Технически университет – Варна
Катедра „Софтуерни и Интернет Технологии“

Курсов проект по ООП-2
Задание №4: Билетен център

Изготвен от:

Ангел Генчев Проданов

Специалност: СИТ
Курс: 3
Ф.н.: 18621668
Група: 1б

Мартин Димитров Иванов

Специалност: СИТ
Курс: 3
Ф.н.: 18621669
Група: 1б

Съдържание:

Описание на заданието:
№4 Билетен център

Да се разработи информационна система предоставяща услуга билетен център. Програмата съхранява и обработва данни за разпространение на билети. Системата позволява множествен достъп.

Системата поддържа два вида потребители администратор и клиенти (организатор, разпространител) с различни роли за достъп до функционалностите в системата.

Операции за работа с потребители:

- Създаване на организатори от администратор;
- Създаване на разпространители от администратор;
- Създаване, редактиране на събития с избор на един или списък разпространители от собственик за продаване на билети
- Поддържане на профили с характеристики на организатори и разпространители (хонорар и др…)
- Рейтинговане на разпространителите

Системата поддържа операции за работа със събития:

- Добавяне на ново събитие от организатор (вид на събитието, брой места, видове места, цена на билетите по видове, ограничение в закупуването на билет от едно лице и др…);
- Продаване на билети за събитие от разпространител, създаване на формуляр за закупуване (информация за купувача, избор на място и др…);

Системата поддържа справки по произволен период за:

- Разпространител (разпродадени билети от различни видове събития, данните на разпространителя, рейтинг)
- Събития (дата, статус, местоположение и др…);

Организатор на събитие достъпва справки само за събитията, на който е организатор.
Организатора достъпва справки за всички организирани от него събития.
Разпространител има право на справки за събития които са му отредени.
Системата поддържа Известия за събития:

- Новопостъпила заявка за събитие (в профила на разпространител);
- Периодично уведомление за продадени билети от събитие (в профила на собственика);
- Наближаващо събитие с не продадени билети (собственик, разпространител на билети)

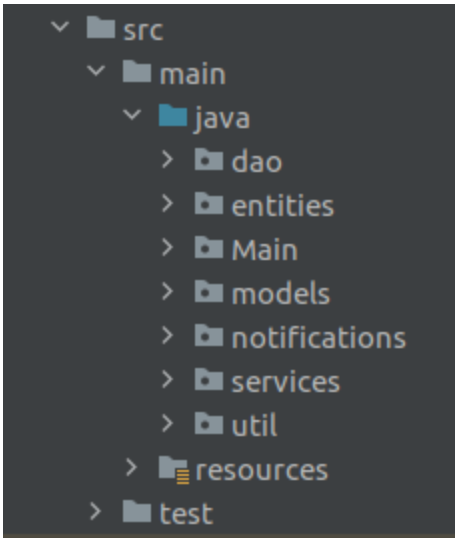
Анализ на проблема:

1. Функционални изисквания
    a. Осъществяване на записването, промяната и изкарването на информация с употребата на база от данни, с помощта на лесна за употреба потребителска графична среда.
    b. Достъпа до информация да бъде ограничен, за да не се позволява появяването на непълна или грешна информация в базата от данни, затова достъпът до нея ще бъде осъществен чрез точно определени функции.
    c. Връзки:
        i. Всеки акаунт има роля и е свързан с User
        ii. Връзка дистрибутор - събитие е много към много и изисква трета таблица Distribution.
        iii. При изтриването на някои редове от таблицата, те трябва да се отделят от свързаните с тях други данни. Други трябва да изтрият всичко свързано с тях.
    d. Справки:
        i. Организатора достъпва справки за всички разпространители на своите събития.
        ii. Разпространителя достъпва справки за всички събития.
    e. Известия:
        i. Трябва да се изпълняват през известен период от време на заден план, без да прекъсват работата на програмата.
        ii. За да информираме потребителя, е нужно да се свържат с нишката, която управлява графичната част на програмата.

2. Структура на проекта

Така като проекта е на Maven , първо се изтеглят външни библиотеки , чрез pom.xml файла , за да можем да работим с Hibernate, MySqlConnector за създаване и свързване с базата данни, също така и JavaFX за графичния интерфейс.

Това са всички модули на файловете в проекта.



3. Дефиниция на модулите на системата

- Main
  - Java
    - Dao - всички Data Access Обекти, с чиито функции достъпваме съответните таблици в базата от данни
    - Entities - Всички entity-та които използваме. Всяко едно репрезентира съответната си таблица от DB
    - Main - Местоположението на Main файла и контролерите.
      - Controllers – Контролерите. Всеки FXML файл си има контролер.
    - Models - Моделите в които записваме информация при правенето на заявки
    - Notifications - За класовете на известията които се изпълняват през определен период от време
    - Services - за Service класовете, с които се свързваме към съответното DAO
    - Util - за HIbernateUtil файла
  - Resources
    - Images - папка с изображенията които използвахме (Създадени набързо от Ангел)
    - Scenes – папка с всички fxml файлове които зареждаме в проекта си.
- Test
  - Java - Тук се намират класовете които използваме за да правим Тестове

Проектиране на системата:

1. Проектиране на отделните модули
    a. база данни

Базата данни е свързана с проекта ни чрез Hibernate. При липсата на схема, я създава.

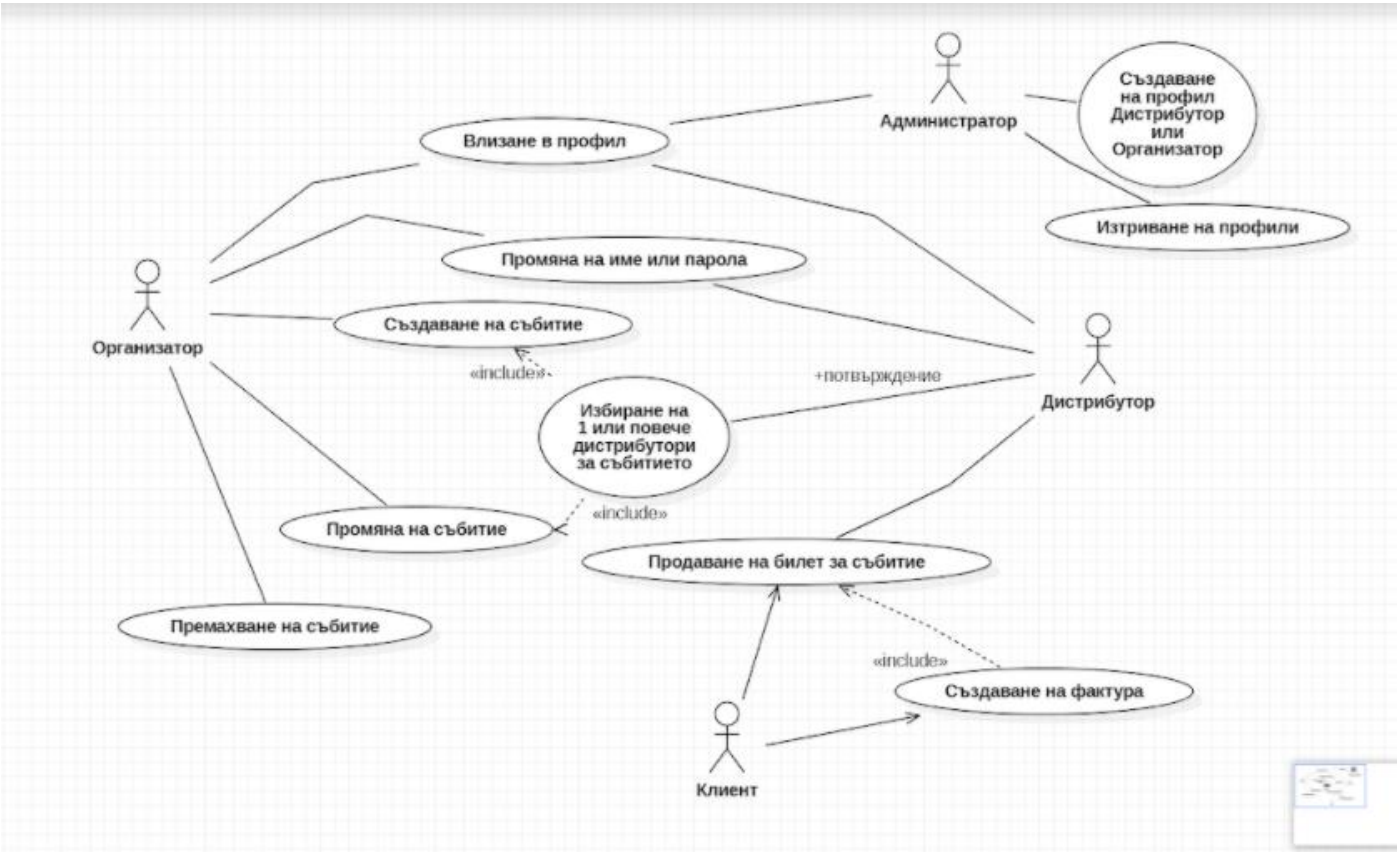
    b. GUI

Крайният потребител използва графичният интерфейс предоставен от FXML файловете и при различни действия, се извикват различни функции от класовете контролери.

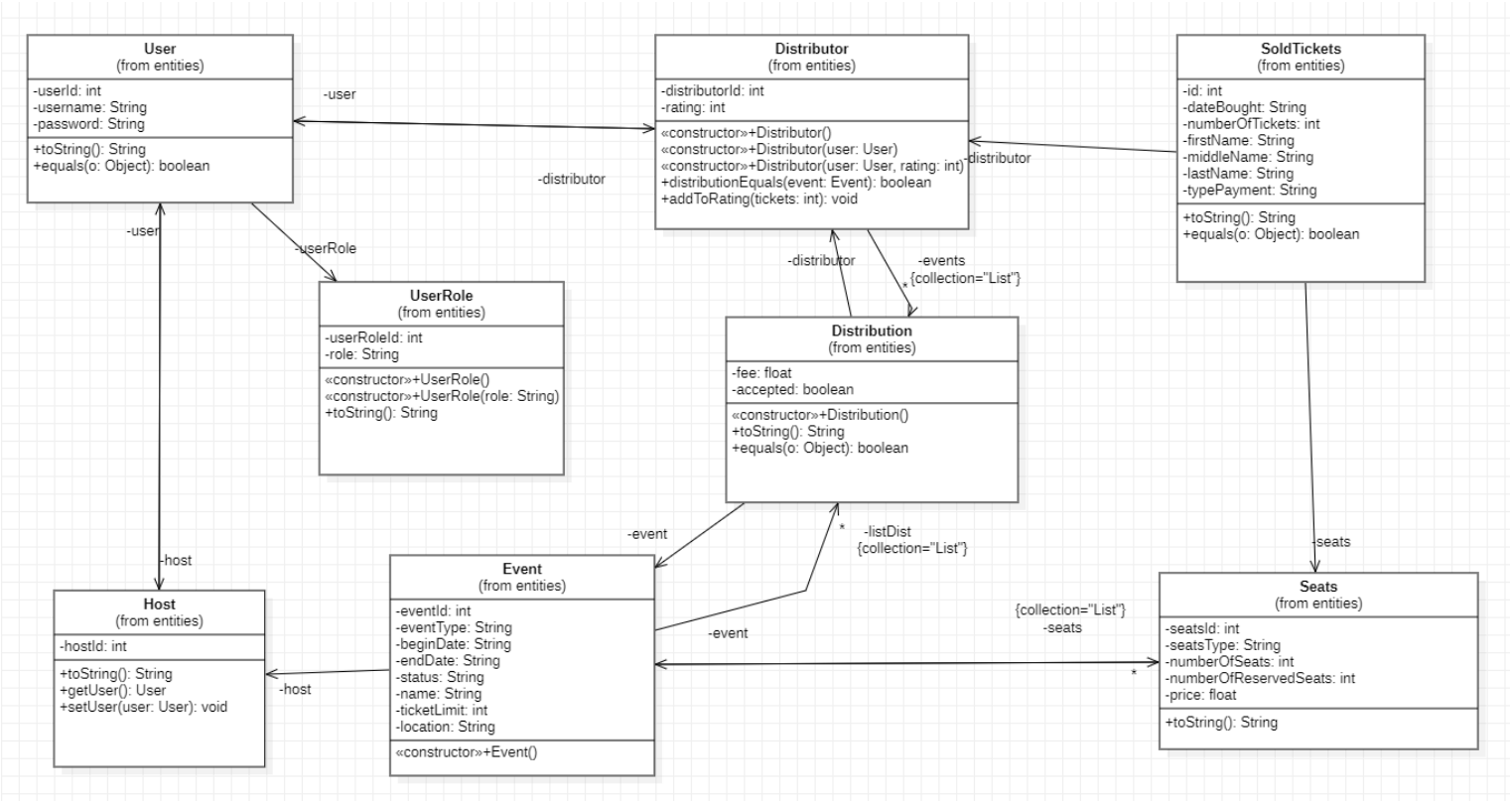
    c. бизнес логика

Осъществява се чрез DAO-pattern манипулацията на данните в базата, чрез сцените в графичния интерфейс.

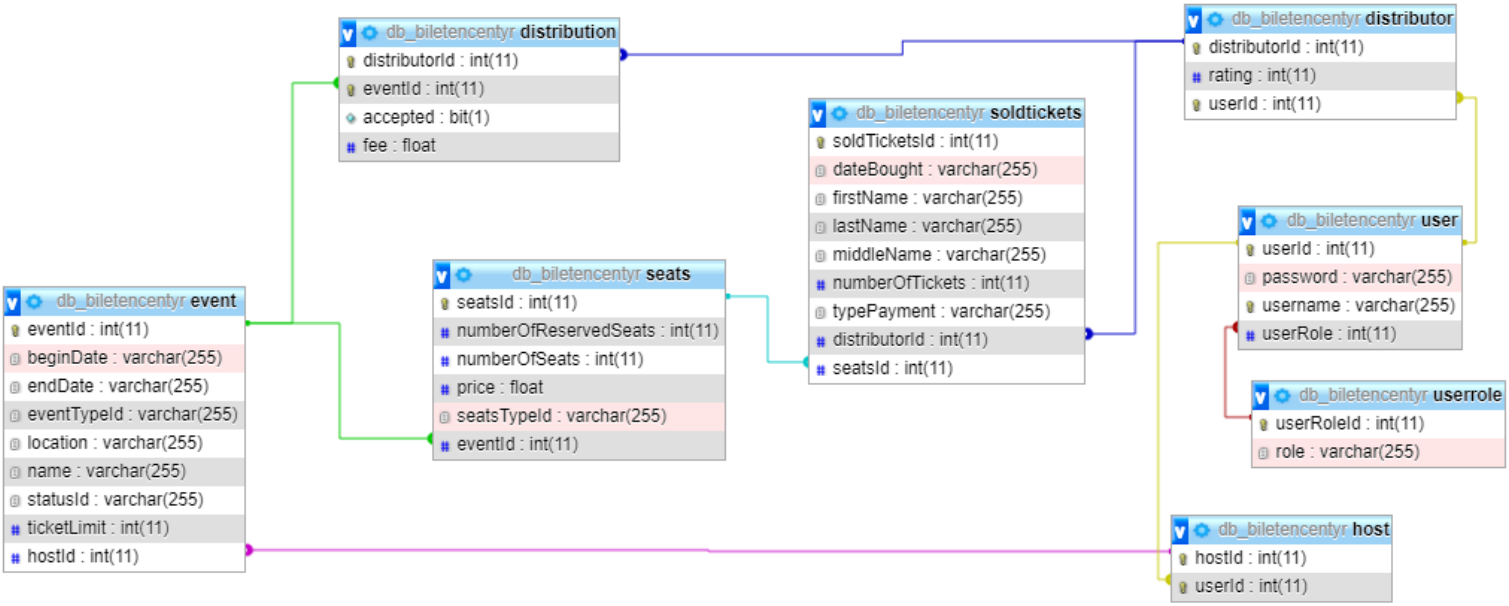
2. UML
    a. Use Case



b. Class Diagram



3. Концептуален модел на базата от данни (ER диаграма)



Реализация на проекта:

1. Реализация на базата от данни:

Класа HibernateUtil служи за създаването и свързването към базата данни. В него се задават и настройките на Hibernate вместо да предоставяме hibernate.cfg.xml файл за създаването на сесия. Тук задаваме конфигурацията за връзката към MySQL сървъра с user “root” и парола “1234”.

```
public class HibernateUtil {

   private static SessionFactory sessionFactory;

   public static SessionFactory getSessionFactory() {
      if (sessionFactory == null) {
         try {
            Configuration configuration = new Configuration();

            Properties settings = new Properties();

            settings.put(Environment.DRIVER, "com.mysql.cj.jdbc.Driver");
            settings.put(Environment.URL,
"jdbc:mysql://localhost:3306/db_biletencentyr?useSSL=false&createDatabaseIfN
otExist=true&serverTimezone=UTC&useLegacyDatetimeCode=false");
            settings.put(Environment.USER, "root");
            settings.put(Environment.PASS, "1234");
            settings.put(Environment.DIALECT,
"org.hibernate.dialect.MySQL57InnoDBDialect");
            settings.put(Environment.SHOW_SQL, "true");
            settings.put(Environment.CURRENT_SESSION_CONTEXT_CLASS,
"thread");
            settings.put(Environment.HBM2DDL_AUTO, "update");

            configuration.setProperties(settings);

            configuration.addAnnotatedClass(Distributor.class);
```

```
            configuration.addAnnotatedClass(Distribution.class);
            configuration.addAnnotatedClass(User.class);
            configuration.addAnnotatedClass(UserRole.class);
            configuration.addAnnotatedClass(Seats.class);
            configuration.addAnnotatedClass(SoldTickets.class);
            configuration.addAnnotatedClass(Host.class);
            configuration.addAnnotatedClass(Event.class);
            configuration.addAnnotatedClass(Distribution.class);

            ServiceRegistry serviceRegistry = new StandardServiceRegistryBuilder()
                  .applySettings(configuration.getProperties()).build();
            sessionFactory = configuration.buildSessionFactory(serviceRegistry);
          } catch (Exception e) {
            e.printStackTrace();
          }

        }
        return sessionFactory;
      }

}
```

Задаваме настройки за конфигурацията на Hibernate чрез settings.put(). Чрез Environment.URL подаваме адреса на сървъра и името на схемата както и дали да я създаде ако не съществува, а със Environment.USER и Environment.PASS , името и паролата. ("jdbc:mysql://localhost:3306/db_biletencentyr?useSSL=false&createDatabaseIfNotExist=true&serverTimezone=UTC&useLegacyDatetimeCode=false")

Ако искаме да изтрива и да създава нови таблици всеки път при изпълнение на програмата, може да променим Environment.HBM2DDL_AUTO от “update” на “create-drop”.

Към конфигурацията добавяме entity-класове , които ще съдържат и създават всичките ни данни в таблиците, чрез addAnnotatedClass().

Задаваме един entity-клас по следния начин:

```
@Entity
```

```
@Table(name = "user")
public class User {
   @Id
   @GeneratedValue(strategy = GenerationType.IDENTITY)
   @Column(name = "userId", unique = true, nullable = false)
   private int userId;

   @Column(name = "username", unique = true, nullable = false)
   private String username;

   @Column(name = "password", nullable = false)
   private String password;

   @OneToOne(mappedBy = "user", cascade = CascadeType.ALL, orphanRemoval
= true)
   private Host host;

   @OneToOne(mappedBy = "user", cascade = CascadeType.ALL, orphanRemoval
= true)
   private Distributor distributor;

   @ManyToOne
   @JoinColumn(name = "userRole")
   private UserRole userRole;
}
```

С анотация @Entity оказваме дадения клас , че се съдържа в базата данни. @Table името на таблицата. @Id оказва primary key-а на дадената таблица. @GeneratedValue - там въвеждаме начина на генериране на колоната (в дадения случай е autoincrement). @Column оказва връзката с колоната. Различните релации се обозначават с анотациите @OneToOne , @OneToMany , @ManyToOne и @ManyToMany спрямо нужната връзка с даденото поле от данни.

При стартирането на програмата може да видим в конзолата съобщение от Hibernate относно връзката с базата данни.

```
дек 13, 2020 1:32:36 ПР.ОБ. org.hibernate.Version logVersion
INFO: HHH000412: Hibernate Core {5.3.7.Final}
дек 13, 2020 1:32:36 ПР.ОБ. org.hibernate.cfg.Environment <clinit>
INFO: HHH000206: hibernate.properties not found
дек 13, 2020 1:32:37 ПР.ОБ. org.hibernate.annotations.common.reflection.java.JavaReflectionManager <clinit>
INFO: HCANN000001: Hibernate Commons Annotations {5.0.4.Final}
дек 13, 2020 1:32:39 ПР.ОБ. org.hibernate.engine.jdbc.connections.internal.DriverManagerConnectionProviderImpl configure
WARN: HHH10001002: Using Hibernate built-in connection pool (not for production use!)
дек 13, 2020 1:32:39 ПР.ОБ. org.hibernate.engine.jdbc.connections.internal.DriverManagerConnectionProviderImpl buildCreator
INFO: HHH10001005: using driver [com.mysql.cj.jdbc.Driver] at URL [jdbc:mysql://localhost:3306/db_biletencentyr?useSSL=false&createDatabaseIfNotExist
дек 13, 2020 1:32:39 ПР.ОБ. org.hibernate.engine.jdbc.connections.internal.DriverManagerConnectionProviderImpl buildCreator
INFO: HHH10001001: Connection properties: {password=****, user=root}
дек 13, 2020 1:32:39 ПР.ОБ. org.hibernate.engine.jdbc.connections.internal.DriverManagerConnectionProviderImpl buildCreator
INFO: HHH10001003: Autocommit mode: false
дек 13, 2020 1:32:39 ПР.ОБ. org.hibernate.engine.jdbc.connections.internal.DriverManagerConnectionProviderImpl$PooledConnections <init>
INFO: HHH000115: Hibernate connection pool size: 20 (min=1)
дек 13, 2020 1:32:40 ПР.ОБ. org.hibernate.dialect.Dialect <init>
INFO: HHH000400: Using dialect: org.hibernate.dialect.MySQL57InnoDBDialect
дек 13, 2020 1:32:48 ПР.ОБ. org.hibernate.resource.transaction.backend.jdbc.internal.DdlTransactionIsolatorNonJtaImpl getIsolatedConnection
INFO: HHH10001501: Connection obtained from JdbcConnectionAccess [org.hibernate.engine.jdbc.env.internal.JdbcEnvironmentInitiator$ConnectionProviderJ
```

Ако няма готови таблици , то ги създава заедно с техните релации.

```
Hibernate: create table distribution (distributorId integer not null, eventId integer not null, accepted bit, fee float, primary key (distributorId, eve
Hibernate: create table distributor (distributorId integer not null auto_increment, rating integer, userId integer, primary key (distributorId)) engine=
Hibernate: create table event (eventId integer not null auto_increment, beginDate varchar(255), endDate varchar(255), eventTypeId varchar(255) not null,
Hibernate: create table host (hostId integer not null auto_increment, userId integer, primary key (hostId)) engine=InnoDB
Hibernate: create table seats (seatsId integer not null auto_increment, numberOfReservedSeats integer, numberOfSeats integer, price float, seatsTypeId v
Hibernate: create table soldtickets (soldTicketsId integer not null auto_increment, dateBought varchar(255), firstName varchar(255), lastName varchar(25
Hibernate: create table user (userId integer not null auto_increment, password varchar(255) not null, username varchar(255) not null, userRole integer,
Hibernate: create table userRole (userRoleId integer not null auto_increment, role varchar(255) not null, primary key (userRoleId)) engine=InnoDB
Hibernate: alter table distributor drop index UK_m53pguuuk2eree5uyul6yiljf
Hibernate: alter table distributor add constraint UK_m53pguuuk2eree5uyul6yiljf unique (userId)
Hibernate: alter table host drop index UK_42hv1ni8e0vka3c6t5u2qeioq
Hibernate: alter table host add constraint UK_42hv1ni8e0vka3c6t5u2qeioq unique (userId)
Hibernate: alter table user drop index UK_sb8bbouer5wak8vyiiy4pf2bx
Hibernate: alter table user add constraint UK_sb8bbouer5wak8vyiiy4pf2bx unique (username)
Hibernate: alter table distribution add constraint FKt0gx9ig1dy2v46t548u9cwfrp foreign key (distributorId) references distributor (distributorId)
Hibernate: alter table distribution add constraint FK7s2woi57d8ykkunjp1tn5s3it foreign key (eventId) references event (eventId)
Hibernate: alter table distributor add constraint FKomat1yocu5ruag2v2n6cposss foreign key (userId) references user (userId)
Hibernate: alter table event add constraint FKsu7s1kapgoc1yu60knpqfaeji foreign key (hostId) references host (hostId)
Hibernate: alter table host add constraint FKehvr0v8svlaou0vi2r9w8kau8 foreign key (userId) references user (userId)
Hibernate: alter table seats add constraint FKjmyywwvb5rh236sg5fwc36sl8 foreign key (eventId) references event (eventId)
Hibernate: alter table soldtickets add constraint FK9r7y60jn0k1q41y6brr19k1g4 foreign key (distributorId) references distributor (distributorId)
Hibernate: alter table soldtickets add constraint FK2w1ig52011wprc8a7sg2jj0eo foreign key (seatsId) references seats (seatsId)
Hibernate: alter table user add constraint FKntqdustq9tymisv4jl5pywyk3 foreign key (userRole) references userRole (userRoleId)
```

2. Реализация на слоя за работа с бази от данни.

DAO

Всяко едно DAO има следните функции:

- void persist(T entity) - Добавя обект към базата данни
- void update(T entity) - Обновява даденият обект
- T findById(int id) - Връща обект по id-то му
- void delete(T entity) - Изтрива обект от базата данни
- List<T> findAll() - Връща лист с всички обекти

Функциите които използваме в ДАО-тата използват основно nativeQuery-та, за да вземат информация от базата данни, но известна част от функциите ни използват други service-и и се намират в service-класа на entity-то

Ето някоѝ от уникалните функции в ДАО-тата ни:

- SeatsDao.reserveSeats - функция която запълва брой места от избраният тип, ако са налични. Ако няма достатъчно, връща грешка.

```
public void reserveSeats(Seats seats,int numberOfReservations) throws Exception {

if(seats.getNumberOfReservedSeats()+numberOfReservations>seats.getNumberOfSeats
()) throw new Exception("You can't reserve this many seats!");

seats.setNumberOfReservedSeats(seats.getNumberOfReservedSeats()+numberOfReserv
ations);

update(seats);}

}
```

- DistributorDao.getFee(int distributorId, int eventId) - връща хонорара на дистрибутора за събитието посочено в параметрите.

```
{       return getCurrentSession().createNativeQuery("SELECT DISTRIBUTION.FEE FROM
DISTRIBUTOR JOIN DISTRIBUTION ON DISTRIBUTOR.DISTRIBUTORID =
DISTRIBUTION.DISTRIBUTORID WHERE DISTRIBUTOR.DISTRIBUTORID = " + distributorId
+ " AND DISTRIBUTION.EVENTID = "+ eventId +";",float.class ).getSingleResult();

 }
```

Services

Във всеки Service има по едно DAO.

При извикване на някоя от функциите на DAO, е нужно да се извика в Service-а му функцията openCurrentSession() първо и при излизането от функцията, closeCurrentSession().

Поради тази причина всяка от по-горе изброените функции в DAO-тата бива извикана от Service-а на DAO-то и присъства със същото име тук.

Пример за уникални функции:

- **EventService.setDistribution(...)** - Преобразува лист от модели на дистрибутор във лист от обекти от таблицата Distribution. Прави връзката между събитието и дистрибуторите, като записва и хонорара към избраното събитие.

```
public void setDistribution(ObservableList<DistributorView> input, Event event)
{
   List<Distribution> listDist = new ArrayList<>();
   DistributorService distributorService = new DistributorService();
   DistributionService distributionService = new DistributionService();
   for(DistributorView x : input)
   {
      Distributor distributor =  distributorService.loadDistributor(x.getDistributorId());
      Distribution distribution = new Distribution(event,distributor,x.getFee());
      if(!listDist.contains(distribution)) {
         distributionService.saveOrUpdate(distribution);
         listDist.add(distribution);
      }
   }
   event.setListDist(listDist);
}
```

- **EventService.loadSeats(int eventId)** - Връща всички места на избраното
- събитие

- **EventService.loadDistributorRow(int eventId)** - преобразува листа от дистрибутори в

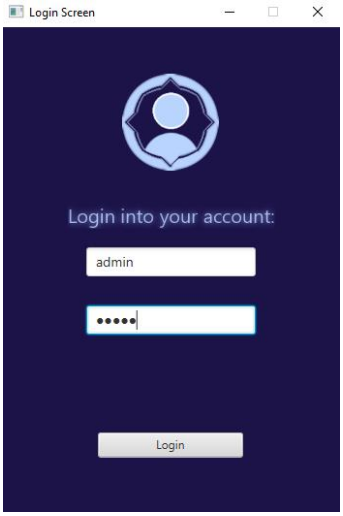
събитието в лист от модели, за да може да се изкарат на екрана.

```
public ObservableList<DistributorView> loadDistributorRow(int eventId) {
    ObservableList<DistributorView> tempDistributors = FXCollections.observableArrayList();
      eventDao.openCurrentSession();
      List<Distribution> list =
eventDao.openCurrentSession().get(Event.class,eventId).getListDist();
      for(Distribution x : list) {
         Distributor dist = x.getDistributor();
         tempDistributors.add(new
 DistributorView(dist.getDistributorId(),dist.getUser().getUsername(),x.getFee(),dist.getRating())
          );
      }
      eventDao.closeCurrentSession();
      return tempDistributors;
   }
```

- **EventService.findByDistributorId(int id, boolean type)** - Връща всички събития на дистрибутор, които са съответно приети или не (в зависимост от type)
- **EventService.toEventView(List<Event> all,int distributorId) -** Връща лист от модели за всяко събитие което е отредено на дадения дистрибутор, чието е id-то от параметъра.
- **EventService.loadDistributors(int eventId) -** взима дистрибуторите на събитието от таблицата Distribution и ги връща като лист от дистрибутори.

- **TicketService.entityToModel(SoldTickets entity) -** преобразува SoldTickets в модела TicketView като събира информацията от другите таблици.

- **TicketService.findAllByEvent(int hostId) -** връща всички продадени билети за събитията на даден организатор.
-



3. Реализация на бизнес логика и графичен интерфейс

Всеки FXML файл е свързан с контролер. Този контролер описва функционалността на сцената.

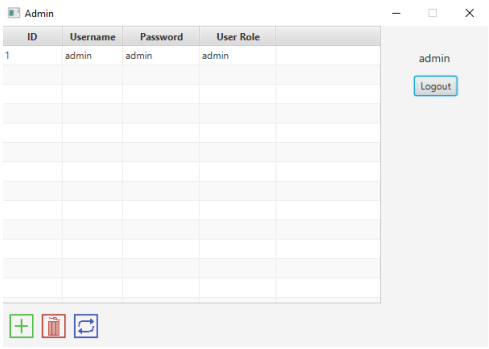
Main.java - При стартирането на проекта, се зарежда направо login.fxml сцената със съответният **Login_controller**

Login_controller

- void log_in_function(ActionEvent actionEvent) - реализира влизането във профил. При идентифицирането на ролята на клиента, го изпраща към съответната сцена и го запазва в статичният клас SessionService.

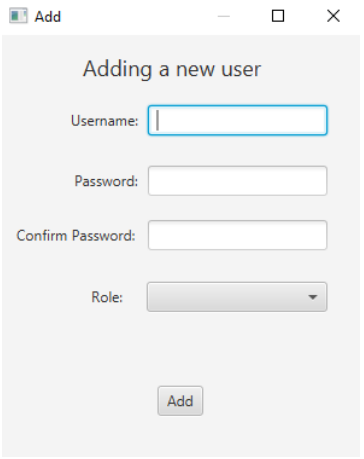
AdminController implements Initializable

- void add_to_db() - зарежда сцената AddControler като дъщерен прозорец, за да се добави акаунт към базата данни
- void delete_db() - изтрива избраният акаунт от таблицата след потвърждение и само ако акаунтът няма връзка с други таблици.
- void refresh() - запълва таблицата наново с акаунтите.



- void logout() - излизане от профила (зареждане на login сцената и нулиране на акаунта в SessionService)

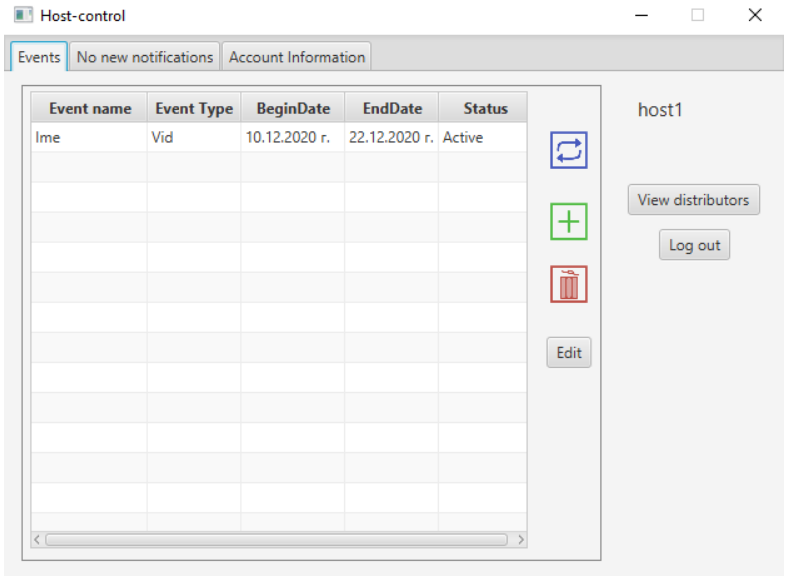
AddController



- boolean findUser(String username) - търси потребител по въведено име в базата данни.
- void create(string,string,string) - създава и добавя нов акаунт към базата данни по предадените параметри.
- void addNewAcc() - след като са били направени нужните проверки, създава нов акаунт чрез create(string,string,string)

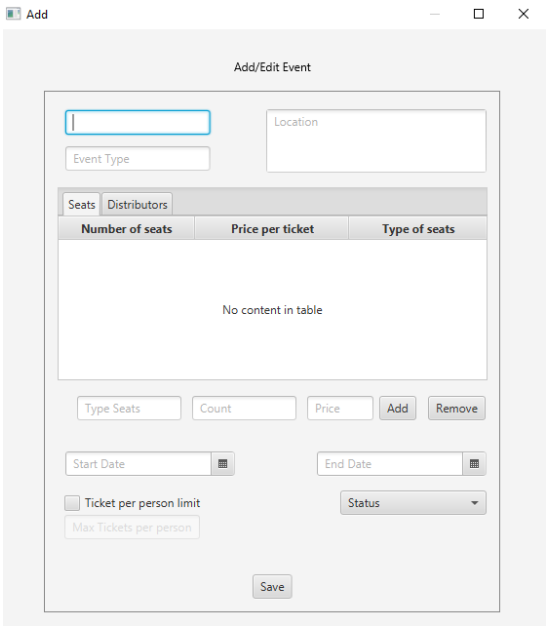
HostController implements Initializable

- void initialize(URL location, ResourceBundle resources) - задава нужните Factory-та на таблицата, запълва я, стартира двете нишки като им задава да се изпълняват през определено време на заден фон (executor.scheduleAtFixedRate)
- void add_new_event() - извиква сцената EventView празна
- void editSelectedEvent() -извиква сцената EventView като зарежда избраното събитие от таблицата чрез setEvent(Event event) на EventViewController-а.
- void refresh_event_table() запълва таблицата със събитията на влезлият организатор.
- void logout() - излизане от профила (зареждане на login сцената и нулиране на акаунта в SessionService)
- viewDistributors() - зарежда сцената queries.fxml
- void updateAccount() - смяна на име/парола на текущият потребител при повторно попълване на старата парола.



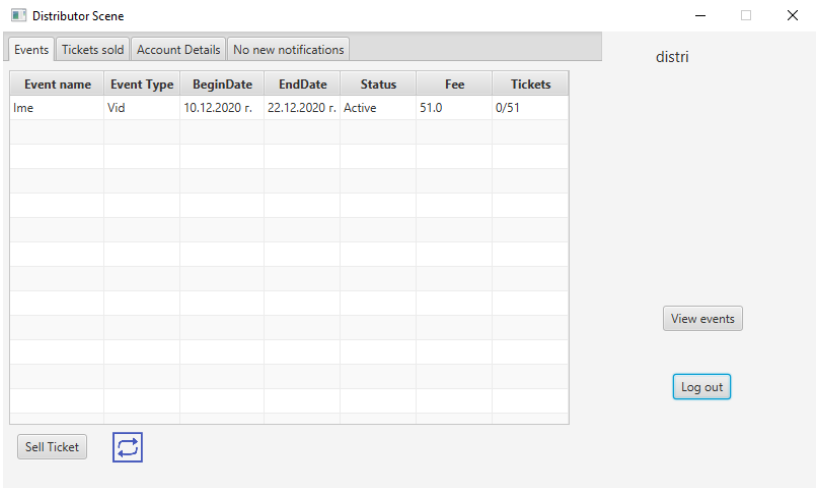
- void upcomingEventNotif(List<Event> upcomingEvents) - запълва listview-то с известията изпратени от класовете изпълняващи се на заден план.
- updateNotification(int num, List<TicketView> list) - показва колко са новите известия и добавя други към листа ако има.
- btn_mark_seen() - изчиства известията

EventViewController implements Initializable



- void initialize() - задава нужните Factory-та на таблиците, запълва comboBox-а с всички дистрибутори.
- setEvent(Event event) - Задава стойността и Id-то на локалното събитие и запълва сцената с информацията за събитието. Премахва дистрибуторите от комбо бокс-а които са вече избрани.
- createNewSeatsType() - Създава нови места и ги добавя към лист и таблицата
- removeSelected() - премахва избраните места от листа
- AddDistributor() Създава нов модел на избрания дистрибутор, маха го от комбо бокса и го добавя към таблицата.
- RemoveDistributor() - премахва избрания модел на дистрибутор и го слага отново във комбо бокса.
- refreshTable() - обновява таблицата
- void SaveEvent() - създава ново събитие по въведената информация. Добавя му избраните места и дистрибутори, като извлече информацията от моделите на дистрибуторите. След успешно създаване, затваря сцената.

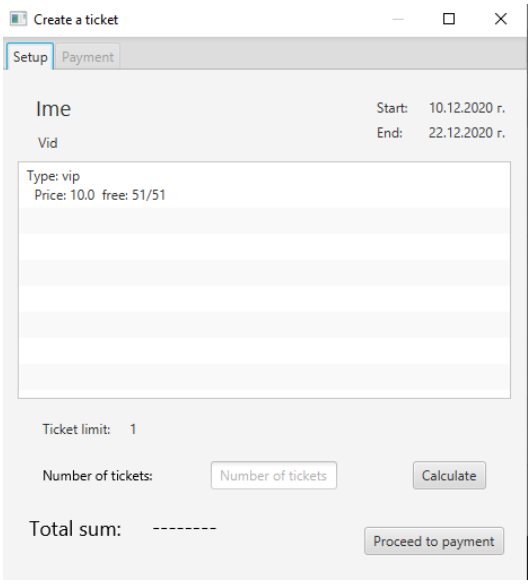
DistributorController implements Initializable



- void initialize(URL location, ResourceBundle resources) - задава нужните Factory-та на таблиците, запълва ги, стартира двете нишки като им задава да се изпълняват през определено време на заден фон (executor.scheduleAtFixedRate)
- void refresh_event_table() - обновява таблицата със събитията за този дистрибутор с тези които той е приел.
- void refresh_ticket_table() - обновява таблицата с продадените билети от този дистрибутор.
- void logout() - излизане от профила (зареждане на login сцената и нулиране на акаунта в SessionService)
- void sell_ticket() - показва сцената TicketEditor за да се продаде билет за избраното събитие. Ако не е избрано, не прави нищо.
- void notificationUpdate(int num) - тази функция бива извиквана от класовете известия които се изпълняват на заден план. Показва броя на новите известия, запълва листа с тях.
- void show_notifications() показва дъщерен прозорец за всяка покана за участие в събитие (всички събития за избрания дистрибутор които не са били приети)
- void viewEvents() - зарежда сцената queries.fxml
- void updateAccount() - смяна на име/парола на текущият потребител при повторно попълване на старата парола.
- void upcomingEventNotif(List<Event> upcomingEvents) - добавя към листа с известия тези за идните събития с непродадени билети.

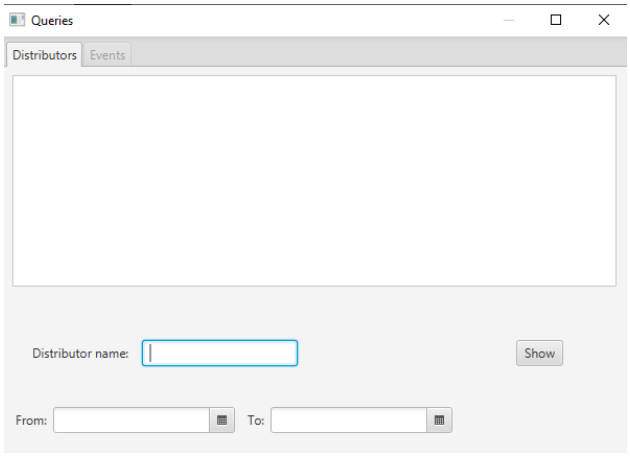
- void clearList() - изпразва листа с известия

TicketEditorController



- void setEvent(Event event) - придава стойност на събитието и добавя местата на събитието в ListView-то.
- void set() - изчислява цената на покупката, ако е възможна
- void proceed() - продължава на следващия раздел, ако покупката е възможна. Там се попълва информацията за купувача.
- void createTicket() - запазва информацията за продадения билет, купувача, намаля се броя на свободните места. Дистрибутора получава +1 рейтинг. Сцената се затваря

QueryController implements Initializable



- void showDistributor() - взима списък от разпространители само за дадено събитие в периода предоставен от потребителя
- void showEvent() - показва списък с всички известия в подадения период
- void initialize() - проверка на потребител , за да се превключи на раздела със справките достъпни на дадения потребител

SessionService - Статичен клас с цел запазване на информацията на влезлия потребител и преизползването на някои функции, като конвертиране на формати дата от един към друг.

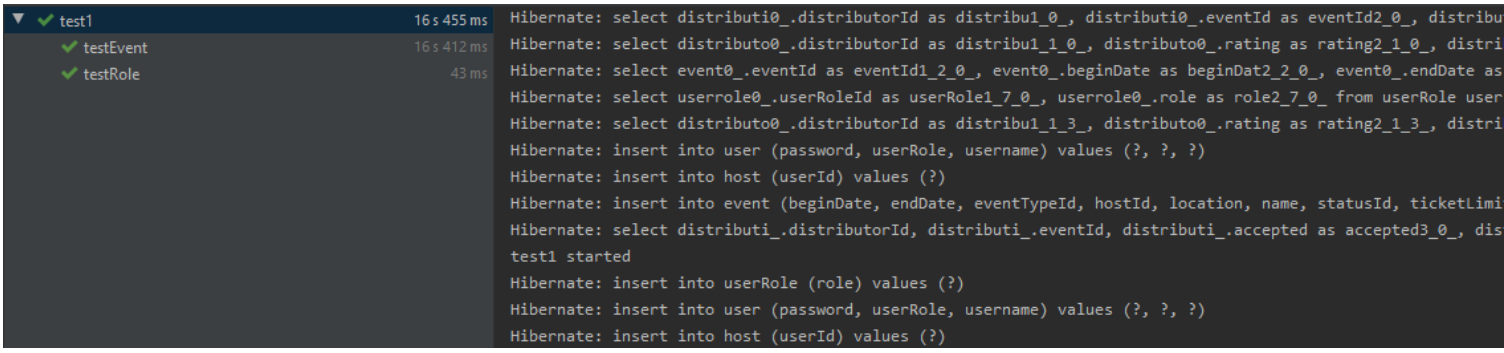
Тестови резултати:

Използваме JUnit за тестването на дадени функции.

Следните тестове проверяват дали се запазват в базата данни , събитието и организатора.

```
public class test1 {
   private Session session;
   private Transaction transaction;
   @Before
   public void setUp() {
      session = HibernateUtil.getSessionFactory().openSession();
      transaction = session.beginTransaction();
   }
   @Test
   public void testRole(){
      System.out.println("test1 started");
      UserRole userRole = createRole();
      session.save(userRole);
      User user = createUser("testuser");
      session.save(user);
      Host host = createHost(user);
      session.save(host);
   }
   @Test
   public void testEvent() {
      DistributionService distributionService = new DistributionService();
      List<Distribution> list = distributionService.findAll();
      User user = createUser("testuser2");
      session.save(user);
      Host host = createHost(user);
      session.save(host);
      Event event = createEvent(list,host);
      session.save(event);
   }
   @After
   public void transactionCommit(){
      transaction.commit();
   }
   private UserRole createRole(){
      UserRole userRole = new UserRole();
      userRole.setRole("testRole");
```

```
        return userRole;
    }
    private User createUser(String username){
        User user = new User();
        user.setUsername(username);
        user.setPassword("12345");
        return user;
    }
    private Host createHost(User user){
        Host host = new Host();
        host.setUser(user);
        return host;
    }
    private Event createEvent(List<Distribution> dist,Host host){
        Event event1 = new Event();
        event1.setName("Championship");
        event1.setBeginDate("Yesterday");
        event1.setEndDate("Tomorrow");
        event1.setEventType("Football");
        event1.setStatus("Cancelled");
        event1.setHost(host);
        event1.setListDist(dist);
        return event1;
    }
}
```



Функцията след анотация @Before се изпълнява преди да се изпълнят тестовете , които са функциите след анотация @Test. Имаме и функция след анотация @After , която да се изпълни след всички тестове.