Search By Category

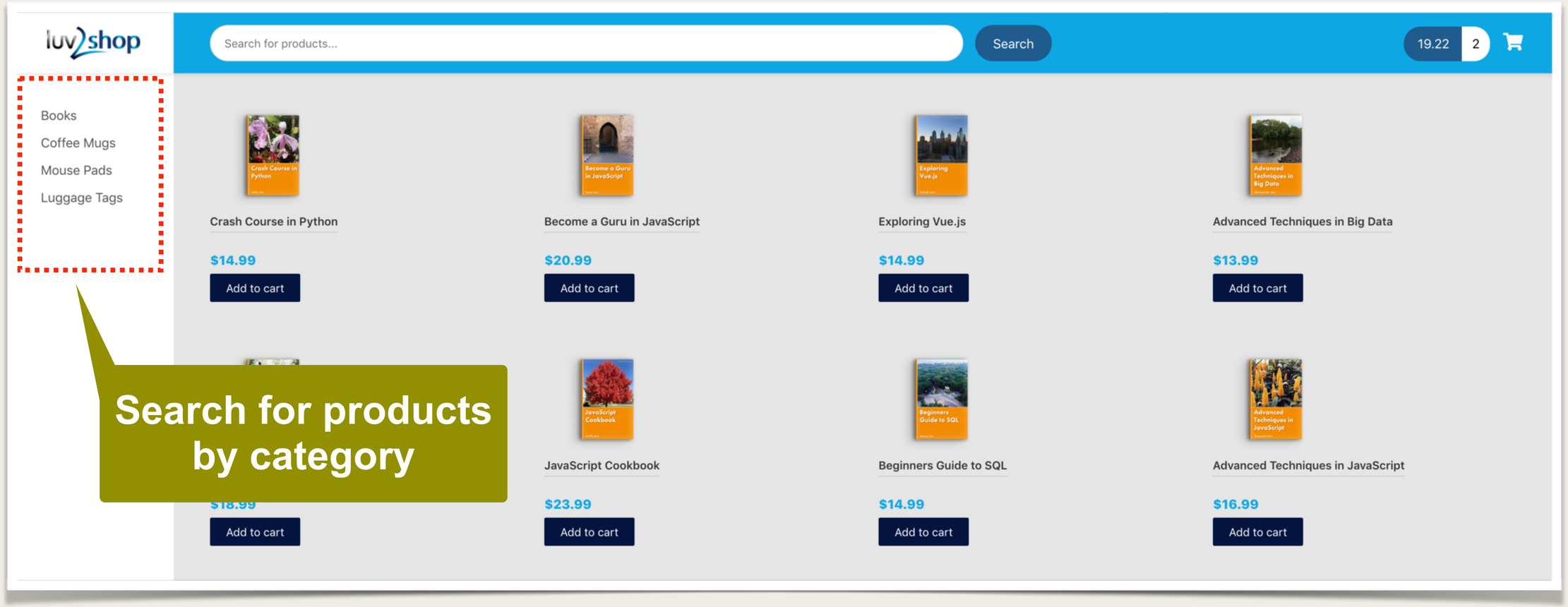
Release 2.0 - Plan

From: The Boss

- Online Shop Template Integration
- Search for products by category
- Search for products by text box
- Master / detail view of products
- Pagination support for products
- Add products to shopping cart (CRUD)
- Shopping cart check out

Current task

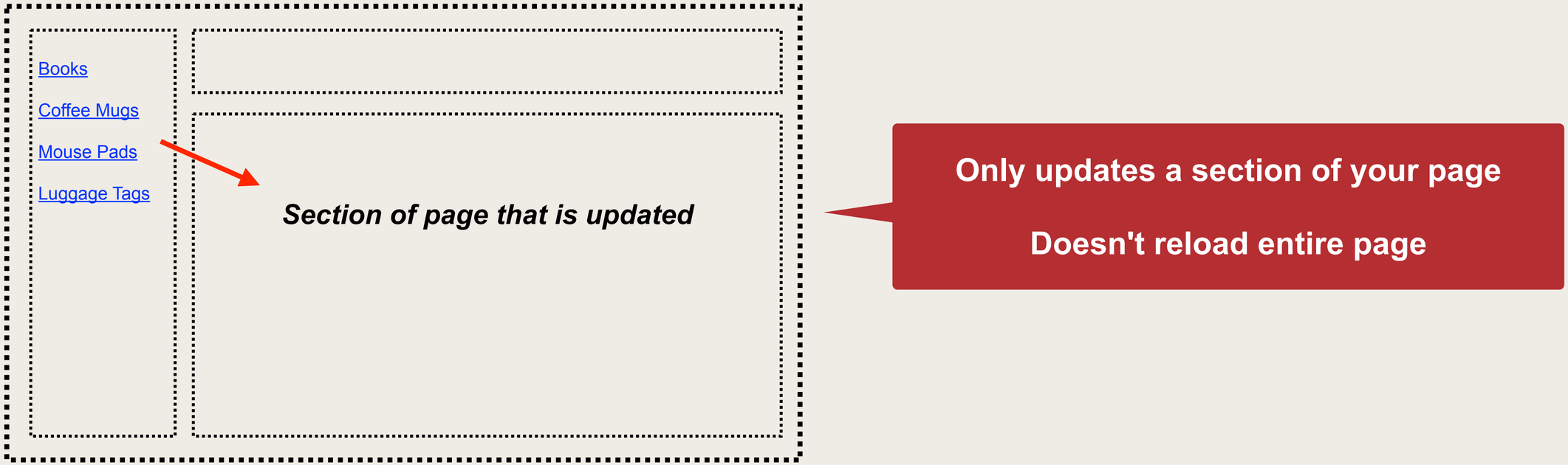
Search for Products by Category



DEMO

Angular Routing

- In Angular, you can add links in your application
- The links will route to other components in your application
- Angular routing will handle updating a view of your application



Key Components

Name	Description
Router	Main routing service. Enables navigation between views based on user actions.
Route	Maps a URL path to a component.
RouterOutlet	Acts as a placeholder. Renders the desired component based on route.
RouterLink	Link to specific routes in your application.
ActivatedRoute	The current active route that loaded the component. Useful for accessing route parameters.
more ….	

We will see code examples in upcoming slides

http://angular.io/guide/router

Development Process

1. Define routes
2. Configure Router based on our routes
3. Define the Router Outlet
4. Set up Router Links to pass category id param
5. Enhance ProductListComponent to read category id param
6. Modify Spring Boot app - REST Repository needs new method
7. Update Angular Service to call new URL on Spring Boot app

Step 1: Define Routes

- A route has a **path** and a reference to a **component**
- When the user selects the link for the route path
  - Angular will create a new instance of component

```
const routes: Routes = [

  {path: 'products', component: ProductListComponent}

];
```

When path matches,
create new instance of component

Path to match

Note: The path has no leading slashes!

Step 1: Define Routes

- Add route to show products for a given category **id**

```
const routes: Routes = [

  {path: 'category/:id', component: ProductListComponent},
  {path: 'products', component: ProductListComponent}

];
```

Category `id` parameter

The component can read this later and show products for this category

Step 1: Define Routes

- Add more routes to handle for other cases ...

```
const routes: Routes = [
  {path: 'category/:id', component: ProductListComponent},
  {path: 'category', component: ProductListComponent},
  {path: 'products', component: ProductListComponent },
  {path: '', redirectTo: '/products', pathMatch: 'full'},
  {path: '**', redirectTo: '/products', pathMatch: 'full'}
];
```

Order of routes is important.
First match wins.
Start from most specific to generic.

This is the generic wildcard. It will match on anything that didn't match above routes.

Step 1: Define Routes

- Can add a custom PageNotFoundComponent ... for 404s

```
const routes: Routes = [
 ...
 {path: '**', component: PageNotFoundComponent}
];
```

Be sure to place this last.

Order of routes is important.
First match wins.
Start from most specific to generic.

Custom component that you create

Can give any name and
provide your own custom view

https://angular.io/guide/router#define-a-wildcard-route

Step 2: Configure Router based on our routes

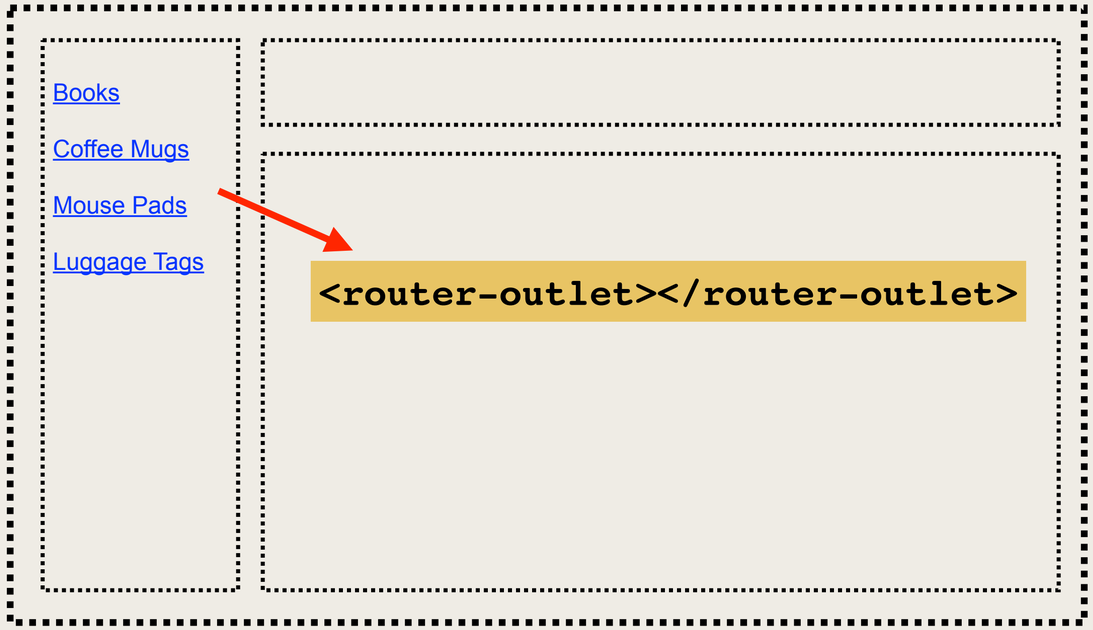
- Configure the router in the application module

File: app.module.ts

```
const routes: Routes = [ ... ];


@NgModule({
  declarations: [
    AppComponent,
    ProductListComponent
  ],
  imports: [
    RouterModule.forRoot(routes),
    BrowserModule,
    HttpClientModule
  ],
  providers: [ProductService],
  bootstrap: [AppComponent]
})
export class AppModule { }
```

Step 3: Define the Router Outlet

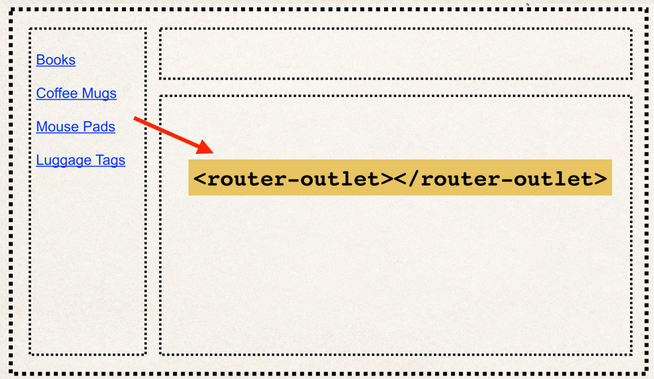
- **Router Outlet** acts as a placeholder.
- Renders the desired component based on route.



Only updates a section of your page

Doesn't reload entire page

Step 3: Define the Router Outlet



- Update app.component.html to use Router Outlet

File: app.component.html

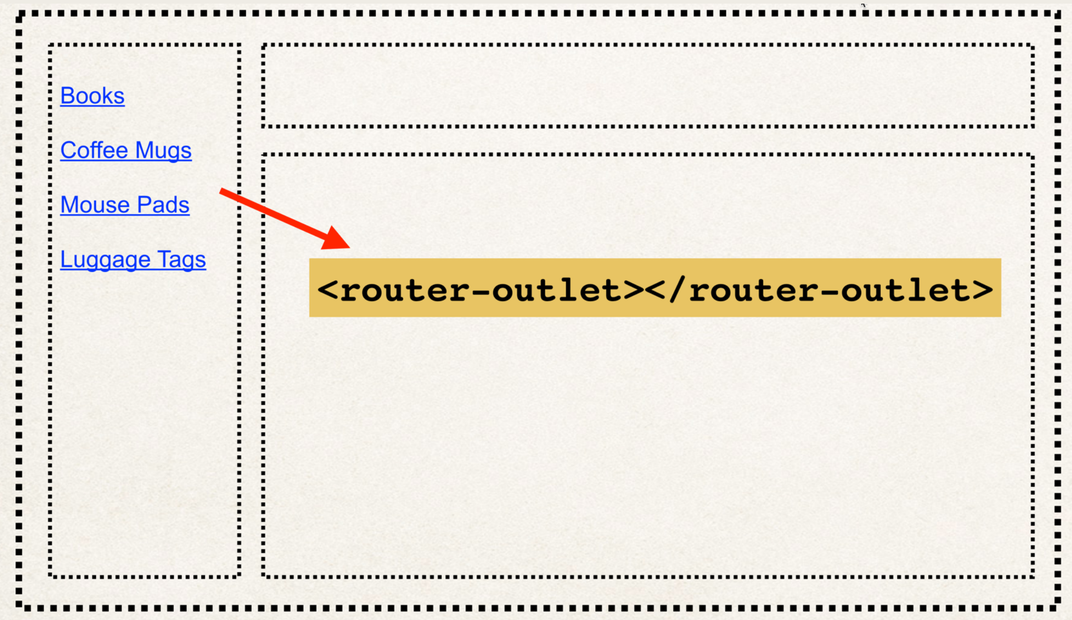
```
...

    <!-- MAIN CONTENT -->
    <router-outlet></router-outlet>

    <!-- <app-product-list></app-product-list> -->

...
```

Based on Router configuration
Display appropriate component here

Comment out or delete the reference to
<app-product-list></app-product-list>

Step 4: Set up Router Links to pass category id param

- In our HTML page, set up links to our route
- Pass category id as a parameter



Step 4: Set up Router Links to pass category id param

- In our HTML page, set up links to our route
- Pass category id as a parameter

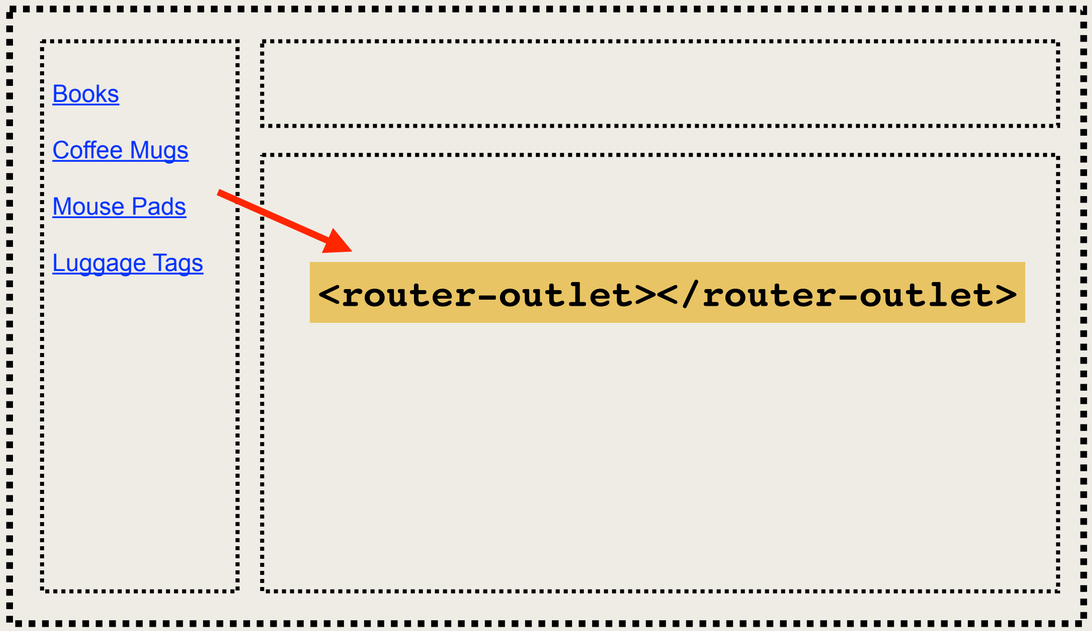
Based on Routes configuration, use ProductListComponent

File: app.component.html

```
...
   <!-- MENU SIDEBAR -->
   <li>
      <a routerLink="/category/1" routerLinkActive="active-link">Books</a>
   </li>

   <li>
      <a routerLink="/category/2" routerLinkActive="active-link">Coffee Mugs</a>
   </li>
...
```

Recall: Router Outlet

- When user clicks the link
  - ProductListComponent will appear in the location of `router-outlet`



Step 5: Enhance ProductListComponent to read category id param

- Need to read the category id parameter

```
currentCategoryId: number;
...
...
...
this.currentCategoryId = +this.route.snapshot.paramMap.get('id');
```

State of route at this given moment in time

Read the `id` parameter

Use the activated route

Map of all the route parameters

Step 6: Modify Spring Boot app - REST Repository needs new method

- Currently, the Spring Boot app, returns products regardless of category
- Need to modify to only return products for a given category id

Step 6: Modify Spring Boot app - REST Repository needs new method

- Spring Data REST and Spring Data JPA supports "query methods"
- Spring will construct a query based on method naming conventions
- Methods starting with: `findBy`, `readBy`, `queryBy`, etc ...

Magic!

File: ProductRepository.java

```
public interface ProductRepository extends JpaRepository<Product, Long> {

    Page<Product> findByCategoryId(@RequestParam("id") Long id, Pageable pageable);

}
```

A query method

Step 6: Modify Spring Boot app - REST Repository needs new method

File: ProductRepository.java

```
public interface ProductRepository extends JpaRepository<Product, Long> {

    Page<Product> findByCategoryId(@RequestParam("id") Long id, Pageable pageable);

}
```

A query method

Match by category id

Use this parameter value

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

```
SELECT * FROM product where category_id=?;
```

Magic!

More on Query Methods

- You can provide your own custom query using `@Query` annotation
- Support is available for conditionals: and, or, like, sort etc
- For details on this topic
  - See the Query Method section in the **Spring Data Reference Manual**

www.luv2code.com/spring-data-query-methods

Step 6: Modify Spring Boot app - REST Repository needs new method

- `Page` and `Pageable` provides support for pagination

File: ProductRepository.java

```
public interface ProductRepository extends JpaRepository<Product, Long> {

    Page<Product> findByCategoryId(@RequestParam("id") Long id, Pageable pageable);

}
```

`Page` is a sublist of a list of objects

Has information such as totalElements, totalPages, curentPosition etc

`Pageable` represents pagination information

Has information such as pageNumber, pageSize, previous, next etc

Step 6: Modify Spring Boot app - REST Repository needs new method

- Spring Data REST automatically expose endpoints for query methods
- Exposes endpoint: `/search/<<queryMethodName>>`

File: ProductRepository.java

```
public interface ProductRepository extends JpaRepository<Product, Long> {

    Page<Product> findByCategoryId(@RequestParam("id") Long id, Pageable pageable);

}
```

http://localhost:8080/api/products/search/findByCategoryId

Step 6: Modify Spring Boot app - REST Repository needs new method

File: ProductRepository.java

```
public interface ProductRepository extends JpaRepository<Product, Long> {

    Page<Product> findByCategoryId(@RequestParam("id") Long id, Pageable pageable);

}
```

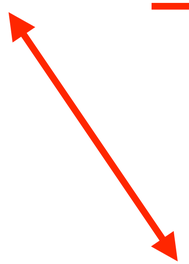
http://localhost:8080/api/products/search/findByCategoryId?id=1

To pass data to REST API

Step 7: Update Angular Service to call new URL on Spring Boot app

File: product.service.ts

```
export class ProductService {

  private baseUrl = 'http://localhost:8080/api/products';

  constructor(private httpClient: HttpClient) { }

  getProductList(theCategoryId: number): Observable<Product[]> {

    const url = `${this.baseUrl}/search/findByCategoryId?id=${theCategoryId}`;

    return this.httpClient.get<GetResponse>(url).pipe(
      map(response => response._embedded.products)
    );
  }
}
```



http://localhost:8080/api/products/search/findByCategoryId?id=1

Development Process

1. Define routes
2. Configure Router based on our routes
3. Define the Router Outlet
4. Set up Router Links to pass category id param
5. Enhance ProductListComponent to read category id param
6. Modify Spring Boot app - REST Repository needs new method
7. Update Angular Service to call new URL on Spring Boot app