Angular training

Introduction to Angular routing

Introduction to Angular Router

Angular Router is a powerful feature in Angular framework that enables navigation between different components in an application. It allows developers to define the navigation paths, handle URL navigation, and manage the application state.
We will see:

- Introduce the Angular router
- Create an Angular application with routing
- Pass parameters to routes
- Enhance navigation with advanced features (guard)

Server Side Rendering Page

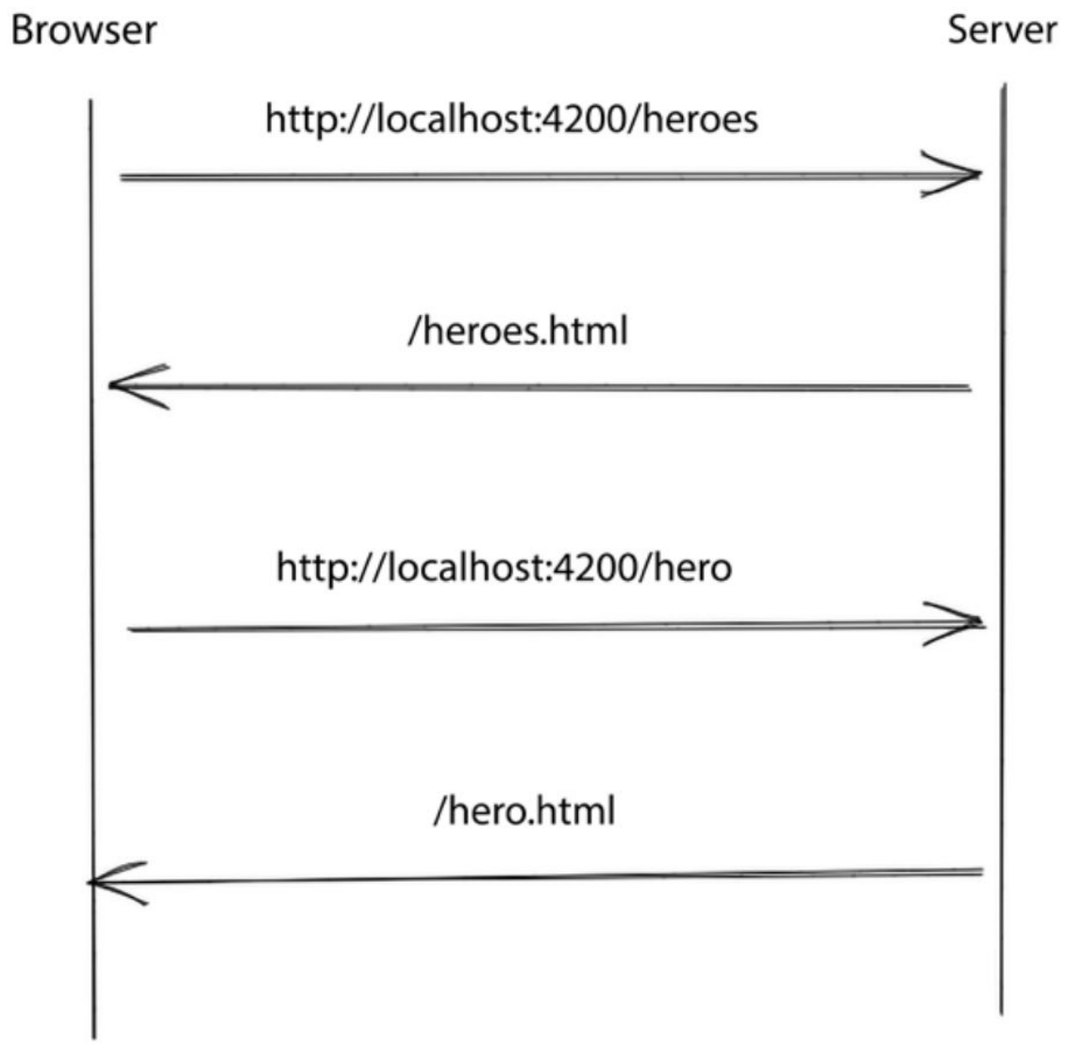
In traditional web applications, when we wanted to change from one view to another, we needed to request a new page from the server.

The browser would create a URL for the view and send it to the server.

The browser would then reload the page as soon as the client received a response.

It was a process that resulted in round trip time delays and a bad user experience for our applications

Server Side Rendering Page



Single Page Application

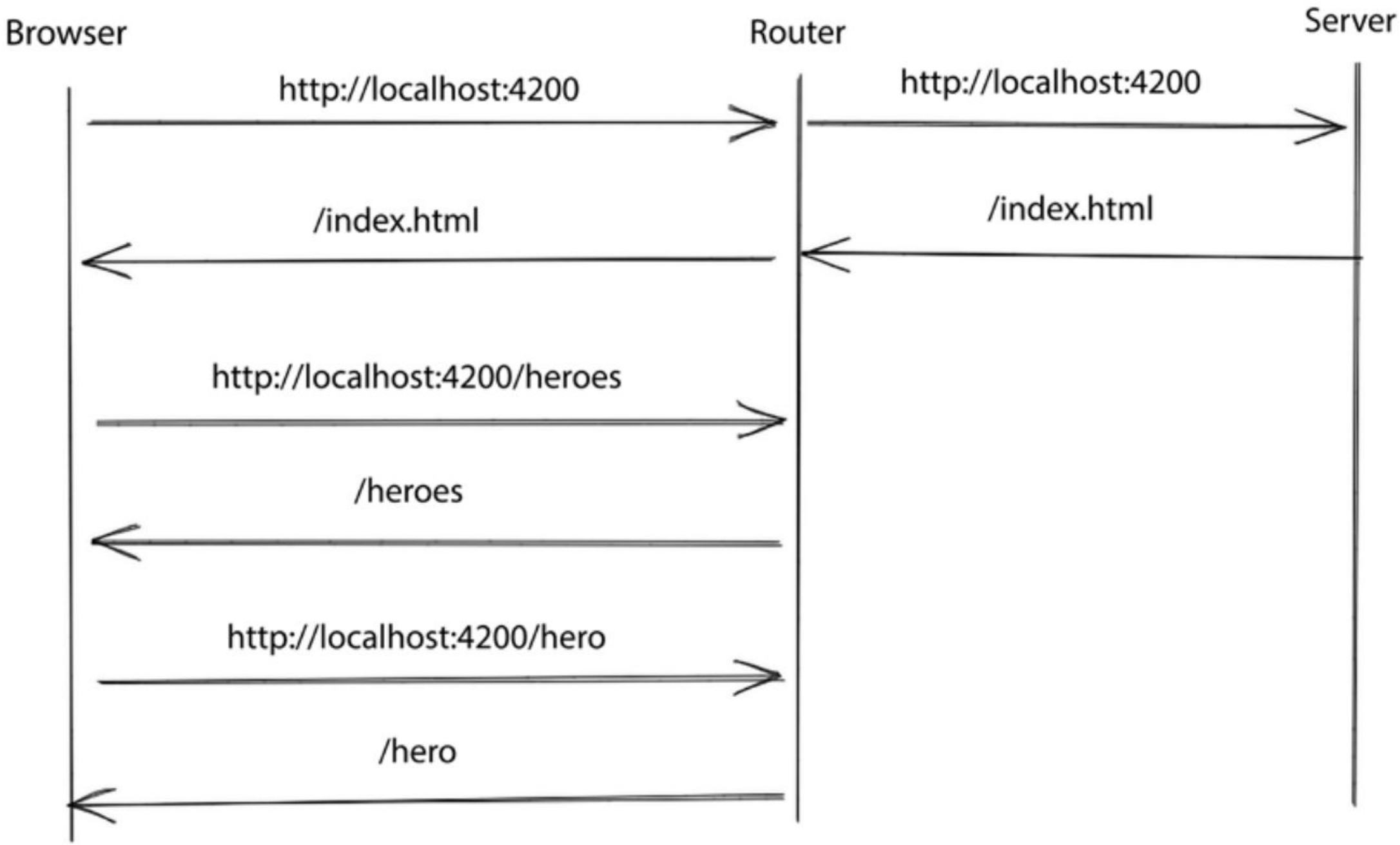
Modern web applications using JavaScript frameworks such as Angular follow a different approach.

They handle changes between views or components on the client side without bothering the server.

They contact the server only once during bootstrapping to get the main HTML file. Any subsequent URL changes are intercepted and handled by the router on the client.

These types of applications are called Single-Page Applications(SPAs) because they do not cause a full reload of a page

Single Page Application



Angular router

The Angular framework provides the @angular/router npm package, which we can use to navigate between different components in an Angular application.

Adding routing in an Angular application involves the following steps:

1. Specify the base path for the Angular application.

2. Use an appropriate Angular module from the @angular/router package.

3. Configure different routes for the Angular application.

4. Decide where to render components upon navigation

Specifying a base path

index.html

```
<!doctype html>
<html lang="en">
<head>
  <meta charset="utf-8">
  <title>HelloWorld</title>
  <base href="/">
  <meta name="viewport" content="width=device-width, initial-scale=1">
  <link rel="icon" type="image/x-icon" href="favicon.ico">
</head>
<body>
  <app-root></app-root>
</body>
</html>
```

Specifying a base path

The href attribute informs the browser about the path it should follow when attempting to load external resources, such as media or CSS files, once it goes deeper into the URL hierarchy.

The Angular CLI automatically adds the tag when creating a new Angular application and sets the href value to the application root, /.

Configuring the router

app.config.ts

```
import { ApplicationConfig } from '@angular/core';
import { provideRouter } from '@angular/router';
import { provideHttpClient } from '@angular/common/http';
import { routes } from './app.routes';

export const appConfig: ApplicationConfig = {
  providers: [provideRouter(routes), provideHttpClient()]
};
```

Configuring the router (2)

The method provideRouter sets up providers necessary to enable `Router` functionality for the application.

It also allows to configure a set of routes as well as extra features that should be enabled.

Configuring the router (3)

The routes variable that we pass in the providerRouter method is a list of Routes objects that specify what routes exist in the application and what components should respond to a specific route. It can look like the following:

```
import { Routes } from '@angular/router';
import { ProductListComponent } from './product-list/product-list.component';
import { PageNotFoundComponent } from './page-not-found/page-not-found.component';

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

Each route definition object contains a `path` property, which is the URL path of the route, and a `component` property that defines which component will be loaded when the application navigates to that path

Configuring the router (4)

Navigation in an Angular application can occur either manually by changing the browser URL or by navigating using in-app links. In the first scenario, the browser will cause the application to reload, while the second will instruct the router to navigate along a route path in the application code.

In our case, when the browser URL contains the /about path, the router creates an instance of *AboutComponent* and displays its template on the page. On the contrary, when the application navigates to / by code, the router follows the same procedure and additionally updates the URL of the browser.

If the user tries to navigate to a URL that does not match any route, Angular activates a custom type of route called a **wildcard**. The wildcard route has a path property with two asterisks and matches any URL. The component property is usually an application-specific *PageNotFoundComponent* or the main component of the app

Configuring the router (5)

app.component.ts

src > app > app.component.html > ...

Go to component

```
1    <router-outlet></router-outlet>
```

Adding new component

We will add several more components to explore more on router:

ng g component cart

ng g component product

ng g component product-list

ng g component page-not-found

Add new path for the new components

app.routes.ts

```
import { Routes } from '@angular/router';
import { ProductListComponent } from './product-list/product-list.comp
import { PageNotFoundComponent } from './page-not-found/page-not-found
import { ProductComponent } from './product/product.component';
import { CartComponent } from './cart/cart.component';
export const routes: Routes = [
    { path: '', component: ProductListComponent },
    { path: 'products', component: ProductComponent },
    { path: 'cart', component: CartComponent },
    { path: '**', component: PageNotFoundComponent }
];
```

routerLink (1)

routerLink is a directives to perform navigation in our application, without refreshing the browser.

We apply the routerLink directive to anchor HTML elements, and we assign the route path in which we want to navigate as a value.

routerLink (2)

app.component.ts

src > app > TS app.component.ts > AppComponent

```
2   import { RouterOutlet } from '@angular/router';
3   import { ProductListComponent } from './product-list/pro
4   import { CommonModule } from '@angular/common';
5   import {RouterModule} from '@angular/router';
6   @Component({
7     selector: 'app-root',
8     standalone: true,
9     imports: [RouterOutlet,ProductListComponent,
10      CommonModule, RouterModule],
11    templateUrl: './app.component.html',
12    styleUrl: './app.component.css'
```

routerLink (2)

app.component.html

```
<div>
    <a routerLink="/products">Products</a>
    <a routerLink="/cart">Cart</a>
</div>
<router-outlet></router-outlet>
```

Handling unknown path

You have seen previously that we set up a wildcard route to display PageNotFoundComponent when our application tries to navigate to a route path that does not exist.

Unknown path in Angular routing is presented as **

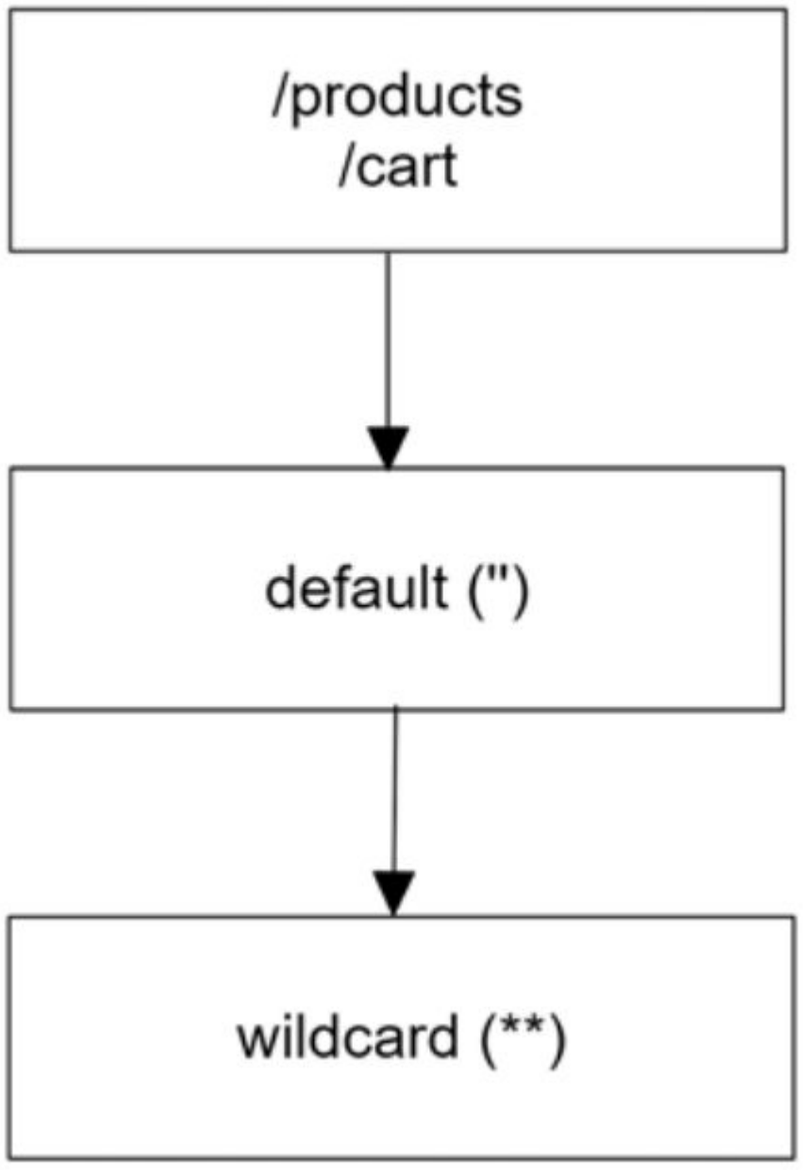
```
import { CartComponent } from './cart/cart.component';
export const routes: Routes = [
    { path: '', component: ProductListComponent },
    { path: 'products', component: ProductComponent },
    { path: 'cart', component: CartComponent },
    { path: '**', component: PageNotFoundComponent }
];
```

Setting default path

We set the path property of a route to an empty string to indicate that the route is the default one for an Angular application. In our case, we want the default route path to display the product list.

```
export const routes: Routes = [
    { path: '', component: ProductListComponent },
    { path: 'products', component: ProductComponent },
    { path: 'cart', component: CartComponent },
    { path: '**', component: PageNotFoundComponent }
];
```

Route path resolution process



Empty route path need to be added after all other routes because, as order of the routes is important.

We want more specific routes before less specific ones.

Navigation programmatically to a route

We can redirect user to a certain page, programmatically from the .ts code, for example to redirect user to order success page upon successful order.

We use router.navigate method from @angular/router library to navigate user to a new page.

Navigation programmatically to a route (2

page-not-found.component.html

src > app > page-not-found > <> page-not-found.compone

Go to component

```
1   <h3>Ooops!</h3>
2   <p>The requested page was not found</p>
3   <button (click)="goHome()">Home</button>
```

Navigation programmatically to a route (3

page-not-found.component.ts

```
import { Component } from '@angular/core';
import { Router } from '@angular/router';

export class PageNotFoundComponent {

  constructor(private router: Router) { }
```

Navigation programmatically to a route (4

page-not-found.component.ts

Decorating router link with styles

The module of the Angular router exports the routerLinkActive directive, which we can use to change the style of a route when it is active.

It accepts a list of class names or a single class that is added when the link is active and removed when it becomes inactive.

We will use routerLinkActive directive so that when we click on a link in our application, its color turns to black to denote that the link is active.

Decorating router link with styles (2)

app.component.html

```
<div>
    <a routerLink="/products" routerLinkActive="active">Products</a>
    <a routerLink="/cart" routerLinkActive="active">Cart</a>
</div>
```

app.component.css

```
.active {
    color: black;
}
```

https://stackoverflow.com/questions/38727582/angular-2-routerlinkactive-always-active-for-base-path

RouterLinkActive with base path

```
<div class="collapse navbar-collapse" id="navbarNavAltMarkup
  <div class="navbar-nav">
    <a class="nav-link active" routerLink="/"
     routerLinkActive="text-primary"
     [routerLinkActiveOptions]="{ exact: true }">Home</a>
    <a class="nav-link" routerLink="/about"
     routerLinkActive="text-primary"
     [routerLinkActiveOptions]="{ exact: true }">About</a>
    <a class="nav-link"
    routerLink="/cart"
    routerLinkActive="text-primary"
    [routerLinkActiveOptions]="{ exact: true }">Cart</a>
  </div>
```

Passing parameter through route

A common scenario in enterprise web applications is to have a list of items and when you click on one of them, the page changes the current view and displays details of the selected item.

The previous approach resembles a master-detail browsing functionality, where each generated URL on the master page contains the identifiers required to load each item on the detail page.

We can represent the previous scenario with two routes navigating to different components. One component is the list of items, and the other is the details of an item. So, we will see how to create and pass dynamic item-specific data from one route to the other.

Passing parameter through route (2)

app.routes.ts

```
import { CartComponent } from './cart/cart.component';
import { ProductDetailComponent } from './product-detail/product-deta
export const routes: Routes = [
    { path: '', component: ProductListComponent },
    { path: 'products/:id', component: ProductDetailComponent },
    { path: 'cart', component: CartComponent },
```

Passing parameter through route (2)

Product-list.component.html

Add routerlink property binding into the a element.

```
<li *ngFor="let product of products | sort" >
  <a [routerLink]="['/products', product.id]">
    {{product.name | uppercase}}</a>
  <p>{{product.price | currency:'USD'}}</p>
/li>
```

Passing parameter through route (3)

product-list.component.ts

Import RouterModule to use routerLink binding

```
import { ProductCreateComponent } from '../product-cr
import { RouterModule } from '@angular/router';
@Component({
  selector: 'app-product-list',
  standalone: true,
  imports: [CommonModule,RouterModule, ProductDetailC
  templateUrl: './product-list.component.html',
```

Passing parameter through route (4)

product-detail.component.ts

Import and inject ActivatedRoute (to retrieve parameter from url)

```
import { ProductsService } from '../products.service';
import { ActivatedRoute } from '@angular/router';
@Component({
```

```
constructor(private productService:ProductsService,
  private route: ActivatedRoute
  ) {

}
```

Passing parameter through route (5)

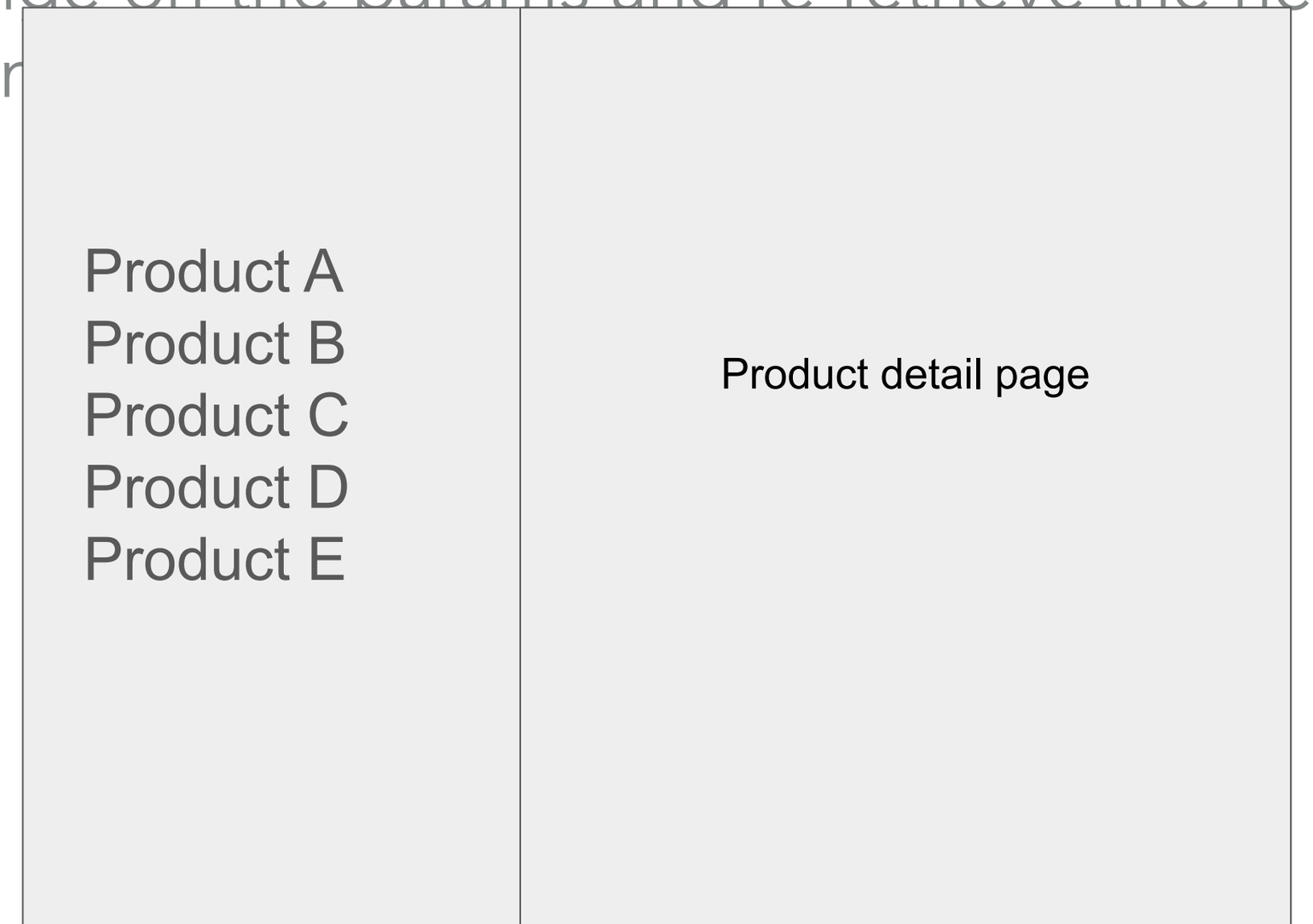
product-detail.component.ts

Retrieve the parameters on onInit and call the API with parameters

```
ngOnInit(): void {
 this.product$ = this.route.paramMap.pipe(
 switchMap(params => {
   return this.productService.getProduct(Number(params.
     get('id'))));
      })
      );
 }
```

Master Detail page

In this scenario, the detail page is not entirely reloaded when user select different product. It is advisable for you to listen to the change on the params and re-retrieve the new product based on



Product A
Product B
Product C

Product Detail

Retrieving parameter snapshot

Another way of retrieving parameter is using parameter snapshot:

```
const id = this.route.snapshot.params['id'];

this.product$ = this.productService.getProduct(id);
```

This approach is more suitable in the case of the website as the router is normally constant.

Retrieving data from query parameter

Query parameters are considered optional because they aim to provide optional services such as sorting or filtering data. Some examples are as follows:

- /products?sortOrder=asc: Sorts a list of products in ascending order
- /products?page=3&pageSize=10: Splits a list of products into pages of 10 records and gets the third page

Query parameters are recognized in a route by the ? character. We can combine multiple query parameters by chaining them with an ampersand (&) character

The ActivatedRoute service contains a queryParamMap observable that we can subscribe to in order to get query parameter values. It returns a ParamMap object, similar to the paramMap observable, which we can query to get parameter values.

Retrieving data from query parameter (2)

```
ngOnInit(): void {

this.route.queryParamMap.subscribe(params => {

console.log(params.get('sortOrder'));

});

}
```

Controlling route access with guard

When we want to prevent unauthorized access to a particular route, we use a specific Angular concept called a **guard**. An Angular guard can be of the following types:

- **canActivate**: Controls whether a route can be activated.
- **canActivateChild**: Controls access to child routes of a route.
- **canDeactivate**: Controls whether a route can be deactivated. Deactivation happens when we navigate away from a route.
- **canLoad**: Controls access to a route that loads a lazy-loaded module.
- **canMatch**: Controls access to the same route path based on application conditions.

https://angular.io/api/router/CanActivateFn

Controlling route access with guard (2)

Create a guard (CanActivate) using the command:

```
ng g guard auth
(press space and enter on canActivate)
```

```
import { CanActivateFn } from '@angular/router';

export const authGuardGuard: CanActivateFn = (route, state) => {
  return true;
};
```

Controlling route access with guard (3)

Use inject method to inject the service in this function.

Verify if user Logged In and redirect to home if not.

```ts
import { CanActivateFn, Router } from '@angular/router';
import { inject } from '@angular/core';
import { AuthService } from './services/auth.service';

export const authGuardGuard: CanActivateFn = (route, state) => {
const authService = inject(AuthService);
 const router = inject(Router);
 if (authService.isLoggedIn) { return true; }
 return router.parseUrl('/');
};
```

product$ = this.produc

Controlling route access with guard (4)

In app.routes.ts, Activate the guard on one of the route, eg: cart

Deactivate guard

We will explore how a guard can be used to control if a route can be deactivated is a function of the CanDeactivateFn type.

https://angular.io/api/router/CanDeactivateFn

Deactivate guard (2)

Create a new Deactivate guard

ng g guard checkout

Deactivate guard (3)

In the checkout.guard.ts , modify as follows

```
import { CanDeactivateFn } from '@angular/router';
import { CartComponent } from './cart/cart.component';

export const checkoutGuard: CanDeactivateFn<CartComponent> = () => {
const confirmation = confirm('You have pending items in your cart. '
+ Do you want to continue?');
 return confirmation;
};
```

Deactivate guard (4)

Add the canDeactivate property in app.routes.ts

```
  { path: 'cart', component: CartComponent, canActivate: [authGuardGuard],
canDeactivate:[checkoutGuard] },
```