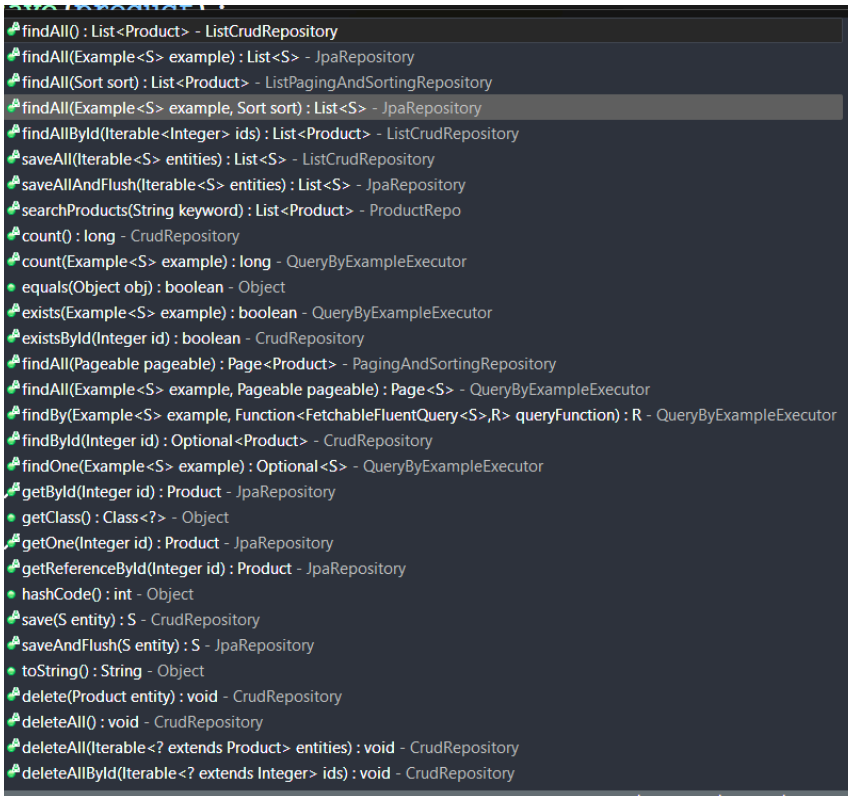
#28-Project using Spring Search Feature

Now we are going to implement the next functionality, which is the search feature in our project. We are using Spring Data JPA in the repository, which looks quite empty in the repository layer. As we are not defining or declaring any methods in it, we get all methods from the JPA repository. This is useful for simple CRUD operations, like searching for a product based on its primary key (ID). However, what if we want a method for searching a product based on fields like its name, category, or any other parameter? Let's check if the JPA repository provides any methods for this.

Methods in JPA Repository



```
findAll() : List<Product> - ListCrudRepository
findAll(Example<S> example) : List<S> - JpaRepository
findAll(Sort sort) : List<Product> - ListPagingAndSortingRepository
findAll(Example<S> example, Sort sort) : List<S> - JpaRepository
findAllById(Iterable<Integer> ids) : List<Product> - ListCrudRepository
saveAll(Iterable<S> entities) : List<S> - ListCrudRepository
saveAllAndFlush(Iterable<S> entities) : List<S> - JpaRepository
searchProducts(String keyword) : List<Product> - ProductRepo
count() : long - CrudRepository
count(Example<S> example) : long - QueryByExampleExecutor
equals(Object obj) : boolean - Object
exists(Example<S> example) : boolean - QueryByExampleExecutor
existsById(Integer id) : boolean - CrudRepository
findAll(Pageable pageable) : Page<Product> - PagingAndSortingRepository
findAll(Example<S> example, Pageable pageable) : Page<S> - QueryByExampleExecutor
findBy(Example<S> example, Function<FetchableFluentQuery<S>,R> queryFunction) : R - QueryByExampleExecutor
findById(Integer id) : Optional<Product> - CrudRepository
findOne(Example<S> example) : Optional<S> - QueryByExampleExecutor
getById(Integer id) : Product - JpaRepository
getClass() : Class<?> - Object
getOne(Integer id) : Product - JpaRepository
getReferenceById(Integer id) : Product - JpaRepository
hashCode() : int - Object
save(S entity) : S - CrudRepository
saveAndFlush(S entity) : S - JpaRepository
toString() : String - Object
delete(Product entity) : void - CrudRepository
deleteAll() : void - CrudRepository
deleteAll(Iterable<? extends Product> entities) : void - CrudRepository
deleteAllById(Iterable<? extends Integer> ids) : void - CrudRepository
```

As we can see, we don't have a method available for finding the product based on name, description, category, etc. So, let's work on creating a custom method capable of doing this, such as searching a product by brand name.

```
public List<Product> findByBrand(String brand) {
}
```

For such custom methods, we don't use normal SQL queries. Instead, we use JPQL (Java Persistence Query Language), where we use column names as fields or variables and table names as class names.

JPQL

The JPQL (Java Persistence Query Language) is an object-oriented query language used to perform database operations on persistent entities. Instead of using a database table, JPQL uses the entity object model to operate SQL queries. The role of JPA is to transform JPQL into SQL, providing an easy platform for developers to handle SQL tasks.

We will create a custom method, searchProducts, for the keyword. For this, we just need to annotate the method with @Query and write our query within it.

@Query

The @Query annotation is used for defining custom queries in Spring Data JPA. When you are unable to use the query methods to execute database operations, you can use @Query to write a more flexible query to fetch data.

- The @Query annotation supports both JPQL and native SQL queries.
- Use @Param in method arguments to bind query parameters.

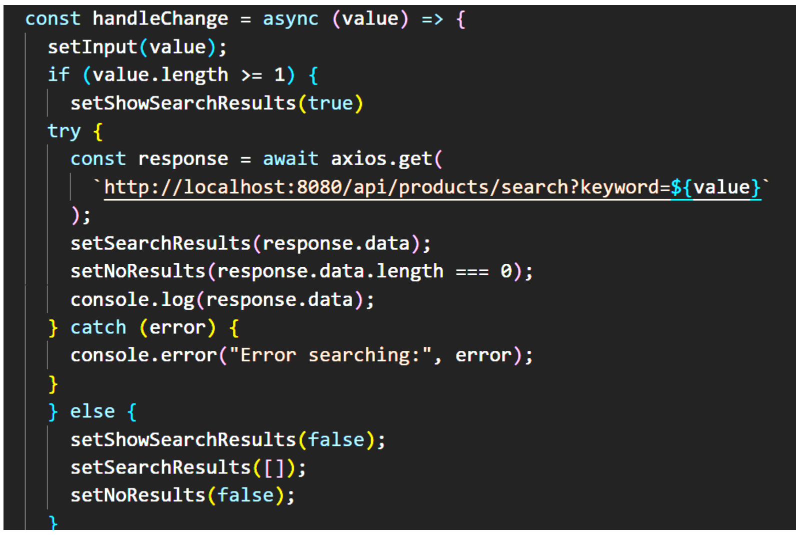
Frontend Changes

Before proceeding to the backend, let’s see what changes regarding the search have been made in our UI frontend part and which APIs we are going to use and handle.

For this chapter, we have updated the UI, so we need to follow the initialization and installation steps and then run the application. In the application code, under the Navbar.jsx component, which has been updated, the search box will manage the handleChange function every time you search for any keyword.

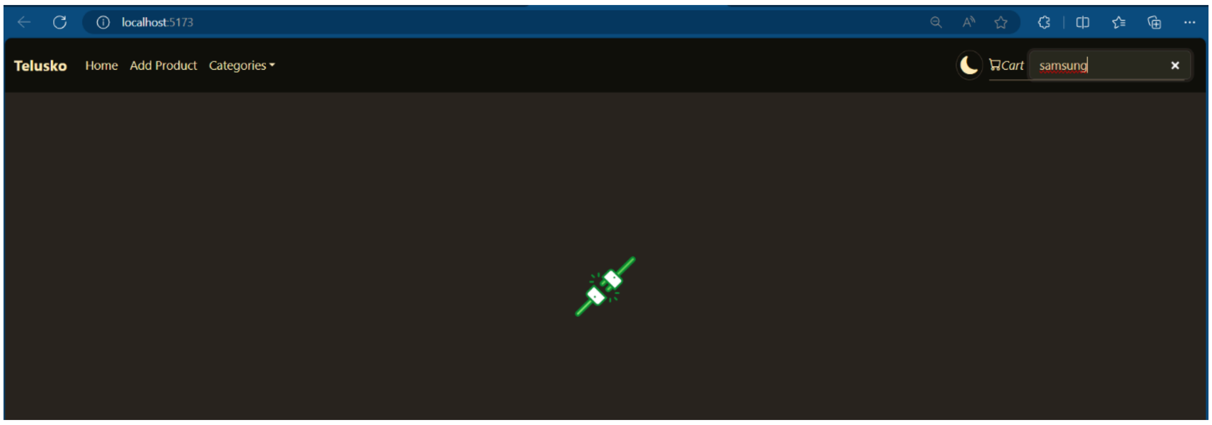
For every word entered, there will be suggestions popping up, making it user-friendly but increasing network calls and server load, as each keyword searched sends a request to the server and returns a response.

We will work with the URL***: localhost:8080/api/products/search?keyword=${value}.***



```
const handleChange = async (value) => {
  setInput(value);
  if (value.length >= 1) {
    setShowSearchResults(true)
  try {
    const response = await axios.get(
      `http://localhost:8080/api/products/search?keyword=${value}`
    );
    setSearchResults(response.data);
    setNoResults(response.data.length === 0);
    console.log(response.data);
  } catch (error) {
    console.error("Error searching:", error);
  }
  } else {
    setShowSearchResults(false);
    setSearchResults([]);
    setNoResults(false);
  }
```

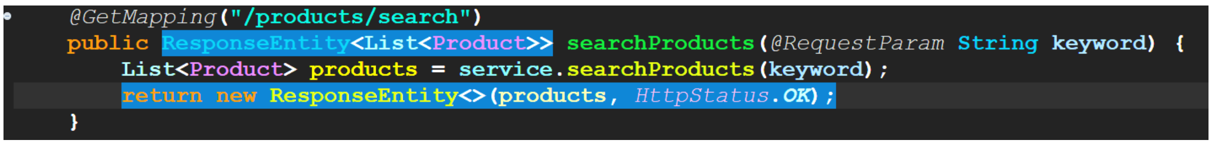
Run the application on port ***localhost:5173/.*** After loading the UI, you can see that the search box in the navbar is not completely functional, as our backend code is not ready, it cannot work yet.



Backend Implementation

Controller Layer

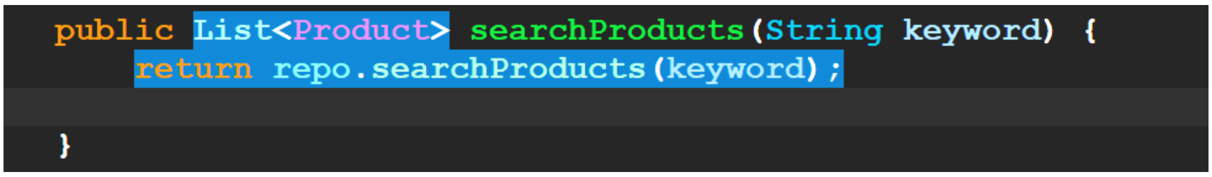
For searching, we will create a method searchProducts returning List<Products>, having a String argument as a keyword annotated with @RequestParam. This is mapped with @GetMapping having the URL ("products/search"), for which the service will provide the implementation, returning the ResponseEntity object with products and status code as "OK".



```
    @GetMapping("/products/search")
    public ResponseEntity<List<Product>> searchProducts(@RequestParam String keyword) {
        List<Product> products = service.searchProducts(keyword);
        return new ResponseEntity<>(products, HttpStatus.OK);
    }
```

Service Layer

Here, we will create the method searchProducts, which will also return List<Products> with a String type keyword argument for which the repository will search the product as we have already had some implementation for it.

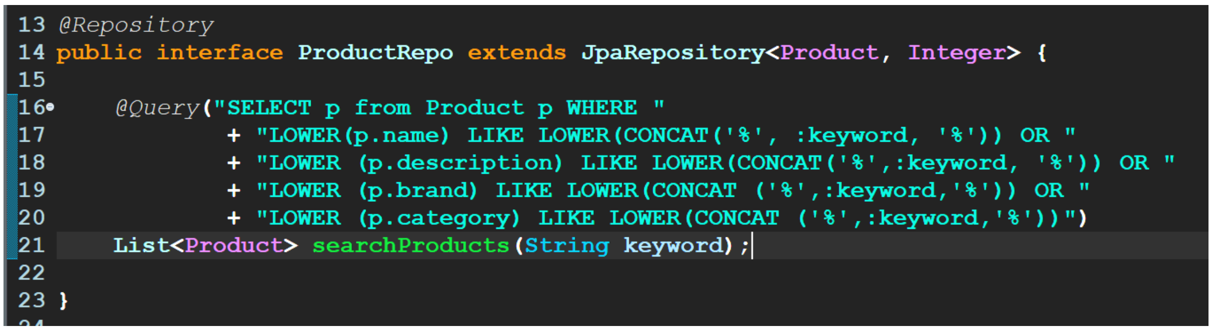


```java
public List<Product> searchProducts(String keyword) {
    return repo.searchProducts(keyword);

}
```

Repository Layer

Here, we will start our actual work for our custom method, searchProducts. In the @Query annotation, we will write our query using a JPQL query, not an SQL query.



```java
@Repository
public interface ProductRepo extends JpaRepository<Product, Integer> {

    @Query("SELECT p from Product p WHERE "
            + "LOWER(p.name) LIKE LOWER(CONCAT('%', :keyword, '%')) OR "
            + "LOWER (p.description) LIKE LOWER(CONCAT('%',:keyword, '%')) OR "
            + "LOWER (p.brand) LIKE LOWER(CONCAT ('%',:keyword,'%')) OR "
            + "LOWER (p.category) LIKE LOWER(CONCAT ('%',:keyword,'%'))")
    List<Product> searchProducts(String keyword);

}
```

Explanation of the Query in searchProducts Method

This query is used to search for products in a database based on a keyword that can match different fields of the Product entity.

ProductRepo: This is a repository interface that extends JpaRepository. It provides CRUD operations for the Product entity with a primary key of type Integer.

JpaRepository<Product, Integer>: This generic interface provides methods for CRUD operations and also allows defining custom queries.

@Query Annotation: This annotation is used to define a custom JPQL (Java Persistence Query Language) query. JPQL is similar to SQL but works with entity objects rather than directly with database tables.

Query Breakdown:

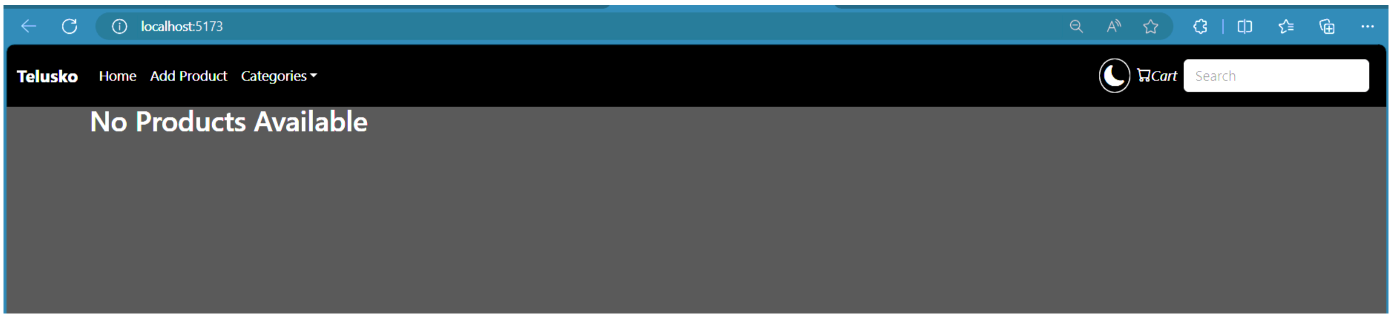
- SELECT p FROM Product p: This part of the query selects all products (p) from the Product entity.
- WHERE: The WHERE clause is used to filter the results based on certain conditions.
- The query checks multiple fields in the Product entity to see if they contain the keyword provided as a method argument.

Conditions Applied in Query:

- **LOWER(p.name) LIKE LOWER(CONCAT('%',:keyword, '%')):** This condition checks if the name field of the product (converted to lowercase for case-insensitive search) contains the keyword. The **CONCAT('%',:keyword, '%')** part ensures that the keyword can be found anywhere in the name (i.e., it can be at the beginning, middle, or end).
- Similar conditions are applied to description, brand, and category fields, allowing the search to match against any of these fields.

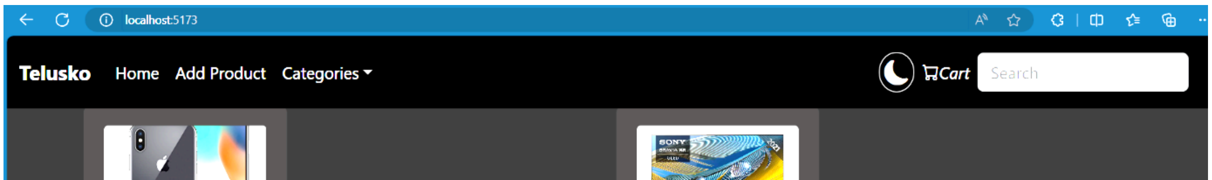
This query is designed to perform a flexible search across multiple fields (**name, description, brand, and category**) of the Product entity.

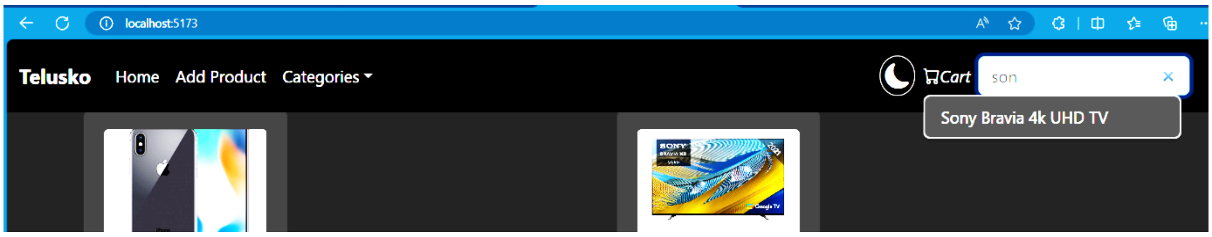
After completing the query, restart the application. Our server is running on Tomcat port 8080. Back to our frontend, which is already running, refresh the home page. As we have no products yet,

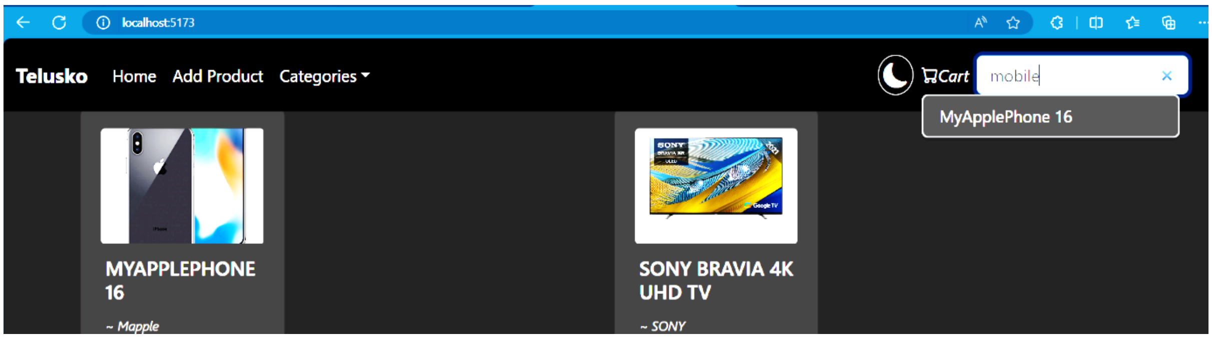


No Products Available

let's add a few products. After that, try to search with the initials of the product or category you want to search. You will find suggestions below, which look amazing with the UI, but a fully functional backend also plays an important role.







Looks good! We have learned this end-to-end project. I hope you have gained some knowledge from it. We will come up with more such projects in the future, so stay tuned.

Till then, keep learning and happy coding! ☺ 👾