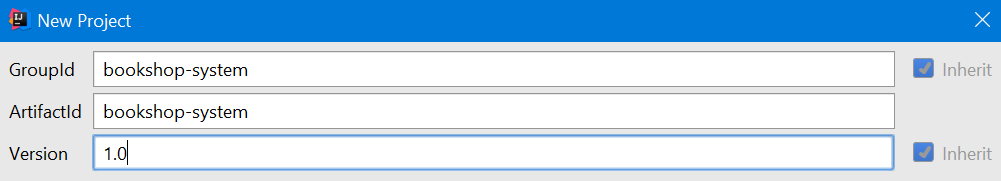
# Exercise: Spring Data - ShampooCompany

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)

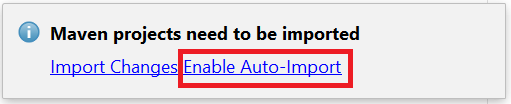
## Create a Database for Shampoo Company using Code First

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



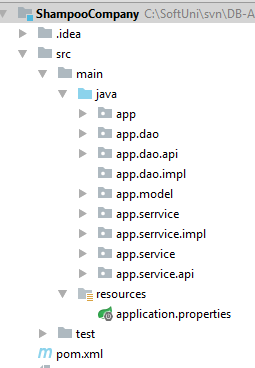
### Step 1 – Prepare Environment

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:

* **app.model** – in that package we will keep all of our entities (classes that would represent our models for the tables in the database)
* **app.dao.api** – will keep interfaces from the data access layer (our repositories) which will perform CRUD operations with the database. (DAO stands for Data Access Objects)
* **app.service.api** – will keep the interfaces four our services
* **app.service.impl** – will keep the actual implementations of the services
* **app** – root package. Here will be the core functionality of the console interface of the project



In the pom.xml file set **spring-boot-starter** as parent and include the following dependencies:

* Spring Boot Starter Data JPA
* MySQL Connector

|  |
| --- |
| pom.xml |
| <**parent**>  <**groupId**>org.springframework.boot</**groupId**>  <**artifactId**>spring-boot-starter-parent</**artifactId**>  <**version**>1.4.1.RELEASE</**version**> </**parent**>  <**dependencies**>  <**dependency**>  <**groupId**>org.springframework.boot</**groupId**>  <**artifactId**>spring-boot-starter-data-jpa</**artifactId**>  </**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 **application.properties** 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.

|  |
| --- |
| application.properties |
| *#Data Source Properties* **spring.datasource.driverClassName** = **com.mysql.jdbc.Driver spring.datasource.url** = **jdbc:mysql://localhost:3306/<DATABASE\_NAME>?useSSL=false spring.datasource.username** = **<DATABASE\_USERNAME> spring.datasource.password** = **<DATABASE\_PASSWORD>**  *#JPA Properties* **spring.jpa.properties.hibernate.dialect** = **org.hibernate.dialect.MySQL5InnoDBDialect spring.jpa.properties.hibernate.format\_sql** = **TRUE spring.jpa.hibernate.ddl-auto** = **create** *###Logging Levels # Disable the default loggers* **logging.level.org** = **WARN logging.level.blog** = **WARN** *#Show SQL executed with parameter bindings* **logging.level.org.hibernate.SQL** = **DEBUG logging.level.org.hibernate.type.descriptor** = **TRACE** |

The property spring.jpa.hibernate.ddl-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.

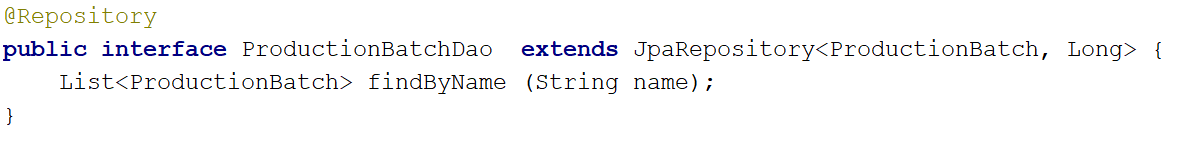
For using the **code first approach** to model database schema we should set the value to **create**.

### Step 2 - Model the Database

Use exactly the same model from the previous lecture with no changes.

### Step 3 - Create the Data Layer

Once the entities are done, our next step is to write the so-called **Data Layer**. We should create **one interface repository** for each of our entities. Each repository should extend **CrudRepository<Object, ID>**. Remember to put **@Repository** to annotate the interface. For Example:

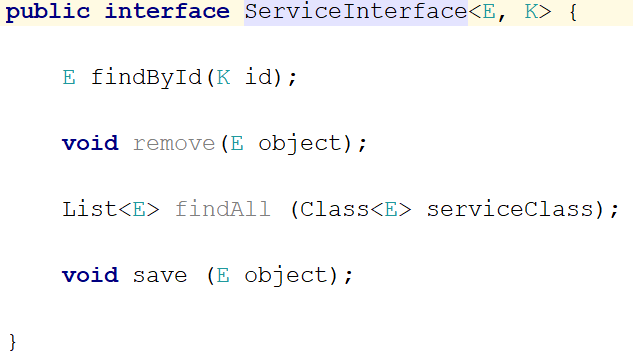


With that repository, you can implement methods that could perform CRUD operations with the database. The names of the methods define the type of query. For example, **findById** automatically means that method would return object with given Id, **findAll** method would automatically return all objects of the given type.

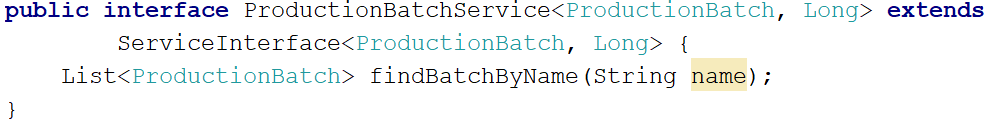
For now, in the **dao.api package** make the other repositories(daos) all other entities.

### Step 4 - Create the Service Layer

**Service layer** is the **link between** the **data access layer** (repositories) and the **presentation layer** (UI). Here we should make service interface for each of our entities. With that interface, we should be able to interact with the database and perform common CRUD operations for all entities.



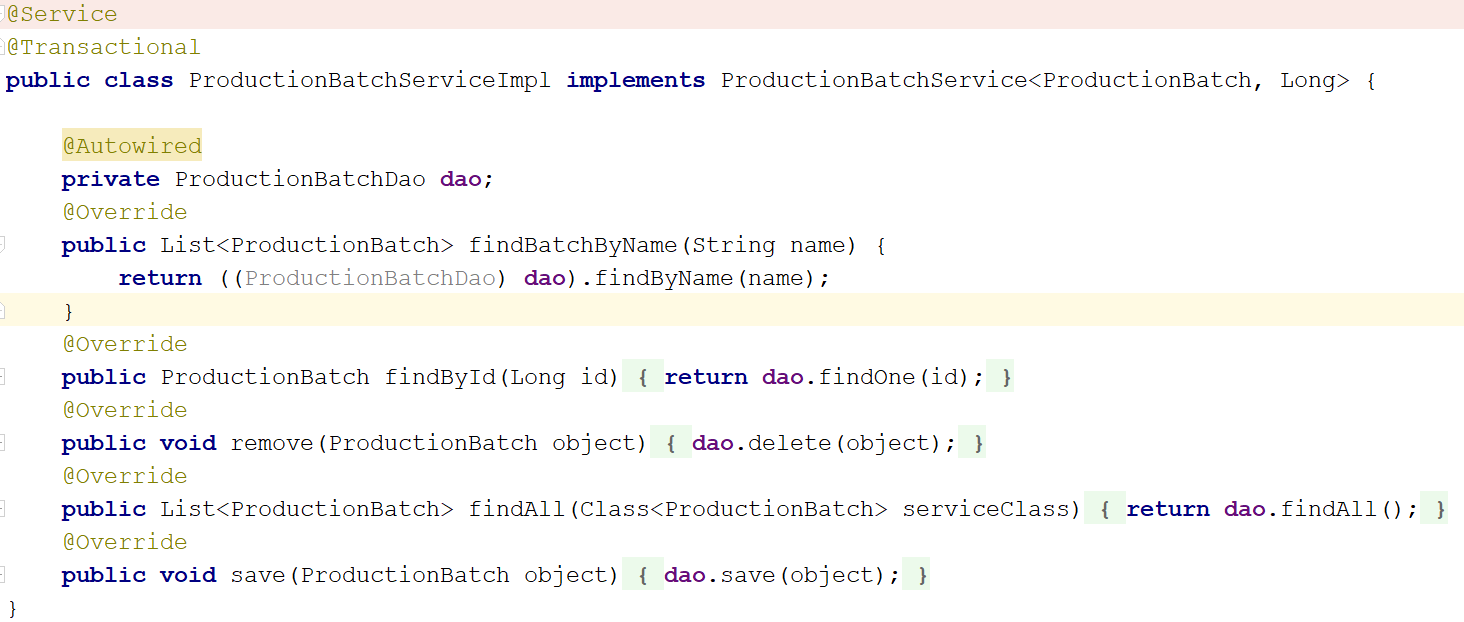
Then define separate Service Interface for all entities :



Once we have interfaces for each of our entities we should implement them. The implementation of the service needs instance of the repository we would use to perform CRUD operations. And we should annotate it with @Autowired.

The implementation class of the service implementation should be annotated with @Service and @Primary.

* @Service – that annotation is used by the spring framework to know where the business logic of the application is contained
* @Primary – that annotation is used by the spring framework to give preference to that class in case multiple candidates are qualified to autowire a single-valued dependency.

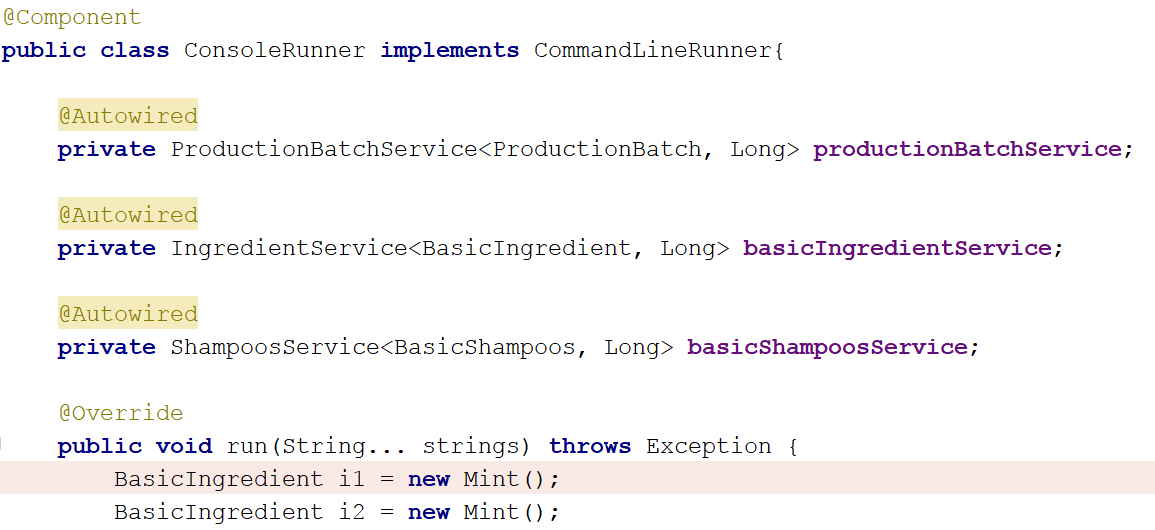


Make **interfaces and their implementations** for all of our remaining entities (**Books** and **Categories**). Put the interfaces in **service package** and their implementations in the **serviceImpl package**.

### Step 5 - 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.

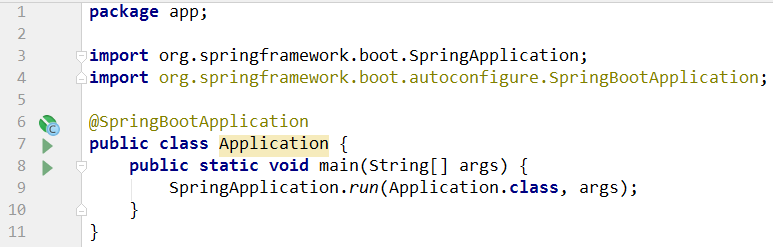
In the console package create new class called ConsoleRunner that would implement CommandLineRunner **interface** from org.springframework.boot package. That method would need instances of every of our services class (and make sure they are annotated with @Autowired so they can be injected by the spring framework). We should override the run() **method** and put our core logic of the program inside that method. Remember to put @Component before the class declaration so the spring framework should know about that class and would be able to use it.



Last but not least create Application class in the app folder. That class would be **the entry point of our program**. Inside of it just make main() **method** and one single line of code is required here:

SpringApplication.run(Application.class, args)

The final touch is to annotate the Application class with @SpringBootApplication so the spring framework should know from where to start executing the code.



### Step 6 – Add Custom Query Methods to Some Repositories

Let's add create a custom query method which is beyound the automaticaly provided ones by **JpaRepository**. Create a custom query method to find all shampoos containing an ingreadiend with given name :

