

# Search By Keyword



# Search for Products by Keyword

The screenshot shows a web-based e-commerce application. At the top, there is a navigation bar with the logo "luv2shop" on the left, a search bar containing "Search for products...", a "Search" button, a user icon with "19.22" and a notification count of "2", and a shopping cart icon with a count of "2". Below the navigation bar, there is a sidebar on the left with categories: Books, Coffee Mugs, Mouse Pads, and Luggage Tags. The main content area features a large green banner in the center with the text "Search for products by keyword". Below this banner, there are four product cards in the top row and four in the bottom row, each with a small image, the product name, and a price. Each product card also has an "Add to cart" button.

Category	Product Name	Price	Add to cart
Books	Crash Course in Python	\$14.99	Add to cart
	JavaScript Cookbook	\$23.99	Add to cart
Books	Beginners Guide to SQL	\$14.99	Add to cart
	Advanced Techniques in JavaScript	\$16.99	Add to cart
Books	Crash Course in Big Data	\$18.99	Add to cart
	Advanced Techniques in Big Data	\$13.99	Add to cart

# Development Process

Step-By-Step

1. Modify Spring Boot app - Add a new search method
2. Create new component for search
3. Add new Angular route for searching
4. Update SearchComponent to send data to search route
5. Enhance ProductListComponent to search for products with ProductService
6. Update ProductService to call URL on Spring Boot app

# Step 1: Modify Spring Boot app - Search Method

- Spring Data REST and Spring Data JPA supports "query methods"
- Spring will construct a query based on method naming conventions

File: ProductRepository.java

Magic!

```
public interface ProductRepository extends JpaRepository<Product, Long> {  
    Page<Product> findByNameContaining(@RequestParam("name") String name, Pageable pageable);  
    ...  
}
```

# Step 1: Modify Spring Boot app - Search Method

- Find products based on name

File: ProductRepository.java

```
public interface ProductRepository extends JpaRepository<Product, Long> {  
    Page<Product> findByNameContaining(@RequestParam("name") String name, Pageable pageable);  
    ...  
}
```



"Containing" ... similar to SQL: "LIKE"

# Step 1: Modify Spring Boot app - Search Method

File: ProductRepository.java

```
public interface ProductRepository extends JpaRepository<Product, Long> {  
    Page<Product> findByNameContaining(@RequestParam("name") String name, Pageable pageable);
```

Behind the scenes, Spring will execute a query similar to this

```
SELECT * FROM Product p  
WHERE  
p.name LIKE CONCAT( '%', :name , '%' )
```

# Step 1: Modify Spring Boot app - Search Method

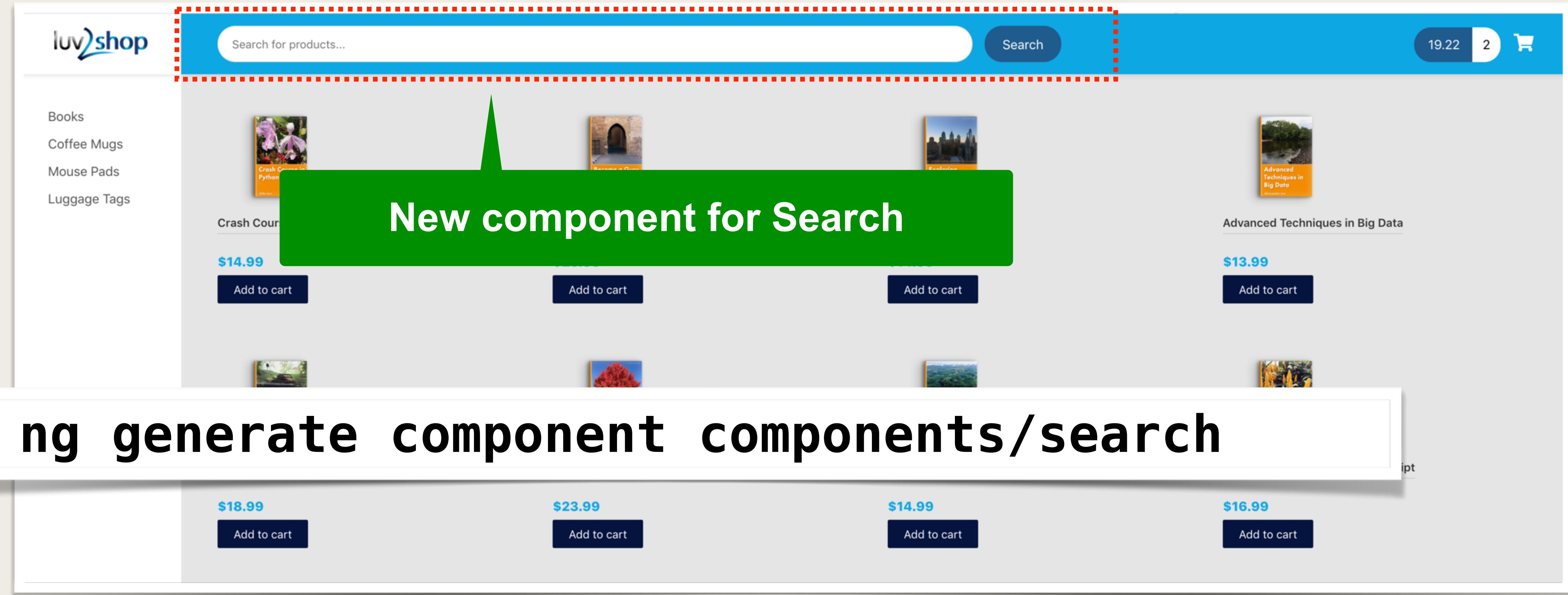
File: ProductRepository.java

```
public interface ProductRepository extends JpaRepository<Product, Long> {  
    Page<Product> findByNameContaining(@RequestParam("name") String name, Pageable pageable);  
    ...  
}
```

<http://localhost:8080/api/products/search/findByNameContaining?name=Python>

To pass data to REST API

# Step 2: Create new component for Search



```
> ng generate component components/search
```

# Step 3: Add new Angular route for searching

File: app.module.ts

```
...  
  {path: 'search/:keyword', component: ProductListComponent},  
...
```

Parameter

# Quick Discussion

## Event Binding

# Quick Discussion - Event Binding



# Event Binding

- In Angular, you can listen for events with "event binding"
- In other languages / frameworks, also known as "Event handler"

```
<button (click)="doMyCustomWork()">Search</button>
```

Listen for "click" event

Call a method in our Angular component code

Can be any method name we define

# Event Binding - Example

Call a method in our Angular component code

```
<button (click)="doMyCustomWork()">Search</button>
```

Listen for "click" event

```
export class SearchComponent implements OnInit {  
  doMyCustomWork() {  
    console.log(`Hey! You pushed my button!`);  
  }  
  ...  
}
```

# Reading User Input

# symbol  
Template reference variable

Provides access to the element

```
<input #myInput type="text"  
      (keyup.enter)="doMyCustomWork(myInput.value)" />
```

Listen for the "enter" key

The text the user typed in

```
export class SearchComponent implements OnInit {  
  
  doMyCustomWork(info: string) {  
    console.log(`Hey! This is your data: ${info}`);  
  }  
  ...  
}
```

# Reading User Input

```
<input #myInput type="text"  
       (keyup.enter)="doMyCustomWork(myInput.value)" />
```

```
<button (click)="doMyCustomWork(myInput.value)">Search</button>
```

Listen for "click" event

```
export class SearchComponent implements OnInit {  
  
  doMyCustomWork(info: string) {  
    console.log(`Hey! This is your data: ${info}`);  
  }  
  ...  
}
```

# symbol  
Template reference variable

Provides access to the element

The text the user typed in

# Other Events

Name	Description
<b>focus</b>	An element has received focus
<b>blur</b>	An element has lost focus
<b>keyup</b>	Any key is released. For a specific key, the <b>enter</b> key, use: <b>keyup.enter</b>
<b>keydown</b>	Any key is pressed
<b>dblclick</b>	The mouse is clicked twice on an element
<i>more ....</i>	

<https://developer.mozilla.org/en-US/docs/Web/Events>

# Step 4: Update SearchComponent to send data to search route

The screenshot shows a web application interface for a shop. At the top, there's a navigation bar with the logo "luv2shop", a search bar containing "Search for products...", a "Search" button, a user icon showing "19.22" and "2", and a shopping cart icon. Below the navigation, there's a sidebar with links to "Books", "Coffee Mugs", "Mouse Pads", and "Luggage Tags". The main content area displays a grid of products. One product is highlighted with a red dashed box around its search bar area. A large purple callout box contains a numbered list of steps: 1. User enters search text, 2. Clicks Search button, 3. SearchComponent has a click handler method, 4. Read search text, 5. Route the data to the "search" route, and 6. Handled by the ProductListComponent. An orange arrow points from the final step to a dark blue callout box on the right that says "To reuse the logic and view for listing products".

1. User enters search text
2. Clicks Search button
3. SearchComponent has a click handler method
4. Read search text
5. Route the data to the "search" route
6. Handled by the ProductListComponent

To reuse the logic and view for listing products

# Step 4: Update SearchComponent to send data to search route

File: search.component.html

```
<div class="form-header">

  <input #myInput type="text"
    placeholder="Search for products . . ."
    class="au-input au-input-xl"
    (keyup.enter)="doSearch(myInput.value)" />

  <button (click)="doSearch(myInput.value)" class="au-btn-submit">
    Search
  </button>

</div>
```

# Step 4: Update SearchComponent to send data to search route

The diagram illustrates the flow of data from the component's logic to its presentation. A red arrow points from the 'doSearch' method in the component code to the 'click' event handler in the template. Another red arrow points from the 'value' binding in the template back to the 'doSearch' method in the component. A callout box highlights the 'doSearch' method as a 'Method defined in our component'.

**File: search.component.ts**

```
export class SearchComponent implements OnInit {  
  constructor(private router: Router) { }  
  
  doSearch(value: string) {  
    console.log(`value=${value}`);  
    this.router.navigateByUrl(`/search/${value}`);  
  }  
  
  ...  
}
```

**Method defined in our component**  
**We can give it any name**

**File: search.component.html**

```
<div class="form-header">  
  <input #myInput type="text"  
        placeholder="Search for products ..."  
        class="au-input au-input-xl"  
        (keyup.enter)="doSearch(myInput.value)" />  
  <button (click)="doSearch(myInput.value)" class="au-btn-submit">  
    Search  
  </button>  
</div>
```

**Route the data to our "search" route**  
**It will be handled by the ProductListComponent**

**To reuse the logic and view for listing products**

# Step 5: Enhance component to search for products with product service

File: product-list.component.ts

```
...
const theKeyword: string = this.route.snapshot.paramMap.get('keyword');

// now search for the products using keyword
this.productService.searchProducts(theKeyword).subscribe(
  data => {
    this.products = data;
  }
)
...
```

File: search.component.ts

```
export class SearchComponent implements OnInit {
  constructor(private router: Router) { }

  doSearch(value: string) {
    console.log(`value=${value}`);
    this.router.navigateByUrl(`/search/${value}`);
  }
}
```

Passed in from  
SearchComponent

# Step 6: Update product service to call URL on Spring Boot app

File: product.service.ts

```
...
searchProducts(theKeyword: string): Observable<Product[]> {
    // need to build URL based on the keyword
    const searchUrl = `${this.baseUrl}/search/findByNameContaining?name=${theKeyword}`;

    return this.httpClient.get<GetResponseProducts>(searchUrl)
        .pipe(
            map(response => response._embedded.products));
}

...
interface GetResponseProducts {
    _embedded: {
        products: Product[];
    }
}
```

Call REST API

URL for searching products

Unwraps the JSON from  
Spring Data REST  
\_embedded entry

Returns an observable  
Map the JSON data from  
Spring Data REST  
to Product array