

# Java završni

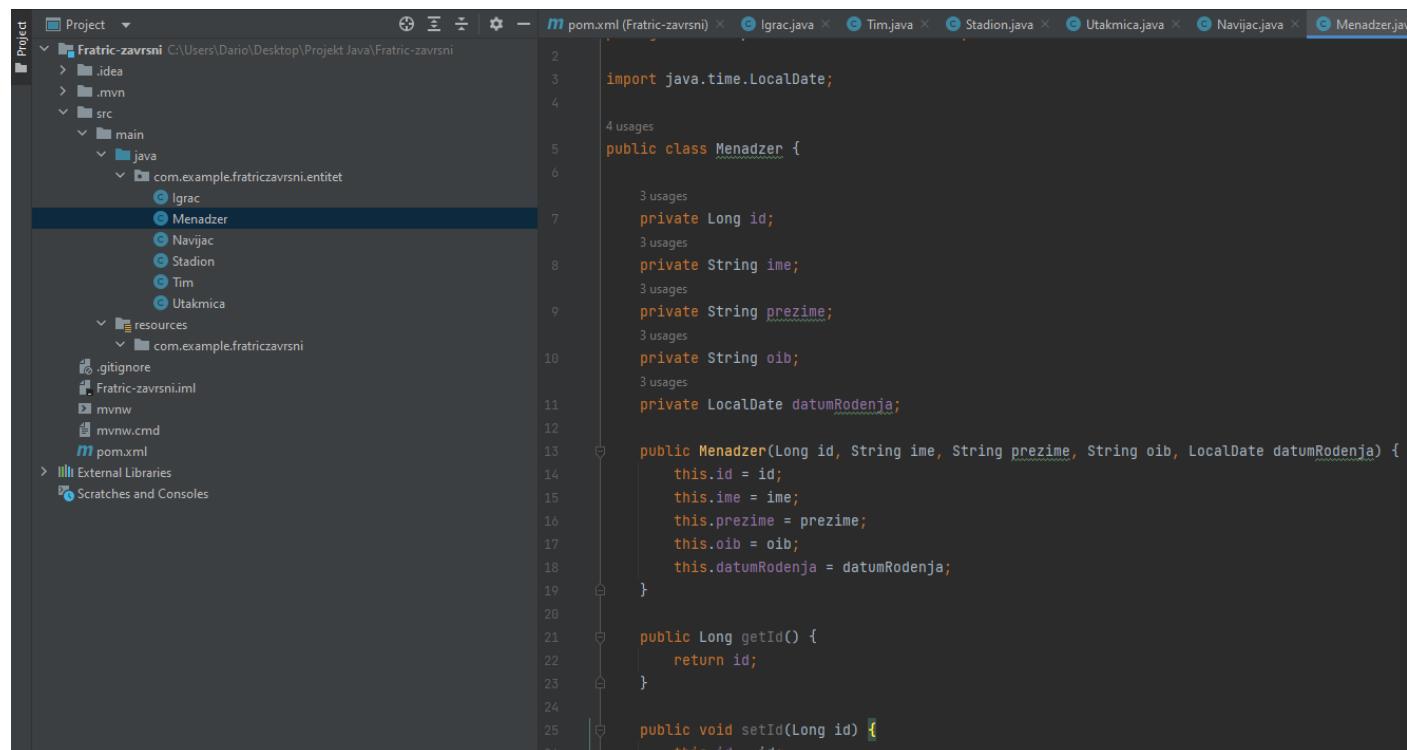
Za početak nam trebaju svi glavni entiteti koji će opisivati naš program.

Odabirem sljedeće entitete: Menadzer, Igrac, Tim, Stadion, Utakmica i Navijac.

Nakon što osposobim sve najvažnije entitete i povežem ih u programu, dodat ću i „Karte“ gdje će navijač moći kupiti karte za utakmice i slično.

Navijac je osoba koja koristi naš program i kupuje karte za utakmice. Ostale komponente će biti hardkodirane (odnosno čitat će se iz datoteke i neće se moći izmjenjivati, samo će se moći nadodavati novi Menadzeri / Igraci / Timovi / Stadioni / Utakmice).

Menadzer ima sljedeće podatke:



The screenshot shows the IntelliJ IDEA interface with the project navigation bar on the left and the code editor on the right. The project structure is visible, showing a package named 'com.example.fratriczavrsni.entitet' containing classes 'Igrac', 'Menadzer', 'Navijac', 'Stadion', 'Tim', and 'Utakmica'. The 'Menadzer.java' file is open in the code editor, displaying the following Java code:

```
import java.time.LocalDate;

public class Menadzer {

    private Long id;
    private String ime;
    private String prezime;
    private String oib;
    private LocalDate datumRodjenja;

    public Menadzer(Long id, String ime, String prezime, String oib, LocalDate datumRodjenja) {
        this.id = id;
        this.ime = ime;
        this.prezime = prezime;
        this.oib = oib;
        this.datumRodjenja = datumRodjenja;
    }

    public Long getId() {
        return id;
    }

    public void setId(Long id) {
        this.id = id;
    }
}
```

## Igrac:

The screenshot shows the IntelliJ IDEA interface with the project structure on the left and the code editor on the right. The code editor displays the `Igrac.java` file from the `com.example.fratriczavrsni.entitet` package. The code defines the `Igrac` class with fields for id, ime, prezime, oib, datumRodjenja, and drzava, along with a constructor and an equals method.

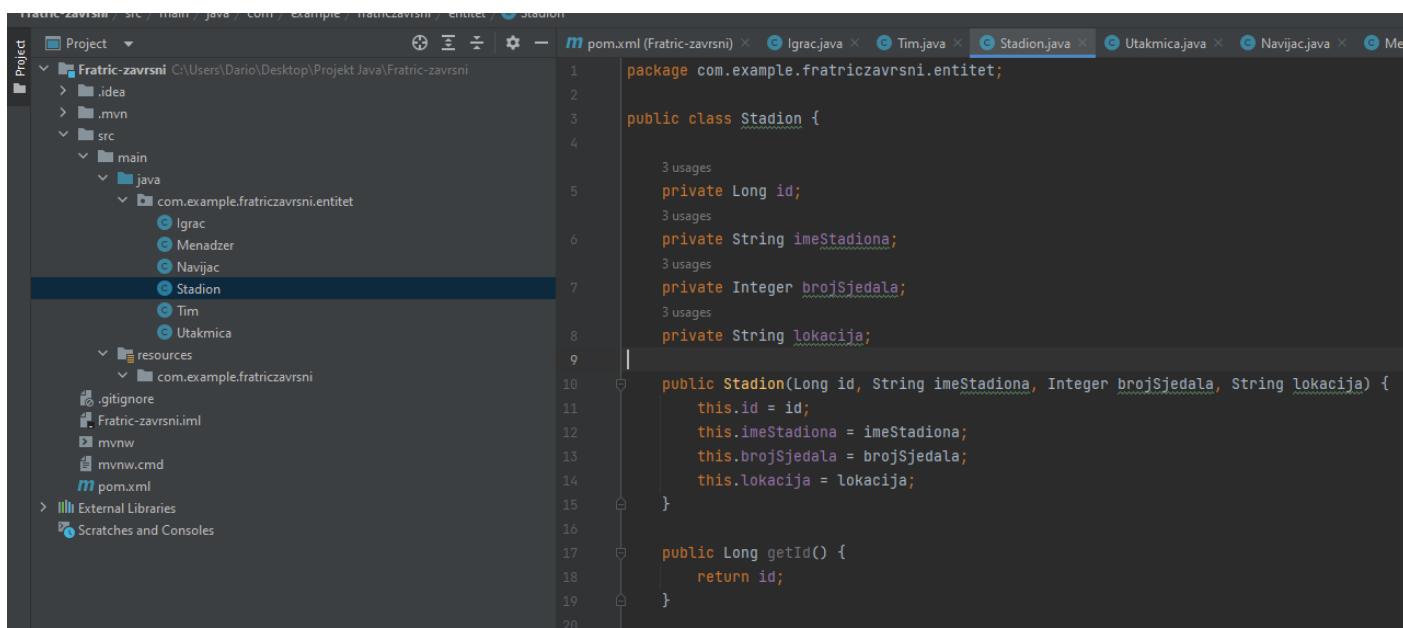
```
1 package com.example.fratriczavrsni.entitet;
2
3 import java.time.LocalDate;
4
5 public class Igrac {
6
7     private Long id;
8
9     private String ime;
10
11    private String prezime;
12
13    private String oib;
14
15    private LocalDate datumRodjenja;
16
17    private String drzava;
18
19
20    public Igrac(Long id, String ime, String prezime, String oib, LocalDate datumRodjenja, String drzava) {
21        this.id = id;
22        this.ime = ime;
23        this.prezime = prezime;
24        this.oib = oib;
25        this.datumRodjenja = datumRodjenja;
26        this.drzava = drzava;
27    }
28
29
30    @Override
31    public boolean equals(Object o) {
32        if (this == o) return true;
33        if (o instanceof Igrac) {
34            Igrac igrač = (Igrac) o;
35            return id.equals(igrač.id) && ime.equals(igrač.ime) && prezime.equals(igrač.prezime) && oib.equals(igrač.oib) && datumRodjenja.equals(igrač.datumRodjenja) && drzava.equals(igrač.drzava);
36        }
37        return false;
38    }
39
40    @Override
41    public String toString() {
42        return "Igrac{" +
43                "id=" + id +
44                ", ime='" + ime + '\'' +
45                ", prezime='" + prezime + '\'' +
46                ", oib='" + oib + '\'' +
47                ", datumRodjenja=" + datumRodjenja +
48                ", drzava='" + drzava + '\'';
49    }
50}
```

## Navijac:

The screenshot shows the IntelliJ IDEA interface with the project structure on the left and the code editor on the right. The code editor displays the `Navijac.java` file from the `com.example.fratriczavrsni.entitet` package. The code defines the `Navijac` class with fields for id, ime, prezime, oib, and datumRodjenja, along with a constructor and a getId method.

```
1 package com.example.fratriczavrsni.entitet;
2
3 import java.time.LocalDate;
4
5 public class Navijac {
6
7     private Long id;
8
9     private String ime;
10
11    private String prezime;
12
13    private String oib;
14
15    private LocalDate datumRodjenja;
16
17
18    public Navijac(Long id, String ime, String prezime, String oib, LocalDate datumRodjenja) {
19        this.id = id;
20        this.ime = ime;
21        this.prezime = prezime;
22        this.oib = oib;
23        this.datumRodjenja = datumRodjenja;
24    }
25
26
27    public Long getId() {
28        return id;
29    }
30}
```

## Stadion:



```
package com.example.fratriczavrsni.entitet;

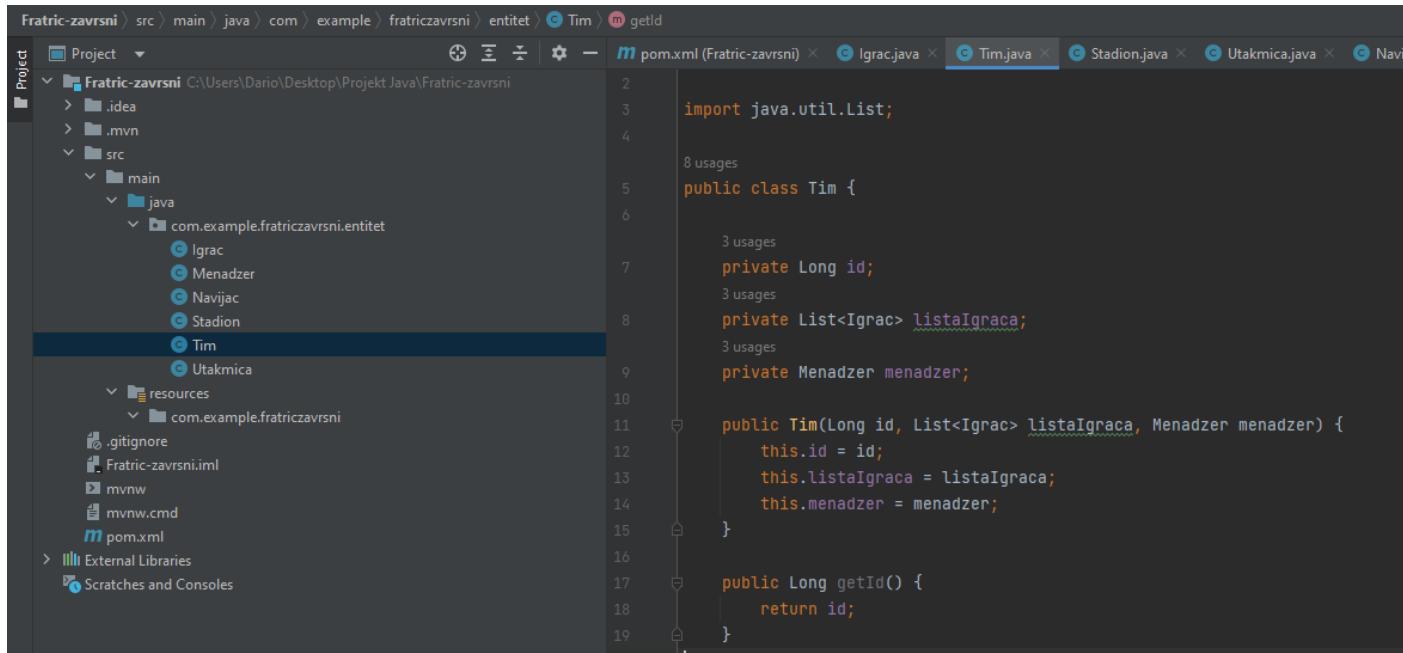
public class Stadion {

    private Long id;
    private String imeStadiona;
    private Integer brojSjedala;
    private String lokacija;

    public Stadion(Long id, String imeStadiona, Integer brojSjedala, String lokacija) {
        this.id = id;
        this.imeStadiona = imeStadiona;
        this.brojSjedala = brojSjedala;
        this.lokacija = lokacija;
    }

    public Long getId() {
        return id;
    }
}
```

## Tim:



```
import java.util.List;

public class Tim {

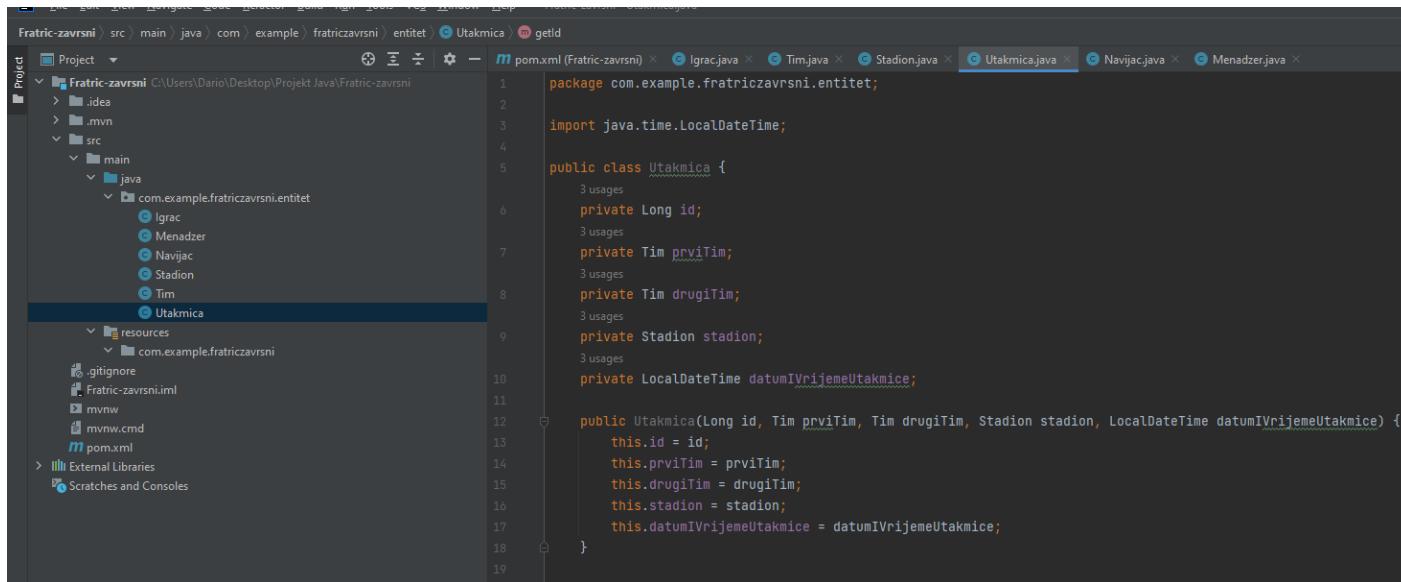
    private Long id;
    private List<Igrac> listaIgraca;
    private Menadzer menadzer;

    public Tim(Long id, List<Igrac> listaIgraca, Menadzer menadzer) {
        this.id = id;
        this.listaIgraca = listaIgraca;
        this.menadzer = menadzer;
    }

    public Long getId() {
        return id;
    }
}
```

Za sad ima menadzera i listu igraca.

## Utakmica:



The screenshot shows the IntelliJ IDEA interface with the project 'Fratric-zavrsni' open. The 'src' directory contains 'main' and 'java' packages. The 'java' package has several classes: Igrac, Menadzer, Navijac, Stadion, Tim, and Utakmica. The 'Utakmica' class is selected in the code editor. The code defines a class with attributes for id, two teams (prviTim and drugiTim), a stadium (stadion), and a date (datumIVrijemeUtakmice). It also includes a constructor and a copy constructor.

```
package com.example.fratriczavrsni.entitet;

import java.time.LocalDateTime;

public class Utakmica {
    private Long id;
    private Tim prviTim;
    private Tim drugiTim;
    private Stadion stadion;
    private LocalDateTime datumIVrijemeUtakmice;

    public Utakmica(Long id, Tim prviTim, Tim drugiTim, Stadion stadion, LocalDateTime datumIVrijemeUtakmice) {
        this.id = id;
        this.prviTim = prviTim;
        this.dugiTim = dugiTim;
        this.stadion = stadion;
        this.datumIVrijemeUtakmice = datumIVrijemeUtakmice;
    }
}
```

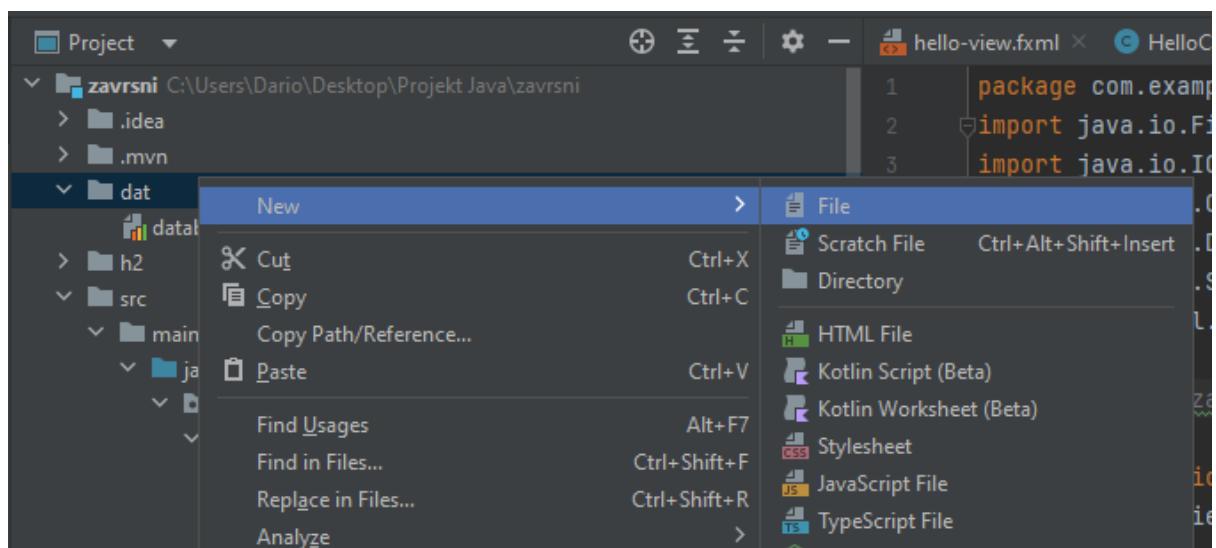
Utakmica ima 2 tima (koja će igrati), Stadion na kojem će se igrati i datum/vrijeme utakmice.

Sa ovim entitetima smo iskoristili skoro sve što smo radili po pitanju varijabli (još nam fale Mape, Setovi i nizovi). Kasnije ću smisliti gdje ću ih iskoristiti.

Napravit ću sve datoteke iz kojih će se čitati većina podataka (nema smisla za unošenjem podataka).

Napravit ćemo novi package „util“ i u njemu ćemo napraviti klasu Baza gdje će nam biti sve metode koje citaju iz baze.

Da bi pročitali sve iz baze moramo se prvo spojiti na bazu, zato ćemo napraviti konekciju na bazu. Dobra je praksa sve podatke vezane za spajanje na bazu zapisati u neku posebnu datoteku.



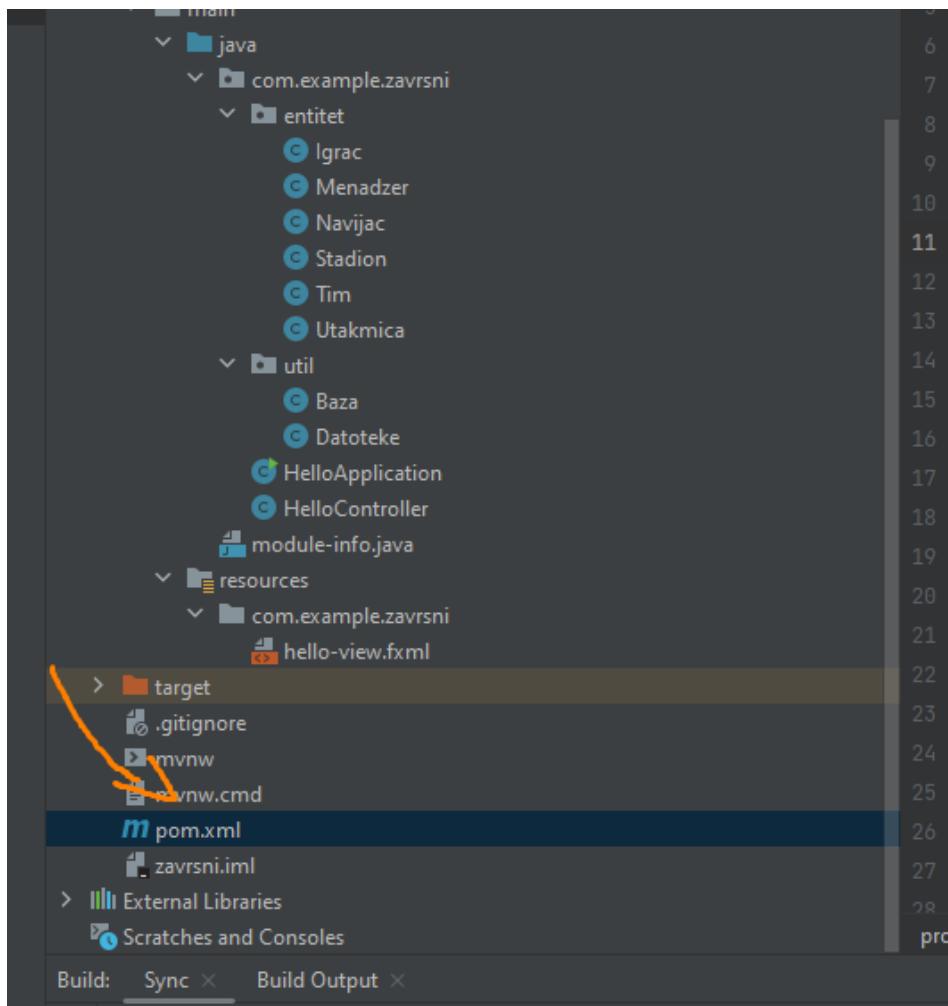
Napravit ćemo datoteku database.properties koja će nam imati zapisane sve podatke za spajanje na bazu.

A screenshot of the IntelliJ IDEA code editor showing the 'database.properties' file. The content is:

```
databaseURL = jdbc:h2:~/zavrsmiprojekt
databaseUsername = fratic
databasePassword = fratic
```

The code editor has tabs for 'hello-view.fxml', 'HelloController.java', 'Stadion.java', 'Menadzer.java', 'Igrac.java', and 'Tim.java'. The 'database.properties' tab is active.

## Da bi nam sve radilo moramo dodati Driver u pom.xml



The screenshot shows the IntelliJ IDEA code editor displaying the 'pom.xml' file. A specific section of the XML code, which defines dependencies for H2 database and JavaFX controls, is highlighted with an orange oval. The code is as follows:

```
<groupId>com.example</groupId>
<artifactId>zavrsni</artifactId>
<version>1.0-SNAPSHOT</version>
<name>zavrsni</name>

<properties>
    <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
    <junit.version>5.8.2</junit.version>
</properties>

<dependencies>
    <dependency>
        <groupId>com.h2database</groupId>
        <artifactId>h2</artifactId>
        <version>2.1.214</version>
        <scope>test</scope>
    </dependency>
    <dependency>
        <groupId>org.openjfx</groupId>
        <artifactId>javafx-controls</artifactId>
        <version>19-ea+7</version>
    </dependency>
    <dependency>
        <groupId>org.openjfx</groupId>
        <artifactId>javafx-fxml</artifactId>
        <version>19-ea+7</version>
    </dependency>
</dependencies>
```

```
2 usages
public class Baza {

    1 usage
    public static Connection connectToDatabase() throws SQLException, IOException {
        Properties configuration = new Properties();
        configuration.load(new FileReader( fileName: "dat/database.properties"));

        String databaseURL = configuration.getProperty("databaseURL");
        String databaseUsername = configuration.getProperty("databaseUsername");
        String databasePassword = configuration.getProperty("databasePassword");

        Connection vezza = DriverManager.getConnection(databaseURL, databaseUsername, databasePassword);

        return vezza;
    }

}
```

Napravit ćemo konekciju identично kao iz predavanja. Iz baze „database.properties“ citamo nase varijable „databaseURL“, „databaseUsername“ i „databasePassword“.

Da bi otvorili svoj database „online“ odemo u H2 > bin > h2

Naziv	Datum izmjene	Vrsta	Veličina
h2	13.6.2022. 21:34	Naredbena datote...	1 KB
h2.sh	13.6.2022. 21:34	SH datoteka	1 KB
h2-2.1.214.jar	13.6.2022. 21:34	JAR datoteka	2.484 KB
h2w	13.6.2022. 21:34	Naredbena datote...	1 KB

Kada prvi put radimo bazu koristimo „Embedded“

The screenshot shows the DBeaver 'Login' interface. In the top right corner, there is a dropdown menu labeled 'Saved Settings' containing a list of database connection profiles. The profile 'Generic H2 (Embedded)' is currently selected, indicated by a blue horizontal bar underneath it. Other options in the list include 'Generic H2 (Server)', 'Generic Derby (Embedded)', 'Generic Derby (Server)', 'Generic HSQLDB', 'Generic MariaDB', 'Generic MySQL', 'Generic PostgreSQL', 'Generic MS SQL Server 2005', 'Generic MS SQL Server 2000', 'Generic Oracle', 'Generic DB2', 'Generic SQLite', 'Generic Firebird Server', 'Generic Azure SQL', 'Generic Hive', 'Generic Hive 2', 'Generic Impala', 'Generic Redshift', and 'Generic Snowflake'. The rest of the dialog includes fields for 'Setting Name', 'Driver Class', 'JDBC URL', 'User Name', and 'Password'.

Nakon sto napravimo bazu koristimo „Server“.

Login

Saved Settings:

Setting Name:

---

Driver Class:

JDBC URL:

User Name:

Password:

U bazi moramo kreirati sve tablice iz kojih ce se citati podaci

Run Run Selected Auto complete Clear SQL statement:

```
CREATE TABLE IGRAC(
    id LONG NOT NULL GENERATED ALWAYS AS IDENTITY,
    ime VARCHAR(30) NOT NULL,
    prezime VARCHAR(30) NOT NULL,
    oib VARCHAR(15) NOT NULL,
    datum_rodjenja DATE NOT NULL,
    drzava VARCHAR(50) NOT NULL,
    PRIMARY KEY(id)
);
```

I moramo u njih unijeti sve podatke.

```
INSERT INTO IGRAC(ime, prezime, oib, datum_rodjenja, drzava)
VALUES ('Luka', 'Modric', '02460234892', '1989-11-15', 'Hrvatska');
```

snip Run Run Selected Auto complete Clear SQL statement:

```
SELECT * FROM IGRAC |
```

5	Iker	Casillas	02719371928	1981-05-20	Spain
6	Wayne	Rooney	02816491722	1987-05-05	England
7	Pedro	Dasilva	02460234892	1989-11-15	Brazil
8	Luka	Modric	01420532193	1980-06-05	Hrvatska
9	Robert	Mane	12421542491	1983-07-15	France
10	Ying	Thei	32124527418	1988-09-20	Japan
11	Neun	Son	02463331825	1990-10-16	Korea
12	Dame	Stirling	02133256223	1991-10-01	England
13	Steve	Dong	02142361728	1990-03-03	France
14	Antonio	Perisic	03442361838	1998-04-15	Hrvatska
15	Odin	Bale	01442566871	1996-05-15	Norway
16	Mark	Houl	02432265867	1994-06-20	Norway
17	Steph	Colw	01422465861	1996-07-21	Morocco
18	Mathew	Andrew	02342267821	1994-09-12	Canada
19	Jererny	Clark	13445752238	1999-01-25	Canada
20	Decklan	Stone	12243721232	1996-02-06	Portugal
21	Mont	Carl	13247721262	1993-04-08	Canada
22	Joseph	Dean	12742731252	1997-05-09	Belgium

Pošto ne možemo jednom timu pridodati 11 igrača kao što smo mogli preko datoteke, napravili smo 2 tima sa id-om menagera

The screenshot shows a database interface with a sidebar containing a tree view of database objects:

- jdbc:h2:tcp://localhost/~/zavrsnip
- IGRAC
- MENADZER
- STADION
- TIM
- TIM\_IGRACI
- INFORMATION\_SCHEMA
- Users
- H2 2.1.214 (2022-06-13)

At the top right are buttons: Run, Run Selected, Auto complete, Clear, and SQL statement:.

The main area contains two queries:

```
SELECT * FROM TIM
```

ID	MENADZER_ID
1	1
2	2

(2 rows, 3 ms)

```
SELECT * FROM MENADZER;
```

ID	IME	PREZIME	OIB	DATUM_RODJENJA
1	Andrea	Fratric	02342342341	1997-02-16
2	David	Daves	00031235123	1989-02-02
3	Jose	Mourinho	33123867129	1963-01-26

(3 rows, 2 ms)

TIM ID 1 ima MENADZER\_ID 1 odnosno > Andrea Fratric

TIM koji ima ID 2 ima MENADZER\_ID 2 odnosno > David Daves

The screenshot shows a database interface with a sidebar containing a tree view of database objects:

- jdbc:h2:tcp://localhost/~/zavrsnip
- IGRAC
- MENADZER
- STADION
- TIM
- TIM\_IGRACI
- INFORMATION\_SCHEMA
- Users
- H2 2.1.214 (2022-06-13)

At the top right are buttons: Run, Run Selected, Auto complete, Clear, and SQL statement:.

The main area contains a query:

```
SELECT * FROM MENADZER;
```

ID	IME	PREZIME	OIB	DATUM_RODJENJA
1	Andrea	Fratric	02342342341	1997-02-16
2	David	Daves	00031235123	1989-02-02
3	Jose	Mourinho	33123867129	1963-01-26

(3 rows, 2 ms)

I zatim smo tim id-evima pridodali igrače sa novom tablicom TIM\_IGRACI.

The screenshot shows a database interface with a sidebar containing schema and connection information, and a main panel for running SQL queries and displaying results.

**Sidebar:**

- jdbc:h2:tcp://localhost/~/zavrsniprojekat
- + IGRAC
- + MENADZER
- + STADION
- + TIM
- + TIM\_IGRACI
- + INFORMATION\_SCHEMA
- + Users
- (i) H2 2.1.214 (2022-06-13)

**Main Panel:**

Run Selected Auto complete Clear SQ

```
SELECT * FROM TIM_IGRACI
```

1	6
1	7
1	8
1	9
1	10
1	11
2	12
2	13
2	14
2	15
2	16
2	17
2	18
2	19
2	20
2	21
2	22

(22 rows, 3 ms)

Sada tim 1 ima igrače sa ID-em od 1 do 11, a tim 2 ima igrače sa ID-em 12-22.

Sada cemo napraviti metodu koja citi redak po redak i vraca sve pročitane retke iz tablice.

Npr: citamo redak po redak u tablici IGRAC.

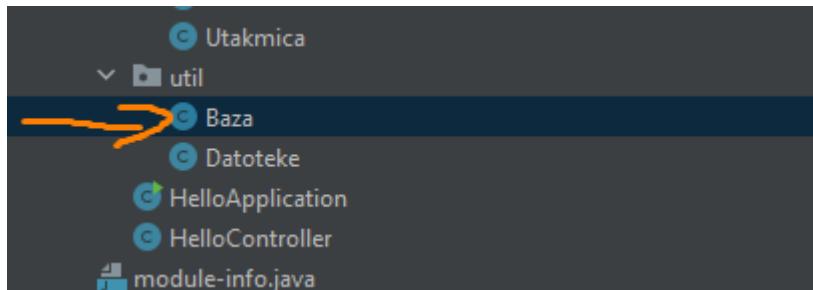
Posto tablica IGRAC izgleda ovako:

SELECT * FROM IGRAC;						
ID	IME	PREZIME	OIB	DATUM_RODJENJA	DRZAVA	
1	Josko	Joskic	01233214421	1999-01-01	Hrvatska	
2	Lionel	Messi	00212345812	1987-06-24	Argentina	
3	Cristiano	Ronaldo	08324823471	1985-02-05	Portugal	
4	Zlatan	Ibrahimovic	05872837824	1981-03-10	Sweden	
5	Iker	Casillas	02719371928	1981-05-20	Spain	
6	Wayne	Rooney	02816491722	1987-05-05	England	
7	Pedro	Dasilva	02460234892	1989-11-15	Brazil	
8	Luka	Modric	01420532193	1980-06-05	Hrvatska	
9	Robert	Mane	12421542491	1983-07-15	France	
10	Ying	Thei	32124527418	1988-09-20	Japan	
11	Neun	Son	02463331825	1990-10-16	Korea	
12	Dame	Strling	02133256223	1991-10-01	England	
13	Steve	Dong	02142361728	1990-03-03	France	
14	Antonio	Perisic	03442361838	1998-04-15	Hrvatska	
15	Odin	Bale	01442566871	1996-05-15	Norway	
16	Mark	Houl	02432265867	1994-06-20	Norway	

Znaci da cemo morati pročitati ID, IME, PREZIME, OIB, DATUM\_RODENJA i DRZAVU od svakog igrača.

Da bi ih procitali moramo napraviti SQL „STATEMENT“. Pošto čitamo iz baze to ćemo napraviti sa QUERY-em „SELECT“.

Sve to ćemo pisati u klasi „Baza“



Primjer koda:

```
1 usage
  ↴   public static List<Igrac> dohvatiIgraceIzBaze(Connection veza) throws SQLException {
      List<Igrac> igracLista = new ArrayList<>();

      Statement sqlStatement = veza.createStatement();

      ResultSet igracResultSet = sqlStatement.executeQuery( sql: "SELECT * FROM igrac");

      while(igracResultSet.next()) {
          Long igracId = igracResultSet.getLong( columnLabel: "ID");
          String ime = igracResultSet.getString( columnLabel: "IME");
          String prezime = igracResultSet.getString( columnLabel: "PREZIME");
          String oib = igracResultSet.getString( columnLabel: "OIB");
          LocalDate datumRodjenja = igracResultSet.getDate( columnLabel: "DATUM_RODJENJA").toLocalDate();
          String drzava = igracResultSet.getString( columnLabel: "DRZAVA");

          Igrac igrac = new Igrac(igracId,ime,prezime,oib,datumRodjenja,drzava);

          igracLista.add(igrac);
      }

      return igracLista;
  }
```

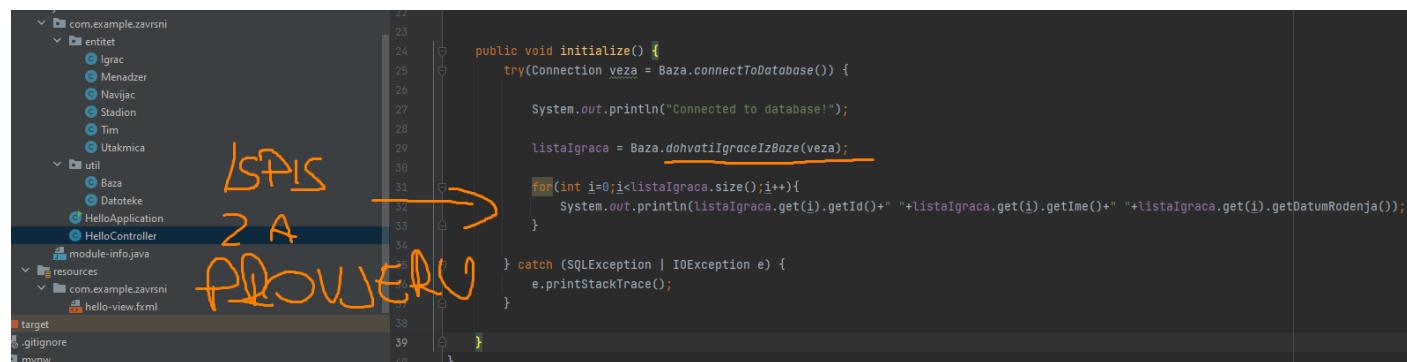
Napravili smo listu igrača da ćemo spremati sve igrače.

Pripremili smo se za čitanje podataka tako da smo napravili „statement“ i zatim smo u tom „statement-u“ executali Query koji želimo da se izvrši u našoj bazi. U našem slučaju napravili smo SELECT \* FROM igrac jer će nam taj Query vratiti sve retke u tablici „igrac“

Sa igracResultSet spremamo sve podatke, sa .next() tohvaćamo redak po redak i zatim čitamo stupac po stupac (id, ime prezime...).

Moramo paziti da DATUM\_RODJENJA pretvorimo u LocalDate() uz pomoć .toLocalDate() jer se u H2 bazi svi datumi zapisuju u obliku „DATE“, a ne „LocalDate“.

Nakon što smo napravili metodu koja dohvaća sve podatke i spremi ih u listu igrača, pozvat ćemo je u „HelloController-u“. Razlog zašto tu metodu pozivamo tamo je taj što se „HelloController“ prvi poziva prilikom paljenja naše aplikacije, tako da želimo da se svi podaci iz baze pročitaju i odmah spreme u zasebne liste.

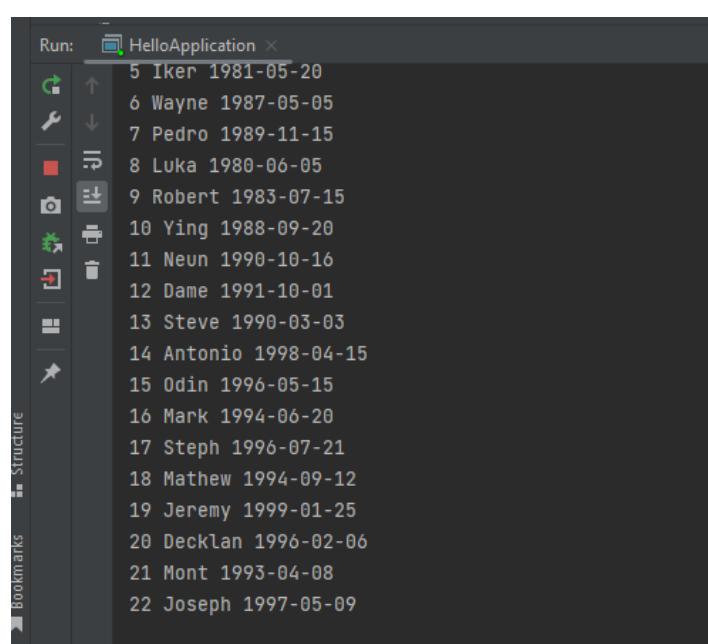


The screenshot shows the Java code for the `loadPlayers` method in the `HelloController` class. The code connects to the database, retrieves a list of players, and prints each player's ID, name, and birth date. The code is annotated with handwritten notes: 'LSPI' and 'Z A' are written above the first two lines of the loop, and 'PROVJERU' is written below the loop body.

```
public void initialize() {
    try( Connection veza = Baza.connectToDatabase() ) {
        System.out.println("Connected to database!");

        listaIgraca = Baza.dohvatiIgraceIzBaze(veza);

        for(int i=0;i<listaIgraca.size();i++){
            System.out.println(listaIgraca.get(i).getId()+" "+listaIgraca.get(i).getIme()+" "+listaIgraca.get(i).getDatumRodjenja());
        }
    } catch (SQLException | IOException e) {
        e.printStackTrace();
    }
}
```



The screenshot shows the application running in the IDE, displaying a list of 22 players. Each player is listed with their ID, name, and birth date. The data is identical to the output shown in the code screenshot.

ID	Ime	Datum Rodjenja
5	Iker	1981-05-20
6	Wayne	1987-05-05
7	Pedro	1989-11-15
8	Luka	1980-06-05
9	Robert	1983-07-15
10	Ying	1988-09-20
11	Neun	1990-10-16
12	Dame	1991-10-01
13	Steve	1990-03-03
14	Antonio	1998-04-15
15	Odin	1996-05-15
16	Mark	1994-06-20
17	Steph	1996-07-21
18	Mathew	1994-09-12
19	Jeremy	1999-01-25
20	Decklan	1996-02-06
21	Mont	1993-04-08
22	Joseph	1997-05-09

Kada smo ustanovili da sve radi, postupak čemo ponoviti za ostale entitete (Menadzere, stadione, timove, igrače, utakmice)

Pošto za timove imamo 2 tablice morat ćemo napraviti 2 SQL upita. Jedan upit će biti za tablicu „TIM“, a drugi upit će biti za tablicu „TIM\_IGRACI“. Razlog zašto to radimo je taj što ne možemo jednom timu dodati 11 igrača od jednom, nego moramo dodavati jednog po jednog, zato smo to napravili u novoj tablici.

```
1 usage
public static List<Tim> dohvatiTimoveIzBaze(Connection veza) throws SQLException {
    List<Tim> timLista = new ArrayList<>();

    Statement sqlStatement = veza.createStatement();

    ResultSet stadionResultSet = sqlStatement.executeQuery( sql: "SELECT * FROM TIM");

    while(stadionResultSet.next()) {
        Long timID = stadionResultSet.getLong( columnLabel: "ID");
        Long menadzer= stadionResultSet.getLong( columnLabel: "MENADZER_ID");

        Tim tim = new Tim(timID,new HashSet<>(), HelloController.listaMenadzera.get(Math.toIntExact(menadzer)-1));

        timLista.add(tim);
    }

    return timLista;
}
```

**Mali update:** umjesto `List<Igrac>` listalgraca koja se nalazila u `Tim` klasi, promijenit ću Listu u Set igrača (tako da iskoristimo Set, pošto je to potrebno iskoristiti za projekt). Set je isti kao i lista samo što ne podržava duplike. Također ćemo kao i prije u klasi „Tim“ Overrideati `equals` i `hashCode` da bi naš Set dobro funkcionirao.

```

java
  com.example.zavrsni
    entitet
      Igrac
      Menadzer
      Navjac
      Stadion
    Tim
      Utakmica
    util
      Baza
      Datoteka
      HelloApplication
      HelloController
    module-info.java
  resources
    com.example.zavrsni

```

```

24   this.listaIgraca = listaIgraca;
25   this.menadzer = menadzer;
26 }
27 
28 @Override
29 public boolean equals(Object o) {
30     if (this == o) return true;
31     if (!(o instanceof Tim tim)) return false;
32     return Objects.equals(getId(), tim.getId()) && Objects.equals(getListaIgraca(), tim.getListaIgraca()) && Objects.equals(getMenadzer(), tim.getMenadzer());
33 }
34 
35 @Override
36 public int hashCode() {
37     return Objects.hash(Id(), getListaIgraca(), getMenadzer());
38 }
39

```

Prvi SQL upit nam vraća samo ID tima i menadzera koji vodi tim. Igrače još nismo dodali jer se nalaze u drugoj tablici. Sada dodajemo još jedan SQL upit koji će pročitati sve igrače koji se nalaze u tablici TIM\_IGRACI i stavitiće te igrače u Set<Igrac> od pojedinog tima.

```

zavrsni C:\Users\Danilo\Desktop\Projekt Java\zavrsni
> .idea
> .mvn
> dat
  database.properties
> h2
> src
  main
    java
      com.example.zavrsni
        entitet
          Igrac
          Menadzer
          Navjac
          Stadion
          Tim
          Utakmica
        util
          Baza
          Datoteka
          HelloApplication
          HelloController
        module-info.java
      resources
        com.example.zavrsni
        hello-view.fxml
> target
  .gitignore
  mvnw
  mvnw.cmd

```

```

102 Statement sqlStatement = veza.createStatement();
103
104 ResultSet stadionResultSet = sqlStatement.executeQuery( sql: "SELECT * FROM TIM");
105
106 while(stadionResultSet.next()) {
107     Long timID = stadionResultSet.getLong( columnLabel: "ID");
108     Long menadzer= stadionResultSet.getLong( columnLabel: "MENADZER_ID");
109
110     Tim tim = new Tim(timID,new HashSet<>(), HelloController.listaMenadzera.getLong( Math.toIntExact(menadzer)-1));
111
112     timLista.add(tim);
113 }
114
115 ResultSet igraciUTimuResultSet = sqlStatement.executeQuery( sql: "SELECT * FROM TIM_IGRACI");
116
117 while(igraciUTimuResultSet.next()) {
118     Long timID = igraciUTimuResultSet.getLong( columnLabel: "TIM_ID");
119     Long igracID= igraciUTimuResultSet.getLong( columnLabel: "IGRAC_ID");
120
121     Igrac igrac = HelloController.listaIgraca.get(Math.toIntExact(igracID)-1);
122
123     timLista.get(Math.toIntExact(timID)-1).getListaIgraca().add(igrac);
124
125 }
126
127 return timLista;
128
129

```

Dodali smo novi upit koji nam čita ID igrača, ako nam u tablici piše da se u TIM-u sa ID-em broj 1 nalazi igrač sa ID-em 2, znači da iz tablice „IGRAC“ uzimamo igrača sa ID-em 2. Ako piše da je ID igrača 15, onda uzimamo igrača sa ID-em 15... itd. Pošto su imena igrači zapisani u listi i njihovi indexi su od 0 do 21 (umjesto 1 – 22), moramo uzeti igrača sa ID-1 (broj manje).

Još jedan mini update: Zaboravih timovima dodati ime.

The screenshot shows two stacked windows of MySQL Workbench. The top window displays an SQL statement to add a column to a table:

```
ALTER TABLE TIM ADD COLUMN imetima VARCHAR(50);
```

The bottom window shows two UPDATE statements and a SELECT query:

```
update tim  
set imetima = 'Champions' where id=1;  
  
update tim  
set imetima = 'Wolfs' where id=2;
```

```
SELECT * FROM TIM;
```

ID	MENADZER_ID	IMETIMA
1	1	Champions
2	2	Wolfs

(2 rows, 1 ms)

U bazi sam im dodao ime i dodao sam liniju koja čita to ime u metodi dohvatiTimoveLzBaze() u klasi „Baza“.

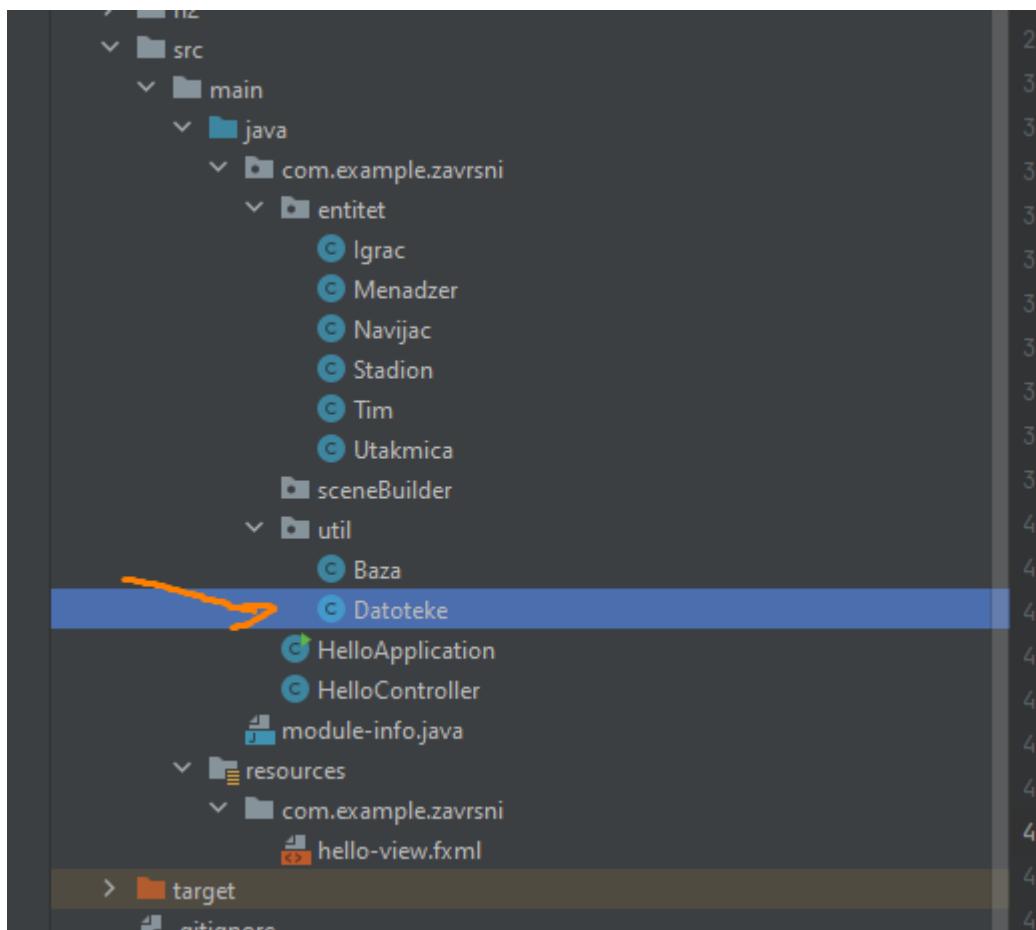
```
CREATE TABLE UTAKMICA(  
    id LONG NOT NULL GENERATED ALWAYS AS IDENTITY,  
    id_prvog_tima LONG NOT NULL,  
    id_drugog_tima LONG NOT NULL,  
    id_stadiona LONG NOT NULL,  
    datum DATE NOT NULL,  
    vrijeme VARCHAR(25) NOT NULL,  
    PRIMARY KEY(id),  
    FOREIGN KEY(id_prvog_tima ) REFERENCES TIM(id),  
    FOREIGN KEY(id_drugog_tima ) REFERENCES TIM(id),  
    FOREIGN KEY(id_stadiona ) REFERENCES STADION(id)|  
);
```

Kada radimo tablicu koja ima npr: ID\_STADIONA (pošto stadioni već postoje u tablici prije) moramo napisati da je ID\_STADIONA FOREIGN KEY koji referencira na stupac ID u tablici „STADION“. Tako dajemo do znanja da je broj koji unosimo isključivo vezan uz tablicu STADION i ne možemo napisati bilo koji ID (jer taj ID mora postojati).

Npr. ako imamo 4 stadiona i njihovi ID-evi su 1,2,3,4 onda ne možemo napisati da se utakmica odvija na stadionu 5 jer taj stadion ne postoji.

Isto tako moramo dodati FOREIGN KEY za tim. Ne možemo dodati tim sa ID-em 5 ako nemamo tim sa ID-em 5 u tablici „TIM“.

Pošto u zadatku piše da se korisniku mora omogućiti „login“ prilikom paljenja aplikacije koji će imati zapisane „username“ i „password“ u datoteci, napravili smo „Datoteke“ klasu.



U toj klasi smo napravili metodu `ucitajNavijace()` koja će učitati usrename i password iz datoteke „navijaci“. Ja koristim „Navijaci“ umjesto „Korisnici“ pošto je tema kupovanje karata za utakmice, ali radi se o istoj stvari.

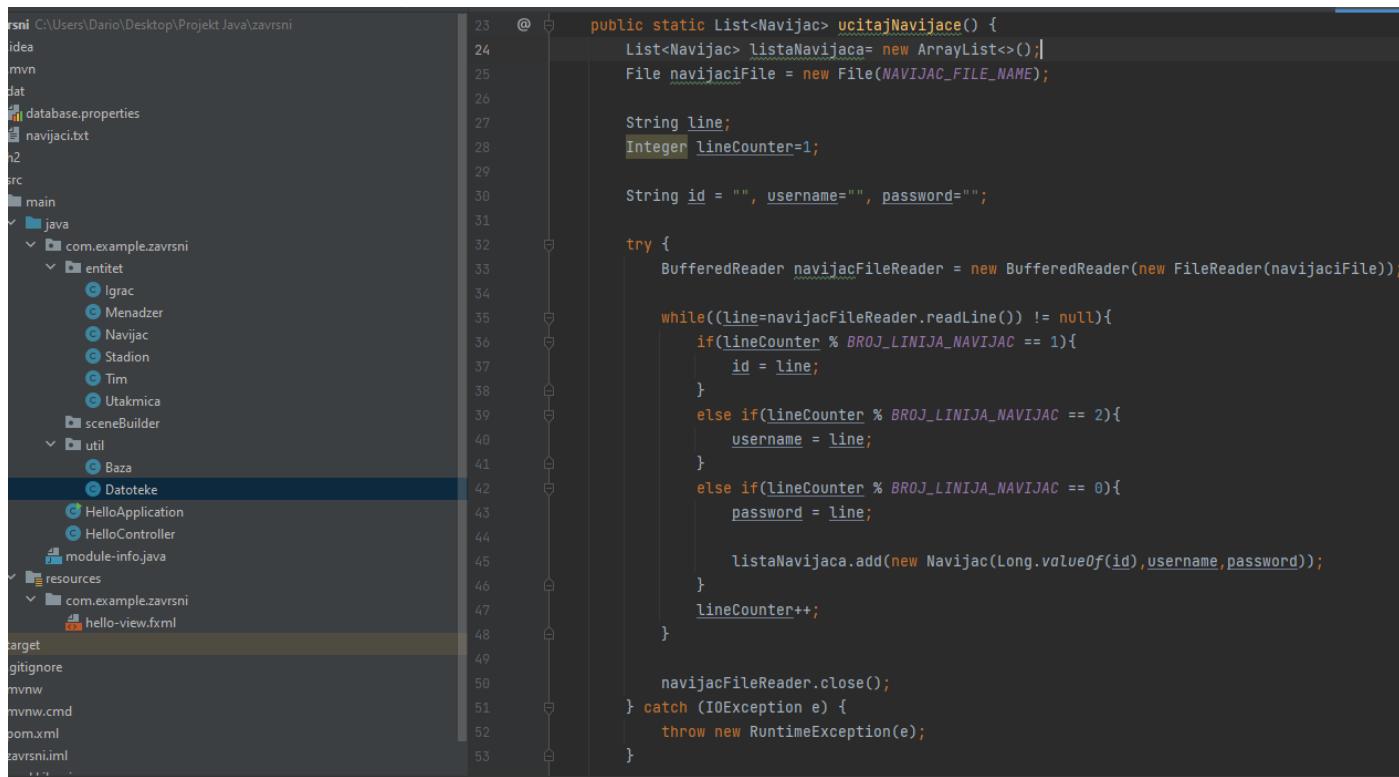
U datoteci smo napravili sljedeće username-ove i password-e:



The screenshot shows a text editor window titled "navijaci - Blok za pisanje". The menu bar includes "Datoteka", "Uređivanje", and "Prikaži". The content of the file is as follows:

```
1
admin
admin
2
andrea
fratric
3
aleksandar
radovan
```

Znači: korisnik može ući u aplikaciju SAMO AKO napiše točan username i password.



The screenshot shows an IDE interface with the following details:

- Project Structure:** The project is named "zavrnsni". It contains:
  - src/main/java/com/example/zavrnsni/
    - entitet (Igrac, Menadzer, Navijac, Stadion, Tim, Utakmica)
    - sceneBuilder
    - util (Baza, Datoteke)
    - HelloApplication
    - HelloController
  - module-info.java
  - resources/com/example/zavrnsni/hello-view.fxml
- Code View:** A Java code editor showing a method `ucitaJNavijace()`. The code reads lines from a file named `NAVIJAC_FILE_NAME` and parses them into `Navijac` objects based on their position in the file (id, username, password).

```
public static List<Navijac> ucitaJNavijace() {
    List<Navijac> listaNavijaca= new ArrayList<>();
    File navijaciFile = new File(NAVIJAC_FILE_NAME);

    String line;
    Integer lineCounter=1;

    String id = "", username="", password="";

    try {
        BufferedReader navijacFileReader = new BufferedReader(new FileReader(navijaciFile));

        while((line=navijacFileReader.readLine()) != null){
            if(lineCounter % BROJ_LINIJA_NAVIJAC == 1){
                id = line;
            }
            else if(lineCounter % BROJ_LINIJA_NAVIJAC == 2){
                username = line;
            }
            else if(lineCounter % BROJ_LINIJA_NAVIJAC == 0){
                password = line;
            }

            listaNavijaca.add(new Navijac(Long.valueOf(id),username,password));
        }
        lineCounter++;
    }

    navijacFileReader.close();
} catch (IOException e) {
    throw new RuntimeException(e);
}
```

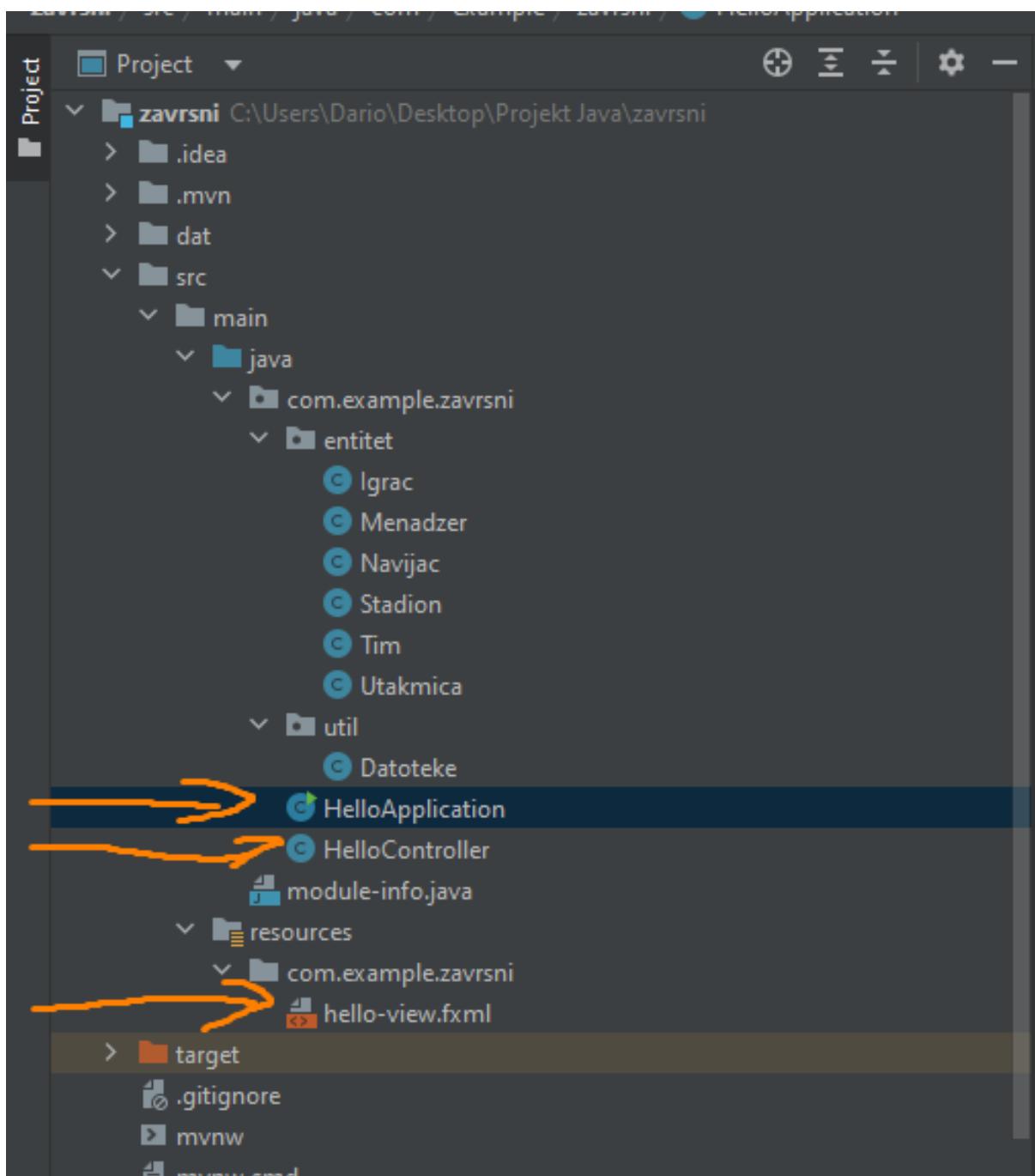
Ova metoda čita redak po redak iz datoteke „navijaci.txt“ koja se nalazi u folderu „dat“. Prvi redak sprema kao ID navijača, drugi redak sprema kao username, a treći sprema kao password.

Citamo liniju po liniju iz datoteke.

Sve procitane linije su automatski tipa „String“. Pošto nam za „id“ treba tip Long morali smo napraviti Long.valueOf(id) da bi pretvorili id koji je tipa „String“ u tip „Long“.

Sada ćemo napraviti JavaFX.

Za JavaFX nam treba Klasa koja će nam pokrenuti cijeli kod (HelloApplication). Napravit ćemo HelloController koji će nam kontrolirati početnu scenu, i fxml file koji će se prikazivati korisniku.

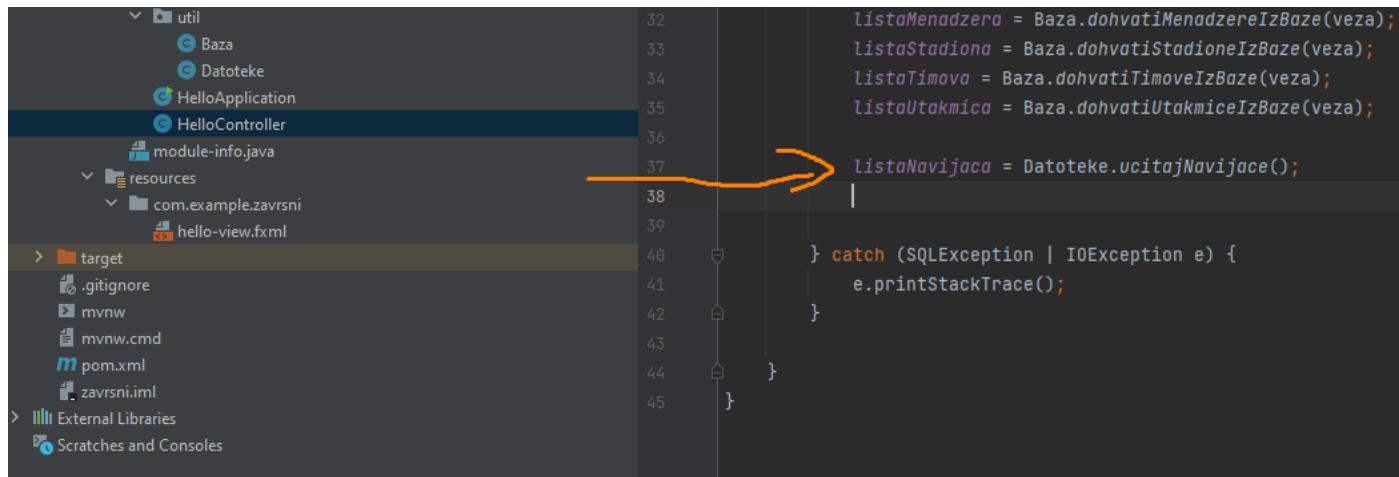


Pošto se HelloController pokreće prvi (odma čim upalimo program), u njegovoj initialize() metodi cemo napraviti inicijalizacije svih lista (Igraca, Menadzera, Stadiona, Timova, Utakmica)

U naše liste spremamo sve ono što smo pročitali iz baze.

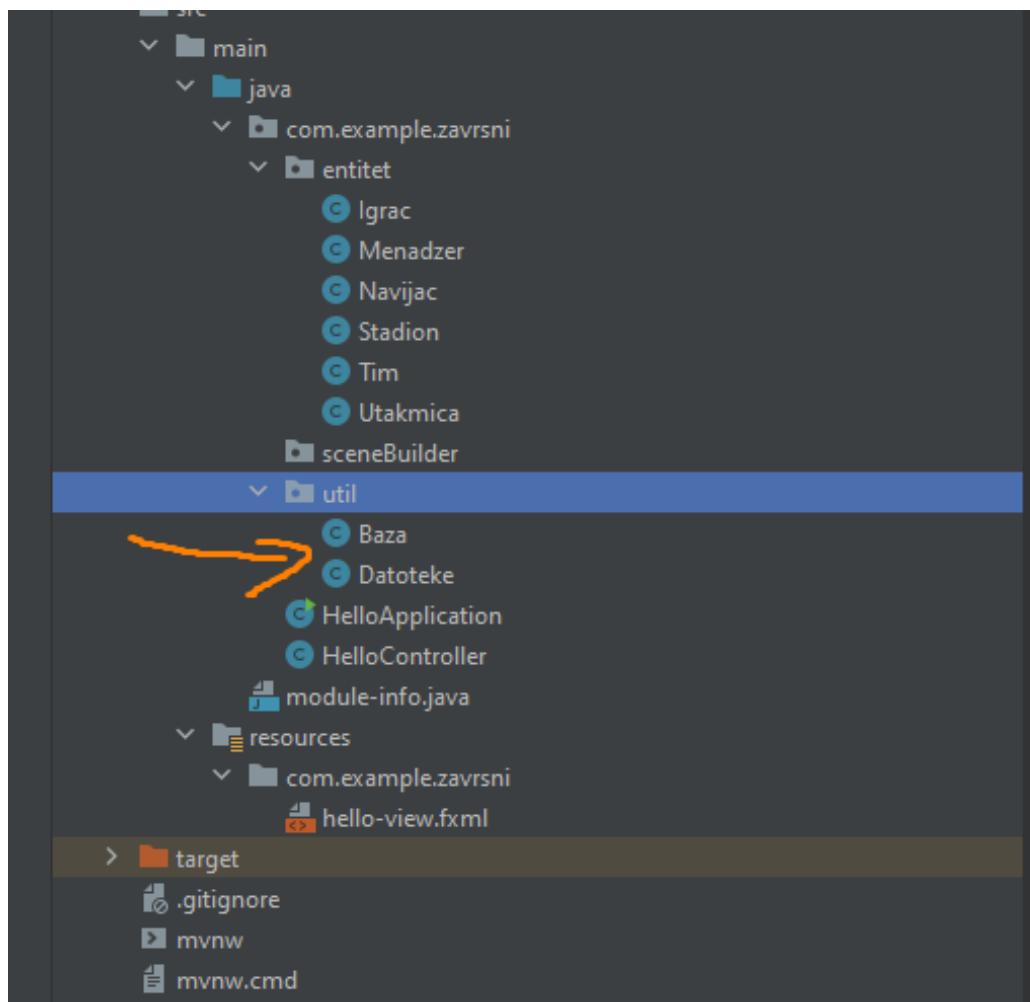
```
7 usages
13  public class HelloController {
14      2 usages
14  public static List<Igrac> listaIgraca = new ArrayList<>();
15      2 usages
15  public static List<Menadzera> ListaMenadzera = new ArrayList<>();
16
17      2 usages
17  public static List<Stadion> listaStadiona = new ArrayList<>();
18
19      3 usages
19  public static List<Tim> listaTimova = new ArrayList<>();
20
21      1 usage
21  public static List<Utakmica> listaUtakmica = new ArrayList<>();
22
23      5 usages
23  public static List<Navijac> listaNavijaca = new ArrayList<>();
24
25
26  public void initialize() {
27      try(Connection veza = Baza.connectToDatabase()) {
28
29          System.out.println("Connected to database!");
30
31          listaIgraca = Baza.dohvatiIgraceIzBaze(veza);
32          listaMenadzera = Baza.dohvatiMenadzereIzBaze(veza);
33          listaStadiona = Baza.dohvatiStadioneIzBaze(veza);
34          listaTimova = Baza.dohvatiTimoveIzBaze(veza);
35          listaUtakmica = Baza.dohvatiUtakmiceIzBaze(veza);
36
```

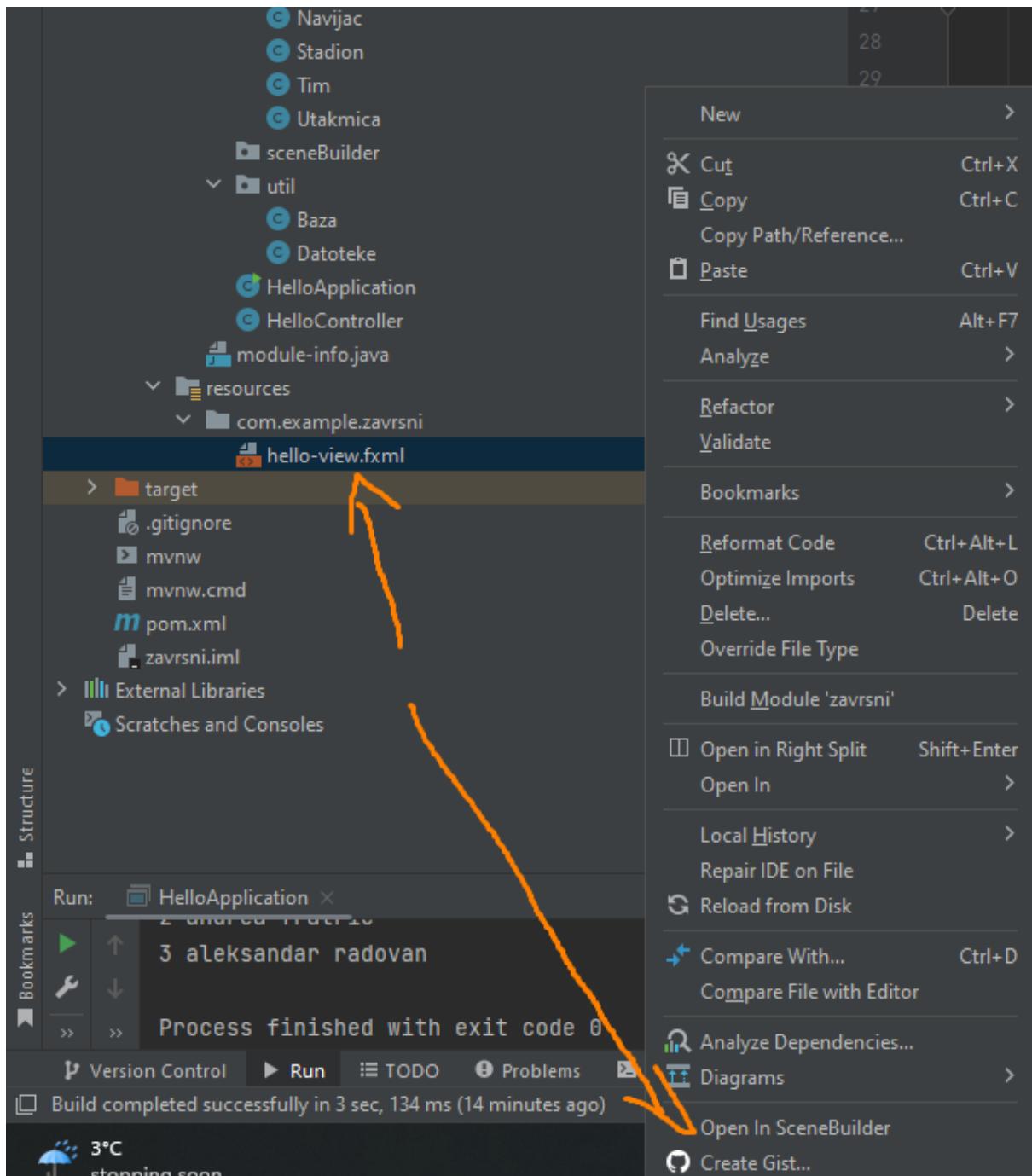
Također spremamo i sve navijače sa njihovim username-ovima i passwordima koje smo zapisali u datoteci.



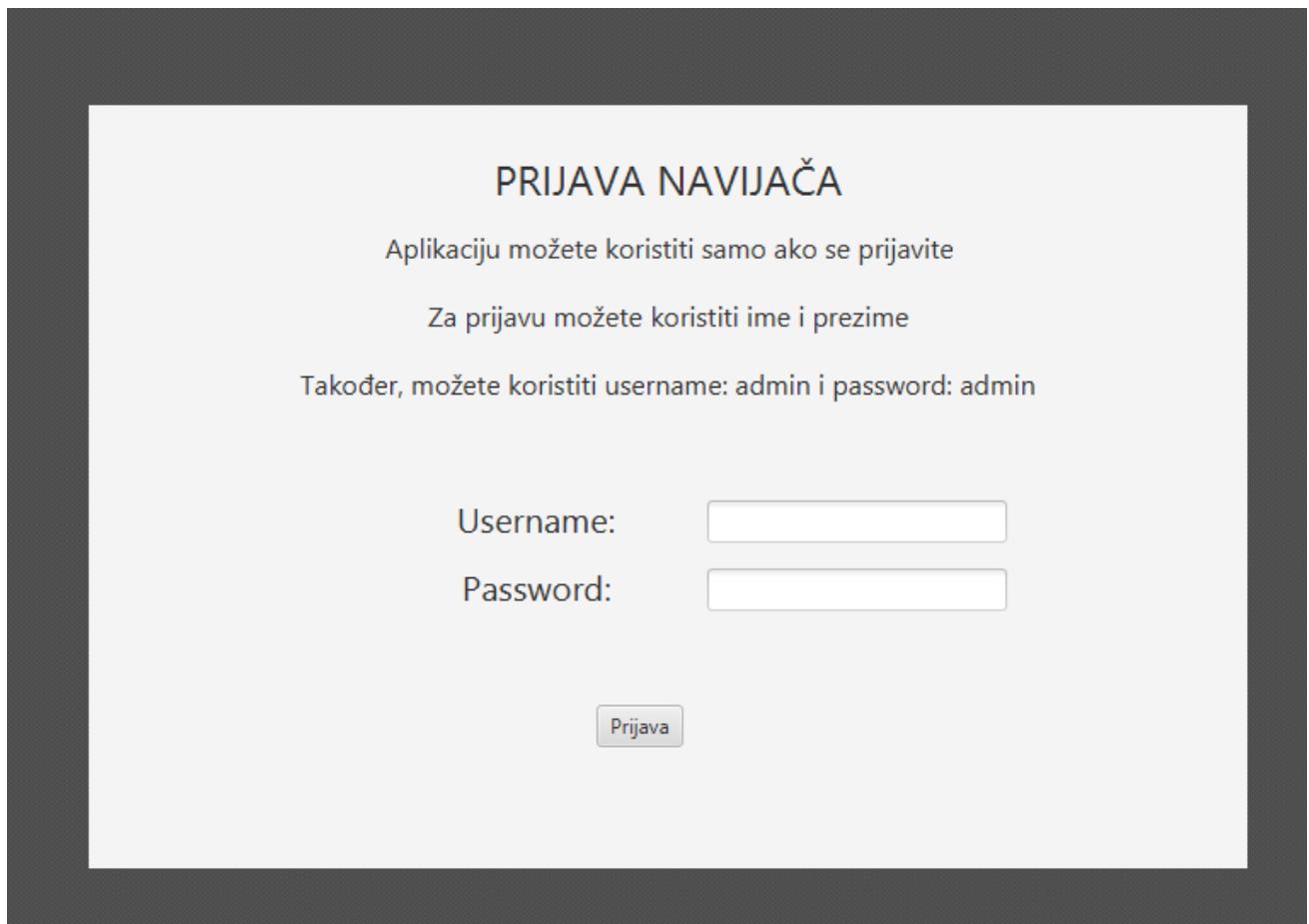
```
32 listaMenadzera = Baza.dohvatiMenadzereIzBaze(veza);
33 listaStadiona = Baza.dohvatiStadioneIzBaze(veza);
34 listaTimova = Baza.dohvatiTimoveIzBaze(veza);
35 listaUtakmica = Baza.dohvatiUtakmiceIzBaze(veza);
36
37 listaNavijaca = Datoteke.ucitajNavijace();
38
39
40 } catch (SQLException | IOException e) {
41     e.printStackTrace();
42 }
43
44 }
45 }
```

Sve metode za čitanje iz Baze se nalaze u klasi „Baza“, a sve metode koje služe za čitanje iz datoteka se nalaze u klasi „Datoteke“.





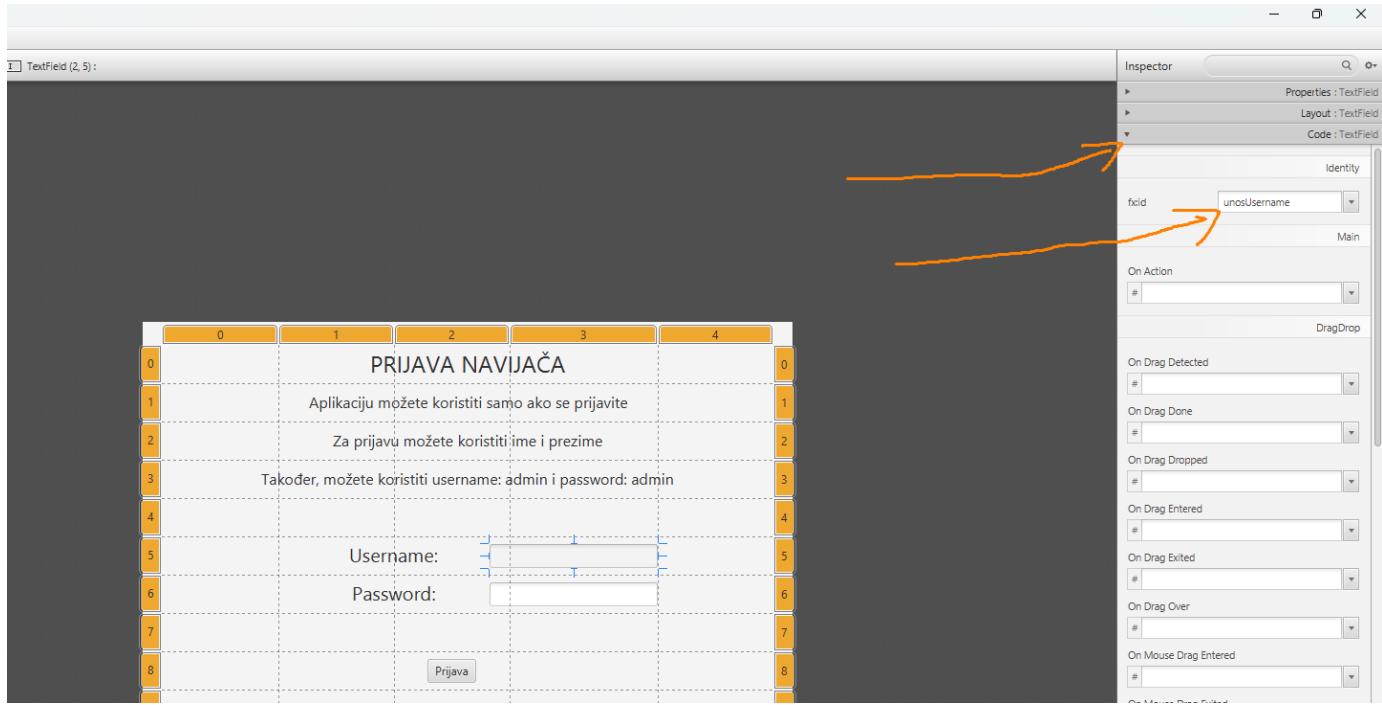
Ako kliknemo desni klik na hello-view možemo kliknuti „Open in SceneBuilder“ i tako oblikovati naš pocetni zaslon.



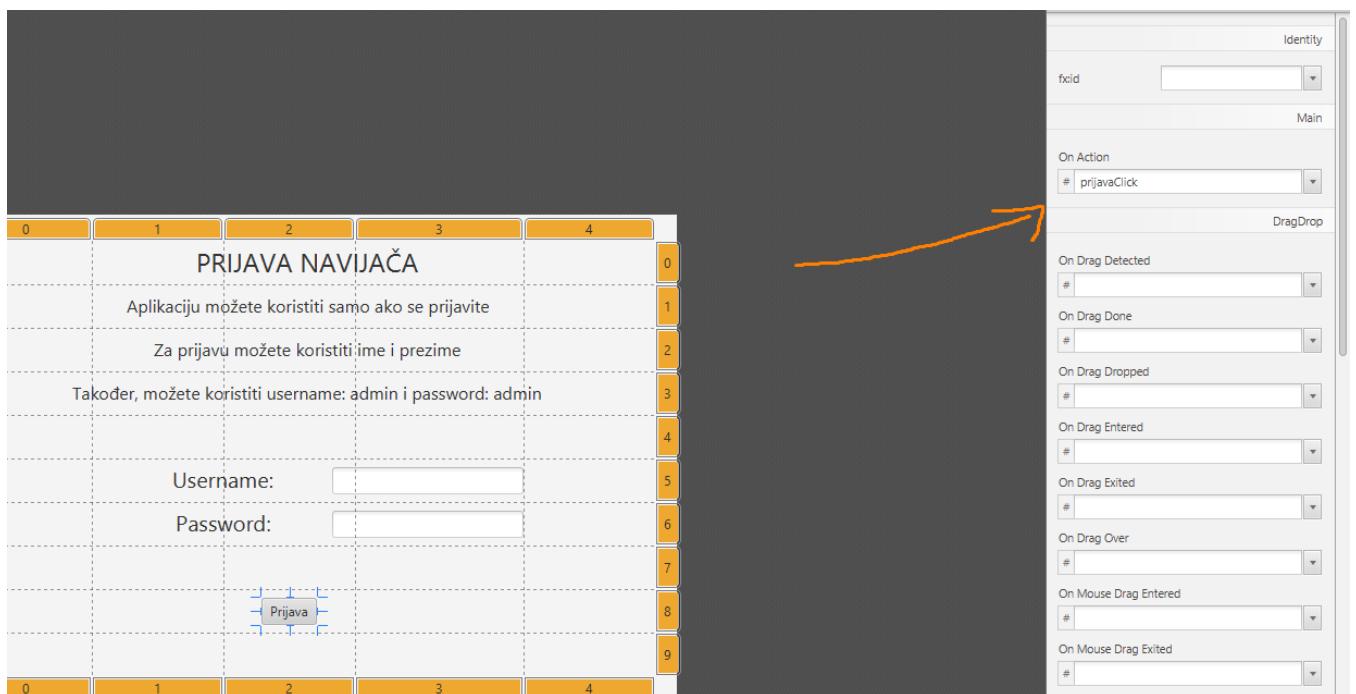
Dizajnirali smo ovaku scenu i povezali smo TextField za username i password sa varijablama u IntelliJ-u.

```
22
23     1 usage
24     public static List<Utakmica> listaUtakmica = new ArrayList<>();
25
26     4 usages
27     public static List<Navijac> listaNavijaca = new ArrayList<>();
28
29     2 usages
30     @FXML
31     private TextField unosUsername;
32
33     2 usages
34     @FXML
35     private TextField unosPassword;
36
37
```

Povezujemo ih tako da u SceneBuilderu odemo na „Code“ i dodamo im id.



Također smo gumb povezali sa metodom „buttonClick“



Ta metoda ide po listi svih navijača koji imaju svoj username i password i gledamo ako su im uneseni username i password isti kao i oni zapisani u datoteci.

The screenshot shows a Java code editor with the following code:

```
1 usage
@FXML
private void prijavaClick(){
    String username = unosUsername.getText();
    String password = unosPassword.getText();

    Boolean provjeraLogina = false;
    for(int i=0;i<listaNavijaca.size();i++){
        if(listaNavijaca.get(i).getUsername().equals(username)){
            if(listaNavijaca.get(i).getPassword().equals(password)){
                provjeraLogina = true;
            }
        }
    }

    if(provjeraLogina == true){
        System.out.println("Uspješan login!");
    }
}
```

On the right side of the code editor, there is a terminal window titled "navijaci - Blok" showing the following output:

Index	Username	Password
1	admin	admin
2	andrea	fratric
3	aleksandar	radovan

Yellow arrows point from the code's list iteration to the terminal output, indicating the comparison of user input against the database entries.

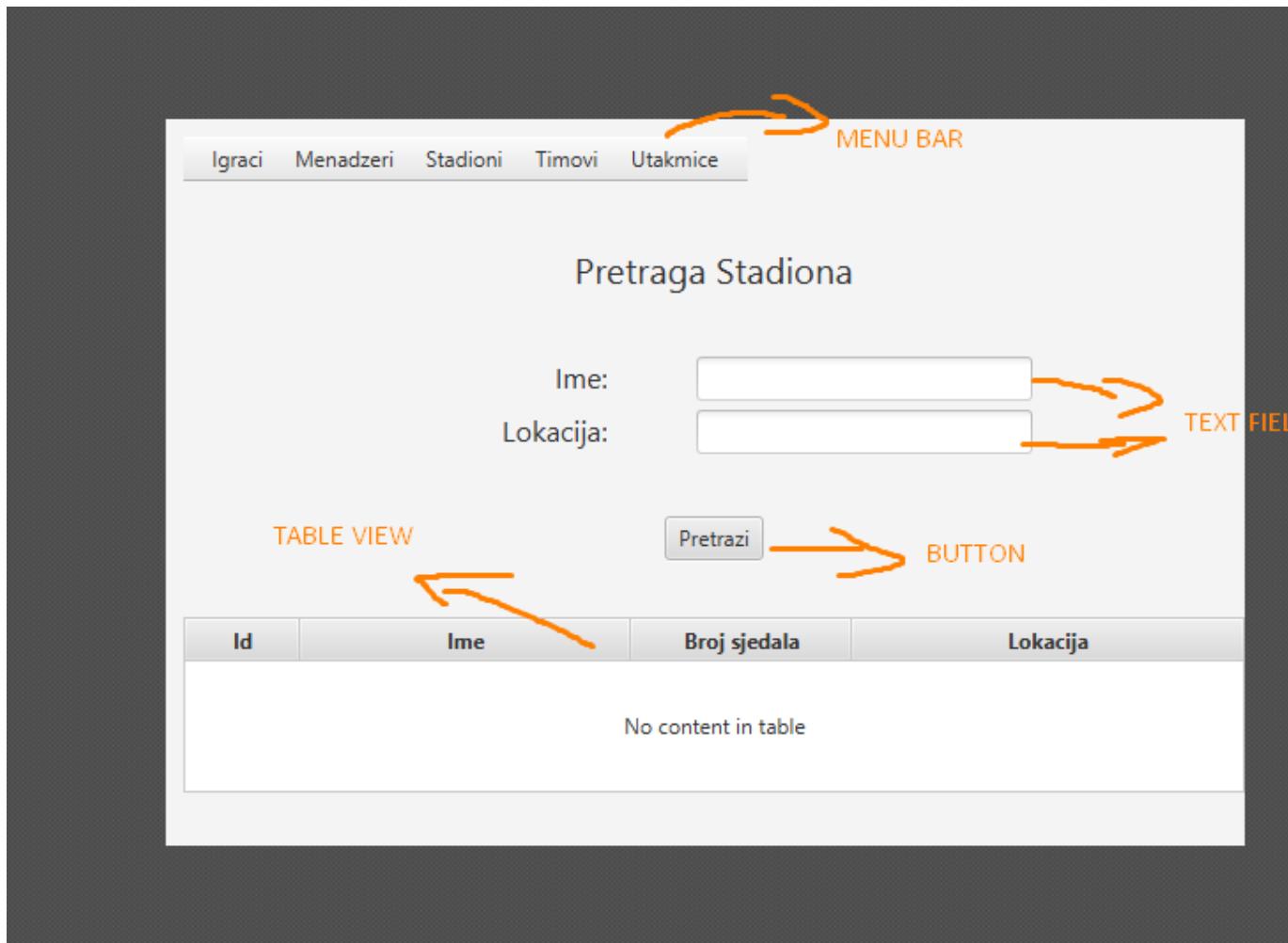
Sada moramo (identично као и у 7. vjezbi) napraviti „Search“ screen za svaki entitet.

Izdvojiti cemoMenuBar u zasebni .fxml da ga ne moramo na svakoj stranici raditi ispočetka. Nakon toga ćemo napraviti 5 fxml scena, svaka će biti za svoj entitet.

Svaka scena mora imati svoj Controller i svoj FXML. FXML nam služi da bi mogli dodavati nove gume / tablice / textFieldove i slično, a Controller nam opisuje što će ti gumbi / tablice / textFieldovi raditi.

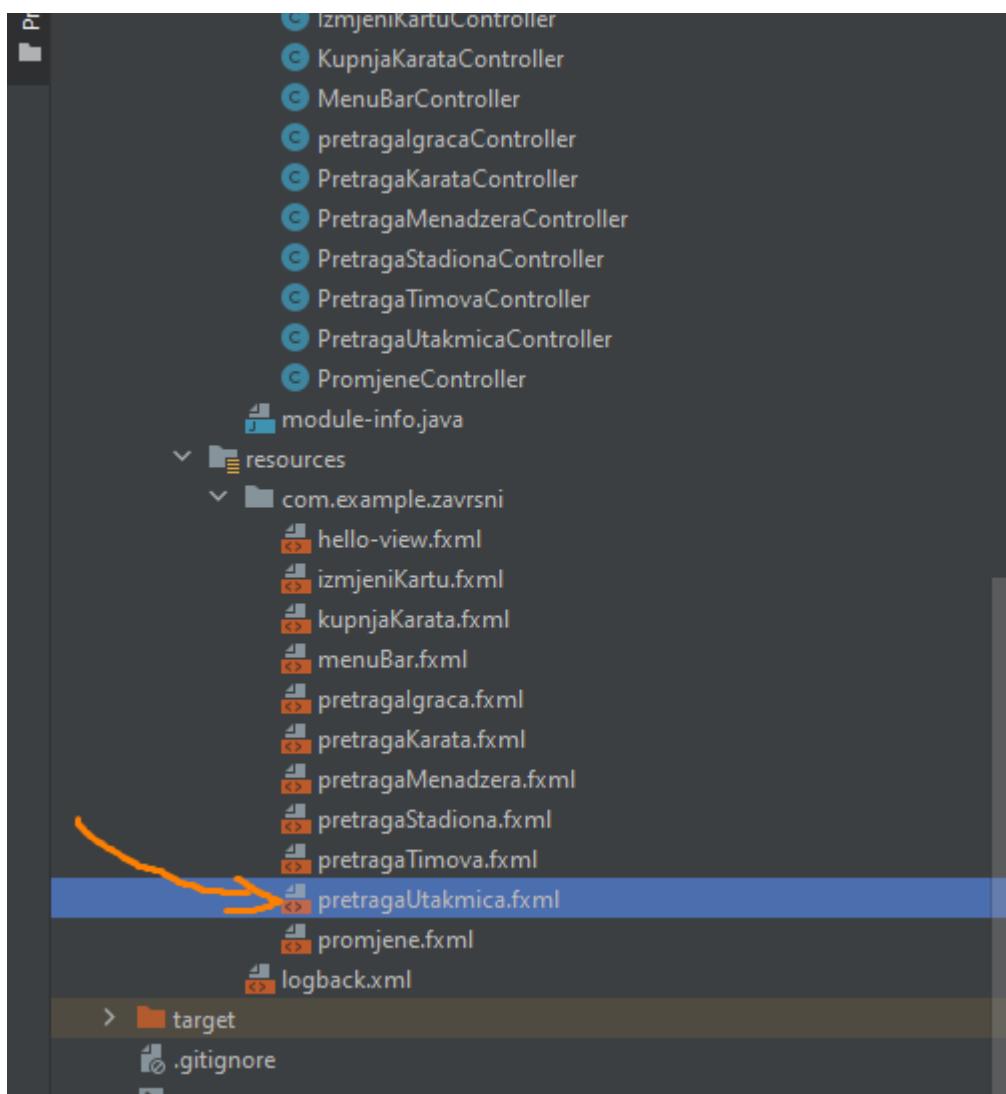
U našem slučaju sve scene će imati TableView da korisniku prikažemo sve entitete, a TextFieldovi će služiti da bi korisnik mogao unijeti neki podatak i prema tome filtrirati tablicu.

Kada kliknemo button, podaci u tablici će se filtrirati ovisno o tome što smo unijeli u TextFieldu.

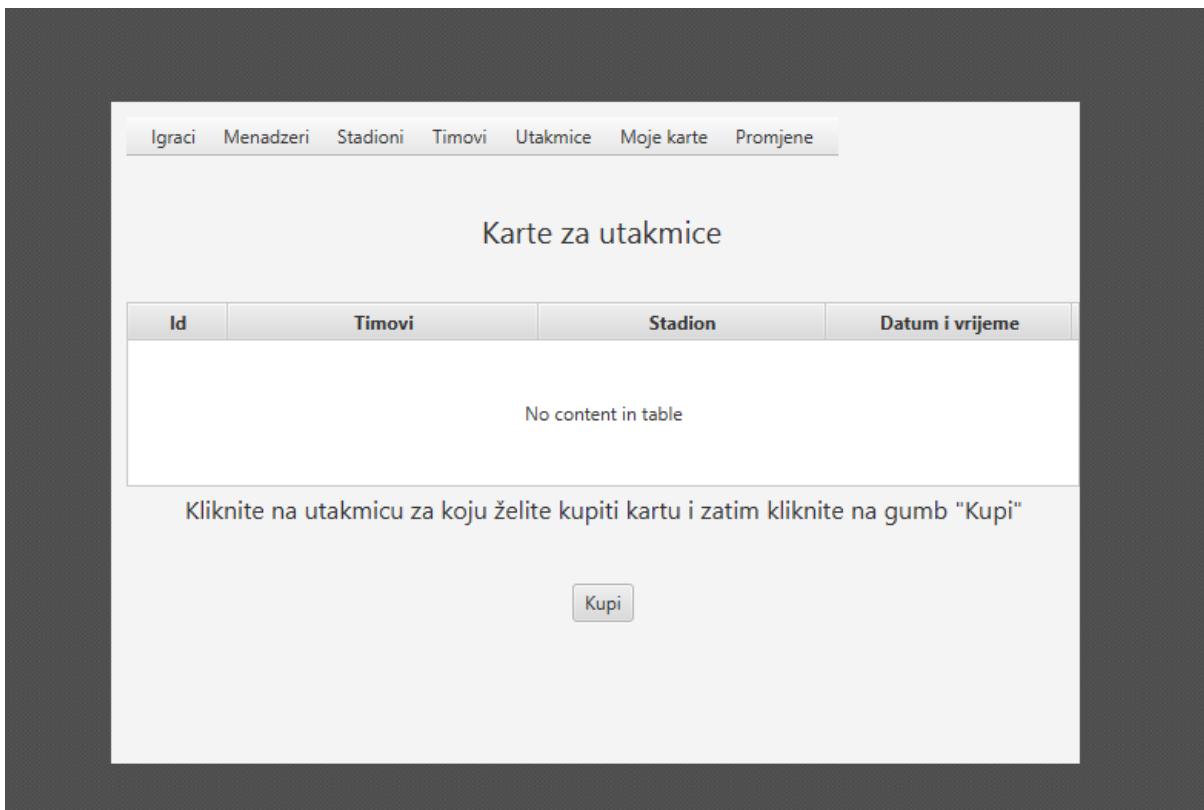


Postupak ponovovimo za sve entitete.

Sada radimo najvažniji dio aplikacije:



U pretrazi utakmica ćemo dati korisniku mogućnost da odabre utakmicu i kupi kartu za nju.



U tablici ćemo ispisati sve utakmice koje smo dohvatali iz baze.

Kada kliknemo gumb „Kupi“ uzima se index odabrane utakmice.

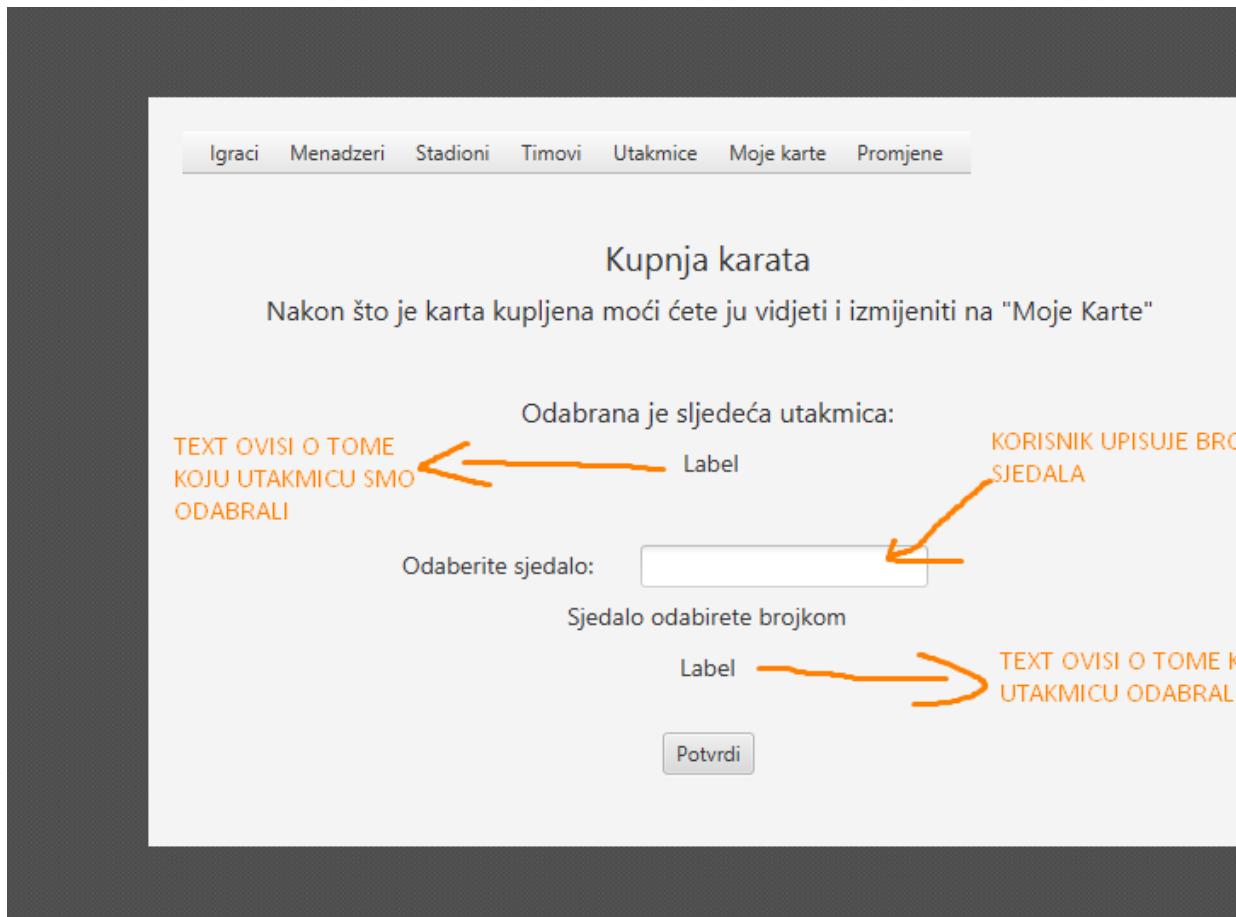
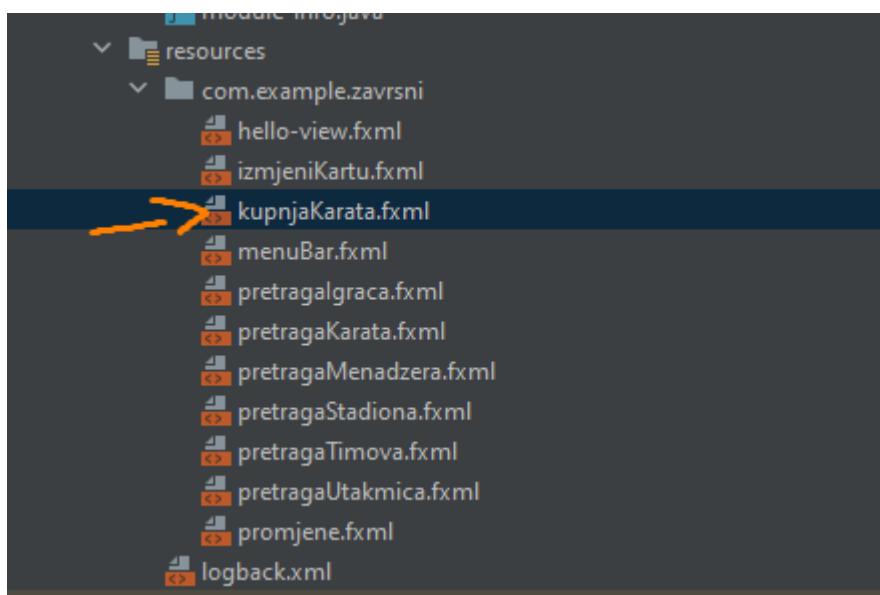
```

1 usage
@FXML
public void onKupiButtonClick() throws IOException {
    int selectedIndex = utakmicaTable.getSelectionModel().getFocusedIndex();

    odabranutakmica = listaUtakmica.get(selectedIndex); DOHVACA UTAKMICU PREKO INDEXA
    FXMLLoader fxmlLoader = new FXMLLoader(HelloApplication.class.getResource( name: "kupnjaKarata.fxml"));
    Scene scene = new Scene(fxmlLoader.load(), v: 600, v: 400);
    HelloApplication.getMainStage().setTitle("Hello!");
    HelloApplication.getMainStage().setScene(scene);
    HelloApplication.getMainStage().show();
}

```

Kada kliknemo „kupi“ prelazimo na kupnju karata



Npr: ako smo odabrali utakmicu koja se nalazi na stadionu sa 81 000 sjedala, onda korisnik može odabrati sjedalo 1 – 81 000 (ni više ni manje).

```

24     @FXML
25     private Label utakmicaLabel;
26
27     2 usages
28     @FXML
29     private Label brojSjedalaLabel;
30
31     2 usages
32     @FXML
33     private TextField odabranoSjedalo;
34
35     @FXML
36     public void initialize(){
37
38         Utakmica utakmica = PretragaUtakmicaController.odabranaUtakmica;
39
40         utakmicaLabel.setText(utakmica.getPrviTim().getImeTim() + " VS " + utakmica.getDrugiTim().getImeTim() + ", Stadion:" + utakmica.getStadion().getNaziv());
41
42     }
43

```

Kada smo kliknuli gumb „Kupi“ utakmica koju smo odabrali se spremila u globalnu varijablu. To znači da ju možemo dohvatiti iz bilo koje druge klase. U klasi „KupnjaKarataController“ smo dohvatili tu globalnu varijablu da bi znali o kojoj se utakmici radi, zatim smo ispisali sve podatke o utakmici u Label.

U prvi Label zapisujemo podatke, a u drugi Label zapisujemo koliko sjedala ima taj stadion (tako da korisnik zna koji broj sjedala može odabrati.)

Ovo je izvrstan trenutak da iskoristimo Interface. Napraviti ćemo interface koji će imati 2 metode: 1. metoda koja provjerava da li je korisnik unio dobar broj sjedala, 2. metoda koja će provjeravati da li je sjedalo koje korisnik želi zauzeto.

```

1 package com.example.zavrnsni.entitet;
2
3 import com.example.zavrnsni.exceptioni.NepostojeceSjedalo;
4 import com.example.zavrnsni.exceptioni.ZauzetoSjedalo;
5
6 1 usage 1 implementation
7 public sealed interface sjedalaStadiona permits Stadion{
8
9     1 usage 1 implementation
10    void provjeraSjedala(String sjedalo) throws NepostojeceSjedalo;
11
12    1 usage 1 implementation
13    void zauzetostSjedala(String sjedalo) throws ZauzetoSjedalo;
14
15 }

```

Također cemo iskoristiti i vlastite Exceptione koje cemo sami napraviti. Prvi exception koji cemo napraviti je NepostojeceSjedalo. Taj exception se baca kada korisnik odabere sjedalo koje ne postoji. Drugi exception će biti ZauzetoSjedalo, taj exception se baca kada korisnik odabere sjedalo koje je zauzeto.

Napravili smo da je taj interface „sealed“ odnosno zapečaćeni interface i napisali smo mu „permits Stadion“ znači da samo stadion može implementirati ovaj interface.

Sada ćemo napraviti implementaciju interface-a.



```
10 import java.io.Serializable;
11 import java.util.List;
12
13 public final class Stadion implements sjedalaStadiona
```

Moramo ga napraviti final zato što „sjedalaStadiona“ može implementirati samo Stadion, što znači, kad bi neka klasa htjela biti podklasa klasi Stadion, ona bi također implementirala sučelje „sjedalaStadiona“, ali to nije moguće jer samo Stadion može implementirati to sučelje.

Sa ključnom riječi final smo dali do znanja da niti jedna druga klasa neće moći nasljediti klasu Stadion.

Nakon što smo implementirali to sučelje moramo override-ati metode iz tog sučelja.

```

62     1 usage
63  ⚡  @Override
64      public void provjeraSjedala(String sjedalo) throws NepostojeceSjedalo {
65
66          Utakmica utakmica = PretragaUtakmicaController.odabranaUtakmica;
67
68          if(Integer.parseInt(sjedalo)<1 || Integer.parseInt(sjedalo)>utakmica.getStadion().getBrojSjedala()) {
69              throw new NepostojeceSjedalo("Odabrali ste sjedalo koje ne postoji!");
70          }
71      }
72
73      1 usage
74  ⚡  @Override
75      public void zauzetostSjedala(String sjedalo) throws ZauzetoSjedalo {
76          List<Karta> listaKarata = HelloController.listaKarata;
77
78          for(int i=0;i<listaKarata.size();i++){
79              if(listaKarata.get(i).getOdabranoSjedalo().equals(Integer.valueOf(sjedalo))) {
80                  throw new ZauzetoSjedalo("Sjedalo koje ste odabrali je zauzeto!");
81              }
82          }
83      }
84

```

Prva metoda uzima našu „globalnu utakmicu“ koju je korisnik odabrao i gleda koliki broj sjedala ima na njenom stadionu. Ako je korisnik unio broj koji je manje od 1 ili veći od maksimalnog broja sjedala na stadionu, onda se baca exception „NepostojeceSjedalo“ sa porukom „Odabrali ste sjedalo koje ne postoji“.

Druga metoda uzima listu karata iz HelloControllera (pošto su u kontroleru inicijalizirane sve liste) i ide po svakoj karti i gleda da li postoji karta sa sjedalom koje je odabrao korisnik. Ako postoji onda se baca exception ZauzetoSjedalo sa porukom „Sjedalo koje ste odabrali je zauzeto!“.

Moramo napraviti novu klasu „Karta“ koja će imati podatke o korisnikovim kartama.

```
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
```

The screenshot shows the Java code editor with the file structure on the left and the code on the right. The code is as follows:

```
@Serial
private static final long serialVersionUID = -685142501662965531L;
3 usages
private Long idKarte;
3 usages
private Navijac navijac;
3 usages
private Utakmica utakmica;
3 usages
private Integer odabranoSjedalo;
3 usages
private LocalDateTime vrijemeKupnje;

public Karta(Long idKarte, Navijac navijac, Utakmica utakmica, Integer odabranoSjedalo,
    this.idKarte = idKarte;
    this.navijac = navijac;
    this.utakmica = utakmica;
    this.odabranoSjedalo = odabranoSjedalo;
    this.vrijemeKupnje = vrijemeKupnje;
}
```

U karti imamo navijača koji je kupio kartu.

Utakmicu za koju je kupio kartu

Sjedalo koje je odabrao.

Vrijeme kad je kupio kartu.

Kako znamo koji navijač je kupio kartu?

Napravili smo početni zaslon „Login“ gdje se korisnik mora ulogirati prije početka korištenja aplikacije.

```
64
65
66
67
68
69
70
71
72
73
74
75
76
77
78
```

The screenshot shows the Java code editor with the file structure on the left and the code on the right. The code is as follows:

```
@FXML
private void prijavaClick() throws IOException {
    String username = unosUsername.getText();
    String password = unosPassword.getText();

    Boolean provjeraLogina = false;

    for(int i=0;i<listaNavijaca.size();i++){
        if(listaNavijaca.get(i).getUsername().equals(username)){
            if(listaNavijaca.get(i).getPassword().equals(password)){
                provjeraLogina = true;
                vlogiraniNavijac = listaNavijaca.get(i);
            }
        }
    }
}
```

Nakon što se korisnik ulogirao, spremili smo njegove podatke u globalnu varijablu (da bi znali koji korisnik se prijavio u sustav).

Dakle:

```
26 usages
public class Karta implements Serializable {

    @Serial
    private static final long serialVersionUID = -685142501662965531L;
    3 usages
    private Long idKarte;
    3 usages
    private Navijac navijac;
    3 usages
    private Utakmica utakmica;
    3 usages
    private Integer odabranoSjedalo;
    3 usages
    private LocalDateTime vrijemeKupnje;

    2 usages
    public Karta(Long idKarte, Navijac navijac, Utakmica utakmica, Integer odabranoSjedalo, LocalDateTime vrijemeKupnje) {
        this.idKarte = idKarte;
        this.navijac = navijac;
        this.utakmica = utakmica;
        this.odabranoSjedalo = odabranoSjedalo;
        this.vrijemeKupnje = vrijemeKupnje;
    }
}
```

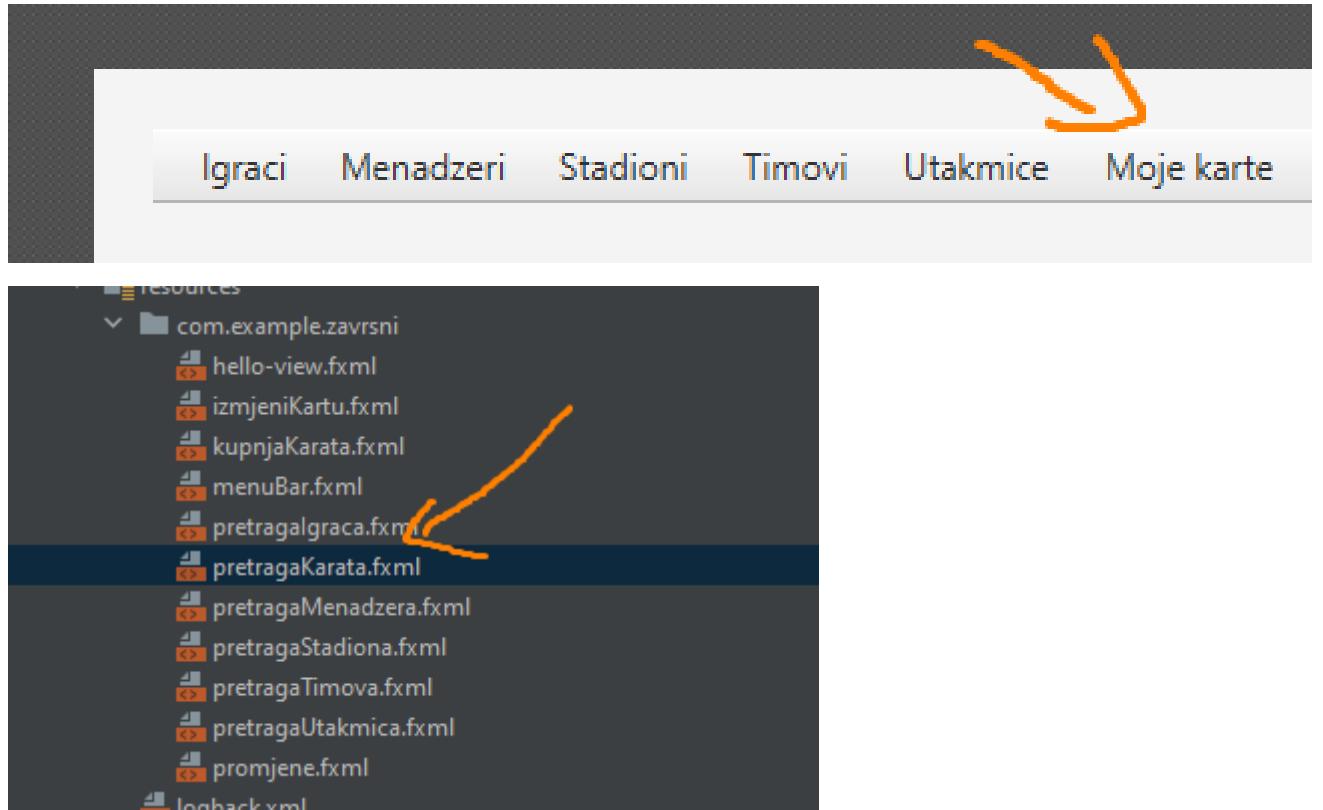
KORISNIK

Navijač koji je spremljen kao podatak o kupljenoj karti je zapravo korisnik koji se ulogirao u sustav. Tako možemo filtrirati karte za svakog korisnika.

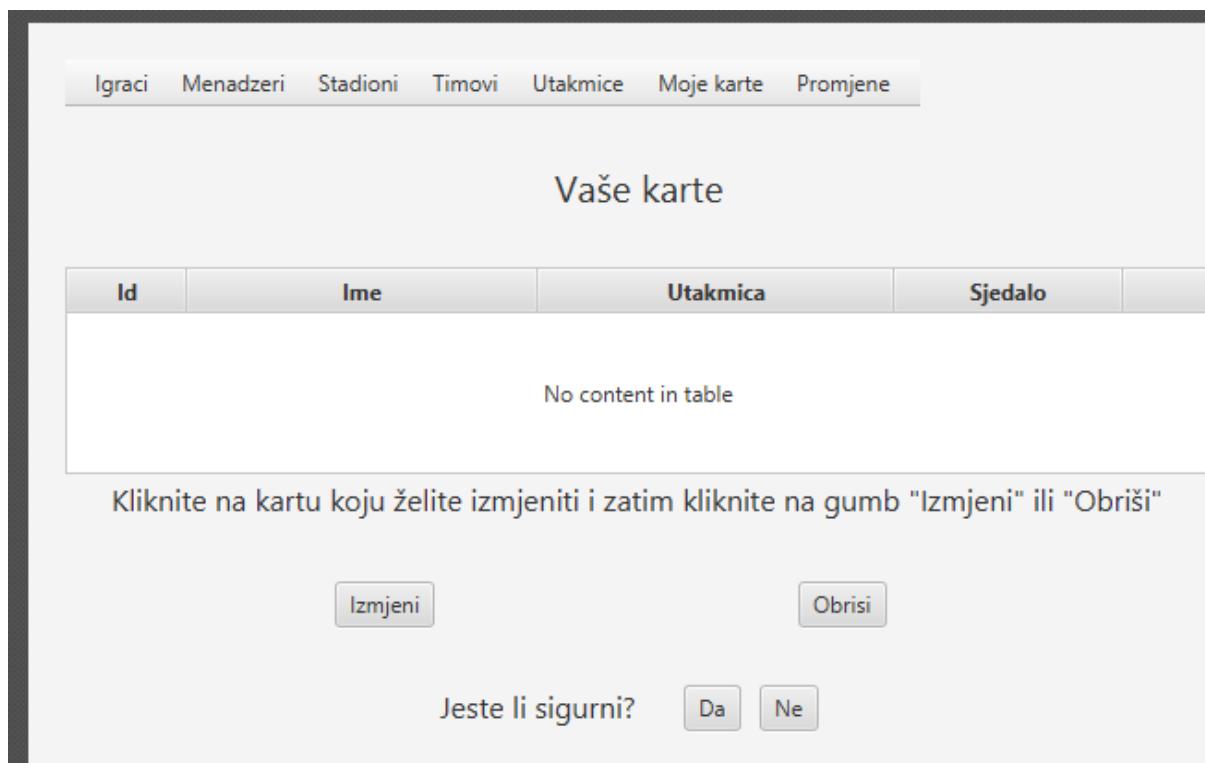
Utakmica je ista ona „globalna varijabla“ utakmice koju smo spremili kada je korisnik kliknuo za koju utakmicu želi kartu.

Sjedalo je broj sjedala koju je korisnik unio na sceni „kupnjaKarata“ (vidi zadnju sliku na strani 33.)

Napraviti ćemo novu scenu koju ćemo dodati u Menu i nazvati „Moje karte“



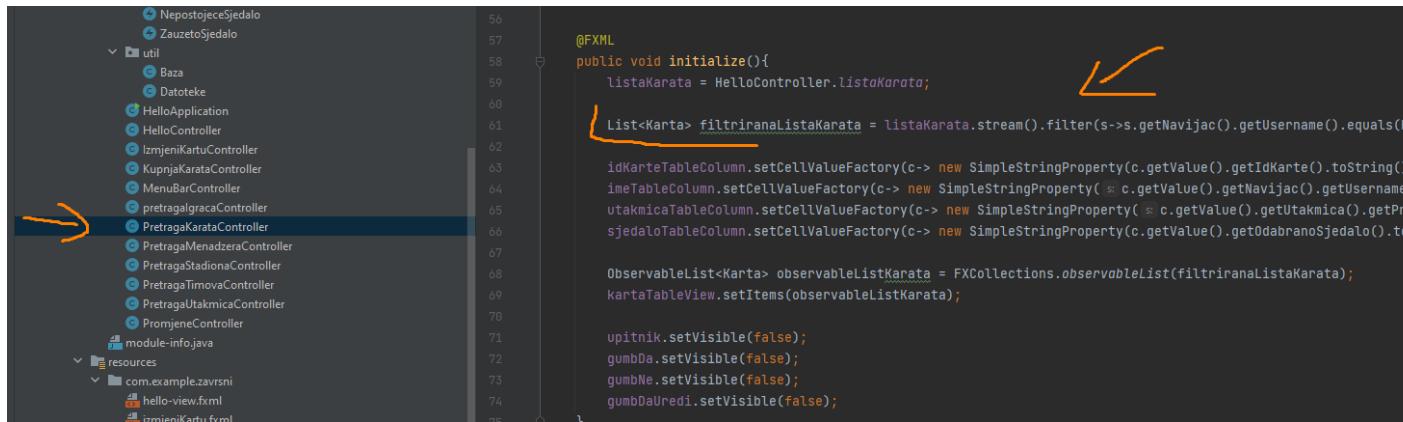
Napraviti ćemo novi fxml koji će nam služiti za uređivanje Scene za pregled karata. Ovako izgleda scena:



U tablici će biti zapisane sve karte od korisnika koji se ulogirao u sustav.

Kako filtrirati samo karte od korisnika koji se ulogirao u sustav?

Pošto smo napravili da svaka karta ima „Korisnika“ odnosno „Navijača“, možemo napraviti da se prilikom Logina u sustav spremi naš navijač u globalnu varijablu. Nakon toga idemo po cijeloj listi karata i filtriramo je tako da ostaju samo one kojima je „Navijač“ na karti isti „Navijaču“ koji se ulogirao u sustav.

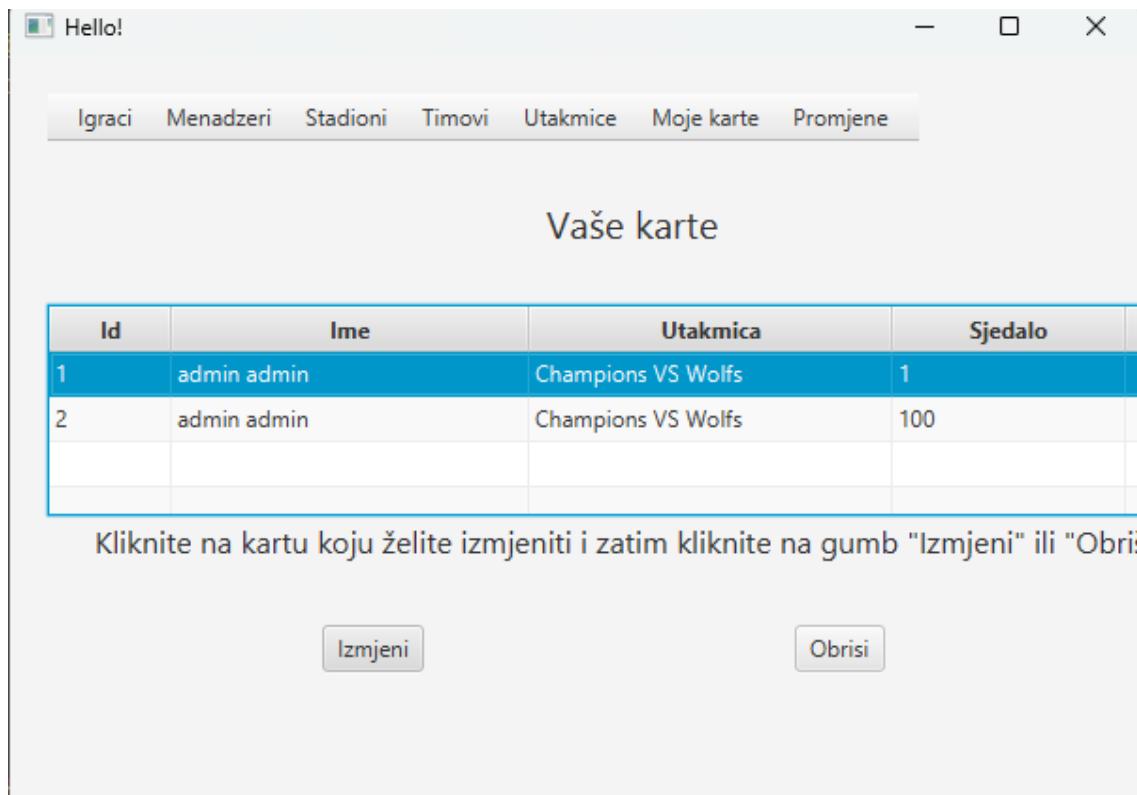


```
56 NepostojeciSjedalo
57 ZauzetoSjedalo
58 
59 @FXML
60 public void initialize(){
61     listaKarata = HelloController.listaKarata;
62 
63     List<Karta> filtriranaListaKarata = listaKarata.stream().filter(s->s.getNavijac().getUsername().equals(navijač))
64         .collect(Collectors.toList());
65 
66     idKarteTableColumn.setCellValueFactory(c-> new SimpleStringProperty(c.getValue().getIdKarte().toString());
67     imeTableColumn.setCellValueFactory(c-> new SimpleStringProperty( s: c.getValue().getNavijac().getUsername());
68     utakmicaTableColumn.setCellValueFactory(c-> new SimpleStringProperty( s: c.getValue().getUtakmica().getNaziv());
69     sjedaloTableColumn.setCellValueFactory(c-> new SimpleStringProperty(c.getValue().getOdarbanoSjedalo().toString());
70 
71     ObservableList<Karta> observableListKarata = FXCollections.observableList(filtriranaListaKarata);
72     kartaTableView.setItems(observableListKarata);
73 
74     upitnik.setVisible(false);
75     gumbDa.setVisible(false);
76     gumbNe.setVisible(false);
77     gumbDaUredi.setVisible(false);
```

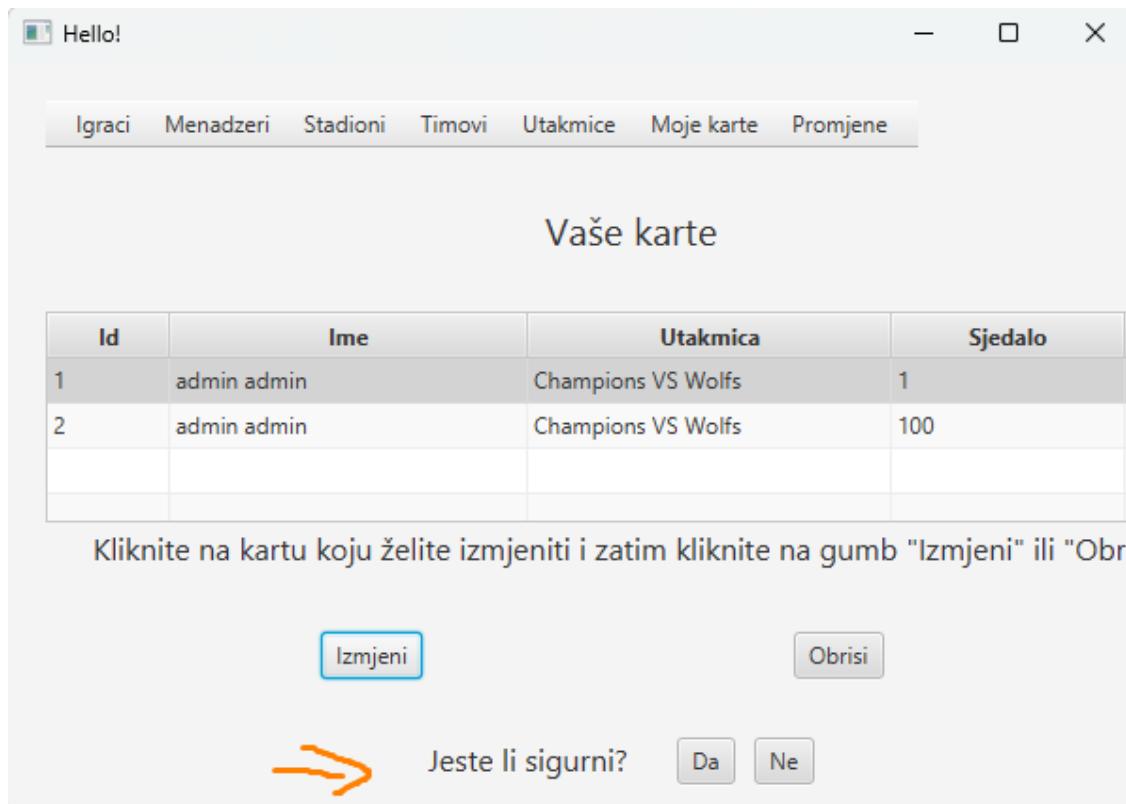
Dakle, idemo po cijeloj listi karata i ostavljamo samo one karte u kojima je username navijača jednak usernameu koji smo koristili prilikom Logina.

Na početku smo sve donje gume i text stavili na .setVisible(false) zato da se korisniku ne pokazuju odmah, nego tek nakon šta klikne na gumb „Obrisi“ ili „Izmjeni“.

Prije nego stisnemo na gumb „Izmjeni“ ili „Obris“



Nakon sta stisnemo:



Napravili smo metodu za gumb „Da“ i metodu za gumb „Ne“

Metoda „Ne“ će samo postaviti gume nazad na nevidljive.

Imamo dvije metode za gume „Da“. (Napravili smo jedan gumb „Da“ za kad korisnik stisne „Izmjeni“ i jedan „Da“ za kad korisnik klikne „Obrisi“), tako da palimo gumb ovisno koji nam treba.

Ako je korisnik Kliknuo „Obrisi“, pa „Da“

```
1 usage
@FXML
public void onDaButtonClick(){
    int indexKarte = kartaTableView.getSelectionModel().getFocusedIndex();
    listaKarata = HelloController.listaKarata;
    KupnjaKarataController.logger.info("Korisnik "+HelloController.ulogiraniNavijac.getUsername()+" obrasio je kartu: ID: "+HelloController.svePromjene.add("Korisnik "+HelloController.ulogiraniNavijac.getUsername()+" obrasio je kartu: ID: "+indexKarte));
    listaKarata.remove(indexKarte);
}
```

Iz liste karata smo pobrisali kartu ovisno o indexu koji je korisnik odabrao (ako je kliknuo na prvu kartu, pobrisala se karta s indexom 0, ako je odabrao drugu kartu, pobrisala se karta s indexom 1 ... itd)

Nakon toga smo cijelu listu karata serijalizirali (spremili u binarnu datoteku) , također smo napravili jednu datoteku di zapisujemo sve promjene (pošto je zadano da moramo imati scenu na kojoj se ispisuju sve promjene koje su se dogodile u aplikaciji)

```
01 try {
02     ObjectOutputStream out = new ObjectOutputStream(new FileOutputStream( name: "dat/serijalizraneKarte.dat"));
03     ObjectOutputStream pogreskeFile = new ObjectOutputStream(new FileOutputStream( name: "dat/serijaliziranePogreske.dat"));
04
05     out.writeObject(listaKarata);
06     pogreskeFile.writeObject(HelloController.svePromjene);
07
08     out.close();
09     pogreskeFile.close();
10 } catch (IOException e) {
11     throw new RuntimeException(e);
12 }
```

Dodali smo i Logger u koji zapisujemo sve promjene (pošto se traži u zadatku).

Napravili smo globalnu listu promjena List<String> svePromjene

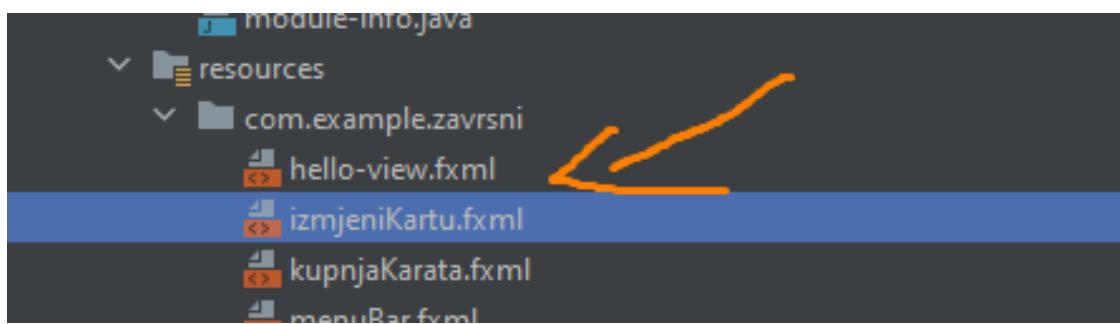
koja se nalazi u HelloController-u

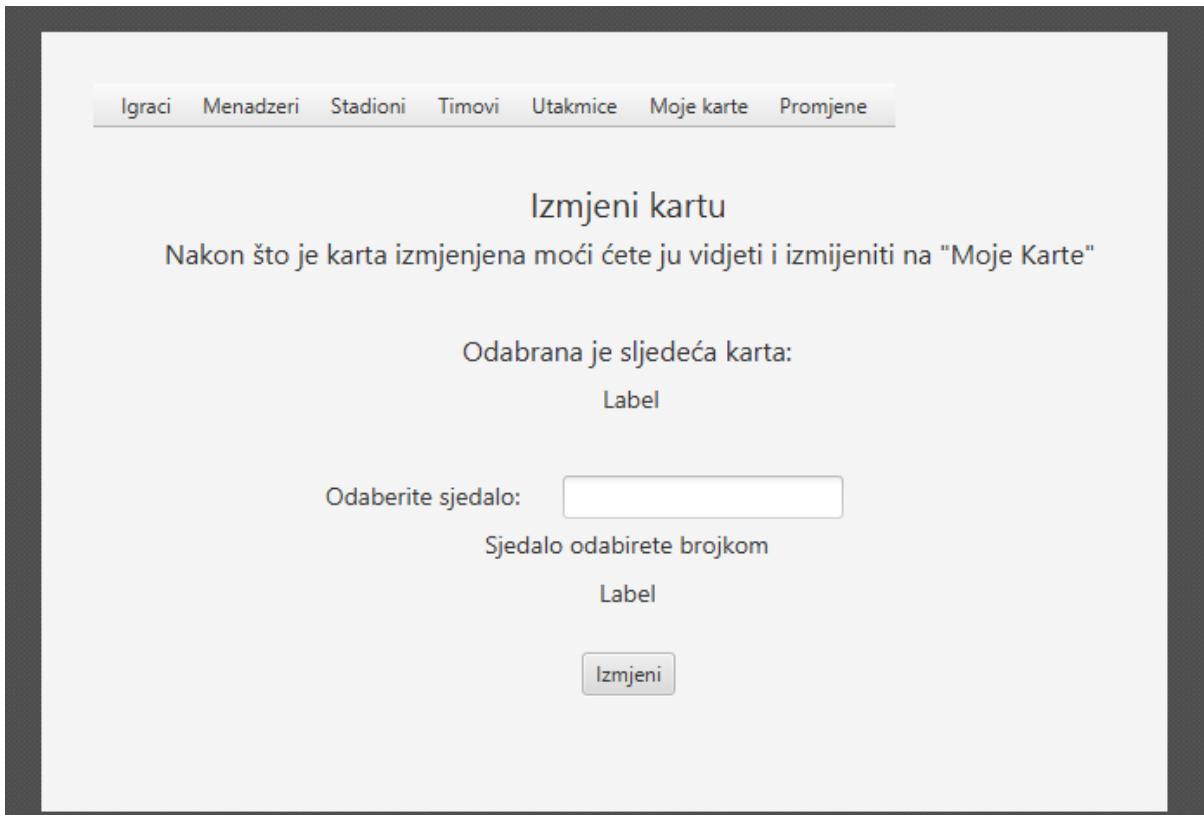
U tu listu stringova se zapisuju sve naše poruke „Korisnik je nešto pobrisao“, „Korisnik je nešto promjenio“ da bi sve te poruke na kraju mogli ispisati na ekran, ali do toga cu doci kasnije.

```
usage
FXML
public void onDaButtonClicked(){
    int indexKarte = kartaTableView.getSelectionModel().getFocusedIndex();
    listaKarata = HelloController.listaKarata;
    KupnjaKarataController.logger.info("Korisnik "+HelloController.ulogiraniNavijac.getUsername()+" obrisao je kartu: "+HelloController.svePromjene.add("Korisnik "+HelloController.ulogiraniNavijac.getUsername()+" obrisao je kartu: "+indexKarte));
}
```

Nakon što se pobrise karta, ta lista se ponovo spremi u binarnu datoteku da bi se zapamtilo brisanje i da te karte sljedeci put nebi bilo prilikom otvaranja datoteke.

Druga metoda za drugi gumb „Da“ radi jako slicno, samo sto ovoga puta korisnika šaljemo na novi ekran gdje će ponovo moći upisati svoje sjedalo.





Scena „Izmjeni kartu“ izgleda isto kao i „kupnjaKarata“, ali je razlika u tome što se karta neće nadodati na postojeće karte, nego će se našoj odabranoj karti samo promjeniti broj sjedala.

```
I usage
@FXML
public void onDaIzmjeniButtonClick() throws IOException {
    index = kartaTableView.getSelectionModel().getSelectedIndex();
    KartaZaIzmjenu = HelloController.listaKarata.get(index);

    FXMLLoader fxmlLoader = new FXMLLoader(HelloApplication.class.getResource( name: "izmjeniKartu.fxml"));
    Scene scene = new Scene(fxmlLoader.load(), v: 600, v1: 400);
    HelloApplication.getMainStage().setTitle("Hello!");
    HelloApplication.getMainStage().setScene(scene);
    HelloApplication.getMainStage().show();
}
```

Spremili smo kartu i njen index u globalne varijable da bi znali za koju kartu moramo promjeniti broj sjedala.

Nakon toga smo korisniku otvorili novi Screen „izmjeniKartu.fxml“

Nakon šta korisnik u sceni „Izmjeni kartu“ odabere sjedalo, sa metodom `.set()` ćemo našoj odabranoj karti promjeniti broj sjedala.

```
1. ZauzeetoSjedalo
util
  Baza
  Datoteke
HelloApplication
HelloController
IzmjeniKartuController
KupnjaKarataController
MenuBarController
pretragalgracaController
PretragaKarataController
PretragaMenadzeraController
PretragaStacionaController
PretragaTimovaController
PretragaUtakmicaController
PromjeneController

36
37 @FXML
38 public void onIzmjeniButtonClicked(){
39     String sjedalo = odabranosjedalo.getText();
40
41     List<Karta> listaKarata = HelloController.listaKarata;
42     Karta karta = PretragaKarataController.KartaZaIzmjenu;
43     int indexKarte = PretragaKarataController.index;
44
45     KupnjaKarataController.logger.info("Korisnik " +HelloController.ulogiraniNavijac.getUsername() + " napravio je
46
47     karta.setOdabranosjedalo(Integer.valueOf(sjedalo));
48     listaKarata.set(indexKarte,karta);
49
```

1. Dohvatili smo listu svih karata (da bi u njoj mogli promjeniti svoju kartu)
  2. Dohvatili smo kartu koju smo u prijašnjoj sceni stavili u globalnu varijablu
  3. Dohvatili smo index na kojem se ta karta nalazi koji smo u prijašnjoj sceni stavili u globalnu varijablu
  4. Svojoj karti smo sa setterom `.setOdabranoSjedalo()` postavili sjedalo na novo sjedalo koje smo unijeli u `TextField`-u

sa metodom `.set(index,karta)` smo promjenili kartu koja se nalazi na indexu „`index`“ sa kartom „`karta`“ kojoj smo promjenili sjedalo.

Nakon toga smo ponovo serijalizirali sve podatke

```
    /**
     * SERIJALIZACIJA */
    try {
        ObjectOutputStream out = new ObjectOutputStream(new FileOutputStream( name: "dat/serijalizraneKarte.dat"));
        ObjectOutputStream pogreskeFile = new ObjectOutputStream(new FileOutputStream( name: "dat/serijaliziranePogreske.dat"));

        out.writeObject(listaKarata);
        pogreskeFile.writeObject(HelloController.svePromjene);

        out.close();
        pogreskeFile.close();

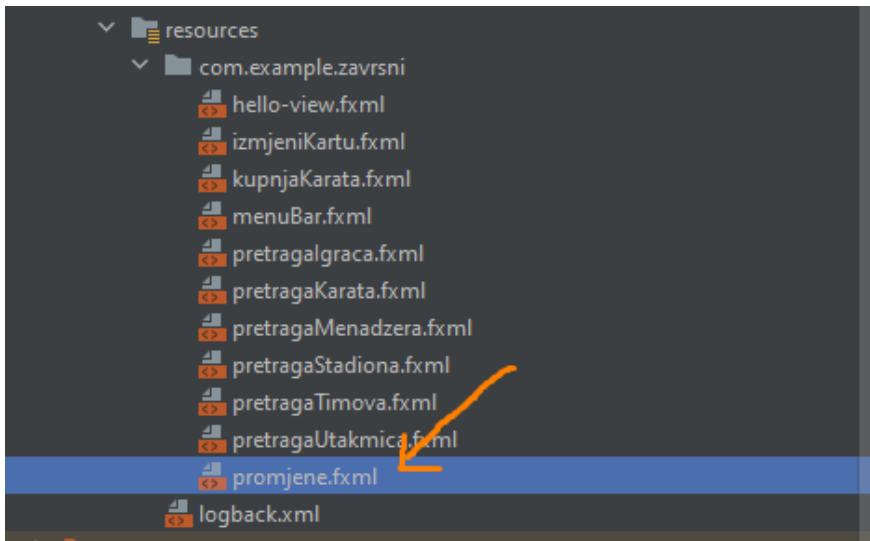
    } catch (IOException e) {
        throw new RuntimeException(e);
    }
}
```

I prikazali korisniku ALERT koji mu ispisuje da je doslo do promjene

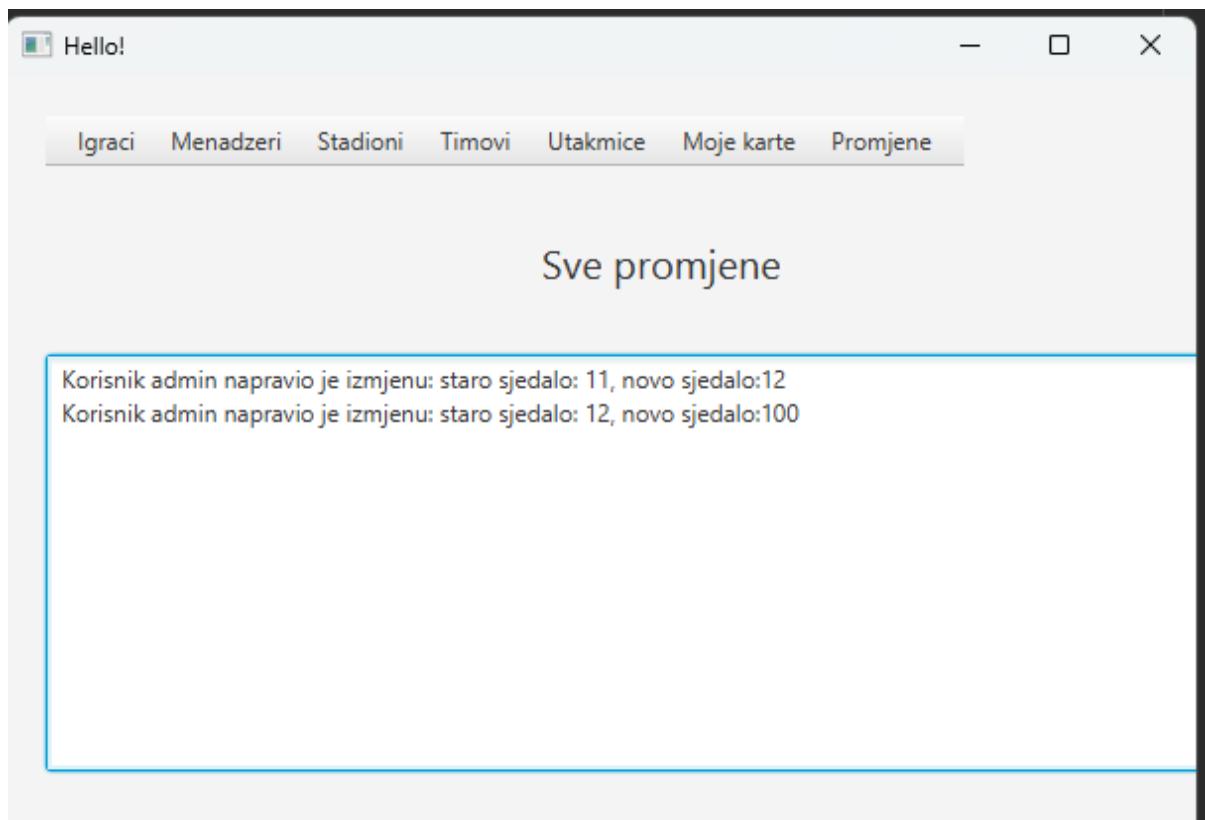
```
        Alert alert = new Alert(Alert.AlertType.INFORMATION);
        alert.setTitle("Uspješno ste izmjenili kartu!");
        alert.setHeaderText("Zahvaljujemo na kupovini");
        alert.setContentText("Karta se nalazi u \"Moje karte\" ");

        alert.showAndWait();
    }
}
```

Pošto se u zadatku traži da ispisemo sve promjene koje su se desile u aplikaciji napravit cemo novu scenu koja ce sluziti za ispisivanje svih promjena koje su se desile na bazi.



Ta scena ima samo 1 TextArea u kojem se zapisuju sve promjene

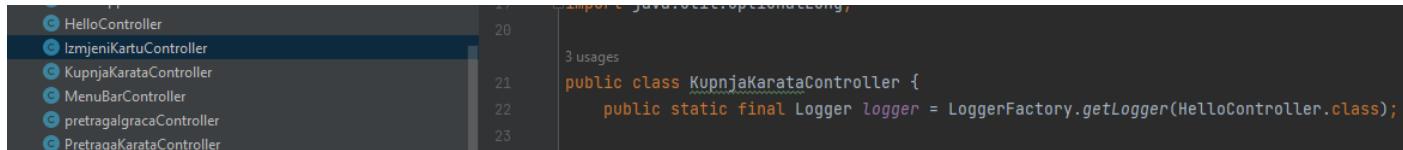


Sve promjene koje se desavaju na aplikaciji spremamo u listu „svePromjene“ koja se nalazi u HelloController-u.

I samo iz te liste ispisujemo sve podatke.

```
public class PromjeneController {  
    private TextArea promjene;  
    public void initialize(){  
        List<String> promjeneUAplicaciji = HelloController.svePromjene;  
        for(int i=0;i<promjeneUAplicaciji.size();i++){  
            promjene.appendText(promjeneUAplicaciji.get(i)+"\n");  
        }  
    }  
}
```

Logger smo inicijalizirali u klasi KupnjaKarataController, ali korisnimo ga gdje god nam treba jer smo ga napravili kao globalnu varijablu koju dohvacamo gdje god nam zatreba.

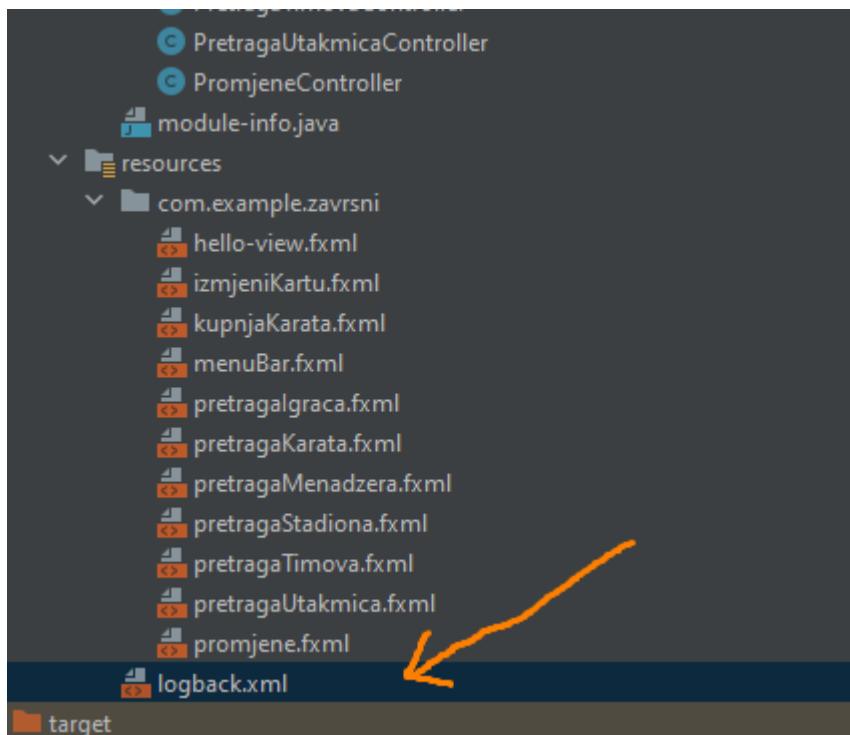


A screenshot of an IDE showing code completion. On the left, a list of controllers is shown: HelloController, IzmjeniKartuController, KupnjaKarataController, MenuBarController, pretragalgracaController, and PretragaKarataController. On the right, a code editor shows a snippet of Java code:

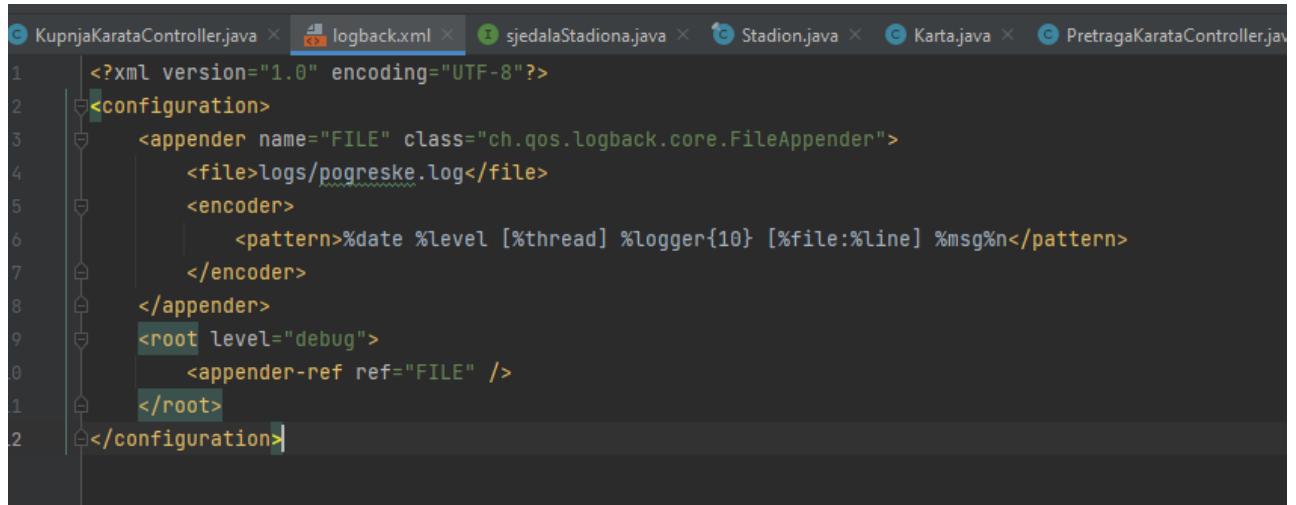
```
import java.util.logging.Logger;
import java.util.logging.Level;
...
20
21 public class KupnjaKarataController {
22     public static final Logger logger = LoggerFactory.getLogger(HelloController.class);
23 }
```

The word "KupnjaKarataController" is highlighted in blue, indicating it's a suggestion from the code completion feature.

Da bi nam logger zapisivao sve podatke u datoteku morali smo u „resources“ dodati logback.xml datoteku



Ta datoteka izgleda ovako:

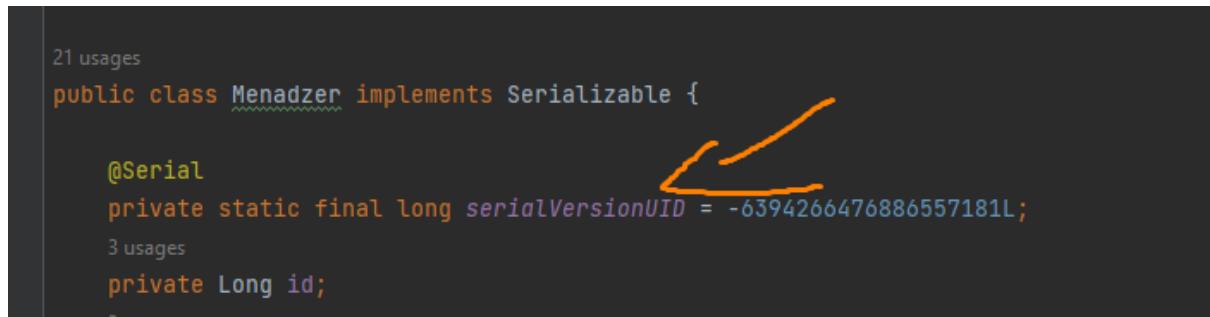


```
<?xml version="1.0" encoding="UTF-8"?>
<configuration>
    <appender name="FILE" class="ch.qos.logback.core.FileAppender">
        <file>logs/pogreske.log</file>
        <encoder>
            <pattern>%date %level [%thread] %logger{10} [%file:%line] %msg%n</pattern>
        </encoder>
    </appender>
    <root level="debug">
        <appender-ref ref="FILE" />
    </root>
</configuration>
```

Copy-paste je iz njegove prezentacije

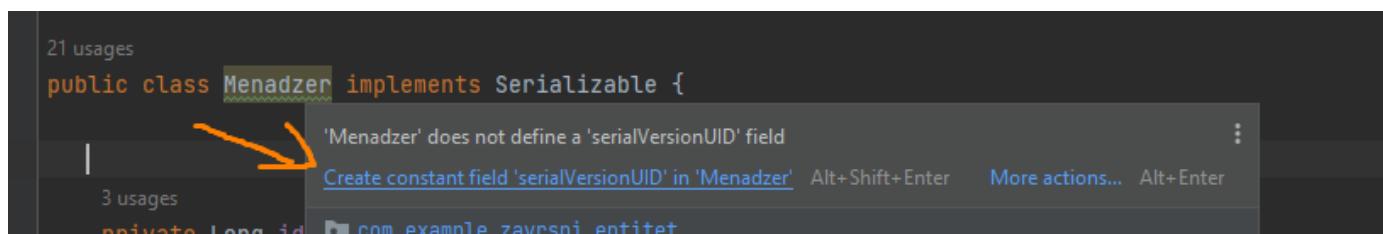
Pošto serijaliziramo podatke o svim kartama (a na kartama su podaci o svemu, utakmica, navijac...) moramo svim Entitetima dodati „implements Serializable“ da bi se podaci uspješno serijalizirali (zapisali u binarnu datoteku)

Pošto serijalizacija nekad zna ne raditi, moramo dodati



```
21 usages
public class Menadzer implements Serializable {
    @Serial
    private static final long serialVersionUID = -6394266476886557181L;
    3 usages
    private Long id;
}
```

Koji se automatski generira



```
21 usages
public class Menadzer implements Serializable {
    | 'Menadzer' does not define a 'serialVersionUID' field
    Create constant field 'serialVersionUID' in 'Menadzer' Alt+Shift+Enter More actions... Alt+Enter
    3 usages
    private Long id;
}
```

Posto mi se taj „Create constant field“ nije automatski pojavio morali smo ga enable-ati u postavkama. Kako se to radi nalazi se na sljedecem linku: <https://mkyong.com/java/how-to-generate-serialversionuid/>

Pošto piše da moramo implementirati BuilderPattern, napravit ću ga na klasi „Menadzer“, te cu umjesto konstruktora koristiti BuilderPattern.

The screenshot shows the IntelliJ IDEA interface. On the left, the project structure is displayed under the package `com.example.zavrsni`. A blue arrow points from the `Menadzer` class in the `entitet` package to the code editor on the right. The code editor contains the following Java code for the `BuilderPattern` class:

```
4 usages
public static class BuilderPattern{
    2 usages
    private Long id;
    2 usages
    private String ime;
    2 usages
    private String prezime;
    2 usages
    private String oib;
    2 usages
    private LocalDate datumRodjenja;

    public BuilderPattern (Long id){
        this.id= id;
    }

    public BuilderPattern imeMenadzera(String ime){
        this.ime = ime;
        return this;
    }

    public BuilderPattern prezimeMendazera(String prezime){
        this.prezime = prezime;
        return this;
    }

    public BuilderPattern oibMenadzera(String oib){
        this.oib = oib;
        return this;
    }

    public BuilderPattern datumRodjenjaMenadzera(LocalDate datumRodjenja)
        this.datumRodjenja = datumRodjenja;
        return this;
    }
}
```

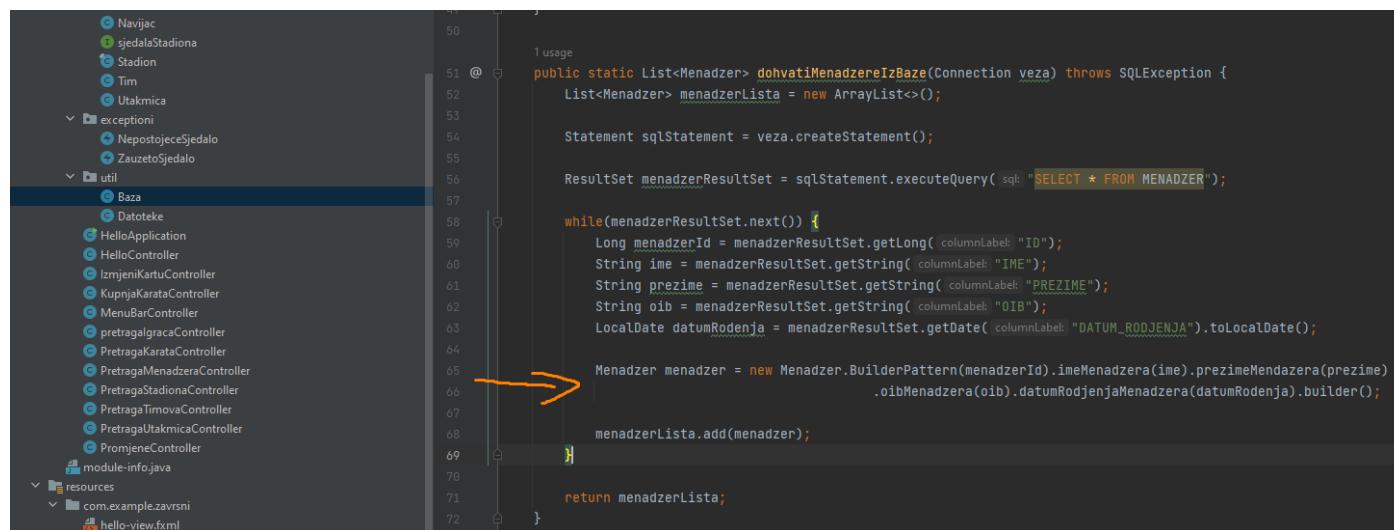
```

48     this.oib = oib;
49     return this;
50 }
51
52 public BuilderPattern datumRodjenjaMenadzera(LocalDate datumRodenja){
53     this.datumRodenja = datumRodenja;
54     return this;
55 }
56
57 public Menadzer builder(){
58     Menadzer menadzer= new Menadzer();
59
60     menadzer.id = this.id;
61     menadzer.ime = this.ime;
62     menadzer.prezime = this.prezime;
63     menadzer.oib = this.oib;
64     menadzer.datumRodjenja = this.datumRodenja;
65
66     return menadzer;
67 }
68
69
70 }
71

```

BuilderPattern koristimo ako ne zelimo unijeti sve podatke od jednog nego mozemo unositi samo odreden broj podataka.

Primjer koristenja:



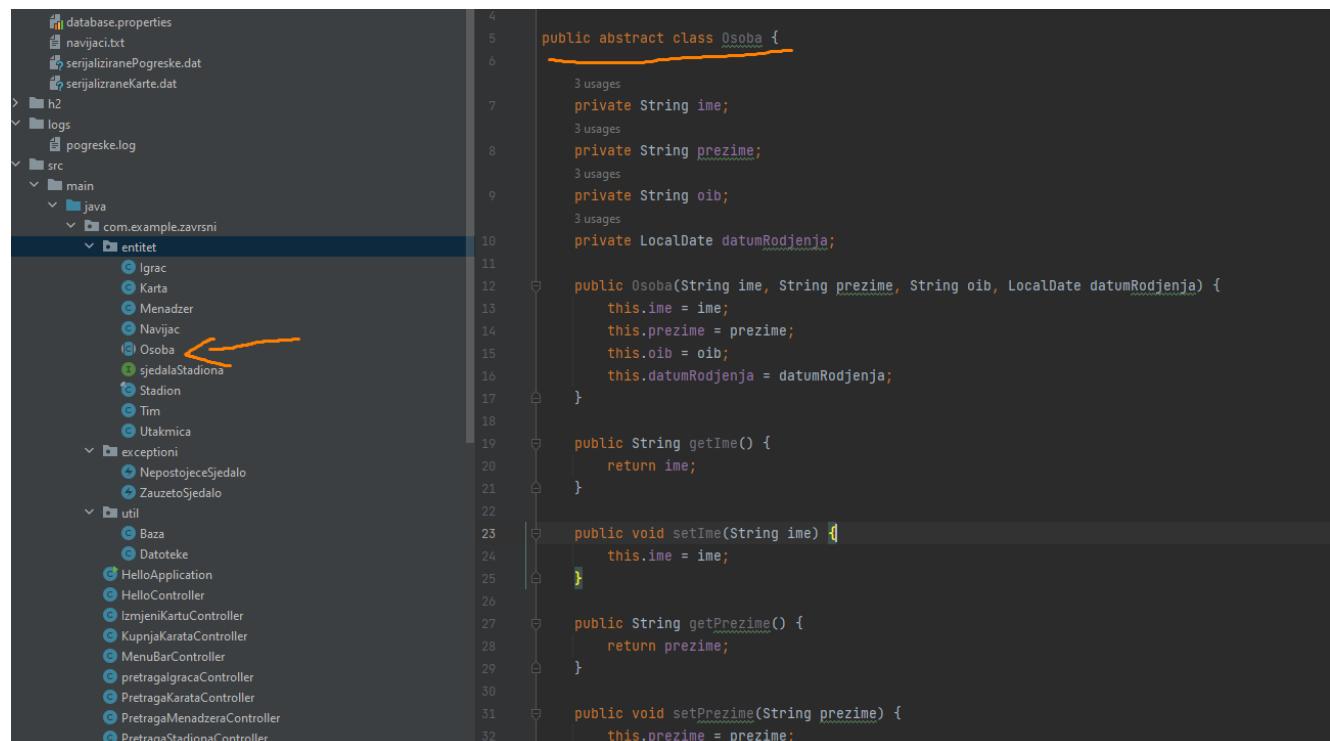
```

1 usage
2 @ ...
3 public static List<Menadzer> dohvatiMenadzereIzBaze(Connection vez) throws SQLException {
4     List<Menadzer> menadzerLista = new ArrayList<>();
5
6     Statement sqlStatement = vez.createStatement();
7
8     ResultSet menadzerResultSet = sqlStatement.executeQuery( sql: "SELECT * FROM MENADZER");
9
10    while(menadzerResultSet.next()) {
11        Long menadzerId = menadzerResultSet.getLong( columnLabel: "ID");
12        String ime = menadzerResultSet.getString( columnLabel: "IME");
13        String prezime = menadzerResultSet.getString( columnLabel: "PREZIME");
14        String oib = menadzerResultSet.getString( columnLabel: "OIB");
15        LocalDate datumRodjenja = menadzerResultSet.getDate( columnLabel: "DATUM_RODJENJA").toLocalDate();
16
17        Menadzer menadzer = new Menadzer.BuilderPattern(menadzerId).imeMenadzera(ime).prezimeMenadzera(prezime)
18                           .oibMenadzera(oib).datumRodjenjaMenadzera(datumRodjenja).builder();
19
20        menadzerLista.add(menadzer);
21    }
22
23    return menadzerLista;
24 }

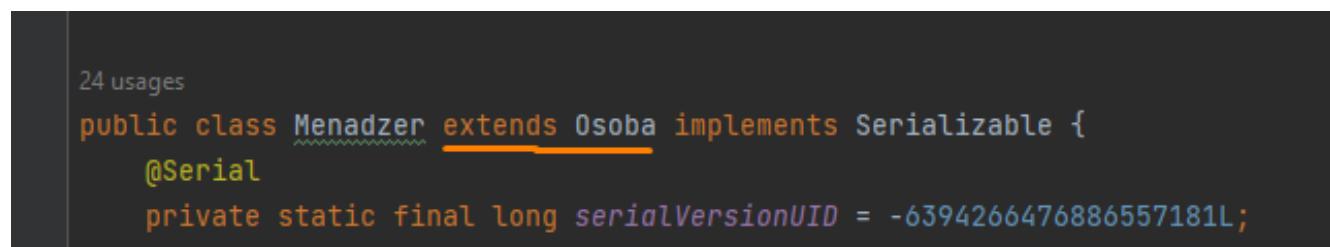
```

Piše u pravilima da napravimo i abstraktnu klasu. Abstraktna klasa je klasa koju druge klase mogu nasljeđivati. Npr ako imamo klasu Jabuka i Banana, možemo napraviti abstraktну klasu Voće koja će biti nadklasa Jabuci i Banani.

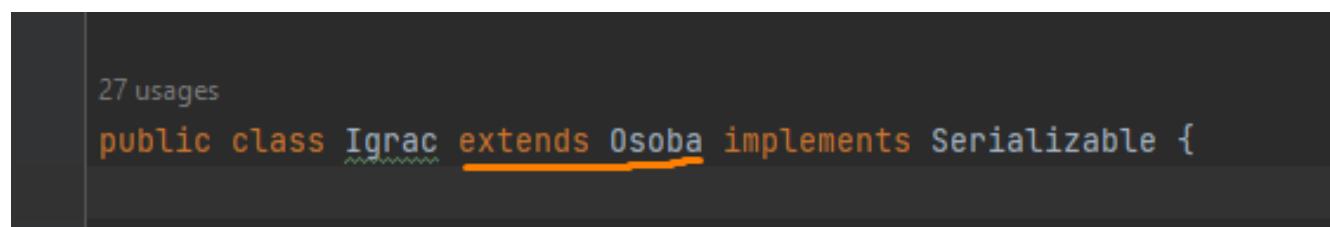
U našem slučaju napraviti cemo abstraktnu klasu Osoba i onda cemo napraviti da Igrac i Menadzer nasljeđuju klasu Osoba.



```
public abstract class Osoba {  
    private String ime;  
    private String prezime;  
    private String oib;  
    private LocalDate datumRodjenja;  
  
    public Osoba(String ime, String prezime, String oib, LocalDate datumRodjenja) {  
        this.ime = ime;  
        this.prezime = prezime;  
        this.oib = oib;  
        this.datumRodjenja = datumRodjenja;  
    }  
  
    public String getIme() {  
        return ime;  
    }  
  
    public void setIme(String ime) {  
        this.ime = ime;  
    }  
  
    public String getPrezime() {  
        return prezime;  
    }  
  
    public void setPrezime(String prezime) {  
        this.prezime = prezime;  
    }  
}
```



```
public class Menadzer extends Osoba implements Serializable {  
    @Serial  
    private static final long serialVersionUID = -6394266476886557181L;
```



```
public class Igrac extends Osoba implements Serializable {
```

```

private LocalDate datumRodjenja;
3 usages
private String drzava;

2 usages
public Igrac(Long id, String ime, String prezime, String oib, LocalDate datumRodjenja, String drzava) {
    super(ime, prezime, oib, datumRodjenja);
    this.id = id;
    this.ime = ime;
    this.prezime = prezime;
    this.oib = oib;
    this.datumRodjenja = datumRodjenja;
    this.drzava = drzava;
}

public Long getId() { return id; }

public void setId(Long id) { this.id = id; }

```

```

main
java
com.example.zavrsni
    entitet
        Igrac
        Karta
        Menadzer
        Navjac
        Osoba
        sjedalaStadiona
        Stadion
        Tim
        Utakmica

2 usages
public Menadzer(Long id, String ime, String prezime, String oib, LocalDate datumRodjenja) {
    super(ime, prezime, oib, datumRodjenja);
    this.id = id;
    this.ime = ime;
    this.prezime = prezime;
    this.oib = oib;
    this.datumRodjenja = datumRodjenja;
}

```

Moramo igracu i menadzeru dodati super() koji ce podatke o imenu, prezimenu, oibu i datumu rodjenja poslati u nadklasu „Osoba“.

Sada možemo napraviti listu Osoba u koju možemo spremati i Igrace i Menadzere. Ali to necemo napraviti jer nam ne treba za aplikaciju. Napravili smo to samo zato jer je zadano u zadatku.

Pošto moramo napraviti generičke klase (ali nemaju nam koristi u programu) napravit ćemo 2 genericke klase, ali necemo ih koristiti.

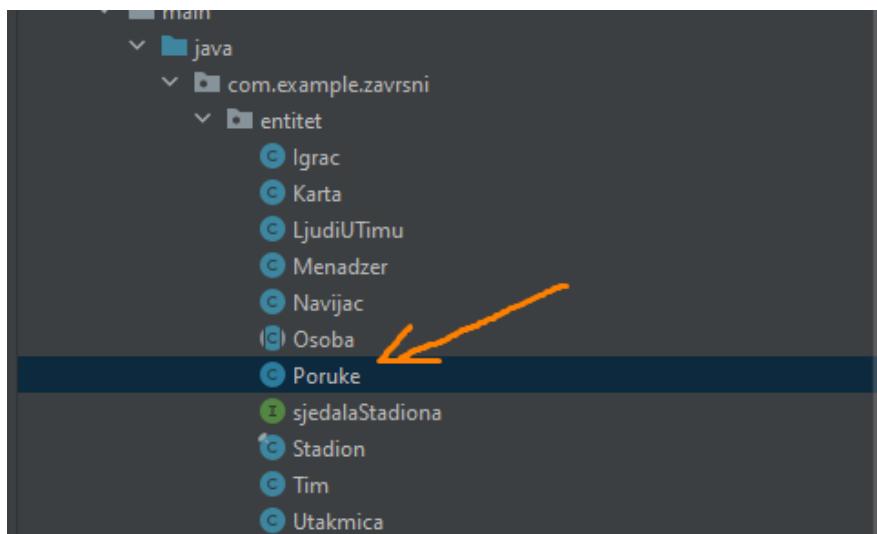
Prva:

The screenshot shows the file structure and code for a Java class named `LjudiUTimu`. The class extends `Osoba` and contains methods for adding and retrieving a list of people from a team. An orange arrow points to the `LjudiUTimu` class in the file tree.

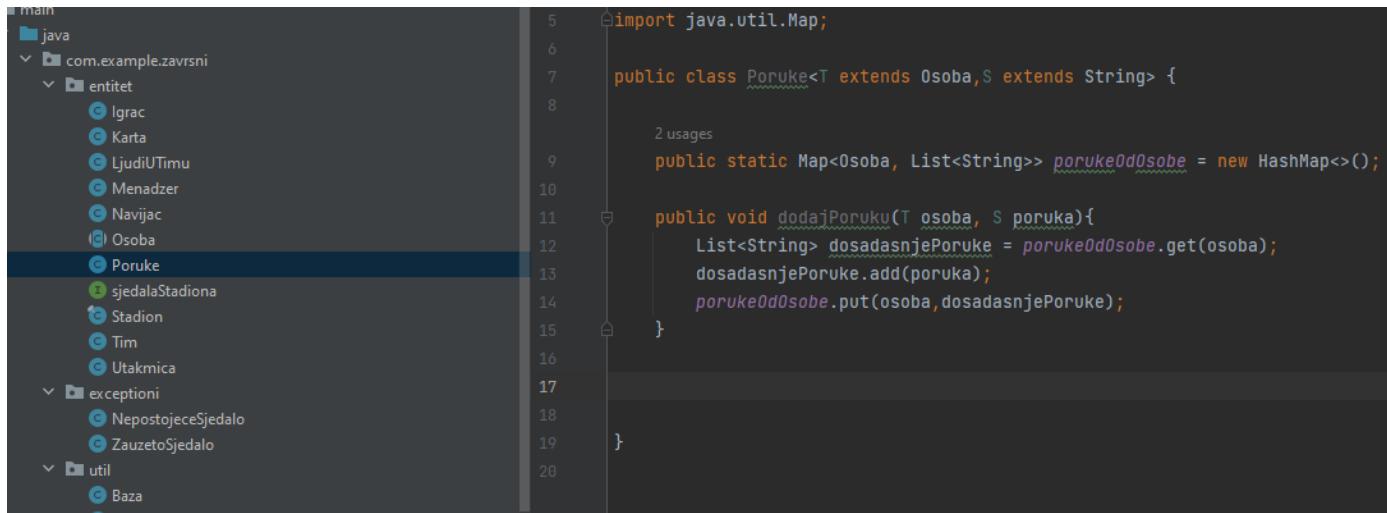
```
5 import java.util.List;
6
7 public class LjudiUTimu<T> extends Osoba{
8
9     2 usages
10    private List<T> ljudiKojiCineTim = new ArrayList<>();
11
12    public LjudiUTimu(String ime, String prezime, String oib, LocalDate datumRodjenja) {
13        super(ime, prezime, oib, datumRodjenja);
14    }
15
16    public LjudiUTimu() {
17    }
18
19    public void dodajOsobuNaTim(T objektOsoba){
20        ljudiKojiCineTim.add(objektOsoba);
21    }
22
23    public List<T> vratiListu(){
24        return ljudiKojiCineTim;
25    }
26
27 }
```

Klasa koja ima „`extends Osoba`“ znaci da će `T` koji saljemo moći biti samo objekt tipa „`Osoba`“ odnosno „`Menadzer`“ ili „`Igrac`“. Napravili smo klasu koja ima listu svih ljudi na timu, u tu listu možemo dodati igrača ili menadzera, te bi oni imali jednu zajednicku listu.

Druga genericka klasa će biti `<T, S>` ti će `T` biti `Osoba`, a `S` će biti `String`, odnosno „poruka“ od igrača / menadzera.



Pošto nam fali Mapa u kodu napraviti ćemo da se sve poruke od osoba spremaju u mapu.



```
import java.util.Map;

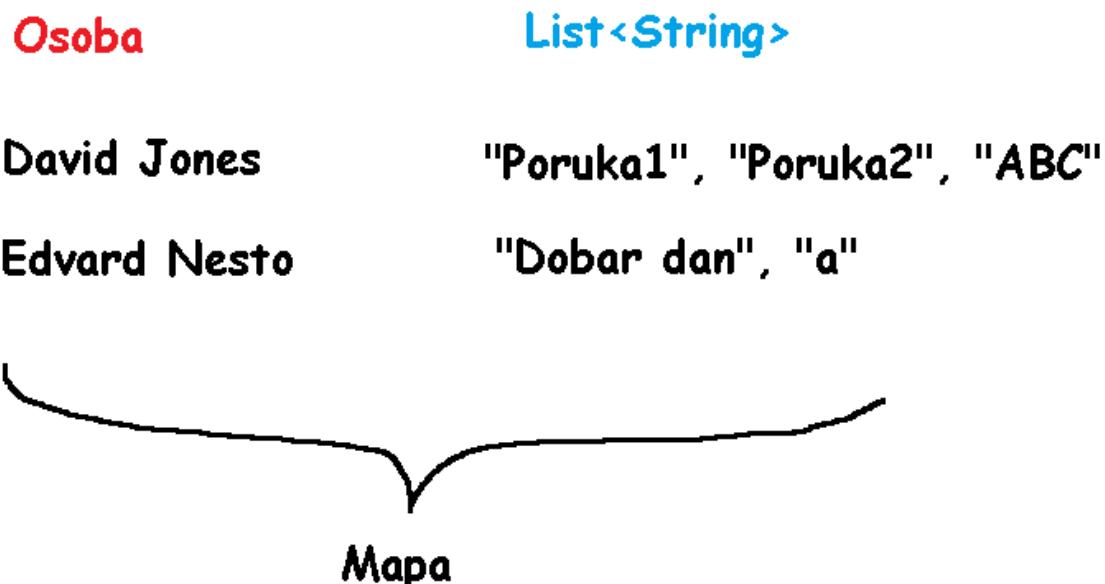
public class Poruke<T extends Osoba, S extends String> {

    public static Map<Osoba, List<String>> porukeOdOsobe = new HashMap<>();

    public void dodajPoruku(T osoba, S poruka){
        List<String> dosadasnjePoruke = porukeOdOsobe.get(osoba);
        dosadasnjePoruke.add(poruka);
        porukeOdOsobe.put(osoba, dosadasnjePoruke);
    }
}
```

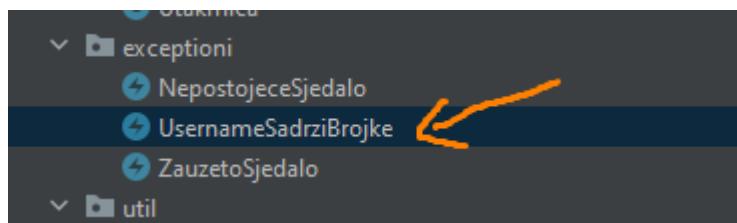
Mapa radi na način da ima jedan ključ i neki keySet (koji je u nasem slučaju lista poruka).

U metodi „dodajPoruku“ smo sa porukeOdOsobe.get(osoba) izvadili sve dosadašnje poruke od osobe, zatim smo na tu listu dodali novu poruku. Na kraju smo tu listu vratili nazad u mapu.



Pošto u zadatku piše da treba napraviti 4 Exceptiona, dodat cu jos 2 (i koristit cemo ih).

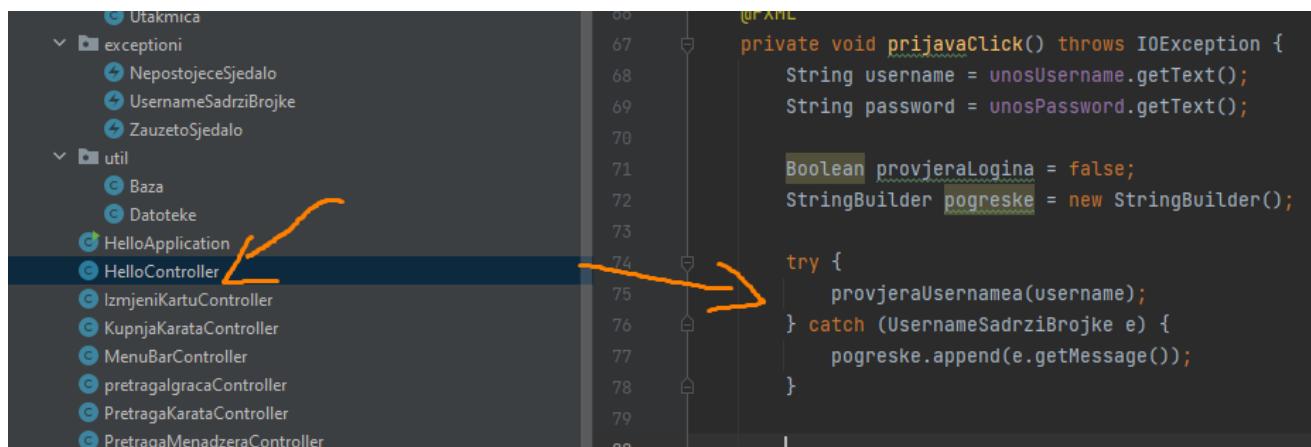
Prvi Exception ce se bacati ako korisnik unese brojku u Username prilikom ulogiravanja (napravit cemo da se ne smiju pisati brojke u username).



```
2
3     public class UsernameSadrziBrojke extends Exception{
4
5         public UsernameSadrziBrojke() {
6
7             super();
8         }
9
10        public UsernameSadrziBrojke(String message) {
11            super(message);
12        }
13
14        public UsernameSadrziBrojke(String message, Throwable cause) {
15            super(message, cause);
16        }
17
18        public UsernameSadrziBrojke(Throwable cause) {
19            super(cause);
20
21        public UsernameSadrziBrojke(String message, Throwable cause, boolean enableSuppression, boolean writableStackTrace) {
22            super(message, cause, enableSuppression, writableStackTrace);
23        }
24    }
```

Napravili smo da on extenda „Exception“ znaci da se mora odma „handle-ati“.

Implementacija tog exceptiona u kodu:



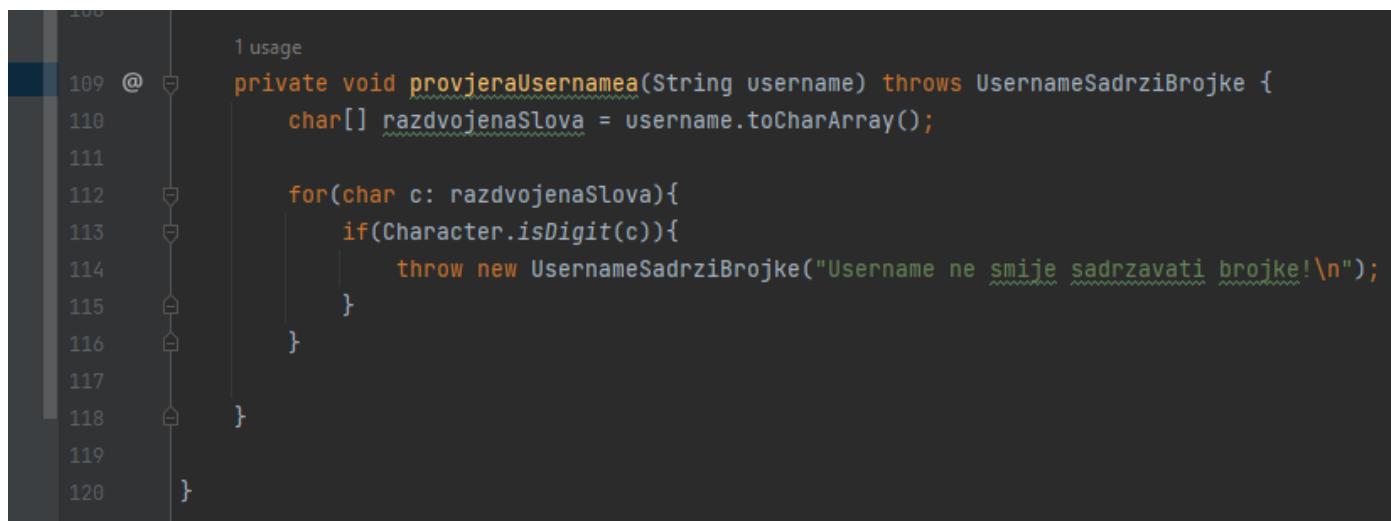
The screenshot shows a Java project structure on the left and a code editor on the right. The project structure includes packages like 'exceptioni' containing 'NepostojecijeSjedalo', 'UsernameSadrziBrojke', and 'ZauzetoSjedalo'; and 'util' containing 'Baza' and 'Datoteke'. The main application class 'HelloApplication' is selected. Below it, 'HelloController' is highlighted with a blue selection bar and has a yellow arrow pointing to it from the left. The code editor shows a method 'prijavaClick()' with annotations '@FXML' and 'throws IOException'. It initializes 'username' and 'password' from text fields, sets a boolean 'provjeraLogina' to false, and creates a 'StringBuilder' 'pogreske'. A try-catch block is present to handle 'UsernameSadrziBrojke' exceptions, appending their messages to 'pogreske'. Lines 74 and 75 are specifically highlighted with orange arrows.

```
private void prijavaClick() throws IOException {
    String username = unosUsername.getText();
    String password = unosPassword.getText();

    Boolean provjeraLogina = false;
    StringBuilder pogreske = new StringBuilder();

    try {
        provjeraUsernamea(username);
    } catch (UsernameSadrziBrojke e) {
        pogreske.append(e.getMessage());
    }
}
```

Pošto se login dešava na samom početku aplikacije (u HelloController-u) dodajemo metodu „provjeraUsernamea“ koja se poziva kada korisnik klikne „prijava“.



The screenshot shows the implementation of the 'provjeraUsernamea' method. It takes a string 'username' and throws an 'UsernameSadrziBrojke' exception if it finds any digits in the input. The method uses a character array to store the individual characters of the input string. It then iterates through each character using a for loop. If a digit is found, the exception is thrown with a message indicating that the username should not contain digits. The code is annotated with '1 usage' at the top.

```
1 usage
private void provjeraUsernamea(String username) throws UsernameSadrziBrojke {
    char[] razdvojenaSlova = username.toCharArray();

    for(char c: razdvojenaSlova){
        if(Character.isDigit(c)){
            throw new UsernameSadrziBrojke("Username ne smije sadrzavati brojke!\n");
        }
    }
}
```

U tu metodu se šalje username koji je korisnik unio i razdvajamo ga na char[] odnosno iz jednog stringa dobivamo niz slova. U for petlji gledamo slovo po slovo i ako je iti jedano od tih slova „Digit“ bacamo Exception „UsernameSadrziBrojke“ koji smo prethodno napravili sa porukom da username ne smije sadrzavati brojke.

Tu poruku ćemo spremiti u „StringBuilder“ koji će nam služiti za spremanje svih pogrešaka koje se mogu dogoditi korisniku.

Prva pogreška koja se može dogoditi je da unese broj u username, a druga pogreska ce se dodati ako je korisnik unio pogresan username + password.

Za drugu pogresku cemo napraviti novi Exception zato da bi ih imali 4.

Inače, kod za provjeru da li je korisnik unio dobar username i password vec postoji u kodu:

```
79
80
81
82 for(int i=0;i<listaNavijaca.size();i++){
83     if(listaNavijaca.get(i).getUsername().equals(username)){
84         if(listaNavijaca.get(i).getPassword().equals(password)){
85             provjeraLogina = true;
86             vlogiraniNavijac = listaNavijaca.get(i);
87         }
88     }
89 }
90 }
```

Ali pošto zadatak kaže da moramo imati 4 exceptiona cemo taj dio koda iskoristiti za novi Exception.

```
3     public class PogresanUserPass extends RuntimeException{
4
5         public PogresanUserPass() {
6     }
7
8         public PogresanUserPass(String message) {
9             super(message);
10        }
11
12        public PogresanUserPass(String message, Throwable cause) {
13            super(message, cause);
14        }
15
16        public PogresanUserPass(Throwable cause) {
17            super(cause);
18        }
19
20        public PogresanUserPass(String message, Throwable cause, boolean enableSuppression,
21            super(message, cause, enableSuppression, writableStackTrace);
22    }
23
24 }
```

## Implementacija exceptiona u kodu:

```
80     Boolean uspjesnostLogina = false;
81
82     try {
83         uspjesnostLogina = provjeraUSernameaIPassworda(username, password);
84     }catch (PogresanUserPass e){
85         pogreske.append(e.getMessage());
86     }
87
88     if(uspjesnostLogina == true){
89         System.out.println("Uspješan login!");
90     }
91 }
```

Ponovo u HelloControlleru, napravili smo metodu u koju saljemo username i password koji je korisnik unio.

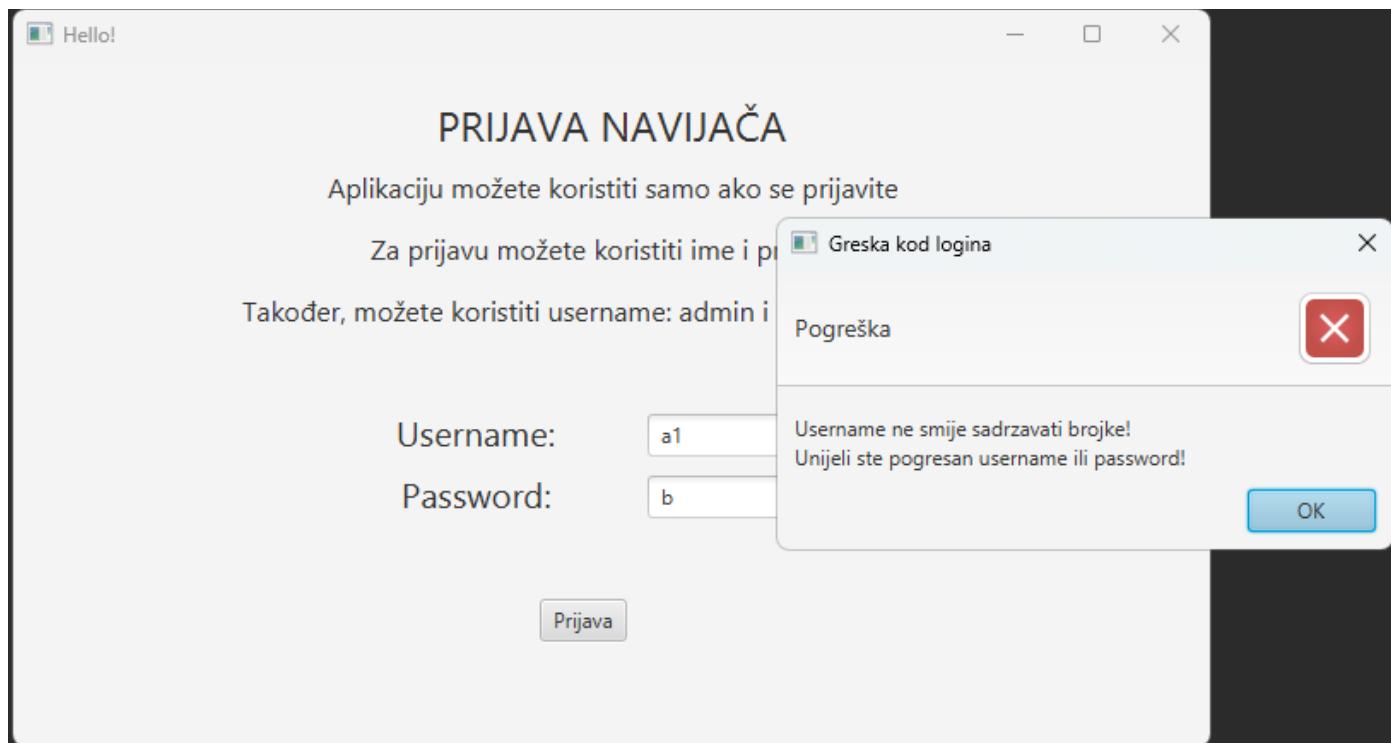
```
1 usage
110     private boolean provjeraUSernameaIPassworda(String username, String password) throws PogresanUserPass{
111         Boolean provjeraLogina = false;
112
113         for(int i=0;i<listaNavijaca.size();i++){
114             if(listaNavijaca.get(i).getUsername().equals(username)){
115                 if(listaNavijaca.get(i).getPassword().equals(password)){
116                     provjeraLogina = true;
117                     ulogiraniNavijac = listaNavijaca.get(i);
118                 }
119             }
120         }
121
122         if(provjeraLogina == false){
123             throw new PogresanUserPass("Unijeli ste pogresan username ili password!\n");
124         }
125
126         return provjeraLogina;
127     }
```

Ta metoda ide po listi svih navijača (korisnika) i gleda ako iti jedan ima isti username kao i username koji smo mi unijeli, ako je to istina onda gleda da li mu je password isti kao i password koji smo mi unijeli. Ako su i username i password dobri, onda metoda vraca „true“ i prosli smo login, ako provjeraLogina ostane „false“ baca se Exception „PogresanUserPass“ sa porukom „Unijeli ste pogresan user ili pass...“.

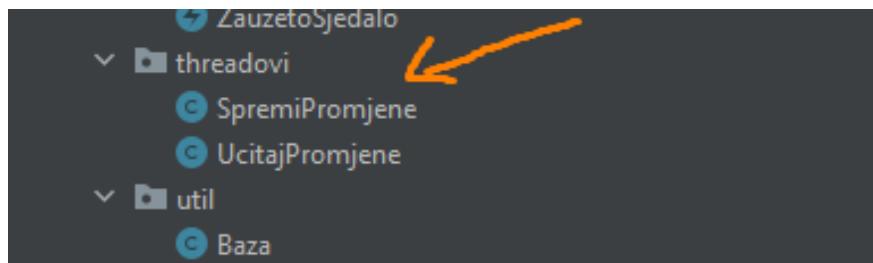
Na kraju, ako smo prosli obije provjere onda nastavljamo dalje sa kodom, odnosno prebacuje nas na novu scenu.

Ako nismo prosli provjeru, onda ispisujemo „ALERT“ korisniku koji ce nam dati do znanja o kojim se sve pogreskama radi. Ispisat ce se sve poruke koje smo spremili u StringBuilder „pogreske“.

```
89
90     if(uspjesnostLogina == true){
91         System.out.println("Uspješan login!");
92
93         FXMLLoader fxmlLoader = new FXMLLoader(HelloApplication.class.getResource( name: "pretragaIgraca.fxml"));
94         Scene scene = new Scene(fxmlLoader.load(), v: 600, vi: 400);
95         HelloApplication.getMainStage().setTitle("Hello!");
96         HelloApplication.getMainStage().setScene(scene);
97         HelloApplication.getMainStage().show();
98
99     }else{
100         Alert alert = new Alert(Alert.AlertType.ERROR);
101         alert.setTitle("Greska kod logina");
102         alert.setHeaderText("Pogreška");
103         alert.setContentText(pogreske.toString());
104
105         alert.showAndWait();
106     }
107
108 }
```



Pošto nam još u kodu fale Niti (Threadovi) dodat cemo 2 Threda.



Prvi Thread ce biti „SpremiPromjene“. Taj thread ce serijalizirati sve dosadasnje promjene u datoteku serijaliziranePogreske.dat

```
9 3 usages
10 public class SpremiPromjene implements Runnable{
11
12     @Override
13     public void run() {
14         try {
15             ObjectOutputStream pogreskeFile = new ObjectOutputStream(new FileOutputStream("dat/serijaliziranePogreske.dat"));
16
17             pogreskeFile.writeObject(HelloController.svePromjene);
18
19             pogreskeFile.close();
20         } catch (IOException e) {
21             throw new RuntimeException(e);
22         }
23     }
24 }
25 }
```

Razlog zasto je ova klasa Thread je zato što ima „implements Runnable“.

Pošto ima „implements Runnable“ mora overrideati metodu „run()“ koja se poziva cim executamo Thread.

Drugi Thread „UcitajPromjene“ ce na nasu listu promjena ucitati sve promjene koje su se procitale iz te binarne datoteke.

```
3 usages
7  public class UcitajPromjene implements Runnable{
8
9      @Override
10     public void run() {
11         PromjeneController.promjeneUApplikaciji = HelloController.svePromjene;
12     }
13
14 }
15
```

Poziv tih Threadova:

The screenshot shows the Java code for the application. On the left, the project structure is visible, showing packages like threadovi, util, and com.example.zavrrsni, along with various controller classes and FXML files. On the right, the code for the PromjeneController is shown. Two arrows point from the numbers 1 and 2 on the left to specific lines of code in the controller's initialize method. Arrow 1 points to line 23, where two threads are instantiated: 'SpremiPromjene prviThread = new SpremiPromjene();' and 'UcitajPromjene drugiThread = new UcitajPromjene();'. Arrow 2 points to line 26, where both threads are submitted to an ExecutorService: 'executorService.execute(prviThread);' and 'executorService.execute(drugiThread);'.

```
3 usages
13  public class PromjeneController {
14
15      2 usages
16      @FXML
17      private TextArea promjene;
18
19      3 usages
20      public static List<String> promjeneUApplikaciji = new ArrayList<>();
21
22      @FXML
23      public void initialize(){
24
25          SpremiPromjene prviThread = new SpremiPromjene();
26          UcitajPromjene drugiThread = new UcitajPromjene();
27
28          ExecutorService executorService = Executors.newCachedThreadPool();
29          executorService.execute(prviThread);
30          executorService.execute(drugiThread);
31
32          executorService.close();
33
34          for(int i=0;i<promjeneUApplikaciji.size();i++){
35              promjene.appendText(promjeneUApplikaciji.get(i)+"\n");
36          }
37      }
38 }
```

1. Napravimo oba Threada

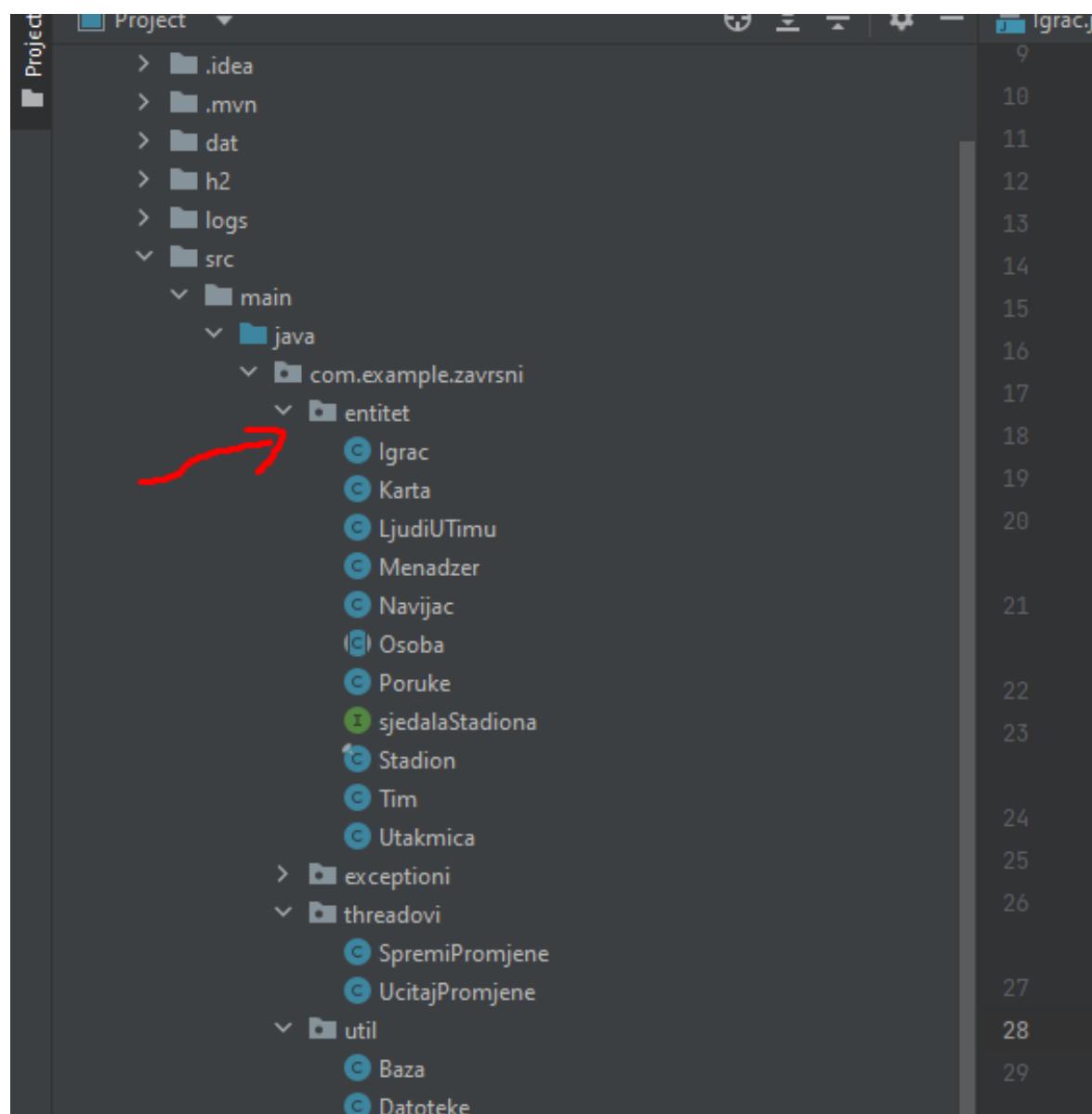
2. Pomocu ExecutorServica ih executamo (pustimo u isto vrijeme“ i s svaki put kada udemo na Screen sa promjenama ce se ti threadovi pustati istovremeno.

1. Implementiranje klase koje utjelovljuju entitete korištene u projektnom zadatku. Svaka klasa mora biti smještena u paket s klasama koje imaju zajednička svojstva (npr. entiteti moraju biti u jednom paketu, a glavna klasa za pokretanje aplikacije u drugom paketu).

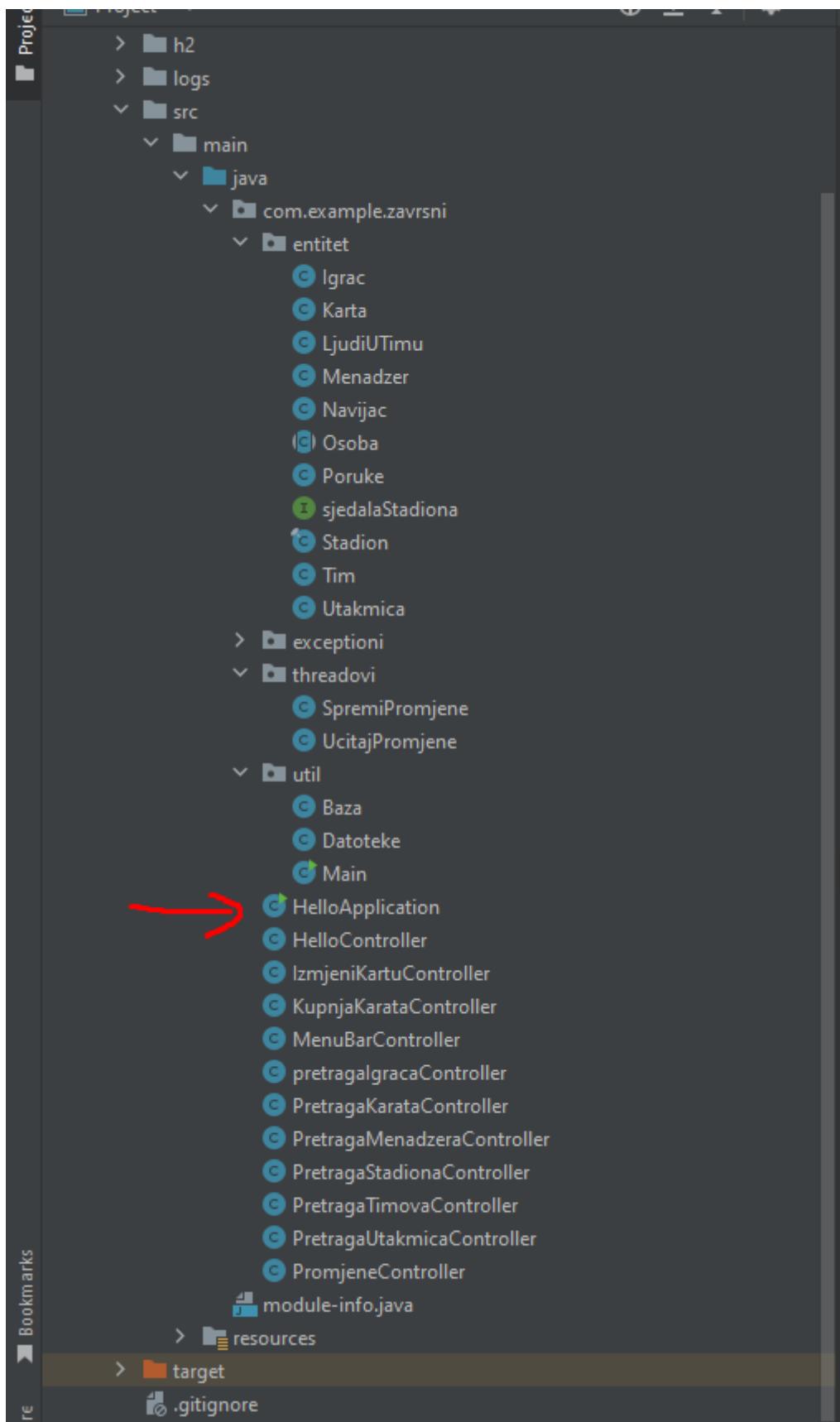
Klase koje utjelovljuju entitete su klase koje sam radio na samom početku: Igrac, Menadzer, Navijac, Stadion, Tim, Utakmica.

Kasnije sam dodao i „Karta“ da se zna koji navijac je kupio kartu.

Nalaze se u paketu „entitet“



Klase za pokretanje se naziva „HelloApplication“. Nalazi se ovdje:

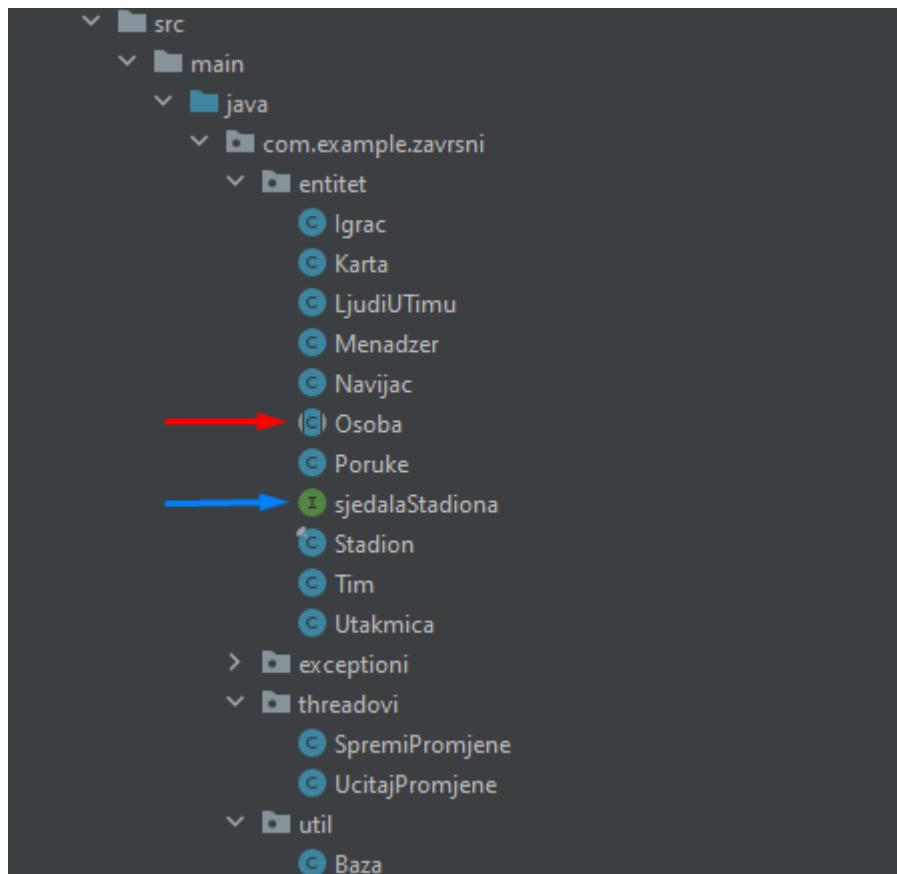


2. Korištenje apstraktnih klasa, sučelja, zapisa, zapečaćenih sučelja te „builder pattern“ oblikovnog obrasca kako bi se iskoristile sve objektno orijentirane paradigme programskog jezika Java.

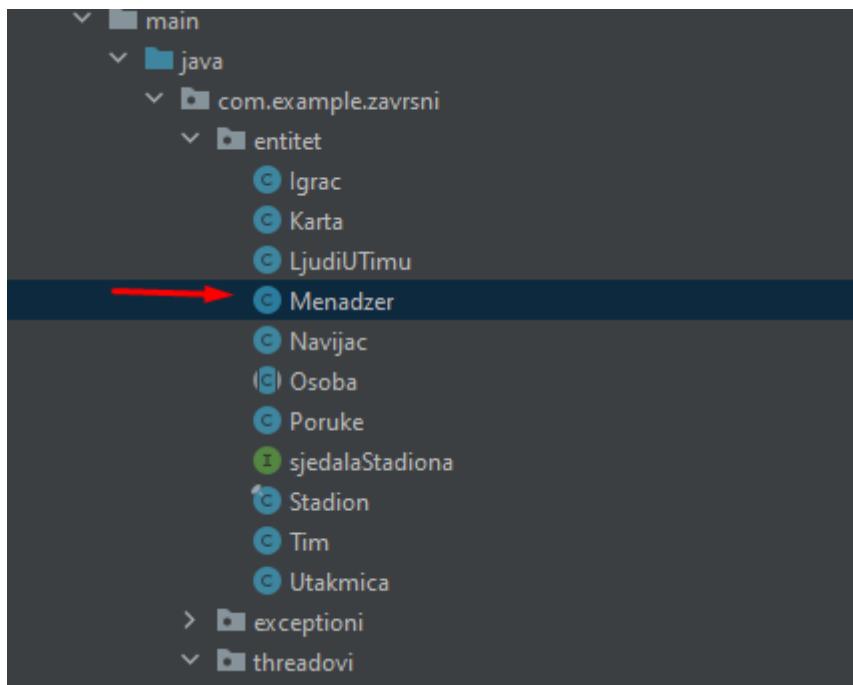
## Apskraktne klase: Osoba

Sučelje: sjedalaStadiona (zapečaćeno)

Također u „entitet“ package-u.

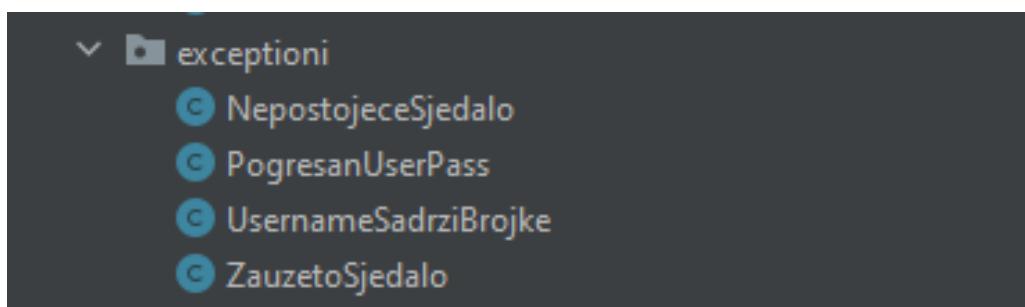


BuilderPattern se nalazi u klasi Menadzer. Dodatno je opisan na strani 50.



- |    |   |
|----|---|
| 3. | Hvatanje i bacanje iznimaka na svim mjestima u programu gdje se mogu dogoditi. Svaka iznimka se mora logirati korištenjem Logback biblioteke. Osim toga je potrebno kreirati barem dvije označene i dvije neoznačene iznimke te ih bacati i hvatati u programskom kodu aplikacije te logirati korištenjem Logback biblioteke. Klase iznimaka moraju biti smještene u zaseban paket. |
|----|---|

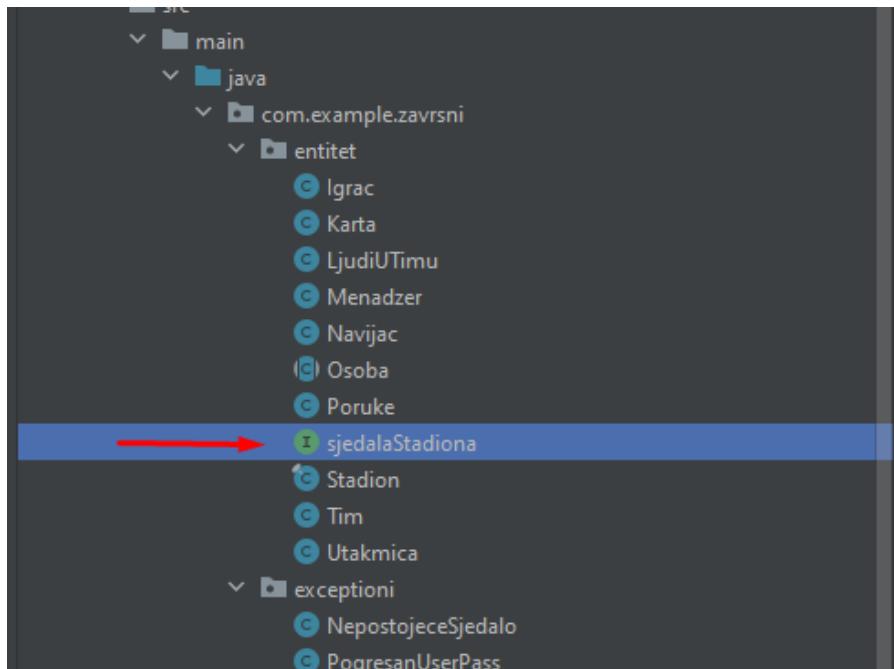
Iznimke se nalaze u package-u exceptioni.



Označeni exceptioni: NepostojeceSjedalo, UsernameSadrziBrojke

Neoznačeni exceptioni: PogresanUserPass, ZauzetoSjedalo

Exceptioni koje smo napravili za provjeru je li korisnik odabrao nepostojeće sjedalo ili sjedalo koje je zauzeto smo implementirali u interface-u (sučelju) „sjedalaStadiona“



```
1 package com.example.zavrnsni.entitet;
2
3 import ...
4
5
6 public sealed interface sjedalaStadiona permits Stadion{
7
8     void provjeraSjedala(String sjedalo) throws NepostojeceSjedalo;
9
10    void zauzetostSjedala(String sjedalo) throws ZauzetoSjedalo;
11 }
```

Ove metode nemaju tijelo jer se nalaze u interface-u. To znači da će ih mogu implementirati klase, a zatim će ti klase odrediti što će taj interface točno raditi. U našem slučaju, ovaj interface može implementirati samo klasa „Stadion“ zbog „permits Stadion“. Dodatno je objašnjeno na stranama 34-36 (na dnu stranice 34 je početak).

Stadion implementira te dvije metode ovako:

The screenshot shows a Java code editor with a file tree on the left and code on the right. The file tree includes packages like entitet, exceptioni, threadovi, and util, and classes like Igrac, Karta, LjudiUTim, Menadzer, Navijac, Osoba, Poruke, sjedalaStadiona, Stadion, Tim, Utakmica, and several HelloController subclasses. The code editor displays two methods in the Stadion class:

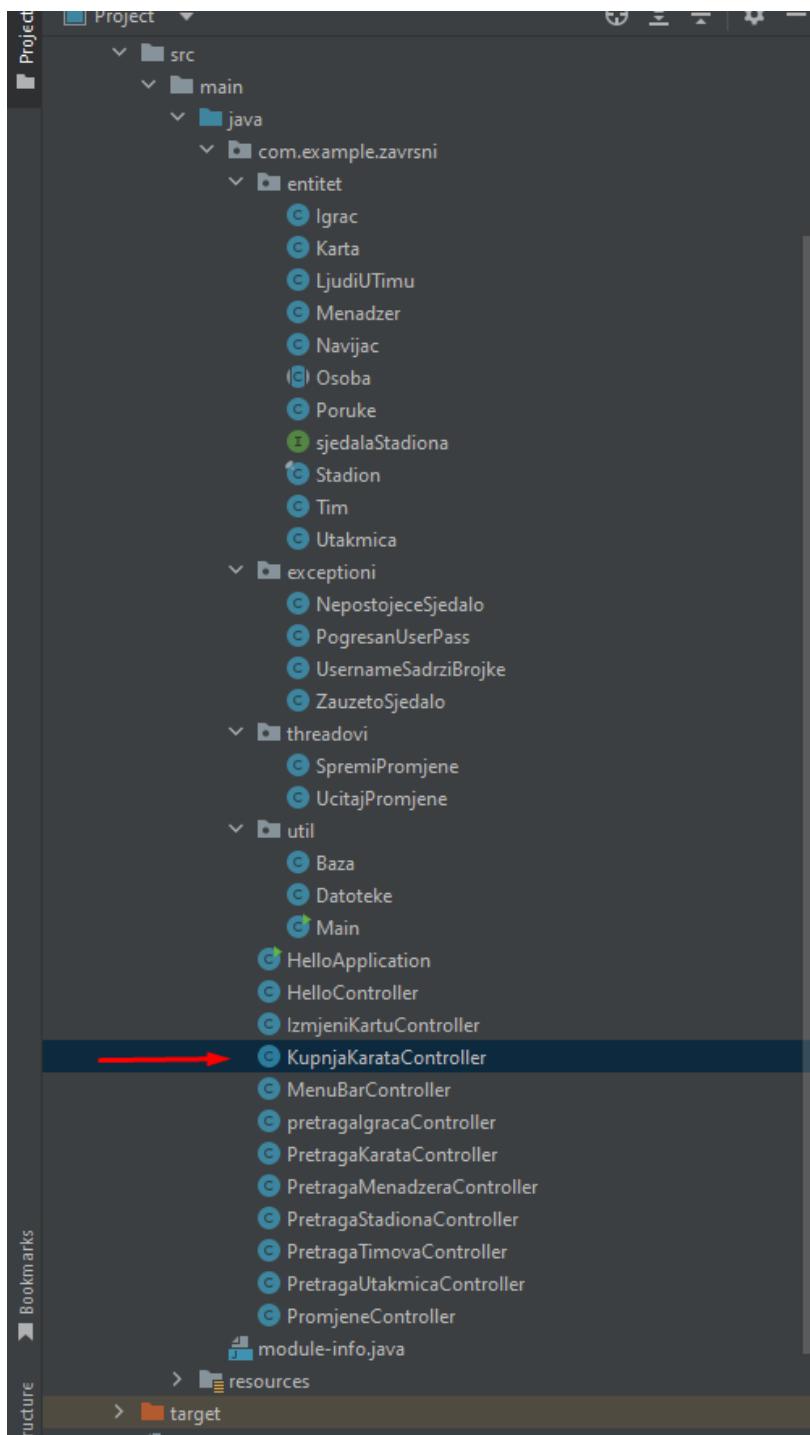
```
1 usage
@Override
public void provjeraSjedala(String sjedalo) throws NepostojeceSjedalo {
    Utakmica utakmica = PretragaUtakmicaController.odabranutaakmica;
    if(Integer.parseInt(sjedalo)<1 || Integer.parseInt(sjedalo)>utakmica.getStadion().getBrojSjedala()) {
        throw new NepostojeceSjedalo("Odabrali ste sjedalo koje ne postoji!");
    }
}

1 usage
@Override
public void zauzetostSjedala(String sjedalo) throws ZauzetoSjedalo {
    List<Karta> listaKarata = HelloController.listaKarata;
    for(int i=0;i<listaKarata.size();i++){
        if(listaKarata.get(i).getOdabranoSjedalo().equals(Integer.valueOf(sjedalo))) {
            throw new ZauzetoSjedalo("Sjedalo koje ste odabrali je zauzeto!");
        }
    }
}
```

Dodatno su objašnjene na strani 36.

Kada god korisnik odabere sjedalo i kupi kartu, taj podatak sa njegovim sjedalom se šalje u te dvije metode da bi se napravila provjera. U slučaju da je sjedalo zauzeto bacit će se exception „ZauzetoSjedalo“, a u slučaju da je odabrao nepostojeće sjedalo bacit će se „NepostojeceSjedalo“.

Te provjere se rade u klasi „KupnjaKarataController“



```
1 usage
@FXML
public void onPotvrдиButtonClick() {

    String sjedalo = odabranoSjedalo.getText();

    StringBuilder pogreske = new StringBuilder();
    Utakmica utakmica = PretragaUtakmicaController.odabranutakmica;

    try {
        utakmica.getStadion().provjeraSjedala(sjedalo);
        utakmica.getStadion().zauzetostSjedala(sjedalo);
    } catch (NepostojeceSjedalo | ZauzetoSjedalo e) {
        pogreske.append(e.getMessage());
        Logger.error("Korisnik "+HelloController.ulogiraniNavijac.getUsername()+" napravio je pogrešku: "+e.getMessage()+"\n");
        HelloController.svePromjene.add("Korisnik "+HelloController.ulogiraniNavijac.getUsername()+" napravio je pogrešku: "+e.getMessage());
    }
}
```

Pozivaju se te dvije metode koje rade provjeru i oko njih se stavlja „try – catch“.

U slučaju da nekad od metoda baci Exception, uhvatiti će se pomocu naredbe „catch“.

Nakon hvatanja exceptiona zapisujemo ga uz pomoć loggера (jer je u zadatku zadano da se sve mora zapisati u logger).

Također se nakon toga sve zapisuje u listu koja se zove „svePromjene“. To radimo zbog zadatka broj 9. (doći ćemo do njega kasnije).

4. Korištenje zbirki iz tipa lista, setova i mapa, uz korištenje lambda izraza za filtriranje i sortiranje svih entiteta u aplikaciji.

Liste koristimo za apsolutno sve entitete koje spremamo. Sve glavne liste se nalaze u klasi „HelloController“.

The screenshot shows the file structure on the left and the code editor on the right. The code in the editor is as follows:

```
46
47     public void initialize() {
48         try(Connection veza = Baza.connectToDatabase()) {
49
50             System.out.println("Connected to database!");
51
52             listaIgraca = Baza.dohvatiIgraceIzBaze(veza);
53             listaMenadzera = Baza.dohvatiMenadzereIzBaze(veza);
54             listaStadiona = Baza.dohvatiStadioneIzBaze(veza);
55             listaTimova = Baza.dohvatiTimoveIzBaze(veza);
56             listaUtakmica = Baza.dohvatiUtakmiceIzBaze(veza);
57
58             listaNavijaca = Datoteke.ucitajNavijace();
59             listaKarata = Datoteke.ucitajKarte();
60
61             svePromjene = Datoteke.ucitajPromjene();
62
63         } catch (SQLException | IOException e) {
64             e.printStackTrace();
65         }
66     }
```

A red arrow points from the word "HelloController" in the file tree to the corresponding line in the code editor where it is used.

Set koristimo u klasi „Tim“. U klasi „Tim“ spremamo Set<Igrac> (set igrača).

The screenshot shows the file structure on the left and the code editor on the right. The code in the editor is as follows:

```
11     @Serial
12     private static final long serialVersionUID = 801126495991438049L;
13
14     private Long id;
15
16     private String imeTimu;
17
18     private Set<Igrac> listaIgraca; ←
19     private Menadzer menadzer;
```

A red arrow points from the word "Tim" in the file tree to the corresponding line in the code editor where it is used.

Zove se listalgrača jer je to ime ostalo od prije, ali to je manje bitno, bitno da je Set.

Mapu koristimo klasi „Poruke“.

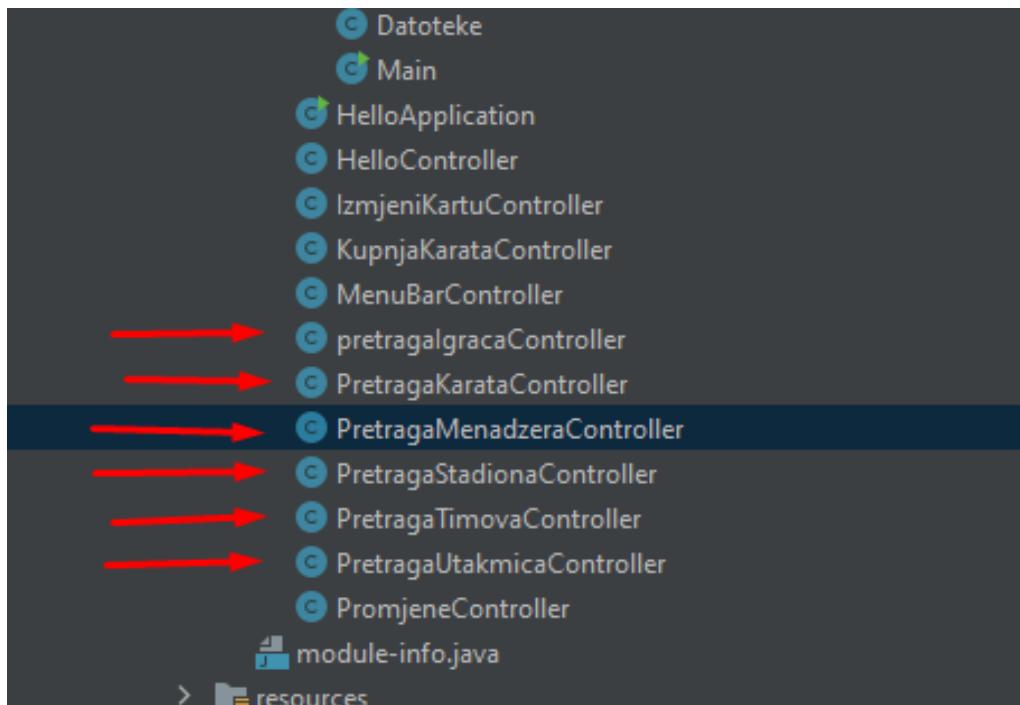
The screenshot shows a Java code editor with the following code:

```
5
6 public class Poruke<T extends Osoba, S extends String> {
7
8     2 usages
9     public static Map<Osoba, List<String>> porukeOdOsobe = new HashMap<>();
10
11    public void dodajPoruku(T osoba, S poruka) {
12        List<String> dosadasnjePoruke = porukeOdOsobe.get(osoba);
13        dosadasnjePoruke.add(poruka);
14        porukeOdOsobe.put(osoba, dosadasnjePoruke);
15    }
16
17
18 }
```

The code defines a generic class `Poruke` with two type parameters: `T` (extending `Osoba`) and `S` (extending `String`). It contains a static map `porukeOdOsobe` of type `Map<Osoba, List<String>>`. A method `dodajPoruku` adds a string `poruka` to the list of messages for a given person `osoba`, and updates the map.

Dodatno objašnjena na strani 55.

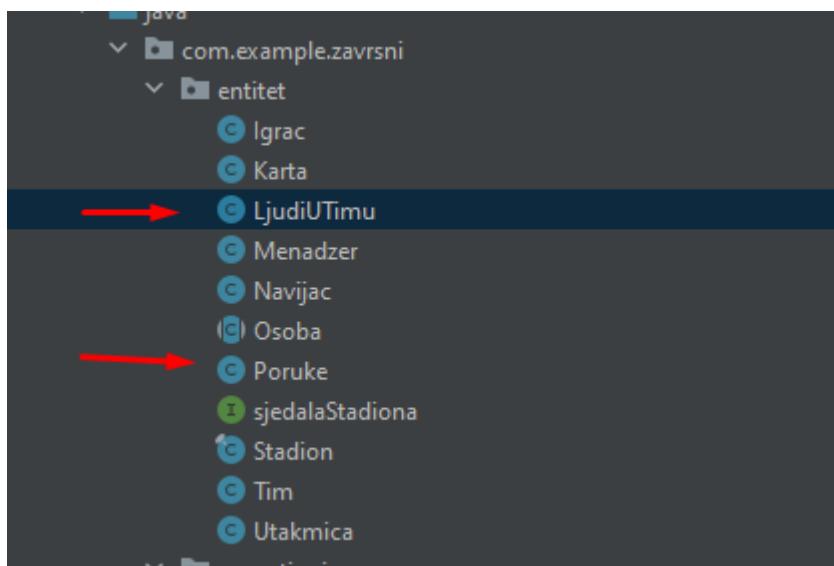
„lambda izrazi za filtriranje svih entiteta u aplikaciji“ se nalazi u svim Klasama koje imaju ime „Pretraga“ u sebi.



```
71  @FXML
72  public void onPretraziButtonClick(){
73      String enteredIme= imeMenadzera.getText();
74      String enteredPrezime = prezimeMenadzera.getText();
75      String enteredOib= oibMenadzera.getText();
76      LocalDate enteredDatum = datumRodenjaMenadzera.getValue();
77
78      List<Menadzera> filtriranaListaMenadzera = new ArrayList<>(ListaMenadzera);
79
80      if(Optional.of(enteredIme).isEmpty() == false){
81          filtriranaListaMenadzera = filtriranaListaMenadzera.stream().filter(s->s.getIme().toLowerCase().contains(enteredIme.toLowerCase())).collect(Collectors.toList());
82      }
83
84      if(Optional.of(enteredPrezime).isEmpty() == false){
85          filtriranaListaMenadzera = filtriranaListaMenadzera.stream().filter(s->s.getPrezime().toLowerCase().contains(enteredPrezime.toLowerCase())).collect(Collectors.toList());
86      }
87
88      if(Optional.of(enteredOib).isEmpty() == false){
89          filtriranaListaMenadzera = filtriranaListaMenadzera.stream().filter(s->s.getOib().toLowerCase().contains(enteredOib.toLowerCase())).collect(Collectors.toList());
90      }
91
92      if(Optional.ofNullable(enteredDatum).isPresent()){
93
94          filtriranaListaMenadzera = filtriranaListaMenadzera.stream().filter(s->{
95              DateTimeFormatter formatter = DateTimeFormatter.ofPattern("yyyy-dd-MM");
96              String formatedString = s.getDatumRodenja().format(formatter);
97
98              return formatedString.equals(enteredDatum.toString());
99          }).collect(Collectors.toList());
100     }
101
102     menadzeraTableView.setItems(FXCollections.observableList(filtriranaListaMenadzera));
103
104 }
```

Svaki put kada korisnik unese neki podatak koji želi filtrirati mu se uz pomoć lambda .filter() vrate filtrirani podaci. Također pogledaj stranu 30.

5. Korištenje barem dvije generičke klase u aplikaciji koje su smještene u paket zajedno s entitetima. Jedna klasa mora imati samo jedan parametar, a druga klasa mora imati dva parametra generičkog tipa.

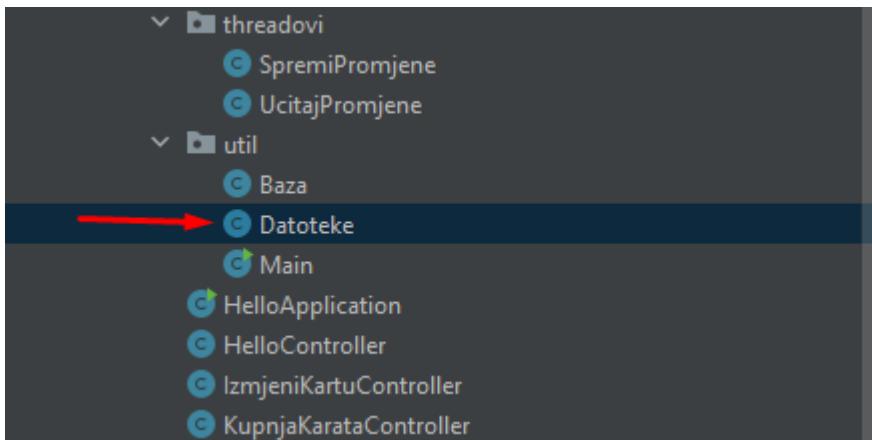


Generička klasa sa jednim parametrom: LjudiUTimu

Sa 2 parametra: Poruke

Pojašnjene na stranicama 54-55

6. Korištenje tekstualnih datoteka koje učitavaju podatke o korisničkim imenima i lozinkama prilikom prijave korisnika u aplikaciju. Potrebno je koristiti i binarne datoteke kojima se serijaliziraju i deserijaliziraju podaci o obavljenim promjenama podataka u projektnom zadatku (na primjer, nakon unošenja novih podataka te promjene postojećih).



U klasi „Datoteke“ citamo sve podatke koji su nam zapisani u datotekama.

„Login“ je objašnjen na stranicama 20-22

U toj se klasi također nalaze metode koje čitaju podatke iz binarne datoteke.

```

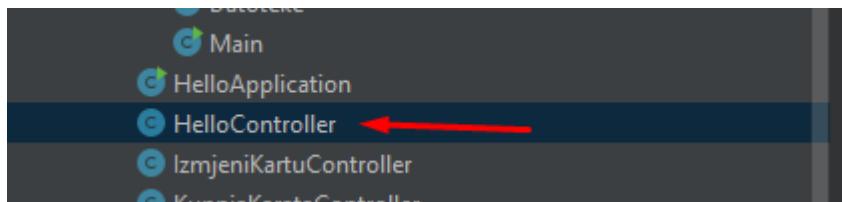
52     1 usage
53     public static List<Karta> ucitajKarte() {
54         List<Karta> lista;
55
56         try {
57             ObjectInputStream in = new ObjectInputStream(
58                 new FileInputStream( name: "dat/serijalizraneKarte.dat"));
59
60             lista = (List<Karta>) in.readObject();
61
62             in.close();
63         } catch (IOException | ClassNotFoundException e) {
64             throw new RuntimeException(e);
65         }
66
67         return lista;
68     }
69
70     1 usage
71     public static List<String> ucitajPromjene() {
72         List<String> promjene;
73
74         try {
75             ObjectInputStream in = new ObjectInputStream(
76                 new FileInputStream( name: "dat/serijaliziranePogreske.dat"));
77
78             promjene = (List<String>) in.readObject();
79
80             in.close();
81         } catch (IOException | ClassNotFoundException e) {
82             throw new RuntimeException(e);
83         }
84
85         return promjene;
86     }

```

Binarne datoteke objašnjene na stranama 42,43,49.

Nakon što smo pročitali sve iz binarne datoteke „castamo“ odnosno pretvaramo sve to pročitano u Listu, nakon toga dalje radimo sa listom.

7. Implementirati JavaFX ekran za prijavu korisnika u aplikaciju koja čita podatke iz tekstualne datoteke o korisničkim imenima i „hashiranim“ lozinkama iz tekstualne datoteke kreirane u šestom koraku. Svaka aplikacija mora imati barem dvije korisničke role.

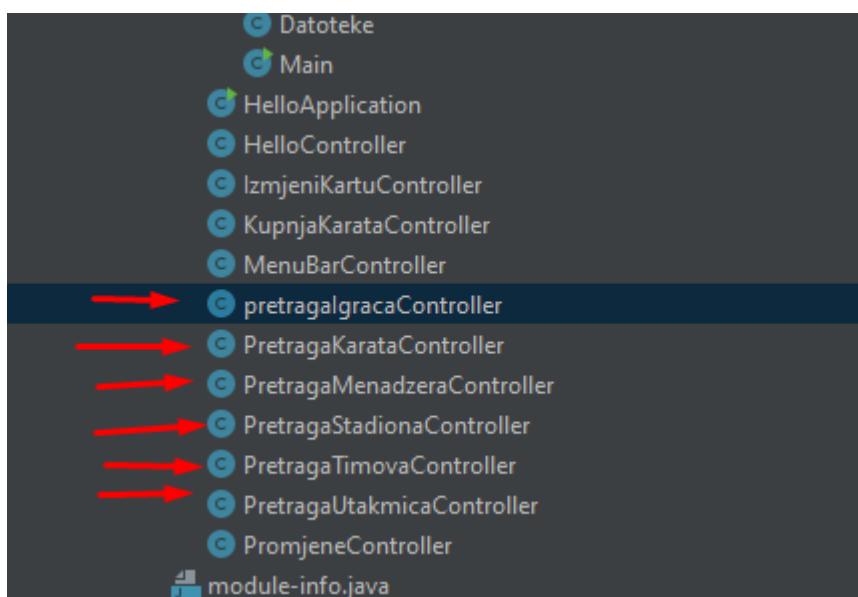


U HelloController-u se nalazi kod koji provjerava je li korisnik unio dobar username i password.

Kako izgleda prijava i kako je povezana sa kodom na strani 27-29.

8. Implementirati JavaFX ekran koji će za svaki entitet omogućavati korištenje funkcionalnosti pretrage i filtriranje podataka (korištenjem tablice TableView), dodavanje novog entiteta, promjene postojećih entiteta te brisanje entiteta. Svaka akcija promjene i brisanja entiteta mora uključivati dodatnu potvrdu korisnika da je suglasan s promjenom ili brisanjem korištenjem JavaFX dijaloga.

Ovo se nalazi u svakoj klasi koja ima „Pretraga“ u imenu.

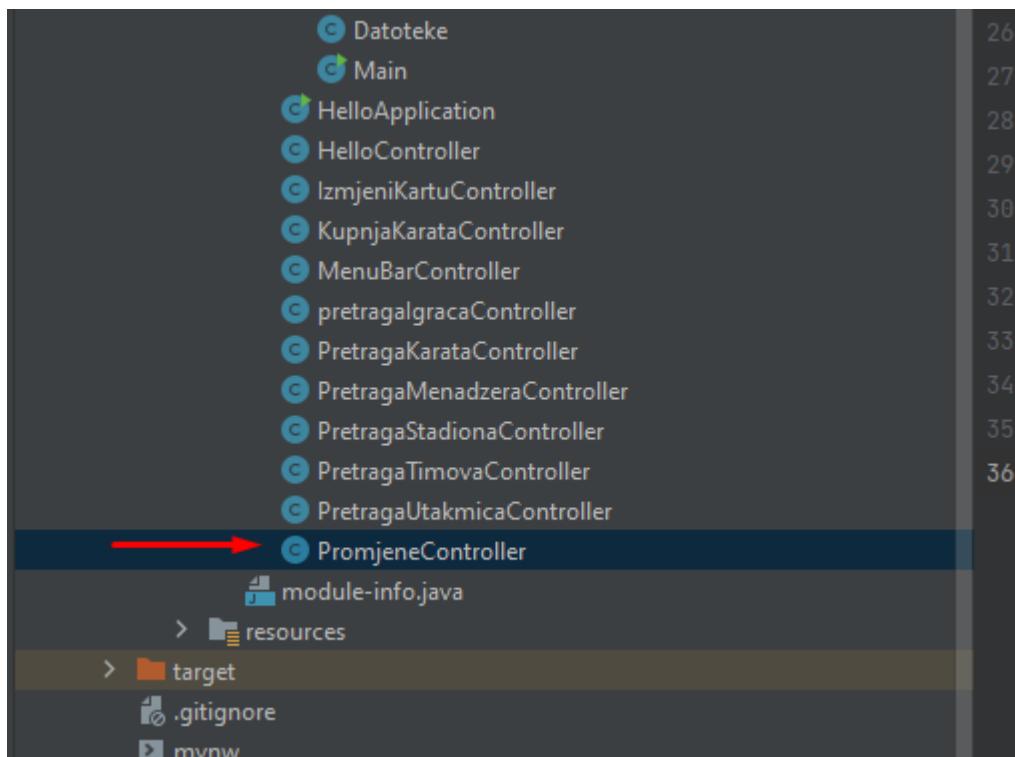


„Akcija promjene i brisanja entiteta“ je napravljena na „PretragaKarataController“.

Ovdje korisnik može brisati i izmjenjivati karte koje do sada ima.

Više na stranicama 40-45

9. Implementirati JavaFX ekran koji će omogućavati prikaz svih promjena koje su obavljene u aplikaciji projektнog zadatka korištenjem serijaliziranih podataka iz šestog koraka. Svaka promjena mora sadržavati podatak koji je promijenjen, staru i novu vrijednost, rolu koja ga je promijenila te datum i vrijeme kad se ta promjena dogodila.



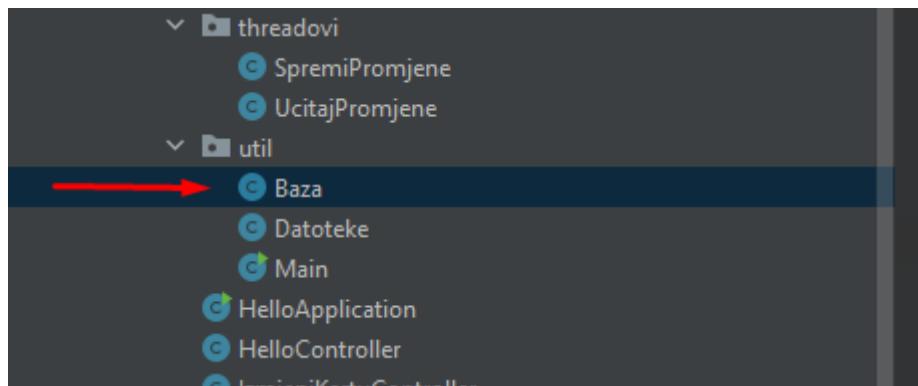
Zapisujemo sve promjene koje su se dogodile od kad smo pokrenuli aplikaciju. U slučaju da je korisnik obrisao kartu, kupio kartu, promjenio kartu, krivo unio sjedalo na karti, zapisuje se poruka u `List<String> svePromjene`.

Ta lista poruka sadrži sve podatke o svim izmjenama koje su se dogodile.

Dodatno objašnjeno na stranicama 43 i 47.

10. Kreirati bazu podataka koja će sadržavati podatke o svim entitetima koji se koriste u aplikaciji te implementirati klasu koja će implementirati funkcionalnosti kreiranje konekcije s bazom podataka, izvršavanje upita nad bazom podataka, dohvaćanje podataka iz baze podataka te zatvaranje konekcije s bazom podataka.

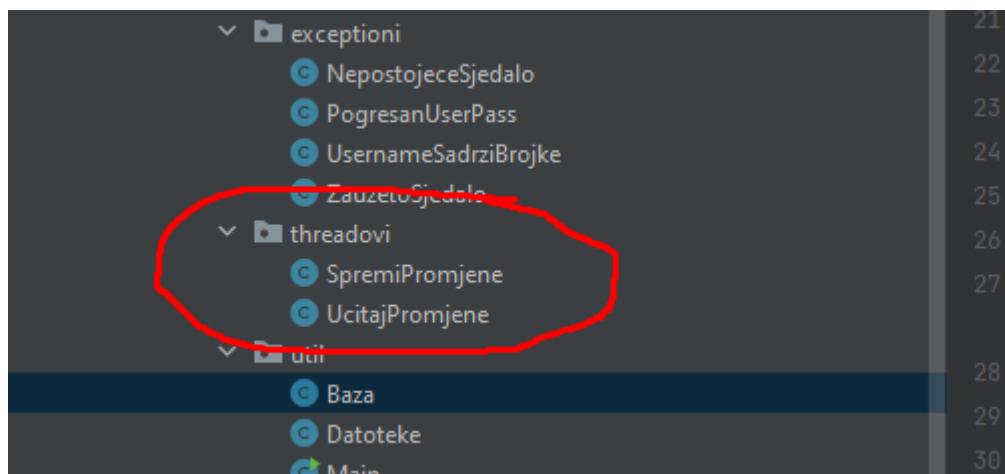
Sve vezano uz bazu podataka se nalazi u sljedećoj klasi:



Dodatno objašnjeno na stranicama 5 – 19.

Ukratko: u klasi „Baza“ imamo metodu sa kojom se connectamo na bazu i sve metode koje čitaju podatke iz baze. Nakon šta su svi podaci pročitani, spremaju se u listu.

11. Korištenjem niti implementirati funkcionalnosti osvježavanja podataka na ekranu aplikacije te konkurentno pristupanje dijeljenom resursu kojem pristupa više niti kroz sinkronizaciju niti (npr. jedna nit ispisuje detalje o posljednje promijenjenom podatu koji dohvata iz serijalizirane datoteke, a za drugu nit koja sprema nove promjene u serijaliziranu datoteku osigurana je sinkronizacija s tom prvom niti).



Threadovi su dodatno objašnjeni na stranicama: 61-62

## UPDATES:

Pošto moramo osigurati da su Threadovi synchronized, umjesto da nam se kod nalazi u metodi „run()“ od Thread-ova, smo ih premjestili u nove metode.

The screenshot shows a Java code editor with two parts. The top part displays a class definition:

```
3 usages
public class UcitajPromjene implements Runnable{
    @Override
    public void run(){
    }
}
```

A red circle highlights the word "run()", and a red arrow points from it to the bottom part of the screen. To the right of the code, the text "KOD KOJI JE BIO TU JE PREMJESTEN" is written in red. The bottom part shows the moved code in a separate method:36 promjene.appendText( s: promjeneUAplicaciji.get(i)+"\n");
37 }
38 }
39
40 1 usage
41 public static synchronized void ucitavanjePromjena(){
42 PromjeneController.promjeneUAplicaciji = HelloController.svePromjene;
43 }
44
45 1 usage
public static synchronized void spremanjePromjena(){

A red arrow points from the word "ucitavanjePromjena()" in the moved code back up to the circled "run()" in the original code.

U novu metodu

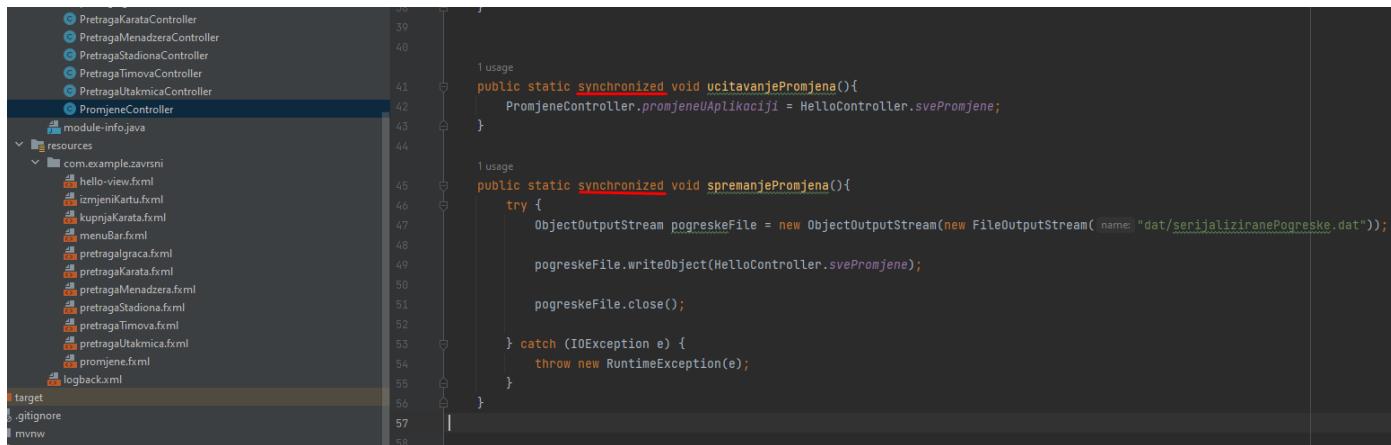
Zatim smo tu metodu u Threadu pozvali.

The screenshot shows the Java code editor again. On the left, the file structure is visible with "threadovi" selected. On the right, the code for the "UcitajPromjene" class is shown:

```
3 usages
public class UcitajPromjene implements Runnable{
    @Override
    public void run(){
        PromjeneController.ucitavanjePromjena();
    }
}
```

A red arrow points from the line "PromjeneController.ucitavanjePromjena();" to the circled "run()" in the original code at the top of the screen.

To smo napravili zato da metodama možemo dodati ključnu riječ „synchronized“.

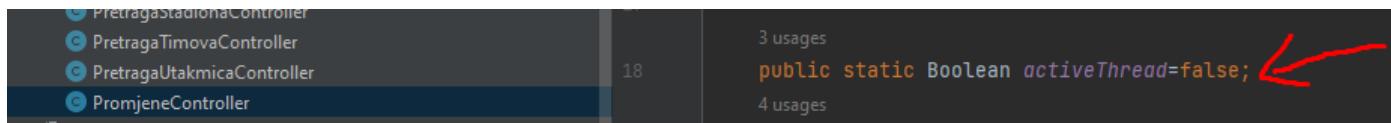


```
39
40
41 1 usage
42 public static synchronized void ucitavanjePromjena(){
43     PromjeneController.promjeneUApplikaciji = HelloController.svePromjene;
44 }
45
46 1 usage
47 public static synchronized void spremanjePromjena(){
48     try {
49         ObjectOutputStream pogreskeFile = new ObjectOutputStream(new FileOutputStream( name: "dat/serijaliziranePogreske.dat"));
50
51         pogreskeFile.writeObject(HelloController.svePromjene);
52
53         pogreskeFile.close();
54     } catch (IOException e) {
55         throw new RuntimeException(e);
56     }
57 }
```

Kod i dalje radi identično kao i prije, samo što su nam sad threadovi sinkronizirani.

Sada kada imamo te dvije metode moramo osigurati da se prvo izvrši spremanje dosadašnjih promjena, a zati da se učitaju na ekran (pošto se threadovi izvršavaju u isto vrijeme)

Za to cemo napraviti globalnu varijablu „activeThread“ koja ce biti „true“ ili „false“, odnosno „true“ ako je trenutno aktivan thread, i „false“ ako trenutno nije aktivan thread.



```
18
3 usages
public static Boolean activeThread=false; ↗
4 usages
```

Kada pozovemo Thread „spremiPromjene“ stavit ćemo tu varijablu na „true“, tako da ostali threadovi znaju da je ona trenutno aktivna.

The screenshot shows a Java code editor with a sidebar containing project files. The file 'SpremiPromjene.java' is open, showing a class implementation. A red arrow points from the word 'activeThread' in the code to the 'SpremiPromjene' file in the sidebar, indicating a reference or inspection result.

```
serijaliziraneKarte.dat  
2  
ogs  
pogreske.log  
rc  
main  
java  
com.example.zavrsni  
    > entitet  
    > exceptioni  
    > threadovi  
        SpremiPromjene  
        UcitajPromjene  
util  
    Baza  
    ...  
3 usages  
public class SpremiPromjene implements Runnable{  
    @Override  
    public void run() {  
        PromjeneController.activeThread=true;  
        PromjeneController.spremanjePromjena();  
    }  
}
```

„spremanjePromjena“ ce raditi isto što i prije (serijalizirati sve u datoteku pogresaka) odnosno spremiti sve pogreške koje imamo do sada.

Nakon toga će se varijabla „activeThread“ staviti na „false“ odnosno više neće biti aktivan taj thread. Da bi dojavili svim threadovima koji su u stanju čekanja da sada mogu krenuti koristit ćemo „Object“ varijablu.

The screenshot shows the IntelliJ IDEA interface with the code editor open to the `PromjeneController` class. The code is annotated with inspection results:

- Line 16: 16 usages
- Line 17: 7 usages
- Line 18: 3 usages
- Line 19: 4 usages
- Line 20: 2 usages

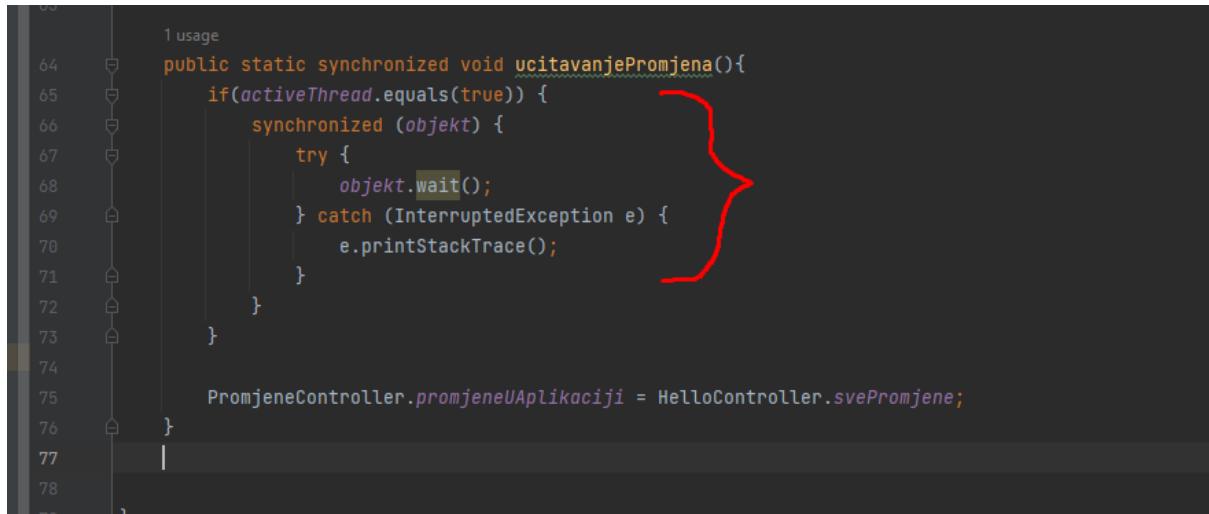
A red arrow points to the line `private final static Object objekt=new Object();`, which has 4 usages.

```
public class PromjeneController {  
    public static Boolean activeThread=false;  
    private final static Object objekt=new Object();  
}
```

Sada ćemo preko te varijable dojaviti svima da je varijable „activeThread“ sada false i moći će nastaviti sa radom.

```
public static synchronized void spremanjePromjena(){  
    try {  
        ObjectOutputStream pogreskeFile = new ObjectOutputStream(new FileOutputStream( name: "dat/serializiranePogreske.dat"));  
  
        pogreskeFile.writeObject(HelloController.svePromjene);  
  
        pogreskeFile.close();  
    } catch (IOException e) {  
        throw new RuntimeException(e);  
    }  
  
    activeThread=false;  
    synchronized (objekt) {  
        objekt.notifyAll();  
    }  
}
```

Na početak druge metode smo dodali ovaj dio koda. (Copy paste je s interneta) On radi sljedeće: ako je varijable „activeThread“ stavljena na „true“ onda će se svi threadovi staviti u stanje „wait“ sve dok varijable „activeThread“ ponovo nije false. Tako smo osigurali da će se threadovi izvršiti kronološki (po nekom redu)



```
1 usage
64     public static synchronized void ucitavanjePromjena(){
65         if(activeThread.equals(true)) {
66             synchronized (objekt) {
67                 try {
68                     objekt.wait();
69                 } catch (InterruptedException e) {
70                     e.printStackTrace();
71                 }
72             }
73         }
74     }
75     PromjeneController.promjeneUAplicaciji = HelloController.svePromjene;
76 }
77
78 }
```

The screenshot shows a portion of Java code in an IDE. A red curly brace is drawn around the entire synchronized block from line 64 to line 73, indicating its scope. The code is as follows:

```
1 usage
64     public static synchronized void ucitavanjePromjena(){
65         if(activeThread.equals(true)) {
66             synchronized (objekt) {
67                 try {
68                     objekt.wait();
69                 } catch (InterruptedException e) {
70                     e.printStackTrace();
71                 }
72             }
73         }
74     }
75     PromjeneController.promjeneUAplicaciji = HelloController.svePromjene;
76 }
77
78 }
```

U kratko :

1. Threadovi „SpremiPromjene“ i „UcitajPromjene“ se stavljaju u fazu „spremni za korištenje“
2. Pošto ne možemo odrediti koji thread će se kada pokrenuti (svi se pokreću u isto vrijeme) moramo ih sinkronizirati da se izvrše jedan pa drugi
3. Radimo varijablu „activeThread“ koja će nam reći ako je thread trenutno aktivan.
4. Kada se pokrene prvi thread on će varijablu „activeThread“ staviti na „true“, ako se iti jedan drugi thread pokuša pokrenuti neće moći sve dok ta varijable nije „false“
5. Kada je prvi thread gotov sa radnjom stavlja varijablu „activeThread“ na false i radi notifyAll() da bi svima dao do znanja da je gotov. Nakon toga se može izvršiti drugi thread.

Razlika između serijalizacije i deserijalizacije.

Serijalizacija služi kako bi neki objekt (ili objekte) spremili kao niz primitivnih podataka da bi taj objekt spremili u trajnu memoriju.

Deserijalizacija služi da bi se taj objekt vratio iz niza primitivnih znakova nazad u pocetni objekt.

Npr: imamo objekt

```
Student student = new Student(“Ivo”, “Ivic”, “0246123123”);
```

-kada bi ovaj objekt htjeli spremiti u tekstualnu datoteku onda bi morali pisati dio po dio :

Ivo

Ivic

0246123123

I onda bi ga procitali red po red i spremili nazad u objekt.

Sa serijalizacijom možemo taj cijeli objekt spremiti od jednom (sa jednom linijom koda), a sa deserijalizacijom pročitati i spremiti u objekt sa jednom linijom koda.

Primjer serijalizacije:

```
ObjectOutputStream file = new ObjectOutputStream(new  
FileOutputStream(“dat/datoteka.dat”));  
  
file.writeObject(student);  
  
file.close();
```

Tako smo objekt zapisali u datoteku datoteka.dat

Primjer deserijalizacije:

```
ObjectInputStream file = new ObjectInputStream(new  
FileInputStream("dat/datoteka.dat"));  
  
Student student = (Student) file.readObject();  
  
file.close()
```

Datkle:

- serijalizacija radi writeObject(), a deserijalizacija radi readObject()
- serijalizacija koristi ObjectOutputStream klasu, a deserijalizacija koristi ObjectInputStream klasu
- bitno je da nakon što smo pročitali objekt ga „castamo“ odnosno pretvorimo u točno onaj koji objekt koji nam treba (npr kako smo u gornjem primjeru napravili (Student) ispred file.readObject())

U našem primjeru na projektu smo napravili identičnu stvar samo što smo umjesto jednog objekta serijalizirali cijelu listu objekta od jednom.

```

1 usage
public static synchronized void spremanjePromjena(){
    try {
        ObjectOutputStream pogreskeFile = new ObjectOutputStream(new FileOutputStream( name: "dat/serijaliziranePogreske.dat"));

        pogreskeFile.writeObject(HelloController.svePromjene);

        pogreskeFile.close();
    } catch (IOException e) {
        throw new RuntimeException(e);
    }

    activeThread=false;
    synchronized (objekt) {
        objekt.notifyAll();
    }
}

1 usage
public static synchronized void ucitavanjePromjena(){
    if(activeThread.equals(true)) {
        synchronized (objekt) {
            try {
                objekt.wait();
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
}

PromjeneController.promjeneUAplicaciji = HelloController.svePromjene;

```

Ako odemo u klasu „Datoteke“ tamo ćemo naći sve metode koje koristimo za deserijalizaciju

```

52     public static List<Karta> ucitajKarte() {
53         List<Karta> lista;
54
55         try {
56             ObjectInputStream in = new ObjectInputStream(
57                 new FileInputStream( name: "dat/serijaliziraneKarte.dat"));
58
59             lista = (List<Karta>) in.readObject();
60
61             in.close();
62         } catch (IOException | ClassNotFoundException e) {
63             throw new RuntimeException(e);
64         }
65
66         return lista;
67     }
68
69     1 usage
70     public static List<String> ucitajPromjene() {
71         List<String> promjene;
72
73         try {
74             ObjectInputStream in = new ObjectInputStream(
75                 new FileInputStream( name: "dat/serijaliziranePogreske.dat"));
76
77             promjene = (List<String>) in.readObject();
78
79             in.close();
80         } catch (IOException | ClassNotFoundException e) {
81             throw new RuntimeException(e);
82         }
83
84         return promjene;
85     }

```

Bitno je da „castamo“ sve pročitane objekte nazad u listu.

Te metode se implementiraju u „HelloController“-u.

The screenshot shows the project structure on the left and the code editor on the right. The code editor displays the `HelloController.java` file with the following content:

```
public void initialize() {
    try( Connection veza = Baza.connectToDatabase() ) {
        System.out.println("Connected to database!");
        listaIgraca = Baza.dohvatiIgraceIzBaze(veza);
        listaMenadzera = Baza.dohvatiMenadzereIzBaze(veza);
        listaStadiona = Baza.dohvatiStadioneIzBaze(veza);
        listaTimova = Baza.dohvatiTimoveIzBaze(veza);
        listaUtakmica = Baza.dohvatiUtakmiceIzBaze(veza); TEKSTUALNA DATOTEKA
        listaNavijaca = Datoteke.ucitajNavijace(); BINARNA DATOTEKA
        listaKarata = Datoteke.ucitajKarte(); BINARNA DATOTEKA
        svePromjene = Datoteke.ucitajPromjene(); BINARNA DATOTEKA
    } catch (SQLException | IOException e) {
        e.printStackTrace();
    }
}
```

Red arrows point from the annotations TEKSTUALNA DATOTEKA, BINARNA DATOTEKA, and BINARNA DATOTEKA to the respective lines of code where lists are populated from database queries.

Binarne datoteke su .dat datoteke (one u koje serijaliziramo i iz kojih deserijaliziramo).

Navijaci su napravljeni kroz tekstualnu datoteku jer tako piše u zadatku

Kod hashiranja passworda morali smo napraviti da se u datoteci ne ispisuju pravi passwordi nego „hashirani“ (skriveni) passwordi.

To je najlakše bilo postići sa klasom „MessageDigest“. I par metoda poput „digest()“ i „encodeToString()“.

Cijeli kod je kopiran s interneta i radi sljedeće:

Zamislimo da imamo password npr: password123

Ali ne želimo da korisnikov password tako bude zapisan u datoteci jer bi bilo tko ko koristi našu aplikaciju mogao vidjeti njegov password.

Rješenje je da se svi password (sva slova iz passworda) pretvore u neki skriveni podatak koji se samo dekodiranjem može razumjeti (npr.

Umjesto password123 se zapiše:

jGI25bVBBBW96Qi9Te4V37Fnqchz/Eu4qB9vKrRlqRg=

Sada nitko tko otvorи aplikaciju ne može zaključiti o kojem se passwordu radi.

Pošto mi nemamo registraciju (nego samo login i 2 accounta) morali smo taj hashirani password tako zapisati u datoteku (doslovno copy paste).

Primjer:

ako korisnik unese „admin“ za password kako da usporedimo „admin“ sa jGI25bVBBBW96Qi9Te4V37Fnqchz/Eu4qB9vKrRlqRg= ?

Kako znamo da je to zapravo jedna te ista stvar?

To radimo tako da korisnikov password koji je unio npr „admin“ pretvorimo u hashirani password i onda ćemo dobiti:

jGI25bVBBBW96Qi9Te4V37Fnqchz/Eu4qB9vKrRlqRg=

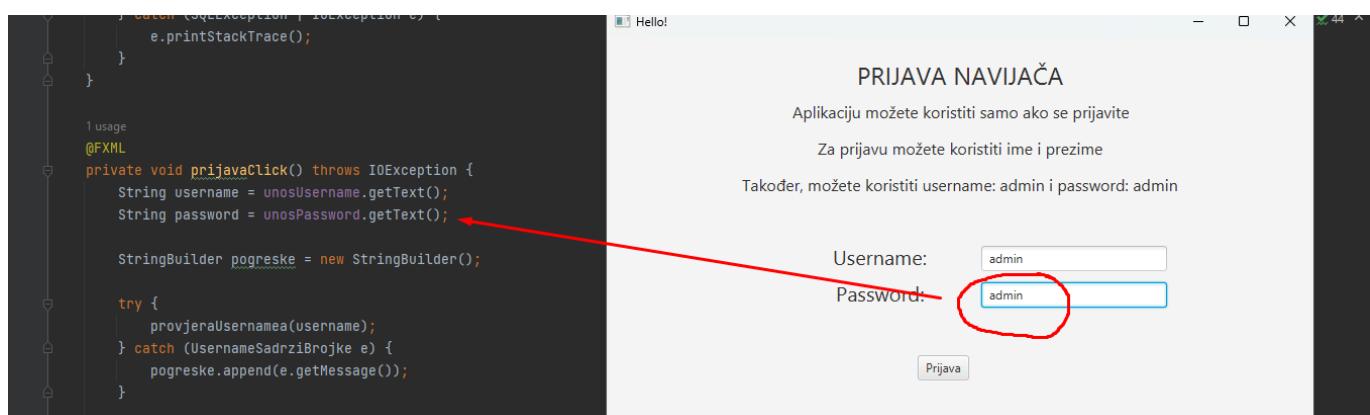
Nakon toga gledamo da li je

jGI25bVBBBW96Qi9Te4V37Fnqchz/Eu4qB9vKrRlqRg=

Jednak kao password koji nam je zapisan u datoteku (odnosno  
jGI25bVBBBW96Qi9Te4V37Fnqchz/Eu4qB9vKrRlqRg=)

Vidimo da je, i zato znamo da je korisnik unio točan password.

Dakle: umjesto da radimo sa passwordima u tekstualnom obliku, pretvaramo ih u hashirani oblik i radimo sa hashiranim passwordima u programu.



Korisnik unese šifru i sprema se ovdje.

```
85
86     try {
87         uspjesnostLogina = provjeraUSernameaIPassworda(username, password);
88     }catch (PogresanUserPass e){
89         pogreske.append(e.getMessage());
90     }
91 }
```

Poziva se metoda koja provjerava da li je unio dobar username i password

```

1 usage
113 @ ...
    private boolean provjeraUsernameIPassworda(String username, String password) throws Pogreška {
        Boolean provjeraLogina = false;
        String hashPassword = "";
        PRETVARA SE U hashirani oblik
        try {
            MessageDigest messageDigest = MessageDigest.getInstance(algorithm: "SHA-256");
            byte[] hash = messageDigest.digest(password.getBytes());
            hashPassword = Base64.getEncoder().encodeToString(hash);
        } catch (NoSuchAlgorithmException e) {

```

Datoteka	Uredivanje	Prikaži
	admin jG125bVBBBW96Qi9Te4V37Fnqchz/Eu4qB9vKrRIqRg=	2
	andrea 6uoF3vFfJVHruE1VmATnEbHXUWh+As82FhOVid6/cSA=	3
	aleksandar u+/2uS09rmFq21/rLLxjHmKyi8Xn+TQJN5TFkg7kkM0=	

```

121     hashPassword = Base64.getEncoder().encodeToString(hash);
122
123
124 } catch (NoSuchAlgorithmException e) {
    throw new RuntimeException(e);
} OVDJE SE NALAZE SVI USERNAME I PASSWORDI KOJI SU
125     PROČITANI IZ DATOTEKE KOJA SE VIDI TU DESNO
126
127 for(int i=0;i<listaNavijaca.size();i++){
128     if(!listaNavijaca.get(i).getUsername().equals(username)){
129         if(listaNavijaca.get(i).getPassword().equals(hashPassword)){
130             provjeraLogina = true; USPOREĐUJEMO HASHIRANI PASSWORD SA
131             uLogiraniNavijac = listaNavijaca.get(i); PASSWORDOM ZAPISANIM
132         }
133     }
134 }
135
136
137 if(provjeraLogina == false){

```

Dakle: u listi navijača se nalaze svi username i passwordi koji se vide tu desno.

Naš uneseni password pretvorimo u hashirani oblik umjesto običnog tekstualnog i onda se uspoređuje sa hashiranim passwordima iz datoteke (odnosno liste, pošto se tamo spreme na početku programa)