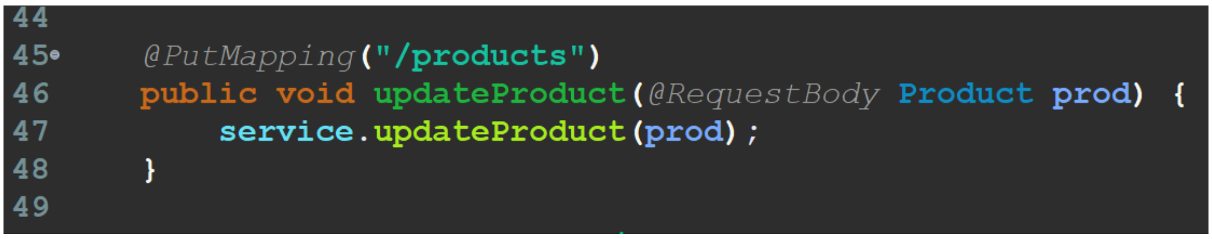
16-Spring Web Put & Delete

Now, let's move on to handling update and delete operations for HTTP requests. In the previous chapter, we demonstrated how to retrieve products using a GET request and how to add a product to the list using a POST request.

Next, we'll focus on updating existing products. Choose any product you'd like to modify, whether it's the price, name, or other details. To start, use the Postman tool to send a GET request to retrieve the full list of products. Once you've obtained the JSON data for a specific product, switch the request type to PUT. Paste the JSON data into the request body, ensuring that the data type is set to JSON, and make the necessary modifications.

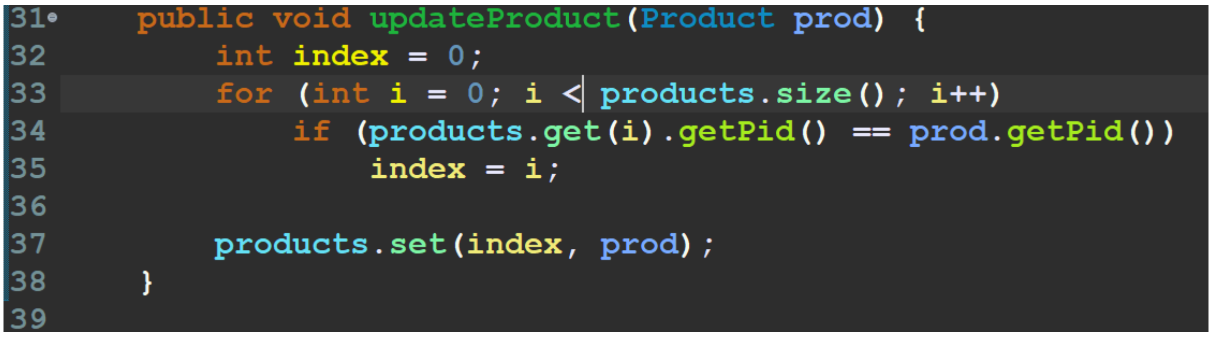
At this point, we won't receive any output or data change for our product, as we haven't yet created the corresponding method for it. So, in our code no, get into the ProductController class and create a method updateProduct accepting the Product object and method annotating with @PutMapping(`/products`), where inside the method is the implementation class of method updateProduct, i.e., our ProductService class.



```java

    @PutMapping("/products")
    public void updateProduct(@RequestBody Product prod) {
        service.updateProduct(prod);
    }

```

In ProductService, we create a method updateProduct that accepts the product as a parameter. To update a specific product, we require its ID. Therefore, we first construct a logic to retrieve its ID, completing our task.

This implementation and thinking of logic to get the ID is a cake-walk thing in Data JPA, and it has various such methods already present in it, which makes our developer life easy.



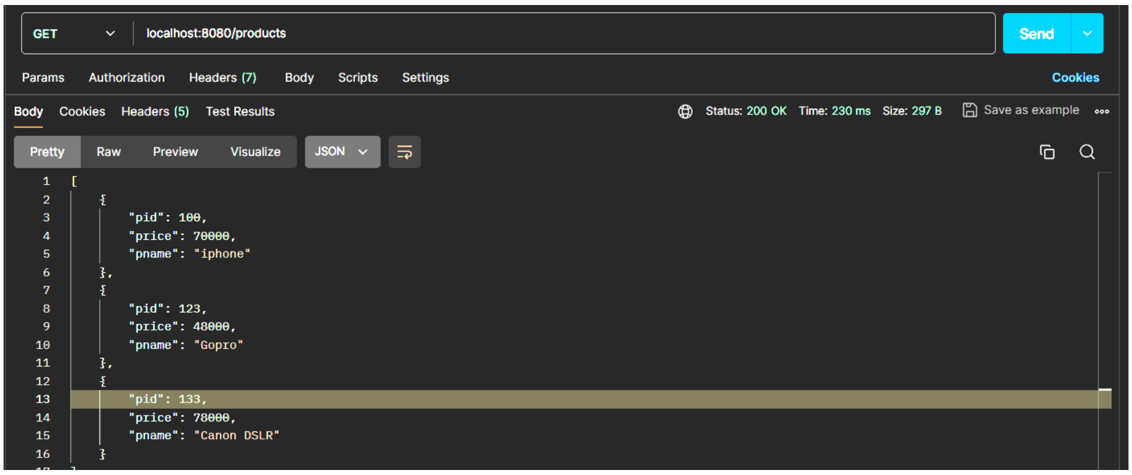
```java
    public void updateProduct(Product prod) {
        int index = 0;
        for (int i = 0; i < products.size(); i++)
            if (products.get(i).getPid() == prod.getPid())
                index = i;

        products.set(index, prod);
    }

```

This method loops through a list of products to find the one with a matching `Product ID` (`Pid`) to the product passed as an argument.

Once the product is found, it replaces the old product in the list with the new product (`prod`) using the `set()` method.

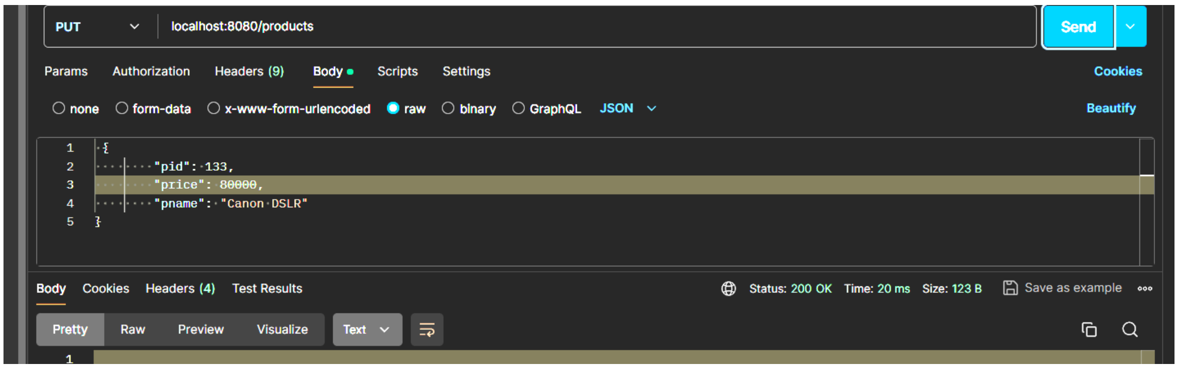
Let's proceed with the project. First, in the Postman tool, retrieve all the product lists. We have hardcoded all the product lists because we are not currently using any databases. Once we've gathered all the products, choose any individual product, modify the request type to PUT, incorporate the data into the body, and submit the request to verify the updated product information.

Output:



```
GET localhost:8080/products
Status: 200 OK  Time: 230 ms  Size: 297 B
```

```
[
    {
        "pid": 100,
        "price": 70000,
        "pname": "iphone"
    },
    {
        "pid": 123,
        "price": 48000,
        "pname": "Gopro"
    },
    {
        "pid": 133,
        "price": 78000,
        "pname": "Canon DSLR"
    }
```



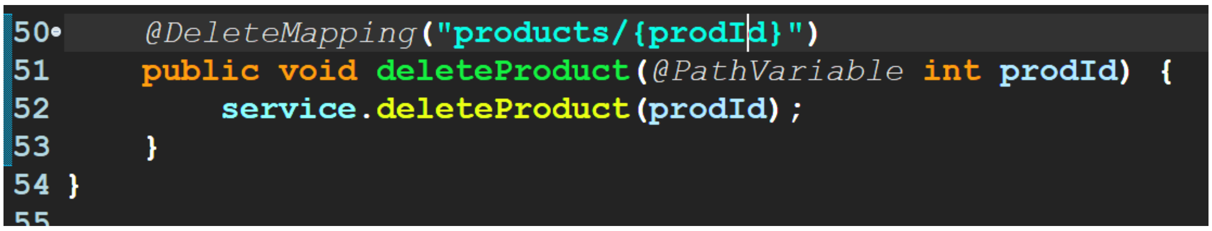
```
PUT localhost:8080/products
```

```
{
    "pid": 133,
    "price": 80000,
    "pname": "Canon DSLR"
}
```

```
Status: 200 OK  Time: 20 ms  Size: 123 B
```

Now moving on to the delete request part, firstly in the code in our ProdcutController create method for it naming as deleteProduct annoataing with ***@DeleteMapping(`/products/{id}`)***

The parameter also includes an annotation for @Pathvariable, which takes the id from the URL.



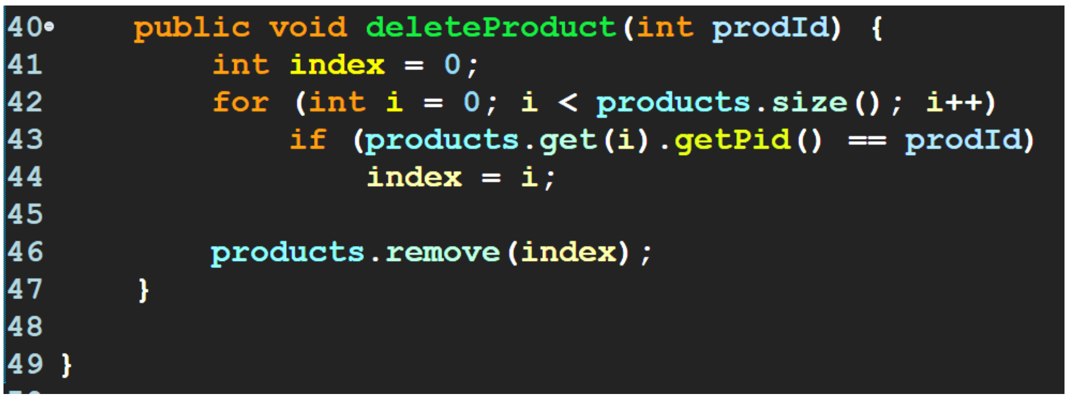
```java
    @DeleteMapping("products/{prodId}")
    public void deleteProduct(@PathVariable int prodId) {
        service.deleteProduct(prodId);
    }
}
```

Create a method in our ProductService class that accepts parameters as our product ID. Since we are deleting products based on ID, we need to implement the same logic as we did in the update part.

The method iterates through the list of products to find the one that matches the given `prodId`.

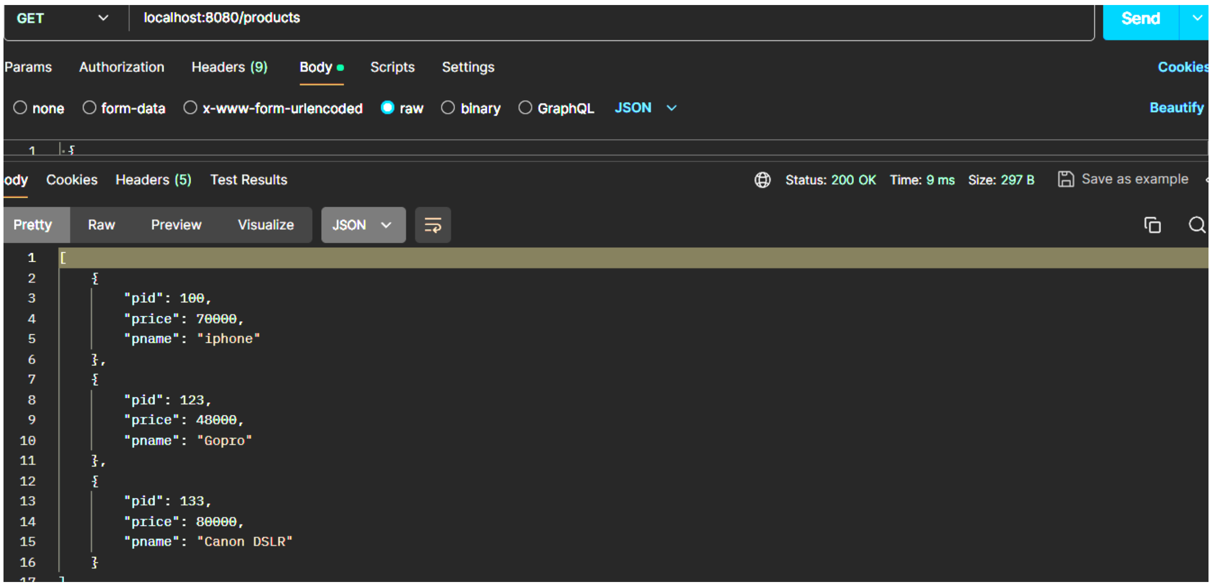
Once the product is found, it is removed from the list using its index.

This will allow us to obtain the product ID. Additionally, we have a method for removing products based on index and object. Since our method accepts arguments as index and objects as ID, we can modify the logic to obtain the ID.



```java
    public void deleteProduct(int prodId)  {
        int index = 0;
        for (int i = 0; i < products.size(); i++)
            if (products.get(i).getPid() == prodId)
                index = i;

        products.remove(index);
    }

}
```

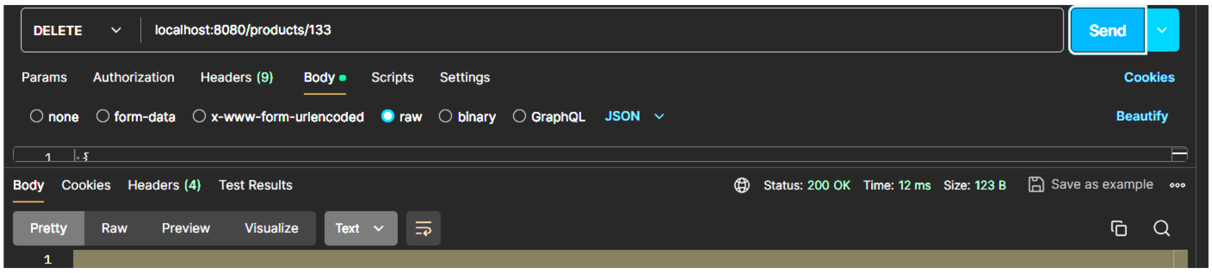
Back in the Postman tool, changing the request type to GET for displaying all lists of products



```
GET  localhost:8080/products
Body: raw  JSON
Status: 200 OK  Time: 9 ms  Size: 297 B
```

```json
[
    {
        "pid": 100,
        "price": 70000,
        "pname": "iphone"
    },
    {
        "pid": 123,
        "price": 48000,
        "pname": "Gopro"
    },
    {
        "pid": 133,
        "price": 80000,
        "pname": "Canon DSLR"
    }
```

and later again changing the request type to Delete, hitting the url: *localhost:8080/products/133*


DELETE localhost:8080/products/133

Send

Params Authorization Headers (9) Body Scripts Settings Cookies

none form-data x-www-form-urlencoded [✓ raw] binary GraphQL JSON Beautify

Body Cookies Headers (4) Test Results Status: 200 OK Time: 12 ms Size: 123 B Save as example

Pretty Raw Preview Visualize Text

We can see the output that our product got deleted with status code 200, i.e., our deletion got successful.

In the next chapter, we will learn about Spring Data JPA.

Happy learning and Coding 😊 👾….