**Full Stack Application : A Real Time Webshop App**

Back-end: Spring-boot(REST API)

Front-end: Angular

Database with full CRUD support

A webshop funkciói:

* Termékek listázása
* Termékek szűrése kategória szerint
* Keresés a termékek között
* Termékek további részleteinek megjelenítése
* Oldalak,lapozás, max méret választása
* Kosár,amelyben gyűjteni lehet a termékeket(darabszám és ár megjelenítésével)
* Checkout(vásárló adatainak bekérése)

Megvalósításhoz telepíteni kell:

* Java Development Kit (JDK)
* Java IDE (jelen esetben IntelliJ)
* Maven
* MySQL Database és MySQL Workbench
* Visual Studio Code

Fontosabb felhasznált technológiák(később kifejtem a konkrétan a szerpüket a projekt során)

REST(**Representational State Transfer**): Kliens- szerver architektúrára épül,ahol a kliensek kéréseket indítanak a szerverek felé; a szerverek kéréseket dolgoznak fel és a megfelelő választ küldik vissza.

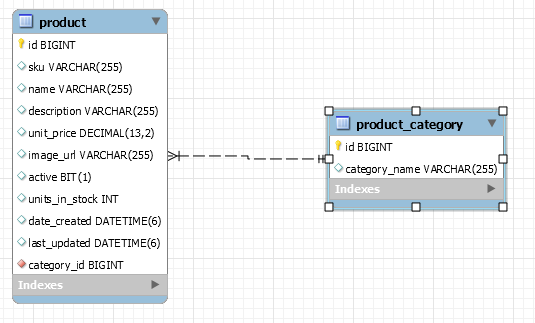
Spring: A **Spring** egy nyílt forráskódú, inversion of controlt (Java objektumok életciklusának kezelése) megvalósító Java alkalmazás keretrendszer.

**Adatbázis séma**: 2 sql scriptet csatoltam

Az első a 01-create-user, amely létrehoz számunkra egy felhasználót(ecommerceapp).

Második 02-create-products, amely létrehozza a szükséges adatbázist full-stack-ecommerce néven. Az adatbázis 2 táblából áll, Product, amely a termékek tulajdonságait tartalmazza és ProductCategory,amely kategóriákba sorolja a termékeket, hogy később szűrni tudjuk őket.

Fontosabb oszlopok: name, description,unit\_price, image\_url(a termékekhez kapcsolódó képekhez)



**Back-end:**

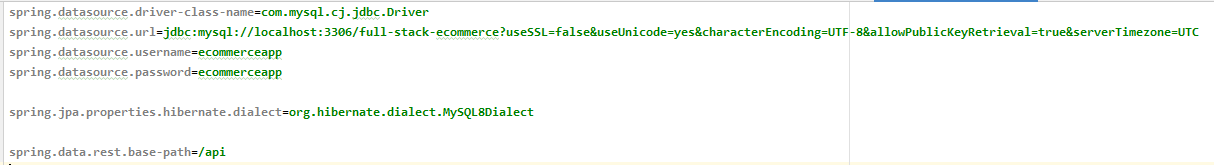
Spring-Boot projekt gyors és egyszerű létrehozásához: **https://start.spring.io/**

Ezen az oldalon megadhatjuk a szükséges dependenciákat,amelyet egyből belerak nekünk a legenerált pom.xml-be, ami által a Maven leszedi nekünk őket.

**Spring-Boot dependencies:**

* spring-boot-starter-data-jpa
* spring-boot-starter-data-rest
* mysql-connector-java
* lombok

Maven projekt létrehozása után, az application.properties fileban megadhatjuk az adatbázis eléréséhez szükséges adatokat. Továbba beállítjuk,hogy a URL-ben minden requestet /api -al kelljen kezdeni.



**Entitások létrehozása:**

Mivel két táblánk van az adatbázisban, ezért ezeket reprezentálnunk kell springen belül osztályonként. Ezeket az entitásoknak olyan változókat kell létrehoznunk, amelyek megegyeznek az adatbázisban deklarált struktúrával és ennek megfelelően fel kell annotálnunk öket. Fontos hogy annotálás során pontosan az adatbázisban lévő neveket kell megadnunk.

ORM(Object-Relational Mapping) használata, amely egy programozási technika adatok konvertálására nem kompatibilis típusos rendszerek és objektumorientált programozási nyelvek között. Úgynevezett „virtuális objektum-adatbázist” hoz létre, amelyet az adott programozási nyelvben használni tudunk.

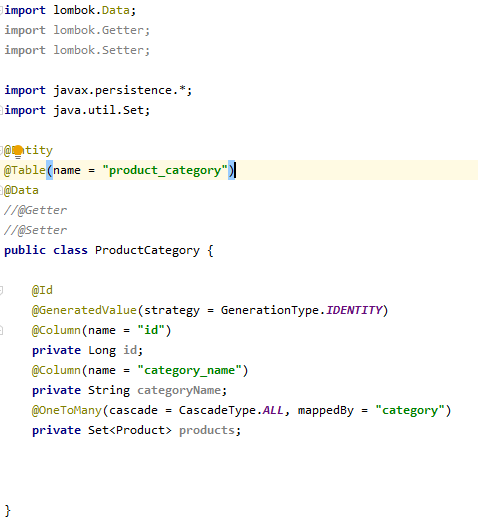
A lombok(@Data) biztosítja, hogy minden propertihez generál nekünk egy setter/gettert.

**Product**



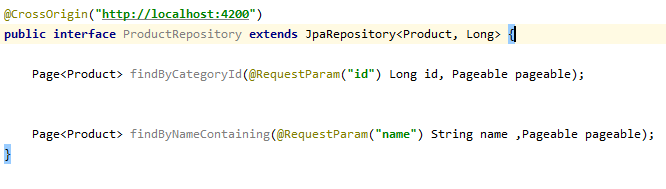
Mivel termékből több van ézert itt @ManyToOne annotációt kell alkalmazni, míg a ProductCategory oldalon @OneToMany-t.

**ProductCategory**



**JpaRepository:**

Miután létrehoztunk egy Interfacet, kitudjuk terjeszteni(extend) JpaRepository-al amelynek két adatot adunk át. <Entity,PrimaryKey>, az entity jelen esetben a Product lesz, a primary key típus pedig Long.



Ehhez hasonlóan létrehozzuk a ProductCategoryRepositoryt is.

**@CrossOrigin** annotáció szükséges,mivel az Angular projektet a 4200 porton fogjuk futtatni, mivel a böngészők nem engedik hogy egy srcipt olyan API-t hívjon meg, aminek nem egyezik meg vele a forrása (scheme/protocol,hostname,port number). Ezért 4200 ra kell állítanunk a portot,hogy az megegyezzen a később használt eCommerce appunkéval.

**REST API:**

A Spring Data REST megkeresi,az imént létrehozott JpaRespositorykat és az ott megadott entitás alapján generál nekünk „végpontokat”. A Product entitás például úgy tudjuk ezáltal elérni,hogy az első betűt kicsire cseréljük és többetszámba tesszük: /products.

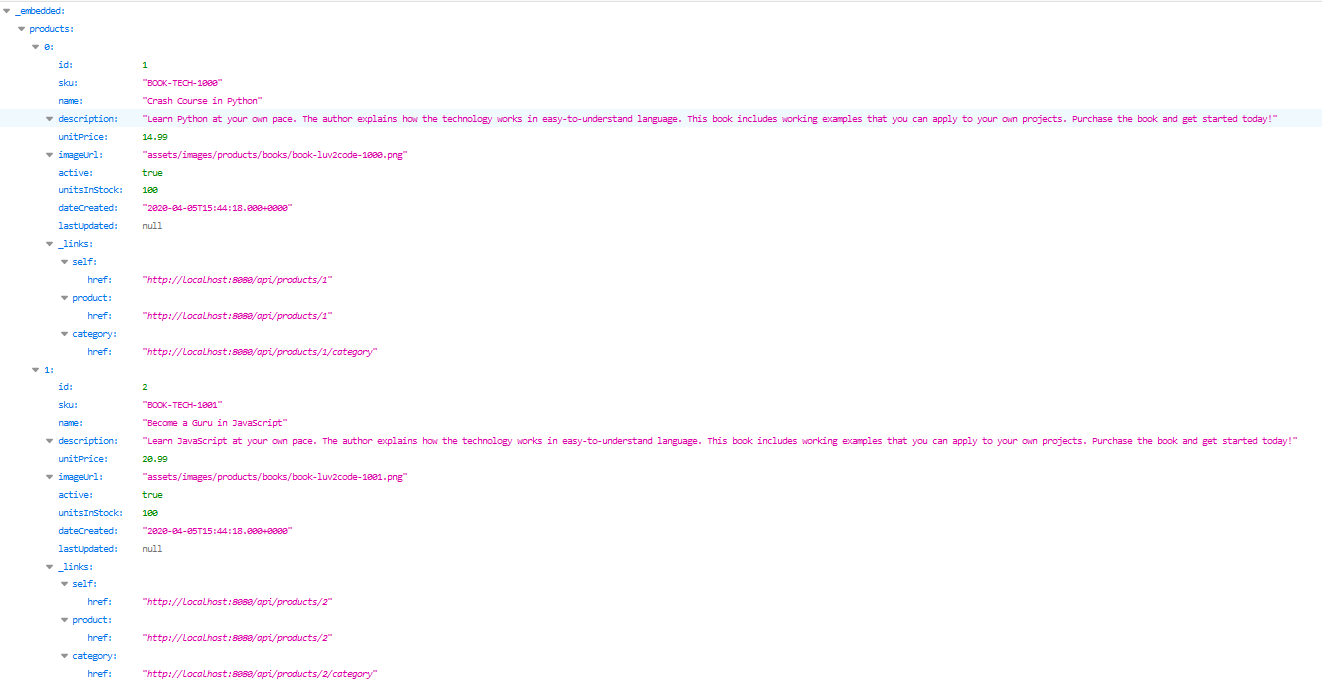
A Spring Data REST ezáltál a végpont által biztosít nekünk teljes CRUD(Create,Read,Update,Delete) funkcionalitást. Mindez azon múlik, hogy milyen HTTP metódust alkalmazunk.

|  |  |  |
| --- | --- | --- |
| HTTP Method |  | CRUD Action |
| POST | /products | Create new product |
| GET | /products | Read a list of products |
| GET | /products{id} | Read a single product |
| PUT | /products{id} | Update an existing product |
| DELETE | /products{id} | Delete an existing product |

Viszont a mi esetünkben csak a Read funkciót fogjuk használni. Ezt a RestDataConfig-ban konfiguráltam.

Az adatokat JSON formában adja vissza nekünk.

http://localhost:8080/api/products



**Front-end**

**Angular**: Egy keretrendszer modern single-page kliens applikációk létrehozásához, HTML-t és TypeScriptet használva.

Angular CLI: Egy parancssorban futtatható interface az Angular számára.

Telepítéshez: **npm install -g @angular/cli**

Új projektet hozunk létre az **ng new angular-ecommerce parancsal**.

Az Angular keretrendszer komponenseket használ az elkülönítés és egyszerűbb átláthatóság érdekében. Ezeknek az egyszerű létrehozása az **ng generate component components/product-list** parancsal történik.

Itt is létre kell hoznunk egy TypeScript osztályt az adatbázisból számazó táblák használatára(Product).

Amit az **ng generate class common/product** parancsal valósítunk meg.

Szervíz: Szükségünk van egy szintén TypeScriptban megírt Service osztályra(**ng generate service services/product**),ami a kliens oldalon fut,amelyen keresztül meghívjuk a Back-enden létrehozott REST API-t. Ezt egy általunk hozzásadott module segítségével éri el, **HttpClientModule**, amelyet az app.module-ban kell hozzáadnunk.

Mivel ezt a servicet más osztályokban is meg fogjuk hívni, lehetővé kell tennünk azt elérést. Ezt alul a providersben adhatjuk meg.



Az Angular támogatja a dependency-injectiont, ezáltal a Service osztályunkban megadhatjuk, hogy bármelyik másik osztály elérhesse a Servicet.

HttpClient használata:

constructor(private httpClient: HttpClient)

A konstruktorban injektáljuk,ezáltal hivatkozni tudunk rá.

Változóban deklaráljuk a REST APInk URL elérését.

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

  getProductList():Observable<Product[]>{

    return  this.httpClient.get<GetResponseProducts>(this. baseurl).pipe(

      map(response => response.\_embedded.products)

    );

  }

interface GetResponseProducts{

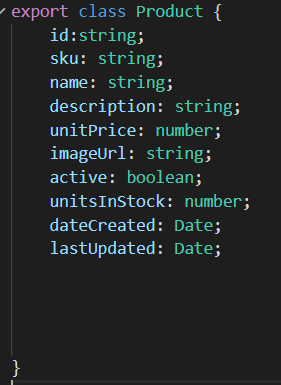
  \_embedded:{

    products:Product[];

  }

A getProductList visszaad nekünk egy Observable(ezekre a többi osztály fel tud iratkozni, és azáltal kezelni tudja az átadott adatokat) Product[] arrayt, a GetResponseProducts interface által a válaszként érkezett adatot belerakja az általunk deklarált arraybe. A válaszként érkező adat a baseURLben megfogalmazott elérési helyről fog származni.

**Product:**



Hasonlóan a back-end oldali Product osztályhoz, front-end vonalon is létre kell hoznunk az adatbátzisban szereplő product táblát megvalósító osztályt. Ezáltal a REST API által kapott adatokat átadhatjuk ezen osztály változóinak.Ugyanígy kell eljárnunk a ProductCategory táblával.

export class ProductCategory {

    id:number;

    categoryName:string;

}

**ProductListComponent:**

A command lineban **ng generate component components/product-list** parancsal létrehozott komponens, amely kezelni fogja a szervíz által átadott adatokat.

@Component({

  selector: 'app-product-list',

  templateUrl: './product-list-grid.component.html',

  styleUrls: ['./product-list.component.css']

})

export class ProductListComponent implements OnInit {

  products:Product[]=[];

constructor(private productService: ProductServiceService) { }

  ngOnInit(): void {

    this.listProducts();

  }

listProducts (){

this.productService.getProductList().subscribe(

data=> {

this.products=data;

}

);

  }

A selector arra szolgál,hogy az ott megadott névvel tudunk később hivatkozni erre a komponensre,koknretan az ehhez a komponenshez tartozó html kódra, amit egy „fő” minden más komponens osszefoglalő html kódban tudunk meghívni. (Erről később részeletesebben).

Mivel kezelnünk kell a Productok adatait és jelenleg nem csak egyről beszélünk, létre kell hoznunk egy Productokat tartalmaző arrayt(products).

A konstruktorban injektáljuk a szervíz osztályt, amit a listProductsban használunk fel. A szervízünkre hivatkozva meghívhatjuk az ott létrehozott getProductList()-et, aminek a visszatérési értékére az Observable segitségével „feliratkozhatunk”(subscribe). Ezutan meg kell adnunk, hogy a visszatérő adatot(data) az általunk korábban létrehozott arraybe tegye. Ezután a html kódban kezelni tudjuk a termékeket és hivatkozni tudunk minden egyes termék tulajdonságára.

A listProducts()-t az ngOnInit()-ben hívjuk meg, ami hasonlóan egy @PostConstuct-hoz, példányosítás után hívodik meg.

**Index.html:**



Egy Angular sok-sok komponensből tevődik össze, de ezeknek a megjelenítése itt öszpontosul. A korábban említett selectorok(<app-root></app-root>) segítségevel az ott megfogalmazott html kódot fogja ennek a helyén megjeleníteni. Ezáltal a kód logikailag elkülöníthető és átlátható lesz.

**app.component.html:**

Az előbb említett <app-root></app-root> selector ezt fogja meghívni. Az összes többi komponens itt lesz meghívva.

**CSS:**

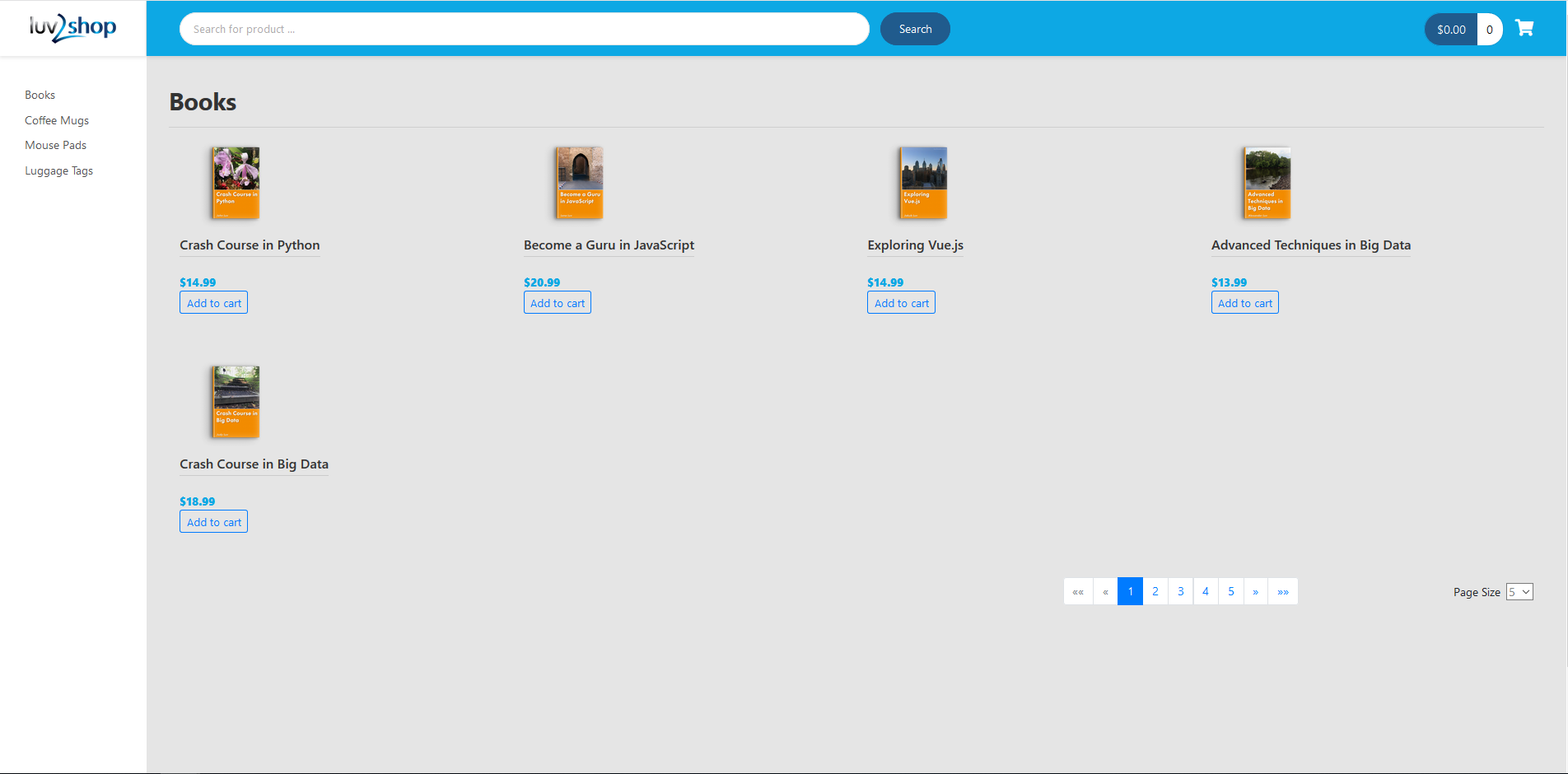
BootStrap hozzáadásához, egyszerűen az index.html-be kell bemásolni a linket.

<link rel="stylesheet" href="https://stackpath.bootstrapcdn.com/bootstrap/4.5.0/css/bootstrap.min.css" integrity="sha384-9aIt2nRpC12Uk9gS9baDl411NQApFmC26EwAOH8WgZl5MYYxFfc+NcPb1dKGj7Sk" crossorigin="anonymous">

**Szerver indídásához** ismét a parancssort kell használnunk, **ng serve –open** parancsal. Ami az Angular által biztosított 4200 prton fog futni.

**Fontos**, hogy a Spring Boot API is fusson a háttérben.

Tervezett végső kínézet:



**Megjelenítés**

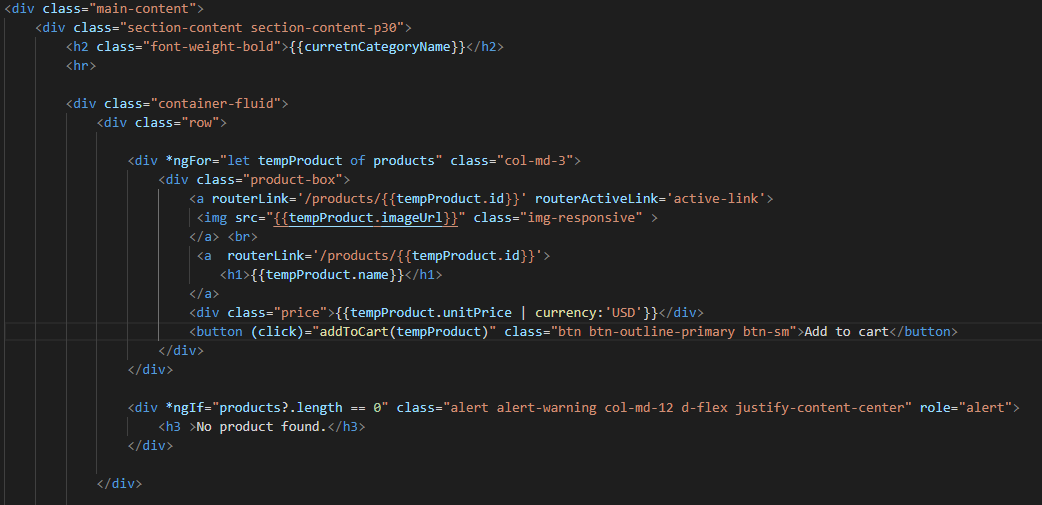
Először megjelenítjük a termékek képeit, nevit és árát.



A product-list.component.html, fileban létrehozzuk ezeknek a megjelenítését. Egyenlőre csak megjelenítésre figyeljünk, a képen látható router és egyéb funkciókat később részletezem.

Mivel mindegyik terméket sorban megszeretnénk jeleníteni, használhatunk irtációt az **\*ngFor** segítségével. Aminek átadhatjuk az általunk a Product-list.component.ts -ben létrehozott **products** arrayt. Ebből mindig kiveszi a soron következőt és átadja a **tempProductnak**. Ez a változó rendelkezik az adott termék adataival így hivatkozhatunk rá.

Ezen úton a htmlen belül is tudunk hivatkozni egy adott termék adataira. {{tempProduct.imageUrl}}



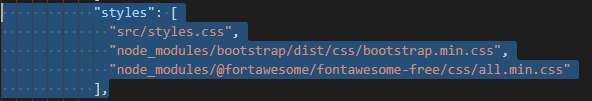
Erre a részre a .ts ben megfogalmazott selectorral tudunk hivatkozni. A képek megjelenítéséhez a képeket az assests mappában kell eltárolni, mivel az adatbázisban az imageUrl oszlopban ezekre az elérési utakra hivatkozunk.

CSS Bootstrap hozzáadása npm haszánálatával. Parancssor: **npm install bootstrap**.

Ikonok hozzáadásához: **npm install @fortawesome/fontawesome-free**

Ezeket a node\_modules-ben találjuk meg. A package.json fileban találjuk meg a telepített verziókat.

A fő konfigurációs fileban (angular.json) is megkell adnunk ezekre a hivatkozásokat, hogy globálisan elérhetőek legyenek.



**Template**

A megjelenítés váza az app.html-ben van megfogalmazva.



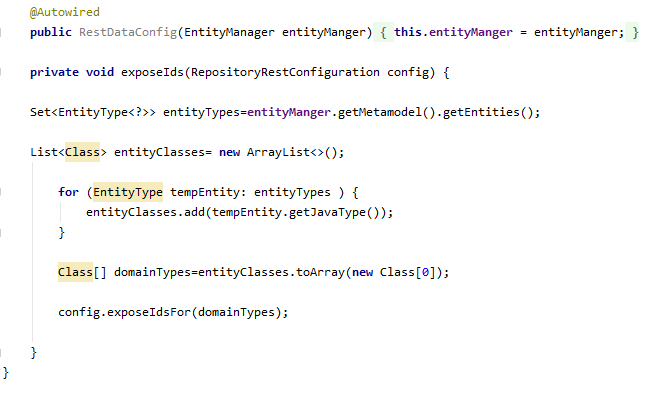
Ezáltal elkülöníthetjük az oldalsó menüsort,ahol a termékeket kategorai szerint tudjuk majd szűrni.

Felül a headert, ahol a keresést és a kosár éléréset fogjuk létrehozni. A középen megjelenő termékek kerülnek a MAIN CONTENTBE. Továbbá alul a footerben linkeket rakunk általános információkért.

**Szűrés kategória szerint:**

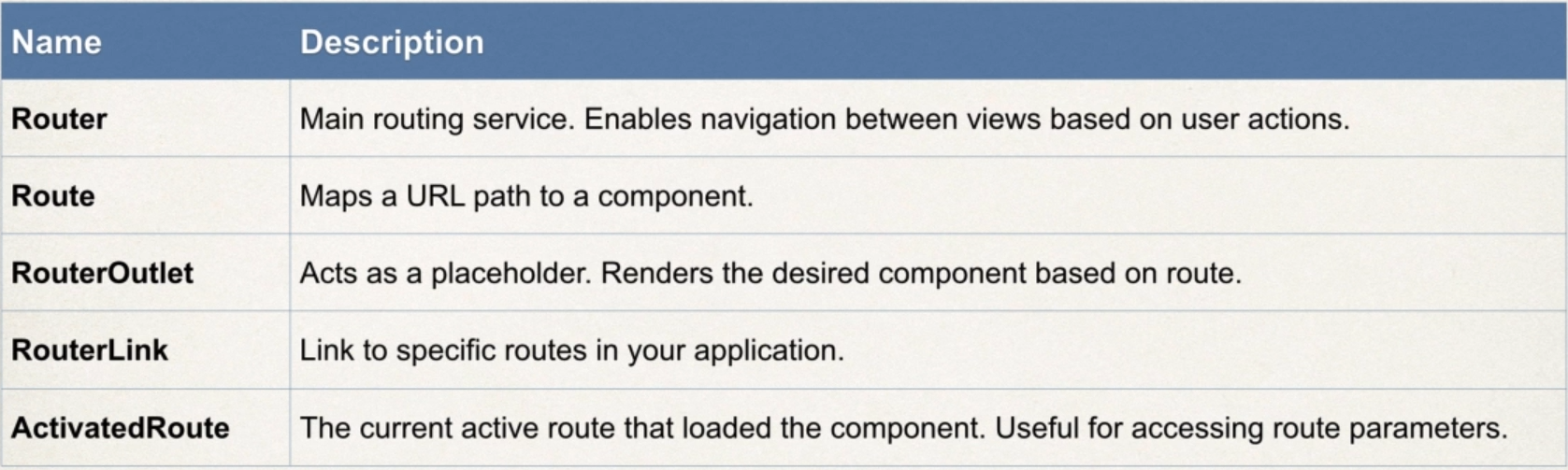
Lényege, hogy az adatbázisban meghatározott category\_id alapján szűrje le a termékeket.

Ahhoz,hogy a REST APIn keresztül megkapjuk a felhasználandó idkat is, módosítanunk kell a back-end vonali konfiguráción.

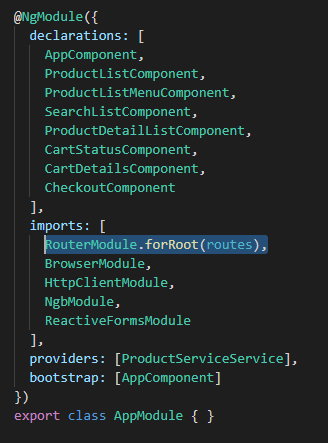


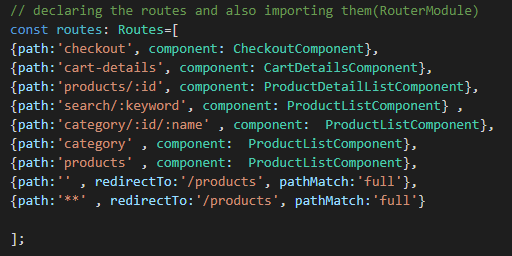
**Routing:** Angularban Routokat tudunk létrehozni, a routoehoz tartozó linkeket különböző komponensekhez tudjuk hozzá kötni. Így a linkeket használva mindig csak egy adott komponenst töltünk ismét be és nem kell az egész oldalt. Amikor a felhasználó kiválaszt egy linket, az Angular létre hoz egy új példányt a linkhez tartozó komponensből és betölti.

Főbb elemek:



Az app.moduler.ts-ben tudjuk ezeket megadni és importálni:





Az alsó két esetben defaultot állítunk be, hogy hova írányítson vissza bennünket. Csak ebben az esetben kell „/” jelet használni a link megadásánál. Fontos a sorrend, mivel fentről lefelé nézi őket, így ha megegyezőket talál,akkor a feljebb lévő fog betöltődni. A betöltés helye azon múlik, hogy hol helyezzük el a **<router-oulet></router-outlet>-**t. Jelen projektben az app.html-ben. Ezáltal mindig csak ezt a részt fogja újratölteni.

Ahogy a pathok között is látható, category/:id/:name el fogjuk elkérni az adott kategóriába tartozó termékeket. Ehhez a back-end vonalon létre kell hoznunk egy függvényt a ProductRepositoryban, amely ehhez igazodik.

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

A Page és Pageable a oldalak és lapozásnál fog segítséget nyújtani,amit később részletezek.

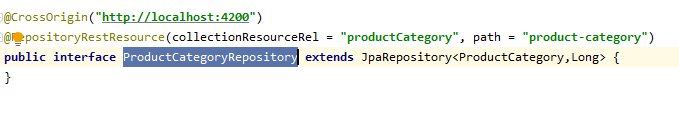
Ez SQL re fordítva annyit jelent hogy: SELECT \* FROM product where category\_id=?;

Mivel ez egy lekérdező metódus, a Spring Data REST generál nekünk végpontot hozzá a /search/metódusnév hozzáadásával és az adott id a végére.

Ezáltal a vegpont [**http://localhost:8080/api/products/search/findByCategoryId?id=1**](http://localhost:8080/api/products/search/findByCategoryId?id=1)

Szükséges oldalt létrehoznunk a menüt ahol a különböző kategoriákat soroljuk fel, és mindegyikhez hozzárendelünk egy routot az adott termék kategóriával. Majd a pathból kiszedett id alapján meghívjuk a az előbb deklarált finByCategoryId függvényünket és egy olyan terméklistát kapunk vissza a REST API-tól ahol csak az adott category\_ID-val rendelkező termékek vannak. Majd ezeket a szokásos módon megjelenítjük.

Ahhoz hogy kinyerjük a kategória neveket és a hozzá tartozó id-at back-end oldalon meg kell hívnunk a ProductCategoryRepositoryt.



Ugye a többesszámá alakított végpont itt nem lenne megfelelő ezért manuálisan product-category -ra lett állítva.

**Kategória nevek a szervíz osztályból:**

private categorySearchUrl='http://localhost:8080/api/product-category';

Új alap URL-t kell megadnunk ami megegyezik az előbb említett végpontal.

Ezt követően újabb Observable listát hozunk létre,ezúttal termék kategóriával.

  getProducCategoryList():Observable<ProductCategory[]> {

    return  this.httpClient.get<GetResponseProductCategory>(this.categorySearchUrl).pipe(

      map(response => response.\_embedded.productCategory)

    );

  }

interface GetResponseProductCategory{

  \_embedded:{

    productCategory:ProductCategory[];

  }

}

Ennek a felépítése megegyezik a termékek lekérdezésénél használtakkal.

Mivel már megvan az id és név, szükségünk van egy osztályra, amelyek feliratkozik erre és ezeket az adatokat html oldalon átadja routerLinkhez.

**Product-list-menu:**

export class ProductListMenuComponent implements OnInit {

   productCategory:ProductCategory[];

  constructor(private productService: ProductServiceService) { }

  ngOnInit(): void {

    this.listOfProductListCategories();

  }

  listOfProductListCategories() {

    this.productService.getProducCategoryList().subscribe(

      data => {

        console.log("The incomming data"+ JSON.stringify(data));

        this.productCategory=data;

      }

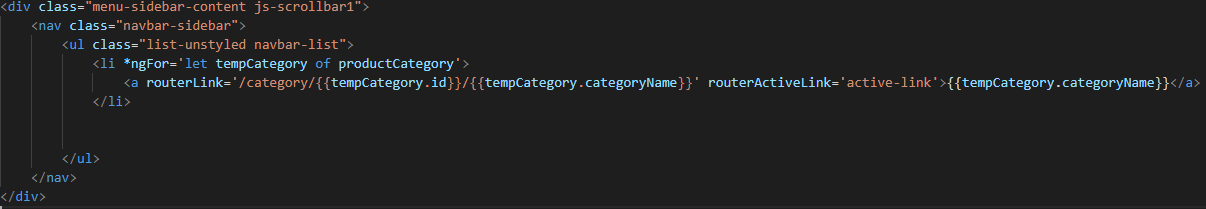
    );

  }

}

Teljesen megegyezik a product-listben használt eljárással, csak ez esetben ProductCategorykat kapunk.

Html oldal:



\*ngFor-t alkalmazva az arrayből minden elemmel hozzárendelünk egy routerLinket a kategória idval és névvel, megjelnéshez pedig a kategória nevét adjuk meg. Ezáltal a kategória nevére kattinva a Router Outlet újratöltődik az új pathal. Az ehhez rendelt komponens(Product-List komponens) űjratöltődik és megvizsgálja az adott URL-t. Kiveszi belőle az adott id-t és felhasználja a Back-end vonalon korábban létrehozott findByCategoryId-t. Ezután rendelkezésünkre állnak a kiválasztott kategória termékei.

**Adott kategória kinyerése szervizben:**

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

    // we have to make that URL access on the backend side

    const searchUrl=`${this.baseurl}/search/findByCategoryId?id=${currentCategoryId}`;

    return  this.httpClient.get<GetResponseProducts>(searchUrl).pipe(

      map(response => response.\_embedded.products)

    );

  }

A baseurl-t kiegészítjük az adott függvényünkhöz szükséges végződéssel, mivel erre a Product-List komponensel fogunk feliratkozni, így ott fogjuk kinyerni a jelenlegi linkből az id-t majd átadni ennek a függvénynek.

**Product-List:**

A routok használatához construktorban injektálnunk kell a ActivatedRoute-t

Majd az ngOnInit-nél feliratkozni rájuk.

 ngOnInit(): void {

    this.route.paramMap.subscribe(

      () =>{

    this. listProducts();

  }

    );

  }

A listProductsban fogalmazzuk meg,hogy a jövőben hozzáadandó keresés funkciót kell a meghívnia vagy a most letrehozandó listProductbyCategory-t.

Ehhez szükséges példányosítanunk egy currentCategoryId-t,aminek a

    this.curretnCategoryId= +this.route.snapshot.paramMap.get('id');

al adunk értéket.

Le kell kezelnünk, hogy adtunk e neki id-t

    const hasId: boolean= this.route.snapshot.paramMap.has('id');

  listProductsByCategory(){

    const hasId: boolean= this.route.snapshot.paramMap.has('id');

    // check if the id is empty or not

    if(hasId){

      // if not empty than we make the id equal with our variable,

      // using the + symbol to convert the String from gthe get method into a number

    this.curretnCategoryId= +this.route.snapshot.paramMap.get('id');

    this.curretnCategoryName= this.route.snapshot.paramMap.get('name');

  }

  else{

    // if it is empty then we set it to 1 by default

    this.curretnCategoryId=1;

    this.curretnCategoryName='Books';

  }

  this.previousCategoryId=this.curretnCategoryId;

    this.productService. getProductList (this.curretnCategoryId).subscribe(

      this.processResultWithPagination()

    );

  }

  processResultWithPagination()  {

    return data => {

      this.products=data.\_embedded.products;

    };

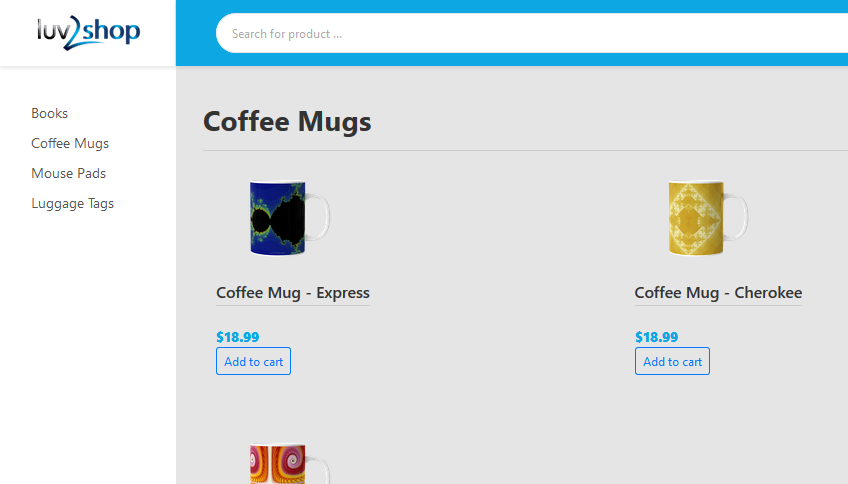
  }

Itt még nem adunk hozzá lapozás segédletet.

Ezzel megkaptuk a kiválasztott kategóriához tartozó termékek listáját, amire html oldalról ugyanúgy tudunk hivatkozni.

Mivel ezáltal megkapjuk a kategória nevét, html oldalon felül kiirathatjuk,hogy jelezzük melyik kategóriára van épp leszűrve.

**Cél:**



Oldalt név szerint leszűrhetőek a termékek kategórai szerint, amit felül ki is ír.

**Keresés a termékek között kulcsszóval:**

Lépések:

1. Back-end vonalon a ProductRepositoryban létre kell hoznunk egy új queryt,ami egy megadott kulcsszóra szűr le a termékek nevei között.
2. Szervízben meg kell hívnunk a REST API-t amivel megkapjuk a kulcsszó alapján leszűrt termék listát.
3. Új route létrehozása,ami ismét betölti a ProductList komponenst és kiszedhetjük belőle a kulcsszót.
4. Search-list komponens létrehozása. Benne egy függvény,ami fogad egy stringet, ami a html oldalról fog érkezni. A függvény meghívja a szervízben deklarált új adatforrásunk.
5. Search-list html oldalán létrehozunk egy inputot és egy gombot, ami meghíja az új függvényünket. Majd ezt mind selector segítségével hozzáadjuk az app.html headeréhez.

**1.**

Page<Product> findByNameContaining(@RequestParam(**"name"**) String name ,Pageable pageable);

Az új lekérdezésünk SQL nyelven a következő: Select \* From product where name LIKE CONCAT(’%’, :name, ’%’)

Ezáltal mindent ki fog listázni,amiben előfordul az adott kulcsszó.

Elérése :

**http://localhost:8080/api/products/search/findByNameContaining?name=Java**

**2.**

  getSearchProduct(theKeyword:string):Observable<GetResponseProducts> {

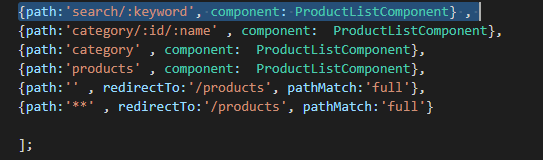
    const searchedUrl=`${this.baseurl}/search/findByNameContaining?name=${theKeyword};

    return  this.httpClient.get<GetResponseProducts>(searchedUrl);

  }

A megszokott módon meghíjukjuk a REST API-t az előzőekben létrehozott végpontal.

**3.**



Lényege:

Route kiegészítése a kulcsszóval. Html oldalon egy textbox által kapjuk meg a az adott szót, amit elküldünk a következőleg létrehozandó Search-list komponensben. Ez a komponens fogadja az adott kulcsszót és beállítja rá a routeot. Ezt követően a route miatt a ProductList komponens újra töltődik,és megvizsgálja,hogy van e a URL-ben keyword, ha van akkor feliratkozik a szervízben deklarált új szűrt adatforrásra. Ezt követően kilistázza azt.

Product-list komponens módosítása:

searchMode:boolean=false;

Új boolean váltoyó deklarálása.

Az ngOnInit-ben feliratkozunk a route paramMap-re,ami által a listProductsban vizsgálni tudjuk az adott routeot.

  ngOnInit(): void {

    this.route.paramMap.subscribe(

      () =>{

    this.listProducts();

  }

    );

  }

  listProducts(){

    const searchMode=this.route.snapshot.paramMap.has('keyword');

    if(searchMode){

      this.listProductsBySearch();

    }

    else{

    this.listProductsByCategory();

  }

  }

A listProducts megvizsgálja, hogy a jelenlegi routeban van e keyword. Ha van akkor a keréséshez megfelelő függvényt hívja meg, különben pedig az eddigi kategória szerintit, ahol defaultbol a könyvekre szűr.

  listProductsBySearch(){

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

this.productService.getSearchProduct(theKeyword).subscribe(

  this.processResultWithPagination()

);

  }

 processResultWithPagination()  {

    return data => {

      this.products=data.\_embedded.products;

    };

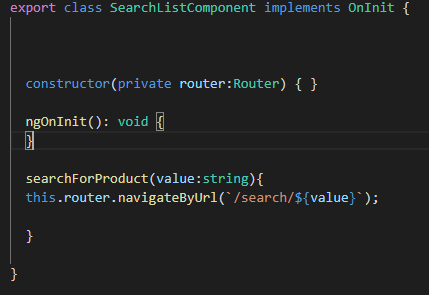
  }

A listProductsBySearch kiszedi a routeból az adott kulcsszót, és meghívja vele a szervízben létrehozott adatforrást.

**4.**

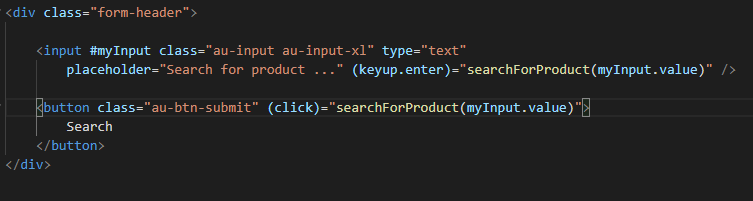
De ahhoz, hogy ez a komponens újratöltődjön, el kell érnünk hogy az adott route meghívodjon.

Ezért létre kell hoznunk a Search-list komponenst.



Injektáljuk a Routert, hogy módosítani tudjuk. A searchForProductot a html fogja meghívni a textboxból kiszedett adattal. Enter megnyomásával vagy a külön ehhez létrehozott gombal meghívódik és kiegészíti a pathot /search/${value} -al. Ezt követően újratöltődik a ProductList komponens.

**5.**

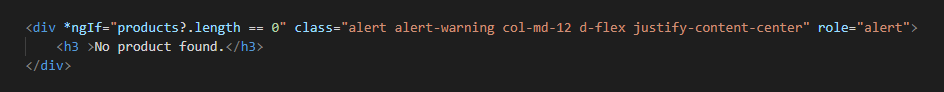


Html oldlon event bindingot hajtunk létre. Vagyis hozzákotjük a korábban létrehozott függvényt az enter lenyomásához és a submit button megnyomásához. A textboxba bekerülendő szövegre myInput névvel tudunk hivatkozni. Ezért az meghívandó searchForProduct függvényünknek áttudjuk adni, és a typeSrcipt osztály letudja kezelni.

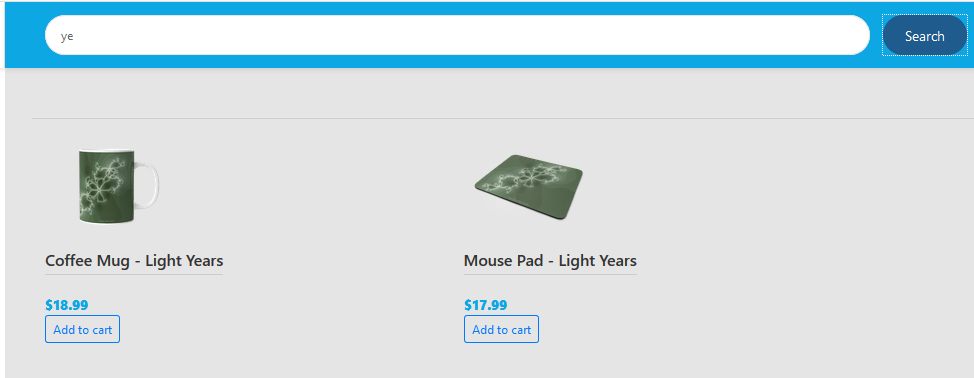
Majd selector segítségvel hozzáadjuk, az app.module header részéhez.

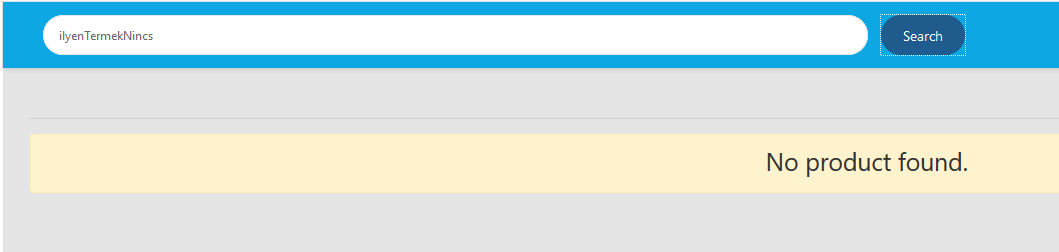
Ha szeretnénk szöveget megjeleníteni abban az esetben, amikor nem lenne termék az adott kulcsszóval, a Product-list html oldalán listázást végrehajto ngFor utan egy ngIf-et kell tennünk.

Ami megvizsgálja, hogy az adott termékeket tartalmazó arrayünk hossza 0 e. Ha igen, akkor kiír egy figyelmeztető szöveget.



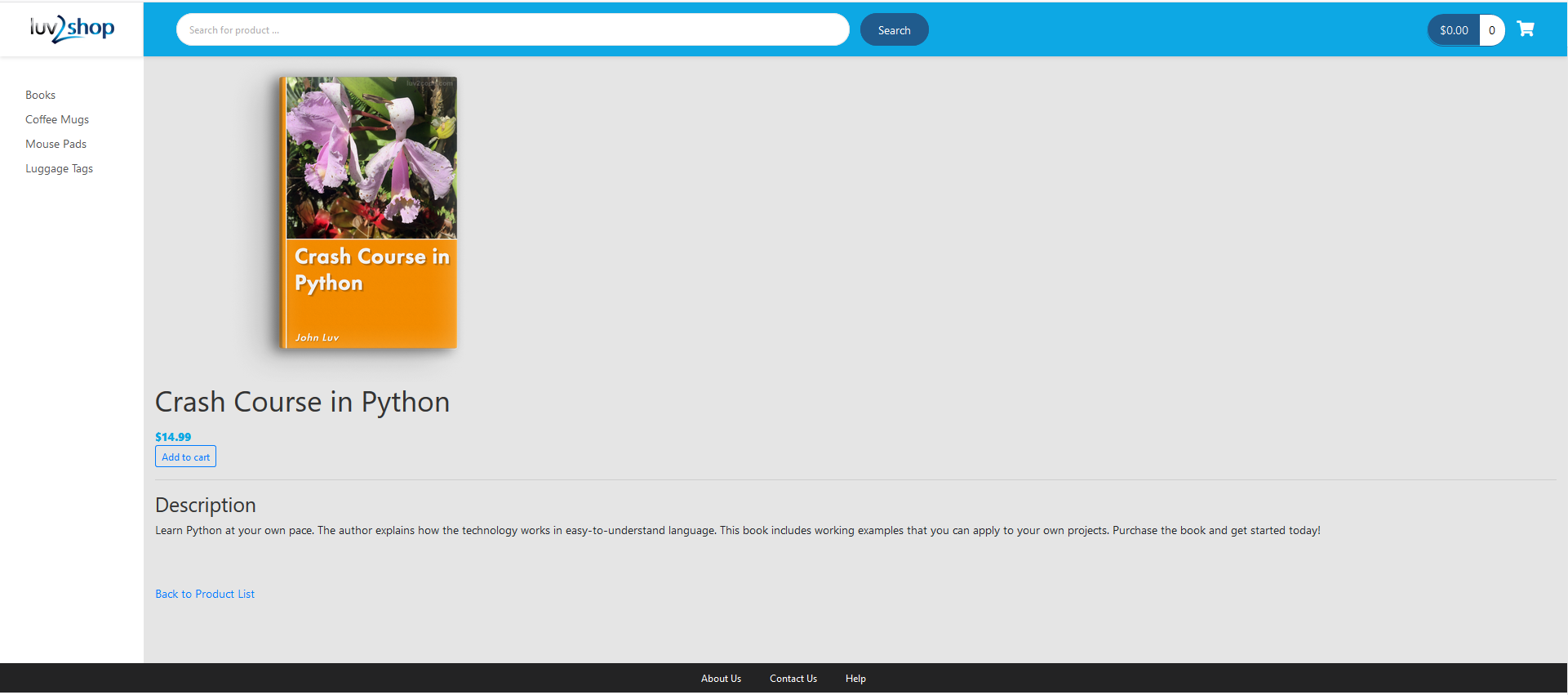
Eredmény:





**Termékek részleteinek megjelenítése(Master view):**

Cél:



A listán a képre vagy a névre kattintva megjelenítjük az adott termékhez tartozó részletes leírást.

Lépések:

1. Új komponens létrehozása, Product-detail-list komponens,amit hozzárakunk a routeokhoz, ahol a products listából id alapján kérdezünk le egy terméket.
2. Route létrehozása,ami az új komponenst tölti be..
3. Szervízben új függvény létrehozása, ami kap egy idt, és az alapján kérést küld a REST APInak, ami a products listából visszaküldi az adott idval rendelkező terméket.
4. Html oldalon létrehozni az új komponenst.
5. A Product-list html vonalon a képhez és névhez hozzá rendelni az új routeot.

**1.**

export class ProductDetailListComponent implements OnInit {

  product:Product=new Product();

  constructor(private productService: ProductServiceService,private route: ActivatedRoute) { }

  ngOnInit(): void {

    this.route.paramMap.subscribe(

      () =>{

    this.getProductById();

  }

    );

  }

  getProductById() {

    const theId=+this.route.snapshot.paramMap.get('id');

    this.productService.getProductByIdService(theId).subscribe(

      data=>{

        this.product=data;

      }

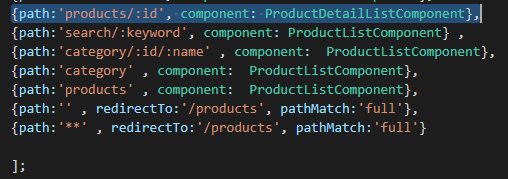
    )

  }

}

Deklarálunk egy Productot,amit a html oldalon megjelenítünk részletesen és injektáljuk a szervízt és az ActivatedRouteot, hogy az ngOnInitnél feliratkozhassunk rá. Majd létrehozunk egy függvényt, ami megvizzgálja a jelenlegi routeot és kiveszi belőle az id-t. Ezzel az idval meghívjuk azt a függvényt, amit majd a szervízben hozunk létre és visszaad egy terméket az id alapján.

**2.**



Az új route a products/:id ra végződik és a korábban létrehozott komponenst tölti be.

**3.**

  getProductByIdService(theId: number):Observable<Product> {

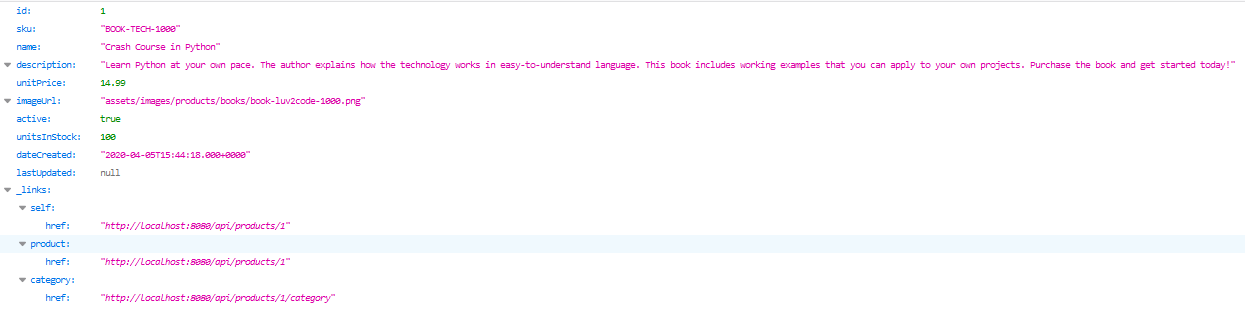
    const detailedUrl=`${this.baseurl}/${theId}`;

    return  this.httpClient.get<Product>(detailedUrl);

  }

Szervíz osztályban létrehozzuk a függvényt, ami meghívja a REST API-t és visszaadja az id alapján a szükséges terméket. Mivel itt csak egy termékről van szó, ezért nem kell külön mappelni az embedded-et, hanem egyből egy Productot tudunk visszaküldeni a feliratkozónak.

Ez új végpont: <http://localhost:8080/api/products/1>



JSON formátumban

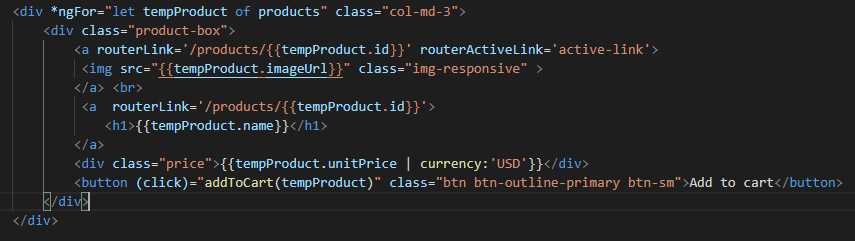
**4.**



A Product-detail-list html oldalon megjelenítjük a képet, az nevet, az árat, egy gombot, ami későbbi kosár funkciót szolgálja(click még nem kell), a leírást, és egy linket vissza a Product listre, ami a routerLink segítségével újra tölti az adott komponens a Product-list el.

A typeScript osztályban deklarált Productra hivatkozva hívjuk meg a termék adatait.

**5.**



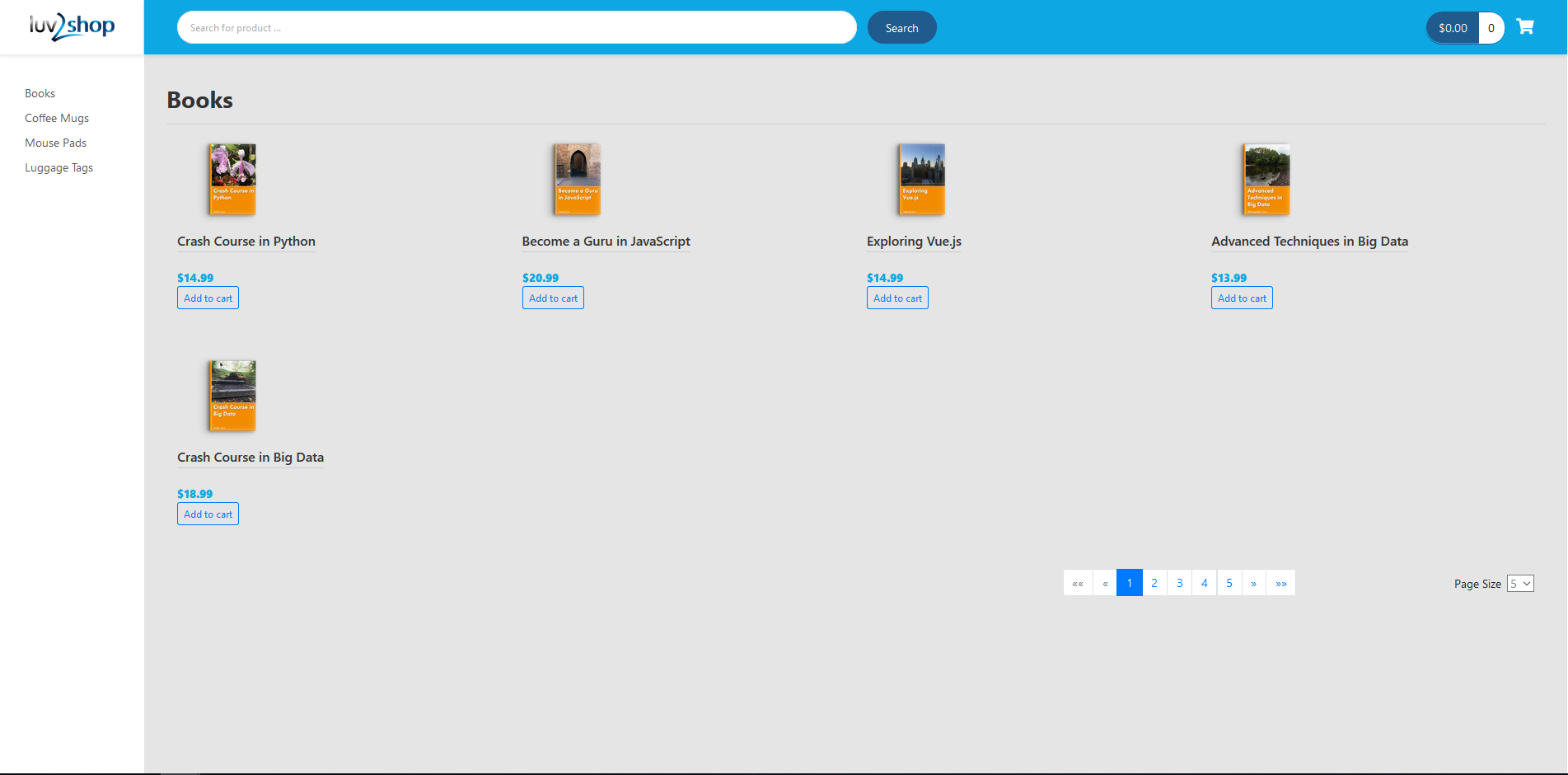
A Product-list html oldalon a képhez és a névhez hozzá rendelünk egy routerLinket, az iterációban éppen soron következő termék id-val. ( routerLink=’/products/{{tempProduct.id}}’)

Ezáltal a a képre kattintva a path kiegészül az id val és betöltődik a Product-detail-list komponens.

A komponens ngOnInitben meghívódik a getProductById() függvény, amely a routeból szedett idval meghívja a szervíz osztály getProductByIdService függvényét. Ez a függvény kérést intéz a REST API hoz az adott id val és visszaküldi a kívánt terméket, amit a html oldalon megtudunk jeleníteni.

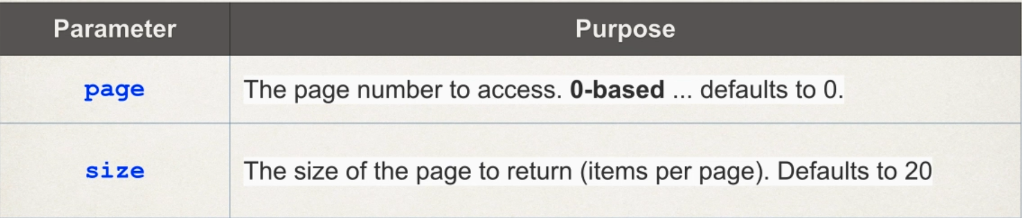
**Lapozás**

Cél:



Alul képeket legyünk lapozni és kiválasztani,hogy egyszerre hány termék jelenhessen meg.

A Paginationnak 2 fontos paramétere van.



A REST API válasza fontos meta információkat tartalmaz.

„page” : {

„size” : 10, az oldal mérete, amennyi elem épp megjelenhet

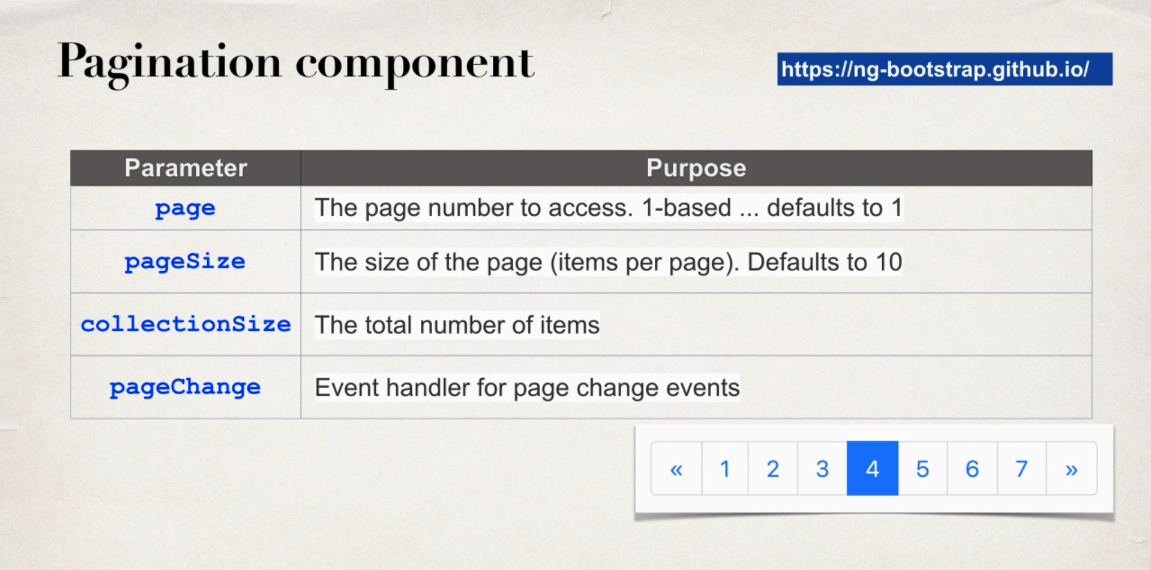
„totalElements” : 200, ennyi eleme van összesan az adatbázisban, csak információs jelleg

„totalPages” : 20, ennyi lap érhető el

„number” : 0 ezen az oldalon vagyunk jelenleg(0 a default)

}

Az ng-bootstrap komponensét fogjuk használni.



Lépések:

1. ng-bootstrap telepítése a komponens használatához
2. Refaktoráljuk a GetResponseProducts interfacet, hogy lekezelje a page-ben lévő metaadatokat. ProductService modosítása,hogy kezelje a Paginationt.
3. ProductList modosítása,hogy kezelje a Paginationt.
4. Html oldalon a Pagination megjelenítése.

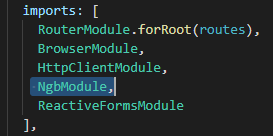
**1.**

A telepítéshez a parancssort kell használnunk, először a dependenciákat Angular 9+-hoz

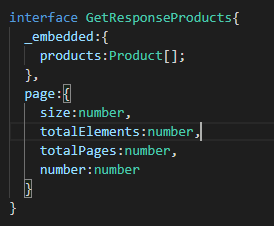
**ng add @angular/localize**

Majd maga a ng-bootstrap telepítése, **npm install @ng-bootstrap/ng-bootsrap**

Ezt az app.moduleba is importálni kell



**2.**



A ProductServiceben módosítjuk az interfacet, hogy lekezelje az új metaadatokat.

  getProductListPaginate(thePage:number,thePageSize:number ,currentCategoryId: number):Observable<GetResponseProducts>{

    // we have to make that URL access on the backend side

    const searchUrl=`${this.baseurl}/search/findByCategoryId?id=${currentCategoryId}`

    + `&page=${thePage}&size=${thePageSize}`;

    return  this.httpClient.get<GetResponseProducts>(searchUrl);

  }

Majd a termékek listázásához új függvényt hozunjk létre, amiben az eddigieket kiegészítjük az oldal számmal, oldal mérettel.

  getSearchProduct(thePage:number,thePageSize:number,theKeyword:string):Observable<GetResponseProducts> {

    const searchedUrl=`${this.baseurl}/search/findByNameContaining?name=${theKeyword}`

      +`&page=${thePage}&size=${thePageSize}`;

    return  this.httpClient.get<GetResponseProducts>(searchedUrl);

  }

Viszont a kereséshez használt függvényt is ki kell egészítenünk, mert ha több termékre is igaz az adott kulcsszó, akkor lehetőség szerint lehessen lapozni köztük.

**3.**

Product-list osztály módosítása.

Deklarálni kell új változókat, amelyekben eltároljuk,hogy épp melyik lapot és milyen méretben jelenítsük meg.

  //for pagination

  thePageNumber:number=1;

  thePageSize:number=5;

  theTotalElements: number=0;

  maxSize:number=3;

  boundaryLinks:boolean=true;

  // pagination search

  thePreviousKeyword:string=null;

Beállíthatjuk a default értékeket. A thePreaviousKeywordot a módosított keresés függvényben fogjuk felhasználni.

  listProductsBySearch(){

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

    if(this.thePreviousKeyword != theKeyword){

      this.thePageNumber=1;

    }

    this.thePreviousKeyword=theKeyword;

this.productService.getSearchProduct(this.thePageNumber-1,this.thePageSize,theKeyword).subscribe(

  this.processResultWithPagination()

);

  }

Miután rákerestünk valamire és megint leszűrjük egy újabb kulcsszóra, megvizsgálja, hogy megegyezik e az előzővel. Amennyiben nem egyeznek meg, a kezdó lapot beállítja az elsőre, majd az előző kulcsszőt egyenlővé teszi az aktuálissal, hogy kovetkezőnek is sikeresen tudja majd vizsgálni.

Ezt követően meghívja a szervíz osztályból a módosított függvényünket, fontos figyelni , hogy a REST API-tót visszaküldött databan a lapszám 0-tól kezdődik. Ezért nekünk le kell vonnunk 1 et a kérés küldésénél.

  processResultWithPagination()  {

    return data => {

      this.products=data.\_embedded.products;

      this.thePageNumber=data.page.number + 1;

      this.thePageSize=data.page.size;

      this.theTotalElements=data.page.totalElements;

    };

  }

Az adat lekezelést is ki kell egészítenünk, hogy a szervíz osztályból érkező új metaadatokat átadja az újonnan létrehozott változóinknak. Itt is figyelembe kell vennünk, hogy a REST APItol a pageNumber 0 indexel kezdődik, nekünk pedig értelem szerűen 1 től indul az oldalak száma, ezért hozzá kell adnunk 1et.

 listProductsByCategory(){

    const hasId: boolean= this.route.snapshot.paramMap.has('id');

    // check if the id is empty or not

    if(hasId){

      // if not empty than we make the id equal with our variable,

      // using the + symbol to convert the String from gthe get method into a number

    this.curretnCategoryId= +this.route.snapshot.paramMap.get('id');

    this.curretnCategoryName= this.route.snapshot.paramMap.get('name');

  }

  else{

    // if it is empty then we set it to 1 by default

    this.curretnCategoryId=1;

    this.curretnCategoryName='Books';

  }

  // if we have different category id than previous

  // then set thePasgeNumber to 1

  if(this.previousCategoryId != this.curretnCategoryId){

    this.thePageNumber=1;

  }

  this.previousCategoryId=this.curretnCategoryId;

  console.log(`the previous category id: ${this.previousCategoryId} and the current: ${this.curretnCategoryId}`);

    this.productService.getProductListPaginate(this.thePageNumber-1,this.thePageSize,this.curretnCategoryId).subscribe(

      this.processResultWithPagination()

    );

  }

A getProductListBHyCategory végső változata.

Először megnézi,hogy a jelenlegi routeban van e id. Ha van, akkor a változóinknak beallítja az értekeket. Ha nincs, akkor default értéknek beállítja a könyveket kategória szűrésre.

Ezt követően megvizsgálja, hogy az előzőleg leszűrt kategóriaId megegyezik a mostanival. Mivel tegyük fel leszűrtük egy kategóriára és ellapoztunk az ötödik oldalig, ha ismét leszűrjük egy másik kategóriára, akkor kezdésnek maradna az ötödik oldalon. Ezért ha nem egyeznek meg beállítjuk hogy az elsó oldlaon kezdjen. Majd hasonlóen a kulcsszónál itt is beállitjuk az előző értékét a jelenlegire.

Majd vegezetül meghíjuk a szervíz osztályból a módosított getProductListPaginate függvényünket az új változókkal. Amely kategóriára szűrve a megfelelő oldalt adja vissza.

**Page Size**

  // pageSize dropdown listhez, html oldalrol bejövő adatot állítja be az itteni size változóra

  // visszarakja az elso oldara es ujra lefuttatja a listazast

  updatePageSize(thePageSize:number){

    this.thePageSize=thePageSize;

    this.thePageNumber=1;

    this.listProducts();

}

Az új változónknak 5 értéket adtunk meg alapból, viszont a lapozás mellett szeretnénk,hogy be tudjuk állítani, hogy mekkora legyen az aktuális oldal mérete. Ezért létrehozunk egy függvényt, amit majd hozzákötünk a html oldalon egy dropdown-listhez. Az ott kiválasztott értéket beállítja a PageSizenak. Mivel új méretet kapott az oldal így visszaállítjuk, hogy az első oldalon legyen. Ekkor még nem történne semmi mivel a termékek még nincsenek leszűrve az új értékek szerint, ezért a végén meg kell hívnunk a listProductsot, ami újból meghívja a REST API-t a szervíz osztály által.

**4.**

            <!-- pagination components-->

<div class="footer-pagination">

<div class="row">

<div class="col-md-6"></div>

<div class="col-md-6">

<div class="row">

    <div class="col-md-9" style="padding-left: 30%;">

        <ngb-pagination [(page)] ="thePageNumber"

                        [pageSize]="thePageSize"

                        [collectionSize]="theTotalElements"

                        [maxSize]="maxSize"

                        [boundaryLinks]="boundaryLinks"

                        (pageChange)= "listProducts()"

                        >

        </ngb-pagination>

    </div>

A Prodcut-List komponens html oldalán a termékek alatt megjelenítjük a paginationt. A kiválasztott page megadja, hogy az oszályban a thePageNumber milyen értéket vegyen fel. A pageSizeban megadjuk, hogy a jelenelegi oldal milyen méretű legyen, a collectionSiteban megadjuk, hogy az elemek száma mennyi lehet. A MaxSizeal a maximum méretet. A boundaryLinksel egy boolean változóval beállíthatjuk, hogy megjelenjenek e oldalt a dupla relácós jelek mikkel az utolsó/legelső oldalra ugorhatunk. A pageChange figyeli, hogy változás esetén meghívja a listProductsot, ami újratölti a termék listát.

    <!-- page size selection-->

    <div class="col-md-3 mt-2" style="text-align: right;">

        <span class="mr-2">Page Size</span>

        <select (change)="updatePageSize($event.target.value)">

            <option selected="true">5</option>

            <option>10</option>

            <option>20</option>

            <option>30</option>

            <option>40</option>

            <option>50</option>

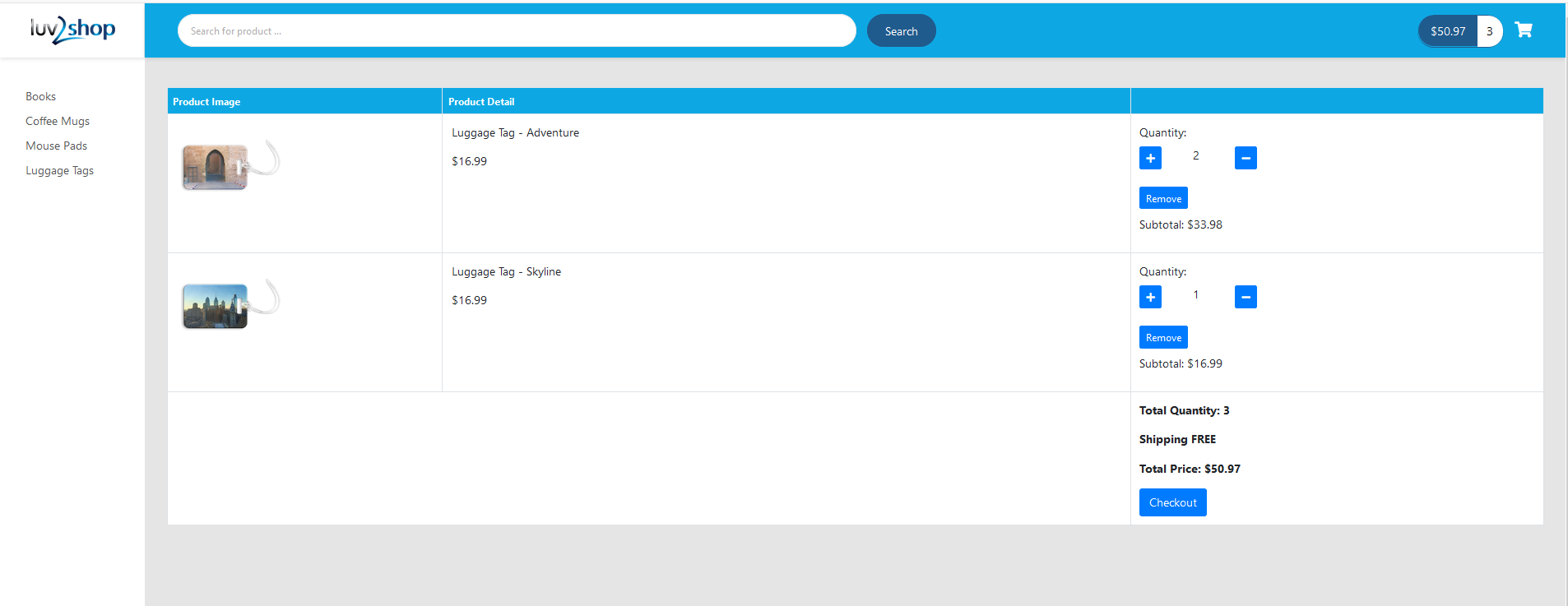
        </select>

    </div>

A méret kiválasztásához használt dropdown-list. Minden kiválasztáskor meghívja az update fuggvényünket, és az épp kiválasztott értéket adja át neki.

**Kosárba rakás, ár és mennyiség jelzése és módosítása**

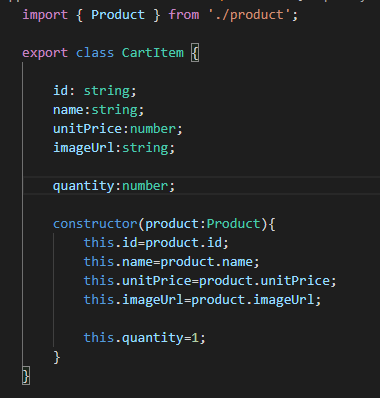
Cél:



Lépések:

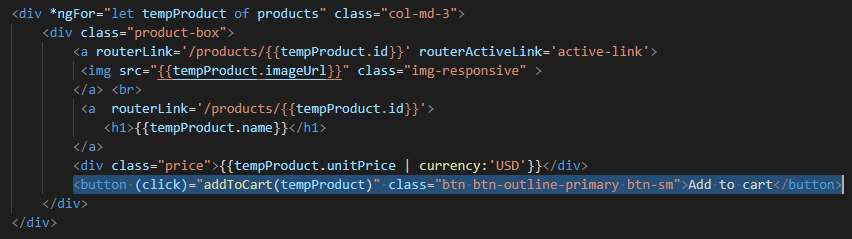
1. Kosárban lévő termék modellezése az aktuális termékekből, ennek segítségével könnyebben számon tarthatjuk a mennyiséget és az összeget.
2. Új szervíz létrehozása, Cart Service.
3. Product-list/detail-list ben addToChart létrehozása.
4. Cart-status komponens létrehozása, aminek segítségével kííratjuk a összmennyiséget és összeget.
5. Új route létrehozása.
6. Cart-status html oldalon a teljes összeg és mennyiség megjelenítése.
7. A kosarat tartalmát részletező komponens létrehozása, Cart-details komponens. CartServiceben az új függvények létrehozása.
8. Html oldlaon a kosár tartalmának megjelenítése.

**1.**



A Productból a fontosabb tulajdonságokat szedjük ki, amiket szeretnénk megjeleníteni. Mivel ez egy kosárban lévő termék tulajdonságait reprezentája, így létrehozáskor a mennyiséget 1 re kell állítani.

**2.**



A Product-list html oldalon felvisszük az Add to cart gombot, ami megnyomáskor meghívja az addToCart függvényt, az eddigiekkel megegyezően event bindigal oldottuk ezt meg.

Ennek a függvénynek átadja az iterációban soron következő terméket.

addToCart

A typeScript osztályban:

  addToCart(tempProduct:Product){

    console.log(`name: ${tempProduct.name} price: ${tempProduct.unitPrice}`);

    const theCartItem=new CartItem(tempProduct);

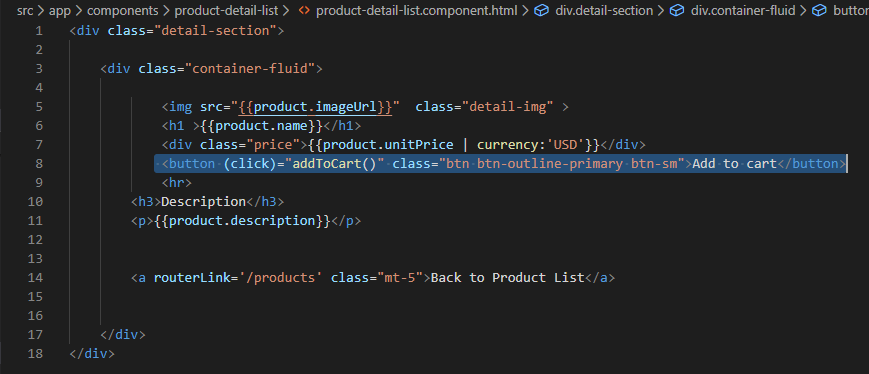
    this.cartService.addToCart(theCartItem);

  }

Bemeneti paraméterként megkapja az adott terméket és létrehoz belőle egy kosárban lévő terméket(a konstruktor miatt a mennyisége alapból 1 lesz). A console.log csak segítség/debugolás miatt van ott.

Ezt követően meghívjuk a Service osztályban létrehozott addToCart függvényt és átadjuk neki a cartItemet.

Hasonlóan járunk el a Product-detail komponensel:



TypeScript oldalon:

addToCart(){

  const cartItem=new CartItem(this.product);

  this.cartService.addToCart(cartItem);

}

Mivel a detail ostályban alapból 1 adott terméket fogalmazunk meg, így az új cartItemnek átadhatjuk az osztály saját termékét. Majd ugyanúgy meghívjuk vele a szervíz osztályt.

**3.**

Cart Service

  CartItems: CartItem[]=[];

  totalPrice: Subject<number>=new Subject<number>();

  totalQuantity: Subject<number>=new Subject<number>();

Létrehozunk egy cartItemeket tároló arrayt, amely számon tartjuk hogy melyik egyedi termékek vannak a kosárban. Ezt követőn létrehozunk két Subjectet(ami az Observablenek egy alosztálya, így más osztályok is feliratkozhanak rá, a szervíz osztály mepid publikálni tudja ezeket az értékeket, így mindig frissülni tud az össz érték és mennyiség.)

addToCart

   addToCart(theCartItem:CartItem){

   let alreadyExist: boolean=false;

   let existingCartItem: CartItem=undefined;

    if(this.CartItems.length>0){

/\*

      for (let tempItem of this.CartItems) {

        if(tempItem.id==theCartItem.id){

          existingCartItem=tempItem;

          break;

        }

      }

\*/

// goes over the array from the first element, if it matches with the specific id then it stops

existingCartItem=this.CartItems.find(tempCartItem => tempCartItem.id==theCartItem.id);

      alreadyExist=(existingCartItem != undefined);

    }

      if(alreadyExist){

        existingCartItem.quantity++;

      }

      else{

        this.CartItems.push(theCartItem);

      }

        this.computeCartTotalts();

   }

Ezt a függvényt a ProductList/DetailList fogja megvíni, az aktuális cartItemet átadva. Kezdő értéknek hamis illetve undefined étékkel felvesz két változót. Arra kellenek,hogy megvizsgáljuk, hogy létezik e már az adott termék a kosárban. Először megnézzük, hogy üres e a kosár,ha nem üres,akkor először végig megy a kosárban lévő termékeken és a soron következő termék ID-át összehasonlítja a bejövő paraméterben átadott termék ID-al. Ha megegyezik akkor azt az existingCartItemet egyenlővé teszi azzal a termékkel.

Emiatt az már nem lesz undefined, emiatt az alreadyExist igaz lesz és a hozzá tartozó if-be bele fog menni. Azon belül megnöveli eggyel az adott termék mennyiségét, mert már létezett, vagyis mi csak hozzáadtunk mégegyet a kosárhoz. Viszont ha nem létezik akkor a beérkező terméket hozzáadja a listához.

Ezt követően meghívjuk a computeCartTotals függvényt, ami a kosában lévő termékek alapján kiszámolja az össz mennyiséget és összeget.

computeCartTotalts

  computeCartTotalts() {

    let totalPriceValue: number=0;

    let totalQuantityValue:number=0;

    for (const currentItem of this.CartItems) {

      totalPriceValue += currentItem.unitPrice\* currentItem.quantity;

      totalQuantityValue += currentItem.quantity;

    }

    this.totalPrice.next(totalPriceValue);

    this.totalQuantity.next(totalQuantityValue);

  }

Létrehozunk benne két ideiglenes változót 0 kezdő értékkel, ezek a totalMennyiség és a totálÖsszeget hivatottak számolni.

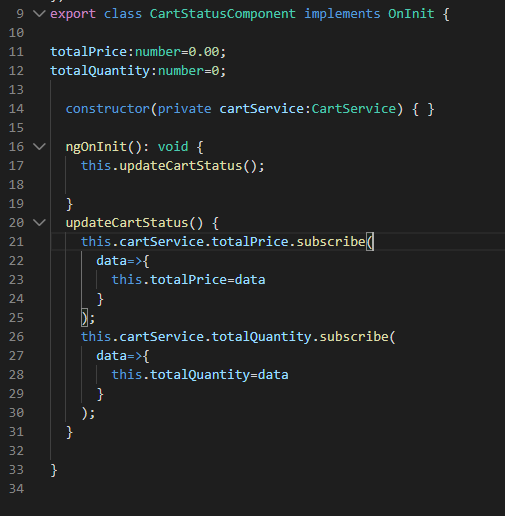
Majd egy forEach iterációt alkalmazunk, ami a kosárban lévő terméklistán egyesével végig megy.

Először termékenként összeszorozza az adott tgermék mennyiségét és egységárát,majd ezzel megnöveli az ideiglenes totalisÖsszeget. Szitén termékenként megnézi az adott termék mennyiségét és hozzáadja az ideiglenes totalisMennyiséghez.

Miutánt végig ment az iteráció az ideiglenes változóink rendelekeznek az össz mennyiséggel és összeggel. Ezért az elején létrehozzott két Subjectünknek átadjuk őket. A next metódust meghívva publikáljuk a két subjectet, átadva a két ideiglenes értékünket. Ezt követően minden osztály,ami fel van iratkozva rájuk megkapja az új értékeket és frissülnek.

**4.**

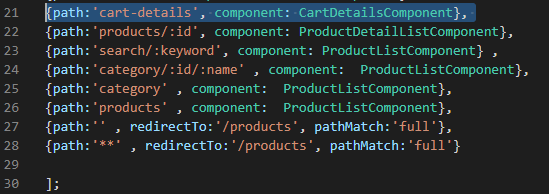
Cart-status komponens létrehozása.



Az új komponensben deklarálunk 2 változót, amit majd megjeleníthetünk a html oldalon, totalPrice és totalQuantity. Injektáljuk a CartServicet. Majd létrehozzuk az updateCartStatus függvényt, amit a ngOninit-ben helyezünk el. A cartServicet használva feliratkozunk a két Subjectnek az eventjére(esemény) és a bejövő értéket átadjuk az osztályban deklarált változókkal, így mindig, amikor meghíjuk a addToCart függvényt a szervíz küld egy eseményt minden felirtakozónak, ezáltal ez az érték mindig frissülni fog ilyenkor.

**5.**

Új route létrehozása



Ezáltal lesz meghívva, a kosár tartalmát részletező CartDetailsComponent

**6.**

<div class="cart-area d-n">

    <a routerLink="/cart-details" >

        <div class="total">{{totalPrice.toFixed(2) | currency:'USD'}} <span> {{totalQuantity}}</span> </div> <i class="fa fa-shopping-cart"

            aria-hidden="true"></i>

    </a>

</div>

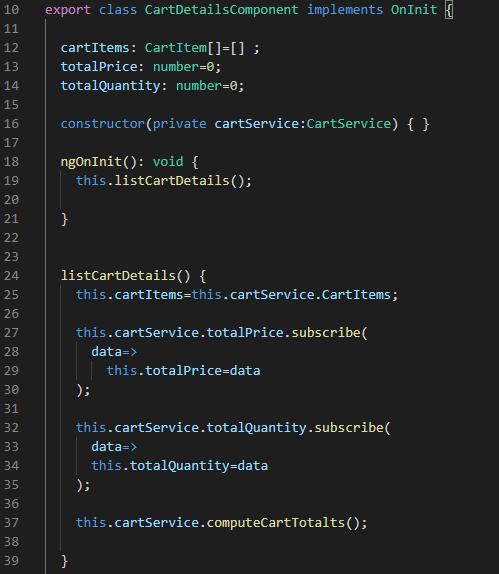
Cart-status html oldalon való megjelenítése, össz mennyiség és összeg. toFixed(2) segítségével 2 tizedesjegyre alakítja. | currency: ’USD’ pedig dollárjelet rak az elejére.

Az egészet beágyazzuk <a ba,hogy linkként tudjuk használni. Megadunk neki egy routerLinket az elobbiekben létrehozott /cart-details el.



**7.**

Cart-details komponens létrehozása.



A Cart-status komponenshez hasonloan epítjük fel ezt a komponenst is.

Kiegészítjük annyival, hogy legyen egy arrayünk a kosárban lévő termékekről, amiknek a tartalmát a szervízen lévő arrayből kapunk meg.

Majd a végén meghyjuk a computeCartTotal, hogy frissítse az adatokat.

Mivel szeretnénk befolyásulni a termékek mennyiségét, szükségünk lesz 3 gombra/függvényre.

incrementQuantity:

 incrementQuantity(cartItem : CartItem) {

    this.cartService.addToCart(cartItem);

  }

A html oldalon minden kilistázott terméknek lesz egz plusz gombja, amihez event binderrel hozzákötjük ezt a függvényt. bemenetnek megkapja a listáról az adott terméket.

Eztunán ezt a terméket átadja a cartServiceről meghívott addToCartnak(mivel a függvény lekezeli,ha már letezik ez a termék és ilyenkor megnöveli az adott mennyiséget eggyel).

  decrementQuantity(cartItem : CartItem){

    this.cartService.decrementQuantity(cartItem);

  }

Hasonlóan ez előzőhoz,csak másik függvényt hív meg a Szervíz osztályból, decrementQuantity.

  decrementQuantity(cartItem: CartItem) {

    cartItem.quantity--;

    if(cartItem.quantity==0){

      this.remove(cartItem);

    }else{

      this.computeCartTotalts();

    }

  }

A függvény csokkenti a bementi paraméterként kapott cartItem mennyiségét eggyel. Majd megvizsgálja, hogy 0 nál több e ezek után a mennyiség, ugyanis ha 0 akkor már nincs ilyen termék a kosárban ezért azt ki kell venni a listából.

Ezt követően meghívjuk a computeCartTotalts-t ,hogy lefrissítse a mennyiséget és összeget.

Remove

  remove(cartItem: CartItem) {

    this.cartService.remove(cartItem);

  }

Minden termékhez tartozni fog egy remove gomb,amelyet hozzákötünk ehhez a függvényhez, és az aktuális terméket adja át neki.

A CartServiceben:

  remove(cartItem: CartItem) {

    const theCartIndex= this.CartItems.findIndex(

      tempCartItem => tempCartItem.id==cartItem.id

    );

    if(theCartIndex > -1){

      this.CartItems.splice(theCartIndex,1);

      this.computeCartTotalts();

    }

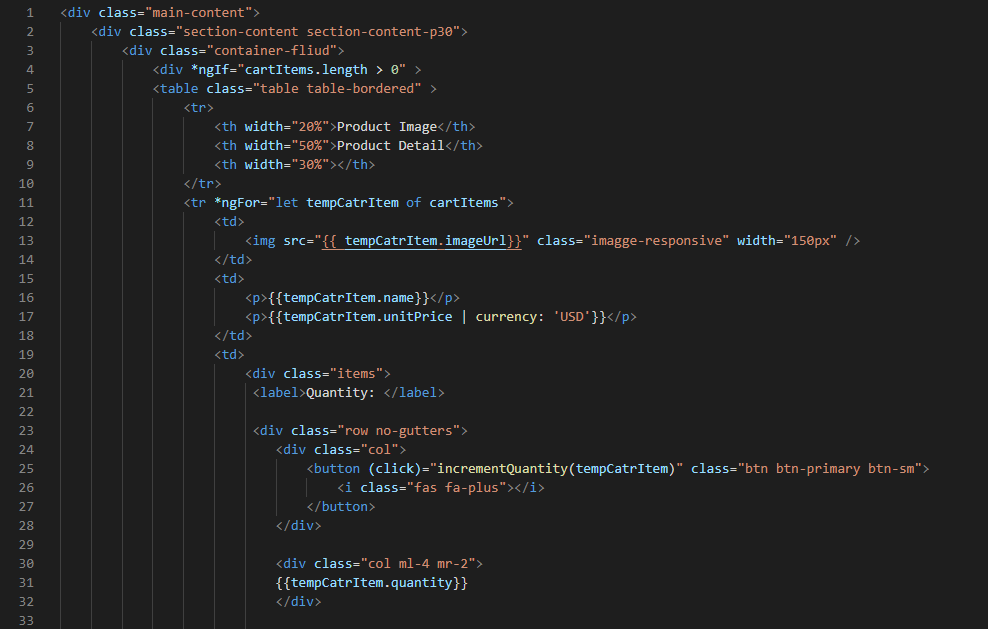
  }

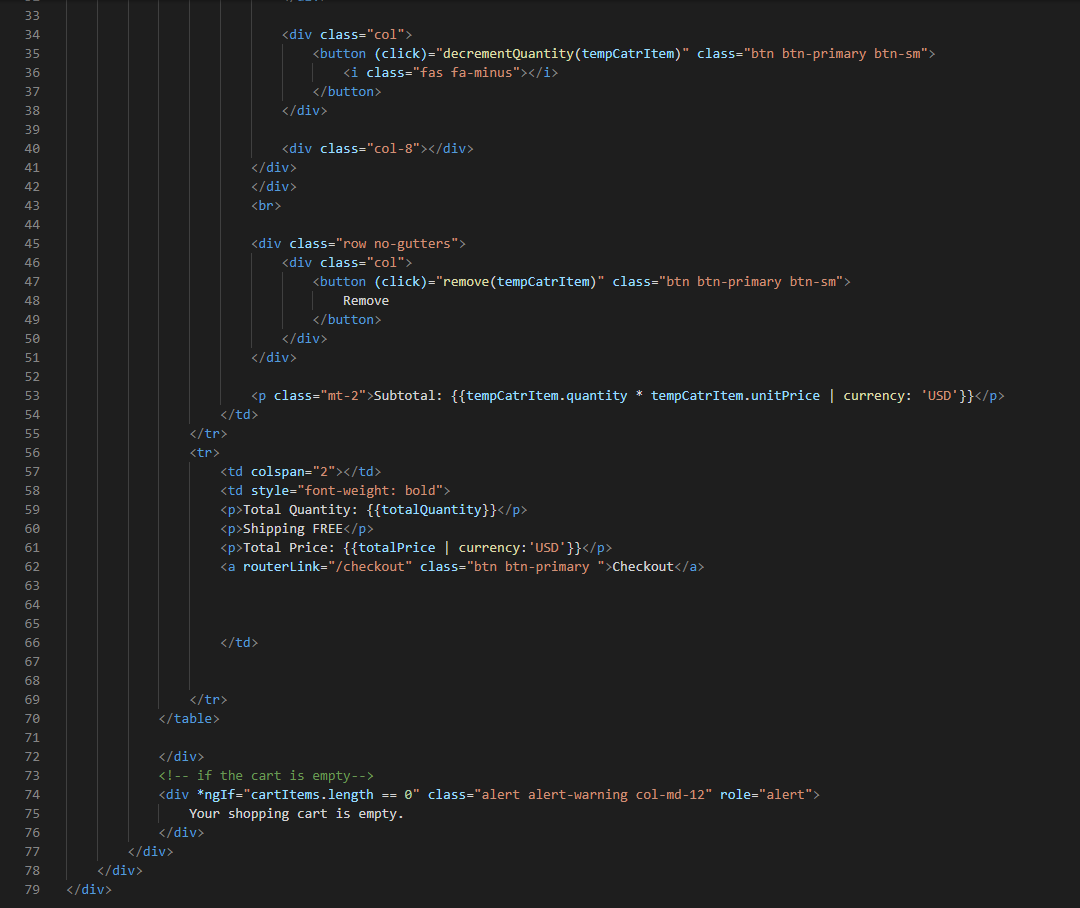
Először végignézi a kosárban lévő terméklistát és mindegyik elemnek összehasonlítja az ID-át a paraméterként átadott termékével. Ha egyezés történik akkor a findIndex visszaadja az adott termék arrayben lévő indexét.

If ben ellenőrizzük, hogy nagyobb e -1nél ergó létezik e az index és megtalálta e. Ha létezik akkor a splice segítségével kiszedjük az adott indexű terméket az arrayből. Ezt követően a computeCartTotalts frissíti az adatokat, hogy kiszedje az össz mennyiségből és összegből a kiszedett terméket.

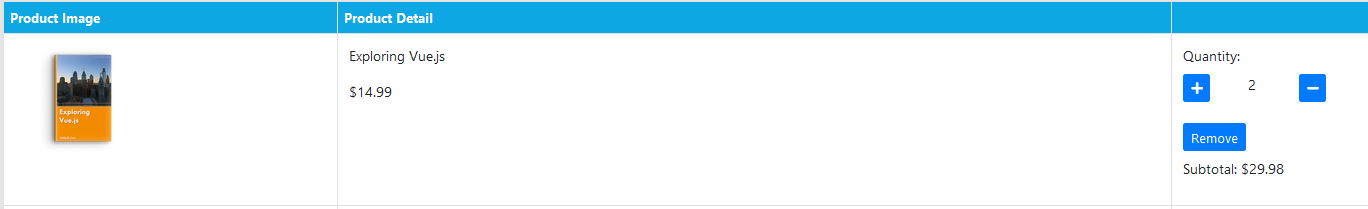
**8.**

Html oldalon való megjelenítés. Ugye routeLinket ranktunk a jobb felső sarokban lévő számlálóra és ikonra, így rákattintás esetén betöltődik a cart-details komponens.





Egy táblazatot hozunk létre első oszlopban a kép majd a név és ár végezetül pedig a mennyiség majd annak növelése és csökkentése, törlése és a végösszeg.



A táblázat sorait (tr) egy ngFor segítségével hozzuk létre, így a kosárban lévő listából minden termékhez rendelünk egy sort és az adott termék mennyiségét és összegét is ki tudjuk számoltatni.

A gombokhoz hozzá kötjük a korábban létrehozott megfelelő függvényeket és átadjuk nekik az adott terméket.

Legalul megjelenítjük a vég összeget és mennyiséget. Egy új <tr/> ben. Ugye ez a tr már nem tartozik bele az ngForba így azután fog megjelenni, hogy a többi sor kigenerálódott. A Checkout gombra tudunk kötni később egy routot amivel egy újabb komponenst tudunk megjeleníteni.



Legalul egy ngIf el lekezeljük, hogy mit írjon ki, ha a kosárban lévő lista üres lenne.

            <!-- if the cart is empty-->

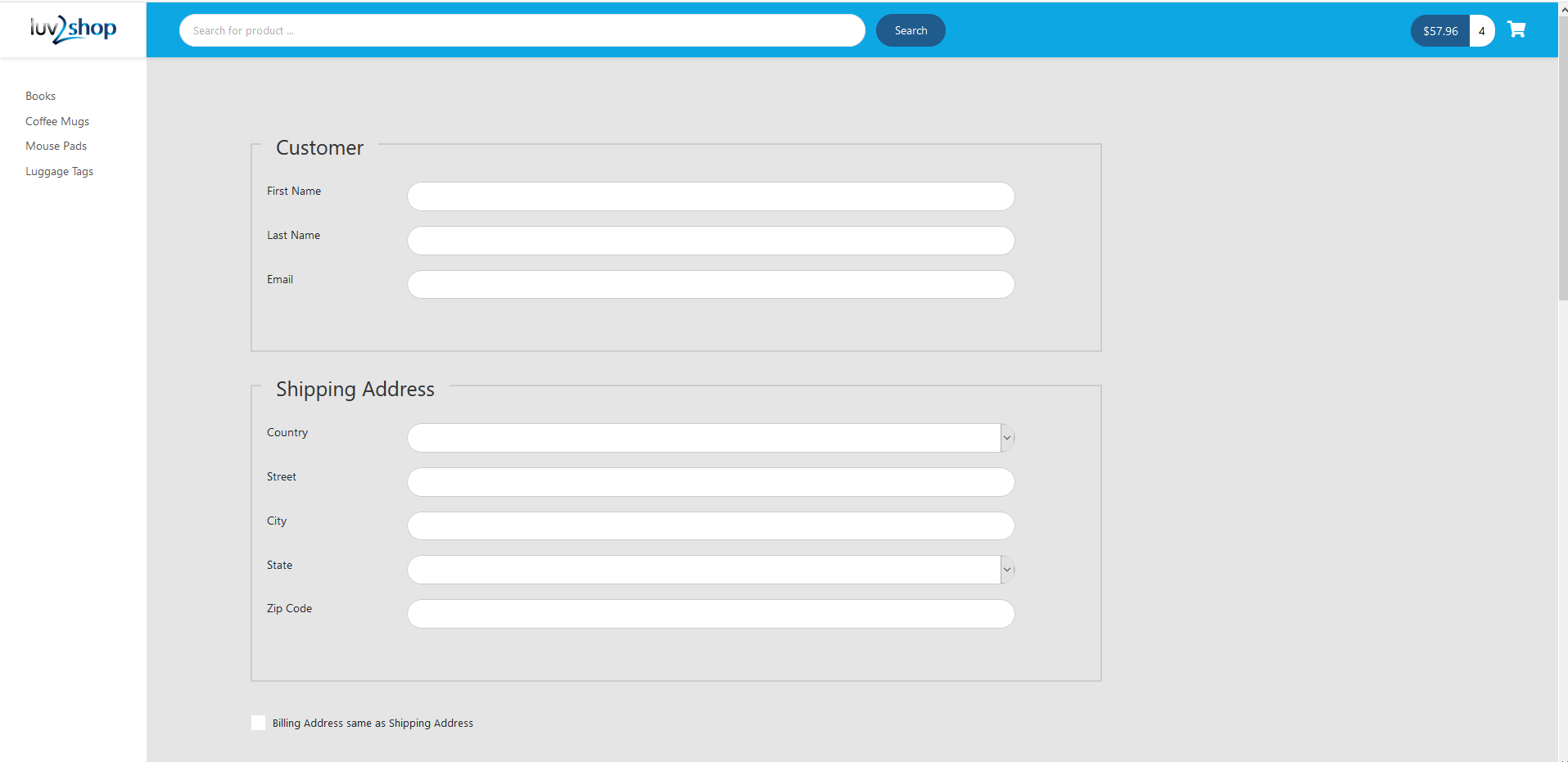
            <div \*ngIf="cartItems.length == 0" class="alert alert-warning col-md-12" role="alert">

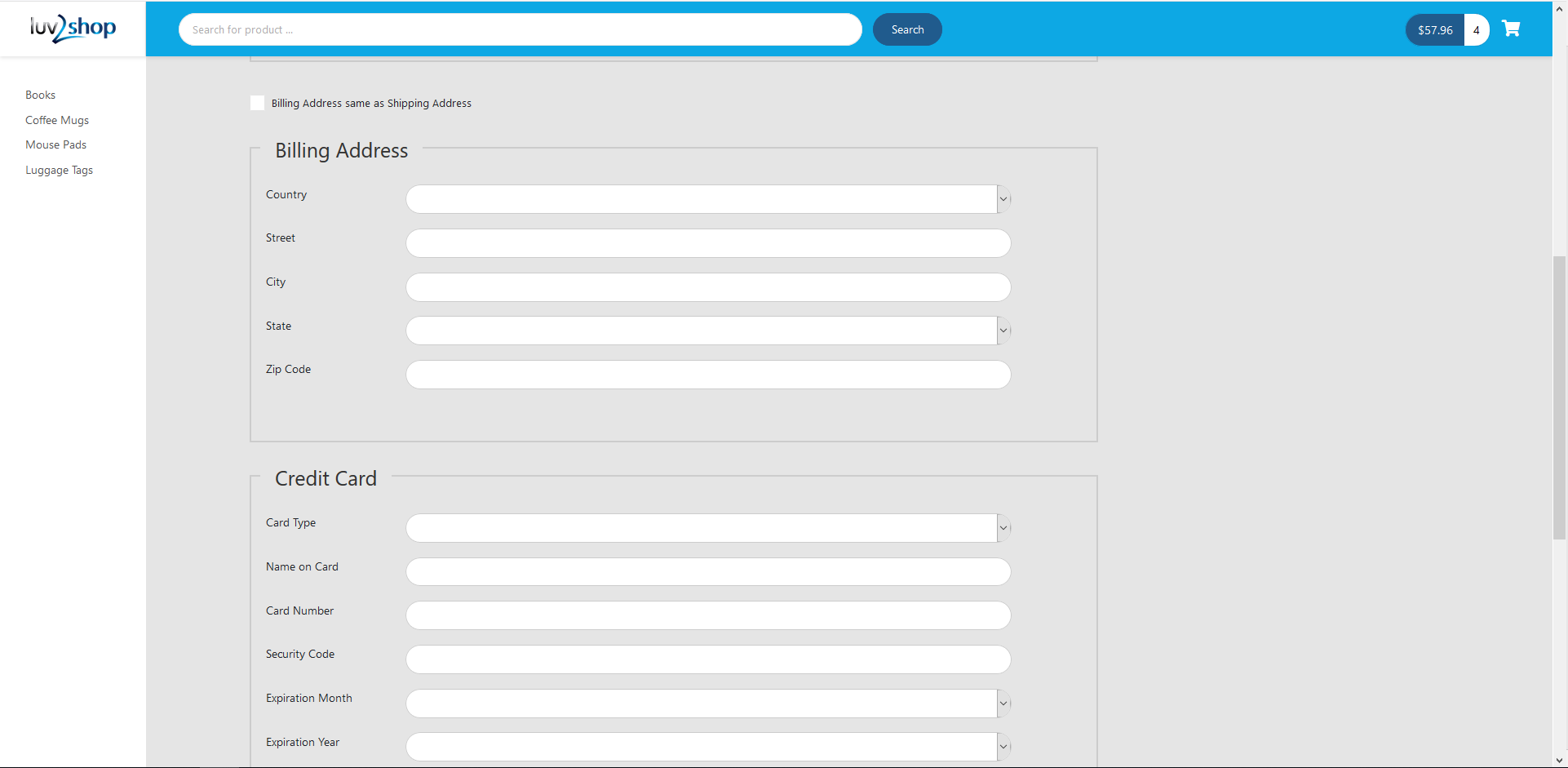
                Your shopping cart is empty.

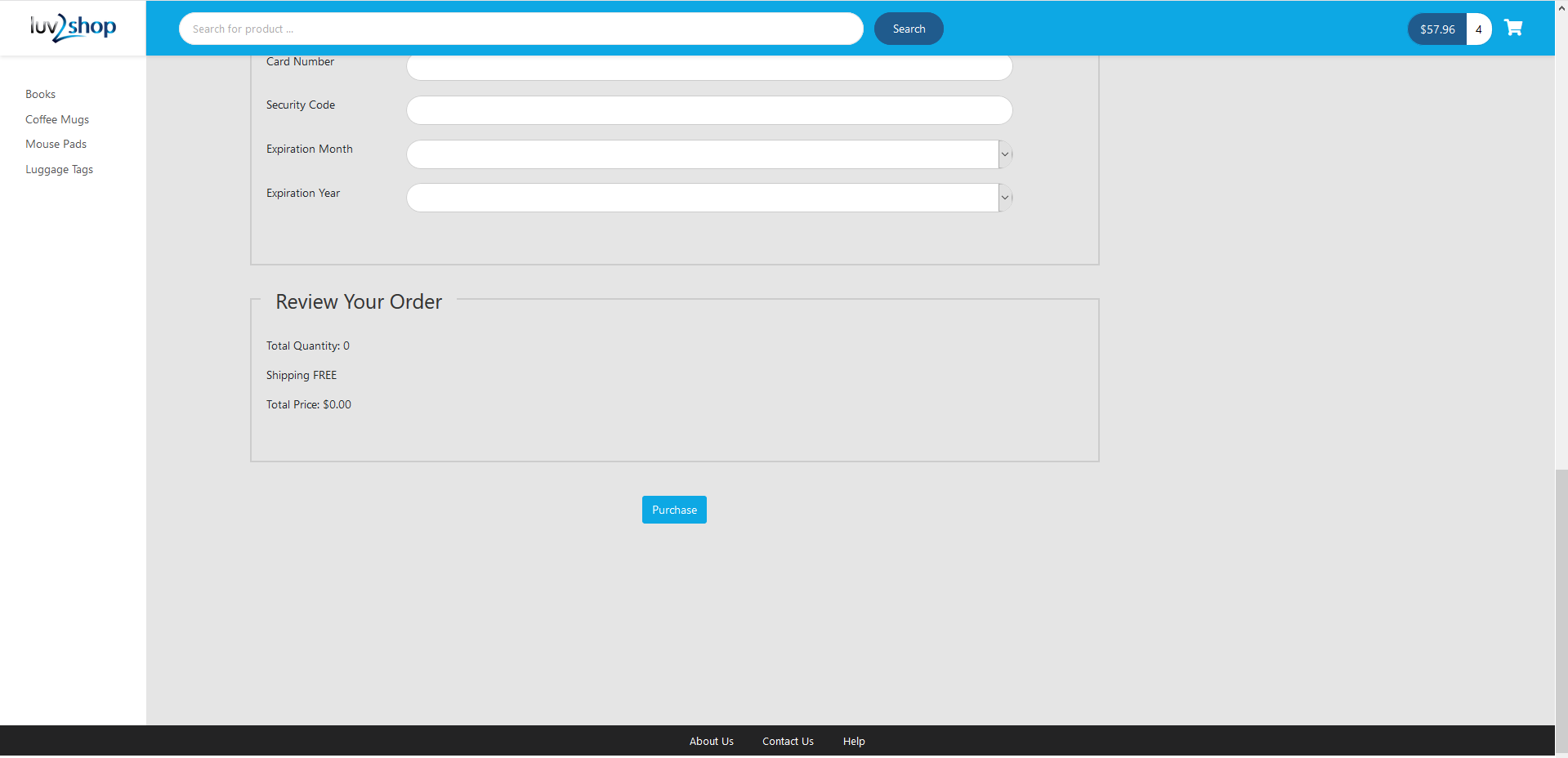
            </div>

**Checkout**

Cél:







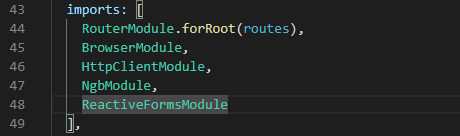
Egy Reactive formot fongunk létrehozni, amelynek különböző adatokat adhatunk meg. Személyes adatok, számlázás, bank kártya stb.

Lépések:

1. Importálni a ReactiveFormsModulet.
2. Új komponens Cheackout komponens generálása.
3. Új route létrehozása és routerLink megadása.
4. Html oldalon való megjelenítés.
5. Checkbox adat kitöltés.
6. Submit és totalPrice,totalQuantity megjelenítés.

**1.**

app.moduleban importáljuk a ReactiveFromsModulet.



**2.**

Új komponens létrehozása. Parancssorban **ng generate component components/checkout.**

export class CheckoutComponent implements OnInit {

  totalQuantity: number=0;

  totalPrice: number=0;

  checkoutFormGroup: FormGroup;

  constructor(private formBuilder: FormBuilder) { }

  ngOnInit(): void {

    this.checkoutFormGroup=this.formBuilder.group({

      customer: this.formBuilder.group({

        firstName: [''],

        lastName: [''],

        email: ['']

      }),

      shippingAddress: this.formBuilder.group({

        country:[''],

        street:[''],

        city:[''],

        state:[''],

        zipCode:[''],

      }),

      billingAddress: this.formBuilder.group({

        country:[''],

        street:[''],

        city:[''],

        state:[''],

        zipCode:[''],

      }),

      creditCard: this.formBuilder.group({

        type:[''],

        name:[''],

        number:[''],

        secCode:[''],

        expirationMonth:[''],

        expirationYear:['']

      })

    });

  }

Deklaráljuk a szokásos két változót az össz mennyiség és összeg reprezentálásához. Majd létrehozunk egy FromGroup változót, amire tudunk hivatkozni majd a html oldalon.

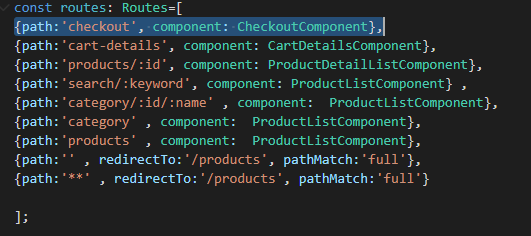
Konstruktorban injektálunk egy FormBuilder, ami által össze tudjuk állítani a reactive formunkat a typeScript oldalon. Az újonnan deklarált checkoutFormGoupunkkat egyenlővé teszzük a FormBuilder group metódusa által generált formal.

A checkoutFormGroup lesz a html oldalon a formGroupName, ami az egész form groupot összefoglalja.

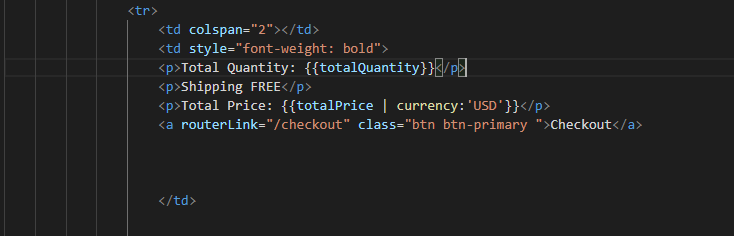
A customer/shippingAddress/billingAddress/creditCard lesznek a formGroupName-ek, amelyeket külön egységet alkotnak, és rájuk hivatkozva tudjuk elérni a bennük lévő formControlName-eket(firstName,lastName stb..).

**3.**

Új route hozzáadása.



Route hozzáadása a Checkout gombhoz, a cart-details komponensen.



Ezáltal rákattintás esetén betöltődik az új checkout formunk.

**4.**



Html oldalon való megjelenítés.

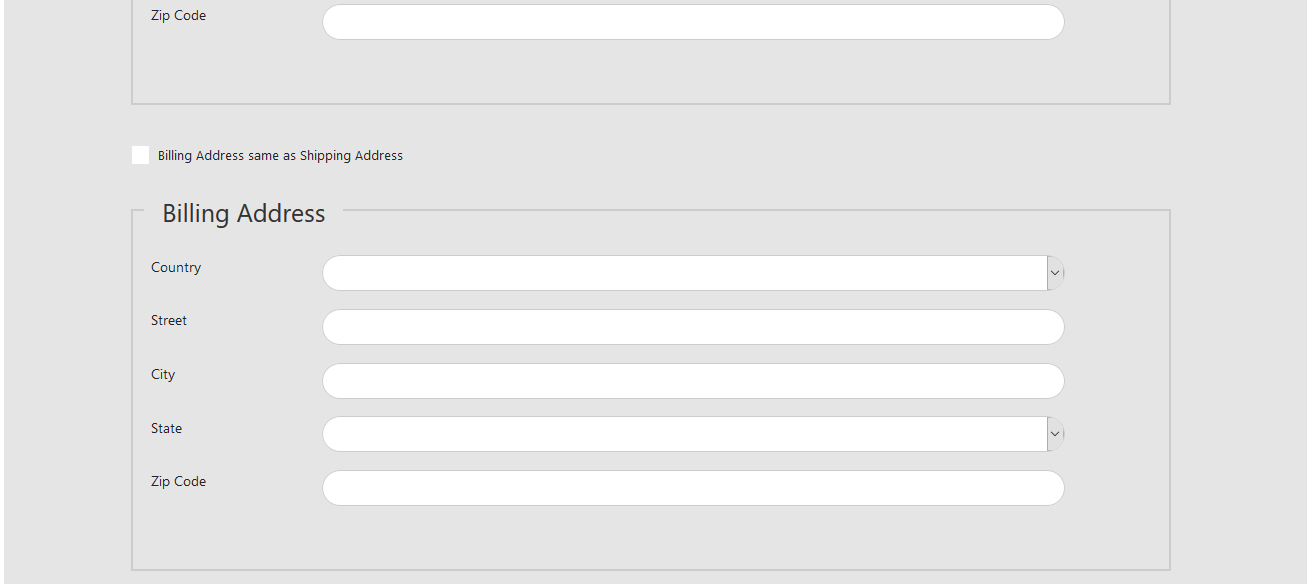
A form tagen belül hívatkozhatunk a formGroupunkra: checkoutFormGroup, ugyanitt a tagen belül definiálhatjuk melyik függvény hívodjon meg, ha submitolunk.

A customer/shippingAddress/billingAddress/creditCard lesznek a formGroupName-ek, amelyeket külön egységet alkotnak, és rájuk hivatkozva tudjuk elérni a bennük lévő formControlName-eket(firstName,lastName stb..).

Egymástól ekülöníthetjük a formGroupNameket: customer/shippingAddress/billingAddress/creditCard.

Ezeket belül megadhatjuk a hozzájuk tartozó formControlNameeket. Customer esetében hivatkozhatunk a firstName,lastName,emailre. Ezáltal azt ezekben megadott adatot tudjuk kezelni tyeScript oldalon.

**5.**



Felviszünk egy checkboxot arra az esetre, ha a Shipping Address megegyezni a Billing Addressel, kipipálás esetén a Shippingben lévő adatokat átmásolja a Billingbe.

                <!-- checkbox-->

                <div class="input-space">

                    <label class="au-checkbox">

                        <input type="checkbox" (change)="copyShippingToBilling($event)">

                        <span class="au-checkmark"></span>Billing Address same as Shipping Address

                    </label>

                </div>

Event bindingal hozzákötjük a copyShippingToBilling metódushoz.

copyShippingToBilling(event){

  if(event.target.checked){

    this.checkoutFormGroup.controls.billingAddress

    .setValue(this.checkoutFormGroup.controls.shippingAddress.value);

  }else{

    this.checkoutFormGroup.controls.billingAddress.reset();

  }

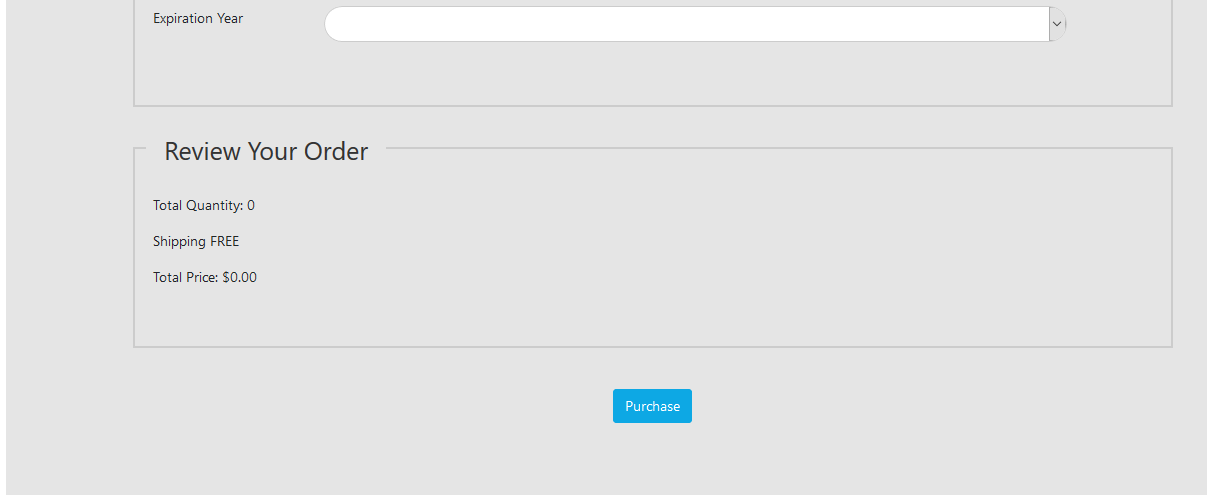
  }

A metódus minden változtatásnál lefut. Az #event et megkapjuk bemeneti paraméterként, így egy ifel megtudjuk vizsgálva, hogy bevan e pipálva. Amennyiben be van pipálva elkérjük a shippingAddressben lévő adatokat, és átadjuk a bullingAddressnek.

Viszont ellenkező esetben pedig billingAddressre meghíjuk a rest metódust, ami kitörli a benne lévő adatokat.

**6.**

A html oldalon az össz mennyiség és összeg megjelenítése. Illetve a Purchase gomb.



onSubmit(){

  console.log(this.checkoutFormGroup.get('customer').value);

  console.log(this.checkoutFormGroup.get('shippingAddress').value);

  console.log(this.checkoutFormGroup.get('creaditCard').value);

}

Az onSubmit() ot a form tagben tudjuk hozzákötni a reaktív formunkhoz.

Egyenlőre submitolás esetén az itt megadott adatokat írhja ki a konzolra.

showPriceAndQuantityStatus() {

  this.cartService.totalPrice.subscribe(

    data=>{

      this.totalPrice=data

    }

  );

  this.cartService.totalQuantity.subscribe(

    data=>{

      this.totalQuantity=data

    }

  );

  this.cartService.computeCartTotalts();

}

A showPriceAndQuantityStatus() metódust az ngOnIniten belül hívjuk meg, ami által feliratkozunk a cartService két eventjére. A computeCartTotalts is meg kell hívnunk, hogy elküldje az adott eventeket a feliratkozók száméra, mivel mikor betöltjük a checkout komponenst még nem kaptuk meg ezeket az adatokat.