|  |  |
| --- | --- |
| **Гордеев Алексей Aleksei.a.Gordeev@gmail.com** |  |

**Task**

It is required to create a web-application that simulates the functionality of mobile network operator information system. Details of subject area and technical requirements are given below.

**Subject area**

There are following kinds of entities:

*Tariff*

* Title
* Price
* List of available options

*Option*

* Title
* Price
* Cost of connection

*Client*

* Name
* Last name
* Birth date
* Passport data
* Address
* List of contracts (telephones numbers of client)
* E-mail
* Password

*Contract*

* Telephone number
* Tariff
* Connected options for tariff

The application must provide the following functionality:

For clients

* + Browse of the contract in a personal cabinet;
  + Browse of all available tariffs and change a tariff;
  + Browse of all available options for tariff, add new options, disable the existing ones;
  + Lock / Unlock of a number (if number were locked, it is not allowed to change the tariff and options; if number was locked not by a client, he can’t unlock it);

For employees

* + Conclusion of contract with a new client: the choice of a new telephone number with the tariff and options. The phone number should be unique.
  + Browse of all clients and contracts;
  + Lock and unlock of client’s contract;
  + Looking up client’s contract by phone number;
  + Change tariff, add and remove options of contract;
  + Add new tariffs, remove existing one;
  + Add / remove option available for the specific tariff;
  + Option management: options may be inconsistent each other or ought to be added with certain options, employee adds and removes these rules.

On the each page during operating with contract before saving the changes the basket must be displayed. It should contain the client's choices.

**Solution description**

**Overview**

The application has a common three-layered desing: presentation, services and persistence.

The DAO-layer is used for basic (create, read, update and delete) operations on the database. The DAO layer is based on Java Persistence API the the Application Server provides the implementation of it by the Hibernate.

As a database management system the application uses MySQL.

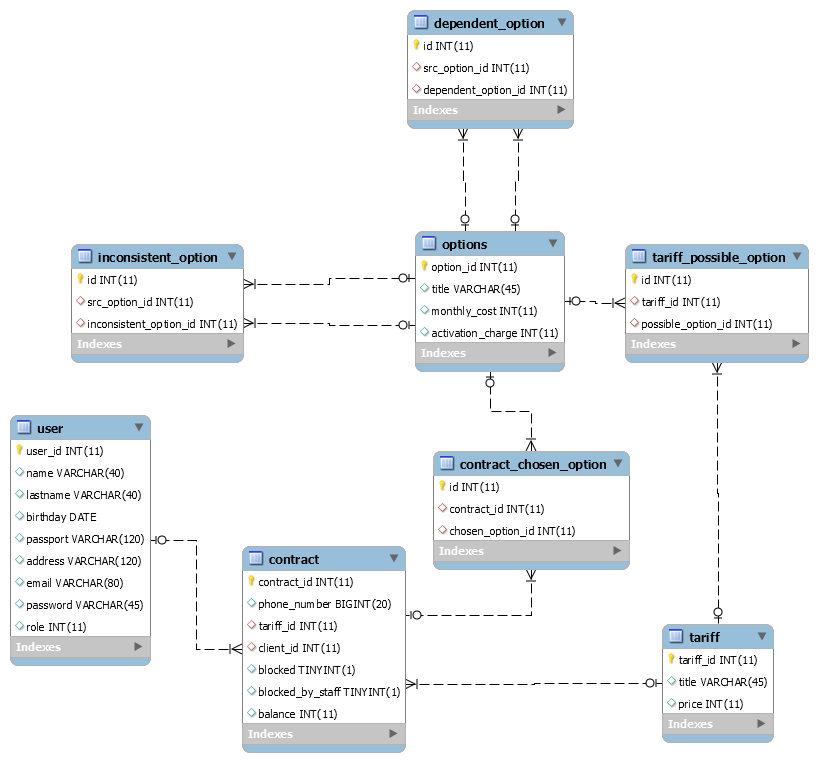
The application runs with a help of the Spring Framework. To be exact there are Spring Core and Spring MVC here. The Spring Core helps to solve the task of injecting of dependencies. While the Spring MVC is responsible for presentation layer.

The application runs under WildFly 9.0 Application Server.

**Datasource**

The Application Server is configured in such a way that it creates and managed the Entity manager. And the Entity managers is obtained from JNDI.

**Database design**

****

First let’s take a look at **tariff** and **options** tables. By condition of the task each tariff may have a specific set of options. At the same time an option may have been linked to the different tariffs. The **tariff\_possible\_option** table serves for the storing this data. The similar things are with **contract** and **options.** Each of the contract may have a definite assortment of option and vice versa. The **contract\_chosen\_option** helps with storing this linking information.

This diagram shows that a user may have several contracts. While a contract may has the only tariff.

It is considered that several options may have been linked to a contract and several contracts may have the same option. So there is an Many-to-Many relationship here.

All the foreign keys in each table have constrains on update and delete operation. Performing these operations with primary key will cause cascade operation on foreign keys. The only exception **tariff** is table. The removing of a tariff must not lead to removing of connected contracts. So in that case the cascade operations set to «SET NULL».

**Input data validation**

For the purpose of validation the user input data I used Java Validation API in Spring MVC. It [requires](http://docs.spring.io/spring/docs/current/spring-framework-reference/html/validation.html) a class for binding it to form’s inputs. But my existing classes (like UserDTO) were not what I need because Java Validation API does not provide @Pattern annotation for Integer fields. Consequently I created a bunch of classes for form validation. All those classes located in «*controllers\_mvc.validationFormClasses*» package.

**Бизнес-логика**

**Опции**

**Option relationship**

**Overview**

По условию, для любой опции может быть наложено ограничение, что она будет работать только одновременно с другой опцией (или с набором опций). Кроме этого некоторые опции могут быть несовместимы друг с другом.

Четыре операции:

* Удаление несовместимости между опциями
* Удаление зависимости одной опции от другой
* Добавление зависимости одной опции от другой
* Добавление несовместимости

Первые два пункта каких-либо проблем не вызывают и сводятся к простым удалениям записей из БД. Последние два пункта сложнее. В приложении довольно большое внимание уделено тому, чтобы добавление зависимости одной опции от другой не противоречило уже существующим зависимостям и несовместимостям. Аналогично - при добавление несовместимости между опциями.

Ниже я более детально рассмотрю решение этих задач.

Далее я буду называть опцию B, которая зависит от другой опции A, зависимой от B и обозначать как A->C. И в обратную сторону: опцию А – обязательной для опции B.



**Criteria of valid option relationship**

Возможность делать одну опцию зависимой от другой приводит к тому, что связи эти могут образовывать графы.

1. Логично предположить, что в случае, когда одна опция зависит от другой, это не значит, что верно и обратное.
2. Зависимости опций не должны образовывать зацикливания. Например, если A->B (опция B зависит от А) и B->C, то нельзя разрешать пользователю добавлять зависимость С->A, поскольку в этом случае структура становится неделимой. В случае подключения к контракту одной из опций такого цикла, все остальные так же должны быть подключены. Тоже самой справедливо и для удаления. Зацикленное множество опций фактически становится одной большой опцией, которую пользователю проще создать как новую. Поэтому зацикливания зависимостей в приложении не разрешены.
3. 

Возможность зависимостей при недопустимости зацикливаний, приводит к тому, что опции могут образовывать древовидные структуры.

Основной критерий того, что все отношения между опциями корректны:

Для любой опции строим множество, состоящее из самой это опции и опций обязательных для нее. И каждое такое множество не должно содержать опций несовместимых с другими опциями этого множества.

Поясню на примере Диаграммы 1. Представим, что мы хотим подключить к контракту опцию 100 SMS. Мы видим, что она зависима от опции SMS, которая в свою очередь зависима от опции Gold. То есть обязательные опции для 100SMS – это {SMS, GOLD}. При подключении 100SMS обе эти опции тоже должны подключаться. Между этими тремя опциями {100SMS, SMS, GOLD} нет попарно несовместимых. Значит, противоречий не возникает.

Diag.1 Valid option tree



Далее посмотрим два примера, когда дерево опций оказывается некорректным.

Diag.2 Invalid option tree Diag.3 Invalid option tree

Теперь снова рассмотрим предыдущую цепочку с тем изменением, что опция Gold несовместима с 100SMS (диаграмма 2). При подключении к контракту опции 100SMS должны также подключаться опции SMS и Gold. Но Gold и SMS – несовсместимы, следовательно, такое дерево опций некорректно.

Аналогичная ситуация отражена на диаграмме 3. Подключая опцию All inclusive, должны подключаться обязательные для нее опции Inet и SMS. Но поскольку Inet и SMS несовместимы, то это дерево опций тоже несовместимо.

**Критерий корректности добавления зависимости**

Приведенные примеры показаны здесь, чтобы показать основную идею. Речь в «основном критерии» идет о проверке корректности уже существующих связей между опциями. Между тем в задаче необходима проверка того, не будет ли добавление еще одной зависимости между двумя опциями противоречить существующим связям.

Предположим, что нам надо проверить на корректность добавление следующей зависимости: A->B (т.е. B зависит от А). Для этого необходимо выполнить следующие шаги:

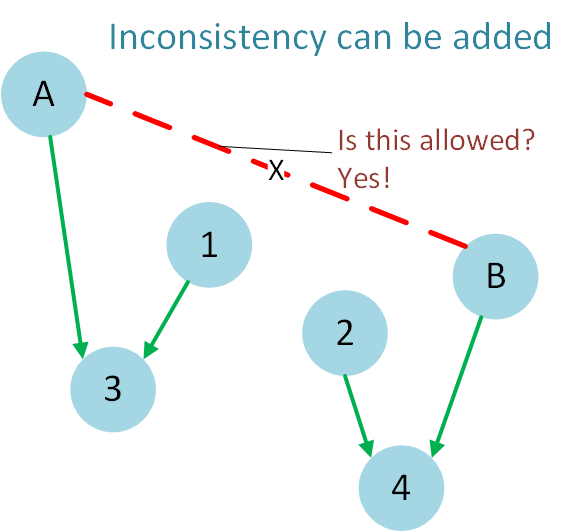
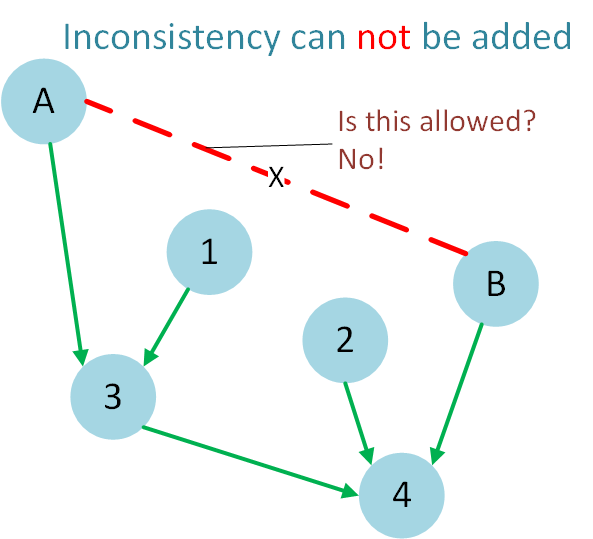
1. Построить множество, состоящее из A и всех обязательных для нее опций.
2. Построить множество, состоящее из B, всех зависимых для нее опций и всех обязательных для нее опций.
3. Проверить, что нет такой опции из первого множества и опции из второго множества, таких что, они будут несовместимы. Добавление будет корректным, если таких опций не нашлось.

**Критерий корректности добавления зависимости**

Предположим, что нам надо проверить на корректность добавление несовместимости опций A и B. Для этого необходимо выполнить следующие шаги:

1. Построить множество, состоящее из A и всех зависимых для нее опций.
2. Построить множество, состоящее из B, всех зависимых для нее опций.
3. Проверить, не пересекаются ли эти множества. Если они пересекаются (хоть одна опция входит в оба множества), то добавление несовместимости будет некорректным.

**Implementation**

Как видно, в основном алгоритмы в этой секции опираются на поиск множества зависимых опций и поиск множества обязательных опций для некоторой заданной опции. В обоих случаях используется один и тот же рекурсивный алгоритм обхода дерева. Только в одном случае он работает с зависимыми опциями, а в другом - с обязательными.

**Тарифы**

Создание тарифа представляет из себя заполнение формы, валидации введенных данных и записи в базу данных. Удаление чуть более сложное: перед удалением записи из базы данных приложение переключает пользователей с этого тарифа на тариф «Базовый». В свою очередь «Базовый» тариф удалить нельзя.

Кроме этого к тарифу можно подключать опции, которые потом могут быть доступны для подключения к контрактам с этим тарифом. Учитывая уже реализованную функциональность нахождения множества всех обязательных опций для выбранной опции, при подключении некоторой опции автоматически происходит и подключение всего дерева обязательных опций. При этом несовместимость опций на данном этапе пока роли не играет (будет играть при подключении опций к контракту).

**Редактирование контракта**

На этой форме пользователь может просматривать контракт, добавлять новый тариф и опции в корзину. Оплачивать корзину. За хранение данных к корзине отвечает объект класса Cart, который хранится в HttpSession.

Про несовместимые опции надо написать.

**Авторизация и аутентификация**

Во время логина, данные о пользователе (UserId и, если пользователь – клиент, ContractID) сохраняются в HttpSession.

Аутентификация реализована с помощью фильтра, который перед передачей запроса Web-контроллеру проверяет, что пользователь вошел в систему. Авторизация также сделана на базе фильтра. Так фильтр не допускает клиента до редактирования тарифов и опций, а так же к редактированию чужого контракта. Кроме этого фильтр не позволяет клиенту проводить операции над контрактом в случае, если контракт пользователя заблокирован.

**Sonar**

