|  |  |
| --- | --- |
|  | ТЕХНИЧЕСКИ УНИВЕРСИТЕТ - ВАРНА  ФАКУЛТЕТ ПО ИЗЧИСЛИТЕЛНА ТЕХНИКА И АВТОМАТИЗАЦИЯ |

**КУРСОВ ПРОЕКТ по ООП-2**

изготвен от

Елица Иванова Тодорова - фак. №19621649

и

Кенан Сейханов Кемалов - фак. №19621655

спец.: Софтуерни и интернет технологии,

III курс, 2-ра група

**ТЕМА** *№16*

**Пътнически превози**

В А Р Н А

2 0 2 2 год.

1. **Задание на проекта.**

**ТЕМА №16**

Пътнически превози

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 клиенти (пътническа компания, разпространител, касиер) с различни роли за достъп до функционалностите в системата.

Операции за работа с потребители:

• Създаване на пътническа 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 с 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
* Касиер

• Закупени билети (дата, статус, и др...);

• Пътувания;

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

Системата поддържа Известия за:

• Новопостъпила заявка за пътуване (в профила на ~~разпространител~~ пътническата компания)

• Периодично уведомление за продадени билети от пътуване (в профила на собственика)

• Отменено пътуване (в профила на касиера и разпространителя)

• Наближаващо пътуване с непродадени билети (собственик, разпространител)

1. **Анализ на проблема.**
2. Функционални изисквания.

Приложението ще има функция за логване. Ще могат да се логват всички типове потребители, които системата поддържа - администратор и клиенти (пътническа компания, разпространител, касиер). Те ще могат да се логват, след като въведат своето потребителско име и парола. Потребителското име и паролата може да включват малки и големи латински букви, цифри и специални символи, и ще се състоят от максимум 50 символа. След логване в системата, потребителите ще видят основната страница на приложението и меню, от което да избират. Всеки от типовете потребители ще разполага с различно меню, предлагащо възможност за достъп до функционалностите, с които всеки от тях разполага.

Администраторът ще има достъп до всички функционалности на системата. Ще може да създава потребители от всеки тип - пътнически компании, разпространители и касиери, като въведе съответно наименованието/името на клиента, потребителското му име, паролата и хонорара, който ще получава. Той ще може да поддържа профилите им, като добавя, променя или изтрива характеристиките в тях (например хонорарът им). Администраторът ще може да добавя видове пътуване и транспорт, нова локация (дестинация), ново пътуване, да създава формуляр за закупуване на билет, както и нови известия в базата данни. Администраторът ще може да прави и промени след добавянето на гореспоменатите данни в базата. Ще може да променя броя на местата при дадено пътуване, името на купувача във формуляра, както и статуса на заявката за допълнителни билети за продажба.

Всяка пътническа компания може да бъде организатор на пътуване. Ако е организатор, ще може да добавя ново пътуване като въведе датата на заминаване и пристигане, броя места, вида на пътуването и на транспорта, локацията на заминаване и дестинацията. Пътническата компания ще потвърждава или отказва заявки за билети за дадено пътуване, изпратени от разпространителя.

Разпространителят ще може да създава касиери след въвеждане на името на касиера, потребителското му име, паролата и хонорара, който ще получава. Той ще може да изпраща заявки за билети за продажба до пътническата компания, когато наличните такива са изчерпани.

Касиерът ще продава билети за пътуванията, като създава формуляр за закупуване на такива. Формулярът ще съдържа трите имена на купувача, номера на мястото в превозното средство, датата на закупуване на билета, името на касиера, както и информация за пътуването – дата на заминаване, пътническата компания, с която ще пътува, както и локацията от-до на пътуването.

Системата ще извежда справки по произволен период, зададен от потребителя:

* за клиентите в системата

Администраторът ще има достъп до справки за всички клиенти в системата – пътнически компании, разпространители и касиери. Пътническата компания ще достъпва справки за разпространителите и касиерите, а разпространителите ще могат да извеждат справки само за касиерите в системата. Касиерите няма да имат достъп до справки за клиенти в системата. В справките ще се извеждат името/наименованието на клиента, потребителското му име и хонорарът, който получава.

* за закупени билети

Ще се извеждат справки за билети за зададения период с информация за датата им на закупуване, статуса им, за лицето, което ги е закупило, както и за пътническата компания-организатор на пътуването и за разпространителя.

* за пътувания

Ще се извеждат справки за всички пътнически компании с налични пътувания. Пътническата компания ще има достъп до справки само за пътувания, на които е организатор. Разпространителят ще достъпва само актуалните пътувания към момента на търсенето. Касиерът ще има право на достъп само до пътувания, за които разпространява билети.

Справките за пътувания ще включват вида на пътуването, началната и крайната му дата, дестинацията, броя места, вид(овете) транспорт, пътническата компания – организатор и разпространителя.

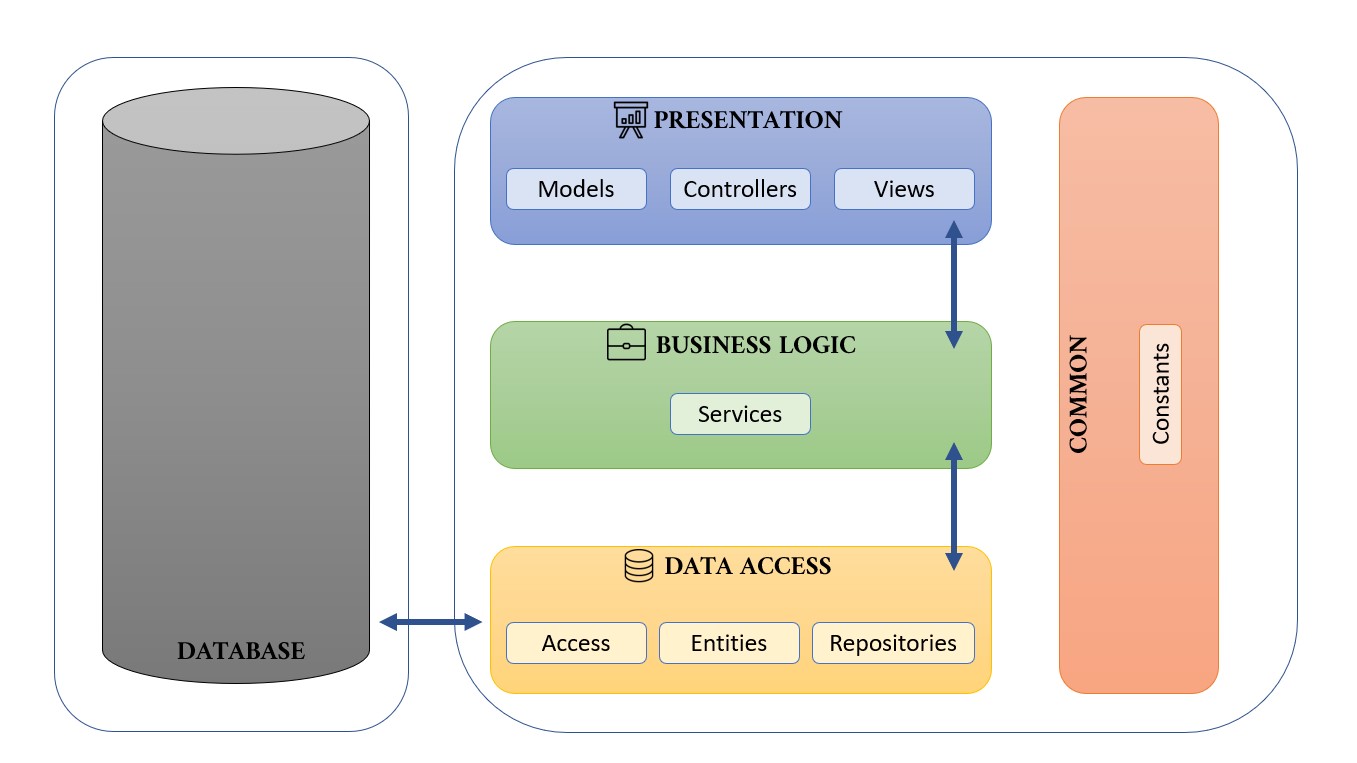
Системата ще поддържа известия. В профила на разпространителя ще възниква известие при новопостъпила заявка за пътуване. В профила на собственика периодично ще се известява за продадени билети от пътуване. В профила на касиера и разпространителя ще се получава известие при отменено пътуване. В профила на собственика и разпространителя ще се получава известие, когато има непродадени билети за наближаващо пътуване.

1. Структура на проекта и дефиниция на модулите на системата.

Структурата на проекта съдържа модулите на приложението.

То е съставено от модул за потребителски интерфейс (Presentation Layer), модул за бизнес логика (Business Logic Layer), модул за бази данни (Database) и модул за комуникация между базата данни и бизнес логиката (Data Access Layer).

За реализация на проекта се използва многослойна архитектура, съдържаща гореспоменатите модули.

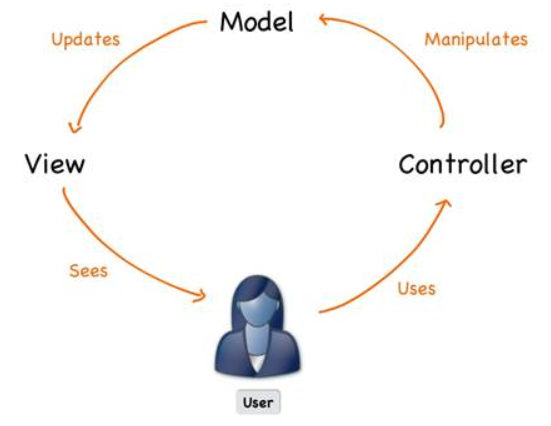
******

**Presentation Layer**

Презентационният слой е на най-високо ниво в приложението и потребителят има директен достъп до него. Той представлява връзката между системата и крайния потребител (изгледът, който виждат потребителите), като предоставя различни видове информация (и опции) на потребителя. За реализиране на презентационния слой е използван архитектурният шаблон MVC (или Model-View-Controller), който се основава на разделянето на бизнес логиката от графичния интерфейс и данните в дадено приложение.

Съставен е от 3 модула:

* Models – данните, въведени от потребителя, които се обработват от системата;
* Views – визуални JavaFX файлове, които дефинират изгледа на потребителския интерфейс; всяко view е обвързанo със съответен контролер;
* Controllers – контролери, които управляват потребителския интерфейс, като осъществяват връзката между въведените от потребителя данни и тяхната обработка в бизнес логиката; логика за управлевление на views;



Презентационният слой комуникира с Бизнес логиката.

**Business Logic Layer**

Бизнес логиката е междинният слой между презентационния и слоя за комуникацията с база данни, който отговаря за изпълнението на функционалните изисквания към създаденото приложение. Той контролира функционалността на приложението, като извършва различни процеси по обработката на данните, предаден от слоя за връзката с база данни. Част от бизнес логиката е модул Services:

* Services – предоставя услуги за извличане на данни от базата данни

**Data Access Layer**

Data Access слоят реализира комуникацията с базата данни. Осъществяват се всички основни операции в базата данни – добавят се нови записи, променят се параметри или се изтриват данни от базата. Слоят за работа с базата данни е съставен от 3 модула:

* Access – осъществява се същинската връзка с базата данни
* Entities – съдържа класове, съответстващи на таблиците в базата данни и връзките между тях; чрез Entity модели се осъществява работа с базата данни
* Repositories – съдържа класове с методи, имплементиращи заявки към базата данни (save, update, delete)

**Common Layer**

Общият слой позволява достъп на всички слоеве до общите ресурси на системата. Включва модул Constants:

* Constants –указвa пътя до fxml файловете, изградащи потребителския изглед, до конфигурационния файл на log4J, използва се за проследяване на потребител в системата

**Database**

Този слой се състои от сървър база данни. Тук информацията се съхранява и чете. Когато данните се съхраняват в отделен слой се увеличава мащабируемостта и се подобрява производителността.

1. **Проектиране на системата.**
2. Проектиране на отделните модули (база данни, графичен потребителски интерфейс GUI, бизнес логика).

1.1. Проектиране на база данни.

Базата данни ще се изгради в Microsoft SQL Server, защото включва професионален софтуер за управление на бази данни, който е лесен за използване и предоставя повече въможности от конкурентни продукти.

Базата данни ще включва 11 таблици, именувани в единично число с главна буква. Те ще са съответно 4 за типовете потребители в системата – Администатор, Разпространител, Пътническа компания и Касиер. Останалите 7 ще са за Вид на пътуване, Вид на транспорт, Локация, Пътуване, Продажба на билет, Заявка за билети, Известие. Всяка от тях ще има подходящи полета за въвеждане на необходимите данни. Ще бъдат свързани помежду си с подходящи връзки.

Базата данни ще има нужда от нормализация, за да се избегнат повторенията в нея и за да се ускори търсенето в базата данни. Дублирането на данни може да доведе до грешки, затова трябва да се избегне, като данните се съхраняват само на едно място. Ускореното търсене в базата данни е възможно, тъй като индексите могат да се създават по-бързо.

1.2. Проектиране на графичен потребителски интерфейс (GUI).

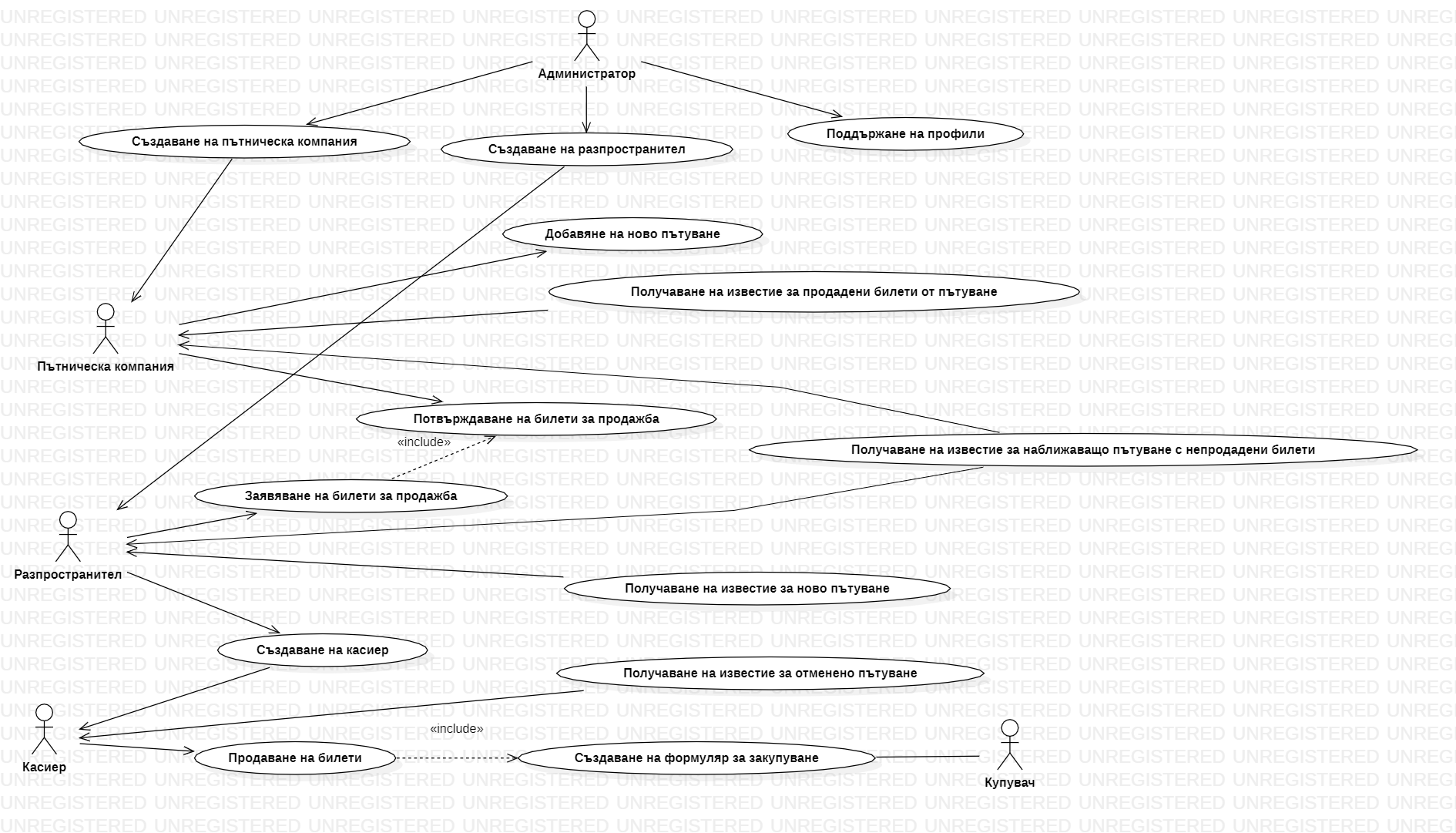
За изграждане на графичния потребителски интерфейс (GUI) на приложението ще се използва JavaFX, тъй като е лека, високопроизводителна графична платформа. Изгледите на интерфейса ще се създават с fxml файлове, които ще включват изображения, label-и, текстови полета, combobox-ове, бутони, table view-та и др. контроли. Ще бъдат подбрани подходящи цветни комбинации за приложението, за да не дразнят окото, но и да са визуално атрактивни. Ще има лек контраст между основните цветове, за да се разграничават по-лесно отделните модули на приложението. Ще се направи дизайн на лого, което ще е неизменна част от всеки изглед на приложението. Логото ще е името на пътническата компания, съчетано с подходящи минималистични изображения.

1.3. Проектиране на бизнес логика.

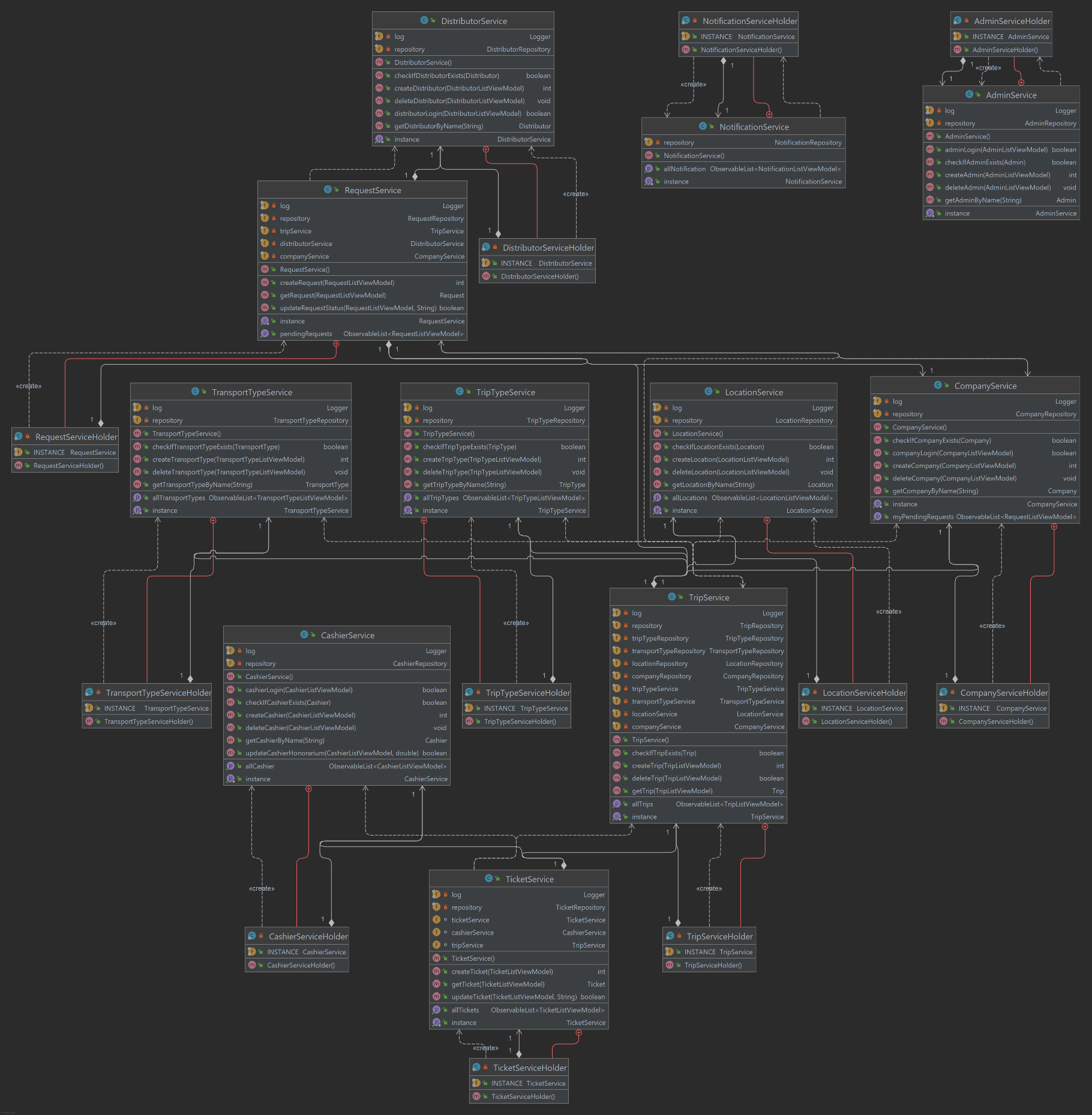
Бизнес логиката в приложението ще бъде реализирана така, че да осигурява изпълнението на всички функционални изисквания към системата. Те ще бъдат разпределени според типа на потребителя и в зависимост от това всеки потребител на системата ще изпълнява конкретни функционалности, реализирани от бизнес логиката.

1. UML диаграми.

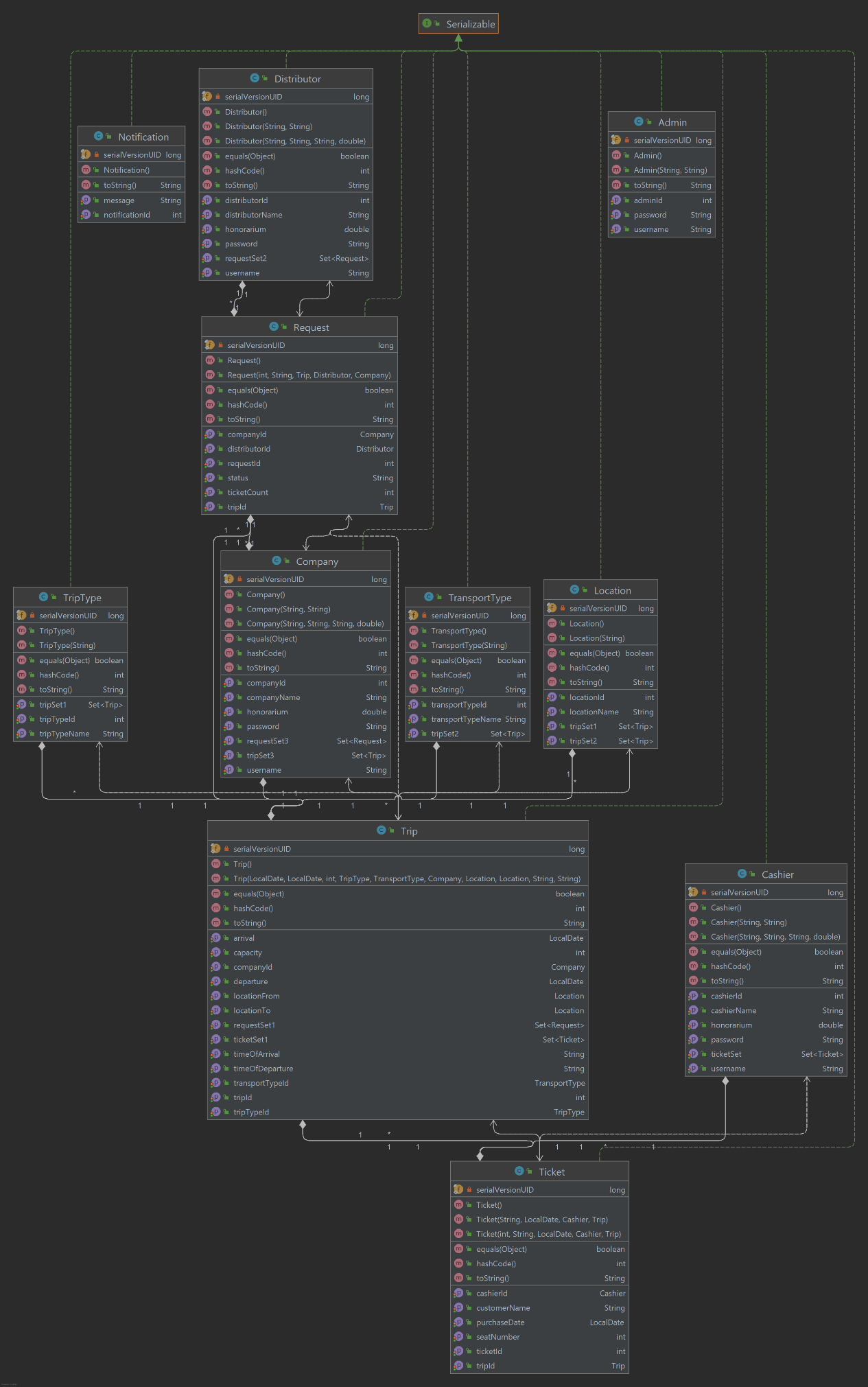
2.1. Use Case Diagram

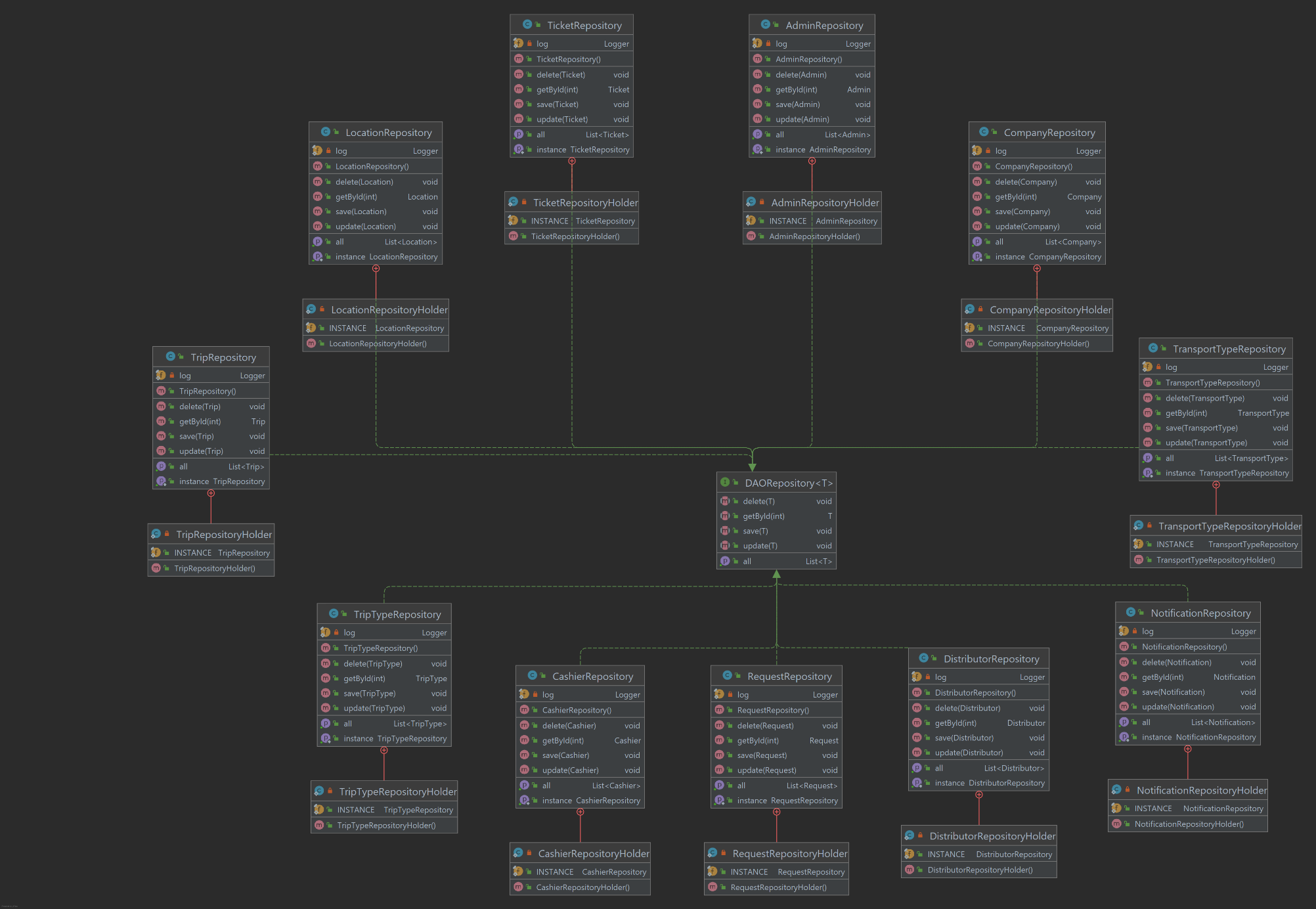


2.2. Class Diagram.

Services:

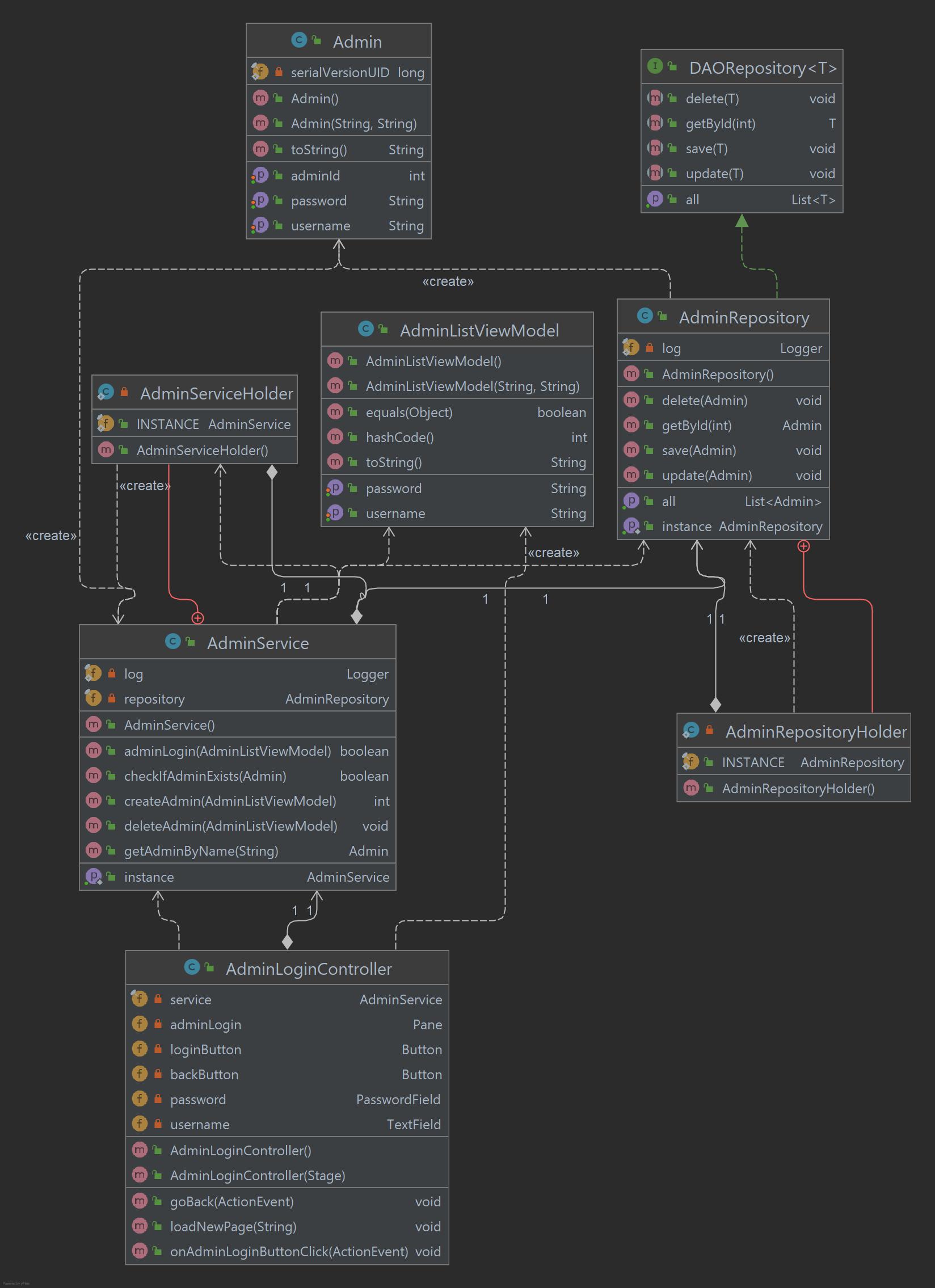
Entites:



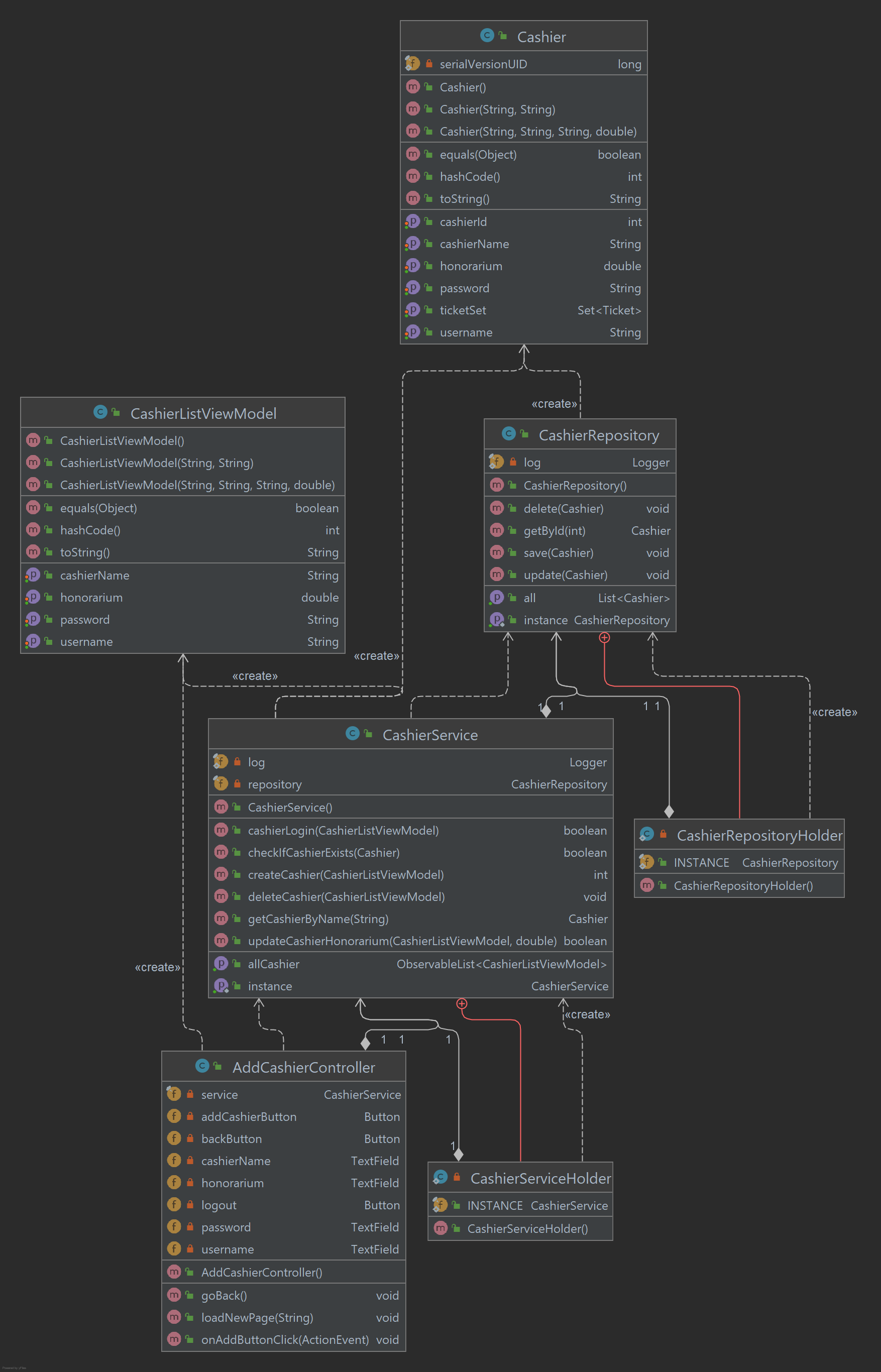
Repositories:

Models:

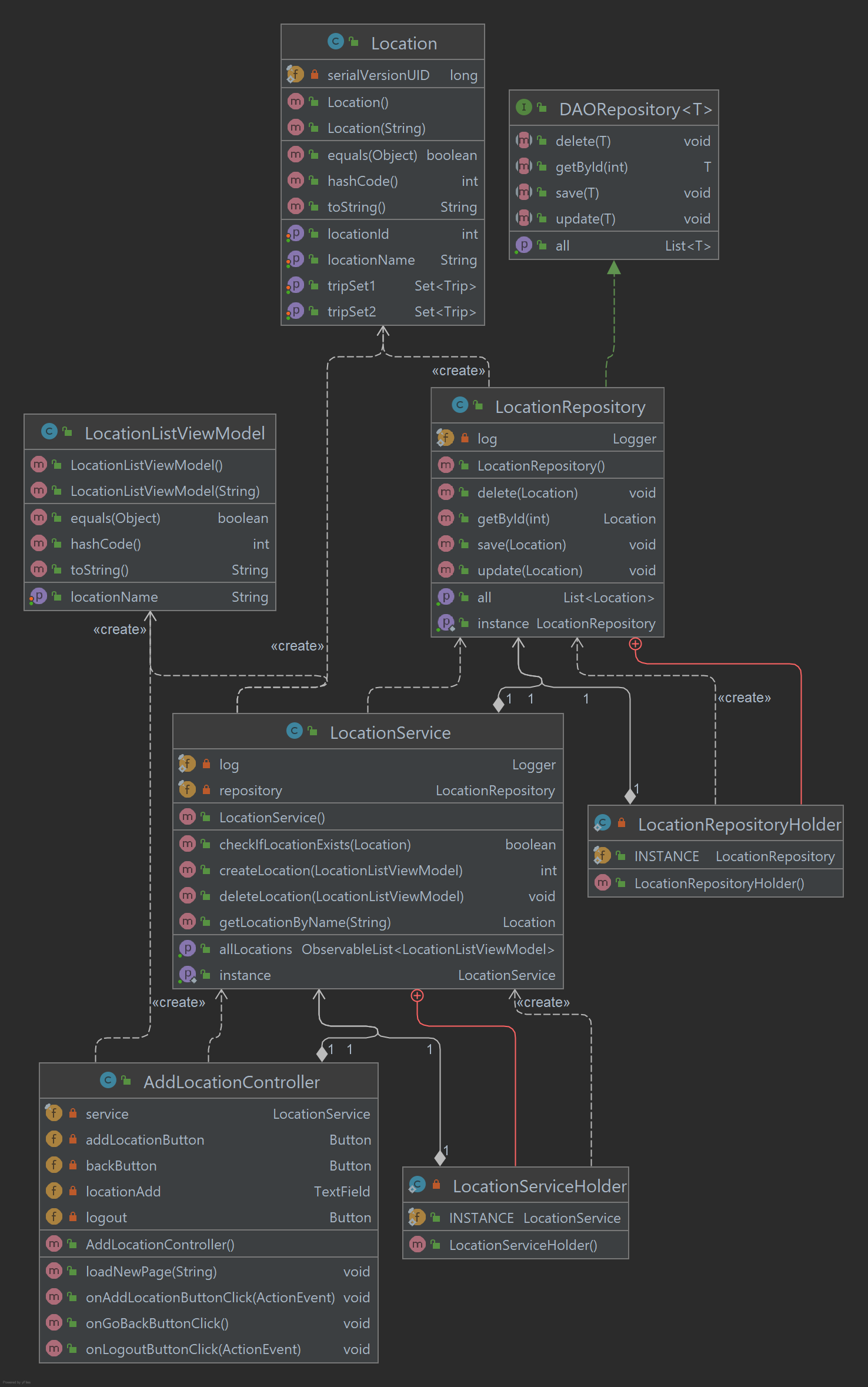
User (Admin) Log In:



Add User (Create Cashier):



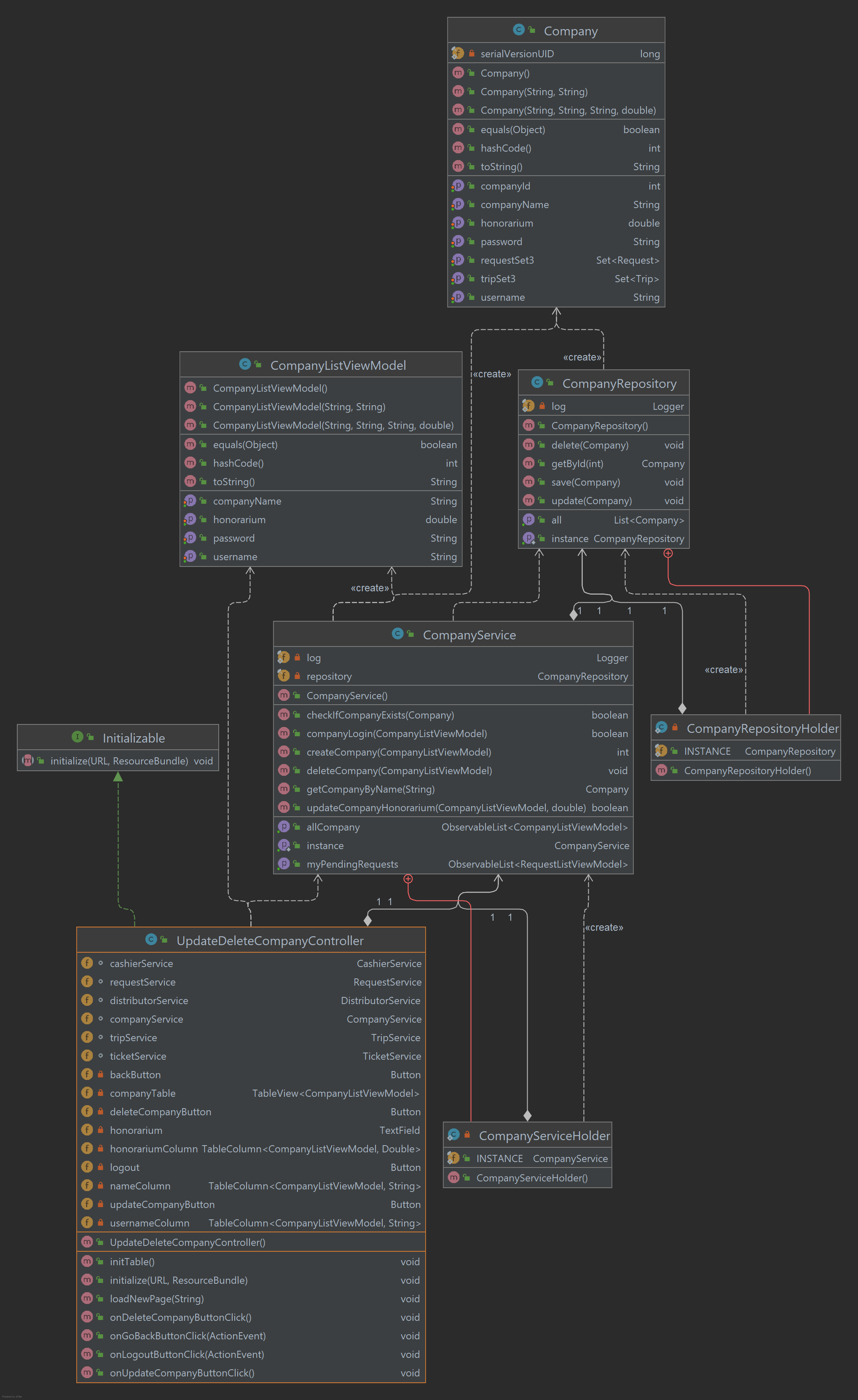
Add Location:



Update Request:

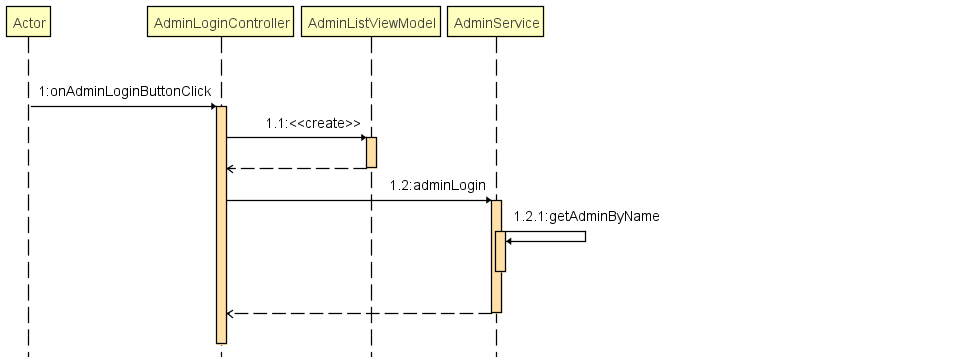


Update/Delete Company:

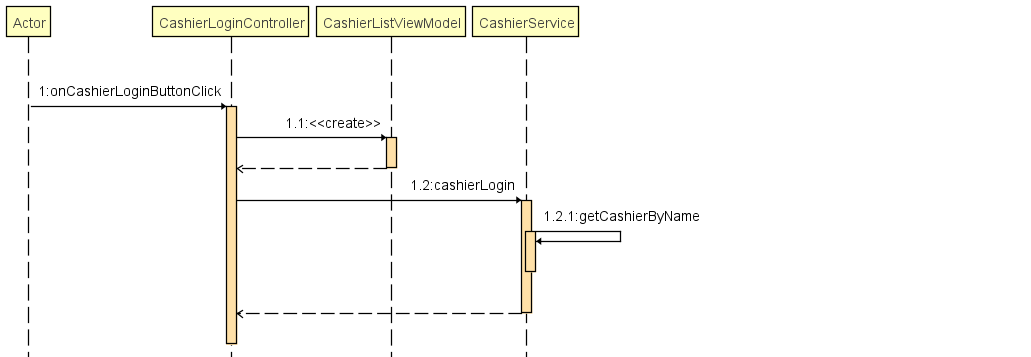


2.3. Sequence Diagram.

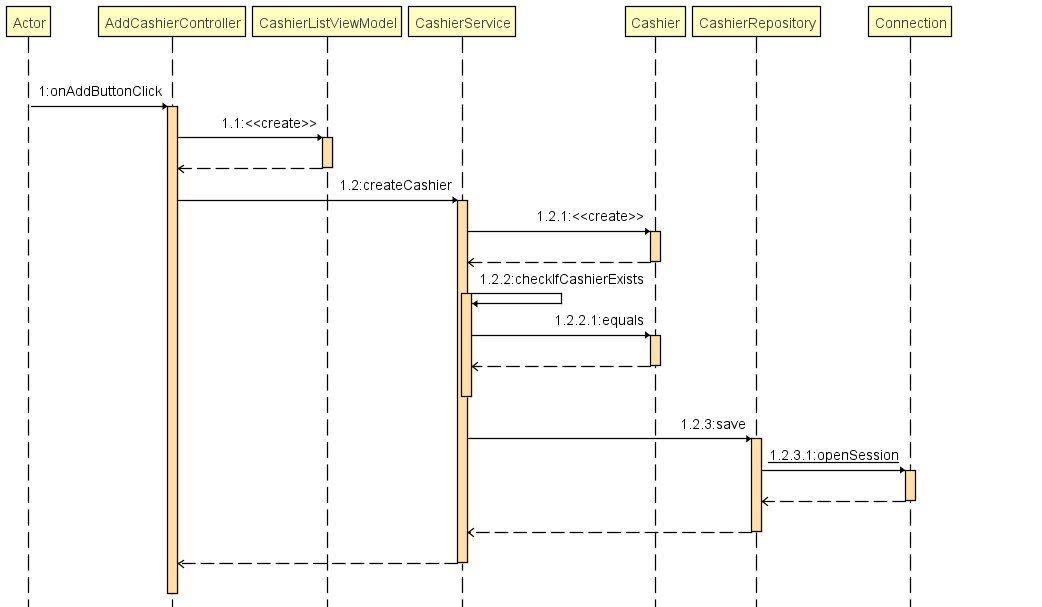
Admin Log In:



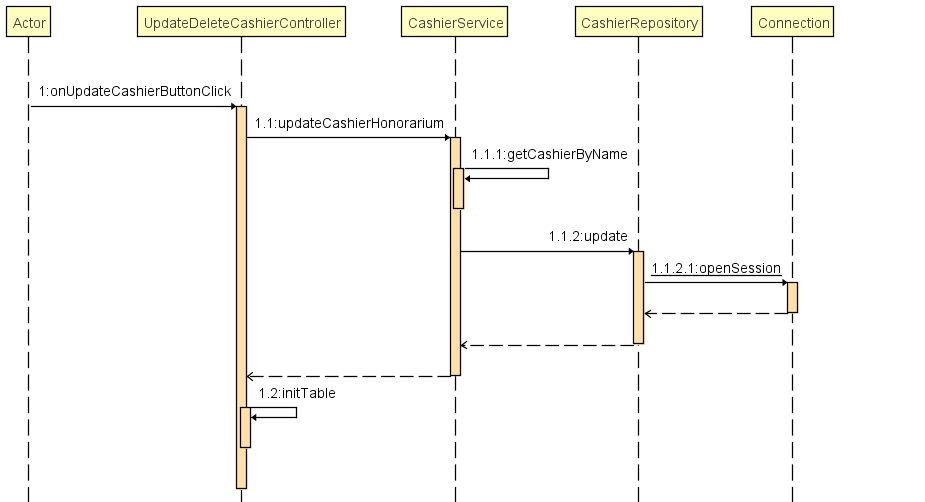
Cashier Log In:



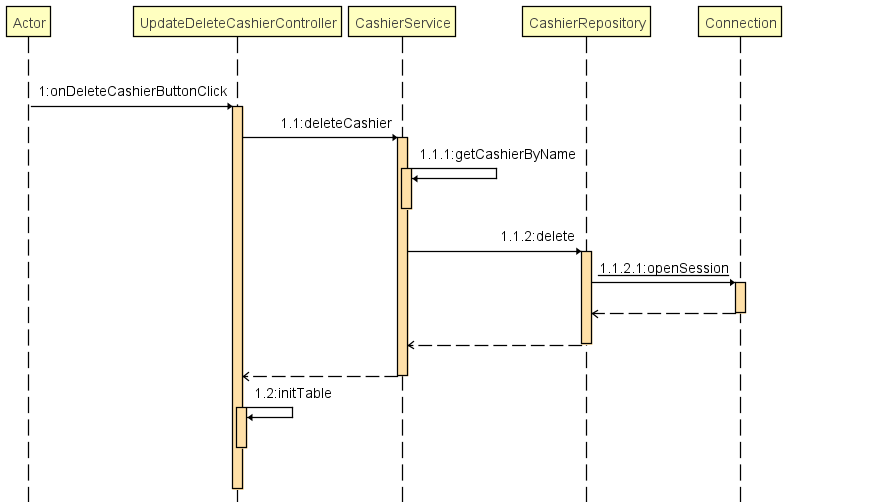
Add Cashier:



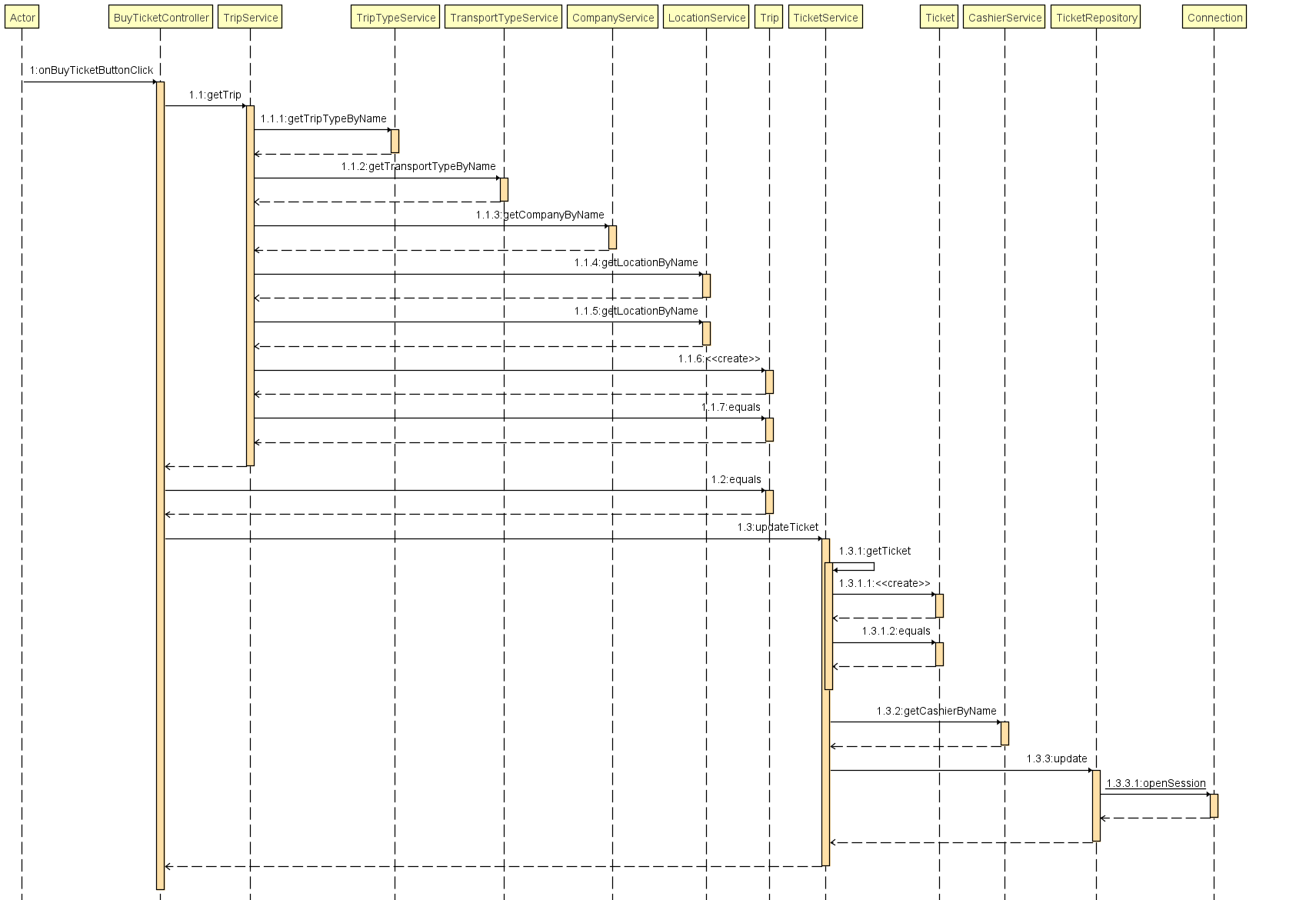
Update Cashier:



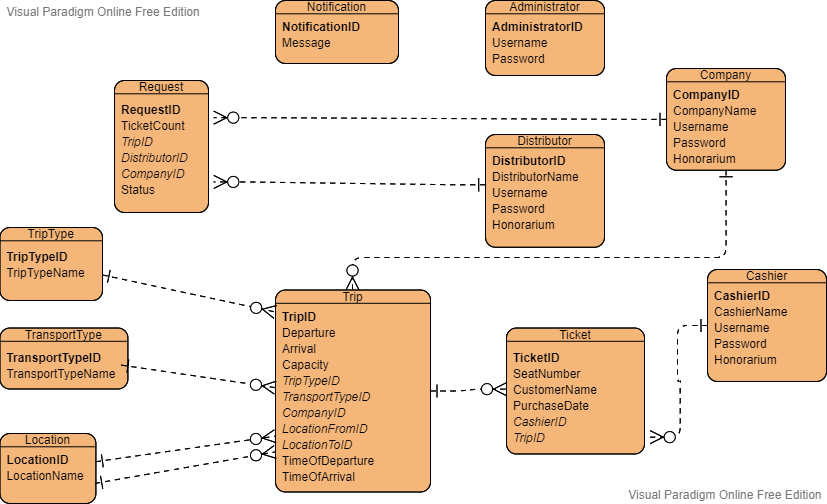
Delete Cashier:



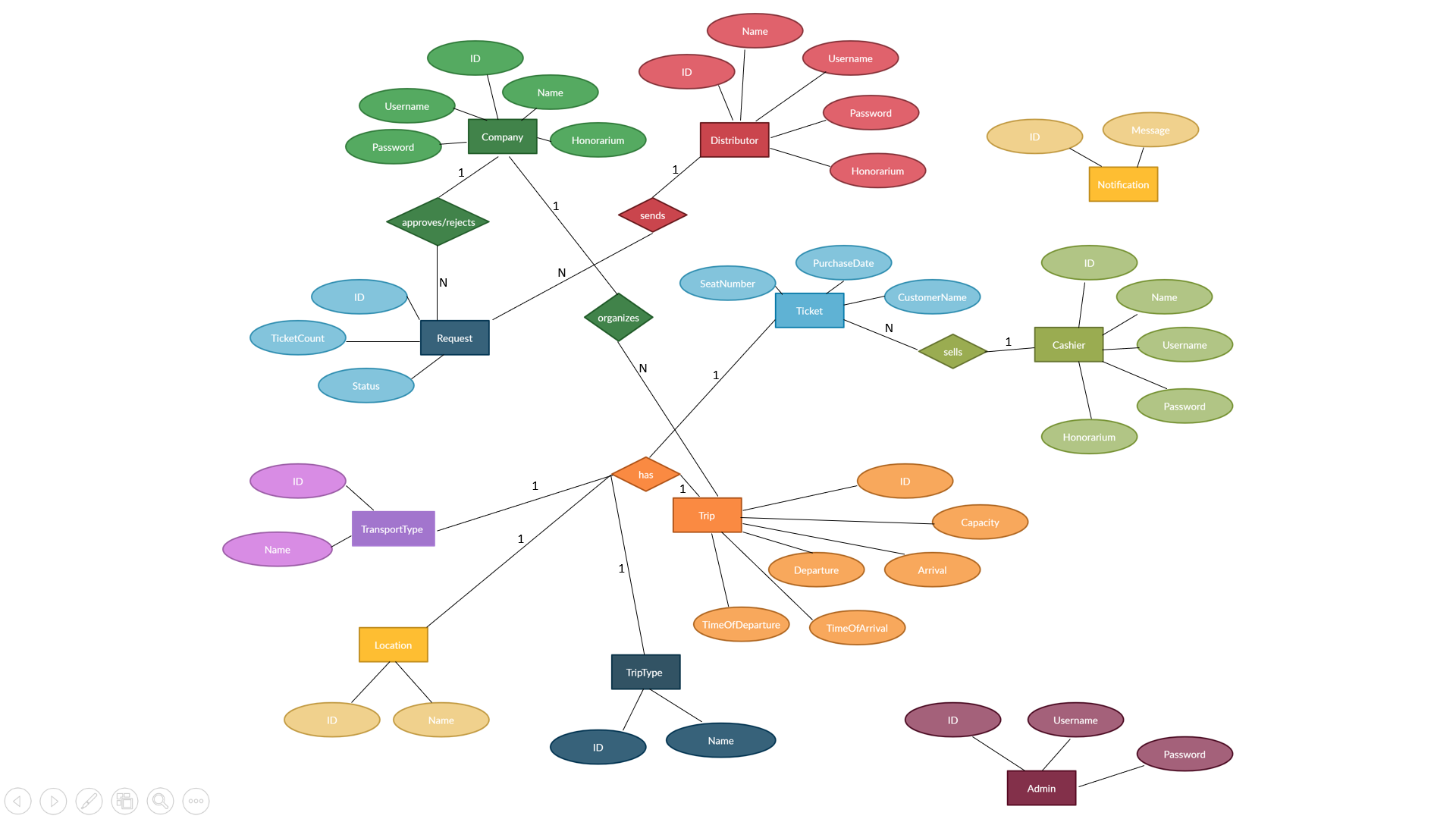
Buy (Add) Ticket:



1. Kонцептуален модел на БД.

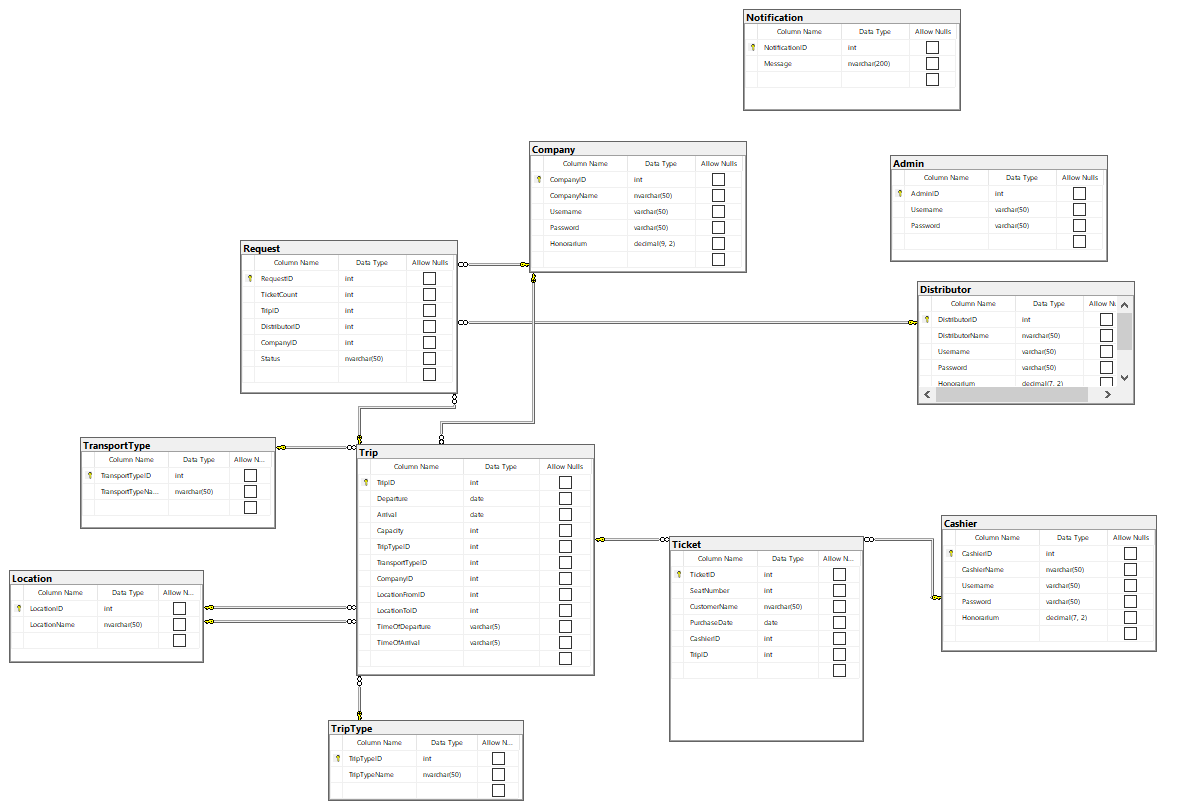


1. Модел на Чен.



1. **Реализация на системата.**
2. Реализация на базата от данни. Релационна схема и описание на таблиците.

За реализация на базата от данни е използван Microsoft SQL Server, защото включва професионален софтуер за управление на бази данни, който е лесен за използване и предоставя повече въможности от конкурентни продукти. MS SQL е подходящ за управление на големи бази от данни и разполага с редица опции за възстановяване на базата данни при случайна повреда, например загуба на мощност или неправилно изключване.

Релационен модел на БД

Таблици и описание

Картина, която съдържа маса

Описанието е генерирано автоматично

В таблица Admin се записват данните за потребител на системата от тип администратор. Таблицата съдържа поле AdminID от тип integer, което се увеличава автоматично (Auto Increment) с единица при всеки нововъведен администратор в системата. В таблицата се записват и неговите потребителско имe (Username) и парола (Password), с които потребителят може да се логне в системата. Те са от тип varchar (тъй като могат да включват само латински букви) и са с размер до 50 символа, за да е възможно съставянето на по-дълги и сложни комбинации от символи за по-голяма сигурност на данните.

Картина, която съдържа маса

Описанието е генерирано автоматично

В таблица Company се записват данните за потребител (клиент) на система от тип пътническа компания. Таблицата съдържа поле CompanyID от тип integer, което се увеличава автоматично (Auto Increment) с единица при всяка нововъведена компания в системата. В таблицата се записват наименованието на компанията (CompanyName), което е от тип nvarchar, тъй като може да е на английски, български, гръцки, арабски или др. езици. Типът данни nvarchar ни осигурява възможността за съхранение на Unicode символи с променлива дължина.

Полето CompanyName е с размер до 50 символа, което осигурява възможност за съхранение на по-дълги имена на компании. Таблицата съдържа потребителско имe (Username) и парола (Password) на компанията, с които да се логне в системата. Те са от тип varchar (тъй като могат да включват само латински букви) и са с размер до 50 символа, за да е възможно съставянето на по-дълги и сложни комбинации от символи за по-голяма сигурност на данните. Таблицата съдържа и поле Honorarium от тип decimal, в което се записва хонорарът на пътническата компания, който може да е десетично число до 9 цифри.

Картина, която съдържа маса

Описанието е генерирано автоматично

В таблица Distributor се записват данните за потребител (клиент) на система от тип разпространител. Таблицата съдържа поле DistributorID от тип integer, което се увеличава автоматично (Auto Increment) с единица при всеки нововъведен разпространител в системата. В таблицата се записват наименованието разпространителя (DistributorName), което е от тип nvarchar, тъй като може да е на английски, български, гръцки, арабски или др. езици. Типът данни nvarchar ни осигурява възможността за съхранение на Unicode символи с променлива дължина. Полето DistributorName е с размер до 50 символа, което осигурява възможност за съхранение на по-дълги наименования на разпространители. Таблицата съдържа потребителско имe (Username) и парола (Password) на разпространителя, с които да се логне в системата. Те са от тип varchar (тъй като могат да включват само латински букви) и са с размер до 50 символа, за да е възможно съставянето на по-дълги и сложни комбинации от символи за по-голяма сигурност на данните. Таблицата съдържа и поле Honorarium от тип decimal, в което се записва хонорарът на разпространителя, който може да е десетично число до 7 цифри.

Картина, която съдържа маса

Описанието е генерирано автоматично

В таблица Cashier се записват данните за потребител (клиент) на система от тип касиер. Таблицата съдържа поле CashierID от тип integer, което се увеличава автоматично (Auto Increment) с единица при всеки нововъведен касиер в системата. Трите имена на касиера се записват в полето CashierName, което е от тип nvarchar, тъй като имената на касиера може да са на английски, български, гръцки, арабски или др. език. Типът данни nvarchar ни осигурява възможността за съхранение на Unicode символи с променлива дължина. Полето CashierName е с размер до 50 символа, което осигурява възможност за съхранение на по-дълги имена на касиери. Таблицата съдържа и потребителско имe (Username) и парола (Password) на касиера, с които да се логне в системата. Те са от тип varchar (тъй като могат да включват само латински букви) и са с размер до 50 символа, за да е възможно съставянето на по-дълги и сложни комбинации от символи за по-голяма сигурност на данните. Таблицата съдържа и поле Honorarium от тип decimal, в което се записва хонорарът на касиера, който може да е десетично число до 7 цифри.

Картина, която съдържа маса

Описанието е генерирано автоматично

Таблицата Location съдържа поле LocationName, в което се записват местата на тръгване и/или пристигане при дадено пътуване. Полето е от тип nvarchar, което позволява въвеждането на локации на различни езици (английски, български, гръцки, арабски и др.). Дължината му е до 50 символа, което осигурява възможност за съхранение на по-дълги имена на локации. Другото поле на таблицата е LocationID, чрез което всяка локация се идентифицира автоматично с число integer. При всяка следваща нововъведена локация това поле се увеличава автоматично с единица (Auto Increment).

Картина, която съдържа маса

Описанието е генерирано автоматично

Таблицата TripType съдържа поле TripTypeName, в което се записват видовете пътувания, организирани от пътническите компании. Полето е от тип nvarchar, което позволява въвеждането на видове пътувания на различни езици (английски, български, гръцки, арабски и др.). Дължината му е до 50 символа, което осигурява възможност за съхранение на по-дълги наименования. Всеки вид пътуване има идентификационен номер TripTypeID от тип integer, който се увеличава автоматично с единица (Auto Increment) при всяко нововъведено такова.

Картина, която съдържа маса

Описанието е генерирано автоматично

Таблицата TransportType съдържа поле TransportTypeName, в което се записват видовете транспорт, с които се осъществяват пътуванията. Полето е от тип nvarchar, което позволява въвеждането на видове транспорт на различни езици (английски, български, гръцки, арабски и др.). Дължината му е до 50 символа, което осигурява възможност за съхранение на по-дълги наименования. Всеки вид транспорт има идентификационен номер TransportTypeID от тип integer, който се увеличава автоматично с единица (Auto Increment) при всяко нововъведено такова.

Картина, която съдържа маса

Описанието е генерирано автоматично

Tаблицата Trip съдържа данни за организирани пътувания. С поле TripID от тип integer се идентифицира всяко пътуване с число, увеличаващо се автоматично с единица (Auto Increment) при всяко нововъведено пътуване. В полетата Departure и Аrrival от тип date се записват съответно датата на заминаване и датата на пристигане. Полето Capacity от тип integer съдържа общия брой на билетите за пътуването. Таблицата съдържа полета TripTypeID, TransportTypeID, CompanyID, LocationFromID, LocationToID, чрез които може да достъпи информация от другите таблици съответно за вида на пътуване, вида транспорт, компанията, която организира пътуването, началната и крайната му локация. В полетата TimeOfDeparture и TimeOfArrival се записва часът на заминаване и пристигане и са от тип varchar.

Картина, която съдържа маса

Описанието е генерирано автоматично

Таблицата Ticket съдържа информация за закупен билет от клиент за дадено пътуване. Тя се използва като формуляр за закупуване (Purchase Form), който касиерът изготвя за всеки клиент, когато закупува билет за пътуване. Таблицата съдържа поле TicketID от тип integer, което записва поредния идентификационен номер на всеки закупен билет. То се увеличава автоматично с единица (Auto Increment) при всеки следващ закупен. В таблицата се включва и поле SeatNumber от тип integer, в което се записва номера на мястото на клиента в превозното средство, с което ще пътува. В полето CustomerName се записват трите имена на клиента, закупил билета. То е от тип nvarchar, тъй като имената му може да са на английски, български, гръцки, арабски или др. език. Типът данни nvarchar ни осигурява възможността за съхранение на Unicode символи с променлива дължина. Полето е с размер до 50 символа, което осигурява възможност за съхранение на по-дълги имена на клиенти. Таблицата Ticket съдържа и поле PurchaseDate от тип date, в което се въвежда датата на закупуване на билета. Полетата CashierID и TripID от тип integer позволяват достъпването на информация от таблиците Cashier и Trip, която би била полезна за купувача на билета. Например чрез идентификационния номер на касиера може да се извлече информация за неговото име, а чрез идентификационния номер на пътуването – за вида на пътуването, транспорта, мястото и часа на заминаване/пристигане, пътническата компания.

Картина, която съдържа маса

Описанието е генерирано автоматично

Таблицата Request съдържа данни за заявка на билети. Всяка заявка има идентификационен номер RequestID от тип integer, който се увеличава автоматично (Auto Increment) с единица при всяка новопостъпила заявка. Когато първоначално предоставените билети се изчерпат, в полето TicketCount oт тип integer се записва броят на заявените билети от разпространителя за дадено пътуване, за което се получава информация чрез полето TripID от тип integer, съдържащ идентификационния му номер. В полето DistributorID от тип integer се записва идентификационния номер на разпространителя, изпращащ заявката, чрез който може да се извлезе информация за разпостранителя. В полето CompanyID от тип integer се записва идентификационния номер на пътническата компания, до която е изпратена заявката за билети и компанията следва да я одобри или откаже. Чрез CompanyID може да се извлезе информация за компанията. В полето Status от тип nvarchar се записва текущото състояние на поръчката (в очакване /одобрена/отказана). След като компанията одобри или откаже заявката, полето Status се променя.

Картина, която съдържа маса

Описанието е генерирано автоматично

В таблица Notification се записват данни за известия. Таблицата съдържа поле NotificationID от тип integer, число, с което се идентифицира всяко известие. То се увеличава автоматично с единица (Auto Increment) при всяко нововъведено известие. В полето Message се записва съобщението (текста) на известието. То е от тип nvarchar, за да се въвежда текст на английски, български, гръцки, арабски или др. език. Типът данни nvarchar ни осигурява възможността за съхранение на Unicode символи с променлива дължина. Полето е с размер до 200 символа, предоставящо възможност за въвеждане на по-дълги известяващи съобщения.

1. Реализация на слоя за работа с базата данни.

За реализация на слоя за работа с базата данни са разработени различни подходи. Два от по-известните са JDBC

и JPA (Hibernate).

По-ранните технологии като JDBC (Java Database Connectivity) срещат проблема с несъотвестствието на импеданса (разлика между обектно-ориентираните (JAVA) и релационните технологии (RDBMS)). Решението на този проблем се осъществява чрез въвеждането на персистентен (постоянен) слой, който капсулира достъпа до базата данни от бизнес логиката. JPA (Java Persistence API) е рамка, предназначена за управление на релационни данни, използвайки слоя за постоянство в Java приложения. JPA позволява свързването на Java обекти към записи в релационна база данни. Hibernate е най-популярната реализация на JPA.

Някои разлики между JDBC и JPA:

* JPA е по-подходящ за по-сложни приложения, докато JDBC се избира за приложения с по-проста база данни.
* JDBC зависи от базата данни, което означава, че трябва да бъдат написани различни скриптове за различните видове бази данни. От друга страна, при JPA един и същ код може да се използва в различни бази данни с малко (или никакви) промени.
* Основното предимство на JPA пред JDBC е, че се използват обектно-ориентирани принципи, а в резултат на това разработването може да бъде завършено по-бързо.
* JDBC не дава гъвкавост за стесняване на търсенето и прави приложението тежко и мудно, докато разработката на JPA - Hibernate опростява взаимодействието на базата данни и Java приложението.

Комуникацията с базата данни е реализирана чрез Hibernate. Обектите на Java и сървърът на бази данни взаимодействат помежду си чрез Hibernate. Hibernate e мощен и лек инструмент за обектно-релационно картографиране (ORM). С прости думи, той създава картографиране между Java класове и таблици в релационни бази данни, също и между Java и SQL типове данни. Може да се използва и за заявки и извличане на данни чрез генериране на SQL повиквания, т.е. Hibernate автоматично генерира повечето заявки (освен по-сложните), за разлика от JDBC. Следователно програмистът е освободен от ръчното боравене с набори от резултати и преобразуване на обекти.

Програмен код за реализация на връзката с базата данни (модул Access от слоя) – class Connection:

package bg.tu\_varna.sit.oop\_project\_demo.data.access;  
  
import org.apache.log4j.Logger;  
import org.hibernate.Session;  
import org.hibernate.SessionFactory;  
import org.hibernate.cfg.Configuration;  
  
public class Connection {  
 private static final Logger *log* = Logger.*getLogger*(Connection.class);  
 private static SessionFactory *sessionFactory*;  
  
 static {  
 try {  
 *sessionFactory* = new Configuration().configure().buildSessionFactory();  
 } catch (Throwable ex) {  
 *log*.error("Initial SessionFactory creation failed" + ex);  
 }  
 }  
 public static Session openSession() {  
 return *sessionFactory*.openSession();  
 }  
 public static void closeSession() {  
 *sessionFactory*.close();  
 }  
}

Част от слоя за работа с базата данни е и модулът Entities. Entities са класове на обекти, които се съхраняват в базата данни. Всяко entity отговаря на таблица в базата данни. Следователно класът съдържа името на таблицата, колоните ѝ, типовете данни, които се записват и връзките ѝ с другите таблици. Entities осигуряват обектно-релационен интерфейс между обектно-ориентирания код и структурата на релационната таблица на базата данни.

Пример как е асоциирано едно Entity с едноименна таблица от базата данни:

Company:

@Table(name = "Company")  
@Entity  
public class Company {  
 private static final long *serialVersionUID*=1L;  
 @Id  
 @GeneratedValue(strategy = GenerationType.*IDENTITY*)  
 @Column(name = "companyId", nullable = false)  
 private int companyId;  
  
 @Column(name = "companyName", nullable = false)  
 private String companyName;  
  
 @Column(name = "username", nullable = false)  
 private String username;  
  
 @Column(name = "password", nullable = false)  
 private String password;  
  
 @Column(name = "honorarium", nullable = false)  
 private double honorarium;  
  
 @OneToMany(mappedBy = "companyId")  
 Set<Request> requestSet3;  
  
 @OneToMany(mappedBy = "companyId")  
 Set<Trip> tripSet3;

Посочено е името на таблицата, която отговаря на Entity класа Company. Съответно са конфигурирани и отделните полета – companyID, companyName, username, password, honorarium. Как изглежда таблицата в базата данни:

Картина, която съдържа маса

Описанието е генерирано автоматично

Пример за Trip:

Entity/клас на обект:

@Table(name = "Trip")  
@Entity  
public class Trip {  
 private static final long *serialVersionUID*=1L;  
 @Id  
 @GeneratedValue(strategy = GenerationType.*IDENTITY*)  
 @Column(name = "tripId", nullable = false)  
 private int tripId;  
  
 @Column(name = "departure", nullable = false)  
 private LocalDate departure;  
  
 @Column(name = "arrival", nullable = false)  
 private LocalDate arrival;  
  
 @Column(name = "capacity", nullable = false)  
 private int capacity;  
  
 @ManyToOne  
 @JoinColumn(name = "tripTypeId", nullable = false)  
 private TripType tripTypeId;  
  
 @ManyToOne  
 @JoinColumn(name = "transportTypeId", nullable = false)  
 private TransportType transportTypeId;  
  
 @ManyToOne  
 @JoinColumn(name = "companyId", nullable = false)  
 private Company companyId;  
  
 @ManyToOne  
 @JoinColumn(name = "locationFromId", nullable = false)  
 private Location locationFrom;  
  
 @ManyToOne  
 @JoinColumn(name = "locationToId", nullable = false)  
 private Location locationTo;  
  
 @Column(name = "timeOfDeparture", nullable = false)  
 private String timeOfDeparture;  
  
 @Column(name = "timeOfArrival", nullable = false)  
 private String timeOfArrival;  
  
 @OneToMany(mappedBy = "tripId")  
 Set<Request> requestSet1;  
  
 @OneToMany(mappedBy = "tripId")  
 Set<Ticket> ticketSet1;

В базата данни:

Картина, която съдържа маса

Описанието е генерирано автоматично

Аналогично са реализирани Entity-тата, съответстващи на всички останали таблици в базата данни.

Третият модул от слоя за работа с базата данни е модул Repositories. Част от него е интерфейсът DAORepository, съдържащ методи, които могат да се извършват върху обекти, съхранявани в базата данни – save, update, delete, getById, getAll. За всеки клас на обект (Entity) има отговарящо Repository, което имплементира методите на DAORepository, като по този начин се осъщестява комуникацията с базата данни.

интерфейс DAORepository:

public interface DAORepository<T> {  
 void save(T obj);  
 void update(T obj);  
 void delete(T obj);  
 T getById(int id);  
 List<T> getAll();  
}

Пример за имплементиране на интерфейса DAORepository от CompanyRepository:

public class CompanyRepository implements DAORepository<Company>{  
 private static final Logger *log* = Logger.*getLogger*(CompanyRepository.class);  
  
 public static CompanyRepository getInstance() {  
 return CompanyRepository.CompanyRepositoryHolder.*INSTANCE*;  
 }  
 private static class CompanyRepositoryHolder {  
  
 public static final CompanyRepository *INSTANCE* = new CompanyRepository();  
 }  
  
 @Override  
 public void save(Company obj) {  
 Session session = Connection.*openSession*();  
 Transaction transaction = session.beginTransaction();  
 try{  
 session.save(obj);  
 *log*.info("Company saved successfully");  
 } catch (Exception ex) {  
 *log*.error("Company save error. " + ex.getMessage());  
 } finally {  
 transaction.commit();  
 }  
 }  
  
 @Override  
 public void update(Company obj) {  
 Session session = Connection.*openSession*();  
 Transaction transaction = session.beginTransaction();  
 try {  
 session.update(obj);  
 } catch (Exception e) {  
 e.printStackTrace();  
 } finally {  
 transaction.commit();  
 session.close();  
 }  
 }  
  
 @Override  
 public void delete(Company obj) {  
 Session session = Connection.*openSession*();  
 Transaction transaction = session.beginTransaction();  
 try {  
 session.delete(obj);  
 } catch (Exception e) {  
 e.printStackTrace();  
 } finally {  
 transaction.commit();  
 session.close();  
 }  
 }  
  
 @Override  
 public Company getById(int id) {  
 Session session = Connection.*openSession*();  
 Transaction transaction = session.beginTransaction();  
 Company company = new Company();  
 try {  
 String jpql = "SELECT a FROM Company a WHERE companyId =" + String.*valueOf*(id);  
 company=session.createQuery(jpql, Company.class).getSingleResult();  
 } catch (Exception e) {  
 e.printStackTrace();  
 } finally {  
 transaction.commit();  
 session.close();  
 }  
 return company;  
 }  
  
 @Override  
 public List<Company> getAll() {  
 Session session = Connection.*openSession*();  
 Transaction transaction = session.beginTransaction();  
 List<Company> companies = new LinkedList<>();  
 try {  
 String jpql = "SELECT r FROM Company r";  
 companies.addAll(session.createQuery(jpql, Company.class).getResultList());  
 *log*.info("Got all companies");  
 } catch (Exception ex){  
 *log*.error("Get all companies error. " + ex);  
 } finally {  
 transaction.commit();  
 }  
 return companies;  
 }  
}

1. Реализация на бизнес логика и графичен интерфейс.

Бизнес логиката е реализирана на езика Java, а графичния интерфейс с помощта на JavaFX.

Всеки изглед на графичния интерфейс има съответен контролер, който реагира на събития (например натискане на бутон), и по този начин контролерът осъществява комуникацията между графичния интерфейс и бизнес логиката.

Изгледът на началната страница на приложението за пътнически превози се състои от следните компоненти: изображение, което илюстрира за какво ще се използва приложението, лого, слоган и 4 бутона, предоставящи възможност на потребителя да избере като какъв тип потребител да влезе в системата – администратор, пътническа компания, разпространител или касиер. За да бъде успешно логването му, той трябва да бъде предварително регистриран.

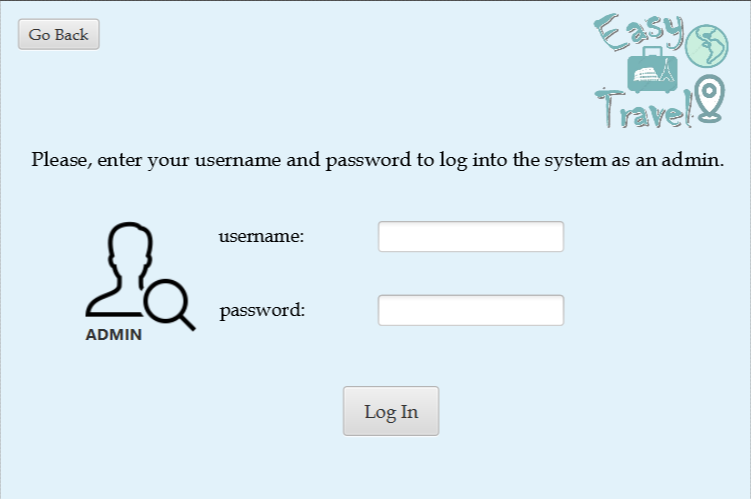
Начален изглед на приложението:

hello-view.fxml

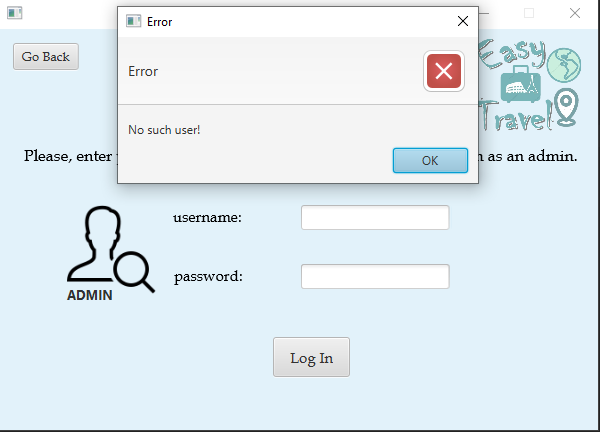
<?xml version="1.0" encoding="UTF-8"?>  
  
<?import javafx.scene.control.Button?>  
<?import javafx.scene.control.Label?>  
<?import javafx.scene.image.Image?>  
<?import javafx.scene.image.ImageView?>  
<?import javafx.scene.layout.Pane?>  
<?import javafx.scene.text.Font?>  
  
<Pane cache="true" maxHeight="-Infinity" maxWidth="-Infinity" minHeight="-Infinity" minWidth="-Infinity" prefHeight="400.0" prefWidth="600.0" style="-fx-background-color: #E2F2FA;" xmlns="http://javafx.com/javafx/17" xmlns:fx="http://javafx.com/fxml/1" fx:controller="bg.tu\_varna.sit.oop\_project\_demo.presentation.controllers.HelloController">  
 <children>  
 <Button fx:id="cashierButton" layoutX="454.0" layoutY="343.0" mnemonicParsing="false" onAction="#cashierLogin" prefHeight="40.0" prefWidth="141.0" text="Log In As Cashier" textAlignment="CENTER" />  
 <ImageView fx:id="iv1" fitHeight="253.0" fitWidth="365.0" layoutX="123.0" layoutY="36.0" pickOnBounds="true" preserveRatio="true">  
 <image>  
 <Image url="@../images/background.png" />  
 </image>  
 </ImageView>  
 <Button fx:id="adminButton" layoutX="6.0" layoutY="343.0" mnemonicParsing="false" onAction="#adminLogin" prefHeight="40.0" prefWidth="141.0" text="Log In As Administrator" />  
 <Button fx:id="distributorButton" layoutX="304.0" layoutY="343.0" mnemonicParsing="false" onAction="#distributorLogin" prefHeight="40.0" prefWidth="141.0" text="Log In As Distributor" />  
 <Button fx:id="companyButton" layoutX="153.0" layoutY="343.0" mnemonicParsing="false" onAction="#companyLogin" prefHeight="40.0" prefWidth="141.0" text="Log In As Company" textAlignment="CENTER" />  
 <Label fx:id="label1" alignment="CENTER" contentDisplay="CENTER" layoutX="-136.0" layoutY="14.0" prefHeight="59.0" prefWidth="600.0" text="Welcome to" textAlignment="CENTER" textFill="#302222" textOverrun="LEADING\_WORD\_ELLIPSIS">  
 <font>  
 <Font name="Broadway" size="45.0" />  
 </font>  
 </Label>  
 <ImageView fx:id="iv2" fitHeight="141.0" fitWidth="173.0" layoutX="419.0" layoutY="5.0" pickOnBounds="true" preserveRatio="true">  
 <image>  
 <Image url="@../images/logoEasyTravel2.png" />  
 </image>  
 </ImageView>  
 <Label fx:id="label2" layoutX="195.0" layoutY="261.0" text="Travel Explore Have Fun">  
 <font>  
 <Font name="Book Antiqua" size="16.0" />  
 </font>  
 </Label>  
 <Label fx:id="label3" layoutX="129.0" layoutY="299.0" text="Everything is with us!">  
 <font>  
 <Font name="Broadway" size="21.0" />  
 </font>  
 </Label>  
 <ImageView fx:id="iv3" fitHeight="65.0" fitWidth="97.0" layoutX="283.0" layoutY="285.0" pickOnBounds="true" preserveRatio="true">  
 <image>  
 <Image url="@../images/easy.png" />  
 </image>  
 </ImageView>  
 <ImageView fx:id="iv4" fitHeight="32.0" fitWidth="31.0" layoutX="163.0" layoutY="258.0" pickOnBounds="true" preserveRatio="true">  
 <image>  
 <Image url="@../images/travel.png" />  
 </image>  
 </ImageView>  
 <ImageView fx:id="iv5" fitHeight="30.0" fitWidth="31.0" layoutX="247.0" layoutY="255.0" pickOnBounds="true" preserveRatio="true">  
 <image>  
 <Image url="@../images/explore.png" />  
 </image>  
 </ImageView>  
 <ImageView fx:id="iv6" fitHeight="59.0" fitWidth="31.0" layoutX="336.0" layoutY="255.0" pickOnBounds="true" preserveRatio="true">  
 <image>  
 <Image url="@../images/havefun.png" />  
 </image>  
 </ImageView>  
 </children>  
</Pane>

При избор на някой от четирите бутона се зарежда следващият изглед в зависимост от типа потребител.

Изглед при логване като администратор:



От потребителя се изисква да въведе своите потребителско име и парола и след това да натисне бутона за вход в системата. При това контролерът (AdminLoginController) подава въведените от потребителя данни на модела (AdminListViewModel), който ги предава на бизнес логиката (AdminService) за обработка и проверка. След това те се предават на AdminRepository, което осъществява комуникацията с базата данни и изпраща заявка до нея. Ако валидацията се извърши успешно, т.е. потребителското име и паролата съвпадат със запис от базата данни, потребителят (в случая администраторът) се допуска до следващия съответен изглед с функционалностите, до които има достъп. Ако потребителят е въвел некоректни данни, се извежда съобщение за грешка, че не е открит такъв потребител в системата. Полетата за потребителско име и парола се изчистват, за да е удобно на потребителя да опита повторно да влезе в системата, въвеждайки други данни.



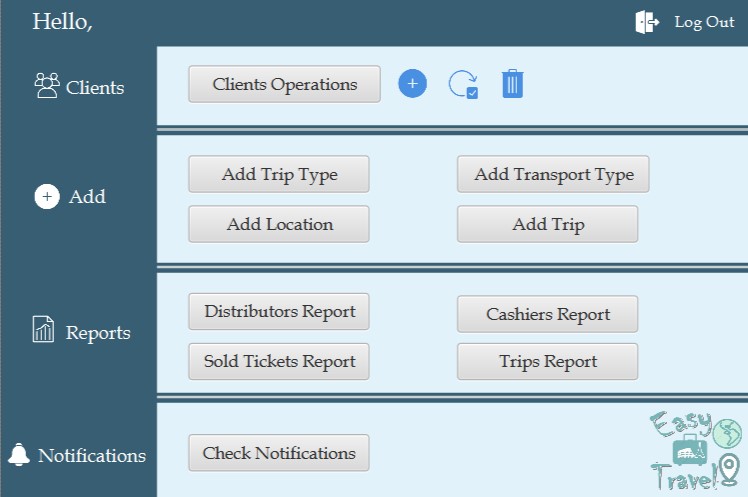
AdminLoginController:

package bg.tu\_varna.sit.oop\_project\_demo.presentation.controllers;  
  
import bg.tu\_varna.sit.oop\_project\_demo.business.services.AdminService;  
import bg.tu\_varna.sit.oop\_project\_demo.presentation.models.AdminListViewModel;  
import javafx.event.ActionEvent;  
import javafx.fxml.FXML;  
import javafx.fxml.FXMLLoader;  
import javafx.scene.Parent;  
import javafx.scene.Scene;  
import javafx.scene.control.\*;  
import javafx.scene.layout.Pane;  
import javafx.stage.Stage;  
import static bg.tu\_varna.sit.oop\_project\_demo.common.Constants.User.*trackUser*;  
import static bg.tu\_varna.sit.oop\_project\_demo.common.Constants.User.*loggedAdminUsername*;  
import static bg.tu\_varna.sit.oop\_project\_demo.common.Constants.View.*ADMIN\_VIEW*;  
import static bg.tu\_varna.sit.oop\_project\_demo.common.Constants.View.*HELLO\_VIEW*;  
  
public class AdminLoginController {  
 private final AdminService service=AdminService.*getInstance*();  
 @FXML  
 private Pane adminLogin;  
  
 @FXML  
 private Button loginButton;  
  
 @FXML  
 private Button backButton;  
  
 @FXML  
 private PasswordField password;  
  
 @FXML  
 private TextField username;  
  
 public AdminLoginController() {  
 }  
  
 public AdminLoginController(Stage stage) {  
 }  
  
 public void onAdminLoginButtonClick(ActionEvent actionEvent) {  
 AdminListViewModel adminToLogIn=new AdminListViewModel(username.getText(),password.getText());  
 if(service.adminLogin(adminToLogIn))  
 {  
 try {  
 *trackUser* = 1;  
 *loggedAdminUsername* = adminToLogIn.getUsername();  
 Stage s = (Stage) loginButton.getScene().getWindow();  
 s.close();  
 FXMLLoader fxmlLoader = new FXMLLoader(getClass().getResource(*ADMIN\_VIEW*));  
 Stage stage = new Stage();  
 Parent root1 = (Parent) fxmlLoader.load();  
 stage.setScene(new Scene(root1));  
 stage.setResizable(false);  
 stage.setWidth(600);  
 stage.setHeight(400);  
 stage.show();  
 } catch(Exception e) {  
 e.printStackTrace();  
 }  
 }  
 else{  
 Alert alert=new Alert(Alert.AlertType.*INFORMATION*,"No such user!", ButtonType.*OK*);

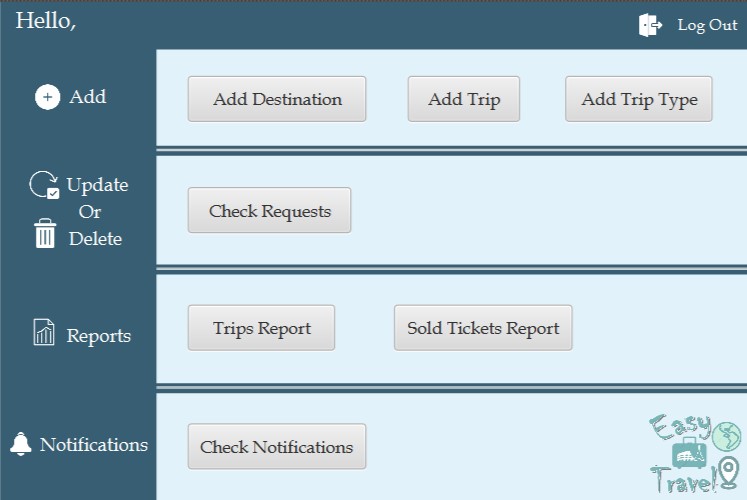
username.setText("");  
 password.setText("");  
 }  
 }  
  
 public void goBack(ActionEvent actionEvent){  
 loadNewPage(*HELLO\_VIEW*);  
 }  
  
 public void loadNewPage(String path){  
 try {  
 Stage s = (Stage) loginButton.getScene().getWindow();  
 s.close();  
 FXMLLoader fxmlLoader = new FXMLLoader(getClass().getResource(path));  
 Stage stage = new Stage();  
 Parent root1 = (Parent) fxmlLoader.load();  
 stage.setScene(new Scene(root1));  
 stage.setResizable(false);  
 stage.setWidth(615);  
 stage.setHeight(440);  
 stage.show();  
 } catch(Exception e) {  
 e.printStackTrace();  
 }  
 }  
}

При успешно логване в системата се зарежда съответният изглед според типа потребител. Изгледите са различни, защото всеки тип потребител има различни функционалности, а именно те разделят потребителите на типове.

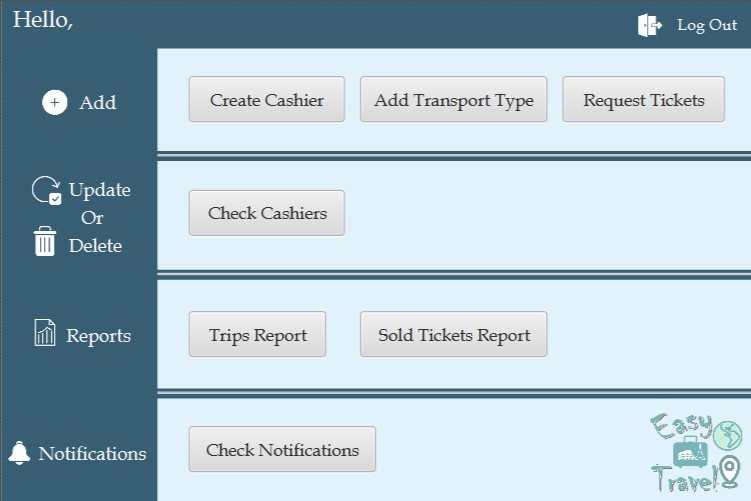
Изглед на администратор:



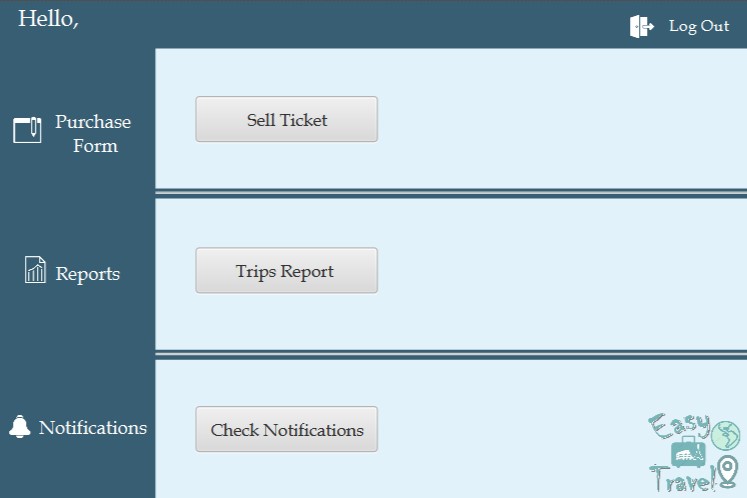
Изглед на пътническа компания:



Изглед на разпространител:



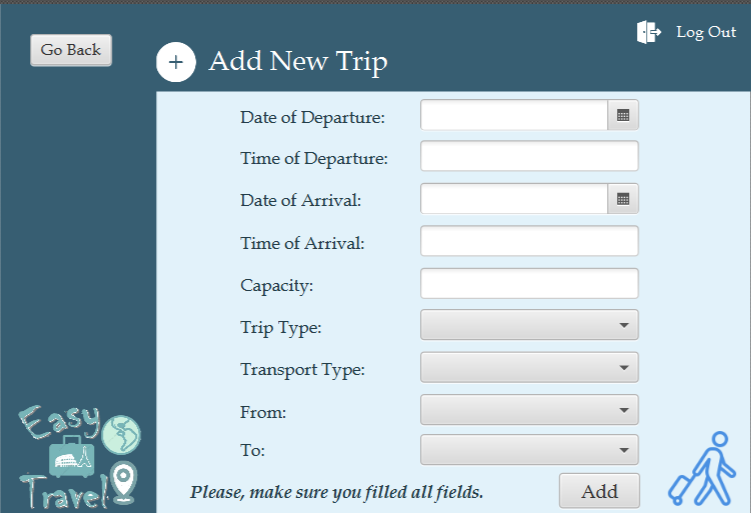
Изглед на касиер:



Всеки от потребителите може да избира желана функционалност, посредством бутоните. След това съответния контролер ще създаде обект от тип модел, който ще се предаде на бизнес логиката, която отговаря за обработката и валидацията на данните, съзадава се обект от тип Entity и чрез репозиторито се осъществява комуникацията с базата данни. По този начин се изпълняват всички функционалности на системата.

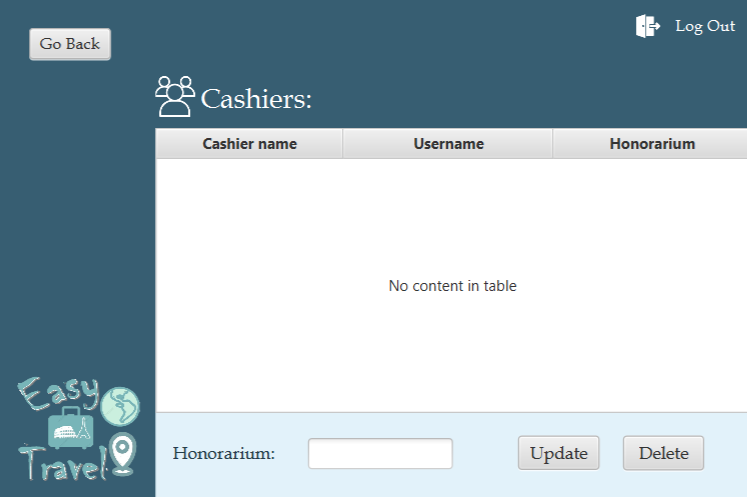
Примерен изглед за добавяне:

* Добавяне на пътуване



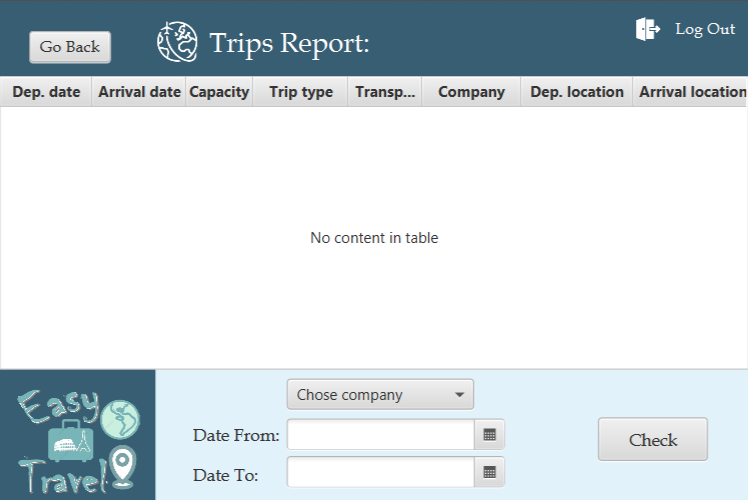
Примерен изглед за промяна/изтриване:

* Промяна на хонорар/изтриване на касиер



Примерен изглед за справка:

* Справка за пътувания на компания



1. Реализация на модул за регистриране на събития в системата – log4J.

За регистриране на събития в системата се използва библиотеката за логване в Java - log4J. Тя предоставя възможност за регистриране на поведението на приложенията, системната активност или други категории данни за събития.

Конфигурационният файл за логване е log4J.properties, който съдържа информацията какви грешки и информация ще се обработват. Основно има пет вида нива на регистрационни файлове, като ние използваме нива на логване ERROR и INFO. Нивото на логване ERROR извежда съобщение за грешка, което може да позволи на системата да продължи, а при INFO се отпечатва информационно съобщение, което подчертава напредъка на приложението. Грешките и информацията ще се извеждат в конзолата, както и ще се записват във файл application.log. 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.

log4j.rootLogger=ERROR, console, file  
  
log4j.appender.console=org.apache.log4j.ConsoleAppender  
log4j.appender.console.layout=org.apache.log4j.PatternLayout  
log4j.appender.console.layout.ConversionPattern=[%t] %-5p %c %x - %m%n  
  
log4j.appender.file=org.apache.log4j.RollingFileAppender  
log4j.appender.file.layout=org.apache.log4j.PatternLayout  
log4j.appender.file.layout.ConversionPattern=[%t] %-5p %c %x - %m%n  
log4j.appender.file.File=application.log

log4j.logger.bg.tu\_varna.sit.oop\_project\_demo=INFO, console, file

Пример за лог на грешкa, записан във файл application.log:

В HelloApplication.java:

public void start(Stage stage) throws IOException { PropertyConfigurator.*configure*(HelloApplication.class.getResource(Constants.Configurations.*LOG4J\_PROPERTIES*));  
  
 URL path = getClass().getResource(*HELLO\_VIEW*);  
  
 if(path != null){  
 Parent root = FXMLLoader.*load*(path);  
 Scene scene = new Scene(root);  
 scene.setFill(Color.*TRANSPARENT*);  
  
 stage.setTitle(Constants.Values.*Title*);  
 stage.setScene(scene);  
 stage.setResizable(false);  
 stage.setWidth(615);  
 stage.setHeight(440);  
 stage.show();  
 } else {  
 *log*.error("Sorry, the main FXML could not be loaded");  
 System.*exit*(-1);  
 }  
}

Въведено е съобщение за грешка log.error(“Sorry, the main FXML could not be loaded”), което да се изведе към лога за грешка във файла application.log при неуспешно зареждане на FXML файла:

[JavaFX Application Thread] ERROR *bg.tu\_varna.sit.oop\_project\_demo.application.HelloApplication* - Sorry, the main FXML could not be loaded

Други примери за логове на ниво ERROR:

Връзка с база данни – class Connection:

public class Connection {  
 private static final Logger *log* = Logger.*getLogger*(Connection.class);  
 private static SessionFactory *sessionFactory*;  
  
 static {  
 try {  
 *sessionFactory* = new Configuration().configure().buildSessionFactory();  
 } catch (Throwable ex) {  
 *log*.error("Initial SessionFactory creation failed" + ex);  
 }  
 }  
 public static Session openSession() {  
 return *sessionFactory*.openSession();  
 }  
 public static void closeSession() {  
 *sessionFactory*.close();  
 }  
}

В TripService:

public int createTrip(TripListViewModel a){  
 TripService tripService=TripService.*getInstance*();  
 Trip trip=new Trip(a.getDeparture(), a.getArrival(), a.getCapacity(), tripTypeService.getTripTypeByName(a.getTripTypeId().getTripTypeName()),  
transportTypeService.getTransportTypeByName(a.getTransportTypeId().getTransportTypeName()), companyService.getCompanyByName(a.getCompanyId().getUsername()),locationService.getLocationByName(a.getLocationFrom().getLocationName()),  
 locationService.getLocationByName(a.getLocationTo().getLocationName()), a.getTimeOfDeparture(), a.getTimeOfArrival());  
 if(checkIfTripExists(trip)){  
 *log*.info("Trip already exists!");  
 return 0;  
 }  
 else {  
 try{  
 repository.save(trip);  
 *log*.info("Trip created!");  
 }  
 catch (Exception e) {  
 e.printStackTrace();  
 *log*.error("Create trip error!");  
 }  
 return 1;  
 }  
}

В CompanyService:

public Company getCompanyByName(String name){  
 List<Company> allCompany=repository.getAll();  
 for(Company a:allCompany){  
 if(a.getUsername().equals(name))  
 {  
 return a;  
 }  
 }  
 *log*.error("Company not found!");  
 return null;  
}

В application.log:

[JavaFX Application Thread] ERROR *bg.tu\_varna.sit.oop\_project\_demo.data.access.Connection* - Initial SessionFactory creation failedorg.hibernate.service.spi.ServiceException: Unable to create requested service [*org.hibernate.engine.jdbc.env.spi.JdbcEnvironment*]



Последните два лога ни уведомяват, че има грешка при създаването на пътуване и че пътническата компания не е намерена в базата данни. Благодарение на log4J, с извеждането на този тип информация, успяваме по-лесно да открием грешката, която сме допуснали, да я поправим и да продължим да работим с приложението.

Примери за регистриране на събития на ниво INFO:

В CashierService:

public boolean cashierLogin(CashierListViewModel a)  
{  
 if(getCashierByName(a.getUsername())==null){  
 *log*.error("Cashier login error!");  
 return false;  
 }  
 *log*.info("Cashier login successful!");  
 return true;  
}

В CashierRepository:

public List<Cashier> getAll() {  
 Session session = Connection.*openSession*();  
 Transaction transaction = session.beginTransaction();  
 List<Cashier> cashiers = new LinkedList<>();  
 try {  
 String jpql = "SELECT r FROM Cashier r";  
 cashiers.addAll(session.createQuery(jpql, Cashier.class).getResultList());  
 *log*.info("Got all cashiers");  
 } catch (Exception ex){  
 *log*.error("Get all cashiers error. " + ex);  
 } finally {  
 transaction.commit();  
 }  
 return cashiers;  
 }  
}

В application.log:

[JavaFX Application Thread] INFO *bg.tu\_varna.sit.oop\_project\_demo.business.services.CashierService* - Cashier login successful!

[JavaFX Application Thread] INFO *bg.tu\_varna.sit.oop\_project\_demo.data.repositories.CashierRepository* - Got all cashiers

С логовете на ниво INFO получаваме информация за напредъка на приложението. В горните два примера се запазва информация във файла application.log, че касиер се е логнал успешно в системата и че са се извели успешно всички касиери в отчета.

1. **Тестови резултати.**

За да се постигне добро качество на даден продукт, е необходимо добро тестване. Именно затова в програмата са реализирани различни видове тестове – unit (единични), функционални и интеграционни.

Единичното тестване е тестване на малка логика или код, за да се провери дали изходът на кода е както се очаква при въвеждане на конкретни данни иили при удовлетворяване на определени условия. Функционалните тестове проверят един метод от програмата, а интеграционните преминават през няколко функции/методи.

JUnit е рамка с отворен код, която се използва за създаване и изпълнение на модулни тестове по автоматизиран начин в Java.

С помощта на тестовете JUnit се пишат независими модули, като по този начин се подобрява покритието на теста и качеството на приложението. Изкарва се ясен и явен отчет за резултатите от тестването.

Извършени са единични, функционални и интеграционни тестове на класовете от Services, Repositories и Models.

Пример за тест на CashierService:

package bg.tu\_varna.sit.oop\_project\_demo.business.services;  
  
import bg.tu\_varna.sit.oop\_project\_demo.data.entities.Cashier;  
import bg.tu\_varna.sit.oop\_project\_demo.data.repositories.CashierRepository;  
import bg.tu\_varna.sit.oop\_project\_demo.presentation.models.CashierListViewModel;  
import javafx.collections.ObservableList;  
import org.junit.jupiter.api.BeforeEach;  
import org.junit.jupiter.api.Test;  
  
import java.util.List;  
  
import static org.junit.jupiter.api.Assertions.\*;  
  
class CashierServiceTest {  
 private CashierService cashierService;  
 private Cashier cashier;  
 private CashierRepository cashierRepository;  
 private CashierListViewModel cashierListViewModel;  
 private CashierListViewModel cashierListViewModel1;  
  
 @BeforeEach  
 void setUp() {  
 cashierService = CashierService.*getInstance*();  
 cashier = new Cashier("TestCashier", "test1", "12345", 2200.00);  
 cashier.setCashierId(1);  
 cashierRepository = CashierRepository.*getInstance*();  
 cashierListViewModel = new CashierListViewModel("test1", "12345", "TestCashier", 2200.00);  
 cashierListViewModel1 = new CashierListViewModel("test2", "test", "test", 1);  
  
 }  
  
 @Test  
 void deleteCashier() {  
 List<Cashier> cashiers=cashierRepository.getAll();  
 cashierService.deleteCashier(cashierListViewModel);  
 *assertNotEquals*(cashiers,cashierRepository.getAll());  
 }  
  
 @Test  
 void cashierLogin() {  
 *assertTrue*(cashierService.cashierLogin(cashierListViewModel));  
 }  
  
 @Test  
 void createCashier() {  
 *assertEquals*(0, cashierService.createCashier(cashierListViewModel1));  
 }  
  
 @Test  
 void checkIfCashierExists() {  
 *assertTrue*(cashierService.checkIfCashierExists(cashier));  
 }  
  
 @Test  
 void getCashierByName() {  
 *assertEquals*(cashier, cashierService.getCashierByName(cashier.getUsername()));  
 }  
  
 @Test  
 void getAllCashier() {  
 ObservableList<CashierListViewModel> cashierListViewModels = cashierService.getAllCashier();  
 *assertEquals*(cashierListViewModels, cashierService.getAllCashier());  
 }  
  
 @Test  
 void updateCashierHonorarium() {  
 *assertTrue*(cashierService.updateCashierHonorarium(cashierListViewModel, 22));  
 }  
}

Пример за тест на LocationRepository:

package bg.tu\_varna.sit.oop\_project\_demo.data.repositories;  
  
import bg.tu\_varna.sit.oop\_project\_demo.business.services.LocationService;  
import bg.tu\_varna.sit.oop\_project\_demo.data.entities.Location;  
import org.junit.jupiter.api.BeforeEach;  
import org.junit.jupiter.api.Test;  
  
import java.util.List;  
  
import static org.junit.jupiter.api.Assertions.\*;  
  
class LocationRepositoryTest {  
 LocationRepository locationRepository;  
 LocationService locationService;  
 Location location;  
  
 @BeforeEach  
 void setUp() {  
 locationRepository = LocationRepository.*getInstance*();  
 locationService = LocationService.*getInstance*();  
 location = new Location("Temporary location");  
 }  
  
 @Test  
 void save() {  
 List<Location> all=locationRepository.getAll();  
 locationRepository.save(location);  
 *assertNotEquals*(all,locationRepository.getAll());  
 }  
  
 @Test  
 void update() {  
 List<Location> all=locationRepository.getAll();  
 Location a=locationRepository.getById(7);  
 a.setLocationName("Still temporary");  
 locationRepository.update(a);  
 *assertEquals*(all,locationRepository.getAll());  
 }  
  
 @Test  
 void delete() {  
 List<Location> all=locationRepository.getAll();  
 locationRepository.delete(location);  
 *assertEquals*(all,locationRepository.getAll());  
 }  
  
  
 @Test  
 void getAll() {  
 List<Location> locations = locationRepository.getAll();  
 List<Location> locations1 = locationRepository.getAll();  
 *assertEquals*(locations, locations1);  
 }  
}

Пример за тест на AdminListViewModel:

package bg.tu\_varna.sit.oop\_project\_demo.presentation.models;  
  
import org.junit.jupiter.api.BeforeEach;  
import org.junit.jupiter.api.Test;  
  
import static org.junit.jupiter.api.Assertions.\*;  
  
class AdminListViewModelTest {  
 AdminListViewModel adminListViewModel;  
 AdminListViewModel adminListViewModel1;  
  
 @BeforeEach  
 void setUp() {  
 adminListViewModel = new AdminListViewModel();  
 adminListViewModel1 = new AdminListViewModel();  
 adminListViewModel1.setPassword("1");  
 adminListViewModel1.setUsername("1");  
 adminListViewModel.setPassword("1");  
 adminListViewModel.setUsername("1");  
 }  
  
 @Test  
 void equals(){  
 *assertTrue*(adminListViewModel.equals(adminListViewModel1));  
 }  
  
 @Test  
 void getUsername() {  
 *assertEquals*("1",adminListViewModel1.getUsername());  
 }  
  
 @Test  
 void getPassword() {  
 *assertEquals*("1", adminListViewModel1.getPassword());  
 }  
}

Успешно изпълнени тестове:

* CashierService

Картина, която съдържа текст

Описанието е генерирано автоматично

* CompanyService

Картина, която съдържа текст

Описанието е генерирано автоматично

* DistributorService

Картина, която съдържа текст

Описанието е генерирано автоматично

* LocationService

Картина, която съдържа текст

Описанието е генерирано автоматично

* RequestService

Картина, която съдържа текст

Описанието е генерирано автоматично

* TicketService

Картина, която съдържа текст

Описанието е генерирано автоматично

* TransportTypeService

Картина, която съдържа текст

Описанието е генерирано автоматично

* TripService

Картина, която съдържа текст

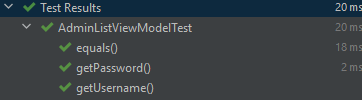
Описанието е генерирано автоматично

* TripTypeService

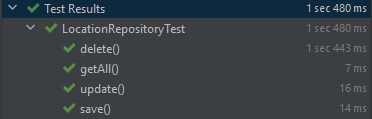
Картина, която съдържа текст

Описанието е генерирано автоматично

* AdminListViewModel



* LocationRepository



В резултат от всички успешно изпълнени тестове програмната логика на софтуера работи коректно, следователно приложението може да се използва.