Angular

Maltar, Jurica

Master's thesis / Diplomski rad

2017

Degree Grantor / Ustanova koja je dodijelila akademski / stručni stupanj: **Josip Juraj Strossmayer University of Osijek, Department of Mathematics / Sveučilište Josipa Jurja Strossmayera u Osijeku, Odjel za matematiku**

Permanent link / Trajna poveznica: https://urn.nsk.hr/urn:nbn:hr:126:762763

Rights / Prava: In copyright / Zaštićeno autorskim pravom.

Download date / Datum preuzimanja: **2024-03-29**



Repository / Repozitorij:

Repository of School of Applied Mathematics and Computer Science







Sveučilište J. J. Strossmayera u Osijeku
Odjel za matematiku
Sveučilišni diplomski studij matematike i računarstva

Jurica Maltar

Angular

Diplomski rad

Osijek, 2017.

Sveučilište J. J. Strossmayera u Osijeku
Odjel za matematiku
Sveučilišni diplomski studij matematike i računarstva

Jurica Maltar

Angular

Diplomski rad

Voditelj: izv. prof. dr. sc. Domagoj Matijević

Osijek, 2017.

Sadržaj

1 Uvod **1**

1.1 Osnovna struktura Angular aplikacije 2

2 Komponente **3**

2.1 Osnovna građa komponente 3

2.2 Vezanje podataka 5

2.3 Projekcija sadržaja 7

2.4 Životni vijek komponente 8

3 Direktive **9**

3.1 Strukturalne direktive 9

3.2 Atributne direktive 11

3.3 Direktive rađene po mjeri 12

4 Forme **13**

4.1 Forme temeljene na predlošku 14

4.2 Forme temeljene na modelu 15

5 Filteri **17**

6 Servisi **18**

6.1 Ubrizgavanje ovisnosti 19

7 Reaktivno programiranje **22**

7.1 Observable klasa 23

7.1.1 Observer obrazac 23

7.1.2 Iterator obrazac 23

7.2 RxJS 23

7.3 Subject klasa 25

8 Jednostranična aplikacija **27**

8.1 Usmjerivač 29

9 React **32**

9.1 Komponente 33

9.2 Flux 35

10 Praktični projekt **36**

10.1 MathosIntranet 36

10.2 Ostale tehnologije korištene pri izradi MathosIntranet-a 37

10.2.1 Node.js 37

10.2.2 MySQL 38

10.2.3 Bootstrap 38

10.3 Klijentska aplikacija 39

10.3.1 StudyProgrammesModule 40

Zaključak **45**

Literatura **46**

Sažetak **47**

Summary **48**

Životopis **49**

1 Uvod



Angular je *okvir* (eng. framework) za razvoj aplikacija razvijen 2009. godine od strane Google-a. U prvoj verziji, ovaj se je okvir nazivao AngularJS, a počevši od druge verzije, samo Angular. Sufiks "JS" upućuje da se radi o JavaScript[1] okviru za razvoj klijentskih web aplikacija. Izbacivši "JS" iz imena, Google ukazuje da se uz razvoj web aplikacija, omogućava i razvoj mobilnih te desktop aplikacija, a time Angular postaje univerzalni okvir.

2016. godine pojavljuje se druga verzija Angular okvira. Angular je iz temelja redizajniran na osnovu savjeta iz prakse, ali i tako da bude u koraku s modernim tehnologijama koje su se pojavile nakon prve verzije. Također, redizajn je bio potreban zbog nauma da Angular postane univerzalni okvir.

Angular okvir napisan je programskom jeziku TypeScript, razvijenom od strane Microsofta. Njegova je svrha nadomjestiti nedostatke JavaScript-a, zato se često u literaturi TypeScript naziva *nadskup* JavaScript-a. TypeScript nije jezik kojeg preglednici mogu interpretirati, stoga se TypeScript kod prevodi (eng. compile) u JavaScript kod. TypeScript omogućava *strogo tipiziranje* (eng. strong typing) nad JavaScript kodom, tj. možemo deklarirati tipove podataka. Time je omogućena provjera nad tipovima podataka prilikom prevođenja koda kako ne bi došlo do greške prilikom izvršavanja koda. Uz strogo tipiziranje i otkrivanje grešaka prilikom prevođenja, najvrijednija nadopuna JavaScript kodu jest uvođenje mehanizama objektno-orijentiranog programiranja.

Osnova namjena Angular okvira je razvoj jednostraničnih web aplikacija. Uz razvoj jednostraničnih web aplikacija, moguć je razvoj i višestraničnih web aplikacija koristeći biblioteku Angular Universal. Nadopuna Angular okviru za razvoj mobilnih aplikacija jest okvir Ionic, a taj se okvir također razvija od strane Google-a. Uz Ionic dolaze elementi koji se slažu na stog (eng. stack) i time čine mobilnu aplikaciju, ali i niz sučelja za komunikaciju s komponentama unutar mobilnog uređaja, kao što su kamera, žiroskop i GPS senzor.

Dobra svojstva prve verzije ovog okvira, poput direktiva i ubrizgavanja ovisnosti, naslijeđena su i poboljšana unutar druge verzije. Mnogi su prijašnji mehanizmi odbačeni radi lakšeg snalaženja prilikom rada s Angular okvirom, ali i radi poboljšanja performansi same aplikacije. Oslanjajući se na TypeScript sintaksu, svaki se entitet unutar Angular okvira definira pomoću TypeScript klase čime je omogućeno jednostavnije pisanje i lakše snalaženje unutar koda. Uz to, Angular okvir dolazi s komandnim alatom Angular-CLI koji vodi računa o stvaranju kostura same aplikacije, te stvaranju predložaka svih entiteta koji se nalaze unutar okvira, što ubrzava razvojni postupak.

[1] Javascript je skriptni jezik koji se izvršava unutar preglednika. Uz HTML (Hyper Text Markup Language) i CSS (Cascading Style Sheets), JavaScript je temeljna web tehnologija.

1.1 Osnovna struktura Angular aplikacije

Angular aplikacija sastoji se od mnoštva komponenti koje tvore stablo (poglavlje 2). Kako uočavamo da skup komponenti tvori logičku cjelinu, taj skup grupiramo u *modul* (eng. module). Svaka aplikacija stvorena Angular okvirom mora se sastojati od barem jednog modula, a taj modul treba sadržavati korijensku komponentu. Ostale komponente se ugnježđuju unutar korijenske komponente.

Modul je TypeScript klasa koja grupira skup funkcionalnosti u jednu cjelinu. Funkcionalnosti koje Angular okvir pruža su: komponente, direktive (poglavlje 3), filteri (poglavlje 5), servisi (poglavlje 6) i moduli. Kako bismo naznačili da se radi o modulu, prije njegove deklaracije smještamo dekorator `@Module` iz `@angular/core` biblioteke. Dekoratoru prosljeđujemo objekt koji sadrži informacije o tome koje su komponente, direktive i filteri deklarirani unutar modula, koji su servisi pruženi unutar modula, koji su dodatni moduli uveženi u modul, te, ukoliko se radi o korijenskom modulu, koja je korijenska komponenta modula. Nepisano je pravilo da se klasa koja predstavlja korijenski modul naziva `AppModule`. Taj korijenski modul uvozi `BrowserModule` modul koji deklarira sve ugrađene funkcionalnosti, poput direktiva i filtera, u svoj djelokrug. Tada se te funkcionalnosti, npr. `NgIf` direktiva ili `async` filter, mogu koristiti uz komponente deklarirane unutar modula.

Naredbom `platformBrowserDynamic().bootstrapModule(AppModule)` vršimo postupak *samopokretanja* (eng. bootstrapping) modula i time naznačavamo da će se Angular aplikacija izvršavati unutar preglednika i to tako da uz aplikaciju dolazi i interni Angular prevoditelj koji vodi računa o generiranju dijelova aplikacije, odnosno on prevodi aplikaciju u trenutku kada je ona otvorena unutar preglednika. Zato se takav postupak prevođenja naziva *just-in-time*. Uz just-in-time, postoji i *ahead-of-time* prevođenje gdje se aplikacija unaprijed generira bez posredstva Angular prevoditelja unutar preglednika. Nakon što smo definirali način samopokretanja krećemo u izgradnju aplikacije.

Za izgradnju aplikacije uobičajno je koristiti Webpack - alat koji mnoštvo manjih datoteka veže u nekolicinu većih paketa, a između ostaloga, vodi računa i o tome da se TypeScript kod prevede u JavaScript kod pomoću TypeScript prevoditelja. Također, Webpack vodi računa o unošenju "3rd party" biblioteka o kojima ovisi aplikacija, ali i o tome da smanji veličinu koda uz postupke *umanjivanja* (eng. minify) i *poružnjivanja* (eng. uglify). Angular-CLI interno koristi Webpack, pa koristeći naredbu `ng build` uz dodatne zastavice gradimo aplikaciju ukoliko je ona inicijalizirana pomoću tog alata. Nakon što je kod preveden i paketi izgrađeni, oni se uvoze unutar glavnog HTML dokumenta i time tvore web aplikaciju. Bitno je naglasiti da se unutar oznake `body` HTML dokumenta mora nalaziti oznaka koja odgovara selektoru (potpoglavlje 2.1) korijenske komponente korijenskog modula kako bi se sav prikaz kojeg predstavlja aplikacija mogao pričvrstiti na tu oznaku.

2 Komponente

Osnovna gradivna jedinica Angular aplikacije je komponenta (eng. component). Koncept komponente generalno se pojavljuje u softverskom inženjerstvu kao nezavisna cjelina koda koja samostalno obavlja neku funkcionalnost nudeći i zahtjevajući pritom različita sučelja. Konkretno, u slučaju Angular aplikacije, sučelje neke komponente sastoji se od ulaznih svojstava (eng. input properties) i izlaznih događaja (eng. output events).

Komponente se koriste zbog nekoliko praktičnih i očiglednih razloga. Naime, sama se komponenta unutar Angular aplikacije pojavljuje u obliku TypeScript klase. Unutar te klase enkapsuliramo aplikacijsku logiku naše komponente tako da uočimo koncept koji ta komponenta treba predstavljati te deklariramo i definiramo članove koji se trebaju nalaziti unutar komponente. Budući da je komponenta definirana klasom, više instanca te komponente može se pojaviti u različitim dijelovima aplikacije, a time se postiže ponovna upotrebljivost komponente. Unutar komponente moguće je ukomponirati i druge komponente. Komponente sastavljene od drugih komponenata time tvore stablo, vjerno oponašajući samu strukturu HTML DOM-a (Document Object Model)[2] - svaka komponenta tvori jedan čvor stabla i prikazuje se unutar preglednika.

2.1 Osnovna građa komponente

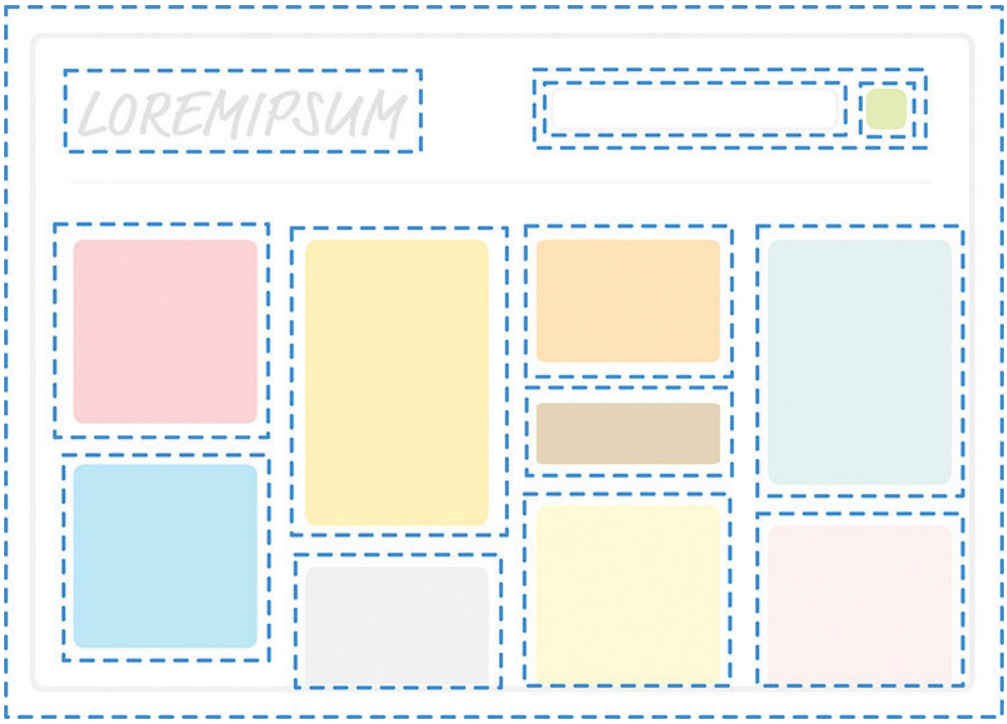
```
1  @Component({
2    selector: 'app-message',
3    templateUrl: './app.component.html'
4  })
5  export class MessageComp {
6
7    public data: string;
8
9    constructor() {
10     this.data = "Hello World";
11   }
12 }
```

Kod 1: Osnovna građa Angular komponente.

Komponentu moramo deklarirati unutar modula u kojem se nalazi, stoga prije ključne riječi `class` koristimo ključnu riječ `export` kako bismo mogli uvesti komponentu u druge di-

[2]DOM (Document Object Model) - sučelje koje omogućava da unutar programskog jezika dohvaćamo i manipuliramo sadržajem nekog strukturiranog dokumenta (konkretno se u slučaju web razvoja radi o upravljanju HTML dokumenta pomoću JavaScript programskog jezika). HTML dokument definira hijerarhijski strukturirane elemente koji se prikazuju unutar preglednika, a takva hijerarhijska struktura vizualno se može predočiti putem n-arnog stabla - svaki je element čvor unutar stabla, a `html` element je korijen stabla čija su djeca `body` i `head` elementi.

jelove aplikacije. Iz `@angular/core` biblioteke uvozimo `@Component` dekorator kojeg postavljamo prije deklaracije klase i prosljeđujemo mu objekt s odgovarajućim svojstvima. Objekt koji prosljeđujemo govori na koji će se način vršiti prikaz HTML koda vezanog uz komponentu. Prije svega, atribut `selector` sadrži string kojim se određuje kojom će oznakom komponenta biti pozvana unutar neke druge komponente. Sve se komponente uvijek pozivaju iz drugih komponenata, osim ukoliko se radi o korijenskoj komponenti nekog modula. U ovom slučaju, komponentu pozivamo oznakom `<app-message></app-message>`. Drugim riječima, komponente se ugnježđuju.



Slika 1: Ilustracija ugnježđivanja komponenti.

Vizualni dio komponente naziva se prikaz (eng. view), a njega definira predložak (eng. template). Predložak se sastoji od HTML koda i koda specifičnog za Angular aplikaciju. Npr., korištenje dvostrukih vitičastih zagrada koje u domeni Angular aplikacije, ali i drugih okvira, omogućavaju interpolaciju stringa (eng. string interpolation). Umjesto atributa `template`, možemo koristiti atribut `templateUrl` čija je vrijednost string relativnog puta do odgovarajućeg koda predloška, a time se postiže bolja preglednost budući da, koristeći atribut template, nije jednostavno pisati HTML i Angular kod unutar JavaScript stringa. Kako koristimo HTML, to je moguće dodati i CSS kod kojim uljepšavamo vizualni prikaz i definiramo stil izgleda, a jednako tako, i ovdje koristimo ili atribut `styles` čija je vrijednost JavaScript polje koje se sastoji od stringova koji predstavljaju CSS kod ili atribut `stylesUrl` čija je vrijednost polje stringova relativnih puteva do odgovarajućih CSS datoteka. Kao i u slučaju predloška, ukoliko "izbacimo" CSS stil izvan komponente u drugu CSS datoteku, dobivamo bolju preglednost koda.

Prilikom prikazivanja komponente, posebno se treba voditi računa o tome na koji će način

komponentna biti prikazana, odnosno enkapsulirana unutar vizualnog izgleda cijele aplikacije. Taj se princip u Angular okviru naziva enkapsulacija prikaza (eng. view encapsulation). Kako smo već spomenuli, prikazu komponente pridružujemo odgovarajući CSS stil kojeg na prijašnje opisani način uvodimo unutar komponente. Enkapsulacija vodi računa o tome kako će se CSS stil prihvaćati na HTML kod komponente. Atribut `encapsulation`, unutar objekta kojeg prosljeđujemo dekoratoru, može imati vrijednosti: `ViewEncapsulation.Emulated`, `ViewEncapsulation.None` ili `ViewEncapsulation.Native`.

Pomoću `ViewEncapsulation.Emulated` načina, CSS kod se primjenjuje jedino na dio HTML-a koji pripada komponenti i to tako da pomoću unutarnjih mehanizama unutar Angular okvira budu pridruženi jedinstveni atributi HTML elementu i CSS selektoru tog elementa. Prilikom `ViewEncapsulation.None` načina, stil definiran unutar neke komponente primjenjuje se globalno, a prilikom `ViewEncapsulation.Native`, sav stil koji bi se inače trebao primjeniti na HTML elemente postaje nedostupan unutar prikaza komponente i jedino stil koji je definiran uz samu komponentu biva primjenjen.

2.2 Vezanje podataka

Jedan od osnovnih principa koji se veže uz samu komponentu unutar Angular okvira je vezanje podataka (eng. data binding). Kako je rečeno, komponente se definiraju TypeScript klasama. U općoj teoriji računarstva, preciznije objektno-orijentiranog programiranja, klasa je logički konstrukt koji definira fizički konstrukt - objekt. Klasa definira svojstva objekta, a ta svojstva mogu biti podaci i metode (specifično se unutar TypeScript-a, ali time JavaScript-a, metode nazivaju funkcije). Pozvavši se na prethodno spomenuto svojstvo komponente - ponovnu upotrebljivost, unutar Angular aplikacije možemo imati mnoštvo instanca iste komponente.

```
1 <app-message></app-message>
2 <app-message></app-message>
3 <app-message></app-message>
```

Kod 2: Višestruko korištenje komponente unutar predloška `AppComp` komponente.

U ovome je slučaju važno naglasiti da je svaka od tri instance `MessageComp` komponente dijete od instance `AppComp`, dok je s druge strane instanca `AppComp` komponente njihov roditelj. To odgovara prethodno napisanoj tvrdnji da komponente čine stablo. Ovakav primjer u prvu ruku nije zanimljiv jer smo dobili prikaz s tri identične poruke, ali time uvodimo princip vezanja podataka kako bi komponenta postala što je moguće generaliziranija. Naime, komponentu možemo promatrati kao kontejner koji će, na određeni način definiran unutar njenog djelokruga (eng. scope), prikazati podatke i omogućiti njihovu manipulaciju. Zato postoje dvije vrste vezanja podataka - vezanje ulaznih svojstava i vezanje izlaznih događaja koji čine sučelje komponente. Budući da se ovaj princip naziva vezanje podataka, potrebni su podaci. Stoga, unutar roditeljske komponente `AppComp` deklarirajmo i inicijalizirajmo

polje poruka.

```
1   messages: string[] = [
2     "Hello World",
3     "Have a nice day",
4     "Nice to see you"
5   ];
```

Kod 3: Polje poruka.

Taj je podatak vidljiv unutar djelokruga same `AppComp` komponente. S njime možemo manipulirati unutar klase, ali i unutar prikaza komponente. Ukoliko predložak `AppComp` komponente nadogradimo retkom `{{ messages | json }}` , tada će unutar preglednika biti prikazano to polje poruka.

Ovaj redak predloška sadrži specifični Angular kod - dvostruke vitičaste zagrade označavaju da se radi o interpolaciji stringa, a vertikalna linija, pomoću mehanizma filtera (poglavlje 5), formatira polje stringova kako bi se to polje pretvorilo u string. Interpolacijom stringa omogućen je prikaz podataka koji se nalaze u djelokrugu komponente. Kako je vidljivo, unutar koda komponente smo definirali podatak `messages`, a ovakvim principom moguće ga je prikazati unutar preglednika. Još uvijek nismo došli do postupka vezanja podataka, ali trenutno imamo podatke koji će biti vezani u druge komponente.

```
1 <app-message
2     *ngFor="let message of messages"
3     [innerMessage]="message">
4 </app-message>
```

Kod 4: Vezanje podataka od roditelja prema djeci.

```
1 @Input('innerMessage') message: string;
```

Kod 5: `@Input` dekorator unutar `MessageComp` komponente.

Pomoću koda `[innerMessage] = "message"` dolazi do vezanja podataka, točnije do vezanja ulaznog svojstva `message` koje je definirano unutar djelokruga `MessageComp` komponente. Koristeći dekorator `@Input('innerMessage')`, naznačeno je da će unutar komponente `MessageComp` postojati podatak nazvan `message` čiji je nadimak `innerMessage`, i atributu `innerMessage` obavijenom uglatim zagradama unutar oznake `MessageComp` komponente koja se nalazi unutar predloška roditeljske komponente pridružit ćemo varijablu `message` koja je trenutno lokalna varijabla predloška[3] komponente `AppComp`, a nastaje iteriranjem polja `messages` koristeći direktivu `NgFor` (poglavlje 3). Trenutno je važno napomenuti da `NgFor` omogućava instanciranje više komponenata istoga tipa iterirajući kroz polje. Postupkom iteriranja instancirane su tri komponente `MessageComp`, a svakoj od instanci smo proslijedili, koristeći mehanizam vezanja podataka, string koji sadrži poruku.

[3]Lokalna varijabla predloška je svaka varijabla koja nastaje unutar predloška i to tako da unutar oznake nekog HTML elementa dodamo oznaku # i definiramo ime varijable.

Uz to što podatke možemo vezati od roditeljskih prema komponentama djece, možemo i u suprotnom smjeru. Time je postignuto da roditeljska komponenta sluša, odnosno reagira na događaj koji je poslan iz nekog od dijeteta. Dijete putem `EventEmitter` objekta šalje podatak, a roditelju je vidljiv taj podatak koji se dalje prosljeđuje u njegov djelokrug.

```
1 @Output('outerMessage') outputEvent = new EventEmitter<string>();
2
3 onClick() {
4     this.outputEvent.emit("Thank you");
5 }
```

Kod 6: `@Output` dekorator unutar `MessageComp` komponente.

```
1 <app-message *ngFor="let message of messages"
2              [innerMessage]="message"
3              (outerMessage)="print($event)">
4 </app-message>
```

Kod 7: Vezanje podataka od djeteta prema roditelju.

Oba ova postupka su tzv. jednosmjerni načini za vezanje podataka. Uz to, postoji i mehanizam za dvosmjerno vezanje podataka koji je bio jedini način u prvoj verziji Angular okvira. Počevši od druge verzije, odlučeno je da će vezanje podataka pretpostavljeno biti jednosmjerno radi efikasnosti. Istovremeno koristimo uglate i obične zagrade kako bismo postigli dvosmjerno vezanje podataka. Ipak, takav način vezanja podataka nije uobičajan u današnje vrijeme, upravo radi efikasnosti.

2.3 Projekcija sadržaja

Kako bi komponenta postala još skalabilnija i fleksibilnija, uveden je mehanizam projekcije sadržaja. Tim mehanizmom prosljeđujemo dodatni HTML kod s ukomponiranim Angular kodom unutar komponente. Unutar predloška komponente u kojoj želimo projicirati sadržaj, dodajemo oznaku `<ng-content></ng-content>`. Oznaci koja instancira komponentu prosljeđujemo HTML kod koji će biti prikazan na mjestu gdje se ta oznaka nalazi. Moguće je smjestiti i više oznaka koje predstavljaju neki sadržaj, i tada je potrebno koristiti atribut `select` unutar `<ng-content>` oznake.

```
1 <ng-content select="h1"></ng-content>
2 <ng-content select="ul"></ng-content>
```

Kod 8: Korištenje višestruke projekcije sadržaja.

Unutar predloška komponente, tj. u oznaci gdje se ta komponenta poziva, neovisno o redoslijedu HTML elemenata `h1` i `ul`, prosljeđeni elementi se prikazuju redoslijedom unutar predloška komponente.

2.4 Životni vijek komponente

Svaka komponenta, počevši od nastanka, tj. instanciranja, pa sve do uništenja, prolazi kroz različite faze koje opisuju njeno stanje. U ovisnosti o stanju, moguće je izvršiti sljedeće funkcije:

- `ngOnChanges()`
- `ngOnInit()`
- `ngDoCheck()`
- `ngAfterContentInit()`
- `ngAfterContentChecked()`
- `ngAfterViewInit()`
- `ngAfterViewChecked()`
- `ngOnDestroy()`

Angular okvir u modulu `@angular/core` sadrži sučelja koja se mogu implementirati unutar komponente kako bi se pozvala određena od gore navedenih funkcija prilikom nekog trenutka u životnom vijeku komponente. Npr., ukoliko naznačimo da komponenta implementira sučelje `OnInit`, tada unutar klase komponente definiramo metodu `ngOnInit()` koja se pokreće svaki puta kada je komponenta inicijalizirana. Takav je mehanizam koristan za pripremu podataka koji se nalaze u komponenti, ali i prilikom uništavanja komponente. Tri najvažnije metode koje se pokreću u životnom vijeku komponente su `ngOnChanges()` - pokreće se svaki puta prilikom promjene ulaznih svojstava pri vezanju podataka, `ngOnInit()` - pokreće se nakon prvog `ngOnChanges()` poziva, i `ngOnDestroy()` - pokreće se prilikom uništavanja komponente, odnosno njenog uklananja iz prikaza neke roditeljske komponente.

Koristeći paradigmu reaktivnog programiranja (poglavlje 7) kojom na funkcionalan način oblikujemo model aplikacije, uobičajno je koristiti klasu `Observable` iz biblioteke RxJS. Tom klasom omogućeno je aktivno praćenje promjene nekog podataka i to tako da se, npr. prilikom inicijalizacije komponente u funkciji `ngOnInit()`, pretplatimo (eng. subscribe) na `Observable` objekt koji će odašiljati podatke, a prilikom promjene o tome obavijestiti komponentu. Kako komponenta aktivno sluša sve promjene podatka jer je pretplaćena na `Observable` instancu, odnosno zauzela je resurs, isti taj resurs radi efikasnosti treba biti odjavljen (eng. unsubscribe), što se može učiniti prilikom poziva `ngOnDestroy()`.

3 Direktive

Direktiva je mehanizam kojim mijenjamo strukturu, ponašanje ili izgled DOM-a. Prethodno opisane komponente (poglavlje 2) su također direktive - očigledno je da komponente utječu na strukturu DOM-a. Preciznije rečeno, komponenta je direktiva s predloškom koji definira prikaz unutar preglednika. Razlika između obične direktive i komponente je u tome da komponenta sadrži, uz ukomponiranu logiku koja utječe na njena svojstva, i predložak kojim je definirano što će unutar preglednika biti prikazano.

U tehničkom pogledu, direktiva je TypeScript klasa koja sadrži članove kojima utječemo na prethodno navedena svojstva DOM-a - strukturu, izgled i ponašanje. Jednako kao i komponenta, direktiva ima svoj selektor. Direktivu koristimo tako da ju "pričvrstimo" na oznaku HTML elementa kojim želimo manipulirati, kao što je i učinjeno prilikom opisivanja vezanja podataka kada smo koristili `NgFor` direktivu. Budući da je direktiva definirana klasom, ona mora imati svoju deklaraciju i definiciju, a kako bismo označili da se radi o direktivi, koristimo `@Directive` dekorator iz `@angular/core` biblioteke. Direktiva se uobičajno deklarira s PascalCase notacijom, dok je kod selektora direktive uobičajno uzeti camelCase notaciju. Budući da se direktive koriste na taj način da ih se pričvrsti na HTML elemente i time se manipulira njihovim svojstvima, ishod korištenja se ne bi trebao mijenjati s obzirom na građu elementa - npr. ukoliko pozivamo direktivu koja mijenja boju pozadine nekog elementa, isti ishod bi se trebao dogoditi i za `h1` i za `p` element. Direktive se dijele u tri kategorije:

- Komponente
- Strukturalne direktive (eng. structural directives)
- Atributne direktive (eng. attribute directives)

Za razliku od komponenata koje se uvijek moraju samostalno razviti, sama Angular aplikacija dolazi s mnoštvom predefiniranih i korisnih strukturalnih i atributnih direktiva koje će biti opisane. Uz predefinirane direktive, moguće je razviti i vlastite direktive ovisno o potrebi same aplikacije.

3.1 Strukturalne direktive

Strukturalne direktive mijenjaju strukturu DOM-a. One mogu duplicirati, ukloniti ili premjestiti neki element unutar DOM-a, a te se radnje manifestiraju na sadržaj unutar preglednika, budući da elementi unutar DOM-a impliciraju sadržaj unutar preglednika. Predefinirane strukturalne direktive unutar Angular aplikacije su `NgIf`, `NgFor` i `NgSwitch`.

Pomoću `NgFor` direktive iteriramo kroz JavaScript polje (eng. array). Kako bismo mogli

pristupiti polju, ono se treba nalaziti u djelokrugu komponente unutar čijeg je prikaza pozvana ova direktiva. Sam manevar `NgFor` direktive koristimo kako bismo dobili više instanci nekog HTML predloška (osnovni HTML element ili predložak komponente) jednakih do na vezanje ili interpolaciju podataka. Dodatne lokalne varijable predloška koje se mogu koristiti prilikom iteriranja pomoću `NgFor` direktive su:

- `index` - označava indeks iteracije
- `first` - Booleova varijabla koja označava radi li se o prvom elementu unutar polja
- `last` - Booleova varijabla koja označava radi li se o zadnjem elementu unutar polja
- `even` - Booleova varijabla koja označava je li indeks paran
- `odd` - Booleova varijabla koja označava je li indeks neparan

Dodatni mehanizam koji možemo koristiti prilikom poziva `NgFor` direktive je `trackBy` atribut kako bismo postigli poboljšane performanse. `trackBy` atributu pridružujemo ime atributa koji sadrži jedinstveni podatak (npr. primarni ključ koji se pohranjuje u bazi podataka), a koji se nalazi unutar svakog elementa polja kroz koji iteriramo. Naime, prilikom iteriranja kroz elemente polja, za svaki element stvaramo novi skup elemenata unutar prikaza. Ukoliko se polje kroz koje iteriramo dinamički promijeni (npr. dođe do osvježavanja podataka), `trackBy` mehanizmom provjerava se je li neki od elemenata polja ostao isti s obzirom na vrijednost jedinstvenog atributa - ukoliko jest, on se ne uklanja iz DOM-a. U ovo se možemo uvjeriti ukoliko prilikom korištenja `NgFor` direktive stvaramo niz instanci neke komponente - metoda životnog vijeka `ngOnDestroy()` bit će pozvana jedino ukoliko je element s jedinstvenim atributom uklonjen iz polja.

Druga bitna strukturalna direktiva je `NgIf`. U ovisnosti o Booleovoj varijabli, ova direktiva uklanja element na koji je pričvršćena. Ukoliko je Booleova varijabla istinita, element ostaje sadržan unutar DOM-a, a u suprotnom, biva ukonjen.

```
1  <p *ngIf="true">I am in DOM</p>
2  <p *ngIf="false">I am not in DOM</p>
```

Kod 9: Primjer `NgIf` direktive.

Direktiva `NgIf` svojim ponašanjem nalikuje svojstvu `hidden` sadržanom unutar svakog HTML elementa, ali postoji velika razlika. Naime, ukoliko je uvjet za `hidden` atribut zadovoljen, element će biti sadržan unutar DOM-a, ali neće biti prikazan, već, kako samo ime atributa kaže, skriven - neće biti vidljiv unutar preglednika i neće utjecati na položaj ostalih HTML elemenata, ali je taj element moguće dohvatiti i manipulirati njime.

Posebna se pažnja treba obratiti ukoliko se radi o kardinalnim dijelovima unutar Angular aplikacije. Primjera radi, kako će biti objašnjeno, `RouterOutlet` je posebna direktiva koja

vodi računa o usmjerivaču (eng. router) – pomoću usmjerivača, moguće je vršiti kretanje unutar Angular aplikacije. Ukoliko u modul u kojem se vrši kretanje uvezemo `RouterModule` modul, potrebno je uključljučiti i `RouterOutlet` direktivu, odnosno, instancirati ju unutar predloška pomoću `router-outlet` selektora. Pretpostavimo da logika neke aplikacije ne dozvoljava prikaz sadržaja kojim se kreće putem usmjerivača ukoliko korisnik nije autoriziran. Tada u ovisnosti o statusu autorizacije korisnika možemo ukloniti sav zaštićeni sadržaj, ali tako da koristimo atribut hidden, a ne `NgIf` direktivu. Ukoliko bismo koristili `NgIf` direktivu koja uvjetno uklanja `RouterOutlet` direktivu, tada ta komponenta u ovisnosti o uvjetu uopće ne mora biti sadržana unutar DOM-a i dolazi do greške u izvršavanju koda.

Pomoću `NgSwitch` direktive, jednako kao i kod `NgIf`, dinamički uklanjamo ili dodajemo element na koji je direktiva pričvršćena. Kod konvencionalnih programskih jezika, `switch` tvrdnjom granamo slučajeve izvršavanja programa s obzirom na zadovoljenost uvjeta. Jednako tako unutar Angular okvira, pomoću `NgSwitch` direktive, dodajemo ili uklanjamo elemente u ovisnosti o grananju slučajeva. Sama se logika `NgSwitch` direktive može zamijeniti i s `NgIf` direktivom, uobičajno kao i kod drugih programskih jezika, ali u drugu ruku, ukoliko provjeravamo višestrukte uvjete, zapis pomoću `NgSwitch` direktive može biti ekonomičniji u pogledu broja redaka koda.

```
1 <div [ngSwitch]="12">
2     <p *ngSwitchCase="12">12</p>
3     <p *ngSwitchCase="13">13</p>
4     <p *ngSwitchCase="14">14</p>
5 </div>
```

Kod 10: Primjer `NgSwitch` direktive.

3.2 Atributne direktive

Pomoću atributnih direktiva manipuliramo atributima elemenata. Te direktive ne mijenjaju strukturu DOM-a već izgled i ponašanje. Tri ugrađene atributne direktive su `NgStyle`, `NgClass` i `NgNonBindable`.

Pomoću direktive `NgStyle` HTML elementu pridružujemo CSS stil na dinamički način. Samoj `NgStyle` direktivi, koja je pričvršćema na neki element putem `ngStyle` selektora prosljeđujemo objekt koji definira njen CSS stil. Npr., ukoliko želimo da tekst unutar nekog elementa bude crvene boje, direktivu pozivamo kodom `[ngStyle]="color:'red'"`. Vidljivo je da `NgStyle` direktiva prihvaća objekt koji definira stil. Taj se objekt može nalaziti u djelokrugu u kojem je direktiva pričvršćena, odnosno, taj objekt može biti lokalna varijabla predloška ili biti definiran unutar klase komponente u kojoj se poziva. Budući da se radi o objektu, njega na dinamički način možemo mijenjati, za razliku od običnog CSS stila koji je definiran statičkim kodom.

`NgClass` direktivom dodajemo CSS klase elementu na koji je ova direktiva pričvršćena. Postupak je analogan kao i kod `NgStyle` direktive, s tim da `NgClass` direktivi prosljeđujemo objekt čiji su atributi imena CSS klasa, a vrijednosti atributa Booleove varijable. Ukoliko je vrijednost nekog atributa unutar objekta istinita, ime tog atributa manifestirat će se kao CSS klasa elementa.

`NgNonBindable` direktivu dodajemo unutar oznake elementa kojeg ne želimo evaluirati u smislu Angular sintakse. Sav sadržaj unutar tog elementa bit će prikazan kao obični tekst (eng. plain text). Ova je direktiva korisna za web stranice koje su implementirane pomoću Angular okvira i koje podućavaju o istom okviru kako se sam kod ne bi evaluirao.

3.3 Direktive rađene po mjeri

Uz ugrađene direktive, moguće je kreirati i one rađene po mjeri s obzirom na specifičnost prikaza unutar preglednika. Direktive se definiraju pomoću TypeScript klase i unutar aplikacije naznačujemo da se radi o direktivi tako da uz klasu dodamo `@Directive` dekorator. Unutar dekoratora prosljeđujemo objekt koji sadrži atribut `selector` kojim naznačujemo kojim imenom pozivamo direktivu koja će biti pričvršćena na određeni element. Selektor se u ovome slučaju, za razliku od selektora komponente, razlikuje u tome da uz ime koje definira selektor dodajemo i uglate zagrade, tj. `'[myDirective]'`, a ne `'myDirective'`. Time naznačavamo da će direktiva biti primjenjena kao atribut elementa. U slučaju bez zagrada, radi se o direktivi koja se primjenjuje kao HTML element, što je u skladu s time kako se definira selektor za komponentu.

Konstruktor klase koja definira neku direktivu prima element tipa `ElementRef` na koji primjenjujemo direktivu. To je smisleno jer direktiva manipulira samim elementom na koji je pričvršćena. U pogledu direktive, element na koji je direktiva primjenjena naziva se domaćin (eng. host element).

Pomoću dekoratora `@HostListener` i `@HostBinding` pratimo stanje domaćina. Dekorator `@HostListener` iz `@angular/core` biblioteke naznačava događaj na koji želimo reagirati (npr. klik mišem) i to tako da mu proslijedimo ime događaja te definiramo funkciju koja će se pozvati prilikom događaja. Pomoću `@HostBinding` dekoratora, naznačavamo koje svojstvo domaćina želimo nadograditi tako da dekoratoru proslijedimo ime svojstva.

Jednako kao što se podaci mogu vezati unutar komponente, mogu se i unutar direktive kako bi ona postala konfigurabilnija. Već smo vidjeli da se za `NgStyle` i `NgClass` vežu objekti koji definiraju stil, odnosno CSS klase elementa. Jednako tako, uz pomoć `@Input` dekoratora unutar direktive, prosljeđujemo podatke.

4 Forme

Prema [3, p. 145], forme su tiskani dokumenti s područjima za popunjavanje informacija. Taj princip je nasljeđen i unutar HTML dokumenata. Unutar preglednika, forme možemo popunjavati koristeći kontrole forme (eng. form contol), a podaci koji su unešeni unutar forme bivaju prosljeđeni na poslužitelj gdje se obrađuju i pohranjuju. U starijim okvirima za web aplikacije, cijela se web stranica u kojoj su unijeti podaci, odnosno, na kojoj se nalazi forma, šalje na poslužitelj te se, s obzirom na jedinstvene oznake koje su pohranjene unutar forme, podaci parsiraju.

U ovisnosti o tome koje podatke želimo prikupiti i na koji način, unutar HTML elementa `form` slažemo kontrole forme. Pritom možemo voditi računa o točnosti podataka, odnosno validirati podatke, a i dodatno implementirati aplikacijsku logiku koja pomaže pri unošenju podataka (npr. ulazna jedinica za tekst sa *autocomplete* opcijom koja nam preporučuje unos nekog teksta). Kontrole forme su:

- ulazna jedinica za tekst (eng. text input)
- ulazna jedinica za datoteku (eng. file input)
- okvir za označavanje (eng. check box)
- radio gumb (eng. radio button)
- padajući izbornik (eng. drop-down box)
- gumb za podnošenje forme (eng. submit button)

Za razliku od konvencionalnih web stranica gdje se cijela stranica, točnije, cijeli HTML dokument s popunjenim podacima šalje na poslužitelj te parsira, unutar Angular aplikacije stvar je u potpunosti drugačija. Naime, budući da je Angular aplikacija po svojoj arhitekturi jednostranična aplikacija (poglavlje 8, str. 27), gdje se vizualni dio vidljiv na pregledniku dinamički generira putem JavaScript koda, nije moguće poslati HTML dokument koji definira stranicu koja sadrži popunjenu formu. Stoga se unutar Angular aplikacije nude mnoge funkcionalnosti kako bismo mogli raditi s formama. Bitno je naglasiti da, za razliku od uobičajnog postupka gdje se popunjena forma šalje kao potpun HTML dokument, podatke prikupljene unutar forme u ovom slučaju šaljemo enkapsulirane unutar JSON objekta putem HTTP protokola.

Princip rada forme unutar Angular aplikacije vrlo je intuitivan. S obzirom da se forma, tj. njen vizualni dio, nalazi unutar prikaza neke komponente, oslanjamo se na koncept vezanja podataka. Oni podaci koji su unešeni unutar neke kontrole forme lako se mogu vezati (u smislu vezanja podataka) za članove komponente, a iz komponente se dalje šalju

do servisa (poglavlje 6, str. 18) koji služe za komunikaciju s poslužiteljem. Uz vizualni dio forme, koji je dovoljan unutar konvencionalnih web stranica budući da se podaci obrađuju na poslužitelju, logika forme može biti ukomponirana i unutar klase komponente u kojoj se forma nalazi. Unutar Angular aplikacije razlikujemo dvije vrste formi - forme temeljene na predlošku i forme temeljene na modelu. Obje vrste formi imaju istu svrhu - omogućiti prikupljanje podataka od strane korisnika.

4.1 Forme temeljene na predlošku

Forme temeljene na predlošku (eng. template-driven forms), kako samo ime govori, definiraju se unutar predloška neke komponente. Njih je lako izgraditi, a kako bismo to učinili, potrebno je uvesti modul `FormsModule` unutar glavnog modula u kojem gradimo aplikaciju. Unutar tog modula definirane su direktive `NgForm` i `NgModel`.

`NgForm` direktiva automatski će biti pričvršćena na svaki `form` element unutar neke komponente. Tada formi možemo pristupiti koristeći lokalnu varijablu predloška. Pomoću koda `#myForm="ngForm"` unutar oznake forme, lokalnoj varijabli predloška `myForm` pridružujemo vrijednost `ngForm` koja enkapsulira formu i njene unutrašnje elemente, odnosno kontrole forme kreirajući pritom podatak tipa `FormGroup`. `FormGroup` je klasa koja enkapsulira logiku kojom pratimo vrijednosti i valjanost niza `FormControl` instanci. Jednako kao i kod običnih formi unutar HTML dokumenta gdje se forma sastoji od niza kontrola forme, `FormGroup` instanca sadrži objekt `controls` koji se sastoji od niza `FormControl` instanci. Odnosno, jedna ili više `FormControl` instanci su djeca od neke `FormGroup` instance. Sama klasa `FormControl` enkapsulira logiku kojom pratimo vrijednost i valjanost jedne kontrole forme.

Unutar predloška u kojem je smještena forma uz koju smo instancirali formu temeljenu na predlošku, nalazi se više kontrola forme. Svakoj kontroli forme dodajemo direktivu `NgModel` koju pozivamo selektorom `ngModel`. Ta direktiva kreira novu instancu klase `FormControl` koja je tada automatski pridružena roditeljskoj `FormGroup` instanci. `FormControl` veže DOM element koji služi kao kontrola forme, a kako je rečeno, tada je dostupno sučelje koje omogućava pristupanje atributima elementa - bitno je pristupiti vrijednosti elementa kako bismo dalje mogli vršiti propagiranje vrijednosti ili ju validirati. Koristeći uobičajni atribut `name` unutar elementa kontrole forme, naznačavamo da tu kontrolu forme pridružujemo u obliku instance `FormControl` roditeljskoj `FormGroup` instanci, a ime atributa koji pohranjuje tu instancu `FormControl` klase bit će jednako vrijednosti `name` atributa. Ukoliko ne dodamo `name` atribut uz kontrolu forme, dolazi do greške prilikom izvršavanja koda. Nakon što su podaci unešeni, pomoću događaja `ngSubmit`, prilikom podnošenja forme koristeći gumb za podnošenje, `myForm` varijabla se prosljeđuje unutar komponente.

Uz direktivu `NgModel`, postoji i direktiva `NgModelGroup` koja omogućuje jednostavno grupiranje različitih dijelova forme kako bi vrijednosti koje su unešene unutar forme bile

bolje strukturirane. Selektor ove direktive je `ngModelGroup`, a njemu prosljeđujemo string koji definira atribut kojim će se grupa više `FormControl` instanci grupirati u novu `FormGroup` instancu. U pogledu vezanja podataka, podaci se mogu vezati tako da direktivi `ngModel` proslijedimo string koji se nalazi unutar djelokruga komponente.

```
1 <input type="text" [ngModel]="someString" name="lastName">
```

Kod 11: Jednosmjerno vezanje podatka unutar kontrole forme.

To je iznimno korisno ukoliko želimo da inicijalna vrijednost unutar kontrole forme bude različita od praznog stringa. To možemo učiniti i na jednosmjeran način, ali i dvosmjeran. U prvom načinu, budući da se on naziva jednosmjernim, dolazi do vezanja početne vrijednosti podatka iz djelokruga komponente. Ukoliko dodatno modificiramo kontrolu uz koju je vezan podatak, vrijednost se mijenja isključivo unutar forme, dok je podatak unutar komponente ostao nepromijenjen. Ukoliko želimo vezati podatke u oba smjera, odnosno, želimo da se promjena unutar kontrole forme direktno manifestira i na podatak unutar komponente, koristimo dvosmjerno vezanje podataka.

```
1 <input type="text" [(ngModel)]="someString" name="lastName">
```

Kod 12: Dvosmjerno vezanje podatka unutar kontrole forme.

4.2 Forme temeljene na modelu

Forme temeljene na modelu (eng. model-driven forms), koje se nazivaju još i reaktivne forme (eng. reactive forms), grade se unutar klase komponente. Krajnji ishod ovakvog postupka izgradnje jednak je postupku izgradnje forme unutar predloška. Kako bismo izgradili formu unutar komponente koja će ju sadržavati, potrebno je unijeti `ReactiveFormsModule` koji implementira logiku za rad s ovakvim formama. Prije svega, `ReactiveFormsModule` modul pruža direktive `FormGroup` i `FormControlName` kako bismo `form` element unutar predloška mogli asocirati sa instancom unutar komponente. Instanciramo `FormGroup` objekt koji se sastoji od više `FormGroup` ili `FormControl` instanci. Gledajući odnos među instancama kao stablo, listovi stabla moraju biti `FormControl` instance, a korijen mora biti `FormGroup` instanca. Instancu `FormGroup` klase vežemo s formom unutar predloška.

```
1 this.myForm = new FormGroup({
2     credentials: new FormGroup({
3         firstName: new FormControl(''),
4         lastName: new FormControl('')
5     }),
6     otherInfo: new FormGroup({
7         years: new FormControl('')
8     }),
9 });
```

Kod 13: Instanciranje reaktivne forme unutar klase komponente.

```
<form [formGroup]="myForm" (ngSubmit)="onSubmit()">
  <div [formGroup]="myForm.controls['credentials']">
    <input type="text"
        formControlName="firstName"
        placeholder="First Name">
    <br>
    <input type="text"
        formControlName="lastName"
        placeholder="Last Name">
  </div>
  <div [formGroup]="myForm.controls['otherInfo']">
    <input type="number"
        formControlName="years"
        placeholder="Years">
  </div>
  <button type="submit">Ok</button>
</form>
```

Kod 14: Vezanje instance reaktivne forme unutar predloška komponente.

Pomoću direktive `FormGroup`, naznačavamo koju `FormGroup` instancu vežemo uz koji HTML element. Tako je u ovome slučaju `myForm` vezan uz `form` element, a dva `div` elementa unutar forme vežu unutrašnje `FormGroup` instance naziva `credentials` i `otherInfo`. Same kontrole forme asociramo s instancama `FormControl` koje se nalaze unutar unutrašnjih `FormGroup` instanci koristeći `FormControlName` direktivu. Nakon popunjavanja forme, vrijednost forme je objekt strukturiran analogno strukturi objekta `myForm`.

Uobičajno je instacirati `FormGroup` objekt unutar poziva metode `ngOnInit()` što je i očigledno budući da je taj objekt potrebno povezati s direktivama unutar prikaza - u suprotnom, `FormGroup` instanca ne bi bila definirana te bi došlo do greške prilikom izvršavanja koda.

Reaktivne forme je moguće validirati koristeći klasu `Validators` iz `@angular/forms` biblioteke. Uz predefinirane validatore kao što su `required`, `pattern()`, `minLength()`... koje prosljeđujemo unutar polja kao drugi argument `FormControl` konstruktora, vodimo računa o točnosti unosa. Također, uz obične, postoje i *asinkroni* (potpotpoglavlje 10.2.1) validatori koji validiraju unos na asinkroni način - najtipičnija situacija za korištenje asinkronog validatora je provjera podatka na poslužitelju (npr. jedinstvenost e-mail adrese). Polje asinkronih validatora prosljeđuje se kao treći argument unutar konstruktora `FormControl` klase.

5 Filteri

Prethodno je objašnjen mehanizam interpolacije stringa (poglavlje 2, str. 3). Ukoliko dodatno želimo promijeniti string ili podatak koji sadrži string prilikom interpolacije, ali samo u smislu onoga što je vidljivo unutar preglednika i pritom ne promijeniti podatak, koristimo filtere. Ipak, ukoliko je potrebno, moguće je i promijeniti podatak na koji se filter primjenjuje. Počevši od druge verzije Angulara, službeni naziv za filter je cijev (eng. pipe). Iako nije službeni, naziv filter bolje objašnjava ulogu ovog mehanizma: filtriranje sadržaja vidljivog unutar preglednika.

Prilikom interpolacije, naznačavamo da želimo primjeniti filter tako da nakon imena podatka smjestimo vertikalnu liniju i ime filtera. Više filtera možemo primjenjivati odjednom, a svakome od njih možemo prosljeđivati dodatne parametre. Naprimjer, u ugrađenom `date` filteru prosljeđujemo parametar `'shortDate'`, odnosno ovaj filter primjenjujemo nad podatkom tipa `Date` kao: `{{ startDate | date:  'shortDate'}}` , a za početni string `“Fri Jul 28 2017 14:59:33 GMT+0200 (Central European Daylight Time)”`, formatirani ispis je `“7/28/2017”`.

Postoje dvije vrste filtera - filteri bez stanja (eng. stateless) i sa stanjem (eng. stateful). Filteri bez stanja su čiste funkcije - primaju parametre, mijenjaju inicijalnu, te vraćaju modificiranu vrijednost, a filteri sa stanjem trajno mijenjaju vrijednost koja je prosljeđena. U tehničkom smislu, filter je TypeScript klasa kojoj pridružujemo dekorator `@Pipe`, i tom dekoratoru prosljeđujemo objekt s atributom `name` kojim se označava ime filtera. Unutar klase filtera, implementira se sučelje `PipeTransform` koje deklarira funkciju `transform` koja, kako ime govori, transformira podatak na kojem primjenjujemo filter.

Kao i u slučaju direktiva, postoji niz korisnih ugrađenih filtera. Neki, po subjektivnom mišljenju, od najkorisnijih su:

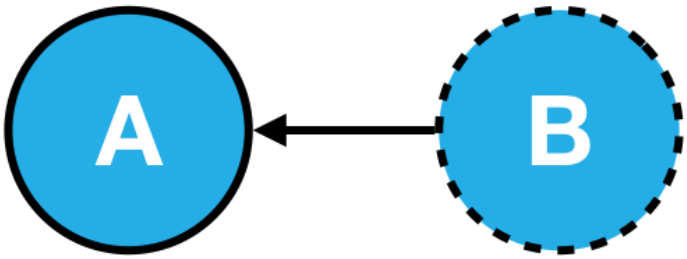
- `json` - pretvara JavaScript objekt u string
- `async` - vodi računa o asinkronim podacima
- `date` - vodi računa o formatu datuma
- `currency` - uz broj pridodaje, odnosno nadodaje znak, odnosno skraćenicu neke valute
- `percent` - za dani broj $x \in [0, 1]$ vraća string u obliku postotka

6 Servisi

Kako sve vrste aplikacija, neovisno o platformi, iz dana u dan postaju sve funkcionalnije i kompleksnije, nužno je pravilno raspodjeliti njihove funkcionalnosti. Jedno od najbitnijih svojstva dobre aplikacije jest ažuriranost podataka. Promjenivši podatak prikazan unutar različitih dijelova aplikacije, nužno je da se promjena, zbog korisničkog iskustva, ali i aplikacijske logike, manifestira u svim dijelovima. Nameće se da taj podatak mora biti jedinstvena instanca neke klase ili jedinstvena vrijednost nekog tipa podataka.

Mehanizmom vezanja podataka možemo osigurati ažuriranost podataka, ali samo među komponentama koje su u relaciji dijete-roditelj. Takvim načinom bismo između svake dvije klase morali vezati podatke, što povlaći mnogo pisanja koda. Same komponente trebaju služiti za vezanje podataka, njihovo prikazivanje te praćenje interakcije korisnika. Iako, u tehničkom smislu, podaci jesu pohranjeni unutar djelokruga komponente, njihov se izvor ne bi trebao nalaziti unutar djelokruga jer se susrećemo sa ovim problemom. Stoga je unutar aplikacije potrebno uvesti "globalne" objekte i vrijednosti koje nazivamo servisi.

Servis je instanca neke klase ili vrijednost nekog tipa podataka dostupna nad svim dijelovima aplikacije. Ideja servisa bazirana je nad konceptom ubrizgavanja ovisnosti (eng. dependency injection). Kažemo da je dio aplikacije A ovisan o dijelu aplikacije B, odnosno B je ovisnost od A, ukoliko je logika koju enkapsulira dio B potrebna za rad dijela A.

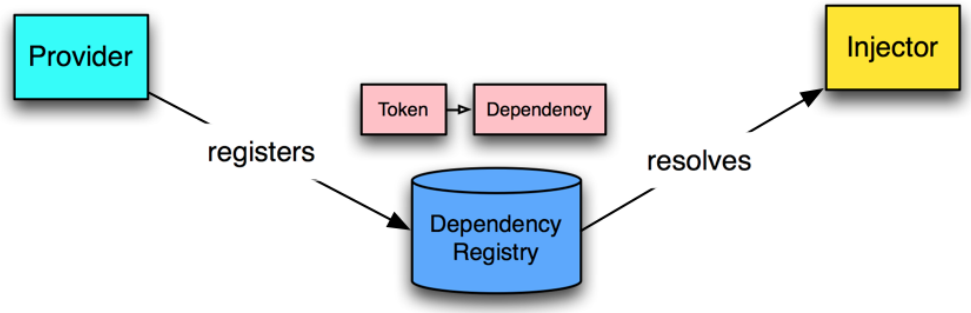


Slika 2: Ilustracija ovisnosti među dijelovima aplikacije.

U pogledu ovisnosti, uobičajan je slučaj da jedna klasa treba drugu klasu za rad. Loša je praksa instancirati objekt klase koja je ovisnost unutar klase koja je ovisna, a ta se praksa naziva *čvrsto spajanje* (eng. tight coupling). U tom slučaju, dolazi do otežanog testiranja i održavanja koda. Temeljna ideja u konceptu ubrizgavanja ovisnosti jest, za razliku od čvrstog spajanja, instancirani objekt koji predstavlja ovisnost proslijediti putem konstruktora klase koja je ovisna. Takva se praksa, suprotna čvrstom spajanju, naziva *labavo spajanje* (eng. loose coupling) i njome postižemo *inverziju kontrole* (eng. inversion of control). Ubrizgavanje ovisnosti jest obrazac za oblikovanje (eng. design pattern) koji slijedi ideju inverzije kontrole.

6.1 Ubrizgavanje ovisnosti

Angular okvir nudi vrlo fleksibilan način za ubrizgavanje ovisnosti. Ideja ubrizgavanja ovisnosti unutar Angulara temelji se na četiri entiteta - token, ubrizgavač (eng. injector), opskrbljivač (eng. provider) i ovisnost.



Slika 3: Dijagram ubrizgavanja ovisnosti unutar Angular okvira.

Niz opskrbljivača je polje koje registrira sve ovisnosti unutar aplikacije ili dijela aplikacije (u smislu nekog podstabla stabla komponenti), a svaki objekt kojim je definiran opskrbljivač sadrži atribut `provide` kojim naznačujemo jedinstveni identifikator ovisnosti, odnosno token te ovisnosti. Drugim riječima opskrbljivač mapira token s odgovarajućom ovisnosti. Ovisnost se može dohvatiti u obliku instance klase, JavaScript objekta, polja ili varijable. Token je jedinstveni identifikator ovisnosti. On može biti string, ime klase (tzv. type token) ili instanca generičke klase `InjectionToken`. Korištenje stringa kao tokena se izbjegava jer dolazi do mogućnosti kolizije.

Opskrbljivač s obzirom na prosljeđeni token definira rezultat koji će biti ubrizgan. Drugi atribut unutar objekta kojim je definiran opskrbljivač može biti `useClass`, `useExisting`, `useValue` ili `useFactory`. `useClass` atribut naznačava da će opskrbljivač s obzirom na prosljeđeni token kreirati jedinstvenu instancu klase, odnosno ta jedinstvena instanca bit će ubrizgana kao ovisnost svim ostalim klasama koje ju zahtjevaju kao ovisnost. Interno se takva instanca pohranjuje unutar registra ovisnosti (eng. dependency registry) kako bismo istu instancu koristili u raznim dijelovima aplikacije.

Uz već registrirani opskrbljivač, moguće je definirati novi opskrbljivač koji koristi vrijednost postojećeg koristeći atribut `useExisting`. Također, opskrbljivač može kreirati i varijablu koristeći `useValue` atribut, ali i pozvati funkciju koristeći atribut `useFactory`. Ukoliko se unutar funkcije koja je vrijednost `useFactory` atributa instancira neki objekt, on se također pohranjuje unutar registra. Funkcija koja je vrijednost `useFactory` atributa može primati parametre koji se moraju nalaziti unutar registra. Polje opskrbljivača vrijednost je atributa providers koji se nalazi unutar objekta kojeg prosljeđujemo `@NgModule`, `@Component` ili `@Directive` dekoratoru. Drugim riječima, moduli, komponente i direktive

mogu imati svoje opskrbljivače.

```
providers: [
    {provide: 'api', useValue: api_url},
    {provide: 'value', useValue: 'init'},
    {provide: MainHttpService, useClass: MainHttpService},
    {provide: 'FactoryDependency', useFactory: (value)=>{
        return new ServiceClass(value)
    }, deps: ['value']}
]
```

Kod 15: Polje opskrbljivača koje unosimo unutar modula.

Mehanizam ubrizgavanja ovisnosti brine o tome da sve ovisnosti sadržane unutar opskrbljivača budu registrirane, ali i da ovisnosti mogu biti ubrizgane tamo gdje su potrebne. Za samo ubrizgavanje ovisnosti brine se ubrizgivač. On s obzirom na prosljeđeni parametar koji predstavlja token ovisnosti, unutar registra ovisnosti putem opskrbljivača dohvaća instancu ili vrijednost ovisnosti i ubrizgava ju. Ubrizgivač se pojavljuje u obliku dekoratora `@Inject`, a smještamo ga prije parametra koji označava ovisnost unutar konstruktora klase koja sadrži ovisnost. Npr., prethodno smo registrirali opskrbljivač s tokenom `'api'` koji opskrbljuje vrijednošću `api_url`. Tu vrijednost želimo ubrizgati unutar `MainHttpService` klase, pa deklariramo parametar unutar konstruktora i dekoriramo ga s `@Inject` dekoratorom:

```
constructor(@Inject('api') private api_url: string) { }
```

Unutar neke komponente koja se brine o korisnikovoj interakciji, odnosno o tome da se podaci prilikom korisnikove interakcije dohvate sa poslužitelja, ubrizgavamo `MainHttpService` instancu:

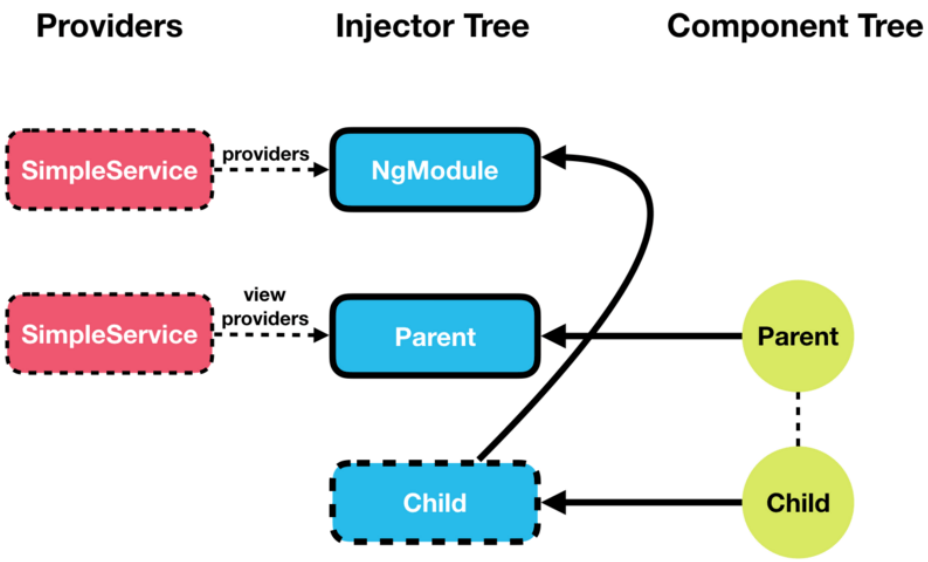
```
constructor(private mainHttpService: MainHttpService) { }
```

Drugi dekorator uz `@Inject` kojim naznačavamo ubrizgavanje ovisnosti jest `@Injectable`. Ovim dekoratorom naznačavamo da želimo ubrizgati ovisnost čiji je token jednak imenu klase, pa Angular interno ubrizgava ovisnost tamo gdje je potrebna. `@Injectable` se koristi kao dekorator uz klase koje nemaju dekoratore `@Component` i `@Directive`, jer i s tim dekoratorima Angular interno ubrizgava ovisnosti ukoliko je token ovisnosti jednak imenu klase koja se instancira.

S obzirom na način registriranja ovisnosti, odnosno s obzirom poredak registriranih opskrbljivača, ovisnosti tvore stablo ovisnosti (eng. dependency tree). Čvorovi stabla ovisnosti predstavljaju ovisnost koja se ubrizgava odgovarajućim ubrizgivačem. Poredak ovisnosti unutar stabla ovisnosti reflektira se na poredak unutar stabla komponenti. Ukoliko registriramo opskrbljivač unutar glavnog modula, ista će ovisnost biti dostupna svim čvorovima stabla komponenti, počevši od korijenske komponente, odnosno korijena stabla, pa sve do listova stabla. Unjevši opskrbljivač nad nekim čvorom komponente, ovisnost će biti dostupna svim komponentama koje su djeca komponente unutar koje je registriran opskrbljivač. Ukoliko

su dvije ovisnosti s istim tokenom dostupne unutar različitih čvorova, a svaki od tih čvorova je predak čvora koji zahtjeva ovisnost, ubrizgana ovisnost bit će ona čiji je oprskrbljivač u najbližem pretku.

Posebna vrsta opskrbljivača jest `viewProvider` koji se odnosi isključivo na komponente. Ukoliko je neki opskrbljivač sadržan nad komponentom kao `viewProvider`, ovisnost koja je definirana opskrbljivačem neće biti dostupna onoj djeci koja su kreirana projekcijom sadržaja. Ipak, takva se djeca još se uvijek nalaze unutar stabla komponenti, pa ukoliko postoji opskrbljivač sa istim tokenom registriran u čvoru koji je predak roditeljske komponente, druga instanca iste ovisnosti postaje dostupna.



Slika 4: Različite instance ovisnosti iste klase s različitim načinima opskrbljivanja.

7 Reaktivno programiranje

Reaktivno programiranje je programska paradigma koja se bazira na kreiranju, transformiranju i reagiranju na tok podataka te propagiranju promjena koje uzrokuje tok podataka. Tok podataka opisuje stanje u kojem se program nalazi, tj. program je definiran kao niz različitih tokova podataka s operacijama koje transformiraju tokove. Više tokova podataka može se povezati tvoreći novi tok.

Uobičajna programska paradigma jest imperativno programiranje, koje u obzir uzima varijable koje predstavljaju stanje programa i izraze koji mijenjaju varijable. Za razliku od varijabli, tok podataka nudi agilniji mehanizam za upravljanje stanjem programa. Zahvaljujući dobrim praksama koje dolaze uz paradigmu reaktivnog programiranja - izbjegavanjem vanjskog stanja (tok podataka ne ovisi o nekoj globalnoj varijabli) i korištenjem čistih funkcija (eng. pure function) koje ne izazivaju nuspojave (eng. side effects) u stanju programa, postižemo konkurentnost. Konkurentnost (eng. concurrency) je istovremeno izvršavanje različitih usklađenih izračuna. Zbog izbjegavanja nuspojava i korištenja čistih funkcija, reaktivno programiranje jest funkcionalno programiranje[4]. Ipak, uključivanje reaktivnog programiranja unutar softverskog projekta nije nužno i na prvi pogled se čini kompliciranim.

Prema [5, p. 4], tok podataka možemo zamišljati kao polje koje je, za razliku od uobičajnog polja čiji su elementi razdvojeni memorijom, razdvojeno vremenom. Sam tok će tijekom vremena odašiljati (eng. emit) više vrijednosti.



Slika 5: Tok podataka je niz podataka razdvojenih vremenom.

Prvobitna implementacija koja je omogućavala uporabu reaktivnog programiranja jest bila Reactive Extensions (skraćeno Rx) bibliteka za .NET okvir napisan u C# programskom jeziku. Implementacija za JavaScript jest RxJS. Prema službenoj web stranici (vidi [10]), RxJS je biblioteka za slaganje asinkronih programa temeljenih na događajima koristeći niz Observable-a. Iako nije nužno koristiti paradigmu reaktivnog programiranja unutar Angular aplikacije, u tehničkom smislu biblioteka RxJS mora biti sadržana unutar projekta koji definira Angular aplikaciju budući da se određene ugrađene klase unutar Angular razvojnog okvira baziraju na RxJS klasama (npr. `EventEmitter`, `HTTP`, `FormGroup`).

[4]Funkcionalno programiranje je programska paradigma u kojoj se program definira kao niz evaluacija matematičkih funkcija pritom izbjegavajući promjenjive podatke i vanjska stanja.

7.1 Observable klasa

Pojam toka podataka jest temeljni koncept reaktivnog programiranja. Direktna implementacija toka podataka jest klasa `Observable`. Observable se može shvatiti kao entitet koji se promatra tijekom vremena i koji odašilje vrijednosti toka podataka. Sam Observable temelji se na dva obrasca oblikovanja: Observer obrazac i Iterator obrazac.

7.1.1 Observer obrazac

Unutar ovog obrasca, dva bitna entiteta su subjekt (eng. subject) - entitet koji pohranjuje niz slušatelja i slušatelj (eng. listener) - entitet koji sluša promjene koje uzrokuje subjekt. Subjekt prolazeći kroz listu slušatelja, nad svakim slušateljem poziva metodu `update()` kojoj prosljeđuje novu vrijednost.

7.1.2 Iterator obrazac

Drugi obrazac za oblikovanje na kojem se temelji Observable jest Iterator. Ovakav obrazac omogućava prolaženje kroz sadržaj nekog podatka. Iterator obrazac deklarira metode `next()` i `hasNext()`.

7.2 RxJS

`Observable` je ključna struktura unutar RxJS biblioteke koja odašilje vrijednosti nekog toka podataka. Te vrijednosti `Observable` odašilje entitetima koji su pretplaćeni na njega. Takvi entiteti u terminima Observable-a se nazivaju Observer-i. Konkretno, RxJS biblioteka nudi generičko sučelje `Observer` koje deklarira metode `next`, `error` i `complete`, dok klasa `Subscription` implementira sučelje `Observer`. Sam Observer nije ništa drugo nego objekt koji definira ove tri metode koje se pozivaju u ovisnosti kako Observable odašilje podatke.

Razlika između Observer obrasca i Observable-a je u tome što Observable ne odašilje vrijednosti ukoliko niti jedan Observer nije pretplaćen na njega. To nije slučaj kod Observer obrasca - činjenično stanje je da ukoliko subjekt nema niti jednog slušatelja, on će trivijalno "proći" kroz prazno polje slušatelja i poslati vrijednosti. S druge strane, Observable jednako kao i Iterator pattern implementira metode kojima se prolazi kroz niz. Koristeći metodu `next`, odašiljemo novu vrijednost toka, jednako kao što u Iterator obrascu koristeći metodu `next` dohvaćamo sljedeću vrijednost unutar niza, a kako bismo naznačili da je tok završen, pozivamo metodu `complete`, analogno tome što provjeravamo postoji li sljedeći element niza unutar Iterator obrasca pozivajući metodu `hasNext` - ukoliko `hasNext` vraća neistinu, ne postoji sljedeća vrijednost koju možemo dohvatiti s `next` metodom.

```
let observable = Rx.Observable.create(function(observer){
    observer.next("First emitted value");
    observer.next("Second emitted value");
    setTimeout(()=>{
        observer.next("Async emitted value");
        observer.complete();
    }, 1000);
 });
```

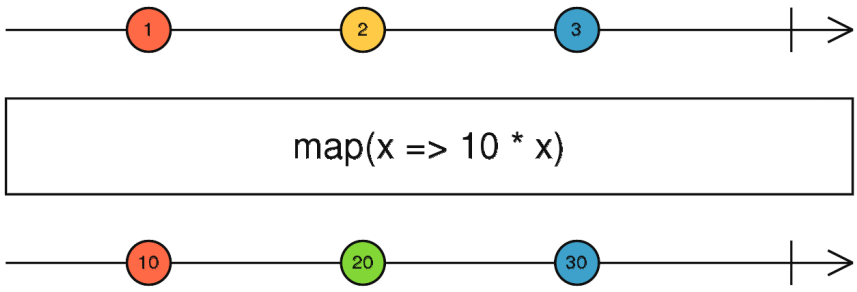
Kod 16: Kreiranje `Observable` instance koristeći metodu `create`.

```
let observer = {
    next: value => console.log(value),
    error: err => console.log(err),
    complete: () => console.log("Stream finished")
}

observable.subscribe(observer);
```

Kod 17: Pretplaćivanje Observera na Observable.

Više je Observable-a moguće nizati, odnosno ulančavati, tvoreći tako lanac Observable-a (eng. Observable chain), a to činimo koristeći operatore koji su implementirani unutar RxJS biblioteke. Operator je čista funkcija kojom je omogućeno funkcionalno programiranje nad tokom podataka. Svaki operator prima ulazni Observable i vraća modificirani izlazni Observable. Poznati operatori koji se pojavljuju unutar svakog funkcionalnog programskog jezika su `map`, `filter` i `reduce`. Uz ove operatore, postoji i pregršt drugih. Kako bismo vizualno predočili utjecaj operatora nad tokom podataka, koristimo se marble-dijagramima.



Slika 6: Marble-dijagram za primjenu operatora `map`.

```
let chained = Rx.Observable
                .interval(1000)
                .filter(value => value % 2 === 0)
                .map(value => value*2);
```

Kod 18: Ulančavanje u kojem koristimo `map` i `filter` operatore te metodu `interval`.

Odašiljujući vrijednosti toka podataka, Observable "gura" (eng. push) te vrijednosti prema svojim pretplatnicima. Zato se kaže da je Observable baziran na *push-protokolu*[5], jednako kao i Promise. Za razliku od Promise-a koji gura jednu vrijednost, Observable može gurati više vrijednosti tijekom vremena. Suprotan push-protokolu je *pull-protokol*[6] kojeg ostvarujemo koristeći uobičajne JavaScript funkcije te njihove specijalizirane oblike za višestruke vrijednosti. Pull i push su protokoli koji određuju na koji način tvorac podataka (eng. data producer) komunicira s potrošačem podataka (eng. data consumer).

Uobičajne podatke unutar JavaScript programskog jezika možemo transformirati u instancu `Observable` klase. Pomoću metode `from` kojoj prosljeđujemo polje vrijednosti dobivamo `Observable` koji odašilje vrijednosti tog polja kao tok podataka. Koristeći metodu `fromEvent` pretvaramo Javascript događaj (eng. event) u `Observable` instancu, a metodom `fromCallback` pretvaramo callback funkciju u `Observable` instancu.

7.3 Subject klasa

Dodatna specijializacija klase `Observable` jest klasa `Subject`. `Subject` klasa istovremeno djeluje kao Observer i Observable. Budući da je Subject Observer, moguće ga je pretplatiti na neki tok podataka, a u drugu ruku, budući da je Subject Observable, on odašilje vrijednosti toka podataka. Ulogu klase `Subject` možemo shvatiti kao posredništvo između nekog toka podataka i pretplate na tok podataka. Pomoću `Subject` instance posredujemo stvarajući višesmjerne (eng. multicasted) Observable-e. Do sada, prilikom pretplate nekog Observera na tok podataka koji je bio ukomponiran u uobičajni, tj. jednosmjerni (eng. unicasted) Observable, svaki je Observer bio pretplaćen sa jednosmjerno izvršavanje Observable-a. Odnosno, Observable se izvršavao posebno za svaki pretplaćeni Observer. Upravo zato, ukoliko jedan tok podataka dijelimo između više entiteta koji su pretplaćeni, koristimo se Subject klasom.

```
1st observer:  1                    observable.component.ts:52
1st observer:  2                    observable.component.ts:52
1st observer:  3                    observable.component.ts:52
2nd observer:  1                    observable.component.ts:56
2nd observer:  2                    observable.component.ts:56
2nd observer:  3                    observable.component.ts:56
```

Slika 7: Tijek odašiljanja toka podataka jednosmjernog `Observable` objekta.

[5]U push-protokolu, tvorac odlučuje kada poslati podatke potrošaču, a potrošač nije svjestan kada će podaci doći.

[6]U pull-protokolu potrošač odlučuje kada će primiti podatak od tvorca koji pritom nije svjestan da šalje podatke potrošaču.

```
let subject = new Rx.Subject();
let shared = Rx.Observable.from([1, 2, 3]).multicast(subject);

shared.subscribe(firstObserver);
shared.subscribe(secondObserver);

shared.connect();
```

Kod 19: Višesmjerno odašiljenje elemenata polja.



Slika 8: Tijek odašiljanja toka podataka višesmjernog `Observable` objekta.

Dodatne specijalizacije `Subject` klase su:

- `AsyncSubject` klasa
- `BehaviorSubject` klasa
- `ReplaySubject` klasa

`AsyncSubject` odašilje zadnju vrijednost od svih vrijednosti unutar toka podataka nakon što izvršavanje unutar `AsyncSubject` instance završi. `BehaviorSubject` se inicijalizira početnom vrijednošću toka i prilikom inicijalizacije pretplate nekog Observera, odmah se prosljeđuje trenutna vrijednost unutar toka. `ReplaySubject` klasa je generaliziraniji oblik `BehaviorSubject` klase čiji konstruktor prima parametre koji određuju koliko prethodnih vrijednosti toka će se pohraniti i u kolikom vremenskom razdoblju (npr. ukoliko naznačimo da se radi o 3 vrijednosti u zadnjih 1000 milisekundi, zadnje 3 vrijednosti odašiljene zadnjih 1000 milisekundi bit će pohranjene unutar spremnika i poslane nekom Observeru koji se pretplati u tom razdoblju).

8 Jednostranična aplikacija

Pomoću mehanizma linkova, omogućeno je kretati se (nestandardnim rječnikom - surfati) s jedne na drugu web stranicu. Kretanje s jedne stranice na drugu obuhvaća: kretanje među različitim web sjedištima[7] , kretanje među različitim stranicama u istome sjedištu i kretanje među istom stranicom (vidi [3, p. 75]). Link se unutar web stranice definira pomoću HTML oznake `a` koja sadrži atribut `href` čija je vrijednost URL[8] sadržaja na koji link vodi. Ta je oznaka akronim za pojam "anchor" (hrv. sidro) koji slikovito opisuje ulogu linka - putem linka se usidravamo na određeni web sadržaj i ostajemo sve dok ne pritisnemo neki drugi link, tj. u slikovitom smislu, podižemo sidro, idemo do drugog sadržaja i usidravamo se. Pošto upišemo web adresu, odnosno URL unutar navigacijske trake ili pritisnemo odgovarajući link koji vodi do nekog sadržaja, web adresa tog sadržaja ostaje unutar navigacijske trake.

Klasični oblik kretanja unutar web sjedišta uzima u obzir više web stranica, odnosno više HTML dokumenata koji će biti poslani unutar preglednika. Time se podrazumijeva da se prilikom svake interakcije koja će dovesti do promjene prikaza unutar preglednika sa poslužitelja dostavi drugi HTML dokument koji predstavlja dio cjeline. Iz tehničke perspektive, prilikom upisa URL-a unutar navigacijske trake ili pritiska linka, poslužitelj šalje odgovarajući HTML dokument. Uobičajno je naziv tog dokumenta jednak dijelu URL-a koji dolazi nakon imena domene i naziva se put (eng. path). Npr. ukoliko je put URL-a "about", zahtijevamo da poslužitelj vrati dokument about.html.



Slika 9: Uobičajne web aplikacije sastoje se od mnoštva HTML dokumenata.

Svaki dio unutar web sjedišta, odnosno svaka stranica biva popunjena podacima od strane poslužitelja. Takvim postupkom poslužitelj troši mnogo računalne moći kako bi podatke

[7]Web sjedište (eng. website) sastoji se od više web stranica koje zajedno predstavljaju smislenu cjelinu koja se naziva web aplikacija.

[8]URL (Uniform Resource Locator) predstavlja web adresu nekog sadržaja.

povezao uz dokument koji definira web stranicu, a istovremeno, preglednik mora dohvatiti veću količinu podataka za svaki od prikaza. Stoga se odnedavno pojavila ideja za drugačijim pristupom kretanja unutar preglednika.

Za razliku od toga da pri svakoj promjeni prikaza, kretajući se unutar web aplikacije, poslužitelj dostavlja odgovarajući dokument, poslužitelj može dostaviti samo jedan dokument prilikom prve posjete na određeno web sjedište. Ukoliko je web aplikacija sastavljena od jednog HTML dokumenta, ona se naziva *jednostranična aplikacija* (eng. single-page application) uz popularan akronim "SPA". Različiti prikazi unutar jednostranične aplikacije ne dostavljaju se putem poslužitelja, već se dinamički stvaraju unutar preglednika u ovisnosti o JavaScript kodu koji je dostavljen uz HTML dokument prilikom dohvaćanja. Nakon što se aplikacija dostavi unutar preglednika, više ne dohvaćamo HTML dokumente koji predstavljaju prikaz, već isključivo podatke koji se vežu uz prikaz, a time je izbjegnuto vezanje sadržaja na poslužitelju i dohvaćanje HTML dokumenta (izuzev glavnog dokumenta koji predstavlja jednostraničnu aplikaciju).



Slika 10: Jednostranična aplikacija sastoji se od jednog HTML dokumenta.

Ipak, za razliku od uobičajne web aplikacije koja se sastoji od niza HTML dokumenata, nedostatak je ovakvog pristupa što inicijalno dohvaćanje aplikacije traje dulje budući da se dohvaća sav kod koji definira sve prikaze unutar web aplikacije. Prilikom razvoja aplikacije

moramo promisliti hoće li ona biti jednostranična ili ne, ali tri najpopularnija okvira za web aplikacije [9] podržavaju razvoj jednostraničnih aplikacija.

Uz Angular okvir dolazi modul `HttpModule` koji enkapsulira logiku za komunikaciju sa poslužiteljem putem HTTP (HyperText Transfer Protocol) protokola. Nakon što se `HttpModule` unese unutar glavnog modula aplikacije, moguće je ubrizgati instancu `Http` klase koja enkapsulira sve metode unutar HTTP protokola (`get`, `post`, `put`, `delete`, `patch`), a svaka od metoda vraća instancu `Observable` klase koja će odašiljati poslužiteljev odgovor na zahtjev. Pomoću instance `Http` klase dohvaćamo i mijenjamo podatke sa poslužitelja. Za kretanje unutar različitih dijelova aplikacije koristimo usmjerivač (eng. router).

8.1 Usmjerivač

Ono što će biti prikazano unutar jednostranične aplikacije su prikazi komponenata koje smještamo na odgovarajući način. Unutar ilustracije koja predstavlja jednostraničnu aplikaciju, vidljivo je da prikaz jedne komponente ostaje fiksan, dok se ostala četiri prikaza drugih komponenti dinamički mijenjaju. Fiksna komponenta koja služi za promjenu sadržaja unutar preglednika naziva se navigacija. Unutar navigacije su smješteni linkovi ili gumbovi koji, bivajući pritisnuti, generiraju novi prikaz. Ova ideja ukomponirana je unutar Angularovog `RouterModule` modula.

Prije svega, potrebno je definirati rute kojima ćemo dinamički stvarati prikaz unutar preglednika, odnosno, kretati se unutar aplikacije. Jedini put kada smo se kretali unutar web sjedišta bio je kada smo inicijalno zatražili HTML dokument Angular aplikacije. O ostalim kretanjima unutar aplikacije brine se usmjerivač.

```
1 const routes: Routes = [
2     {path: '', redirectTo: 'home', pathMatch:'full'},
3     {path: 'home', component: HomeComponent},
4     {path: 'about', component: AboutComponent},
5     {path: 'products', component: ProductsComponent},
6     {path: '**', redirectTo: 'home'}
7 ];
```

Kod 20: Poljem ruta naznačavamo da se želimo kretati kroz prikaze komponenti `HomeComponent`, `AboutComponent` i `ProductsComponent`. Za praznu rutu (`path: ''`) ili bilo koju drugu rutu koja nije definirana unutar polja ruta (`path: '**'`) prikazujemo prikaz `HomeComponent` komponente.

Polje ruta tipa `Routes`, gdje je svaka ruta predstavljena objektom, prosljeđuje se statičkoj metodi `RouterModule.forRoot` koja instancira modul koji sadrži sve direktive, rute i servise potrebne za rad s usmjerivačem. Instancu `RouterModule` klase unosimo unutar glavnog modula.

[9] Uz Angular, React i Vue.js.

```
<nav>
    <ul>
        <li><a [routerLink]="['home']" routerLinkActive="active">Home</a
            ></li>
        <li><a [routerLink]="['about']" routerLinkActive="active">About</
            a></li>
        <li><a [routerLink]="['products']" routerLinkActive="active">
            Products</a></li>
    </ul>
</nav>
<router-outlet></router-outlet>
```

Kod 21: Unutar prikaza u kojem želimo da se prikazi komponenta registriranih unutar ruta prikazuju, ubacujemo `RouterOutlet` direktivu, a na istom prikazu možemo definirati i navigaciju.

Svaki link sadrži odgovarajuću direktivu `RouterLink` čijem selektoru prosljeđujemo polje koje sadrži put do komponente unutar usmjerivača, ali može sadržavati i dodatne parametre. Jednako tako, unutar navigacijske trake preglednika možemo direktno unijeti adresu koja nakon imena domene sadrži put do komponente koja treba biti prikazana. `RouterLinkActive` direktiva odgovarajućem linku koji vodi do prikaza koji je aktivan pridružuje CSS klasu `'active'` kako bismo dodatnim CSS kodom mogli naglasiti na kojem se prikazu nalazimo i time poboljšati korisničko iskustvo.

```
<ul>
  <li *ngFor="let product of products">
    <a [routerLink]="['/product', product.id]">
      {{product.name}}
    </a>
  </li>
</ul>
```

Kod 22: Unutar prikaza komponente `ProductsComponent` izlistavamo sve proizvode.

Svaki element liste unutar prikaza sadrži link do proizvoda. U ovome je kodu naznačeno da želimo putem usmjerivača prikazati prikaz `ProductComponent` komponente, pritom proslijedivši identifikator proizvoda kako bi on mogao biti dohvaćen. Rutu proizvoda trebamo registrirati unutar polja ruta, odnosno, dodati objekt `{path: 'product/:id', component: ProductComponent}`. Dvotočka unutar puta naznačava da prilikom posjete određenoj komponenti prosljeđujemo dodatne parametre (eng. route parameters), a njih unutar komponente koja će biti prikazana dohvaćamo pretplativši se na `Observable` instancu naziva `params` unutar instance `ActivatedRoute` koju ubrizgavamo unutar komponenete.

```
constructor(private activatedRoute: ActivatedRoute) {
    this.sub = this.activatedRoute.params.subscribe(data=>{
        this.id = data.id;
    });
```

```
5 }
```

Kod 23: Ubrizgavanje instance ActivatedRoute klase.

Ukoliko dodatno želimo usmjerivač unutar usmjerivača, koristimo atribut `children` unutar objekta koji predstavlja rutu, a vrijednost tog atributa je novo polje ruta.

```
1 {path: 'product/:id', component: ProductComponent, children: [
2         {path: 'details', component: ProductDetailsComponent},
3         {path: 'order', component: ProductOrderComponent}
4     ]
5 }
```

Kod 24: Dodatno definiranje ugnježđenih ruta unutar `'product'` rute.

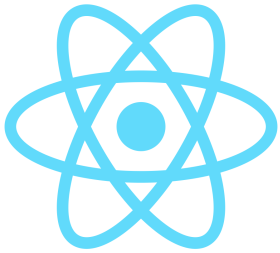
Kako se radi o novom usmjerivaču, ponovno moramo koristiti `RouterOutlet` direktivu unutar prikaza `ProductComponent` komponente. Kako bismo dohvatili prosljeđene parametre (oni su prosljeđeni komponenti `ProductComponent`, ali ne i `ProductDetailsComponent` i `ProductOrderComponent` komponentama), ponovno ubrizgavamo instancu `ActivatedRoute` klase i pretplaćujemo se na parametre roditelja (`this.activatedRoute.parent.params`).

Dodatno možemo restringirati kretanje unutar umjserivača koristeći čuvare (eng. guards). Čuvari su posebne metode koje vraćaju Booleovu varijablu, `Promise` koji razrješava Booleovu varijablu ili `Observable` koji odašilje Booleovu varijablu u ovisnosti o tome je li korisniku dozvoljeno kretati se unutar usmjerivača. Dvije osnovne vrste čuvara su oni koji implementiraju `CanActivate` i `CanDeactivate` sučelja te dopuštaju ili ne dopuštaju da usmjerivač prikaže odnosno ukloni prikaz neke komponente. Sve čuvare za određenu rutu stavljamo unutar polja koje je vrijednost atributa `canActivate` unutar objekta koji definira rutu.

```
1  @Injectable()
2  export class IsAuthorizedGuard implements CanActivate {
3
4    constructor(private authService: AuthService){}
5
6    canActivate(
7      next: ActivatedRouteSnapshot,
8      state: RouterStateSnapshot): Observable<boolean> | Promise<boolean> |
           boolean {
9      return this.authService.isAuthorized();
10   }
11 }
```

Kod 25: Uobičajno je koristiti čuvar koji implementira `CanActivate` sučelje za provjeru je li korisnik ulogiran ili autoriziran za kretanje po određenoj ruti.

9 React



Najpopularniji okvir za razvoj klijentskih web aplikacija jest React. Nastao na temelju iskustva pri razvoju društvenih mreža Facebook i Instagram, React omogućava jednostavan način izgradnje skalabilnih web aplikacija. Inicijalna verzija ovog okvira pojavljuje se 2013. godine, i sama kompanija Facebook vodi računa o njegovom razvoju. U međuvremenu, React postaje univerzalni okvir u kojem je, uz web aplikacije, moguće razvijati i mobilne aplikacije.[10]

Razlog popularnosti React okvira je njegova minimalistička filozofija. Prema službenoj web stranici (vidi [9]), React je okvir za razvoj korisničkih sučelja. Sve ostale funkcionalnosti unutar aplikacije koja je razvijena React okvirom, dobavljaju se putem ostalih "3rd party" biblioteka. Za razvoj React aplikacije potrebne su biblioteke `React` i `ReactDOM`. U početku su ove dvije biblioteke činile jednu, ali su, upravo zbog nauma da React postane univerzalni okvir, razdvojene. `ReactDOM` biblioteka služi za dodavanje sadržaja, odnosno prikaza unutar preglednika, dok se sva ostala aplikacijska logika implementira pomoću `React` biblioteke uz dodatak drugih "3rd party" biblioteka.

React okvirom stvaramo aplikacije koje, koristeći mehanizam virtualnog DOM-a, upravljaju s DOM-om unutar preglednika. Virtualni DOM na optimizirani način kreira i nadograđuje DOM unutar preglednika. On se sastoji od niza React elemenata koji strukturom (ne u smislu sintakse, već u smislu sadržaja) nalikuju HTML elementima, ali sadržavaju dodatna svojstva za efikasnije prikazivanje sadržaja. React elementi su JavaScript objekti. Kako stvorimo neki React element, on će biti dio virtualnog DOM-a, a taj virtualni DOM će se "pretvoriti" u DOM unutar preglednika koristeći metodu `ReactDOM.render` kojoj prosljeđujemo korijenski element, kojeg pričvršćujemo na DOM element unutar preglednika. Svi se ostali elementi ugnježđuju unutar korijenskog elementa, i time tvore stablo.

Kao i u slučaju Angular okvira, React okvirom gradimo jednostranične aplikacije koristeći usmjerivač, a osnovna gradivna jedinica za izgradnju aplikacije jest komponenta. Gotovo svi koncepti vezani uz komponente jednaki su unutar oba okvira. Ovdje svaka sličnost prestaje - razlika između Angular i React okvira je u njihovim filozofijama. Angular okvir nudi čitav niz funkcionalnosti za svaki dio izgradnje aplikacije, počevši od direktiva, filtera, formi... pa do ugrađenih metoda za HTTP pozive, dok je React okvir koncentriran isključivo na komponente, njihovo međudjelovanje te kako na što efikasniji način prikazivati i nadograđivati sadržaj unutar preglednika.

[10] React Native - skup biblioteka za razvoj mobilnih aplikacija koje koriste React okvir.

9.1 Komponente

Budući da se React aplikacija sastoji od niza komponenti koje tvore stablo, razvoj takve aplikacije svodi se na razvoj pojedinih komponenti i njihovog međudjelovanja. U React okviru, komponenta je JavaScript klasa[11] koja nasljeđuje apstraktnu klasu `Component` iz `React` biblioteke i sadrži članove za prikaz podataka unutar preglednika. Budući da definiramo komponentu kao klasu, potrebno ju je, jednako kao i u slučaju Angular aplikacije, instancirati unutar djelokruga aplikacije. No međutim, komponenta se u React aplikaciji ne instancira unutar HTML predloška, već unutar JavaScript koda.

```
1 class Children extends Component {
2     render(){
3         return (<p>{this.props.info}</p>);
4     }
5 }
```

Kod 26: Osnovna građa React komponente.

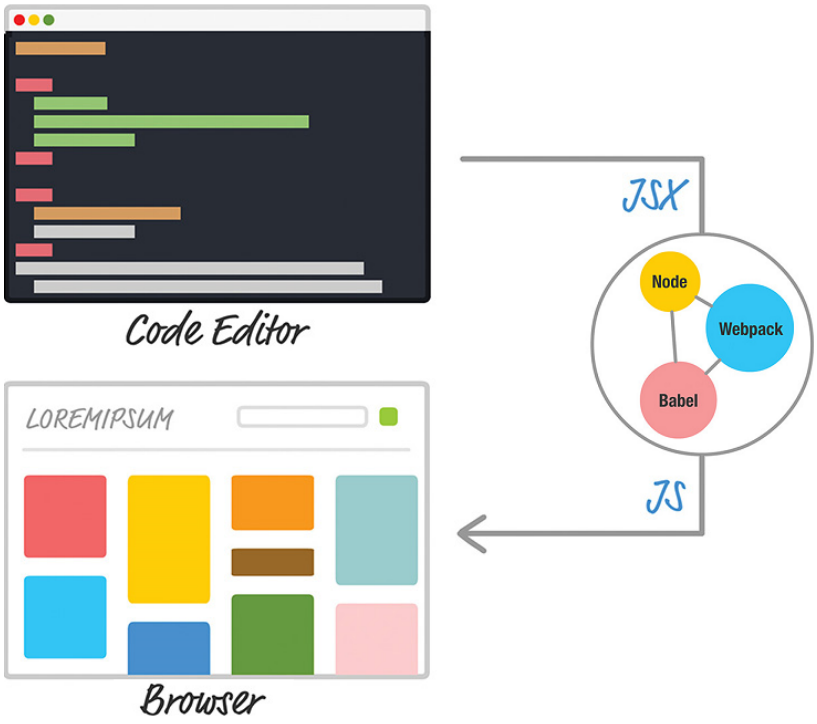
Metodom `render` naznačavamo što će biti prikazano unutar preglednika. Primjetimo da `render` vraća kod koji je izgledom jednak HTML kodu. Ipak, unutar React aplikacije, sve se definira isključivo unutar JavaScript koda uz dodatnu nadogradnju koja se naziva JSX. JSX kod služi za jednostavno slaganje komponenata, a sintaksom nalikuje na HTML kod. Kako se ne radi o standardnom JavaScript kodu kojeg je preglednik u mogućnosti interpretirati, JSX kod potrebno je *prevesti* u uobičajni JavaScript kod pomoću Babel paketa. U drugu ruku, kako se React okvir bazira na najnovijim JavaScript standardima, moguće je da kod koji čini aplikaciju nije podržan od strane nekog preglednika, zato uz postupak prevođenja JSX koda u JavaScript kod, taj kod, koji odgovara ECMAScript 2015 standardima, dodatno prevodimo u neki stariji standard (uobičajno u ECMAScript 5 iz 2009. godine) kako bi se aplikacija mogla izvršavati na što većem broju preglednika.

Komponenta se instancira tako da njeno ime obgrlimo izlomljenim zagradama, a dodatne podatke prosljeđujemo unutar zagrada, jednako kao u slučaju HTML koda. JSX kod je sve ono što se nalazi unutar izlomljenjih zagrada, a sve izvan je standardni JavaScript kod. Ukoliko unutar JSX koda želimo izvršavati JavaScript izraze, koristimo se vitičastim zagradama. Posebnu pozornost prilikom kodiranja JSX-a treba obratiti na rezerviranu riječ `class`. U novom JavaScript standardu, `class` je rezervirana riječ kojom naznačavamo klasu u smislu objektno-orijentiranog programiranja. Ukoliko HTML elementu unutar React komponente želimo prirodati CSS klasu, ne koristimo atribut `class`, već `className`.

```
1 <Parent data={myArray} />
```

Kod 27: Instanciranje komponente.

[11] Klase se unutar JavaScript jezika pojavljuju u verziji ECMAScript 2015.



Slika 11: Tijek prevođenja JSX koda u JavaScript kod.

Kako je već rečeno, neku komponentu možemo više puta instancirati, a ono po čemu će se instance neke komponente razlikovati su svojstva. Svojstva se unutar React komponente nalaze u atributu `props`. Svaki podatak kojeg proslijedimo kao atribut unutar izlomljenih zagrada, bit će sadržan unutar djelokruga komponente. Tako za oznaku `data = {myArray}`, atribut `data` postaje dohvatljiv unutar djelokruga komponente, i dohvaćamo ga izrazom `this.props.data`. Tijekom životnog vijeka komponente (i ovdje postoje *metode životnog vijeka*), svojstva se ne mijenjaju. S obzirom na relacije među komponentama, svojstva se prosljeđuju iz korijenske komponente prema djeci.

```
render() {
    return (
        <div className="parent-div">
            {this.props.data.map((item, i) => {
                return <Children info={item} key={i} />
            })}
        </div>
    );
}
```

Kod 28: Instanciranje komponenata-djece unutar roditeljske komponente.

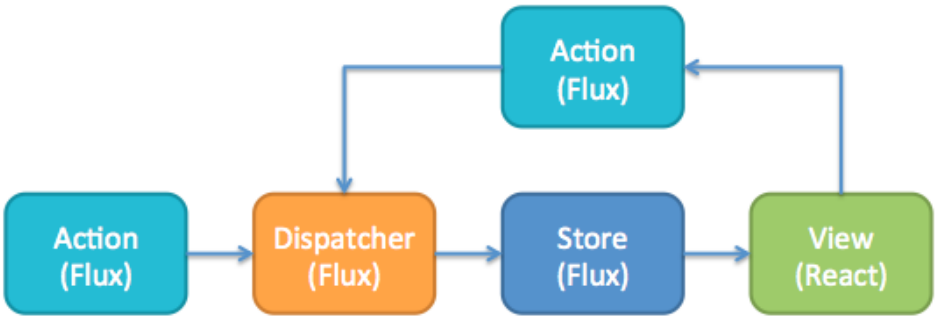
Primjetimo da u ovome kodu postižemo isto što u Angular okviru postižemo s `NgFor` direktivom - iterirajući kroz polje unutar djelokruga roditeljske komponente, instanciramo potomke. Ono što nalikuje `trackBy` mehanizmu unutar `NgFor` direktive je u ovome slučaju svojstvo `key` kojem prosljeđujemo jedinstveni identifikator kojim, kao i u slučaju Angular

aplikacije, optimiziramo stvaranje sadržaja unutar DOM-a. Također, iz polja koje se nalazi unutar djelokruga roditeljske komponente, podatke dalje vežemo prema djeci.

Uz svojstva, koja su nepromijenjiva tijekom životnog vijeka komponente, postoji i *stanje* (eng. state) komponente. Ono se također sastoji od podataka, ali u ovom slučaju, dopušteno ga je mijenjati tijekom životnog vijeka komponente. Uobičajna je praksa da se stanje nalazi u roditeljskoj komponenti koja instancira komponente-djecu i prosljeđuje im dijelove stanja kao svojstva - time lakše nadograđujemo i testiramo kod budući da smo sigurni da jedino roditeljska komponenta uzrokuje promjene podataka. Problemi koji se vežu uz vezanje podataka među članovima stabla jednaki su kao i u slučaju Angular okvira, stoga se sva logika za podatke definira izvan komponenti pomoću *Flux* obrasca oblikovanja, odnosno njegove implementacije `Redux`.

9.2 Flux

Flux je obrazac oblikovanja temeljen na ideji *jednosmjernog protoka podataka* (eng. unidirectional data flow). Implementirajući ovaj obrazac, svi podaci unutar aplikacije jednosmjerno "kruže" unutar djelokruga aplikacije. Tri osnovna entiteta Flux obrasca su: *akcija* (eng. action), *otpremnik* (eng. dispatcher) i *spremnik* (eng. store).



Slika 12: Jednosmjerni protok podataka unutar Flux obrasca.

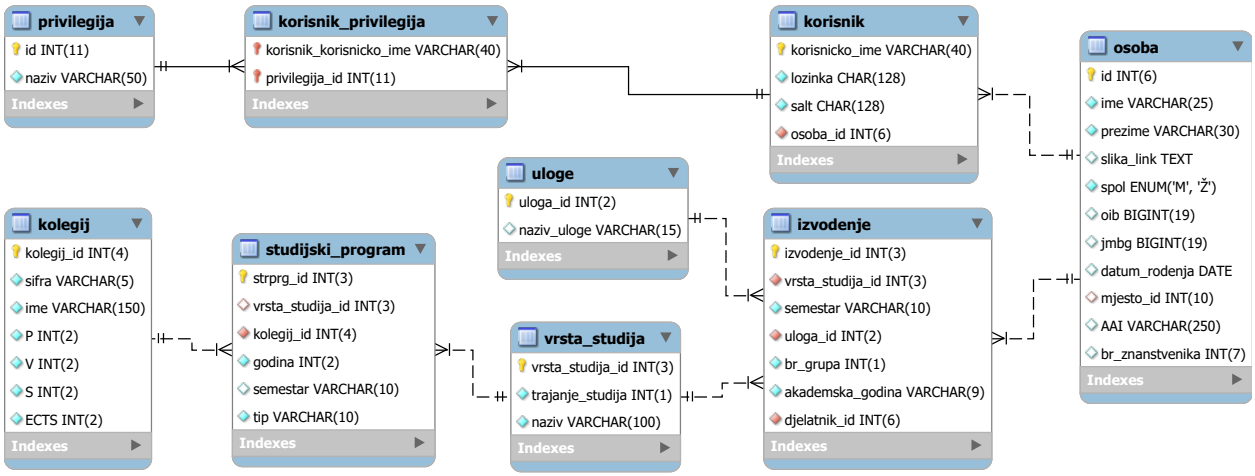
Svaka promjena nad stanjem aplikacije, gdje stanje predstavlja skup podataka, počinje kreiranjem akcije od strane tvorca akcije. Akcija uvijek sadrži atribut `type` čija je vrijednost string kojim se opisuje ime akcije. Spremnik raspoznaje akciju s obzirom na njen tip i na osnovu toga izvršava određene instrukcije. Uz ime, akcija može sadržavati i podatke potrebne za nadogradnju stanja unutar spremnika. Nakon što je akcija stvorena, otpremnik otprema akciju do spremnika. Prilikom stvaranja spremnika, uz njega se asocira jedinstveni otpremnik. Spremnik pohranjuje skup podataka koji predstavljaju stanje aplikacije. Stanje se mijenja isključivo unutar spremnika u ovisnosti o dopremljenim akcijama. Kako se podatak promijeni, spremnik odašilje događaj o promjeni podataka svim komponentama koje su pretplaćene.

10 Praktični projekt

10.1 MathosIntranet

Praktični dio diplomskog rada odnosi se na izradu klijentske web aplikacije za MathosIntranet. MathosIntranet projekt nastao je sa svrhom da studenti diplomskog studija Matematike i računarstva primjene znanja stečena na kolegijima "Klijentsko web programiranje" i "Dizajniranje i modeliranje baza podataka" te steknu iskustvo rada u timu, pritom stvarajući konkretan proizvod. Cilj je MathosIntranet-a obuhvatiti sve segmente poslovanja Odjela za matematiku kako bi se upravljanje poslovnim postupcima[12] olakšalo te integriralo u jednu cjelinu. Zbog područja na koje se primjenjuje, MathosIntranet moguće je uvesti i u ostale odjele i fakultete uz minimalne preinake.

Temelj svake aplikacije jest njen model baze podataka. Budući da koristi relacijsku bazu podataka (potpotpoglavlje 10.2.2), najbolji uvid o tome što MathosIntranet predstavlja dobivamo pregledom tablica unutar baze podataka. Baza podataka sastoji se od 71 tablice koje zajedno čine spremište za informacije o kolegijima, studijskim programima, izvedbenom planu, djelatnicima, korisnicima aplikacije...



Slika 13: Dijagram modela jednog dijela baze podataka za MathosIntranet.

Model baze podataka dalje se preslikava u model unutar poslužitelja i klijenta. Npr., za tablicu `kolegij` stvaramo odgovarajuću TypeScript klasu `Course` čija su imena i tipovi atributa jednaka imenima i tipovima čelija unutar tablice kako postepeno ne bismo morali pretvarati podatke.

MathosIntranet bazira se na *klijentsko-poslužiteljskoj* (eng. client-server) arhitekturi, stoga je uz Angular bitno spomenuti i ostale tehnologije koje se koriste u tom projektu.

[12] Softver koji vodi računa o opravljanju poslovim postupcima općenito se naziva softver za *planiranje resursa poduzeća* (eng. enterprise resource planning), ili skraćeno "ERP".

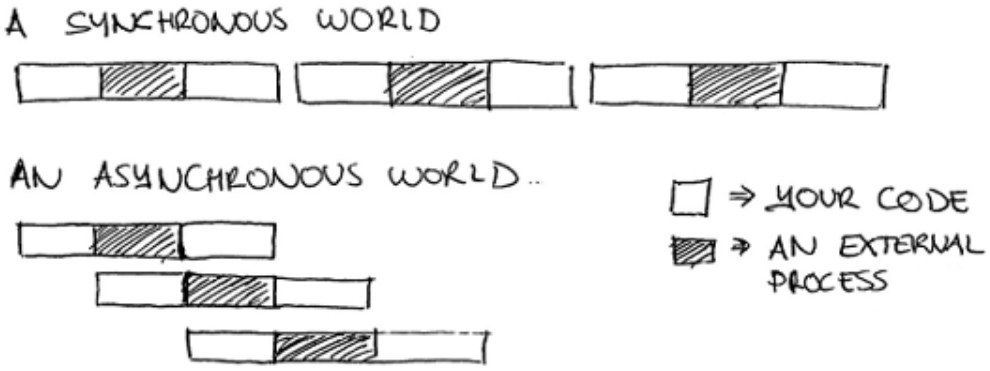
10.2 Ostale tehnologije korištene pri izradi MathosIntranet-a

10.2.1 Node.js



Node.js okruženje je za izvršavanje JavaScript koda izvan preglednika. Prvobitno je JavaScript bio skriptni jezik za funkcionalnosti klijentskih dijelova web aplikacija, tj. jezik web stranica. 2009. godine ostvarena je ideja da JavaScript jezikom stvaramo poslužiteljske aplikacije.

U srcu Node.js-a nalazi se V8 JavaScript interpreter preuzet iz preglednika Chrome. Budući da je V8 najbolji interpreter za JavaScript u smislu performansi, Node.js postiže zapanjujuće rezulate u brzini. S obzirom da se radi o JavaScript-u, tijek izvođenja koda može biti asinkron čime se izbjegavaju zagušenja prilikom izvođenja koda. Npr., za upit nad bazom podataka, poslužitelj asinkrono šalje upit te nastavlja izvršavati ostale zadaće sve dok baza podataka ne odgovori na upit i dostavi odgovarajući rezultat. Zbog činjenice da se kod unutar Node.js okruženja može izvršavati asinkrono, dovoljna je jedna *nit izvršavanja* (eng. thread), za razliku od uobičajnih poslužitelja koji se sastoje od više niti i izvršavaju kod sinkrono.



Slika 14: Ilustracija sinkronog i asinkronog izršavanja koda.

Uz Node.js dolazi komandni alat `npm` kojim upravljamo bibliotekama unutar Node.js projekta. U današnje vrijeme Node.js, uz to što služi kao okruženje za izgradnju poslužiteljskih aplikacija, služi i za razne zadaće prilikom razvoja bilo koje vrste aplikacija. U slučaju razvoja Angular aplikacije, `npm` se brine o tome da se sve biblioteke potrebne za razvoj dohvate unutar repozitorija projekta, dok pomoću Node.js-a možemo izvršavati postupak izgradnje aplikacije (npr., Webpack koji je implementiran u Node.js okruženju). Budući da Node.js uzima u obzir razvoj svih dijelova softverskog projekta, on je tzv. *full-stack* razvojno okruženje. U pogledu razvoja poslužiteljskih aplikacija, najpopularniji okvir baziran na Node.js-u je `Express`.

10.2.2 MySQL



MySQL je *sustav za upravljanje relacijskim bazama podataka* (eng. relational database management system). MySQL je najpopularniji izbor baze podataka za web sjedišta, a najčešće se koristi uz PHP skriptni jezik kojim se razvijaju poslužiteljske aplikacije.

Unutar relacijskog sustava, svaki je skup podataka nekog tipa smješten u jednu tablicu, a svaki je podatak nekog tipa jedan redak tablice koja pohranjuje podatke tog tipa. Pomoću strukturiranog jezika za upite (eng. structured query language), skraćeno "SQL", nad relacijskim bazama podataka, vršimo operacije nad podacima, točnije šaljemo *upite* (eng. query) za određene podatke ili ih *modificiramo* (eng. modification). Također, pomoću strukturiranog jezika za upite stvaramo ili uređujemo tablice, prikaze (eng. view, predefinirani upiti), okidače (eng. trigger, dijelovi koda koji se izvršavaju prilikom određenog događaja)... Relacije među podacima ostvarene su korištenjem ključeva, koji mogu biti: *primarni ključ* (eng. primary key) ili *strani ključ* (eng. foreign key). Primarni je ključ jedinstveni identifikator podatka, dok je strani ključ nekog podatka poveznica na podatak drugog tipa. Time se različite vrste podataka mogu *spojiti* (eng. join) u jedan podatak.

Kako bi poslužiteljski dio aplikacije mogao komunicirati s MySQL sustavom, korištena je Node.js biblioteka `mysql` koja *omotava* (eng. wrap) JavaScript naredbe u naredbe razumljive MySQL-u, a također vodi računa o povezivanju poslužitelja sa MySQL sustavom. Rezultati upita su SQL retci koji se pretvaraju u odgovarajuću JavaScript strukturu kako bi se koristili unutar klijenta i poslužitelja.

10.2.3 Bootstrap

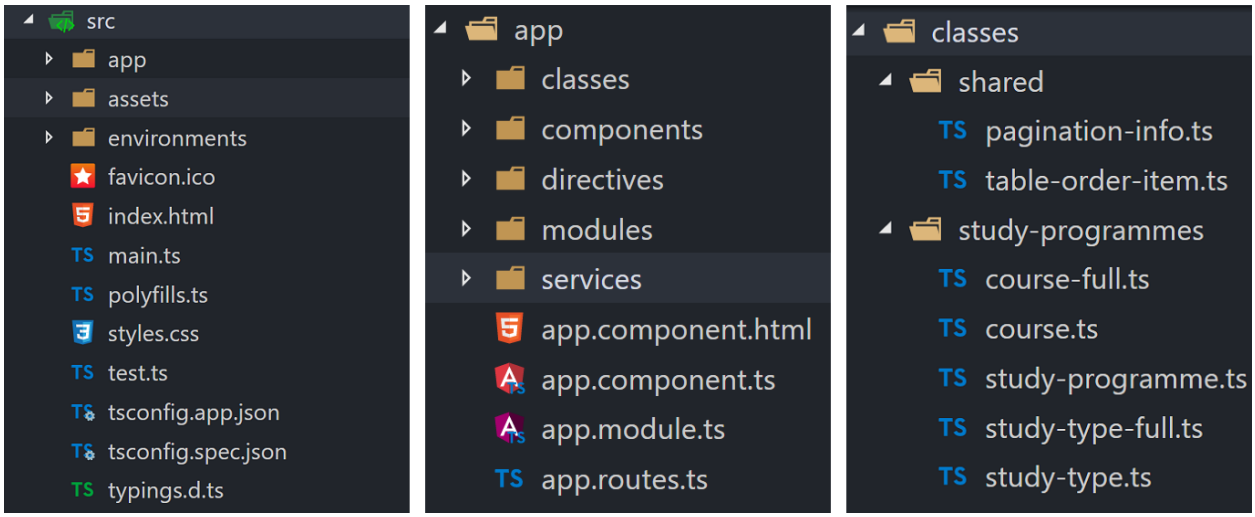


Bootstrap je okvir za razvoj klijentskih web aplikacija. Za razliku od Angular i React okvira koji su koncentrirani na funkcionalnost aplikacije, Bootstrap je koncentriran na vizualni dio aplikacije. Zato se Bootstrap koristi kao dodatak okvirima s mnogo većim skupom funkcionalnosti radi poboljšanja korisničkog sučelja i vizualnog identiteta aplikacije.

Bootstrap se sastoji od komponenata koje nisu funkcionalne poput prethodno spomenutih komponenata, već predstavljaju vizualnu cjelinu i sastoje se od jednog ili više HTML elemenata. Bootstrap primjenjuje responzivni dizajn u kojem se prikaz unutar preglednika prilagođava širini zaslona, a za to je zaslužan njegov mrežni sustav (eng. grid system). Unutar mrežnog sustava, prikaz se sastoji od redaka, a svaki redak od najviše 12 stupaca čiji se poredak mijenja u ovisnosti o širini zaslona. Stoga je prikaz jedne aplikacije prilagođen za uređaje različitih rezolucija.

10.3 Klijentska aplikacija

Klijentska je aplikacija inicijalizirana pomoću Angular-CLI alata. Naredbom `ng new` unutar naredbenog retka stvaramo kostur Angular aplikacije, a time se kreiraju svi dokumenti koji ju tvore. Budući da se radi o osnovnoj strukturi, ostale je dijelove aplikacije potrebno samostalno razviti. Uz to što pomoću prethodno navedene naredbe stvaramo kostur aplikacije, pomoću `ng generate` stvaramo određene entitete unutar aplikacije. Angular aplikacija sastoji se od mnoštva datoteka kojima konfiguriramo razvojnu okolinu te datoteka koje sadrže izvorni kod koji definira ono što aplikacija predstavlja i taj se kod nalazi se unutar mape `src`. Entiteti Angular aplikacije poput klasa, komponentata, direktiva, servisa... nalaze se unutar mape `app`. Unutar svake od mapa entiteta Angular aplikacije nalaze se podmape koje se odnosne na specifični modul. Npr., za klase postoje mape `classes/shared` i `classes/study-programmes` u koje smještamo datoteke s definicijama klasa u ovisnosti o tome u kojem će se modulu koristiti.



Slika 15: Struktura Angular aplikacije.

`AppModule`, definiran unutar datoteke `app.module.ts` koja se nalazi u ishodištu mape s izvornim kodom, glavni je modul aplikacije kojeg samopokrećemo prilikom izgradnje. Unutar tog modula uvozimo ostale module koji tvore našu aplikaciju - `SharedModule` i `StudyProgrammesModule`. `SharedModule` uvozi se u ostale module, a kako se uveze u neki drugi modul, unutar tog su modula deklarirane komponente, direktive i filteri, pruženi servisi i uveženi ostali moduli koje `SharedModule` izvozi.

Budući da koristimo forme te HTTP pozive prema poslužitelju, unutar `SharedModule` potrebno je, kako je spomenuto u prethodnim poglavljima, uvesti module `FormsModule`, `ReactiveFormsModule`, `HttpModule`. Za vizualni dio aplikacije brine se Bootstrap, stoga dodatno uvozimo `NgbModule` - modul s Bootstrap komponentama prilagođen Angular okviru. Za Bootstrap komponente dodatno je potrebna CSS datoteka koja vodi računa o njihovom izgledu i mrežnom sustavu.

Komponente poput `NavigationComponent` i `FooterComponent`, koje predstavljaju navigacijsku traku i podnožje aplikacije, nisu usko vezane uz pojedini modul stoga ih deklariramo unutar glavnog modula kako bismo ih koristili unutar korijenske komponente `AppComponent`. Budući da se radi o jednostraničnoj aplikaciji u kojoj se sadržaj prikaza mijenja dinamički u ovisnosti o kretanju po rutama, koristimo usmjerivač. Selektor direktive `RouterOutlet` smješta se između oznaka komponenata za navigacijsku traku i zaglavlje, kako je i uobičajno za jednostranične aplikacije. Kako je i navedeno prilikom opisa jednostranične aplikacije (poglavlje 8), navigacijska traka i zaglavlje ostaju fiksirani, odnosno njihov se prikaz ne uklanja pa ponovno dodaje prilikom kretanja po rutama.

```
<div id="main" class="d-flex flex-column">
    <app-navbar removeHost></app-navbar>
    <router-outlet></router-outlet>
    <div class="mb-auto"></div>
    <app-footer removeHost></app-footer>
</div>
```

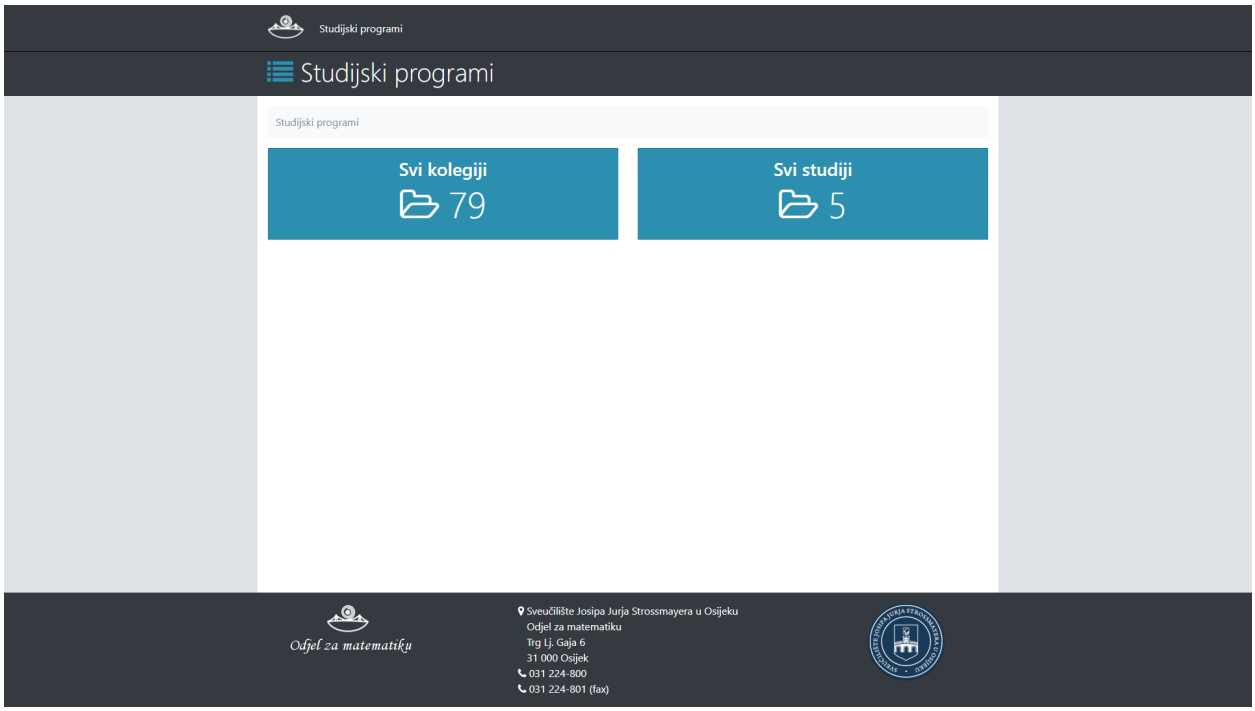
Kod 29: Predložak korijenske komponente `AppComponent`.

```
@NgModule({
  declarations: [
    AppComponent,
    NavbarComponent,
    FooterComponent,
    RemoveHostDirective
  ],
  imports: [
    BrowserModule,
    SharedModule.forRoot(),
    StudyProgrammesModule,
    AppRoutingModule
  ],
  bootstrap: [AppComponent]
})
export class AppModule { }
```

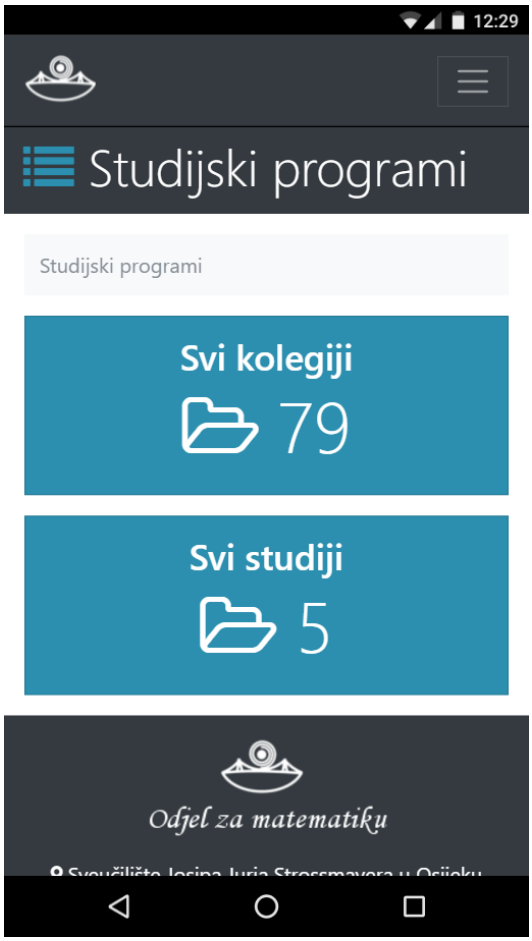
Kod 30: Glavni modul `AppModule`.

10.3.1 StudyProgrammesModule

Unutar `StudyProgrammesModule` modula smještena je sva logika kojom vodimo računa o informacijama vezanima uz studijske programe. Stoga se unutar ovog modula stvaraju, uređuju ili uklanjaju svi kolegiji te svi studiji. Unutar dijagrama baze podataka vidljivo je da je veza između tablica `kolegij` i `vrsta_studija` više-prema-više, odnosno jedan kolegij može biti sadržan u više studija, a jednako tako jedan studij može imati više od jednog kolegija. Budući da su informacije o kolegijima i studijima usko vezane, te su dvije vrste entiteta smještene unutar istoga modula.

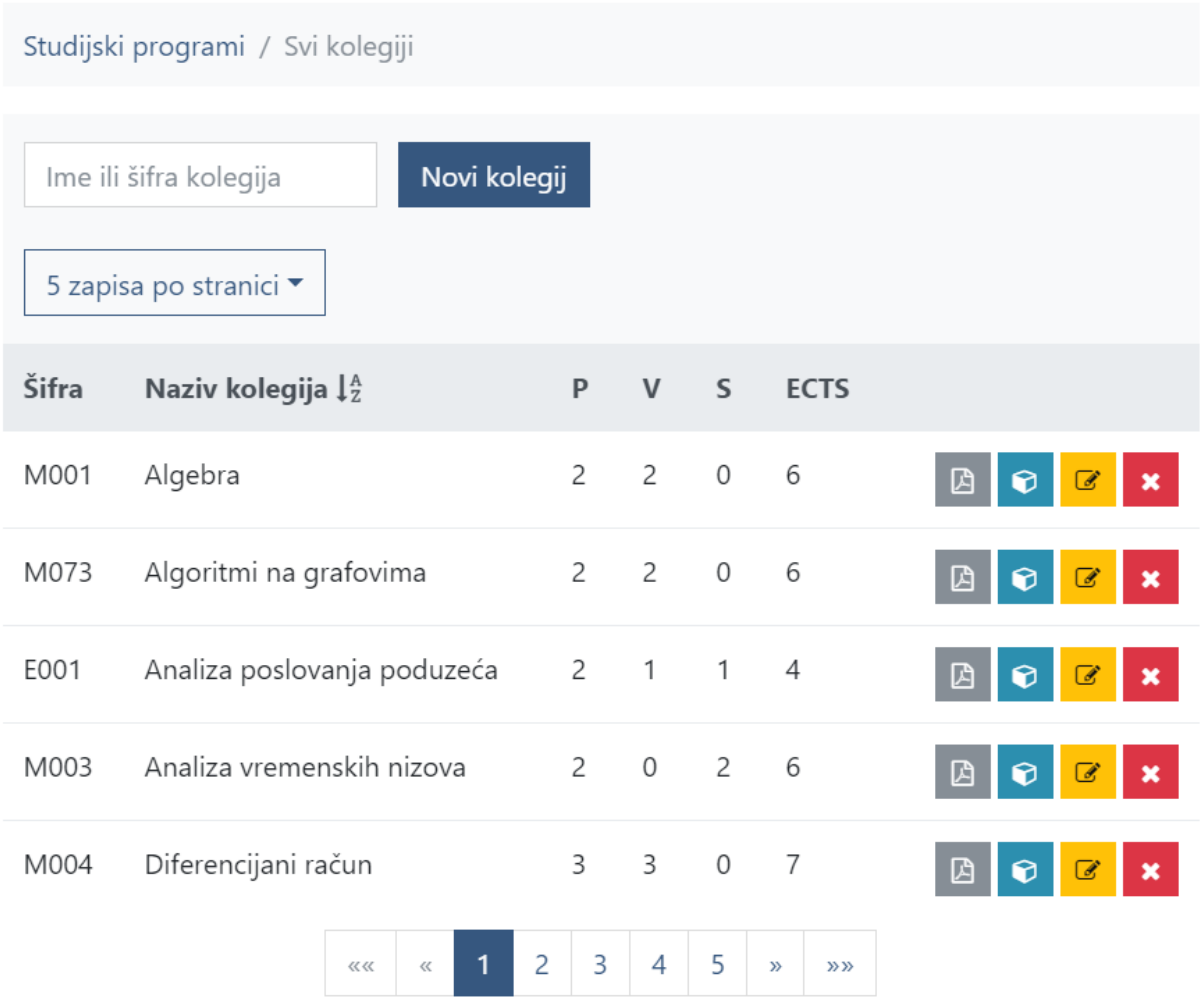


Slika 16: Početni prikaz studijskih programa.



Slika 17: Zahvaljujući mrežnom sustavu Bootstrap okvira, sadržaj prikaza prilagođava se širini zaslona (rezolucija: 412×732 piksela).

Kako pritisnemo tipke koje vode do prikaza svih kolegija ili svih studija, unutar preglednika se prikazuju prikazi odgovarajućih komponenata. Polje svih kolegija i polje svih studija prilikom inicijalizacije aplikacije dohvaća se unutar `StudyProgrammesService` servisa, a zatim sprema u odgovarajuće instance `BehaviorSubject` klasa kako bi podaci bili ažurni. Npr., ukoliko želimo dodati novi kolegij poslavši `post` zahtjev prema poslužitelju i poslužitelj odgovori potvrdno, svim pretplaćenim Observerima istog se trena šalje novo polje kolegija, odnosno, ažuriraju se podaci.

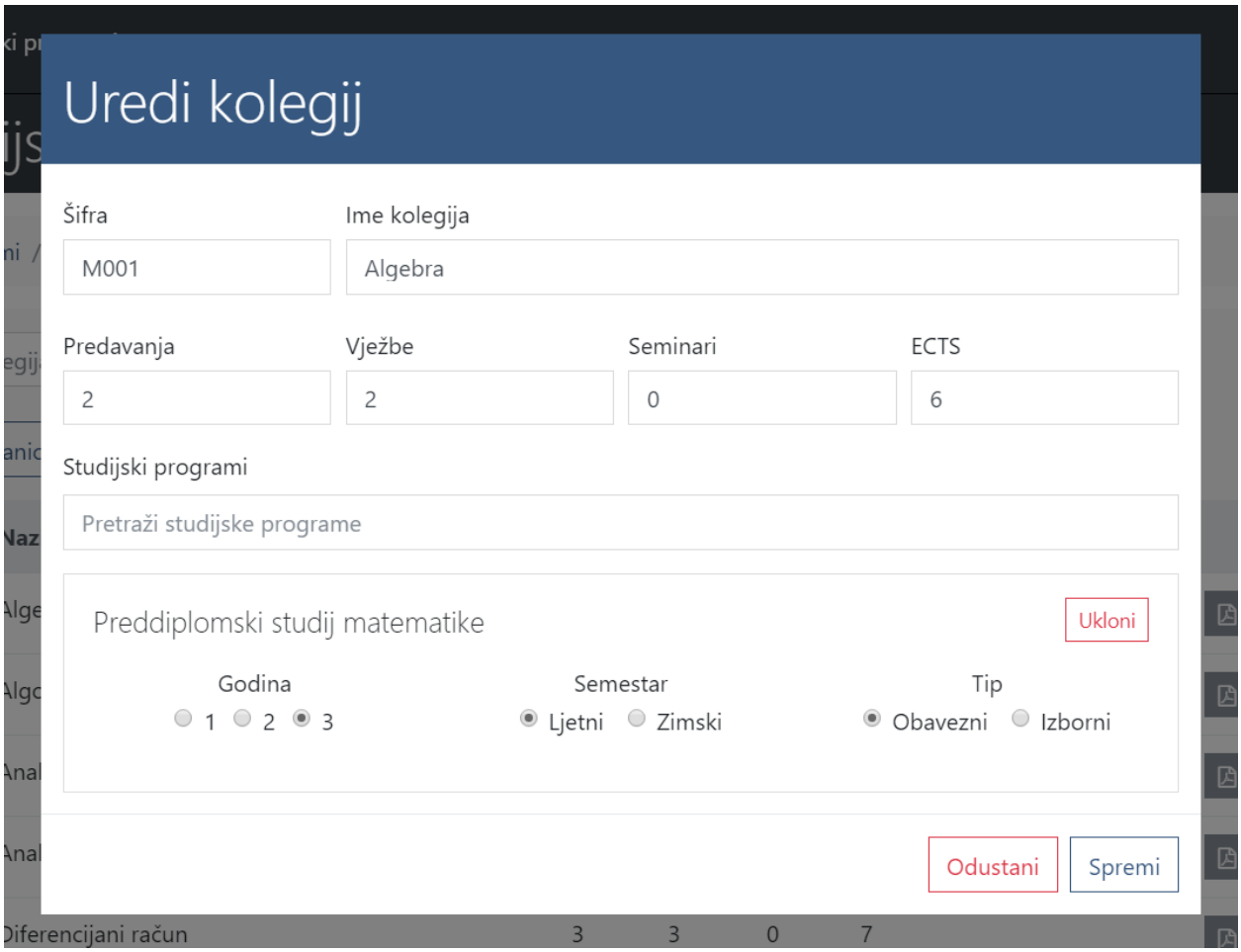


Slika 18: Tablica kolegija. Tipke na kraju svakog retka tablice otvaraju redom: novu karticu s PDF dokumentom o sadržaju kolegija, modal s izlistanim studijima koji sadrže kolegij, modal za uređivanje kolegija te modal s upozorenjem o brisanju kolegija.

Dodatno se tablica kolegija filtrira, sortira te se obilježavaju stranice. U suprotnom, svih 79 kolegija koji se trenutno nalaze u bazi podataka bilo bi izlistano unutar jednog prikaza. Filtriranje, sortiranje i obilježavanje stranica nije nužno za mali broj podataka (npr. ti se mehanizmi ne koriste unutar tablice studija). Budući da se u djelokrugu komponente koja se brine o tablici kolegija nalazi polje koje sadrži *sve* kolegije, na to polje primjenjujemo odgovarajuće operatore kako bismo postigli odgovarajući rezultat.

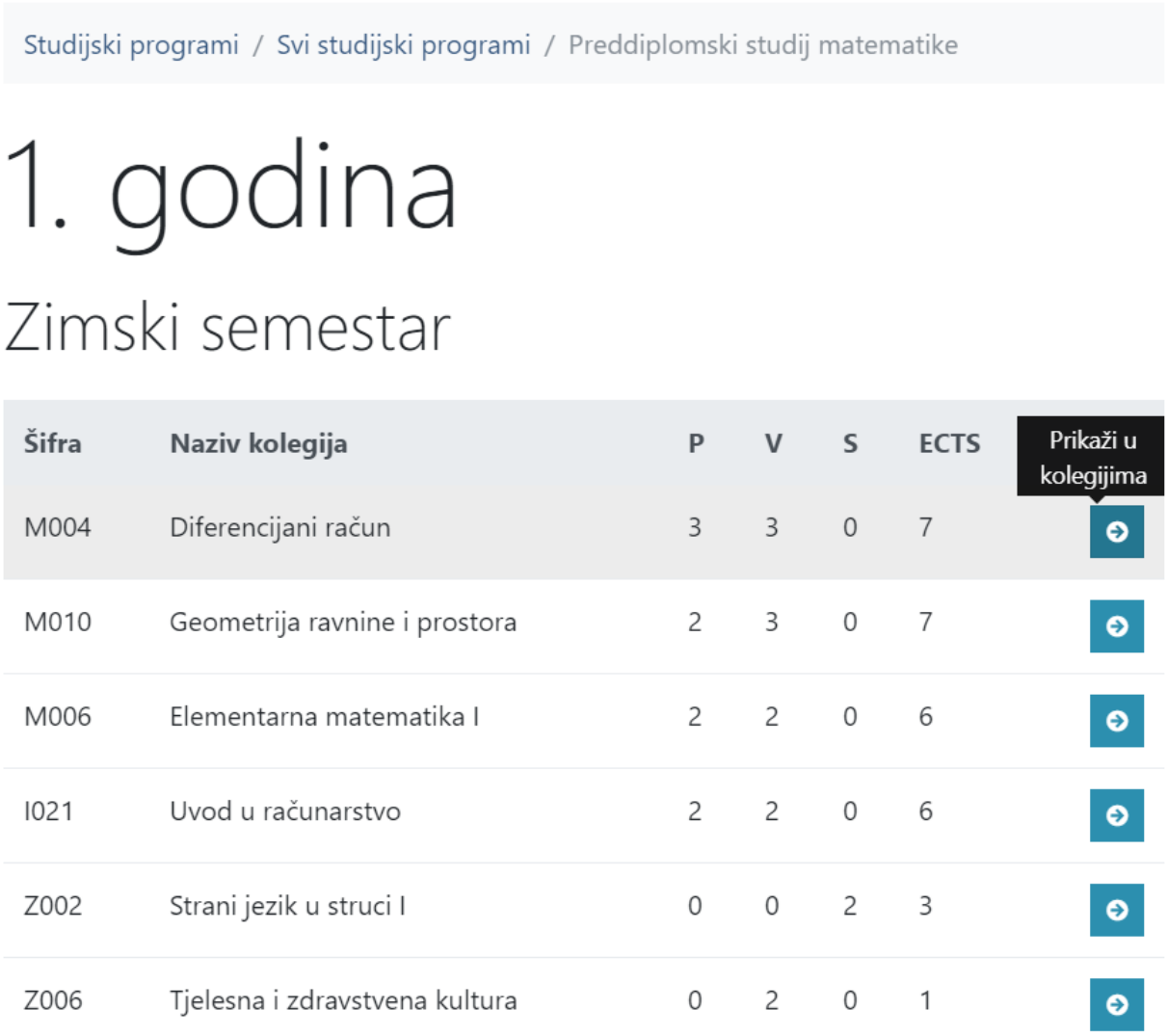
```
selectCourses() {
  let temp: CourseStudyProgramme[];
  temp = this.courses
    .sort(
      this.utilsService
          .dynamicSort(
            this.tableSortingService.getActive(this.tableOrderItems)
          )
    )
    .filter((item: CourseStudyProgramme)=>{
      return this.checkIfContained(item, this.term);
    });
  this.paginationInfo.totalItems = temp.length;
  let pageNo: number = this.paginationInfo.pageNo;
  let pageSize: number = this.paginationInfo.pageSize;
  this.selectedCourses = temp
    .slice((pageNo - 1)*pageSize, pageNo*pageSize);
}
```

Kod 31: Filtriranje, sortiranje i uzimanje potpolja kolegija.



Slika 19: Modal za uređivanje postojećeg i dodavanje novog kolegija.

Tablica studijskih programa na jednak način izlistava sve studijske programe, no budući da se radi o malom skupu podataka, nije potrebno filtriranje, sortiranje i obilježavanje stranica. Kako studijski program sadrži mnoštvo obaveznih i izbornih kolegija za svaku od godina, posebni prikaz izlistava sve kolegije pridružene određenom studijskom programu.



Slika 20: Kolegiji pridruženi određenom studijskom programu.

Zaključak

Koristeći Angular okvir, mehanizme koji dolaze uz njega i reaktivno programiranje, razvio sam klijentsku web aplikaciju MathosIntranet. Angular okvir potiće razvoj skalabilnih aplikacija koje sastoje od modula, stoga je moguće stvoriti druge module koji se na jednostavan način mogu ukomponirati unutar postojeće aplikacije. Time je Angular odlično rješenje za rad u timu. Također, budući da je Angular univerzalni okvir, napor za razvoj aplikacija nad različitim platformama je smanjen što povlači uštedu u novcu i vremenu.

Zbog prethodno navedene činjenice, smatram da je razvoj aplikacija uz pomoć univerzalnih okvira budućnost, ali i sadašnjost koju većina kompanija i timova treba usvojiti kako bi bili u koraku s vremenom te konkurentni. U drugu ruku, svjedoci smo sve većeg broja informacija u svijetu, stoga je za dobar softverski proizvod nužno slijediti programske paradigme i implementirati obrasce oblikovanja kako bismo osigurali efikasnost izvršavanja te ažuriranost podataka.

Literatura

[1] A. Banks, E. Porcello, *Learning React: Functional Web Development with React and Redux*, O'Reilly Media, Inc., 2017.

[2] K. Chinnathambi, *Learning React*, Pearson Education, Inc., 2017.

[3] J. Duckett, *HTML and CSS: Design and Build Websites*, John Wiley & Sons, Inc., 2011.

[4] A. Hussain, *Angular 4: From Theory To Practice*, Daolrevo Ltd., 2017.

[5] S. Mansilla, *Reactive Programming with RxJS*, The Pragmatic Programmers, 2015.

[6] N. Murray, F. Coury, A. Lerner, C. Taborda, *ng-book: The Complete Guide to Angular 4*, FULLSTACK.io, 2017.

[7] *Angular*, `https://angular.io/`

[8] M. Schwarzmüller, *Angular 2 - The Complete Guide*, `https://www.udemy.com/the-complete-guide-to-angular-2/`

[9] *React*, `https://facebook.github.io/react/`

[10] *RxJS Overview*, `http://reactivex.io/rxjs/manual/overview.html`

Sažetak

Ovaj diplomski rad opisuje Angular okvir i pripadne mehanizme koji omogućuju brz razvoj klijentskih aplikacija, neovisno radi li se o mobilnoj ili web platformi. Budući da se radi o univerzalnom okviru, nije potrebno poznavati više programskih jezika za različite platforme i većina se izvornog koda može primjeniti na svakoj od platformi. Angular aplikacija sastoji se od komponenata koje se ugnježđuju u stablo, a skup komponenata koje tvore logičku cjelinu smješten je unutar modula. Za manipulaciju prikaza brinu se direktive, dok ubrizgavanjem ovisnosti opskrbljujemo različite dijelove aplikacije s objektima koji su potrebni za njihov rad. Razvijajući aplikacije Angular okvirom, primjenjujemo programske paradigme poput reaktivnog programiranja kako bismo na što efikasniji način vodili računa o ažuriranosti podataka koji predstavljaju stanje aplikacije. Uz Angular, bitno je spomenuti i React - konkurentski i trenutno najpopularniji univerzalni okvir.

Ključne riječi: Angular, komponenta, modul, direktiva, ubrizgavanje ovisnosti, programska paradigma, obrazac oblikovanja, React

Summary

This master's thesis describes the Angular framework and the mechanisms included within that allow for the fast development of client applications both on mobile and web platforms. Since it is a universal framework, knowledge of multiple programming languages for different platforms is not required and the majority of the source code can be used on every platform. An Angular application consists of components nested within a tree and a set of components comprises a logical unit is located within a module. View manipulation is the task of the directives, while by dependency injection we supply various parts of the application with objects that are necessary for its functioning. By developing applications using the Angular framework we apply programming paradigms such as reactive programming to efficiently take into account whether the data that represents the state of the application is up-to-date. Alongside Angular it is important to mention React as well - a rival and currently most popular universal framework.

Key words: Angular, component, module, directive, dependency injection, programming paradigm, design pattern, React

Životopis

Jurica Maltar rođen je 1. travnja 1993. godine u Osijeku. Osnovnoškolsko obrazovanje završio je u Osnovnoj školi Matije Petra Katančića u Valpovu te u Glazbenoj školi Franje Kuhača u Osijeku, a srednjoškolsko u Isusovačkoj klasičnoj gimnaziji s pravom javnosti u Osijeku. 2012. godine upisuje preddiplomski studij matematike na Odjelu za matematiku Sveučilišta Josipa Jurja Strossmayera u Osijeku. 2015. godine završava preddiplomski studij s završnim radom na temu "Elektronička pošta" pod vodstvom izv. prof. dr. sc. Domagoja Matijevića, a iste godine upisuje diplomski studij matematike, smjer Matematika i računarstvo. Tijekom diplomskog studija sudjeluje na programerskom natjecanju IEEEXtreme, pohađa studentske prakse u poduzećima Adacta d.o.o. i Farmeron d.o.o., obavlja studentski posao u poduzeću Rocksoft obrt za razvoj softvera te sudjeluje kao predavač Ljetne škole informatike u organizaciji Osijek Software City-a.