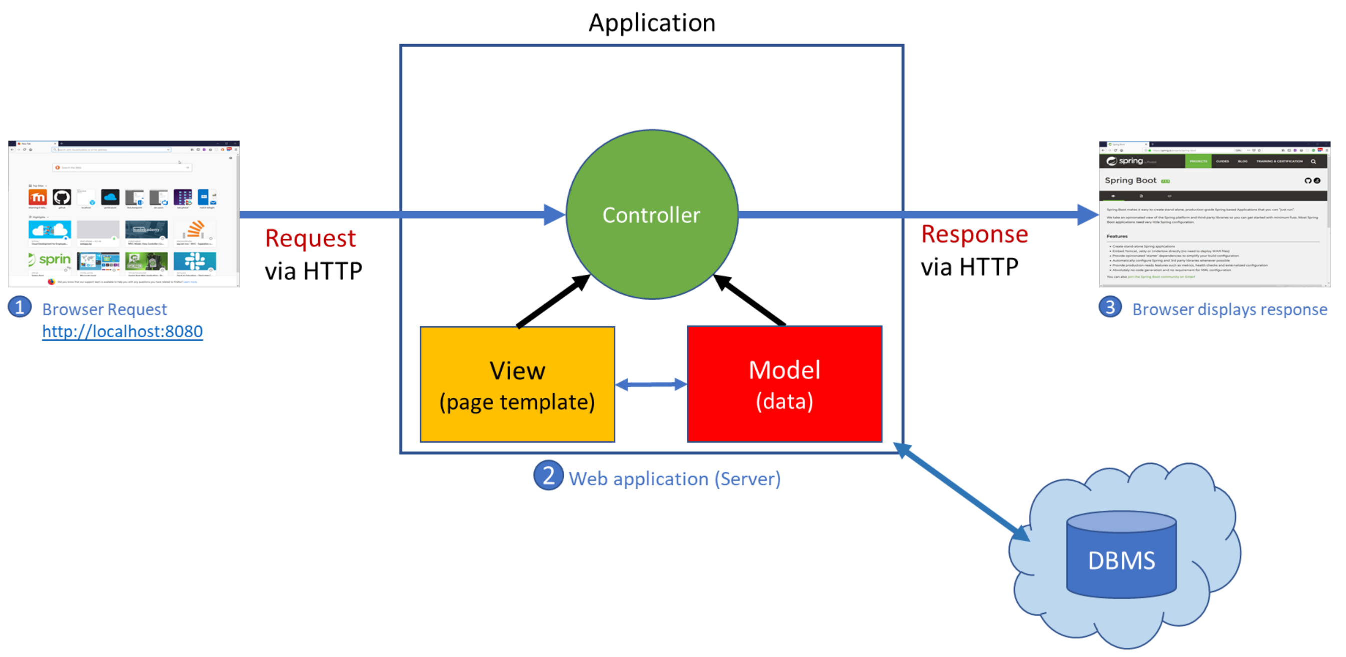
# Spring Boot web application with JDBC (Part 1)

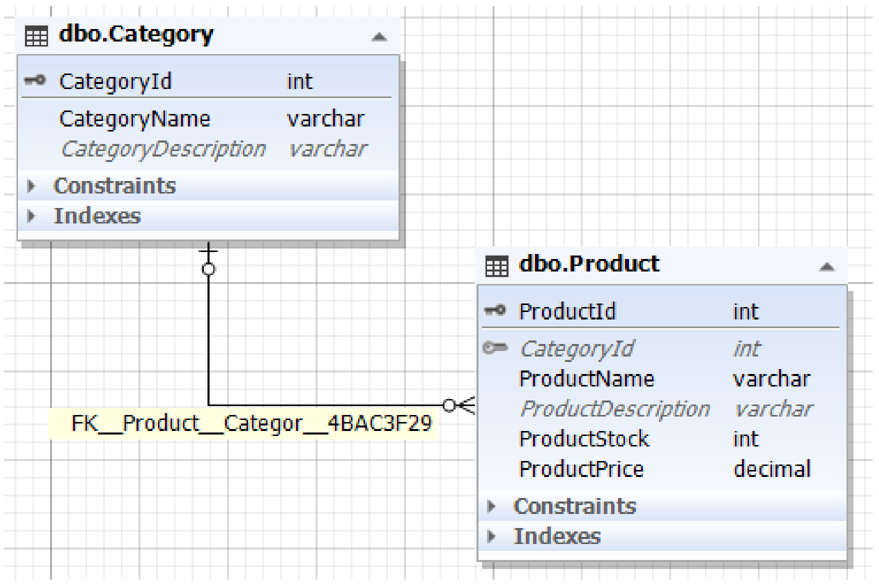
## Overview

At the end of part 1 you should have had a working Spring Boot application, using view templates and linking pages together.

This tutorial will database functionality, using JDBC (Java Database Connectivity) to retrieve stored data. `Model` classes will be used to represent that data in the application.



The database will be the SQL database previously configured in Microsoft Azure. It contains two tables in a one-to-many relationship.



**dbo.Category**

| | |
|---|---|
| CategoryId | int |
| CategoryName | varchar |
| *CategoryDescription* | *varchar* |

▸ Constraints
▸ Indexes

FK__Product__Categor__4BAC3F29

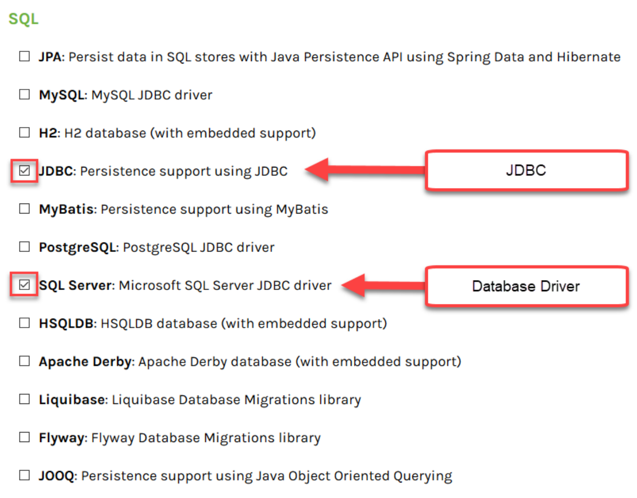
**dbo.Product**

| | |
|---|---|
| ProductId | int |
| *CategoryId* | *int* |
| ProductName | varchar |
| *ProductDescription* | *varchar* |
| ProductStock | int |
| ProductPrice | decimal |

▸ Constraints
▸ Indexes

## 1. Adding support for JDBC and MS SQL to the Spring Boot web app

The web app from the last tutorial is missing SQL dependencies required to access a database. These should be selected in the SQL section of dependencies when creating a new project.

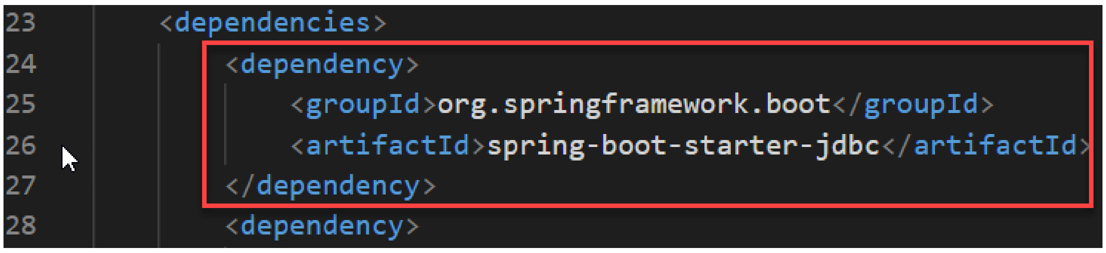


SQL

- ☐ **JPA**: Persist data in SQL stores with Java Persistence API using Spring Data and Hibernate
- ☐ **MySQL**: MySQL JDBC driver
- ☐ **H2**: H2 database (with embedded support)
- ☑ **JDBC**: Persistence support using JDBC
- ☐ **MyBatis**: Persistence support using MyBatis
- ☐ **PostgreSQL**: PostgreSQL JDBC driver
- ☑ **SQL Server**: Microsoft SQL Server JDBC driver
- ☐ **HSQLDB**: HSQLDB database (with embedded support)
- ☐ **Apache Derby**: Apache Derby database (with embedded support)
- ☐ **Liquibase**: Liquibase Database Migrations library
- ☐ **Flyway**: Flyway Database Migrations library
- ☐ **JOOQ**: Persistence support using Java Object Oriented Querying

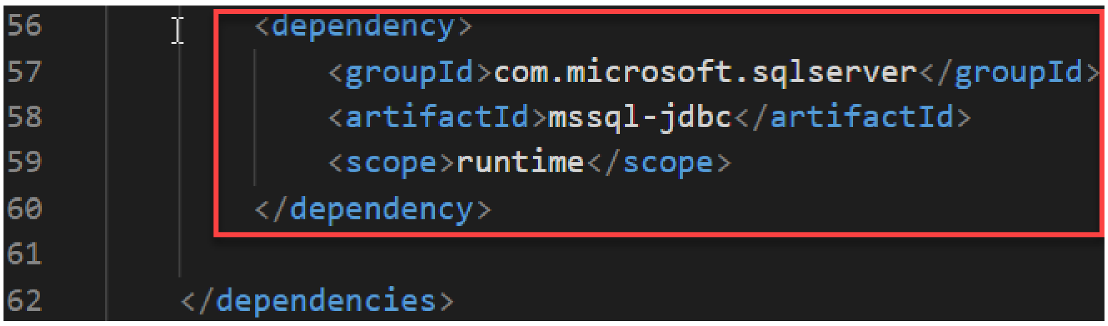
These dependencies can also be added to an existing application by editing the `pom.xml` file and adding to the dependencies section. If unsure, create a new app with all the required dependencies, download, and examine the pom.xml.

The first dependency required is `spring-boot-starter-jdbc`:



```
<dependencies>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-jdbc</artifactId>
    </dependency>
    <dependency>
```

The second is the database driver `com.microsoft.sqlserver`:



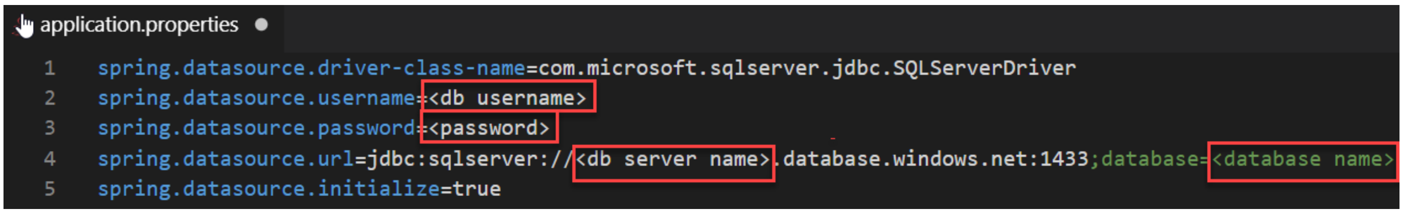
```
    <dependency>
        <groupId>com.microsoft.sqlserver</groupId>
        <artifactId>mssql-jdbc</artifactId>
        <scope>runtime</scope>
    </dependency>

</dependencies>
```

Add these, save `pom.xml`. Then run `mvn install` in the terminal.

## 2. Configure the application for database connectivity

The application needs to be configured so that it can connect to the database, using the previously setup **server address**, **username**, and **password**.

These settings are stored in the **application.properties** file which can be found in the **src/main/resources** directory of your application. Note that the <> brackets are not required!

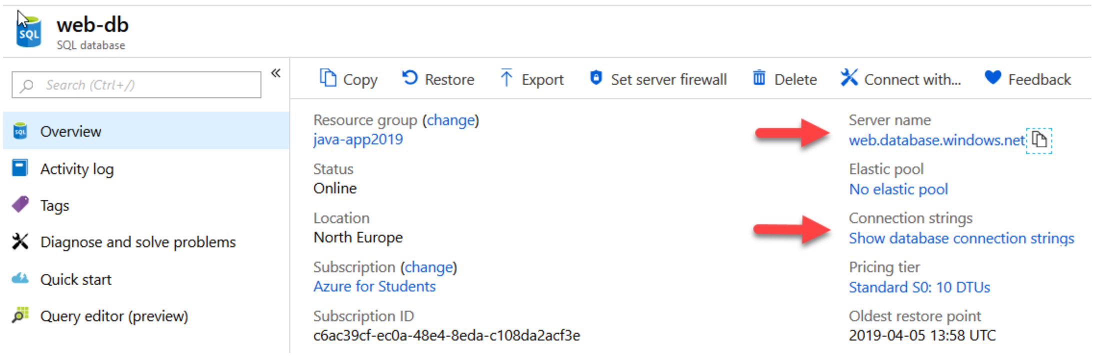


```
spring.datasource.driver-class-name=com.microsoft.sqlserver.jdbc.SQLServerDriver
spring.datasource.username=<db username>
spring.datasource.password=<password>
spring.datasource.url=jdbc:sqlserver://<db server name>.database.windows.net:1433;database=<database name>
spring.datasource.initialize=true
```

The following text can be copied:

```
spring.datasource.driver-class-name=com.microsoft.sqlserver.jdbc.SQLServerDriver
spring.datasource.username=<db username>
spring.datasource.password=<password>
spring.datasource.url=jdbc:sqlserver://<db server name>.database.windows.net:1433;database=<database name>
spring.datasource.initialize=true
```

The details can be found in the Azure control panel for the SQL database, e.g.

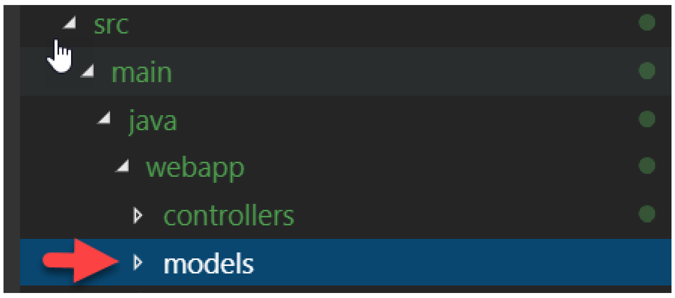


Resource group (change)
java-app2019
Status
Online
Location
North Europe
Subscription (change)
Azure for Students
Subscription ID
c6ac39cf-ec0a-48e4-8eda-c108da2acf3e
Server name
web.database.windows.net
Elastic pool
No elastic pool
Connection strings
Show database connection strings
Pricing tier
Standard S0: 10 DTUs
Oldest restore point
2019-04-05 13:58 UTC

## 3. The model classes

The two main data objects which in the application are **Category** and **Product**. These are the Java class equivalents of the database tables created previously and will be used to represent objects of each in the application. Note the public getter and setter accessor methods for each private property.

First add a folder, named **models**, to the application:



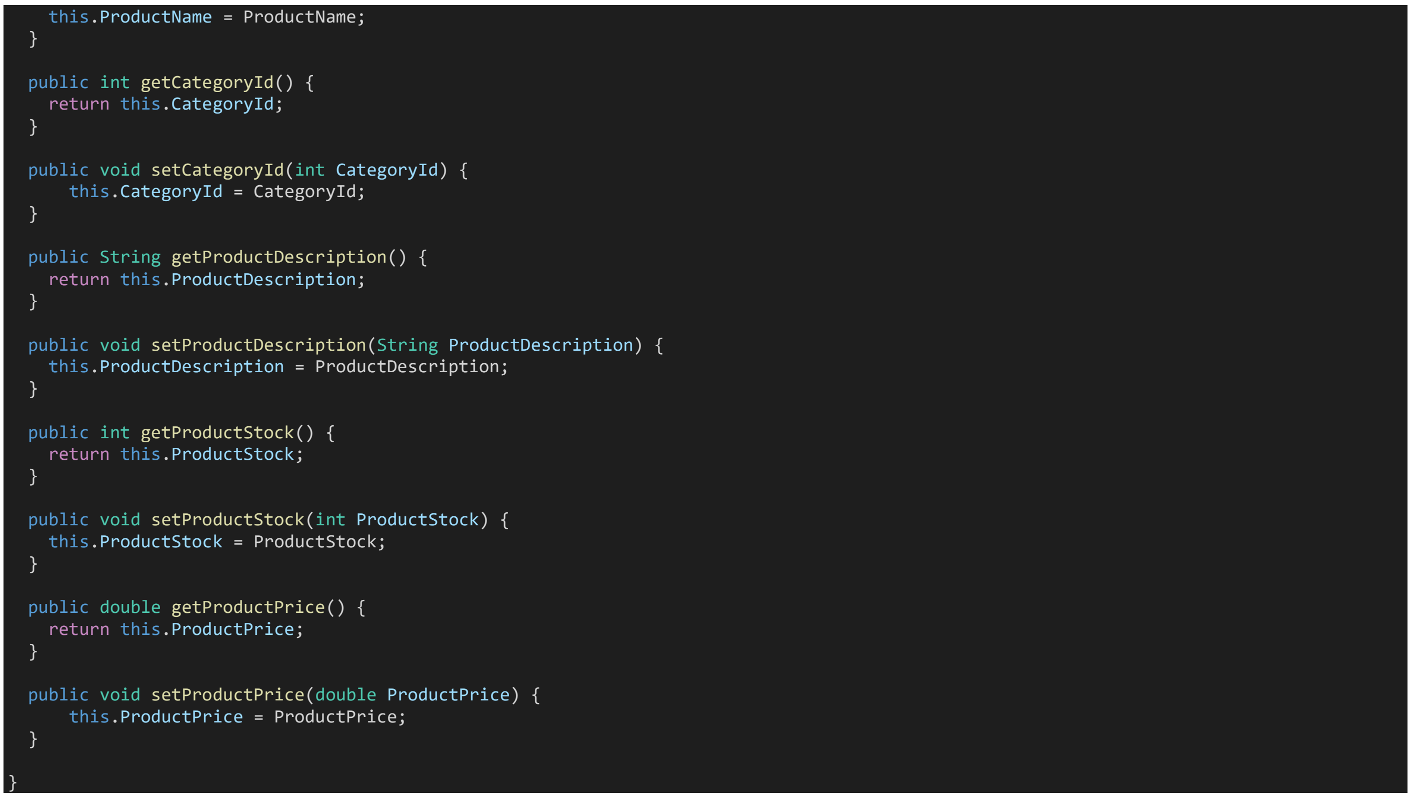
Then add two files, **Category.java** and **Product.java** to the model folder.

## Category.java

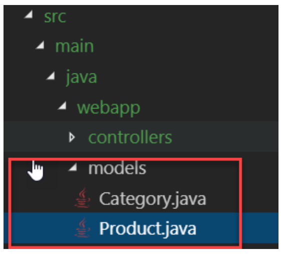
```java
package webapp.models;

public class Category {

    // Properties
    private int CategoryId;
    private String CategoryName;
    private String CategoryDescription;

    // Constructors
    public Category() {

    }

    public Category(int CategoryId, String CategoryName, String CategoryDescription) {
        this.CategoryId = CategoryId;
        this.CategoryName = CategoryName;
        this.CategoryDescription = CategoryDescription;
    }

    // Accessor Methods
    public int getCategoryId()  {
        return this.CategoryId;
    }

    public void setCategoryId(int CategoryId) {
        this.CategoryId = CategoryId;
    }

    public String getCategoryName() {
        return this.CategoryName;
    }

    public void setCategoryName(String CategoryName) {
        this.CategoryName = CategoryName;
    }

    public String getCategoryDescription()  {
        return this.CategoryDescription;
    }

    public void setCategoryDescription(String CategoryDescription) {
        this.CategoryDescription = CategoryDescription;
    }
}
```

## Product.java

```java
package webapp.models;

public class Product {

  // Properties
  private int ProductId;
  private int CategoryId;
  private String ProductName;
  private String ProductDescription;
  private int ProductStock;
  private double ProductPrice;

  // Constructors
  public Product() {
  }

  public Product(int ProductId, String ProductName, int CategoryId, String ProductDescription, int ProductStock, double ProductPrice) {
      this.ProductId = ProductId;
      this.ProductName = ProductName;
      this.CategoryId = CategoryId;
      this.ProductDescription = ProductDescription;
      this.ProductStock = ProductStock;
      this.ProductPrice = ProductPrice;
  }

  // Accessor Methods
  public int getProductId() {
    return this.ProductId;
  }

  public void setProductId(int ProductId) {
    this.ProductId = ProductId;
  }

  public String getProductName() {
    return this.ProductName;
  }

  public void setProductName(String ProductName) {
```



```
    this.ProductName = ProductName;
  }

  public int getCategoryId() {
    return this.CategoryId;
  }

  public void setCategoryId(int CategoryId) {
      this.CategoryId = CategoryId;
  }

  public String getProductDescription() {
    return this.ProductDescription;
  }

  public void setProductDescription(String ProductDescription) {
    this.ProductDescription = ProductDescription;
  }

  public int getProductStock() {
    return this.ProductStock;
  }

  public void setProductStock(int ProductStock) {
    this.ProductStock = ProductStock;
  }

  public double getProductPrice() {
    return this.ProductPrice;
  }

  public void setProductPrice(double ProductPrice) {
      this.ProductPrice = ProductPrice;
  }

}
```

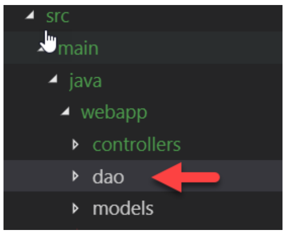


## 4. Data Access – getting the data

The next step is to add a data access layer. This will serve as interface between the database and the object model used in Java (i.e. the model classes created earlier). **`JDBC will be used here`** as it is based on the SQL already covered in the course. *In the future you may want to investigate using the Java Persistence Architecture (JPA) as an alternative as it provides automatic object relational mapping (ORM) between Database entities and Java Objects.*

We will create Data Access Objects to query the database and convert the result into objects of the model classes.

First add another new folder named **dao** to the application



Add ProductDao.java, the DAO for Product, to the **dao** folder. This is an **interface** and not a class. It defines the public face of the class and contains **abstract** properties and methods but no implementation. See https://www.geeksforgeeks.org/interfaces-in-java/

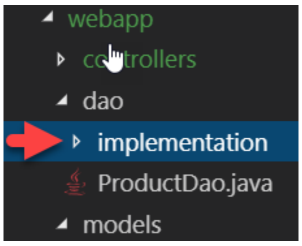
The actual implementation will be defined separately (**why?**)

**ProductDao.java**

```
package webapp.dao;

import java.util.List;
import webapp.models.Product;

// ProductDAo Interface definition
public interface ProductDao {

    // Return a list containing all the product objects
    public List<Product> findAll();

    // Return a product with matching id
    public Product findById(int id);

    // return a list of products in a category
    public List<Product> findByCategory(int id);
}
```

Next, we will implement **ProductDao**. Separating the interface and its implementation makes it easier to change how data access works without changing how it is accessed. Comparable to buying a new computer but keeping your existing keyboard and mouse!

Create a new folder, named **implementation**, inside the **dao** folder



Add a new file, **ProductDaoImpl.java**, to this folder. This file will contain a class which implements **ProductDao** using Spring JDBC.

**ProductDaoImpl.java**

```
package webapp.dao.implementation;

import java.sql.Connection;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.Statement;
import java.util.List;


import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.jdbc.core.JdbcTemplate;
import org.springframework.jdbc.core.PreparedStatementCreator;
import org.springframework.jdbc.core.RowMapper;
import org.springframework.jdbc.support.GeneratedKeyHolder;
import org.springframework.jdbc.support.KeyHolder;
import org.springframework.stereotype.Repository;

import webapp.dao.ProductDao;
import webapp.models.Product;

@Repository
public class ProductDaoImpl implements ProductDao {

    // SQL for selecting products
    private final String SELECT_SQL = "SELECT * FROM dbo.Product";
    private final String SELECT_SQL_BY_ID = "SELECT * FROM dbo.Product WHERE ProductId = ?";
    private final String SELECT_SQL_BY_CAT_ID = "SELECT * FROM dbo.Product WHERE CategoryId = ?";

    // Spring JdbcTempate helps with storing and retrieving data
    @Autowired
    private JdbcTemplate jdbcTemplate;
    // Implement findAll() which retrieves all products from the DB
    // ProductMapper() converts rows from the result into Product objects
    public List<Product> findAll() {
        return jdbcTemplate.query(SELECT_SQL, new ProductMapper());
    }

    // Return a single product matching id
    public Product findById(int id) {
        return jdbcTemplate.queryForObject(SELECT_SQL_BY_ID, new Object[] { id }, new ProductMapper());
    }

    // return products for a category
    public List<Product> findByCategory(int id) {
        return jdbcTemplate.query(SELECT_SQL_BY_CAT_ID, new Object[] { id }, new ProductMapper());
    }
}

// This class converts resultset rows (from the sql execution) into Java objects
class ProductMapper implements RowMapper<Product> {
    @Override
    public Product mapRow(ResultSet rs, int rowNum) throws SQLException {
        Product p = new Product();
        p.setProductId(rs.getInt("ProductId"));
        p.setCategoryId(rs.getInt("CategoryId"));
        p.setProductName(rs.getString("ProductName"));
        p.setProductDescription(rs.getString("ProductDescription"));
        p.setProductStock(rs.getInt("ProductStock"));
        p.setProductPrice(rs.getDouble("ProductPrice"));
        return p;
    }
}
```

## 5. Display Product data

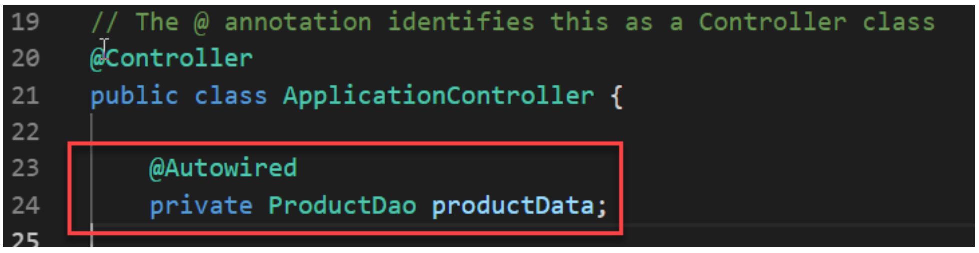
At this point we have everything required to get data from the database into Java. Now it will be displayed in a page.

Start with the controller, **`ApplicationController.java`**

1. Add the required imports at the top

```
import org.springframework.beans.factory.annotation.Autowired;
import java.util.*;

import webapp.models.Product;
import webapp.dao.ProductDao;
```

2. Declare an **`@Autowired`** instance of **`ProductDao`** – this will be used to access the required data



```
// The @ annotation identifies this as a Controller class
@Controller
public class ApplicationController {

    @Autowired
    private ProductDao productData;
```

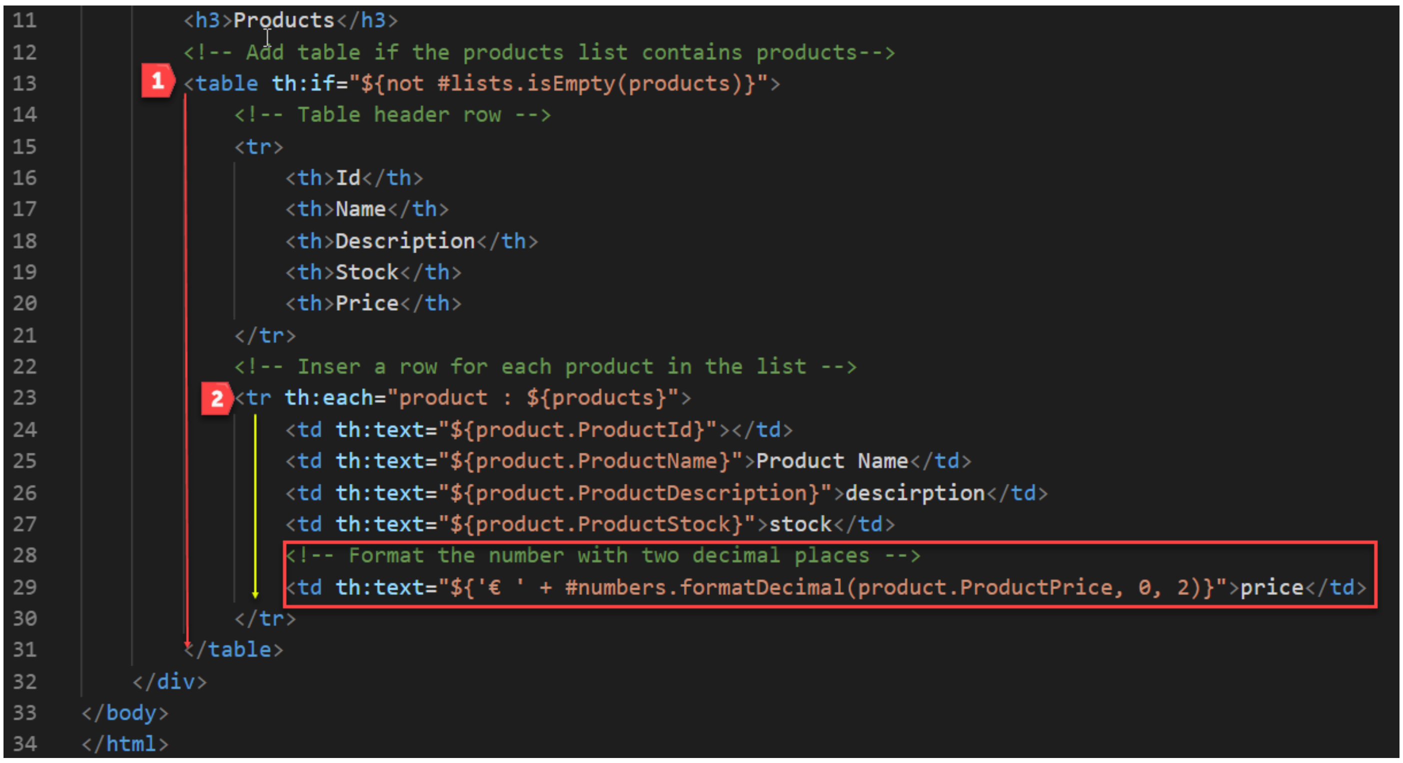
3. Add a method to handle an HTTP request for /products. Name the method **`products`** (note the code comments):

```
    // This method displays the product page
    @RequestMapping(value = "/products", method = RequestMethod.GET)
    // Uses a Model instance - which will be passed to a view
    public String products(Model model){
        // Get the product list from the ProductDao instance
        List<Product> products = productData.findAll();
        model.addAttribute("products", products);

        // Return the view
        return "products";
    }
```

4. Add a view template **products.html** to **main/resources/templates**. Use emmet to add an HTML table with two rows (one for headers). See https://www.w3schools.com/html/html_tables.asp

```html
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Products</title>
</head>
<body>
    <!--/*/ <th:block th:include="fragments/navigation :: navigation"></th:block> /*/-->
    <div>
        <h3>Products</h3>
        <table>
            <tr>
                <th>Id</th>
                <th>Name</th>
                <th>Description</th>
                <th>Stock</th>
                <th>Price</th>
            </tr>
            <tr>
                <td></td>
                <td></td>
                <td></td>
                <td></td>
                <td></td>
            </tr>
        </table>
    </div>
</body>
</html>
```

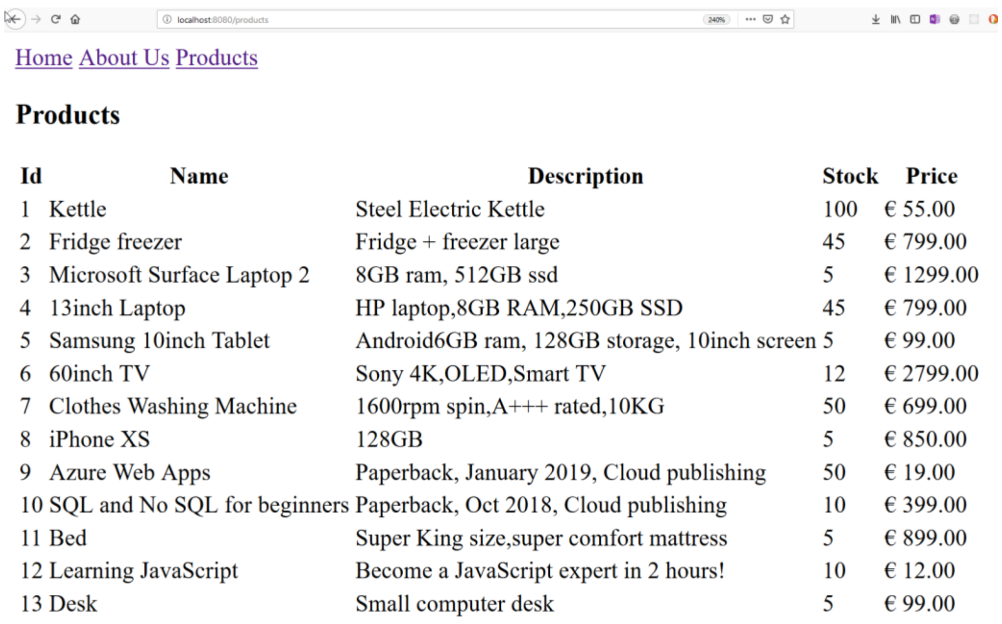
5. Use Thymeleaf template language to display products. See the documentation for a full description of functions: https://www.thymeleaf.org/documentation.html
    a. The table will only be inserted **if** there are products to display.
    b. The **the:each** loop will insert a row for each product.



```html
        <h3>Products</h3>
        <!-- Add table if the products list contains products-->
        <table th:if="${not #lists.isEmpty(products)}">
            <!-- Table header row -->
            <tr>
                <th>Id</th>
                <th>Name</th>
                <th>Description</th>
                <th>Stock</th>
                <th>Price</th>
            </tr>
            <!-- Inser a row for each product in the list -->
            <tr th:each="product : ${products}">
                <td th:text="${product.ProductId}"></td>
                <td th:text="${product.ProductName}">Product Name</td>
                <td th:text="${product.ProductDescription}">descirption</td>
                <td th:text="${product.ProductStock}">stock</td>
                <!-- Format the number with two decimal places -->
                <td th:text="${'€ ' + #numbers.formatDecimal(product.ProductPrice, 0, 2)}">price</td>
            </tr>
        </table>
    </div>
</body>
</html>
```

## 6. Run the application

Open http://localhost:8080/products in a browser. You should see something like this:



localhost:8080/products

Home About Us Products

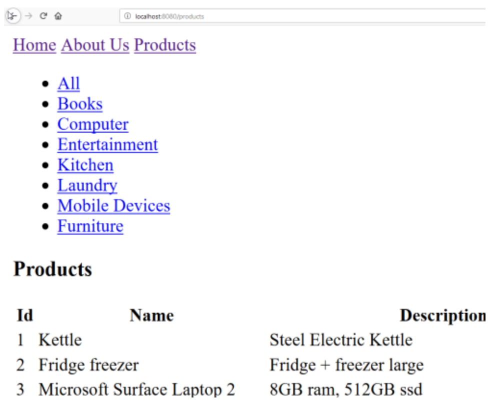
**Products**

| Id | Name | Description | Stock | Price |
|---|---|---|---|---|
| 1 | Kettle | Steel Electric Kettle | 100 | € 55.00 |
| 2 | Fridge freezer | Fridge + freezer large | 45 | € 799.00 |
| 3 | Microsoft Surface Laptop 2 | 8GB ram, 512GB ssd | 5 | € 1299.00 |
| 4 | 13inch Laptop | HP laptop,8GB RAM,250GB SSD | 45 | € 799.00 |
| 5 | Samsung 10inch Tablet | Android6GB ram, 128GB storage, 10inch screen | 5 | € 99.00 |
| 6 | 60inch TV | Sony 4K,OLED,Smart TV | 12 | € 2799.00 |
| 7 | Clothes Washing Machine | 1600rpm spin,A+++ rated,10KG | 50 | € 699.00 |
| 8 | iPhone XS | 128GB | 5 | € 850.00 |
| 9 | Azure Web Apps | Paperback, January 2019, Cloud publishing | 50 | € 19.00 |
| 10 | SQL and No SQL for beginners | Paperback, Oct 2018, Cloud publishing | 10 | € 399.00 |
| 11 | Bed | Super King size,super comfort mattress | 5 | € 899.00 |
| 12 | Learning JavaScript | Become a JavaScript expert in 2 hours! | 10 | € 12.00 |
| 13 | Desk | Small computer desk | 5 | € 99.00 |

Right click the page (in your browser). You should see the html table rows which were generated. Why are the comments repeated?

## 7. Exercises

Display a list of categories in the products page



localhost:8080/products

Home About Us Products

- All
- Books
- Computer
- Entertainment
- Kitchen
- Laundry
- Mobile Devices
- Furniture

**Products**

| Id | Name | Description |
|---|---|---|
| 1 | Kettle | Steel Electric Kettle |
| 2 | Fridge freezer | Fridge + freezer large |
| 3 | Microsoft Surface Laptop 2 | 8GB ram, 512GB ssd |

1. Add `CategoryDao` and its implementation
   a. Only `findAll()` and `findById()` methods are required.
2. Retrieve the `categories` list in the `products()` controller method and add to the model.
3. Use a `<ul>` (unordered list) element to display the categories. See https://www.w3schools.com/html/html_lists.asp
4. Make each item in the list a link.
   a. The links should call `/products` with a parameter, e.g. `/products/?cat=1`