

[Home](#)

[Java Core](#)

[Java SE](#)

[Java EE](#)

[Frameworks](#)

[Servers](#)

[Coding](#)

[IDEs](#)

[Books](#)

[Videos](#)

[Certifications](#)

[Testing](#)

[Home](#) ▶ [Frameworks](#) ▶ [Spring Boot](#)

Learn Spring framework:

- [Understand the core of Spring](#)
- [Understand Spring MVC](#)
- [Understand Spring AOP](#)
- [Understand Spring Data JPA](#)
- [Spring Dependency Injection \(XML\)](#)
- [Spring Dependency Injection \(Annotations\)](#)

- [Spring Dependency Injection \(Java config\)](#)
- [Spring MVC beginner tutorial](#)
- [Spring MVC Exception Handling](#)
- [Spring MVC and log4j](#)
- [Spring MVC Send email](#)
- [Spring MVC File Upload](#)
- [Spring MVC Form Handling](#)
- [Spring MVC Form Validation](#)
- [Spring MVC File Download](#)
- [Spring MVC JdbcTemplate](#)
- [Spring MVC CSV view](#)
- [Spring MVC Excel View](#)
- [Spring MVC PDF View](#)
- [Spring MVC XstlView](#)
- [Spring MVC + Spring Data JPA + Hibernate - CRUD](#)
- [Spring MVC Security \(XML\)](#)
- [Spring MVC Security \(Java config\)](#)
- [Spring & Hibernate Integration \(XML\)](#)
- [Spring & Hibernate Integration \(Java\)](#)

config)

- [Spring & Struts Integration \(XML\)](#)
- [Spring & Struts Integration \(Java config\)](#)
- [14 Tips for Writing Spring MVC Controller](#)

Spring Boot CRUD Example with Spring MVC – Spring Data JPA – ThymeLeaf - Hibernate - MySQL

Written by [Nam Ha Minh](#)

Last Updated on 04 August 2019 | [Print](#) [Email](#)

In this Spring Boot tutorial, you will learn develop a Java web application that manages information in a database – with standard [CRUD operations](#): Create, Retrieve, Update and Delete. We use the following technologies:

- **Spring Boot:** enables rapid application development with sensible defaults to reduce boilerplate code. Spring Boot also helps us create a standalone, executable Java web application with ease.
- **Spring MVC:** simplifies coding the controller layer. No more boilerplate code of Java Servlet classes.
- **Spring Data JPA:** simplifies coding the data access layer. No more boilerplate code of DAO classes.
- **Hibernate:** is used as an ORM framework – implementation of JPA. No more boilerplate JDBC code.
- **ThymeLeaf:** simplifies coding the view layer. No more cluttered JSP and JSTL tags.
- And **MySQL** as the database.

For project development, we use Eclipse IDE 2018-12, JDK 8, and Maven.

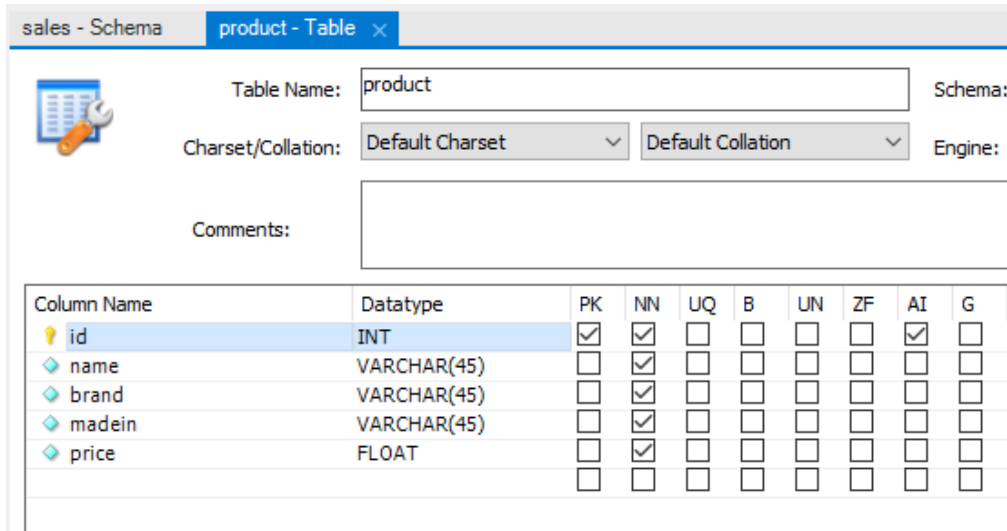
Please follow this table of content:

1. [Create MySQL Database](#)
2. [Create Maven Project in Eclipse](#)
3. [Configure Data Source Properties](#)
4. [Code Domain Model Class](#)
5. [Code Repository Interface](#)
6. [Code Service Class](#)
7. [Code Spring MVC Controller Class](#)
8. [Code Spring Boot Application Class](#)

9. Implement List Products Feature
10. Implement Create Product Feature
11. Implement Edit Product Feature
12. Implement Delete Product Feature
13. Test and package the Spring Boot CRUD Web Application

1. Create MySQL Database

Suppose that our Spring Boot web application will manage product information in this table:



The screenshot shows the MySQL Workbench interface for creating a table. The 'sales' schema is selected, and the 'product' table is being defined. The table name is 'product', and the schema is 'sales'. The charset/collation is set to 'Default Charset' and 'Default Collation'. The engine is set to 'InnoDB'. The table structure is as follows:

Column Name	Datatype	PK	NN	UQ	B	UN	ZF	AI	G
id	INT	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>
name	VARCHAR(45)	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
brand	VARCHAR(45)	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
madein	VARCHAR(45)	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
price	FLOAT	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>

You can execute the following MySQL script to create the **product** table:

```
1 CREATE TABLE `product` (  
2   `id` int(11) NOT NULL AUTO_INCREMENT,  
3   `name` varchar(45) NOT NULL,  
4   `brand` varchar(45) NOT NULL,  
5   `madein` varchar(45) NOT NULL,  
6   `price` float NOT NULL,  
7   PRIMARY KEY (`id`)  
8 ) ENGINE=InnoDB DEFAULT CHARSET=utf8;
```

The name of the database schema is **sales**.

2. Create Spring Boot Maven Project in Eclipse

In Eclipse, create a simple Maven project (skip archetype selection). Update the pom.xml file to have the following code:

```

1  <project xmlns="http://maven.apache.org/POM/4.0.0"
2      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
3      xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
4          http://maven.apache.org/xsd/maven-4.0.0.xsd">
5      <modelVersion>4.0.0</modelVersion>
6      <groupId>net.codejava</groupId>
7      <artifactId>ProductManager</artifactId>
8      <version>0.0.1-SNAPSHOT</version>
9      <packaging>jar</packaging>
10
11      <parent>
12          <groupId>org.springframework.boot</groupId>
13          <artifactId>spring-boot-starter-parent</artifactId>
14          <version>2.1.3.RELEASE</version>
15      </parent>
16
17      <dependencies>
18          <dependency>
19              <groupId>org.springframework.boot</groupId>
20              <artifactId>spring-boot-starter-web</artifactId>
21          </dependency>
22          <dependency>
23              <groupId>org.springframework.boot</groupId>
24              <artifactId>spring-boot-starter-data-jpa</artifactId>
25          </dependency>
26          <dependency>
27              <groupId>org.springframework.boot</groupId>
28              <artifactId>spring-boot-starter-thymeleaf</artifactId>
29          </dependency>
30          <dependency>
31              <groupId>mysql</groupId>
32              <artifactId>mysql-connector-java</artifactId>
33              <scope>runtime</scope>
34          </dependency>
35      </dependencies>
36
37      <build>
38          <plugins>
39              <plugin>
40                  <groupId>org.springframework.boot</groupId>
41                  <artifactId>spring-boot-maven-plugin</artifactId>
42              </plugin>
43          </plugins>
44      </build>
45  </project>

```

As you can see, with Spring Boot we have to specify only few dependencies: Spring Boot Starter Web, Spring Boot Data JPA, Spring Boot Thymeleaf and MySQL JDBC driver.

And create the main Java package **net.codejava**.

3. Configure Data Source Properties

Create the **application.properties** file under the **src/main/resources** directory with the following content:

```

1  spring.jpa.hibernate.ddl-auto=none
2  spring.datasource.url=jdbc:mysql://localhost:3306/sales
3  spring.datasource.username=root
4  spring.datasource.password=password
5  logging.level.root=WARN

```

First line tells Hibernate to make no changes to the database. And we specify the database connection properties in the next 3 lines (change the values according to your settings). And the last line we set the logging level to WARN to avoid too verbose output in the console.

4. Code Domain Model Class

Create the domain model class **Product** to map with the **product** table in the database as follows:

```
1 // copyright www.codejava.net
2 package net.codejava;
3
4 import javax.persistence.Entity;
5 import javax.persistence.GeneratedValue;
6 import javax.persistence.GenerationType;
7 import javax.persistence.Id;
8
9 @Entity
10 public class Product {
11     private Long id;
12     private String name;
13     private String brand;
14     private String madein;
15     private float price;
16
17     protected Product() {
18     }
19
20     @Id
21     @GeneratedValue(strategy = GenerationType.IDENTITY)
22     public Long getId() {
23         return id;
24     }
25
26     // other getters and setters are hidden for brevity
27 }
```

This is simple JPA entity class with the class name and field names are identical to column names of the table **product** in the database, to minimize the annotations used.

5. Code Repository Interface

Next, create the **ProductRepository** interface as simple as follows:

```
1 package net.codejava;
2
3 import org.springframework.data.jpa.repository.JpaRepository;
4
5 public interface ProductRepository extends JpaRepository<Product, Long> {
6
7 }
```

As you can see, this interface extends the **JpaRepository** interface from [Spring Data JPA](#). **JpaRepository** defines standard CRUD methods, plus JPA-specific operations. We don't

to write implementation code because Spring Data JPA will generate necessary code at runtime, in form of proxy instances.

So the purpose of writing the repository interface is to tell Spring Data JPA about the domain type (`Product`) and ID type (`Long`) to work with.

6. Code Service Class

Next, we need to code the `ProductService` class in the service/business layer with the following code:

```
1 // copyright www.codejava.net
2 package net.codejava;
3
4 import java.util.List;
5
6 import org.springframework.beans.factory.annotation.Autowired;
7 import org.springframework.stereotype.Service;
8 import org.springframework.transaction.annotation.Transactional;
9
10 @Service
11 @Transactional
12 public class ProductService {
13
14     @Autowired
15     private ProductRepository repo;
16
17     public List<Product> listAll() {
18         return repo.findAll();
19     }
20
21     public void save(Product product) {
22         repo.save(product);
23     }
24
25     public Product get(long id) {
26         return repo.findById(id).get();
27     }
28
29     public void delete(long id) {
30         repo.deleteById(id);
31     }
32 }
```

In this class, we inject an instance of `ProductRepository` via private field using `@Autowired` annotation. At runtime, Spring Data JPA will generate a proxy instance of `ProductRepository` and inject it to the instance of `ProductService` class.

You might see this service class is redundant as it delegates all the calls to `ProductRepository`. In fact, the business logic would be more complex over time, e.g. calling two or more repository instances.

So we create this class for the purpose of extensibility in future.

7. Code Spring MVC Controller Class

... create the **AppController** class acts as a **Spring MVC controller** to handle requests
... the clients – with the initial code as follows:

```
1 package net.codejava;
2
3 import org.springframework.stereotype.Controller;
4
5 @Controller
6 public class AppController {
7
8     @Autowired
9     private ProductService service;
10
11     // handler methods...
12 }
```

As you can see, we inject an instance of the **ProductService** class to this controller – Spring will automatically create one at runtime. We will write code for the handler methods when implementing each CRUD operation.

8. Code Spring Boot Application Class

Next, we create a class with **main()** method to bootstrap our Spring Boot application:

```
1 package net.codejava;
2
3 import org.springframework.boot.SpringApplication;
4 import org.springframework.boot.autoconfigure.SpringBootApplication;
5
6 @SpringBootApplication
7 public class AppMain {
8     public static void main(String[] args) {
9         SpringApplication.run(AppMain.class, args);
10    }
11 }
```

Here, the **@SpringBootApplication** annotation does all the magic stuffs such as create the web server instance and Spring MVC dispatcher servlet.

9. Implement List Products Feature

The website's home page displays a list of all products, so add the following handler method into the Spring MVC controller class:

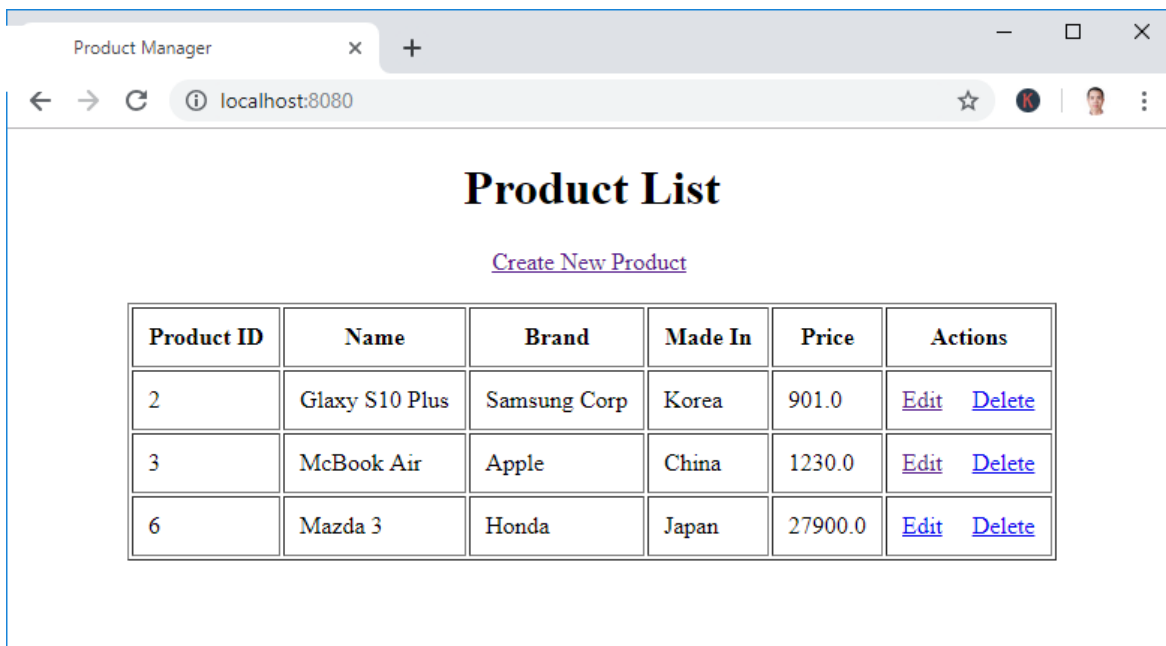
```
1 @RequestMapping("/")
2 public String viewHomePage(Model model) {
3     List<Product> listProducts = service.listAll();
4     model.addAttribute("listProducts", listProducts);
5
6     return "index";
7 }
```

We use Thymeleaf instead of JSP, so create the **templates** directory under **src/main/resources** to store template files (HTML).

[illegible]

Now we can run the `AppMain` class to test our Spring Boot web application. You should see the Spring Boot logo appears in the Console view of Eclipse:





You see, the list of products gets displayed nicely – Suppose that you inserted some rows in the **product** table before.

10. Implement Create Product Feature

You can see in the `index.html`, we have a hyperlink that allows the user to create a new product:

```
1 <a href="/new">Create New Product</a>
```

The relative URL `new` is handled by the following method in the `AppController` class:

```
1 @RequestMapping("/new")
2 public String showNewProductPage(Model model) {
3     Product product = new Product();
4     model.addAttribute("product", product);
5
6     return "new_product";
7 }
```

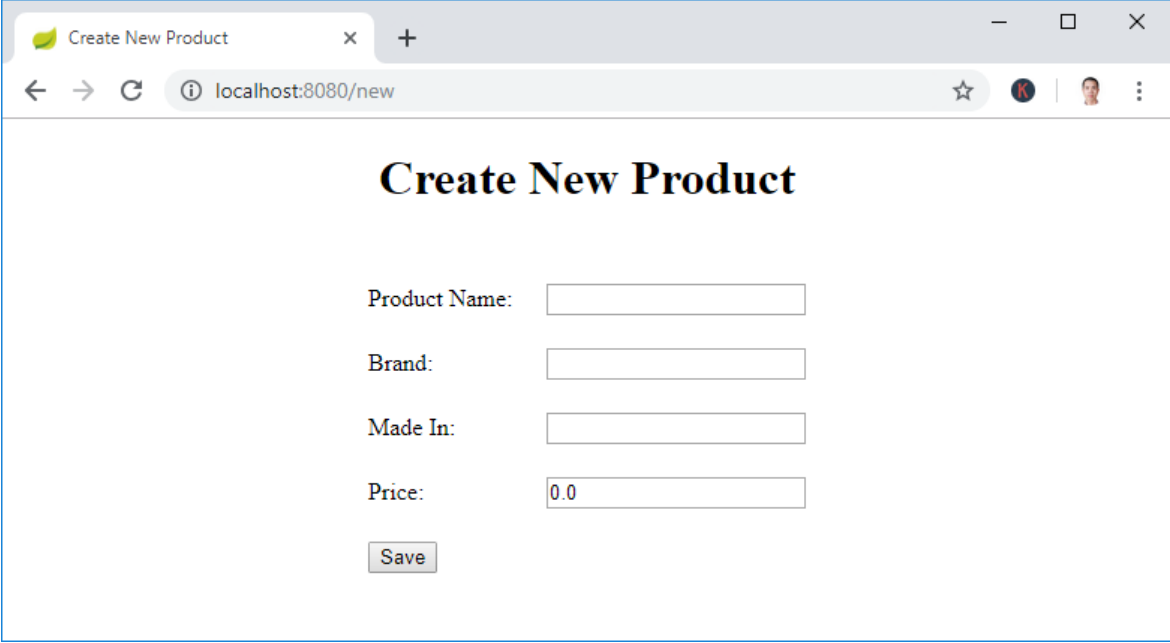
For the view, create the `new_product.html` file with the following code:

```

1 <!DOCTYPE html>
2 <html xmlns="http://www.w3.org/1999/xhtml"
3     xmlns:th="http://www.thymeleaf.org">
4 <head>
5 <meta charset="utf-8" />
6 <title>Create New Product</title>
7 </head>
8 <body>
9     <div align="center">
10         <h1>Create New Product</h1>
11         <br />
12         <form action="#" th:action="@{/save}" th:object="${product}"
13             method="post">
14
15             <table border="0" cellpadding="10">
16                 <tr>
17                     <td>Product Name:</td>
18                     <td><input type="text" th:field="*{name}" /></td>
19                 </tr>
20                 <tr>
21                     <td>Brand:</td>
22                     <td><input type="text" th:field="*{brand}" /></td>
23                 </tr>
24                 <tr>
25                     <td>Made In:</td>
26                     <td><input type="text" th:field="*{madein}" /></td>
27                 </tr>
28                 <tr>
29                     <td>Price:</td>
30                     <td><input type="text" th:field="*{price}" /></td>
31                 </tr>
32                 <tr>
33                     <td colspan="2"><button type="submit">Save</button> </td>
34                 </tr>
35             </table>
36         </form>
37     </div>
38 </body>
39 </html>

```

As you can see, here we use Thymeleaf syntax for the form instead of Spring form tags. The Create New Product page looks like this:



The screenshot shows a web browser window with the title 'Create New Product'. The address bar shows 'localhost:8080/new'. The page content is centered and features a large heading 'Create New Product'. Below the heading is a form with four input fields: 'Product Name:', 'Brand:', 'Made In:', and 'Price:'. The 'Price' field is pre-filled with '0.0'. At the bottom of the form is a 'Save' button.

we need to code another handler method to save the product information into the database:

```
1 @RequestMapping(value = "/save", method = RequestMethod.POST)
2 public String saveProduct(@ModelAttribute("product") Product product) {
3     service.save(product);
4
5     return "redirect:/";
6 }
```

After the product is inserted into the database, it redirects to the homepage to refresh the product list.

11. Implement Edit Product Feature

In the home page, you can see there's a hyperlink that allows the users to edit a product:

```
1 <a th:href="@{'/edit/' + ${product.id}}">Edit</a>
```

Code the handler method in the controller class as follows:

```
1 @RequestMapping("/edit/{id}")
2 public ModelAndView showEditProductPage(@PathVariable(name = "id") int id) {
3     ModelAndView mav = new ModelAndView("edit_product");
4     Product product = service.get(id);
5     mav.addObject("product", product);
6
7     return mav;
8 }
```

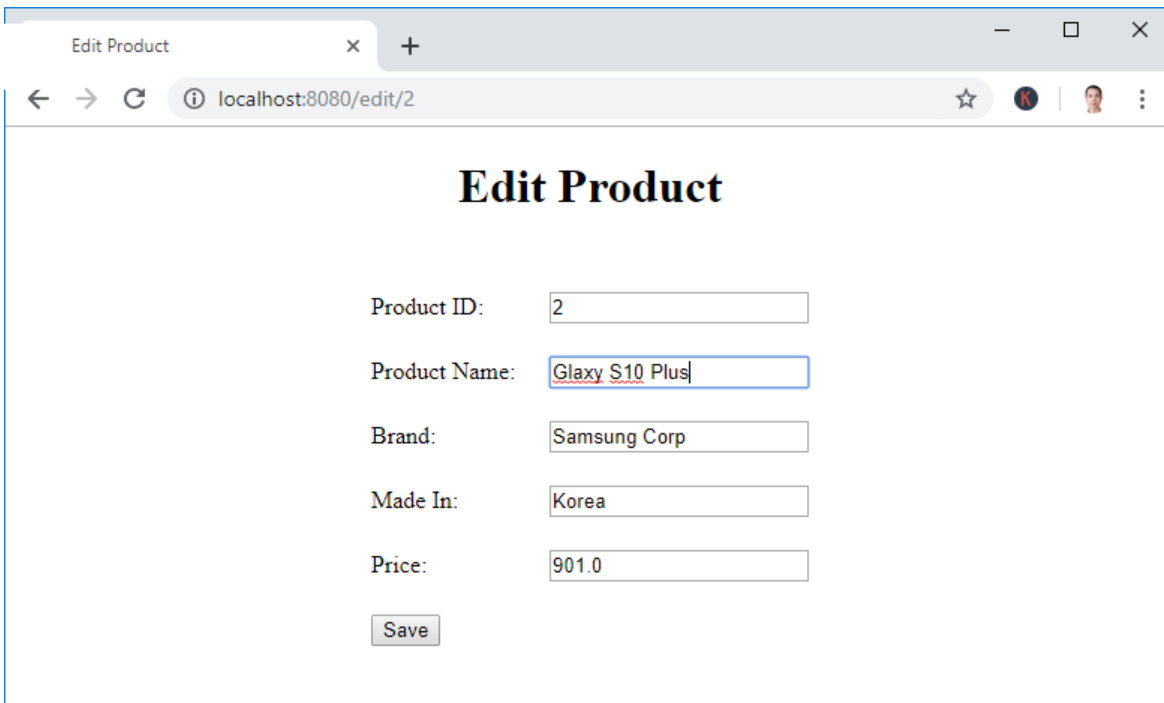
And code the view page `edit_product.html` with the following code:

```

1 <!DOCTYPE html>
2 <html xmlns="http://www.w3.org/1999/xhtml"
3     xmlns:th="http://www.thymeleaf.org">
4 <head>
5 <meta charset="utf-8" />
6 <title>Edit Product</title>
7 </head>
8 <body>
9     <div align="center">
10         <h1>Edit Product</h1>
11         <br />
12         <form action="#" th:action="@{/save}" th:object="${product}"
13             method="post">
14
15             <table border="0" cellpadding="10">
16                 <tr>
17                     <td>Product ID:</td>
18                     <td>
19                         <input type="text" th:field="*{id}" readonly="readonly" />
20                     </td>
21                 </tr>
22                 <tr>
23                     <td>Product Name:</td>
24                     <td>
25                         <input type="text" th:field="*{name}" />
26                     </td>
27                 </tr>
28                 <tr>
29                     <td>Brand:</td>
30                     <td><input type="text" th:field="*{brand}" /></td>
31                 </tr>
32                 <tr>
33                     <td>Made In:</td>
34                     <td><input type="text" th:field="*{madein}" /></td>
35                 </tr>
36                 <tr>
37                     <td>Price:</td>
38                     <td><input type="text" th:field="*{price}" /></td>
39                 </tr>
40                 <tr>
41                     <td colspan="2"><button type="submit">Save</button> </td>
42                 </tr>
43             </table>
44         </form>
45     </div>
46 </body>
47 </html>

```

The edit product page should look like this:



The screenshot shows a web browser window with the title 'Edit Product'. The address bar shows 'localhost:8080/edit/2'. The page content includes a heading 'Edit Product' and a form with the following fields:

- Product ID: 2
- Product Name: Galaxy S10 Plus
- Brand: Samsung Corp
- Made In: Korea
- Price: 901.0

At the bottom of the form is a 'Save' button.

Click the Save button will update the product information into the database. The handler method `saveProduct()` is reused in this case.

12. Implement Delete Product Feature

You can see the hyperlink to delete a product in the home page:

```
1 <a th:href="@{'/delete/' + ${product.id}}">Delete</a>
```

So code the handler method in the controller class as follows:

```
1 @RequestMapping("/delete/{id}")
2 public String deleteProduct(@PathVariable(name = "id") int id) {
3     service.delete(id);
4     return "redirect:/";
5 }
```

When the user clicks the Delete hyperlink, the corresponding product information is removed from the database, and the home page gets refreshed.

13. Test and package the Spring Boot CRUD Web Application

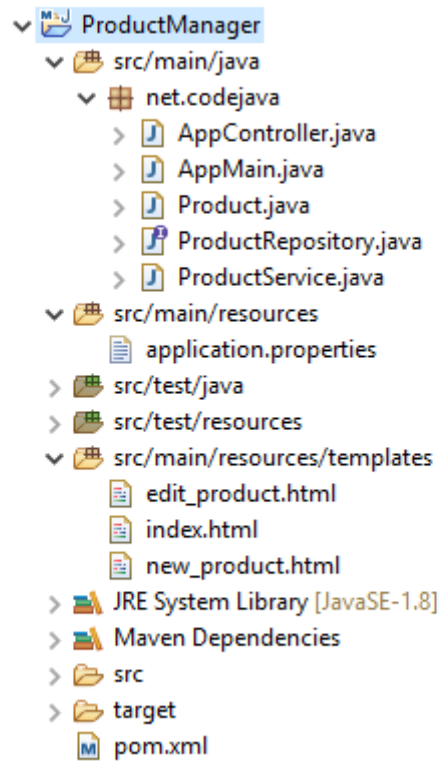
To test the Spring Boot web application we have developed in Eclipse, run the `AppMain` class as Java Application.

To package the web application as an execute JAR file in Eclipse, right-click on the project, and select **Run As > Maven build...** then enter package as the goal name, and click **Run**. If the build succeeded, you will see a JAR file is generated under the project's `target` directory, with the name like `ProductManager-0.0.1-SNAPSHOT.jar`.

.. you can use the `java` command to run this JAR file:

```
1 | java -jar ProductManager-0.0.1-SNAPSHOT.jar
```

For your reference, here's the screenshot of the project structure:



That's how to develop a Spring Boot CRUD application with Spring MVC, Spring Data JPA, ThymeLeaf, Hibernate and MySQL. You can download the sample project in the attachment section below.

Spring Boot CRUD Tutorial with Spring MVC, Spring Data JPA, ...



Other Spring Boot Tutorials:

- [Spring Boot Hello World Example](#)
- [How to create a Spring Boot Web Application \(Spring MVC with JSP/ThymeLeaf\)](#)
- [Spring Boot - Spring Data JPA - MySQL Example](#)

About the Author:



[Nam Ha Minh](#) is certified Java programmer (SCJP and SCWCD). He started programming with Java in the time of Java 1.4 and has been falling in love with Java since then. Make friend with him on [Facebook](#).

Attachments:

 [Sample Spring Boot CRUD project](#) [] 24 kB

Add comment

500 symbols left

☐ Notify me of follow-up comments



Je ne suis pas un robot

reCAPTCHA
Confidentialité - Conditions

Send

Comments

1

2

3

#11 **Premkumar** 2019-10-24 14:26

Hi,

The below snippet code is not working :

Edit

Instead I used :

Edit

Worked fro me. Can you please brief the difference for the above codes.

[Quote](#)

#10 **Rajveer Singh** 2019-10-13 11:31

Hey ! Nam Ha Minh, can you please also give a tutorial on spring maven module based project.
which is having repository module, service module, web module.

I am unable to understand spring maven web mvc project flow

[Quote](#)

#9Ravi B 2019-10-10 07:46

how to add phonenumber by throwing an error if we give a alphabets instead of numbers.

[Quote](#)

#8Sandhya 2019-09-11 05:10

for edit option, i am getting below error:
org.thymeleaf.exceptions.TemplateProcessingException: Could not parse as expression:

[Quote](#)

#7Nam 2019-08-23 16:58

Quoting Amit Sahoo:

... there is no ProductManagerApplication.java file...

It is actually the AppMain class which is described in the section 8.

[Quote](#)

[1](#) [2](#) [3](#)

[Refresh comments list](#)

About CodeJava.net:

CodeJava.net shares Java tutorials, code examples and sample projects for programmers at all levels.
CodeJava.net is created and managed by Nam Ha Minh - a passionate programmer.

[About](#) [Advertise](#) [Contact](#) [Terms of Use](#) [Privacy Policy](#) [Sitemap](#) [Newsletter](#) [Facebook](#) [Twitter](#) [YouTube](#)

Copyright © 2012 - 2019 CodeJava.net, all rights reserved.