

# Spring Boot Microservices: Building RESTful API Services

## Contents

Course Overview .....	3
Performing Read Operations .....	4
1. Create Spring Boot Project Application .....	4
2. Generated Spring Boot Entry Point.....	5
3. Prepare Model Object .....	6
4. Prepare Service Class .....	7
5. Prepare Controller Class.....	8
6. Run and Test Application.....	9
7. Make new feature / functionality by updating the code on Service and Controller .....	11
Performing Create Operations.....	13
1. Update the Service Class .....	13
2. Update the Controller Class.....	14
3. Run and Test the Application.....	15
Performing Update Operations.....	17
1. Update the Service Class .....	17
2. Update the Controller Class.....	18
3. Run and Test .....	19
Performing Delete Operations .....	21
1. Update the Server Class .....	21
2. Update the Controller Class.....	22
3. Run and Test .....	23
Setup MySQL Database on Docker .....	25
Integrating With MySQL .....	26
1. Create Database.....	26
2. Create Maven Project.....	27
3. Preparing Model Object .....	29
4. Prepare Repository Interface.....	31
5. Prepare Service Class .....	32
6. Prepare Custom Exception Controller Class .....	33
7. The Controller Class .....	34
8. Entry Point Spring Boot Application Class .....	35

# Spring Boot Microservices: Building RESTful API Services

9. Update database info config properties.....	36
10. Run and Test .....	37
Performing CRUD operation via Web UI .....	45
1. Update Dependency in pom.xml .....	45
2. Create Web Controller Class.....	46
3. Create View HTML Page .....	47
5. Create Additional View HTML Page .....	50
6. Updating Records Using a Web UI .....	55
7. Deleting Records Using a Web UI .....	60
Caching Using @Cacheable.....	65
1. Update Dependency in pom.xml .....	65
2. Enable Caching on Entry Point Spring Boot Application .....	66
3. Enable Caching on Service Layer.....	67
Clearing Caches Using @CacheEvict.....	74
Summary.....	83
Quiz.....	84

# Spring Boot Microservices: Building RESTful API Services

## Course Overview

Spring Boot is an opinionated framework, which makes building applications easy by providing sensible defaults for most of the libraries that you need. Yet, allowing you the flexibility of configuring specific classes. In this course, we will study how you can build a RESTful API service to perform, create, read, update, and delete operations. We'll use the advance REST client to test our API service. We'll then wire up a MySQL database using JPA spring data and Hibernate. And add a front end to our application using the Thymeleaf template engine.

Once you're done with this course, you will be able to build RESTful microservices in Spring Boot. Integrate with a relational database to store your data. Set up a front end for your application. And add caching for common requests.

# Spring Boot Microservices: Building RESTful API Services

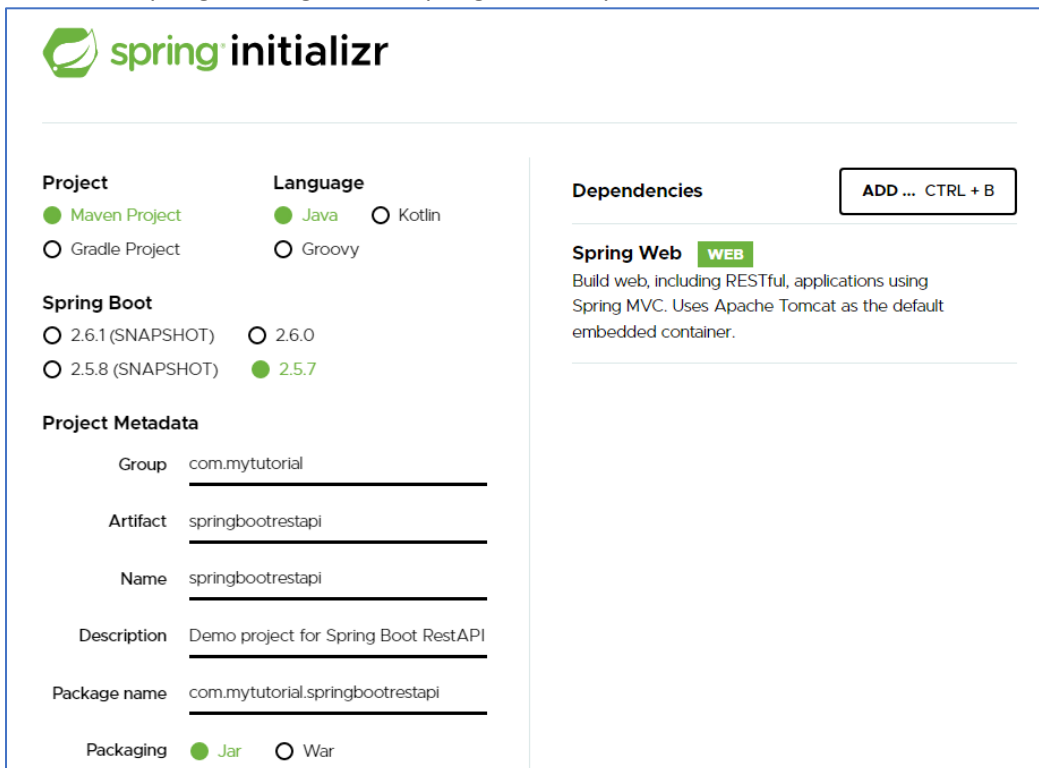
## Performing Read Operations

We'll be iteratively adding to the RESTful API that we build using Spring Boot. We'll first set up APIs for the CRUD operations, create, read, update, and delete. We'll then wire up our web application to a MySQL database. We'll use JPA and Hibernate to work with that database.

Then we'll add a few web pages to our application in order to help us perform these CRUD operations. And finally, we'll see how we can implement caching in our web app. We'll add the starter dependencies to our pom.xml as we need them.

### 1. Create Spring Boot Project Application

Go to [start.spring.io](https://start.spring.io) and generate spring boot template



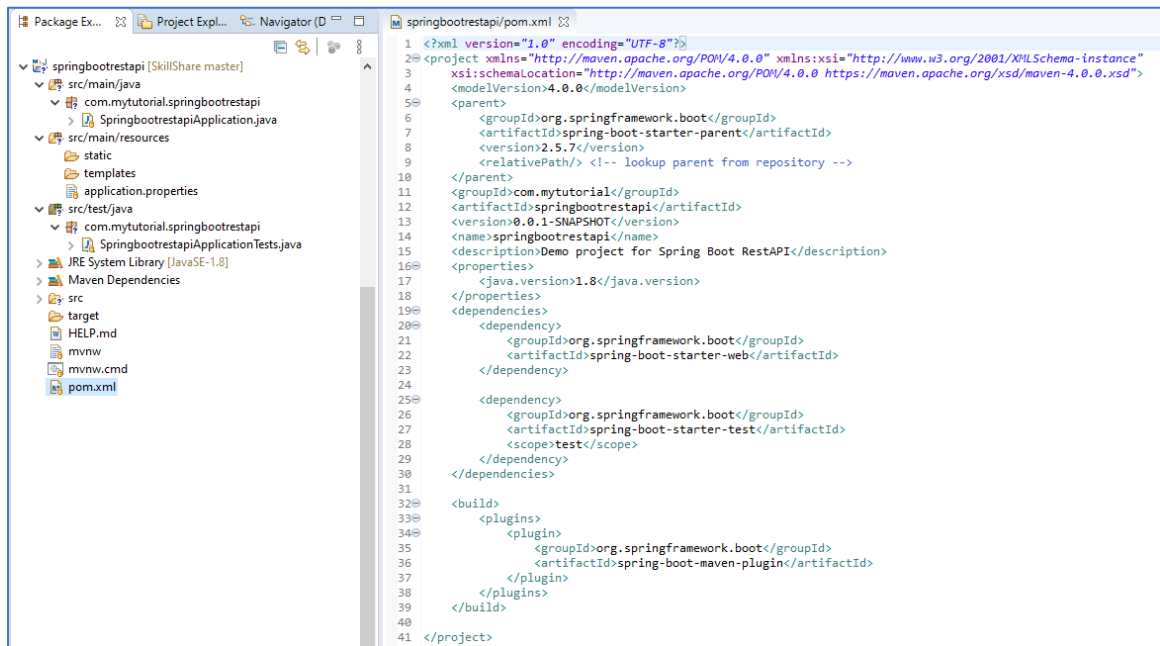
The screenshot shows the Spring Initializr web application interface. At the top left is the "spring initializr" logo. The interface is divided into several sections:

- Project:** Includes radio buttons for "Maven Project" (selected) and "Gradle Project".
- Language:** Includes radio buttons for "Java" (selected) and "Kotlin", and "Groovy".
- Spring Boot:** Includes radio buttons for "2.6.1 (SNAPSHOT)", "2.6.0", "2.5.8 (SNAPSHOT)", and "2.5.7" (selected).
- Project Metadata:** Includes input fields for "Group" (com.mytutorial), "Artifact" (springbootrestapi), "Name" (springbootrestapi), "Description" (Demo project for Spring Boot RestAPI), and "Package name" (com.mytutorial.springbootrestapi). There is also a "Packaging" section with radio buttons for "Jar" (selected) and "War".
- Dependencies:** Includes a button "ADD ... CTRL + B" and a section for "Spring Web" with a "WEB" tag. The description for "Spring Web" is: "Build web, including RESTful, applications using Spring MVC. Uses Apache Tomcat as the default embedded container."

We start off with just the Spring Boot starter web. Click on the **GENERATE** button and the project template with all of its structure will be downloaded as a Zip file onto your local machine.

# Spring Boot Microservices: Building RESTful API Services

Extract the generated template project zip file and import it into Eclipse, here is the generated **pom.xml** :



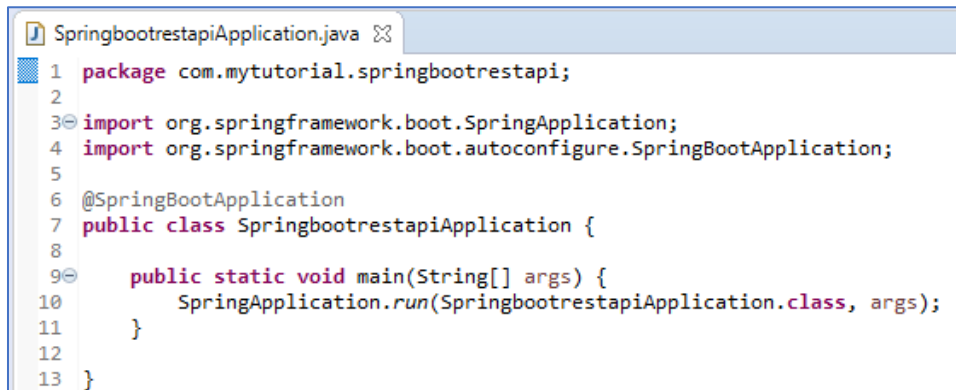
The screenshot shows the Eclipse IDE with a project named 'springbootrestapi'. The left sidebar displays the project structure, including 'src/main/java' with 'com.mytutorial.springbootrestapi' and 'SpringbootrestapiApplication.java', 'src/main/resources' with 'static', 'templates', and 'application.properties', and 'src/test/java' with 'com.mytutorial.springbootrestapi' and 'SpringbootrestapiApplicationTests.java'. The right pane shows the 'pom.xml' file with the following content:

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
3   xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 https://maven.apache.org/xsd/maven-4.0.0.xsd">
4   <modelVersion>4.0.0</modelVersion>
5   <parent>
6     <groupId>org.springframework.boot</groupId>
7     <artifactId>spring-boot-starter-parent</artifactId>
8     <version>2.5.7</version>
9     <relativePath/> <!-- lookup parent from repository -->
10  </parent>
11  <groupId>com.mytutorial</groupId>
12  <artifactId>springbootrestapi</artifactId>
13  <version>0.0.1-SNAPSHOT</version>
14  <name>springbootrestapi</name>
15  <description>Demo project for Spring Boot RestAPI</description>
16  <properties>
17    <java.version>1.8</java.version>
18  </properties>
19  <dependencies>
20    <dependency>
21      <groupId>org.springframework.boot</groupId>
22      <artifactId>spring-boot-starter-web</artifactId>
23    </dependency>
24
25    <dependency>
26      <groupId>org.springframework.boot</groupId>
27      <artifactId>spring-boot-starter-test</artifactId>
28      <scope>test</scope>
29    </dependency>
30  </dependencies>
31
32  <build>
33    <plugins>
34      <plugin>
35        <groupId>org.springframework.boot</groupId>
36        <artifactId>spring-boot-maven-plugin</artifactId>
37      </plugin>
38    </plugins>
39  </build>
40
41 </project>
```

## 2. Generated Spring Boot Entry Point

Our main entry point is very simple, the Spring Boot application annotated using **@SpringBootApplication**.

### SpringbootrestapiApplication.java



The screenshot shows the 'SpringbootrestapiApplication.java' file in the Eclipse IDE. The code is as follows:

```
1 package com.mytutorial.springbootrestapi;
2
3 import org.springframework.boot.SpringApplication;
4 import org.springframework.boot.autoconfigure.SpringBootApplication;
5
6 @SpringBootApplication
7 public class SpringbootrestapiApplication {
8
9     public static void main(String[] args) {
10         SpringApplication.run(SpringbootrestapiApplication.class, args);
11     }
12
13 }
```

# Spring Boot Microservices: Building RESTful API Services

## 3. Prepare Model Object

Let's take a look at Product.java. This is the object that represents our model.

```
Product.java
1 package com.mytutorial.springbootrestapi.model;
2
3 public class Product {
4
5     private String id;
6     private String name;
7     private String category;
8
9     public Product() {
10    }
11
12    public Product(String id, String name, String category) {
13        this.id = id;
14        this.name = name;
15        this.category = category;
16    }
17
18    public String getId() {
19        return id;
20    }
21
22    public void setId(String id) {
23        this.id = id;
24    }
25
26    public String getName() {
27        return name;
28    }
29
30    public void setName(String name) {
31        this.name = name;
32    }
33
34    public String getCategory() {
35        return category;
36    }
37
38    public void setCategory(String category) {
39        this.category = category;
40    }
41
42 }
```

The Product class is a plain old Java object. Every product has an id of type string, a name and a category.

We have two constructors for this class, one on line 9 and one on line 12. One is a default no argument constructor and another takes in input arguments.

The remaining code in this class are simply getters and setters for all of the member variables of this model object.

# Spring Boot Microservices: Building RESTful API Services

## 4. Prepare Service Class

I'm going to add in a service class called ProductService. It's tagged using the **@Service** annotation.

### ProductService.java

```
ProductService.java
1 package com.mytutorial.springbootrestapi.service;
2
3 import java.util.Arrays;
4 import java.util.List;
5
6 import org.springframework.stereotype.Service;
7
8 import com.mytutorial.springbootrestapi.model.Product;
9
10 @Service
11 public class ProductService {
12
13     private List<Product> products = Arrays.asList(
14         new Product("P101", "Monitor", "Electronics"),
15         new Product("P102", "Blanket", "Household"),
16         new Product("P103", "Laptop", "Electronics"),
17         new Product("P104", "Shirt", "Fashion"),
18         new Product("P105", "Pens", "School"));
19
20     public List<Product> getAllProducts() {
21         return products;
22     }
23
24 }
```

The service layer in any web application is what contains the business logic of our code. The **@Service** annotation basically means that this is a spring managed component. It also declares intent, indicating that this is a class that contains code with some business logic.

We are not retrieving these products from a database yet. So I'm going to hard code a list of products in this ProductService class and when getAllProducts is invoked, we return this hard coded list of products. Every product has an ID, a name and a category.

# Spring Boot Microservices: Building RESTful API Services

## 5. Prepare Controller Class

In the very first iteration of this web service, we'll set up a handler mapping for a get request for a read operation. Here is our `ProductController`, annotated as **@RestController** indicating that the responses from the handler methods should be treated as response bodies, or web responses rendered to the user.

### ProductController.java

```
ProductController.java
1 package com.mytutorial.springbootrestapi.controller;
2
3 import java.util.List;
4
5 import org.springframework.beans.factory.annotation.Autowired;
6 import org.springframework.web.bind.annotation.GetMapping;
7 import org.springframework.web.bind.annotation.RestController;
8
9 import com.mytutorial.springbootrestapi.model.Product;
10 import com.mytutorial.springbootrestapi.service.ProductService;
11
12 @RestController
13 public class ProductController {
14
15     @Autowired
16     private ProductService productService;
17
18     @GetMapping("/products")
19     public List<Product> getAllProducts() {
20         return productService.getAllProducts();
21     }
22 }
```

We have a `getAllProducts` method here, which returns a list of product objects. It's annotated using `@GetMapping("/products")`.

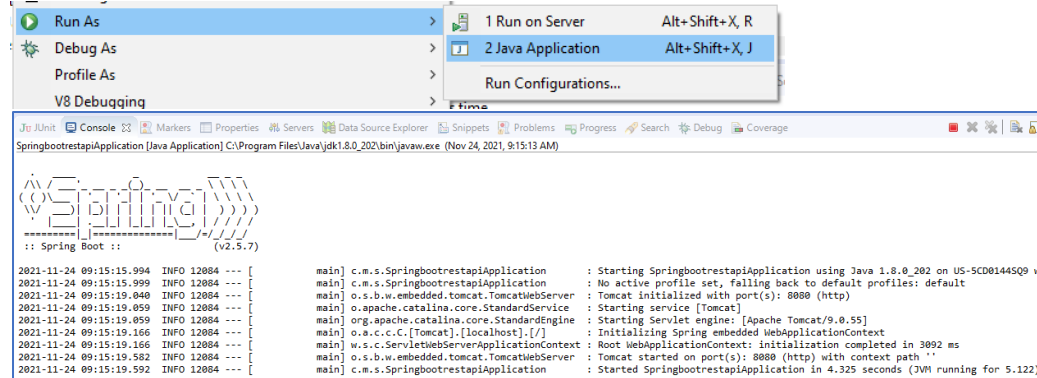
The `ProductController` simply calls the `ProductService` to get the list of products. A reference to the `ProductService` will be injected into this class because of the **Autowired** annotation that we have on the `ProductService` member variable.



# Spring Boot Microservices: Building RESTful API Services

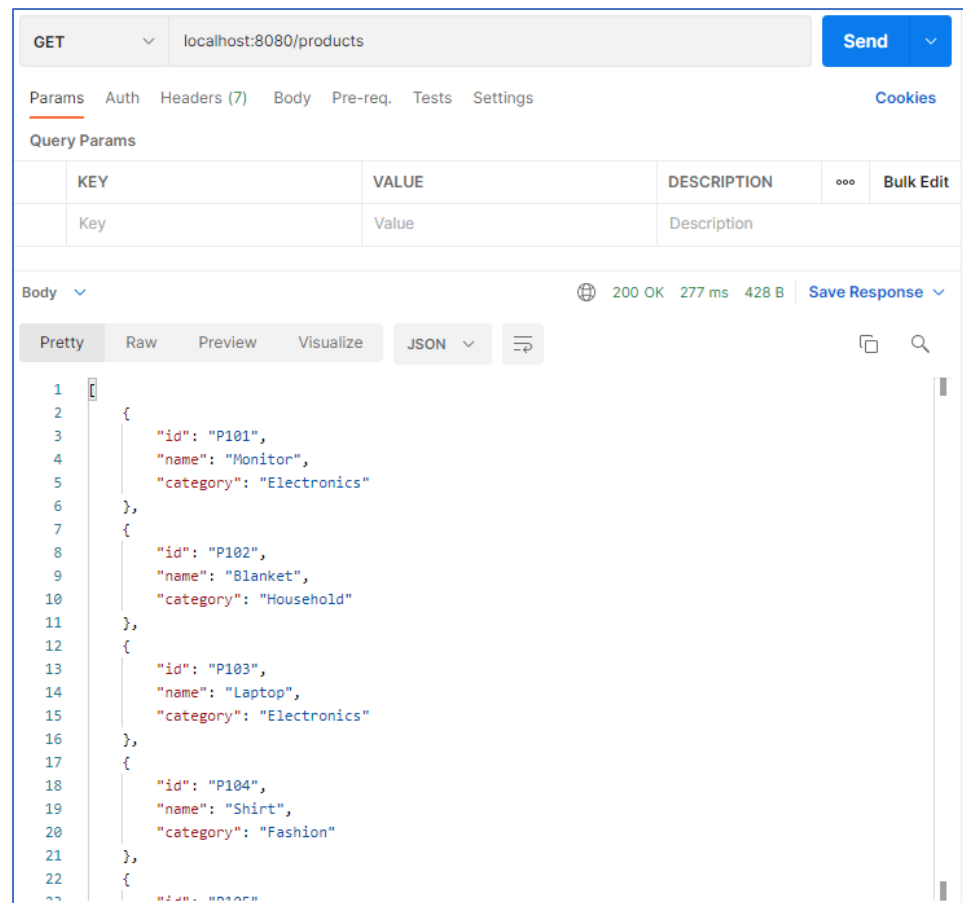
## 6. Run and Test Application

Let's check that the simple read operation works. I'm going to run this code  
Right Click on “SpringbootrestapiApplication.java” and Select “Run As > Java Application”



and then switch over to the Postman client.

Method I set to GET, the Request URL will be localhost:8080/products, click on the SEND button. And the response from our web application in the form of JSON will be available here right within our REST client. You can see the nicely formatted JSON response that we get from the server.



# Spring Boot Microservices: Building RESTful API Services

Select “Headers” from output dropdown and the status returned by this response was 200 OK.

The screenshot shows a REST client interface with the following details:

- Method: GET
- URL: localhost:8080/products
- Buttons: Send, Cookies
- Tabs: Params, Auth, Headers (7), Body, Pre-req, Tests, Settings
- Query Params table:

KEY	VALUE	DESCRIPTION	...	Bulk Edit
Key	Value	Description		
- Headers table:

KEY	VALUE
Content-Type ①	application/json
Transfer-Encoding ①	chunked
Date ①	Wed, 24 Nov 2021 01:16:18 GMT
Keep-Alive ①	timeout=60
Connection ①	keep-alive
- Response status: 200 OK, 277 ms, 428 B
- Buttons: Save Response

Go back and select “Body” output from dropdown menu and select Raw tab

The screenshot shows the REST client interface with the Body tab selected and the Raw sub-tab active. The response status is 200 OK, 19 ms, 428 B.

Body Cookies Headers (5) Test Results

200 OK 19 ms 428 B Save Response

Pretty Raw Preview Visualize

```
[{"id": "P101", "name": "Monitor", "category": "Electronics"}, {"id": "P102", "name": "Blanket", "category": "Household"}, {"id": "P103", "name": "Laptop", "category": "Electronics"}, {"id": "P104", "name": "Shirt", "category": "Fashion"}, {"id": "P105", "name": "Pens", "category": "School"}]
```

Which gives us the raw unformatted JSON. This is the JSON response that was returned from our RESTful API.

# Spring Boot Microservices: Building RESTful API Services

## 7. Make new feature / functionality by updating the code on Service and Controller

Let's add one more method to the productService. This method called *getProduct* will retrieve a single product with a specified id.

### ProductService.java

```
ProductService.java
1 package com.mytutorial.springbootrestapi.service;
2
3 import java.util.Arrays;
4 import java.util.List;
5
6 import org.springframework.stereotype.Service;
7
8 import com.mytutorial.springbootrestapi.model.Product;
9
10 @Service
11 public class ProductService {
12
13     private List<Product> products = Arrays.asList(
14         new Product("P101", "Monitor", "Electronics"),
15         new Product("P102", "Blanket", "Household"),
16         new Product("P103", "Laptop", "Electronics"),
17         new Product("P104", "Shirt", "Fashion"),
18         new Product("P105", "Pens", "School"));
19
20     public List<Product> getAllProducts() {
21         return products;
22     }
23
24     public Product getProduct(String id) {
25         return products.stream().filter(p -> p.getId().equals(id)).findFirst().get();
26     }
27 }
```

We have a hard coded list of products. We invoke the stream function on this list, convert the list of products to a stream and perform a filter operation. And we will retrieve just that product with the ID that we have specified as an input argument.

We'll add a handler mapping to the ProductController class as well and this handler mapping uses a path variable.

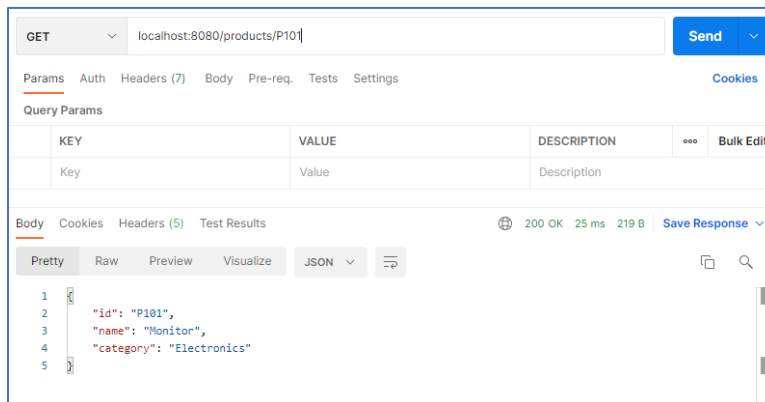
### ProductController.java

```
ProductController.java
1 package com.mytutorial.springbootrestapi.controller;
2
3 import java.util.List;
4
5 import org.springframework.beans.factory.annotation.Autowired;
6 import org.springframework.web.bind.annotation.GetMapping;
7 import org.springframework.web.bind.annotation.PathVariable;
8 import org.springframework.web.bind.annotation.RestController;
9
10 import com.mytutorial.springbootrestapi.model.Product;
11 import com.mytutorial.springbootrestapi.service.ProductService;
12
13 @RestController
14 public class ProductController {
15
16     @Autowired
17     private ProductService productService;
18
19     @GetMapping("/products")
20     public List<Product> getAllProducts() {
21         return productService.getAllProducts();
22     }
23
24     @GetMapping("/products/{pId}")
25     public Product getProduct(@PathVariable("pId") String id) {
26         return productService.getProduct(id);
27     }
28 }
```

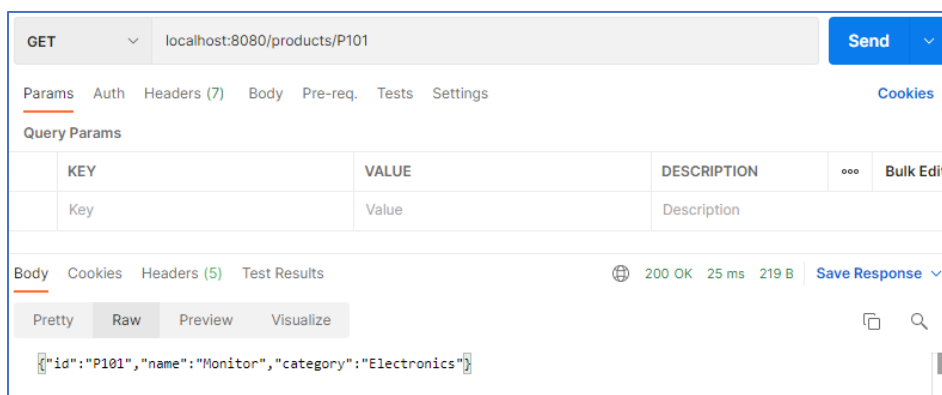
# Spring Boot Microservices: Building RESTful API Services

We have a `getProduct` method. It's annotated using the `@GetMapping` annotation and the path is `("/products/{pId}")`. `pId` is the dynamic portion of this URL that we inject into the method using the `@PathVariable` annotation. The ID of the product that we want to retrieve will be a path element that will be injected into our `getProduct` method. And then we use the `productService` to retrieve the product with this ID.

Having made all of these changes, we can now head over to the advanced REST client after running our app. Let's retrieve a single product. Notice the path to which I make the request, `/products/P101`. This will retrieve the product with ID 101. Hit SEND, there you can see the Pens product, with the category school has been retrieved.



Let us take a look at the source view, which gives us the raw unformatted JSON. This is the JSON response that was returned from our RESTful API. If you want a slightly better format, you can select the WRAP TEXT option, which will wrap the JSON representation of the list of products that we receive from the server.



You can take a look at the source view and view the same information in the JSON format.

# Spring Boot Microservices: Building RESTful API Services

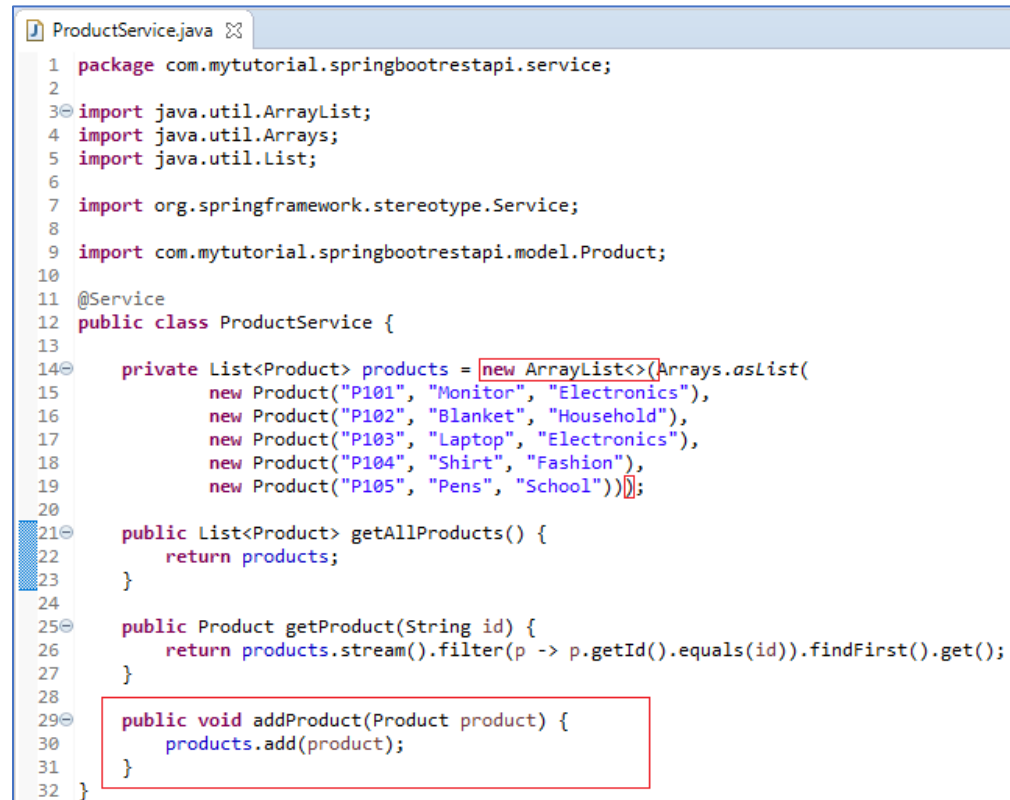
## Performing Create Operations

In this demo, we'll further enhance our RESTful APIs. We'll add the ability to add a new product by sending a POST request to our API service.

### 1. Update the Service Class

The change that we'll make first is in the ProductService class.

#### ProductService.java



```
1 package com.mytutorial.springbootrestapi.service;
2
3 import java.util.ArrayList;
4 import java.util.Arrays;
5 import java.util.List;
6
7 import org.springframework.stereotype.Service;
8
9 import com.mytutorial.springbootrestapi.model.Product;
10
11 @Service
12 public class ProductService {
13
14     private List<Product> products = new ArrayList<>(Arrays.asList(
15         new Product("P101", "Monitor", "Electronics"),
16         new Product("P102", "Blanket", "Household"),
17         new Product("P103", "Laptop", "Electronics"),
18         new Product("P104", "Shirt", "Fashion"),
19         new Product("P105", "Pens", "School")));
20
21     public List<Product> getAllProducts() {
22         return products;
23     }
24
25     public Product getProduct(String id) {
26         return products.stream().filter(p -> p.getId().equals(id)).findFirst().get();
27     }
28
29     public void addProduct(Product product) {
30         products.add(product);
31     }
32 }
```

We'll continue working with our in memory list of products. We haven't connected to a database yet. Notice that I've instantiated list of products on line 14 as an **new ArrayList()**, which is a mutable list. A mutable list will allow me to add or remove products from this list as I choose.

I've also added a method here called *addProduct*, which takes a single input argument, the product to be added to our list of products.

# Spring Boot Microservices: Building RESTful API Services

## 2. Update the Controller Class

We'll need to expose this ability to add a new product instance via a product controller handler mapping. Notice at the very bottom, we have an `addProduct` method which is annotated using an `@PostMapping` to `("/products")`.

### ProductController.java

```
ProductController.java
1 package com.mytutorial.springbootrestapi.controller;
2
3 import java.util.List;
4
5 import org.springframework.beans.factory.annotation.Autowired;
6 import org.springframework.web.bind.annotation.GetMapping;
7 import org.springframework.web.bind.annotation.PathVariable;
8 import org.springframework.web.bind.annotation.PostMapping;
9 import org.springframework.web.bind.annotation.RequestBody;
10 import org.springframework.web.bind.annotation.RestController;
11
12 import com.mytutorial.springbootrestapi.model.Product;
13 import com.mytutorial.springbootrestapi.service.ProductService;
14
15 @RestController
16 public class ProductController {
17
18     @Autowired
19     private ProductService productService;
20
21     @GetMapping("/products")
22     public List<Product> getAllProducts() {
23         return productService.getAllProducts();
24     }
25
26     @GetMapping("/products/{pId}")
27     public Product getProduct(@PathVariable("pId") String id) {
28         return productService.getProduct(id);
29     }
30
31     @PostMapping("/products")
32     public void addProduct(@RequestBody Product product) {
33         productService.addProduct(product);
34     }
35 }
```

This method will handle incoming POST requests made to this `/products` path. The input argument to this controller method is a product instance that we want added to our list of products.

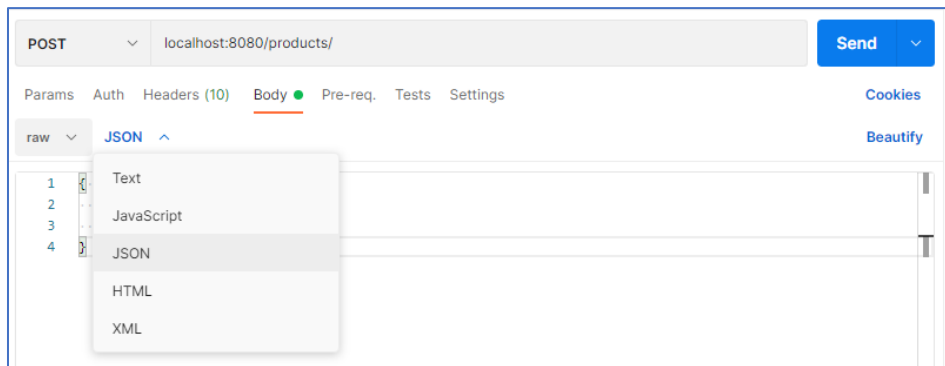
We have tagged this input argument using `@RequestBody`. This means that the body of the incoming web request will contain the details to instantiate a new product object. Spring will ensure that the request body is converted to a product instance. And this product we'll add to our list of products by invoking the `productService.addProduct` method.

# Spring Boot Microservices: Building RESTful API Services

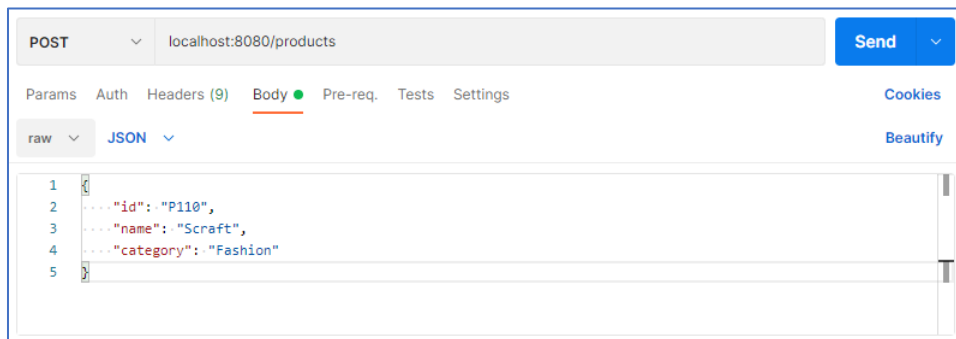
## 3. Run and Test the Application

Time to see how this works, go ahead and run this code and set up our server. Back to our *Postman* client, this time I'm first going to make a *POST* request to add a new product instance. Select the POST option and then make a request to `http://localhost:8080/products`.

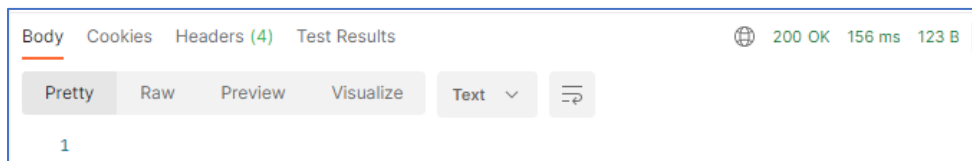
Now, in order to specify additional parameters with this request, you can expand the request option. And down here at the bottom, you'll see the option to specify a request body for your web request.



Your request body can be specified in a number of different formats. We'll select the raw with application/json format. This is an easy one to use. And this is one that is handled by Spring automatically. Specify the request body, which is basically the details of a single product. You can use FORMAT JSON so your JSON is in a pretty format.



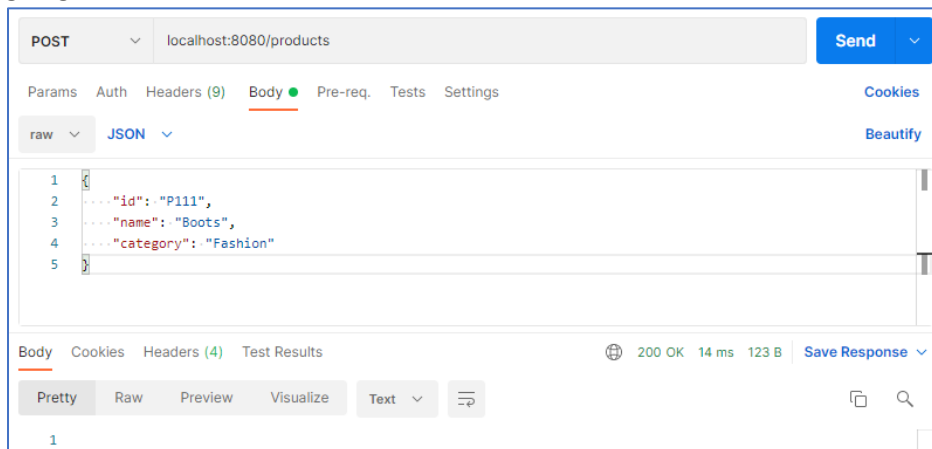
The product we're going to add has id *P110*, it's a Scarf in the fashion category. I'm now going to go up and collapse this expanded portion of my Request URL and then hit the *SEND* button. This will make a POST request to our server.



And you can see from the **200 OK** at the bottom, our new product has been successfully added.

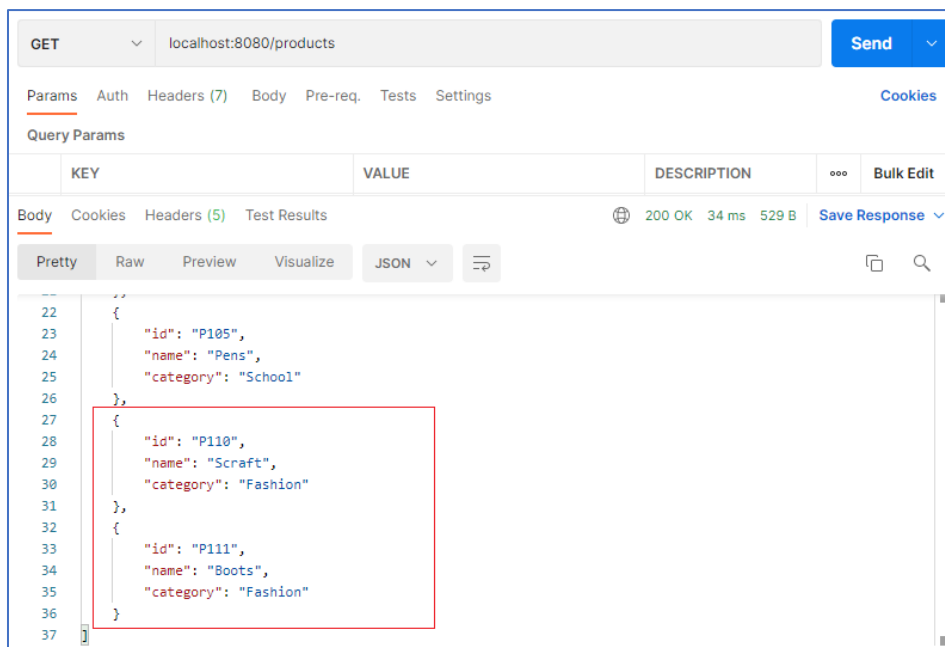
# Spring Boot Microservices: Building RESTful API Services

Let's add one more product before we test what products exist in our web application. Here is a new product with id P111. This is the Boots product in the fashion category. Once again, I'm going to FORMAT JSON and then hit the SEND button.



I get a **200 OK** in the response, indicating this product has been added successfully as well. How do we know this?

Let's make a GET request to all of the products that exist in our web service. Now you have to get rid of this body content here. That's not part of our GET request to `/products`. Hit the SEND button.



Let's take a look at the response here. And the very bottom you can see that Scarf and Boots, which are part of the fashion category, are now available in our list of products.



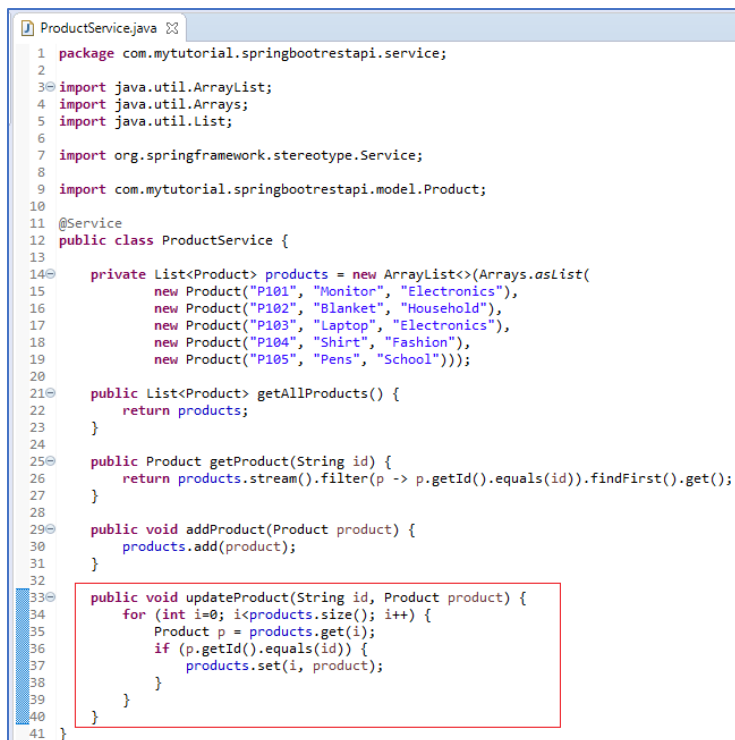
# Spring Boot Microservices: Building RESTful API Services

## Performing Update Operations

We'll now add one more operation to the APIs that our web service provides, the ability to update the details of a particular product. The first change that we'll make will be to the ProductService class. I have a method here called updateProduct which takes in two input arguments. The id of the product and a new instance of the product with the updated details.

### 1. Update the Service Class

In order to update this product, we'll run a for loop starting at 0, going up to the number of products that we have.



```
1 package com.mytutorial.springbootrestapi.service;
2
3 import java.util.ArrayList;
4 import java.util.Arrays;
5 import java.util.List;
6
7 import org.springframework.stereotype.Service;
8
9 import com.mytutorial.springbootrestapi.model.Product;
10
11 @Service
12 public class ProductService {
13
14     private List<Product> products = new ArrayList<> (Arrays.asList(
15         new Product("P101", "Monitor", "Electronics"),
16         new Product("P102", "Blanket", "Household"),
17         new Product("P103", "Laptop", "Electronics"),
18         new Product("P104", "Shirt", "Fashion"),
19         new Product("P105", "Pens", "School")));
20
21     public List<Product> getAllProducts() {
22         return products;
23     }
24
25     public Product getProduct(String id) {
26         return products.stream().filter(p -> p.getId().equals(id)).findFirst().get();
27     }
28
29     public void addProduct(Product product) {
30         products.add(product);
31     }
32
33     public void updateProduct(String id, Product product) {
34         for (int i=0; i<products.size(); i++) {
35             Product p = products.get(i);
36             if (p.getId().equals(id)) {
37                 products.set(i, product);
38             }
39         }
40     }
41 }
```

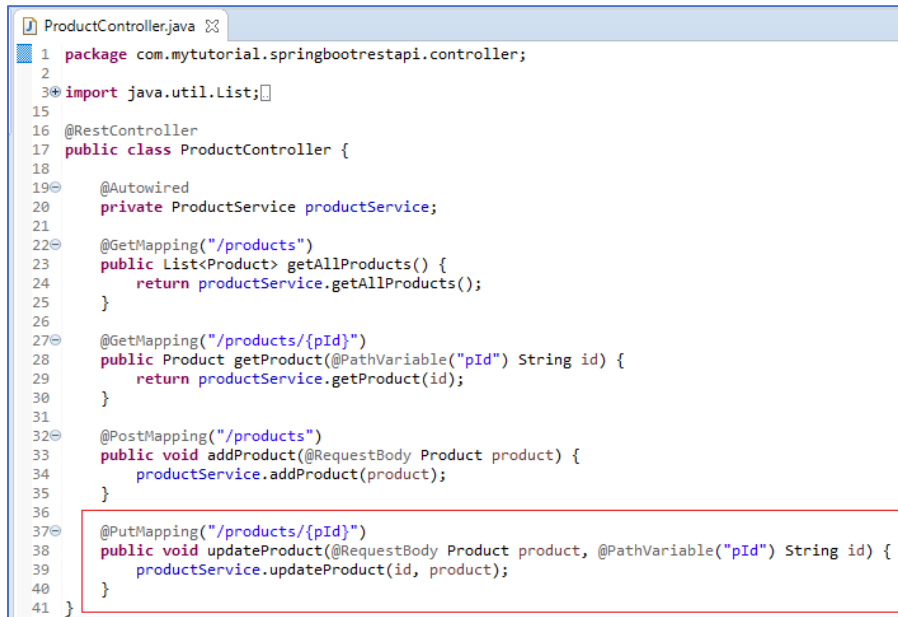
For every product, we'll check to see whether the id of the product is equal to the id that we passed in as an input argument to this method. If this is true, we'll go ahead and set a new product instance at that index. This will effectively get rid of the old product instance and the new product instance with the updated detail will become part of our web app.

Everything is still running in memory, keep that in mind. There's no database yet.

# Spring Boot Microservices: Building RESTful API Services

## 2. Update the Controller Class

Let's head over to the ProductController where we'll add a handler mapping. In order to be able to handle product update requests from the client at the very bottom, I have a method here called update product, which has an **@PutMapping** annotation. This responds to http Put requests.



```
ProductController.java
1 package com.mytutorial.springbootrestapi.controller;
2
3 import java.util.List;
15
16 @RestController
17 public class ProductController {
18
19     @Autowired
20     private ProductService productService;
21
22     @GetMapping("/products")
23     public List<Product> getAllProducts() {
24         return productService.getAllProducts();
25     }
26
27     @GetMapping("/products/{pId}")
28     public Product getProduct(@PathVariable("pId") String id) {
29         return productService.getProduct(id);
30     }
31
32     @PostMapping("/products")
33     public void addProduct(@RequestBody Product product) {
34         productService.addProduct(product);
35     }
36
37     @PutMapping("/products/{pId}")
38     public void updateProduct(@RequestBody Product product, @PathVariable("pId") String id) {
39         productService.updateProduct(id, product);
40     }
41 }
```

Update operations are typically Put request rather than Post request. Because Put implies that any updates that we perform are idempotent.

Updating a product is idempotent even if you call this method multiple times. The product will be updated in exactly the same way, there'll be no change to the result.

Notice that this PutMapping contains a dynamic element in its URL path, the id of the product that we want to update. This is extracted in the id member variable because of the **@PathVariable** annotation that you see on it.

The details of the updated product are present in the RequestBody. The product instance is an input argument here, it's annotated using **@RequestBody**.

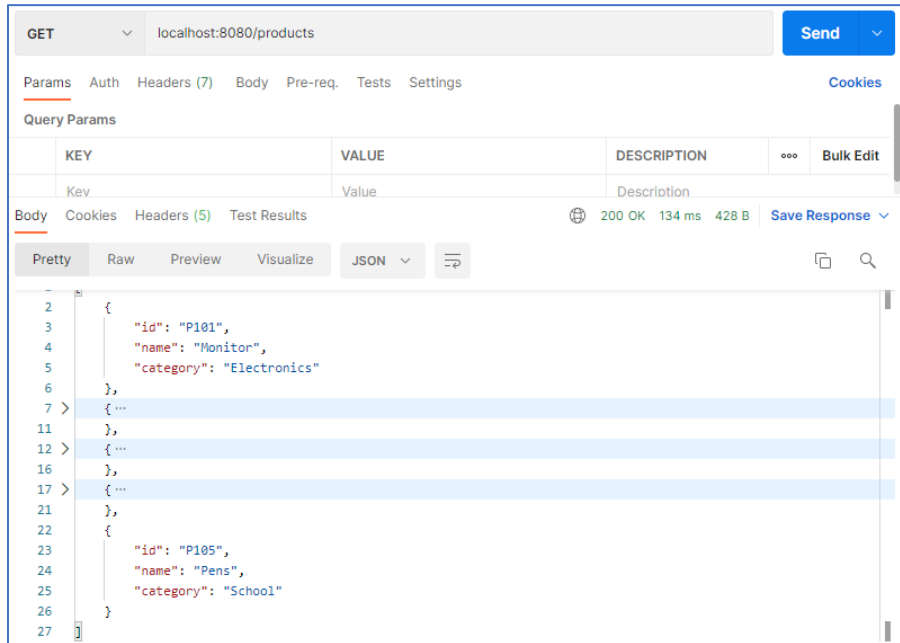
We call *productService.update*, pass in the id and the product and the update will be performed.

# Spring Boot Microservices: Building RESTful API Services

## 3. Run and Test

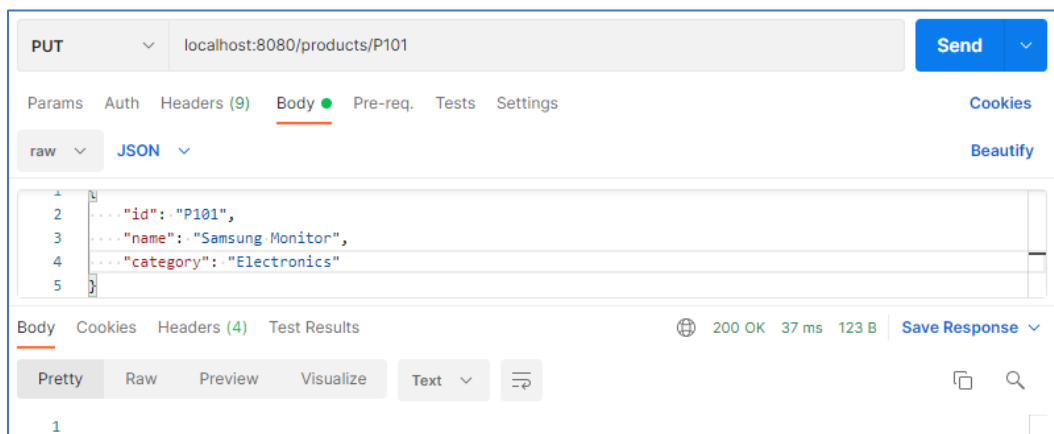
We've now set up stuff, so we can run this code and see how this update operation works. Head back to the advanced REST client.

Let's do a GET request to `/products`. You can see the state of the list of products here as it exists before we perform any updates. Make special note of P101 and 105. Those are the products that we'll update in just a bit.



Let's first update the product with id `101`. I'm going to change the method here so that we make **PUT** HTTP request to our API service to perform updates. Now our Request URL will contain the product id that we want updated, `P101`. The details of the updated product we specify within the request body. Click on the **BODY** tab, and let's specify the product details in the json format.

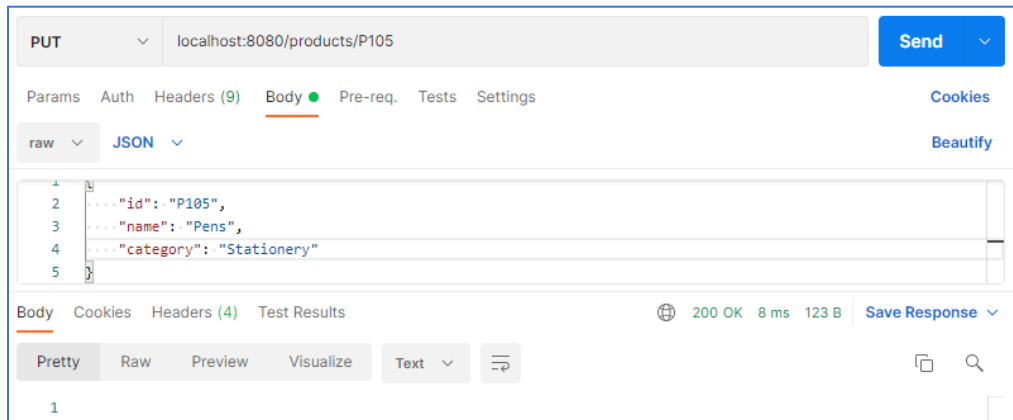
Make sure you update the *body* content type to be *application/json*. I want the product with id P101 to have the name Samsung Monitor. Let's format our JSON so that it's all pretty. And let's now make a PUT request to our server.



# Spring Boot Microservices: Building RESTful API Services

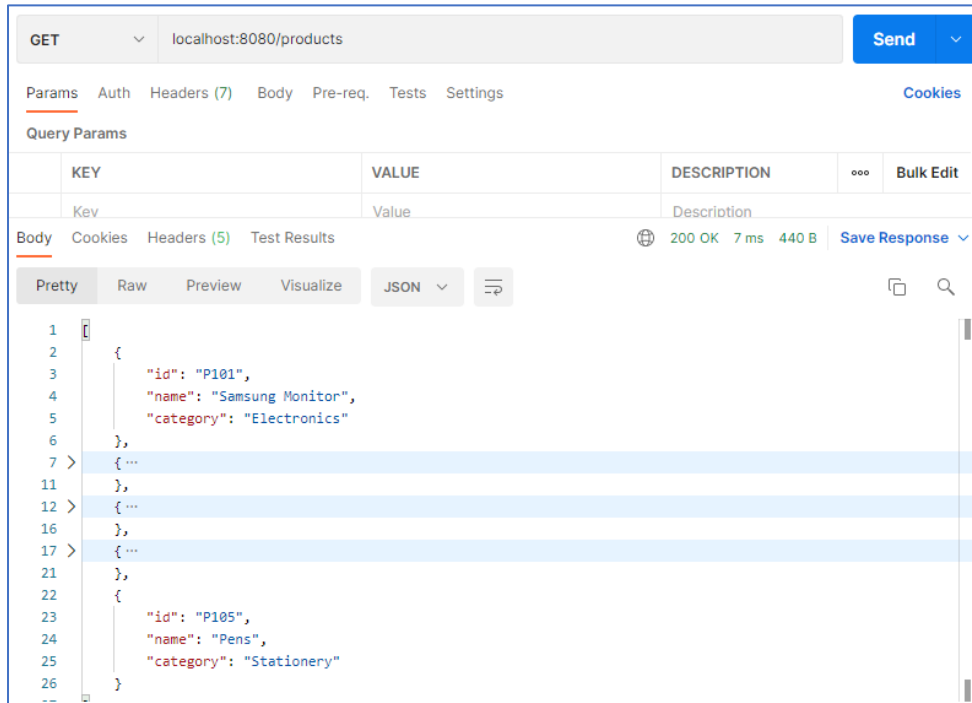
Click on the SEND button to make a PUT request to update product with id 101. The PUT request goes through successfully.

I'm now going to change the request body to update product 105.



Notice I've made a corresponding change to the URL for update as well. The URL contains *P105*, it's once again a **PUT** request. The name of the product is Pens, the category is Stationery rather than school. Click on SEND, we get a **200 OK** in the response, our PUT request or update was successful.

Now let's switch over and make a GET request to the list of products in our app. We need to get rid of this request body here, that's not part of the GET request for the list of products.



Click on the SEND button and let's see the list of products retrieved from our server. You can see that P101 now has the name Samsung Monitor and the product with id 105 is now in the Stationery category.

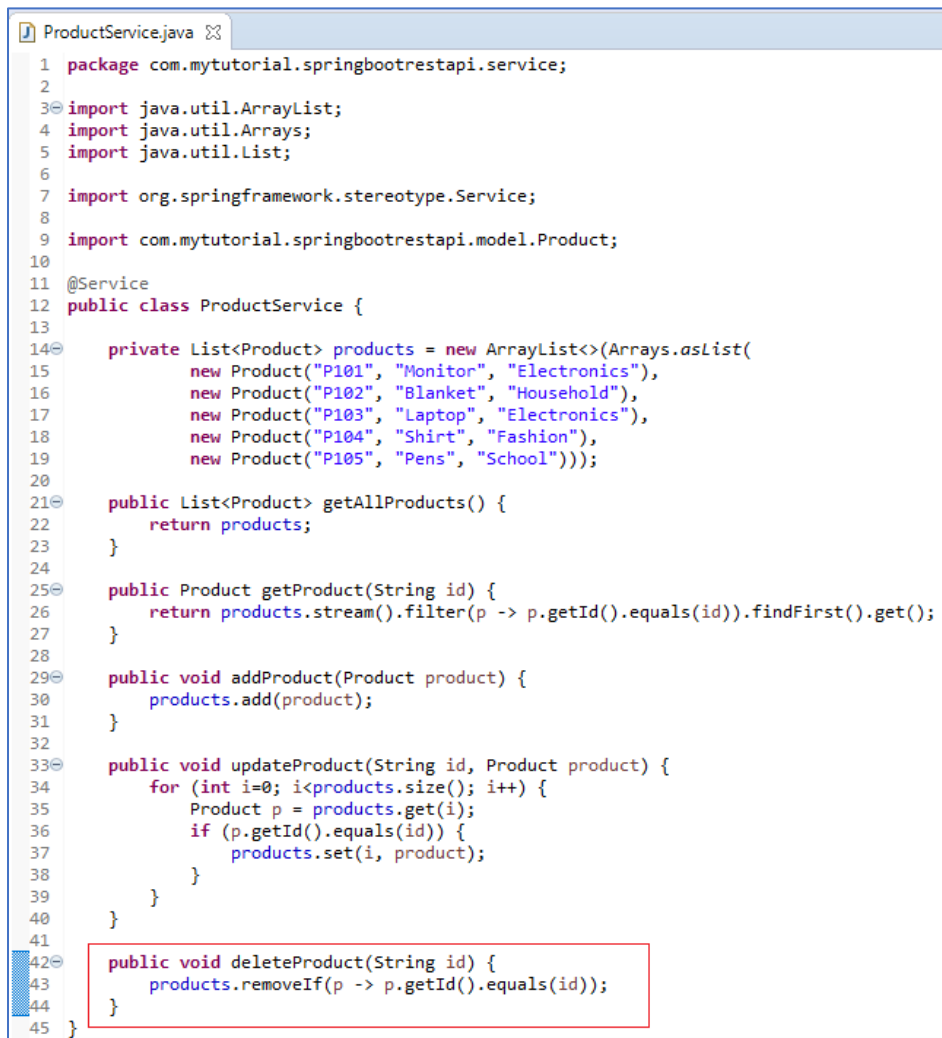
# Spring Boot Microservices: Building RESTful API Services

## Performing Delete Operations

Amongst the CRUD operations we are yet to implement deletion. And that's exactly what we'll do here in this demo.

### 1. Update the Server Class

So the changes that we'll make are in the ProductService class here, scroll down to the very bottom. Notice that I've added a new method called deleteProduct, which takes as an input argument the string id of a product.



```
1 package com.mytutorial.springbootrestapi.service;
2
3 import java.util.ArrayList;
4 import java.util.Arrays;
5 import java.util.List;
6
7 import org.springframework.stereotype.Service;
8
9 import com.mytutorial.springbootrestapi.model.Product;
10
11 @Service
12 public class ProductService {
13
14     private List<Product> products = new ArrayList<>(Arrays.asList(
15         new Product("P101", "Monitor", "Electronics"),
16         new Product("P102", "Blanket", "Household"),
17         new Product("P103", "Laptop", "Electronics"),
18         new Product("P104", "Shirt", "Fashion"),
19         new Product("P105", "Pens", "School")));
20
21     public List<Product> getAllProducts() {
22         return products;
23     }
24
25     public Product getProduct(String id) {
26         return products.stream().filter(p -> p.getId().equals(id)).findFirst().get();
27     }
28
29     public void addProduct(Product product) {
30         products.add(product);
31     }
32
33     public void updateProduct(String id, Product product) {
34         for (int i=0; i<products.size(); i++) {
35             Product p = products.get(i);
36             if (p.getId().equals(id)) {
37                 products.set(i, product);
38             }
39         }
40     }
41
42     public void deleteProduct(String id) {
43         products.removeIf(p -> p.getId().equals(id));
44     }
45 }
```

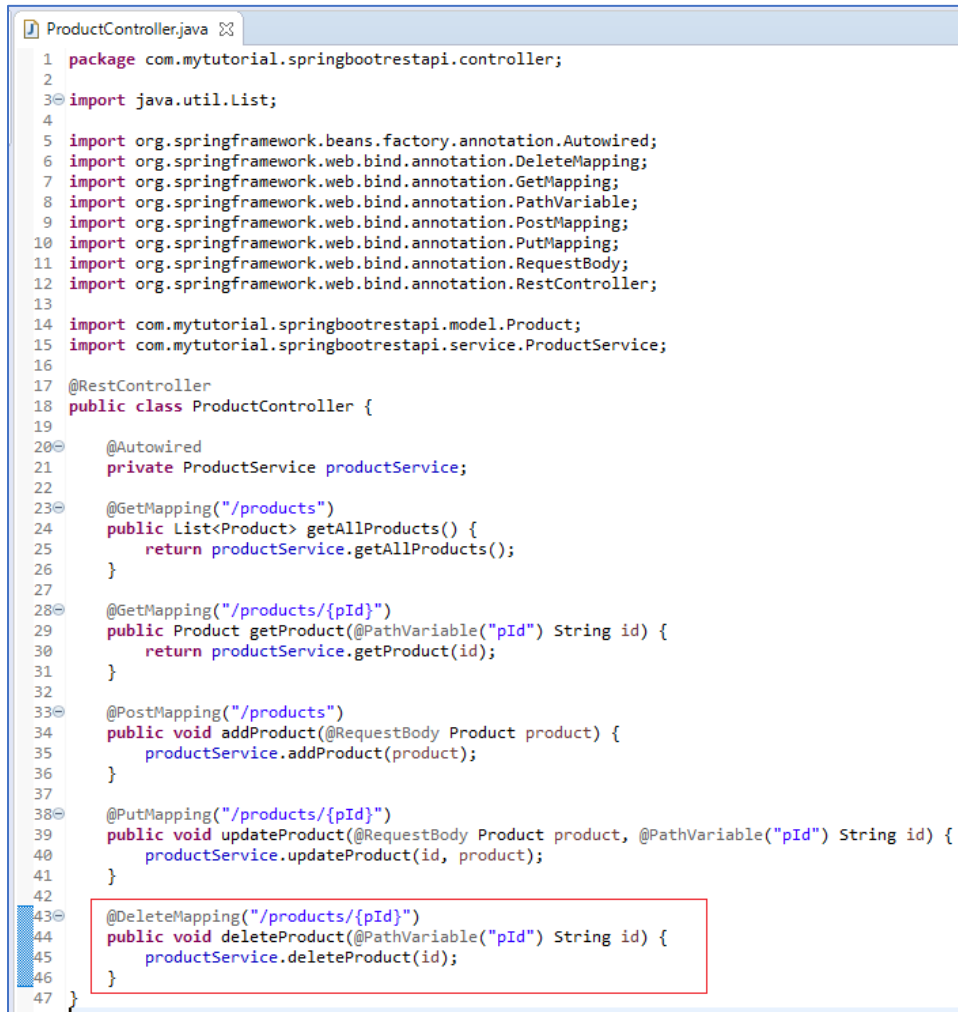
We then perform a **products.removeIf** operation.

If the product ID is equal to the ID that is passed in, that product will be removed from our list.

# Spring Boot Microservices: Building RESTful API Services

## 2. Update the Controller Class

The next step is to expose this delete operation via the product controller handler mapping. Here we are on the ProductController. Scroll down to the very bottom and you can see we have a deleteProduct method, which has an **@DeleteMapping** annotation.



```
1 package com.mytutorial.springbootrestapi.controller;
2
3 import java.util.List;
4
5 import org.springframework.beans.factory.annotation.Autowired;
6 import org.springframework.web.bind.annotation.DeleteMapping;
7 import org.springframework.web.bind.annotation.GetMapping;
8 import org.springframework.web.bind.annotation.PathVariable;
9 import org.springframework.web.bind.annotation.PostMapping;
10 import org.springframework.web.bind.annotation.PutMapping;
11 import org.springframework.web.bind.annotation.RequestBody;
12 import org.springframework.web.bind.annotation.RestController;
13
14 import com.mytutorial.springbootrestapi.model.Product;
15 import com.mytutorial.springbootrestapi.service.ProductService;
16
17 @RestController
18 public class ProductController {
19
20     @Autowired
21     private ProductService productService;
22
23     @GetMapping("/products")
24     public List<Product> getAllProducts() {
25         return productService.getAllProducts();
26     }
27
28     @GetMapping("/products/{pId}")
29     public Product getProduct(@PathVariable("pId") String id) {
30         return productService.getProduct(id);
31     }
32
33     @PostMapping("/products")
34     public void addProduct(@RequestBody Product product) {
35         productService.addProduct(product);
36     }
37
38     @PutMapping("/products/{pId}")
39     public void updateProduct(@RequestBody Product product, @PathVariable("pId") String id) {
40         productService.updateProduct(id, product);
41     }
42
43     @DeleteMapping("/products/{pId}")
44     public void deleteProduct(@PathVariable("pId") String id) {
45         productService.deleteProduct(id);
46     }
47 }
```

Notice that this annotation contains a dynamic path variable. The delete mapping is for the `/products/{pId}` path. This product ID is injected as an input argument to this method. It's annotated using the **@PathVariable** annotation.

We then call `productService.deleteProduct` and pass in the `id` to the `productService`.

# Spring Boot Microservices: Building RESTful API Services

## 3. Run and Test

Let's run this code and see how things work. We are still working in memory, we haven't connected to a database yet. Here we are on the Advanced REST Client. I'll first make a GET request to the /products path to get the list of products that we are working with. I've sent this GET request, now let's scroll down and you can see that we have a total of five products here. The name and category of each product.

The screenshot shows the Advanced REST Client interface. The top bar indicates a GET request to localhost:8080/products. The 'Send' button is visible. Below the URL bar, the 'Params' tab is selected, showing a table for Query Params. The 'Body' tab is also visible, showing the response body in JSON format. The response is a JSON array of 5 products:

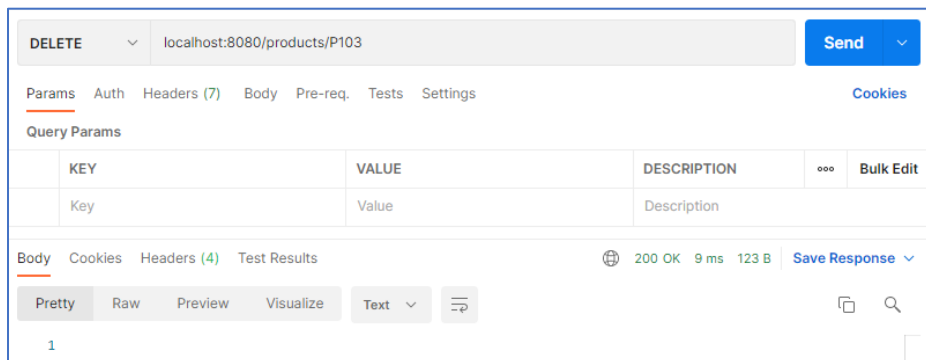
```
[
  {
    "id": "P101",
    "name": "Monitor",
    "category": "Electronics"
  },
  {
    "id": "P102",
    "name": "Blanket",
    "category": "Household"
  },
  {
    "id": "P103",
    "name": "Laptop",
    "category": "Electronics"
  },
  {
    "id": "P104",
    "name": "Shirt",
    "category": "Fashion"
  },
  {
    "id": "P105",
    "name": "Pens",
    "category": "School"
  }
]
```

We are now ready to test the deletion operation. I'm going to make a **DELETE** request to the server. And in the **DELETE** request path, I'll specify the ID of the product that we want to delete. We'll delete the product with id *P101*. This will be extracted using the **@PathVariable** annotation and this product will be deleted.

The screenshot shows the Advanced REST Client interface. The top bar indicates a DELETE request to localhost:8080/products/P101. The 'Send' button is visible. Below the URL bar, the 'Params' tab is selected, showing a table for Query Params. The 'Body' tab is also visible, showing the response body in Text format. The response is 200 OK.

# Spring Boot Microservices: Building RESTful API Services

This request returns with a **200 OK**. Deletion seems to be working fine, but let's try this with one more product. This time we'll delete the product with id P103 as you can see from the path that we have specified. Hit **SEND**, we get a **200 OK** in the response.

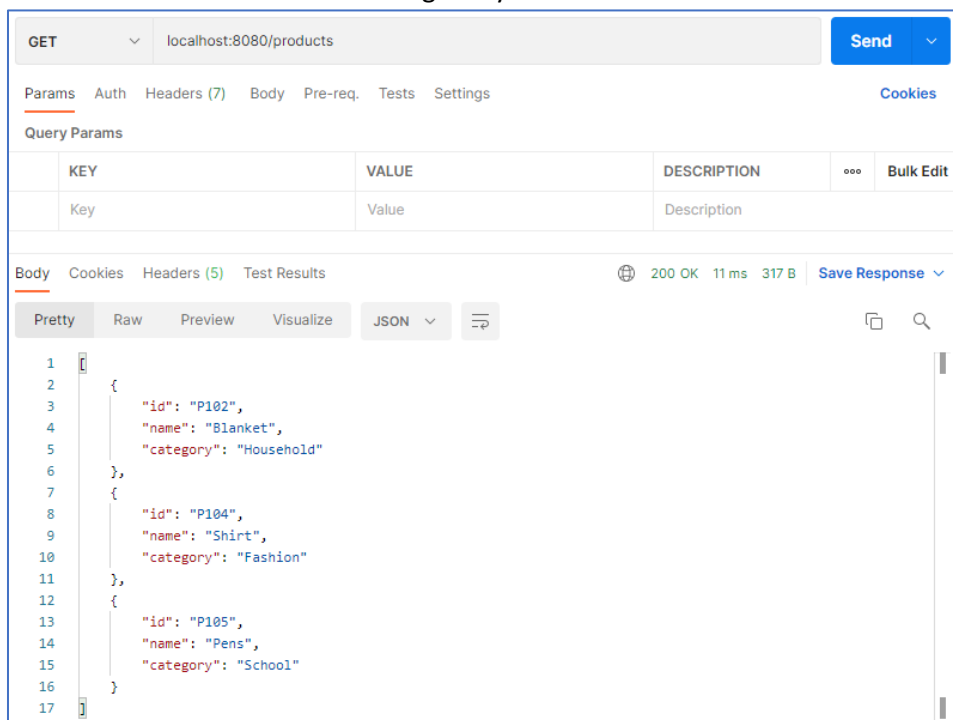


The screenshot shows a REST client interface with the following details:

- Method:** DELETE
- URL:** localhost:8080/products/P103
- Buttons:** Send, Cookies
- Tabs:** Params, Auth, Headers (7), Body, Pre-req., Tests, Settings
- Query Params Table:**

KEY	VALUE	DESCRIPTION	...	Bulk Edit
Key	Value	Description		
- Response:** 200 OK, 9 ms, 123 B. Buttons: Save Response, Pretty, Raw, Preview, Visualize, Text.

Let's check to see whether the products have indeed been deleted, we'll change our HTTP method to be GET rather than DELETE. And let's make a GET request to get a list of all products. When we hit **SEND**, you'll see that the products that have been retrieved are exactly three. Products P101 and P103 are missing. They have been deleted.



The screenshot shows a REST client interface with the following details:

- Method:** GET
- URL:** localhost:8080/products
- Buttons:** Send, Cookies
- Tabs:** Params, Auth, Headers (7), Body, Pre-req., Tests, Settings
- Query Params Table:** (Same as the previous screenshot)
- Response:** 200 OK, 11 ms, 317 B. Buttons: Save Response, Pretty, Raw, Preview, Visualize, JSON.
- JSON Response:**

```
[
  {
    "id": "P102",
    "name": "Blanket",
    "category": "Household"
  },
  {
    "id": "P104",
    "name": "Shirt",
    "category": "Fashion"
  },
  {
    "id": "P105",
    "name": "Pens",
    "category": "School"
  }
]
```



# Spring Boot Microservices: Building RESTful API Services

## Setup MySQL Database on Docker

Prepare **docker-compose.yml** with following content

```
docker-compose.yml
1  version: '3.3'
2  services:
3    db:
4      image: mysql:5.7
5      container_name: mysqlldb
6      restart: always
7      environment:
8        MYSQL_DATABASE: 'db'
9        # So you don't have to use root, but you can if you like
10       MYSQL_USER: 'user'
11       # You can use whatever password you like
12       MYSQL_PASSWORD: 'password'
13       # Password for root access
14       MYSQL_ROOT_PASSWORD: 'password'
15     ports:
16       # <Port exposed> : <MySQL Port running inside container>
17       - '3306:3306'
18     expose:
19       # Opens port 3306 on the container
20       - '3306'
```

And run following command on terminal

**\$> docker compose up -d**

To Check if docker container is running with following command

**\$> docker ps**

Make sure the docker container is up and running

```
PS C:\AnnualScienceFair> docker compose up -d
[+] Running 1/1
- Container mysqlldb Started      3.9s

PS C:\AnnualScienceFair> docker ps
CONTAINER ID   IMAGE                                COMMAND                  CREATED        STATUS        PORTS                               NAMES
3ffa5b8e551e   2c9028880e58   "docker-entrypoint.s..." About a minute ago   Up About a minute   0.0.0.0:3306->3306/tcp, :::3306->3306/tcp, 33060/tcp   mysqlldb

PS C:\AnnualScienceFair>
```

To Start MySQL Service on Docker Container

**\$> docker start mysqlldb**

To Stop MySQLService on Docker Container

**\$> docker stop mysqlldb**

To Remove MySQL Container on Docker

**\$> docker container rm mysqlldb**

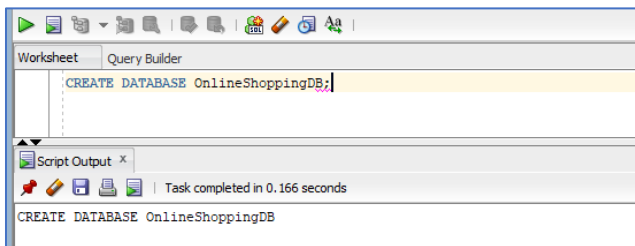
# Spring Boot Microservices: Building RESTful API Services

## Integrating With MySQL

Now that we have MySQL installed and set up on our local machine, we are finally ready to wire up our API service to an underlying MySQL database. You'll see how easy it is to do in Spring Boot using Spring Data, JPA, and Hibernate.

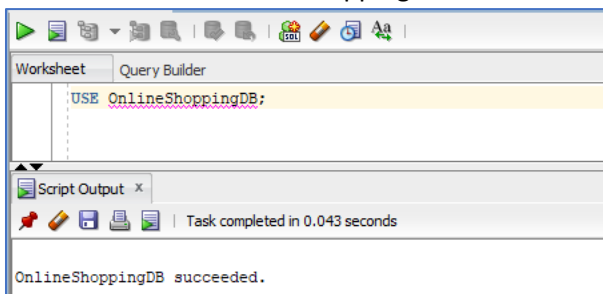
### 1. Create Database

Here we are on the MySQL Workbench, I'm going to create a database that we'll connect to from our application.



Use the CREATE DATABASE command to create a database named OnlineShoppingDB. In order to execute this command, click on this little lightning bolt icon, that will execute the statement under the keyboard cursor. We've created the database OnlineShoppingDB, as you can see from the bottom green check mark here.

Let's now use this OnlineShoppingDB database so that we can run queries on tables within it.



Now that we are within the OnlineShoppingDB database, let's run a SELECT\* operation from the products table. Now clearly we haven't created this table yet. So this statement returns an error. This table will be created by JPA and the Hibernate ORM framework when we run our application.

# Spring Boot Microservices: Building RESTful API Services

## 2. Create Maven Project

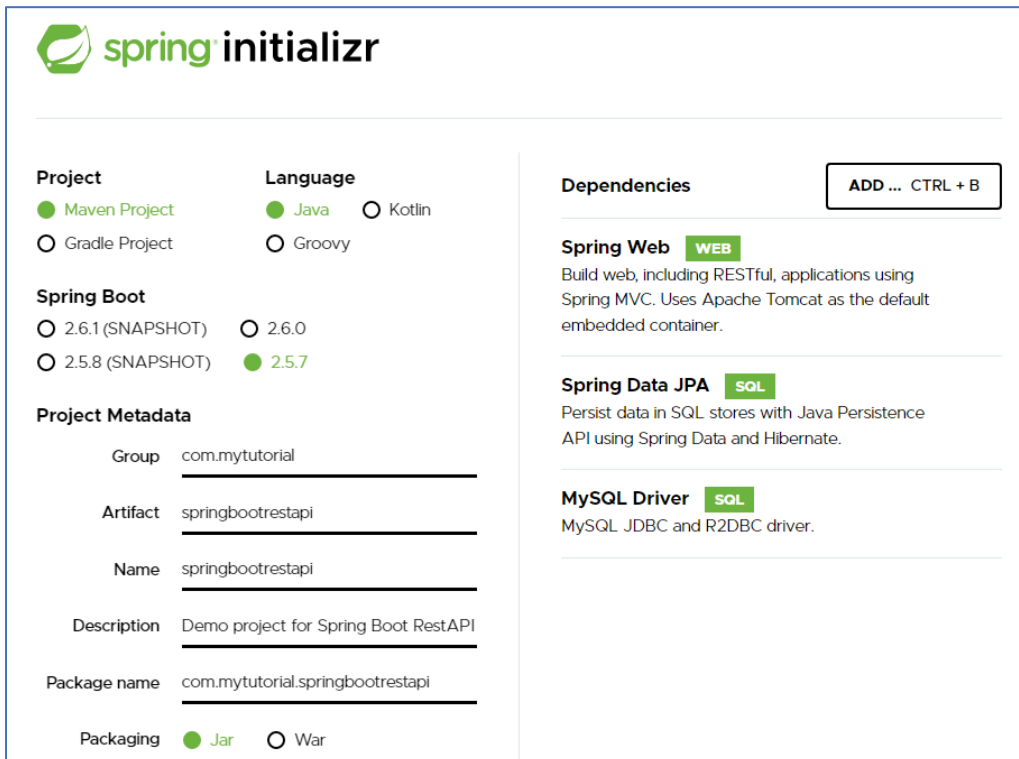
Now I'm going to set up a new project because we need a few new dependencies. And I'm going to use spring initializr for this. Once again, it's a Maven Java project using Spring Boot 2.5.7. We'll continue to work in the same package as we have before. In fact, we'll reuse the code that we've written in previous demos. The package name is *com.mytutorial*.

Now let's add the dependencies that we need. Now this is going to be a web application because we are exposing a RESTful service. So make sure you include **Spring Web** as a dependency. There's no change there.

Click on Add Dependencies once again. This time we'll choose **Spring Data**. The Spring Data JPA starter template allows you to persist data in a SQL database using the Java Persistence API and the frameworks Spring Data and Hibernate.

Hibernate is an object relational mapping framework, which maps your Java objects or classes to underlying database tables. JPA or the Java Persistence API is what you'll use to work with Hibernate. And Spring Data does a lot of magic for you, basically provides implementations for many common database operations for your table.

Now, the Hibernate ORM framework connects to the underlying database table using JDBC. And for that, you need to add the **MySQL Driver** as a dependency. These are the three dependencies that we need in order to connect the API service that we've built so far to a MySQL database.



The screenshot shows the Spring Initializr web application interface. It is divided into several sections for configuring a new project:

- Project:** Includes radio buttons for **Maven Project** (selected) and **Gradle Project**.
- Language:** Includes radio buttons for **Java** (selected) and **Kotlin**, and **Groovy**.
- Spring Boot:** Includes radio buttons for versions **2.6.1 (SNAPSHOT)**, **2.6.0**, **2.5.8 (SNAPSHOT)**, and **2.5.7** (selected).
- Project Metadata:** A form with fields for:
  - Group:** `com.mytutorial`
  - Artifact:** `springbootrestapi`
  - Name:** `springbootrestapi`
  - Description:** `Demo project for Spring Boot RestAPI`
  - Package name:** `com.mytutorial.springbootrestapi`
  - Packaging:** Includes radio buttons for **Jar** (selected) and **War**.
- Dependencies:** A section with a button **ADD ... CTRL + B** and a list of selected dependencies:
  - Spring Web** (tagged **WEB**): Build web, including RESTful, applications using Spring MVC. Uses Apache Tomcat as the default embedded container.
  - Spring Data JPA** (tagged **SQL**): Persist data in SQL stores with Java Persistence API using Spring Data and Hibernate.
  - MySQL Driver** (tagged **SQL**): MySQL JDBC and R2DBC driver.

Click on GENERATE and a zip file with the project structure will be generated for you. I've imported this project into my Eclipse IDE and also set up some of the classes that we had in the previous demo. I've made a few enhancements to those classes as well.

# Spring Boot Microservices: Building RESTful API Services

And we'll look at what those enhancements are. This is what will allow us to connect to the underlying MySQL database. **pom.xml** deserves our attention first.



```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
3   xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">
4   <modelVersion>4.0.0</modelVersion>
5   <parent>
6     <groupId>org.springframework.boot</groupId>
7     <artifactId>spring-boot-starter-parent</artifactId>
8     <version>2.5.7</version>
9     <relativePath/> <!-- lookup parent from repository -->
10  </parent>
11  <groupId>com.mytutorial</groupId>
12  <artifactId>springbootrestapi</artifactId>
13  <version>0.0.1-SNAPSHOT</version>
14  <name>springbootrestapi</name>
15  <description>Demo project for Spring Boot RestAPI</description>
16  <properties>
17    <java.version>1.8</java.version>
18  </properties>
19  <dependencies>
20
21    <dependency>
22      <groupId>org.springframework.boot</groupId>
23      <artifactId>spring-boot-starter-data-jpa</artifactId>
24    </dependency>
25
26    <dependency>
27      <groupId>org.springframework.boot</groupId>
28      <artifactId>spring-boot-starter-web</artifactId>
29    </dependency>
30
31    <dependency>
32      <groupId>mysql</groupId>
33      <artifactId>mysql-connector-java</artifactId>
34      <scope>runtime</scope>
35    </dependency>
36
37    <dependency>
38      <groupId>org.springframework.boot</groupId>
39      <artifactId>spring-boot-starter-test</artifactId>
40      <scope>test</scope>
41    </dependency>
42  </dependencies>
43
44  <build>
45    <plugins>
46      <plugin>
47        <groupId>org.springframework.boot</groupId>
48        <artifactId>spring-boot-maven-plugin</artifactId>
49      </plugin>
50    </plugins>
51  </build>
52
53 </project>
```

This is where we've specify the dependencies for the Spring Boot starters that we need to build a RESTful web service which is connected to an underlying SQL database.

We have the **spring-boot-starter-data-jpa** on line 24, **spring-boot-starter-web** on line 28, and the **mysql-connector-java** on line 33.

# Spring Boot Microservices: Building RESTful API Services

## 3. Preparing Model Object

Our RESTful API service works on a list of products and we can perform CRUD operations on these products.

The product is a model object which was previously a plain old Java object. I've made a few changes to this product object so that it's now an entity that will be stored as a record in the underlying products database table.

```
Product.java
1 package com.mytutorial.springbootrestapi.model;
2
3 import javax.persistence.Entity;
4 import javax.persistence.GeneratedValue;
5 import javax.persistence.GenerationType;
6 import javax.persistence.Id;
7
8 @Entity(name = "products")
9 public class Product {
10
11     @Id
12     @GeneratedValue(strategy = GenerationType.IDENTITY)
13     private Long id;
14     private String name;
15     private String category;
16
17     public Product() {
18     }
19
20     public Product(String name, String category) {
21         this.name = name;
22         this.category = category;
23     }
24
25     public Long getId() {
26         return id;
27     }
28
29     public void setId(Long id) {
30         this.id = id;
31     }
32
33     public String getName() {
34         return name;
35     }
36
37     public void setName(String name) {
38         this.name = name;
39     }
40
41     public String getCategory() {
42         return category;
43     }
44
45     public void setCategory(String category) {
46         this.category = category;
47     }
48
49 }
```

The **@Entity** annotation that you see here on the Product class is a part of the **Java Persistence API**. This tags this class as mapped to a underlying database table. The name of the table is *products*.

Every object of this class, that is, every product instance in our application maps to a record or a row in the products table. Every entity in JPA has to have a **primary key**. Here the primary key I have specified to be of type *Long*. You know this is the primary key because of the **@Id** annotation we have on that member variable.

# Spring Boot Microservices: Building RESTful API Services

The **@GeneratedValue** annotation means that JPA and Hibernate will automatically generate values for this primary key.

The member variables defined in any entity class correspond to the columns of the database table. The products database table that the product entity is mapped to will have three columns, the id, name, and category of the product. Just by using these JPA annotations, you've indicated to the Spring framework that instances of the product object are basically records in the underlying database table.

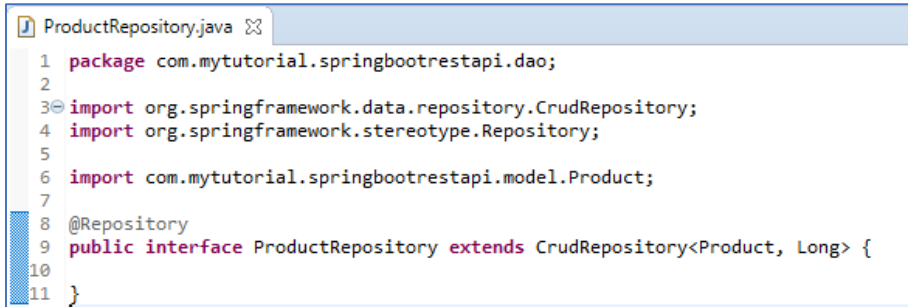
Make sure you set up the constructors for the product object. You need a default no argument constructor. You need a constructor with arguments, one which you can use within your code.

The other methods that you see here are simply getters and setters for the member variables of this class.

# Spring Boot Microservices: Building RESTful API Services

## 4. Prepare Repository Interface

Let's now head over to the **ProductRepository.java** file.

A screenshot of a code editor showing the ProductRepository.java file. The code is as follows:

```
1 package com.mytutorial.springbootrestapi.dao;
2
3 import org.springframework.data.repository.CrudRepository;
4 import org.springframework.stereotype.Repository;
5
6 import com.mytutorial.springbootrestapi.model.Product;
7
8 @Repository
9 public interface ProductRepository extends CrudRepository<Product, Long> {
10
11 }
```

The product repository is in the *com.mytutorial.springbootrestapi.dao* namespace. The DAO namespace contains classes which interact directly with our underlying database. These are the data access classes. Now this product repository class seems surprisingly sparse. How does this actually work? It's not really connecting to a database. All we have is an **interface**. This is the magic of Spring Data.

Now there are a bunch of things going on here. Let's walk through them step by step. Notice the **@Repository** annotation. This basically tells Spring that this is a spring controlled component, Spring managed component. Just like any other Spring bean, except that this component will be used to access data from an underlying database table. Exceptions thrown by this component will be converted to Data Access exceptions by Spring.

Now, you'll see that this product repository is just an interface. *Where is the implementation?*

This is where we'll talk about the **CrudRepository** interface that product repository extends. The first thing to note is that the **CrudRepository** interface is a **generic interface** with two generic type parameters. We have specified the values of these type parameters as *Product*, *Long*. This means that the *CrudRepository* interface will expose, **create, read, update, and delete** operations for the product entity that we have defined. And this product entity has a primary key of type *Long*.

Now let's answer the question, *if this is an interface, where is the implementation?* Now it turns out, that the magic of Spring Data is such that **You don't have to specify the implementation** yourself. You simply indicate that your *ProductRepository* extends the *CrudRepository*. You specify the type of your entity, the type of the primary key, and **The implementation will be auto-generated by the Spring Data module**.

The implementations for all of the methods expose by the *CrudRepository* interface will be automatically available to you. Some examples of the methods that the *CrudRepository* exposes are **findById**, **findAll**, **save**, **saveAll**, **delete**, **deleteAll**, **existById**, **count**, and *so on*. You can just invoke these methods on any member variable of type *ProductRepository* and the invocations will work fine.

# Spring Boot Microservices: Building RESTful API Services

## 5. Prepare Service Class

As you shall see in the ProductService class that we have set up here. Now we've changed the ProductService class. **ProductService.java**

```
ProductService.java
1 package com.mytutorial.springbootrestapi.service;
2
3 import java.util.ArrayList;
4 import java.util.List;
5 import java.util.Optional;
6
7 import org.springframework.beans.factory.annotation.Autowired;
8 import org.springframework.stereotype.Service;
9
10 import com.mytutorial.springbootrestapi.dao.ProductRepository;
11 import com.mytutorial.springbootrestapi.model.Product;
12
13 @Service
14 public class ProductService {
15
16     @Autowired
17     private ProductRepository productRepository;
18
19     public List<Product> getAllProducts() {
20         List<Product> products = new ArrayList<>();
21         productRepository.findAll().forEach(products::add);
22         return products;
23     }
24
25     public Optional<Product> getProduct(Long id) {
26         return productRepository.findById(id);
27     }
28
29     public void addProduct(Product product) {
30         productRepository.save(product);
31     }
32
33     public void updateProduct(Long id, Product product) {
34         if (productRepository.findById(id).get() != null)
35             productRepository.save(product);
36     }
37
38     public void deleteProduct(Long id) {
39         productRepository.deleteById(id);
40     }
41 }
```

It's tagged using **@Service** as before, but instead of holding the products in memory, the products will be stored in our MySQL database. We inject the ProductRepository into this product service using **@Autowired**. And all operations on the product service, we delegate to the ProductRepository.

For example, *getAllProducts* which retrieves a list of all the products in our database. All we have to do is call *productRepository.findAll* and add all of the product entities retrieved to our products list.

Let's take a look at the other methods as well. All we do is call the ProductRepository, *getProduct* which takes as input argument, the primary key of the product Long id. We'll simply call *productRepository.findById*. *getProduct* returns an **optional** product, because a product with that Id may not exist.

*Add Product* simply calls *productRepository.save*.

*Update Product*, which takes an input argument the primary key and a product instance uses *productRepository.findById* to check whether a particular product with this ID exists. If it does, it updates that product by calling *productRepository.save*.



# Spring Boot Microservices: Building RESTful API Services

And finally, *Delete Product* simply invokes *productRepository.deleteById*.

## 6. Prepare Custom Exception Controller Class

We've enhanced our application a bit by adding a *ProductNotFoundException*. If a product with an ID is not found, this exception will be thrown.

```
ProductNotFoundException.java
1 package com.mytutorial.springbootrestapi.exception;
2
3 public class ProductNotFoundException extends RuntimeException {
4
5     private static final long serialVersionUID = 1L;
6
7     public ProductNotFoundException(Long id) {
8         super(String.format("The prodduct with %d cannot be found", id));
9     }
10
11 }
```

We've also added a programmatic exception handler to our RESTful API service. This *ProductNotFoundResponse* is the response that is sent back to the user when a product is not found. You can see that this class is annotated using **@ControllerAdvice**.

```
ProductNotFoundResponse.java
1 package com.mytutorial.springbootrestapi.exception;
2
3 import org.springframework.http.HttpStatus;
4 import org.springframework.web.bind.annotation.ControllerAdvice;
5 import org.springframework.web.bind.annotation.ExceptionHandler;
6 import org.springframework.web.bind.annotation.ResponseBody;
7 import org.springframework.web.bind.annotation.ResponseStatus;
8
9 @ControllerAdvice
10 public class ProductNotFoundResponse {
11
12     @ResponseBody
13     @ExceptionHandler(ProductNotFoundException.class)
14     @ResponseStatus(HttpStatus.NOT_FOUND)
15     public String productNotFoundHandler(ProductNotFoundException exception) {
16         return exception.getMessage();
17     }
18 }
```

This annotation is meant for classes that specify methods that are used across multiple controllers such as exception handlers. Notice, that we have a single method here, *productNotFoundHandler*, which takes as an input argument, the *ProductNotFoundException*, and returns the message of that exception.

The **@ResponseBody** annotation basically says that the message of the exception is basically the web response that should be rendered to the user.

The **@ExceptionHandler** annotation declares this method as one which handles exceptions, specifically the *ProductNotFoundException*.

The **@ResponseStatus** annotation indicates that when this exception handler is hit and a response is returned to the user, the HTTP status should be not found or 404.

# Spring Boot Microservices: Building RESTful API Services

## 7. The Controller Class

The code in the ProductController class hasn't changed at all. All of the CRUD operations that we saw in the previous demo continue to be exactly the same.

```
ProductController.java
1 package com.mytutorial.springbootrestapi.controller;
2
3 import java.util.List;
4
5 import org.springframework.beans.factory.annotation.Autowired;
6 import org.springframework.web.bind.annotation.DeleteMapping;
7 import org.springframework.web.bind.annotation.GetMapping;
8 import org.springframework.web.bind.annotation.PathVariable;
9 import org.springframework.web.bind.annotation.PostMapping;
10 import org.springframework.web.bind.annotation.PutMapping;
11 import org.springframework.web.bind.annotation.RequestBody;
12 import org.springframework.web.bind.annotation.RestController;
13
14 import com.mytutorial.springbootrestapi.exception.ProductNotFoundException;
15 import com.mytutorial.springbootrestapi.model.Product;
16 import com.mytutorial.springbootrestapi.service.ProductService;
17
18 @RestController
19 public class ProductController {
20
21     @Autowired
22     private ProductService productService;
23
24     @GetMapping("/products")
25     public List<Product> getAllProducts() {
26         return productService.getAllProducts();
27     }
28
29     @GetMapping("/products/{pId}")
30     public Product getProduct(@PathVariable("pId") Long id) {
31         return productService.getProduct(id).orElseThrow(() -> new ProductNotFoundException(id));
32     }
33
34     @PostMapping("/products")
35     public void addProduct(@RequestBody Product product) {
36         productService.addProduct(product);
37     }
38
39     @PutMapping("/products/{pId}")
40     public void updateProduct(@RequestBody Product product, @PathVariable("pId") Long id) {
41         productService.updateProduct(id, product);
42     }
43
44     @DeleteMapping("/products/{pId}")
45     public void deleteProduct(@PathVariable("pId") Long id) {
46         productService.deleteProduct(id);
47     }
48 }
```

# Spring Boot Microservices: Building RESTful API Services

## 8. Entry Point Spring Boot Application Class

There's one small change that we have made to the main entry point of our application, the `SpringBootApplication` class. Notice that we've injected a *ProductRepository* directly to the SpringBoot application. We also have the application implement the **CommandLineRunner** interface indicating that this is an executable.

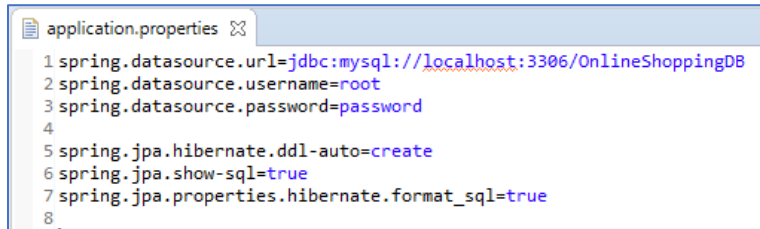
```
SpringbootrestapiApplication.java
1 package com.mytutorial.springbootrestapi;
2
3 import org.springframework.beans.factory.annotation.Autowired;
4 import org.springframework.boot.CommandLineRunner;
5 import org.springframework.boot.SpringApplication;
6 import org.springframework.boot.autoconfigure.SpringBootApplication;
7
8 import com.mytutorial.springbootrestapi.dao.ProductRepository;
9 import com.mytutorial.springbootrestapi.model.Product;
10
11 @SpringBootApplication
12 public class SpringbootrestapiApplication implements CommandLineRunner {
13
14     @Autowired
15     private ProductRepository productRepository;
16
17     public static void main(String[] args) {
18         SpringApplication.run(SpringbootrestapiApplication.class, args);
19     }
20
21     @Override
22     public void run(String... args) throws Exception {
23         productRepository.save(new Product("Television", "Electronics"));
24         productRepository.save(new Product("Air Conditioner", "Electronics"));
25         productRepository.save(new Product("Sofa", "Electronics"));
26         productRepository.save(new Product("Cushions", "Home Essentials"));
27         productRepository.save(new Product("Wardrobe", "Furniture"));
28     }
29 }
30 }
```

We've overridden the `run` method within this class, and basically saved a bunch of products to the underlying database table. So each time we run this application, for the purposes of this demo, we'll have these five products in our products table.

# Spring Boot Microservices: Building RESTful API Services

## 9. Update database info config properties

One last detail to note here in this code. The properties which we use to connect to the underlying MySQL database, we specify in the **application.properties** file.

A screenshot of a code editor showing the content of the application.properties file. The file is titled 'application.properties' with a small icon and a refresh symbol. The code is as follows:

```
1 spring.datasource.url=jdbc:mysql://localhost:3306/OnlineShoppingDB
2 spring.datasource.username=root
3 spring.datasource.password=password
4
5 spring.jpa.hibernate.ddl-auto=create
6 spring.jpa.show-sql=true
7 spring.jpa.properties.hibernate.format_sql=true
8
```

The **spring.datasource.url** is the URL that we use to connect to our MySQL database using JDBC. Notice that our MySQL database is running on localhost:3306, and the database is OnlineShoppingDB.

We have the **username** and **password** for our connection root and password.

The **spring.jpa.hibernate.ddl-auto** is equal to **create**, this basically tells *hibernate to recreate tables each time we run our app*. This is great for prototyping.

The **spring.jpa.hibernate.show-sql** and **spring.jpa.hibernate.format\_sql** properties ensures that the SQL statements that are executed are displayed within our console window for the purposes of debugging.

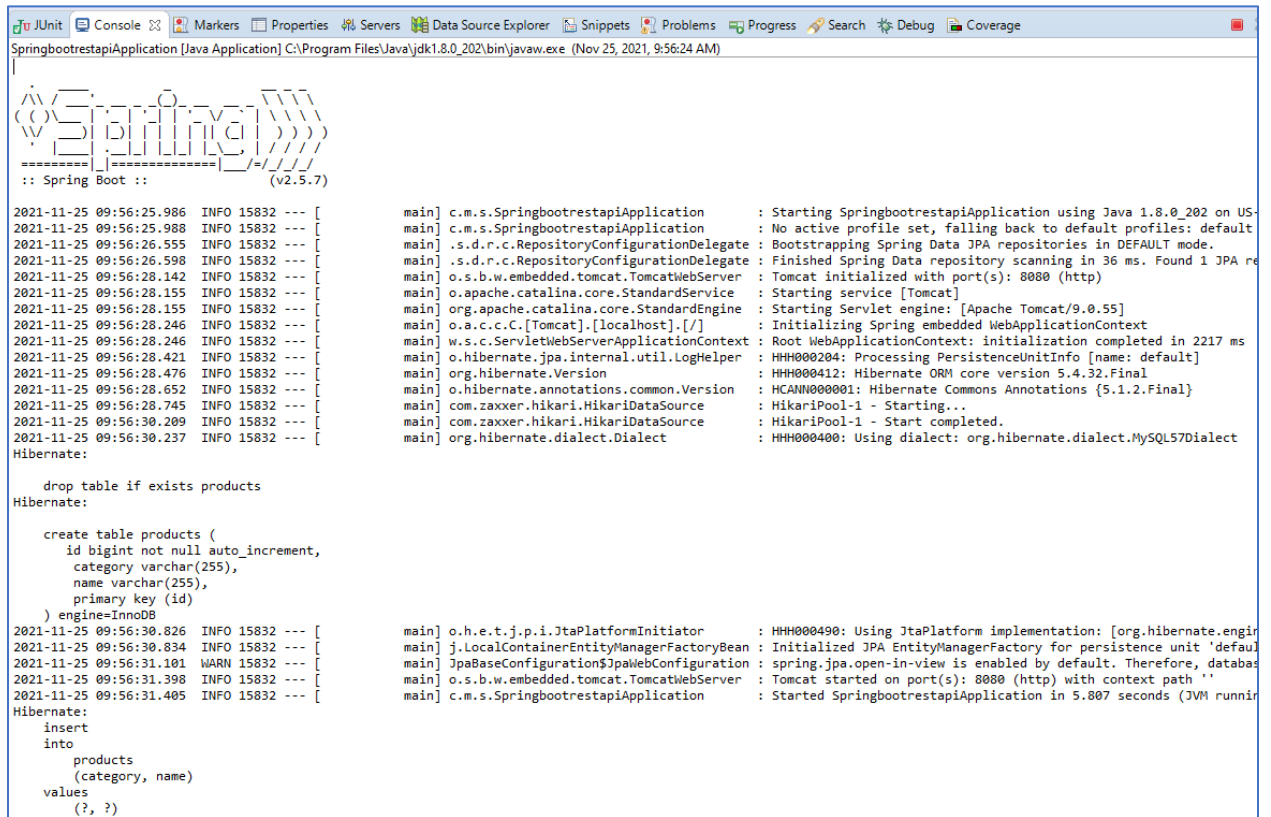
# Spring Boot Microservices: Building RESTful API Services

## 10. Run and Test

In this demo, we'll use the advanced REST client to see how our REST API service works. Remember, this time our REST API is connected to a back end SQL database.

- *Inspect During Start up Process*

Go ahead and run this application and let's take a look at the console messages.



```
SpringbootrestapiApplication [Java Application] C:\Program Files\Java\jdk1.8.0_202\bin\javaw.exe (Nov 25, 2021, 9:56:24 AM)

:: Spring Boot :: (v2.5.7)

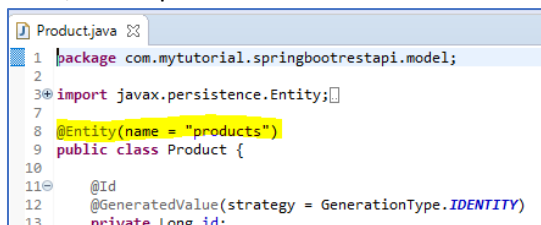
2021-11-25 09:56:25.986 INFO 15832 --- [main] c.m.s.SpringbootrestapiApplication : Starting SpringbootrestapiApplication using Java 1.8.0_202 on US-
2021-11-25 09:56:25.988 INFO 15832 --- [main] c.m.s.SpringbootrestapiApplication : No active profile set, falling back to default profiles: default
2021-11-25 09:56:26.555 INFO 15832 --- [main] s.d.r.c.RepositoryConfigurationDelegate : Bootstrapping Spring Data JPA repositories in DEFAULT mode.
2021-11-25 09:56:26.598 INFO 15832 --- [main] s.d.r.c.RepositoryConfigurationDelegate : Finished Spring Data repository scanning in 36 ms. Found 1 JPA re
2021-11-25 09:56:28.142 INFO 15832 --- [main] o.s.b.w.embedded.tomcat.TomcatWebServer : Tomcat initialized with port(s): 8080 (http)
2021-11-25 09:56:28.155 INFO 15832 --- [main] o.apache.catalina.core.StandardService : Starting service [Tomcat]
2021-11-25 09:56:28.155 INFO 15832 --- [main] org.apache.catalina.core.StandardEngine : Starting Servlet engine: [Apache Tomcat/9.0.55]
2021-11-25 09:56:28.246 INFO 15832 --- [main] o.a.c.c.C.[Tomcat].[/localhost].[/] : Initializing Spring embedded WebApplicationContext
2021-11-25 09:56:28.246 INFO 15832 --- [main] w.s.c.ServletWebServerApplicationContext : Root WebApplicationContext: initialization completed in 2217 ms
2021-11-25 09:56:28.421 INFO 15832 --- [main] o.hibernate.jpa.internal.util.LogHelper : HHH000204: Processing PersistenceUnitInfo [name: default]
2021-11-25 09:56:28.476 INFO 15832 --- [main] org.hibernate.Version : HHH0000412: Hibernate ORM core version 5.4.32.Final
2021-11-25 09:56:28.652 INFO 15832 --- [main] o.hibernate.annotations.common.Version : HCANN000001: Hibernate Commons Annotations {5.1.2.Final}
2021-11-25 09:56:28.745 INFO 15832 --- [main] com.zaxxer.hikari.HikariDataSource : HikariPool-1 - Starting...
2021-11-25 09:56:30.209 INFO 15832 --- [main] com.zaxxer.hikari.HikariDataSource : HikariPool-1 - Start completed.
2021-11-25 09:56:30.237 INFO 15832 --- [main] org.hibernate.dialect.Dialect : HHH0000400: Using dialect: org.hibernate.dialect.MySQL57Dialect

Hibernate:
    drop table if exists products
Hibernate:
    create table products (
      id bigint not null auto_increment,
      category varchar(255),
      name varchar(255),
      primary key (id)
    ) engine=InnoDB
2021-11-25 09:56:30.826 INFO 15832 --- [main] o.h.e.t.j.p.i.JtaPlatformInitiator : HHH000490: Using JtaPlatform implementation: [org.hibernate.engin
2021-11-25 09:56:30.834 INFO 15832 --- [main] j.LocalContainerEntityManagerFactoryBean : Initialized JPA EntityManagerFactory for persistence unit 'default
2021-11-25 09:56:31.101 WARN 15832 --- [main] JpaBaseConfiguration$JpaWebConfiguration : spring.jpa.open-in-view is enabled by default. Therefore, databas
2021-11-25 09:56:31.398 INFO 15832 --- [main] o.s.b.w.embedded.tomcat.TomcatWebServer : Tomcat started on port(s): 8080 (http) with context path ''
2021-11-25 09:56:31.405 INFO 15832 --- [main] c.m.s.SpringbootrestapiApplication : Started SpringbootrestapiApplication in 5.807 seconds (JVM runnin

Hibernate:
    insert
    into
      products
    (category, name)
    values
      (?, ?)
```

There are a few interesting details in there. Notice that thanks to our property, which says **spring.jpa.hibernate.ddl-auto=create**, the Hibernate framework that allows us to work with the MySQL database drops the products table if it already exists.

Whenever you tag your class with **@Entity**, that is automatically considered to be mapped to a table, so the products table will be deleted and then recreated.



```
Product.java
1 package com.mytutorial.springbootrestapi.model;
2
3 import javax.persistence.Entity;
4
5 @Entity(name = "products")
6 public class Product {
7
8     @Id
9     @GeneratedValue(strategy = GenerationType.IDENTITY)
10    private Long id;
```

If you scroll down a bit, you'll see a create table statement that is executed on our MySQL database.

Notice that the columns of the table are id category and name, with id as the primary key. These correspond exactly to the member variables of our product class.

# Spring Boot Microservices: Building RESTful API Services

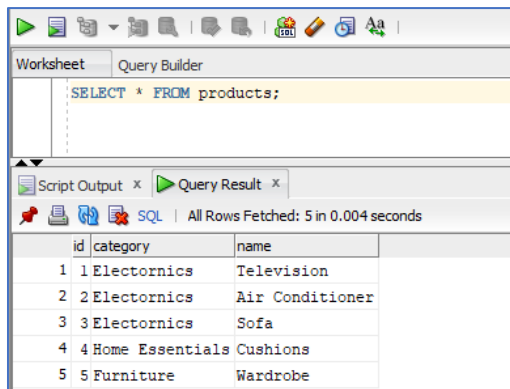
If you scroll further down the console window, you'll see a few insert statements, which add products to our Products table. You'll find five separate insert statements corresponding to the *productRepository.save* method invocation that we have within our Spring Boot application.

Remember, the Spring Boot application implements the command line runner. And in the run method, we save five products that we can work with when we run our app.

```
11 @SpringBootApplication
12 public class SpringbootrestapiApplication implements CommandLineRunner {
13
14     @Autowired
15     private ProductRepository productRepository;
16
17     public static void main(String[] args) {
18         SpringApplication.run(SpringbootrestapiApplication.class, args);
19     }
20
21     @Override
22     public void run(String... args) throws Exception {
23         productRepository.save(new Product("Television", "Electornics"));
24         productRepository.save(new Product("Air Conditioner", "Electornics"));
25         productRepository.save(new Product("Sofa", "Electornics"));
26         productRepository.save(new Product("Cushions", "Home Essentials"));
27         productRepository.save(new Product("Wardrobe", "Furniture"));
28     }
29 }
```

Each *productRepository.save* translates to an insert.

Once your application is running, you can go to MySQL Workbench and run a `SELECT * FROM products;`



The screenshot shows the MySQL Workbench interface. The Query Builder tab is active, displaying the query `SELECT * FROM products;`. Below the query, the Query Result tab shows the results of the query. The results are displayed in a table with 5 rows and 3 columns: id, category, and name.

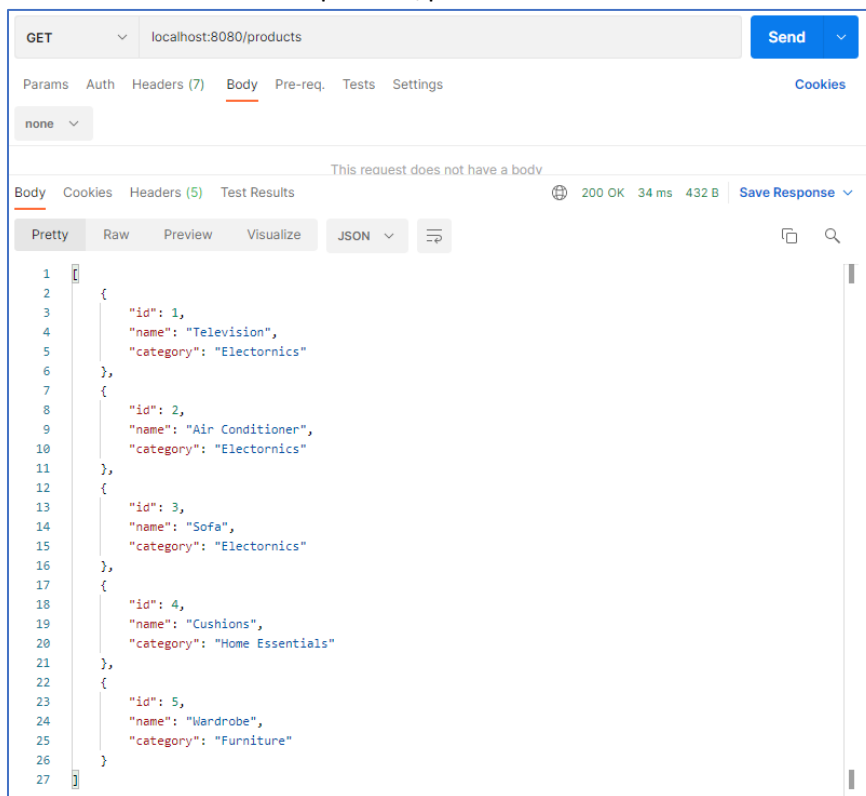
	id	category	name
1	1	Electornics	Television
2	2	Electornics	Air Conditioner
3	3	Electornics	Sofa
4	4	Home Essentials	Cushions
5	5	Furniture	Wardrobe

You can see the five products that we inserted from our Spring Boot application are now present in the products database table.

# Spring Boot Microservices: Building RESTful API Services

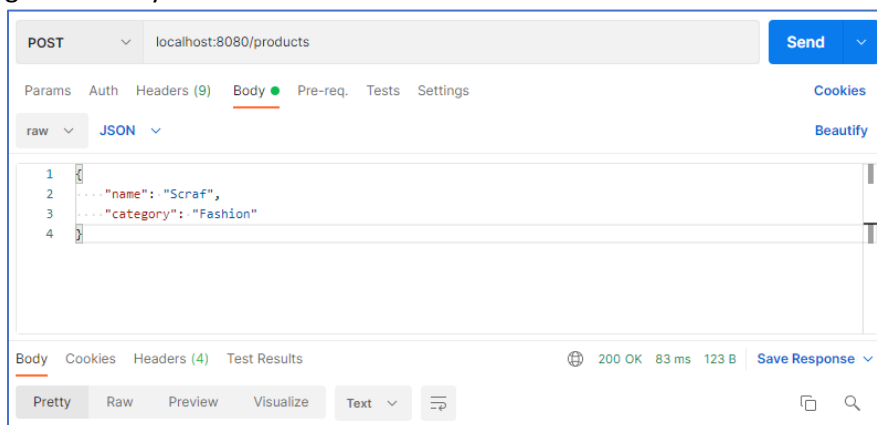
- *Test Insert Operation*

We'll now head over to the advanced REST client and make sure that our CRUD operations still work. Let's make a GET request to /products.



And if you scroll down to the bottom, here are the five products available in our products database table. They have been retrieved successfully.

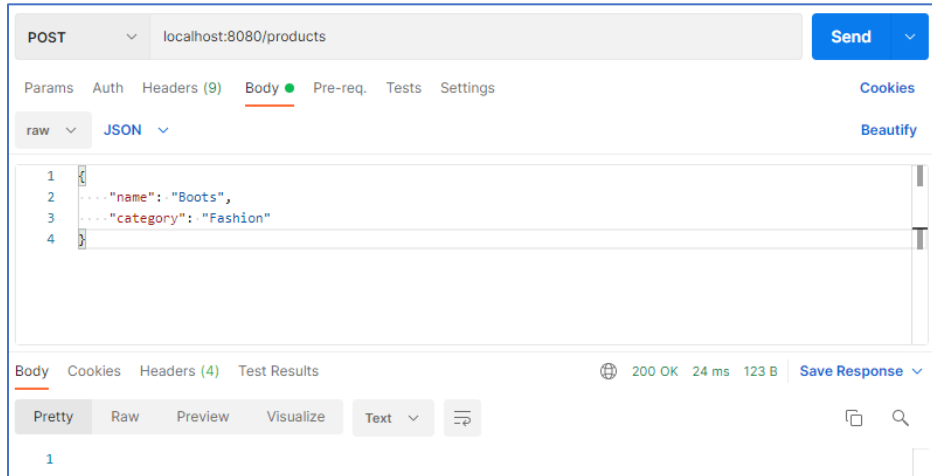
Let's try a create operation where we make a POST request to add new products to the table. The URL path is still /products. Remember, the POST request expects the new product details to be part of the request body. Select the BODY tab for our request. The Body content type we'll set as application/json. We now need to specify the product details in the form of JSON. Only the name and category, the ID that is the primary key of the product will be automatically generated by Hibernate.



# Spring Boot Microservices: Building RESTful API Services

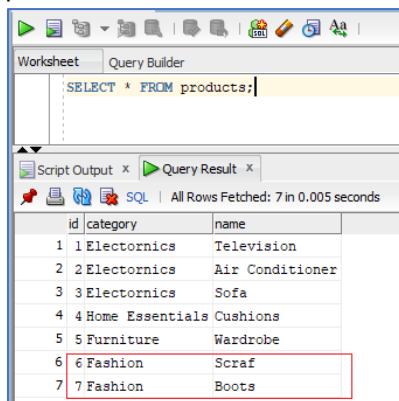
Hit SEND to perform this POST operation on our API service. If you get a **200** response, everything is fine and that's exactly what we get, as you can see from the bottom of the screen.

Let's add another product to our database table, this time we'll add Boots. Once again, go ahead and hit the SEND button in order to make a POST request to the server.



The path is still `/products`. If you get **200 OK**, this product has also been added to your database table and we can confirm this by heading over to MySQL workbench.

Then you do a `SELECT * FROM products`, notice that our database table now has two new product records.



With ids 6 and 7, we have Scarf and Boots. We only specified the name and category of the products. The ID was auto-generated thanks to the **@GeneratedValue** annotation on the `Id` field of the product class.



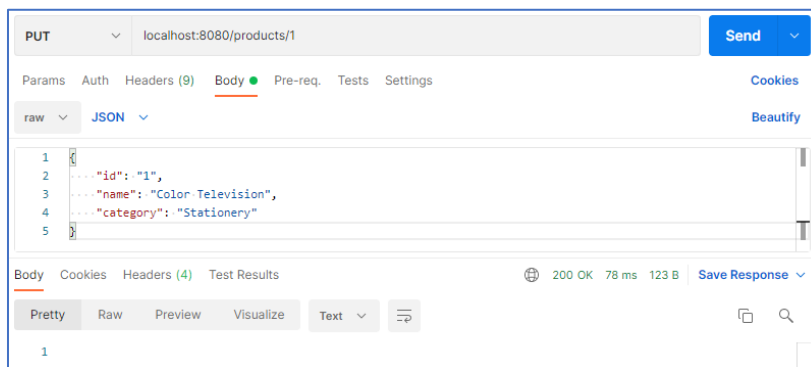
# Spring Boot Microservices: Building RESTful API Services

- *Test Update Operation*

Back to our advanced REST client here where we perform a *PUT* operation to update an existing product in our database.

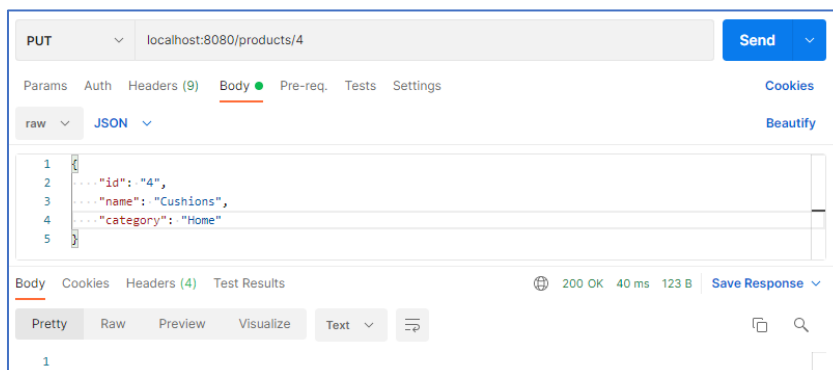
Notice the path of the URL that we're about to hit the product ID is part of the path. Our Update operation will be on the product with ID 1.

The updated product details need to be specified as a part of the request body, go to the BODY tab and specify the content type as application/json. While updating the product in the request body, make sure you specify the id of the product as well. The product with id 1 we change the name to Color Television, it's still part of the Electronics category.



In order to execute this Update operation for product id 1, I'll hit the SEND button, and when I get the **200 OK** response, I know my update has been successful.

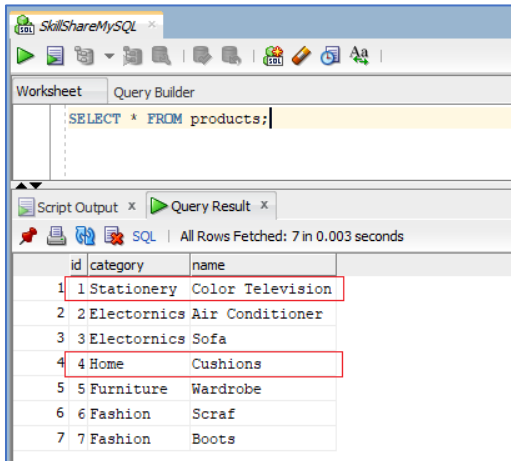
Let's perform another update before we take a look at the database. This time we'll update the product with id 4. Make sure you specify 4 as a part of the URL path, and let's now specify the request body, which contains the updated details of our product. The updated details include the name Cushions in the category Home.



Let's now hit the SEND button in order to send this PUT request to perform the product update. I get 200 OK, everything seems good, but this time, I'll verify using the MySQL Workbench.

I'll run a `SELECT * FROM products`, notice that Cushions is now part of category Home, that is a product with id 4. The name of the product with id 1 is Color Television, both of our updates have been applied successfully.

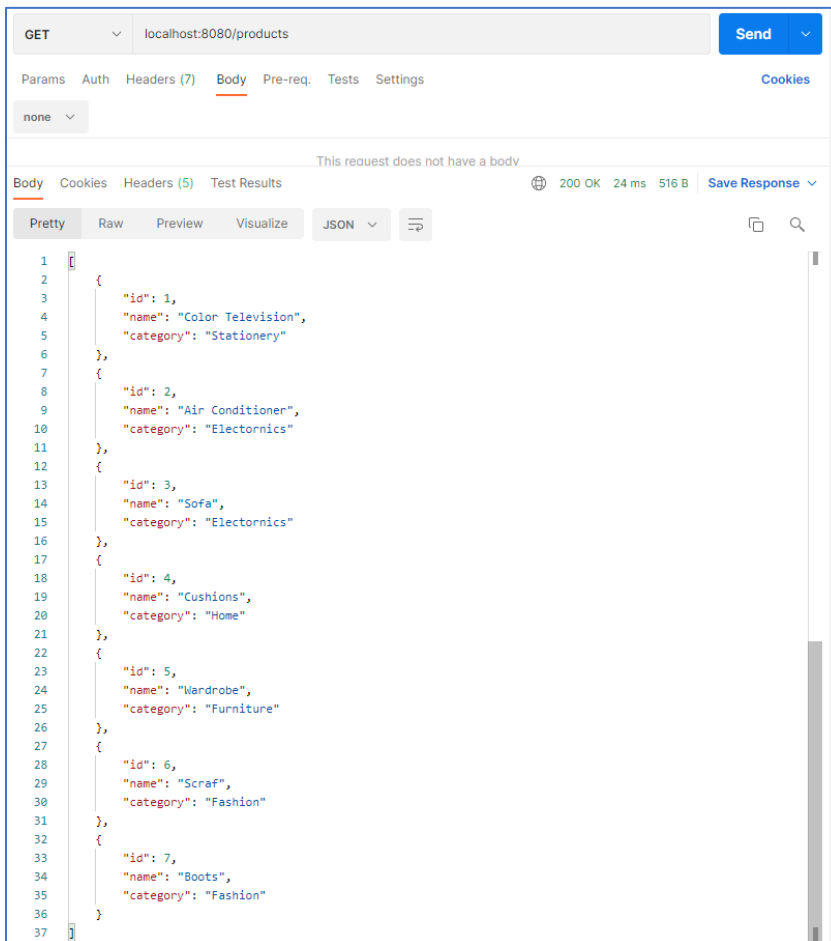
# Spring Boot Microservices: Building RESTful API Services



The screenshot shows a web-based SQL client interface. The 'Query Builder' tab is active, displaying the query `SELECT * FROM products;`. Below the query, the 'Query Result' tab shows a table with 7 rows. The columns are 'id', 'category', and 'name'. The data is as follows:

	id	category	name
1	1	Stationery	Color Television
2	2	Electronics	Air Conditioner
3	3	Electronics	Sofa
4	4	Home	Cushions
5	5	Furniture	Wardrobe
6	6	Fashion	Scraf
7	7	Fashion	Boots

Back to the advanced REST client, this time we perform a GET operation to ensure that the products retrieved from the database have the right updated values. If you scroll down and take a look at the list of products, all of our updates and additions are visible here.



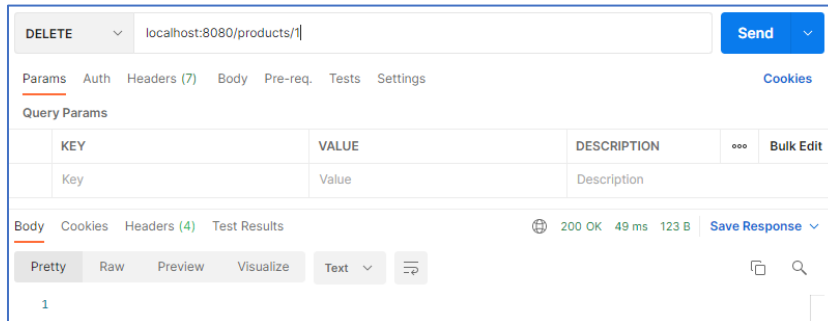
The screenshot shows an advanced REST client interface. The 'Send' button is visible. The request is a GET operation to `localhost:8080/products`. The response is a JSON array of 7 objects, each representing a product. The response status is 200 OK, 24 ms, 516 B. The response body is as follows:

```
1 {
2   {
3     "id": 1,
4     "name": "Color Television",
5     "category": "Stationery"
6   },
7   {
8     "id": 2,
9     "name": "Air Conditioner",
10    "category": "Electronics"
11  },
12  {
13    "id": 3,
14    "name": "Sofa",
15    "category": "Electronics"
16  },
17  {
18    "id": 4,
19    "name": "Cushions",
20    "category": "Home"
21  },
22  {
23    "id": 5,
24    "name": "Wardrobe",
25    "category": "Furniture"
26  },
27  {
28    "id": 6,
29    "name": "Scraf",
30    "category": "Fashion"
31  },
32  {
33    "id": 7,
34    "name": "Boots",
35    "category": "Fashion"
36  }
37 }
```

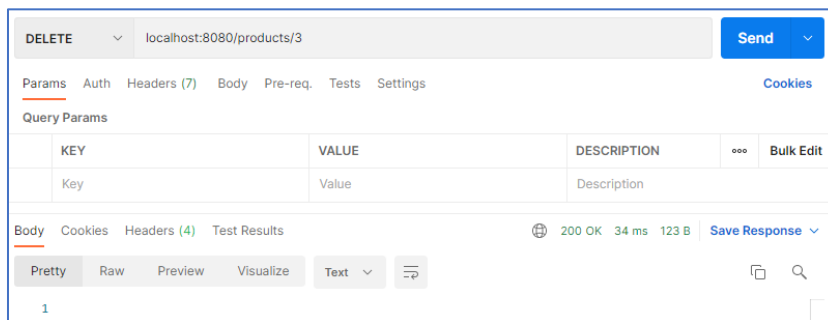
# Spring Boot Microservices: Building RESTful API Services

- *Test Delete Operation*

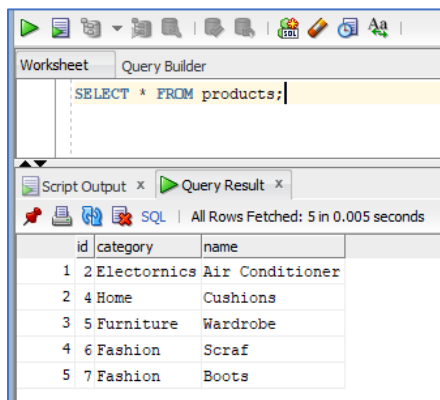
Everything looks good so far, we have one last operation left to test and that is the DELETE operation. The http method is DELETE, the URL includes the ID of the product that we want to delete.



This time we'll delete the product with ID 1. Hit SEND and the **200 OK** tells us the deletion was successful. Let's perform one more DELETE, this time I'll delete the product with ID 3.



Make sure you specify 3 as a part of your URL, hit SEND, and the **200 OK** tells me this deletion was successful as well. The deletion should be visible in your MySQL database. A `SELECT * FROM products` tells us that the product with id 1 and the product with id 3 are no longer present in our database table.

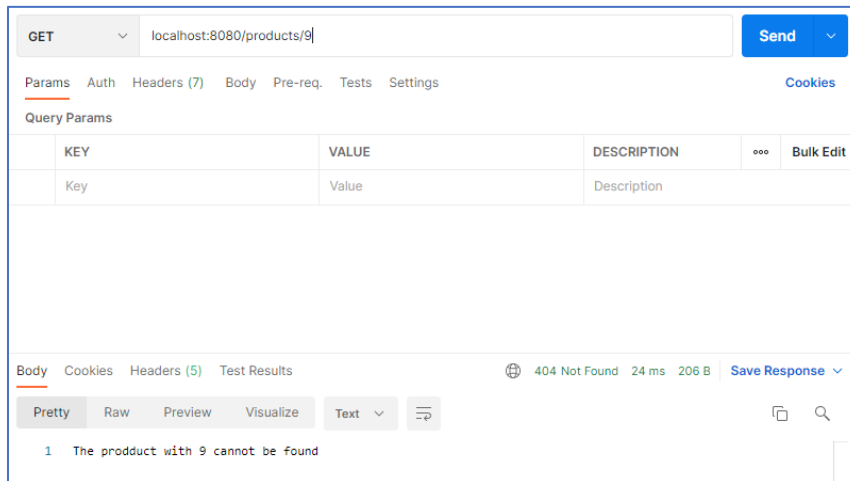


# Spring Boot Microservices: Building RESTful API Services

- *Test Error Handling*

One last thing to test in our REST API is our error handling mechanism. I'm going to try and retrieve the product with id 9, this is not a product that is present in our database table.

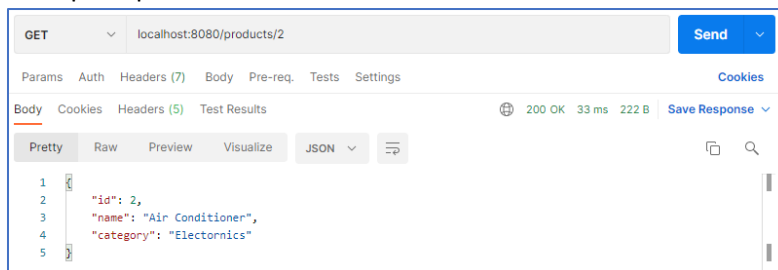
Notice that we get a 404 status, and we get a message which says the product with 9 cannot be found. This is the error message from the *product exception class*.



```
18 @RestController
19 public class ProductController {
20
21     @Autowired
22     private ProductService productService;
23
24     public List<Product> getAllProducts() {}
25
26     @GetMapping("/products/{pId}")
27     public Product getProduct(@PathVariable("pId") Long id) {
28         return productService.getProduct(id).orElseThrow(() -> new ProductNotFoundException(id));
29     }
30 }
31
32
33
```

```
3 public class ProductNotFoundException extends RuntimeException {
4
5     private static final long serialVersionUID = 1L;
6
7     public ProductNotFoundException(Long id) {
8         super(String.format("The prodduct with %d cannot be found", id));
9     }
10
11 }
```

Example if product exists :



# Spring Boot Microservices: Building RESTful API Services

## Performing CRUD operation via Web UI

Now that we've integrated our REST API with the SQL database, we're ready to take on the next step.

We'll place a nice web UI in front of our REST service. So the create, read, update, and delete operations we'll perform using the web interface, rather than the advanced REST client.

### 1. Update Dependency in pom.xml

If you're going to build a web user interface, we need a template engine which means within pom.xml I'll add in an additional starter dependency **spring-boot-starter-thymeleaf**. Thymeleaf is a template engine that we'll use to set up our UI.

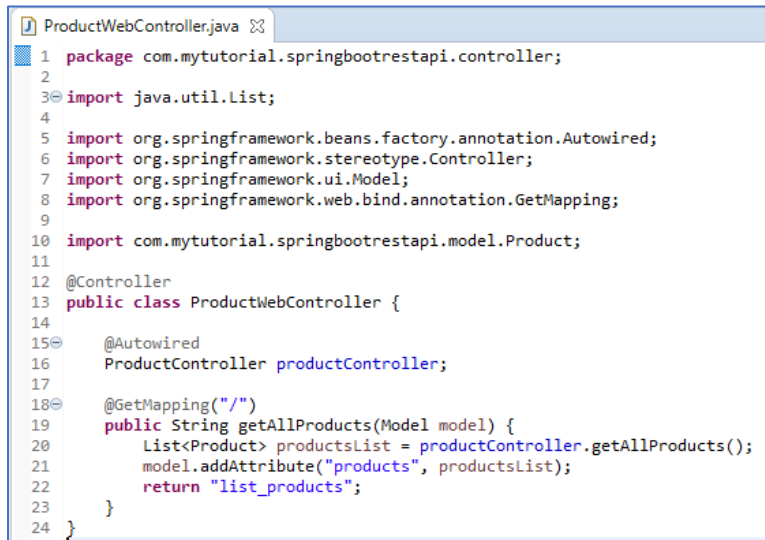


```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
3   xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 https://maven.apache.org/xsd/maven-4.0.0.xsd">
4   <modelVersion>4.0.0</modelVersion>
5   <parent>
6     <groupId>org.springframework.boot</groupId>
7     <artifactId>spring-boot-starter-parent</artifactId>
8     <version>2.5.7</version>
9     <relativePath/> <!-- lookup parent from repository -->
10  </parent>
11  <groupId>com.mytutorial</groupId>
12  <artifactId>springbootrestapi</artifactId>
13  <version>0.0.1-SNAPSHOT</version>
14  <name>springbootrestapi</name>
15  <description>Demo project for Spring Boot RestAPI</description>
16  <properties>
17    <java.version>1.8</java.version>
18  </properties>
19  <dependencies>
20
21    <dependency>
22      <groupId>org.springframework.boot</groupId>
23      <artifactId>spring-boot-starter-data-jpa</artifactId>
24    </dependency>
25
26    <dependency>
27      <groupId>org.springframework.boot</groupId>
28      <artifactId>spring-boot-starter-web</artifactId>
29    </dependency>
30
31    <dependency>
32      <groupId>mysql</groupId>
33      <artifactId>mysql-connector-java</artifactId>
34      <scope>runtime</scope>
35    </dependency>
36
37    <dependency>
38      <groupId>org.springframework.boot</groupId>
39      <artifactId>spring-boot-starter-thymeleaf</artifactId>
40    </dependency>
41
42    <dependency>
43      <groupId>org.springframework.boot</groupId>
44      <artifactId>spring-boot-starter-test</artifactId>
45      <scope>test</scope>
46    </dependency>
47  </dependencies>
48
49  <build>
50    <plugins>
51      <plugin>
52        <groupId>org.springframework.boot</groupId>
53        <artifactId>spring-boot-maven-plugin</artifactId>
54      </plugin>
55    </plugins>
56  </build>
57
58 </project>
```

# Spring Boot Microservices: Building RESTful API Services

## 2. Create Web Controller Class

Another change that we'll make is to add an additional controller class. We already have the `ProductController`, which basically is the controller that exposes our REST API. I've added another controller here called product web controller. This **ProductWebController** will handle incoming web request from the user interface that we set up.



```
1 package com.mytutorial.springbootrestapi.controller;
2
3 import java.util.List;
4
5 import org.springframework.beans.factory.annotation.Autowired;
6 import org.springframework.stereotype.Controller;
7 import org.springframework.ui.Model;
8 import org.springframework.web.bind.annotation.GetMapping;
9
10 import com.mytutorial.springbootrestapi.model.Product;
11
12 @Controller
13 public class ProductWebController {
14
15     @Autowired
16     ProductController productController;
17
18     @GetMapping("/")
19     public String getAllProducts(Model model) {
20         List<Product> productsList = productController.getAllProducts();
21         model.addAttribute("products", productsList);
22         return "list_products";
23     }
24 }
```

Notice that the `ProductWebController` is tagged using the **@Controller** annotation, indicating that the response from our handler method should be interpreted as logical views, rather than as web responses directly.

The `ProductWebController` will not actually query the database directly. If we did that, that's just duplication of code. Instead, we'll reference the `ProductController` and use the `ProductController` to create, read, update, and delete our database.

On line 16, you can see a reference to the `ProductController`, which is injected automatically by spring thanks to our use of the **@Autowired** annotation. The only method that we've implemented within this product web controller is a `get` it's annotated using **@GetMapping("/")**

This is to the path `/`, all we do is get a list of products. And return this list of products to the client. To get the list of products we call `productController.getAllProducts`. And then we add this as an attribute on the model. And we'll render the `list_products.html` file.

# Spring Boot Microservices: Building RESTful API Services

## 3. Create View HTML Page

Let's take a look at this **list\_products.html**, which is some new code that we have within this app. You can see it's an HTML file which uses **Thymeleaf** to render data. You can see a reference to **Thymeleaf** ("<http://www.thymeleaf.org>") on line 2.

```
list_products.html
1 <!DOCTYPE html>
2 <html xmlns:th="http://www.thymeleaf.org">
3 <head>
4   <meta charset="utf-8">
5   <meta http-equiv="x-ua-compatible" content="ie=edge">
6   <title>Products</title>
7   <meta name="viewport" content="width=device-width, initial-scale=1">
8   <link rel="stylesheet" href="https://stackpath.bootstrapcdn.com/bootstrap/4.1.3/css/bootstrap.min.css">
9   <link rel="stylesheet" href="https://use.fontawesome.com/releases/v5.4.1/css/all.css">
10 </head>
11 <h2 align="center">List of Products</h2>
12 <body>
13   <div class="container my-2">
14     <div class="card">
15       <div class="card-body">
16         <div th:switch="${products}" class="container my-4">
17           <p class="my-5">
18             <a class="btn btn-primary">Add Product</a>
19           </p>
20
21           <div class="col-md-10">
22             <h2 th:case="null">No product found!</h2>
23             <div th:case="*">
24               <table class="table table-striped table-responsive-md">
25                 <thead>
26                   <tr>
27                     <th>Product ID</th>
28                     <th>Product Name</th>
29                     <th>Product Category</th>
30                     <th></th>
31                     <th></th>
32                   </tr>
33                 </thead>
34                 <tbody>
35                   <tr th:each="product : ${products}">
36                     <td th:text="${product.id}"></td>
37                     <td th:text="${product.name}"></td>
38                     <td th:text="${product.category}"></td>
39                     <td><a class="btn btn-primary"> Edit</a></td>
40                     <td><a class="btn btn-danger"> Delete</a></td>
41                   </tr>
42                 </tbody>
43               </table>
44             </div>
45           </div>
46         </div>
47       </div>
48     </div>
49   </div>
50 </body>
51 </html>
```

We also use CSS from bootstrap and fontawesome to get the right look and feel for our products display.

The references to these style sheets are

["https://stackpath.bootstrapcdn.com/bootstrap/4.1.3/css/bootstrap.min.css"](https://stackpath.bootstrapcdn.com/bootstrap/4.1.3/css/bootstrap.min.css)

["https://use.fontawesome.com/releases/v5.4.1/css/all.css"](https://use.fontawesome.com/releases/v5.4.1/css/all.css)

When the list\_products view is rendered, the list of products is available in the products variable on following line.

```
<div th:switch="${products}" class="container my-4">
```

You can see that we have a **Thymeleaf** switch statement on this products variables on following line, we check for the case that products is null.

```
<h2 th:case="null">No product found!</h2>
```

# Spring Boot Microservices: Building RESTful API Services

In that case we will display No product found!

On next line.

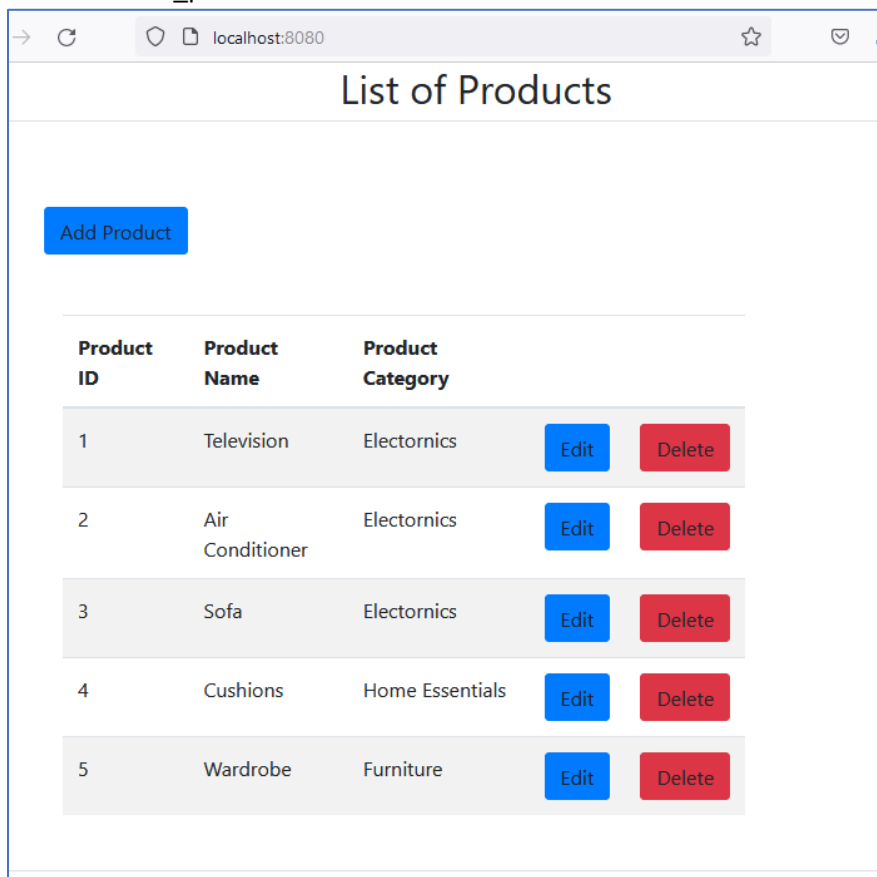
```
<div th:case="*">
```

We check for all other cases, case=\*. In that case, we'll simply display the list of products within a nicely formatted table. The header of this table gives us the thead element. And within the tbody we run a for each loop th:each on line.

```
<tr th:each="product : ${products}">
```

For every product we'll display the id, name and category of the product. Each row will contain buttons to edit and delete the product. We haven't wired up those buttons yet.

Let's run this code and take a look at our UI go to localhost:8080. Just / is enough and this will render the list\_products.html file.



Product ID	Product Name	Product Category		
1	Television	Electornics	Edit	Delete
2	Air Conditioner	Electornics	Edit	Delete
3	Sofa	Electornics	Edit	Delete
4	Cushions	Home Essentials	Edit	Delete
5	Wardrobe	Furniture	Edit	Delete

Complete with all of the products that we've queried from our underlying database. You have the Add Product button, the Edit and Delete product buttons. None of them are wired up yet.



# Spring Boot Microservices: Building RESTful API Services

## 4. Wired Up the UI Operation

Let's head back to our code on to **ProductWebController**. Down here at the very bottom. I've added some additional code in order to add a new product using our UI.

```
ProductWebController.java
1 package com.mytutorial.springbootrestapi.controller;
2
3 import java.util.List;
4
5 import org.springframework.beans.factory.annotation.Autowired;
6 import org.springframework.stereotype.Controller;
7 import org.springframework.ui.Model;
8 import org.springframework.web.bind.annotation.GetMapping;
9 import org.springframework.web.bind.annotation.ModelAttribute;
10 import org.springframework.web.bind.annotation.PostMapping;
11
12 import com.mytutorial.springbootrestapi.model.Product;
13
14 @Controller
15 public class ProductWebController {
16
17     @Autowired
18     ProductController productController;
19
20     @GetMapping("/")
21     public String getAllProducts(Model model) {
22         List<Product> productsList = productController.getAllProducts();
23         model.addAttribute("products", productsList);
24         return "list_products";
25     }
26
27     @GetMapping("/new_product")
28     public String addProduct(Model model) {
29         model.addAttribute("product", new Product());
30         return "new_product";
31     }
32
33     @PostMapping("/save_new")
34     public String saveNewProduct(@ModelAttribute("product") Product product) {
35         productController.addProduct(product);
36         return "redirect:/";
37     }
38 }
```

The first add product method here is annotated using **@GetMapping("/new\_product")** the path is new product. This will render a new page where we can fill in the details of the product that we want added. Product details will be added using a form, so we instantiate a new product object on line 29. And we add this product object to the model and render the *new\_product* page. This product instance will be the model object that is bound to our form user interface.

We've also added the Save new product method which accepts as an input argument, a product instance where the details of the product have been filled in via the form on the front end.

This is tagged using **@ModelAttribute** indicating this is the instance bound to our form. We then call the *ProductController* and add this new product and redirect to *"/*. *"/* is the URL, where we can see the list of products.

The **@PostMapping** annotation tells us that this method will be invoked when a POST request is made to the path */save\_new*.

# Spring Boot Microservices: Building RESTful API Services

## 5. Create Additional View HTML Page

In order to wire up the Add Product button. We need to make a slight change to our **list\_products.html** file.

```
list_products.html
1 <!DOCTYPE html>
2 <html xmlns: th="http://www.thymeleaf.org">
3 <head>
4   <meta charset="utf-8">
5   <meta http-equiv="x-ua-compatible" content="ie=edge">
6   <title>Products</title>
7   <meta name="viewport" content="width=device-width, initial-scale=1">
8   <link rel="stylesheet" href="https://stackpath.bootstrapcdn.com/bootstrap/4.1.3/css/bootstrap.min.css">
9   <link rel="stylesheet" href="https://use.fontawesome.com/releases/v5.4.1/css/all.css">
10 </head>
11 <h2 align="center">List of Products</h2>
12 <body>
13   <div class="container my-2">
14     <div class="card">
15       <div class="card-body">
16         <div th:switch="${products}" class="container my-4">
17           <p class="my-5">
18             <a href="/new_product" class="btn btn-primary">Add Product</a>
19           </p>
20         </div>
21         <div class="col-md-10">
22           <h2 th:case="null">No product found!</h2>
23           <div th:case="*">
24             <table class="table table-striped table-responsive-md">
25               <thead>
26                 <tr>
27                   <th>Product ID</th>
28                   <th>Product Name</th>
29                   <th>Product Category</th>
30                   <th></th>
31                 </tr>
32               </thead>
33             </table>
34             <tbody>
35               <tr th:each="product : ${products}">
36                 <td th:text="${product.id}"></td>
37                 <td th:text="${product.name}"></td>
38                 <td th:text="${product.category}"></td>
39                 <td><a class="btn btn-primary"> Edit</a></td>
40                 <td><a class="btn btn-danger"> Delete</a></td>
41               </tr>
42             </tbody>
43           </table>
44         </div>
45       </div>
46     </div>
47   </div>
48 </div>
49 </div>
50 </body>
51 </html>
```

The change here is on line 18 where we've wired up the add button to the path `/new_product`.

# Spring Boot Microservices: Building RESTful API Services

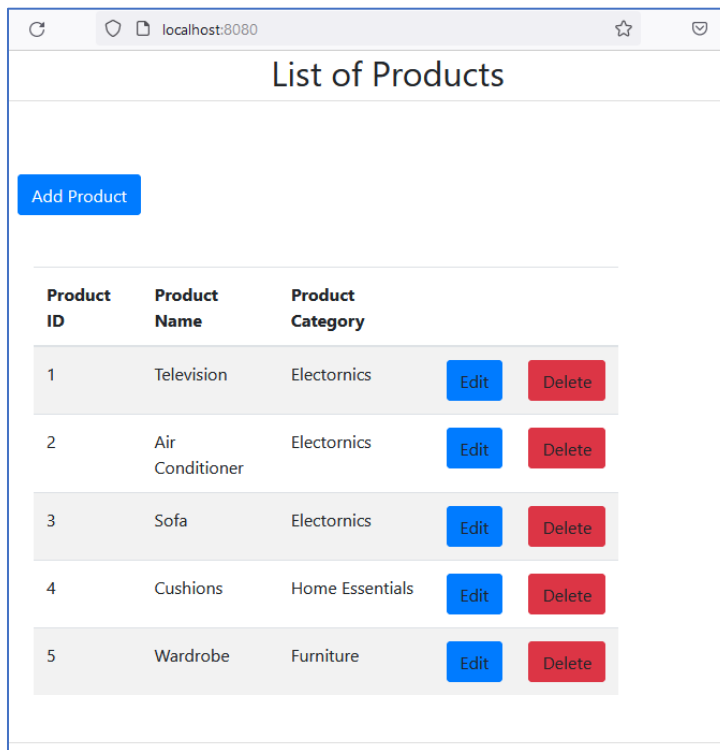
Clicking on this button will make a get request to the new product path and that will take us to the **new\_product.html** file.

```
new_product.html
1 <!DOCTYPE html>
2 <html xmlns: th="http://www.thymeleaf.org">
3 <head>
4 <meta charset="utf-8">
5 <title>Add New Product</title>
6 <style type="text/css">
7     .button {
8         border: none;
9         color: white;
10        padding: 10px 130px;
11        text-align: center;
12        font-size: 16px;
13    }
14    .button1 {
15        background-color: #2790F3;
16    }
17    table, th, td {
18        padding: 10px;
19    }
20 </style>
21 <meta name="viewport" content="width=device-width, initial-scale=1">
22 <link rel="stylesheet" href="https://stackpath.bootstrapcdn.com/bootstrap/4.1.3/css/bootstrap.min.css">
23 <link rel="stylesheet" href="https://use.fontawesome.com/releases/v5.4.1/css/all.css">
24 </head>
25 <body>
26 <div align="center">
27     <h2>Add New Product</h2>
28     <br/>
29     <form action="#" th:action="@{/save_new}" th:object="${product}" method="post">
30         <table>
31             <tr>
32                 <td>Name: </td>
33                 <td><input type="text" th:field="*{name}" /></td>
34             </tr>
35             <tr>
36                 <td>Category: </td>
37                 <td><input type="text" th:field="*{category}" /></td>
38             </tr>
39             <tr>
40                 <td colspan="2"><button class="button button1 btn" type="submit">Save</button>
41             </tr>
42         </table>
43     </form>
44 </div>
45 </body>
46 </html>
```

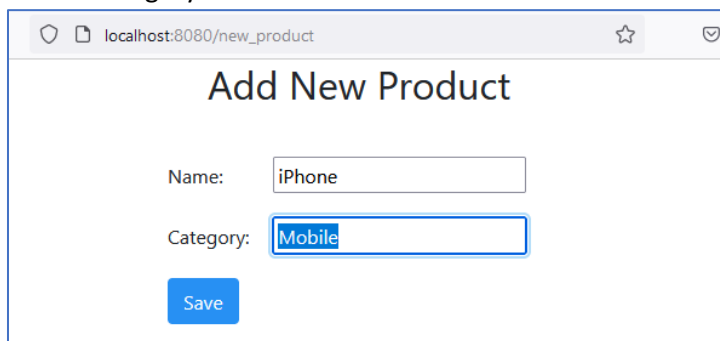
Within this file, we display a form to the user, the **Thymeleaf** action that this form is wired up to is `/save_new`. The **Thymeleaf** object that this form is bound to is the product instance that we instantiated on the server. We have a table here where the user types in the name and category of the product and a submit button a Save button.

# Spring Boot Microservices: Building RESTful API Services

Let's run our application and head over to localhost 8080 and try adding a new product. Here is the current list of products in our database, click on the Add Product button, and that makes a get request to the path /new\_product that renders this page.



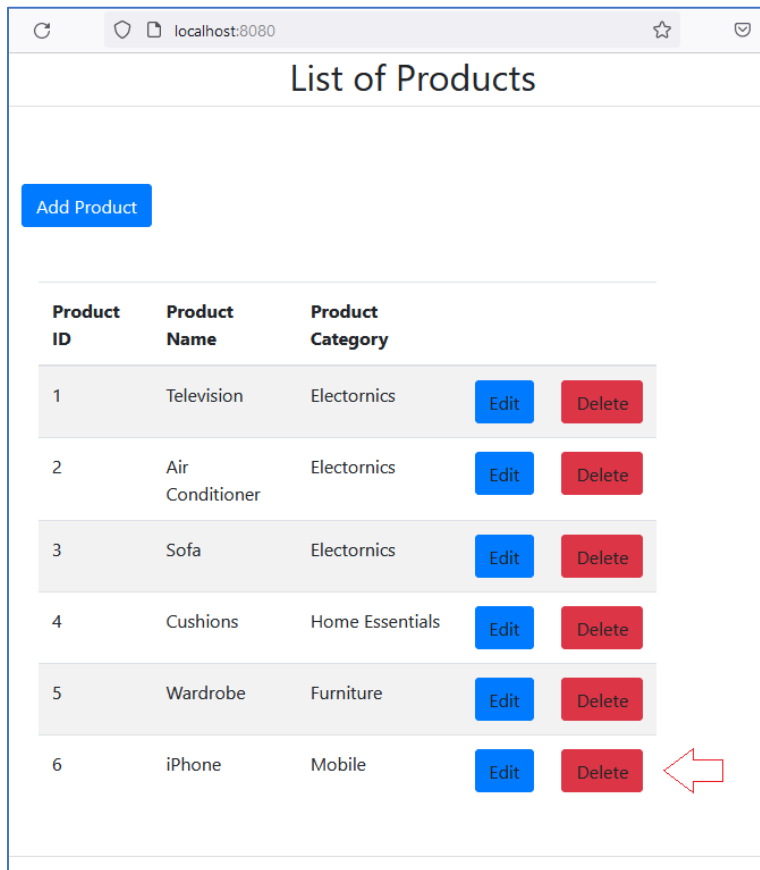
Let's type in the new product that we want to add to our database. This is an *iPhone* in the *Mobile* category.



Clicking on the Save button, will save the product to the database at the back end.

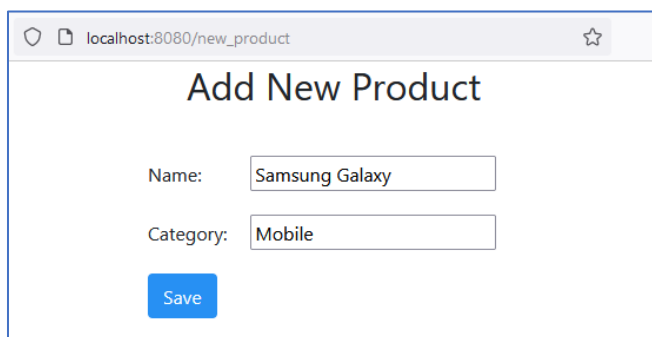
# Spring Boot Microservices: Building RESTful API Services

And when we redirect to the list products view you can see that the iPhone with the Mobiles category is present here at the very end with ID 6.



Product ID	Product Name	Product Category	Edit	Delete
1	Television	Electornics	Edit	Delete
2	Air Conditioner	Electornics	Edit	Delete
3	Sofa	Electornics	Edit	Delete
4	Cushions	Home Essentials	Edit	Delete
5	Wardrobe	Furniture	Edit	Delete
6	iPhone	Mobile	Edit	Delete

Let's add one more product here we are in the new product page. The product name is *Samsung Galaxy* the category is *Mobile* Click on Save.



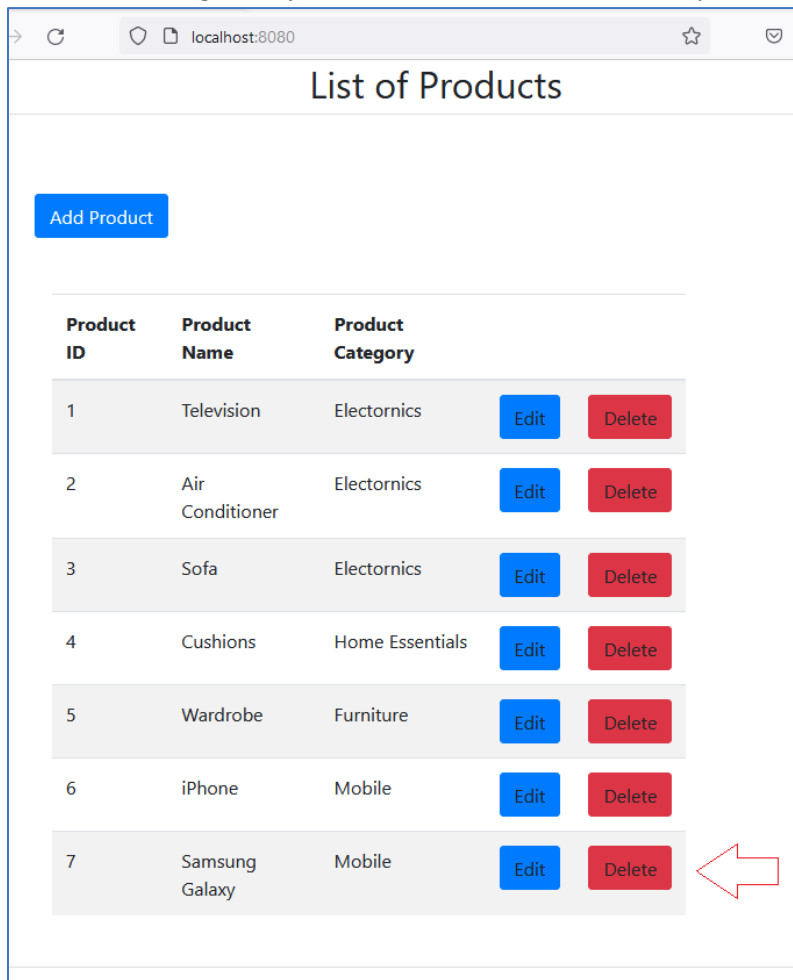
**Add New Product**

Name:

Category:

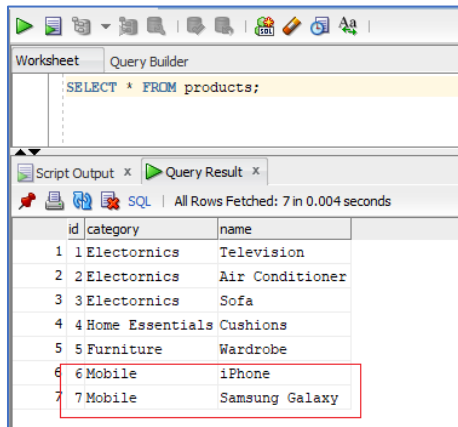
# Spring Boot Microservices: Building RESTful API Services

And the *Samsung Galaxy* has been added with ID 7 to our products list here.



Product ID	Product Name	Product Category		
1	Television	Electornics	Edit	Delete
2	Air Conditioner	Electornics	Edit	Delete
3	Sofa	Electornics	Edit	Delete
4	Cushions	Home Essentials	Edit	Delete
5	Wardrobe	Furniture	Edit	Delete
6	iPhone	Mobile	Edit	Delete
7	Samsung Galaxy	Mobile	Edit	Delete

Let's verify this using MySQL Workbench. I do a `SELECT * FROM products` and at the very end the two mobile phones that we added to our database are present.



id	category	name
1	Electornics	Television
2	Electornics	Air Conditioner
3	Electornics	Sofa
4	Home Essentials	Cushions
5	Furniture	Wardrobe
6	Mobile	iPhone
7	Mobile	Samsung Galaxy

# Spring Boot Microservices: Building RESTful API Services

## 6. Updating Records Using a Web UI

Now we'll wire up the update and delete buttons in our web interface, perform update and delete operations. First, let's start with update. Here within the **ProductWebController**, if you scroll down to the very bottom, you'll see that I've added a few new methods in here. I have an *editForm* method here on line 40, which responds to get requests on the path `"/update_product/{pId}"`

```
ProductWebController.java
1 package com.mytutorial.springbootrestapi.controller;
2
3 import java.util.List;
4
5 import org.springframework.beans.factory.annotation.Autowired;
6 import org.springframework.stereotype.Controller;
7 import org.springframework.ui.Model;
8 import org.springframework.web.bind.annotation.GetMapping;
9 import org.springframework.web.bind.annotation.ModelAttribute;
10 import org.springframework.web.bind.annotation.PathVariable;
11 import org.springframework.web.bind.annotation.PostMapping;
12
13 import com.mytutorial.springbootrestapi.model.Product;
14
15 @Controller
16 public class ProductWebController {
17
18     @Autowired
19     ProductController productController;
20
21     @GetMapping("/")
22     public String getAllProducts(Model model) {
23         List<Product> productsList = productController.getAllProducts();
24         model.addAttribute("products", productsList);
25         return "list_products";
26     }
27
28     @GetMapping("/new_product")
29     public String addProduct(Model model) {
30         model.addAttribute("product", new Product());
31         return "new_product";
32     }
33
34     @PostMapping("/save_new")
35     public String saveNewProduct(@ModelAttribute("product") Product product) {
36         productController.addProduct(product);
37         return "redirect:/";
38     }
39
40     @GetMapping("/update_product/{pId}")
41     public String editForm(@PathVariable(name = "pId") Long id, Model model) {
42         model.addAttribute("product", productController.getProduct(id));
43         return "update_product";
44     }
45
46     @PostMapping("/save_update")
47     public String saveUpdateProduct(@ModelAttribute("product") Product product) {
48         productController.updateProduct(product, product.getId());
49         return "redirect:/";
50     }
51 }
```

The product ID here is a path element which has to be extracted using `@PathVariable(name = "pId")`. You can see we have the `Long id` input argument annotated using `@PathVariable` for **pId**. We also have the model as an input argument. We use **model.addAttribute** to add an instance of the product object retrieved from our database.

# Spring Boot Microservices: Building RESTful API Services

So we call `productController.getProduct` for this ID, add that to the model and render the `update_product` view. The `update_product` view will display a form which we'll use to edit the details of the product.

Now to actually update the product details, we have the `save_update` product method, which is annotated using a `@PostMapping` `PostMapping` to the path (`"/save_update"`).

The product instance with the updated details is injected in as an input argument to this method. We have the annotation **@ModelAttribute**, binding this product instance with the form where users can edit the product.

Once we call `productController.updateProduct` to update the details of this product. We simply redirect to `/`, which will list out the products that we have in our database.

So let's take a look at the changes that we need to make to our user interface. In **list\_products.html**, we'll wire up the edit button. Notice the edit button on line 40, which we wired up to the path `/update_product/pld`, and we get the product ID from `product.id`.

```

1 <!DOCTYPE html>
2 <html xmlns: th="http://www.thymeleaf.org">
3 <head>
4   <meta charset="utf-8">
5   <meta http-equiv="x-ua-compatible" content="ie=edge">
6   <title>Products</title>
7   <meta name="viewport" content="width=device-width, initial-scale=1">
8   <link rel="stylesheet" href="https://stackpath.bootstrapcdn.com/bootstrap/4.1.3/css/bootstrap.min.css">
9   <link rel="stylesheet" href="https://use.fontawesome.com/releases/v5.4.1/css/all.css">
10 </head>
11 <h2 align="center">List of Products</h2>
12 <body>
13   <div class="container my-2">
14     <div class="card">
15       <div class="card-body">
16         <div th:switch="${products}" class="container my-4">
17           <p class="my-5">
18             <a href="/new_product" class="btn btn-primary">Add Product</a>
19           </p>
20           <div class="col-md-10">
21             <h2 th:case="null">No product found!</h2>
22             <div th:case="*">
23               <table class="table table-striped table-responsive-md">
24                 <thead>
25                   <tr>
26                     <th>Product ID</th>
27                     <th>Product Name</th>
28                     <th>Product Category</th>
29                   </tr>
30                 </thead>
31                 <tbody>
32                   <tr th:each="product : ${products}">
33                     <td th:text="${product.id}"></td>
34                     <td th:text="${product.name}"></td>
35                     <td th:text="${product.category}"></td>
36                     <td><a th:href="@{/update_product/{pid}(pid=${product.id})}" class="btn btn-primary"> Edit</a></td>
37                     <td><a class="btn btn-danger"> Delete</a></td>
38                   </tr>
39                 </tbody>
40               </table>
41             </div>
42           </div>
43         </div>
44       </div>
45     </div>
46   </div>
47 </body>
48 </html>

```



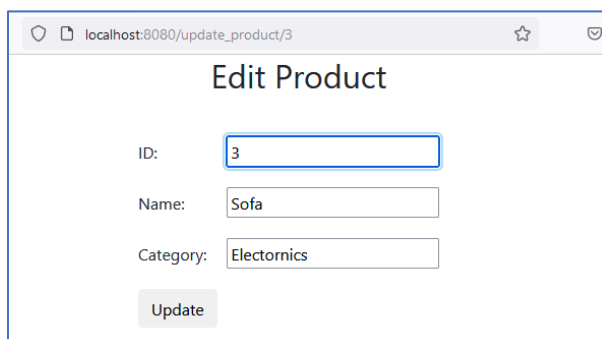
# Spring Boot Microservices: Building RESTful API Services

In addition to this change, we've also added a new HTML `update_product.html` file to display the product form for update. On line 23 the Thymeleaf action for this form is mapped to `/save_update`, a **post** request will be made to this path.

```
update_product.html
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 <meta name="viewport" content="width=device-width, initial-scale=1">
8 <link rel="stylesheet" href="https://stackpath.bootstrapcdn.com/bootstrap/4.1.3/css/bootstrap.min.css">
9 <link rel="stylesheet" href="https://use.fontawesome.com/releases/v5.4.1/css/all.css">
10 </head>
11 <style>
12     table, th, td {padding: 10px};
13 </style>
14 <body>
15 <div align="center">
16 <h2>Edit Product</h2>
17 <br/>
18 <form action="#" th:action="@{/save_update}" th:object="${product}" method="post">
19 <table>
20 <tr>
21 <td>ID:</td>
22 <td><input type="text" th:field="*{id}" readonly="readonly" /></td>
23 </tr>
24 <tr>
25 <td>Name:</td>
26 <td><input type="text" th:field="*{name}" /></td>
27 </tr>
28 <tr>
29 <td>Category:</td>
30 <td><input type="text" th:field="*{category}" /></td>
31 </tr>
32 <tr>
33 <td colspan="2"><button class="button button1 btn" type="submit">Update</button>
34 </tr>
35 </table>
36 </form>
37 </div>
38 </body>
39 </html>
```

The form is also bound to the product instance sent down from the server as you can see from the **th:object** attribute. On line 21 you can see that the *id* field of this form is *read only*. You cannot update the id for a product, you can update the name and category for the product.

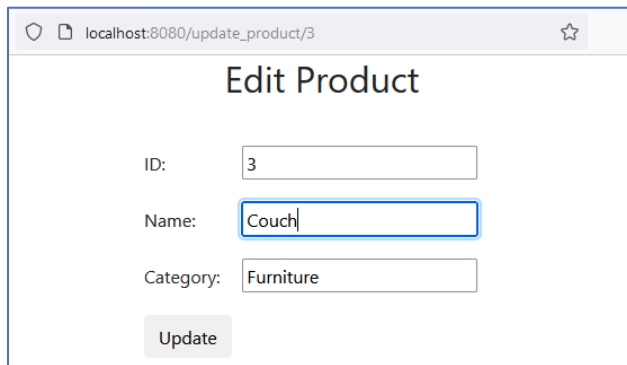
Now let's run this code and head over to localhost:8080, where we'll see the list of products in our database. We've started off with the five products that are added during initialization. Let's edit the product with ID 3.



The screenshot shows a web browser window with the address bar displaying 'localhost:8080/update\_product/3'. The page title is 'Edit Product'. The form contains three input fields: 'ID' with the value '3', 'Name' with the value 'Sofa', and 'Category' with the value 'Electronics'. Below the input fields is an 'Update' button.

# Spring Boot Microservices: Building RESTful API Services

The Edit button will bring up this edit product page. Let's change the name of the product. You won't be able to edit the ID even if you want to. Click on the Update button.



localhost:8080/update\_product/3

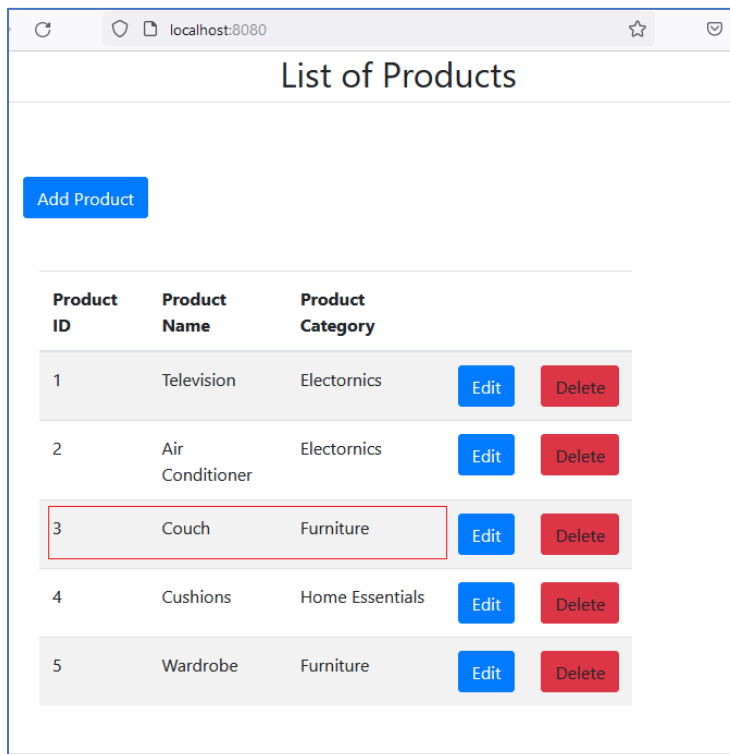
## Edit Product

ID:

Name:

Category:

And once the update is performed, you can see on this list of products that the product with ID 3 now has the name Couch.



localhost:8080

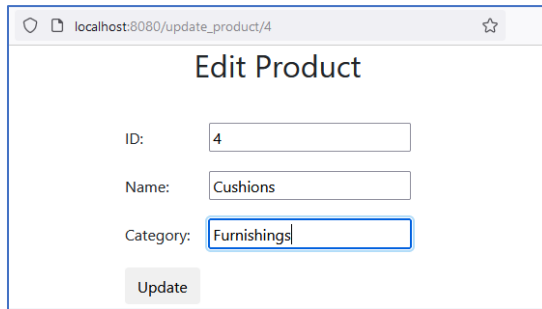
## List of Products

Product ID	Product Name	Product Category		
1	Television	Electornics	<input type="button" value="Edit"/>	<input type="button" value="Delete"/>
2	Air Conditioner	Electornics	<input type="button" value="Edit"/>	<input type="button" value="Delete"/>
3	Couch	Furniture	<input type="button" value="Edit"/>	<input type="button" value="Delete"/>
4	Cushions	Home Essentials	<input type="button" value="Edit"/>	<input type="button" value="Delete"/>
5	Wardrobe	Furniture	<input type="button" value="Edit"/>	<input type="button" value="Delete"/>

Let's perform another edit, this time we'll edit the product with ID 4, Cushions. Here's the page to edit a product, map to the URL path `/update_product/4`.

# Spring Boot Microservices: Building RESTful API Services

Let's change the category to Furnishings and click on the Update button.



localhost:8080/update\_product/4

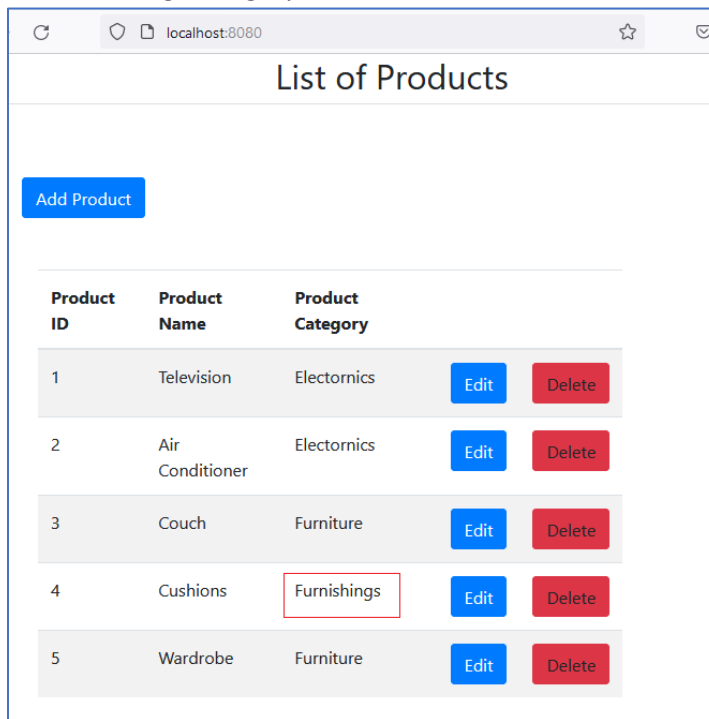
### Edit Product

ID:

Name:

Category:

And when you render the list of products, you can see that the product with ID 4 now belongs to the Furnishings category.

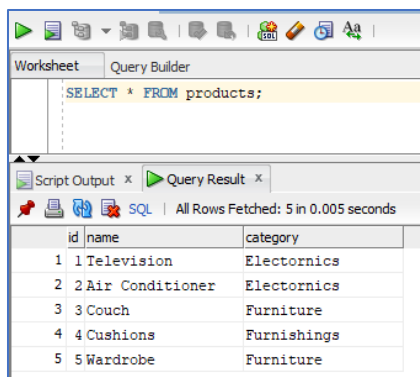


localhost:8080

### List of Products

Product ID	Product Name	Product Category		
1	Television	Electronics	<input type="button" value="Edit"/>	<input type="button" value="Delete"/>
2	Air Conditioner	Electronics	<input type="button" value="Edit"/>	<input type="button" value="Delete"/>
3	Couch	Furniture	<input type="button" value="Edit"/>	<input type="button" value="Delete"/>
4	Cushions	Furnishings	<input type="button" value="Edit"/>	<input type="button" value="Delete"/>
5	Wardrobe	Furniture	<input type="button" value="Edit"/>	<input type="button" value="Delete"/>

We'll now confirm that these updates are reflected in our MySQL database. `SELECT * FROM products;`



Worksheet Query Builder

```
SELECT * FROM products;
```

Script Output x Query Result x

SQL | All Rows Fetched: 5 in 0.005 seconds

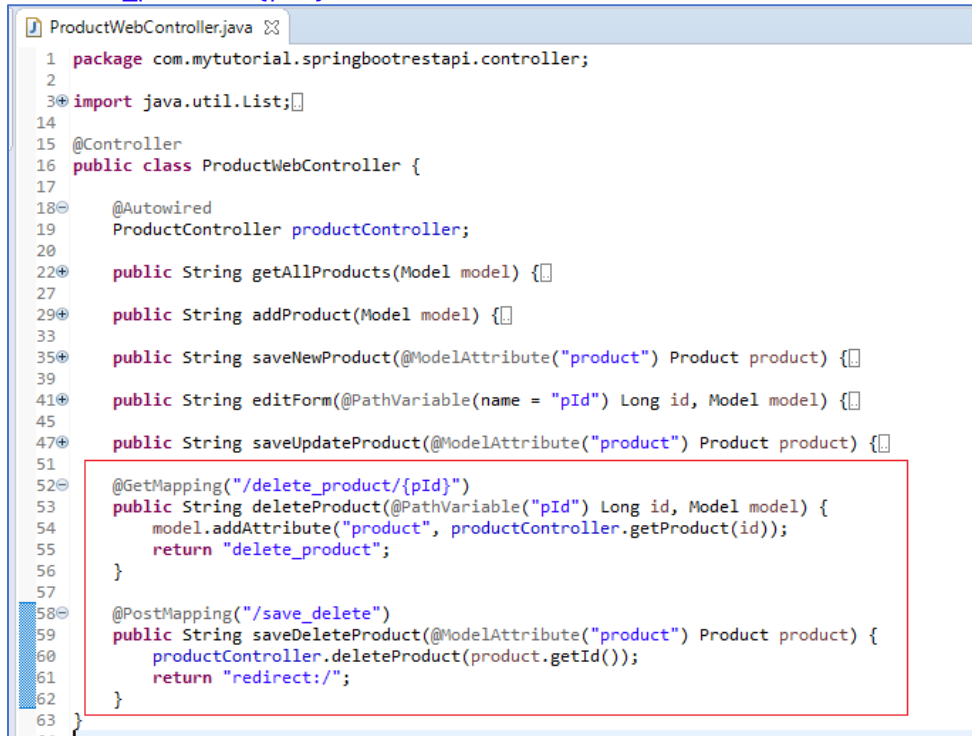
id	name	category
1	Television	Electronics
2	Air Conditioner	Electronics
3	Couch	Furniture
4	Cushions	Furnishings
5	Wardrobe	Furniture

We have Cushions in the Furnishings category and Couch in Furniture.

# Spring Boot Microservices: Building RESTful API Services

## 7. Deleting Records Using a Web UI

The only thing left now is to wire up the delete button. Back to **ProductWebController.java**, scroll down at the very bottom. I've added two methods here to delete a product from our database. The `deleteProduct` method is invoked when we make a **GET** request to `"/delete_product/{pId}"`



```
1 package com.mytutorial.springbootrestapi.controller;
2
3 import java.util.List;
4
14 @Controller
15 public class ProductWebController {
16
17     @Autowired
18     ProductController productController;
19
20     public String getAllProducts(Model model) {}
21
22     public String addProduct(Model model) {}
23
24     public String saveNewProduct(@ModelAttribute("product") Product product) {}
25
26     public String editForm(@PathVariable(name = "pId") Long id, Model model) {}
27
28     public String saveUpdateProduct(@ModelAttribute("product") Product product) {}
29
30     @GetMapping("/delete_product/{pId}")
31     public String deleteProduct(@PathVariable("pId") Long id, Model model) {
32         model.addAttribute("product", productController.getProduct(id));
33         return "delete_product";
34     }
35
36     @PostMapping("/save_delete")
37     public String saveDeleteProduct(@ModelAttribute("product") Product product) {
38         productController.deleteProduct(product.getId());
39         return "redirect:/";
40     }
41
42 }
```

So this path is dynamic, we extract the product ID using a path variable. Within this `deleteProduct` method, we simply add an instance of the product that will be deleted to the model using `model.addAttribute`. And we render the `"delete_product"` html view.

The `delete_product` view will contain a form bound to that product instance, we'll view the details of the product before we hit Delete. And when we actually hit Delete, a post request will be made to this method, `saveDeleteProduct`, map to the path `"/save_delete"`.

The product to be deleted will be injected as an input argument tagged using `@ModelAttribute`. And we simply call `productController.deleteProduct` on this ID and we redirect to the lists space `"redirect:/"`.

# Spring Boot Microservices: Building RESTful API Services

This will require a change to the **list\_products.html** page where we wire up the delete button.

```
list_products.html
1 <!DOCTYPE html>
2 <html xmlns:th="http://www.thymeleaf.org">
3 <head>
4   <meta charset="utf-8">
5   <meta http-equiv="x-ua-compatible" content="ie=edge">
6   <title>Products</title>
7   <meta name="viewport" content="width=device-width, initial-scale=1">
8   <link rel="stylesheet" href="https://stackpath.bootstrapcdn.com/bootstrap/4.1.3/css/bootstrap.min.css">
9   <link rel="stylesheet" href="https://use.fontawesome.com/releases/v5.4.1/css/all.css">
10 </head>
11 <h2 align="center">List of Products</h2>
12 <body>
13   <div class="container my-2">
14     <div class="card">
15       <div class="card-body">
16         <div th:switch="${products}" class="container my-4">
17           <p class="my-5">
18             <a href="/new_product" class="btn btn-primary">Add Product</a>
19           </p>
20
21           <div class="col-md-10">
22             <h2 th:case="null">No product found!</h2>
23             <div th:case="*">
24               <table class="table table-striped table-responsive-md">
25                 <thead>
26                   <tr>
27                     <th>Product ID</th>
28                     <th>Product Name</th>
29                     <th>Product Category</th>
30                     <th></th>
31                   </tr>
32                 </thead>
33                 <tbody>
34                   <tr>
35                     <th:each="product : ${products}">
36                       <td th:text="${product.id}"></td>
37                       <td th:text="${product.name}"></td>
38                       <td th:text="${product.category}"></td>
39                       <td><a th:href="@{/update_product/{pId}(pId=${product.id})}" class="btn btn-primary">Edit</a></td>
40                       <td><a th:href="@{/delete_product/{pId}(pId=${product.id})}" class="btn btn-danger">Delete</a></td>
41                     </tr>
42                   </tbody>
43                 </table>
44               </div>
45             </div>
46           </div>
47         </div>
48       </div>
49     </div>
50   </div>
51 </body>
52 </html>
```

The href for the delete button goes to `/delete_product/{pId}`.

```
<a th:href="@{/delete_product/{pId}(pId=${product.id})}" class="btn btn-danger">
```

# Spring Boot Microservices: Building RESTful API Services

Next, we also add a new page for actual product deletion. This **delete\_product.html** page is a simple form where the submit action of the form is to the `/save_delete` path, you can see this on line 18.

```
delete_product.html
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>Delete Product</title>
7 <meta name="viewport" content="width=device-width, initial-scale=1">
8 <link rel="stylesheet" href="https://stackpath.bootstrapcdn.com/bootstrap/4.1.3/css/bootstrap.min.css">
9 <link rel="stylesheet" href="https://use.fontawesome.com/releases/v5.4.1/css/all.css">
10 </head>
11 <style>
12   table, th, td {padding: 10px;}
13 </style>
14 <body>
15 <div align="center">
16 <h2>Delete Product?</h2>
17 <br/>
18 <form action="#" th:action="@{/save_delete}" th:object="${product}" method="post">
19 <table>
20 <tr>
21 <td>ID:</td>
22 <td><input type="text" th:field="*{id}" readonly="readonly" /></td>
23 </tr>
24 <tr>
25 <td>Name:</td>
26 <td><input type="text" th:field="*{name}" readonly="readonly" /></td>
27 </tr>
28 <tr>
29 <td>Category:</td>
30 <td><input type="text" th:field="*{category}" readonly="readonly" /></td>
31 </tr>
32 <tr>
33 <td colspan="2"><button class="btn-primary btn" type="submit">Delete</button>
34 </td>
35 </tr>
36 </table>
37 </form>
38 </div>
39 </body>
40 </html>
```

Note on lines 22, 26 and 30, all of the form fields are *readonly*.

Time to run this code and see the DELETE operation on our web UI in action. Let's go to `localhost:8080`, here are the products that we initialize our database with. Let's delete the product with ID 2, click on the Delete button.

localhost:8080/delete\_product/2

## Delete Product?

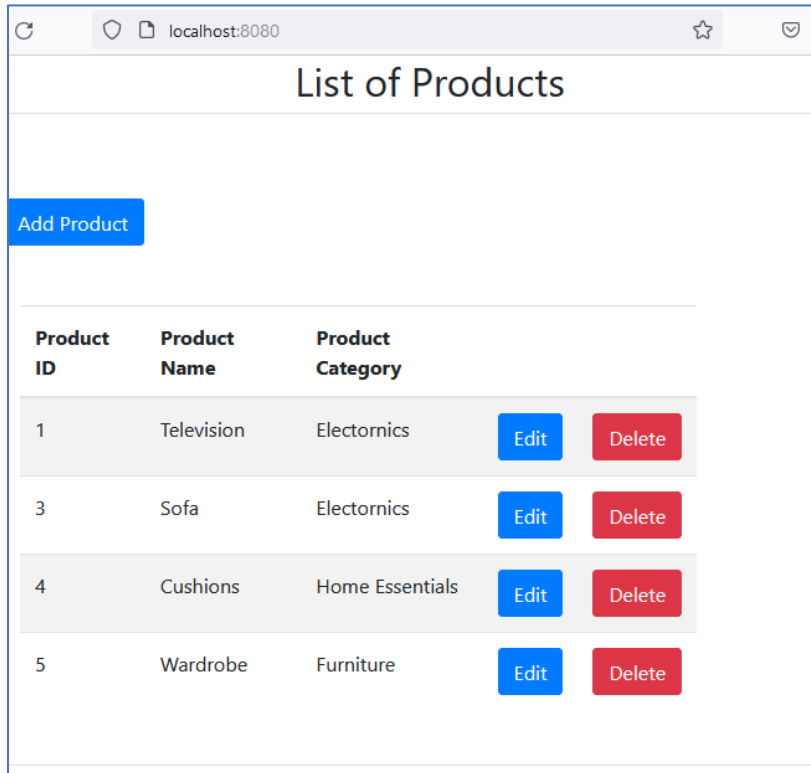
ID:

Name:

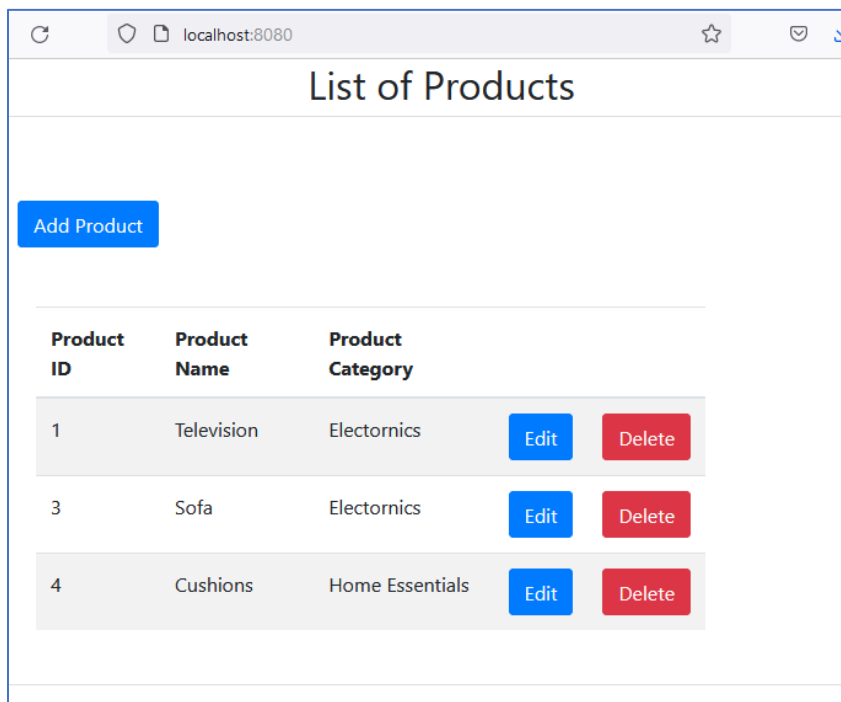
Category:

This is the Delete page. Notice the URL, `delete_product/2`. Let's hit Delete, that will confirm deletion. And when we go back to our products list page, you'll see that product has disappeared.

# Spring Boot Microservices: Building RESTful API Services

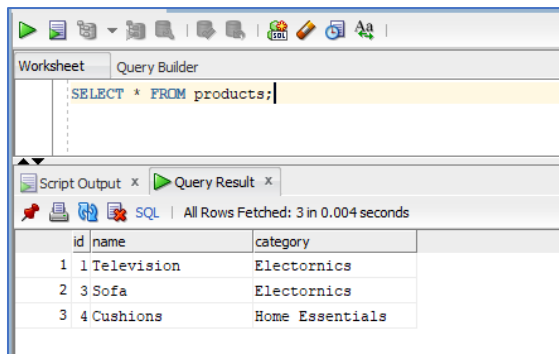


We'll delete one more product here. This time I'll get rid of the Wardrobe with ID 5, hit Delete. Let's confirm deletion on this page. When you hit Delete here, we'll go back to our products list page. And the product with ID 5 has disappeared as well.



# Spring Boot Microservices: Building RESTful API Services

We'll confirm deletion from the MySQL Workbench. A `SELECT * FROM products` shows us that we only have three products in our database after deletion.



The screenshot shows the MySQL Workbench interface. The 'Query Builder' tab is active, displaying the SQL query `SELECT * FROM products;`. Below the query editor, the 'Query Result' tab shows the results of the query. The status bar indicates 'All Rows Fetched: 3 in 0.004 seconds'. The results are displayed in a table with three columns: 'id', 'name', and 'category'.

	id	name	category
1	1	Television	Electornics
2	3	Sofa	Electornics
3	4	Cushions	Home Essentials



# Spring Boot Microservices: Building RESTful API Services

## Caching Using @Cacheable

Now if you were to retrieve products each time they were requested from the database, your app can get pretty slow. Also, if these products are listed on some kind of e-commerce site, the most common operation will be retrieval or read of products. Products won't be updated or deleted that often. One way to speed up the display of your products is by **caching**. And Spring Boot makes caching easy, where you can cache your products in memory or with some kind of **in-memory database using just annotations**.

1. Update Dependency in pom.xml

And in order to enable this, we'll work with the **spring-boot-starter-cache** dependency descriptor.

**pom.xml**

```
19<dependencies>
20
21    <dependency>
22        <groupId>org.springframework.boot</groupId>
23        <artifactId>spring-boot-starter-data-jpa</artifactId>
24    </dependency>
25
26    <dependency>
27        <groupId>org.springframework.boot</groupId>
28        <artifactId>spring-boot-starter-web</artifactId>
29    </dependency>
30
31    <dependency>
32        <groupId>mysql</groupId>
33        <artifactId>mysql-connector-java</artifactId>
34        <scope>runtime</scope>
35    </dependency>
36
37    <dependency>
38        <groupId>org.springframework.boot</groupId>
39        <artifactId>spring-boot-starter-thymeleaf</artifactId>
40    </dependency>
41
42    <dependency>
43        <groupId>org.springframework.boot</groupId>
44        <artifactId>spring-boot-starter-cache</artifactId>
45    </dependency>
46
47    <dependency>
48        <groupId>org.springframework.boot</groupId>
49        <artifactId>spring-boot-starter-test</artifactId>
50        <scope>test</scope>
51    </dependency>
52</dependencies>
```

# Spring Boot Microservices: Building RESTful API Services

## 2. Enable Caching on Entry Point Spring Boot Application

In order to enable caching within your application, you need to make a few changes. Here is our **SpringbootrestapiApplication.java** file. Notice that I've added another annotation to the class **@EnableCaching**.

```
SpringbootrestapiApplication.java
1 package com.mytutorial.springbootrestapi;
2
3 import org.springframework.beans.factory.annotation.Autowired;
4 import org.springframework.boot.CommandLineRunner;
5 import org.springframework.boot.SpringApplication;
6 import org.springframework.boot.autoconfigure.SpringBootApplication;
7 import org.springframework.cache.annotation.EnableCaching;
8
9 import com.mytutorial.springbootrestapi.dao.ProductRepository;
10 import com.mytutorial.springbootrestapi.model.Product;
11
12 @SpringBootApplication
13 @EnableCaching
14 public class SpringbootrestapiApplication implements CommandLineRunner {
15
16     @Autowired
17     private ProductRepository productRepository;
18
19     public static void main(String[] args) {
20         SpringApplication.run(SpringbootrestapiApplication.class, args);
21     }
22
23     @Override
24     public void run(String... args) throws Exception {
25         productRepository.save(new Product("Television", "Electronics"));
26         productRepository.save(new Product("Air Conditioner", "Electronics"));
27         productRepository.save(new Product("Sofa", "Electronics"));
28         productRepository.save(new Product("Cushions", "Home Essentials"));
29         productRepository.save(new Product("Wardrobe", "Furniture"));
30     }
31 }
32 }
```

When you use Spring Boot, the mere presence of the starter template that you specified in pom.xml along with this enable caching annotation, tells your Spring application that caching is enabled for this app.

# Spring Boot Microservices: Building RESTful API Services

## 3. Enable Caching on Service Layer

Now you can decide which layer you want your caching to be performed. I've explicitly chosen the Service layer for caching, so that not only my REST API can take advantage of caching, but also the web UI. Make sure you specify your caching annotations in a layer that is common to all of your clients. In our case, it is the service layer. Here we are within the *ProductService* class. I'm going to cache the response of exactly one method call here. This is the *getAllProducts*. Notice on line 20 I have an **@Cacheable** annotation with a *name*.

```
ProductService.java
1 package com.mytutorial.springbootrestapi.service;
2
3 import java.util.ArrayList;
4 import java.util.List;
5 import java.util.Optional;
6
7 import org.springframework.beans.factory.annotation.Autowired;
8 import org.springframework.cache.annotation.Cacheable;
9 import org.springframework.stereotype.Service;
10
11 import com.mytutorial.springbootrestapi.dao.ProductRepository;
12 import com.mytutorial.springbootrestapi.model.Product;
13
14 @Service
15 public class ProductService {
16
17     @Autowired
18     private ProductRepository productRepository;
19
20     @Cacheable("products")
21     public List<Product> getAllProducts() {
22         List<Product> products = new ArrayList<>();
23         productRepository.findAll().forEach(products::add);
24         return products;
25     }
26
27     public Optional<Product> getProduct(Long id) {
28         return productRepository.findById(id);
29     }
30
31     public void addProduct(Product product) {
32         productRepository.save(product);
33     }
34
35     public void updateProduct(Long id, Product product) {
36         if (productRepository.findById(id).get() != null)
37             productRepository.save(product);
38     }
39
40     public void deleteProduct(Long id) {
41         productRepository.deleteById(id);
42     }
43 }
```

Products will be the name of the cache, where the response from *getAllProducts* will be stored. In order to demonstrate the effects of caching, notice that I've added a *Thread.sleep* on line 24 within this method.

```
20 @Cacheable("products")
21 public List<Product> getAllProducts() {
22     List<Product> products = new ArrayList<>();
23     try {
24         Thread.sleep(3000); // to simulate the long process
25     } catch (Exception ex) {}
26     productRepository.findAll().forEach(products::add);
27     return products;
28 }
```

# Spring Boot Microservices: Building RESTful API Services

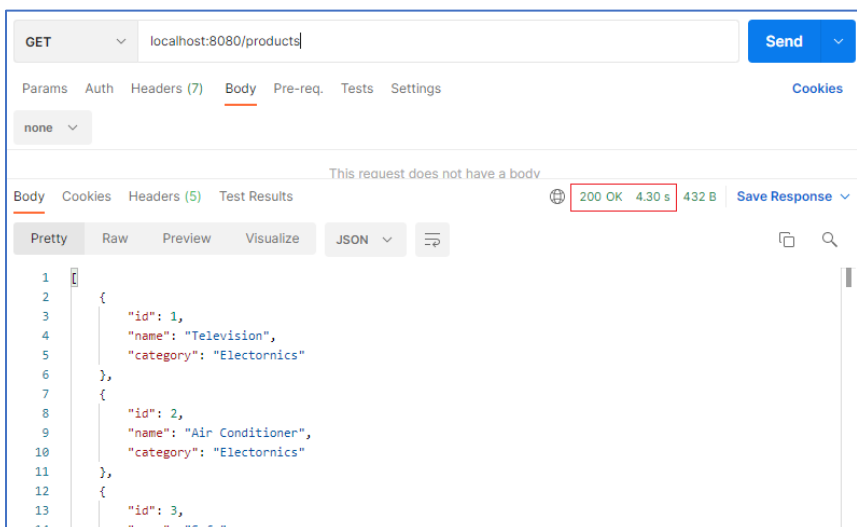
This will cause a normal execution of this method to run for at least three seconds.

Here is how the **@Cacheable** annotation works. If the list of products that is the response of this method is present within the products cache, that response is returned directly to the user. The code within this method will not be executed. When you invoke this method for the first time or if the cache has been previously cleaned up, the response for this method will not be present in that cache. In that case, the code for this method will be executed along with the sleep of 3 seconds, and the response will be cached in the products cache.

```
30 public Optional<Product> getProduct(Long id) {
31     try {
32         Thread.sleep(3000); // to simulate the long process
33     } catch (Exception ex) {}
34     return productRepository.findById(id);
35 }
36
37 public void addProduct(Product product) {
38     try {
39         Thread.sleep(3000); // to simulate the long process
40     } catch (Exception ex) {}
41     productRepository.save(product);
42 }
43
44 public void updateProduct(Long id, Product product) {
45     try {
46         Thread.sleep(3000); // to simulate the long process
47     } catch (Exception ex) {}
48     if (productRepository.findById(id).get() != null)
49         productRepository.save(product);
50 }
51
52 public void deleteProduct(Long id) {
53     try {
54         Thread.sleep(3000); // to simulate the long process
55     } catch (Exception ex) {}
56     productRepository.deleteById(id);
57 }
58 }
```

Now if you look at the remaining methods here within this product service, I haven't added caching to any other method. But to all methods, I've added the **Thread.sleep** for 3 seconds. So all methods will take at least 3 seconds to execute.

Let's run our application and see how caching affects the response time of our GET request and I'm going to make a **GET** request to *localhost:8080/products*. Click on the **SEND** button and wait for about 3 seconds.

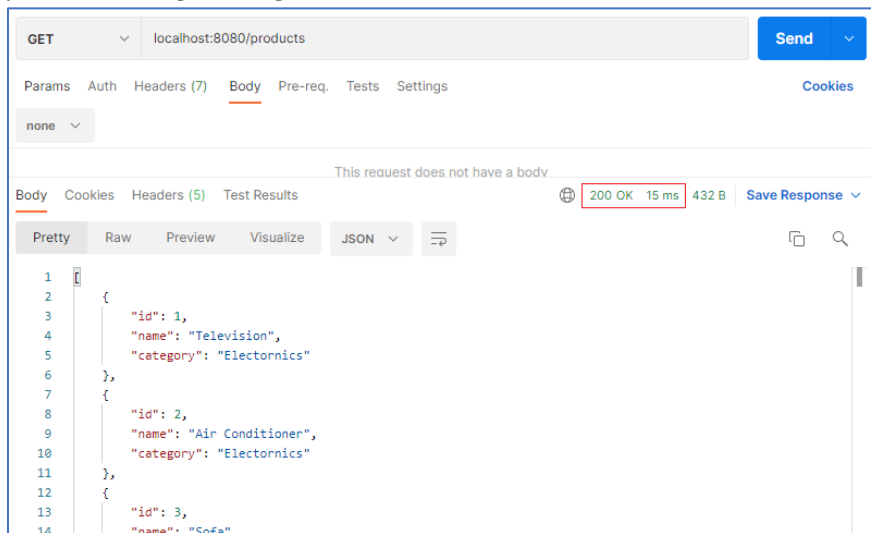


The screenshot shows a REST client interface with a GET request to `localhost:8080/products`. The response is a JSON array of three products. The response status is 200 OK, and the response time is 4.30 s. The response body is:

```
[
  {
    "id": 1,
    "name": "Television",
    "category": "Electronics"
  },
  {
    "id": 2,
    "name": "Air Conditioner",
    "category": "Electronics"
  },
  {
    "id": 3,
    "name": "Sofa"
  }
]
```

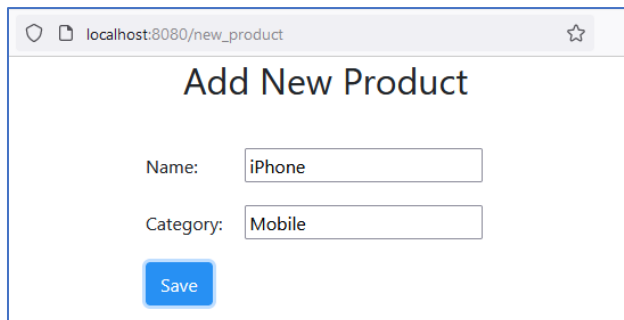
# Spring Boot Microservices: Building RESTful API Services

So you can see that this request took about 4 seconds to run right next to the 200 OK. It's pretty clear that these results were not retrieved from the cache. Let's go back and hit SEND on the products listing once again.



The previous request should have cached the results and notice next to the 200 OK we took only about 15 milliseconds to retrieve the results. So you can see that use of the **@Cacheable** annotation allowed us to cache our results in memory.

Now let's head over to our `localhost:8080` web user interface, and add some new products to our database. I click on Add Product and I'm going to add the iPhone in the Mobiles category and hit Save.



When we come back to our product listing, after having added this new product, you'll see something strange. Where is the new iPhone product that we'd added to the Mobiles category?

# Spring Boot Microservices: Building RESTful API Services

List of Products				
<a href="#">Add Product</a>				
Product ID	Product Name	Product Category		
1	Television	Electornics	<a href="#">Edit</a>	<a href="#">Delete</a>
2	Air Conditioner	Electornics	<a href="#">Edit</a>	<a href="#">Delete</a>
3	Sofa	Electornics	<a href="#">Edit</a>	<a href="#">Delete</a>
4	Cushions	Home Essentials	<a href="#">Edit</a>	<a href="#">Delete</a>
5	Wardrobe	Furniture	<a href="#">Edit</a>	<a href="#">Delete</a>

It's not present here in this list of products. Well, that's because this list of products has been retrieved from the cache. And we hadn't emptied the cache when we added a new product. So we still get the old response, the list of the original five products that were originally cached. Well, that's not really that good. Let's try deletion.

Click on the Delete button for say the product with ID 3.

localhost:8080/delete\_product/3

Delete Product?

ID:

3

Name:

Sofa

Category:

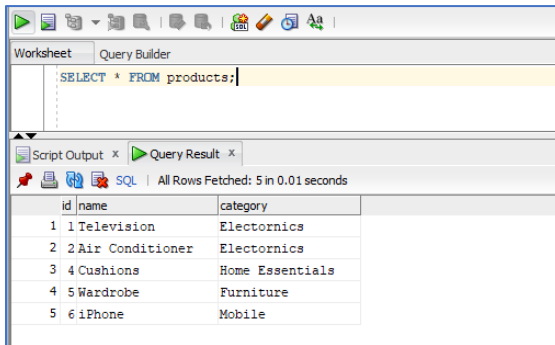
Electornics

Delete

Select the Delete option again to confirm deletion. When we go back to the list of products, the product with ID 3, or Sofa, is still present. We're still getting our cache results and not the updated list of products.

# Spring Boot Microservices: Building RESTful API Services

If you head over to the MySQL Workbench and do a `SELECT * FROM products;`

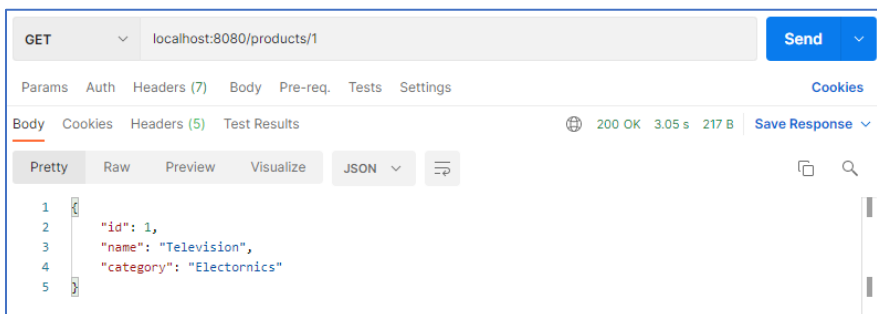


The screenshot shows the MySQL Workbench interface. The 'Query Builder' tab is active, displaying the query `SELECT * FROM products;`. Below the query, the 'Query Result' tab shows the results of the query. The results are displayed in a table with 5 rows and 3 columns: id, name, and category. The data is as follows:

id	name	category
1	Television	Electornics
2	Air Conditioner	Electornics
3	Cushions	Home Essentials
4	Wardrobe	Furniture
5	iPhone	Mobile

you will see that the new product that we added the iPhone is present here, also the product with ID 3 has been deleted. Our database reflects the current state of the world. But because we cached our responses in the service layer, anything that uses the service layer will get a stale response. So that isn't great. That's something we need to take care of and we will in a bit.

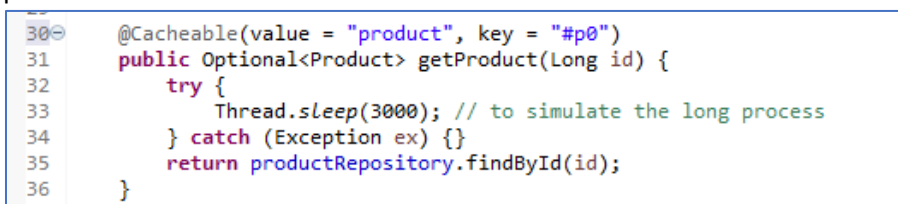
Now we cache the response where we retrieved all products. But what about accessing a single product? That's exactly what we'll do now. I'm going to make a GET request for the product with ID 1.



When you hit SEND, you'll see that this response takes about 3 seconds to run, right next to the 200 OK.

This response is clearly not from the cache. It has actually run the code to retrieve products. I'm going to hit SEND once again, and once again, the response will take about 3 seconds. We haven't enabled caching for the method that retrieves a single product, which is why we always have to hit the database to get this product.

Let's fix this. Let's add caching to the retrieval of a single product by ID as well. Here we are in the `ProductService` class. We already have an `@Cacheable` annotation on the `getAllProducts` method. And here I've added another `@Cacheable` annotation for the `getProduct` method, which takes as an input *argument the ID* of the product to retrieve. Now **the name of the cache** where we'll store this product retrieve by ID is **different**. It's called simply `product` rather than `products`.



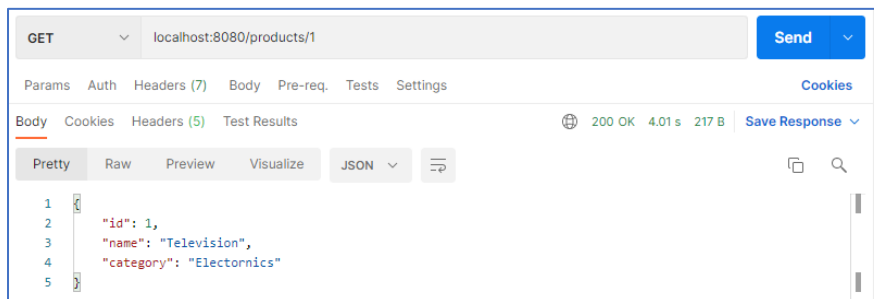
```
30 @Cacheable(value = "product", key = "#p0")
31 public Optional<Product> getProduct(Long id) {
32     try {
33         Thread.sleep(3000); // to simulate the long process
34     } catch (Exception ex) {}
35     return productRepository.findById(id);
36 }
```

# Spring Boot Microservices: Building RESTful API Services

In addition, we also specify a key that'll be used to store the response for one invocation of this method. The key is equal to **#p0**, which means the key will be the *first input argument* to this method, which is the ID of the product.

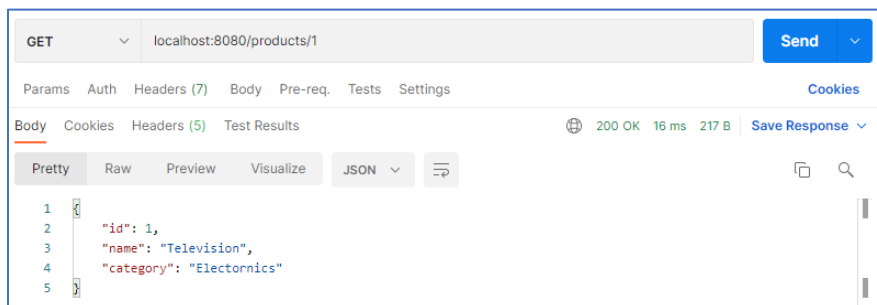
The response of this method will be stored in the cache keyed by the ID of the product. So if a product with ID 5 is say present in the cache, that does not mean that a product with ID 10 is also present in the cache. I still haven't added any caching to the other methods in this service, not to update product, not to delete product.

Time to run this code and see how the caching of single products work. Let's head over to our advanced REST client, and I'm going to make a request to the product with ID 1. This is a GET request.



Click on SEND and you can see next to the 200 OK that the response took about 4 seconds to get back to us. Remember, we have the Thread.sleep in there.

Now let's try and hit SEND once again and make the same request again.

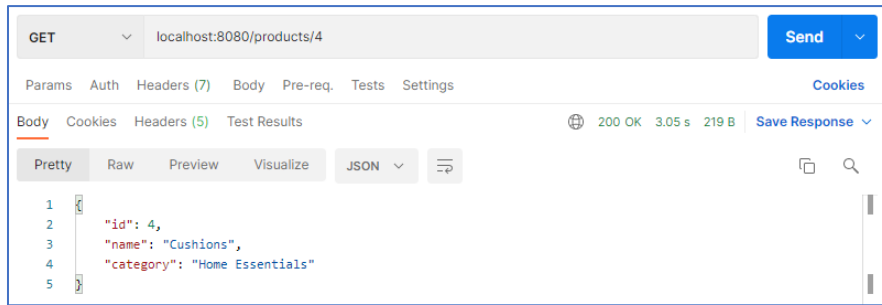


And this time, we got the response in 16 milliseconds. That's because this product with ID 1 was found in the cache and retrieved and returned. The code with the Thread.sleep wasn't executed at all.

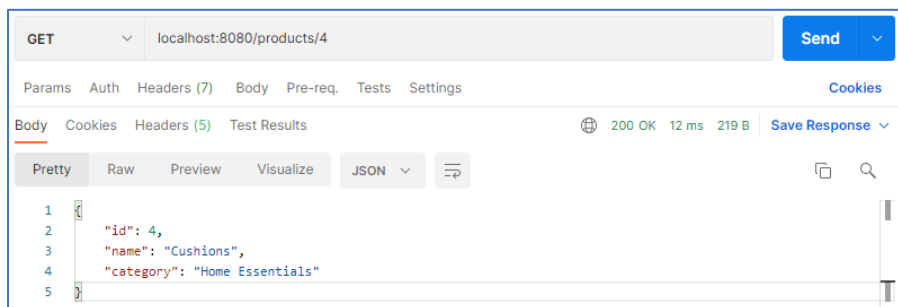


# Spring Boot Microservices: Building RESTful API Services

Let's try retrieving the product with ID 4, hit SEND.



And since this is the first time we're accessing this product, you'll see that the response takes over 3 seconds to get back to us. Right next to the 200 OK, you can see the response time. Now if you hit SEND once again, the second time, we get a response in 12 milliseconds. The product with ID 4 is now in the cache as well.



# Spring Boot Microservices: Building RESTful API Services

## Clearing Caches Using @CacheEvict

So far, we've added caching to two methods in our service.

### ProductService.java

```
20 @Cacheable("products")
21 public List<Product> getAllProducts() {
22     List<Product> products = new ArrayList<>();
23     try {
24         Thread.sleep(3000); // to simulate the long process
25     } catch (Exception ex) {}
26     productRepository.findAll().forEach(products::add);
27     return products;
28 }
29
30 @Cacheable(value = "product", key = "#p0")
31 public Optional<Product> getProduct(Long id) {
32     try {
33         Thread.sleep(3000); // to simulate the long process
34     } catch (Exception ex) {}
35     return productRepository.findById(id);
36 }
```

One which retrieves a list of all products from the database, and another that retrieves the product by ID. Now let's set up the remaining caching annotations so our service works correctly.

We need to evict the cache at the right point in time. We saw earlier that when we add a new product, our cache wasn't updated. That's because we were not evicting the responses from cache, let's fix that using the **@CacheEvict** annotation.

### ProductService.java

```
39 @CacheEvict(value = "products", allEntries = true)
40 public void addProduct(Product product) {
41     try {
42         Thread.sleep(3000); // to simulate the long process
43     } catch (Exception ex) {}
44     productRepository.save(product);
45 }
46
47 public void updateProduct(Long id, Product product) {
48     try {
49         Thread.sleep(3000); // to simulate the long process
50     } catch (Exception ex) {}
51     if (productRepository.findById(id).get() != null)
52         productRepository.save(product);
53 }
54
55 public void deleteProduct(Long id) {
56     try {
57         Thread.sleep(3000); // to simulate the long process
58     } catch (Exception ex) {}
59     productRepository.deleteById(id);
60 }
```

We've added this annotation to the addProduct method on our service. We perform a **cache eviction** on the cache named *products*. If you remember, the cache with the name *products* holds a list of all of the products that we have.

# Spring Boot Microservices: Building RESTful API Services

Also note that we evict **allEntries** within the *products*' cache, allEntries is set to **true**. This means, this cache will be completely cleared so that the new product will be visible in the response next time we retrieve all products.

This is the only new annotation I've added. I haven't added any cache eviction when we update a product or when we delete a product.

Let's give our app a test run to see whether this CacheEvict annotation does what it should. I'm going to make a request to localhost:8080, here's the list of products available in our database.

List of Products				
<a href="#">Add Product</a>				
Product ID	Product Name	Product Category		
1	Television	Electronics	<a href="#">Edit</a>	<a href="#">Delete</a>
2	Air Conditioner	Electronics	<a href="#">Edit</a>	<a href="#">Delete</a>
3	Sofa	Electronics	<a href="#">Edit</a>	<a href="#">Delete</a>
4	Cushions	Home Essentials	<a href="#">Edit</a>	<a href="#">Delete</a>
5	Wardrobe	Furniture	<a href="#">Edit</a>	<a href="#">Delete</a>

I'm going to click on the Add Product button here and add a new product.

localhost:8080/new\_product

Add New Product

Name:

iPhone

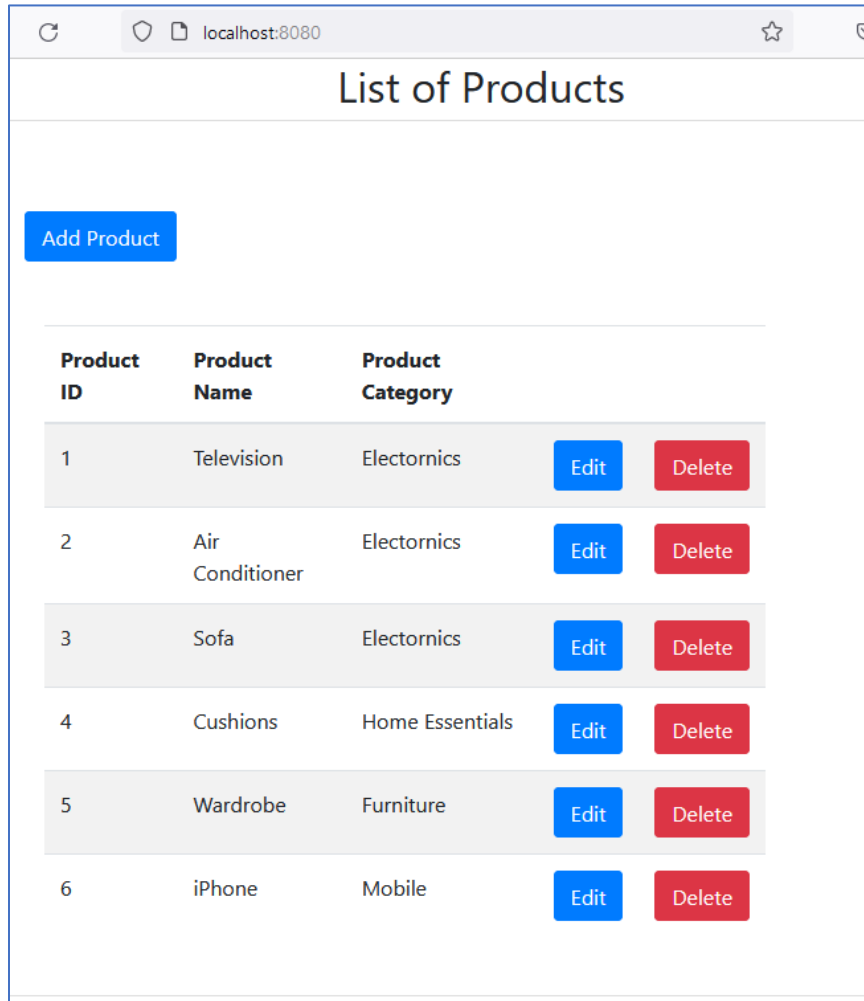
Category:

Mobile

Save

This will be the iPhone in the Mobile category. Click on the Save button, and when we head back to our list of products, you can see the iPhone is available at the very bottom.

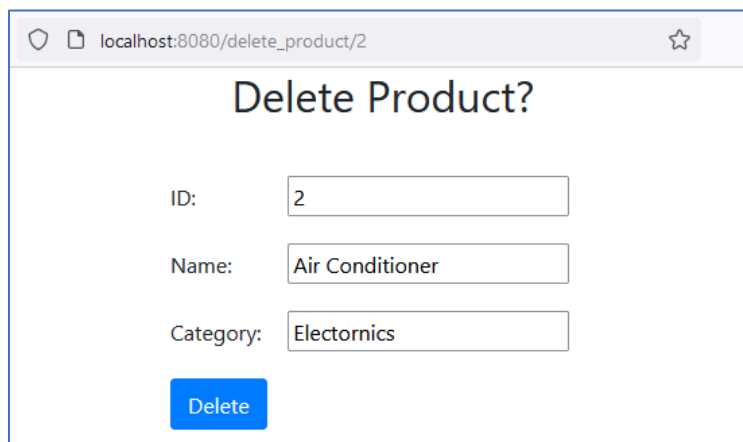
# Spring Boot Microservices: Building RESTful API Services



Product ID	Product Name	Product Category		
1	Television	Electornics	Edit	Delete
2	Air Conditioner	Electornics	Edit	Delete
3	Sofa	Electornics	Edit	Delete
4	Cushions	Home Essentials	Edit	Delete
5	Wardrobe	Furniture	Edit	Delete
6	iPhone	Mobile	Edit	Delete

That's because adding a new product evicted all of the products which were in the cache. This means the next time we retrieve the list of all products, we had to get the products once again from the database.

Let's try some of the other operations here. I'm going to delete the product Air Conditioner from my list of products.



Delete Product?

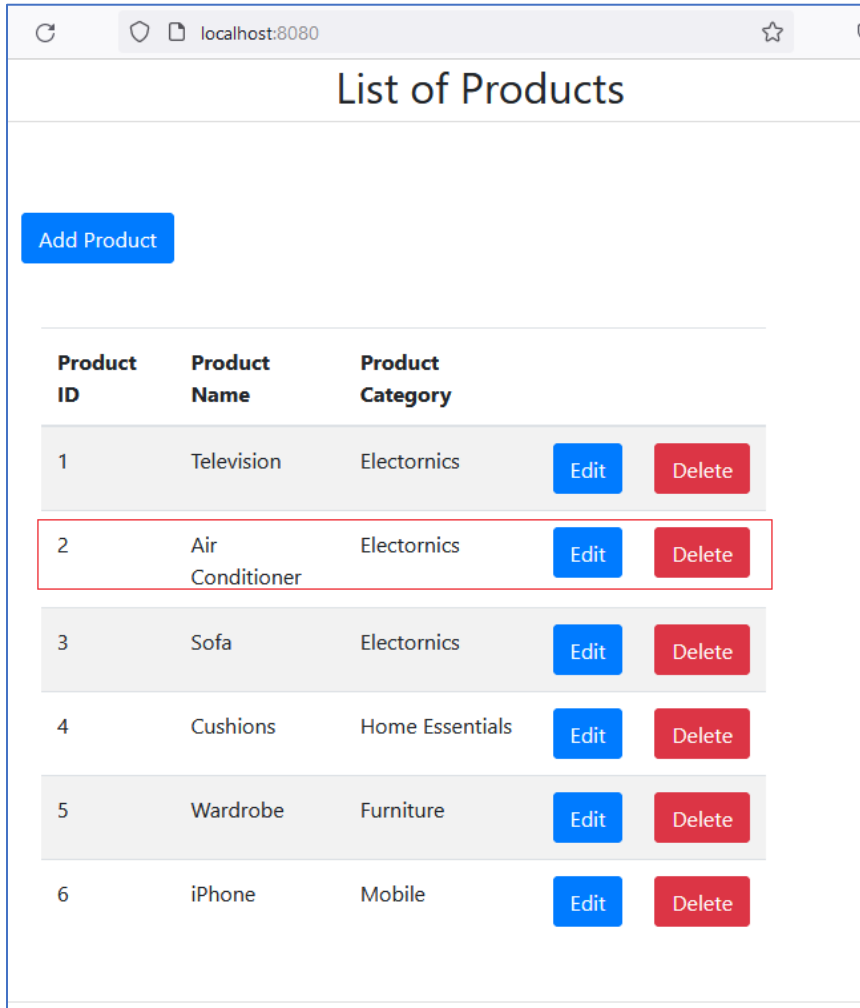
ID:

Name:

Category:

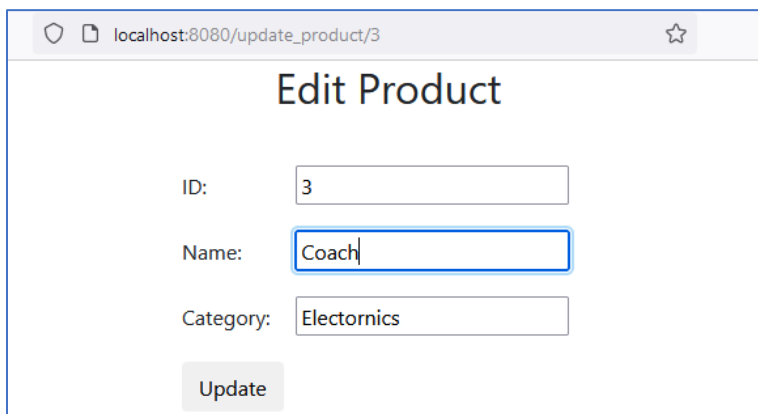
# Spring Boot Microservices: Building RESTful API Services

I'm going to confirm deletion here. But because we haven't explicitly evicted our cache, you can see that even after deletion, the Air Conditioner is still present in our list of products.



Product ID	Product Name	Product Category		
1	Television	Electornics	Edit	Delete
2	Air Conditioner	Electornics	Edit	Delete
3	Sofa	Electornics	Edit	Delete
4	Cushions	Home Essentials	Edit	Delete
5	Wardrobe	Furniture	Edit	Delete
6	iPhone	Mobile	Edit	Delete

Let's try editing a product. I'm going to edit this product Sofa. I'm going to call it Couch instead and then hit Update.



**Edit Product**

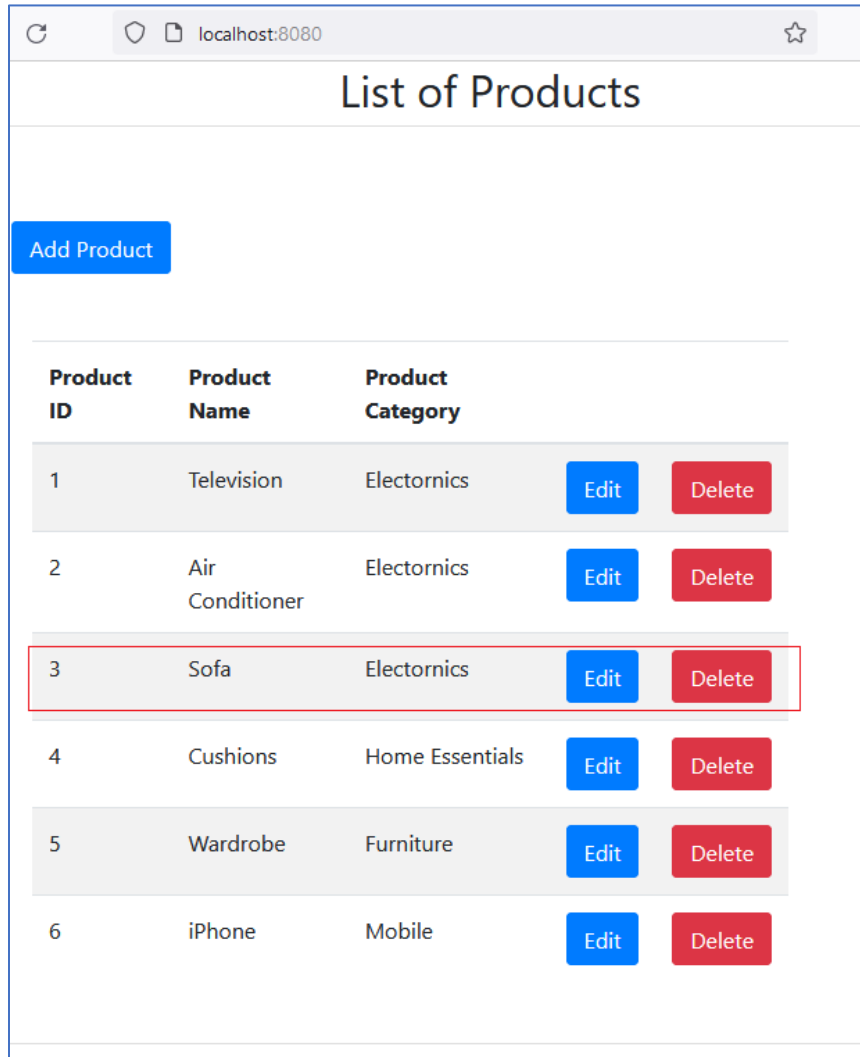
ID:

Name:

Category:

We haven't cleared our caches on product update.

# Spring Boot Microservices: Building RESTful API Services



Product ID	Product Name	Product Category		
1	Television	Electornics	Edit	Delete
2	Air Conditioner	Electornics	Edit	Delete
3	Sofa	Electornics	Edit	Delete
4	Cushions	Home Essentials	Edit	Delete
5	Wardrobe	Furniture	Edit	Delete
6	iPhone	Mobile	Edit	Delete

So when we go back to our list of products, you can see that the product with ID 3 still says Sofa instead of Couch.

Let's fix our product service once and for all by performing cache evictions at the right time. Scroll down to the very bottom and let's get to the method where we update an existing product in our database. Here's the `updateProduct` method, notice that we have a bunch of `evict` annotations in here.

# Spring Boot Microservices: Building RESTful API Services

Now we have to use the `@Caching` annotation to specify multiple cache evicts. Why do we need multiple evicts? Well, that's because we have two caches, one to cache all of the products in our database and another to cache a product with a specific ID. The first cache evict on line 68 is for the product with a specific ID, `key="#p0"`.

## ProductService.java

```
1 package com.mytutorial.springbootrestapi.service;
2
3 import java.util.ArrayList;
4 import java.util.List;
5 import java.util.Optional;
6
7 import org.springframework.beans.factory.annotation.Autowired;
8 import org.springframework.cache.annotation.CacheEvict;
9 import org.springframework.cache.annotation.Cacheable;
10 import org.springframework.cache.annotation.Caching;
11 import org.springframework.stereotype.Service;
12
13 import com.mytutorial.springbootrestapi.dao.ProductRepository;
14 import com.mytutorial.springbootrestapi.model.Product;
15
16 @Service
17 public class ProductService {
18
19     @Autowired
20     private ProductRepository productRepository;
21
22     @Cacheable("products")
23     public List<Product> getAllProducts() {
24         List<Product> products = new ArrayList<>();
25         productRepository.findAll().forEach(products::add);
26         return products;
27     }
28
29     @Cacheable(value = "product", key = "#p0")
30     public Optional<Product> getProduct(Long id) {
31         return productRepository.findById(id);
32     }
33
34     @CacheEvict(value = "products", allEntries = true)
35     public void addProduct(Product product) {
36         productRepository.save(product);
37     }
38
39     @Caching(evict = {
40         @CacheEvict(value = "product", key = "#p0"),
41         @CacheEvict(value = "products", allEntries = true)
42     })
43     public void updateProduct(Long id, Product product) {
44         if (productRepository.findById(id).get() != null)
45             productRepository.save(product);
46     }
47
48     @Caching(evict = {
49         @CacheEvict(value = "product", key = "#p0"),
50         @CacheEvict(value = "products", allEntries = true)
51     })
52     public void deleteProduct(Long id) {
53         productRepository.deleteById(id);
54     }
55
56 }
```

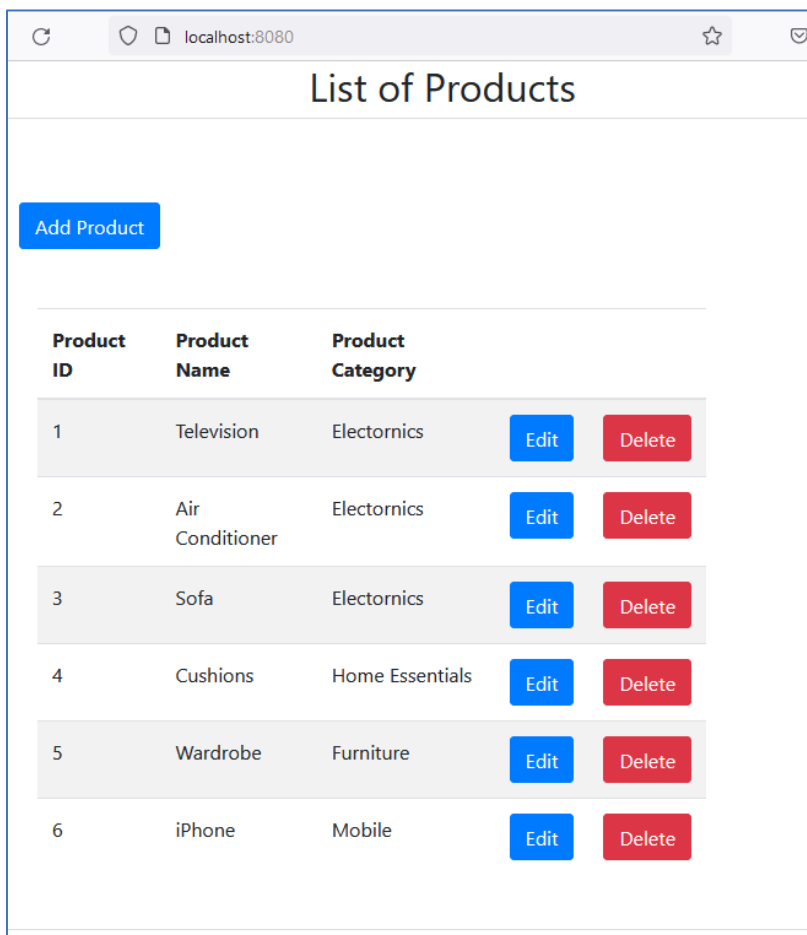
Evict the response corresponding to the key that is the first input argument to this method, the ID of the product. The second `CacheEvict` on line 69 is for the products cache. Here `allEntries=true` meaning, when we update a single product invalidate the entire cache. Because we don't know which particular product has been updated in our list of products.

# Spring Boot Microservices: Building RESTful API Services

When we delete a product, we'll perform the same set of cache evictions. We have the **@Caching** annotation and two evict annotations within it. We **CacheEvict** the product with **key="#p0"**, that is the ID of the product. And we perform a **CacheEvict** for **products**, **allEntries=true**.

```
@Caching(evict = {  
    @CacheEvict(value = "product", key = "#p0"),  
    @CacheEvict(value = "products", allEntries = true)  
})
```

Now our application will work as we expect, so let's run our code and head over to localhost:8080. Here's the list of all products currently existing in our database. Let's use the Add Product button and add in a new product iPhone in the Mobiles category. Hit Save and when we go back to our list of products, you will find the iPhone right there at the bottom of your screen.

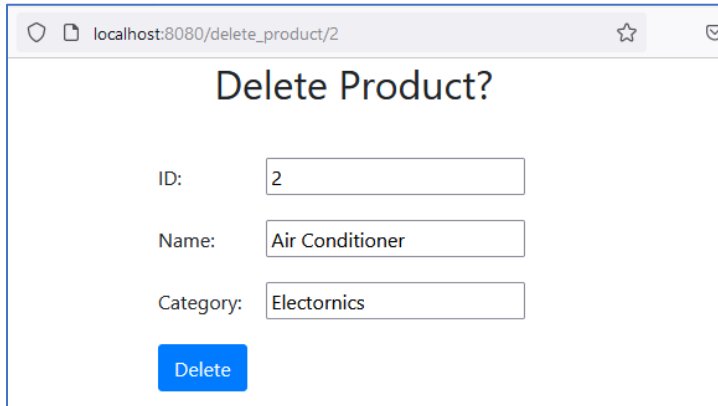


Product ID	Product Name	Product Category		
1	Television	Electornics	Edit	Delete
2	Air Conditioner	Electornics	Edit	Delete
3	Sofa	Electornics	Edit	Delete
4	Cushions	Home Essentials	Edit	Delete
5	Wardrobe	Furniture	Edit	Delete
6	iPhone	Mobile	Edit	Delete



# Spring Boot Microservices: Building RESTful API Services

Now let's delete one of the products here. I'm going to get rid of the Air Conditioner, the product with ID 2. Hit Delete to confirm deletion.



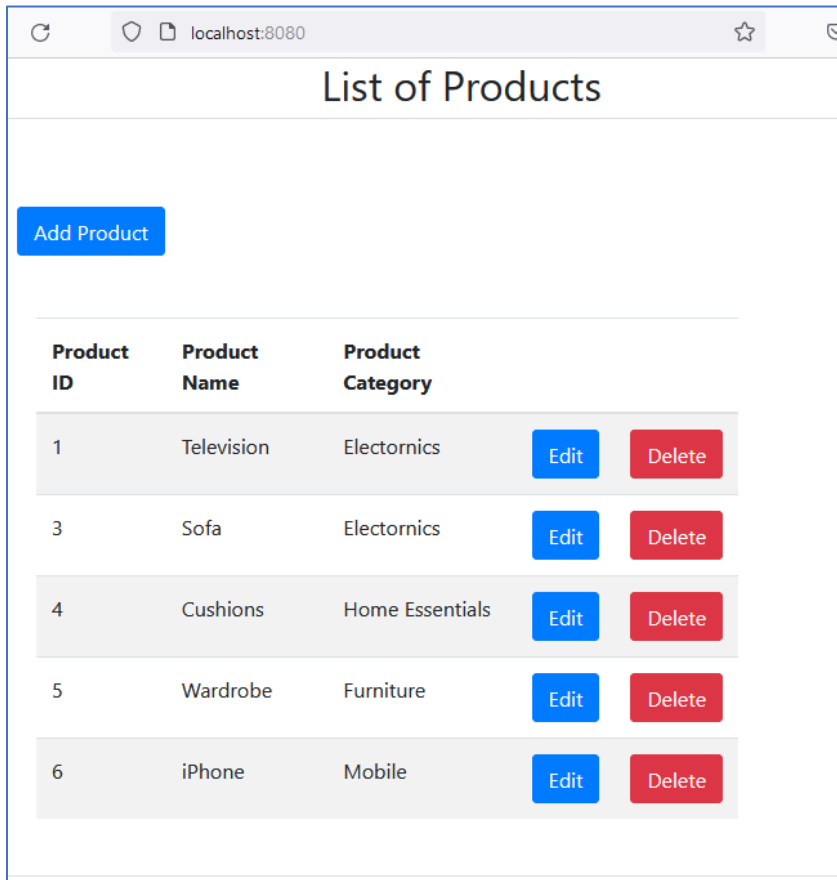
Delete Product?

ID:

Name:

Category:

And when we go back to the list of products here, you can see that the Air Conditioner has disappeared.



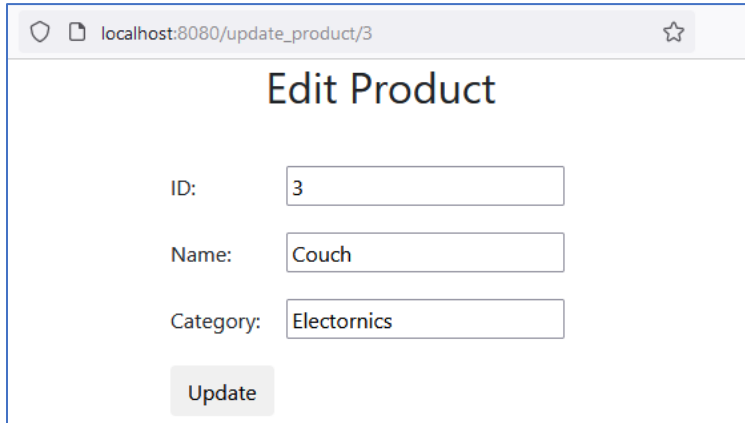
List of Products

Product ID	Product Name	Product Category		
1	Television	Electornics	<input type="button" value="Edit"/>	<input type="button" value="Delete"/>
3	Sofa	Electornics	<input type="button" value="Edit"/>	<input type="button" value="Delete"/>
4	Cushions	Home Essentials	<input type="button" value="Edit"/>	<input type="button" value="Delete"/>
5	Wardrobe	Furniture	<input type="button" value="Edit"/>	<input type="button" value="Delete"/>
6	iPhone	Mobile	<input type="button" value="Edit"/>	<input type="button" value="Delete"/>

Our cache evictions were performed correctly and our list of products here does not contain a deleted product.

# Spring Boot Microservices: Building RESTful API Services

Let's edit one of the products here. Let's edit the Sofa product, instead of Sofa, I'm going to call it Couch.



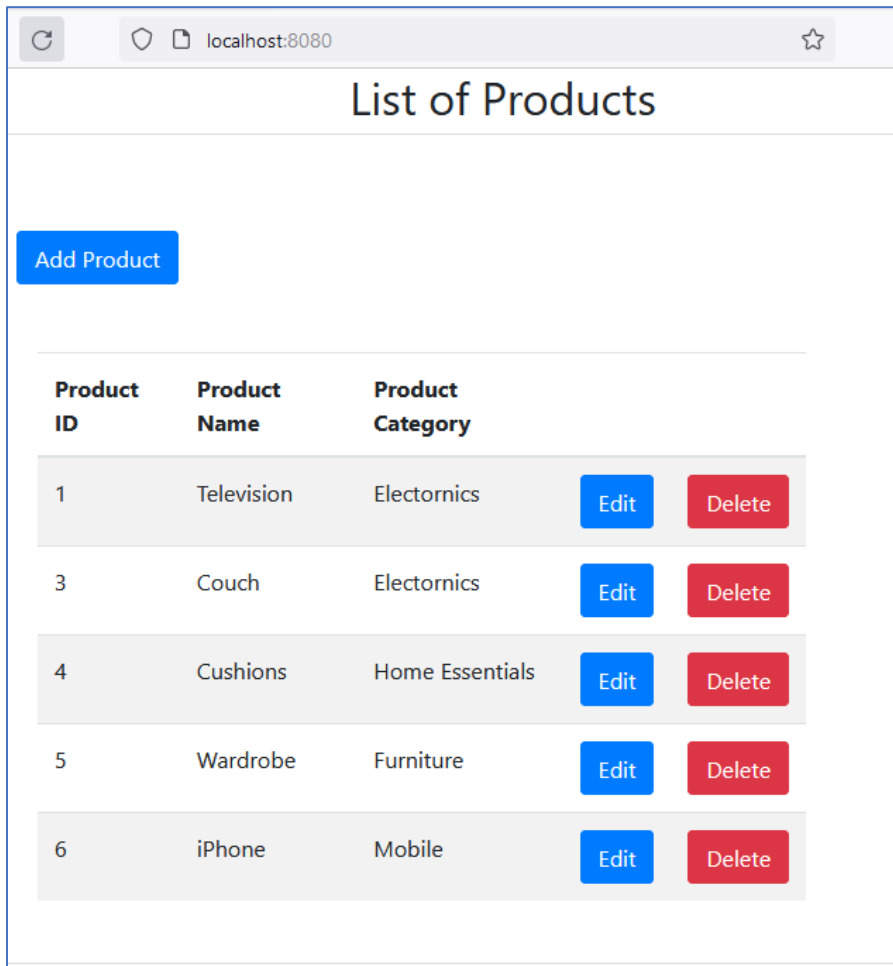
**Edit Product**

ID:

Name:

Category:

Hit Update after making this change. And when you go back to our list of products here, you can see that the product with ID 3 is now called Couch.



**List of Products**

Product ID	Product Name	Product Category		
1	Television	Electornics	<input type="button" value="Edit"/>	<input type="button" value="Delete"/>
3	Couch	Electornics	<input type="button" value="Edit"/>	<input type="button" value="Delete"/>
4	Cushions	Home Essentials	<input type="button" value="Edit"/>	<input type="button" value="Delete"/>
5	Wardrobe	Furniture	<input type="button" value="Edit"/>	<input type="button" value="Delete"/>
6	iPhone	Mobile	<input type="button" value="Edit"/>	<input type="button" value="Delete"/>

# Spring Boot Microservices: Building RESTful API Services

## Summary

In this course, we explored RESTful microservices in Spring Boot. We integrated our application with a relational database, set up a front end for our application. And added caching to improve the performance of our application's responses. We started off by building a simple API which worked with in memory data in order to perform create, read, update and delete operations, in response to HTTP web requests from the user.

We made these web requests using the advanced REST client. We then moved on to wiring up a MySQL database on the back end to store the data that could be queried using the REST API. We installed MySQL and MySQL Workbench on our local machines. We then used Spring Data with JPA and Hibernate to integrate with this database, and store our entities as records in the underlying table. We rounded off our application by adding a web front end using the Thymeleaf template engine. We then improved the responses from our application using spring caching.

# Spring Boot Microservices: Building RESTful API Services

## Quiz

1. What is the Advanced Rest Client?  
*A debugging tool for web applications*  
✓ An API testing tool which helps make HTTP requests to URLs  
*A request generation tool which helps generate a large number of queries per second*  
*A dependency injection tool which injects the right parameters in URLs*
2. What annotation would you apply to the component which holds business logic in your application?  
*@Bean*  
*@Controller*  
*@Component*  
✓ @Service
3. What kind of HTTP request would you make to add a new record to your database?  
*OPTIONS*  
*GET*  
*HEAD*  
✓ POST  
*DELETE*
4. What kind of HTTP request would you make to update an existing record in your database?  
*HEAD*  
*DELETE*  
*OPTIONS*  
✓ PUT  
*GET*
5. What is the most relevant HTTP request that you can make to remove an existing record in your database?  
*HEAD*  
*GET*  
*POST*  
✓ DELETE  
*PUT*  
*OPTIONS*
6. What is MySQL Workbench?  
✓ A user interface that can be used with the MySQL database to run queries  
*A SQL database that stores data in tables*  
*A debugging tool to be used with a SQL database*  
*A performance tool that can be used with the MySQL database to monitor queries*

# Spring Boot Microservices: Building RESTful API Services

7. MySQL runs on which default port?

*7077*

✓ 3306

*8080*

*8888*

8. Which Spring module automatically provides implementations for CRUD operations on your database entities?

*Spring ORM*

*Spring MVC*

*Spring Security*

✓ Spring Data

9. Match the CRUD operation with the correct HTTP request:

DELETE D:Delete

PUT C:Update

POST A:Create

GET B:Read

10. Which Thymeleaf construct is used to display different UIs based on a condition?

*th:object*

*th:text*

✓ *th:switch*

*th:each*

11. How do you specify that an input text box cannot be edited?

*Set enabled=false*

✓ Specify a value for the "readonly" attribute

*Set disabled=true*

*Specify a value for the "editable" attribute*

12. Which statements accurately describe the @Cacheable annotation?

✓ Cache responses are not automatically updated when a change has occurred

✓ Once the response is cached, the code for the method is not invoked

*Cache responses are automatically updated when a change has occurred*

*Even though the response is cached, the code for the method is invoked*

13. What annotation would you use to clear the cache in case of deletes or updates?

✓ @CacheEvict

*@NoCache*

*@CacheDelete*

*@CacheRemove*