

Angular

Getting Started



January 1, 2019

Napredne web tehnologije

Sadržaj

[1 Basics 3](#_Toc535833611)

[1.1 Moduli 3](#_Toc535833612)

[1.2 Komponenta 3](#_Toc535833613)

[1.3 Angular module – Bootstrapping AppComponent 3](#_Toc535833614)

[1.4 Template komponente 4](#_Toc535833615)

[1.5 Binding 4](#_Toc535833616)

[1.6 Directives 4](#_Toc535833617)

[1.7 Data Binding & Pipes 5](#_Toc535833618)

[1.7.1 Property binding 5](#_Toc535833619)

[1.7.2 Event binding 5](#_Toc535833620)

[1.7.3 Two way binding 5](#_Toc535833621)

[1.8 Pipes 6](#_Toc535833622)

[2 Component Basics 7](#_Toc535833623)

[2.1 Strong types + Interfaces 7](#_Toc535833624)

[2.2 Lifecycle hooks 7](#_Toc535833625)

[2.3 Custom Pipes 7](#_Toc535833626)

[2.4 Filtriranje liste (podataka) !!! 8](#_Toc535833627)

[3 Nested components 9](#_Toc535833628)

[4 Services and dependency injection 11](#_Toc535833629)

[4.1 Kreiranje servisa 11](#_Toc535833630)

[4.2 Registriranje servisa 11](#_Toc535833631)

[4.3 Injectanje servisa 12](#_Toc535833632)

[5 Dohvaćanje podataka preko HTTP-a + Observables 13](#_Toc535833633)

[5.1 Observables i Reactive Extensions(RxJS) 13](#_Toc535833634)

[5.1.1 Observable operatori 13](#_Toc535833635)

[5.1.2 Promise vs Observable 13](#_Toc535833636)

[5.2 Slanje HTTP zahtjeva 14](#_Toc535833637)

[5.3 Handleanje grešaka kod dohvaćanja podataka sa servera 14](#_Toc535833638)

[5.4 Subscribeanje na Observable 15](#_Toc535833639)

[6 Navigation and Routing Basics 16](#_Toc535833640)

[6.1 Funkcioniranje rutiranja 16](#_Toc535833641)

[6.2 Konfiguriranje ruta 16](#_Toc535833642)

[6.3 Vezanje ruta s akcijama i prikazivanje viewova 17](#_Toc535833643)

[7 Navigation and Routing Additional Techniques 18](#_Toc535833644)

# Basics

## Moduli

Angular koristi Angular module, JS module i TypeScript module.

Js moduli su uvedeni standardom ES6 ( to je ono import export).

Angular moduli su moduli koje mora imat svaka Angular aplikacija, dakle u svakoj Angular app imamo **barem 1 Angular modul,** a taj se modul zove App module.

Es moduli: - Odnose se na code fileove, organiziraju ih, import i export rade, modulariziraju kod i promiču što veću reusabilnost koda

Angular moduli : - Odnose se na aplikaciju, organiziraju aplikaciju u jednu kohezivnu strukturu koja ima više funkcionalnosti, modularizira aplikaciju i promiče granice unutar aplikacije

* Ako imamo dva modula, svaki ce imati svoje komponente koje medusobno nisu vidljive onom drugom modulu, pa mozemo i kreirati shared modul za komponente koje mogu koristiti oba modula

## Komponenta

Komponenta se sastoji od templatea, klase i metadata. **Template** je view i kreiran je s html-om, a na njega ćemo kasnije vezati binding i directive. **Klasa** sadrži kod za taj view, kreiramo je u TypeScriptu, a ima properties koje sadrže podatke i metodu koje sadrže logiku aplikacije. **Metadata** su extra podaci koji nam služe kako bi Angular prepoznao ovo kao komponentu, a definirane su s dekoratorom.

COMPONENT = TEMPLATE + CLASS + METADATA

**Dekorator** je funkcija koja klasi, njenim članovima i metodama daje metadata kako bi je Angular prepoznao kao komponentu, uvijek ima prefiks @ kako bi je lako uočili u kodu.

DIREKTIVA = IME HTML TAGA

Angular ima neke već ugrađene dekoratore. Npr. @Component() – unutar poziva funkcije dekoratora najčešće se nalazi objekt. Taj objekt ima 2 propertyja – **selector i template**. Selektor ima definiran naziv direktive koji se koristi u HTML-u, odnosno svi tagovi u HTMLu s nazivom te direktive primjenit će na sebe template zadan kao 2.property ovog objekta. Template sadrži html kod koji će se prikazati, a u njemu može biti poziva prema drugim propertyjima. Tako da u templateu imamo binding koji se oznacava s {{ }} i dohvaca nam property od klase. Direktiva je zapravo custom made html tag koji koristimo.

Klasa nije komponenta ako joj se ne pridjeli dekorator funkcija sa metadata !!!!

## Angular module – Bootstrapping AppComponent

U app.module.ts exportamo klasu AppModule i povezemo je sa metadata preko dekoratora **@NgModule** u kojem u propertyju imports navedemo dodatne ukljucene module, u declarations i bootstrap dodamo komponente koje ćemo koristiti u tom modul pri čemu je jako bitno da je html tag u indexu definiran u selektoru te komponente. Komponente koje ukljucujemo dodajemo preko importa.

## Template komponente

Razlikujemo dvije vrste templatea koje možemo pozvati u dekoratoru kao property. **Inline template** poziva se kao template property i to je obicni HTML unutar stringa, a **linked template** poziva se kao templateUrl property i to je URL do datoteke u kojoj se nalazi HTML.

Svaki selektor koji smo pridjelili nekoj komponenti (to je direktiva) unutar dekoratora možemo koristiti u templateima unutar drugih komponenti odnosno unutar njihovih pripadajućih dekoratora. Odnosno to znaci da mozemo ubacivati komponente jedne unutar drugih, ali one moraju biti definirane u istom modulu da bi bile vidljive, a ako želimo koristiti komponente drugog modula samo importamo taj modul.

## Binding

Binding nam povezuje template sa klasom i njenim propertyjima i metodama, odnosno binding je taj koji prenosi podatke između njih dvoje. Binding je uvijek definiran u template-u, a ne u klasi. Razlikujemo više vrsta bindinga, najviše se koristi **string interpolation,** a označavamo ga sa {{ text }} i onda umjesto toga se to poveže s definiranim propertyjem klase text i ispiše string koji je tu zapisan, a tu može bit i metoda i operacija.

Primjeri: <h1>{{pageTitle}}</h1> {{'Title: ' + getTitle() }}

<h1 innerText= {{pageTitle}}></h1>

{{pageTitle}} 🡪 template expression 🡪 ovo je 1 way binding( property)

## Directives

Custom HTML tagovi ili atributi koji se koriste kako bi nadogradili naš HTML. Imamo custom i builtIn angular tagove. BuiltIn tagovi još se nazivaju i **strukturalne direktive**. Prepoznajemo ih tako što imaju oznaku \* ispred svoga imena. Primjer su **\*ngIf** i \*ngFor. One utječu na strukturu HTML-a, odnosno hoće li se nešto prikazati ili neće ovisno o određenom uvjetu (\*ngIf). Tako se ako uvjet nije ispunjen u DOM-u ne prikazuje ni roditelj niti jedno njegovo dijete. **\*ngFor** određuje koliko će se puta ponoviti određeni dio strukture HTML-a zajedno sa svojom djecom. Primjer:

\*ngIf = 'products && products.length ' \*ngFor='let product **of** products'

U \*ngFor se koristi **of** jer to ispisuje elemente objekta, a da koristimo in ispisivalo bi indeks elementa.

U TypeScriptu se koristi tip podataka **any** kada ne znamo ili nam nije bitno koji tip podataka imamo, ali kako je TS strogo određen i mora imati tipove podataka zato je to i uvedeno.

## Data Binding & Pipes

**Binding** može ići u jedan smjer i oba smjera. Kada idemo iz klase u template odnosno html onda je to property smjer, a kada želimo od usera dobiti neku informaciju npr. kada on klikne na gumb onda je to event smjer. Postoji i obostrani smjer, kada želimo postaviti propertyje i primati eventove. Razlikujemo 4 vrste bindinga (3 + interpolation).

### Property binding

<img [src] = 'product.imageUrl' > 🡪 koristi se kada ima više elemenata u nekom nizu i za svaki treba prikazati sliku tj.svaki je source drukciji, to je 1 way binding od propertyja klase do propertyja elementa

Element property je src, a Template Expression je product.ImageUrl.

Binding target je u [ ], a binding source je u ' '.

### Event binding

<button (click) = 'toggleImage()'> 🡪 koristi se kada zelimo uhvatiti korisnikovu radnju klikavanja, a to je 1 way binding od propertyja elementa odnosno eventa do metode klase koja odgovara na taj event

Target event je u ( ), a Template Statement u ' '.

### Two way binding

<input [(ngModel)] = 'listFilter'> - u templateu

Export class ListComponent { - u klasi

listFIlter: string = 'cart';

}

<h4>Filtered by: {{listFilter}}</h4>

Za 2 way binding koristi se i event i property binding, te se uvijek koristi **ngModel** unutar [()] odnosno **banana in the box**, a izjednaci se sa template expressionom u ' '. ngModel je direktiva koju moramo importati pomoću FormsModule inače nam neće moći raditi program.

## Pipes

Pipes se koriste kako bi mi neki podatak prikazali u željenom formatu, npr.lowecase, uppercase, određena valuta, a pipes su najjednostavnije upravo za to. Pipes se pišu unutar template expressions, dakle ili u {{ }} ili unutar ' ' ( u bilo kojem tipu bindinga ih možemo stavit ).

{{ Product.price | currency }} 'product.productName | lowercase'

# Component Basics

## Strong types + Interfaces

Kad nemamo tip podatka koji nam odgovara od postojećih u TypeScriptu kreiramo svoj preko interface-a te ga možemo svugdje koristiti u aplikaciji ako ga export/importamo.

**Interface** nema inicijalizirane propertyje i metode već samo deklarirane i svi moraju imati definiran tip podatka. Ime interfacea počinje sa I, a nastavlja se sa imenom koje sami odabiremo ovisno o kontekstu toga što radimo. Ako želimo dodati neke metode interfaceu najbolje je business klasu deklarirati i u njoj staviti metode, a ta klasa će naslijeđivati interface.

## Lifecycle hooks

Svaka komponenta ima svoj životni vijek. Naprije se kreira, pa zatim renderira, i onda kreira i renderira svoju djecu. Nakon toga obrađuje sve promjene i taj se proces odvija sve dok se komponenta ne uništi.

**OnInit:** koristi se pri kreiranju komponente, dohvaća podatke

**OnChanges:** koristi se kada pratimo promjene na inputu i moramo reagirati s nekom akcijom

**OnDestroy:** koristi se pri uništavanju komponente

Lifecycle hooks su interfacei i ako ih želimo uključiti implementamo ih u klasi i importamo prije. Metoda tog interfacea zove se kao i interface te ima prefiks ng. Interfacei su dio TypeScripta stoga je obavezno navesti tip podataka koji ta metoda vraća.

## Custom Pipes

Kad nam ne odgovaraju ugrađeni pipeovi možemo napraviti svoje. Kreiranje je jako slično kao kreiranju komponente. Imamo klasu za taj **Pipe** koja implementa **PipeTransform**. Tu klasu povežemo s dekoratorom @Pipe kojem specificiramo ime pipea. Unutar klase imamo **metodu transform** koja će promijeniti danu vrijednost u ono što joj mi odredimo.

Da bi koristili custom pipe samo ju dodamo unutar pipea i sa : ' ' deklariramo koju promjenu želimo. Pipe moramo deklarirati unutar modula u kojem je komponenta koja koristi pipe.

<td>{{ product.productCode | convertToSpaces: '-'}}</td>

Transform(value: string, character: string): string {

}

Value = product.productCode, character = '-', ime pipea = convertToSpaces ( name property u dekoratoru)

## Filtriranje liste (podataka) !!!

Angular developeri ne preporučaju filtriranje preko pipea (kad se učitavaju elementi preko \*ngFor).

Najbolja je praksa kako ne bi promijenili listu elemenata sa filtriranjem, stvoriti novu listu elemenata u koje ćemo spremati one elemente koji zadovoljavaju kriterij filtriranja. Zato treba provjeravati što je korisnik unio u filtar redovito i usklađivati to s prikazanim elementima kako bi imali dobar user experience.

Najbolji način za to ostvariti je za property u koji spremamo string koji je korisnik unio u filtar postaviti **getter i setter metode**. Tako svaki put kada se promijeni property od strane korisnika zovemo setter, a kad želimo dohvatiti vrijednost zovemo getter. U setteru možemo implementirati funkcionalnost filtriranja kako bi što brže dobili filtrirane elemente.

S obzirom da nam taj property ima getter i setter, ime propertyja ima prefiks \_.

\_listFilter : string;

get listFilter() : string {

return this.\_listFilter;

};

set listFilter(value:string) {

this.\_listFilter = value;

this.filteredProducts = this.listFilter ? this.performFilter(this.listFilter) : this.products;

}

filteredProducts: IProduct[];

constructor() {

this.filteredProducts = this.products;

this.listFilter = 'cart';

}

performFilter(filterBy:string) : IProduct[] {

filterBy = filterBy.toLocaleLowerCase();

return this.products.filter((product: IProduct) => product.productName.toLocaleLowerCase().indexOf(filterBy) !== -1);

}

Metoda performFilter vraća niz elemenata koje unutar svog naziva imaju substring koji je jednak filterBy (to provjeravamo sa metodom indexOf).

# Nested components

Komponente možemo staviti jednu u drugu, pa onda još i u treću odnosno komponenta u komponenti unutar komponente. Komponentu možemo koristiti kao direktivu tj. pozvati je preko custom HTML taga ili kao routing target (koncept koji se obrađuje kasnije).

Nested komponente su one komponente čiji templateovi upravljaju samo fragmentom većeg viewa tj. container komponente. Nested = parent, Container = child

Nested komponenta prima informacije od Container komponente pomoću **input propertyja**, a šalje informacije Container komponenti pomoću **output eventa**.

**@Input() dekorator** (koji je funkcija) koristimo unutar klase nested komponente kako bi definirali da je to property koji ćemo primiti od container komponente. Taj property šaljemo preko property bindinga u html-u.

**@Output() dekorator** (koji je funkcija) također koristimo unutar klase nested komponente, ali kako on vraća event onda mora biti **tipa objekta EventEmitter**. Taj event ima payload u kojem se šalju podaci nazad do container komponente.

U nested komponenti deklariramo @output dekorator s nekim imenom i za određenu komponentu kojoj želimo pratiti akciju u html-u za tu komponentu napravimo event binding. Tako se svaki put kad npr.kliknemo na div preko funkcije koja prati klikove zovemo naš output. Taj se output šalje u html container komponente tamo gdje želimo dohvatiti podatke. Payload koji smo poslali dohvaća se sa **$event**. Na kraju, u deklaraciji container komponente definiramo funkciju koju smo dohvatili iz htmla zajedno sa njenim payloadom i definiramo što želimo da se dogodi.

1. Definirati radnju koju ćemo slati iz child u parent komponentu preko event bindinga u child komponenti.

(click)="onClick()

onClick(): void {

this.ratingClicked.emit(`The rating ${this.rating} was clicked!`);

}

1. Definirati @Output u child komponenti koji će se slati parent komponenti, a mora biti tipa EventEmitter.

@Output() ratingClicked: EventEmitter<string> = new EventEmitter<string>();

1. U parent komponenti unutar html-a dohvatiti @Output preko event bindinga. Dodatni podaci koji se šalju sa outputom dohvaćaju se sa znakom $event.

(ratingClicked)='onRatingClicked($event)'

1. U parent komponenti definirati metodu koja dohvaća payload outputa i definirati ono što želimo da se s tim podacim dogodi.

onRatingClicked(message: string): void {

this.pageTitle = 'Product List: ' + message;

}

# Services and dependency injection

Servise koristimo kako bi spremili informacije koje nam nisu vezane uz komponentu tj. uz view nego nam trebaju kroz cijelu aplikaciju, a da bi ih ubacili u kod koristimo dependency injection. Servisi su klasa neovisna o komponentama i daju nam podatke i logiku koju ćemo ubacivat kroz komponente.

Kako bi ubacili servis u komponentu koristimo dependency injection. **Dependency injection** je uzorak kodiranja u kojem će klasa komponente dobiti instancu objekta koji joj je potreban ( dependency ) iz vanjskog izvora, a ne da sama kreira tu instancu. Vanjski izvor u Angularu je **Angular Injector.**

U **Angular injector** nalaze se svi servisi koje trebaju koristiti komponente. Na ovaj način servisi su odvojeni od komponenata i u zasebnom su dijelu aplikacije pa tako svaka komponenta odmah vidi promjene na servisima tj. promjene na podacima (globalno). Da sami instanciramo objekt unutar komponente te bi promjene nad servisom bile lokalne i ne bi ih druge komponente vidjele.

Kako bi iskoristili Angular Injector, registriramo samo jednu instancu servisa nazvanu **singleton** unutar cijelog Angulara. Sve te instance servisa registrirane su unutar Angular Injectora i on ih mijena, a tako su vidljive svim komponentama. Kada komponenta želi koristiti neki servis mora ga dodati kao **dependency** (instancu objekta) u konstruktoru tj. mora ga injectati pri instanciranju same sebe. Taj postupak nazivamo dependency injection.

## Kreiranje servisa

Servis se kreira jako slično kao i komponente i custom pipes. Kreiramo klasu, dekorator **@Injectable** sa metapodacima te importamo sve što nam treba. Servis se može koristiti za dijeljenje podataka, ali i za enkapsulaciju podataka te tako osiguravamo da sve komponente prime ispravne podatke te da odredimo koje komponente imaju pristup podacima (servisu), a koje ne.

## Registriranje servisa

Servis registriramo u **Angular Injectoru**. Razlikujemo dvije vrste injectora: root i component. Root injector je injector koji djeluje na cilu aplikaciju, a servisi koji su u njemu registrirani vidljivi su svim komponentama i drugim servisima. Component injector je injector samo za tu komponentu i njenu djecu ili nested komponente, servisi u tom injectoru vidljivi su samo komponenti i njenoj djeci. Za razliku od root-a, component injector nam omogućava više instanci servisa, svaku za po jedan chil od parent komponente.

Root injector tj. servisi koji su vidljivi u cijeloj aplikaciji označavaju se u dekoratoru @Injectable sa propertyjem **providedIn**.

@Injectable({

providedIn: 'root'

})

Component injector tj. servisi koji su vidljivi samo komponenti i njenoj djeci označavaju se unutar te komponente sa propertyjem **providers** i nazivom servisa.

@Component({

selector: 'pm-products',

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

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

providers: [ProductService]

})

## Injectanje servisa

Dependency injection u Angularu tj. u TypeScriptu radi se unutar konstruktora aplikacije. Konstruktor se pokreće kad se komponenta inicijalizira. Kao parametar konstruktora uzme se servis i onda se u tijelu konstruktora on postavi kao servis (privatna varijabla). Za to postoji i kraća verzija koda u TypeScriptu.

Duža verzija:

private \_productService : ProductService;

constructor(productService : ProductService) {

this.\_productService = productService;

}

Kraća verzija:

constructor(private productService : ProductService) {

}

Sada kada smo uključili servis u komponentu preko konstruktora trebamo dohvatiti podatke i to je najbolje napraviti u **ngOnInit()**.

ngOnInit(): void {

this.products = this.productService.getProducts();

}

# Dohvaćanje podataka preko HTTP-a + Observables

Naš web browser da bi učitao stranicu šalje http request web serveru/servisu i onda on pristupa bazi podataka i daje nam te tražene podatke preko http responsea koji mi onda dohvaćamo i oblikujemo u tip podataka kako bi dalje mogli raditi s njim i prikazati ih korisniku.

## Observables i Reactive Extensions(RxJS)

Podaci odnosno neki eventovi dolaze asinkrono sa back-enda i onda mi predstavljamo preko **RxJS** sekvencu podataka kao **observable sekvencu** tj. observable. Observables upravljaju asinkronim podacima i tretiraju dolazeće eventove kao kolekciju, tj. to je niz u kojem elementi niza dolaze asinkrono tokom vremena.

U kodu možemo naše metode **subscribeat na observable** kako bi dobili obavijesti kad dođu novi podaci, kada više nema podataka ili kada se dogodi greška.

### Observable operatori

Observables nam omogućavaju da manipuliramo eventovima s operatorima. Operatori su metode na observablima koje stvaraju nove observable, transformiraju izvorni observable u nešto drugo. Oni ne obrađuju sve podatke skupa već obrađuju jedan po jedan podatak kad se emitira. Primjeri su: map, filter, take, merge, …

Korisni stranica za pogledat primjere observable operatora: <https://rxmarbles.com/>.

Stvaramo observable operatore sa pipe metodom pa se nekad zovu i **pipeable operatori.**

Varijable u koje je spremljen Observable označavamo s imenom koje slijedi sufiks $. Npr. **source$** Zatim na tom Observableu pozivamo metodu **pipe** u kojoj dodajemo proizvoljan broj pipeova(operatora) i pritom na rezultat pipe metode pozivamo metodu **subscribe** kako bi na bili vidljivi rezultati tog Observablea, inače ih ne možemo vidjeti. Bez subscribea nikakve promjene neće biti vidljive.

import {Observable, range} from 'rxjs';

import {map, filter} from 'rxjs/operators';

const source$: Observable<number> = range(0,10);

source$.pipe(map(x => x\*3),filter(x => x % 2 === 0))

.subscribe(x => console.log(x));

### Promise vs Observable

Promise nam sigurno vraća buduću vrijednost i nije lazy jer vraća resolve ili reject, ali se ne može cancellati. Observable emitira mnogo vrijednosti kroz vrijeme, za svaki element emitira jer subscribea, lazy je, ali se može cancellati tako što se unsubscribeamo, a također podržava map, filter i slične operatore.

## Slanje HTTP zahtjeva

Naš servis šalje zahtjev prema web serveru kako bi prikupio podatke, a to radi preko HTTP servisa preko requesta(GET), a dobije nazad response. Da bi mogli koristiti HTTP servise moramo ga importati kao HttpClientModule u app.module.ts gdje su definirani svi moduli. **Response** najčešće bude u JSON formatu, ali ako metodi get dodamo generički tip u kojem želimo podatke ona će ih automatski mapirati za nas. Međutim neće biti mapirani u taj tip već u **Observable** čiji generički tip je taj koji smo definirali. Observable nam je neophodan jer podatke dobivamo asinkrono, a on radi s asinkrono dobivenim elementima u nizu.

import { HttpClient } from "@angular/common/http";

import { Observable } from "rxjs";

@Injectable({

providedIn: 'root'

})

export class ProductService {

private productUrl = 'api/products/products.json';

constructor(private http : HttpClient) {

}

getProducts(): Observable<IProduct[]> {

return this.http.get<IProduct[]>(this.productUrl);

}

}

## Handleanje grešaka kod dohvaćanja podataka sa servera

Da bi mogli koristiti observable operatore moramo pozvati metodu **pipe** i unutar nje zvati operatore. Operator **tap** koristi nam kako bi provjerili jesu li podaci ispravni, a da pritom ne mijenja izvorne podatke. Operator **catchError** poziva metodu handleError koja na temelju greške koja se dogodila postupa na određeni način.

getProducts(): Observable<IProduct[]> {

return this.http.get<IProduct[]>(this.productUrl)

.pipe(tap(data => console.log('All: ' + JSON.stringify(data))),

catchError(this.handleError));

}

private handleError(err: HttpErrorResponse) {

...

}

Sve dok ne subscribeamo Observable na neku komponentu kod neće raditi.

## Subscribeanje na Observable

Observable je lazy i sve dok ne pozovemo subscribe on neće emitirati podatke. **Subscribe** metoda ima maksimalno 3 argumenta koji su handler funkcije. Prvi argument je funkcija **nextFn** koja obrađuje sljedeću emitiranu vrijednost, a pozvana je za sve vrijednosti koje su emitirane. Drugi argument je funkcija **errorFn** koja je pozvana u slučaju da imamo error. Treći argument je opcionalan i to je funkcija **completeFn** koja je pozvana samo u slučaju kad je observable završen.

Subscribe funkcija vraća nam subscription koji kasnije možemo koristiti kako bi se unsubscribeali.

Sljedeći kod prikazuje nam da je pri inicijaliziranju komponente pozvan servis preko kojeg ćemo dohvatiti podatke tako što se subscribeamo i spremimo ih u propertyje klase.

ngOnInit(): void {

this.productService.getProducts()

.subscribe(products => {

this.products = products;

this.filteredProducts = this.products;

}, error => this.errorMessage = <any>error);

}

Kada nam treba HTTP najbolje ga je definirati unutar servisa koji ima Observable. Zatim taj Observable dajemo klasi kojoj trebaju ti podaci dohvaćeni preko HTTP requesta, a ta klasa se subscribea na Observable i sada može koristiti servis preko kojeg dobiva podatke.

# Navigation and Routing Basics

Navigacija i rutiranje odnose se na to da se možemo kretati između više stranica pomoću određenih akcija. Moramo konfigurirati rute i povezati ih sa ciljanim akcijama da bi se mogli kretati između stranica, a na svakoj stranici je različiti view odnosno izgled.

**Safe navigation operator ?** koristi se kada nismo sigurni da ćemo dobiti podatak pa time izbjegavamo da nam se pojavi greška. Npr. {{product?.productName}}

Međutim taj operator ne može se koristiti kada imamo 2 way binding odnosno banana in the box [(ngModel)]='product?.productName'

Sigurnije je koristiti **\*ngIf** strukturalnu direktivu koja ako nema producta ništa ne prikazuje pa ni nema grešaka.

Rutiranje je bazirano na komponentama, pa kao routing target treba specificirati set komponenti koje će se prikazati.

## Funkcioniranje rutiranja

Rutiranje u Angularu radi na način da mi u index.html imamo definirano sve moguće stranice tj. komponente koje je moguće prikazati pri čemu svaka stranica ima svoju rutu, a da bi se ta stranica prikazala trebamo definirati neku akciju koja se mora ispuniti da bi se ruta aktivirala.

Uobičajeno je da na vrhu stranice postoji header s botunima koji nas klikom na pojedini botun preusmjeri na određenu stranicu. Taj link na botunu naziva se **routerLink** i njime definiramo dodatak na URL potreban za prikazivanje tog viewa. Bitno je da specificiramo path URL skupa s komponentom koju će prikazivati, a ta komponenta uvijek se prikazuje unutar direktive (HTML taga) **<router-outlet></router-outlet>**.

## Konfiguriranje ruta

Da bi mogli napraviti sve rute za različite komponente koje želimo prikazati kao stranice moramo uključiti Angularov **RouterModule** u glavni modul naše aplikacije. Unutar RouterModulea definirane su direktive routerLink i router-outlet koje ćemo koristiti i RouterService, a trebamo definirati naše rute unutar RouterModula kako bi nam radile za cijelu aplikaciju. Za to koristimo funkciju **forRoot([], { useHash: true})**, useHash je true ako koristimo hash styles, a isključeno ako koristimo HTML5 styles.

RouterModule.forRoot([

{ path: 'products', component: ProductListComponent },

{ path: 'products/:id', component: ProductDetailComponent },

{ path: 'welcome', component: WelcomeComponent },

{ path: '', redirectTo: 'welcome', pathMatch: 'full' },

{ path: '\*\*', redirectTo: 'welcome', pathMatch: 'full' }

])

Jako je bitno kojim redoslijedom definiramo rute jer se čitaju od prve prema zadnjoj.

## Vezanje ruta s akcijama i prikazivanje viewova

Da bi povezali rutu s akcijom definiramo property routerLink i specificiramo path naveden u nizu ruta koji nas onda usmjerava na jednu od ruta i učitava komponentu s kojom smo spojili tu rutu.

template: `

<nav class='navbar navbar-expand navbar-light bg-light'>

<a class='navbar-brand'>{{pageTitle}}</a>

<ul class='nav pills'>

<li><a class='nav-link' [routerLink]="['/welcome']">Home</a></li>

<li><a class='nav-link' [routerLink]="['/products']">Product List</a></li>

</ul>

</nav>

<router-outlet><router-outlet>

`

Da bi se ta komponenta na navedenoj ruti prikazala u index.html definiramo direktivu tj. html tag <router-outlet></router-outlet>.

# Navigation and Routing Additional Techniques

## Slanje parametara s rutom

{ path: 'products/:id', component: ProductDetailComponent }

Jedna od dodatnih stvari kod rutiranja je slanje parametara ruti kako bi nas mogla preusmjeriti na točno određeni proizvod u listi proizvoda. Tako u **routerLink** treba u nizu navesti i path i dodatni parametar koji šaljemo.

<td>

<a [routerLink]="['/products', product.productId]">

{{product.productName}}

</a>

</td>

Kako bi pročitali parametre koji su došli s rutom i prikazali baš taj određeni proizvod u komponentu koja se koristi za prikaz treba kao dependency uključiti **ActivatedRoute** servis u konstruktoru. Zatim u konstruktoru pozivamo ili metodu snapshot na tu rutu ili Observable. **Metoda snapshot** poziva se u slučaju kada se iz nekog proizvoda vraćamo na listu proizvoda te treba točno nazvati parametar koji smo poslali inače neće raditi (u našem slučaju id). **Observable** se poziva u slučaju kada iz nekog proizvoda idemo na idući proizvod, a ne na listu.

constructor(private route: ActivatedRoute) {

}

ngOnInit() {

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

. . .

}

Da bi prikazali taj proizvod opet nam treba HTTP request i response.

Znak + koji se nalazi prije pozivanja metode snapshot označava skraćen zapis za pretvorbu iz stringa u number.

## Aktiviranje rute preko koda

Da bi aktivirali rutu preko koda, trebamo uključiti **Router servis** unutar komponente odakle će se preusmjeravati ruta. Taj dependency ( router servis ) treba injectati u konstruktoru. Kad smo ga injectali u konstruktoru možemo definirati metodu za akciju na nekom elementu kojem želimo i preko **metode rute navigate** otići na drugu rutu odnosno na stranici otvoriti drugu komponentu.

onBack() : void {

this.router.navigate(['/products']);

}

## Zaštita ruta sa čuvarima

Čuvari se koriste kada neke rute želimo ograničiti npr. samo za administratore, za korisnike kada treba potvrditi žele li izaći iz edit stranice prije spremanja.

Razlikujemo CanActivate, CanDeactivate, Resolve i CanLoad **Guard Service**.

**CanActivate** koristi se kada treba čuvati navigaciju do rute tj. na neki je definiran način ograničiti.

**CanDeactivate** koristi se kada treba čuvati navigaciju od rute.

**Resolve** se koristi kada trebamo dohvatiti neke podatke prije aktiviranja rute.

**CanLoad** se koristi kada trebamo ograničiti da nema asinkronog rutiranja.

Guarda kreiramo kao i service jer on to i je što znači da kreiramo klasu, **dekorator @Injectable** te importamo sve što nam treba. Klasi implementiramo CanActivate koji smo prethodno importali te joj moramo implementirati metodu canActivate().

CanActivate metoda ima 2 parametra: **next i state**. Next nam daje informacije o trenutnoj ruti, State nam daje informacije o stanju rute.

CanActivate Guard nas treba usmjeriti na novu rutu ako je dobar parametar poslan s rutom, ali u slučaju da nije dobar parametar mora nas vratiti nazad na prethodnu rutu stoga unutar deklaracije Guarda trebamo uključiti i **Router.**

@Injectable({

providedIn: 'root'

})

export class ProductDetailGuard implements CanActivate {

constructor(private router: Router) {

}

canActivate(

next: ActivatedRouteSnapshot,

state: RouterStateSnapshot): Observable<boolean> | Promise<boolean> | boolean {

...

this.router.navigate(['/products']);

return false;

}

return true;

}

}

Guarda koristimo tako što u pathu kojem želimo dodamo **property canActivate** i nabrojimo niz guardova koje želimo uključiti. Tako smo ograničili rutu da najprije prođe sve Guardove pa se tek onda aktivira.

{ path: 'products/:id', canActivate: [ProductDetailGuard], component: ProductDetailComponent },

# Angular moduli

Pravila kojih se treba pridržavat kod modula:

1. Bootstrap niz
   1. Svaka aplikacija mora bootstrapat barem jednu komponentu, root aplikacijsku komponentu.
   2. Bootstrap niz ( niz bootstrapanih komponenti ) trebao bi se koristiti samo u root aplikacijskom modulu, AppModule.
2. Declarations niz
   1. Svaka komponenta, direktiva i pipe koji smo kreirali pripada samo jednom Angular modulu.
   2. Deklariramo samo komponente, direktive i pipeove, a nikako ne servise i klase.
   3. Nikad nemojmo ponovno deklarirati komponente, direktive i pipe ako smo ih već deklarirali u nekom drugom modulu.
   4. Svi koji su deklarirani u istom modulu **po defaultu su private,** vide ih samo oni koji su deklarirani u tom istom modulu. Da bi ih mogli koristiti u drugim modulima moramo ih exportati pa importati.
   5. Angular modul neće moći učitati i prikazati neku direktivu ukoliko ona nije definirana u modulu ili importana kroz neki drugi modul.
3. Export niz
   1. Exportamo bilo koju komponentu, direktivu ili pipe ako ih neka druga komponenta triba.
   2. Možemo re-exportat komponente, najbolji primjer za to je shared module.
   3. Možemo re-exportat iako nismo importali.
   4. Nikad ne exportamo servis, on je dijeljen unutar cijele aplikacije.
4. Import niz
   1. Kada importamo modul sve što smo u njemu exportali postaje dostupno.
   2. Ne importaj viška komponenti, samo one koje nam trebaju.
   3. Ako importamo neki modul nemamo importano ono šta taj modul ima importano. Importi nisu inherited.
5. Providers
   1. Bilo koji servis koji je dodan u niz servisa registriran je u rootu aplikacije.
   2. Servise ne smijemo dodavati u niz servisa u Shared modulu, jer onda imamo više registriranih servisa.