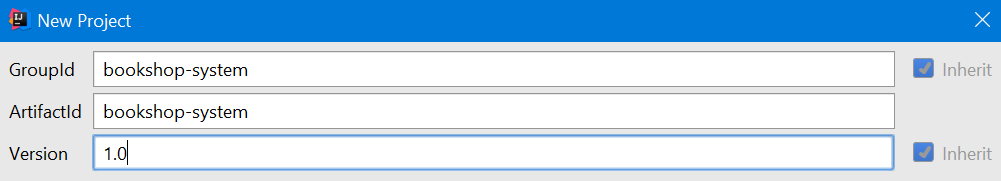
# Exercise: Hibernate Code First - Bookshop

This document defines the **exercise assignments** for the ["Databases Advanced – Hibernate" course @ Software University.](https://softuni.bg/trainings/1635/databases-frameworks-hibernate-and-spring-data-june-2017)

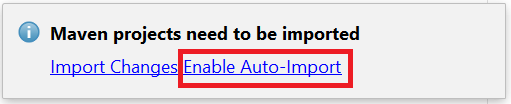
## Create a Database for Student System using Code First

First of all, create a **new Maven project** and name it accordingly - in our case, **bookshop-system** sounds like a nice name.



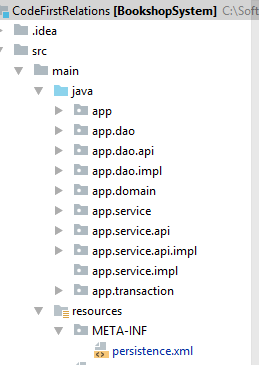
### Step 1 – Prepare Environment

When you create new Maven project the Intellij idea ask you **enable auto import dependencies**. Make sure you turned that on.



First of all, we should make the structure of our project. In the java package create new package app, where we will keep all of our classes. Inside it add the following packages:

* **domain** – in that package we will keep all of our entities (classes that would represent our models for the tables in the database)
* **dao.api –** will keep all dao interfaces from the data access layer defining all CRUD operations
* **dao.imp** – will keep implementational classes from the data access layer which will perform CRUD operations with the database. (DAO stands for Data Access Objects)
* **service.api** – will keep the interfaces four our services
* **service.impl** – will keep the actual implementations of the services
* **transaction** – will keep our transaction management interfaces
* **app** – root package for the application. Here will be the entry point of the project



In the pom.xml include the following dependencies:

* Hibernate Core
* MySQL Connector

|  |
| --- |
| pom.xml |
| <**groupId**>soft-uni</**groupId**> <**artifactId**>code.first.relations</**artifactId**> <**version**>1.0-SNAPSHOT</**version**>  **<dependencies>**  **<dependency>**  **<groupId>org.hibernate</groupId>**  **<artifactId>hibernate-core</artifactId>**  **<version>5.2.3.Final</version>**  **</dependency>**  **<dependency>**  **<groupId>mysql</groupId>**  **<artifactId>mysql-connector-java</artifactId>**  **<version>6.0.4</version>**  **</dependency>**  **</dependencies>** |

Also, we need to change the build configuration to target Java 8 instead of Java 5. Add the following code snipped to the pom.xml file.

|  |
| --- |
| pom.xml |
| <**build**>  <**plugins**>  <**plugin**>  <**groupId**>org.apache.maven.plugins</**groupId**>  <**artifactId**>maven-compiler-plugin</**artifactId**>  <**version**>3.5.1</**version**>  <**configuration**>  <**source**>1.8</**source**>  <**target**>1.8</**target**>  </**configuration**>  </**plugin**>  </**plugins**> </**build**> |

In **src\resources\** package create new **META-INF\persistence.xml** file and put the following template in it and provide appropriate data for connection to the database. Set the database name to **bookshop\_system** and use your username and password to access the database.

|  |
| --- |
| persistence.xml |
| *<?***xml version="1.0" encoding="UTF-8"***?>* <**persistence xmlns="http://java.sun.com/xml/ns/persistence" version="2.0"**>  <**persistence-unit name="bookshop"**>  <**properties**>  <**property name = "hibernate.connection.url" value="jdbc:mysql://localhost:3306/<DATABASE\_NAME>?createDatabaseIfNotExist=true"**/>  <**property name = "hibernate.connection.driver\_class" value="com.mysql.jdbc.Driver"**/>  <**property name = "hibernate.connection.username" value="<USERNAME>"**/>  <**property name = "hibernate.connection.password" value="<PASSWORD>"**/>  <**property name = "hibernate.dialect" value="org.hibernate.dialect.MySQL5Dialect"**/>  <**property name = "hibernate.hbm2ddl.auto" value="create"**/>  <**property name = "hibernate.show\_sql" value = "true"** />  </**properties**>  </**persistence-unit**> </**persistence**> |

The property hibernate.hbm2ddl.auto automatically validates or exports schema DDL to the database. Possible values are:

* validate - hibernate only validates whether the table and columns are existing or not. If the table doesn’t exist, then hibernate throws an exception. Validate is the default value for hibernate.ddl-auto.
* update - hibernate checks for the table and columns. If table doesn’t exist, then it creates a new table and if a column doesn’t exist it creates new column for it.
* create – hibernate first drops the existing table, then creates new table and then executes operations on the newly created table
* create-drop - hibernate first checks for a table and do the necessary operations and finally drops the table after all the operations are completed.

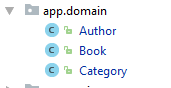
For using the **code first approach** to model database schema we should set the value to **create**.

### Step 2 - Model the Database

A bookshop keeps **books**. A book can have one **author** (for the sake of simplicity) and many **categories**. And each category can be placed on many books. Let's create a class for each of the main tables.

* **Book** - **id**, **title** (between 1..50 symbols), **author** (Author), **categories** (Set), **price**, **copies**, **release date** (optional), **description** (optional, up to 1000 symbols, **related books** (Set)
* **Author** - **id**, **first name** (optional) and **last name, author’s books (**Set**)**
* **Category** - **id**, **name**, **books** (Set)

Assume everything **not market optional** is mandatory.



The **classes** should describe with **properties** each of the **table columns**. **Do NOT forget to create empty constructor for each entity class.**



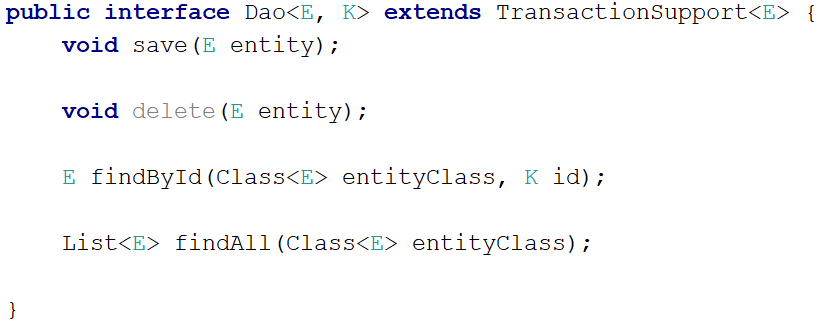
Add constraints and validations for fields as described above. Do the same for the **Book** and **Category** models.

The final schema of the database should look like that:

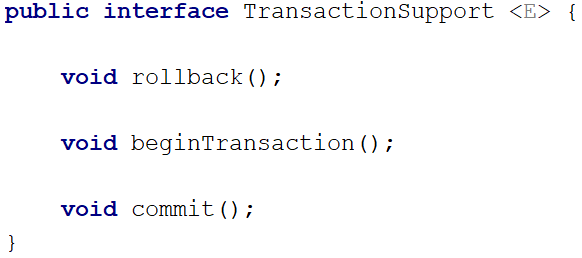


### Step 3 - Create the Data Access Layer

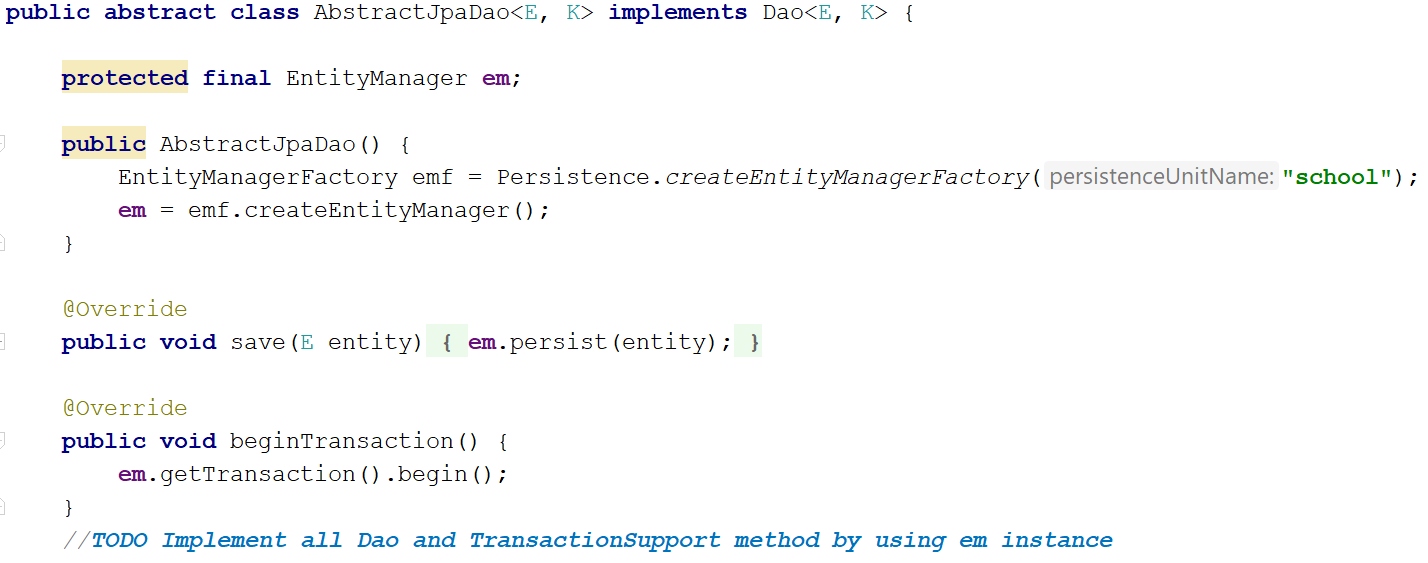
Once the entities are done, our next step is to write the so-called **Data Access Layer**. We should create **one generic dao interface** for common CRUD operations for all our entities. Inside **dao.api** create :



**TransactionSupport** interface should be created in **transaction** package as follows :

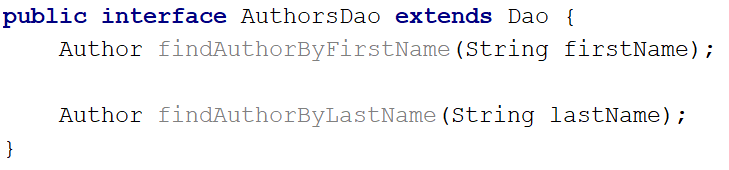


Now we are ready to implement Dao logic at abstract level. Inside **dao.impl** package create **AbstractJpaDao** abstract class as follows :



Implement all **Dao** and **TransactionSupport** methods as if the class is not abstract (leave it abstract anyway).

Now we are ready to create Dao interfaces defining specific CRUD operations for each entity. By extending **Dao interface** we allready have the four basic operation, lets define some specific Dao operations. Under **dao.api** package create **AuthorsDao** interface which adds two specific operations to the four base operations :



Analogically create Dao interfaces with specific operations for **Book** and **Category** entities.

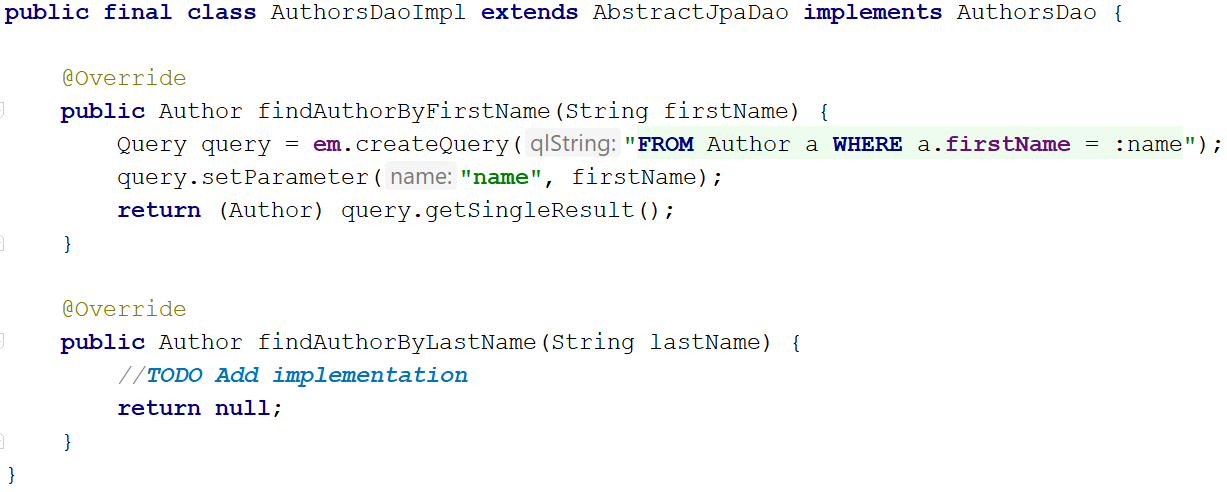
**Inside BooksDao interface add methods :**

Book findByTitle(String id);  
  
Book findByAuthor(Author author);

**Inside CategoriesDao interface add method :**

Category findByName(String categoryName);

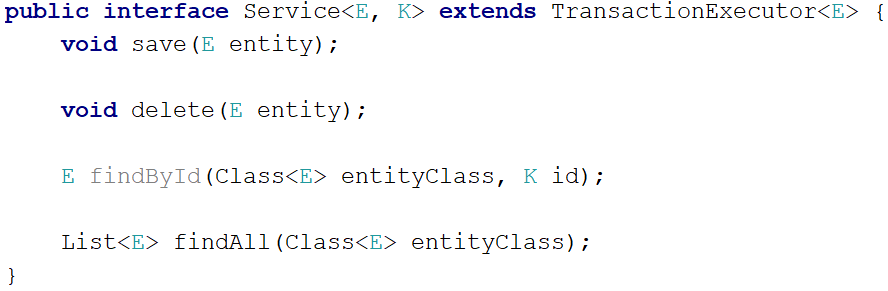
Finally we are ready to create concrete implementational classes for each specific dao interface. Inside **dao.impl** package create **AuthorsDaoImpl** class :



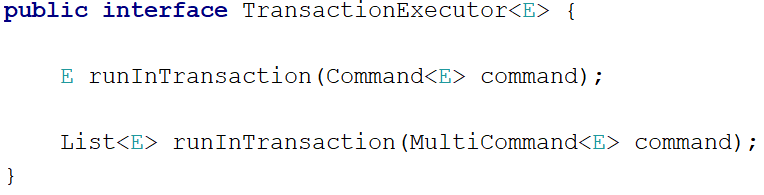
Inside **dao.impl** package create **BooksDaoImpl** and **CategoriesDaoImpl** and implement specific dao methods.

### Step 4 - Create the Service Layer

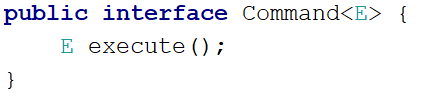
**Service layer** is the **link between** the **data access layer** and the **presentation layer** (UI). Here we should make service interface according our business logic recuirements. With that interface, we should be able to perform logic at service (domain or business) level and interact with the database by using dao interfaces to perform CRUD operations. Base generic **Service** interface duplicates Dao interface methods. Inside **service.api** package create :

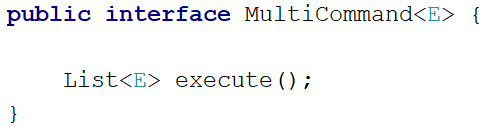


One important feature of the **Service Layer** is to be transactional, meaning that all its method should be executed in transaction. **TransactionExecutor** interface helps to achieve transactional service method. Inside **transaction** package create :

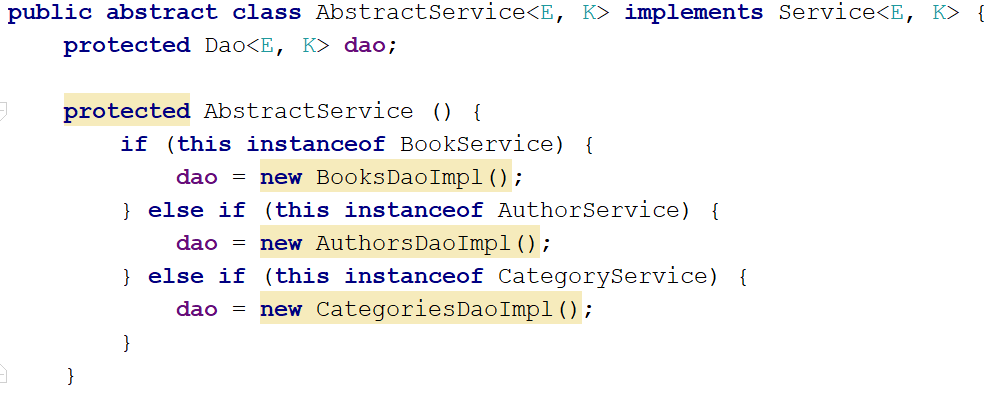


**Command** and **MultiCommand** interfaces define a single **execute** method which is supposed to hold logic which is meant to be executed inside a transaction. Inside **transaction** package create the two interfaces :

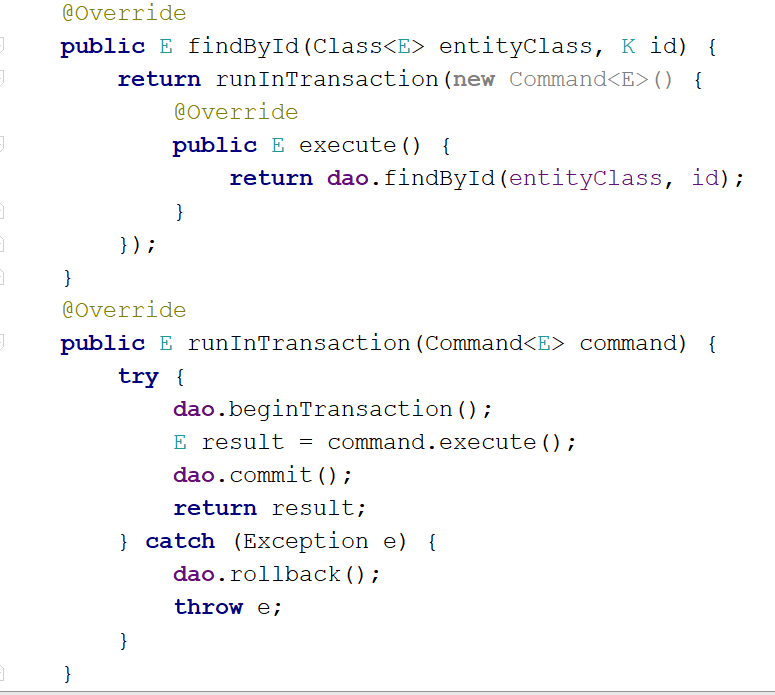




Now we are ready to implement **Service Layer** logic at abstract level. Inside **service.impl** package create **AbstractService** abstract class as follows :

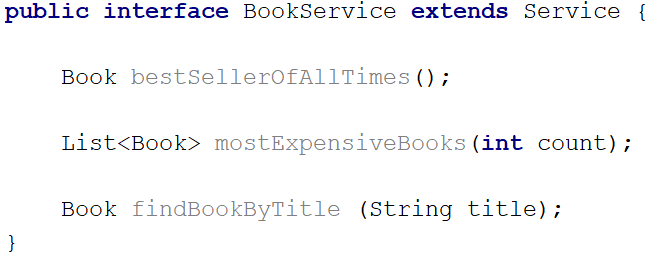


Here are implementations of most interesting methods of **AbstractService** class :



Implement all other methods accordingly.

Now we will define interfaces for specific service layer operations. Example will be given with **BooksService** interface this time. Inside **service.api** package create **BooksService** interface :



Finally we are ready to implement concreate implamentation class for **BooksService** interface. Inside **service.impl** package create **BooksServiceImpl** class :



Make specific service layer **interfaces and their implementations** for all of our remaining entities (**Authors** and **Categories**). Put the interfaces in **service.api package** and their implementations in the **service.impl package**.

### Step 7 - Console Client

Finally, it’s time to make our presentation layer (UI). For sake of simplicity we would use command line interface or simply said our beloved console. Inside root **app** package create **BookshopSystem** class with main method :

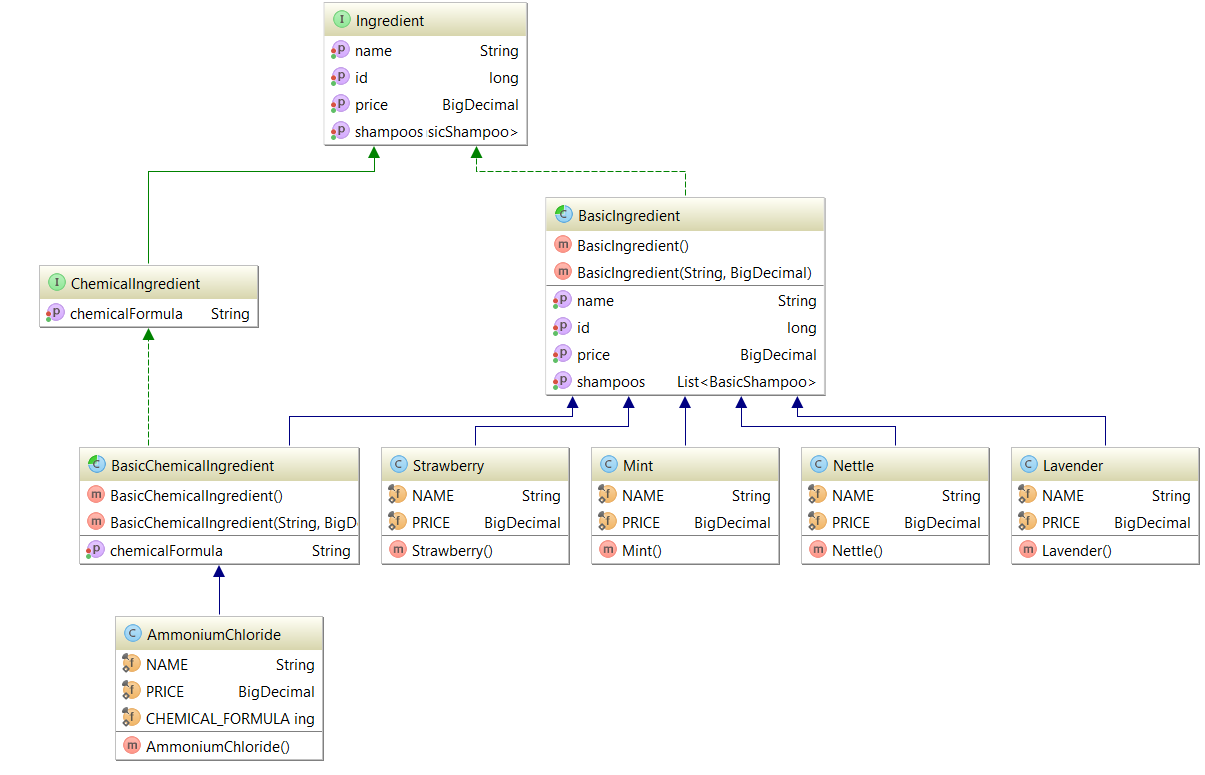


Check database three categories, one author, and one book should be inserted.

**Console output (in case of generated toString methods for all entities):** [Author{id=1, firstName='Karl', lastName='Mai', booksByAuthor=[Book{id=1, title='Vinetu 1', categories=[Category{name='Wild West'}, Category{name='Indians'}, Category{name='Adventure'}], price=null, copies=0, releaseDate=null, description='null', relatedBooks=null}]}]

# Shampoo Company

## Create Ingredients

Every shampoo has ingredients. Here is the expected structure of the ingredients:

Define two interfaces as follows:

**public interface** Ingredient **extends** Serializable{  
  
 **long** getId();  
  
 **void** setId(**long** id);  
  
 String getName();  
  
 **void** setName(String name);  
  
 BigDecimal getPrice();  
  
 **void** setPrice(BigDecimal price);  
}

**public interface** ChemicalIngredient **extends** Ingredient{  
  
 String getChemicalFormula();  
  
 **void** setChemicalFormula(String chemicalFormula);  
}

There are two types of ingredients:

* **Basic Ingredient**. It has the following information:
  + **Id**
  + **Name**
  + **Price**
* **Chemical Ingredient** which add additional information
  + **Chemical formula**

We have 4 types of Basic Ingredients:

* **Mint**
  + Price 3.54
* **Nettle**
  + Price 6.12
* **Strawberry**
  + Price 4.85
* **Lavender**
  + Price 2

And only one Chemical Ingredient:

* **Ammonium Chloride**
  + Price 0.59
  + Formula NH4Cl

**Create the required classes.**

## Implement the inheritance in the database

**All the ingredients should be saved in MySQL as a Single Table. Use the following strategy:**

@Inheritance(strategy = InheritanceType.***SINGLE\_TABLE***)  
@DiscriminatorColumn(name = **"type"**, discriminatorType = DiscriminatorType.***STRING***)

## Create Enumerator Size

**Create an enumerator Size with the following constants:**

***SMALL***, ***MEDIUM***, ***BIG***;

## Create Label

Implement the following interface:

**public interface** Label **extends** Serializable {  
  
 **long** getId();  
  
 **void** setId(**long** id);  
  
 String getTitle();  
  
 **void** setTitle(String title);  
  
 String getSubTitle();  
  
 **void** setSubTitle(String subTitle);  
}

We need to create one class Classic Label with the following fields:

* **ClassicLabel**
  + **Id**
  + **Title**
  + **Subtitle**

**Create the class.**

## Create Batch

Implement the following interface:

**public interface** Batch **extends** Serializable{  
  
 **long** getId();  
  
 **void** setId(**long** id);  
  
 Date getBatchDate();  
  
 **void** setBatchDate(Date batchDate);  
  
 Set<BasicShampoo> getShampoos();  
  
 **void** setShampoos(Set<BasicShampoo> shampoos);  
}

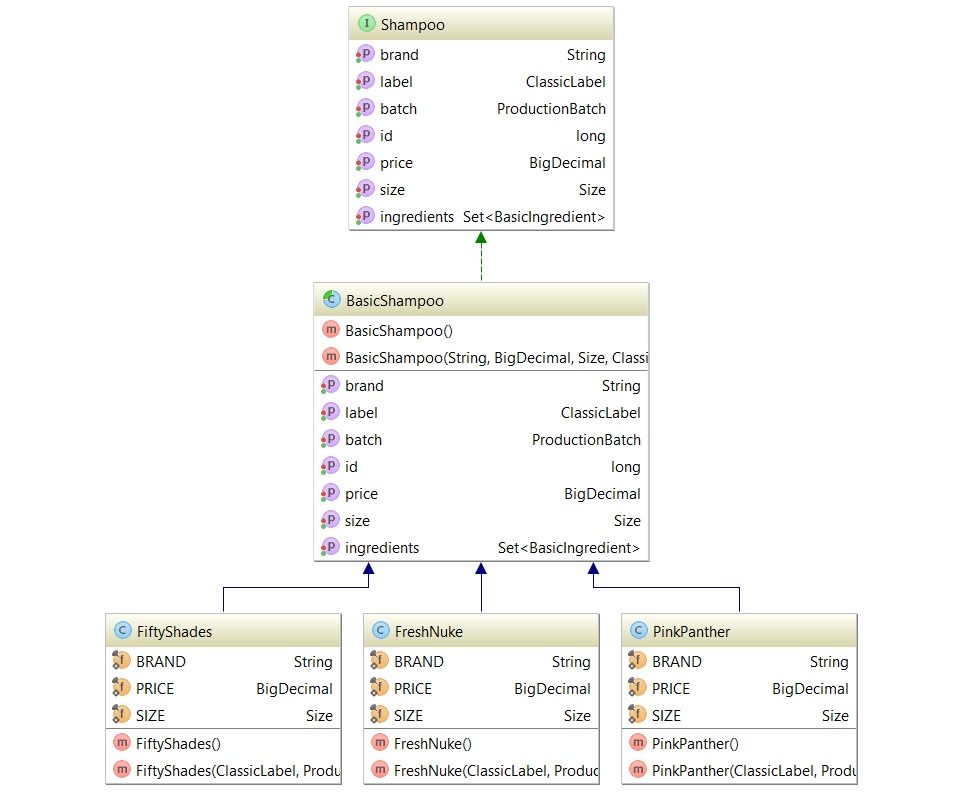
We need a class Production Batch that will hold set of shampoos.

* **ProductionBatch**
  + **Id**
  + **Batch Date**
  + **Shampoos**

**Create the class.**

## Create Shampoos

It is required to create the following structure:



Implement the following Interface:

**public interface** Shampoo **extends** Serializable{  
  
 **long** getId();  
  
 **void** setId(**long** id);  
  
 String getBrand();  
  
 **void** setBrand(String brand);  
  
 BigDecimal getPrice();  
  
 **void** setPrice(BigDecimal price);  
  
 Size getSize();  
  
 **void** setSize(Size size);  
  
 ClassicLabel getLabel();  
  
 **void** setLabel(ClassicLabel label);  
  
 ProductionBatch getBatch();  
  
 **void** setBatch(ProductionBatch batch);  
  
 Set<BasicIngredient> getIngredients();  
  
 **void** setIngredients(Set<BasicIngredient> ingredients);  
}

We need an abstract class BasicShampoo that hold the following information:

* **BasicShampoo**
  + **Id**
  + **Brand**
  + **Price**
  + **Size**
  + **Label**
  + **Batch**
  + **Set of Ingredients**

We have 4 different types of Shampoos:

* **Fresh Nuke**
  + **Brand “Fresh Nuke”**
  + **Price 9.33**
  + **Size Big**
  + **Label with title "Fresh Nuke" and subtitle "Explosively cool"**
  + **It’s made of Mint, Nettle and Ammonium Chloride**
* **Pink Panther**
  + **Brand “Pink Panther”**
  + **Price 8.50**
  + **Size Medium**
  + **Label with title "Pink Panther" and subtitle "Pleasure in pink "**
  + **It’s made of Lavender and Nettle**
* **Fifty Shades**
  + **Brand “Fifty Shades”**
  + **Price 6.69**
  + **Size Small**
  + **Label with title "Fifty Shades" and subtitle "Tie the aroma"**
  + **It’s made of Strawberry and Nettle**

**Create the classes. The inheritance should be presented in a single table.**

## Persistent enumerator

**Make the field size in BasicShampoo persistent. You can use the following annotation:**

@Enumerated(value = EnumType.***STRING***)

## One-To-One Relation

**Create a unidirectional relation between Shampoo and Label which is type one-to-one:**

@OneToOne(optional = **false**)  
@JoinColumn(name = **"label\_id"**, referencedColumnName = **"id"**)

## One-To-Many/Many-To-One Relation

**Create a bidirectional relation between Shampoo and Batch:**

@ManyToOne(optional = **false**)  
@JoinColumn(name = **"batch\_id"**, referencedColumnName = **"id"**)  
**private** ProductionBatch **batch**;

@OneToMany(mappedBy = **"batch"**, targetEntity = BasicShampoo.**class**)  
**private** Set<Shampoo> **shampoos**;

## Many-To-Many Relation

Add one more property to our interface Ingredient:

Set<BasicShampoo> getShampoos();  
  
**void** setShampoos(Set<BasicShampoo> shampoos);

Every ingredient should have a list of shampoos.

**Create a bidirectional relation between Shampoo and Ingredient:**

@ManyToMany  
@JoinTable(name = **"shampoos\_ingredients"**,  
joinColumns = @JoinColumn(name = **"shampoo\_id"**, referencedColumnName = **"id"**),  
inverseJoinColumns = @JoinColumn(name = **"ingredient\_id"**, referencedColumnName = **"id"**))  
**private** Set<BasicIngredient> **ingredients**;

@ManyToMany(mappedBy = **"ingredients"**, targetEntity = BasicShampoo.**class**)  
**private** List<BasicShampoo> **shampoos**;

## Create Data Access Layer (Or Maybe Miss It ???)

**Create Dao interfaces and implementations for every entity we have. If you prefer architecture without Data Access Layer go straight to Service Layer.**

## Services

**Create single generic Service for all major operations the application should support (Seve, Remove, FindById, FindAll) for all entities. Think of Specific Service Interfaces and implementations for each entity.**

## Create Objects

**Create one shampoo of each type and save it to the database.**