# Part I

# Custom Repository Queries with forms

# 1

# Custom database queries

## 1.1 Search for exact property value (project `query01`)

Let's create a simple database schema for hardware products, and then write some forms to query this database.

Use `make:entity` to create a new Entity class Product, with the following properties:

- description: String
- price: Float
- category: String

Use `make:crud` to generate the CRUD pages for Product entities.

## 1.2 Fixtures

NOTE: You may need to add the ORM Fixtures library to this project:

```
composer req orm-fixtures --dev
```

Use `make:fixtures ProductFixtures` to create a fixtures class, and write fixtures to enter the following initial data:

```php
$p1 = new Product();
$p1->setDescription('bag of nails');
```

```
$p1->setPrice(5.00);
$p1->setCategory('hardware');
$manager->persist($p1);

$p2 = new Product();
$p2->setDescription('sledge hammer');
$p2->setPrice(10.00);
$p2->setCategory('tools');
$manager->persist($p2);

$p3 = new Product();
$p3->setDescription('small bag of washers');
$p3->setPrice(3.00);
$p3->setCategory('hardware');
$manager->persist($p3);
```

Now migrate your updated Entity structure to the database and load those fixtures. Figure 1.1 shows the list of products you should visiting the `/product` route.

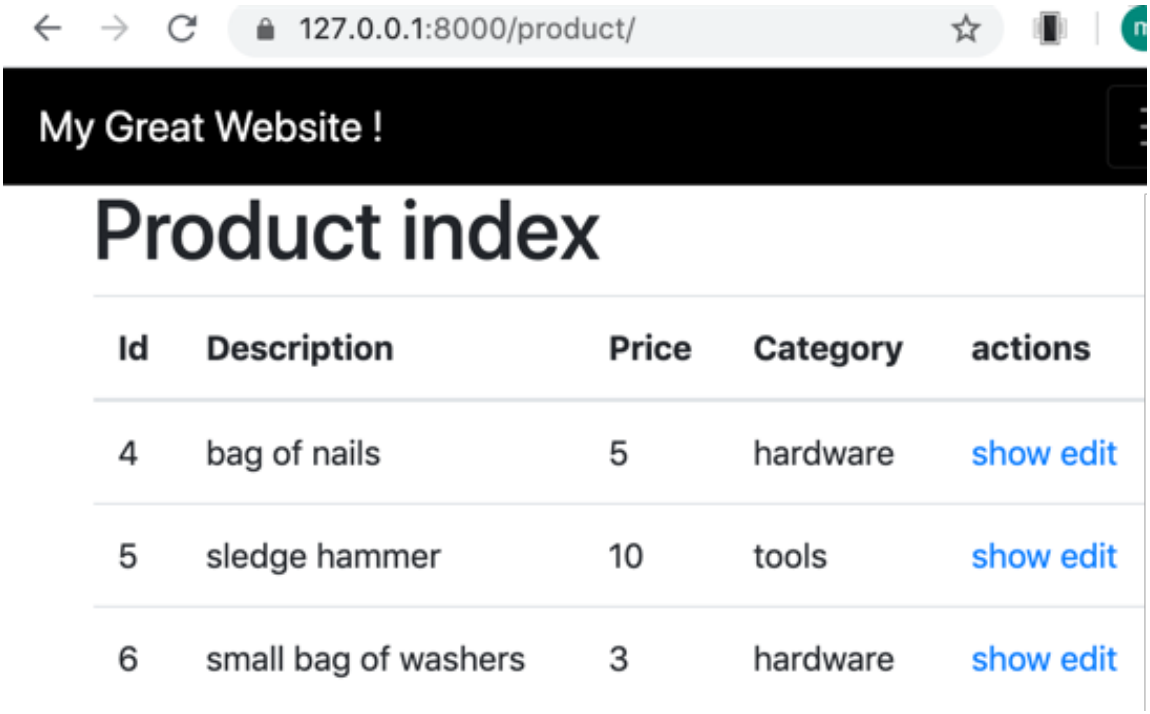


Figure 1.1: Animated hamburger links for narrow browser window.

## 1.3 Add new route and controller method for category search

Add a new method to the `ProductController` that has the URL route pattern `/product/category/{category}`. We'll name this method `categorySearch(...)` and it will allow us to refine the list of products to only those with the given `Category` string:

```
/**
 * @Route("/category/{category}", name="product_search", methods={"GET"})
 */
public function search($category): Response
{

    $productRepository = $this->getDoctrine()->getRepository('App:Product');
    $products =  $productRepository->findByCategory($category);

    $template = 'product/index.html.twig';
    $args = [
        'products' => $products,
        'category' => $category
    ];

    return $this->render($template, $args);
}
```

First, we are getting a string from the URL that follows `/product/category/`. **All** routes defined in the CRUD generated `ProductController` are prefixed with `/product`, due to the annoation comment that is declared **before** the class declaration:

```
/**
 * @Route("/product")
 */
class ProductController extends AbstractController
{
    ... controller methods here ...
}
```

Whatever appears **after** `/product/category/` in the URL will be put into variabled `$category` by the Symfony routing system, because of the `Route` anntation comment:

```
/**
 * @Route("/category/{category}", name="product_search", methods={"GET"})
 */
```

We get a reference to an object that is an instance of the `ProductRepository` from this line:

```
$productRepository = $this->getDoctrine()->getRepository('App:Product');
```

We could get an array `products` of **all** `Product` objects from the database by writing:

```
$products =  $productRepository->findByCategory($category);
```

But Doctrine repository classes also give us free **helper** methods, that provide `findBy` and `findOneBy` methods for the properties of an Entity class. Since Entity class `Product` has a property `name`, then we get for free the Doctrine query method `findByName(...)` to which we can pass a value of `name` to search for. So we can get the array of `Product` objects whose `name` property matches the paramegter `category` as follows:

```
$products =  $productRepository->findByCategory($category);
```

Finally, we'll pass both the `$products` array, and the text string `$category` as variables to the `index` list Products Twig template:

```
$template = 'product/index.html.twig';
$args = [
    'products' => $products,
    'category' => $category
];

return $this->render($template, $args);
```

## 1.4 Aside: How to the free 'helper' Doctrine methods work?

PHP offers a runtime code reflection (or interpreter pre-processing if you prefer), that can intercept calls to non-existent methods of a class. We use the special **magic** PHP method `__call(...)` which expects 2 parameters, one for the non-existent method name, and one as an array of argument values passed to the non-existent method:

```
public function __call($methodName, $arguments)
{
    ... do something with $methodName and $arguments
}
```

Here is a simple class (put it in `/src/Util/ExampleRepository.php` in you want to try this) that demonsrtates how Doctrine uses '___call' to identify which Entity property we are trying to query by:

```
<?php
namespace App\Util;
```

```
/*
 * class to demonstrate how __call can be used by Doctrine repositories ...
 */
class ExampleRepository
{
    public function findAll()
    {
        return 'you called method findAll()';
    }

    public function __call($methodName, $arguments)
    {
        $html = '';
        $argsString = implode(', ', $arguments) . "\n";

        $html .= "you called method $methodName\n";
        $html .= "with arguments: $argsString\n";

        $result = $this->startsWithFindBy($methodName);
        if($result){
            $html .= "since the method called started with 'findBy'"
            . "\n it looks like you were searching by property '$result'\n";
        }

        return $html;
    }

    private function startsWithFindBy($name)
    {
        $needle = 'findBy';
        $pos = strpos($name, $needle);

        // since 0 would evaluate to FALSE, must use !== not simply !=
        if (($pos !== false) && ($pos == 0)){
            return substr($name, strlen($needle)); // text AFTER findBy
        }

        return false;
    }
}
```

You could add a new method to the `DefaultController` class to see this in action as follows:

```
/**
 * @Route("/call", name="call")
 */
public function call()
{
    // illustrate how __call workds
    $exampleRepository = new ExampleRepository();

    $html = "<pre>";
    $html .=  "----- calling findAll() -----\n";
    $html .= $exampleRepository->findAll();

    $html .=  "\n\n----- calling findAllByProperty() -----\n";
    $html .= $exampleRepository->findByName('matt', 'smith');

    $html .=  "\n----- calling badMethodName() -----\n";
    $html .= $exampleRepository->badMethodName('matt', 'smith');

    return new Response($html);
}
```

See Figure 1.2 shows the `ExampleRepository` output you should visiting the `/call` route. We can see that:

- a call to `findAll()` works fine, since that is a defined public method of the class
- a call to `findByName(...)` would work fine, since we can use `__call(...)` to identify that this was a call to a helper `findBy<property>(...)` method
  - and we could add logic to check that this is a property of the Entity class and build an appropriate query from the arguments
- a call to `badMethodName(...)` is caught by `__call(...)`, but fails our test for starting with `findBy`, and so we can ignore it
  - or log error or throw Exception or whatever our program spec says to do in these cases…

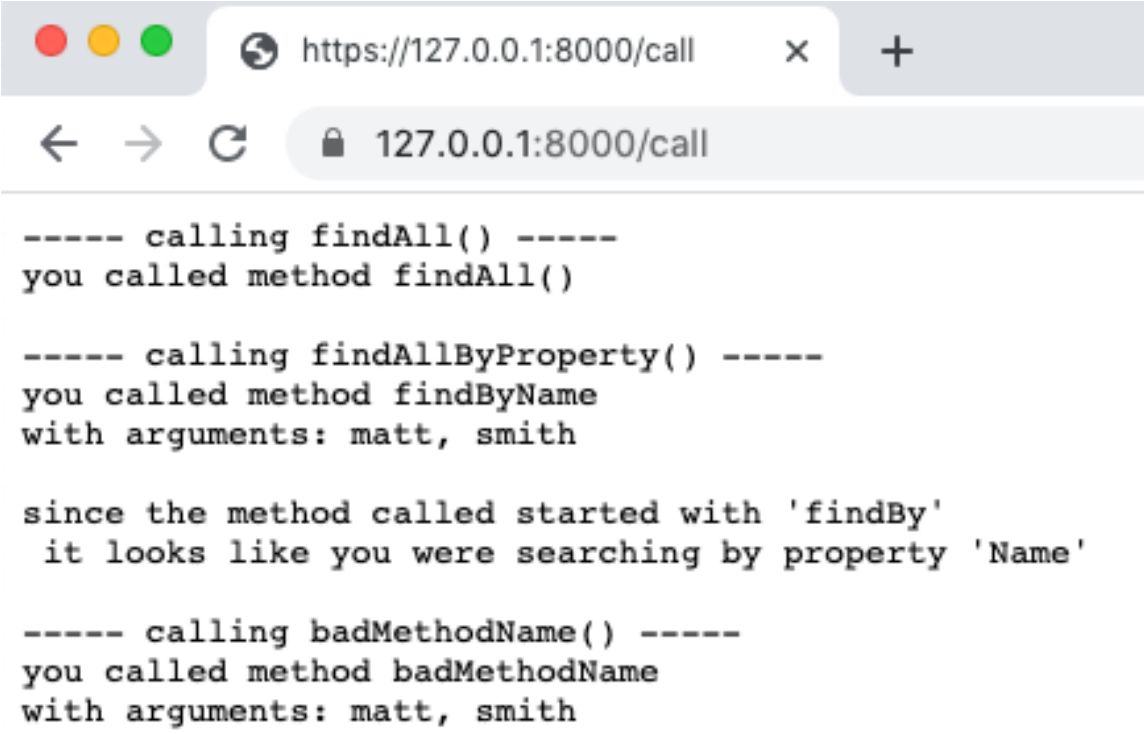


Figure 1.2: Output from our ExampleRepository `__call` demo.

## 1.5 Testing our search by category

If we now visit `/products/category/tools` we should see a list of only those Products with category = `tools`. See Figure 1.3 for a screenshot of this.

Likewise, for `/products/category/hardware` - see Figure 1.4.

If we try to search with a value that does not appear as the `category` String property for any Products, no products will be listed. See Figure 1.5.

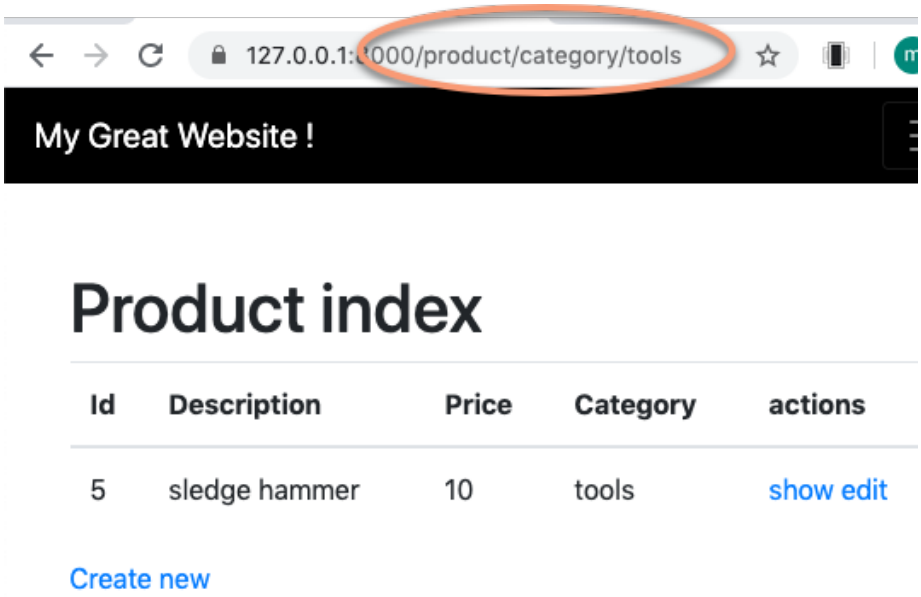


Figure 1.3: Only `tools` Products.

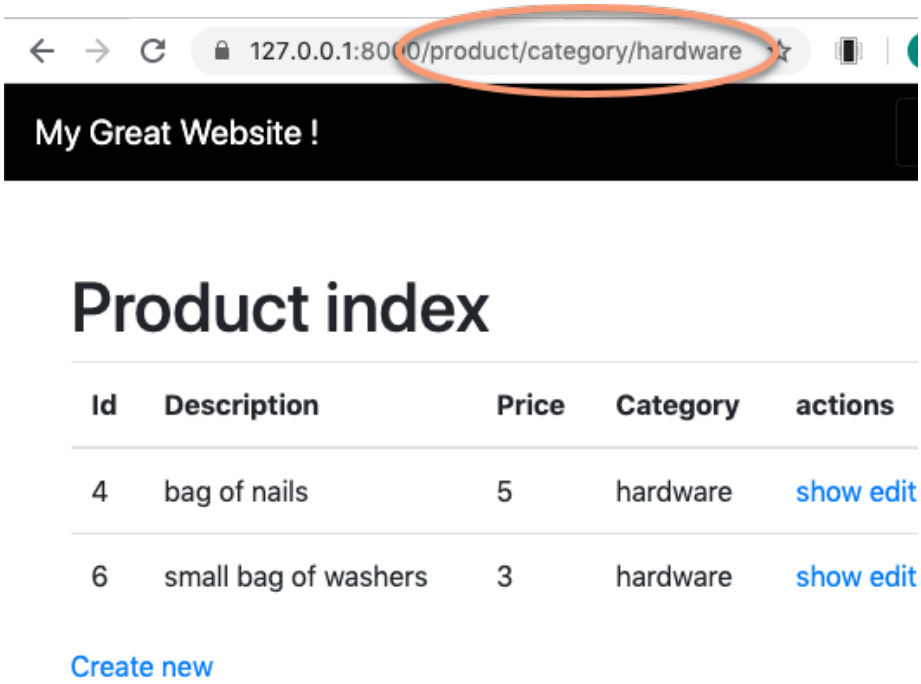


Figure 1.4: Only `hardware` Products.

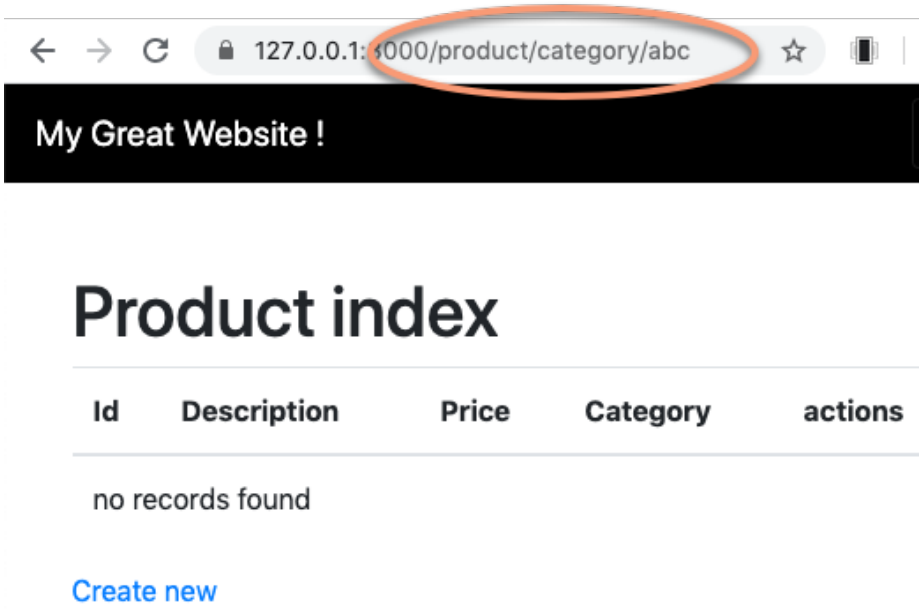


Figure 1.5: Only abc Products (i.e none!).

# 2

# Custom database queries

## 2.1 Search Form for exact property value (project `query02`)

Searching by having to type values in the URL isn't ideal. So let's add an HTML form in the list of projects page, allowing users to enter the category that way. Figure 2.1illustrates what we are going to create.

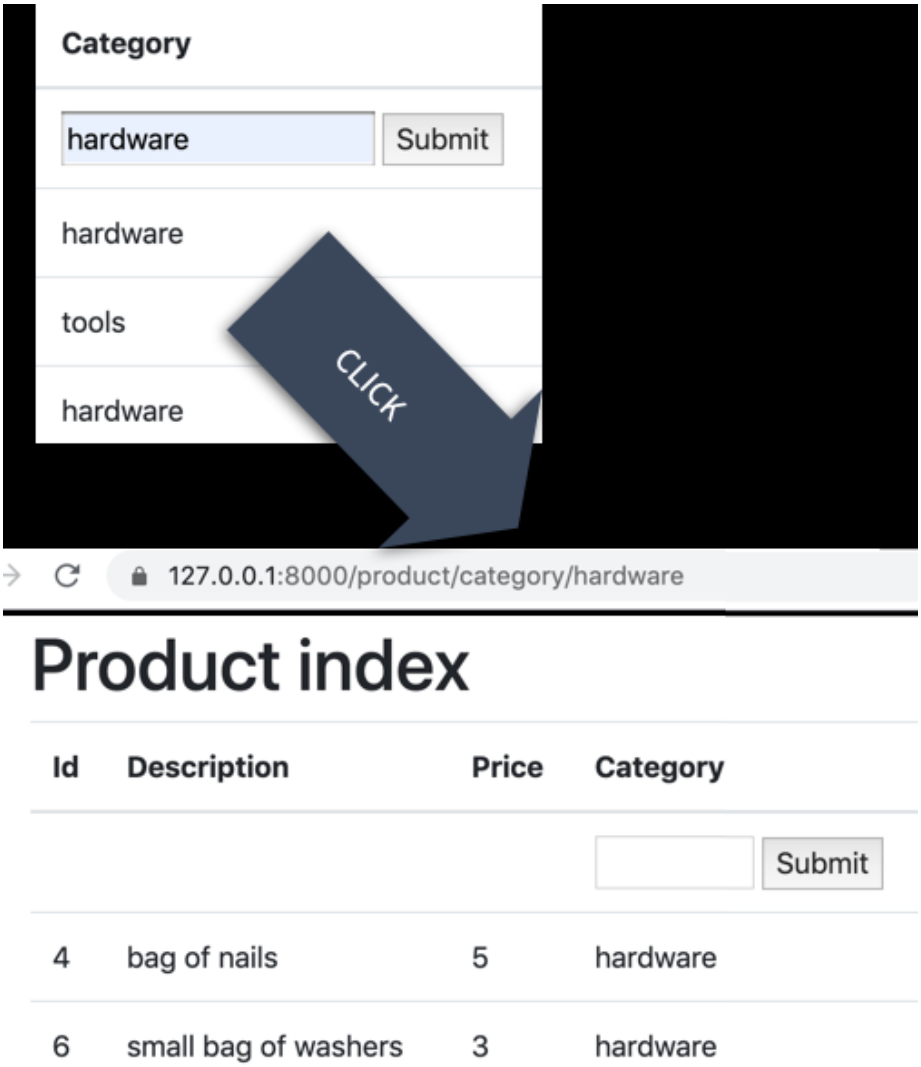


Figure 2.1: Form to search for category.

## 2.2 The form in Twig template

Let's write the HTML code for the submission form for the Products list Twig template in `/templates/product/index.html.twig`.

At present we have a table `<thead>` with a row of column headers, and then a loop for each Product:

```
{% extends 'base.html.twig' %}

{% block title %}Product index{% endblock %}

{% block body %}
    <h1>Product index</h1>

    <table class="table">
        <thead>
            <tr>
                <th>Id</th>
                <th>Description</th>
                <th>Price</th>
                <th>Category</th>
                <th>actions</th>
            </tr>
        </thead>
        <tbody>

<<<< TABLE ROW WITH FORM TO GO HERE >>>>

        {% for product in products %}
            <tr>
                <td>{{ product.id }}</td>
```

We need to add a new table row between the table headers and the loop of Products:

```
<tr>
    <th></th>
    <th></th>
    <th></th>
    <th>
        <form action="{{ url('search_category') }}" method="post">
            <input name="category">
            <input type="submit">
        </form>
```

```
        </th>
        <th></th>
    </tr>

    {% for product in products %}
        <tr>
            ... as before
```

The row has empty cells, except for the 4th cell (the Category column), where we create a simple form. The form has:

- a method of `post`
- an action of `url('search_category')`
  - we'll have to create this new route in the `ProductController` to process submission of this form
- a text box named `category`
  - since this text box will appear in the Categorty column, we don't need to give a text prompt
  - the HTML default `<input>` type is `text`, so we don't need to specify this either
- a Submit button

## 2.3 Controller method to process form submission

Here is the new method in `ProductController` to process submission of this form - implementing the route `search_category`:

```
/**
 * @Route("/searchCategory", name="search_category", methods={"POST"})
 */
public function searchCategory(Request $request): Response
{
    $category = $request->request->get('category');

    if(empty($category)){
        return $this->redirectToRoute('product_index');
    }

    return $this->redirectToRoute('product_search', ['category' => $category]);
}
```

The annotation comments specify the URL route `/searchCategory`, the internal route name `search_category`, and that we expect the request to be submitted using the `POST` method:

```
/**
 * @Route("/searchCategory", name="search_category", methods={"POST"})
 */
```

We need to extract the `category` variable submitted in the HTTP Request, so we need access to the Symfony `Request` object. The simnplest way to get a reference to this object is via the Symfony **param converter**, by adding `(Request $request)` as a method parameter. This means we now have Request object variable `$request` available to use in our method:

```
public function searchCategory(Request $request): Response
```

We can retrieve a value from the submitted `POST` variables int the request using the `get->` method naming the variable `category`. NOTE: In this instance `get` is a **getter** (accessor method) - not to be confused with the HTTP `GET` Request method…

```
$category = $request->request->get('category');
```

Finally, we can do some logic based on the value of form submitted variable `$category`. If this variable is an emptuy string, let's just redirect Symfonhy to run the method to list all products, route `product_index`:

```
if(empty($category)){
    return $this->redirectToRoute('product_index');
}
```

If `$category` was **not** empty then we can redirect to our category search route, passing the value to this route:

```
return $this->redirectToRoute('product_search', ['category' => $category]);
```

## 2.4 Getting rid of the URL search route

If we no longer wanted the URL search route, we could replace the final statement in our `searchCategory(...)` method to the following (and remove method `search(...)` altogether):

```
/**
 * @Route("/searchCategory", name="search_category", methods={"POST"})
 */
public function searchCategory(Request $request): Response
{
    $category = $request->request->get('category');

    if(empty($category)){
```

```
        return $this->redirectToRoute('product_index');
    }

    // if get here, not empty - so use value to search...
    $productRepository = $this->getDoctrine()->getRepository('App:Product');
    $products =  $productRepository->findByCategory($category);

    $template = 'product/index.html.twig';
    $args = [
        'products' => $products,
        'category' => $category
    ];

    return $this->render($template, $args);
}
```

# 3

# Wildcard vs. exact match queries

## 3.1 Search Form for partial description match (project `query03`)

Let's add a query form in the **description** column, so we need to edit Twig template `/templates/product/index.html.twig`.

So we add a new search form in the second table header row, for internal route name `search_description`, and passing form variable `keyword`:

```
<tr>
    <th></th>
    <th>
        <form action="{{ url('search_description') }}" method="post">
            <input name="keyword">
            <input type="submit">
        </form>
    </th>
    <th></th>
    <th>
        <form action="{{ url('search_category') }}" method="post">
            <input name="category">
            <input type="submit">
```

```
            </form>
        </th>
        <th></th>
    </tr>
```

Let's write the controller method to process our keyword form submission - edit `/src/ProductController.php` and add a new method:

```
    /**
     * @Route("/searchDescription", name="search_description", methods={"POST"})
     */
    public function searchDescription(Request $request): Response
    {
        $keyword = $request->request->get('keyword');

        if(empty($keyword)){
            return $this->redirectToRoute('product_index');
        }

        // if get here, not empty - so use value to search...
        $productRepository = $this->getDoctrine()->getRepository('App:Product');
        $products =  $productRepository->findByDescription($keyword);

        $template = 'product/index.html.twig';
        $args = [
            'products' => $products,
            'keyword' => $keyword
        ];

        return $this->render($template, $args);
    }
```

The above is just like our category search - but does only work for an exact match of `keyword` with the value of the `description` property.

What we want is to implement something similar the SQL `LIKE "%wildcard%"` query, where a word **anywhere** in the text of the `description` property will be matched.

## 3.2 Customer queries in our Repository class

The solution is to write a custom query method `findByLikeDescription($keyword)` in our `ProductRepository` class as follows:

```
...

class ProductRepository extends ServiceEntityRepository
{
    ...

    /**
     * @return Product[] Returns an array of Drill objects
     */
    public function findByLikeDescription($keyword)
    {
        return $this->createQueryBuilder('p')
            ->andWhere('p.description LIKE :keyword')
            ->setParameter('keyword', "%$keyword%")
            ->getQuery()
            ->getResult()
            ;
    }
}
```

We can now use this method in our `ProductController` controller method `searchDescription(..)`:

```
// if get here, not empty - so use value to search...
$productRepository = $this->getDoctrine()->getRepository('App:Product');

$products =  $productRepository->findByLikeDescription($keyword);
```

See Figure 3.1 illustrates a wildcard search for any `Product` with description containing texgt `bag`.

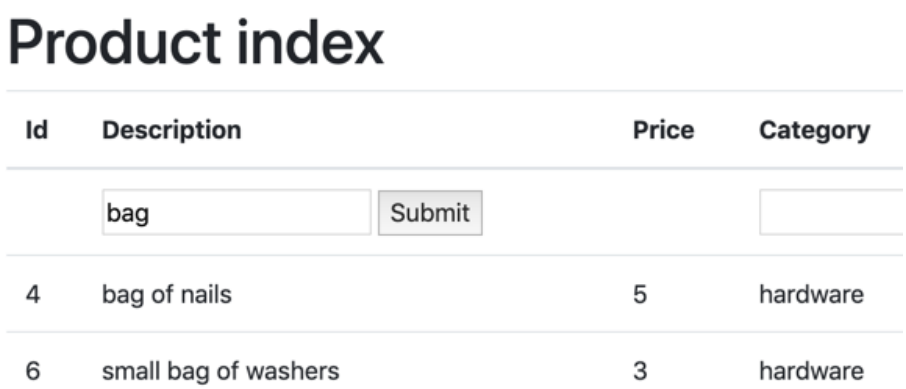


Figure 3.1: Form to wildcard search for description.

## 3.3 Making wildcard a sticky form

You may have noticed that in `ProductController` method `searchDescription(...)` we are passing the value of `$keyword` as well as the array `$products` to our Twig template.

This means that when the Product index page is called from our search method there will be an extra Twig variable `keyword` defined, which we can detect and use as a default value for our search form - so the user can **see** the wildcard value for which we are seeing a list of products:

```
<form action="{{ url('search_description') }}" method="post">
    <input name="keyword"
           {% if keyword is defined %}
           value="{{ keyword }}"
           {% endif %}
    >
    <input type="submit">
</form>
```

If there is no `keyword` Twig variable (it is not defined), then we don't add a `value` attribute to this form input.

NOTE: There is a difference between a variable existing and containing `NULL` versus no such variable being defined at all - ensure you write the correct test in Twig to distinguish between these differences…