Spring Boot web application with JDBC – Update Product

Overview

Updating an existing product is like creating and inserting a new product with one important difference existing products already have an id. Attempting to re-insert an existing product (using a SQL insert) will result in a primary key violation error.

You will implement the following steps to update an existing product

1. Select the product to update from the product list
    a. This will pass the product id as a parameter
    b. http://localhost:8080/**updateProduct/?id=1**
2. Display the update form (**/updateProduct**)
3. Fill in form values and **submit**
4. update the product in the database.

1. Show edit links in the product list

The goal is to add an edit button for each link. Clicking the button should call and pass the product id to **updateProduct()** in the controller. The result should look like this:

Id	Name	Description	Stock	Price	
1	Kettle	Steel Electric Kettle	97	€ 35.60	Edit
2	Fridge freezer	Fridge + freezer large	45	€ 799.00	Edit
3	Microsoft Surface Laptop 2	8GB ram, 512GB ssd	5	€ 1299.00	Edit

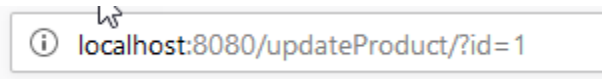
Add a new table column (**<td>** element) to the product row for the edit button and link:

```
    </tr>
    <!-- Insert a row for each product in the list -->
    <tr th:each="product : ${products}">
        <td th:text="${product.ProductId}"></td>
        <td th:text="${product.ProductName}">Product Name</td>
        <td th:text="${product.ProductDescription}">description</td>
        <td th:text="${product.ProductStock}">stock</td>
        <!-- Format the number with two decimal places -->
        <td class="text-right" th:text="${'€ ' + #numbers.formatDecimal(product.ProductPrice, 0, 2)}">price</td>
        <td><a th:href="@{'/updateProduct/?id=' + ${product.ProductId}}" class="btn-sm btn-danger" role="button">Edit</a></td>
    </tr>
</table> <!-- End table -->
```

Text version for copying:

```
<td><a th:href="@{'/updateProduct/?id=' + ${product.ProductId}}" class="btn-sm btn-danger" role="button">Edit</a></td>
```

The links generated will look like this:



2. Product DAO update method

Add a method to **ProductDao** and its implementation to perform the update

First **ProductDao** (the interface)

The create method accepts an existing product object parameter and returns the number of rows affected by the update (usually 1).

```
ProductDao.java ×
22      public Product create(final Product product);
23
24      // update an exiting product - return the number or fows affected
25      public int update(final Product product);
26
27      // Delete product with matching id
28      public int delete(int id);
```

The product implementation, **ProductDaoImpl.java**

The SQL statement required for update. This is an SQL statement with value placeholders (**?**)

```
private final String UPDATE_SQL = "UPDATE dbo.Product SET ProductName = ?, CategoryId = ?,
ProductDescription = ?, ProductStock = ?, ProductPrice = ? WHERE ProductId = ?";
```

The **update** method which will fill in the values and execute the query (**see code comments**)

This is more straight forward than an insert as the id value already exists.

```
// Update an existing product
    public int update(final Product product) {

        // Update product using values in product object passed as a parameter
        // As this is an update a new primary key id is not required

        // The quesry requires 6 parameters which will be passed as an object
        Object[] params = {
            product.getProductName(),
            product.getCategoryId(),
            product.getProductDescription(),
            product.getProductStock(),
            product.getProductPrice(),
            product.getProductId()
        };

        // execute the query using params, returning the number of rows affected
        return jdbcTemplate.update(UPDATE_SQL, params);
    }
```

3. A Controller Method to load the update Product page

This method will load a view named **updateProduct**. The view requires a product object (to be updated) and a list of categories for the form:

```
    // The newProduct page will be accessed using /updateProduct from the browser
    @RequestMapping(value = "/updateProduct", method = RequestMethod.GET)
    public String updateProduct(@RequestParam(name = "id", required = true) String pId, Model model) {

        Product product;
        // Initialise id (default value used to get all products)
        int id = 0;

        // The parameter may be not be valid - which could crash the application
        // This trys to parse the string converting it to an it
        // If successfull id will be assigned the cat value
        // Otherwise - catch any exception
        // If it fails (i.e an exception occurs) id value will not be changed (from 0).
        try {
            id = Integer.parseInt(pId);
         }
        catch(NumberFormatException e) {
            System.out.println("Bad input for id: " + e);
        }

        // If id is 0 then get all products otherwise get products for cat id
        if (id == 0) {
            // product id=0 does not exist - return to product list
            return "redirect:/products";
        } else {
            // Otherwise find the product matching the id
            product = productData.findById(id);
        }

        // add product to the model
        model.addAttribute("product", product);

        // Get a list of categories and add to the model
        List<Category> categories = categoryData.findAll();
        model.addAttribute("categories", categories);

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

4. The update product view (form)

The update form is almost identical to the existing **addProduct.html.** The same form could be shared for insert and update, but we will do it separately to keep things simple.

Make a copy of **newProduct.html**, naming the new file **updateProduct.html**. Then make the following changes:

1. Change the form action so that the form data will be submitted to **/updateProduct**

```
<div class="col-sm-9">
    <h3>Update Product</h3>
    <!-- https://getbootstrap.com/docs/4.0/components/forms/ -->
    <form th:object="${product}" th:action="@{/updateProduct}" method="post" class="needs-validation">
        <input type="hidden" th:field="*{ProductId}"/>
```

2. Change the text above the form to Update Product.
3. Add a cancel button – in addition to the submit button. This will link back to the products list without making changes.

```
        <button type="submit" class="btn btn-warning">Submit</button>
        <a href="/products" th:href="@{/products}" class="btn btn-primary" role="button">Cancel</a>
    </form>
</div> <!-- End Products col -->
```

5. Handling form submit

Back to the controller to handle the incoming form values (after it is submission).

Our new method to handle the form submit will re-use **`/updateProduct`** but for **`POST`** requests.

```
    // Handle form submit via HTTP POST
    @RequestMapping(value = "/updateProduct", method = RequestMethod.POST)
    // Form data will be supplied as a filled in Product object
    public String editProduct(Product product) {

        // Use the Dao to update the product
        // To do: check for errors and return to form if any found
        // https://www.journaldev.com/2668/spring-validation-example-mvc-validator
        int rows = productData.update(product);

        // output result in server side console
        System.out.println(rows + " rows were updated");

        // Redirect back to the products list
        return "redirect:/products";
    }
```

6. The completed form