Komunikacija klijenata u distribuiranoj Java aplikaciji

Čizmar, Ivan

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:200:841730

Rights / Prava: In copyright / Zaštićeno autorskim pravom.

Download date / Datum preuzimanja: **2024-03-29**

Repository / Repozitorij:

Faculty of Electrical Engineering, Computer Science and Information Technology Osijek

zir.nsk.hr

Image not found or type unknown

SVEUČILIŠTE JOSIPA JURJA STROSSMAYERA U OSIJEKU
ELEKTROTEHNIČKI FAKULTET

Sveučilišni studij

KOMUNIKACIJA KLIJENATA U DISTRIBUIRANOJ JAVA APLIKACIJI

Diplomski rad

Ivan Čizmar

Osijek, 2018.

1 Sadržaj

1. UVOD 1
   1.1. Zadatak diplomskog rada 2
2. *REST API* SERVIS 3
   2.1. *Web hosting* 4
   2.2. Baza podataka 4
   2.3. *REST PHP* servis 6
   2.4. Forma 11
   2.5. Povezivanje forme i *REST* servisa 17
3. *JAVA* KLIJENTSKA APLIKACIJA 20
   3.1. *MVC (Model-View-Controller)* 20
   3.2. *Model* i *ORM* 22
   3.3. *Controller* 26
   3.4. *View* 28
   3.5. Dohvat podataka s *REST* Servisa 30
4. PRINCIP RADA *JAVA* KLIJENT APLIKACIJE 33
   4.1. Izbornik 33
   4.2. Prozor studentskog doma 34
   4.3. Prozor za upravljanje paviljonima 37
   4.4. Prozor za upravljanje sobama i studentima 39
   4.5. Spajanje na *REST API* 41
   4.6. Prozor za upravljanje računima 44
   4.7. Prozor za upravljanje opomenama 46
   4.8. Lambda izrazi 50
      4.8.1. Anonimna implementacija sučelja (eng. *interface*) kroz lambda izraz 51
      4.8.2. Funkcionalno sučelje (eng. *Functional interface*) 54
      4.8.3. forEach() metoda 56
5. ZAKLJUČAK 58

LITERATURA 59
SAŽETAK 60
ABSTRACT 61
ŽIVOTOPIS 62
PRILOZI 63

1. UVOD

U današnje doba postoji mnogo aplikacija, kako mobilnih tako i računalnih. Te aplikacije su pojednostavile ljudski život u gotovo svakom njegovom aspektu, primjerice u osobne ili poslovne svrhe. Zbog ogromne rasprostranjenosti interneta, dijeljenje informacija, kao i pristup njima, postali su svakodnevnica. Stoga, veliki broj današnjih aplikacija na neki način komunicira s internetom. Kako bi pristup podacima pohranjenima na internetu, odnosno nekom serveru, bio omogućen, na njemu se mora nalaziti odgovarajući *Web* servis koji nudi potrebne podatke ili odrađuje zatražene radnje. Komunikacija između distribuirane aplikacije i *Web* servisa omogućena je pomoću *software-ske* veze koja se naziva *API* (engl. *Application programming interface,* API). *API* se nalazi u pozadini svake aplikacije koja je u interakciji sa *Web* servisima. Time su moguće radnje kao što su vremenska prognoza, slanje i primanje e-maila, pregled slika i sl. Također, poslovnim korisnicima omogućeno je jednostavnije i bolje oglašavanje te poslovanje.

U sljedećim poglavljima bit će prikazan cjelokupni postupak izrade *Web* servisa, *Java* klijentske aplikacije te *API* veze zbog ostvarivanja komunikacije između njih. Kreiranje *web API* servisa izvršit će se pomoću *PHP* programskog jezika na serveru *000webhost* te će služiti za dohvaćanje i kreiranje studenata u online bazu podataka. Servis će se podijeliti u tri djela: konfiguracija, objekt i student. Unutar konfiguracije definirat će se veza prema online bazi podataka radi mogućnosti čitanja i zapisivanja podataka u nju. Objekt će sadržavati klasu Student koja služi kao opisni podatak koji će se spremati ili dohvaćati. Također će sadržavati metode koje to omogućavaju. Student dio sadržavat će datoteke čijim pozivanjem bit će vraćeni svi studenti iz baze podataka ili će biti zapisani u bazu podataka. Unos studenata unutar online baze podataka bit će omogućeno kreiranjem forme koja će pozivati datoteku za spremanje studenata u bazu podataka iz Student dijela *Web* servisa. *Web* servis bit će stvoren na osnovu *REST* arhitekturnog stila koji u svojoj pozadini radi pomoću *HTTP* protokola. Također će detaljnije biti opisane *GET* i *POST* metode korištene pri pozivanju servisa. Podaci koje nudi *Web* servis moći će se dohvatiti pomoću Java klijentske aplikacije. Klijentska aplikacija predstavlja desktop aplikaciju za praćenje evidencije procesa unutar studentskih domova.

Aplikacija će se sastojati od 6 modela: studentski dom, paviljon, soba, student, račun i opomena. Kako bi se mogla primijeniti unutar bilo kojeg studentskog doma, nudit će mogućnosti *CRUD-a* (engl. *Create, Read, Update, Delete,* CRUD) nad gotovo svim kreiranim modelima. Unutar projekta primijenit će se *MVC* stil, kako bi se pojednostavila njena struktura i odvojila logika od prikaza i modela. Aplikacija će se spajati na lokalnu bazu podataka koja sadrži šest tablica (po jedna za svaki model). Pomoću *Hibernate ORM-a* omogućit će joj se povezivanje modela unutar baze podataka te manipulacija nas istim tim podacima. Unutar aplikacije, bit će stvoren *API*, pomoću kojeg će se s prethodno kreiranim *REST API* servisa dohvatiti svi studenti, pri preuzimanju rasporediti u željene sobe te spremiti u bazu podataka. Time će biti zaključena distribuirana aplikacija.

1.1. Zadatak diplomskog rada

Napraviti Java aplikaciju koja će komunicirati s API web servisom. Aplikacija treba imati praktičnu primjenu. Objasniti korištene protokole za ovakvu komunikaciju.

2. *REST API* SERVIS

Zbog potreba za javnim prikazivanjem željenih korporativnih podataka razvijeno je standardno sučelje (*Interface*) za potrebe korištenja istih. Kako bi se izbjeglo razmatranje izgleda korisničkog sučelja kreirani su *Web* servisi. *Web* servis predstavlja mehanizam pomoću kojeg se objavljuju korisni podaci koje je moguće „dohvatiti“ pomoću *HTTP* upita. Osim na tisuće javno dostupnih *Web* servisa koji pružaju *API* sučelja za očitavanje podataka različitih tipova postoje i privatni koje koriste kompanije za internu uporabu. Prvi standard za korištenje i objavljivanje *Web* servisa bio je *SOAP* (*Simple Object Access Protocol*) koji je zasnovan na *XML* (*Extensible Markup Language*) kodu. Formiranjem *HTTP* zahtjeva korisnicima je bilo omogućeno primanje odgovora, ali morao je biti poznat direktorij servisa, funkcija, kao i naziv krajnjih adresa. Direktorij servisa mogao je biti objavljen pomoću *WSDL* (*Web Services Description Language*), ali zbog zasnovanosti na *XML* kodu povećavala se kompleksnost procesa. Zbog toga je razvijen koncept *REST* servisa.

REST je skraćenica od „*Representational State Transfer“* te *web* servis kreiran pomoću *REST* principa naziva se *RESTful web* servis. Za razliku od *SOAP* protokola, *REST* predstavlja kreiranja *web* servisa. Dr. Roy Filding identificirao je *REST* principe:

- Svaki *Web* resurs ima svoj jedinstveni identifikator,
- Za dohvaćanje podataka koristi se uniformno sučelje: *HTTP Get*, *Post*, *Put*, *Delete* i sl.,
- Upućeni zahtjevi nemaju stanje što znači da se između zahtjeva ne čuva nikakva informacija specifična za klijenata,
- Resurs može imati više oblika zapisa (*JSON*, *XML*, *PDF*, tekstualni zapis i sl.),
- Moguća je povezanost resursa,
- *REST* aplikacija omogućava slojevitost,
- te resurs treba imati mogućnost keširanja prema [1].

Resurs predstavlja sve ono čemu je moguće pristupiti pomoću hiperlinka te svaki resurs sadrži jedinstveni *URI* (*Uniform resource identidier*). Kako bi resurs bio *REST* resurs, potrebno je da podržava standardne *HTTP* zahtjeve bez stanja. Zbog toga *RESTful web* servisi omogućuju korištenje *GET* metode za dohvaćanje podataka, *POST* metode za kreiranje novih resursa, *PUT* metode za promjenu postojećih te *DELETE* metode za uklanjanje određenih resursa.

U ovoj distribuiranoj aplikaciji korištena je *GET* metoda koja ima određene karakteristike. *GET* metoda omogućuje modifikaciju resursa, keširanje rezultata te „idempotentnost“ što znači da koliko god puta se izvrši metoda, rezultat se neće promijeniti.

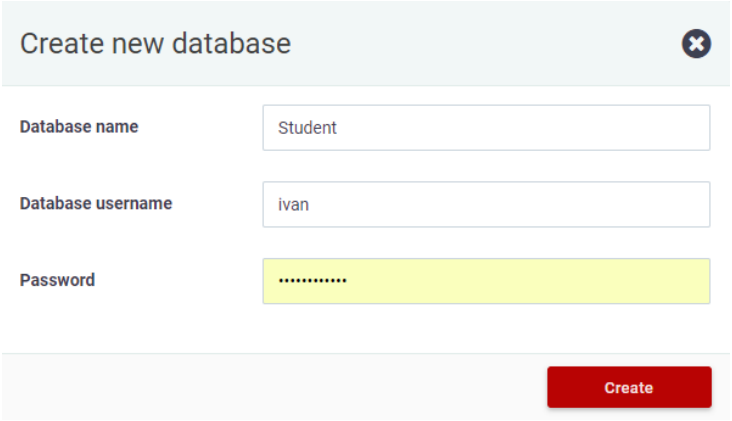
2.1. *Web hosting*

Kako bi se za potrebe klijentske aplikacije izradio prikladan *REST API* servis korišten je besplatan server koji se naziva „*000webhost“* . On nudi podršku za *MySQL* radi mogućnosti izrade baze podataka kojoj se pristupa pomoću *phpMyAdmin* sučelja, *PHP* (eng. *Hypertext Preprocessor*) koji omogućuje rad s podacima iz baze podataka. Također ima podršku *FTP*-a (eng. *File Transfer Protocol*) kojim je omogućeno sigurno i brzo prebacivanje datoteka sa računala na server. Nakon otvaranja korisničkog računa, server dodjeljuje domenu za pristup istom (korisničkoIme.000webhost.com) kako bi se omogućio daljnji pristup početnom direktoriju *(root directory).*

Sl. 2.1. *Logo servera 000webhost*

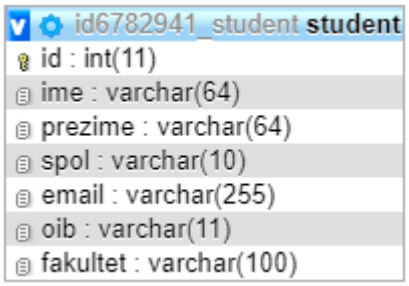
2.2. Baza podataka

Zbog podržanosti *MySQL-a* , *000webhost* nudi mogućnost kreiranja nove ili manipulaciju nad već postojećom bazom podataka. Također je moguće mijenjati lozinku, samu bazu ili kompletnu obrisati. Kako bi se spriječila prenatrpanost memorije, baza podataka je ograničena na maksimalnih 1 GB veličine te na maksimalno 150 tablica. Zbog instaliranog *phpMyAdmin* panela uvelike je olakšano kreiranje nove baze podataka. Prilikom kreiranja nove baze podataka, potrebno je upisati naziv baze podataka (*Database Name*), korisničko ime (*Database Username*) te lozinku(*Password*).



Sl. 2.2. *Prozor za kreiranje nove baze podataka naziva Student*

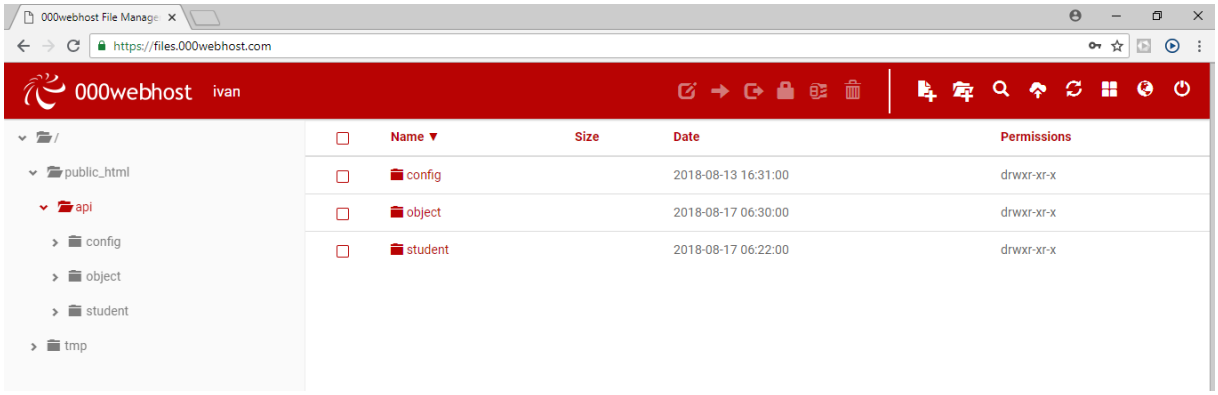
Nakon kreiranja nove baze podataka server implicitno dodaje *id* s nasumičnim brojem ispred naziva baze i korisničkog imena kako bi eliminirao mogućnost dupliciranja. Važno je dobro se pobrinuti o tim podacima jer su potrebni pri spajanju aplikacije na bazu podataka. *REST* servis koristi bazu podataka sa jednom tablicom imena „student“. Tablica sadrži sedam atributa. Primarni ključ entiteta je *id* koji se pomoću naredbe *auto_increment* generira svaki put nakon unosa novog studenta te je tipa *INT(11)*. Tablica također sadrži entitete; *ime* koje je tipa *VARCHAR*(64), *prezime* koje je također *VARCHAR*(64), *spol* koje je tipa *VARCHAR*(10), *email* tipa *VARCHAR*(255), *oib* tipa *VARCHAR*(11) i *fakultet* tipa *VARCHAR*(100).



Sl. 2.3. *Tablica unutar kreirane baze podataka Student*

2.3. *REST PHP* servis

Kako bi prikupljeni podaci nakon procesa unosa, potvrđivanja, odnosno prosljeđivanja i validacije mogli biti spremljeni u prethodno kreiranu bazu podataka ili iščitani iz iste, potrebno je napraviti odgovarajući *REST* servis.



Slika 2.4. *Struktura REST PHP servisa*

REST servis kreiran je pomoću *PHP* jezika, a na slici 2.4. prikazana je struktura servisa. Servis je podijeljen u tri direktorija : *config*, *object* i student direktorij. Unutar *config* direktorija nalazi se *database.php* file. Uloga mu je stvaranje konekcije na prethodno kreiranu bazu. *File* zapravo predstavlja klasu *Database* kojoj je zadaća kreiranje veze prema bazi podataka kako bi se omogućilo izvršavanje željenih upita (eng. *query*).

```
<?php
Class Database(){
      private $host = "localhost";
      private $db_name = "id6782941_student";
      private $username = "id6782941_ivan";
      private $password = "ivan123";
      public $conn;
      public function getConnection(){
        $this->conn = null;
        try{
            $this->conn = new PDO("mysql:host=" . $this->host . ";dbname=" .
$this->db_name, $this->username, $this->password);
            $this->conn->exec("set names utf8");
        }catch(PDOException $exception){
            echo "Connection error: " . $exception->getMessage();
        }
        return $this->conn;
    }
} ?>
```

Primjer koda 2.1. *Klasa Database() s pripadajućim svojstvima i metodama*

Poslužitelj na kojem se nalazi baza podataka je *localhost* (127.0.0.1) iz razloga što se *REST* servis nalazi na istom serveru kao i kreirana baza podataka. Kako bi konekcija bila upotpunjena potrebno je ime baze podataka, korisničko ime te lozinka definirani pri njenom kreiranju. Dostupnost konekcije izvan klase omogućeno je javnim svojstvom *$conn* u koje se sprema cjelokupna veza prema bazi podataka. Kreiranje konekcije odrađuje se pomoću *getConnection()* metode. Kao prvi korak definirana je varijabla *$conn* na *null* vrijednost. Zatim se unutar *try-catch* bloka kreira konekcija pomoću *PDO-a*. *PDO (PHP Data Object)* je apstraktno sučelje koje stvara konekciju te mu je potrebno mu je proslijediti svojstva klase prema [2].

```
//String
"mysql:host=localhost; dbname=id6782941_student, id6782941_ivan, ivan123"
.....
//Omogucavanje dijakritičkih znakova
$this->conn->exec("set names utf8");
```

Primjer koda 2.2. *Format Stringa koji je proslijeđen konstruktoru PDO-a te omogućavanje dijakritičkih znakova*

Ukoliko se dogodi pogreška prilikom kreiranja konekcije, u slučaju neispravnog korisničkog imena, lozinke, naziva baze podataka ili čak *hosta*, *try-catch* blok će preuzeti pogrešku (eng. *Exception* – iznimka), a korisnik će biti o tome obaviješten porukom kako konekcija nije uspješno kreirana uz ispisanu vrstu iznimke.

Unutar *object* direktorija nalazi se datoteka *student.php*. Unutar datoteke nalazi se klasa nazvana *Student*.

```
<?php
class Student{

      //konekcija na bazu i ime tablice
      private $conn;
      private $table_name = "student";
      //svojstva studenta
      public $id;
      public $ime;
      public $prezime;
      public $email;
      public $spol;
      public $oib;
      public $fakultet;

      public function __construct($db){
            $this->conn = $db;
      }
      ..... ?>
```

Primjer koda 2.3. *Klasa Student s pripadajućim svojstvima i konstruktorom koji prima konekciju na bazu podataka kao parametar*

Klasa sadrži sva ona svojstva koja su potreba za kreiranje studenta pomoću forme , ali i za dohvaćanje studenata koji će biti proslijeđeni klijentskoj aplikaciji. Također sadrži i privatna svojstva u koja će se spremiti konekcija na bazu *($conn)* i tablica na kojoj će biti izvršen upit *($table_name)*. Kako bi se omogućilo postavljanje studenata pomoću prethodno stvorene forme unutar klase „Student“ potrebno je definirati funkciju koja kreira *INSERT query*. Zbog toga, klasa sadrži *create()* funkciju.

```
function create(){

    // query za insert studenta
    $query = "INSERT INTO
                " . $this->table_name . "
            SET
                ime=:ime, prezime=:prezime, email=:email,
fakultet=:fakultet, spol=:spol, oib=:oib";
    // pripremi query
    $stmt = $this->conn->prepare($query);

    // Validiranje
    $this->ime=htmlspecialchars(strip_tags($this->ime));
    //Postupak je jednak za sve preostale varijable

    // zamjeni vrijednosti u query-u
    $stmt->bindParam(":ime", $this->ime);
      //Za sve ostale varijable je isti postupak

    // execute query
    if($stmt->execute()){
        return true;
    }

    return false;
}
```

Primjer koda 2.4. *Funkcija create() kojom se studenti zapisuju unutar baze podataka*

Prije svega, potrebno je definirati varijablu *$query* kojoj je kao vrijednost dodijeljen upit koji zapisuje novog studenta u bazu podataka. Upit je definiran tako da se u daljnjem kodu omogući umetanje vrijednosti u njega (*ime=:ime*). Definirani $*query* priprema se za izvršenje spremanjem u varijablu *$stmt* (eng. *statement*). Korištenjem *strip_tags()* funkcije izbjegavaju se implementacije *HTML, XML* i *PHP* oznake. Također, dodatno je zaštićeno uvođenjem *htmlspecialchars()* metode. Postupak je jednak za svaku varijablu koja opisuje studenta. Nakon validacije podataka potrebno je kreirati završni upit pomoću metode *bindParam()* kojom se određeni dio *String-a*, definiranog u *$query*, zamjenjuje željenom vrijednošću. *bindParam()* metodu potrebno je izvršiti za svako svojstvo studenta kako bi se zapis odradio u potpunosti bez *null* vrijednosti. Pomoću metode *execute()* izvršava se kreirani upit, a student se zapisuje u bazu podataka.

Druga funkcija koja je implementirana unutar *Student* klase, a ujedno omogućuje dohvaćanje svih studenata iz baze jest *read()*.

```
function read(){

    // select all query
    $query = "SELECT *
            FROM " . $this->table_name;

    // prepare query statement
    $stmt = $this->conn->prepare($query);

    // execute query
    $stmt->execute();

    return $stmt;
}
```

Primjer koda 2.5. *Funkcija read() kojom se dovaćaju studenti iz baze podataka*

Funkcija definira upit koji iz prethodno postavljene tablice dohvaća sve studente. Upit se priprema za izvršenje pomoću *prepare()* funkcije, a izvršava pomoću *execute()* funkcije. Povratne informacije dobivene izvršenjem *execute()* funkcije „vraćaju“ se pomoću *return* naredbe.

Kako bi *REST* servis bio upotpunjen za povrat informacija iz baze podataka unutar direktorija Student kreirana je datoteka *read.php*. Datoteka svojim pozivanjem ujedno poziva i *read()* funkcije definiranu unutar *Student* klase.

```
<?php
      // required headers
      header("Access-Control-Allow-Origin: *");
      header("Content-Type: application/json; charset=UTF-8");

      //include database i objekt file
      include_once '../config/database.php';
      include_once '../object/student.php';

      // instanciranje database i student objekta
      $database = new Database();
      $db = $database->getConnection();
      $student = new Student($db);

      //query studenti
      $stmt = $student->read();
      $num = $stmt->rowCount();

      //provjerava ima li više od 0 zapisa u bazi
      if($num > 0){
```

```
            //lista studenata
            $students_arr=array();

            while ($row = $stmt->fetch(PDO::FETCH_ASSOC)){
            // extract row
            // to ce napraviti od $row['ime']
            // samo $ime
            extract($row);
            $student_proper=array(
            "ime" => $ime,
            "prezime" => $prezime,
            "email" => $email,
            "spol" => $spol,
            "oib" => $oib,
            "fakultet" => $fakultet
             );
            array_push($students_arr, $student_proper);
            }

      echo json_encode($students_arr);
      }
      else{
            echo json_encode(
                  array("message" => "U bazi nema studenata.")
            );
      }
?>
```

Primjer koda 2.6. *Funkcija read() kojom se pokreće dohvaćanje studenata iz baze podataka*

Kako bi se omogućilo dobivanje odgovora, bez obzira na koji način ili iz kojeg izvora je kreiran zahtjev te da bi odgovor mogao biti zapisan u *JSON* formatu, potrebno je definirati odgovarajuća zaglavlja kao što je vidljivo unutar *read.php* datoteke. Korištenjem naredbe *includ_once*, čitav kod sadržan u navedenim datotekama (datoteka na kraju putanje nakon *include_once* naredbe) kopiran je unutar read.php file-a. Za doseg svih studenata iz baze potrebno je kreirati instancu klase *Student* pomoću konstruktora koji prima konekciju na bazu kao parametar, a sprema ga u svojstvo kreirane instance. Pozivanjem metode *read()* kroz instanciranog studenta, u varijablu *$stmt* se sprema lista dohvaćenih studenata, a pomoću funkcije *rowCount()* u varijablu *$num* spremljen je točan broj elemenata liste. Pomoću *if* uvjetnog grananja provjera se je li lista prazna. Ako u bazi nema unesenih studenata, program porukom obavještava korisnika o tome. Ukoliko je broj elemenata liste veći od 0 *($num*>0) kreće obrada dohvaćenih podataka. Najprije se funkcijom *array()* kreira prazna lista u koju će biti spremljeni dohvaćeni studenti. Kako bi se prikupili i obradili svi podaci iz liste koristi se *while* petlja koja obrađuje blok koda sve dok ne prođe sve elemente liste $*stmt*. Unutar uvjeta *while* petlje svaki element $*stmt* sprema se unutar varijable *$row,* identično svakom retku dohvaćenom iz baze. To je omogućeno pomoću *PDO fetch* naredbe u kombinaciji s *PDO::FETCH_ASSOC* parametrom, što rezultira spremanjem svih vrijendosti dohvaćenih iz baze kao asocijativne liste. Zatim funkcija *extract()* pretvara listu

row u lokalne varijable. Te varijable spremaju se u listu *$student_proper* koja se pomoću *array_push()* funkcije pohranjuje unutar prethodno kreirane liste *$student_array*. Dakle, u listu *$student_array* spremljeni su svi dohvaćeni studenti koji se ispisuju pomoću *echo* funkcije u *JSON* formatu.

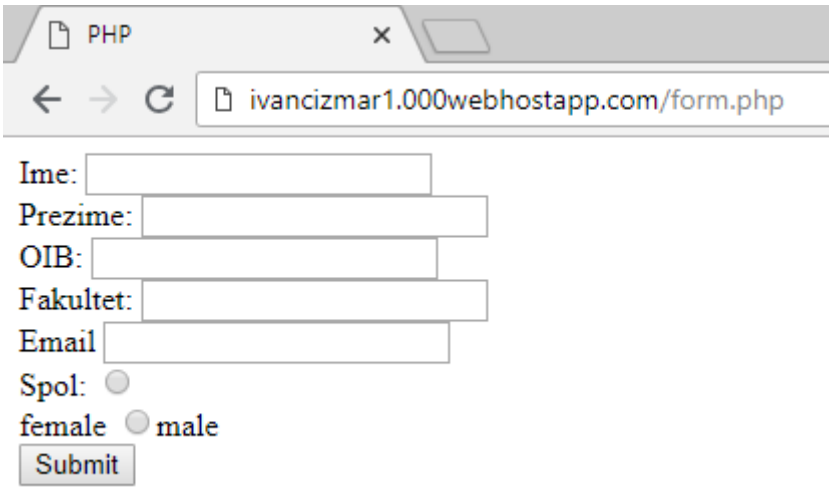
2.4. Forma

Kako bi se omogućila prijava studenata, odnosno njihovo kreiranje u bazi podataka, kreirana je *PHP* datoteka „form.php“ koja sadrži formu za unos. Forme su jedini način u *HTML(HyperText Markup Language)* kojim korisnik može komunicirati s *web* stranicom, unijeti podatke i slično. S obzirom da se *HTML* stranica sastoji od *head-a* i *body-a*, forme se kreiraju unutar *body-a*.

```
<form method="post" action="">
<!-- elementi forme -->
      Ime: <input type="text" name="ime"><br>
</form>
```

Primjer koda 2.7. *Naredba za kreiranje tekstualnog polja za unos imena studenta unutar forme*

One primaju mnoštvo atributa, a dva najvažnija su *action* i *method*. *Action* atribut predstavlja „metu“, odnosno stranicu koja će biti korištena za prosljeđivanje podataka iz forme, a njegovim postavljanjem na vrijednost praznog *String-a* označava se trenutna stranica iz koje će se prosljeđivati podaci forme. *Method* atribut zadržava vrijednost *HTTP* upita ili *HTTP* metode. Početna vrijednost je *GET*, te ukoliko se napravi prosljeđivanje podataka ili forme tom metodom, podaci forme bit će sadržani u *URL-u (?name=value).* Kako bi se omogućio unos u bazu važno je da *HTTP* metoda bude *POST* (*HTTP* metoda koja uvjetuje stvaranje zapisa u bazi podataka), iako *HTTP* na tim mjestima pruža mnoštvo ostalih metoda (*PUT, GET, DELETE, HAD, OPTION, TRACE*). *POST* metoda nema limita pa je moguće i prosljeđivanje datoteka. Podaci neće biti sadržani u *URL-u*, već će biti sadržani u tijelu *HTTP* upita, a prikladna *PHP* lista koja sadržava sve podatke je *$_POST*. Pozivanjem *URL-a* „*ivancizmar1.000webhost.com/form.php*“ u internetskom pregledniku otvara se stranica koja sadržava kreiranu formu te izgleda:



Sl. 2.5. *Forma za unos podataka studenta radi njihovog spremanja u bazu podataka*

Forma se sastoji od pet tekstualnih polja, odnosno jednorednih tekstualnih polja koja omogućavaju unos podataka o pojedinom studentu kao što su ime, prezime, OIB, fakultet koji pohađa ili koji će tek upisati (ukoliko se radi o studentima prvih godina), te *email-a*. U gore navedenom primjeru kreiranje forme (Sl. 2.5.) bitno je označiti tip forme (eng. *type*) kao i njeno ime (eng. *name*) jer to su vrijednosti koje će se proslijediti bazi nakon pritiska na gumb *„Submit“*. Također su korišteni i *Radio buttons* za koje je specifična pojava u grupama što omogućava odabir između ponuđenih opcija. Grupira ih se postavljanjem istog imena svim radio button-ima koji pripadaju istoj grupi, ali vrijednosti su im drukčije. Ukoliko je korisnik odabrao „Musko“ u bazu se prosljeđuje *true*, a odabirom „Zensko“ prosljeđuje se *false* čime je korisnik ograničen na ponuđeni odabir. Kako bi se formi promijenio klasični izgled, korišten je *Bootstrap frameworka*. *Bootstrap* predstavlja *front end framwork* za kreiranje *web* stranica prilagodljivog izgleda. Implementiran je metodom *copy-paste* na način da je *stylesheet* (lista željenih dizajna) hiperveza kopirana u *head* dio *HTML-a* kako bi omogućio učitavanje svih dostupnih lista stilova u *CSS* (*Cascading Style Sheets*) prema [3].

```
<head>
      <title>PHP</title>

      <link rel="stylesheet"
      href="https://stackpath.bootstrapcdn.com/bootstrap/4.1.3/css/bootstrap.m
      in.css" integrity="sha384-
      MCw98/SFnGE8fJT3GXwEOngsV7Zt27NXFoaoApmYm81iuXoPkFOJwJ8ERdknLPMO"
      crossorigin="anonymous">

</head>
```

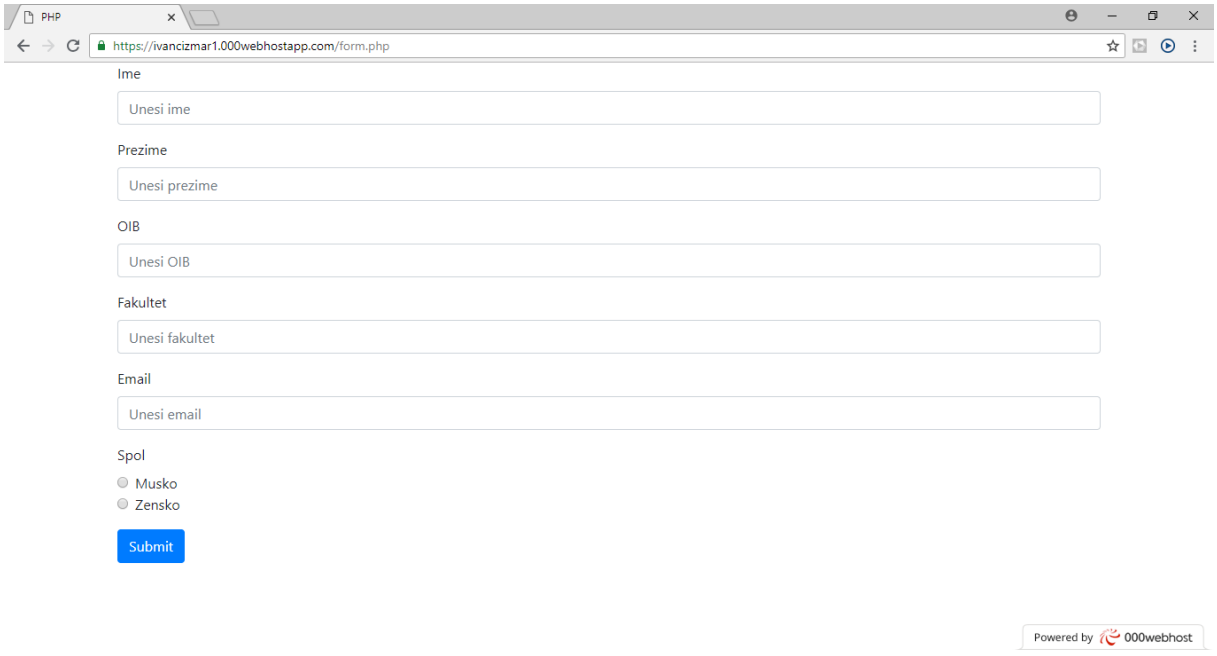
Primjer koda 2.8. *Omogućavanje primjene Bootstrap-a putem hiperveze kopirane unutar head dijela HTML-a*

Kako bi se primijenio željni izgled elemenata forme potrebno ih je staviti u divizije (eng. *division*) radi primjene *CSS-a*. Cijela forma postavljena je u diverziju koja se naziva „*container*“, a svaki pojedini element forme u zasebnu diviziju. Time je omogućen zaseban odabir klase forme, naslova elementa forme, dok su tip i ime nepromijenjeni. Kako bi se korisniku olakšao unos umetnuti su i *placeholderi* koji ukazuju na podatke koje treba unijeti u polja elemenata forme.

```
<form method="post" action="">
  <div class="form-group">
    <label for="ime">Ime</label>
      <input type="text" name="ime" value="<?php echo
      htmlspecialchars($ime);?>" class="form-control" placeholder="Unesi ime"
      id="ime" />
  </div>
```

Primjer koda 2.9. *Implementacija Bootstrap-a na prethodno kreirano tekstualno polje forme*

Ovom implementacijom uvelike je povećana preglednost forme, izgled je elegantniji i ugodniji. Na slici 2.5. može se zamijetiti kako kod odabira spola nije jasno definirano koji *radio button* je određen za koji spol, dok to na slici 2.6. nije slučaj. Sl. 2.6. prikazuje nastalu formu nakon obrade *CSS-om*.



Sl. 2.6. *Prethodno kreirana forma uz implementirani Bootstrap*

Svaka forma sadrži *Submit* tipku te je potrebno iskoristiti tu informaciju kako bi se ustanovilo treba li formu proslijediti ili ne. Zbog toga je moguće pomoću *PHP-a* ustanoviti je li forma potvrđena. Unutar *body-a HTML*-a dodavanjem *PHP* odjeljka (<?*php* …?>) omogućeno je izvršavanje *PHP* upita. Zbog postojanja globalnih varijabli unutar *PHP-a (Superglobals)* omogućena je provjera je li došlo do pritiska na tipku. Prilikom pritiska na tipku, zbog postavljenog atributa *method* na vrijednost *POST*, kreira se upit koji u svojim zaglavljima ima upisanu metodu forme. Toj metodi moguće je pristupiti kroz globalnu varijablu *$_SERVER*. Varijabla *$_SERVER* sadrži element „*REQUEST_METHOD*“ koji vraća metodu korištenu za pristup stanici (u ovom slučaju *POST*) prema [4].

```
<?php
      if($_SERVER['REQUEST_METHOD'] === 'POST'){
      //Odradi određeni kod (Spremanje studenta u bazu)
```

Primjer koda 2.10. *Provjera proslijeđenosti koda pritiskom na tipku Submit*

Na taj način je omogućeno odrađivanje željenog koda ukoliko je forma potvrđena pritiskom na tipku *Submit*. Iako na taj način forma prosljeđuje vrijednosti unesene u tekstualna polja, ti podaci mogu biti štetni za bazu ili formu. Ukoliko treća strana unosi podatke, oni nisu pouzdani. Uneseni podaci također mogu sadržavati kod *HTML-a*. Kako bi se izbjegao unos *HTML* koda, korištena je *PHP* metoda *htmlspecialchars().*

```
<div class="form-group">

      <label for="ime">Ime</label>
      <input type="text" name="ime"
      value="<?php echo htmlspecialchars($ime);?>"
      class="form-control" placeholder="Unesi ime" id="ime" />

</div>
```

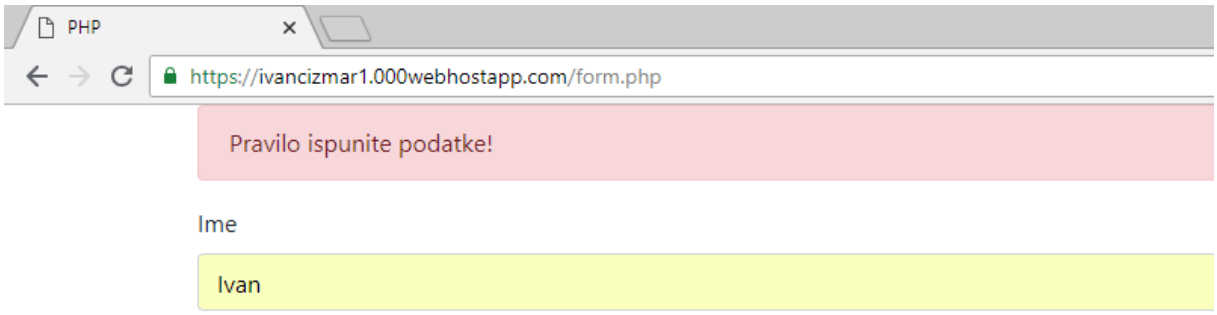
Primjer koda 2.11. *Izbjegavanje implementacije HTML koda pri unosu podataka studenta*

U primjeru je vidljivo da ukoliko se želi koristiti *htmlspecialchars()* metoda, potrebno ju je pozvati unutar *PHP* odjeljka. Odjeljak se otvara unutar vrijednosti (eng. *value*) koja se dodjeljuje tekstualnom polju. Na taj način je elimininiran unos znakova <, >, “, &, kao i *HTML* naredbi, već prosljeđuje navedene znakove kao običan tekst. Zbog važnosti svih podataka, potrebno je odraditi njihovu validaciju. Prije prosljeđivanja podataka potrebno je provjeriti jesu li tekstualna polja ispravno popunjena, odnosno postoji li zapis unutar tekstualnog polja ili ne. Također, znajući da vrijednost *radio buttona* mora biti proslijeđena serveru (bazi podataka) potrebno je provjeriti je li neki od njih odabran. Kako bi se to omogućilo, potrebno je provjeriti nalaze li se u listi *$_POST* svi elementi forme nakon pritiska na gumb *Submit*.

```
<?php

	$ime = '';
	//Ostale varijable također se postavljaju kao prazan String

	if($_SERVER['REQUEST_METHOD'] === 'POST'){
		$ok = true;
		if(!isset($_POST['ime']) || $_POST['ime']===''){
			$ok= false;
		}else{
			$ime = $_POST['ime'];
		}
		//Isti je postupak za sve elemente forme ......


		if($ok){
			//Spremi studenta u bazu

		}else{
			echo <div class="alert alert-denger" role="alert">
			Pravilno ispunite podatke!</div>
		}
		.......
?>
```

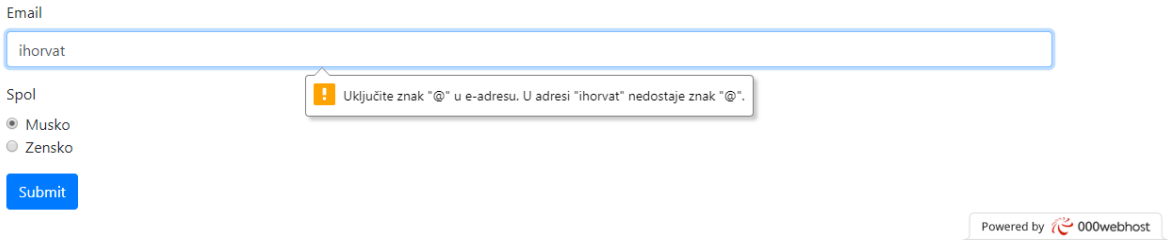
Primjer koda 2.12. *Provjera popunjenosti svih elemenata forme*

Nakon pritiska na *Submit* gumb, podaci iz elemenata formi pohranjuju se u *$_POST* listu. Provjerom potvrde podataka iz prijašnjih primjera moguće je provjeriti jesu li svi elementi ispravno ispunjeni. Provjera je integrirana tako da se postavlja nova varijabla koja je tipa *boolean* na *true* vrijednost. Korištenjem *PHP* metode *isset()* koja provjerava nalazi li se željena vrijednost u *$_POST* listi, odrađena je provjera validnosti proslijeđenih podatka. Ukoliko nije proslijeđena vrijednost tekstualnog polja „ime“ ili ukoliko je proslijeđena vrijednost jednaka praznom *String-u,* tada će varijable *$ok* biti postavljena na vrijednost *false*. Sprječavanje gubitka već unesenih podataka nakon neuspješnog ili neispravnog slanja podataka unutar tekstualnih polja ili *radio buttona*, kreirane su varijable čije su vrijednosti postavljenje kao prazan *String (`$ime = '';`)*. Ukoliko je neko od tekstualnih polja ispravno popunjeno, u *else* dijelu uvjetnog grananja vrijednost koja je trebala biti proslijeđena bazi sprema se u prethodno definirane varijable. Kao posljedica toga slijedi obustavljanje prosljeđivanja podataka bazi, te obavještavanje korisnika kako forma nije pravilno ispunjena (Sl. 2.7.). Ukoliko nakon provjera varijabla *$ok* ostane ne promijenjena, izvršava se blok koda koji prikuplja proslijeđene podatke, od njih kreira objekt studenta, a isti se sprema u bazu podataka.



Sl. 2.7. *Obavijest o nepravilno ispunjenoj formi*

Kako bi proces validacije bio upotpunjen, bitna je provedba kontrole unosa koji obavlja korisnik. Vrijednost koju tekstualno polje prosljeđuje bazi podataka može se definirati pod *value* atributom, ali unutar *PHP* odjeljka. Kao što je već prikazano u prijašnjim primjerima, unutar *inputa* elementa forme dodan je atribut *value* koji za vrijednost prima vrijednost nad kojom je primijenjen *PHP* kod. Unutar *PHP* odjeljka korištena je metoda *htmlspecialchars()* radi eliminacije mogućih implementiranja *HTML* koda. Isto tako korištena je i metoda *echo()* koja omogućava ispis unesene tekstualne vrijednosti.

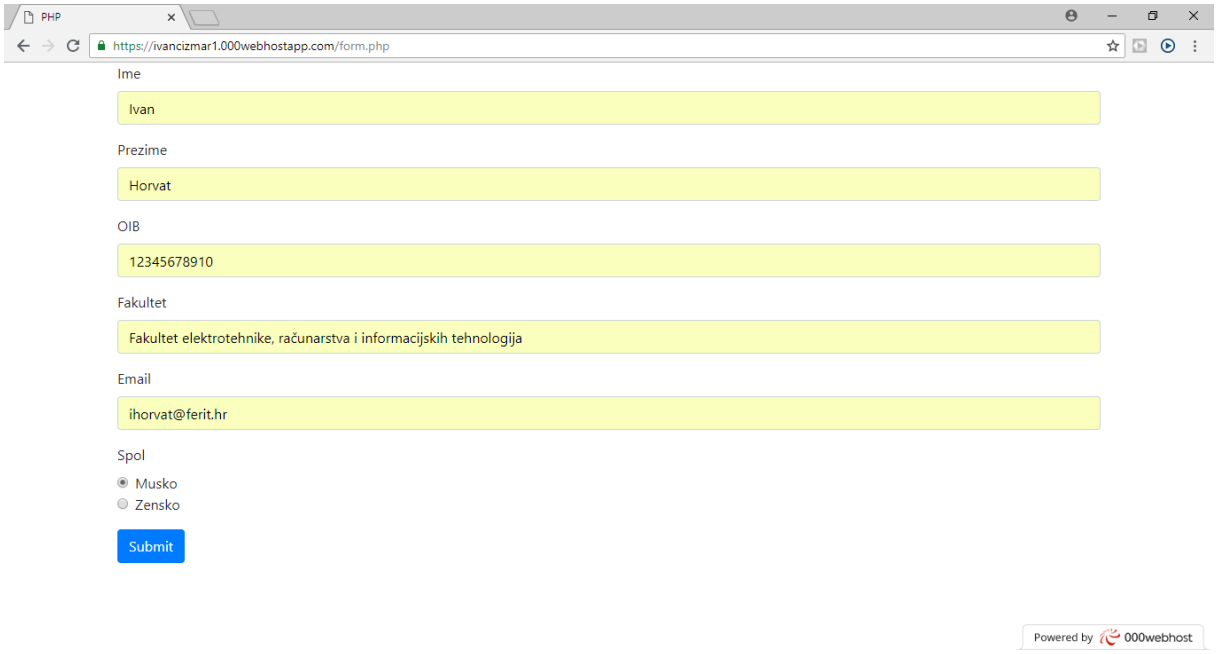


Sl. 2.8. *Kontrola ispravnosti unesene e-mail adrese pomoću Bootstrap-a*

Bitno je napomenuti da implementacija *Bootstrap-a* povećava i sigurnost same forme. To je vidljivo na primjeru krivog unosa *e-mail* adrese. Ukoliko se unese e-mail adresa u kojoj se ne nalazi znak „@“, *Bootstrap* prepoznaje pogrešno napisanu *e-mail* adresu, zaustavlja se prosljeđivanje svih elemenata u bazu, a pravilno uneseni podaci ostaju pohranjeni u tekstualnim poljima kao i odabrani *radio button* dok se ispod polja za unos *e-mail* adrese prikaže skočni prozor s obrazloženjem nevaljanosti unesenog podatka (Sl. 2.8.).

2.5. Povezivanje forme i *REST* servisa

Kreirani *REST* servis, uz mogućnost dohvaćanja studenata, ima i implementiranu i funkciju za dodavanje studenata u bazu podataka. Spremanje studenata u bazu podataka omogućeno je kroz prethodno kreiranu formu. Kao što je navedeno u prethodnom poglavlju, validacijom podataka određuje se daljnji proces izvršavanja koda unutar forme. Ukoliko validacija bude neuspješna, korisnik se obavješćuje o istom, a podaci nisu proslijeđeni.



Sl. 2.9. *Ispravno popunjena forma*

Pravilnim popunjavanjem svih elemenata forme (Sl. 2.9.) student se sprema u bazu podataka. Pritiskom na tipku *Submit*, pokreće se validacija, vrijednost varijable *$ok* ostaje nepromijenjena *($ok = true;)*, a pokreće se i blok koda koji vrši spremanje studenta u bazu.

```
If($ok){

        // Dohvati database konekciju
        include_once './api/config/database.php';
        include_once './api/object/student.php';
        $database = new Database();
        $db = $database->getConnection();
        $student = new Student($db);

        // set student property vrijednosti
        $student->ime = $ime;
        $student->prezime = $prezime;
        $student->spol = $spol;
```

```
$student->email = $email;
$student->oib = $oib;
$student->fakultet = $fakultet;
```

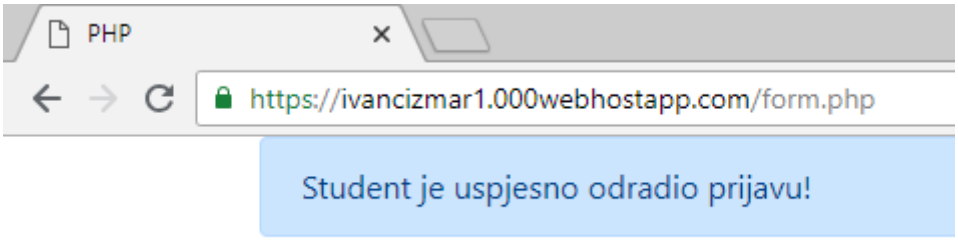
Primjer koda 2.13. *Proces uvoza potrebnih metoda, instanciranje konekcije i studenta te postavljanje vrijednosti forme unutar objekta studenta*

Prije svega potrebno je uvesti kod (pomoću naredbe *include_once*) važan za instanciranje konekcije na bazu podataka i studenta koji će biti spremljen u bazu podataka. Instanciranje studenta vrši se pozivanjem konstruktora koji kao parametar prima prethodno instanciranu konekciju na bazu podataka. Zatim, u stvorenu instancu studenta spremaju se svojstva proslijeđena kroz formu. Nadalje, poziva se metoda *create()* koja sprema studenta u bazu podataka.

```
//create student
if($student->create()){
      echo '<div class="alert alert-primary" role="alert">
      Student je uspjesno odradio prijavu!</div>';
}
//ako krejiranje nije uspjelo , obavjesti korisnika
else{
      echo '<div class="alert alert-danger" role="alert">
      Prijava neuspjela!</div>';
}
```

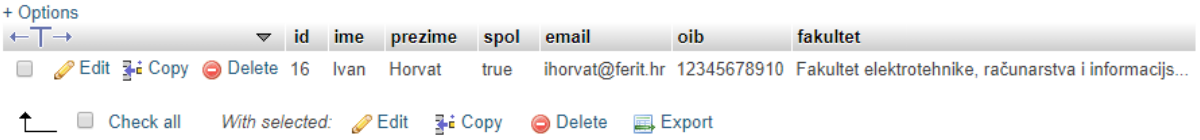
Primjer koda 2.14. *Obavještavanje korisnika o ispravnosti i uspješnosti prijave u sustav*

Kreiranje je omogućeno na način da je instanci studenta proslijeđena konekcija prema prethodno stvorenoj bazi podataka (*database name:* id6782941_student) te je u istom tom konstruktoru spremljena u svojstvo kreiranog studenta. Pozivanjem *create()* metode kreira se upit u koji se postavljaju svojstva prosljeđenog studenta. Spremanje u bazu se završava funkcijom *execute()* koja za rezultat vraća *true* ili *false* ovisno o točnosti izvršenja upita. Zbog toga je moguće kvalitetno obavijestiti korisnika o uspješnoj prijavi, a izgled obavijesti prikazan je na slici 2.10.



Sl. 2.10. *Obavijest o uspješnoj prijavi studenta*

Nakon uspješnog kreiranja, student se pojavljuje u bazi podataka sa svim proslijeđenim vrijednostima iz forme čime je moguće njegovo dohvaćanje za potrebe klijentske *Java* aplikacije.



Sl. 2.11. *Kreirani student prikazan alatom phpMyAdmin*

Kako bi se dohvatili svi studenti unutar baze podataka, potrebno je u web pretraživaču pozvati *URL* „*https://ivancizmar1.000webhostapp.com/api/student/read.php*“ (poziva se izvršavanje *read.php* datoteke).

```
[
    {
        "ime":"Ivan",
        "prezime":"Horvat",
        "email":"ihorvat@ferit.hr",
        "spol":"true",
        "oib":"12345678910",
        "fakultet":"Fakultet elektrotehnike, ra\u010dunarstva i informacijskih
tehnologija"
    }
]
```

Primjer koda 2.15. *Dohvaćeni podaci iz baze podataka u JSON formatu zapisa*

3. *JAVA* KLIJENTSKA APLIKACIJA

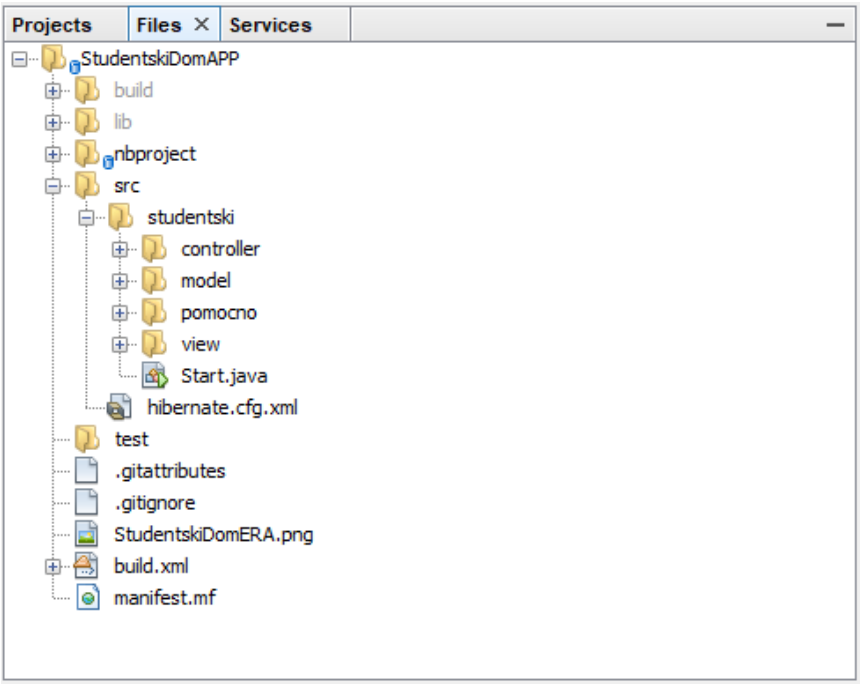
Radi obrade podataka koje je moguće dohvatit uz prethodno kreirani *REST API* servis, potrebno je razviti odgovarajućeg java klijenta. To znači da prikupljeni podaci odgovaraju potrebama klijentske aplikacije te ih ona preuzima i nastavlja s njihovom daljnjom uporabom. Klijentska aplikacija predstavlja aplikaciju namijenjenu evidenciji procesa koji se odvijaju unutar studentskih domova. Ti procesi pokrivaju područje useljavanja studenata u odgovarajuće sobe unutar studentskog doma, evidenciju plaćanja računa kao i izdavanje opomena ukoliko se uplate ne izvrše kroz određeni period. Kako bi aplikacija mogla biti primijenjena u bilo kojem studentskom domu, omogućeno je kreiranje studentskog doma po želji klijenta, te paviljona i soba prema potrebama studentskog doma. Radi izvršavanja koda na Java virtualnoj mašini (engl. *Java Virtual Machine,* JVM) potrebno je pomoću *Java compilera* kreirati *bytecode (.class)* iz izvornog koda *(.java).* Budući da je za izvršavanje aplikacije potreban samo JVM, aplikaciju je moguće pokretati na svim platformama koje imaju JVM. Kreirana klijentska aplikacija će biti razjašnjena na *Windows* operacijskom sustavu.

3.1. *MVC (Model-View-Controller)*

Kreiranje projekta bez vođenja računa o njegovoj arhitekturi nije dobra praksa. Ukoliko bi se sav kod „natrpao“ unutar jednog paketa (direktorij na disku) uvelike bi se povećala njegova nepreglednost kao i mogućnost pogrešaka. Time se može narušiti daljnji razvoj ili unaprjeđenje projekta odnosno aplikacije, jer ukoliko se javi potreba nadogradnje projekta osoba koja se prvi puta susreće s projektom će imati problema s nadogradnjom postojećeg sustava. Zbog povećanja organiziranosti strukture i koda aplikacije korišten je *MVC*. *MVC* je arhitekturni stil (eng. *design pattern*) koji dijeli aplikaciju na tri djela:

- *Model,*
- *View,*
- *Controller.*

Model predstavlja objekt ili *Java POJO (engl. Plain Old Java Object,* POJO- običan stari Java objekt*)* koji je nositelj podataka. Također može sadržavati i logiku za implementaciju *controllera* ukoliko mu se podaci mijenjaju. *View* predstavlja vizualizaciju podataka sadržanih unutar *modela*. *Controller* djeluje i na model i na *view*. Zadaća mu je kontroliranje protoka podataka unutar modela te nadopuna *view-a* ukoliko dođe do promijene podataka. Omogućava odvajanje *modela* i *view-a*

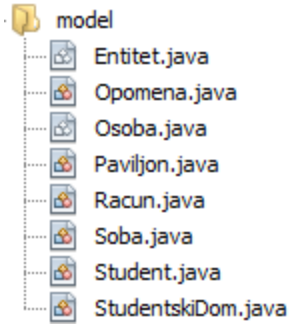


Sl. 3.1. *Struktura Java klijentske aplikacije*

Slika 3.1. prikazuje strukturu aplikacije. *Src* predstavlja direktorij unutar kojeg se nalazi kod koji se izgrađuje. Taj direktorij sadrži *hibernate.cfg.xml* datoteku i direktorij student koji predstavlja paket aplikacije. Unutar paketa student nalaze se paketi: *controller*, *model*, *view*. Na taj način aplikacija je strukturirana na ispravan način te je omogućen paralelni rad na svakoj od komponenti *MVC-a.*

3.2. *Model* i *ORM*

Kao što je već navedeno, *model* predstavlja objekt ili *POJO* koji nosi atribute. Model ove aplikacije sadrži osam klasa kao što je prikazano na slici (Sl. 3.2.).



Sl. 3.2. *Slika strukture modela*

Svaki model predstavlja jedan od entiteta koje je potrebno kreirati unutar baze podataka kako bi se omogućila manipulacija nad podacima. Dvije od njih kreirane su kao abstraktne *(*eng. *abstract)* klase što znači da instanciranje objekata pomoću njih nije moguće, već samo instanciranje objekata pomoću klasa koje nasljeđuju *(*eng. *extends)* abstraktnu klasu, uz mogućnost korištenja svih njenih svojstava za opisivanje instance. Klasa *Osoba* sadrži atribute kao što su:

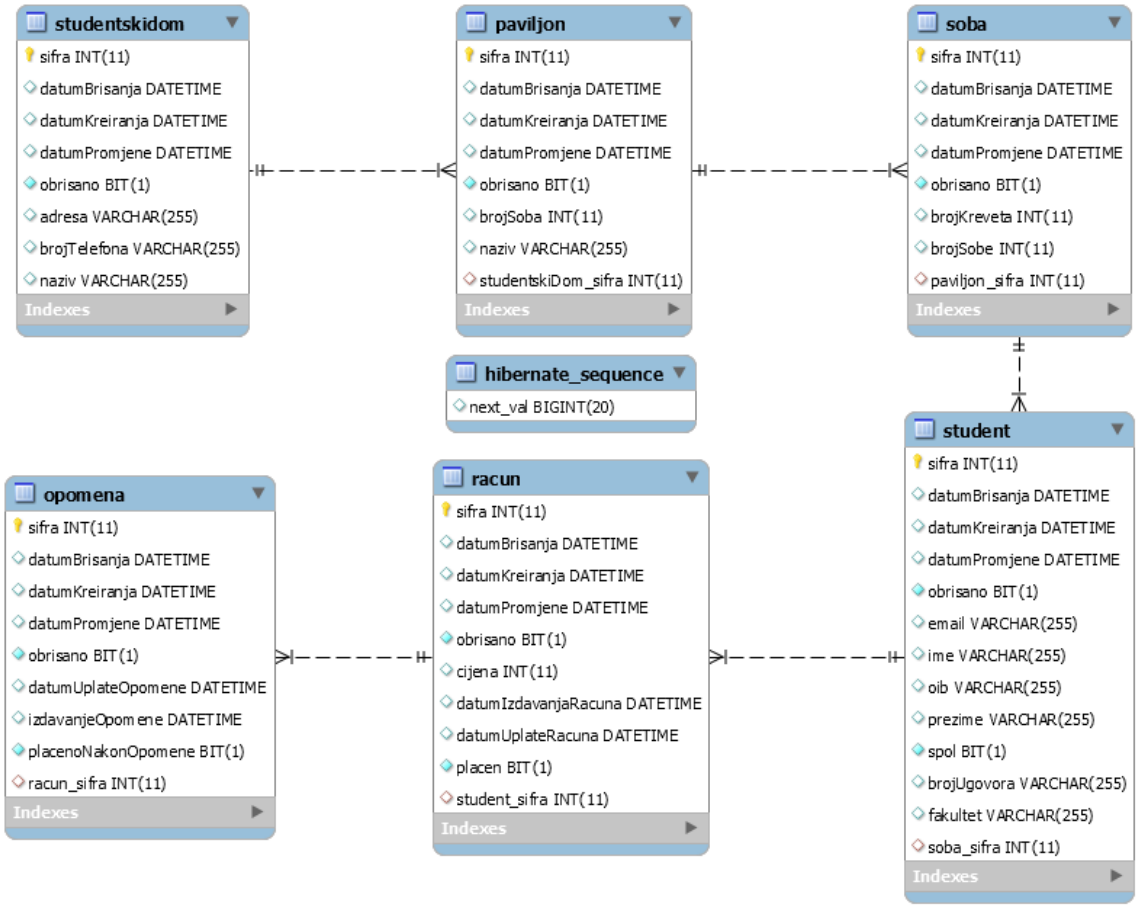
- oib
- ime
- prezime
- *email*
- spol

Ta klasa predstavlja sve ono što je zajedničko za svaku osobu vezanu za studentski dom, bio to student ili zaposlenik. Nju nasljeđuje samo klasa *Student* koji se od zaposlenika ili neke treće strane razlikuje po broju ugovora sklopljenog sa studentskim domom, te upisanog fakulteta. Abstraktna klasa *Entitet* sadrži atribute kao što su:

- šifra
- datum kreiranja
- datum promjene
- datum brisanja
- obrisano (*boolena*)

Klasu *Entitet* nasljeđuju sve klase pa čak i klasa *Osoba*. Njena svrha je da prilikom kreiranja objekta i njegovog zapisivanja u bazu podataka generira šifru koja će taj objekt jednoznačno razlikovati od svih ostalih objekata zapisanih u bazu podataka, odnosno služi za generiranje primarnog ključa. Isto tako ukoliko se dogodi promjena ili brisanje podataka u bazi, zapisuje se vrijeme promjene, odnosno vrijeme brisanja. Kako se ne bi izgubili obrisani podaci, prilikom procesa brisanja, vrijednost svojstva „obrisano“ se postavlja iz *false* u *true* što služi kao indikator je li objekt obrisan.

Zbog jednostavnijeg kreiranja baze podataka na osnovu kreiranih modela i uz prije definirane veze između njih, primijenjen je *Hibernate ORM*. *Hibernate ORM* (engl. *Object-Relational Mapping,* ORM) predstavlja mehanizam pomoću kojeg je moguće adresiranje, pristup i manipulacija nad objektima bez vođenja brige o tome kako su ti objekti vezani za njihov izvor. Omogućava programeru održavanje konciznog pregleda podataka bez obzira dođe li do promjene izvora ih kojih se dohvaćaju, baza unutar kojih se spremaju ili aplikacije koja im pristupa. Pravilno postavljenim vezama između modela i prethodno kreiranom bazom podataka, *ORM* kreira sve tablice i veze između njih prema [5].



Sl. 3.3. *ER dijagram klijentske aplikacije*

Slika 3.1. prikazuje ER (engl. *Entity Relationship*, ER) dijagram kreiran uz pomoć *ORM-a*. To je postignuto pravilno postavljenim anotacijama (eng. *annotation*) unutar modela i pravilno postavljenjenom konfiguracijskom datotekom *Hibernatea*. Anotacije su specifične riječi označene s „@“ koje predstavljaju *Javadoc* metapodatke koji opisuju linkove, verziju, autora i povezane klase u Java dokumentaciji. Abstraktne klase *Osoba* i *Entitet* iznad sebe imaju anotaciju *„@MappedSuperclass“* koja komunicira s *Hibernateom* i govori mu da klasa ne treba biti kreirana unutar baze, već će njihova svojstva biti sadržana unutar klasa koje ih nasljeđuju (u bazi su to atributi). Klasa *Entitet* iznad definicije svojstva šifra ima anotacije *„@Id“* i *„@GeneratedValue“* koje *ORM-u* ukazuju da prilikom upisa objekata u bazu, šifra će predstavljati jedinstvenu vrijednost za svaki objekt te da će biti generira od strane *Hibernatea*, a ne programera. Ostale klase iznad sebe imaju anotacije *„@Entity“* i *„@Table“* koje ukazuju da se radi o entitetima unutar baze, te da je za njih potrebno kreiranje tablica. Kako bi se veze između tablica kreirale ispravno korištene su anotacije *„@ManyToOne“* i *„@OneToMany()“* prema [5].

```
@ManyToOne
private Soba soba;
@OneToMany(mappedBy = "student")
private List<Racun> racuni = new ArrayList<>();
```

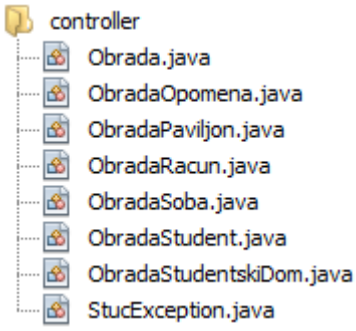
Primjer koda 3. 1. *Primjena anotacija „@ManyToOne“ i „@OneToMany()“*

Kod iz primjer 3.1. prikazuje način korištenja navedenih anotacija za spajanje tablica pomoću *Hibernatea*. Sadržan je unutar klase *Student*. Varijable „soba“ i „racuni“ također predstavljaju atribute studenta. Kako student boravi u studentskom domu duži period, za svaki mjesec stanovanja izdaje mu se račun. Zbog toga veza između studenta i računa mora biti 1:N (jedan na više). Realizacija te veze omogućena je anotacijom *„@OneToMany()“* iznad liste računa koji će biti vezani uz studenta. Isto tako, student fizički može biti vezan samo za jednu sobu, ali unutar sobe može se nalaziti više različitih studenata te je iz perspektive studenta veza N:1 (više na jedan). Veza je omogućena anotacijom *„@ManyToOne“* dok će unutar klase „Soba“ kao svojstvo biti lista studenata uz anotaciju *„@OneToMany()“*. Treba zamjetiti kako veza 1:N prima parametar *mappedBy* koji kazuje da će unutar svakog računa biti student za kojeg je izdan račun kao što je to soba za koju je vezan student. Time je upotpunjena veza 1:N što je vidljivo na slici 3.3.

Kako bi *Hibernate* znao treba li kreirati tablice unutar baze podataka, ili je potrebna samo manipulacija nad podacima iz već kreiranih tablica unuta „src“ direktorija nalazi se *„hibernate.cfg.xml“* datoteka. Ta datoteka predstavlja *XML* konfiguracijsku datoteku *Hiberantea* u kojoj je definirano na koji način će se Hibernate „ponašati“. Ukoliko je potrebno kreirati tablice unutar baze podataka, potrebno je kreirati samu bazu. Pomoću XAMPP platforme potrebno je pokrenuti *MySql* bazu podataka. Zatim, alatom *MySQL Workbench* potrebno je spojiti se na pokrenutu bazu podataka (*XAMPP* pokreće *MySql* na lokalnom serveru). Nakon uspješnog spajanja na *MySql* bazu podataka, naredbom „*create database studentski character set utf8 collate utf8_general_ci;*“ kreira se prazna baza podataka. Unutar konfiguracijske datoteke potrebno je odrediti konekcijski *URL* prema prethodno kreiranoj bazi podataka, korisničko ime i zaporku potrebnu za spajanje, *driver* baze, dialekt i slično. Kako bi *hiberante* znao da se radi o kreiranju tablica potrebno je navesti unutar svojstva „hbm2ddl.auto“ vrijednost *create* te pravilnim redosljedom navesti klase koje je potrebno kreirati i povezati. Pokretanjem *main* klase, *hibernate* kreira tablice i pokreće aplikaciju. Nakon uspješnog kreiranja važno je unutar svojstva *„hbm2ddl.auto“* postaviti vrijednost *update* koja govori *ORM-u* potrebna manipulacija nad podacima.

3.3. *Controller*

Kao što je prije navedeno, *controller* stvara vezu između *modela* i *view-a*. Omogućava korisniku da zahtjevom koji je kreiran kroz grafičko sučelje view-a, određenom logikom prikaže zatražene podatke sadržane unutar modela na *view-u*.



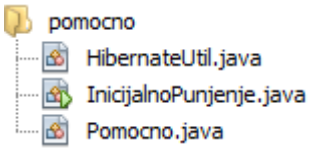
Sl. 3.4. *Klase definirane unutar controller paketa*

Controller ove aplikacije sastoji se od obrade specifične za svaku od klasa modela. Razlog tome je raspodjela logike za svaku od klasa modela zasebno, jer time je povećana preglednost koda. Kako bi bilo omogućeno spremanje, brisanje ili mijenjanje podataka, kreirana je klasa *Obrada*. Unutar te klase definirane su metode pomoću kojih se izvršavaju navedene akcije. Klasa *Obrada* generičkog je tipa što znači da njene metode mogu biti primijenjene na sve klase koje ju pozovu. Svojstvo klase je prikazano u primjeru 3.2.:

```
public class Obrada<T extends Entitet> {

    private final Session session;

    public Obrada() {
        session = HibernateUtil.getSession();
     }
```

Primjer koda 3.2. *Svojstvo klase Obrada*

Kako bi se prilikom kreiranja instance klase *Obrada* definiralo i njeno svojstvo, unutar konstruktora poziva se *HibernateUtil.getSession()* metoda. Ta metoda kao rezultat vraća session koja predstavlja sesiju pomoću koje se vrši spremanje, brisanje i promjena podataka. *getSession()* metoda definirana je unutar klase *HibernateUtil*. Ta klasa sadrži statičko svojstvo *session* koje je tipa Sesson. Također spomenuta metoda getSession() definirana je kao statička, što omogućava njeno pozivanje bez potrebe instanciranja *HibernateUtil* klase.



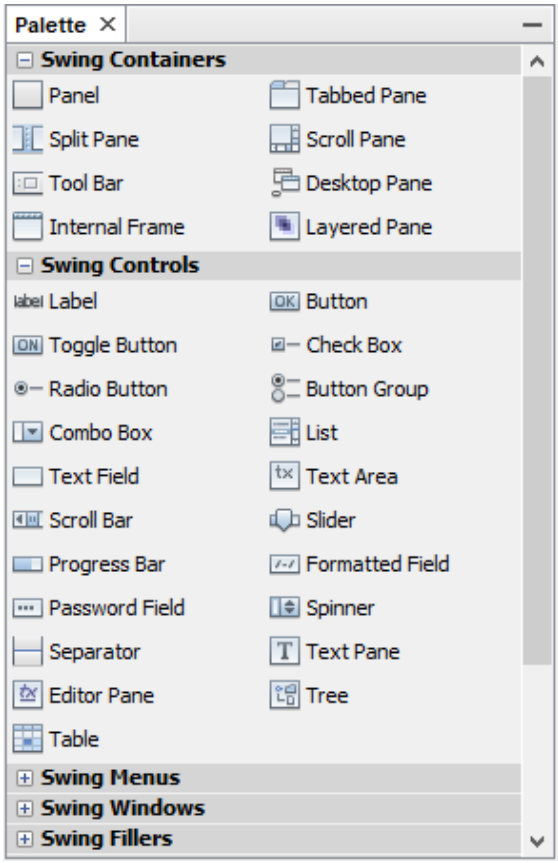
Sl. 3.5. *Klase definirane unutar paketa* pomocno

HibernateUtil zapravo predstavlja *Java Singleton* klasu koja stvara samo jednu svoju instancu. Nakon instanciranja klase *Obrada*, omogućene su navedene akcije nad podacima uz pomoć instance klase *„HibernateUtil“*. Ukoliko je potrebno spremiti zapis unutar baze, poziva se metoda *save(T entitet)* koja kao argument prima podatak koji je potrebno spremiti. Ukoliko je podatku proslijeđenom metodi šifra jednaka nuli, znači da je taj podatak novo kreirani te se podešava datum kreiranja na trenutni. Također ukoliko mu je datum brisanja jednak nuli, radi se o promjeni postojećeg, te mu se postavlja datum promjene na trenutni. To znači da metoda *save(T entitet)* kreira novog u bazi ili mijenja postojećeg. Ukoliko je potrebno brisanje podataka, poziva se metoda *delete(T entitet)* koja objektu postavlja datum brisanja na trenutni, te vrijednost obrisan na *true*. Time podaci nisu zaista obrisani iz baze, već prilikom dohvaćanja podataka u upitima se definira dohvaćanje onih kojima svojstvo obrisano ima vrijednost *false*. Klasa sadrži i metodu *save(List<T> lista)* koja kao argument prima liste objekata, i omogućava njihovo spremanje. Za kreiranje upita na bazu radi dohvaćanja podataka, klasa sadrži metodu *createQuery(String hql)* koja izvršava proslijeđeni upit na bazu podataka te za rezultat vraća listu određenog tipa.

Kako bi ostale klase *controller* paketa mogle zasebno spremati ili mijenjati objekte zbog kojih su kreirane, u sebi imaju instancu klase *Obrada*. Ostale metode u tim klasama služe za dohvaćanje objekata iz baze podataka pomoću metode *createQuery(String hql).*

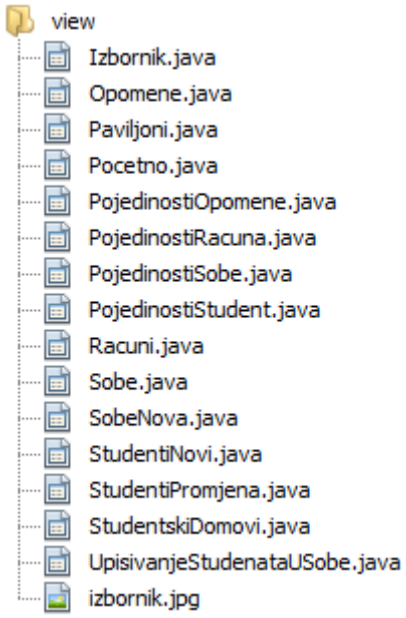
3.4. *View*

Kako bi korisnik što lakše manipulirao nad podacima, potrebno je izgraditi što bolje korisničko sučelje *(engl. User Interface,* UI*).* Zbog toga Java sadrži biblioteku za kreiranje grafičkog sučelja pod nazivom *Swing*. Ona pruža skup elementarnih komponenti *UI-a*, uz očuvanje osnovne ideje da razvoj sučelja bude neovisan od operativnog sistema krajnjeg korisnika. Pokrene li se isti program na *Windows* ili *Mac* operativnom sustavu, *GUI* komponente će izgledati u skladu sa odgovarajućim operativnim sustavom. *Swing* pruža sve što je neophodno za kreiranje *UI-a* te sadržava kontrole kao što su tipke, mreže, liste, padajuće menije i sl.



Sl. 3.6. *Swing komponente koje pruža IDE NetBeans*

Korišteni IDE (engl. Integrated development envirement, IDE) za izradu klijentske aplikacije je *NetBeans*. Izrada korisničkog sučelja funkcionira na principu „*drag and drop*“ što znači da se željena komponenta „uhvati“ pomoću miša te se postavi na željeno mjesto na kontejneru. Pod kontejnere spadaju *JPanel* ili *JFrame* koji podržavaju različite rasporede elemenata što omogućava da se kontrole rasporede onako kako je zamislio grafički dizajner. Slika 3.6. prikazuje sve komponente koje pruža *Swing* biblioteka.



Sl. 3.7. *Klase unutar view paketa*

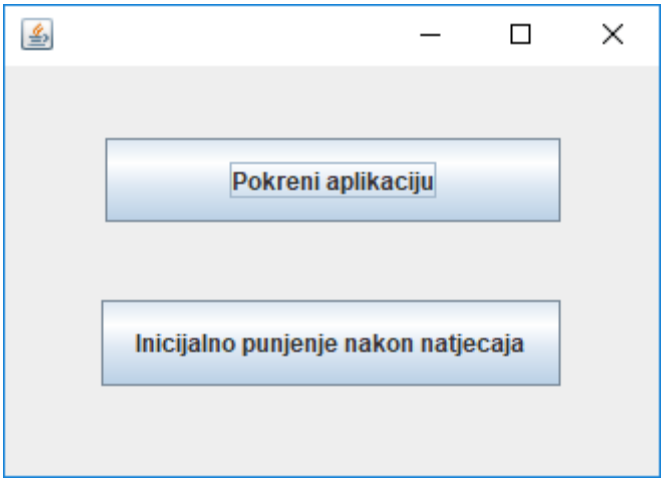
Slika 3.7. prikazuje strukturu *view* paketa. Zbog lokacije *Swing* klase *unutar javax.swing* paketa, svaka klasa prikazana na slici 3.7. nasljeđuje *„javax.swing.JFrame“*. Prozor najviše razine, zajedno s naslovnom linijom i rubom, kreira se na način da se kreira instanca *JFrame* klase. Tako je kreiran kontejner, odnosno *UI* koji uz određenu kombinaciju *Swing* komponenti korisniku omogućava jednostavan pregled objekata, te manipulaciju nad istim. Svaki od *JFrame-ova*, u kombinaciji s *JButton*, *JList*, *JLable* i slično omogućava da korisnik dohvati željene podatke, oni mu se prikažu, te na njegov zahtjev kroz *UI* izvrši se spremanje, promjena ili brisanje podataka. *UI* funkcionira na način da se u *JList-e* dohvate i prikažu željene rezultate, a pomoću *JButton-a* i *JTextField-a* u kombinaciji s *JComboBox-om* vrši kreiranje novih, brisanje i mijenjanje. Zbog toga *Swing* klasa sadrži *ActionListener* koji služi kao „indikator“ događaja. Primjer koda 3.3. prikazuje izvršavanje metode ukoliko *ActionListener* prepozna je pritisnuto na tipku dodavanja novog paviljona i bazu podataka.

```
private void btnNoviActionPerformed(java.awt.event.ActionEvent evt) {
      //Određene provjere ……
      try {
            obradaPaviljon.spremi(paviljon);
      } catch (StucException e) {
            //Ispis pogrešku ……
            return;
      }
}
```

Primjer koda 3.3. *Pritiskom na tipku pokreće se kod unutar metode „btnNoviActionPerformed“*

3.5. Dohvat podataka s *REST* Servisa

Kako bi prethodno kreirane aplikacije tvorile distribuiranu aplikaciju potrebno ih je međusobno povezati. Povezivanje se vrši preko *API-a* koji iz *REST* servisa preuzima podatke te ih sprema lokalno za daljnju uporabu. *API* predstavlja softverskog posrednika koji omogućava komunikaciju između dvije nezavisne aplikacije. Kao što je objašnjeno u poglavlju 1, *REST* servis pozivanjem određenog *URL-a* kao rezultat vraća sve zapise studenata koji se nalaze u bazi podataka na serveru. Rezultat koji *REST* servis daje zapisan je u *JSON* formatu. Zbog toga potrebno je unutar projekta uvesti (eng. *import*) *GSON driver* koji omogućava pretvaranje *JSON* zapisa u odgovarajući objekt i obrnuto. Pri pokretanju aplikacije otvara se *JFrame* koji sadrži dvije tipke (Sl. 3.8.).



Sl. 3.8. *JFrame za odabir pokretanja aplikacije ili inicijalnog punjenja nakon natječaja*

Klikom na tipku označenu s „Pokreni aplikaciju“ pokreće se glavni izvornik aplikacije. Ukoliko se odabere na „Inicijalno punjenje baze nakon natječaja, *ActionListener* prepoznaje da je došlo do pojave događaja te izvršava metodu *btnInicijalnoPunjenjeActionPerformed()*. Ulaskom u metodu izvršava se sljedeći kod:

```
if(obradaStudent.imaLiStudenataUDomu()){
      JOptionPane.showMessageDialog(getRootPane(), "U Dom su već uneseni " +
      "studenti nakon natječaja!");
      return;
}
```

Primjer koda 3.4. *Provjera postojanja studenata unutar studentskog doma*

Kod iz primjera 3.4. pokazuje izvršavanje provjere postoje li unutar studentskog doma studenti. Provjera se izvršava na način da se kroz instancu klase *ObradaStudent* pozove metoda *imaLiStudenataUDomu()* koja je tipa *boolena* i koja pomoću *createQuery(String hql)* radi *HQL (Hibernate Query Language)* upit na bazu. Upit kao rezultat vraća broj studenata unutar doma i ukoliko je broj jednak nuli, metoda vraća *false* vrijednost te program ne ulazi u petlju iz primjera koda 3.4. Ukoliko metoda vrati *true*, znači da unutar studentskog doma već ima studenata, a to ujedno znači da je inicijalno punjenje već izvršeno. O tome se obavještava korisnik aplikacije pomoću *JOptionPane* poruke te se pomoću *return* naredbe prekida izvođenje metode *btnInicijalnoPunjenjeActionPerformed().* Nakon toga slijedi instanciranje objekata koji su potrebni za dohvaćanje i obradu podataka preko *API-a*.

```
//String koji predstavlja URL konvertira se u URL
URL url = null;
//gson objekt sluzi za konvertiranje iz JSON u Java objekt
Gson gson = new Gson();
//pretvara byte stream u znakovni
InputStreamReader reader = null;
//Instanca JsonElementa
JsonElement json = null;
```

Primjer koda 3.5. *Instanciranje potrebnih objekata*

Zbog izbjegavanja prekida programa uslijed dohvaćanja podataka putem *API-a* uzrokovano iznimkama koristi se *try-catch* blok.

```
try {
      url = new
      URL("https://ivancizmar1.000webhostapp.com/api/student/read.php");
      reader = new InputStreamReader(url.openStream());
      json = gson.fromJson(reader, JsonElement.class);
} catch (IOException ex) {
      Logger.getLogger(Pocetno.class.getName()).log(Level.SEVERE, null, ex);
}
```

Primjer koda 3.6. *Dohvat podataka preko API-a*

Prije svega, iz Stringa koji predstavlja *URL* prema *REST* servisu koji vraća podatke u *JSON* formatu, kreira se nova instanca klase *URL* u prije inicijaliziranu varijablu *url*. Pozvani url vraća rezultat koji je pomoću *InputStreamReader()* klase, iz *byte streama* (otvara se metodom *openStream()*) pretvara se u *stream* znakova te se sprema u prethodno instanciran objekt *reader*. Zatim se *stream* znakova pohranjenih unutar *reader* instance po uzorku na *JsonElement.class* pretvara u *JSON* zapis pomoću metode *fromJson()*. Na taj način podaci koje vraća *REST* servis u *JSON* obliku, preko *API-a* su dohvaćeni i pohranjeni u instancu *JsonElement* klase.

Kako se dohvaćeni podaci ne bi izgubili, potrebno ih je konvertirati u odgovarajuće objekte i spremili u lokalnu bazu podataka.

```
List<Student> listOfStudents = gson.fromJson(json,
      new TypeToken<List<Student>>(){}.getType());
listOfStudents.forEach(x -> {
      obrada.save(x);
});
JOptionPane.showMessageDialog(getRootPane(), "Unjeli ste " +
listOfStudents.size() +
            " studenata u bazu!");
new UpisivanjeStudenataUSobe().setVisible(true);
```

Primjer koda 3.7. *Pretvorba objekata studenta zapisanih u JSON formatu u listu objekata tipa*

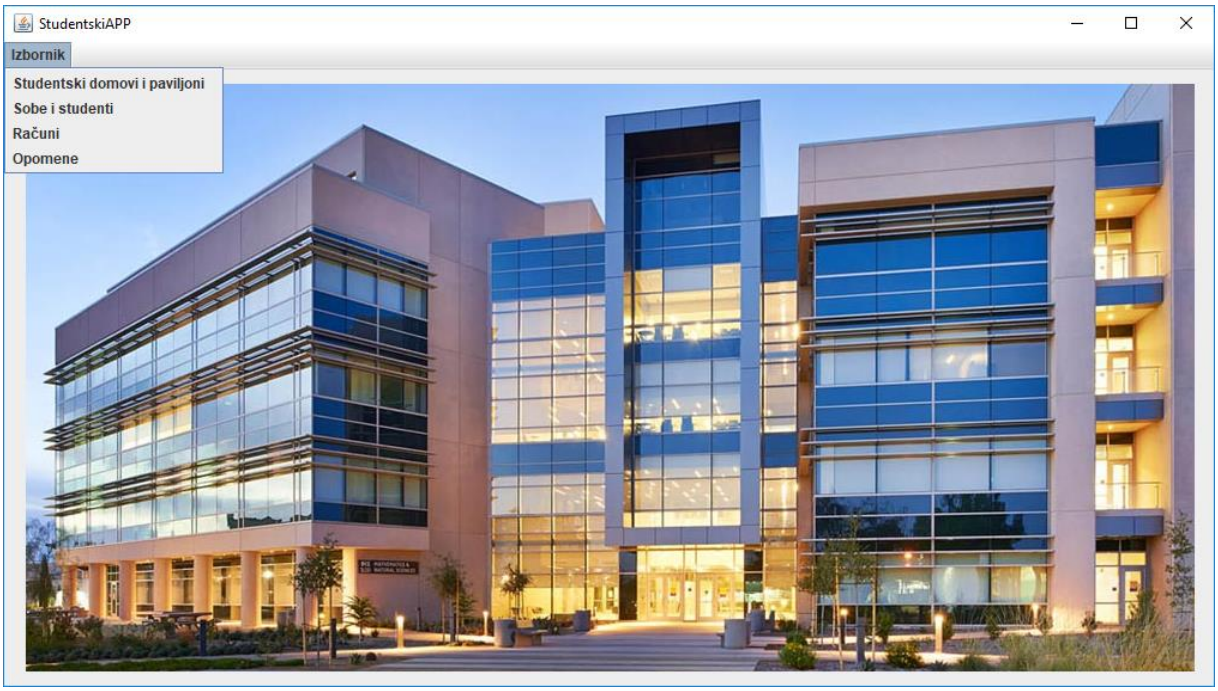
Pomoću generičke *TypeToken* klase koja vrši se konverzija *JSON* zapisa podataka u željeni objekt. Taj proces se izvršava na način da se klasi *TypeToken* proslijedi tip parametra u koji se treba konvertirati, a to je lista studenata *(List<Student>)* te pozivanjem metode *getType()* svaki element *JSON* zapisa parsira se u objekt klase Student i sprema se u listu. Zatim svaki element kreirane liste studenata se pomoću *forEach()* lambda izraza sprema u bazu podataka pozivanjem *save(T entitet)* metode iz klase *Obrada*. Na kraju procesa korisnik se obavještava koliko je studenta preuzeto i spremljeno u lokalnu bazu, te se otvara *JFrame* pomoću kojeg se izvrši upisivanje spremljenih studenata u sobe po izboru.

4. PRINCIP RADA *JAVA* KLIJENT APLIKACIJE

Java klijentska aplikacija nudi jednostavnu manipulaciju nad podacima. Kombinacijom *JFrame* kao kontejnera i odgovarajućih elemenata Swing biblioteke moguće je vršiti pregled svih podataka koji su upisani u bazu. Isto tako moguće je vršiti *CRUD (Create, Read, Update, Delete)* operacije gotovo nad svim modelima. Vrlo je pregledna, i time jednostavna za korištenje. Omogućeno je praćenje evidencije svih procesa koji se događaju unutar jednog studentsko doma. Pri samom pokretanu *main* metode potrebno je pozivanje *HibernateUtil.getSession()* kako bi se podigla *ORM-ova* sesija, i njegovi podaci.

4.1. Izbornik

Pokretanjem aplikacije otvara se prozor koji predstavlja glavni izbornik aplikacije (Sl. 4.1.). Prozor zapravo predstavlja *JFrame*, koji na sebi ima *JPanel* i padajući izbornik.



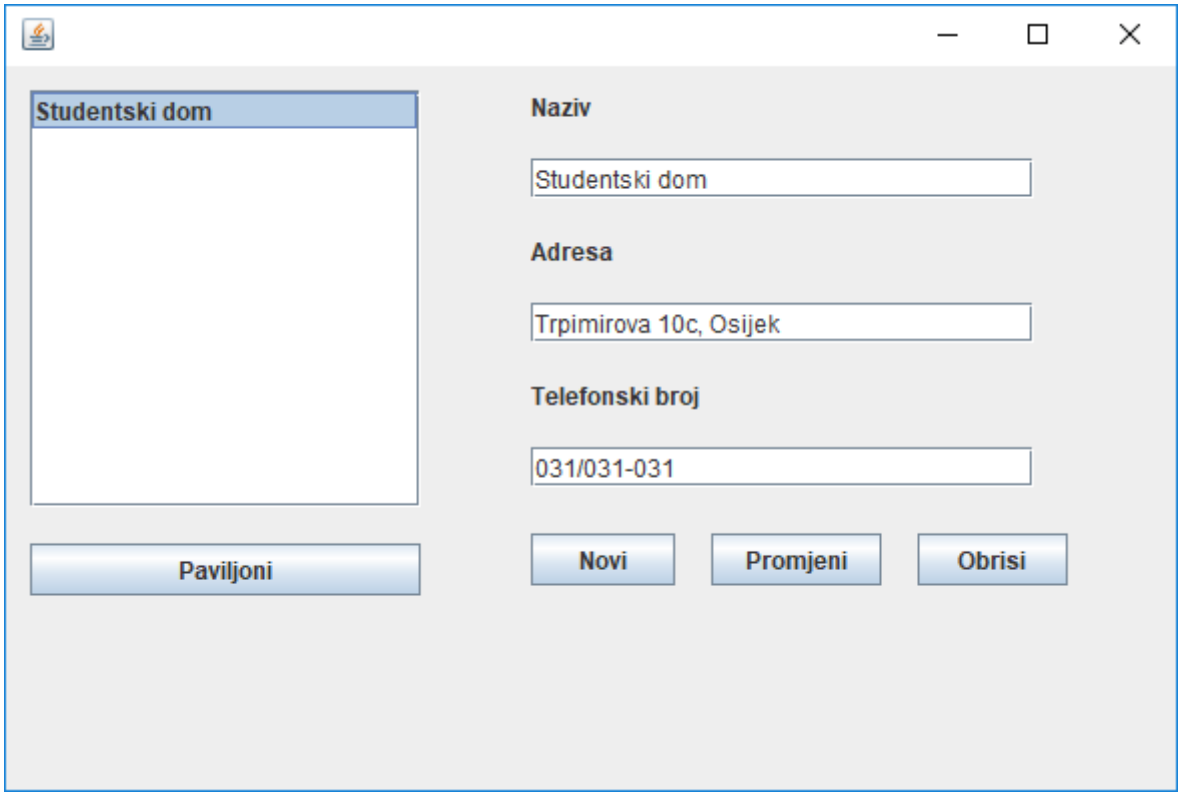
Sl. 4.1. *Glavni izbornik*

JPanel služi kao kontejner, te radi ljepšeg izgleda izbornik na njega je postavljena slika studentskog doma koja je u .jpg formatu. Kako bi se omogućio pristup svim ostalim *JFrame-ovima* izbornik sadrži *JMenuBar* na kojem se nalazi *JMenu*. *JMenu* predstavlja padajući izbornik

unutar kojeg se nalaze četiri *JMenuItem-a*. Svaki od njoj vodič je do ostalih *JFrame-ova* namijenjenih za prikaz i manipulaciju nad podacima. Na svakom od *JMenuItem-a* uključen je *actionPerformer* koji pomoću *ActionListener-a* „osluškuje“ događaj nad njim, te pomoću metode *setVisible(true)* pali odabrani JFrame. Kako bi se aplikacija zaustavila nakon zatvaranja izbornika, svojstvo *JFrame-a* pod nazivom *defaultCloseOperation* mora biti podešeno na vrijednost *EXIT_ON_CLOSE*.

4.2. Prozor studentskog doma

Pritiskom na prvi *JMenuItem* u padajućem izborniku unutar glavnog izbornika sa slike 4.1. otvara se prozor sa slike 4.2. Prozor predstavlja *JFrame* koji služi za *CRUD* nad studentskim domovima. Unutar konstruktora *JFrame* nalaze se metode za inicijalizaciju odabranih *Swing* komponenti, centriranje *JFrame-a* na zaslonu, instanciranje obrada radi mogućnosti manipulacije nad podacima te metoda *ucitajPodatke()*.



Sl. 4.2. *Prozor namijenjen za rad sa studentskim domovima*

JFrame sastoji se od *JList-e,* tri *JTextField-a* i četiri *JButton-a*. *JList* „komunicira“ s klasom *ObradaStudentskiDom* koja se nalazi unutar *controller* paketa. Komunicira na način da kroz instancu klase *ObradaStudentskiDom* poziva metodu koja pomoću *HQL* upita dohvaća sve studentske domove koji su zapisani unutar baze podataka kao što je to prikazano u primjeru koda 4.1. Cijeli proces sadržan je unutar prije navedene metode *ucitajPodatke()* kako bi se već prilikom pokretanja prozora za upravljanje studentskim domovima oni prikazali unutar *JList-e*.

```
//Kod unutar JFrame-a
List<StudentskiDom> listaStudentskihDomova =
obradaStudentskiDom.getSviDomovi();
.....
/*metoda koja vraća sve studentske domove i nalazi se unutar
ObradaStudentskiDom klase*/
public List<StudentskiDom> getSviDomovi() {
      Session s = HibernateUtil.getSession();
      s.clear();
      return s.createQuery(
                  "from StudentskiDom a where a.obrisano = false ").list();
}
```

Primjer koda 4.1. *Pozivanje metode „getSviDomovi()“ koja pomoću HQL upita vraća sve postojeće studentske domove*

Nakon što metoda *getSviDomovi()* vrati rezultat kao *List<StudentskiDomovi>*, vraćenu listu potrebno je pretvoriti u *DefaultListModel* kako bi studentske domove bilo moguće prikazati kroz *JList*. Kako bi se izvršila određena radnja, ukoliko se na *JList-i* odabere neki od prikazanih domova, potrebno je za njen događaj (eng. *Events*) *valueChanged* podesiti vrijednost *listaValueChanged*. Time je omogućeno da se odabirom željenog studentskog doma, unutar *JTextField-a,* ispišu svojstva specifična za odabrani studentski dom (Sl. 4.2.). Ukoliko se želi unijeti novi studentski dom u bazu podataka, potrebno je unijeti željene podatke unutar tekstualnih polja, te pritisnuti tipku *Novi*. Time *ActionListener* prepozna događaj i pokrene metodu *btnDodajNoviActionPerformed()*. Prvi korak metode je uz pomoć metode *kontrola()* provjeriti jesu li sva tekstualna polja ispravno popunjena. Ukoliko je neko od polja prazno, metoda vraća *false* vrijednost, korisnik se obavještava o tome tako da se obrub tekstualnog polja koje nije ispravno popunjeno oboji u crveno (Primjer koda 4.2.), te se prekida izvršenje metode *btnDodajNoviActionPerformed()* pomoću *return* naredbe.

```
//Metoda za provjeru tekstulnih polja
private boolean kontrola() {
      if(txtNaziv.getText().trim().length() == 0){
            oznaciPogresku(txtNaziv);
            return false;
      }
      ......//Postupak je isti za svaki JTextField
      return true;
}
//Metoda koja boja obrub tekstualnog polja
public void oznaciPogresku(JTextField polje){
      polje.setBorder(BorderFactory.createLineBorder(
      Color.decode("#FF0000")));
      polje.requestFocus();
}
```

Primjer koda 4.2. *Provjera popunjenosti tekstualnih polja i bojanje obruba istih ukoliko su prazna*

U slučaju da kontrola vrati vrijednost *true*, instancira se novi objekt klase *StudentskiDom*, tom objektu se postavljaju svojstva iz tekstualnih polja pomoću *Setter* metoda, te se pomoću *save(T entitet)* metode sprema unutar baze podataka.

Ukoliko je potrebna promjena postojećeg, potrebno je odabrati odgovarajući studentski dom sa *JList-e*. U tekstualnim poljima pojave se vrijednosti njegovih svojstava. Potrebno je izmijeniti svojstvo na željeni način te pritisnuti tipku *Promjeni*. Time se odabrani studentski dom sprema u novu instancu. Provjerava se je li on jednak *null* vrijednosti (je li pravilno odabran) ili nije unesen naziv unutar tekstualnog polja. Ukoliko je proces izveden ispravno, poziva se metoda *save(T entitet)* koja prepoznaje da se radi o već postojećem domu, dodaje mu novi datum promjene, te vrši promjene unutar baze podataka sukladno onima sadržanim u proslijeđenom objektu studentskog doma.

Kako bi se izvršio proces brisanja, potrebno je odabrati studentski dom za koji želimo izvršiti tu radnju. Ukoliko dom nije odabran, korisnik se o tome obavještava putem *JOptionPane* skočnog prozora.

```
//Indikator
boolean mozeObrisati=true;
//Provjer postoji li unutar studentskog doma kreirani Paviljon
for (Paviljon p : sd.getPaviljoni()) {
      if(!p.isObrisano()){
            mozeObrisati=false;
            break;
      }
}
```

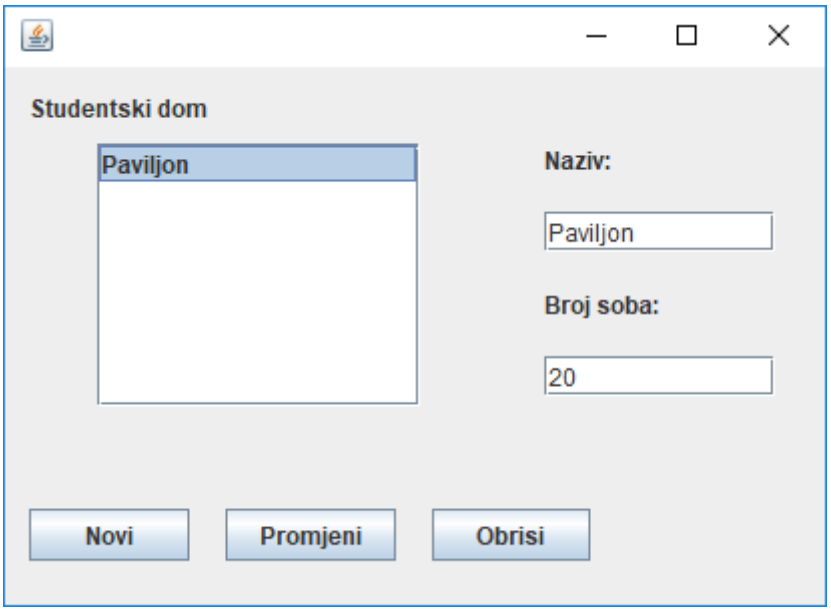
Primjer koda 4.3. *Provjera postoji li kreiran paviljon unutar studentskog doma*

Da bi bilo moguće obrisati studentski dom, potrebno je napraviti provjeru postojanja paviljona unutar istog (Primjer koda 4.3.). Ako odabrani studentski dom unutar svoje liste paviljona ima sadržan paviljon, studentski dom je nemoguće obrisati ukoliko se prethodno ne obrisa paviljon koji se nalazi u njegovoj listi paviljona.

JFrame sadrži i tipku *Paviljoni* koja vodi do svih paviljona koji se nalaze u odabranom studentskom domu. To znači da je potrebno odabrati studentski dom za koji se žele urediti paviljoni kako bi se mogao otvoriti *JFrame* namijenjen isključivo tome. Za potrebe demonstracije rada unesen je novi studentski dom sa svojstvima kao sa slike 4.2.

4.3. Prozor za upravljanje paviljonima

Pritiskom na tipku *Paviljoni* sa slike 4.2. otvara se prozor koji nudi mogućnost *CRUD* akcija nad paviljonima koji se nalaze unutar prethodno odabranog studentskog doma. Pri samom otvaranju prozora iz baze podataka se dohvate svi paviljoni za odabrani studentski dom unutar *JList-e.*



Slika 4.3. *Prozor za rad s paviljonima*

JFrame na sebi sadrži jednu listu, tri tipke te dva tekstualna polja. U lijevom vrhu prozora ispisuje se studentski dom za kojeg se rade promjene paviljona. To je omogućeno prosljeđivanjem prethodno odabranog studentskog doma kroz konstruktor *Paviljoni(StudentskiDom studentskiDom)* prilikom pokretanja samog *JFrame-a*. Proces kreiranja

novog, promjene i brisanja jednak je kao i za studentske domove. Razlika je u kontroli prilikom izvođenja akcija.

Unutar paketa *contoller* nalazi se klasa koja se naziva *StucException*. To je klasa koja nasljeđuje klasu *Exception* te kao svojstvo ima *String* koji se naziva *komponenta*. Također sadrži i parametrizirani konstruktor koji prima dva parametra: *String* poruka te *String* komponenta. Unutar konstruktora, poruka se prosljeđuje *nadklasi* naredbom *super(poruka).*

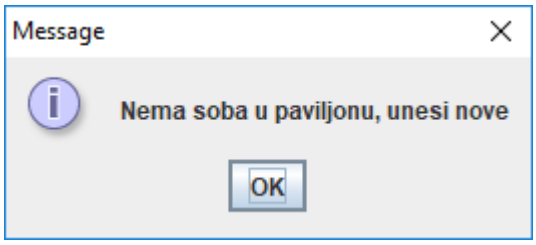
```
try {
      obradaPaviljon.spremi(pav);
} catch (StucException e) {
      switch(e.getKomponenta()){
            case "naziv":
                  txtNaziv.requestFocus();
                  break;
      }
      JOptionPane.showMessageDialog(getRootPane(), e.getMessage());
      return;
}
```

Primjer koda 4.4. *Primjena StucException radi provjere ispravnosti popunjenosti naziva paviljona*

Pritiskom na tipku *Novi*, izvršava se „punjenje“ objekta s podacima iz tekstualnih polja te se u svojstvo *studentskiDom* postavlja proslijeđeni studentski dom. Slijedi *try-catch* blok kao iz primjera koda 4.3. Pomoću obrade specifične za paviljon, pokušava se spremiti novo kreirani paviljon. Ako je tekstualno polje *txtNaziv* neispravno popunjeno, program će javiti iznimku te kursor za unos teksta postaviti unutar tekstualnog polja *txtNaziv*. Na taj način korisnik je izoliran od krivog unosa naziva paviljona. Prije promjene postojećeg, kontrolu čini samo provjera jesu li tekstualna polja prazna. Ukoliko je barem jedno od njih prazno, promjena se prekida. U suprotnom, promjena se izvršava te se o tome obavještava korisnik. Ako je potrebno obrisati neki od paviljona, potrebno ga je odabrati, i pritisnuti tipku *Obrisi*. Nakon toga vrši se provjera kao u primjeru koda 4.4., te ukoliko u paviljonu nema soba, aplikacija pomoću *JOptionPane Confirm Dialog-a* nudi mogućnost odustajanja od brisanja ili potvrdu istog. Radi demonstracije rada unutar prethodno kreiranog studentskog doma, unesen je paviljon pod nazivom *Paviljon*.

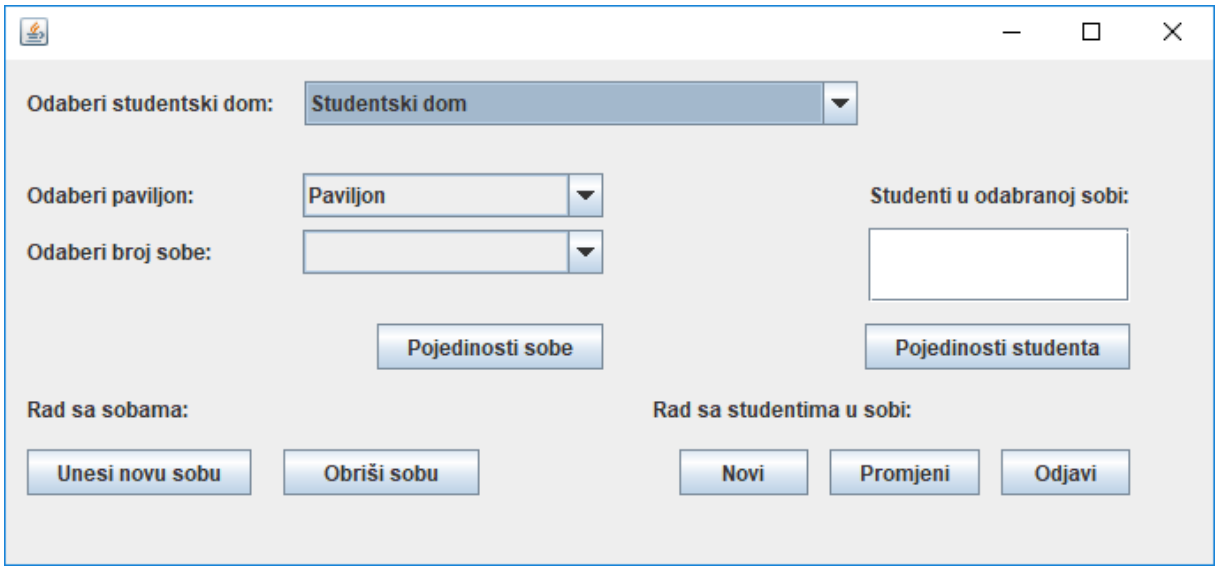
4.4. Prozor za upravljanje sobama i studentima

Nakon unesenog paviljona u studentski dom, potrebno je unijeti i sobe, kako bi se omogućilo upisivanje studenata koji su uneseni u bazu podataka uz pomoć *API-a*.



Sl. 4.4. *Obavijest o popunjenosti paviljona pri pokretanju prozora za upravljanje soba i studenata*

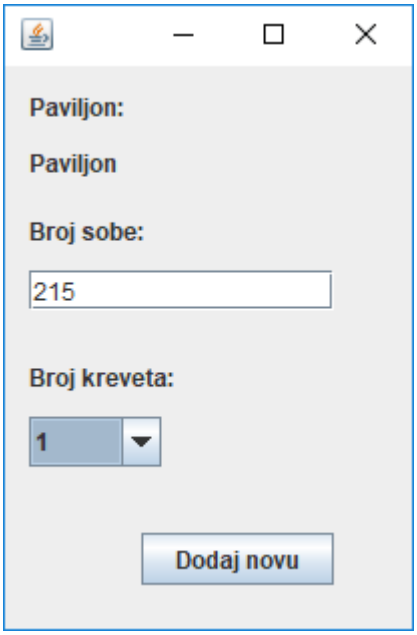
S obzirom da prethodno nije unesena niti jedna soba u tek kreirani paviljon, korisnik je o tome obaviješten (Sl. 4.4.). Pritiskom na *OK* tipku, otvara se prozor za upravljanje sobama i studentima unutar tih soba (Sl. 4.5.).



Sl. 4.5. *Prozor za upravljanje sobama i studentima*

Za razliku od prijašnjih *JFrame-ova*, ovaj sadrži i *JComboBox Swing* komponente. Na taj način je korisniku ograničen izbor i izvršena je prevencija od pogrešaka. Pri samom pokretanju pomoću *obradaStudentskiDom* i *obradaPaviljoni* napune se *JComboBox-ovi*. Prvo se u

JComboBox, namjenjen za studentske domove *(cmbStudentskiDom)*, učitaju svi studentski domovi iz baze podataka te se automatski odabere prvi s liste. Zatim se, pomoću *ActionListener-a* koji osluškuje promjene na *cmbStudentskiDom*, aktivira metoda koja puni *JComboBox* namijenjen za paviljone *(cmbPaviljon)* na način da se metodi koja dohvaća paviljone proslijedi odabrani studenti dom. Prosljeđuje se zbog toga što je u *cmbPaviljoni* potrebno dohvatiti samo one paviljone koji se nalaze u odabranom studentskom domu. Proslijeđeni studentski dom se uvrštava u *where* dio *HQL* upita. Nakon završenog punjenja *cmbPaviljon*, njegovim *ActionListenerom* aktivira se punjene *JComboBox-a* namijenjenom za prikaz soba na isti način kao i paviljona. Pri završetku punjenja soba, automatski se odabire prva. *ActionListener* prepozna događaj te za odabranu sobu vrati studente koje prikaže unutar *JList-e*. Cijeli taj lanac procesa učitavanja pokrenut je metodom *napuniDomove()* koja se nalazi u konstruktoru klase *Sobe*. Na taj način je omogućen pristup podacima pri samom otvaranju prozora sa slike 4.5. S obzirom da unutar odabranog paviljona nema kreiranih soba, te je lista studenata prazna, potrebno je unijeti sobe u odabrani paviljon. Taj proces započinje pritiskom na tipku *Unesi novu sobu*. Pritiskom na tipku, otvara se novi *JFrame* koji kroz parametar prima odabrani paviljon (Sl. 4.6.). To znači da će se novonastala soba nalaziti unutar tog paviljona.



Sl. 4.6. *Prozor za unos nove sobe u odabrani paviljon*

U gornjem lijevom uglu *JFrame* prikazan je paviljon unutar kojeg će soba biti spremljena. Prije unosa nove sobe potrebno je unijeti broj sobe te koliko kreveta soba ima. Pritiskom na tipku *Dodaj novu* kreira se nova instance klase *Soba* kojoj se svojstva „pune“ pomoću *Setter* metoda. Nakon toga vrši se kontrola svojstva sobe primjernom metode *provjeri()* (Primjer koda 4.4.).

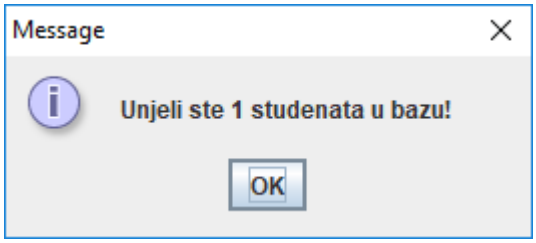
```
boolean kontrola = false;
kontrola = provjeri(soba);
if(!kontrola){
      return;
}
```

Primjer koda 4.5. *Pozivanje metode provjera() radi provjere ispravnosti unosa nove sobe u paviljon*

Metoda *provjeri() boolean* je tipa, što je i vidljivo iz primjera koda 4.5. Najprije metoda provjerava je li kreirana nova instanca klase *Soba* ili je ona jednaka *null* vrijednosti. Zatim provjera sadrži li kreirana soba njen broj i broj kreveta unutar nje. Ukoliko sve kontrole prođu, metoda vraća *true* i kreira se nova soba. Kako bi se nakon kreiranja novih soba one odmah učitale u *cmbSobe*, kroz konstruktor *SobeNova.class* prosljeđuje se cijela klasa *Sobe*. Unutar samog konstruktora se instancira te se nakon kreiranja nove sobe poziva metoda *napuniSobe()* kako bi se automatski unutar *cmbSobe* učitala i novo kreirana soba.

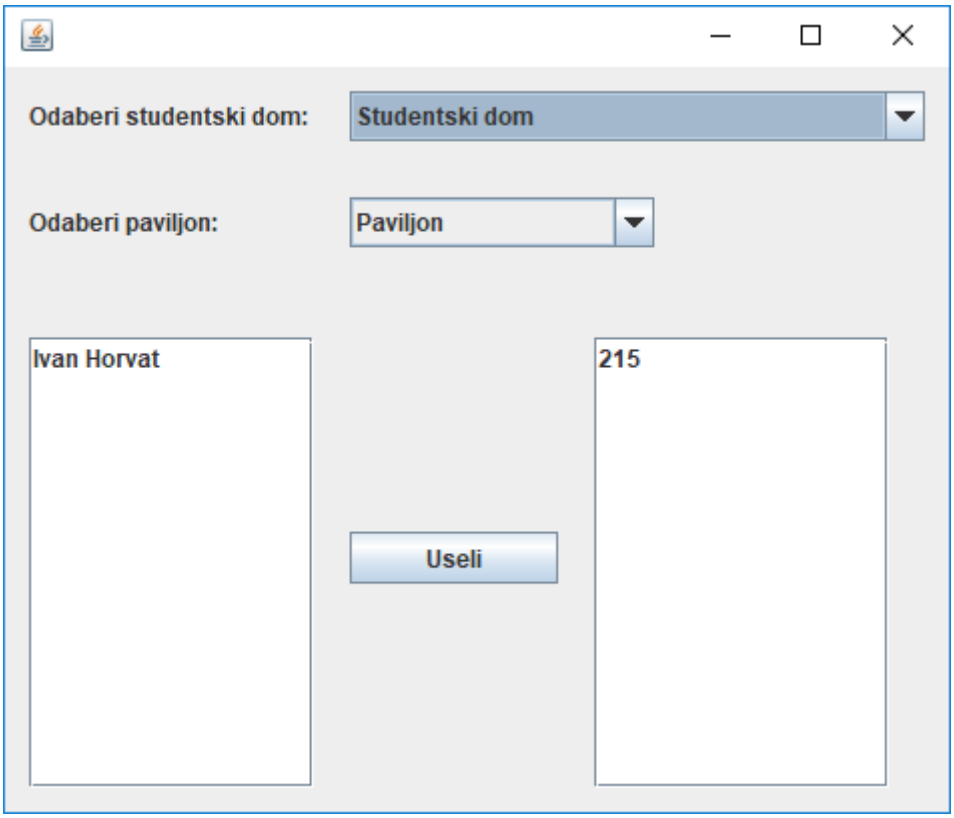
4.5. Spajanje na *REST API*

Prije svake akademske godine potrebno je izvršiti ponovno upisivanje studenata u studentske domove. Po završetku natječaja za smještaj u studentske domove, objavljuju se liste studenata koji su ostvarili pravo na smještaj u studentskom domu. Potrebno je preuzeti tu listu studenata te im dati na izbor sobe u koje se žele upisati.



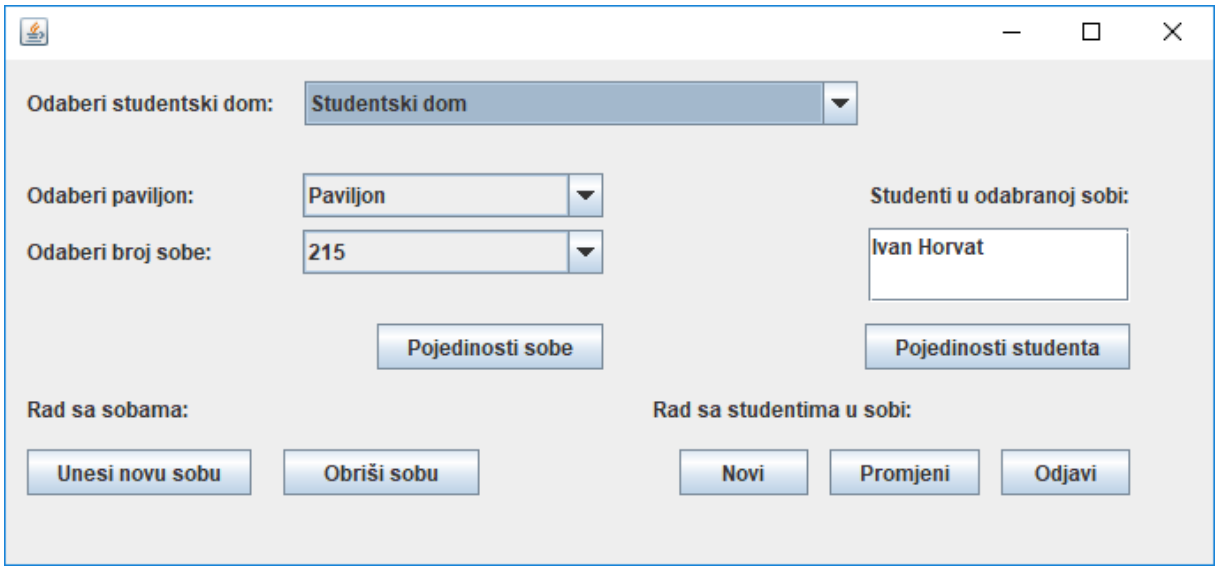
Sl. 4.7. *Obavijest o završetku i broju studenata zapisanih u bazu podataka pomoću API-a*

Pritiskom na tipku *Inicijalno punjenje* nakon natječaja sa slike 3.8. poziva se prethodno kreirani *API* koji dohvaća sve studente koji su ostvarili pravo na smještaj u studentskom domu. U ovom slučaju preuzet je samo jedan student te je korisnik o tome obaviješten. Pritiskom na tipku *OK*, otvara se prozor kao sa slike 4.8.



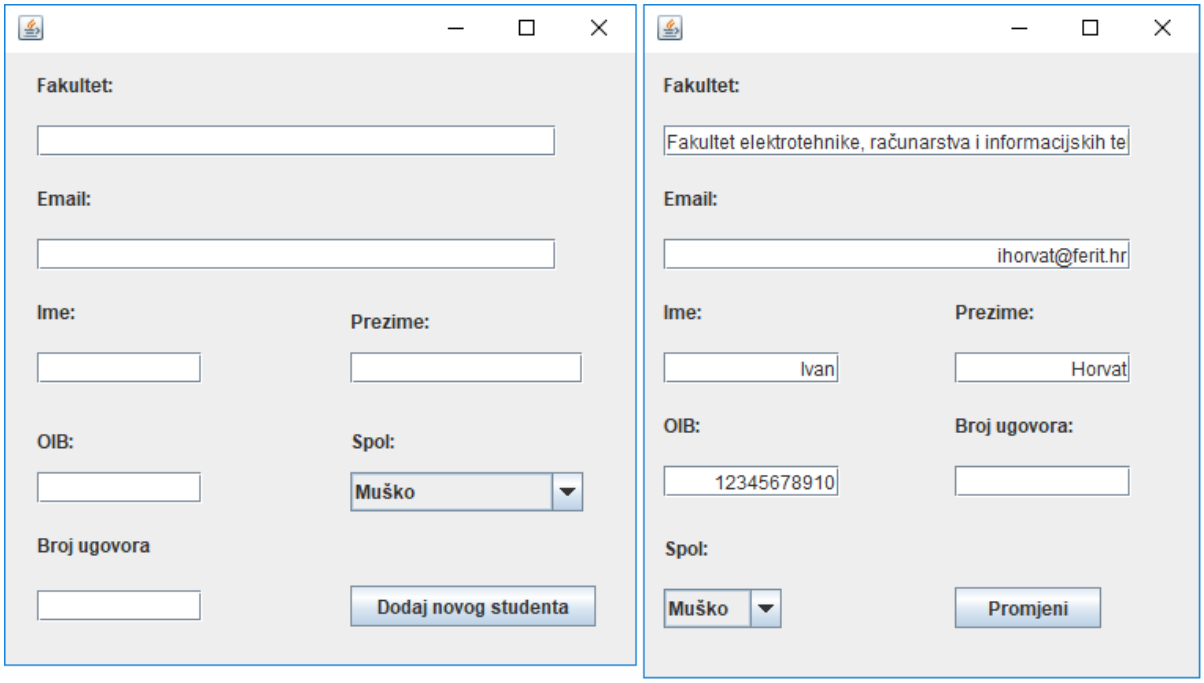
Sl. 4.8. *Prozor koji omogućava upisivanje studenata u željenu sobu*

Prozor predstavlja *JFrame* koji sadrži dva *JCombobox-a*, dvije *JListe* i jedan *JButton*. Pri samom pokretanju, *Combo Box* puni se svim studentskim domovima unutar baze podataka, automatski se odabire prvi i time se puni drugi *Combo Box* sa svim paviljonima unutar tog studentskog doma. Kada se završi punjenje paviljona, automatski se odabire prvi te se u listi s desne strane pojavljuju sve sobe iz odabranog paviljon, ali kojima je broj studenata unutar njih manji od broja kreveta. Paralelno tome, unutar lijeve *JList-e* prikazuju se svi studenti preuzeti s *REST* servisa koji nemaju zapisanu sobu unutar svog svojstva. Ukoliko student (u ovom slučaju Ivan Horvat) želi useliti u sobu 215 potrebno je označiti i studenta i željenu sobu. Nakon toga, potrebno je pritisnuti na tipku *Useli* kako bi se student uselio u odabranu sobu. Nakon pritiska na tipku *Useli*, pod svojstvo *„soba“* unutar studenta upisuje se odabrana soba, dok je za sobu potrebno dodati odabranog studenta u listu studenata koji se nalaze u toj sobi. Nakon toga i student i soba u koju je upisan će nestati s lista jer student ima unesenu sobu unutar svojstva, a prethodno kreirana soba ima samo jedan krevet.



Sl. 4.9. *Pojava studenta unutar sobe nakon upisa u željenu sobu*

Ponovnim otvaranjem prozora za upravljanje sobama, može se zamijetiti da se student pojavljuje na listi koja prikazuje studente unutar sobe. Ukoliko korisnik želi vidjeti pojedinosti sobe ili pojedinosti studenta, to može učiniti pritiskom na odgovarajuće tipke pomoću kojih se otvaraju novi prozori koji sadrže tekstualna polja pomoću kojih su opisani odabrani objekti.

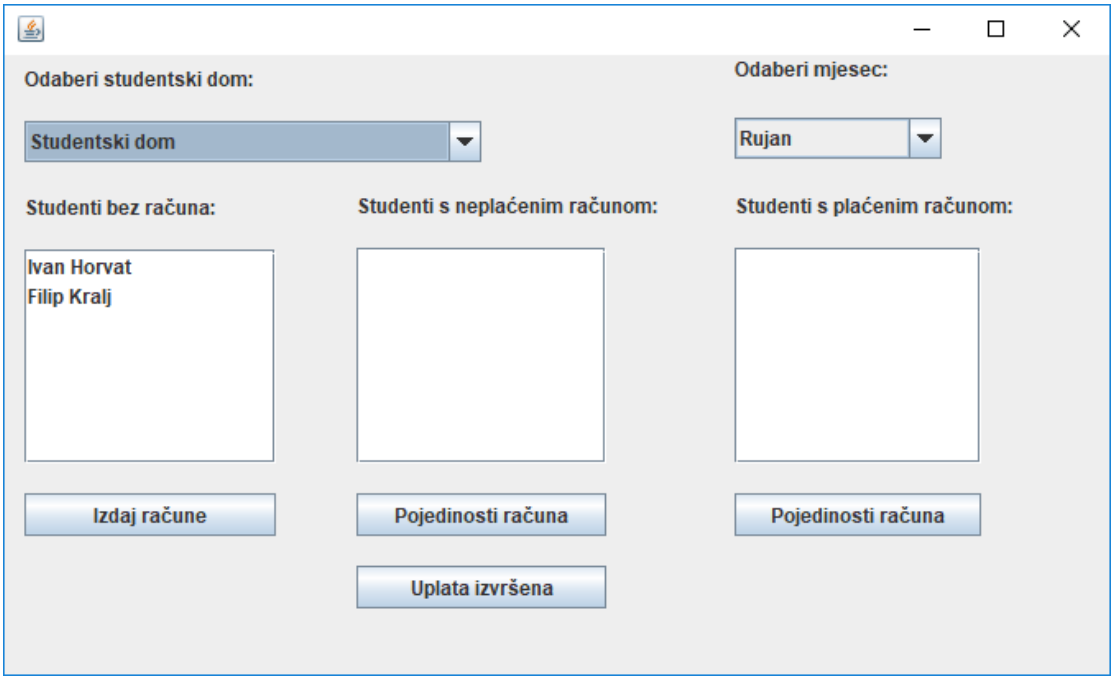


Sl. 4.10. *Dodavanje novog studenta(lijevo) ili promjena postojećeg (desno)*

Ukoliko je potrebno dodati novog studenata potrebno je pritisnuti tipku *Novi* na prozoru prikazanom na slici 4.9. Ako se u sobi nalaze dva kreveta i ako u sobi već postoje dva studenta, kreiranje novog će se odbiti te će korisnik o tome biti obaviješten. Pritiskom na tipku *Novi* otvara se novi *JFrame* (Sl. 4.10. lijevo) s jednom tipkom i sedam tekstualnih polja. Kako bi se kreirano novi potrebno je popuniti tekstualna polja sa željenim podacima te pritisnuti tipku *Dodaj novog studenta*. Provjera ispravnosti unesenih podataka jednaka je kao i kod unosa paviljona. Ukoliko se neispravno unesu ime ili prezime studenta, program „baca“ *StucException* iznimku te se prekida dodavanje novog, a kursor za pisanje se postavlja na tekstualno polje koje je neispravno popunjeno. Ako je cijeli proces kontrole pošao, poziva se obrada koja sprema studenta unutar baze podataka. Postupak je identičan i kod promjene postojećeg studenta (Sl. 4.10. desno) samo što obrada prepoznaje da odabrani student već postoji unutar baze podataka i izvrši se samo promjena njegovih svojstva.

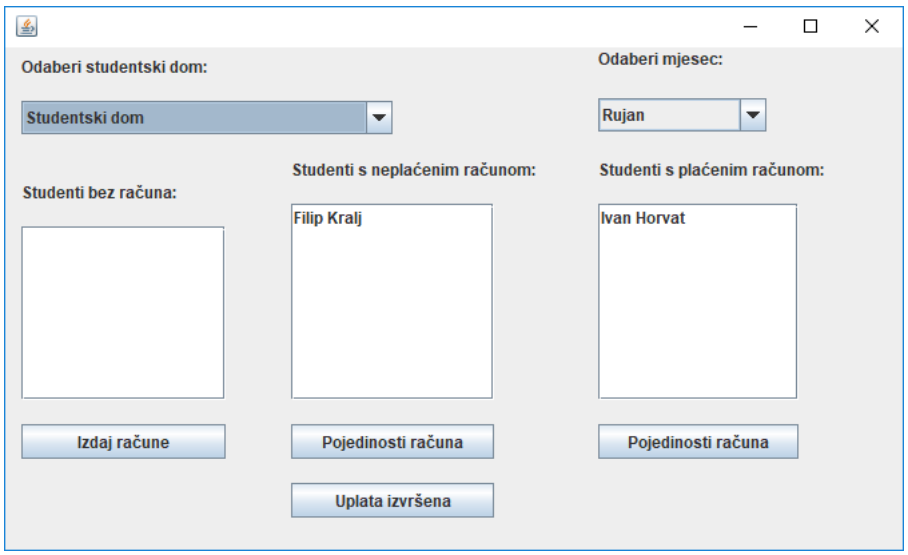
4.6. Prozor za upravljanje računima

Prozoru za upravljanje računima pristupa se kroz glavni izbornik. Klikom na *JMenuItem* pod nazivom *Računi* otvara se prozor kao na slici 4.11. Radi lakše demonstracije rada unesen je još jedan student (Filip Kralj) u novu sobu (broj sobe = 100).



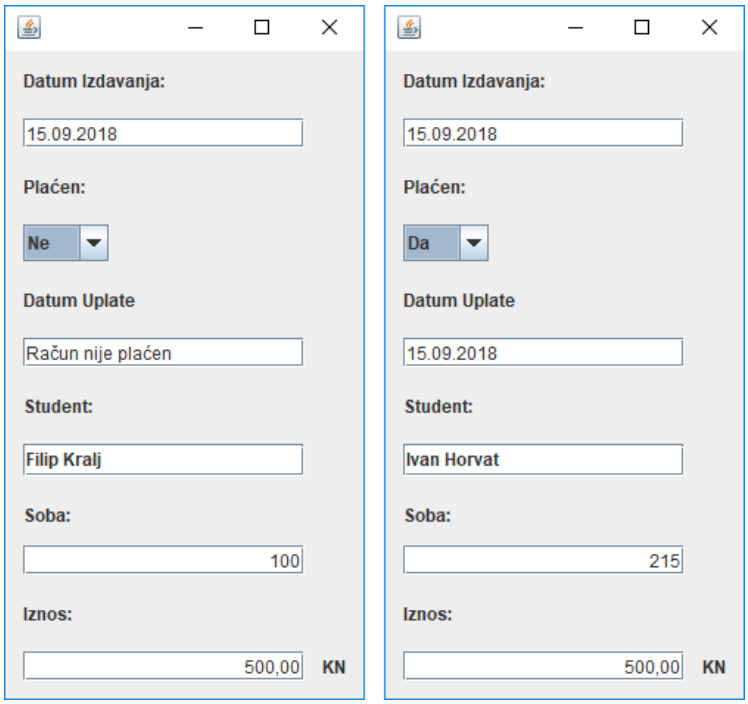
Sl. 4.11. *Prozor za izdavanje i evidenciju o plaćenosti računa*

JFrame sadrži tri liste i dva *Combo Box-a*. Pri samom pokretanju prozora, lijevi *Combo Box* puni se sa svim studentskim domovima, te po završetku punjenja, odabire se prvi. Paralelno s tim, u desnom *Combo Box-u* sadržani su svi mjeseci u godini, te se po završetku punjenja *Combo Box-a* studentskim domovima odabire se trenutni mjesec. Odabirom mjeseca kreće punjenje *JLista*. Unutar prva *JList-e* pojavljuj se svi oni studenti koji za odabrani mjesec nemaju izdan račun. U drugoj *JList-i* pojavljuju se svi studenti koji imaju izdan račun, ali on nije plaćen. Unutar treće *JList-e* pojavljuju se studenti koji su platili izdani račun. Pritiskom na tipku *Izdaj račune* provjerava se je li lista studenata prazna. Ukoliko lista nije prazna, pomoću *for* petlje prolazi se kroz svaki element liste studenata bez računa te se za svakoga kreira novi račun s trenutnim datumom izdavanja. Time se studenti prebacuju na srednju *JList-u*. Ukoliko je neki od studenata izvršio uplatu, potrebno je odabrati tog studenta i pritisnuti tipku *Uplata izvršena*. Time se student prebacuje na desnu *JList-u*.



Sl. 4.12. *Promjena na listama prozora za vođenje evidencije o računima nakon izdavanja i uplate računa*

Slika 4.12. prikazuje studenta kojem je izdan račun, ali koji nije izvršio uplatu te studenta koji je izvršio uplatu računa.

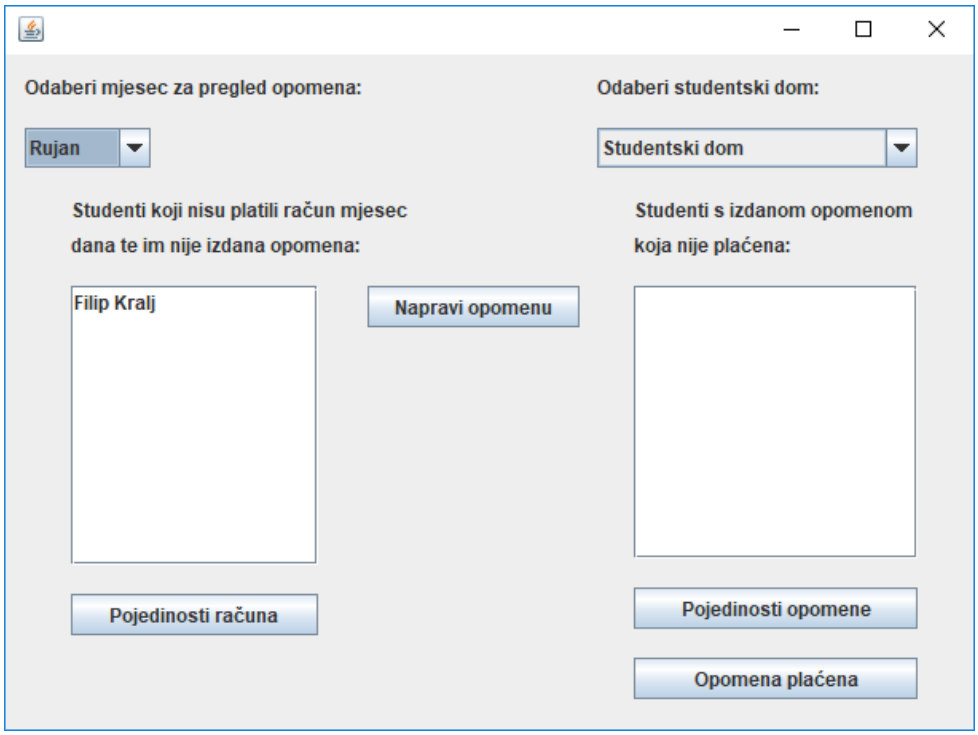


Slika 4.13. *Prozori za prikaz pojedinosti izdanog i plaćenog računa*

Ukoliko korisnik želi vidjeti pojedinosti svakog od računa, to može napraviti pritiskom na tipku *Pojedinosti računa*. Na slici 4.13. prikazani su prozori pojedinosti izdanog i ne plaćenog računa (lijevo) te pojedinosti izdanog i plaćenog računa (desno). Neplaćeni račun nema datum uplate dok uplaćeni ima. Iz pojedinosti se također može vidjeti datum izdavanja računa, iznos koji je potrebno uplatiti te soba i ime studenta za kojeg je izdan račun.

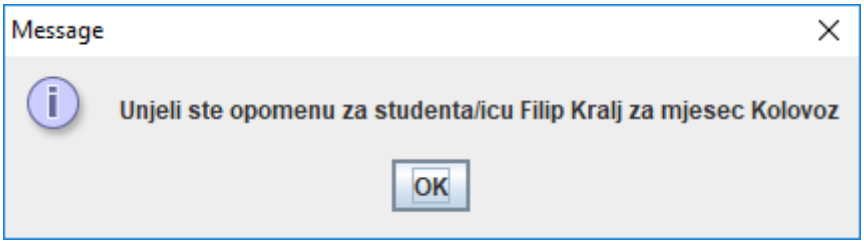
4.7. Prozor za upravljanje opomenama

Kao i prozoru za upravljanje računima, tako i prozoru za upravljanje opomenama pristupa se iz glavnog izbornika. Kako bi bila moguća demonstracija rada s opomena unutar baze podataka direktno je unesen novi račun za studenta Filipa Kralja te je datum izdavanja računa postavljen na 20.kolovoza 2018. godine.



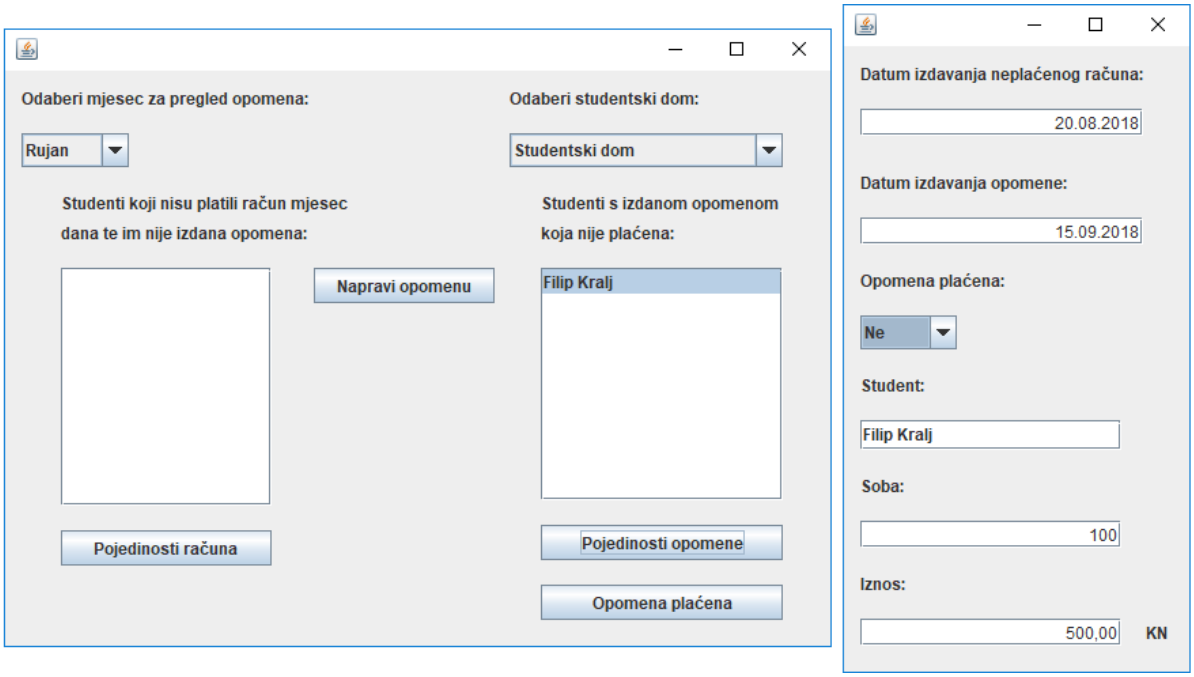
Sl. 4.14. *Prozor za upravljanje opomenama za neplaćene račune*

Prozor za upravljanje opomenama (Sl. 4.14.) sastoji se od dvije *JList-e* i dva *JCombo Box-a*. Pri pokretanju prozora, desni *Combo Box* puni se sa svim studentskim domovima iz baze podataka. Završetkom punjenja automatski se odabire prvi element *Combo Boxa*. Unutar lijevog *Combo Box-a* definirani su svi mjeseci u godini dana te se automatski odabire trenutni mjesec. Postavljanjem *Combo Box-a* na trenutni mjesec, unutar lijeve *JList-e* pojavljuju se svi studenti koji imaju izdan račun u prethodnom mjesecu, te svi studenti koji još nisu podmirili račun. To znači da je potrebno izdavanje opomene studentu u svrhu ubrzavanja plaćanja računa. Ukoliko se pritisne na tipku *Pojedinosti računa* uz prethodno odabranog studenta, dobiju se pojedinosti kao sa slike 4.13. (desno) gdje je vrijednost datuma izdavanja jednaka 20. kolovoza 2018. godine.



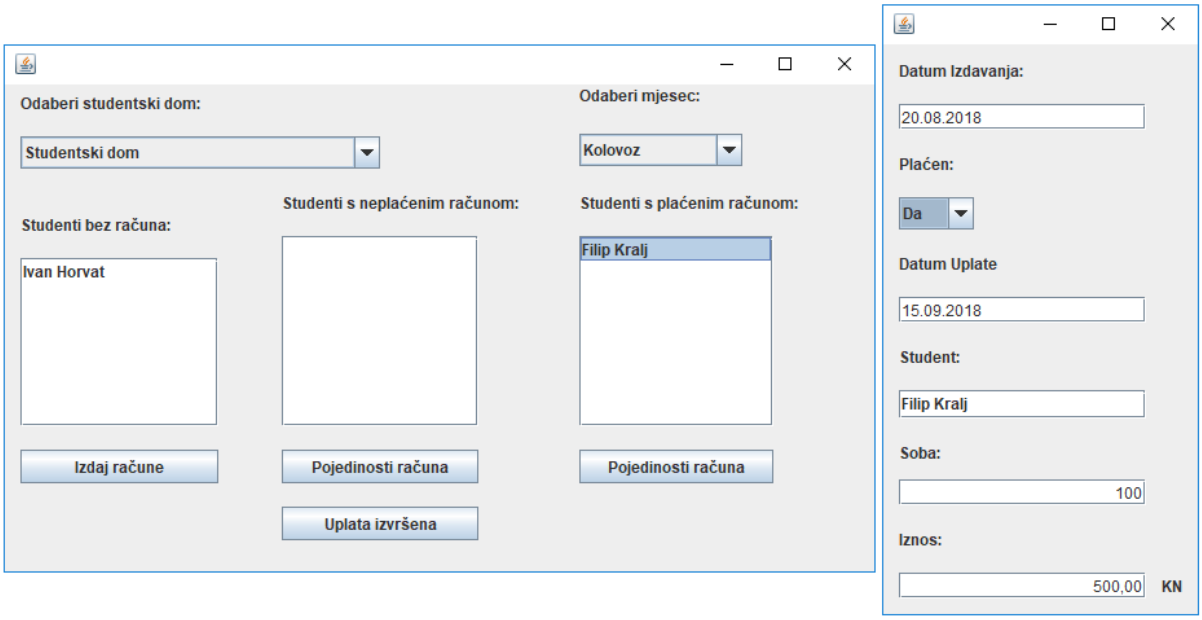
Sl. 4.15. *Obavijest o izdanoj opomeni za određenog studenta zbog računa iz prethodno mjeseca*

Kreiranje opomene vrši se pritiskom na tipku *Napravi opomenu* uz prethodni odabir studenta s lijeve *JList-e*. Time se pokreće provjera ima li student već izdanu opomenu. Ukoliko nema računa, pomoću *obradeRacuna* dohvaća se račun za koji je potrebno kreirati opomenu jer se opomena veže za točno taj račun. Postavlja se datum kreiranja opomene na trenutni te svojstvo plaćenosti opomene na *false*. Po završetku, pomoću *JOptionPane-a* se obavještava korisnik da je kreiranje opomene završeno uspješno (Sl. 4.15.).



Sl. 4.16. *Promjena na listama prozora za upravljanje opomenama te prozor s pojedinostima izdane opomene*

Nakon završetka metode koja kreira opomenu, student se prebacuje na desnu *JList-u* na kojoj se nalaze studenti koji imaju izdanu opomenu, ali račun svejedno nije podmiren. To se može vidjeti sa slike 4.16. Pritiskom na tipku *Pojedinosti opomene* moguće je vidjeti datum izdavanja računa za kojeg je kreirana opomena. Isto tako, vidljiv je i datum izdavanja opomene, ime studenta za kojeg je kreirana opomena te soba u kojoj je smješten. Ukoliko je student podmirio račun nakon izdane opomene, potrebno je na desnoj *JList-i* odabrati studenta, te pritisnuti tipku *Opomena plaćena*.



Sl. 4.17. *Prikaz lista prozora za upravljanje računima nakon uplate računa iz kolovoza za koji je bila izdana opomena*

Time se pokreće metoda koja sprema studenta u novu instancu te tu instancu, zajedno s datumom početka i kraja trenutnog mjeseca, prosljeđuje metodi koja vraća opomenu za račun koji je uplaćen. Tada se postavlja datum uplate opomene, a ujedno i računa, na trenutni datum. Time student nestaje s *JList-a* unutar prozora za upravljanje opomenama te se pojavljuje u desnoj *JList-i* u prozoru za upravljanje računima (Sl. 4.17.).

4.8. Lambda izrazi

Java kao objektno-orijentirani programski jezik predstavlja imperativni stil programiranja što se suprotstavlja funkcionalnom stilu programiranja. Kao takav, prije samog pokretanja bilo kakvog programa potrebno je definiranje klasa (eng. *class*) s njenim odgovarajućim atributima i metodama kako bi bilo moguće instanciranje objekta definirane klase te korištenje njegovih metoda. Ukoliko je neka od metoda deklarirana kao statička (eng. *static*), moguće je njeno pozivanje bez prethodnog instanciranja objekta, ali prethodno je potrebno kreiranje klase koja je opisnik instanciranog objekta kao i deklariranje statičke metode. Na primjer, kako bi se izvršila akcija pritiska „*Swing JButton"* tipke potrebna je implementacija i instanciranje klase koja „osluškuje" pojavu događaja, a koja sadrži samo jednu metodu koja se naziva „*actionPerformed()*". Postoji veliki broj programskih jezika koji su zasnovani na funkcionalnom programiranju gdje nije potrebno definiranje funkcija u klasama. Kreiranje objekata s poljima koja se mogu modificirati (mijenjati) kodom metoda može biti podložno pogreškama te se otežava njegovo debagiranje (eng. *debuging*). Funkcije kao takve ne zavise od eksternih sadržaja već preuzimaju vrijednosti kroz proslijeđene argumente, na njih primjenjuju logiku te vraćaju rezultat. Ona može preuzeti jednu vrijednost i generirati drugu, a da pri tome ne mijenja ulaz. Zahvaljujući tome, bez obzira koliko puta se dogodi njeno izvršavanje s istim ulaznim parametrima, neće se dogoditi njihova promjena. U funkcionalnim jezicima *(JavaScript, Ruby, Scala, Haskell)* funkcije se tretiraju kao glavni elementi te je s njima moguće izvršiti:

- definiranje funkcije unutar druge funkcije,
- prosljeđivanje funkcije kao argument drugoj funkciji,
- dodjeljivanje funkcije varijabli,
- te vraćanje funkcije iz druge funkcije.

Java 8 uvodi Lambda izraze koje su zapravo anonimne funkcije. One kao takve nemaju naziv i zbog toga se mogu dodjeli varijabli. Njihovim uvođenje omogućeno je programiranje u kombiniranom stilu. Jedan od primjera usporedbe navedena dva stila programiranja je korištenje metoda *equals()* suprotno korištenju metode *contains()* pri traženju točno određenog elementa tipa *String* unutar kolekcije *„ArrayList"*.

```
public class ImperativniVSFunkcionalni {
    public static void main(String[] args) {
        List<String> lista = new ArrayList<>();
        lista.add("Jedan");
        lista.add("Dva");
        lista.add("Tri");
        //imperativni stil
        boolean postoji = false;
        for (String element : lista) {
            if (element.equals("Dva")) {
                postoji = true;
                System.out.println("Imperativni stil. Postoji: " + postoji);
                break;
            }
        }
        //funkcijonalni stil
        System.out.println("Funkcionalni stil. Postoji: " +
      lista.contains("Dva"));
    }
}
Izlaz:
Imperativni stil. Postoji: true
Funkcionalni stil. Postoji: true
```

Primjer koda 4.6. *Izvođenje iste provjere na imperativni i funkcionalni način*

U primjeru koda 4.6. je vidljivo da ukoliko se imperijskim stilom želi doći do traženog elementa prije svega, potrebno je definirati varijablu *boolean* tipa koja služi kao indikator pronalaska. Zatim, potrebno je uz pomoć *for* petlje odabrati svaki element liste te pomoću metode *equals()* provjeriti je li trenutni element traženi element. Ukoliko je, indikator mijenja vrijednost na *false*, ispisuje se tekst i pomoću *break* naredbe izlazi se iz petlje. Time je stvoren uvid kako imperijski stil programiranja upravlja načinom izvršavanja koda. S druge strane, funkcionalni stil je kratak, koncizan i lako ga je razumjeti. Uklonjeno je kreiranje indikatora i stvaranje petlja. *Contains()* metoda ugrađena je unutar *„ArrayList“* kolekcije što omogućava njeno korištenje. Međutim, korištenjem lambda izraza moguće je definiranje funkcije po želji bez pravljenja klase i definiranja metoda u njima.

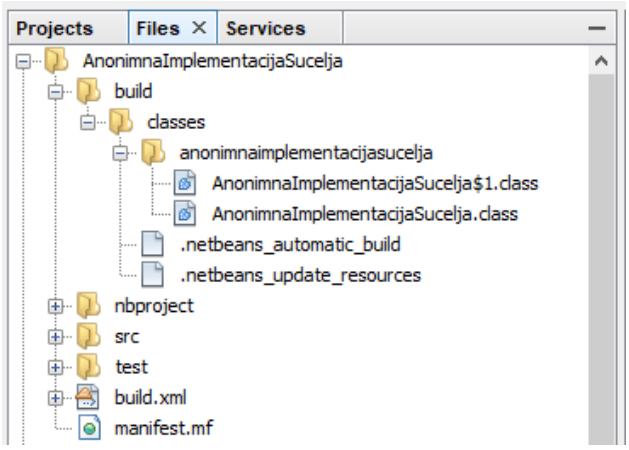
4.8.1. Anonimna implementacija sučelja (eng. *interface*) kroz lambda izraz

Kao što je već navedeno, lambda izrazi su anonimne funkcije koje se mogu dodjeljivati izravno varijablama, moguće ih je proslijediti kao argument ili vratiti iz metode. Time zamjenjuju praksu kreiranja anonimnih unutarnjih klasa koje služe kao omotač metoda.

```
public class AnonimnaImplementacijaSucelja {
    public static void main(String[] args) {
        Comparator<String> comparator = new Comparator<String>() {
            @Override
            public int compare(String o1, String o2) {
                return o1.compareTo(o2);
            }
        };
        int compareVrijednost = comparator.compare("Dobar", "Dan");
        System.out.println(compareVrijednost);
    }} //Izlaz: 14
```

Primjer koda 4.7. *Anonimna implementacija sučelja*

U primjeru koda 4.7. je vidljiva implementacija sučelja *Comparator* koje sadrži apstraktnu metodu *compare()*. Kako bi se izbjeglo kreiranje klase koja implementira to sučelje te kreiranje objekta iste klase radi omogućavanja korištenja te metode, implementacija se može izvršiti pomoću anonimne unutarnje klase. Nakon pozivanja konstruktora *Comparator* sučelja otvaraju se i zatvaraju vitičaste zagrade, kreira se metoda *compare()* koja prima dva *String* argumenta te uspoređuje njihove vrijednosti i kao rezultat vraća *int* vrijednost njihove razlike.



Sl. 4.18. *Pojava anonimne klase unutar klase anonimnaimplementacijasucelja*

Nakon pokretanja programa u konzoli se ispiše rezultat „14“ kao rezultat *compareTo()* metode koja prima *String-ove Dobar* i *Dan*, dok unutar projekta dolazi do promjene. Nakon pokretana *java compiler-a,* unutar direktorija *build* vidljive su dvije *.class* datoteke (Sl. 4.18.). To su klase *AnonimanaImplementacijaSucelja.class* i *AnonimnaImpelentacijaSucelja$1.class.* Prva predstavlja klasu u kojoj je sadržana *main* metoda iz koje se pokreće program, a druga predstavlja unutarnju anonimnu klasu sadržanu unutar *AnonimanaImplementacijaSucelja.class*. Dio imena prije *$1* ukazuje na klasu u kojoj je kreirana. Uporabom lambda izraza eliminira se potreba za kreiranjem unutarnjih anonimnih klasa.

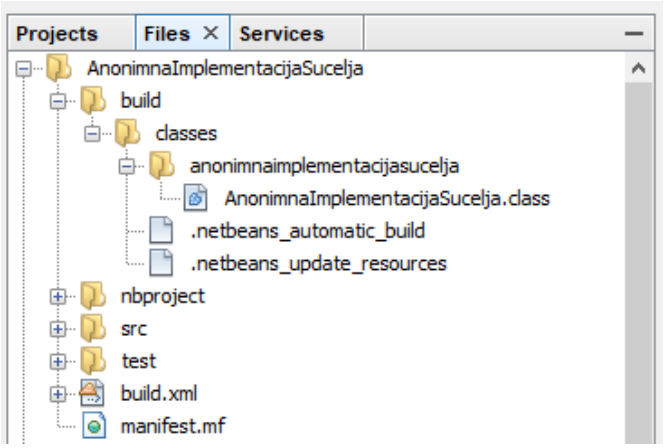
```
public class AnnonimnaMetoda {
    public static void main(String[] args) {

        Comparator<String> comparatorLambda =
                (String o1, String o2) -> {return o1.compareTo(o2);};

        int compareVrijednost = comparator.compare("Dobar", "Dan");
        System.out.println(compareVrijednost);

    }
}
Izlaz:
14
```

Primjer koda 4.8. *Implementacija lambda izraza*

Iz primjera koda 4.8. je vidljivo kako je cijela unutarnja klasa zamijenjena lambda izrazom. Lambda izraz sastoji se od argumenata koji su obavijeni običnim zagradama, *array token*-a te tijela koje je obavijeno vitičastim zagradama. Specificiranje tipova argumenata je opcionalno jer Java *compile-a* (eng. *compiler*) ispravno određuje tipove argumenata iz samog koga. Ukoliko tijelo lambda izraza ima samo jednu liniju koda, moguće je i izostaviti *return* naredbu te nakon tih promjena lambda izraz izgleda kao u primjeru 4.9. :

```
public class AnnonimnaMetoda {
    public static void main(String[] args) {

        Comparator<String> comparatorLambda =
                (o1, o2) -> o1.compareTo(o2);

        int compareVrijednost = comparator.compare("Dobar", "Dan");
        System.out.println(compareVrijednost);

    }
}
Izlaz:
14
```

Primjer koda 4.9. *Skraćeni zapis lambda izraza*



Sl. 4.19. *Nestanak anonimne klase unutar klase AnonimnaImplementacijasucelja*

Nakon pokretanja programa, rezultat je u sva tri primjera jednak, ali nakon korištenja lambde dogodila se promjena u direktoriju projekta. Naime, nakon *compail-a* vidljiva je samo jedna *AnonimnaImplementacijaSucelja.class* što ukazuje na to da je izbjegnuto kreiranje unutarnje anonimne klase (Sl. 4.19.). Time je prikazana implementacija funkcionalnog stila programiranja u objektno-orijentiranom programskom jeziku te je dobivena veća preglednost i urednost koda.

4.8.2. Funkcionalno sučelje (eng. *Functional interface*)

Kako bi metode definirane u sučelju mogle biti implementirane kao lambda izrazi, sučelje mora definirati jednu ne implementiranu metodu. Takva sučelja nazivaju se funkcionalna sučelja. S tehničke strane, funkcionalna sučelja mogu sadržavati više metoda, ali značajan dio istoga je postojanje samo jedne koja je apstraktna te se ona mora implementirati. U aplikaciji je korišteno sučelje *ActionListener*, a standardni način za implementaciju tog sučelja je pomoću anonimne unutarnje klase.

```
cmbStudentskiDom.addActionListener(new ActionListener(){
      public void actionPerformed(ActionEvent actionEvent){
            //Kod
      }
};
```

Primjer koda 4.10. *Implementacija sučelja pomoću anonimne unutarnje klase*

Pošto je implementacija izvršena u konstruktoru *JFrame-a,* uz dodatna dva *ComboBoxModel*-a metodu stvara nepreglednom. Pošto *ActionListener* sučelje definira samo jednu metodu koja se naziva *actionPreformed(),* sučelje se može nazvati funkcionalnim i moguće ga je implementirati pomoću lambda izraza:

```
cmbStudentskiDom.addActionListener((ActionEvent e) -> {
      //Kod
});
```

Primjer koda 4.11. *Implementacija sučelja pomoću lambda izraza*

Ovakav način implementacije mnogo je čitljiviji i jednostavniji za pisanje. Na ovaj način izbjegnuto je deklariranje i instanciranje anonimnih klasa. Iz koda je vidljivo da lambda izraz prima jedan argument koji je tipa *ActionEvent* u koji će biti vraćen rezultat koda koji se odradi unutar vitičastih zagrada. Metoda *actionPerformed()* „osluškuje“ događaj kojem će biti proslijeđena instanca „e“. Pošto lambda izraz prima samo jedan argument, *compiler* sam otkriva kojeg je tipa.

4.8.3. forEach() metoda

Kako bi se omogućilo ponavljanje koda ili pristupanje svakom elementu polja ili kolekcije, jedan od standardnih načina je uporabom *for* petlje. for petlja se koristi ukoliko postoji definirano i napunjeno polje kojemu je poznat broj elemenata.

```
int[] array = new int[10];
...
for (int i = 0; i < 10; i++) {
    System.out.println(array[i]);
}
```

Primjer koda 4.12. *Iteracija kroz polje pomoću for petlje*

Također, ukoliko postoji kolekcija određenog tipa (npr. *String*), ali joj se ne zna točan broj elemenata, svakoj od njih pristupa se naprednom *for* petljom.

```
List<String> list = new ArrayList<>();
list.add(„prviElement");
...
for (String element : list)
    System.out.println(element);
}
```

Primjer koda 4.13 .*Iteracija kroz listu naprednom for petljom*

U oba slučaja prikazan je imperativni pristup te „diktira" kodom na koji način ga treba izvršiti te eksternu iteraciju kroz polje. U prvom primjeru prije svega se definira *int* varijabla koja služi za simulaciju indeksa svakog od elemenata polja. Vezano uz to, postavlja se uvjet *for* petlje na broj elemenata minus jedan (u Java programskom jeziku indeksi polja ili kolekcija kreću od 0) te se postavlja inkrement petlje. Drugi primjer prikazuje napredniju *for* petlju (*Enhanced For loop*) gdje se za svaki element liste *list* instancira objekt tipa String koji simulira element iz *list* kolekcije, te se u tijelu petlje izvršava određeni kod (u ovom slučaju se „element" samo ispisuje u konzolu). Java 8 nudi funkcionalan pristup za iteraciju kroz kolekcije. Kako bi to bilo moguće, kolekciji se prosljeđuje funkcija koja će se primijeniti nad svakim elementom kolekcije. Na taj način kolekcija sama određuje na koji način će proslijeđenu funkciju izvršiti. Zbog toga kolekcija *ArrayList* sadrži metodu *forEach()* koja prima samo jedan argument tipa *Consumer* (funkcionalno sučelje) što znači da *forEach()* prima lambda izraze. U ovoj distribuiranoj aplikaciji za svaki rad nad elementima kolekcije korištena je metoda *forEach()* kojoj je bio proslijeđen lambda izraz.

```
List<Opomena> listaOpomena = HibernateUtil.getSession.createQuery(....
List<Student> listaStudenataNeplacenaOpomena = new ArrayList<>();
...
listaOpomena.forEach(opomena -> {
    listaStudenataNeplacenaOpomena.add(opomena.getRacun().getStudent());
});
```

Primjer koda 4.14. *Pomoću metode forEach() svakom elementu liste proslijeđen je lambda izraz*

Zbog daljnjeg rada sa studentima kojima je izdana opomena, ali nisu svejedno izvršili uplatu računa za koji je ona izdana, potrebno ih je prikupiti te spremiti u listu. Zbog toga je pomoću *HibernateUtil* klase pozvana metoda *getSession()* pomoću koje je iz baze obavljeno dohvaćanje svih opomena izdanih u trenutnom mjesecu. Zatim je kreirana nova *ArrayList* kolekcija koja je tipa *Student* kako bi se u nju moglo izvršiti spremanje studenata. Pri tome, korišten je funkcionalni pristup gdje je nad *listaOpomena* pozvana *forEach()* metoda. Kao argument metode proslijeđen je lambda izraz koji za svaki element liste preko *get* metoda dohvaća studenta koji je vezan, preko računa, svojom šifrom za tu opomenu. Time je izbjegnuta eksterna iteracija u kojoj je potrebno striktno navoditi „kako se kod treba ponašati“, a implementirana je interna iteracija kojoj kolekcija logiku obavlja u pozadini, te je tako kod postao čitljiviji i koncizniji.

5. ZAKLJUČAK

Ovaj diplomski rad prikazuje *Java* aplikaciju koja za praktičnu primjenu ima preuzimanje studenta iz *online* baze podataka te njihovo pohranjivanje u lokalnu bazu podataka. Primjer kreiranja studenta uz pomoć forme primjenjuje se u gotovo svim online prijavama te je proces njegovog kreiranja realiziran uz pomoć *REST API* servisa. *REST* servis kreiran je uz pomoć *PHP* programskog jezika, ali isto tako mogao je biti kreira uz pomoć *Java EE*, *Spring-a*, *Groovy*-a i sl. Isto tako, kreirani server mogao je biti preseljen na bilo koji drugi server koji prethodno podržava neku od baza podataka (*MySql*, *Oracle*, *DB2* i sl.). *REST* kao stil izvedbe servisa, u pozadini koristi *HTTP* protokol koji se pokazao kao idealno rješenje. Ista aplikacija mogla je biti izvedena pomoću *SOAP* protokola, ali zbog njegovih putanja koje se postavljaju putem *XML-a* znatno bi zakomplicirale realizaciju servisa. Pri kreiranju studenata korištena je *POST* metoda *HTTP-a* koja u kombinacijom s *create.php* datotekom servisa zapisuje studenta unutar baze podataka. Ukoliko bi bila dodana mogućnost postavljanja određenih dokumenata, moglo bi se realizirati zbrajanje bodova te kreiranje rang lista. Unutar klijentske aplikacije, kreirane uz pomoć *Java* programskog jezika, kreiran je *API*-servis koji se uz pomoć *GET* metode *HTTP* protokola spaja na *read.php* datoteku *REST* servisa koja vraća studente zapisane u *JSON* formatu. Klijentska aplikacija preuzima listu studenata, te ju formulira na način kako njoj odgovara. Isto tako, rezultat iz *REST* servisa mogao je biti vraćen u *XML* ili *PDF* formatu. Tada bi bilo potrebno promijeniti prihvat klijentske aplikacije na način da iz *XML* formata formira objekte studente koje je potrebno spremiti u bazu podataka. Nakon preuzimanja i spremanja studenata aplikacija nudi upisivanje istih unutar prethodno kreirane sobe te izdavanje računa za svakog studenta na mjesečnoj razini. To klijentskoj aplikaciji omogućava primjenu u bilo kojem studentskom domu. Pravilnom realizacijom *API-a* kreirana je funkcionalna distribuirana aplikacija s praktičnom primjenom.

LITERATURA

[1] Y. Fain, Java Programiranje, Kompjuter biblioteka, Beograd, 2015.

[2] (The only proper) PDO tutorial, https://phpdelusions.net/pdo, datum posjete: 05.09.2018.

[3] Forms, https://getbootstrap.com/docs/4.1/components/forms, datum posjete: 05.09.2018.

[4] W3Schools, https://www.w3schools.com/php/php_superglobals.asp, datum posjete: 07.09.2018.

[5] Hibernate ORM, http://hibernate.org/orm/, datum posjete: 09.09.2018.

SAŽETAK

Svrha rada bila je kreirati distribuiranu aplikaciju uz njenu praktičnu primjenu. Prikazana je i objašnjena izrada *REST API* servisa pomoću kojeg su se dohvaćali i spremali podaci te njegovo naseljavanja na server. Prikazana je izrada forme pomoću koje se izvršavalo spremanje studenata unutar baze podataka pozivanjem *REST* servisa. Dokazano je da je *HTTP* protokol kao podloga *REST* servisa idealan za ostvarenje komunikacije između dva sustava. Objašnjen je način kreiranja *POST* metode kroz formu kreiranu pomoću *HTML-a*, te dohvaćanje podataka s *REST* servisa pomoću *GET* metode unutar *API-a*. Prikazana je i objašnjena metoda koja koristi *API* unutar klijentske aplikacije te način na koji su dohvaćeni podaci pretvoreni iz J*SON* formata zapisa u objekte korisne klijentskoj Java aplikaciji. Objašnjena je primjena *Hibernate ORM-a* koji je uvelike olakšao kreiranje baze te njeno povezivanje s klijentskom aplikacijom. Demonstriran je rad klijentske java aplikacije, te je dokazano kako se može primijeniti za bilo koji studentski dom.

Ključne riječi: *API*, distribuirana, *GET*, *HTTP*, *HTML*, *Hibernate, ORM, POST, REST*

ABSTRACT

COMMUNICATION OF CLIENTS IN DISTRIBUTED JAVA APPLICATION

The purpose of the paper was to create a distributed application with its practical use. It was displayed and explained making of The REST API service by which the data was taken and saved and its population on the server. It is shown making of form which is used to save students inside the database by calling the REST service. It has been proven that the HTTP protocol as a base of REST service is ideal for communication between two systems. It was explained how to create POST methods through a form created using HTML, and fetching data from REST service using the GET method within the API. It was showed and explained the method that uses the API within the client application and the way in which the retrieved data is converted from JSON format records to objects useful for client Java application. It was explained appliance of Hibernate ORM which greatly facilitated the creation of a database and its connection with client applications. It was demonstrated the work of client java applications, and has been proven to be applicable to any student dorm.

Keywords: API, distributed, GET, HTTP, HTML, Hibernate, ORM, POST, REST

ŽIVOTOPIS

Ivan Čizmar je rođen 04.12.1994. godine u gradu Spittal an der Drau u Austriji. Dolazi iz mjesta Popovac gdje je 2001. godine upisao osnovnu školu. 2009. godine upisuje srednju školu u Belom Manastiru, smjera elektrotehničar te ju završava 2013. godine. Po završetku srednje škole upisuje preddiplomski studij elektrotehnike na fakultetu Elektrotehnike, računarstva i informacijskih tehnologija u Osijeku kojeg završava 2016. godine kada upisuje diplomski studij na istom fakultetu, smjer komunikacijske tehnologije. 2017. godine upisuje tečaj Java programer u školi Edunova te ga završava 2018. godine. U slobodno vrijeme trenira nogomet, te je kroz period studiranja radio više studentskih poslova. Trenutno je zaposlen kao Junior Java *developer* u Zagrebačkoj banci u Zagrebu.

Potpis_____

PRILOZI

Na optičkom disku u prilogu nalazi se ovaj rad u .docx i .pdf formatu kao i cjelokupni kod aplikacije opisane u radu.