ПРОЕКТ

Университет: Технически университет – Варна

Катедра: ФИТА

Специалност: СИТ

Тема на проекта: VI. Пътнически превози

Реализиран от: Максимилиян Георгиев (Фак. № 18621814) и Дончо Дончев (Фак. № 18621818)

Сорс код достъпен на: https://github.com/tuvarna-coursework/transport-system-management

ЗАДАНИЕ НА ПРОЕКТА

Да се разработи информационна система, предоставяща услуга пътнически превози. Програмата съхранява и обработва данни за разпространение на билети.

Системата поддържа два вида потребители администратор и клиент (пътническа компания, разпространител, касиер) с различни роли за достъп до функционалностите в системата.

Операции за работа с потребители:

* Създаване на пътнически компании от администратор;
* Създаване на разпространители от администратор;
* Създаване на касиери от разпространител
* Поддържане на профили с характеристики на клиентите в системата (хонорар и др..)
* Рейтинговане на клиентите в системата

Системата поддържа операции за работа със пътувания:

* Добавяне на ново пътуване от организатор (вид на пътуването, дестинация, дата на заминаване и пристигане, брой места, вид(ове) транспорт, ограничение в закупуването на билет от едно лице и др..)
* Заявяване на билети за продажба от разпространител и потвърждаване от пътническа компания
* Продаване на билети за пътуване от касиер, създаване на формуляр за закупуване (информация за купувача, избор на място и др…)

Системата поддържа Справки по произволен период за:

* Клиентите в системата:

Пътническа компания с наличните пътувания (за разпространители);

Разпространители

Касиери

* Закупени билети (дата, статус, и др…);
* Пътувания;

Пътническата компания достъпва справки само за пътувания, на които е организатор. Разпространителя достъпва справки за всички актуални пътувания. Касиерите имат право на справки само за пътувания, чиито билето-разпространители са.

Системата поддържа Известия за:

* Новопостъпила заявка за пътуване (в профила на разпространител);
* Периодично уведомление за продадени билети от пътуване (в профил на собственика)
* Отменено пътуване (в профила на касиера и разпространителя)
* Наближаващо пътуване с не продадени билети (собственик, разпространител)

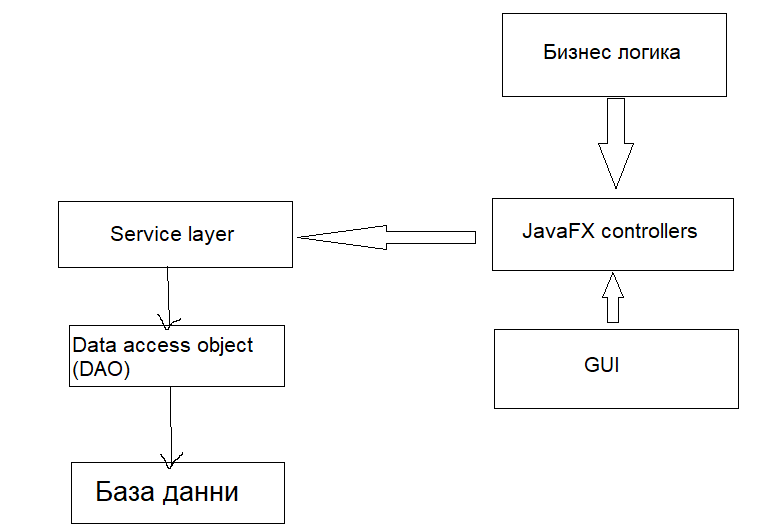
ФУНКЦИОНАЛНИ ИЗИСКВАНИЯ КЪМ ПРОЕКТА

|  |  |
| --- | --- |
| № | Описание на функционално изискване |
| 1 | Администратор създава пътнически компании (генерира потребителско име и парола, назначава роля и запазва в базата данни) |
| 2 | Администратор създава разпространител (генерира потребителско име и парола, назначава роля и запазва в базата данни) |
| 3 | Администратор поддържа профили с характеристики за потребителите (обновява рейтинг и хонорар ръчно) |
| 4 | Разпространител създава касиери (генерира потребителско име и парола, назначава роля и запазва в базата данни) |
| 5 | Разпространител назначава касиери към дадена компания |
| 6 | Разпространител назначава касиери от дадена компания към дадено пътуване |
| 7 | Валидация при назначаване на касиер към пътуване: локацията на касиера съвпада с точка, през която минава пътуването. |
| 8 | Валидация за правилен вход в администраторския панел (съвместими типове данни) |
| 9 | Организатор създава ново пътуване (въвежда вид на пътуването, начално и крайна точка, дата и час на заминаване, брой места, видове транспорт, ограничение в закупуването на билети, въвежда междинни точки в маршрута и др.) |
| 10 | Новото пътуване на пътническата компания се присвоява към компанията (UsersTrip таблица) |
| 11 | Валидация в панела за създаване на ново пътуване (правилен тип данни, датата да е след текущата дата, локациите да са различни и др.) |
| 12 | Заявки за допълнителни билети от разпространител към пътническа компания (добавяне на нова заявка в базата данни). |
| 13 | Зареждане на заявките за конкретната пътническа компания |
| 14 | Функционалност за приемане или отхвърляне на дадената заявка, обновление на статуса на заявката (REJECTED, PENDING, ACCEPTED). |
| 15 | Валидация на заявката: допълнително предложените билети + текущите билети не могат да надхвърлят капацитета на автобуса |
| 16 | Функционалност за продаване на билети от касиер (избиране на купувач, ако е регистриран или създаване на нов потребител-гост). Създава се нов билет и той се назначава на потребителя. |
| 17 | Валидация при продаване на билети: касиер продава билети само за пътувания, към които е назначен, и неговата локация съвпада с локацията, от където иска да продаде билет. Следи се дали има достатъчно билети за продан. |
| 18 | Допълнителна функционалност: потребител купува сам билети. Въвежда критерии за търсене и вижда отговарящите пътувания. При закупуване се генерира нов билет и се назначава към потребителя. |
| 19 | Валидация при покупка от клиент: достатъчен брои билети достъпни |
| 20 | Функционалност за показване на разписание за пътническа компания (техните пътувания) |
| 21 | Функционалност за показване на разписание за разпространител (всички пътувания) |
| 22 | Функционалност за показване на разписания за касиер (само за такива, към които е назначен) |
| 23 | Нотификации за новопостъпило пътуване (за разпространител) |
| 24 | Нотификации за продадени билети от пътувания (на дневна база пътническата компания получава доклад) |
| 25 | Отменено пътуване: ако компания отмени пътуването, разпространителя получава нотификация |
| 26 | Нотификации за наближаващо пътуване с не продадени билети (за собственик и разпространител) |
| 27 | Логин функционалност: дешефриране на кодираната парола на потребителя и пренасочване към съответното ниво на достъп. Назначава текущия усер като логнат и чрез него се правят заявки към базата. |
| 28 | Валидация при логин: непразни полета, неправилен вход. |
| 29 | Функционалност за регистрация: (само на краен потребител) |
| 30 | Валидация при регистрация: дължина на потребителското име и парола, без специални символи в потребителското име. |
| 31 | Функционалност касиер да вижда билетите, които е продал и на кого. |
|  |  |

СТРУКТУРА НА ПРОЕКТ

Проектът е реализиран на **Maven**. Използвани са следните dependency-та:

* **PostgreSQL** – връзка с базата данни
* **Hibernate** – слой за работа с бази данни
* **JavaFX** – реализация на графичен интерфейс и бизнес логика чрез съответните контролери
* **JBCrypt** – функционалност за криптиране и декриптиране на потребителска парола
* **Log4J** – логове за всички контролери за проследяване на логиката на приложението. Допълнително използване, ако се налага да се дебъгва
* **JUnit** – Unit тестове

Структурата на проекта е следната: 

**ДЕФИНИЦИЯ НА МОДУЛИТЕ НА СИСТЕМАТА**

**DatabaseUtils –** статичен клас, който отговаря за глобалната сесия, тук се инициализира, тук се пази и текущия потребител, който е логнат в системата. Притежава и спомагателни функции, които подпомагат работата с бази данни. Зареждат се и примерни данни или такива, които никога не се променят и се запазват в базата данни.

**NotificationUtils –** статичен клас, отговорен за системата за нотификации. Цикличните нотификации се извикват със стартирането на програмата и има приложена периодизация. Тук се генерират всички типове нотификации, запазват се в базата данни и се изпращат до съответния получател.

**Entities –** това са всички класове, които са участници в системата и са обекти, които се съхраняват в база данни и са мапнати от Hibernate.

**DAO –** слой за работа с бази данни. Тук се осъществяват всички основни операции (CRUD) с базата данни. Тук се извличат Entity-та от базата данни (по даден критерии), правят се нови записи, променят се параметри или се изтриват от базата данни. Служи само и единствено за работа с база данни. Имплементира интерфейс **GenericDAOInterface<T>,** в който се съдържат CRUD операциите и се разширява функционалността от там спрямо нуждите на потребителя и Entity-то. За всяко Entity съществува точно едно DAO. Тук се осъществяват всички транзакции и заявки, използва се частен EntityManager, а за сесия се използва глобална, инициализирана от **DatabaseUtils** при стартиране на приложението (обърнете се към описанието на DatabaseUtils).

**Service layer** – Допълнително ниво на абстракция в приложението. В този слой се съдържа частно DAO съответстващо на Entity-то, за което е написан този сървис. Подобрява се скаларността на системата, тъй като може да се използва повече от едно DAO. Пример: Ако приложението работи с две различни бази данни и на нас ни се налага да извличаме от двете, просто добавяме още едно DAO и отново достъпваме функционалността чрез този Wrapper class, или ако се наложи смяна на начина на свързане към база данни, то Service класа ще си остане същия, а ще се пренапише DAO-то. Имплементира CrudService<T>, където отново са заявени основните CRUD операции. Целта е било максимално да се намали зависимостта между отделните класове, така че те да са изцяло независими.

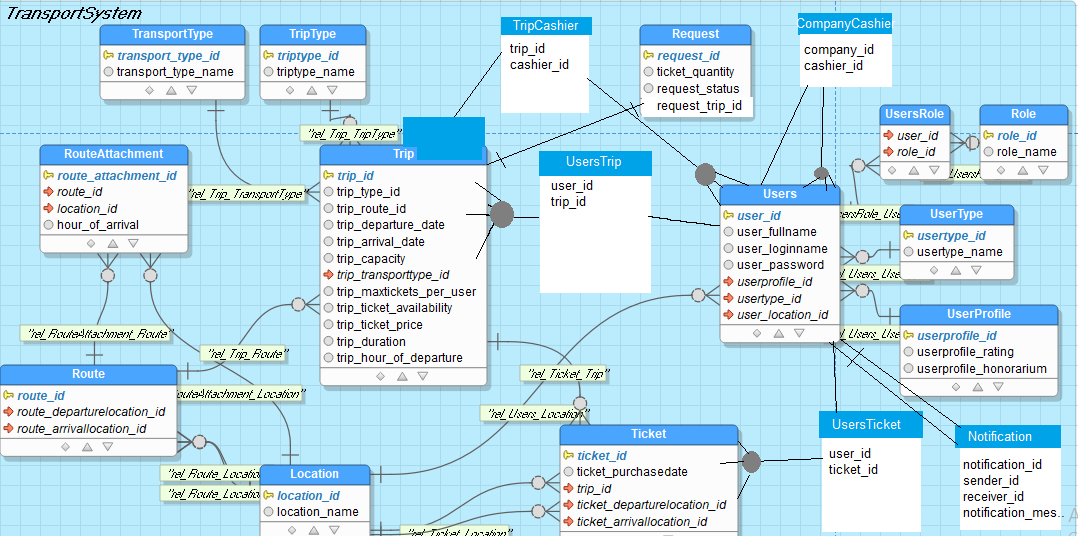
Оптимизация: в нашата система има един service за всяко едно entity. Може да се комбинират спрямо функционалността си: UserService да съдържа UserDAO, UserTypeDAO, UserProfileDAO.

**Controllers** – контролерите извършват цялата бизнес логика в приложението. Абсолютно всеки изглед от JavaFX е обвързан със съответен контролер. Тук има валидации, тук се изобразява графично информацията на потребителя и всички заявки тръгват и се връщат тук.

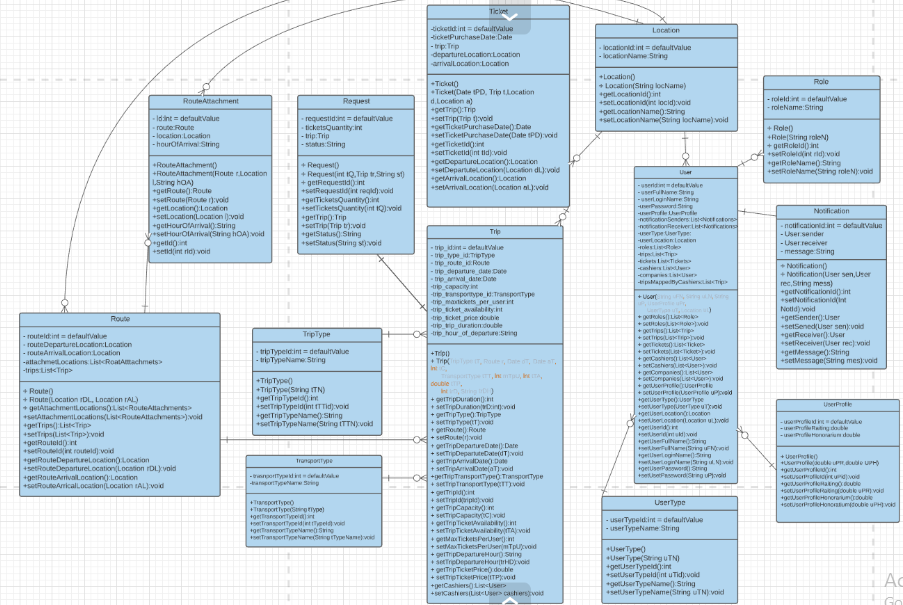
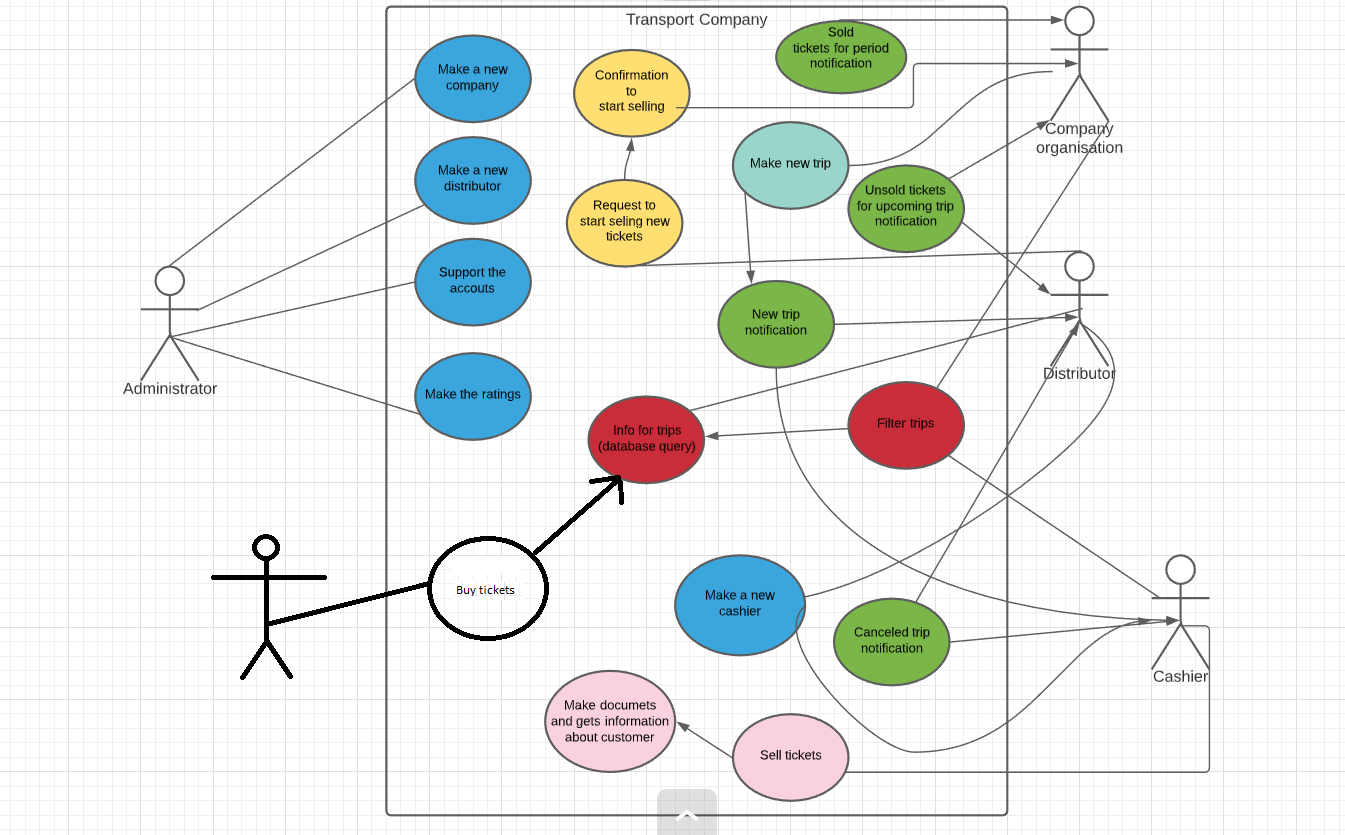
**Диаграми**

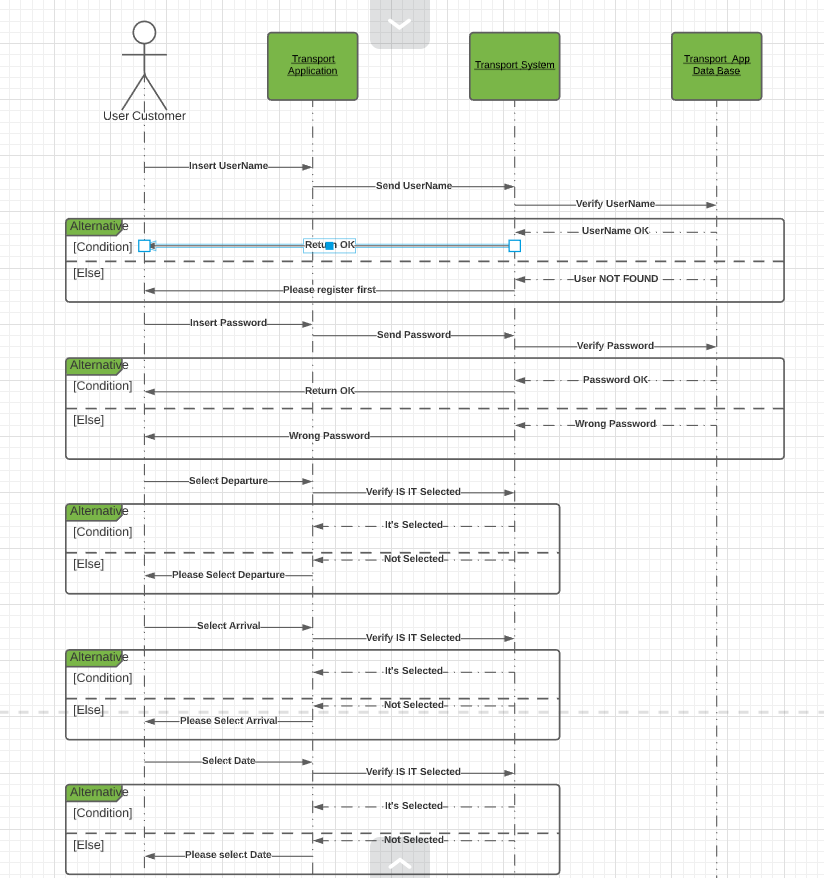
**Пълният размер на снимките са налични на:** [**https://github.com/tuvarna-coursework/transport-system-management/tree/master/documentation**](https://github.com/tuvarna-coursework/transport-system-management/tree/master/documentation)

**Релационна схема описание на таблиците и UML – Use Case Diagram:**

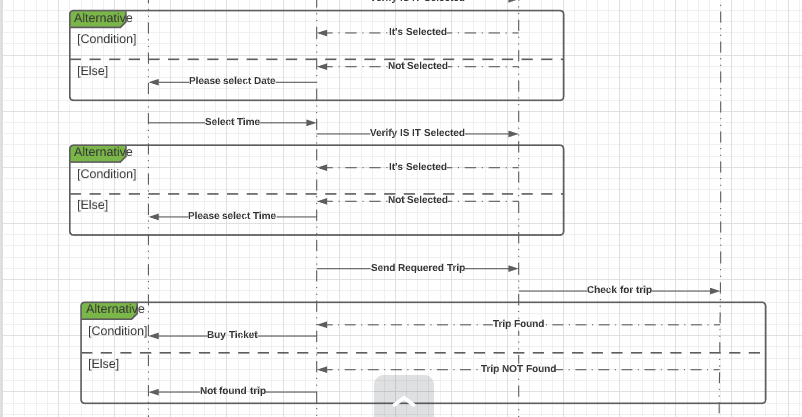
****

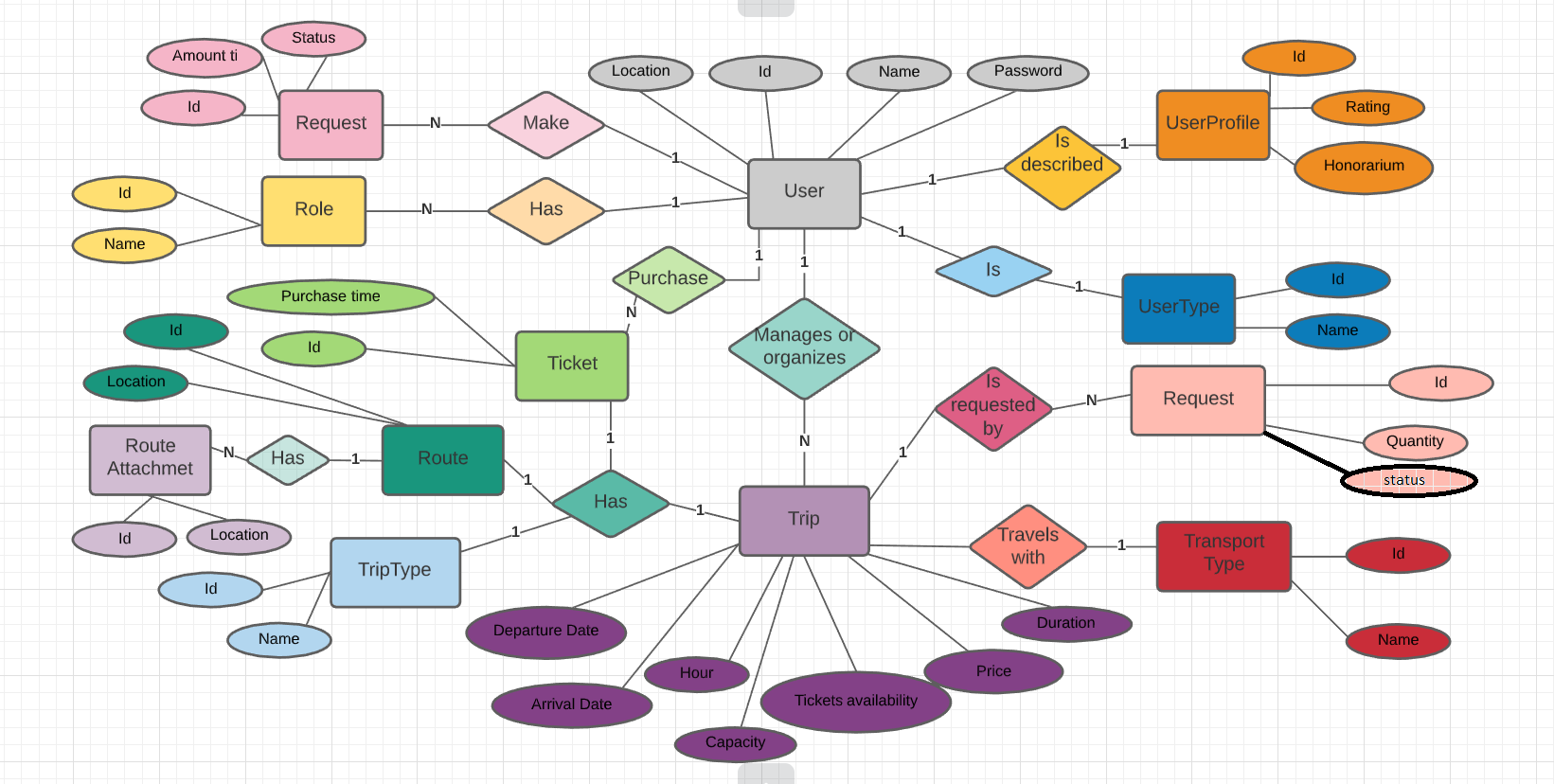
**UML – Use Case**

**UML – Class Diagram**

**UML –Sequence Diagram (За потребител: )**

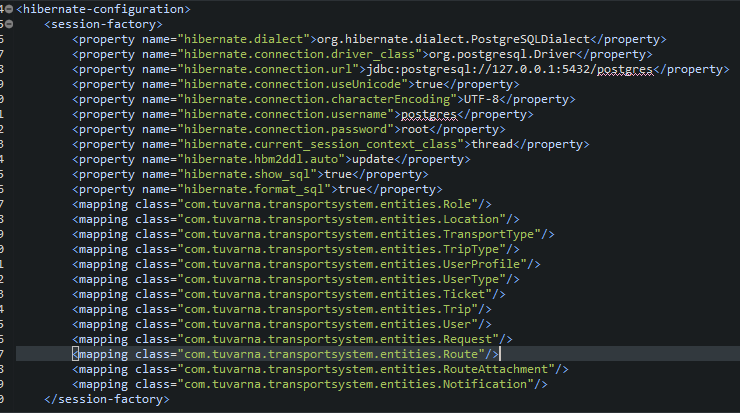
**Sequence Diagram #2:**

****

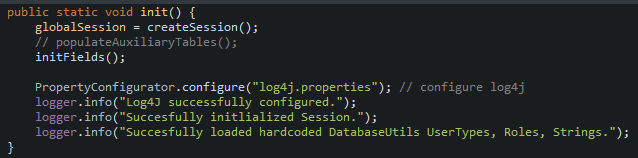
**Модел на Чен (ER диаграма)**

**Реализация на база данни**

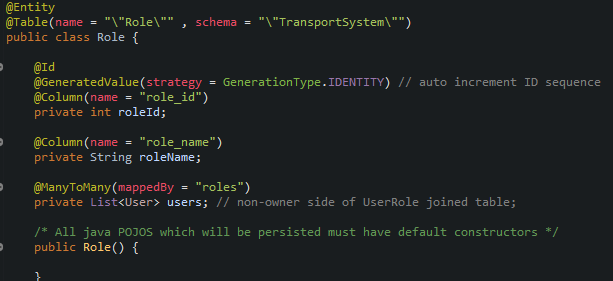
Използваната база данни е PostgreSQL. Избрана е, тъй като е гъвкава и удобна и решихме да използваме нещо различно.

Конфигурацията на връзката, както и всички класове, които са Entity-та, са описани в конфигурационния файл **hibernate.cfg.xml**:

Сесията е глобална и тя се създава при стартиране на главното приложение. Извиква се функцията **DatabaseUtils.init(),** където се създава сесията, зареждат се примерни данни (ако е първо пускане на програмата и няма нищо в базата данни) и се инициализират статични стойности които се ползват из цялото приложение (роли, тип потребители, стрингове и т.н.)

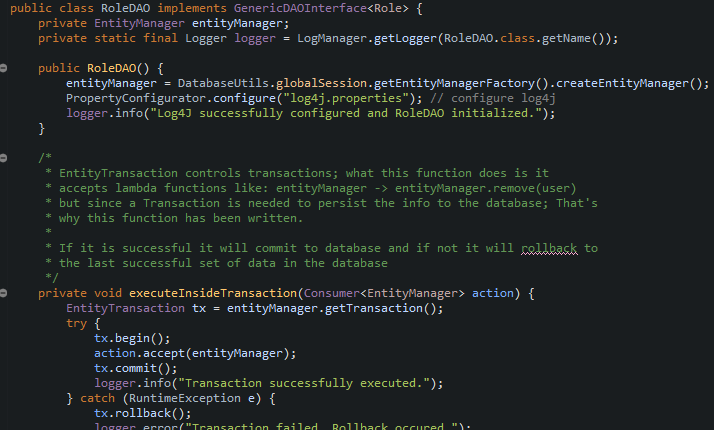


Следва пример как е асоциирано едно Entity с едноименна таблица от базата данни. Ще вземем пример **Role**.

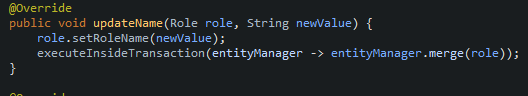


Тук е указано, че типа на обекта е Entity, посочена е таблицата, към която трябва да се свърже. Съответно са конфигурирани и отделните полета.

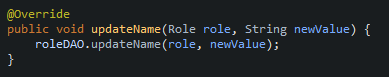
Следва да разгледаме работата на DAO-то. В конструктора се създава entityManager, който ръководи всички транзакции. Този конструктор се ползва от съответния Service на класа. executeInsideTransaction е нашата функция, която осъществява **UPDATE, INSERT, DELETE**. Опитва да извърши операцията, но при възникнала грешка възстановява базата данни до момента, в който е била преди това и така не може да има непредвидени обстоятелства и загуба на данни.



Пример на извикване на тази функция:

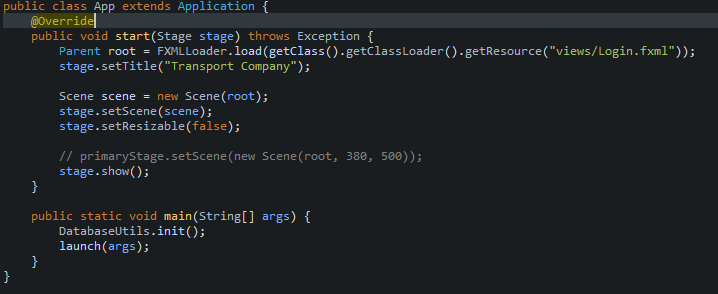


Пример за извикване на тази функция от Service, който се ползва навсякъде в приложението вместо работа директно с DAO:

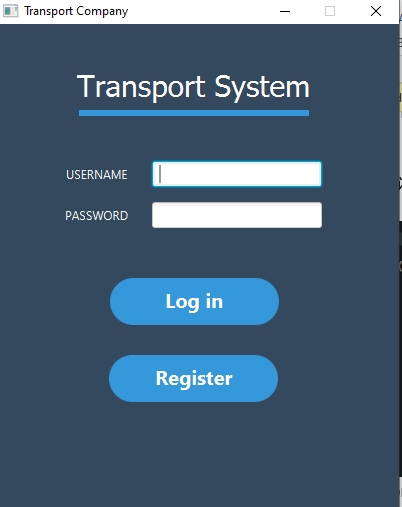


**БИЗНЕС ЛОГИКА И ГРАФИЧЕН ИНТЕРФЕЙС**

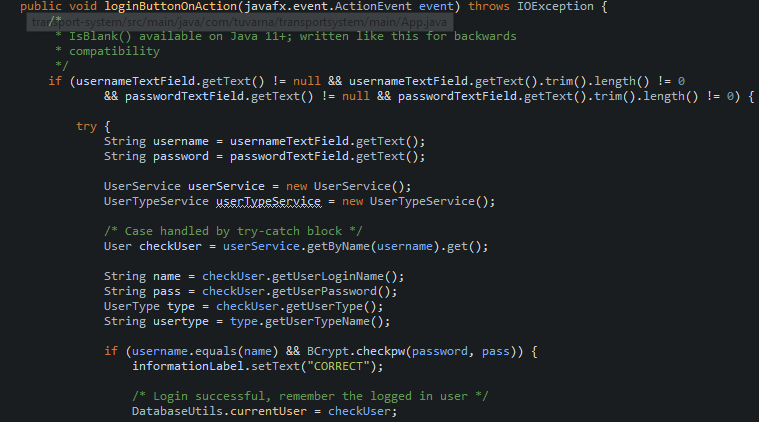
Както бе упоменато, бизнес логиката и графичния интерфейс се контролира от JavaFX. Принципа на работа е следния:



1. При стартиране се зарежда главната сцена за логин в приложението и се извиква инициализирането на базата данни.
2. Всеки изглед е в папка ресурси и е с разширение .fxml
3. Всеки изглед има съответен контролер, които реагира на събития (натискане на бутон, валидация на текстови полета и др.)

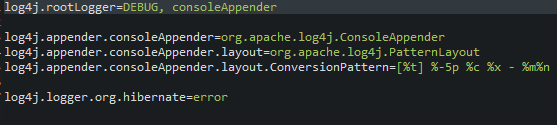


Това е първоначалния изглед и следва да разгледаме част от функционалността на съответния **LoginController**:



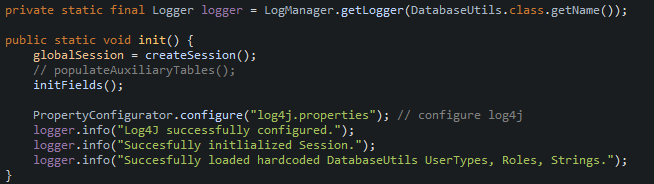
**МОДУЛ ЗА РЕГИСТРИРАНЕ НА СЪБИТИЯ**

Използва се библиотеката log4J, която е конфигурирана да разпечатва само в конзолата. Сметнато е за ненужно да се записва и в лог файл. Разполага с конфигурационен файл **log4j.properties:**



Нивото на логване е DEBUG, а формата, в който се разпечатва е максимално подробен и включва нивото на логване, мястото от където е извикан текущия ред, и съответния клас, в който се случва събитието. Указано е да се пропуска лога на Hibernate, тъй като той пише огромни количества логове и е ограничено само до **ERROR** ниво. Към всеки клас има прикачен **Logger**. Всяко действие се логва, било то успех или Exception. Единствено е пропуснато там, където графично се изобразява проблема (напр. грешно въведено име при регистрация).

Пример с DatabaseUtils:



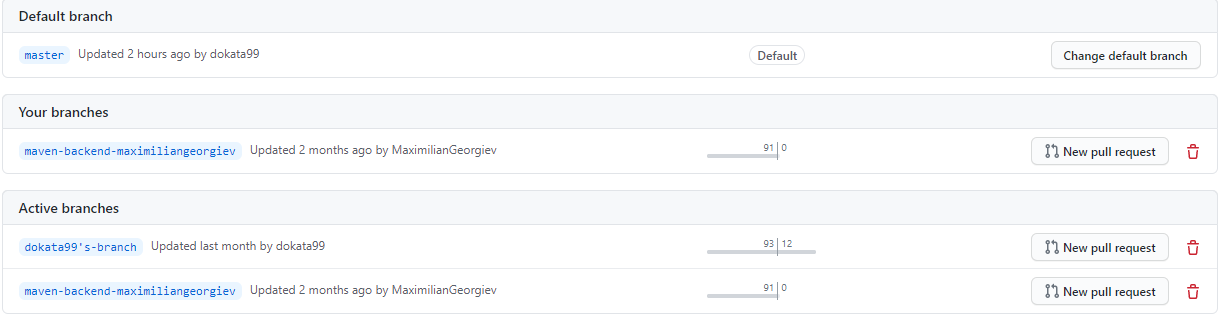
Инициализацията на логера, както и самото логване, което се осъществява навсякъде. Ето го и него:



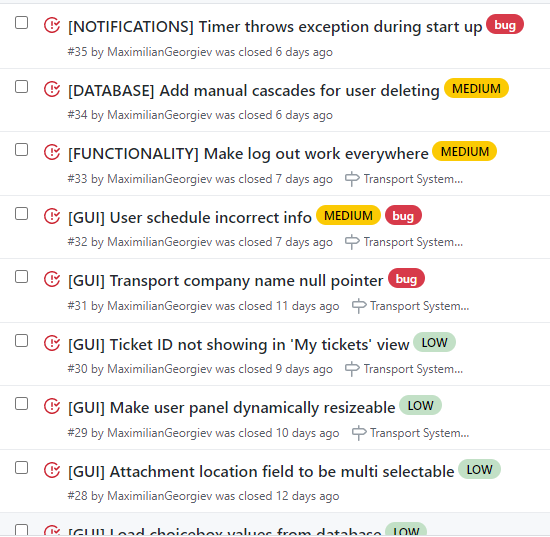
**SOURCE CONTROL И РАЗПРЕДЕЛЕНИЕ НА ЗАДАЧИТЕ**

Платформата, която сме използвали за source control e **Github**. Работата се е осъществявала паралелно. Всеки участник е имал задачи, които са съобразени, така че да са максимално независими от другия.

Започнахме работата със собствени бранчове. Максимилиян имаше за цел да реализира структурата на проекта и да навърже базата с данни. Дончо имаше за цел да направи изгледите. След осъществяването на тези задачи, проектът бе убединен в **master branch.**



За да имаме ясна концепция какво има да се прави и какво е належащо, ние се възползвахме от системата за Issue tracking на Github:



**ВЪЗМОЖНА ОПТИМИЗАЦИЯ НА ПРОГРАМАТА**

1. Въвеждане на автоматизирани CASCADE операции от Hibernate вместо ръчните такива
2. Нотификациите да се осъществяват асинхронно
3. Графични подобрения: оптимизации за различните резолюции, при дърпане, zoom и тн
4. Зареждане на стойностите от избираемите менюта със стойности директно извлечени от базата данни
5. Функционалност за триене на нотификации или заявки, повече функционалност за търсене

**ИЗПОЛЗВАНА ЛИТЕРАТУРА**

<https://www.baeldung.com/hibernate-one-to-many>

<https://www.baeldung.com/jpa-many-to-many>

https://www.baeldung.com/jpa-cascade-types