Java and Spring MVC

1 Overview

For this website, I chose to build it using Spring MVC tools. The spring web Model-View-Controller architecture and some components that can be used to develop flexible and loosely coupled web application. The spring Model-View-Controller framework is designed around a Dispatcher Servlet that handles all the HTTP requests and responses. The diagram below shows how it works within a Spring MVC framework.

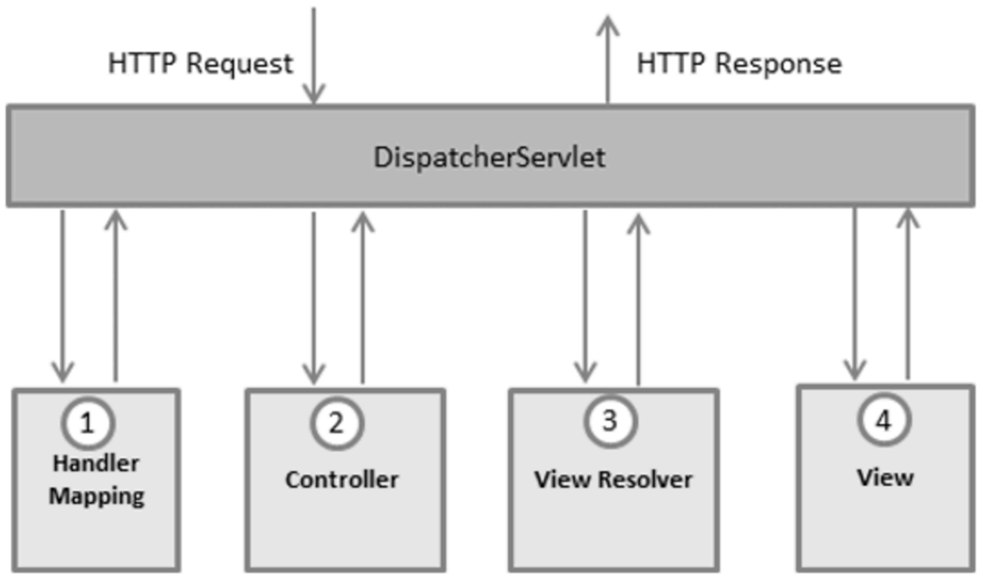


figure 4.1 Spring MVC structure

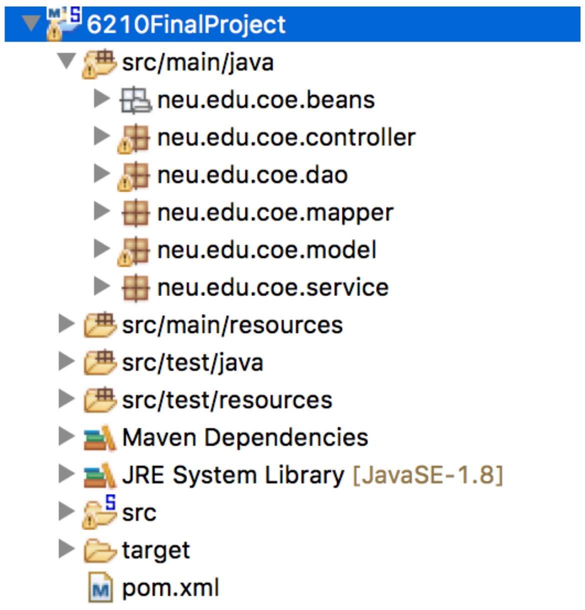
When browsers sending out requests to the DispatcherServlet, it will consult the Handler Mapping to call the appropriate Controller. Then the controller takes the request and calls the appropriate service method based on the used GET or POST method. The service method will set model data based on defined business logic and returns view name to the Dispatcher Servlet. The Dispatcher Servlet will take help from ViewResolver to pick up defined view for the request. Once view is finalized, the Dispatcher Servlet passes the model data to the view which is finally rendered on the browser.

Why Spring MVC?

As you can see, using Spring MVC can effectively separate business logic from back end database and front end user. This kind of well-structured architecture will upgrade the application's security. Spring also have a lot of other advantages. For example, spring enables the developers to develop the enterprise application using Plain Old Java Objects(POJOs), so that I do not need an enterprise container such as an application server but I have the option of using a robust servlet container. Meantime, spring is lightweight and it minimum invasive development with POJOs.

2 Project repository

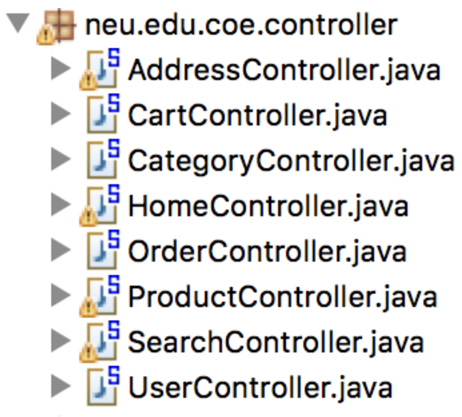
Below is a screen shot of my project repository. The neu.edu.coe.controller package contains all the controller class. The neu.edu.coe.model package contains all the POJOs of my project. The neu.edu.coe.mapper package contains all the mappers for the POJOs. Same as POJO class, it is related to the corresponding tables in the database. The neu.edu.coe.dao package contains all the classes which is responsible for interactions with database. I'm using maven to manage the dependencies injection, the pom.xml file contains all the maven dependencies I used in my project.



```
6210FinalProject
  src/main/java
    neu.edu.coe.beans
    neu.edu.coe.controller
    neu.edu.coe.dao
    neu.edu.coe.mapper
    neu.edu.coe.model
    neu.edu.coe.service
  src/main/resources
  src/test/java
  src/test/resources
  Maven Dependencies
  JRE System Library [JavaSE-1.8]
  src
  target
  pom.xml
```

3 Controllers

The controller is responsible for processing user requests and building an appropriate model and pass it to the view for rendering. As you can see, for this application, I have several different controllers handling different functions. HomeController is in charge of user actions. For example, the function that direct users to their home

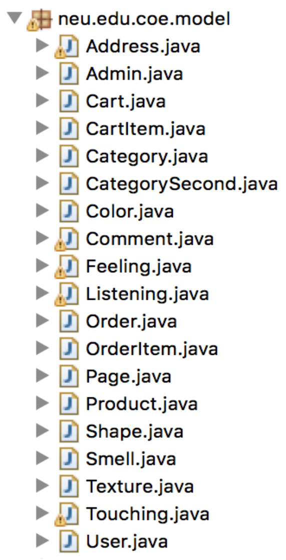


```
neu.edu.coe.controller
  AddressController.java
  CartController.java
  CategoryController.java
  HomeController.java
  OrderController.java
  ProductController.java
  SearchController.java
  UserController.java
```

page is mapped inside the HomeController. So is the mapping function when user tries to log out or register. The UserController contains several functions that will change the user tables behind the scene, like when a user tries to update his/her profile information. The CategoryController is in charge of all the functions which is category related. Category is a table which have a many-to-one relationship with product table. Through category class, it will be more convenience for administrator to manage all the products. Like its name, the SearchController is responsible for search related functions. For my website, I have a convenience search engine that user will be able to gradually specifies their search keyword based on their needs. Similarly, OrderController and ProductController gathers all the functions which interacts with order and product table, respectively. The CartController have all those functions that will enable users to add any products into their cart and when they check out, the cart will be cleared up. When user tries to check out, the website will need a shipping address for the order. All the functions which is address related, like create new address, are mapped inside the AddressController. It associates with some Jsp pages stored inside the views file, which represents the View of Model-View-Controller framework.

4 Models

Here is a view of the neu.edu.coe.model package. It contains all the model class of my project. The model encapsulates the application data and in general they will consist of POJOs. All the POJOs are related with the database tables. Each table corresponds to one POJO inside this package. A POJO is a Java class not bound by any restriction other than those forced by the Java Language Specification. In this project, I have 18 tables inside database, and most of them are mapped to a POJO in the model package. Each POJO defines all the variables which are all attributes inside the corresponding entity. Take user as an example, User POJO have 7 variables. They are uid (integer), username(string), password(string), enabled(Boolean), authority(string), email(string) and phone(string). The users table, which is corresponding to the User POJO, share those 7 attributes with the User POJO.



```
neu.edu.coe.model
    Address.java
    Admin.java
    Cart.java
    CartItem.java
    Category.java
    CategorySecond.java
    Color.java
    Comment.java
    Feeling.java
    Listening.java
    Order.java
    OrderItem.java
    Page.java
    Product.java
    Shape.java
    Smell.java
    Texture.java
    Touching.java
    User.java
```

5 DAO classes

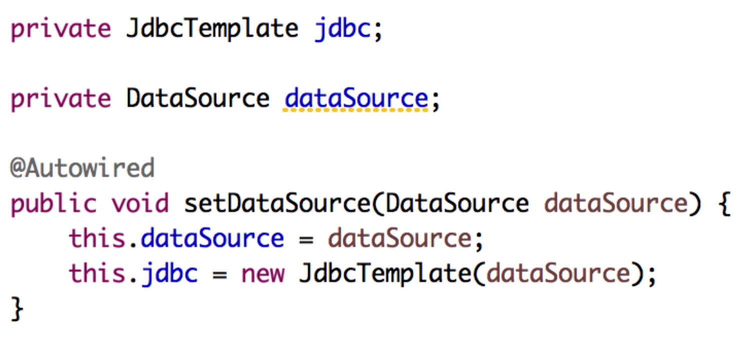
Below is a screen shot of what's inside the neu.edu.coe.dao package. As you can see, every model I built, have a corresponding Dao class for it. This is because dao classes are responsible for the connection between the database and the business logic of this project. Each Dao class contains the SQL queries for the related entity. Dao stands for Data Access Object, an object that provides an abstract interface to some type of database or persistence mechanism, providing some specific operations without exposing details of the database. Having a dao layer will separate the database operation behind the scene with the business logic of the application.

To illustrate more details about the Dao class, I need to explain how I connect the database with my Spring MVC project first.

6 JDBC and JdbcTemplate

JDBC stands for Java Database Connectivity. It is an application programming interface for Java language, which defines how a client may access a database. It provides methods to query and update data in a database, and is oriented towards relational databases.

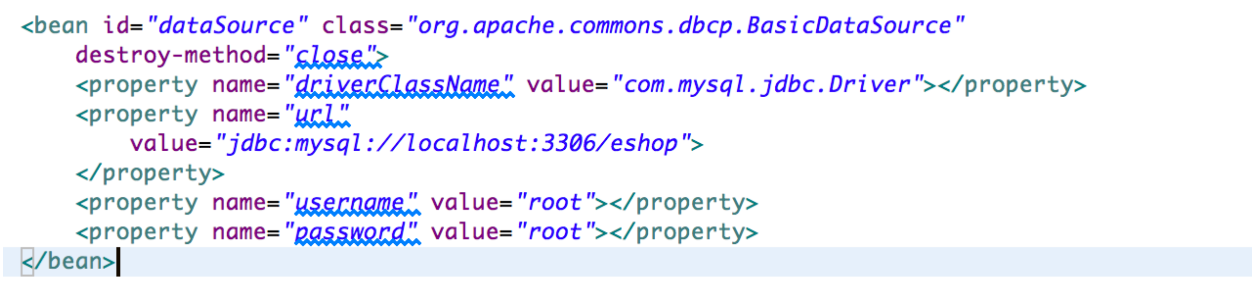
For this project, I use JdbcTemplate to connect with database. JdbcTemplate is the central class in the JDBC core package. It simplifies the use of JDBC and helps to avoid common errors. Below is a fragment of codes inside



```java
private JdbcTemplate jdbc;

private DataSource dataSource;

@Autowired
public void setDataSource(DataSource dataSource) {
    this.dataSource = dataSource;
    this.jdbc = new JdbcTemplate(dataSource);
}
```

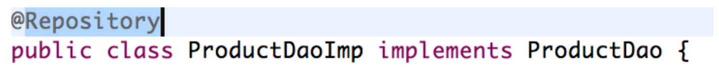


```xml
<bean id="dataSource" class="org.apache.commons.dbcp.BasicDataSource"
    destroy-method="close">
    <property name="driverClassName" value="com.mysql.jdbc.Driver"></property>
    <property name="url"
        value="jdbc:mysql://localhost:3306/eshop">
    </property>
    <property name="username" value="root"></property>
    <property name="password" value="root"></property>
</bean>
```

Dao class. I initialize a JdbcTemplate with a DataSource variable. That is where I obtain connection from. Meantime, I initialize a bean named dataSource, contains all the information and JDBC driver class needed to build a JDBC connection. The configuration is shown below. The property “url” stands for the database I use in MySQL workbench. The property “username” and “password” stands for the database username and password. A Java application need a software component to interact with a database. So JDBC requires drivers for each database. The property “driverClassName” specifies the driver package I’m using. From the above, I can make a connection between the application and database. Now we can get into more details of the Dao layer.

7 Dao layer

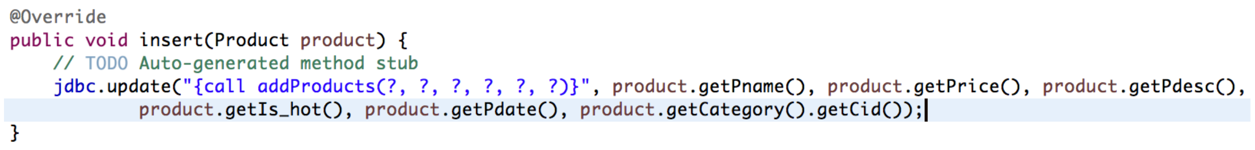
As I said before, Dao layer is the realization of connectivity between the application and the database. Take ProductDaoImp as an example.



```
@Repository
public class ProductDaoImp implements ProductDao {
```

I use annotation @Repository to annotate all the DaoImp class. There are several different annotations in Spring MVC. For example, @Component is a generic stereotype for any spring-managed component. @Repository is a specific type of @Component. It is a specific use case of @Component for persistence layer.

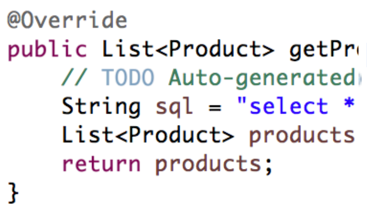
There are some functions related to products inside ProductDaoImp, they all interact



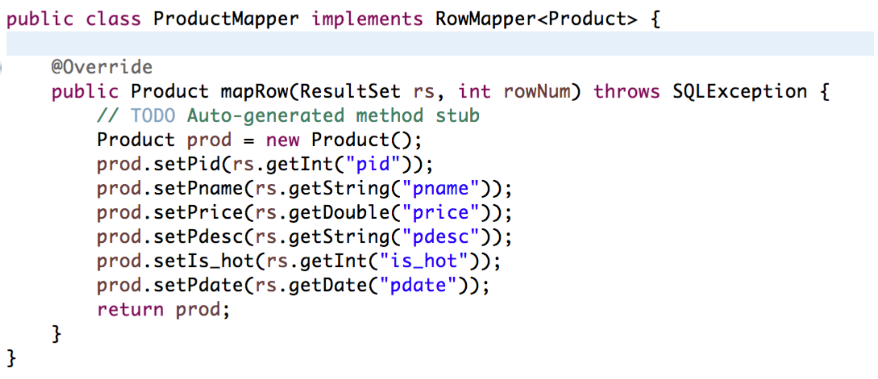
```java
@Override
public void insert(Product product) {
    // TODO Auto-generated method stub
    jdbc.update("{call addProducts(?, ?, ?, ?, ?, ?)}", product.getPname(), product.getPrice(), product.getPdesc(),
            product.getIs_hot(), product.getPdate(), product.getCategory().getCid());
}
```

with Product entity in the database using JdbcTemplate. The diagram below shows a fragment of codes in ProductDaoImp. This insert method takes a product object as argument and calls the update method from JdbcTemplate. The first argument it passes in is a string, it is also an executable SQL query which will call the stored procedure “addProduct” defined in the database. All the stored procedure explanation will be at the SQL explanation part of this report. But from the appearance of this SQL, it will add a product record based on 6 attributes which is represented by 6 question marks in the string. Followed by the string, there are 6 more arguments for this update method, each extract a variable from the passed in product object. Those 6 variables will replace the 6 question marks and being added to the Product entity as a record in the back-end database. Thus, complete the adding product function for the website administrator.

Below is another function inside the ProductDaoImp. This getProduct() function takes



```java
@Override
public List<Product> getPr
    // TODO Auto-generated
    String sql = "select *
    List<Product> products
    return products;
}
```



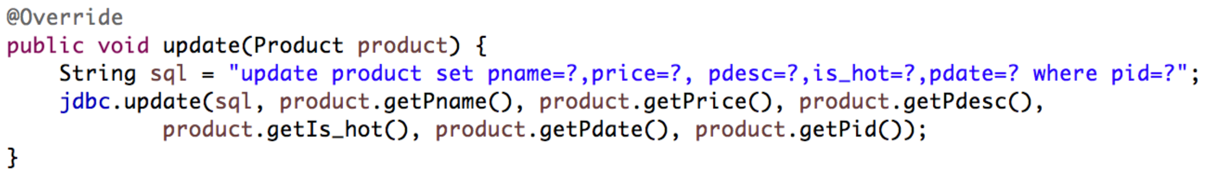
```java
public class ProductMapper implements RowMapper<Product> {

    @Override
    public Product mapRow(ResultSet rs, int rowNum) throws SQLException {
        // TODO Auto-generated method stub
        Product prod = new Product();
        prod.setPid(rs.getInt("pid"));
        prod.setPname(rs.getString("pname"));
        prod.setPrice(rs.getDouble("price"));
        prod.setPdesc(rs.getString("pdesc"));
        prod.setIs_hot(rs.getInt("is_hot"));
        prod.setPdate(rs.getDate("pdate"));
        return prod;
    }
}
```

no argument and return a List of products. The

String inside this method is simply a select query, returns all the products stored in the Product entity. This time, I use the query method provided by the JdbcTemplate, pass in the select SQL query and a ProductMapper object. The result get from a JDBC connection is an object of type ResultSet. To make it become the POJO objects I expected, I need a Mapper. ProductMapper is a RowMapper object. The diagram shows its implementation of RowMapper interface. RowMapper is an interface used by JdbcTemplate for mapping rows of a ResultSet on a per-row basis. As we can see, implementations of this interface perform the actual work of mapping each row to a result object, and we don't need to worry about exception handling. Then the passed in argument, which is an object of ResultSet will be transformed as a Product object. I created RowMapper for POJOs based on their needs and put them inside the neu.edu.coe.mapper package. Right now, if we pass in the RowMapper for product when we try to execute the select query using JdbcTemplate, we can get parsed data from the database as product objects.

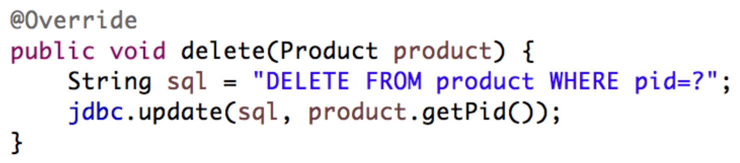
The next method is update method inside the ProductDaoImp class. As you can see,



```
@Override
public void update(Product product) {
    String sql = "update product set pname=?,price=?, pdesc=?,is_hot=?,pdate=? where pid=?";
    jdbc.update(sql, product.getPname(), product.getPrice(), product.getPdesc(),
            product.getIs_hot(), product.getPdate(), product.getPid());
}
```

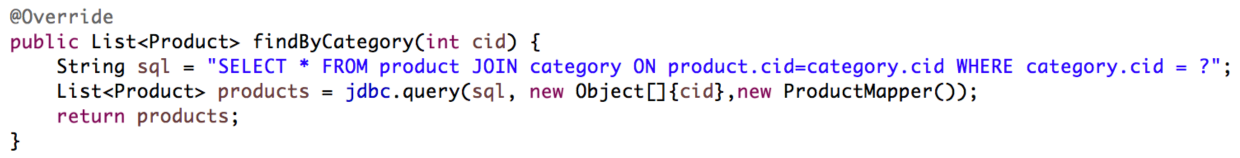
the update method uses the same function of JdbcTemplate as insert method. In fact, those two have no difference except the string passed in, which is the actual SQL query. Similarly, the 6 arguments will replace the six questions mark inside the string based on their position, form an actual executable SQL query, then executed by the JdbcTemplate. The syntax of the query is exactly the same with the actual MySQL query.

Next is the delete method of ProductDaoImp class. From the fragment, we can see the delete method also take a product object as an argument. Inside the string, I wrote a delete SQL query, ask for delete a row of Product entity based on the primary key of Product entity, which is “pid”. At the end of this method, I passed in the string and get the pid from the passed-in product object then passed it in to replace the question mark inside the string query.



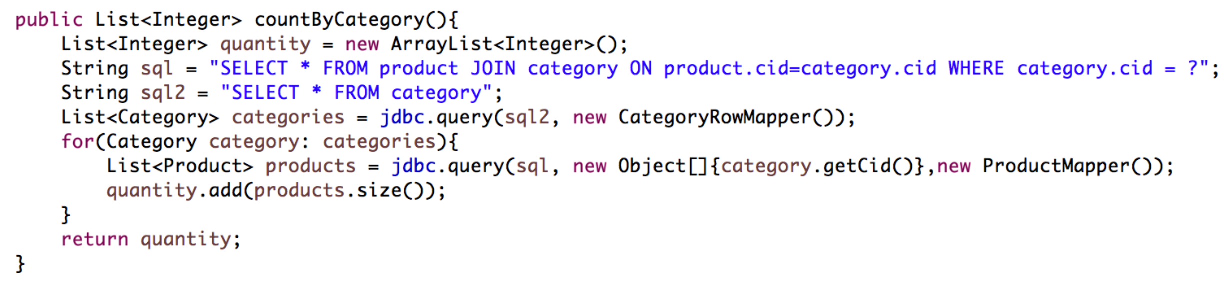
```
@Override
public void delete(Product product) {
    String sql = "DELETE FROM product WHERE pid=?";
    jdbc.update(sql, product.getPid());
}
```

Product entity and Category entity have an optional many to mandatory one relationship. The findByCategory method gives a way to find all the products which belongs to certain category. As show below, the category as a condition should be passed in as an argument, so the primary key of Category entity, the “cid”, be passed into the method. The string here is a select query to product entity with an JOIN with category entity. The syntax here is the same with MySQL query. For this method, I use query method provided by the JdbcTemplate and passed in ProductMapper for row mapping.



```
@Override
public List<Product> findByCategory(int cid) {
    String sql = "SELECT * FROM product JOIN category ON product.cid=category.cid WHERE category.cid = ?";
    List<Product> products = jdbc.query(sql, new Object[]{cid},new ProductMapper());
    return products;
}
```

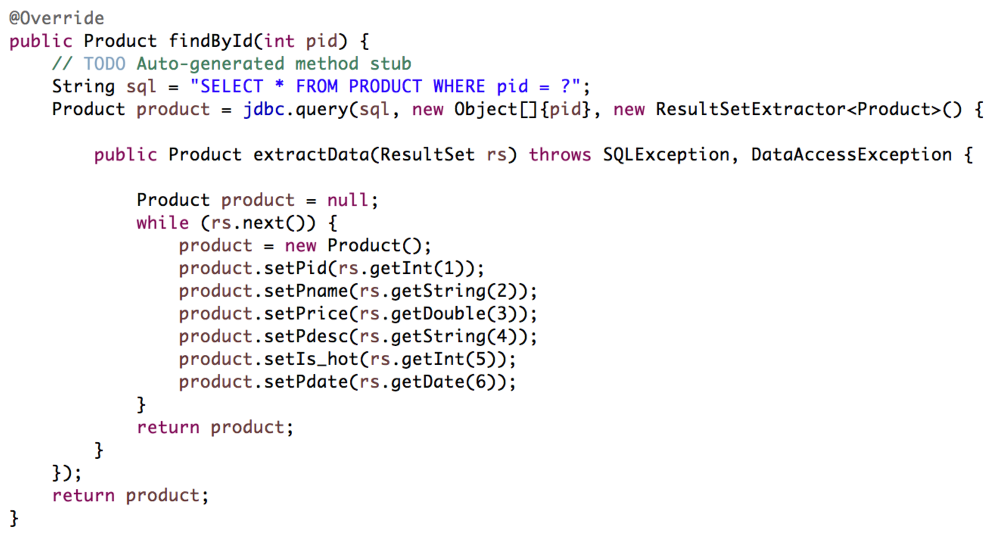
To count how many products belongs to each certain category, I added countByCategory() method. In this method, I first use the query method provided by the



```
public List<Integer> countByCategory(){
    List<Integer> quantity = new ArrayList<Integer>();
    String sql = "SELECT * FROM product JOIN category ON product.cid=category.cid WHERE category.cid = ?";
    String sql2 = "SELECT * FROM category";
    List<Category> categories = jdbc.query(sql2, new CategoryRowMapper());
    for(Category category: categories){
        List<Product> products = jdbc.query(sql, new Object[]{category.getCid()},new ProductMapper());
        quantity.add(products.size());
    }
    return quantity;
}
```

JdbcTemplate and get all the categories I have from Category Entity. Then I use a for loop to iterate through all the categories, query and get all the products that belongs to each category then save all the numbers into an Integer List. The SQL I use here is the same with the SQL from the findByCategory() method. They are both standard SQL queries.

The following diagram shows a method which can get certain product based on the primary key. This method takes a pid, the primary key of Product entity, as an argument, then return a Product object. The SQL query here is a simple select query based on the pid value. What's different is, I pass in a ResultSetExtractor()in this method instead of Mapper. The difference is that



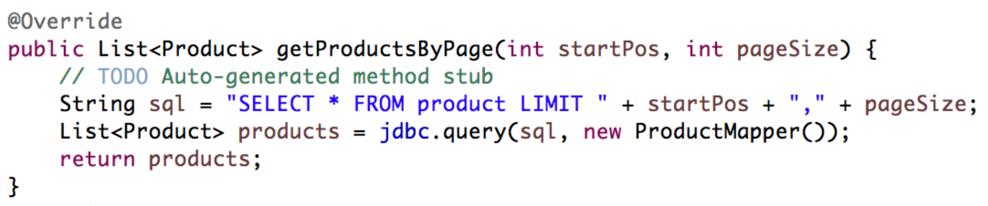
```
@Override
public Product findById(int pid) {
    // TODO Auto-generated method stub
    String sql = "SELECT * FROM PRODUCT WHERE pid = ?";
    Product product = jdbc.query(sql, new Object[]{pid}, new ResultSetExtractor<Product>() {

        public Product extractData(ResultSet rs) throws SQLException, DataAccessException {

            Product product = null;
            while (rs.next()) {
                product = new Product();
                product.setPid(rs.getInt(1));
                product.setPname(rs.getString(2));
                product.setPrice(rs.getDouble(3));
                product.setPdesc(rs.getString(4));
                product.setIs_hot(rs.getInt(5));
                product.setPdate(rs.getDate(6));
            }
            return product;
        }
    });
    return product;
}
```

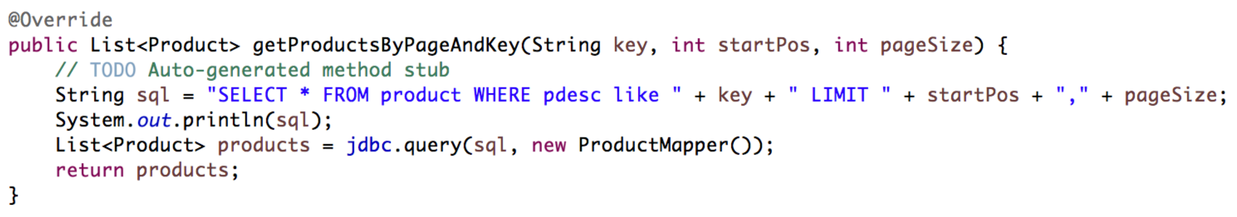
when I pass in a string and mapper to the query function of JdbcTemplate, the default result type would be a collection of an object, which in this case we need the result type to be one specific object type. So I choose to pass in a ResultSetExtractor(), which does pretty much the same thing with a mapper. They all transforms a ResultSet into certain type of object.

Here are two methods that have a different kind of function than others. The



```java
@Override
public List<Product> getProductsByPage(int startPos, int pageSize) {
    // TODO Auto-generated method stub
    String sql = "SELECT * FROM product LIMIT " + startPos + "," + pageSize;
    List<Product> products = jdbc.query(sql, new ProductMapper());
    return products;
}
```

getProductsByPage() method. Apparently, this method is a simply select method that will get all the products except it constrains the result show only certain few records which is between requested region. The startPos and pageSize are two passed in argument and they form the needed section for the result. The SQL in this method use a keyword “LIMIT” to accomplish that.



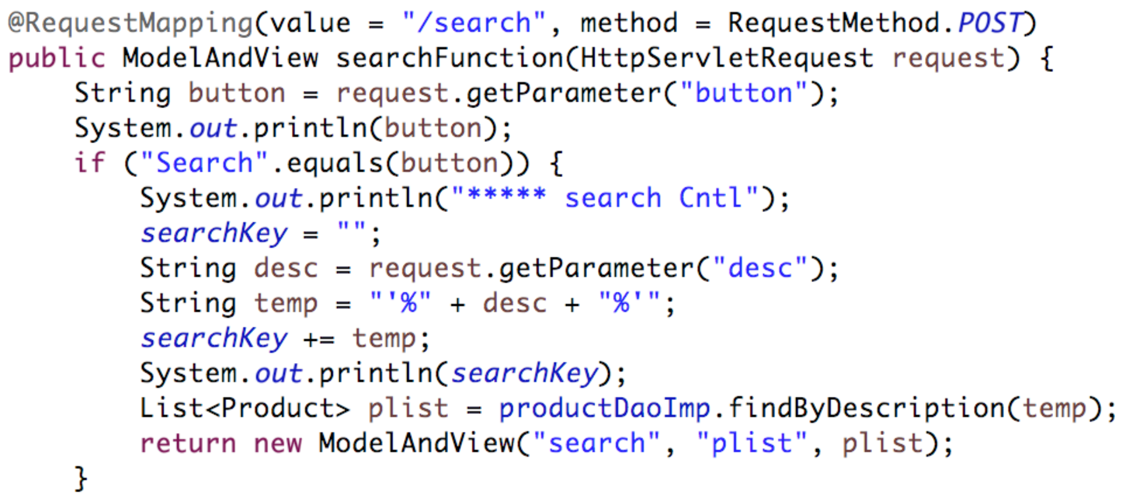
```java
@Override
public List<Product> getProductsByPageAndKey(String key, int startPos, int pageSize) {
    // TODO Auto-generated method stub
    String sql = "SELECT * FROM product WHERE pdesc like " + key + " LIMIT " + startPos + "," + pageSize;
    System.out.println(sql);
    List<Product> products = jdbc.query(sql, new ProductMapper());
    return products;
}
```

Similarly, getProductsByPageAndKey() method shown below can explain itself. As we can see, this method needs one more argument than the one before. It needs a String to be the key. From the SQL query we can see it is a simple select method which will find all the products whose description (“pdesc”) contains the key word. Then again, it limits the result to show only those ones which is within the required section.

8 Search Function

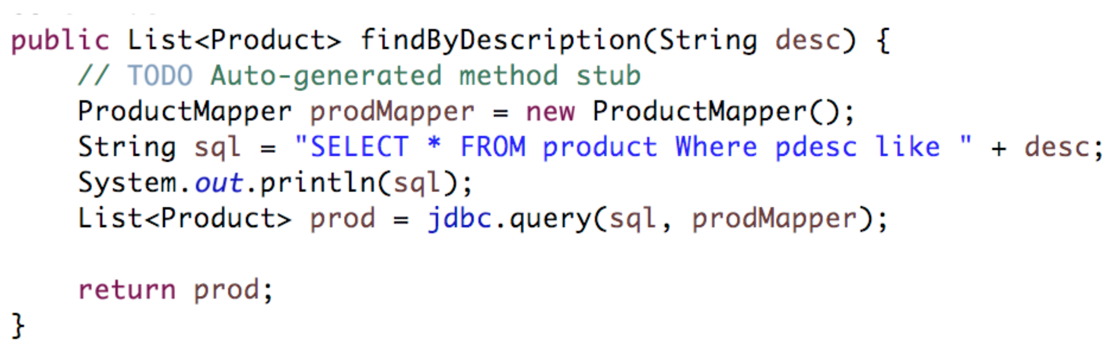
In this part, I will be introducing the function named as “deep search”. I made effort to try to simulate how visual impairment people use search engine. And I found the answer was it was difficult for them to use a normal search engine like normal people use. Because it is hard for them to modify and delete a specific letter that they typed. So I redesigned the search engine to simplify the usage of people who have visual impairment.

Here’s the source code.



```
@RequestMapping(value = "/search", method = RequestMethod.POST)
public ModelAndView searchFunction(HttpServletRequest request) {
    String button = request.getParameter("button");
    System.out.println(button);
    if ("Search".equals(button)) {
        System.out.println("***** search Cntl");
        searchKey = "";
        String desc = request.getParameter("desc");
        String temp = "'%" + desc + "%'";
        searchKey += temp;
        System.out.println(searchKey);
        List<Product> plist = productDaoImp.findByDescription(temp);
        return new ModelAndView("search", "plist", plist);
    }
```

The First line is mapping url to a search service. If a user using the search function it will be mapped to this controller. I made 2 bottoms one is “Search” and one is “Search again”. If a user trigger the “Search” bottom, the controller will read what user typed as “desc” and pass the “desc” to database, using the sql like:

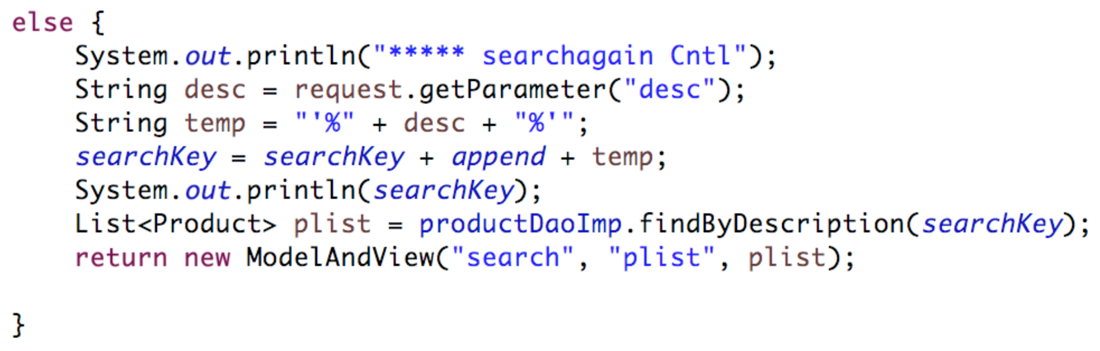


```
public List<Product> findByDescription(String desc) {
    // TODO Auto-generated method stub
    ProductMapper prodMapper = new ProductMapper();
    String sql = "SELECT * FROM product Where pdesc like " + desc;
    System.out.println(sql);
    List<Product> prod = jdbc.query(sql, prodMapper);

    return prod;
}
```

“SELECT * FROM product Where pdesc like” + “%” “desc” “%”. And retrieve data from database to show on the page. Then we finish a one-time search.

When a user want to have a deeper search, for example, a user searches “table” first and then he wants to be more specific, so he want to add an attribute like “rectangular”. All he needs to do is type “rectangular” and trigger “Search again”. The program will automatically add two keys words into one query.



```
else {
    System.out.println("***** searchagain Cntl");
    String desc = request.getParameter("desc");
    String temp = "'%" + desc + "%'";
    searchKey = searchKey + append + temp;
    System.out.println(searchKey);
    List<Product> plist = productDaoImp.findByDescription(searchKey);
    return new ModelAndView("search", "plist", plist);

}
```

When a user uses “Search”, the key word will store in a parameter called “searchKey” and when a user uses “Search again” the application will add the key word to “searchKey” and inject SQL to database like:

“SELECT * FROM product Where pdesc like” + “%” “desc1” “%” + “%” “desc2” “%”

So, no matter how many times a user will search, the more times the user search the more accurate answer the user will get.

If the user want to have a new search, he just need to trigger the “Search” bottom and all records will be removed.