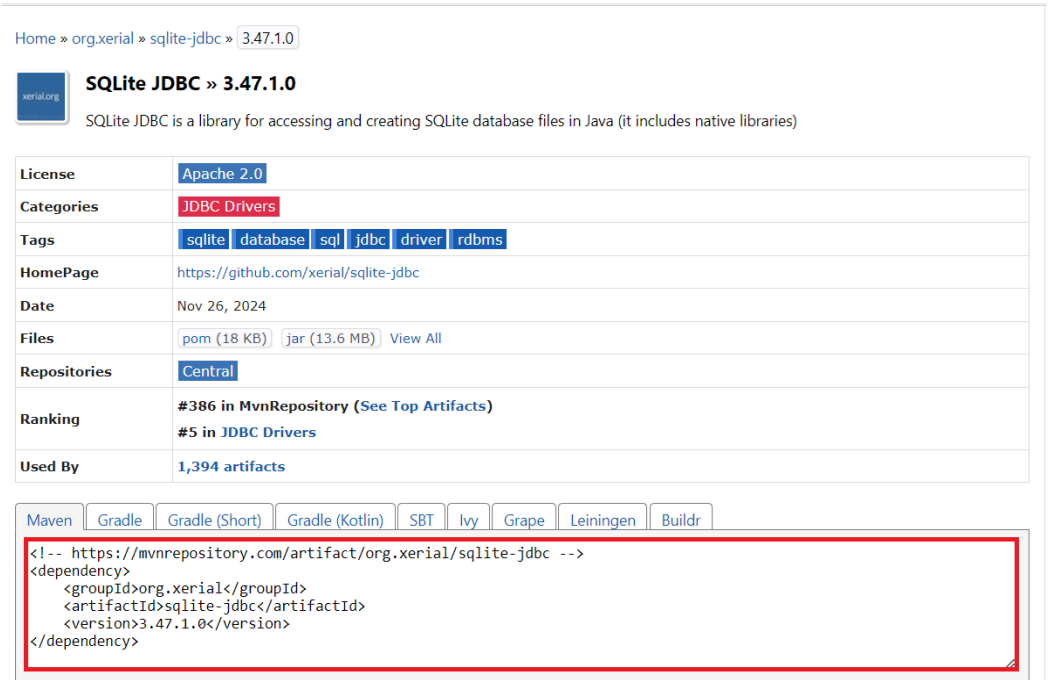
Laboratorijska vježba 9.

Rad sa bazom podataka

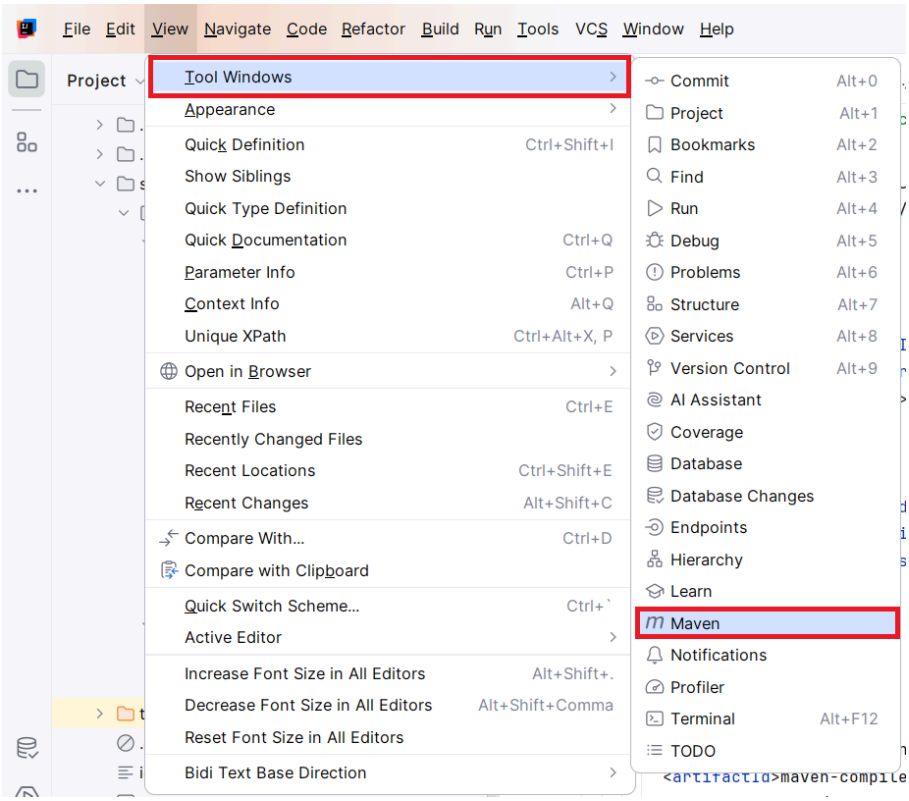
9.1. SQLite baza podataka

Baza podataka je strukturirana kolekcija podataka koja omogućava efikasnu pohranu, preuzimanje i upravljanje informacijama. Baze podataka se često koriste u JavaFX aplikacijama za čuvanje korisničkih podataka, stanja aplikacije ili bilo kojih drugih strukturiranih informacija. Postoje različiti tipovi baza podataka a na laboratorijskim vježbama će se koristiti relaciona SQLite baza podataka. Za korištenje ove baze podataka, u IntelliJ, prvo je potrebno uključiti *dependency* koji se odnosi na *SQLite JDBC driver*. Izgled ovog *dependency* se može pronaći na sljedećem linku. Izaberite najnoviju verziju i kopirajte *dependency* (dio koji je označen na Slici 1) u *pom.xml* datoteku vašeg JavaFX projekta. Vodite računa da *dependency* kopirate u dio koji se nalazi između ***dependencies*** otvarajućeg (*<dependencies>*) i zatvarajućeg taga (*</dependencies>*). Također, verzija *SQLite JDBC driver dependency* ne mora biti ista kao ona koja je prikazana na Slici 1.

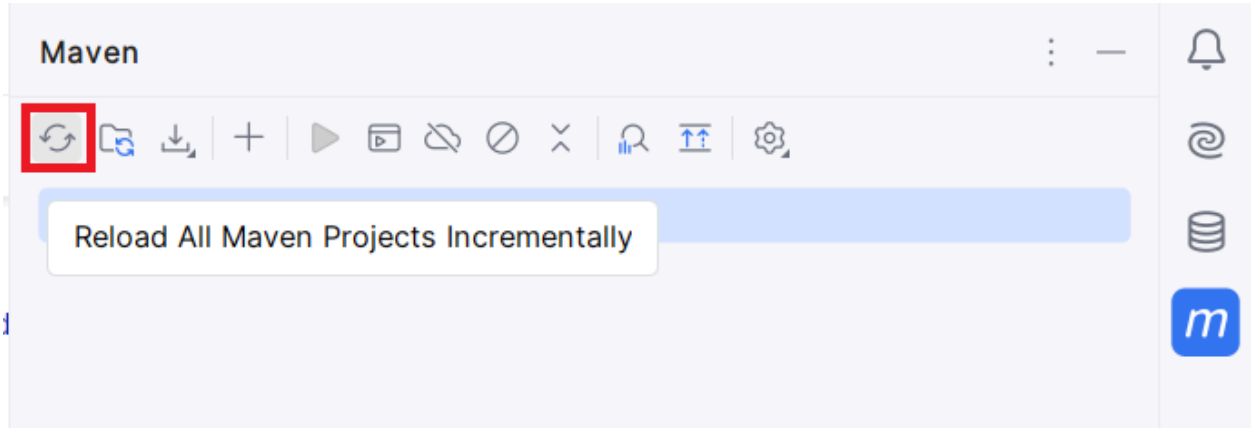


Slika 1. SQLite JDBC driver dependency

Nakon uključivanja *dependency* u projekat, moguće je da će se prikazati greška da okruženje ne prepoznaje dodani *dependency*. U tom slučaju, u glavnom meniju izaberite *View* → *Tool Windows* → *Maven* i uradite *Reload Maven* projekta. Ovo će učitati paket i otkloniti grešku. Prethodno opisani koraci su prikazani na Slici 2 (prvi dio koraka) i Slici 3 (ponovno učitavanje *Maven* projekta).



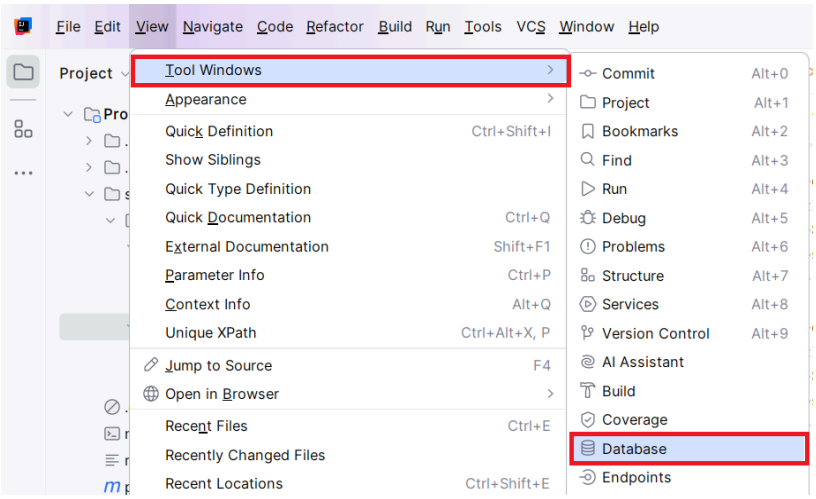
Slika 2. Izbor opcije Maven u View dijelu menija



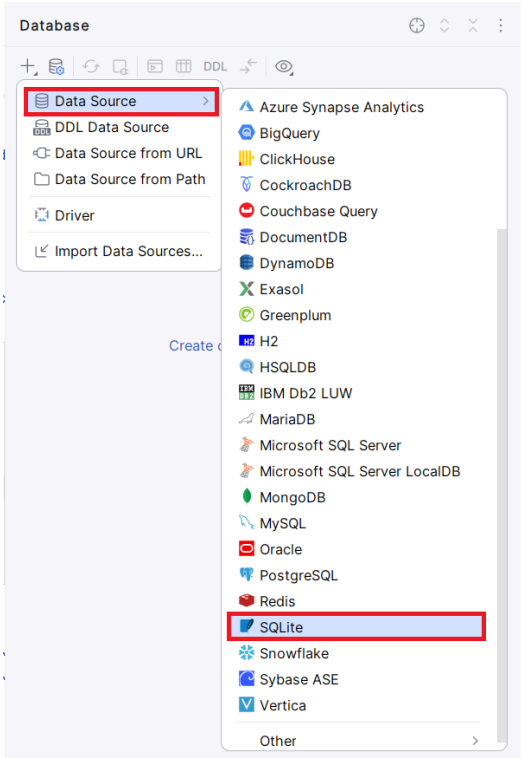
Slika 3. Maven reload projekta

9.2. Kreiranje baze podataka

Kako bi rad sa bazom podataka bio moguć, potrebno je kreirati datoteku koja se odnosi na samu bazu podataka. Neka je to datoteka naziva ***baza.db***. Ukoliko imate *IntelliJ Ultimate Edition*, ovu datoteku je moguće jednostavno kreirati izborom *View → Tool Windows →* *Database* opcije u glavnom meniju (Slika 4). U prozoru koji se otvori, izaberite opciju *New* (znak "+") i u *Data Source* izaberite *SQLite* (Slika 5).

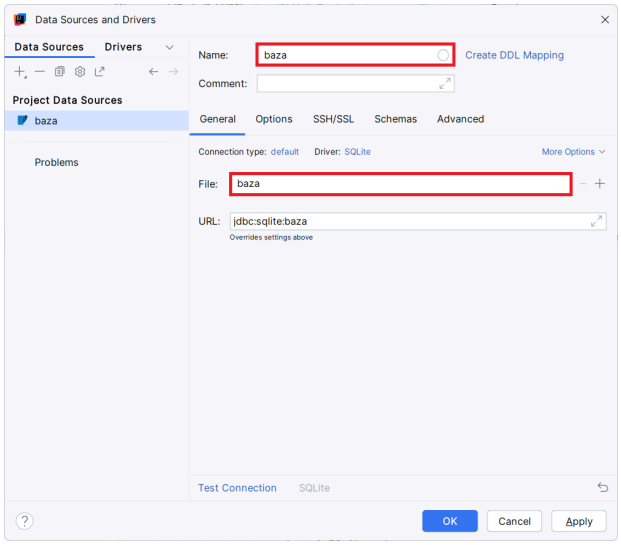


Slika 4. Izbor Database opcije u View → Tool Windows glavnog menija



Slika 5. Kreiranje baze podataka kroz Database prozor

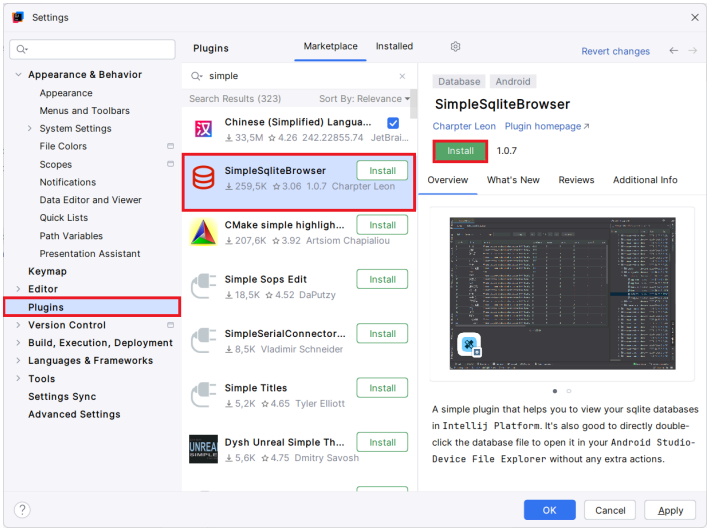
Bazi podataka dajte naziv ***baza*** kroz polja *Name* i *File* u prozoru koji se otvorio nakon prethodnog koraka (Slika 6). Kada ste završili sa dodjelom naziva, kliknite OK.



Slika 6. Davanje naziva bazi podataka

9.2.1. SimpleSqlite Browser plugin

Za lakši rad sa bazom podataka (ili ako nemate *IntelliJ Ultimate Edition*), možete instalirati *SimpleSqlite Browser plugin*. Koraci za preuzimanje ovog *plugin*-a će biti opisani u nastavku. U glavnom meniju izaberite *File* → *Settings*. U prozoru koji se otvorio izaberite dio *Plugins*. U polju za pretragu ukucajte *SimpleSqlite Browser* (dovoljno je ukucati samo "simple" i jedan od početnih rezultata pretraživanja bi trebao biti traženi *plugin*). Izaberite opciju *Install*. Koraci su prikazani na Slici 7. Nakon instalacije bit će potrebno uraditi restart okruženja klikom na dugme "Restart IDE", koje se pojavi kada se završi instalacija *plugin*-a.



Slika 7. Instalacija SimpleSqlite Browser plugin ekstenzije

9.3. Povezivanje sa bazom podataka i pisanje upita

9.3.1. Povezivanje sa bazom podataka

JDBC je API koji pruža Java za povezivanje i rad sa relacionim bazama podataka. Nakon kreiranja baze podataka, istu je potrebno povezati sa JavaFX aplikacijom. Kao polazni primjer kreirat će se klasa naziva *Database*, čiji je izgled prikazan u Listingu 1.

```
public class Database {
    private static final String DB_URL = "jdbc:sqlite:baza.db";

    public static Connection connect() {
        Connection conn = null;
        try {
            conn = DriverManager.getConnection(DB_URL);
            System.out.println("Povezano s bazom podataka!");
        } catch (Exception e) {
            System.out.println(e.getMessage());
        }
        return conn;
    }
}
```

Listing 1. Klasa za rad s bazom podataka

Prva linija koda klase *Database* se koristi za deklaraciju URL-a (putanje) do baze podataka. Varijabla *DB_URL* je konstanta tipa *String* a njen sadržaj (tj. "**jdbc:sqlite:baza.db**") predstavlja *JDBC URL* koji specificira lokaciju SQLite baze podataka. Dijelovi URL-a imaju sljedeće značenje:

- *jdbc* - označava da se koristi *Java Database Connectivity* za povezivanje sa bazom podataka.
- *sqlite* - označava da se koristi SQLite baza podataka.
- *baza.db* - relativna putanja do SQLite baze podataka. Ovo znači da se baza nalazi u *root* direktoriju projekta (na istom nivou hijerarhije kao *src*). Ukoliko ste bazu pohranili na neku drugu lokaciju, promijenite ovaj dio URL-a tako da je putanja ispravna. Da biste znali koja je ispravna putanja, možete izabrati opciju *Properties* nad bazom podataka u *Database* prozoru (uradite desni klik na naziv baze podataka nakon čega izaberete *Properties* i u novootvorenom prozoru iščitate vrijednost polja URL). Osim relativne, možete koristiti i apsolutnu putanju.

Metoda *connect* se koristi za povezivanje sa bazom podataka i vraća objekat tipa *Connection*. Ova metoda je označena kao *static*, što znači da pripada klasi kao cjelini a ne

pojedinim objektima klase. Unutar ove metode deklarisana je varijabla *conn* tipa *Connection*. Njena inicijalna vrijednost je *null*. Ovo osigurava da varijabla ima početnu vrijednost koja se kasnije može provjeriti kako bi se znalo da li je povezivanje sa bazom podataka uspješno završeno.

U nastavku metode vrši se pokušaj povezivanja sa bazom podataka. Obzirom da povezivanje može biti neuspješno, potrebno je koristiti *try-catch* blok za ispravno upravljanje izuzecima. Dijelovi programskog koda u *try* bloku se odnose na:

- *DriverManager.getConnection(DB_URL)*:
  - *DriverManager* - klasa iz JDBC-a koja upravlja konekcijama sa bazom podataka.
  - *getConnection(String url)* - metoda koja otvara konekciju prema bazi podataka navedenoj u *DB_URL*.
  - Ako je povezivanje sa bazom podataka uspješno, vraća se objekat tipa *Connection*, koji se dodjeljuje varijabli *conn*.

Ako se konekcija uspješno uspostavi, korisniku se daje povratna informacija o tome u vidu konzolnog ispisa. Blok *catch* je zadužen za hvatanje izuzetka i ispisivanje poruke izuzetka u konzolu (ukoliko dođe do toga). Konačno, nakon što se konekcija uspostavi, iz metode se vraća objekat *Connection*. Ako konekcija nije uspjela (npr. *DriverManager.getConnection* je bacio izuzetak), varijabla *conn* ostaje *null*.

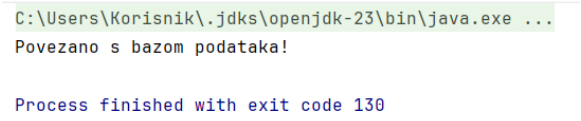
Metodu za povezivanje sa bazom podataka je moguće provjeriti kroz *main* metodu *Main* klase. Modificirajte ovu metodu tako što ćete zakomentarisati poziv *launch()* metode i dodati poziv *connect()* metode iz klase *Database*, kao što je prikazano u Listingu 2.

```
public static void main(String[] args) {
    //launch();

    Connection connect = Database.connect();
}
```

Listing 2. Poziv metode za konekciju s bazom podataka

Rezultat izvršavanja bi trebao biti kao na Slici 8. Ukoliko je došlo do neke greške, ispravite grešku (najvjerovatnije je riječ o neispravnoj putanji do datoteke koja predstavlja bazu podataka) i ponovo pokrenite program sve dok ne dobijete ispravan izlaz.



Slika 8. Konzolni ispis nakon ispravnog povezivanja sa bazom podataka

9.3.2. Pisanje SQL upita

Nakon što je uspješno uspostavljena konekcija sa bazom podataka, potrebno je napraviti tabele i pisati SQL upite nad tim tabelama. U nastavku će biti prikazana sintaksa na apstraktnom nivou. Dakle, isječci koda iz ovog dijela laboratorijske vježbe se ne mogu samo kopirati bez obavljanja određenih modifikacija. Na ovaj način će biti opisane osnovne klase i funkcionalnosti koje se koriste za rad sa bazom podataka a konkretna implementacija nad bazom podataka će biti prikazana u nastavku. Apstraktni primjer sintakse koja se koristi za pisanje upita u JavaFX je prikazan u Listingu 3.

```
String SQL_QUERY = "SELECT * FROM korisnici WHERE ime = ? AND
godine > ?";
try (Connection conn = DriverManager.getConnection(DB_URL);
    PreparedStatement pstmt = conn.prepareStatement(SQL_QUERY)) {

   // (1) Postavljanje parametara u PreparedStatement
   pstmt.setString(1, parametar1);
   pstmt.setInt(2, parametar2);

   // (2) Izvršavanje upita
    ResultSet rs = pstmt.executeQuery(); // Koristi se za SELECT
upite
     int affectedRows = pstmt.executeUpdate(); // Koristi se za
INSERT, UPDATE, DELETE

   // (3) Obrada rezultata (samo za SELECT upite)
   while (rs.next()) {
       String ime = rs.getString("ime");
       int godine = rs.getInt("godine");
       // Rad s dobijenim podacima
   }

} catch (SQLException e) {
   // Obrada greške
   System.out.println("SQL Greška: " + e.getMessage());
}
```

Listing 3. Primjer Java SQL upita

Osim prethodno spomenute klase *Connection*, ključne klase i metode za rad sa bazom podataka su:

1. *PreparedStatement* - SQL upit koji je unaprijed kompajliran i omogućava sigurno postavljanje parametara. Primjeri metoda za postavljanje parametara upita su:

- *pstmt.setString(1, "Neko ime");* - postavlja prvi parametar na “Novo ime”.
- *pstmt.setInt(2, 25);* - postavlja drugi parametar na 25.

2. *Statement* - statički SQL upit bez parametara. Ova klasa se koristi kada se pišu jednostavni upiti bez dinamičkih parametara. Pored ovoga, još jedna razlika između *PreparedStatement* i *Statement* je što PreparedStatement unaprijed kompajlira upit, što je brže za upite koji se ponavljaju više puta. Za razliku od njega, *Statement* parsira i kompajlira SQL upit svaki put kada se izvršava.
3. *ResultSet* - rezultat SQL *SELECT* upita. Čuva podatke dohvaćene iz baze podataka i omogućava iteriranje kroz rezultate. Za iteriranje kroz rezultate, koji su vraćeni iz baze podataka, koristi se metoda *next()*. Konkretno, ona pozicionira pokazivač na sljedeći red u rezultatu upita. Kada se prvi put pozove, metoda pomijera pokazivač na prvi red rezultata. Nakon svakog poziva, metoda pomijera pokazivač na sljedeći red. Ova metoda vraća *boolean* vrijednost *true* (ako postoji sljedeći red u rezultatu) ili *false* (ako nema više redova, tj. pokazivač je prošao kroz sve rezultate). Prije prvog poziva ove metode, pokazivač se nalazi prije prvog reda rezultata. Osim metode za iteriranje kroz rezultate, *ResultSet* pruža i metode za dohvaćanje podataka. Primjeri ovih metoda su:
   - *rs.getString("kolona")* - dohvaća vrijednost kolone tipa String.
   - *rs.getInt("kolona")* - dohvaća vrijednost kolone tipa *int*.
4. *executeQuery()* - koristi se za *SELECT* upite. Vraća *ResultSet*.
5. *executeUpdate()* - koristi se za *INSERT*, *UPDATE*, i *DELETE* upite. Vraća broj redova koji su promijenjeni usljed izvršavanja upita.

9.4. MVC i baza podataka

Na osnovu implementacije MVC šablona dizajna za klasu *Osoba*, koja je prikazana u prethodnim laboratorijskim vježbama, u ovoj laboratorijskoj vježbi će se uraditi povezivanje tog projekta sa bazom podataka. Klasa koja će doživjeti najviše promjena je *OsobaModel*, obzirom da ona pohranjuje podatke o osobama i odgovorna je za komunikaciju sa bazom podataka. Nakon kreiranja baze podataka naziva *baza.db* na način koji je opisan na početku, u klasu *OsobaModel* treba dodati varijablu *DATABASE_URL*, koja definira URL baze podataka koja će se koristiti (Listing 4).

```
private static final String DATABASE_URL = "jdbc:sqlite:baza.db";
```

Listing 4. Konekcijski string

Za uspostavljanje konekcije sa bazom podataka, koristit će se metoda *connect()* (Listing 5).

```
private static Connection connect() throws SQLException {
   return DriverManager.getConnection(DATABASE_URL);
}
```

Listing 5. Konekcijska metoda

Iako je povezivanje sa bazom podataka završeno sa prethodnim linijama koda, ona je trenutno prazna. Zbog toga je potrebno napraviti metodu koja će kreirati odgovarajuće tabele u bazi podataka. Trenutno je dovoljna samo tabela *Osoba*. Tabela se kreira na osnovu polja koja su sadržana u *Osoba* klasi a to su ime, prezime, adresa, datumRodjenja, maticniBroj i uloga. U SQLite ne postoji tip podataka koji odgovara datumu, tako da će se *datumRodjenja* u bazi podataka spašavati kao *String*. Metoda *kreirajTabeluAkoNePostoji()* će se koristiti za kreiranje tabele *Osoba*, ako ona već ne postoji. Dakle, prvo se provjerava da li je u bazi podataka već sadržana tabela *Osoba*. Ako nije, onda se kreira. Ukoliko tabela postoji, onda se neće ponovo kreirati. Programski kod navedene metode je prikazan u Listingu 6.

```
public static void kreirajTabeluAkoNePostoji() {
    String kreirajOsobaTabeluSql = """
        CREATE TABLE IF NOT EXISTS Osoba (
            id INTEGER,
            ime TEXT,
            prezime TEXT,
            adresa TEXT,
            datumRodjenja TEXT,
            maticniBroj TEXT,
            uloga TEXT
        );
    """;

    try (Connection conn = connect();
         Statement stmt = conn.createStatement()) {
        stmt.execute(kreirajOsobaTabeluSql);
        System.out.println("Tabela je kreirana ili vec postoji!");
    } catch (SQLException e) {
        System.out.println(e.getMessage());
    }
}
```

Listing 6. Kreiranje SQL tabele kroz Java metodu

Obratite pažnju na korištenje **trostrukih znakova navodnika** prilikom pisanja SQL upita. Naime, ova sintaksa se koristi za definiranje višelinijskog tekstualnog bloka za lakše formatiranje, čitanje i održavanje. Također, nad varijablom koja predstavlja konekciju na bazu (*conn*), korištena je metoda *createStatement()*, jer je riječ o statičkom SQL upitu (tj. upitu bez dinamičkih parametara).

Nakon kreiranja tabele u bazi podataka, istu je potrebno popuniti određenim podacima. Za obavljanje ove operacije koristi se metoda *napuniInicijalnimPodacima()*, čiji je programski kod prikazan u Listingu 7. Ova metoda koristi *prepareStatement()* metodu nad konekcijskim objektom, jer se upit sastoji od parametara (tj. nije statički).

```
public static void napuniInicijalnimPodacima() {
   String insertSQL = """
       INSERT INTO Osoba (id, ime, prezime, adresa, datumRodjenja,
maticniBroj, uloga)
      VALUES (?, ?, ?, ?, ?, ?, ?);
   """;

   try (Connection conn = connect();
         PreparedStatement pstmt = conn.prepareStatement(insertSQL))
{

        pstmt.setInt(1, 1);
        pstmt.setString(2, "John");
        pstmt.setString(3, "Doe");
        pstmt.setString(4, "Some Address");
        pstmt.setString(5, "1995-01-15");
        pstmt.setString(6, "1501995123456");
        pstmt.setString(7, "STUDENT");
        pstmt.executeUpdate();

        pstmt.setInt(1, 2);
        pstmt.setString(2, "Alice");
        pstmt.setString(3, "Alister");
        pstmt.setString(4, "Another Address");
        pstmt.setString(5, "1980-05-20");
        pstmt.setString(6, "2005980444444");
        pstmt.setString(7, "NASTAVNO_OSOBLJE");
        pstmt.executeUpdate();

        System.out.println("Ubaceni pocetni podaci!");
   } catch (SQLException e) {
        System.out.println(e.getMessage());
   }
}
```

Listing 7. Punjenje SQL tabele podacima kroz Java metodu

Za brisanje redova tabele koristit će se metoda *isprazniTabeluOsoba()* (Listing 8). Ova metoda briše sve redove, obzirom da upit u sebi ne sadrži *WHERE* klauzu. Ideja upotrebe ove metode je da se na početku isprazni tabela sa osobama, nakon čega se pozove metoda za popunjavanje tabele podacima, što omogućava da podaci budu u istom početnom stanju prilikom svakog novog pokretanja aplikacije.

```
public static void isprazniTabeluOsoba() {
   String upit = "DELETE FROM Osoba";

   try (Connection conn = connect();
        Statement stmt = conn.createStatement()) {
       int brojObrisanihRedova = stmt.executeUpdate(upit);
         System.out.println("Obrisani redovi tabele. Broj obrisanih
redova: " + brojObrisanihRedova);
   } catch (SQLException e) {
       System.out.println(e.getMessage());
   }
}
```

Listing 8. Brisanje sadržaja SQL tabele kroz Java metodu

U nastavku će biti opisane preostale CRUD operacije upotrebom baze podataka. Metoda *dajSveOsobe()* vraća sve osobe spašene u bazi podataka. Programski kod ove metode je prikazan u Listingu 9.

```
public static List<Osoba> dajSveOsobe() {
   List<Osoba> osobe = new ArrayList<>();
   String upit = "SELECT * FROM Osoba";

   try (Connection conn = connect();
        Statement stmt = conn.createStatement();
        ResultSet rs = stmt.executeQuery(upit)) {

       while (rs.next()) {
           Osoba osoba = new Osoba(
                   rs.getInt("id"),
                   rs.getString("ime"),
                   rs.getString("prezime"),
                   rs.getString("adresa"),

OsobaModel.dateFormat.parse(rs.getString("datumRodjenja")),
                   rs.getString("maticniBroj"),
                   Uloga.valueOf(rs.getString("uloga"))
           );
           osobe.add(osoba);
       }
   } catch (SQLException e) {
       System.out.println(e.getMessage());
   }
   catch (ParseException e) {
```

```
        System.out.println(e.getMessage());
    }
    return osobe;
}
```

Listing 9. Pregled sadržaja SQL tabele kroz Java metodu

Najprije se kreira prazna lista osoba i smješta se u varijablu *osobe*. SQL upit u ovoj metodi je jednostavan, zbog čega se koristi *Statement* klasa za izvršavanje istog. Na osnovu *ResultSet*, koji se dobije nakon izvršavanja upita, pojedina polja klase *Osoba* se preuzimaju korištenjem metoda za dohvaćanje podataka na osnovu tipa tih podataka. Podaci koji se tiču datuma rođenja i uloge osobe se moraju pretvoriti iz *String* tipa u odgovarajući tip. Zbog toga se radi parsiranje ovih vrijednosti prilikom instanciranja objekta klase *Osoba*. Parsiranje može dovesti do bacanja izuzetka, zbog čega je potrebno dodati još jedan *catch* blok, koji će upravljati ovim tipom izuzetaka. Za parsiranje datuma koristi se statička varijabla *dateFormat*, koja definira format u kojem se datum zapisuje (Listing 10).

```
private static final SimpleDateFormat dateFormat = new
SimpleDateFormat("yyyy-MM-dd");
```

Listing 10. Format za parsiranje datuma

SQL upit iz prethodne metode se može modificirati tako da sadrži parametar koji se odnosi na *id* osobe. Na taj način, moguće je uraditi vraćanje osobe po njenom *id*-u. Ovo zahtijeva proširenje upita tako da on sadrži *WHERE* klauzu. Također, za razliku od prethodne metode, ova metoda sadrži parametar, koji predstavlja *id* osobe koja se pretražuje. Programski kod metode *dajOsobuPoId()* je prikazan u Listingu 11.

```
public static Osoba dajOsobuPoId(Integer id) {
    Osoba osoba = null;
    String upit = "SELECT * FROM Osoba WHERE id = ?";

    try (Connection conn = connect();
         PreparedStatement pstmt = conn.prepareStatement(upit)) {

        pstmt.setInt(1, id);
        ResultSet rs = pstmt.executeQuery();

        if (rs.next()) {
            osoba = new Osoba(
                    rs.getInt("id"),
                    rs.getString("ime"),
                    rs.getString("prezime"),
                    rs.getString("adresa"),
```

```
OsobaModel.dateFormat.parse(rs.getString("datumRodjenja")),
                    rs.getString("maticniBroj"),
                    Uloga.valueOf(rs.getString("uloga"))
            );
        }
    } catch (SQLException e) {
        System.out.println(e.getMessage());
    } catch (ParseException e) {
        System.out.println(e.getMessage());
    }
    return osoba;
}
```

Listing 11. Java metoda za pregled podataka iz SQL tabele s parametrom

Inicijalno se varijabla *osoba* postavlja na *null*. Ukoliko osoba sa navedenim *id*-em postoji, onda će se varijabla *osoba* promijeniti tako da sadrži podatke koji odgovaraju osobi vraćenoj iz baze podataka. Međutim, ako osoba nije pronađena, njena vrijednost će ostati *null* i kao takva će se vratiti iz metode.

Metoda za ažuriranje osobe se naziva *azurirajOsobu()* i ažurira podatke o osobi, čiji je *id* jednak *id*-u koji se proslijedi metodi kao parametar. Za svaki parametar, metoda provjerava da li je jednak *null*. Ako jeste, onda se smatra da nije potrebno raditi ažuriranje te vrijednosti, i samim time se izostavlja dio koji se odnosi na ažuriranje tog polja u tabeli *Osoba* u bazi podataka. Početak upita za ažuriranje osobe u bazi podataka je smješten u varijabli *upit*, koja je tipa *StringBuilder*. Iako se na ovom mjestu mogao koristiti i uobičajeni *String*, postoji nekoliko prednosti koje posjeduje *StringBuilder* a koje ga čine boljim izborom u situaciji kada se upiti dinamički generišu. Prilikom generisanja dinamičkih upita neophodno je raditi nadovezivanje (eng. *concatenate*) više stringova, što se sa *String* klasom radi upotrebom operatora “+”. Ovaj pristup često dovodi do grešaka, sporiji je, teži za održavanje i proizvodi ogramski kod koji je manje čitljiv. Za razliku od ovog pristupa, korištenje *StringBuilder* je efikasnije, omogućava jasnije rukovanje formatiranjem i generalno je fleksibilnije za dinamičko kreiranje upita. *StringBuilder* pruža metode poput:

- *append()* - dodaje novi string na kraj.
- *insert()* - ubacuje string na određenu poziciju.
- *replace()* - mijenja određeni dio stringa.
- *setLength()* - smanjuje ili povećava dužinu stringa.

Pomoću ovih metoda se mogu izbjeći greške u formatiranju, koje mogu nastati u raznim situacijama, poput viška zareza ili zaboravljenih razmaka između dijelova stringa koji predstavlja upit. Listing 12 prikazuje primjer korištenja *String* prilikom pisanja upita i grešaka koje mogu nastati, dok Listing 13 prikazuje kako se ove greške mogu otkloniti korištenjem klase *StringBuilder*.

```
String upit = "UPDATE Osoba SET ime = 'Neko', ";
upit += "prezime = 'Nekic'"; // Zaboravljen razmak
upit += " WHERE id = 1"; // SQL greska zbog formatiranja
```

Listing 12. Formiranje SQL upita bez formatiranja

```
StringBuilder upit = new StringBuilder("UPDATE Osoba SET ");
upit.append("ime = 'Neko', ");
upit.append("prezime = 'Nekic' ");
upit.append("WHERE id = 1");
```

Listing 13. Formiranje SQL upita sa formatiranjem

U metodi *azurirajOsobu()*, čiji je prvi dio prikazan u Listingu 14, varijabla *imaPromjene* čuva informacije o tome da li se desila promjena barem jednog polja. Ukoliko nije, nema potrebe raditi povezivanje na bazu podataka i to je situacija u kojoj metoda vraća poruku "Sva polja su ista kao i prije!". U listi *parametri*, čuvaju se vrijednosti svih polja koja će se ažurirati u bazi podataka. Na osnovu parametara, metoda *azurirajOsobu()* za svaki od njih koji nije *null* i koji nije prazan, dodaje odgovarajući string u *upit* metodom *append*, dodaje vrijednost tog parametra u listu parametri i postavlja *imaPromjene* na *true*, kako bi se znalo da se upit treba izvršiti nad bazom podataka i da postoje polja koja su ažurirana. Nakon što se kreira string koji predstavlja upit, pozivom metode *delete* nad varijablom *upit* se uklanjaju posljednja dva znaka (tj. znak zareza i znak razmaka) kako bi upit imao ispravan oblik. Inače, ova metoda prima dvije vrijednosti tipa *int* i briše znakove koji se nalaze na indeksima između te dvije vrijednosti. Prvi parametar predstavlja početni indeks (uključivo) od kojeg treba započeti brisanje u stringu. Karakter na ovoj poziciji će biti uklonjen zajedno sa svim karakterima između njega i drugog parametra. Drugi parametar predstavlja krajnji indeks (isključivo) do kojeg će se vršiti brisanje. Karakter na ovom indeksu neće biti uklonjen, već se brisanje zaustavlja neposredno prije njega. Konačno, u upit se dodaje i *WHERE* klauza sa parametrom *id*. Parametar *id* ima vrijednost koja je proslijeđena metodi, tako da je tu vrijednost potrebno dodati u listu *parametri*.

```
public static String azurirajOsobu(Integer id, String novoIme,
String novoPrezime, String novaAdresa, Date noviDatumRodjenja,
String noviMaticniBroj, Uloga novaUloga) {
   StringBuilder upit = new StringBuilder("UPDATE Osoba SET ");
   boolean imaPromjene = false;

   // lista koja cuva parametre
   List<Object> parametri = new ArrayList<>();

   // provjera vrijednosti pojedinih polja (da li su prazna ili ne)
   if (novoIme != null && !novoIme.isEmpty()) {
       upit.append("ime = ?, ");
```

```
            parametri.add(novoIme);
            imaPromjene = true;
        }
        if (novoPrezime != null && !novoPrezime.isEmpty()) {
            upit.append("prezime = ?, ");
            parametri.add(novoPrezime);
            imaPromjene = true;
        }
        if (novaAdresa != null && !novaAdresa.isEmpty()) {
            upit.append("adresa = ?, ");
            parametri.add(novaAdresa);
            imaPromjene = true;
        }
        if (noviDatumRodjenja != null) {
            upit.append("datumRodjenja = ?, ");
                parametri.add(dateFormat.format(noviDatumRodjenja));    //
Format the date
            imaPromjene = true;
        }
        if (noviMaticniBroj != null && !noviMaticniBroj.isEmpty()) {
            upit.append("maticniBroj = ?, ");
            parametri.add(noviMaticniBroj);
            imaPromjene = true;
        }
        if (novaUloga != null) {
            upit.append("uloga = ?, ");
            parametri.add(novaUloga.name());
            imaPromjene = true;
        }

        // izadji ranije ako nema polja za azuriranje
        if (!imaPromjene) {
            return "Sva polja su ista kao i prije!";
        }

        // uklanjanje zareza na kraju upita i razmaka iz SQL upita
        upit.delete(upit.length() - 2, upit.length());
        upit.append(" WHERE id = ?");

        // dodaj id kao parametar
        parametri.add(id);
```

Listing 14. Provjera da li se ažuriranje može izvršiti

Drugi dio metode *azurirajOsobu()* prikazan je u Listingu 15. Nakon uspostavljanja konekcije sa bazom podataka kroz *conn*, varijabla *upit* se pretvara u *String* i koristi se za kreiranje *PreparedStatement* objekta. Preostalo je još dodati odgovarajuće parametre upitu koji se treba izvršiti nad bazom podataka. Vodite računa da se vrijednosti parametara moraju navoditi u istom redoslijedu u kojem se navode u stringu upita. Pri tome, indeksacija parametara kreće od 1. Metodom *setObject()* se parametar postavlja na odgovarajuću vrijednost. Prilikom poziva ove metode, najprije se navodi pozicija parametra kao *int* vrijednost a zatim se navodi njegova vrijednost. U metodi *azurirajOsobu()*, ove akcije su obavljene unutar *for* petlje. Poziv *executeUpdate()* metode vraća broj promijenjenih redova kao *int*. Ukoliko je broj promijenjenih redova veći od 0, to znači da je uspješno završeno ažuriranje navedene osobe. U suprotnom, osoba sa navedenim *id*-em ne postoji u bazi podataka.

```
    try (Connection conn = connect(); PreparedStatement pstmt =
conn.prepareStatement(upit.toString())) {

        // dodavanje parameatara u PreparedStatement
        for (int i = 0; i < parametri.size(); i++) {
            pstmt.setObject(i + 1, parametri.get(i));
        }

        int promijenjeniRedovi = pstmt.executeUpdate();
        if (promijenjeniRedovi > 0) {
            return "Osoba je uspjesno azurirana";
        } else {
            return "Ne postoji osoba sa datim id-em";
        }
    } catch (SQLException e) {
        return e.getMessage();
    }
}
```

Listing 15. Ažuriranje sadržaja SQL tabele kroz Java metodu

9.5. Singleton šablon dizajna

U prethodnim dijelovima laboratorijske vježbe prikazane su sve CRUD operacije nad bazom podataka. Ipak, klasu *OsobaModel* treba izmijeniti u odnosu na onu koja je prikazana u laboratorijskoj vježbi 8 na način da ona prati *Singleton* šablon dizajna. *Singleton* je šablon dizajna koji osigurava da postoji samo jedna instanca klase tokom izvršavanja aplikacije i da ta instanca bude dostupna globalno kroz statičku metodu. Ta se metoda obično obično naziva *getInstance()*. Detalji ovog šablona dizajna neće biti analizirani, ali je bitno zapamtiti da se ovaj pristup koristi kako bi se osigurala jedna globalna instanca *OsobaModel* klase i kako bi se omogućila konzistentnost podataka i jednostavan pristup modelu iz bilo kojeg dijela aplikacije. Bez upotrebe *Singleton* šablona dizajna, svaki kontroler bi mogao kreirati svoju instancu

modela, što bi značilo da oni upravljaju različitim podacima. Zbog svega navedenog, klasu *OsobaModel* treba modificirati tako da ima statički atribut koji predstavlja njenu instancu. Dodatno, metoda *getInstance()* kreira instancu klase *OsobaModel*, ako je ona *null* i vraća je kao povratnu vrijednost, dok metoda *removeInstance* postavlja instancu na *null*. Implementacija ovih metoda je prikazana u Listingu 16.

```
private static OsobaModel instance = null;

public static OsobaModel getInstance() {
    if (instance == null) {
        instance = new OsobaModel();
    }
    return instance;
}

public static void removeInstance() {
    instance = null;
}
```

Listing 16. Primjena Singleton šablona dizajna u MVC Java aplikaciji

Važno je napomenuti da ova klasa ne smije imati javni konstruktor. Ako je konstruktor javni, to bi značilo da ništa ne spriječava neki dio aplikacije da kreira novu instancu klase pomoću *new OsobaModel()*. Ovo narušava princip *Singleton*-a. Dakle, ako je konstruktor neophodan, mora biti privatni. Na ovaj način su završene modifikacije u model komponenti MVC JavaFX aplikacije.

View komponentu nije potrebno mijenjati u odnosu na laboratorijsku vježbu 8. Međutim, kontroler komponentu treba modificirati tako da koristi podatke preuzete iz baze podataka. Pomoćna metoda *ucitajOsobeIzBaze()* se koristi za učitavanje podataka o osobama iz kreirane baze podataka (Listing 17).

```
private void ucitajOsobeIzBaze() {
    List<Osoba> osobe = OsobaModel.dajSveOsobe();
    osobeObservableList.setAll(osobe);
}
```

Listing 17. Korištenje baze podataka u okviru MVC kontrolera

Metoda *initialize()* poziva metode za kreiranje tabele ako ona već ne postoji, pražnjenje tabele *Osoba* i popunjavanje ove tabele inicijalnim podacima. Nakon toga se učitavaju sve osobe iz baze podataka pozivom pomoćne metode *ucitajOsobeIzBaze*. Ostatak ove metode je isti kao na laboratorijskoj vježbi 8. Kompletan programski kod metode *initialize()* kontrolera *OsobaController* je prikazan u Listingu 18.

```
@FXML
public void initialize() {
   OsobaModel.kreirajTabeluAkoNePostoji();
   OsobaModel.isprazniTabeluOsoba();
   OsobaModel.napuniInicijalnimPodacima();
   ucitavanjeLabel.setText("Ucitani podaci");
   ucitavanjeLabel.setStyle("-fx-background-color: green;");

   azurirajOsobuButton.setText("Azuriraj");

                       ulogaChoiceBox.getItems().addAll(Uloga.STUDENT,
Uloga.NASTAVNO_OSOBLJE);

   ucitajOsobeIzBaze();
   osobeListView.setItems(osobeObservableList);

   // dodavanje listener-a za klik dugmeta
    azurirajOsobuButton.setOnAction(new EventHandler<ActionEvent>()
{
       @Override
       public void handle(ActionEvent event) {
           azurirajOsobu();
       }
   });

   // dodavanje listener-a za izbor osobe iz listview

osobeListView.getSelectionModel().selectedItemProperty().addListene
r((observable, starVrijednost, novaVrijednost) -> {
       if (novaVrijednost != null) {
            izabranaOsoba = novaVrijednost;  // azuriranje varijable
koja predstavlja trenutno izabranu osobu
                ispuniPolja(novaVrijednost);   // ispunjavanje polja
detaljima izabrane osobe
               porukaLabel.setVisible(false); // sakrij labelu koja
sadrzi poruku
       }
   });
}
```

Listing 18. Inicijalizacija MVC kontrolera koja koristi metode za rad sa bazom podataka

Osim ovih modifikacija, u metodi *azurirajOsobu()* kontrolera, potrebno je dodati poziv metode *ucitajOsobeIzBaze()* prije poziva *osobeListView.refresh()* u posljednoj liniji koda ove metode. Konačno, metodu *start()* unutar *HelloApplication* klase treba promijeniti tako da se više ne

poziva konstruktor klase *OsobaModel*, jer javni konstruktor više ne postoji. Ovo je prikazano u Listingu 19.

```
OsobaModel osobaModel = OsobaModel.getInstance();
```

Listing 19. Upotreba Singleton šablona dizajna u MVC kontroleru

Aplikaciju je sada moguće pokrenuti na isti način kao na laboratorijskoj vježbi 8. Izvršavanje aplikacije proizvodi ispis kao na Slici 9.

```
Tabela je kreirana ili vec postoji!
Obrisani redovi tabele. Broj obrisanih redova: 2
Ubaceni pocetni podaci!
```

Slika 9. Konzolni ispis nakon pokretanja aplikacije

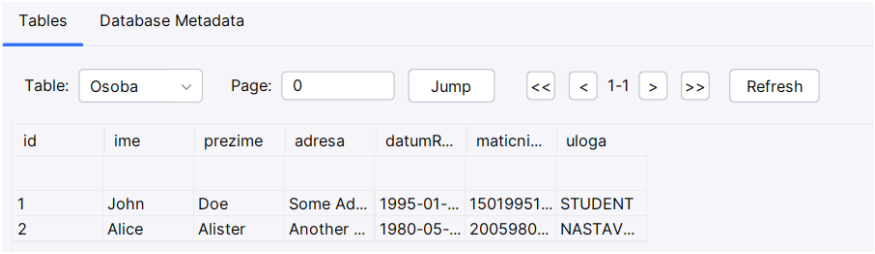
Neobično ponašanje: Nakon pokretanja aplikacije, izbora osobe iz liste, promjene određenih polja i ažuriranja klikom na dugme “Azuriraj”, ne prikazuje se labela sa porukom da je osoba uspješno ažurirana. Ponovnim klikom na dugme, poruka se ispisuje. Da li je ovo slučajnost, *bug* ili očekivano ponašanje? U ovoj situaciji dolazi do neobičnog ponašanja zbog načina na koji JavaFX vrši prikaz komponenti (tzv. *rendering*). Naime, JavaFX koristi jednonitni model (više detalja o korištenjima principa višenitnosti će biti opisano u sklopu laboratorijske vježbe 10). Dovoljno je znati da poziv *model.azurirajOsobu()* nakon kojeg slijedi poziv *ucitajOsobeIzBaze()* i *refresh()*, predstavljaju “skupe” operacije (posebno zbog toga što *ucitajOsobeIzBaze()* komunicira sa bazom podataka). Sve ove operacije izvršavaju se u jednoj JavaFX niti, što blokira ažuriranje grafičkog korisničkog interfejsa dok se operacije ne završe. Zbog ovoga, JavaFX odlaže *rendering* (tj. prikaz) vrijednosti labele *porukaLabel*. Tek kada se nit oslobodi, JavaFX prikazuje promjene. Ovo se može otkloniti korištenjem isječka koda prikazanog u Listingu 20. Korištenjem metode *Platform.runLater()* se osigurava da se ažuriranje elemenata grafičkog korisničkog interfejsa dešava nakon trenutnih operacija, ali prije *renderinga* “skupih” operacija.

```
Platform.runLater(() -> {
   porukaLabel.setVisible(true);
   porukaLabel.setText(poruka);
});
```

Listing 20. Poboljšavanje korisničkog iskustva utjecajem na redoslijed rendering operacija

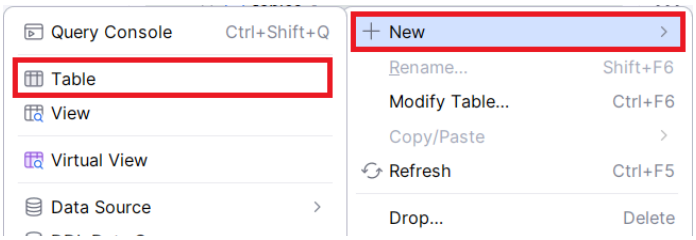
9.6. Pregled baze podataka kroz okruženje

Sve informacije o bazi podataka moguće je pregledati putem datoteke koja predstavlja bazu podataka. Dvostrukim klikom na datoteku *baza.db* otvara se prozor kao na Slici 10.



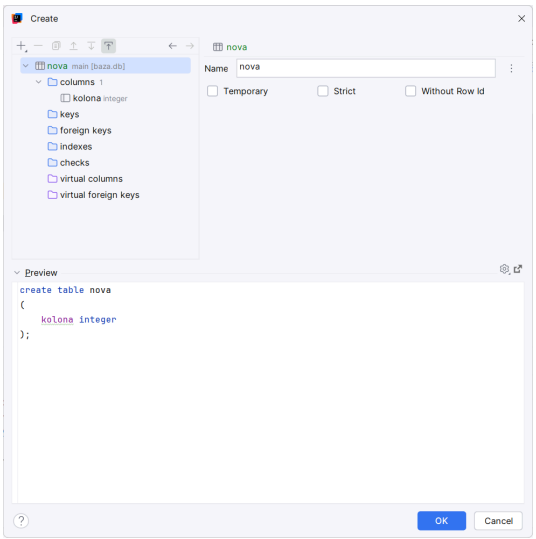
Slika 10. Prikaz tabela i podataka u bazi podataka baza.db

Sve operacije koje se tiču dodavanja tabela, pisanja upita i slično, moguće je realizirati i kroz *Database* prozor. Naravno, određene funkcionalnosti zahtijevaju interakciju korisnika sa grafičkim korisničkim interfejsom i u tim situacijama nije moguće raditi isključivo kroz vanjske skripte ili *plugin*-e u sklopu okruženja, već se te promjene moraju zapisati u programskom kodu. Primjer dodavanja nove tabele kroz *Database* prozor će ukratko biti opisan u nastavku. Desnim klikom na dio označen kao *tables*, otvaraju se razne opcije a jedna od njih je *New →Table* (Slika 11).



Slika 11. Kreiranje nove tabele u bazi podataka kroz Database prozor

Nakon što se otvori prozor za kreiranje nove tabele, istoj je potrebno dati naziv. Za dodavanje kolona, potrebno je kliknuti na znak plus (“+”) u gornjem lijevom uglu prozora. Pokazna tabela naziva “nova” sa kolonom “kolona”, koja je tipa *integer* je prikazana na Slici 12.



Slika 12. Kreiranje tabela i kolona kroz okruženje

Slično, izborom opcije *Query Console* (desni klik na *tables* → *New* → *Query console*), otvara se konzola za pisanje upita u kojoj je moguće zapisivati i izvršavati pojedine upite nad bazom podataka.

9.7. Zadaci za samostalni rad

Za sticanje bodova na prisustvo laboratorijskoj vježbi, potrebno je uraditi sljedeće zadatke tokom laboratorijske vježbe i postaviti ih u repozitorij studenta na odgovarajući način:

Zadatak 1.

U OsobaModel dodati metodu *obrisiOsobuPoId(Integer id)*, koja iz baze podataka briše osobu čiji je id naveden kao parametar metode. Metoda vraća *String*, koji predstavlja poruku o uspješnosti brisanja. Ako se desi neki izuzetak, onda metoda vraća poruku tog izuzetka. Ukoliko osoba sa navedenim id-em ne postoji, onda se vraća poruka "Ne postoji osoba sa datim id-em", dok ukoliko je osoba obrisana, vraća se poruka "Osoba je uspjesno obrisana".

U metodi *azurirajOsobu()* dodajte provjeru da li je dužina imena i ispravna i da li se matični broj poklapa sa datumom rođenja. Tek kada su ti uslovi ispunjeni, moguće je raditi ažuriranje navedenih polja. Situaciju kada neki od ovih uslova nije ispunjen tretirajte kao da je odgovarajuće polje postavljeno na *null*, odnosno kao da korisnik ne želi modificirati to polje.

Zadatak 2.

Po uzoru na implementaciju kojom su se MVC komponente spojile sa bazom podataka i tabelom za osobu, uraditi implementaciju odgovarajućih komponenti za predmet. Kao polazni programski kod iskoristite implementaciju MVC JavaFX aplikacije sa laboratorijske vježbe 8. Slično kao *OsobaModel*, klasa *PredmetModel* omogućava dohvaćanje svih predmeta, dohvaćanje predmeta po id-u, ažuriranje predmeta i brisanje predmeta po id-u. *View* komponentu ne morate modificirati u odnosu na polazni programski kod. Dovoljno je da je ispravno povežete da sada radi sa bazom podataka. Ukoliko vam ostane vremena, dodajte novi *View* koji omogućava izbor id-a predmeta iz neke vrste liste ili padajućeg menija (komponentu koju ćete koristiti izaberite po vlastitoj želji). Nakon što korisnik izabere id predmeta, u odgovarajućim poljima prikažite informacije o izabranom predmetu. Također, probajte i slučaj kada korisnik ne bira id iz padajućeg menija, nego id predmeta ukucava u polje za unos ispod kojeg se nalazi dugme. Klikom na to dugme, pretražuje se predmet po id-u koji je korisnik ukucao i u odgovarajućim poljima se prikazuju informacije o izabranom predmetu (isto kao prethodno).