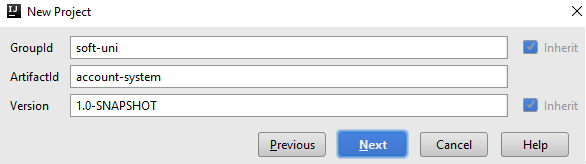
# Lab: Spring Data – Account System

This document defines the lab assignments for the [“Databases Frameworks” course at Software University](https://softuni.bg/courses/databases-advanced-hibernate).

Your task is to create a **simple account system** that has **users with accounts** and **manages money transfer or withdrawal**. Build the system using **code-firs**t and **Spring Data**. The goal is to implement **services** and **repositories**.

## Project Setup

Create new maven project:



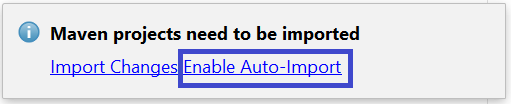
Add several dependencies in the pom.xml file:

|  |
| --- |
| 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**>  *<!-- https://mvnrepository.com/artifact/org.hibernate/hibernate-core -->* <**dependency**>  <**groupId**>org.hibernate</**groupId**>  <**artifactId**>hibernate-core</**artifactId**>  <**version**>5.2.3.Final</**version**>  </**dependency**>  <**dependency**>  <**groupId**>org.hibernate</**groupId**>  <**artifactId**>hibernate-entitymanager</**artifactId**>  <**version**>5.2.3.Final</**version**>  </**dependency**>  *<!-- https://mvnrepository.com/artifact/mysql/mysql-connector-java -->* <**dependency**>  <**groupId**>mysql</**groupId**>  <**artifactId**>mysql-connector-java</**artifactId**>  <**version**>5.1.6</**version**>  </**dependency**> </**dependencies**> |

In the resources folder, add new **applications.properties** file, which will hold the Spring configuration of the project:

|  |
| --- |
| application.properties |
| *#Data Source Properties* **spring.datasource.driverClassName** = **com.mysql.jdbc.Driver spring.datasource.url** = **jdbc:mysql://localhost:3306/account\_system?createDatabaseIfNotExist=true&amp;useSSL=false" spring.datasource.username** = **root spring.datasource.password** = **1234** *###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** *#JPA Properties* **spring.jpa.properties.hibernate.dialect** = **org.hibernate.dialect.MySQL5InnoDBDialect spring.jpa.properties.hibernate.format\_sql** = **TRUE spring.jpa.hibernate.ddl-auto** = **update** |

Enable auto-import so Maven can download the external libraries:



Start splitting the java directory into packages. Create several ones to help you organize your project:

* **models** – the directory of our database models(entities)
* **repositories** – the package where we will hold the repository intefaces
* **services** – where our service interfaces and implementations will be stored

## Database Models

Start by setting up the database models. Each one of them will be as follows:

* **User**
* **Id** – long value**, primary key**
* **Username** – **unique** for each user
* **Age** – integer value
* **Accounts** – each user can have **many accounts**, which will be **identified by their id**
* **Account**
* **Id** – long value, **primary key**
* **Balance** – BigDecimal, cannot be negative
* **User** – an account can be owned by a **single user**

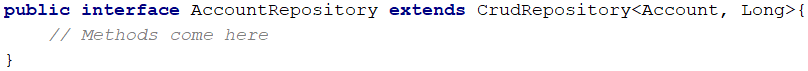
Set up appropriate tables, columns, column properties and table relations.

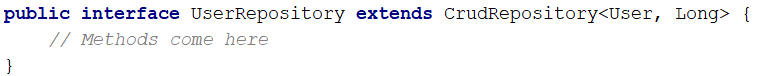
## Repositories

Spring Data reduces the amount of boiler-plate code by using a central interface Repository.   
The CrudRepository inteface contains methods like:

* save(E entity)
* findOne(Id primaryKey)
* findAll()
* count()
* delete(E entity)
* exists(Id primaryKey)

You can define a **custom repository**, which extends the CrudRepository and defines several methods for operating with data besides those exposed by the greater inteface. The query builder mechanism of Spring Data requires following several rules when you define custom methods. Query creation is done by parsing method names by prefixes like find…By, read…By, query…By, count…By, and get…By. You can add more criteria by concatenating And and Or or apply ordering with OrderBy with sorting direction Asc or Desc.

Create two Repository interfaces – UserRepository and AccountRepository. 

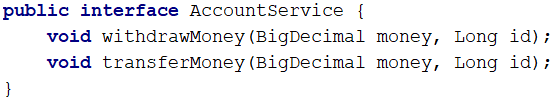


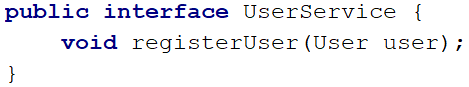
Add several methods to help you look up the data source, for example getByUsername(String username) in the UserRepository inteface.

## Services

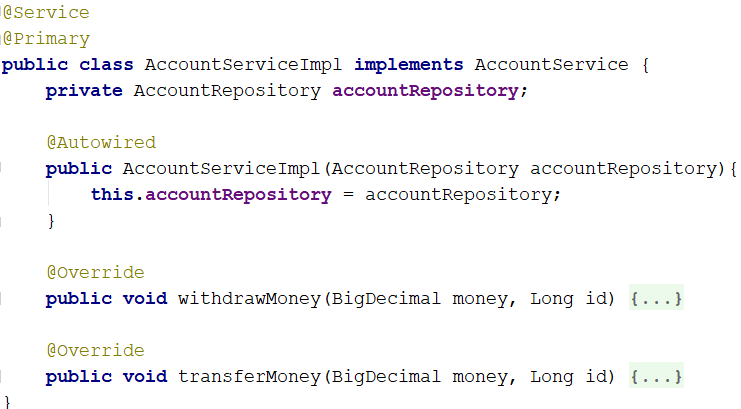
In bigger applications mixing business logic and crud operations to the database is not wanted. Having a repository objects is implementing the **Domain Driven Design**. Repositories are classes reposible **only for write/transactional operations** towards the data source. Any business logic like validation, calculations and so on is implemented by **a Service Layer**. One of the most important concepts to keep in mind is that a **service** should **never expose details of the internal processes** or the business entities used within the application.

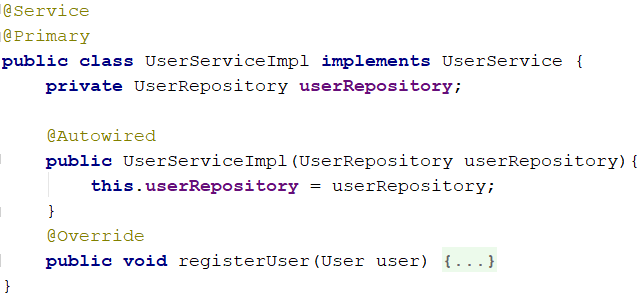
Define several service **intefaces**:





Implement those services with classes AccountServiceImpl and UserServiceImpl. Those classes will do the business logic of the application. In order to do that, they should have certain type of Repository available – AccountRepository or UserRepository according to the service type.





In Spring Data Framework the usage of @Service, @Repository or @Component annotations is needed to separate different “**layers**” in the application. They are mainly used for programmers to know a class’s role and which logical layer it belongs to.

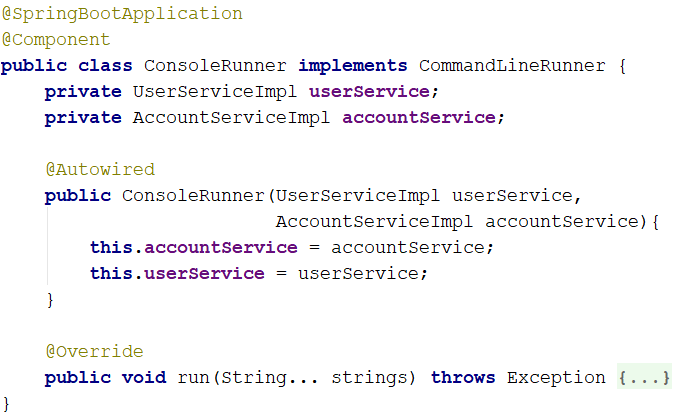
The @Autowired annotation is required when **injecting a resource**, e.g. **Repository** to **Service**.

The implementation of the methods is up to you. Here are some several tips:

* AccountServiceImpl
  + Money withdrawal – should only happen if account **is present** in the database, **belongs to user** and **has enough balance**
  + Money transfer – should only happen if **account belongs to user** and transfer value **is not negative**
* UserServiceImpl
  + User registration – should only happen if user does not exist in the database

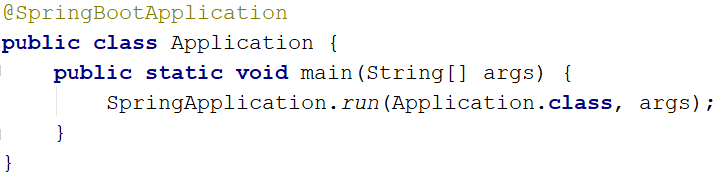
## ConsoleRunner and Application

We will test our application in a ConsoleRunner class. Create such and inject needed repositories:



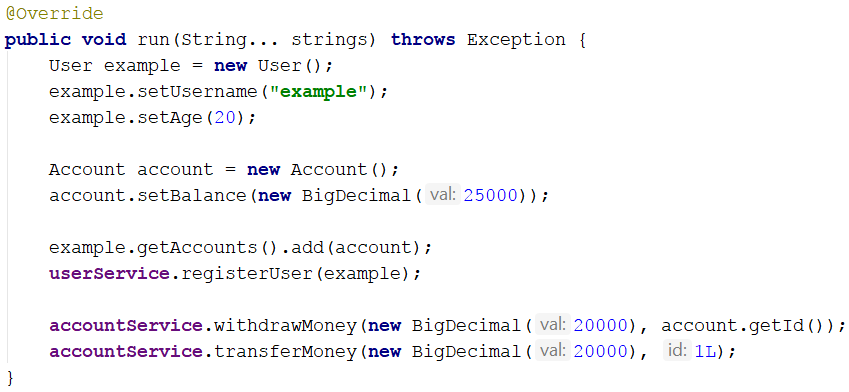
The @SpringBootApplication annotation is used to annotate classes, which will be **starters** to our Spring application.

Create a class Application, which will be used to start our app.



## Test

Test the application by adding some logic in the ConsoleRunner class’s method run:



If you’ve written everything correctly, an **account\_system** database should be created with tables:

* **users**
* **accounts**
* **users\_accounts**