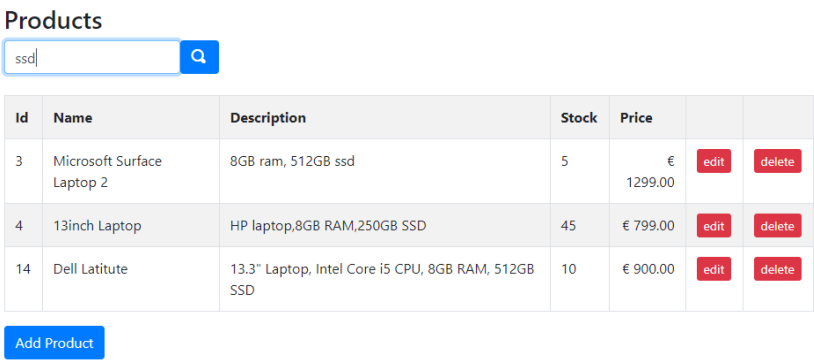
# Spring Boot web application with JDBC – Search Products

## Overview

The final tutorial will add a product search to the **`products.html`** view. Users will enter a search term which will then be used to search for products with matching name or description. The finished page will look like this:




## 1. Add a search form to the products page

The search input field and button, displayed above, are part of an HTML form – styled with BootStrap 4.

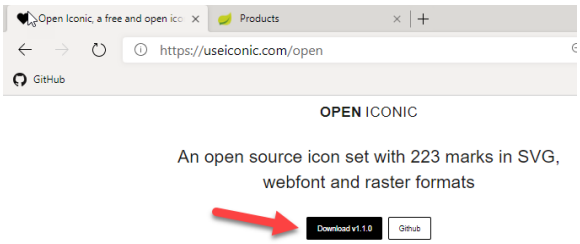
The HTML code should be added just above the product table.

1. The form is contained in a **`<div>`** element.
2. This form is not associated with an object like the others we have used.
    a. action will submit data to **`/searchProducts`**
3. The form method is **`get`** so data will be sent via URL parameters.
4. Styled as a BootStrap 4 **`form-inline`** so that the input and button will be on the same line.
5. The text input field is named **`search`** which will be attached to the URL.
    a. E.g. http://localhost:8080/searchProducts?search=ssd
6. The submit button, note the span class which assign a magnifying glass icon to the button

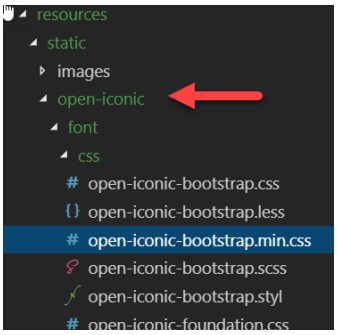
```
products.html
27          <!-- Column 2 - Product List -->
28          <div class="col-sm-10">
29
30              <h3>Products</h3>
31              <!-- Search box-->
32              <!-- mb-4 BootStrap class adds a bottom margin -->
33          1   <div class="mb-4"> <!-- https://getbootstrap.com/docs/4.0/utilities/spacing/ -->
34                  <!-- https://getbootstrap.com/docs/4.0/components/forms/ -->
35              2   <form th:action="@{/searchProducts}" method="get" class="form-inline needs-validation">  4
36                      <div class="form-group">
37                          5 <input id="search" name="search" type="text" class="form-control" placeholder="search products"  required>
38                      </div>
39                  6 <button type="submit" class="btn btn-primary"><span class="oi oi-magnifying-glass"></span></button>
40                  </form>
41              </div>
42
43              <th:block th:if="${message} != null">
```

## 1.1 The Search button Icon

The icon, s applied by **BootStrap 4** but not included by default. Icons are provided by open iconic and must downloaded and placed in the static folder. Download from https://useiconic.com/open




Unzip, rename the folder as **open-iconic** and move it to **resources/static** in your application

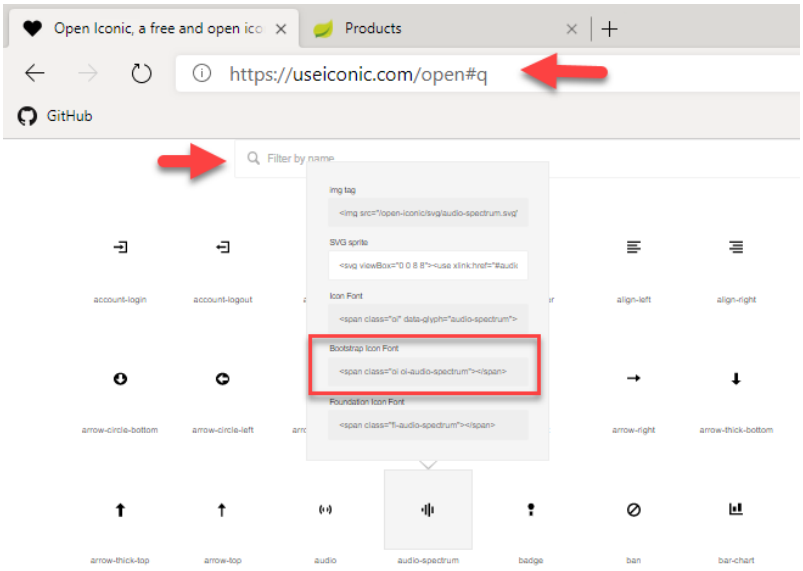



The icons can now be used in the application but fist must be linked in pages. Do this in the **htmlHead.html** fragment. Just after the BootStrap 4 Link


Copy code:

```
<link rel="stylesheet" th:href="@{/open-iconic/font/css/open-iconic-bootstrap.min.css}" >
```

Browse the full library of available icons at https://useiconic.com/open#q Click an icon to see the BootStrap code.



## 2. Product DAO search method

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

First **ProductDao** (the interface)

The **findBySearchText** method accepts a String parameter and returns a list of products matching that search (if found).

```
ProductDao.java ×
17
18      // return a list of products matching search term
19      public List<Product> findBySearchText(String searchText);
20
```

The product implementation, **ProductDaoImpl.java**

The SQL statement required for the search. It uses the like operator to compare the named parameter, **:search**, with **ProductName** and **ProductDescription** in the database.

```
// Search query - note named parameter
private final String SELECT_SQL_BY_SEARCH = "SELECT * FROM dbo.Product WHERE ProductName like
:search or ProductDescription like :search";
```

The **findBySearchText** method assigns the search value to **:search** and executes the query. Note that the search term is surrounded by **%** which is the wildcard value for TSQL.

Working with named parameters is a little different to the previous JDBC methods. The following Java imports are required (add at top of **ProductDaoImpl.java** with the existing imports.

```
import org.springframework.jdbc.core.namedparam.MapSqlParameterSource;
import org.springframework.jdbc.core.namedparam.NamedParameterJdbcTemplate;
```

### Read code comments for more detailds

```
// return a list of products matching search term
    public List<Product> findBySearchText(String searchText)  {

        // The named parameter template assigns values to the named parameters (as opposed to ?) in an SQL statement
        NamedParameterJdbcTemplate namedParamJdbcTemplate = new NamedParameterJdbcTemplate(jdbcTemplate);
        MapSqlParameterSource parameters = new MapSqlParameterSource();

        // Set the :search parameter
        // % is a wildcard - the search term will be used to match files using the like operator
        // https://www.w3schools.com/SQL/sql_like.asp
        parameters.addValue("search", "%" + searchText + "%");

        // execute the query with named parameters
        // use ProductMapper() to process the resultset and return the resulting product list
        return namedParamJdbcTemplate.query(SELECT_SQL_BY_SEARCH, parameters, new ProductMapper());
    }
```

## 3. A Controller Method to execute the search

The **`searcProducts`** method will accept the search parameter from the view (via HTTP get), call the DAO search method and pass the results back to the products view.

The method operation is like the products method.

```
   // This method displays the product page
   @RequestMapping(value = "/searchProducts", method = RequestMethod.GET)
   // Uses a Model instance - which will be passed to a view
   // cat parameter is for category id
   public String searchProducts(@RequestParam(name = "search", required = false, defaultValue = "") String search,
Model model) {

       // If search is blank then redirect to the products page
       if (search == "") {
           return "redirect:/products";
       }

       // Do the search and get the results
       List<Product> products = productData.findBySearchText(search);

       // Get all categories
       List<Category> categories = categoryData.findAll();

       model.addAttribute("products", products);
       model.addAttribute("categories", categories);

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

That's it! The method uses the existing **`products.html`** view without further modifications.