

Dizajn i programiranje korisnickog interfejsa



Mlađan Jovanović
Nikola Savanović



UNIVERZITET SINGIDUNUM

Mlađan Jovanović
Nikola Savanović

DIZAJN I PROGRAMIRANJE KORISNIČKOG INTERFEJSA

Prvo izdanje

Beograd, 2023.

DIZAJN I PROGRAMIRANJE KORISNIČKOG INTERFEJSA

Autori:

Prof. dr Mlađan Jovanović
Nikola Savanović, MA

Recenzenti:

Prof. dr Mladen Veinović
Prof. dr Aleksandar Jevremović

Izdavač:

Univerzitet Singidunum
Danijelova 32
www.singidunum.ac.rs

Za izdavača:

Prof. dr Milovan Stanišić

Priprema za štampu:

Mlađan Jovanović

Dizajn korica:

Aleksandar Mihajlović, MA

Godina izdanja:

2023.

Tiraž:

800

Štampa:

Birograf, Beograd

ISBN: 978-86-7912-792-1

Copyright:

© 2023. Univerzitet Singidunum

Izdavač zadržava sva prava.

Reprodukcija pojedinih delova ili celine ove publikacije nije dozvoljena.

Predgovor

Udžbenik je rezultat višegodišnje saradnje sa studentima 3. godine smera Računarske Nauke na predmetu Programiranje i dizajn korisničkih interfejsa i studentima 3. godine smera Softversko i Informaciono Inženjerstvo na predmetu Klijentske Veb aplikacije, Univerziteta Singidunum. Namenjen je i široj publici zainteresovanoj za oblasti dizajna i programiranja korisničkog interfejsa.

Intuitivan i privlačan za korišćenje korisnički interfejs predstavlja ključni faktor prihvatanja računarskih proizvoda i sistema. Korisnik čitav sistem vidi preko korisničkog interfejsa. Stoga uspeh nekog softverskog i hardverskog proizvoda zavisi od pogodnosti korišćenja funkcija sistema putem interfejsa od strane krajnjih korisnika. Ako korisnik ne može lako i intuitivno koristiti interfejs, vrlo je verovatno da sistem neće postati prihvaćen. Zbog toga je razvoj korisničkih interfejsa sve bitniji deo razvoja nekog programskog sistema.

Udžbenik najpre daje širi kontekst oblasti korisničkog interfejsa kroz istorijski razvoj i vrste. Zatim opisuje grafički korisnički interfejs (GKI) kao najzastupljeniju vrstu korisničkog interfejsa. Prikazana je arhitektura sa obrascima, pristupi izradi, i ulazni i izlazni mehanizmi GKI. Implementacija principa dizajna u izradi GKI čini korisnički interfejs estetski privlačnijim i lakšim za korišćenje te je njima posvećena posebna sekcija.

Sekcije u nastavku daju pregled programskih jezika za izradu Veb GKI i tehnologija zasnovanih na ovim jezicima. Prikazani su osnovni principi i mehanizmi JavaScript i TypeScript jezika čije je poznavanje neophodno za implementaciju Veb GKI. Jezici predstavljaju osnovu Angular okvira za razvoj korisničkog interfejsa. Angular Material je biblioteka standardnih elemenata GKI sa unapređenim izgledom i funkcijama. Pored toga, opisane su i tehnologije za vizuelni dizajn Veb GKI, FlexBox i FlexLayout.

Na kraju je korišćenje navedenih tehnologija demonstrirano kroz prototip aplikacije za podršku i promociju fizičke aktivnosti korisnika.

Zahvaljujemo se našim studentima i recenzentima na savetima i sugestijama za unapređenje udžbenika.

Sugestije i komentare u vezi sa udžbenikom uputiti na adresu mjovanovic@singidunum.ac.rs ili nsavanovic@singidunum.ac.rs.

Autori,

Beograd, Mart 2023.

Sadržaj

1. Korisnički interfejs: uvod, razvoj i ciljevi	7
1.1. Razvoj	7
1.2. Terminologija i konvencije udžbenika.....	13
1.3. Cilj i struktura udžbenika	14
2. Grafički korisnički interfejs	14
2.1. Arhitektura interfejsa	15
2.1.1. Stablo prikaza.....	15
2.1.2. Događaji i osluškivači.....	16
2.1.3. Model-prikaz-kontroler obrazac	18
2.2. pristupi programiranju interfejsa	22
2.3. Obrasci arhitekture interfejsa	25
2.3.1. Hijerarhija	25
2.3.2. Centralizovani obrazac.....	28
2.3.3. Ugnježdjeni obrazac.....	29
2.3.4. Kartični obrazac	30
2.3.5. Komandna tabla	31
2.3.6. Filter	32
2.4. Ulaz interfejsa	34
2.5. Izlaz interfejsa	36
3. Dizajn grafičkog korisničkog interfejsa.....	40
3.1. Jednostavnost	40
3.2. Kontrast.....	47
3.3. Ponavljanje.....	50
3.4. Ravnanje.....	50
3.5. Blizina	51
3.6. Boje i tipografija	52
4. JavaScript i TypeScript jezici	55

4.1. Proceduralni JavaScript.....	56
4.1.1. Operatori i naredbe.....	56
4.1.2. Varijable.....	57
4.1.3. Funkcije.....	59
4.1.4. Tipovi.....	62
4.2. Objektno-orijentisani JavaScript.....	70
4.2.1. Objekti i funkcije	70
4.2.2. Prototipi.....	76
4.2.3. Klase.....	81
4.3. Elementi funkcionalnog programiranja	84
4.4. Interaktivni elementi JavaScript jezika	89
4.4.1. Stablo objekata.....	89
4.4.2. Događaji.....	98
4.5. TypeScript jezik	108
4.5.1. Osnovni tipovi.....	108
4.5.2. Interfejsi i klase	110
4.5.3. Funkcije.....	116
4.5.4. Prevođenje.....	119
5. Angular okvir za razvoj korisničkog interfejsa.....	123
5.1. Gradivni elementi Angular okvira	125
5.2. Modul.....	127
5.3. Komponenta	131
5.3.1. Struktura.....	131
5.3.2. Životni ciklus komponente.....	138
5.3.3. Povezivanje podataka.....	140
5.3.4. Direktive.....	148
5.4. Servisi.....	153
5.5. Rutiranje.....	159
6. Angular Material elementi korisničkog interfejsa	163

6.1. Instalacija	165
6.2. Grupisanje AM komponenti.....	166
6.3. Teme.....	168
6.4. Animacije AM komponenti	170
6.5. Dugme	171
6.6. Ikona.....	174
6.7. Polje za unos	176
6.8. Padajuća lista.....	179
6.9. Polje za potvrdu	181
6.10. Forma	183
6.11. Kalendar	186
6.12. Vertikalna linija alata	187
6.13. Horizontalna linija alata	188
6.14. Lista.....	189
6.15. Višestruki pogledi	191
6.16. Kartica	193
6.17. Tabela.....	195
6.17.1. Kreiranje tabele	195
6.17.2. Sortiranje tabele	201
6.17.3. Straničenje tabele	204
6.17.4. Filtriranje tabele	206
6.18. Dijalog.....	208
6.19. Korak.....	214
6.20. Klizač	220
7. Tehnologije za dizajn korisničkog interfejsa	222
7.1. FlexBox biblioteka.....	222
7.1.2. Direktive rasporeda roditeljskih elemenata.....	224
7.1.2. Raspoređivanje elemenata dece	230
7.2. FlexLayout biblioteka	232

7.2.1. Raspoređivanje roditeljskih elemenata	233
7.2.2. Direktive rasporeda elemenata dece	237
8. Studija slučaja Angular aplikacije.....	241
8.1. Uvod u strukturu projekta i instalacija potrebnih biblioteka.....	241
8.1.1. Razvojno okruženje i kreiranje projekta	241
8.1.2. Korišćenje razvojnog servera.....	244
8.1.3. Kreiranje komponenti	245
8.1.4. Instalacija AM biblioteke.....	246
8.1.5. Kreiranje modula za registraciju AM komponenti	247
8.1.6. Navođenje modula za registraciju AM komponenti u korenom modulu	248
8.1.7. Dodavanje dugmeta u templejt komponente.....	249
8.1.8. Implementacija rutiranja	250
8.2. Prikaz i raspored elemenata GKI aplikacije.....	252
8.2.1. Korišćenje predefinisanih tema i animacija	253
8.2.2. Podrška za ekrane osjetljive na dodir.....	253
8.2.3. Dodavanje ikona	254
8.2.4. Instalacija FlexLayout biblioteke.....	255
8.2.5. Dodavanje forme.....	256
8.3. Uređivanje forme	258
8.3.1. Dugme za slanje podataka forme	258
8.3.2. Pristup i korišćenje podataka forme.....	258
8.3.3. Smernice za korišćenje polja forme	260
8.3.4. Obrada grešaka i validacija unosa u polje forme	260
8.3.5. Dodavanje kalendara.....	261
8.3.6. Korišćenje polja za potvrdu	262
8.3.7. Dodatne funkcije forme	262
8.4. Dodavanje navigacije.....	264
8.4.1. Vertikalna linija alata za navigaciju.....	264

8.4.2. Horizontalna linija alata za navigaciju.....	267
8.4.3. Uređivanje izgleda navigacije.....	267
8.5. Prilagođavanje interfejsa različitim dimenzijama prikaza.....	271
8.5.1. Dodavanje podrške za različite veličine ekrana.....	271
8.5.2. Dodavanje navigacione liste.....	273
8.5.3. Dodavanje višestrukih pogleda.....	276
8.5.4. Dodavanje kartice.....	277
8.6. Razmena podataka između komponenti.....	278
8.6.1. Dodavanje padajuće liste.....	278
8.6.2. Povezivanje komponenti treninga.....	279
8.6.3. Dodavanje progres elementa.....	283
8.6.4. Uređivanje progres elementa.....	284
8.7. Dodavanje tabele sa simulacijom pozadinske logike interfejsa.....	287
8.7.1. Priprema podataka tabele korišćenjem servisa.....	287
8.7.2. Kreiranje prikaza tabele.....	289
8.7.3. Popunjavanje tabele.....	290
8.7.4. Sortiranje tabele istorijata treninga.....	291
8.7.5. Straničenje tabele istorijata treninga.....	293
8.8. Filtriranje tabele i dodavanje dijaloga.....	296
8.8.1. Filtriranje tabele istorijata treninga.....	296
8.8.2. Dodavanje dijaloga za rad sa tekućim treningom.....	297
8.8.3. Prosleđivanje podataka dijalogu.....	300
8.9. Upravljanje dijalogom i dodavanje teme.....	302
8.9.1. Prosleđivanje podataka od dijaloga ka pozivajućoj komponenti.....	303
8.9.2. Kreiranje teme.....	305
8.9.3. Uređivanje početne stranice.....	308
8.10. Simulacija pozadinske logike.....	309
8.10.1. Kreiranje servisa za rad sa korisničkim nalogima.....	309
8.10.2. Izmene komponente za registraciju korisnika.....	312

8.10.3. Izmene komponente za prijavu korisnika	313
8.11. Ažuriranje profila korisnika	315
8.11.1. Proširenje servisa za rad sa korisničkim nalogima.....	316
8.11.2. Kreiranje komponente profila korisnika	318
8.11.3. Rad sa komponentom profila korisnika	323
9. Literatura	325
Spisak slika i tabela.....	331
Spisak skraćenica	339

1. Korisnički interfejs: uvod, razvoj i ciljevi

Za korisnika računarskog sistema korisnički interfejs predstavlja čitav sistem. Korisnik delove sistema i njegovu funkcionalnost vidi isključivo preko odgovarajućih elemenata tog interfejsa, pa zbog toga uspeh nekog softverskog proizvoda zavisi od pogodnosti i prihvatanja korisničkog interfejsa od strane korisnika. Ako korisnik ne može lako i intuitivno koristiti programski sistem, vrlo je verovatno da sistem neće postati šire prihvaćen, i zbog toga je razvoj korisničkih interfejsa sve bitniji deo razvoja nekog programskog sistema. Dobro dizajniran, intuitivan i privlačan za korišćenje korisnički interfejs predstavlja ključni faktor uspeha računarskih proizvoda i sistema. Pored toga, nove unapređene mogućnosti računara, kao i pojava novih računarskih uređaja i aplikacija, donele su i potrebu za unapređenjem interakcije između čoveka i računara.

1.1. Razvoj

Da bismo bolje sagledali trenutno stanje i pravce razvoja, u ovom delu dajemo kratak pregled istorije korisničkih interfejsa. Ovde ćemo se prvenstveno zadržati na opisu početaka razvoja korisničkih interfejsa, osnovnim faktorima koji su upravljali ovim razvojem, i pregledu postojećih tehnologija.

Termin "korisnički interfejs" nije bio relevantan u prvim računarskim sistemima jer su prvi korisnici računara bili su inženjeri, a korišćenje računara je zahtevalo potpuno poznavanje hardvera. Inženjeri su interakciju sa računarom obavljali neposrednim radom na hardveru, na primer, podešavanjem prekidača, a osnovni oblik komunikacije je bio razmena brojeva u binarnom, oktalnom ili heksadecimalnom brojevnom sistemu. Pri tome su inženjeri direktno radili sa registrima i memorijskim lokacijama, a težište u radu sa računarima je bilo poboljšanje performansi izvršavanja instrukcija.

U početku je hardver bio centralni deo korisničkog interfejsa, a tipični korisnici su bili inženjeri i programeri. Unapređenje korisničkog interfejsa išlo je u tri pravca [1]. Prvo, razvijan je pouzdaniji hardver sa više mogućnosti, tako da su korisnici mogli da sa manje komandi i većom pouzdanošću koriste računare. Drugo, ergonomska prezentacija i manipulacija hardverskih komponenti je unapređivana, na primer, pogodnijem rasporedom komponenti i dodavanjem oznaka na prekidače. Treće, korisnici računara, prvenstveno programeri, su postepeno bili oslobođani potrebe da poznaju sve detalje funkcionisanja hardvera, na primer, uvođenjem koncepta virtuelne memorije. Čak i 70-tih godina dvadesetog veka, računari su imali sličan hardverski

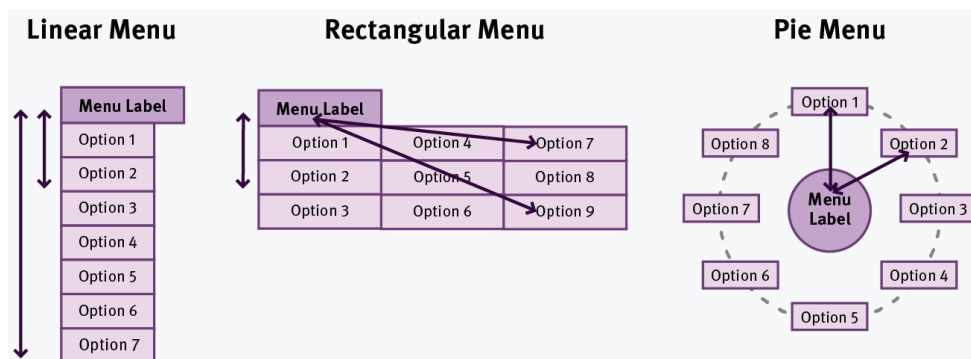
korisnički interfejs, dok su napredniji računari često još više smanjivali udaljenost između korisnika i hardvera [2]. Većina korisnika je radila sa računarima prema precizno definisanim uputstvima. Fizička ograničenja, kao što je veličina memorije, bila su izuzetno velika. Pored toga, računarska oprema i radno vreme računara bili su veoma skupi. Na primer, prvi ekrani sa katodnom cevi (eng. *cathode ray tube* - CRT) koštali su više od 10.000 dolara.

Tokom 60-tih i 70-tih godina dvadesetog veka, programeri su ostali glavni korisnici računara. Prvi editori teksta bili su linijski orijentisani editori namenjeni programerima. Upotreba računara u cilju proizvoljne obrade teksta bila je preskupa. U osnovi, unapređenje korisničkog interfejsa značilo je unapređenje efikasnosti programera. Kritična unapređenja, iako još nisu bila okarakterisana kao unapređenja korisničkog interfejsa, bili su viši programski jezici, "debageri" i operativni sistemi. Međutim, navedena unapređenja obično nisu uzimala u obzir ljudske faktore poput čitljivosti koda ili unapređenja pamćenja termina. U slučaju programskih jezika, na primer, osnovni fokus je bio stavljen na boljem strukturiranju koda i podizanju nivoa apstrakcije, a ne na unapređenju ergonomske karakteristike izrade koda [3]. Tek kasnije su se pojavile studije o programiranju kao obliku rešavanja problema, o strategijama otkrivanja grešaka (eng. *debugging*), o efektivnoj programskoj dokumentaciji i o vizuelizaciji programa. Čak i početkom 80-tih godina dvadesetog veka, većina istraživanja je i dalje bila fokusirana na programiranje kao osnovni oblik interakcije sa računarom i zasnovana na pretpostavci da je osnovni korisnik računara programer.

Od niza unapređenja, za programere su najznačajnije novine predstavljale mogućnosti rada sa više procesa (eng. *multitasking*), virtuelna memorija i interaktivni terminali. Interaktivni terminali su doneli najznačajnije promene, jer su ne samo povećali broj korisnika već su doveli do pojave nove klase korisnika sistema koji nisu bili programeri. Termin "korisnički interfejs" je uskoro postao često korišćen, a nova oblast istraživanja interakcije između čoveka i računara počela je da se formira. Iako je rad na unapređenju programerskih okruženja i dalje nastavljen, težište rada u oblasti interakcije između čoveka i računara je prešlo sa pomoći programerima kao korisnicima, na pomoć programerima da razviju bolje interfejse za korisnike koji nisu programeri.

Vizuelni displeji i interaktivne mogućnosti terminala otvorile su novu oblast razvoja i istraživanja. Uskoro se pojavio velik broj istraživanja o percepcijskim problemima interakcije između čoveka i računara poput čitljivosti ispisa ili istraživanja motorike u radu sa mehaničkim ulaznim uređajima. Postojeća istraživanja ergonomije i drugih ljudskih faktora, počela su da se primenjuju i proširuju na oblast interakcije između čoveka i računara.

Kao tipičan primer možemo navesti Fittsov zakon koji opisuje zavisnost između vremena potrebnog za označavanje nekog objekta do njegove udaljenosti i veličine [4]. Ovaj zakon je definisan 1954. godine, ali je vrlo uspešno korišćen i kasnije [5], kao što je prikazano na Slici 1.1.



Slika 1.1. Fitts-ov zakon kaže da vreme pokreta ka cilju zavisi od veličine cilja i udaljenosti do cilja. Na slici, prosečna udaljenost od naziva menija do stavke menija zavisi od tipa menija: linearni meni je manje efikasan od pravougaonog menija, koji je dalje manje efikasan od kružnog menija. Izvor: <https://www.nngroup.com/articles/fitts-law/>.

Fleksibilnost i laka proširivost računarskih sistema uskoro je dovela do pojave interaktivnih elemenata korisničkih interfejsa poput funkcijskih tastera, komandnih jezika i menija. Novi elementi doveli su do proširenja istraživanja na teme iz sazajne psihologije poput učenja motoričkih radnji, formiranja konceptata, semantičke memorije i akcija.

Rad na percepcijskim i osnovnim sazajnim procesima zajedničkim za većinu korisnika ostao je dominantan fokus istraživanja u oblasti interakcije između čoveka i računara. Neki od osnovnih istraživačkih tema o interakciji između čoveka i računara, koje su aktuelne i danas, definisao je 1986. godine Ben Shneiderman na CHI'86 konferenciji [6]. Oblast interakcije između čoveka i računara je okupila veliki broj inženjera i istraživača iz psihologije, medicine, sazajnih nauka, ali i određen broj umetnika i dizajnera.

Nakon formiranja oblasti interakcije između čoveka i računara, broj istraživača i radova je značajno porastao. Pored istraživanja percepcijskih i sazajnih mehanizama pojedinačnih korisnika, počela su da se primenjuju i istraživanja iz sociologije u cilju podrške grupnim aktivnostima korisnika.

Bez obzira na činjenicu da mnoga istraživanja iz oblasti interakcije između čoveka i računara nisu odmah našla praktičnu primenu, neka od njih su imala značajan uticaj na razvoj današnjih računarskih sistema. Na primer, grafički interfejs koji danas postoji u *Mac*, *Windows* i *Linux* operativnim sistemima,

zasnovan je na rešenjima koja su početkom 80-tih godina prošlog veka koristili *Apple* računari. Sa druge strane, *Apple* računari su primenili istraživanja iz Xerox PARC (Palo Alto Research Center) laboratorija, zasnovana na istraživanjima iz 60-tih godina dvadesetog veka na Univerzitetu Stanford [7].



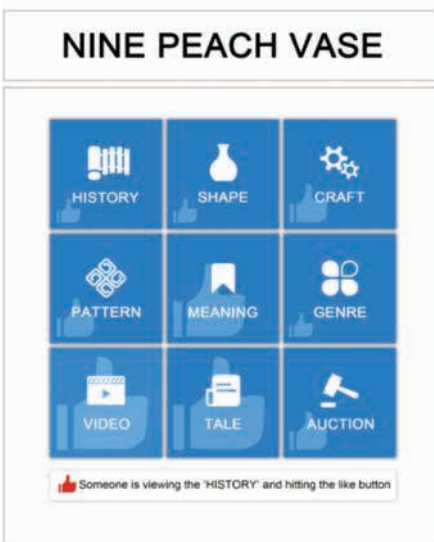
Slika 1.2. Xerox PARC radna stanica iz 1973. godine. Izvor: <https://www.computerhistory.org/revolution/input-output/14/347>.

Iako je postignut veliki komercijalni uspeh GKI (eng. *graphical user interfaces* - *GUI*), interakcija između čoveka i računara nije značajno unapređena u periodu od tri decenije. Mada je GKI danas *de facto* standard, on još uvek nije dovoljno efikasan i intuitivan u praksi. Komunikacija sa računarom preko ovih interfejsa ne koristi dovoljno čovekove percepcijske i izražajne mogućnosti i značajno je drugačija od uobičajne međuljudske komunikacije. Zbog toga je sproveden veći broj istraživanja i eksperimenata u cilju unapređenja interakcije između čoveka i računara eksploatišući u većoj ili manjoj meri karakteristike čovekovog percepcijskog aparata. Nove paradigme interakcije između čoveka i računara polaze od pretpostavke da komunikacija između njih treba da bude zasnovana na istim principima koje obični ljudi svakodnevno koriste pri međusobnoj komunikaciji ili pri opažanju sveta oko sebe. To je dovelo do pojave više novih vrsta korisničkih interfejsa kao što su percepcijski korisnički interfejsi [8], korisnički interfejsi zasnovani na pažnji [9], elektro-fiziološki korisnički interfejsi [10], dodirni korisnički

interfejsi [11], konverzacijski korisnički interfejsi [12], i korisnički interfejsi virtualne i proširene stvarnosti [13].

Preciznije, prema korišćenoj tehnološkoj bazi i karakteristikama interakcije sa korisnikom, korisničke interfejse možemo podeliti u sledeće grupe:

- *Hardverski korisnički interfejsi*. Vezani za početke komunikacije čoveka i računara koje karakteriše ručno povezivanje hardverskih komponenti, upotreba dugmadi, prekidača, signalnih lampica i bušenih kartica.
- *Terminalski korisnički interfejsi*. Podrazumevaju korišćenje teleprinterskih ili video terminala gde je najviše korišćeni oblik interakcije bio putem tekstualnog interfejsa komandne linije.
- *Grafički korisnički interfejsi*. Zasnovani na WIMP (eng. *windows, icons, menus, pointer*) načinu interakcije [14].
- *Percepcijski korisnički interfejsi*. Imaju sposobnost opažanja korisnika i njegovih aktivnosti [8]. Primer predstavlja *Microsoft Azure Kinect* kamera koja koristi računarsku viziju (eng. *computer vision*) za detekciju pokreta radi prepoznavanja aktivnosti korisnika [15].
- *Korisnički interfejsi zasnovani na pažnji* (eng. *attentive user interfaces - AUI*) kao osnovni cilj imaju prepoznavanje prostora korisnikovih namera u cilju optimizacije resursa za obradu informacija korisnika i računara. Ovo postižu merenjem i modelovanjem usmerenosti korisnikove pažnje ka zadatku, uređajima i drugim korisnicima [9].
- *Elektro-fiziološke korisnički interfejsi*. Kombinuju tehnologije za opažanje fizioloških signala iz medicinskih aplikacija sa raznim tehnikama izrade interaktivnih računarskih sistema [10]. Predstavljaju platformu za mnoge aplikacije kao što su interfejsi između mozga i računara (eng. *brain-computer interfaces - BCI*) [16].
- *Osetni korisnički interfejsi* (eng. *tangible user interfaces - TUI*). Vezani za disciplinu interakcije čoveka i računara zasnovanu na korišćenju metafora koje oblikuju digitalne informacije u fizičke forme iz realnog sveta (eng. *Tangible Interaction*) [17]. Cilj je postizanje integracije fizičkog i digitalnog sveta kako bi se unapredila sama komunikacija čoveka i računara. Ovo se postiže angažovanjem znanja i veština čoveka u korišćenju svakodnevnih objekata iz realnog sveta poput prostornih, motoričkih i socijalnih veština [18].
- *Konverzacijski korisnički interfejsi* (eng. *Conversational User Interface - CUI*) su zasnovani na konverzaciji ili dijalogu sa korisnikom korišćenjem prirodnog jezika (eng. *natural language - NL*). Konverzacija može biti tekstualna ili govorna [19].
- *Korisnički interfejsi virtualne i proširene stvarnosti* (eng. *virtual reality - VR, augmented reality - AR*). Kod interfejsa virtualne stvarnosti okruženje koje korisnik vidi je u potpunosti računarski generisano ili sintetičko, dok se kod interfejsa proširene stvarnosti računarski kreirani objekti dodaju preko prikaza realnog sveta korisniku, kao na slici ispod.



Slika 1.3. Primer korisničkog interfejsa proširene stvarnosti. Izvor [13].

Sve češći zahtevi za značajnim izmenama paradigme interakcije, posledica su i pojave novih računarskih uređaja uređaji poput prenosivih računara i mobilnih telefona koji ne mogu da iskoriste rešenja sa stonih računara. Kao što je predvideo Mark Weiser u Xerox PARC laboratoriji, doba sveprisutnog računarstva (eng. *ubiquitous computing*) je u toku [20]. Weiser je predvideo da će okruženja za život i rad biti ispunjena računarskih i komunikacionim servisima koji su orijentisani ka čoveku i da će čovek u svakom trenutku i na svakom mestu imati pristup ovim servisima. Danas su ovi servisi dostupni čoveku zahvaljujući uređajima kao što su *IPhone* mobilni telefoni ili *IPad* uređaji, koji imaju integrisane tekstualno-govorne mehanizme, ekrane osetljive na dodir i čitav skup servisa poput *location-based* servisa (*GoogleMaps*) ili servisa društvenih mreža (*Facebook*, *Twitter*, *LinkedIn*). Sa druge strane, prikazne tehnologije velikih dimenzija, poput displeja veličine zida (eng. *wall-size displays*) ili displeja proširene stvarnosti (eng. *augmented-reality displays*) su već dostupni u obliku projekcionih uređaja ili u obliku displeja velikih dimenzija. Interfejsi na ovim veoma malim ili veoma velikim displejima tipično ne mogu koristiti standardni *desktop* model. Pored toga, *Windows* elementi korisničkog interfejsa poput padajućih menija nisu pogodni

ni za velike displeje veličine zida, jer meniji mogu biti nedostupni za određenu grupu korisnika. Nadalje, korisnici koriste više uređaja istovremeno, tako da će uređaji morati da komuniciranju i usklađuju svoje funkcije. Ove promene iziskuju razvoj odgovarajućih korisničkih interfejsa. Raznolikost uređaja za interakciju postavlja nove izazove za razvoj korisničkog interfejsa [21], kao što su:

- Kreiranje i održavanje verzija jednog korisničkog interfejsa za različite tipove ili dimenzije uređaja.
- Konzistentnost između verzija radi održavanja istovetnog načina interakcije sa korisničkim interfejsom na različitim uređajima.
- Dinamičko prilagođavanje korisničkog interfejsa promenama u fizičkom okruženju i lokaciji korisnika, karaktteristikama uređaja, kao i potrebama i osobinama konkretnog korisnika.

1.2. Terminologija i konvencije udžbenika

U ovom delu navodimo konvencije koje smo koristili pri pisanju teksta, kao i definicije korišćenih termina.

Nazivi na engleskom jeziku, kao i imena varijabli, funkcija i drugih objekata iz programskog koda se navode kao *kurziv*. Delovi teksta koji sadrže programske naredbe i primere koda koriste `code` familiju fontova i nalaze se u oivičenim paragrafima. U nekim primerima koda rezultat je prikazan kao slika u nastavku. U zavisnosti od složenosti, primeri programa često prikazuju samo relevantnije skupove naredbi umesto kompletnog koda.

Terminologija koja se koristi u oblasti dizajna i programiranja korisničkog interfejsa još uvek nije ujednačena. Značenje termina često zavisi od konteksta u kojem se oni upotrebljavaju. Radi ispravnog razumevanja sadržaja udžbenika, u nastavku su definisani ključni termini korišćeni u tekstu.

- *Dizajn korisničkog interfejsa* predstavlja proces uređivanja izgleda GKI u pogledu vizuelnih karakteristika kao što su veličina, boja, kontrast i raspored elemenata.
- *Programiranje korisničkog interfejsa* se odnosi na implementaciju izgleda i ponašanja GKI.
- *Element GKI* je standardni deo GKI koji ima unapred definisan osnovni izgled, ponašanje i namenu koji se mogu menjati i prilagoditi potrebama konkretnih GKI. Na primer, dugme, meni, tekstualno polje i drugi. Sinonim je *kontrola GKI*.
- *Interaktivni prototip GKI* je interfejs koji nema implementiranu logiku aplikacije po dubini u smislu funkcija u pozadini i skladišta podataka. Funkcije i podaci se simuliraju u samom interfejsu.
- *Korisnik* označava krajnjeg korisnika, tj. osobu koja koristi GKI.

1.3. Cilj i struktura udžbenika

Osnovni cilj udžbenika je pregled principa i tehnika dizajna i implementacije GKI sa naglaskom na Veb korisnički interfejs. Udžbenik podrazumeva poznavanje programskih jezika za kreiranje i vizuelni dizajn Veb strana, tj. HMTL (eng. *HyperText Markup Language*) i CSS (eng. *Cascading Style Sheets*). Pored toga, poželjno je poznavanje principa programiranja i algoritama i struktura podataka, uključujući i objektno-orijentisane programske jezike.

Udžbenik sadrži osam poglavlja.

Poglavljje 2 daje detaljniji pregled oblasti GKI kroz softversku arhitekturu i njene obrasce, pristupe programiranju, i ulazne i izlazne mehanizme.

Poglavljje 3 opisuje principe dizajna strukture i sadržaja GKI kako bi bio privlačniji i lakši za korišćenje za krajnje korisnike.

Poglavljje 4 obrađuje programske jezike koji se koriste za izradu Veb GKI, JavaScript i TypeScript. Ovi jezici predstavljaju osnovu naprednijih tehnologija za kreiranje interfejsa.

Poglavljje 5 predstavlja arhitekturu Angular tehnologije za dizajn i programiranje Veb GKI kroz analizu gradivnih elemenata i relacija između njih.

Poglavljje 6 ilustruje korišćenje Angular Material biblioteke standardnih elemenata GKI. Elementi se nazivaju i Angular Material komponente.

Poglavljje 7 prikazuje FlexBox i FlexLayout tehnologije za dizajn kreiranih interfejsa. Tehnologije čine raspoređivanje sadržaja interfejsa lakšim i efikasnijim za programere.

Poglavljje 8 integriše prethodna poglavlja kroz studiju slučaja, odnosno primer aplikacije.

2. Grafički korisnički interfejs

GKI je i danas najzastupljenija vrsta korisničkog interfejsa. Kod ove vrste interfejsa, interakcija sa korisnikom se odvija korišćenjem:

- *Prozora*, kao radnih površina na kojima korisnici obavljaju zadatke pozivajući funkcije aplikacije koju interfejs reprezentuje;
- *Ikona*, kao simbola ili oznaka akcija na korisničkom interfejsu, odnosno njegovih funkcija;
- *Menija*, kao mehanizama koji omogućavaju kretanje (navigaciju) u interfejsu, selekciju elementa iz grupe elemenata, i pokretanje akcije iz grupe akcija;
- *Pokazivača*, kao trenutnog položaja korisnika (orijentacija) u virtuelnom svetu korisničkog interfejsa, odnosno aplikacije koju predstavlja.

Pravila opisana iznad su poznata i pod nazivom *WIMP* (eng. *Windows, Icons, Menus, Pointer*) princip [22].

2.1. Arhitektura interfejsa

Softversku arhitekturu GKI koja i danas predstavlja osnovu implementacije je ustanovio Den Olsen u svojoj knjizi *Implementacija korisničkih interfejsa* 1998. godine [23]. Olsen je objedinio i sistematizovao postojeća znanja tog vremena u arhitekturu koju i danas realizuje većina korisničkih interfejsa.

Softversku arhitekturu GKI čine tri dela:

1. *Stablo prikaza* (eng. *view tree*) kao struktura za organizaciju elemenata korisničkog interfejsa. Drugim rečima, elementi (kao prikazi) su organizovani u obliku hijerarhije stabla.
2. *Događaji i osluškivači* (eng. *events and listeners*) kao mehanizam komunikacije između korisnika (čoveka) i interfejsa u kojem se fizičke akcije korisnika (npr. klik miša, dodir dugmeta ili pritisak tastera tastature) transformišu u logički, tj. programski objekat događaja (*event*) i aplikacija kao odgovor izvršava funkciju vezanu za nastanak događaja (*listener*).
3. *Model-prikaz-kontroler obrazac* (eng. *model-view-controler pattern*) koji raspodeljuje odgovornosti kako na nivou pojedinačnog elementa, tako i na nivou čitavog korisničkog interfejsa. Model čuva stanje, odnosno podatke. Prikaz ili pogled prikazuje sadržaj. Kontroler definiše pozadinsku logiku interfejsa ili elementa kao komunikaciju između modela i pogleda.

2.1.1. Stablo prikaza

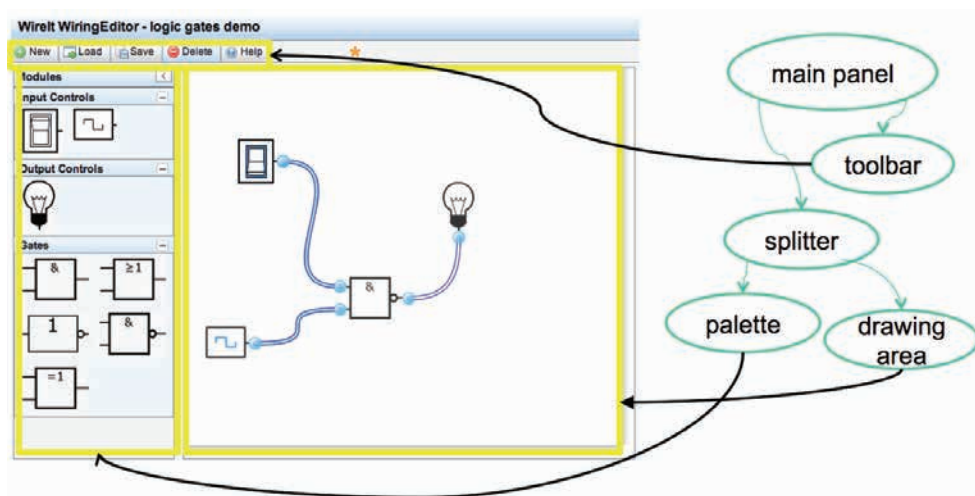
Stablo prikaza organizuje sadržaj korisničkog interfejsa kao skup prikaza ili elemenata (čvorovi stabla) povezanih ivicama (konekcijama između čvorova). Ivice predstavljaju relaciju sadržavanja. Ovakav način modelovanja strukture GKI omogućava fleksibilnost i raznovrsnost sadržaja korisničkog interfejsa i interakcije između njegovih delova. Prikaz je element korisničkog interfejsa koji se prikazuje u određenom delu korisničkog interfejsa i obično zauzima pravougaonu površinu. U zavisnosti od primarne namene, razlikuju se sledeće vrste elemenata prikaza:

- Ulazni element (eng. *input element*) prihvata ulazne podatke korisnika i prosleđuje logici korisničkog interfejsa na obradu. Korisnik može prosleđivati podatke korišćenjem tastature, miša ili dodira. Primeri ulaznih elemenata su tekstualno polje (eng. *text field*), tekstualno dugme (eng. *button*) i dugme ikona (eng. *icon*). Kod tekstualnog polja podaci su tekstualni sadržaj, dok se kod tekstualnog dugmeta i ikone prosleđuju naredbe ili akcije koje pokreću funkcije korisničkog interfejsa.
- Izlazni element (eng. *output element*) čija je osnovna funkcija prosleđivanje informacija ka korisniku. Informacija može biti tekstualna

ili vizuelna. Primer prosleđivanja tekstualne informacije predstavlja labela (eng. *label*), dok vizuelnu informaciju može prikazivati progres element (eng. *progress bar*) i slika (eng. *image*). Ovi elementi menjaju izgled korisničkog tako što modifikuju deo stablo prikaza na koji se promene odnose, kao izmene ili uklanjanje postojećih elemenata, ili dodavanje novih elemenata.

- Element kontejner (eng. *container element*) služi za grupisanje (eng. *grouping*) i raspoređivanje (eng. *layout*) jednostavnijih elemenata koji mogu biti ulazni, izlazni i kontejneri. Primeri kontejnerskih elemenata su panel (eng. *panel*), prozor (eng. *window*) i linija alata (eng. *toolbar*).

U tipičnom scenariju (kao na Slici 2.1), korisnik zadaje komande preko ulaznih elemenata, kao što su dugmad ili graničnik (eng. *splitter*). Na osnovu zadatih komandi akcija (čuvanje promena ili promena veličine specifičnog prikaza), prikaz odgovarajućih izlaznih elemenata će biti ažuriran. Posmatrano iz perspektive korisnika, ulazni i izlazni elementi omogućavaju razmenu informacija sa sistemom putem korisničkog interfejsa. Elementi kontejneri pomažu ovoj razmeni organizacijom informacija na interfejsu tako da ih korisnik može pravovremeno opaziti i ispravno interpretirati.



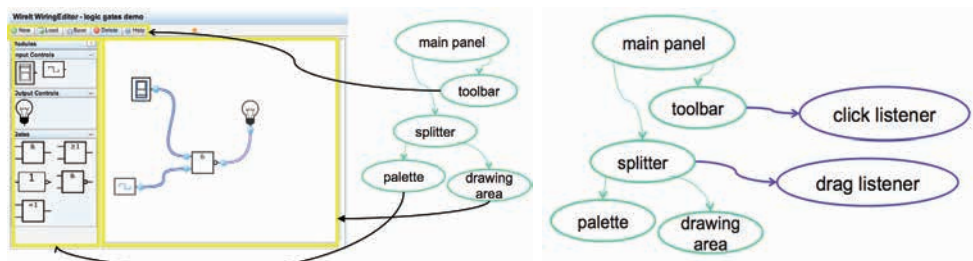
Slika 2.1. Elementi stabla prikaza u GKI. Izvor [23].

2.1.2. Događaji i osluškivači

Mehanizam komunikacije korisnika i korisničkog interfejsa se odvija putem događaja i osluškivača. Poznat je i pod nazivom mehanizam obrade događaja (eng. *event handling*). Jedan ili više osluškivača se pridružuju izvoru događaja. Osluškivač predstavlja logiku (funkciju) koja će se izvršiti kada događaj

nastane nad izvorom događaja (vidljivim delom ili elementom stabla prikaza). Izvor može biti dugme ili tekstualno polje korisničkog interfejsa, dok su odgovarajući događaji klik mišem ili dodir dugmeta i pritisak tastera tastature pri unosu u tekstualno polje, respektivno. Programski objekti događaja sadrže informacije koje mogu biti korisne funkcijama osluškivača, kao što su identifikatora tastera tastature ili dugmeta miša, trenutak nastanka događaja i dvodimenzionalne (x i y) koordinate mesta na ekranu ili prozoru aplikacije na kojem je događaj nastao. Pored toga, jednom izvoru događaja može biti dodeljen veći broj osluškivača i obrnuto.

Oslušivači se uglavnom pridružuju ulaznim elementima GKI. Na taj način se obrada ulaza (eng. *input handling*) korisnika realizuje u okviru osluškivača koji su dodeljeni prikazima. Inače, opisani mehanizam je konkretan slučaj projektnog obrasca pod nazivom Posmatrač (eng. *Observer design pattern*) [24]. Projektni obrasci predstavljaju apstraktna rešenja karakterističnih problema u razvoju softvera koja se mogu implementirati u praksi. Jedan od problema jeste međuzavisnosti softverskih elemenata gde promena jednog elementa utiče na promene drugih elemenata sa kojima je element povezan i obrnuto. Promena može biti stanje (atributi), izgled ili ponašanje elementa. Obrazac definiše da se elementi registruju (eng. *subscribe*) za praćenje ili osluškivanje promena u elementu od interesa i otuda i naziv Posmatrači za te elemente. Često su relacije obostrane, u smislu da povezani elementi prate promene međusobno. U kontekstu GKI, funkcije osluškivača se registruju kod izvora događaja koji zauzvrat kreira programske objekte događaja koji se prosleđuju svim registrovanim funkcijama koje dalje obrađuju događaj, tj. izvršavaju se. Više informacija u vezi sa ovim i drugim obrascima se može naći u [24].



Slika 2.2. Oslušivači događaja se dodeljuju čvorovima - vidljivim elementima stabla prikaza GKI koji predstavljaju izvore događaja. Dugmad imaju osluškivač registrovan za klik događaj, dok graničnik (eng. *splitter*) generiše i registrovanom osluškivaču prosleđuje događaj prevlačenja (eng. *drag*). Izvor [23].

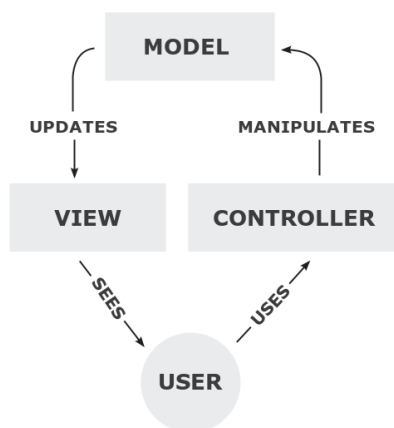
2.1.3. Model-prikaz-kontroler obrazac

Model-prikaz-kontroler obrazac predstavlja generalizaciju projektnog obrasca Posmatrač. Može se posmatrati i kao nadogradnja stabla prikaza i događaja i osluškivača. Obrazac je formulisan 1979. godine u Xerox PARC laboratoriji tokom razvoja personalnog računara, odnosno korisničkog interfejsa operativnog sistema [25, 26].

Obrazac predviđa raspodelu odgovornosti (eng. *separation of responsibilities*) elemenata GKI [27]. Zadatak obrade ulaza korisnika dodeljuje ulaznim elementima i osluškivačima. Konkretno, ulazni elementi GKI obezbeđuju generisanje ulaznih podataka korisnika čiji se sadržaj putem događaja prosleđuje osluškivačima na obradu. Za izlaz je zaduženo celokupno stablo prikaza. Potrebno je još predstaviti i opisati stanje, odnosno podatke korisničkog interfejsa. Ovde treba napraviti razliku između podataka pozadinske logike (eng. *backend*) koji se prosleđuju korisničkom interfejsu i podataka korisničkog interfejsa. U širem kontekstu, model je zajednički naziv za obe vrste podataka koji zajedno čine stanje (eng. *state*) aplikacije. Međutim, može se posmatrati i kao skup podataka kojima se upravlja na nivou korisničkog interfejsa, kao što su događaji, sadržaji elemenata stabla prikaza i vrednosti internih promenljivih i atributa. Udžbenik razmatra model na nivou GKI.

U tom pogledu model čuva i održava podatke GKI, i obezbeđuje pristup podacima. Prikaz je odgovoran za izlaz, odnosno prezentaciju podataka modela korisniku i unos podataka korisnika. Promene modela se reflektuju na promene prikaza. Kontroler ima dve glavne odgovornosti - prijem događaja kao posledica unosa podataka korisnika i upravljanje modelom.

U praksi se gore navedene odgovornosti mogu i drugačije rasporediti između elemenata, u zavisnosti od konkretne tehnologije ili funkcionalnih zahteva. Slika 2.3 ilustruje uloge modela, prikaza i kontrolera, kao i njihove međusobne relacije.

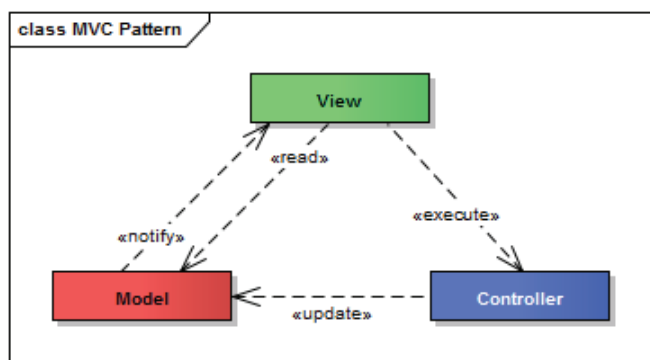


Slika 2.3. Raspodela odgovornosti elemenata model-prikaz-kontroler obrasca. Izvor: RegisFrey - Own work, Public Domain, <https://commons.wikimedia.org/w/index.php?curid=10298177>.

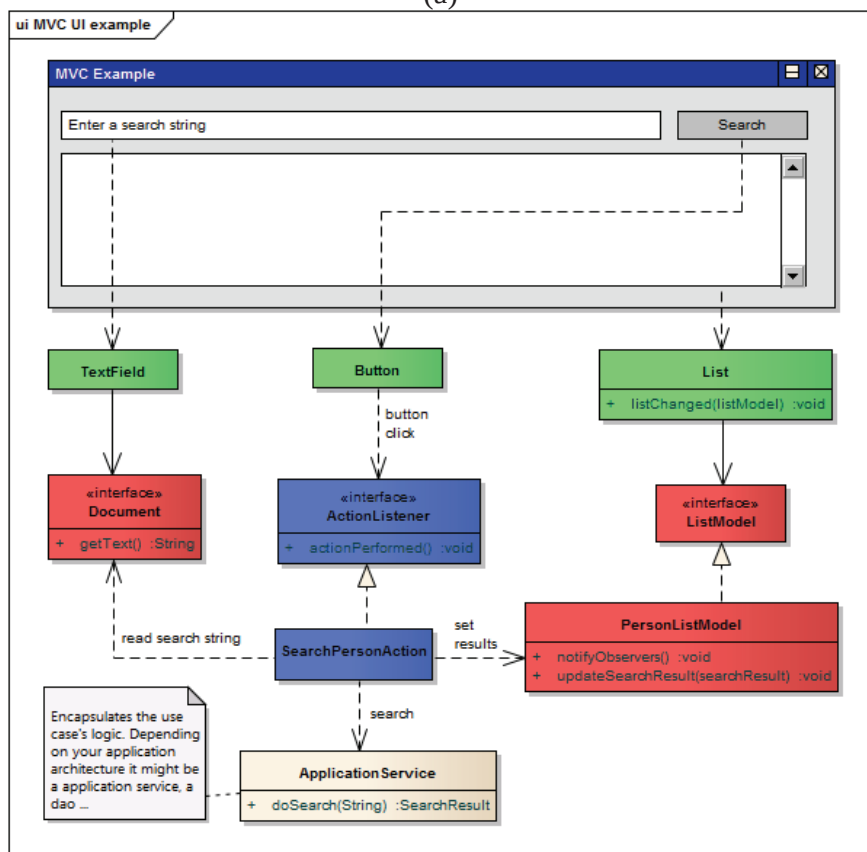
Raspodela odgovornosti odvojenim elementima GKI ima nekoliko prednosti. Pre svega, različiti prikazi mogu prezentovati iste podatke (modele) interfejsa. Na primer, podaci entiteta aplikacije (pregled proizvoda na osnovu zadatih filtera/kriterijuma) u jednom delu korisničkog interfejsa mogu biti prikazani tako da se ne mogu menjati, dok se u drugom delu interfejsa mogu menjati (forma za ažuriranje ili uklanjanje proizvoda). Pored toga, model i prikaz se mogu nezavisno menjati što olakšava razvoj i održavanje, tj. izmene GKI u toku korišćenja. Ponovna upotrebljivost (eng. *reusability*) se ogleda u tome da se prikazi i modeli kao odvojeni elementi mogu koristiti u različitim aplikacijama. Vođene ovim principom, pojavile su se biblioteke komponenti (eng. *toolkit*) za razvoj korisničkih interfejsa u različitim programskim jezicima.

Komponenta `TextField` [28], Java Swing generička implementacija tekstualnog polja, predstavlja jednostavniji primer obrasca. Model predstavlja sadržaj polja kao niz karaktera. Prikaz je pravougaoni, oivičen prostor koji prezentuje tekst. Kontroler je funkcija osluškivača koja registruje unos korisnika putem tastature (eng. *keystroke*) preko odgovarajućeg tipa događaja i obrađuje tekst.

Obrazac je po prirodi skalabilan i ima i širu upotrebu, na nivou dela ili celog GKI. Gore opisano tekstualno polje može biti deo prikaza (dijalog za pretragu podataka aplikacije) sa specifičnim kontrolerom (osluškivač koji registruje sekvencu događaja unosa korisnika i vrši tekstualnu pretraga podataka aplikacije) i prilagođenim modelom (dokumenti kao predmet pretrage). Primer takve implementacije obrasca je prikazan na Slici 2.4.



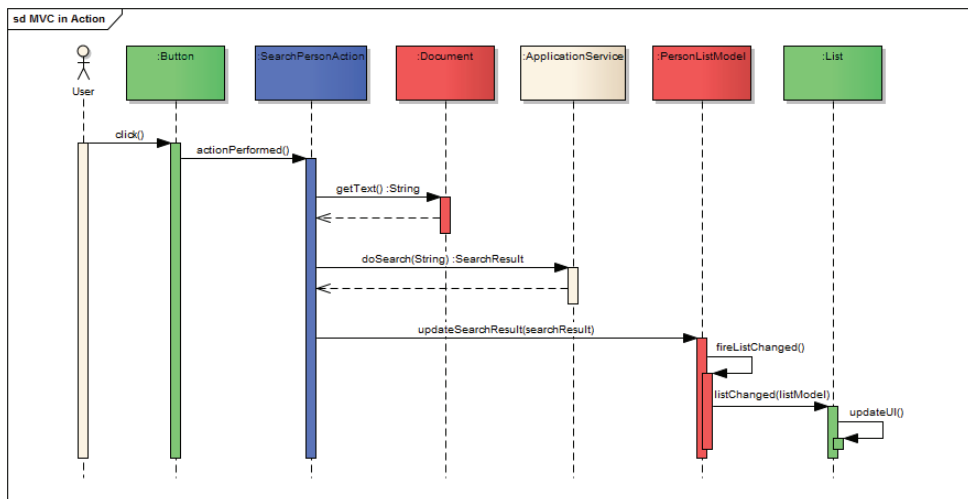
(a)



(b)

Slika 2.4. Model-prikaz-kontroler (a) šema i (b) implementacija u Java Swing biblioteci za GKI. Korisnik: 1) unosi kriterijum za pretragu (model) u tekstualno polje (prikaz); 2) aktivira pretragu klikom na dugme (prikaz) koje generiše događaj i inicira funkcije pretrage u osluškivaču (kontroler); i 3) interfejs prikazuje rezultat pretrage (model) u višelinijnskom tekstualnom polju (prikaz). Izvor [29].

Dijagram sekvence (eng. *sequence diagram*) na Slici 2.5 daje hronološki prikaz interakcije korisnika sa dijalogom za pretragu.



Slika 2.5. Vremenski redosled izvršavanja elemenata model-prikaz-kontroler obrasca u dijalogu za pretragu. Izvor [29].

Opšte posmatrano, obrazac razdvaja prezentacionu (eng. *look*) i interaktivnu (eng. *feel*) logiku na nivou korisničkog interfejsa. Prezentacionu logiku čine ulaz korisnika (kontroler) i izlaz ka korisniku (prikaz). Interaktivna logika se sastoji od funkcija korisničkog interfejsa poput čuvanja i pristupa podacima i notifikacije putem događaja u slučaju promena podataka (model), i dvosmerne sinhronizacije prikaza i modela u slučaju njihovih promena (kontroler).

Kod GKI ulaz i izlaz se ne mogu u potpunosti posmatrati nezavisno. Na primer, postojanje kontrolera kao pozadinske logike interfejsa mora biti vidljivo, tj. intuitivno iz izgleda prikaza. Vizuelni tragovi (eng. *visual cues*) mogu biti naziv dugmeta, simbol dugmeta ikone i ručice *scrollbar* elementa ili klizača. Nakon što korisnik interaguje sa prikazom, kao naznaka pokretanja i izvršenja kontrolera u pozadini prikaz mora promeniti izgled, tj. dati korisniku vizuelnu povratnu informaciju (eng. *feedback*). Na primer, promena boje dugmeta kada korisnik izabere dugme.

Sa obzirom da su ulaz i izlaz GKI u logičkoj sprezi, nekada nije praktično razdvojiti ih fizički, kao odvojene datoteke ili klase. Zbog toga su prikaz i kontroler često objedinjeni u jedan element tako da prikaz integriše kontroler i upravlja ulazom i izlazom. Primer su komponente ili vidžeti (eng. *widget*) Java Swing biblioteke.

U nekim slučajevima, kontroler se ponaša kao posrednik (eng. *mediator*) između modela i prikaza. Tada prikaz registruje i generiše događaje. Kontroler

osluškuje promene u modelu i prikazu i prosleđuje ih, tj. sinhronizuje dva elementa.

Martin Fauler je proširio i primenio obrazac na arhitekturu softverske aplikacije [30]. Varijante obrasca opisuju različite arhitekture softverskog sistema po vertikali kroz kombinacije sloja skladištenja (baza ili izvor podataka) kao modela, sloja poslovne logike (eng. *backend*) kao kontrolera, i sloja korisničkog interfejsa (eng. *frontend*) kao prikaza.

2.2. Pristupi programiranju interfejsa

U zavisnosti od načina zapisivanja naredbi i njihove reprezentacije, tj. značenja, postoje dva osnova pristupa u programiranju GKI - *proceduralni* i *deklarativni* [31].

Proceduralni pristup je zasnovan na metafori proceduralnog programiranja, odnosno programskom kodu koji sadrži naredbe koje govore *kako* se kreira i koje su funkcije stabla prikaza, tj. elemenata korisničkog interfejsa. Program je sekvencijalan skup naredbi u definisanom redosledu tako da postoji tok kontrole izvršavanja. Na primer, registracija (eng. *import*) predefinisanih objekata i funkcija, kreiranje elemenata prikaza, definisanje vrednosti atributa kreiranih prikaza, kreiranje kontrolera (osluškivača i drugih funkcija) i povezivanje kreiranih objekata u stablo prikaza.

Kod *deklarativnog* pristupa programer direktno navodi strukturu i sadržaj stabla prikaza, odnosno definiše *šta* stablo treba da sadrži, tj. koje elemente prikaza. Ovaj pristup uglavnom podržavaju jezici označavanja (eng. *markup languages*) kao što je HTML (eng. *HyperText Markup Language*). Struktura koda programa predstavlja strukturu stabla prikaza i ne postoji, odnosno u ovom pristupu nema potrebe za tokom kontrole programa. Prevodilac jezika označavanja transformiše deklarativni sadržaj programa u izvršno (eng. *runtime*) stablo prikaza.

HTML jezik (Slika 2.6 levo) omogućava deklarativan način definisanje strukture i sadržaja stabla prikaza. Elementi stabla prikaza se navode kao tagovi (eng. *tag*). Tag definiše tip elementa (*div*, *textarea*, *button*) koji ima pridružene attribute (*id* kao jedinstven identifikator elementa) i sadržaj (između otvarajuće i zatvarajuće oznake taga). Veb pretraživač kreira stablo prikaza na osnovu HTML deklaracija, tj. tagova. HTML parser sekvencijalno analizira tag po tag i kreira stablo elemenata kao rezultat.

JavaScript jezik (Slika 2.6 desno) kreira identično stablo prikaza proceduralno, tako što konstruiše DOM (eng. *Document Object Model*) strukturu. DOM predstavljala stablo prikaza i obezbeđuje i skup objekata i metoda za rad sa istim. Struktura kao takva postoji u različitim programskim jezicima.

```

<div>Deklarativni GKI</div>
<div>
<textarea></textarea>
</div>
<div>
<button
id="sendButton">Pošalji</button>
</div>

```

Deklarativni GKI

Pošalji

```

var doc = document;
var div1 =
doc.createElement("div");
div1.appendChild(doc.createTextNode(
"Proceduralni GKI"));
var div2 =
doc.createElement("div");
var textElement =
document.createElement("textarea");
div2.appendChild(textElement);
var div3 =
doc.createElement("div");
var textButton =
doc.createElement("button");
textButton.setAttribute("id", "sendB
utton");
textButton.appendChild(doc.createTe
xtNode("Pošalji"));
div3.appendChild(textButton);
doc.body.appendChild(div1);
doc.body.appendChild(div2);

doc.body.appendChild(div3);

```

Proceduralni GKI

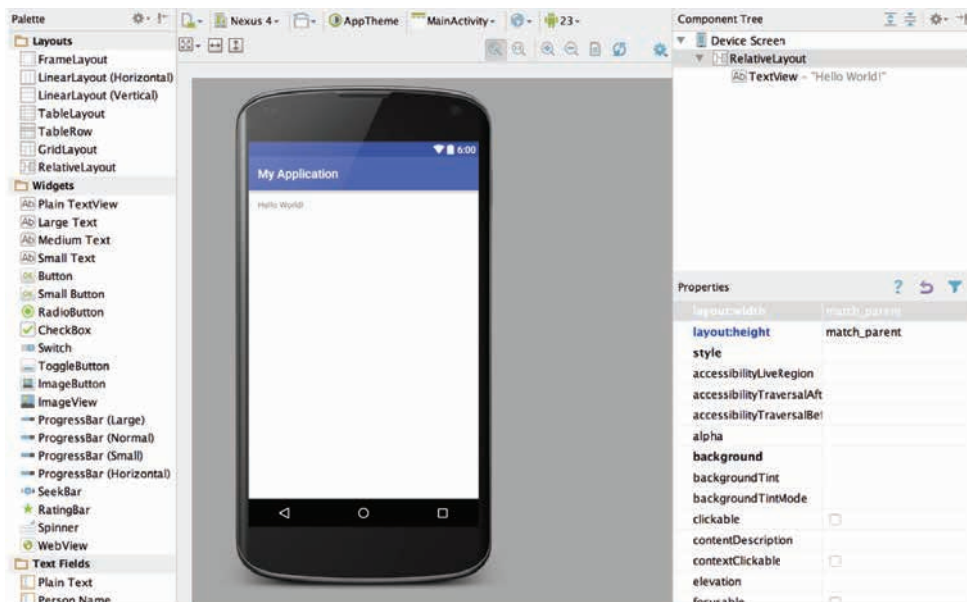
Pošalji

Slika 2.6. Uporedni prikaz deklarativnog pristupa u jeziku HTML (levo), i proceduralnog pristupa u jeziku JavaScript (desno) u kreiranju identičnog prikaza GKI.

Deklarativne naredbe su apstraktnije (govore šta program radi), i samim tim kompaktnije i čitljivije. Sa druge strane, one ne omogućavaju kontrolu toka programa kao kod proceduralnog pristupa.

Pristupi po svojoj prirodi ne mogu samostalno realizovati kompletnu arhitekturu GKI tako da se kombinuju u praksi.

Pojedini alati koriste deklarativni pristup za kreiranje korisničkog interfejsa direktnom manipulacijom (eng. *direct manipulation*), tj. prevlačenjem i podešavanjem elemenata korisničkog interfejsa iz predefinisane palete (Slika 2.7). Deklarativna specifikacija se zatim u pozadini prevodi u proceduralni kod koji se dalje doraduje ručno. Na ovaj način se dva pristupa mogu kombinovati u praksi.



Slika 2.7. Primer deklarativne specifikacije korisničkog interfejsa pomoću direktne manipulacije.

Drugi način kombinovanja je prisutan kod biblioteka (eng. *library*) i okvira (eng. *framework*) za dizajn i programiranje korisničkih interfejsa. U deklarativnom delu se koriste i proceduralne naredbe. Na ovaj način se kreiraju deklarativne konekcije prikaza i modela, i ne moraju se dodavati osluškivači. U primeru na Slici 2.8, tekstualno polje prikaza (*input*) se povezuje sa varijablom modela (*user_name*). Na ovaj način se uspostavlja obostrana sinhronizacija - kada korisnik promeni sadržaj tekstualnog polja varijabla će primiti tu vrednost i obrnuto, ako se vrednost varijable promeni programski, sadržaj tekstualnog polja će biti ažurirana vrednost. Mehanizam je poznat pod nazivom povezivanje podataka (eng. *data binding*) i opisan je u Sekciji 5.3.3. Paragraf (p) u nastavku prikazuje vrednost varijable modela u prikazu (sintaksa `{{...}}`).

```
<input type="text" [(ng-model)]="user_name">
<p>Hello, {{user_name}}!</p>
```

Slika 2.8. Kombinacija deklarativnog i proceduralnog pristupa u Angular okviru za dizajn i programiranje korisničkih interfejsa.

2.3. Obrasci arhitekture interfejsa

Korišćenje mobilnih uređaja prevazilazi stone uređaje, 59,81% u Novembru 2022. godine [32]. Pored toga, Veb saobraćaj ostvaren korišćenjem mobilnih uređaja iznosi 58,99% (Jun, 2022) na globalnom nivou [33]. Korisnici aplikacije na mobilnim uređajima koriste uglavnom putem GKI. Stoga se obrasci arhitekture GKI u praksi razmatraju u kontekstu mobilnih uređaja.

Obrasci arhitekture se razmatraju sa aspekta opšte raspodele i organizacije informacija u GKI [34] i pojavljuju se kao:

- Hijerarhija,
- Centralizovana,
- Ugnježdjena,
- Kartična,
- Komandna tabla, i
- Filter.

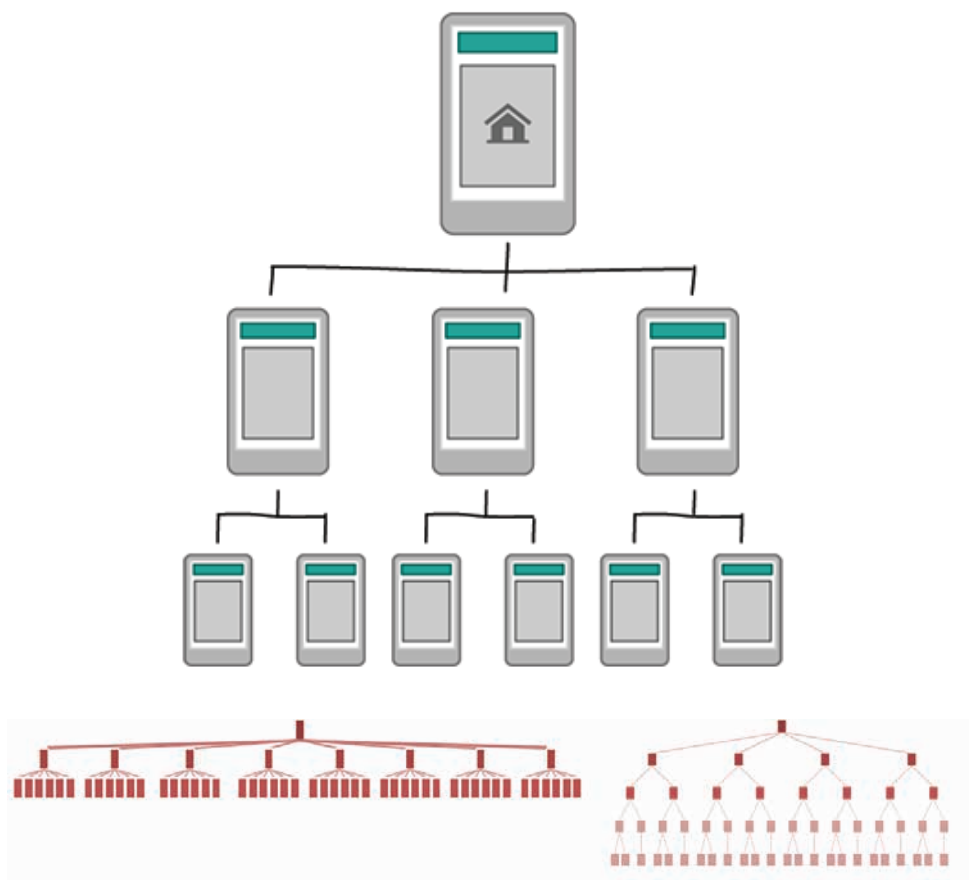
U nastavku se opisuju navedeni obrasci sa napomenom da se u praksi najčešće kombinuje više obrazaca.

2.3.1. Hijerarhija

Hijerarhija (eng. *Hierarchy*) obrazac ima strukturu stabla, sa glavnom (index) strana na vrhu i skupom podređenih strana koje omogućavaju navigaciju po širini (strane na istom nivou) i dubini (strane na različitim nivoima). U praksi se sreću dva tipa hijerarhije - duboka (eng. *deep*) i ravna ili plitka (eng. *flat*) [35].

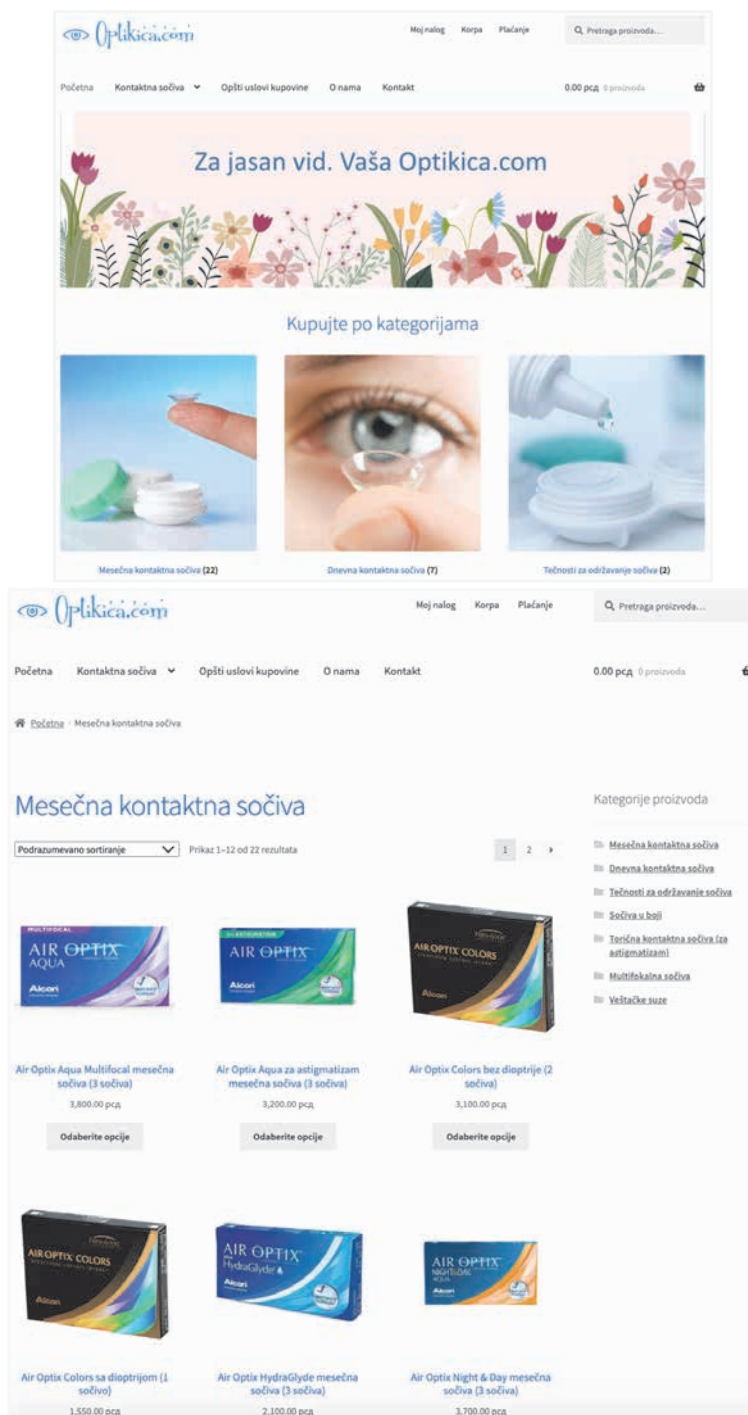
Glavna prednost jeste mogućnost organizacije velike količine raznovrsnih informacija GKI. Sa druge strane, navigacija, tj. kretanje kroz interfejs može biti otežano (ili usporeno) ukoliko je struktura stabla preduboka (duboka hijerarhija) ili preširoka (ravna hijerarhija)

Ravne hijerarhije su pogodne za strane čiji sadržaji predstavljaju različite kategorije koje se kao takve mogu lako prepoznati. Međutim, ako postoji veliki broj različitih kategorija onda bi trebalo razmotriti povećanje dubine hijerarhije tako što će se identifikovati podgrupe kategorija koje dele neke zajedničke karakteristike, odnosno pripadaju istom kontekstu. Uočeni konteksti će tada biti strane-kategorije koje sadrže konkretne kategorije kao neposrednu decu stabla.



Slika 2.9. Hijerarhijska arhitektura GKI (gore), izvor [34], i tipovi hijerarhije (ravna dole levo, duboka dole desno), izvor [35].

U nastavku je dat primer sajta sa ravnom hijerarhijom.

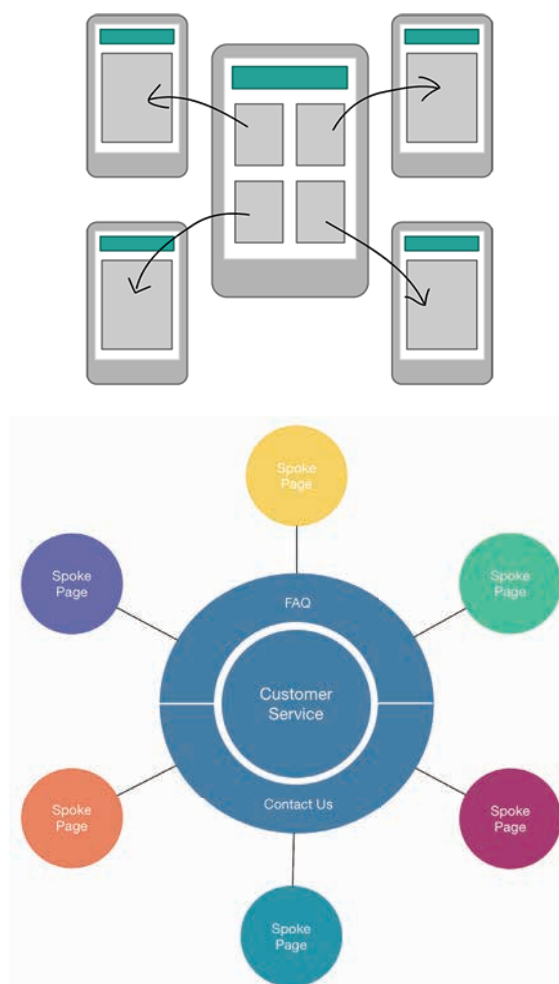


Slika 2.10. Primer ravne hijerarhije - prvi nivo (gore) prikazuje kategorije proizvoda, drugi nivo (dole) nudi proizvode izabrane kategorije. Izvor: <https://optikica.com/>.

2.3.2. Centralizovani obrazac

Centralizovani obrazac je poznat i pod nazivom *hub and spoke* [36]. Obrazac ima centralnu stranu (eng. *hub*) sa koje se može doći na ostale strane (eng. *spoke*) raspoređene radijalno. Pogodan je za multifunkcionalne alate, gde svaki alat ima zasebnu namenu i navigaciju unutar alata. U slučaju obavljanja više zadataka u isto ili približno vreme (eng. *multitasking*) prelazak na novi zadatak znači kretanje preko centralne strane.

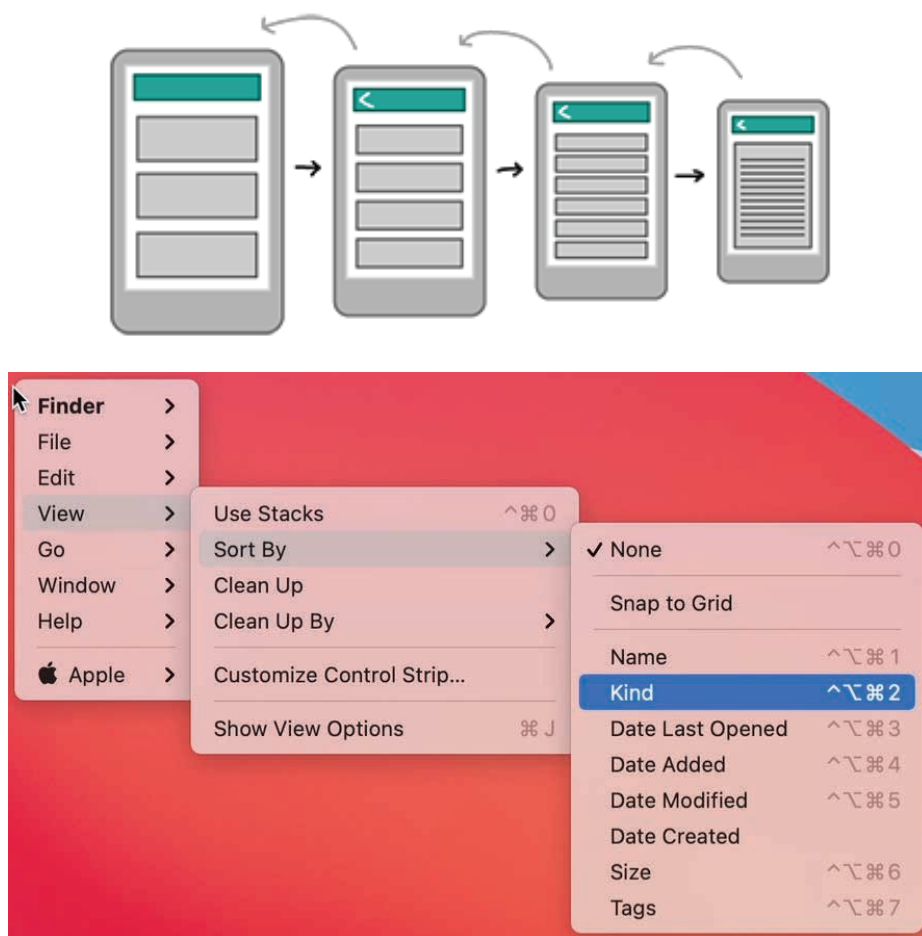
Arhitektura je pogodna za klijent-servis (eng. *customer-service*) sadržaje gde centralna strana sumira dostupne servise i sadrži direktne linkove ka stranama sa detaljima servisa. U praksi su često i strane sa detaljima međusobno povezane radi brže navigacije između strana.



Slika 2.11. Centralizovana arhitektura GKI (gore) i primena u klijent-servis scenariju (dole). Izvor [34].

2.3.3. Ugnježdjeni obrazac

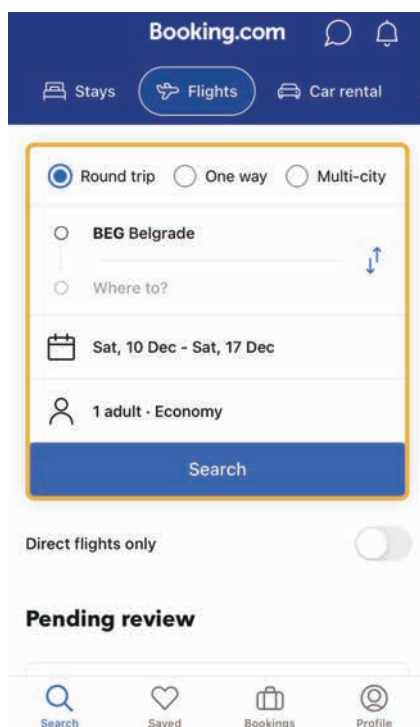
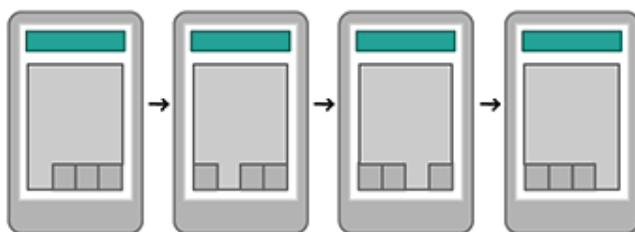
Ugnježdjena arhitektura raspoređuje informacije tako da se korisnik postupno usmerava ka sadržajima sa više detalja. Korisniku obezbeđuje doživljaj orijentacije u strukturi sadržaja korisničkog interfejsa usled percepcije kretanja napred i nazad. Kao takva pogodna je za GKI sa jednom ili nekoliko sličnih tema sadržaja. Može se koristiti i za struktuiranje sekcija u okviru roditeljskih (nadređenih) arhitektura kao što su centralizovana ili hijerarhija. Često se koriste hijerarhijski meniji (eng. *hierarchical menus*). Obrazac može usporiti navigaciju između sekcija sa drugačijim temama/sadržajima.



Slika 2.12. Ugnježdjena arhitektura GKI (gore), izvor [34], i hijerarhijski meni kao često korišćen element za navigaciju (dole).

2.3.4. Kartični obrazac

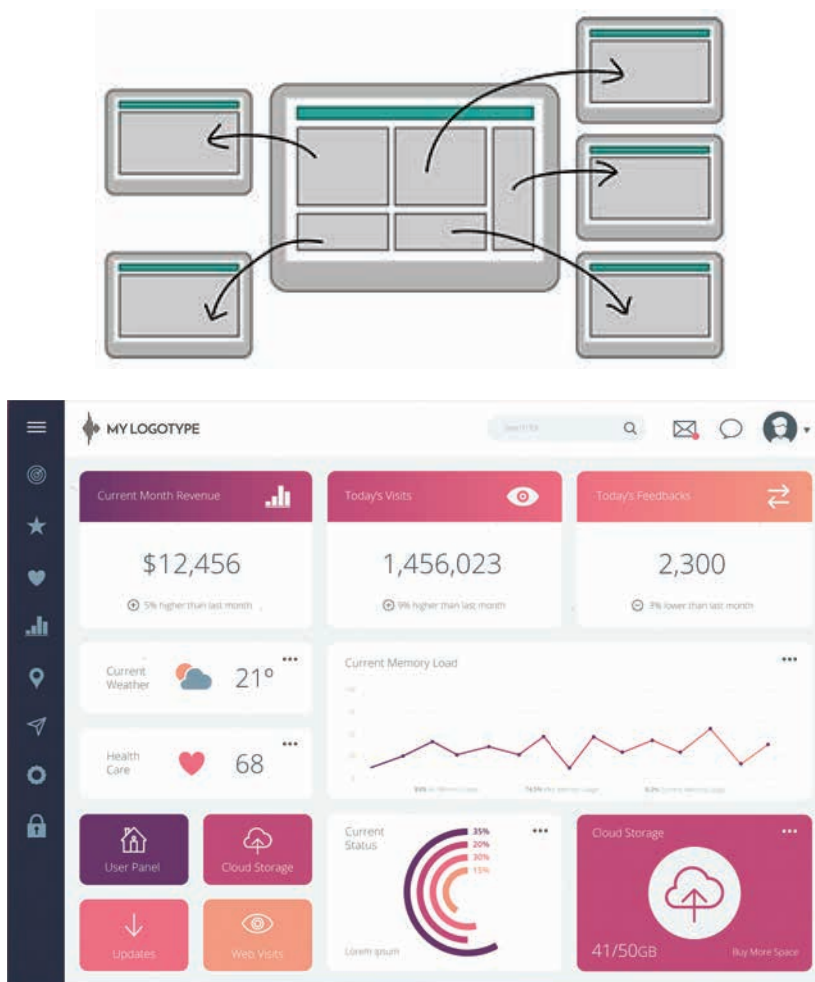
Kartična arhitektura organizuje kolekciju sekcija korišćenjem linije alata (eng. *toolbar*) ili tab elemenata (eng. *tab*). Korisniku daje kompletniju predstavu o celokupnom sadržaju aplikacije i bržu promenu strana. Pogodna je za aplikacije sa sadržajima ili funkcijama koji su na neki način povezani, i za *multitasking* korišćenje. Česta je kod mobilnih telefona. Povećanje kompleksnosti u smislu raznovrsnosti sadržaja aplikacije iziskuje povećanje broja alata ili tab elemenata što može otežati korišćenje GKI. Tab elementi se koriste za sadržaje koji pripadaju istom kontekstu i preporuka je imati maksimalno 4 do 5 tab elemenata [37].



Slika 2.13. Kartična arhitektura GKI (gore), izvor [34], i primer linije alata (dole), izvor: <https://www.booking.com>.

2.3.5. Komandna tabla

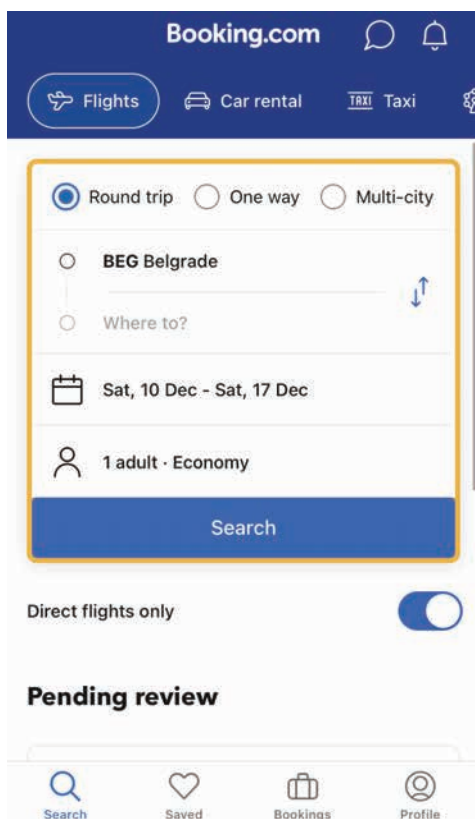
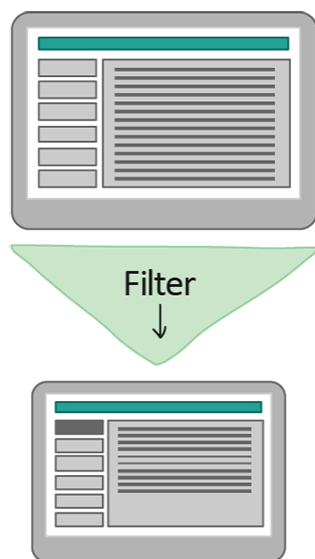
Kod obrasca komandna tabla glavna strana ima bogatiji i detaljniji sadržaj. Ona sadrži sažetke prikaza ostalih strana interfejsa kao zasebne komponente GKI. Pogodna je za tablet uređaje zbog većih dimenzija interfejsa. Glavna prednost je što korisnik na osnovu izgleda glavne strane brzo stiče predstavu o čitavom sadržaju aplikacije. Ovo istovremeno zahteva pažljiv dizajn sadržaja glavne strane. Obrazac pogoduje interfejsu aplikacija alata i sadržaja koji su slični ili imaju zajednički kontekst.



Slika 2.14. Arhitektura GKI tipa Komandna tabla (gore), izvor [34], i primer komandne table (dole), izvor: https://www.freepik.com/free-vector/gradient-elements-collection-dashboard-template_5500787.htm Image by pikisuperstar on Freepik.

2.3.6. Filter

Filter obrazac omogućava kretanje kroz sadržaj aplikacije kao kroz skupove selektovanih podataka zadavanjem kriterijuma (filtera). Strukture strana su identične, sa podacima istog formata ali različitog sadržaja u zavisnosti od zadatih vrednosti filtera. Obrazac je pogodan za GKI aplikacija sa velikom količinom podataka (tzv. *data-driven*) kao što su članci, slike, video materijali, tekstualni i numerički podaci. U praksi se često kombinuje sa drugim obrascima. Na primer, kao detaljna pretraga sa glavne strane kartičnog obrasca, korene ili ugnježdene strane hijerarhije, ili *hub* strane centralizovane arhitekture. Složenost i količina podataka koji se filtriraju utiču na prostor potreban za prikaz kriterijuma (kategorija) i vrednosti filtera, kao i sadržaj rezultata filtriranja.



Slika 2.15. Filter arhitektura GKI (gore), izvor [34], i primer filtera za pretragu (dole), izvor: <https://www.booking.com>.

2.4. Ulaz interfejsa

Interakcija sa GKI počinje tako što korisnik inicira akciju nad interfejsom (npr. selekcija dugmeta ili stavke menija). Akcija se najpre transformiše u programsku reprezentaciju događaja, zatim se događaj prosleđuje kroz stablo prikaza i dodeljuje odgovarajućim obrađivačima ili osluškivačima, i sprovode se promene kao rezultat izvršavanja osluškivača i vidljive kroz izlaz GKI. U skladu sa prethodnim, svaka akcija se postupno obrađuje ulaznim mehanizmima GKI [23] u koje spadaju:

- Ulazni događaji (eng. *input events*), i
- Raspoređivanje i prenošenje događaja (eng. *event dispatch and propagation*).

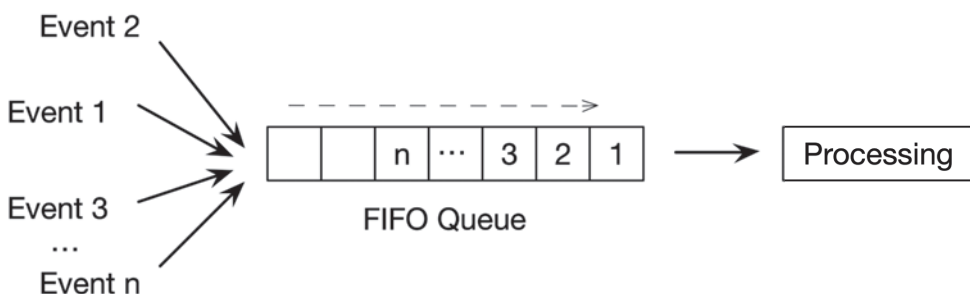
Ulazni događaji su programski objekti koji se generišu kao posledica fizičkih akcija korisnika nad GKI. Skupu mogućih akcija nad elementima GKI su pridruženi odgovarajući logički objekti događaja u programu (Tabela 2.1).

Tabela 2.1. Vrste događaja u GKI.

Ulazni događaj (naziv)	JavaScript događaj (ime)
Pritisnut taster tastature (<i>key pressed</i>)	keydown
Otpušten taster tastature (<i>key released</i>)	keyup
Pomeraj miša (<i>mouse moved</i>)	mousemove
Pritisnuto dugme miša (<i>button pressed</i>)	mousedown
Otpušteno dugme miša (<i>button released</i>)	mouseup
Klik dugmeta miša (<i>click</i>)	click
Dupli klik dugmeta miša (<i>double click</i>)	dblclick
Pritisnut karakter tastature (<i>character down</i>)	keypress
Promena vrednosti elementa forme (<i>input</i>)	input
Pozicioniranje miša u region objekta (<i>bounding box</i>)	mouseenter
Pozicioniranje miša van regiona objekta	mouseleave
Dodir (spuštanje prsta)	touchstart
Kraj dodira (podizanje prsta)	touchend
Pomeranje tačke dodira	touchmove

Ulazni događaj sadrže informacije o stanju hardverskog uređaja koji je generisao događaj. Informacije opisuju identifikator uređaja (dugme miša ili taster tastature), stanje dugmeta (pritisnuto, otpušteno), pozicija miša u dvodimenzionalnom prostoru ekrana i vreme nastanka događaja (eng. *timestamp*).

Sa obzirom da se događaji generišu u kontinuitetu potrebno ih je skladištiti kako bi svaki od njih bio procesuiran. Red događaja (eng. *event queue*) je struktura podataka koja se koristi za skladištenje. Princip pristupa i obrade skladištenih događaja sa odvija po principu prvi-smešten-prvi-opslužen (eng. *first-come-first-served* - *FIFO*).



Slika 2.16. Red događaja skladišti događaje po redosledu njihovog pojavljivanja. Po istom redosledu događaje procesiraju osluškivači. Izvor: <https://circuitcellar.com/resources/event-queues/>.

Tipičan scenario korišćenja GKI se može posmatrati kao “*period razmišljanja ... period intenzivnog unosa*”. Zbog toga se generisanje ulaznih događaja često odvija ubrzano kao rezultat korišćenja interfejsa od strane jednog ili grupe korisnika. Red događaja predstavlja memorijski bafer (eng. *buffer*) između prezentacione i interaktivne logike GKI i olakšava procesiranje događaja.

Jedna od tehnika optimizacije jeste savravanje većeg broja događaja u jedan događaj (eng. *coalesce*). Događaji koji se mogu savrniti su oni čije pojavljivanje prati određen obrazac. Na primer, pokreti miša kao što su pritisak dugmeta, pomeranje i otpuštanje dugmeta kod prevlačenja objekata se mogu savrniti kao jedan događaj predstavljen poslednjim poznatim stanjem. Sa druge strane, kod crtanja oblika u slobodnoj formi mišem događaji se ne mogu savrnjavati zbog toga što forma ne bi bila očuvana već prikazana kao povezani pravolinijski segmenti.

Petlja događaja (eng. *event loop*) je proces koji čita događaje iz reda događaja i raspoređuje ih odgovarajućim čvorovima stabla prikaza, tj. osluškivačima događaja. Petlja je interni proces modernih tehnologija za razvoj GKI. Svaki

GKI ima svoju instancu petlje. Raspoređivanje događaja predstavlja izbor elemenata GKI za obradu događaja. Događaj miša se prvo prosleđuje elementu iznad kojeg se nalazi pokazivač miša u trenutku nastanka događaja. Ukoliko taj element nema pridružen osluškivač za obradu datog događaja, tada se događaj prenosi dalje kod stabla prikaza dok se ne nađe na element čiji je osluškivač registrovan za obadu tipa prenešenog događaja. Tada se događaj obrađuje. Po definiciji i u većini tehnologija, događaj se najpre kreće ka roditeljskim čvorovima stabla prikaza, uz hijerarhiju sve dok se ne dođe do korenog elementa stabla. Ukoliko se ne nađe osluškivač, događaj se odbacuje, tj. dealocira iz memorije programa. Nakon obrade, kod nekih tehnologija se događaj i dalje prosleđuje ka vrhu stabla i može se eksplicitno zaustaviti iz programa GKI (JavaScript), dok se kod drugih dalje ne prosleđuje (Java Swing).

U slučaju Veb tehnologija za GKI, događaji su bili prosleđivani na dva načina - ka vrhu hijerarhije ili po dubini, u zavisnosti od vrste Veb pretraživača (eng. *Web browser*). W3C (eng. *World Wide Web Consortium*) kao preporuku kombinuje oba načina. Događaj se najpre prosleđuje po dubini i ova faza je poznata pod nazivom *event capture*. Zatim se kreće ka vrhu stabla prikaza, takozvani *event bubble*. Većina Veb pretraživača podržava preporuku i omogućava programsko definisanje načina prosleđivanja u funkcijama osluškivača događaja.

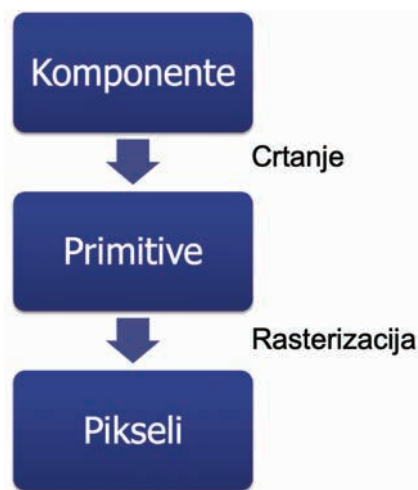
2.5. Izlaz interfejsa

Reprezentacija izlaza GKI se može posmatrati na tri nivoa [23]:

- Komponente (eng. *components*),
- Primitive (eng. *primitives*), i
- Pikseli (eng. *pixels*).

Komponente su elementi stabla prikaza, tj. ulazni, izlazni i kontejnerski elementi GKI. Grafičke primitive su geometrijski objekti od kojih su komponente sačinjene. Na primer, linija (eng. *line*), kriva (eng. *curve*), pravougaonik (eng. *rectangle*), luk (eng. *arc*) i tekst (eng. *text*). Pikseli se posmatraju kao memorija ekrana, odnosno dvodimenzionalni niz tačaka koji prikazuje sadržaj GKI.

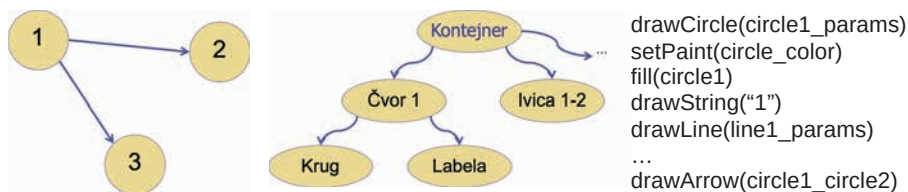
Svaki GKI sadrži navedene nivoe reprezentacije. Komponente se definišu u fazi dizajna i implementacije GKI. Zajedno čine stablo prikaza. Svaka komponenta se iscrtava pomoću skupa geometrijskih primitiva i njene unutrašnjost ispunjava odgovarajućom bojom. Kreiranje i iscrtavanje omogućavaju tehnologije ili biblioteke programskih jezika za izradu GKI. Posebna grafička biblioteka (eng. *graphical library*) konvertuje primitive u matricu piksela koja će biti prikazana na ekranu.



Slika 2.17. Transformacije između nivoa prikaza GKI. Komponente GKI se sastoje od geometrijskih primitiva koje se prikazuju kao dvodimenzionalni niz piksela na ekranu.

Svaki čvor i ivica stabla prikaza predstavljaju zasebne objekte. Čvorovi mogu sadržavati druge čvorove (kao kontejnerski elementi) ili biti ulazni i izlazni elementi GKI. Svaki element list je sastavljen od dece grafičkih primitiva (Slika 2.17). Na primer, dugme sadrži pravougaonik i tekst, dok se ikona sastoji od pravougaonika i simbola (koji opet ima granicu i unutrašnjost). Celo stablo je zapravo kreirano iscrtavanjem grafičkih primitiva (`drawCircle`, `drawText`, `drawRectangle` i druge). Na kraju se GKI prikazuje na ekranu kao rezultat procesa rasterizacije koji preslikava grafičke primitive u matricu piksela.

Za primitive se kaže da su vektorski objekti zbog toga što se definišu matematičkim formulama. U računarskoj grafici vektorski objekat se sastoji od granice (eng. *stroke*) i unutrašnjosti (eng. *fill*). Elementi GKI koji se kreiraju i koriste u programu predstavljaju vektorske objekte. Zbog načina funkcionisanja prikaznih tehnologija, tj. monitora oni se transformišu u rasterske objekte kako bi bili prikazani. Postoje i rasterski objekti koji se sastoje od piksela, gde svaki piksel sadrži informacije poput boje, dubine i druge. Slike formata kao što su .JPG i .PNG predstavljaju rasterske objekte. Više informacija u vezi sa primitivima, vektorskim i rasterskim objektima i rasterizaciji se može naći u [38].



Slika 2.18. Ilustracija iscrtavanja elemenata stabla prikaza (levo) kao skupa primitiva (sredina) i naredbi za crtanje primitiva (desno).

Iscrtavanje stabla prikaza kreće od vrha, tj. korenog elementa, ka dubini stabla i prikazima listovima. (eng. *top-down*). Svaki objekat se iscrtava, a zatim i njegova deca ukoliko ih ima po istom principu, odnosno rekurzivno.

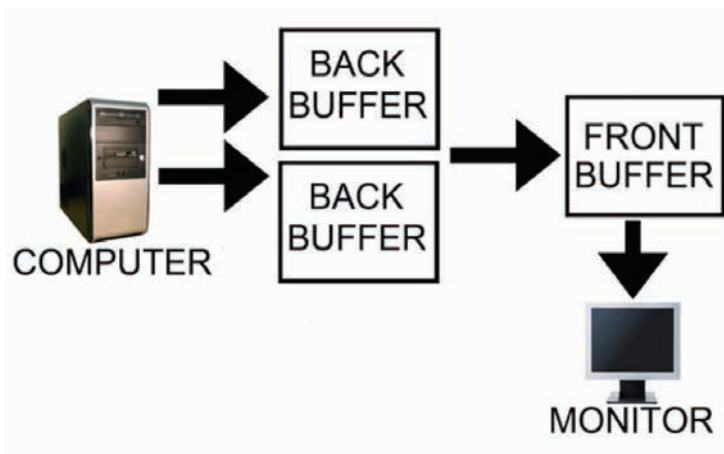
Maska (eng. *clipping area*) određuje deo stabla koji će biti iscrtan, tj. prikazan. Prikazi obuhvaćeni maskom se iscrtavaju, dok se prikazi koji nisu obuhvaćeni maskom ne iscrtavaju. Pravougaonog je oblika (eng. *bounding box*) i najčešće je određena veličinom i pozicijom prozora pretraživača ili aplikacije.

Objekti u GKI se mogu preklapati te je potrebno definisati koji objekti će biti prikazani u prvom planu, a koji neće biti vidljivi. Uvodi se treća koordinata, z, kao vrednost koja se pridružuje svakom pikselu i govori o redosledu datog piksela. Preklapajući objekti, tj. njihovi pikseli imaju identične pozicije pa se na osnovu ove vrednosti zna čiji će piksel biti prikazan i preklapati piksele “dubljih” objekata. Ovaj princip se naziva i z-redosled (eng. *z-order*). Z-osa kao takva ne postoji, vrednost se koristi za određivanje redosleda prikaza objekata na ekranu u slučaju preklapanja. Podrazumevani redosled kod stabla prikaza je da se prvo iscrtava roditeljski objekat, a zatim se deca objekti iscrtavaju preko roditelja za svaki čvor stabla.

Pomeranja objekata GKI ili promena sadržaja može izazvati vizuelni efekat treperenja prikaza (eng. *flickering*). Kada se objekat pomera, uklanja se sa prethodne pozicije i iscrtava na novoj poziciji. Ovo se može odnositi i na druge vidljive objekte tako da se ponovo iscrtava deo stabla prikaza. U zavisnosti od frekvencije osvežavanja ekrana i složenosti sadržaja koji se iscrtava, može se pojaviti treperenje ekrana.

Problem se rešava mehanizmom duplog baferovanja (eng. *double buffering*). Kopija sadržaja ekrana se čuva u memorijskom baferu. Može biti deo ili čitavo stablo prikaza. Memoriju popunjava algoritam koji iscrtava i ažurira prikaz GKI. Po završetku iscrtavanja GKI, sadržaj ove memorije se kopira u memoriju ekrana koja prikazuje matricu piksela. Ovakav mehanizam eliminiše treperenje zbog toga što je kopiranje memorije piksela brža operacija od

ponovnog iscrtavanja stabla prikaza. Pored toga, nepromenjeni delovi stabla često i ne budu kopirani. Naziv potiče od postojanja dva bafera u video memoriji (Slika 2.18). Jedan bafer prikazuje sadržaj GKI, dok se u drugom ažurira prikaz GKI (iscrtavanje promena). Kada se završi ažuriranje, vrši se zamena prikaza dva bafera (eng. *swap*). Prethodno ažurirani se prikazuje, dok se prethodno prikazani ažurira i tako dalje. Zamena je veoma brza operacija koja se odigrava na nivou video hardvera i u vreme osvežavanja ekrana (eng. *refresh rate*) tako da se treperenje ne pojavljuje.



Slika 2.19. Duplo baferovanje razdvaja brzinu iscrtavanja stabla prikaza od frekvencije osvežavanja ekrana. Prikazuje frontalni bafer (front), iscrtava u pozadinski bafer (back), i zamenjuje uloge dva bafera. Izvor [38].

3. Dizajn grafičkog korisničkog interfejsa

Izgled i ponašanje su podjednako važni za prihvatanje GKI od strane korisnika, odnosno sistema koji interfejs predstavlja, i moraju se uvek razmatrati zajedno. Dizajn i programiranje GKI su dve usko povezane, ali odvojene uloge u razvoju GKI. Dok se programiranje fokusira na implementaciju i izgled interfejsa kroz funkcije koje treba da obezbedi korisniku, akcenat dizajna je korisnikov doživljaj izgleda i ponašanja interfejsa. Programiranje primarno određuje korisnost GKI (eng. *utility*), tj. da li interfejs obezbeđuje potrebne funkcije. Dizajnom je u najvećoj meri određena upotrebljivost GKI (eng. *usability*), tj. da li su sadržane funkcije korisniku lake sa korišćenje. Obe karakteristike su jednako značajne za uspeh GKI. Više informacija o upotrebljivosti i dizajnu GKI se može naći u [39]. U praksi je čest slučaj da jedna osoba obavlja uloge programera i dizajnera. Zbog toga je primenjivo znanje dizajna GKI korisno za programere i obrnuto. Razmatranje principa dizajna u toku programiranja vodi ka upotrebljivijim GKI. Pored toga, na ovaj način se na vreme (u toku razvoja) mogu uočiti i rešiti problemi upotrebljivosti čije otkrivanje i otklanjanje u fazi korišćenja GKI može biti znatno skuplje i složenije.

Principi grafičkog dizajna koji se koriste u dizajnu GKI obuhvataju [40]:

- Jednostavnost (eng. *simplicity*),
- Kontrast (eng. *contrast*),
- Ponavljanje (eng. *repetition*),
- Ravnanje (eng. *alignment*), i
- Blizina (eng. *proximity*).

Dizajn GKI je složen i uslovljen većim brojem faktora, kao što su domen interfejsa, funkcije, krajnji korisnici, raspoloživi resursi i drugi. Principi su generalizovane preporuke, ne pravila kako rešiti konkretan problem. Stoga njihova primena zahteva pažljivu analizu i kritičko razmišljanje o dizajnu elemenata GKI.

3.1. Jednostavnost

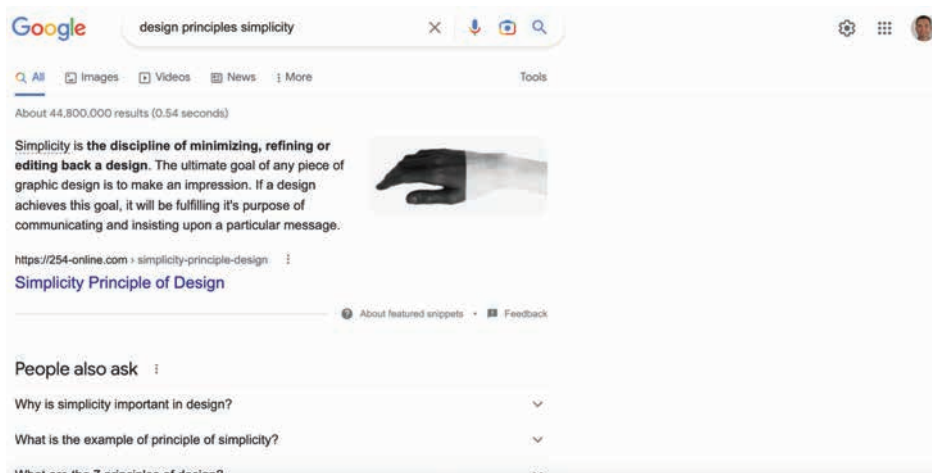
Jednostavnost dizajna GKI implicira korišćenje što je moguće manje elemenata tako da čine potpun i skladan izgled i funkcionalnost [41]. Princip je po prirodi proces uklanjanja, ne dodavanja. Zbog toga može biti u suprotnosti sa drugim principima ili tendencijom dodavanja sadržaja u GKI. U tom pogledu, jednostavnost zahteva balans između dodavanja sadržaja u GKI i sadržaja koji neophodan i relevantan za funkciju dela ili čitavog GKI.



Slika 3.1. Princip jednostavnosti je univerzalan i nalaže onu količinu sadržaja neophodnu za funkcije sistema ili uređaja. Izvor [40].

GKI koji sadrže samo neophodne informacije su generalno lakši za učenje, i korisnici ih brže koriste i prave manje grešaka prilikom korišćenja. Drugim rečima, veća količina informacija na interfejsu, naročiti ako nisu relevantne za funkciju interfejsa, može otežati nalaženje informacija potrebnih korisniku, produžiti vreme potrebno za nalaženje informacija od interesa, i uzrokovati greške korisnika u smislu biranja pogrešnih opcija ili sadržaja.

Svaki GKI je specifičan u smislu domena ili funkcionalnosti koje pruža, i grupe korisnika kojima je namenjen. Zbog toga se jednostavnost mora razmatrati u odnosu na navedene faktore (Slika 3.2). Na primer, ako govorimo o GKI alata za programiranje ili grafički dizajn, interfejs je bogat sadržajem zbog toga što je namenjen korisnicima sa specifičnim znanjima i veštinama kojima su sadržane opcije neophodne za rad. Sa druge strane, GKI aplikacije za pretragu ili naručivanje mora biti jednostavan i sadržati samo neophodne informacije za korisnike koji ne moraju imati znanje i veštine u određenoj oblasti.



Slika 3.2. Princip jednostavnosti se posmatra u kontekstu namene GKI i ciljnih korisnika. Interfejs Miro kolaborativnog alata za dijagrame i projektne timove (gore), izvor: <https://miro.com>, i interfejs Google Search alata za pretragu namenjenog korisniku Interneta (dole), izvor: <https://www.google.com>.

Postoje tri opšte tehnike za jednostavnost GKI [40]:

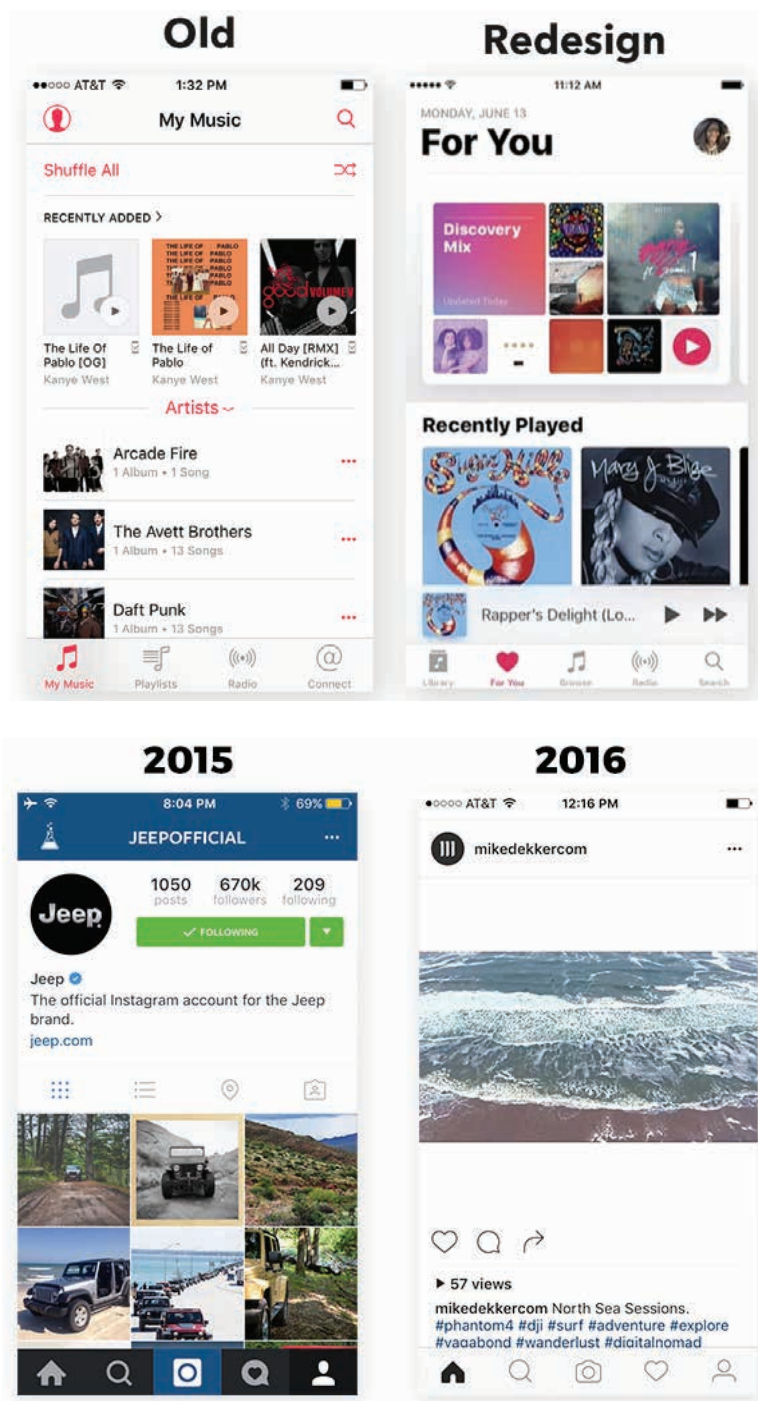
- Redukcija (eng. *reduction*),
- Regularnost (eng. *regularity*), i
- Višestruke funkcije (eng. *multiple functions*).

Redukcija nalaže uklanjanje elemenata koji nisu relevantni za funkciju tekućeg prikaza ili čitavog GKI. Tehnika se primenjuje u tri koraka:

- a) Definirati cilj dizajna prikaza GKI kao poruku koju prenosi korisniku.
- b) Kritički ispitati da li svaki element prikaza (dugme, ikona, boja, tekst) služi cilju dizajna.
- c) Ukloniti sve elemente koji ne služe cilju, odnosno nisu relevantni za kontekst prikaza.

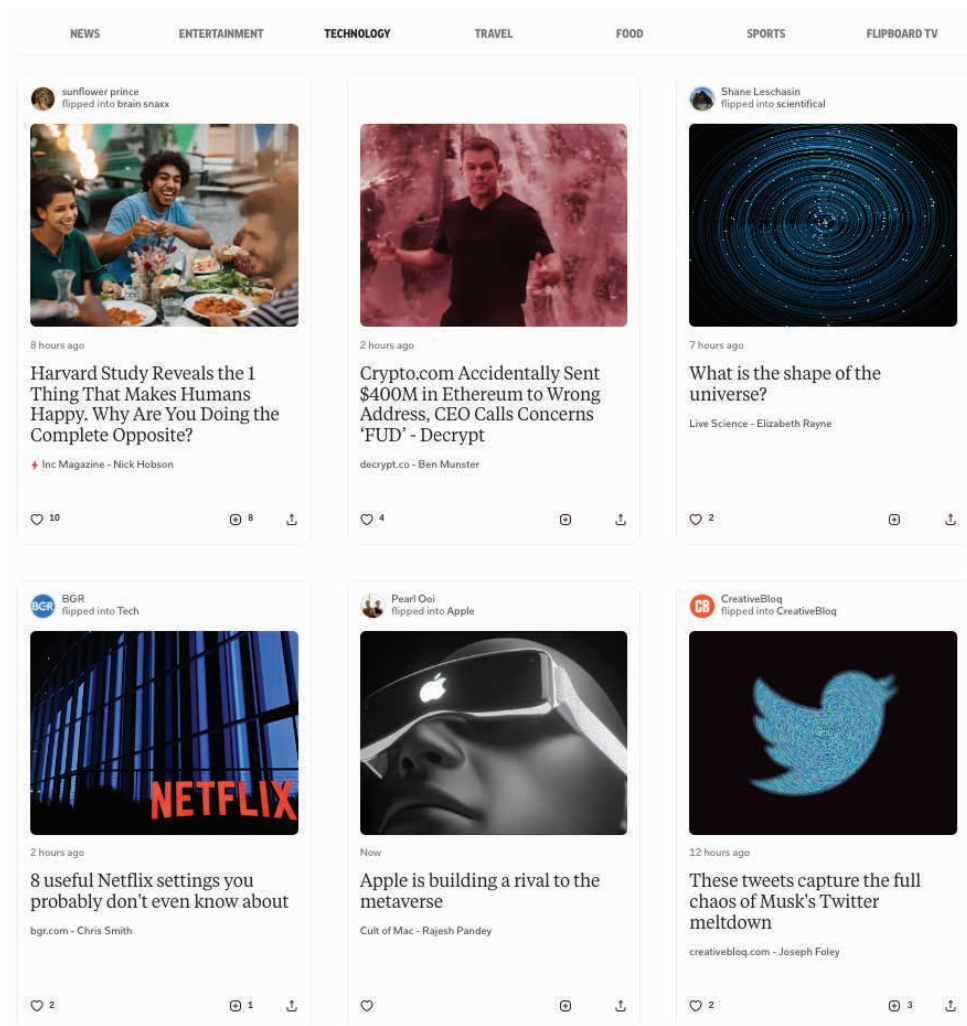
Slika 3.3 ilustruje redukovanje sadržaja za različite verzije aplikacija. Posmatrajući dva različita GKI mogu se uočiti sledeći zajednički elementi:

- Manji broj korišćenih *boja*.
- Isticanje naziva sekcija *fontom* odgovarajuće veličine, debljine i boje.
- *Ikone* koje imaju jednostavnu formu i asociraju na akciju u pozadini.
- Postojanje *praznog prostora* kako bi sadržaj bio jasno razgraničen (eng. *empty space*).



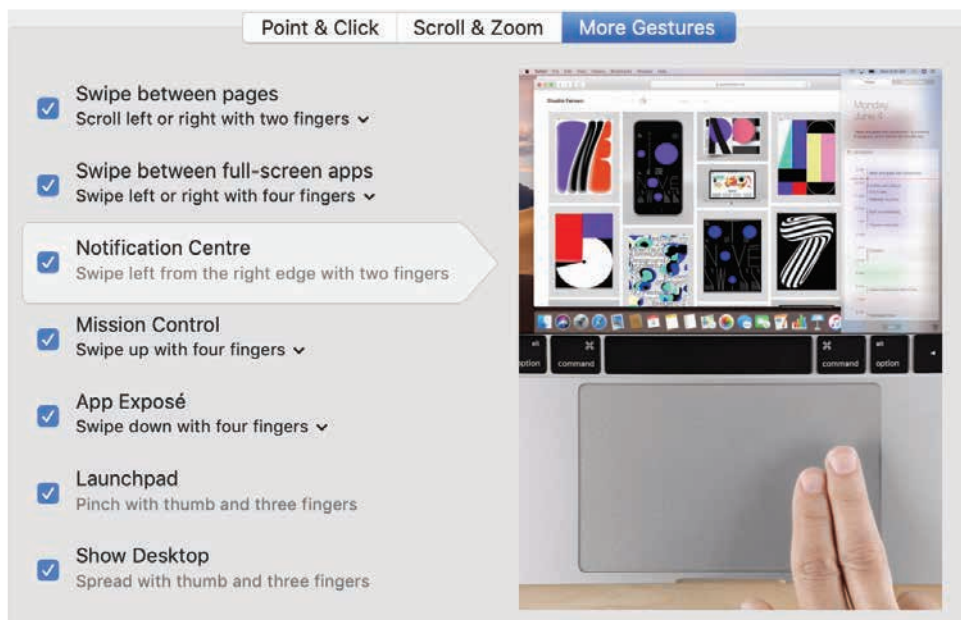
Slika 3.3. Primena tehnike redukcije na primeru poznatih aplikacija, Apple Music (gore) i Instagram (dole). Izvor: <https://medium.com/slate-teams/complexion-reduction-a-new-trend-in-mobile-design-cef033a0b978>.

Regularnost podrazumeva primenu identičnih ili sličnih vizuelnih formi. Drugim rečima, potrebno je definisati šablon prikaza elemenata i koristiti u dizajnu GKI. Šablon je kombinacija vizuelnih elemenata, kao što su font, boja, stil linije, oblik, orijentacija, dimenzije i pozicija. Ako je dizajn regularan, lakše se mogu istaći važni delovi GKI, tj. korisnik će ih lakše i brže opaziti. Razlog tome je što vizuelna percepcija čoveka prirodno grupiše informacije GKI u logičke celine (eng. *chunk*) kako bi ih lakše obradio. Sadržaj koji prati odgovarajući obrazac olakšava grupisanje i obradu informacija od strane korisnika.



Slika 3.4. Primer regularnosti predstavlja mrežni raspored (eng. *grid*) čije ćelije imaju uniformne dimenzije, kao i strukturu sadržaja svake ćelije. Izvor: <https://flipboard.com>.

Šabloni su raznovrsni u zavisnosti od namene GKI, ciljne grupe korisnika i iskustva i inspiracije programera, tj. dizajnera. Primeri se mogu naći i kod GKI standardnih aplikacija, kao što je prikazano na Slici 3.5.



Slika 3.5. Šablon za opis komandi koje koriste pokrete prstiju i šake korisnika (eng. *gesture*) je primer regularnosti. Izvor: <https://www.apple.com/macbook-pro/>.

Elementi GKI koji objedinjavaju veći broj uloga doprinose jednostavnosti jer eliminišu potrebu za dodavanje većeg broja elemenata sa odvojenim funkcijama. Neki od standardnih elemenata GKI imaju veći broj funkcija. Ručica elementa za skrolovanje (eng. *scrollbar handle*) omogućava korisniku kretanje kroz prikaz koji se ne može ceo prikazati na strani interfejsa i pruža korisniku približnu informaciju o veličini sadržaja na osnovu pozicije ručice. Elementi za navigaciju, kao što su straničenje (eng. *pagination*) i mrvice (eng. *breadcrumbs*), omogućavaju kretanje (ili navigaciju) kroz sadržaj, daju informaciju o orijentaciji (ili položaju) korisnika u sadržaju, i obezbeđuju informaciju o veličini sadržaja. Vidljive i precizne informacije u vezi sa navigacijom i orijentacijom su važne za uspešno korišćenje Veb GKI. Razlog tome je što u virtuelnom prostoru GKI korisnici generalno nemaju identičan doživljaj prostora kao u fizičkom svetu i način na koji je taj prostor predstavljen vizuelno može olakšati ili otežati korišćenje GKI.



Slika 3.6. Mrvice omogućavaju korisnicima kretanje kroz GKI (funkcija navigacije) i pružaju informaciju o trenutnoj poziciji korisnika u GKI (funkcija orijentacije). Izvor: <https://www.justinmind.com/blog/breadcrumb-website-navigation/>.

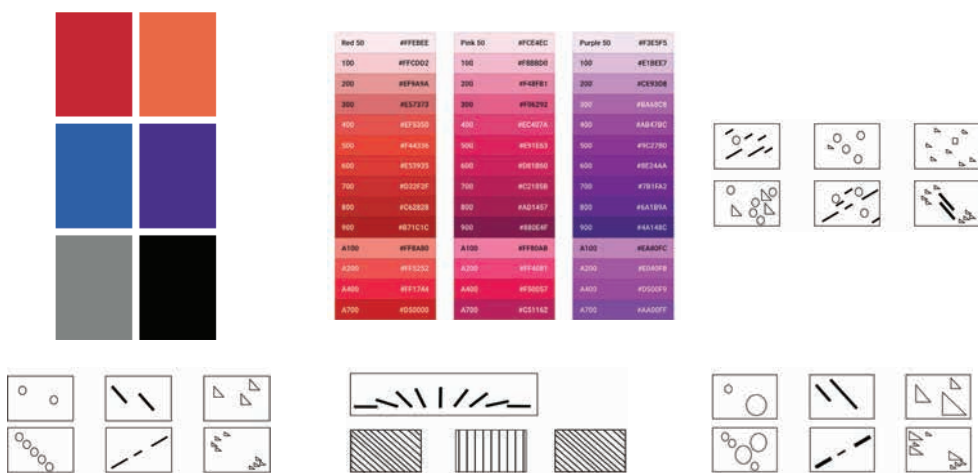
3.2. Kontrast

Kontrast se može definisati kao primetne razlike u vizuelnim karakteristikama sadržaja, odnosno razlike koje korisnik može neposredno opaziti gledajući u sadržaj. U cilju kreiranja i upravljanja kontrastom, definisane su i opisane vizuelne karakteristike kao vizuelne varijable [42].

U vizuelne varijable spadaju:

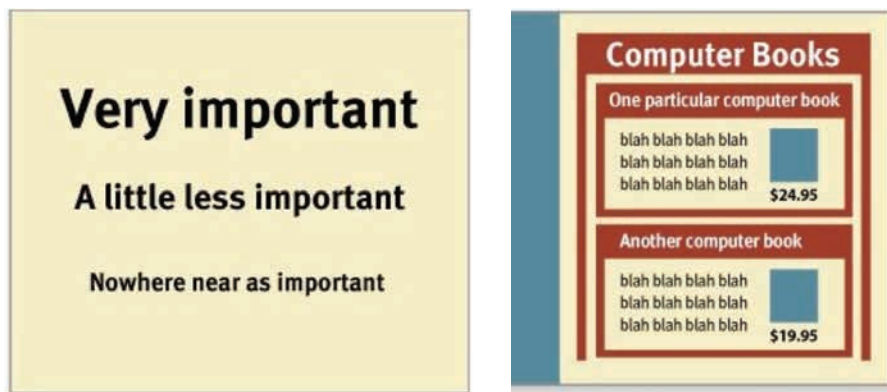
- Boja (eng. *hue*),
- Nijansa (eng. *value*),
- Oblik (eng. *shape*),
- Pozicija (eng. *position*),
- Orijentacija (eng. *orientation*), i
- Veličina (eng. *size*).

Vizuelne varijable se mogu posmatrati kao sredstva za potizanje kontrasta. Kontrast se javlja kao posledica promena vrednosti jedne ili većeg broja vizuelnih karakteristika. Odatle i dolazi korišćenje termina varijabla. Motiv može biti čitljivost elementa ili dela GKI, ili njihovo isticanje (eng. *highlighting*) kako bi se korisniku dodatno skrenula pažnja na specifičan sadržaj. Obično se ističu oni elementi koji su važniji u odnosu na ostali sadržaj strane GKI (npr. glavne ili često korišćenje funkcije i sadržaji). Kontrast kreira vizuelne hijerarhije u GKI time što određene delove ističe u odnosu na druge.



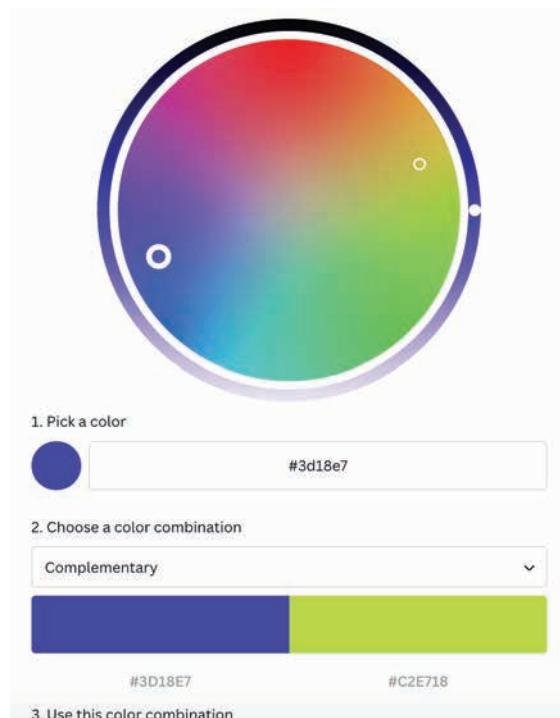
Slika 3.7. Primeri vizuelnih varijabli kao zasebnih promena (sa leva desno, odozgo na dole): boje, nijanse, oblika, pozicije, orijentacije i veličine.

U praksi, dizajner ili programer se odlučuje za korišćenje specifičnih vizuelnih varijabli za kreiranje kontrasta. Izbor zavisi od formata i sadržaja podataka interfejsa, i zadatka korisnika.



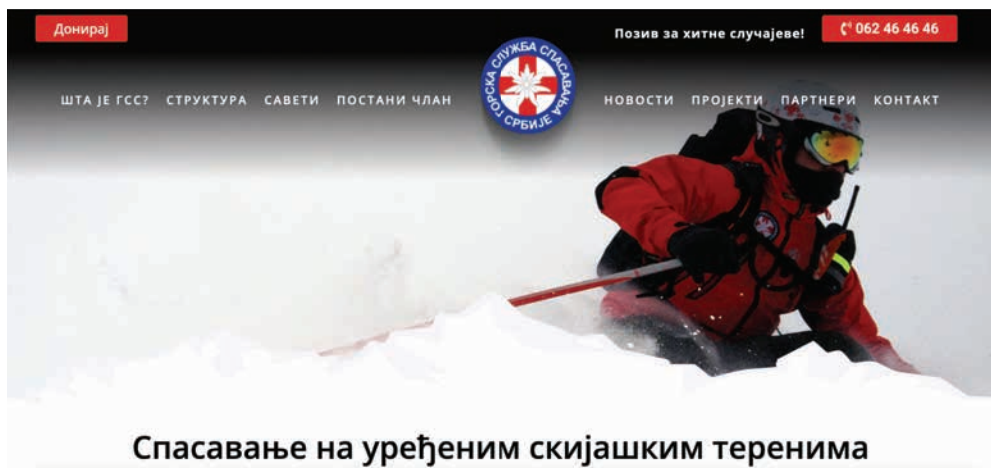
Slika 3.8. Kontrast korišćenjem veličine i pozicije (levo), i veličine, pozicije, oblika i boje (desno). Izvor: [43].

U dizajnu GKI, čest problem u vezi sa kontrastom predstavlja izbor boja, tj. kombinacija frontalne boje i boje pozadine tako da se naziv ili simbol elementa ne može lako prepoznati. Primeri mogu biti dugme, ikona, ili stavka menija. Točak boja (eng. *color wheel*) definiše parove boja koje obezbeđuju dobar kontrast (komplementarne su) i može se koristiti za izbor kombinacija boja (Slika 3.9).



Slika 3.9. Točak boja organizuje boje na taj način da su dijametralno raspoređene boje komplementarne, dok susedne boje nisu. Izvor: <https://www.canva.com/colors/color-wheel/>.

Kreiranje kontrasta je važno za efikasno razumevanje sadržaja GKI i pronalaženje željenih opcija, kao u primeru u nastavku.



Slika 3.10. Dobar kontrast se često postiže kombinacijom boje, pozicije i veličine elemenata GKI. Izvor: <https://gss.rs> (Gorska Služba Spasavanja Srbije).

3.3. Ponavljanje

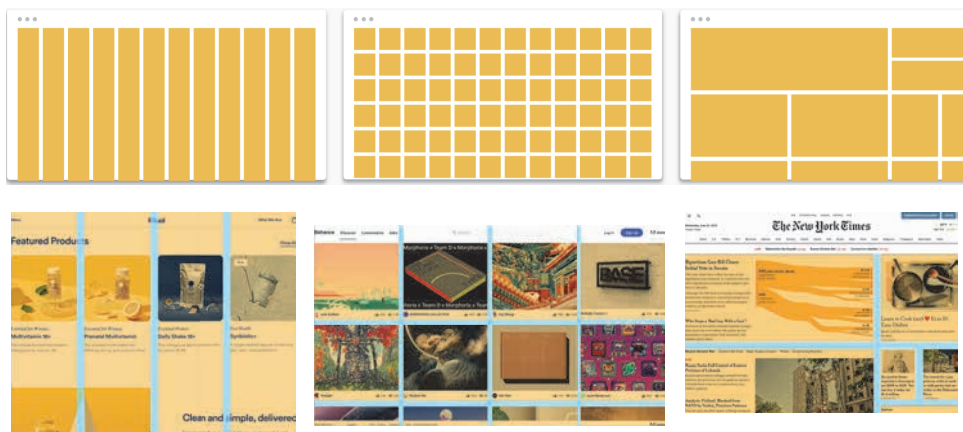
Ponavljjanje podrazumeva ponovnu upotrebu identičnih ili sličnih elemenata u dizajnu GKI. Mogu se ponavljati različiti elementi, kao što su boje, oblici, fontovi, elementi GKI ili kombinacija prethodno navedenih. Za razliku od kontrasta koji ističe razlike, ponavljanje daje utisak kohezije elemenata i pripadanja celini. Ovo zahteva balans u primeni dva principa u dizajnu GKI. Pored toga, ponavljanje većeg broja elemenata GKI može proizvesti utisak monotonog i neinteresantnog dizajna. Stoga je preporučljivo istaći i ponavljati mali broj elemenata.



Slika 3.11. Ponavljajuće forme u dizajnu GKI doprinose utisku celine.

3.4. Ravnanje

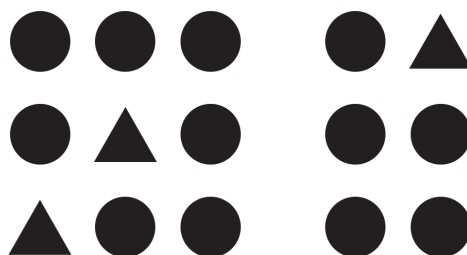
Ravnanje doprinosi povezivanju informacija u prikazu GKI čime povećava čitljivost istog. Elementi GKI se po pravilu ne raspoređuju slučajno, već po određenom obrascu. Mreža (eng. *grid*) se često koristi kao referentna struktura u okviru koje se raspoređuje sadržaj GKI [44]. Na ovaj način se ostvaruje vizuelna povezanost i doživljaj celine elemenata prikaza koji olakšavaju skeniranje interfejsa od strane korisnika [43]. Poznato je da korisnici najpre letimično pregledaju sadržaj GKI (ili skeniraju) dok ne nađu delove ili informacije od interesa koje onda detaljnije analiziraju [43]. Slika 3.12. prikazuje varijante mreže (gore) sa odgovarajućim primerima dizajna (dole).



Slika 3.12. Varijante mreže mogu definisati ravnanje po kolonama (levo), modularno kao uniformne ćelije (sredina), i kao hijerarhiju sa ćelijama različitih dimenzija i raspored (desno). Hijerarhija može istaći određene delove prikaza u odnosu na ostatak sadržaja. Izvor: [44].

3.5. Blizina

Blizina vizuelno definiše logičku povezanost elemenata GKI. Drugim rečima, fizički bliski sadržaji se opažaju kao logički povezani, dok se fizički udaljeni elementi opažaju kao logički odvojeni. Logička povezanost se ogleda u sličnim funkcionalnostima ili zajedničkom kontekstu. Poreklo ovog principa je u Geštalt principima vizuelne percepcije čoveka koji kažu da objekte koji su bliži jedni drugima opažamo kao celinu za razliku od udaljenih objekata. U dizajnu GKI pravilo se ogleda u pravilnom korišćenju praznog prostora (eng. *empty space*) i to kao manje praznog prostora između sličnih elemenata i više praznog prostora između elemenata koji nisu logički povezani (Slika 3.13). Više informacija o Geštalt principima se može naći u [39].



Slika 3.13. Zbog principa blizine se u skupu levo opažaju redovi, dok se u skupu desno opažaju kolone čak i kada su izgledi elemenata različiti. Izvor: [45].

Podela teksta na paragrafe predstavlja primenu principa blizine. Paragraf predstavlja deo teksta koji ima specifičan kontekst ili poruku i razdvojen je većim vertikalnim proredom od susednih paragrafa. Na ovaj način se grupe povezanog teksta mogu uočiti i pre čitanja teksta paragrafa (Slika 3.14 desno).

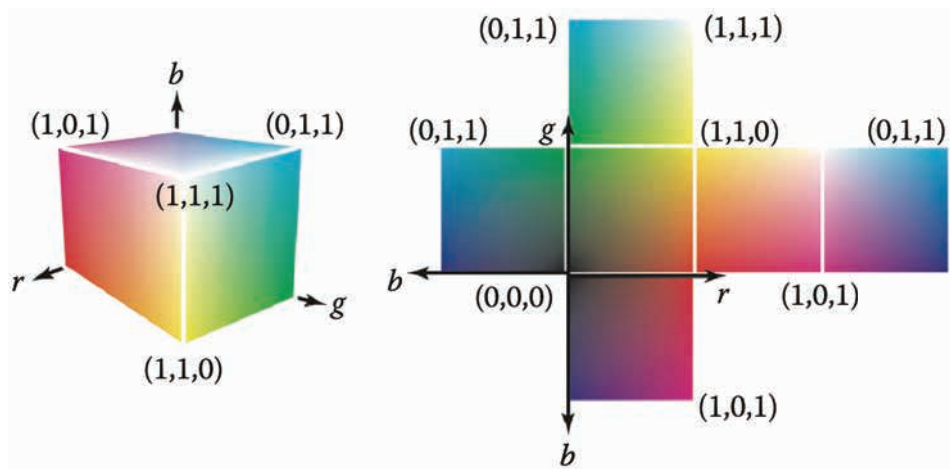


Slika 3.14. Paragrafi teksta predstavljaju primer principa blizine. Izvor [45].

Pravilno korišćenje praznog prostora je važno za dizajn GKI sa stanovišta principa jednostavnosti i blizine. Prazan prostor grupiše sadržaje GKI na osnovu sličnosti. U tom pogledu dizajn GKI se može posmatrati i kao balans između potrebe za praznim prostorom i dodavanjem informacija i kontrola. Sa jedne strane, GKI ne treba imati previše praznog prostora sa sadržajima manjih veličina (npr. dugmad, tekst i drugi) tako da se teže opažaju. Sa druge strane, prazan prostor ne sme biti suviše mali tako GKI ima previše sadržaja (eng. *visual clutter*) zbog toga što će tada korisnici imati poteškoća u nalaženju informacija od interesa.

3.6. Boje i tipografija

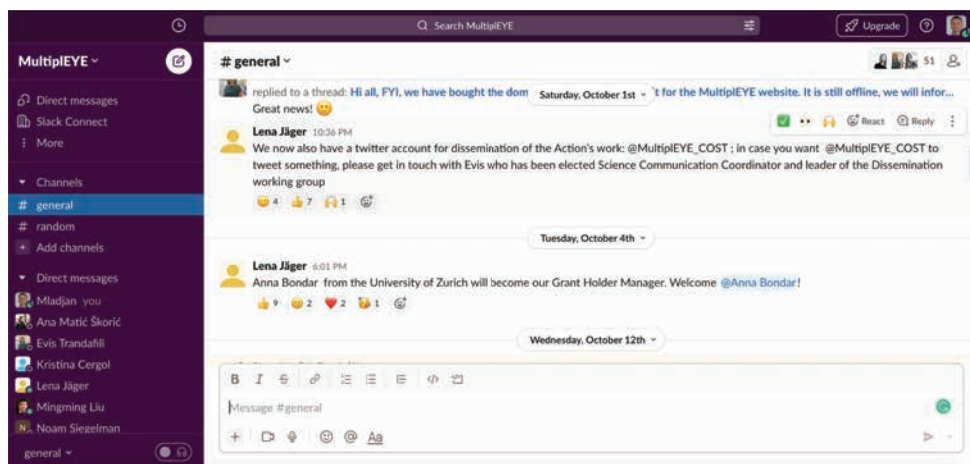
Boje predstavljaju važan element dizajna GKI. Standard za reprezentaciju boja u GKI jeste RGB model (eng. *Red Green Blue model*) koji kreira različite boje kao kombinacije osnovnih boja. tj. crvene, zelene i plave. Mogu se koristiti različite skale za definisanje udela osnovnih boja, procenti (0-100) ili normalizovane vrednosti (0.0-1.0). Kocka boja (Slika 3.15) olakšava kombinovanje boja kao vrednosti trodimenzionalnog koordinatnog sistema definisanog osnovnim bojama, gde je koordinatni početak crna boja (0,0,0) i teme sa koordinatama (1,1,1) bela boja. RGB model koriste displeji računara sa obzirom da svaki piksel sadrži odvojene crvene, zelene i plave komponente boja.



Slika 3.15. RGB kocka boja omogućava definisanje boja u trodimenzionalnom referentnom sistemu čije ose predstavljaju osnovne boje (crvena, zelena i plava). Izvor: [38].

Preporuke za korišćenje boja u dizajnu GKI su sledeće [46]:

- Koristiti do tri boje - manji broj boja kreira jednostavniju vizuelnu hijerarhiju koja ne odvlači pažnju kao što bi mogao biti slučaj kod korišćenja velikog broja boja. Pored toga, treba izbegavati korišćenje zasićenih (eng. *saturated*) boja, tj. boja visokog intenziteta.
- Koristiti pravilo 60-30-10 u kombinovanju odabranih boja - primarna boja pokriva oko 60% površine GKI, sekundarna oko 30% i tzv. akcentovana boja oko 10%. Primarna i sekundarna mogu biti neutralne, dok akcentovana može biti i naglašenija tako da ističe delove prikaza. Ovakvi udeli kreiraju dobar vizuelni balans boja za korisnika (Slika 3.16).
- Koristiti boje konzistentno - odabrane boje treba koristiti na svim prikazima GKI, tj. ne treba primenjivati različite palete boja u različitim delovima interfejsa.



Slika 3.16. Pravilo 60-30-10 preporučuje balans boja u pogledu relativnih udela. Primarna boja je bela, sekundarna ljubičasta i akcentovana je siva (poruka u fokusu). Izvor: <https://www.slack.com>.

Tipografija (eng. *typography*) je oblast koja izučava korišćenje i prikaz teksta. U dizajnu GKI koristi se termin glensing (eng. *glancing*) [47] za brzo čitanje kratkog teksta (obično nekoliko reči). Glensing je tipičan za GKI zbog toga što se na ovaj način čitaju i kontrole kao što su nazivi dugmadi, stavke menija, opcije za navigaciju i druge. Oblik karaktera (eng. *font*) i prostor između karaktera, reči, redova i paragrafa (eng. *spacing*) primarno određuju u kojoj meri je podržan glensing i tekst čitljiv za korisnika [48].

Klasa ili familija fontova je poznata pod nazivom tajpfejs (eng. *typeface*), dok se naziv font koristi za konkretan primerak familije. Fontovi jedne familije dele specifične oblike (forme). *Serif* fontovi, kao što su *Georgia* i *Times*, se koriste za tekst paragrafa. Oštrije forme karaktera ove familije fontova olakšavaju prepoznavanje reči i čitanje. *Sans Serif* fontovi, kao što su *Verdana* i *Arial*, su pogodni za izdvojen tekst (naslovi i zaglavlja) zbog karakteristične forme koja obezbeđuje dobar kontrast u odnosu na tekst paragrafa. Ne treba koristiti više od dve ili tri različite familije fontova (jedna za naslove i druga za tekst paragrafa). Krupniji font poboljšava glensing, kao i korišćenje velikog početnog slova reči na displejima manjih dimenzija (mobilni uređaji) [47].

Horizontalni prostor između reči mora obezbediti prepoznavanje i spajanje reči. Sa druge strane, suviše veliki prostor ometa ritam čitanja. Prostor između linija teksta mora omogućiti prepoznavanje reči vertikalno. Uvek treba koristiti margine i ne smeštati tekst uz ivice prikaza GKI jer će korisnici teže opaziti takav tekst.

4. JavaScript i TypeScript jezici

JavaScript je najčešće korišćen programski jezik [49] i vodeći jezik za implementaciju Veb GKI. Jezik je nastao 1995. godine, glavni tvorac jezika je Brendan Eich, sofer inženjer i jedan od osnivača kompanije Mozilla [50]. Jezik je nastao kao proceduralni i namenjen za pisanje programa koje izvršavaju Veb pretraživači. Jezik je dalje razvijan tako da su dodati elementi objektno-orijentisanog i funkcionalnog programiranja.

Na osnovu evolucije jezika, udžbenik opisuje JavaScript kroz tri grupe elemenata:

- *Proceduralni elementi* - operatori, varijable, funkcije, tipovi podataka i kolekcije.
- *Objektno-orijentisani elementi* - klase, metode, funkcije kao objekti prvog reda, nasleđivanje i prototipi.
- *Funkcionalni elementi* - anonimne funkcije i arrow funkcije.

JavaScript programi se učitavaju i izvršavaju u Veb pretraživačima. Prilikom učitavanja i izvršavanja HTML koda ispod, pretraživač analizira sadržaj HTML datoteke, i iz *script* taga čita naziv JavaScript datoteke. Pretraživač može poslati zahtev serverskoj strani za datoteku *fajl.js*, slično kao za CSS datoteku ili sliku ukoliko datoteka nije lokalna. Serverska strana šalje JavaScript datoteku kao odgovor. Zatim se JavaScript datoteka se izvršava na klijentskoj strani, tj. u Veb pretraživaču računara korisnika.

```
<!DOCTYPE html>
<html>
  <head>
    <title>Dizajn i programiranje korisničkog
interfejsa</title>
    <link rel="stylesheet" href="style.css" />
    <script src="fajl.js"></script>
  </head>
  <body>
    ... sadržaj stranice ...
  </body>
</html>
```

U nastavku je dat opis elemenata JavaScript jezika čije je poznavanje neophodno za dizajn i programiranje GKI. Kompletna dokumentacija jezika se može naći u [51].

4.1. Proceduralni JavaScript

Kod proceduralnog jezika instrukcije, tj. programske naredbe se izvršavaju sekvencijalno, od početka ka kraju po redosledu navođenja. Izvorni fajlovi (sa .js ekstenzijom) ne prevodi programer eksplicitno, već se u približno isto vreme prevode i izvršavaju od strane pretraživača. Princip je poznat pod nazivom *Just-in-time* (JIT) prevođenje ili kompajliranje. JavaScript prevodilac je sastavni deo Veb pretraživača i prevodi izvorni kod u binarni (eng. *byte*) oblik koji se odmah potom izvršava. Binarni kod izvršava interpreter (eng. *interpreter*). Poznatiji interpreteri su *V8 Ignition* (Google Chrome pretraživač) [52] i *SpiderMonkey* (Mozilla pretraživač) [53].

4.1.1. Operatori i naredbe

JavaScript operatori su slični kao u drugim programskim jezicima (npr. Java). Korišćenje operatora dodele vrednosti i standardnih aritmetičkih i logičkih operatora je prikazano u kodu ispod. Pored toga, jezik implementira i naredbe premeštanja toka programa (ili skoka). *Continue* nastavlja izvršavanje u narednoj iteraciji petlje. *Break* izlazi iz petlje, tj. nastavlja izvršavanje u prvoj naredbi nakon petlje. *Return* vraća vrednost funkcije i završava sa izvršavanjem funkcije.

```
//Standardni operatori
i = 3;
i = i * 10 + 3 + (i / 10);
while (i >= 0) {
    sum += i*i;
    i--;
}

//Standardne operacije
/ % + - ! >= <= > < && || ?:

//Naredbe skoka
continue/break/return
```

Jezik sadrži i standardne upravljačke naredbe, poput petlji i uslovnih grananja. *For* petlja ima iterator čija se vrednost inkrementira za jedan nakon svake iteracije petlje i koji određuje ukupan broj iteracija. Izvršavanje *while* petlje zavisi od navedenog uslova. Naredba uslovnog grananja (*if/else*) definiše tokove izvršavanja programa u zavisnosti od specifične kombinacije *if* i *else* uslova. Komentarisati se mogu pojedinačne linije (//) ili veći broj linija programa (*/* */*).


```
//for petlja
    for (let i = 0; i < 5; i++) { ... }

//while petlja
    while (notFinished) { ... }

//komentari
    //jednolinijski
    /* višelinijski ili blok */

//uslovne (if) naredbe
    if (...) {
        ...
    } else {
        ...
    }
}
```

4.1.2. Varijable

Varijable se mogu definisati korišćenjem pomoću tri ključne reči koje su ujedno i modifikatori vidljivosti. Primeri su dati u kodu u nastavku.

```
//Opseg vidljivosti je globalan
var x = 15;

//Opseg vidljivosti je blok
let voće = 'banana';

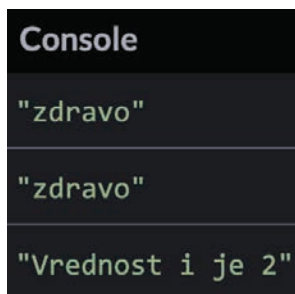
//Konstanta
const isHungry = true;
```

Može se primetiti da tip varijable nije naveden prilikom njihovog definisanja. U JavaScript jeziku varijable imaju dinamičke tipove i on je određen tipom dodeljene vrednosti sa desne strane operatora dodele vrednosti. Drugim rečima, varijable dobijaju tip poslednje dodeljene vrednosti. Princip je poznat pod nazivom *dinamičko tipiziranje*.

Ključna reč *var* označava globalan opseg vidljivosti, odnosno varijabla je vidljiva (može se pristupiti njenoj vrednosti) i izvan bloka naredbi u kojem je prethodno definisana. Blok je skup naredbi ograničen vitičastim zagradama (`{}`). Zbog toga je u primeru u nastavku varijabla *i* vidljiva i nakon završetka bloka naredbi *for* petlje u kojoj je definisana. Petlja je definisana u okviru

funkcije *štampaPoruka* koja se najpre definiše a zatim poziva. *Console* je predefinisani JavaScript objekat koji pokazuje na *Console* prikaz tekućeg prozora Veb pretraživača koji izvršava programski kod. Metoda *log()* ispisuje svoj argument u *Console* prikazu.

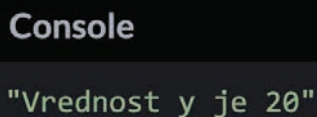
```
function štampaPoruka(poruka, broj) {  
  for (var i = 0; i < broj; i++) {  
    console.log(poruka);  
  }  
  console.log('Vrednost i je ' + i);  
}  
štampaPoruka('zdravo', 2);
```



```
Console  
"zdravo"  
"zdravo"  
"Vrednost i je 2"
```

U sledećem primeru, varijabla *y* se definiše u okviru bloka *if* naredbe i njena vrednost ispisuje nakon završetka bloka.

```
var x = 20;  
if (x > 0) {  
  var y = 20;  
}  
console.log('Vrednost y je ' + y);
```



```
Console  
"Vrednost y je 20"
```

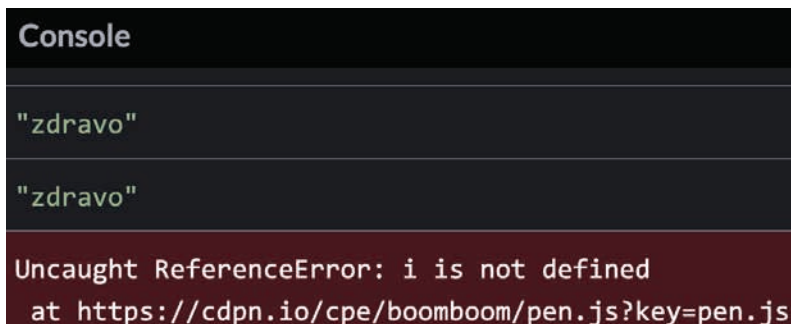
Ključna reč *let* definiše lokalni opseg vidljivosti varijable. U primeru u nastavku varijabla *i* je definisana kao *let* u *for* petlji. Ako se vrednosti pokuša pristupiti van bloka u kojem je definisana, pojaviće se greška koja govori da vrednost nije definisana.

```
function štampaPoruka(poruka, broj) {
```

```

for (let i = 0; i < broj; i++) {
  console.log(poruka);
}
console.log('Vrednost i je ' + i);
}
štampaPoruka('zdravo', 2);

```



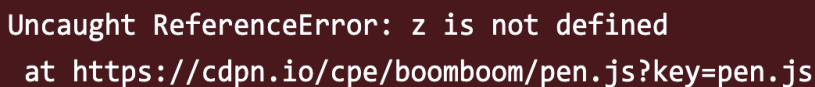
The screenshot shows a browser console with a dark background. At the top, the word "Console" is written in white. Below it, there are two lines of output in green: `"zdravo"`. The third line is an error message in white text on a dark red background: `Uncaught ReferenceError: i is not defined`. Below the error message, the source location is shown in white: `at https://cdpn.io/cpe/boomboom/pen.js?key=pen.js`.

Ključna reč *const* definiše lokalni opseg vrednosti varijable uz koju se navodi. Zbog toga se u primeru ispod javlja greška. Prilikom definisanja konstanti u programu treba voditi računa o tome u kojim delovima će se koristiti i shodno tome ih definisati na odgovarajuće mestu.

```

let x = 20;
if (x > 10) {
  const z = 30;
}
console.log(z);

```



The screenshot shows a browser console with a dark background. It displays an error message in white text on a dark red background: `Uncaught ReferenceError: z is not defined`. Below the error message, the source location is shown in white: `at https://cdpn.io/cpe/boomboom/pen.js?key=pen.js`.

4.1.3. Funkcije

Značenje i korišćenje funkcija je uglavnom identično kao u drugim programskim jezicima. Sa druge strane, JavaScript funkcije su specifične u pogledu definisanja i načina na koji su predstavljene u toku izvršavanja programa. Stoga je opis funkcija skoncentrisan na njihove specifičnosti u jeziku.

Jedan od načina za definisanje JavaScript funkcije je kao u kodu u nastavku. Ključna reč *function* označava početak definicije funkcije kod koje se najpre

navodi *ime* funkcije i opciono parametri u običnim zagradama (*()*). Telo funkcije je blok naredbi. Funkcija ispod nema parametre.

```
function ime() {  
    naredba;  
    naredba;  
    ...  
}
```

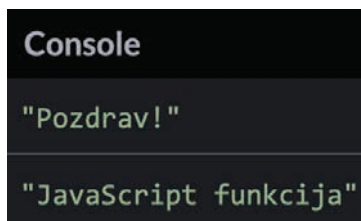
Ukoliko funkcija ima parametre, oni se navode u običnim zagradama. Nije potrebno navoditi tipove parametara, kao što je prikazano u primeru u nastavku. Pored toga, parametri funkcije se ne deklariraju sa modifikatorima vidljivosti, tj. *var*, *let*, ili *const*. Primer ispod ilustruje definisanje funkcije sa parametrima.

```
function štampaPoruka(poruka, broj) {  
    for (var i = 0; i < broj; i++) {  
        console.log(poruka);  
    }  
}
```

Hojsting funkcije (eng. *function hoisting*) je proces u kojem JavaScript interpreter deklaraciju funkcije smešta u globalni opseg vidljivosti pre izvršavanja programa. Posledica toga jeste da se funkcija definisana korišćenjem ključne reči *function* i imena funkcije može pozivati u programu pre nego je definisana.

U sledećem primeru funkcija je definisana, a zatim pozvana.

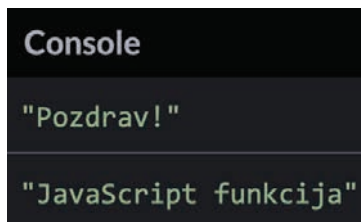
```
function pozdrav() {  
    console.log('Pozdrav!');  
    console.log('JavaScript funkcija');  
}  
pozdrav();
```



Identična funkcija se u narednom primeru najpre poziva a zatim definiše. Rezultat je isti kao u prethodnom primeru. Važno je napomenuti da hojsting

važi samo za navedeni način definisanja funkcija. Više informacija o hojstingu funkcije i hojstingu generalno se može naći u [54] i [55], respektivno.

```
pozdrav();  
function pozdrav() {  
    console.log('Pozdrav!');  
    console.log('JavaScript funkcija');  
}
```



JavaScript funkcije imaju promenljiv broj argumenata i svi argumenti čije se vrednosti ne navedu, tj. ne proslede prilikom poziva funkcije dobijaju vrednost tipa *undefined*.

Podrazumevana povratna vrednost svake funkcije je vrednost tipa *undefined*, ukoliko se eksplicitno ne definiše sa *return* naredbom.

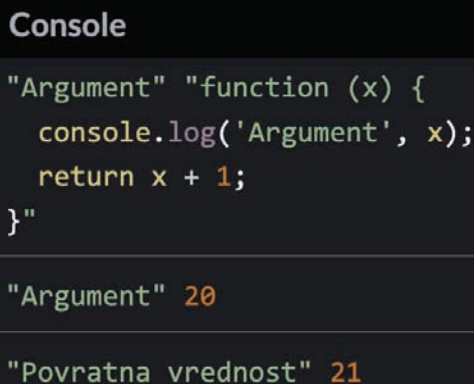
Funkcije su objekti prvog reda (eng. *first-class objects*). To znači da mogu imati svojstva ili atribute, biti dodeljene kao vrednost varijabli, predstavljati element kolekcije vrednosti, prosledene kao parametar druge funkcije ili povratna vrednost funkcije. Primer u nastavku ilustruje korišćenje funkcije kao objekta.

Na početku se definiše varijabla *funkVar* kojoj se kao vrednost dodeljuje funkcija. Ta varijabla dobija tip *function*. Funkcija ispisuje vrednost argumenta i vraća vrednost argumenta uvećanu za 1. Funkcija kreirana na opisani način se naziva anonimna funkcija (eng. *anonymous function*). Ime takve funkcije je naziv varijable kojoj je funkcija dodeljena kao vrednost.

Funkcija *mojaFunk* će kao argument imati drugu funkciju (*funkcija* parametar). Ona najpre ispisuje vrednost argumenta, zatim poziva funkciju-argument i povratnu vrednost dodeljuje varijabli *rezultat*. Na kraju ispisuje vrednost varijable *rezultat* i vraća vrednost te varijable kao povratnu vrednost. Funkcija se nakon definisanja poziva i rezultat ispod ilustruje prethodno opisane mehanizme.

```
var funkVar = function (x) {  
    console.log('Argument', x);  
    return x+1;  
};
```

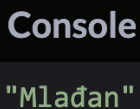
```
function mojaFunk(funkcija) {
  console.log('Argument', funkcija.toString());
  var rezultat = funkcija(20);
  console.log('Povratna vrednost', rezultat);
  return rezultat;
}
mojaFunk(funkVar);
```



```
Console
"Argument" "function (x) {
  console.log('Argument', x);
  return x + 1;
}"
"Argument" 20
"Povratna vrednost" 21
```

Sledeći primer definiše varijabla *stampajme* tako što joj kao vrednost dodeljuje anonimnu funkciju koja ima argument *ime* čiju vrednost prikazuje. Funkcija se poziva tako što se ime varijable kojoj je funkcija dodeljena kao vrednost koristi kao ime funkcije sa vrednošću parametra.

```
var stampajme = function(ime) {
  console.log(ime);
}
stampajme("Mlađan");
```



```
Console
"Mlađan"
```

4.1.4. Tipovi

U JavaScript jeziku varijable nemaju tipove dok vrednosti imaju tipove. Dodelom vrednosti varijabli ona prima tip dodeljene vrednosti. Osnovni tipovi koji postoje u jeziku su:

1. Logički tip (eng. *Boolean*) koji ima dve vrednosti, istinito (eng. *true*) ili neistinito (eng. *false*).
2. Broj tip (eng. *Number*) koji se odnosi na brožčane vrednosti koje su izražene kao realni brojevi dvostruke preciznosti (eng. *double*).
3. String tip (eng. *String*) predstavlja nisku znakova koji se tretiraju kao tekst, gde znakovi mogu biti alfanumerički i specijalni znaci i navode se pod jednostrukim ili dvostrukim znacima navoda.
4. Simbol tip (eng. *Symbol*) označava jedinstven simbol ili identifikator kao objekat.
5. Nulti tip (eng. *Null*) definiše nultu vrednost kao tip.
6. Nedefinisano tip (eng. *Undefined*) situaciju kada varijabli nije dodeljena vrednost označava kao zaseban tip.

Pored navedenih osnovnih tipova postoje i složeni tipovi, kao što je struktura, tj. kolekcija objekat (eng. *object*).

Svi brojevi su predstavljeni kao realni brojevi. Koriste se i tri simboličke konstante kao *Number.MIN_VALUE* za minimalnu brojčanu vrednost, *Number.MAX_VALUE* za maksimalnu brojčanu vrednost, i *NaN* (*Not-a-Number*) za slučajeve aritmetičkih operacija sa vrednostima koje nisu brojčanu.

Za navođenje string vrednosti se mogu koristiti jednostruki ili dvostruki znaci navoda. Jednom kreirane niske znakova su nepromenjive (eng. *immutable*) u memoriji programa. To znači da svaka operacija nad prethodno kreiranom niskom rezultuje novim stringom dok originalni ostaje nepromenjen. Primer u nastavku ilustruje kreiranje i operacije nad stringom, poput konkatencije (operator +).

```
let mojString = 'primer';
mojString += 'stringa';
mojString = mojString.toUpperCase();
console.log("Sadržaj stringa: " + mojString);
```

Šablonski string (eng. *template string*) jeste niska znakova sa dinamičkim sadržajem, odnosno unapred definisanim delovima promenljivog sadržaja. Sadržaj može biti i izraz čija se vrednost izračunava i prikazuje. Sintaksa za navođenje šablonskog stringa je sledeća:

```
`sadržaj stringa ${izraz} sadržaj stringa`
```

Simbol \$ i vitičaste zagrade ({}) označavaju mesto za definisanje promenljivog sadržaja stringa (eng. *placeholder*). Važno je napomenuti da se pri navođenju ove vrste stringa ne koriste znaci navoda već *back-tick* simboli (`). Prilikom prikazivanja stringa računa se vrednost izraza, konvertuje u string i spaja sa nepromenljivim delovima navedenim pre i posle izraza.

U primeru ispod, šablonski string ispisuje rezultat operacija sabiranja vrednosti varijabli *a* i *b*.

```
var a = 10;
var b = 10;
console.log(`Zbir a i b je ${a + b} a ne ${2 * a + b}.`);
```

Console

"Zbir a i b je 20 a ne 30."

Standardni operatori nad vrednostima logičkog tipa su Logičko I (&&), Logičko ILI (||), i Negacija (!). Vrednosti logičkog i drugih tipova se mogu koristiti u proveru jednakosti ili nejednakosti u uslovnim naredbama. U tom slučaju važe specifična pravila u zavisnosti od operatora koji se primenjuju.

Ukoliko se koriste starije verzije operatora jednakosti, odnosno == (jednako) i != (različito) primenjuju se pravila u datom redosledu:

- Null, undefined, 0, NaN, i string bez sadržaja ("", "") se posmatra kao false vrednost.
- Vrednosti svih tipova se tretiraju kao true vrednost.
- Porede se vrednosti operanada.

Sa obzirom da se pre poređenja vrši opisana konverzija tipova, mogu se dobiti rezultati kao ispod. Komentari označavaju rezultat poređenja.

```
' ' == '0' //false
' ' == 0 //true
0 == '0' //true
20 == '20' //true
[10, 20] == '10,20' //true
NaN == NaN //false
[''] == '' //true
false == undefined //false
false == null //false
null == undefined //true
```


Može se videti da su za specifične vrednosti operanada rezultati neočekivani i neusklađeni sa logikom.

Zbog toga se za ovaj metod poređenja koristi termin slaba jednakost (eng. *loose equality*).

Preporuka je koristiti noviju verziju operatora jednakosti, tj. `===` (jednako) i `!==` (različito). Tada ne postoje specifične konverzije između tipova vrednosti kao u prethodnoj i pravila poređenja su sledeća:

- Vrednosti različitih tipova nisu jednake.
- Vrednosti istog tipa se porede u skladu sa pravilima za konkretan tip.

U ovom slučaju rezultati poređenja jednakosti izgledaju kao u primeru.

```
' ' === '0' //false
' ' === 0 //false
0 === '0' //false
[''] === '' //false
false === undefined //false
false === null //false
null === undefined //false
```

Navedeni metod poređenja jednakosti poznat je i pod nazivom striktna jednakost (eng. *strict equality*). Slika ispod daje uporedni prikaz poređenja jednakosti korišćenjem dva opisana metoda.

x	y	==	===
undefined	undefined	true	true
null	null	true	true
true	true	true	true
false	false	true	true
'foo'	'foo'	true	true
0	0	true	true
+0	-0	true	true
+0	0	true	true
-0	0	true	true
0	false	true	false
""	false	true	false
""	0	true	false
'0'	0	true	false

Slika 4.1. Poređenje JavaScript vrednosti korišćenjem metoda slabe i striktnje jednakosti. Izvor [56].

Tip objekat opisuje vrednosti kao kolekcije svojstava (eng. *property*) gde se svako svojstvo sastoji od imena (eng. *name*) i vrednosti (eng. *value*). Ime se navodi kao tekst, dok vrednost može biti proizvoljnog tipa, uključujući i funkciju. U slučaju da je vrednost funkcija, za svojstvo se koristi i naziv metod (eng. *method*). U donjem primeru, kreiraju se dva objekta i dodeljuju varijablama *obj1* i *obj2*, respektivno.

```
var obj1 = {};
var obj2 = {ime: "Marija",
            starost: 22,
            zemlja: "Srbija"};
```

Vrednostima svojstava iznad kreiranih objekata se može pristupiti na dva načina. Prvi način je kao strukturi podataka korišćenjem sintakse *ime_objekta.ime_svojstva*. Drugi način pristupa je kao heš (eng. *hash*) tabeli sa ključevima, odnosno pomoću sintakse *ime_objekta["ime_svojstva"]*.

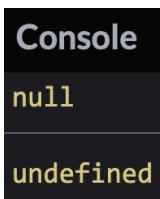
Korišćenjem navedenih sintaksi svojstvima se mogu dodeljivati nove vrednosti ili se mogu dodavati nova svojstva sa vrednostima. U primeru ispod varijabli *obj1* se dodaje vrednost tipa Objekat bez svojstava. Zatim se objektu dodaje svojstvo *ime* sa vrednošću. Na kraju se svojstvo briše iz objekta pomoću naredne *delete*.

```
var obj1 = {};  
  
obj1.ime = "Dragana"; //ili obj1[ime] = "Dragana"  
  
delete obj1.ime;
```

Više informacija u vezi sa kolekcijom objekat može se naći u [57].

Nulti tip definiše nultu vrednost, dok tip nedefinisano opisuje slučaj u kojem varijabli nije dodeljena vrednost. U primeru je ilustrovano kreiranje i prikaz vrednosti varijabli dva tipa.

```
var x = null;  
var y;  
console.log(x);  
console.log(y);
```

A screenshot of a console window with a dark background. The word "Console" is at the top in white. Below it, the word "null" is displayed in yellow, and further down, the word "undefined" is also displayed in yellow.

Niz (eng. *array*) predstavlja indeksiranu kolekciju u kojoj su vrednosti uređeni po unapred definisanom redosledu, odnosno indeksu. Drugim rečima, niz je uređena lista vrednosti kojima se pristupa navođenjem imena niza i indeksa vrednosti. Početna vrednost indeksa je nula. Vrednosti mogu biti prethodno opisanih tipova. Dužina niza je promenjiva i vrednosti elemenata niza mogu biti različitih tipova.

Primer pokazuje različite načine kreiranja niza. Prvi primer koristi uglaste zagrade (`[]`) za navođenje elemenata niza. Naredni primeri koriste predefinisani objekat Niz (eng. *Array*) koji sadrži metode za rad sa nizovima.

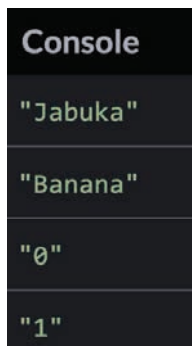
```
var niz1 = [elem0, elem1, /* ... ,*/ elemN];  
var niz2 = new Array(elem0, elem1, /* ... ,*/ elemN);  
var niz3 = Array(elem0, elem1, /* ... ,*/ elemN);  
var niz4 = new Array(dužinaNiza);
```

Ispod su prikazani kreiranje niza, ispisivanje dužine niza (atribut *length*), i pristup elementima niza kao dodela vrednosti odgovarajućim varijablama.

```
var voće = ['Jabuka', 'Banana'];  
console.log(voće.length);  
  
var prvi = voće[0];  
var poslednji = voće[voće.length - 1];
```

Postoje dva predefinisana načina pristupa elementima niza. Naredba oblika *for...let...of* vraća vrednost elementa dok varijanta naredbe *for...let...in* daje poziciju (ili indeks) elementa. Primeri u nastavku pokazuju rezultat korišćenja dve naredbe.

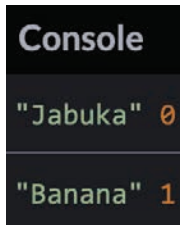
```
for(let vrsta of voće) {  
    console.log(vrsta); }  
  
for(let indeks in voće) {  
    console.log(indeks); }
```



```
Console  
"Jabuka"  
"Banana"  
"0"  
"1"
```

Niz sadrži *forEach()* metodu koja omogućava kretanje (ili iteraciju) kroz elemente niza. Funkcija se poziva nad kreiranim nizom i može imati parametre kao u donjem primeru. U svakoj iteraciji pristupa elementima niza u redosledu indeksa (od prvog do poslednjeg). Značenje prvog parametra je vrednost tekućeg elementa niza, dok drugi parametar označava indeks tekućeg elementa niza.

```
voće.forEach(function(element, indeks) {  
  console.log(element, indeks);  
});
```



Za dodavanje i uklanjanje elemenata niza mogu se koristiti dva para metoda u zavisnosti od toga na kojim pozicijama se menja sadržaj niza.

Metoda *push()* dodaje jedan ili više elemenata na kraj niza. Kao parametre prima vrednosti elemenata i vraća novu dužinu niza. Metoda *pop()* uklanja element sa kraja niza i daje vrednost uklonjenog elementa.

```
var novaDužina = voće.push('Pomorandža');  
var poslednji = voće.pop();
```

Metoda *shift()* uklanja prvi element niza i vraća vrednost tog elementa. Metoda *unshift()* dodaje jedan ili više elemenata na početak niza. Kao parametre prihvata vrednosti elemenata i daje novu dužinu niza.

```
var prvi = voće.shift();  
var novaDužina = voće.unshift('Jagoda');
```

Iz niza se može izdvojiti i sekvenca elemenata, tj. podniz. Metoda *splice()* uklanja jedan ili više elemenata niza i vraća podniz koji sadrži uklonjene elemente. U primeru ispod najpre se kreira i prikazuje sadržaj niza *povrće*. Zatim se iz niza uklanjaju dva elementa počevši od indeksa sa brojem jedan i dodeljuju kao vrednost promenljivoj *uklonjeniElementi*. Na kraju se ispisuju sadržaji dva niza.

```
var povrće = ['Kupus', 'Krompir', 'Luk', 'Šargarepa'];  
console.log(povrće);  
  
var pozicija = 1, n = 2;  
  
var uklonjeniElementi = povrće.splice(pozicija, n);
```

```
console.log(povrće);  
console.log(uklonjeniElementi);
```

Console

```
// [object Array] (4)  
["Kupus", "Krompir", "Luk", "Šargarepa"]  
  
// [object Array] (2)  
["Kupus", "Šargarepa"]  
  
// [object Array] (2)  
["Krompir", "Luk"]
```

Detaljnija pojašnjenja nizova u JavaScript jeziku se mogu naći u [58].

4.2. Objektno-orijentisani JavaScript

Elementi objektno-orijentisanog programiranja, poput apstrakcije, enkapsulacije i nasleđivanja, se mogu realizovati na više načina. U nastavku su opisi postojećih načina u JavaScript jeziku potkrepljeni primerima.

4.2.1. Objekti i funkcije

Objektno-orijentisani elementi se mogu implementirati korišćenjem tipa objekat. U primeru ispod objekat *o* se kreira tako da sadrži svojstva *broj* čija je vrednost tipa broj i *inkrement* čija je vrednost tipa funkcija. Na ovaj način se svojstva mogu posmatrati kao atribut i metoda instance objekta, respektivno. Kod prvog poziva funkcije ne prosleđuje se argument i *broj* dobija vrednost jedan, dok se kod drugog postojeća vrednost *broj* uvećava za navedenu vrednost argumenta.

```
var o = { broj: 0 };  
o.inkrement = function(iznos) {  
  if (iznos === undefined) {  
    iznos = 1; }  
  this.broj += iznos;  
  return this.broj;  
};  
console.log(o.inkrement());  
console.log(o.inkrement(5));
```

Console

1

6

U sledećem primeru objekat *o* se definiše tako da sadrži svojstvo tipa string. Zatim se objektu dodaje svojstvo tipa funkcija (*metod*) i okviru funkcije se definiše novo svojstvo objekta *o* (*novoSvoj*). Objekat *Object* je predstavlja tip iz kog su izvedeni drugi tipovi JavaScript jezika. Sadrži svojstva koja omogućavaju nasleđivanje između objekata i metode za čuvanje i prikaz kolekcija kao što je *keys()*. U ovom slučaju pokazivač *this* ukazuje na instancu tekućeg objekta (*o*), slično kao u instancama klasa kod objektno-orientisanih jezika. U ovom primeru metoda *keys()* kao rezultat vraća nazive svojstava objekta-parametra kao ključeve.

```
var o = { posSvoj: "postojeće svojstvo" };
o.metod = function() {
  this.novoSvoj = "novo svojstvo";
  return Object.keys(this);
};
console.log(o.metod());
```

Console

```
// [object Array] (3)
["posSvoj", "metod", "novoSvoj"]
```

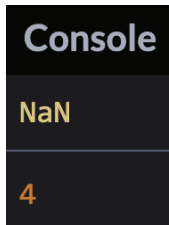
Naredni primer u okviru funkcije kao objekta prvog reda (*plus1*) definiše svojstvo *pozivi*. Prilikom prvog poziva funkciji se dodaje svojstvo sa vrednošću nula i pri svakom sledećem pozivu vrednost svojstva se poveća za jedan. Rezultat *NaN* kod prvog poziva se javlja kao posledica pokušaja sabiranja vrednosti *undefined* (funkciji nije prosleđen parametar) i vrednosti 1.

```
function plus1(vrednost) {
  if (plus1.pozivi === undefined) {
    plus1.pozivi = 0;
  }
  plus1.pozivi++;
  return vrednost + 1;
}
```

```

}
console.log(plus1());
console.log(plus1(3));

```



Funkcija *Pravougaonik* se može posmatrati kao definicija klase. Sadrži dva svojstva kao atribut (*sirina* i *visina*) i svojstvo čija je vrednost anonimna funkcija. Funkcija se kreira kao objekat *p* korišćenjem naredbe *new* i imena funkcija sa parametrima slično kao kod konstruktora klase. *This* ukazuje na tekući objekat funkciju. Svojstvo *constructor* je nasleđeno iz tipa *Object*. Predstavlja objekat koji ima atribut *name* kao ime konstruktora, odnosno funkcije u ovom slučaju. Drugi poziv *log()* funkcije prikazuje sadržaj prethodno kreiranog objekta funkcije *p*.

```

function Pravougaonik(sirina, visina) {
  this.sirina = sirina;
  this.visina = visina;
  this.povrsina = function() { return this.sirina *
this.visina; };
}
var p = new Pravougaonik(30, 20);

console.log(p.constructor.name);
console.log(p);

```

Console

"Pravougaonik"

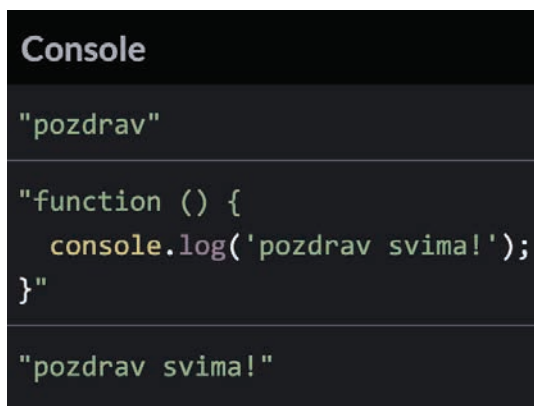
```

// [object Object]
{
  "sirina": 30,
  "visina": 20,
  "povrsina": "function () {return this.sirina * this.visina;}"
}

```


Funkcija u JavaScript jeziku je objekat tipa *Function*. Kada se definiše funkcija, kreira se objekat tipa *Function* čiji je sadržaj programski kod funkcije. Tip sadrži atribut i metode kao što su *name* (ime funkcije), *toString()* (string reprezentacija funkcije), i *call()* (izvršava funkciju). Sledeći primer ilustruje njihovo korišćenje. Anonimna funkcija je definisana i dodeljena kao vrednosti varijabli *pozdrav*.

```
const pozdrav = function() {  
  console.log('pozdrav svima');  
};  
  
console.log(pozdrav.name);  
console.log(pozdrav.toString());  
pozdrav.call();
```



```
Console  
"pozdrav"  
"function () {  
  console.log('pozdrav svima!');  
}"  
"pozdrav svima!"
```

Kao objekti prvog reda i konkretnog tipa funkcije se mogu prosleđivati kao parametri drugih funkcija. Tada se još nazivaju i *callback* funkcije. Primer predstavlja registrovanje osluškivača događaja gde se ime funkcije prosleđuje kao parametar *addEventListener()* metodi.

Primeri u nastavku prikazuju različite načine definisanja funkcije.

Prva funkcija prihvata *callback* funkciju kao argument (*pozvanaFunkcija*) i poziva prosleđenu funkciju. Funkcije u nastavku se definišu kao anonimna i *arrow* funkcija, respektivno. Na kraju se dve funkcije prosleđuju kao parametri početne funkcije koja ih izvršava.

```
function imeFunkcije(pozvanaFunkcija) {  
  pozvanaFunkcija();  
}
```

```

var varijablaFunkcije = function() {
  console.log('funkcija kao varijabla');
};

var arrowFunkcija = () => {
  console.log('arrow funkcija');
};

imeFunkcije(varijablaFunkcije);
imeFunkcije(arrowFunkcija);

```

Console

"funkcija kao varijabla"

"arrow funkcija"

Sledeći primer definiše *callback* funkciju bez argumenata koja prikazuje poruku u *alert* prozoru za poruke. *testFunkcija* kao argument prihvata funkciju koju dalje poziva. Rezultat je kao na slici.

```

//Imenovana funkcija kao callback
function imenovaniCallback(){
  alert("imenovaniCallback()");
}

function testFunkcija(callback){
  callback();
}

testFunkcija(imenovaniCallback);

```

An embedded page at cdpn.io says

imenovaniCallback()

OK

Naredni primer definiše *callback* funkciju sa argumentom čiju vrednost prikazuje u *alert* prozoru. Kao parametar testne funkcije prosleđuje se poziv *callback* funkcije. Kao što se može videti prikazuje se tekst sa vrednošću parametra prosleđenog preko varijable *pozivalac*.

```
//Imenovana callback funkcija sa parametrima
var pozivalac = "pozivalac";
function imenovaniCallback(parametar){
    alert("imenovaniCallback() pozvan sa " + parametar);
}

function testFunkcija(callback){
    callback();
}

testFunkcija(imenovaniCallback(pozivalac));
```

An embedded page at cdpn.io says

imenovaniCallback() pozvan sa pozivalac

OK

Primer ispod kombinuje argumente funkcije kao string vrednost (prvi argument) i *callback* funkciju (drugi argument). Testna funkcija prikazuje vrednost prvog argumenta u *alert* prozoru, proverava da li je drugi argument prosleđen i tipa funkcije, i poziva *callback* funkciju. Prvi *alert* prozor ispisuje vrednost prvog parametra, dok drugi ispisuje vrednost iz *callback* funkcije kao drugog parametra.

```
// Kombinacija callback funkcije i parametra kao
promenljive
var poruka = "zdravo svima!";
function callbackFunkcija(){
    alert("callbackFunkcija je pozvana.");
}
function testCallback(parametar, callback){
    alert(parametar);
    if(callback && typeof(callback) === "function"){
```

```
        callback();  
    }  
}  
testCallback(poruka, callbackFunkcija);
```

An embedded page at cdpn.io says
zdravo svima!

OK

An embedded page at cdpn.io says
callbackFunkcija je pozvana.

OK

Prethodni primeri pokazuju kako se elementi objektno-orijentisanog programiranja mogu realizovati korišćenjem kolekcije objekata i funkcije kao objekta prvog reda. U nastavku je opisana implementacija mehanizma nasleđivanja.

4.2.2. Prototipi

Prototipi omogućavaju nasleđivanje osobina između JavaScript objekata na sledeći način.

Tip Objekat ima atribut prototip (*Object.prototype*). Svaki objekat je instanca Objekat tipa i sadrži dati atribut. Vrednost prototip atributa tekućeg objekta se inicijalizuje prilikom kreiranja objekta i ukazuje na objekat pomoću kojeg je kreiran dati objekat. Kaže se da je taj objekat prototip na osnovu kojeg je kreiran posmatrani, tj. tekući objekat. Na ovaj način svaki kreirani objekat ima svoj prototip te se na taj način formira lanac prototipa (eng. *prototype chain*). Lanac prototipa se proteže do objekta prototipa čiji atribut prototip ima vrednost *null*.

Opisani mehanizam predstavlja osnovu nasleđivanja. Objekti predstavljaju dinamički skup atributa gde se novi atributi se mogu dodeljivati objektima, tj. instancama u toku njihovog postojanja. Kada se pristupa atributu objekta, atribut će najpre biti tražen u tom objektu, zatim u njegovom prototipu, prototipu prototipa i dalje sve dok se atribut ili ne pronađe ili se ne dođe do kraja lanca prototipa. Atributi objekta se mogu posmatrati kao oni koje on

sadrži i atributi objekata u lancu prototipa kojem dati objekat pripada. Na ovaj način se nasleđivanje zasniva na prototipima.

Naredni primer ilustruje korišćenje prototip atributa funkcije. Funkcija *Pravougaonik* je definisana sa dva svojstva (*sirina* i *visina*). Zatim se preko atributa prototip funkcije dodaje svojstvo *povrsina* kao funkcija. Objekat funkcije se kreira sa parametrima korišćenjem naredbe *new* i dodeljuje kao vrednost promenljivoj *p*. Svojstvo površina se poziva i izvršava kao funkcija i dodeljuje kao vrednost promenljivoj *v*. Na kraju se ispisuje sadržaj promenjive *p* kao objekta funkcije i promenjive *v* kao površine pravougaonika, respektivno.

```
function Pravougaonik(sirina, visina) {
  this.sirina = sirina;
  this.visina = visina;
}
Pravougaonik.prototype.povrsina = function()
{
  return this.sirina*this.visina;
}
var p = new Pravougaonik(30, 20);
var v = p.povrsina();
console.log(p);
console.log(v);
```

Console

```
// [object Object]
{
  "sirina": 30,
  "visina": 20,
  "povrsina": "function () {return this.sirina * this.visina;}"
}
```

600

Primeri koji slede pokazuju pravila nasleđivanja u lancu prototipa JavaScript objekata.

Funkcija *Auto* je definisana bez svojstava. Korišćenjem atributa prototip funkciji su dodata tri svojstva sa vrednostima (*tockovi*, *volan* i *boja*). Zatim se kreiraju dva objekta funkcije (*auto1* i *auto2*). Potom se pokušava pristupiti i ispisati vrednost dva svojstva u kreiranim objektima (*auto1.stereo* i

auto2.gps). Sa obzirom da svojstva nisu prethodno definisana, prikazuje se vrednost *undefined*. Nakon toga se ispisuju vrednosti svojstava definisanih u prototip objektu funkcije za obe kreirana objekta funkcije. Može se videti da su kreirani objekti funkcije *auto1* i *auto2* nasledili svojstva sa vrednostima iz prototipa funkcije *Auto*. Odavde proizilazi pravilo da *objekti deca nasleduju svojstva sa vrednostima iz njihovih roditelja ili prototipa*.

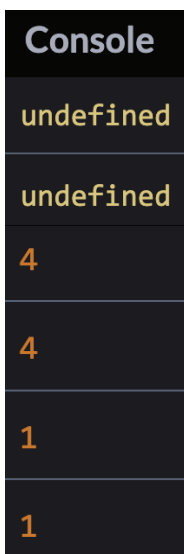
```
function Auto() {}

Auto.prototype.tockovi = 4;
Auto.prototype.volan = 1;
Auto.prototype.boja = "metalik";

var auto1 = new Auto();
var auto2 = new Auto();

console.log(auto1.stereo);
console.log(auto2.gps);

console.log(auto1.tockovi);
console.log(auto2.tockovi);
console.log(auto1.volan);
console.log(auto2.volan);
```



Console
undefined
undefined
4
4
1
1

U sledećem primeru objekti funkcije se kreiraju na isti način kao u prethodnom. Zatim se svojstvu *tockovi* objekta *auto1* dodeljuje nova vrednost. Vrednost je prikazana kao prvi rezultat. Vrednosti svojstva za preostala dva

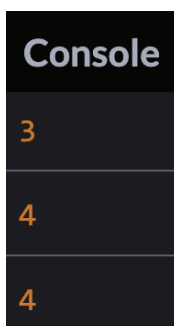
objekta su one navedene u objektu *Auto*. Rezultat izgleda skladu sa pravilima nasleđivanja gde objekat dete dete (*auto1*) može redefinisati vrednost svojstva roditelja (*Auto*). Međutim, u ovom slučaju objekat dete ne menja vrednost postojećeg atributa roditelja ili prototipa. Pravilo glasi da *objekti deca ne mogu menjati vrednosti atributa roditelja ili prototipa, već se kreira novi atribut deteta sa istim imenom i navedenom vrednošću*.

```
function Auto() {}

Auto.prototype.tockovi = 4;
Auto.prototype.volani = 1;
Auto.prototype.boja = "metalik";

var auto1 = new Auto();
var auto2 = new Auto();

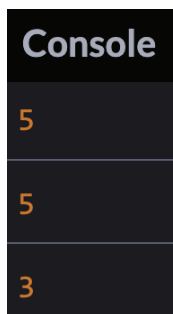
auto1.tockovi = 3;
console.log(auto1.tockovi);
console.log(auto2.tockovi);
console.log(Auto.prototype.tockovi);
```



Primer koji sledi kreira identične objekte funkcije kao raniji primeri. Svojstvu *tockovi* objekta *auto1* se dodeljuje nova vrednost i objektima *auto1* i *auto2* nova svojstva *stereo* i *gps*, respektivno. Svojstvo *tockovi* dobija novu vrednost, ali u prototipu objekta *Auto*. Prikaz vrednosti ovog svojstva u objektima funkcije pokazuje da je vrednost atributa ažurirana samo u objektu *auto2*, dok je objekat *auto1* zadržao postojeću vrednost. Zaključak je da se *promene vrednosti svojstava u roditelju ili prototipu odražavaju na vrednosti ažuriranih svojstava objekata dece*. Izuzetak su svojstva čije su vrednosti prethodno ažurirane u objektima deci (*auto1.tockovi*).

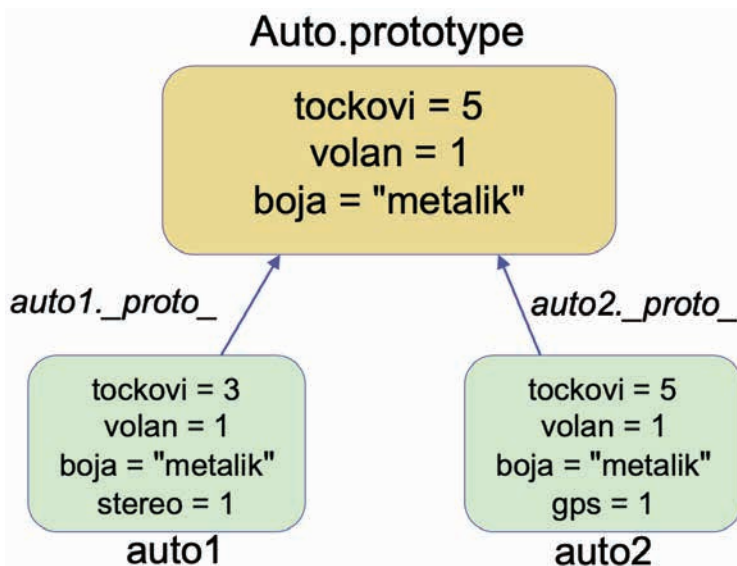
```
function Auto() {}
```

```
Auto.prototype.tockovi = 4;  
Auto.prototype.volan = 1;  
Auto.prototype.boja = "metalik";  
  
var auto1 = new Auto();  
var auto2 = new Auto();  
  
auto1.tockovi = 3;  
  
auto1.stereo = 1;  
auto2.gps = 1;  
  
Auto.prototype.tockovi = 5;  
console.log(Auto.prototype.tockovi);  
console.log(auto2.tockovi);  
console.log(auto1.tockovi);
```



Console
5
5
3

Slika 4.2. ilustruje strukturu i sadržaj objekata funkcije korišćenih u prethodnim primerima prototipa.



Slika 4.2. Šematski prikaz objekata funkcije sa vrednostima svojstava u lancu prototipa.

Svaka funkcija se može posmatrati kao objekat gde postoji korena roditeljska instanca (*prototype*) i izvedeni objekti ili deca. Instance su međusobno povezane pomoću prototip svojstva i formiraju hijerarhiju u obliku lanca prototipa. Sekvenca instanci koje čine lanac počinje od korene instance i nastavlja ka deci po redosledu njihovog kreiranja. Ažuriranje objekta u lancu će se odraziti samo na objekte na nižim nivoima, tj. u nastavku datog lanca. Na primer, dodavanje i uklanjanje svojstava kao atributa i funkcija, ili promene vrednosti svojstava.

Detalji prototipa u JavaScript jeziku se mogu naći u [59].

4.2.3. Klase

Klase kao složeni i apstraktni tipovi podataka postoje u jeziku [60]. Primer u nastavku daje strukturu JavaScript klase sa tipičnim elementima. Klasa se definiše korišćenjem ključne reči *class*. Klasa može naslediti drugu klasu pomoću ključne reči *extends*. Konstruktor klase se navodi pomoću ključne reči *constructor*, dok se *super* koristi za inicijalizaciju konstruktora roditeljske klase. *This* ukazuje na tekuću instancu klase. Statički atributi i metode se označavaju ključnom rečju *static*. Instanca ili objekat klase se kreira pomoću ključne reči *new*.

```
class Pravougaonik extends Oblik {
  constructor(visina, sirina) {
```

```

        super(visina, sirina);
        this.visina = visina;
        this.sirina = sirina;
    }
    površina() {
        return this.sirina * this.visina;
    }
    static brojInstanci() {
        // ...
    }
}
var p = new Pravougaonik(10,20);

```

Sledeći primer apstraktni tip *Auto* navodi kao definiciju klase. Ključne reči *get* i *set* se koriste za definisanje pristupnih (eng. *accessor*) metoda atributima klase. *Get* vraća vrednost atributa, dok *set* postavlja vrednost datog atributa. Definisanim atributima klase se automatski dodaje prefiks *_* (eng. *underscore*). Zbog toga se atributima nadalje pristupa dodavanjem prefiksa, kao u pristupnim metodama ispod. Prikaz sadržaja instance klase *autoInstanca* pokazuje da je simbol *_* dodat nazivima atributa klase *Auto*.

```

//Definicija klase
class Auto {
    constructor(tockovi, volan, boja) {
        this.tockovi = tockovi;
        this.volan = volan;
        this.boja = boja;
    }

    get tockovi() { return this._tockovi; }
    set tockovi(tockovi) { this._tockovi = tockovi; }

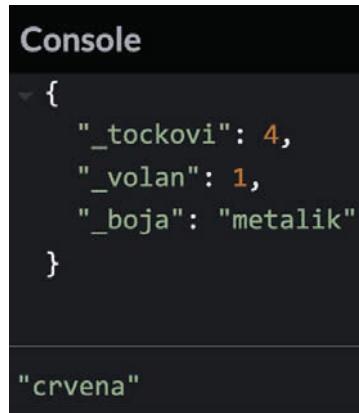
    get volan() { return this._volan; }
    set volan(volan) { this._volan = volan; }

    get boja() { return this._boja; }
    set boja(boja) { this._boja = boja; }
}

//Instanciranje klase
var autoInstanca = new Auto(4,1,'metalik');
console.log(autoInstanca);

```

```
//Koriscenje get/set metoda objekta klase
autoInstanca.boja = 'crvena';
console.log(autoInstanca.boja);
```



Kod nasleđivanja u konstruktoru izvedene klase se poziva konstruktor roditeljske klase (ključna reč *super*) sa parametrima za inicijalizaciju roditelja. U primeru ispod klasa *Auto1* nasleđuje klasu *Auto* i dodaje atribut *stereo*. Metoda *autoInfo()* prikazuje vrednosti prethodno definisanih atributa klase.

```
class Auto1 extends Auto {
  constructor(tockovi, volan, boja, stereo) {
    super(tockovi, volan, boja);
    this.stereo = stereo;
  }

  get stereo() { return this._stereo; }
  set stereo(stereo) { this._stereo = stereo; }

  autoInfo() {
    console.log("Tockovi:" + this._tockovi +
      " volan:" + this._volan +
      " boja:" + this._boja +
      " stereo:" + this._stereo)
  }
}

var auto1Instanca = new Auto1(4,1,'crvena',1);
auto1Instanca.autoInfo();
```

Console

```
"Tockovi:4 volan:1 boja:crvena stereo:1"
```

Važno je napomenuti da je mehanizam nasleđivanja klasa u pozadini implementiran kao nasleđivanje objekata prototipa. Više informacija u vezi sa tim se nalazi u Sekciji 4.5.4.

4.3. Elementi funkcionalnog programiranja

Funkcionalno programiranje je zasnovano na ideji da se računarski program tretira kao matematička funkcija. Na ovaj način je programski kod većinom izraz i eksplicitno navođenje stanja programa (varijable, polja, objekti i slično) je redukovano što je više moguće.

Zbog toga većina programskih jezika implementira elemente funkcionalnog programiranja uz postojeće elemente, uključujući proceduralne i objektno-orijentisane.

JavaScript jezik elemente funkcionalnog programiranja realizuje prvenstveno uvođenjem *arrow* funkcija. Ovaj pristup može obezbediti sažetiji način za definisanje funkcije u odnosu na proceduralni pristup. Naziv potiče od korišćenja *flat arrow* simbola ($\Rightarrow$) kao operatora za definisanje funkcije. U nastavku je dat pregled varijanti sintaksi koje definišu *arrow* funkciju. Princip je da se argumenti funkcije navode sa leve strane *flat arrow* operatora u običnim zagradama ($()$), dok se telo funkcije definiše sa desne strane operatora u vitičastim zagradama ($\{\}$). Ukoliko postoji samo jedan argument ne mora se navoditi u zagradama. Slično tome, kada je izraz jedna naredba ne moraju se koristiti zagrade. Ako funkcija nema argumente, moraju se koristiti obične zagrade.

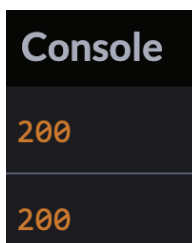
```
argument => izraz
(argument) => izraz
argument => { izrazi }
(argument1, argument2, ..., argumentN) => { izrazi }
(argument1, argument2, ..., argumentN) => izraz
//ekvivalentno sa: => { return izraz; }
//Spisak argumenata može biti prazan
() => { izrazi }
```

U nastavku su primeri varijanti sintaksi *arrow* funkcija. Radi boljeg razumevanja, svaki primer daje uporedan prikaz identične funkcionalnosti definisane na proceduralan način i kao *arrow* funkcije.

Primer ispod ilustruje sintaksu sa većim brojem argumenata. Može se primetiti da se *arrow* funkcija definiše kao anonimna, odnosno dodeljuje kao vrednost varijabli. Naziv varijable se koristi kao ime funkcije prilikom njenog pozivanja.

```
//Osnovna sintaksa sa višestrukim argumentima
//Proceduralan način
var mnozenjeProc = function(x, y) {
    return x * y;
};
console.log(mnozenjeProc(10,20));

//Arrow funkcija
let mnozenjeArrow = (x, y) => { return x * y; };
console.log(mnozenjeProc(10,20));
```



Sledeći primer pokazuje *arrow* funkciju sa jednim argumentom. Funkcija *split()* obrađuje string tako što ga deli skup jednostavnijih string elemenata na osnovu graničnog simbola (eng. *delimiter*) kao parametra.

```
//Osnovna sintaksa sa jednim argumentom
//Proceduralan način
var frazaPodelaProc = function frazaPodeli(fraza) {
    return fraza.split(' ');
};
console.log(frazaPodelaProc("Osnovna sintaksa sa jednim parametrom"));

//Arrow funkcija
var frazaPodelaArrow = fraza => fraza.split(" ");
console.log(frazaPodelaArrow("Osnovna sintaksa sa jednim parametrom"));
```

Console

```
// [object Array] (5)
["Osnovna","sintaksa","sa","jednim","parametrom"]

// [object Array] (5)
["Osnovna","sintaksa","sa","jednim","parametrom"]
```

Kod *arrow* funkcije koja nema argumente uvek se koriste obične zagrade sa leve strane *flat arrow* operatora, kao i u primeru ispod.

```
//Osnovna sintaksa bez argumenata
//Proceduralan način
var dokLogProc = function dokLog() {
    console.log(document);
};

//Arrow funkcija
var dokLogArrow = () => { console.log(document); };
dokLogArrow(); // #document... <html> ...
```

Čest slučaj korišćenja *arrow* funkcija jesu operacije sa nizovima. Tri tipične operacije uključuju:

- Mapiranje (eng. *map*) kao izdvajanje vrednosti kao niza iste dužine iz početnog niza složenijih elemenata.
- Filter (eng. *filter*) predstavlja izdvajanje podniza vrednosti iz niza elemenata koji zadovoljavaju navedeni kriterijum, tj. filter.
- Redukcija (eng. *reduce*) podrazumeva agregaciju vrednosti elemenata niza.

Sa obzirom na veliku zastupljenost operacija, JavaScript implementira operacije kao odgovarajuće funkcije. Svaka od operacija se može navesti i na proceduralan i funkcionalan način. Zajedničko za sve funkcije je da imaju ugrađen mehanizam petlje koji obezbeđuje kretanje kroz niz. Na taj način se u telu funkcije dodaju samo željene operacije nad elementima niza.

Svaka operacija je ilustrovana primerom koji uključuje proceduralnu i funkcionalnu implementaciju logike radi boljeg razumevanja *arrow* funkcija.

Sintaksa *arrow* funkcije mapiranja je prikazana ispod. Funkcija iz svakog objekta izvornog niza izdvaja svojstvo i smešta kao element mapiranog niza koji je ujedno i povratna vrednost.

```
mapiran_niz = originalni_niz.map(objekat =>
objekat.svojstvo)
```

Mapiranje je ilustrovano na primeru niza *uređaji* čiji su elementi objekti sa dva svojstva, *model* i *cena*. Cilj je izdvojiti cene modela u odvojen niz. Može se videti da *arrow* funkcija pruža pregledniju definiciju zahtevane funkcionalnosti u odnosu na proceduralan pristup. Funkcija se poziva se nad nizom *uređaji* i povratna vrednost dodeljuje varijabli *ceneArrow*.

```
//Mapiranje nizova
const uređaji = [
  { model:'iphone', cena:550 },
  { model:'Galaxy S8', cena:660 },
  { model:'Galaxy Note 7', cena:770 }
];

//Proceduralan način
var ceneProc = uređaji.map(function(uređaj) {
  return uređaj.cena;
});
console.log(ceneProc);

//Arrow funkcija
var ceneArrow = uređaji.map(uređaj => uređaj.cena);
console.log(ceneArrow);
```

Console

```
// [object Array] (3)
[550,660,770]
```

```
// [object Array] (3)
[550,660,770]
```

Šema *arrow* funkcije *filter* je u nastavku. Iz originalnog niza se izdvajaju elementi koji zadovoljavaju navedeni uslov u filtriran niz povratnu vrednost.

```
filtriran_niz = originalni_niz.filter(element => uslov)
```

Filter funkcija je u primerima ispod. Niz sadrži brođane vrednosti. Potrebno je elemente deljive sa tri izdvojiti u podniz. Arrow funkcija i u ovom slučaju daje

sažetiji opis tražene funkcionalnosti. Poziva se nad originalnim nizom i izdvojeni podniz dodeljuje kao vrednost varijabli *filterArrow*.

```
//Filtriranje nizova
const niz = [11, 12, 13, 14, 15, 16,
             17, 18, 19, 20, 21,
             22, 23, 24, 25];

//Proceduralan način
var filterProc = niz.filter(function (vrednost){
  return vrednost % 3 === 0;
});
console.log(filterProc);

//Arrow funkcija
var filterArrow = niz.filter(vrednost => vrednost % 3 ===
0);
console.log(filterArrow);
```

Console

```
// [object Array] (5)
[12,15,18,21,24]
```

```
// [object Array] (5)
[12,15,18,21,24]
```

Struktura funkcije redukcije je data ispod. Iz sadržaja, tj. vrednosti elemenata niza se izvodi skalarna vrednost. Argumenti funkcije nose predefinisano značenje. U početnoj iteraciji, *ukupno* ima vrednost prvog elementa niza, *tekuće* ima vrednost drugog elementa. U svakoj narednoj iteraciji drugi parametar (*tekuće*) prima vrednost sledećeg elementa niza i dodaje prvom parametru (*ukupno*).

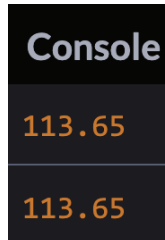
```
vrednost = originalni_niz.reduce((ukupno,tekuće) =>
izraz sa ukupno i tekuće)
```

Funkcija redukcije u narednom primeru računa sumu elemenata niza *iznosi*.

```
//Redukcija nizova
```



```
var iznosi = [27.85, 37.85, 47.95];  
//Klasičan način  
var sumaProc = iznosi.reduce(function(ukupno, tekuće){  
    return ukupno + tekuće });  
console.log(sumaProc);  
//Arrow funkcija  
var sumaArrow = iznosi.reduce((ukupno, tekuće) => ukupno  
+ tekuće);  
console.log(sumaArrow);
```



Više informacija u vezi sa *arrow* funkcijama se može naći u [61].

4.4. Interaktivni elementi JavaScript jezika

Bazične strukture podataka JavaScript jezika koje realizuju interakciju korisnika sa Veb GKI su hijerarhija stabla objekata i događaji.

4.4.1. Stablo objekata

Elementi svake Veb stranice, odnosno HMTL (eng. *HyperText Markup Language*) dokumenta u vreme prikazivanja su predstavljeni kao hijerarhija objekata tipa stabla u memoriji. Ovakva organizacija je poznata pod nazivom DOM (eng. *Document Object Model*) stablo. Stablo je skup objekata koji čine strukturu i sadržaj dokumenta Veb stranice.

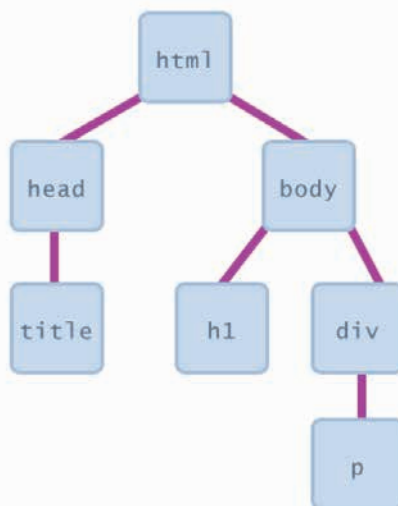
Istovremeno DOM obezbeđuje i aplikacioni programski interfejs (eng. *Application Programming Interface - API*) Veb stranica. Drugim rečima, iz JavaScript programa se može pristupiti stablu i menjati njegova struktura, sadržaj i izgled. Čvorovi stabla su objekti preko kojih program interaguje sa stranicom. Slika 4.3 ilustruje DOM reprezentaciju Veb stranice pojednostavljene strukture.

```

<html>
  <head>
    <title></title>
  </head>
  <body>
    <h1></h1>
    <div>

      <p></p>
    </div>
  </body>
</html>

```



Slika 4.3. DOM reprezentacija Veb stranice. Izvor: [62].

DOM predstavlja dinamičko stablo HTML objekata, tj. čvorova u vreme izvršavanja programa. JavaScript program može pristupati čvorovima kako bi ustanovio stanje elementa GKI (na primer, da li je korisnik uneo sadržaj u tekstualno polje). Mogu su ažurirati atributi elemenata, poput stila ili sadržaja. Izgled stranice se može menjati dodavanjem novih ili uklanjanjem postojećih elemenata GKI kao čvorova stabla.

DOM stablo se sastoji od čvorova koji su instance tipa Čvor (eng. *Node*). Dva osnovna podtipa Čvora su Element (eng. *Element*) koji predstavlja HTML tag i Tekst (eng. *Text*) koji predstavlja sadržaj taga u tekstualnom formatu, uključujući i prazan prostor. Tip Čvor ima, između ostalih, i sledeće atribute:

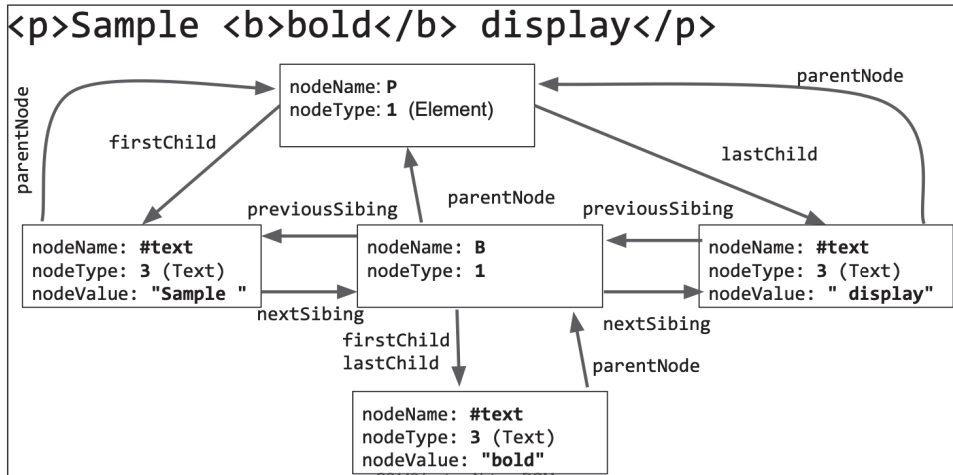
- *nodeType* – tip čvora (*Element*, *Text*, *Comment*),
- *nodeName* – ime čvora,
- *nodeValue* – vrednost čvora (*null* za *Element*, sadržaj za *Text* tip čvora), može se koristiti i za čitanje i za dodelu vrednosti,
- *textContent* – tekstualni sadržaj čvora i čvorova dece,
- *childNodes* – niz čvorova dece,
- *parentNode* – referenca na roditeljski čvor.

Pored navedenih atributa, tip sadrži i metode za kretanje kroz DOM hijerarhiju objekata stranice, kao što su:

- *parentNode()* – vraća roditeljski čvor,
- *nextSibling()* – vraća naredni čvor u istom nivou hijerarhije,
- *previousSibling()* – vraća prethodni čvor u istom nivou hijerarhije,
- *firstChild()* – vraća prvi element dete,

- *lastChild()* – vraća poslednji element dete.

Slika 4.4. ilustruje strukturu i sadržaj DOM stabla HTML paragrafa. Mogu se приметiti odvojeni čvorovi za HTML tagove (podtip Element) i sadržaje tagova (podtip Tekst).



Slika 4.4. DOM stablo HTML paragrafa. Izvor: [63].

Postoji više načina, odnosno JavaScript metoda za pristup elementima GKI kao čvorovima DOM stabla. Jedan je korišćenje *querySelector* metode koja kao argument prihvata CSS identifikator (ili selektor) odgovarajućeg HTML elementa. Metoda se poziva nad predefinisanim JavaScript objektom *document* koji ukazuje na DOM stablo tekuće Veb stranice u prozoru pretraživača. Primeri sintaksi i naredbi su dati u nastavku.

```

document.querySelector('css selector');
//vraća prvi element koji odgovara datom selektoru

document.querySelectorAll('css selector');
//vraća sve elemente koji odgovaraju datom selektoru

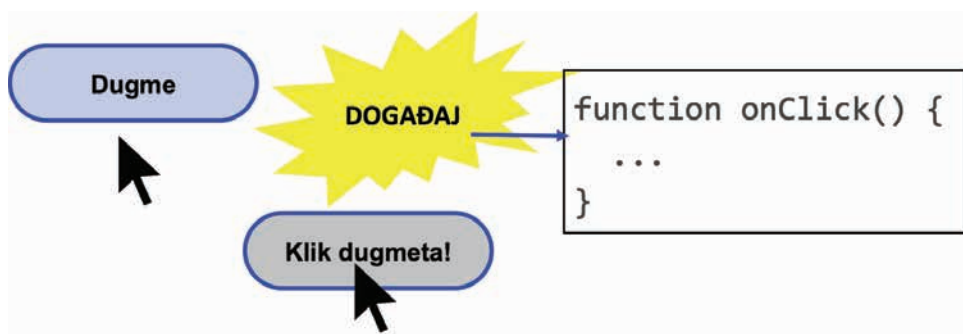
let element = document.querySelector('#dugme');
//vraća sve DOM čvorove koji predstavljaju HTML
//elemente sa id='dugme'

let listeElemenata = document.querySelectorAll('.klasa');
//vraća listu DOM čvorova koji predstavljaju HTML
//elemente sa class="klasa"
  
```

Skup metoda može pristupati čvorovima stabla korišćenjem različitih identifikator HTML elemenata. Identifikatori mogu biti vrednosti *id* atribut i *class* atributa HTML taga i ime taga, respektivno. Primeri u nastavku ilustruju korišćenje metoda.

```
document.getElementById('id elementa');  
//vraća element sa vrednošću id atributa  
  
document.getElementsByClassName('class vrednost');  
//vraća sve elemente sa vrednošću class atributa  
  
document.getElementsByTagName('ime taga');  
//vraća sve elemente sa imenom taga  
  
HTML: <div id="div1">...</div>  
let element = document.getElementById('div1');  
  
HTML: <div class="div2"> ... </div>  
let element = document.getElementsByClassName('div2');  
  
HTML: <div class="div3"> ... </div>  
let element = document.getElementsByTagName('div');
```

Prethodno opisane metode omogućavaju dinamički prikaz HTML dokumenta, odnosno promenu izgleda u vreme prikazivanja. Promena je inicirana nastankom programskih objekata događaja iz akcija korisnika nad GKI, kao što je prikazano na Slici 4.5. Kada korisnik klikne na dugme generiše se događaj i poziva funkcija koja osluškuje nastanak tog tipa događaja. Funkcija implementira promenu, tj. izvršava JavaScript kod GKI kao odgovor na nastanak događaja.



Slika 4.5. Interakcija korisnika i interfejsa odvija se putem objekata događaja.

Funkcije osluškivači događaja se mogu dodavati dinamički, u vreme izvršavanja aplikacije. Svaki DOM čvor ima sledeću metodu koja povezuje događaj i osluškivač.

```
addEventListener(ime događaja, ime funkcije)

//ime događaja čiji nastanak se registruje ili osluškuje,
//predefinisani tipovi: click, focus, blur, ...
//ime funkcije koja će biti pozvana kada događaj nastane
```

Postojeći osluškivač se može ukloniti programski korišćenjem metode u nastavku koja razdvaja događaj i osluškivač.

```
removeEventListener(ime događaja, ime funkcije)
//ime događaja koji se dalje ne osluškuje
//ime funkcije koja se poziva kada događaj nastane
```

Primeri u nastavku ilustruju primenu opisanih metoda u promeni sadržaja HTML stranice u toku njenog prikazivanja. Svaki primer navodi sadržaj HTML dokumenta i JavaScript kod koji menja njegov sadržaj.

Funkcija *otvoriPoklon()* pronalazi čvor *img* taga DOM stabla i dodeljuje kao vrednost varijabli *slika*. Varijabla ima vrednost tipa Čvor i sadrži atribut *src* kojem se dodeljuje vrednost putanje ka novoj slici. Rezultat je promena vrednosti *src* atributa *img* taga. Na kraju se uklanja prethodno dodeljeni osluškivač *img* tagu za događaj klik (eng. *click*). Može se primetiti da je funkcija osluškivač razdvojena od događaja (*removeEventListener()*) prilikom definisanja, tj. u telu te funkcije. Nakon što je funkcija definisana, ona se proglašava osluškivačem za događaj klika na sadržaj *img* taga HTML dokumenta (*addEventListener()*). Metode dodavanja i uklanjanja osluškivača događaja se kombinuju tako da će se na događaj reagovati samo kod prvog klika na *img* tag. Tada će se slika koja se prikazuje promeniti kao u rezultatu ispod.

```
<html>
<head>
<title>DOM primer 1</title>
</head>
<body>
<h1>Izaberi poklon:</h1>
```

```

</body>
</html>
```

```
function otvoriPoklon() {
const slika = document.querySelector('img');
slika.src =
'https://upload.wikimedia.org/wikipedia/commons/1/15/Gift
_Cake_%2812661117903%29.jpg';
slika.removeEventListener('click', otvoriPoklon);
}
const slika = document.querySelector('img');
slika.addEventListener('click', otvoriPoklon);
```

Izaberi poklon:



Izaberi poklon:



Sledeći primer menja sadržaj dva HTML taga, slike (*img*) i zaglavlja prvog nivoa (*h1*). Sadržaj taga slike se menja na isti način kao u prethodnom primeru. Funkcija *otvoriPoklon()* je proširena tako što se varijabli *naslov* dodeljuje vrednost čvora taga *h1* i dodaje novi tekst zaglavlja preko atributa *textContent*.

```
<html>
<head>
<title>DOM primer 2</title>
</head>
<body>
<h1>Izaberi poklon:</h1>

</body>
</html>
```

```
function otvoriPoklon() {
const slika = document.querySelector('img');
slika.src =
'https://upload.wikimedia.org/wikipedia/commons/1/15/Gift
_Cake_%2812661117903%29.jpg';
const naslov = document.querySelector('h1');
naslov.textContent = 'Poklon izabran!';
slika.removeEventListener('click', otvoriPoklon);
}
const slika = document.querySelector('img');
slika.addEventListener('click', otvoriPoklon);
```

Izaberi poklon:



Poklon izabran!



U prethodnom primeru se funkcija *querySelector()* poziva na identičan način (pristup *img* čvoru) na različitim mestima u kodu.

Element izvor događaja se može identifikovati na drugi način. U tom smislu JavaScript *event* objekat ima dva atributa:

- *event.target* – predstavlja element izvor događaja, tj. nad kojim je događaj generisan. U slučaju da elementu nije pridružen osluškivač, događaj se prenosi kroz DOM stablno dok se ne nađe na element koji ima osluškivač registrovan za taj tip događaja. Drugim rečima, izvor ne mora imati pridružen osluškivač. Mehanizam je poznat pod nazivom delegiranje događaja (eng. *event delegation*).
- *event.currentTarget* – označava element kojem je pridružen osluškivač za obradu nastalog događaja.

U slučaju da je konkretnom HTML elementu pridružen osluškivač događaja, dva atributa ukazuju na isti element. U suprotnom mogu vratiti dva različita elementa. Na ovaj način se od korisnika ne mora zahtevati visoka preciznost prilikom selekcije elemenata GKI kako bi odgovarajuće funkcije bile pozvane. Stilovi se mogu dodeljivati ili uklanjati elementima pomoću *classList* atributa čvora DOM stabla. Atribut sadrži nazive klasa stilova koji se primenjuju na element i njegov sadržaj se može menjati.

```
classList.add('klasa1', 'klasa2')
//dodavanje CSS klasa koje će se primenjivati na
//element, nazivi klasa se navode kao parametri

classList.remove('klasa1', 'klasa2')
//uklanjanje CSS klasa koje se primenjuju na element

classList.toggle('klasa1', 'klasa2')
//ako se navedene klase primenjuju one se uklanjaju;
//ako se ne primenjuju //dodaju se elementu
```

Primer u nastavku ilustruje korišćenje prethodno opisanih atributa, *event.currentTarget*, *classList.add* i *classList.remove*. Tagovima *div* su dodeljeni identifikatori, *div1* i *div2*. Svaki *div* sadrži tekst zaglavlja prvog nivoa (*h1*) i sliku (*img*). Klasa stila *sakriven* definiše da element na koji se primenjuje ne bude prikazan. Osluškivač *otvoriPoklon* kao argument prihvata objekat događaj (*event*). Funkcija najpre identifikuje objekat izvor događaja i dodeljuje varijabli *slika*. Zatim uklanja osluškivač događaja za dati izvor i tip događaja kako bi se na događaj reagovalo samo na prvi klik slike, slično kao u prethodnom primeru. Zatim se varijablama *div1* i *div2* dodeljuje vrednosti *div* čvorova DOM stabla na osnovu njihovih identifikatora (*id*). Prvom *div* elementu se dodaje stil *sakriven*, dok se drugom uklanja dati stil koji se primenjuje na element. Nakon definicije osluškivača, slici u *div1* elementu se pridružuje kreirani osluškivač na događaj klika. Kao rezultat, klikom na sliku u *div1* elementu on će postati sakriven, dok će inicijalno sakriven *div2* element postati vidljiv.

```
<html>
<head>
<title>DOM primer 3 </title>
</head>
<body>
<div id="div1">
<h1>Izaberi poklon:</h1>

</div>
<div id="div2" class="sakriven">
<h1>Poklon izabran!</h1>
```



```


</div>
</body>
</html>

```

```

.sakriven {
display: none;
}

```

```

function otvoriPoklon(event) {
const slika = event.currentTarget;
slika.removeEventListener('click', otvoriPoklon);

```

```

const div1 = document.querySelector('#div1');
const div2 = document.querySelector('#div2');
div1.classList.add('sakriven');
div2.classList.remove('sakriven');
}

```

```

const slika = document.querySelector('#div1 img');
slika.addEventListener('click', otvoriPoklon);

```

Izaberi poklon:



Poklon izabran!



Prethodno primeri ilustruju načine promene izgleda HTML stranice iz JavaScript programa. Može se menjati sadržaj *pojedinačnih elemenata* GKI poput primera u kojima su menjani sadržaji *img* i *h1* taga, ili *grupe elemenata* GKI kao u primeru koji menja delove stranice u okviru elemenata grupisanja (*div*, *section*).

Više informacija u vezi sa DOM stablom se može naći u [62].

4.4.2. Događaji

Interakcija GKI počinje tako što korisnik inicira akciju nad interfejsom. Fizička akcija korisnika se transformiše u programski objekat događaja i aplikacija zna koji element je odgovoran za obradu događaja na osnovu funkcije osluškivača. Događaji mogu biti klik dugmetom miša nad elementom GKI, klik na taster tastature, dodir i pokreti dodira, i pomeranje, promena veličine ili zatvaranje prozora aplikacije.

Opšte gledano, događaja opisuje dešavanje značajnu za funkcionisanje aplikacije kao što je pristizanje zahteva od druge aplikacije, pristizanje traženih podataka u GKI iz pozadinske logike softverskog sistema, ili završetak učitavanja Veb stranice u prozor pretraživača. U daljem razmatranju događaji se razmatraju u kontekstu interakcije korisnika sa GKI.

JavaScript događaji nisu deo jezgra jezika, njihova realizacija je prepuštena rutinama koje implementiraju Veb pretraživači. Tipovi događaja u vezi sa GKI obuhvataju:

- Događaji miša (klik, pomeraj, ulazak ili izlazak pokazivača iz regiona stranice),
- Događaji tastature (pritisak tastera, otpuštanje tastera, produžen pritisak tastera),
- Događaji fokusa (element u fokusu, ulazak i izlazak elementa iz fokusa),
- Događaji forme (slanje podataka forme, reset forme),
- Događaji promene prikaza (promena veličine prozora, promena vidljivog dela prozora), i
- Događaji učitavanja resursa ili pristizanja zahteva.

Korisnik često ne pokrene akciju nad GKI dovoljno precizno, tj. direktno nad željenim elementom. Pored toga, različiti elementi mogu reagovati na isti tip događaja. Svrsishodno upravljanje događajima podrazumeva sledeće korake:

1. Generisanje i opisivanje događaja od interesa;
2. Otkrivanje mesta nastanka, tj. izvora događaja; i
3. Definisanje odgovora aplikacije na nastanak događaja kao prosleđivanje i dodelu na obradu odgovarajućim elementima.

Za opisivanje događaja JavaScript je uveo tip *Event* iz kog su izvedeni tipovi za specifične tipove poput događaja miša (*MouseEvent*) i događaja tastature (*KeyboardEvent*). U attribute ovog tipa, između ostalih spadaju:

- *type* – tip događaja kao string simbolička konstanta (*click*, *mousedown*, *mouseup* i drugi),
- *timeStamp* – vreme nastanka događaja,
- *currentTarget* – element kojem je pridružen osluškivač, tj. koji je registrovan za prijem i obradu događaja, i

- *target* – element nad kojim je generisan događaj (ne mora imati osluškivač).

Tip događaj miša je jedan od najzastupljenijih u korišćenju GKI. Događaji ovog tipa se generišu kako korišćenjem miša kao pokazivačkog uređaja, tako i površine osetljive na dodir (eng. *trackpad*). Radnje i stanja ovih ulaznih uređaja opisane su skupom atributa kao što su:

- *button* – identifikator dugmeta miša (na primer, 0 – levo, 1 – srednje, 2 – desno),
- *click* – klik dugmeta miša,
- *mouseDown* – pritisak dugmeta miša,
- *mouseUp* – otpuštanje dugmeta miša,
- *mouseMove* – pomeranje miša,
- *mouseenter*, *mouseleave* – pokazivač miša ulazi ili napušta površinu elementa kojem je pridružen osluškivač, respektivno,
- *pageX*, *pageY* – pozicija miša relativna u odnosu na gornji, levi ugao tekuće stranice kao referentni početak (x-horizontalno, y-vertikalno),
- *screenX*, *screenY* – pozicija miša relativna u odnosu na gornji, levi ugao ekrana (x-horizontalno, y-vertikalno).

Naredni primer ilustruje korišćenje događaja miša za prikaz pozicije miša na HTML stranici. Pozicija se ispisuje kao sadržaj *span* tagova za grupisanje teksta. Svakom tagu je dodeljen identifikator (*x-osa*, *y-osa*) kako bi mu se pristupilo kao čvoru DOM stabla. Varijabli *region* se dodeljuje vrednost DOM čvora *div* elementa koji će osluškivati događaje miša, dok se varijablama *xOsa* i *yOsa* dodeljuju vrednosti čvorova *span* tagova u kojima će se ispisivati koordinate pozicije miša. Funkcija osluškivača *azurirajKoordinate()* iz argumenta tipa događaja miša čita vrednosti atributa *pageX* i *pageY* i dodeljuje kao vrednosti sadržajima čvorova DOM stabla (*innerText*). Konkretan tip događaja će biti prepoznat i prosleđen kao parametar *event* osluškivača. Na kraju se varijabli *region* (odnosno njenom DOM čvoru) pridružuje osluškivač za tri vrste događaja miša - *mousemove*, *mouseenter* i *mouseleave*. Prva vrsta će omogućiti registrovanje pozicije miša unutar *div* elementa, dok će preostale dve obezbediti čitanje pozicije kada miš ulazi ili napušta površinu koju zauzima *div* element.

```
<div class="region">
<p> Pozicija miša u okviru regiona. </p>
<p> <code>X-osa</code>: <span id="x-osa">van
regiona</span> </p>
<p> <code>Y-osa</code>: <span id="y-osa">van
regiona</span> </p>
</div>
```

```

.region {
width: 400px;
height: 200px;
background-color: green;
color: white;
}

var region = document.querySelector(".region");
var xOsa = document.getElementById("x-osa");
var yOsa = document.getElementById("y-osa");

function azurirajKoordinate(event) {
    xOsa.innerText = event.pageX;
    yOsa.innerText = event.pageY;
}

region.addEventListener("mousemove", azurirajKoordinate);
region.addEventListener("mouseenter",
azurirajKoordinate);
region.addEventListener("mouseleave",
azurirajKoordinate);

```

Pozicija miša u okviru regiona.

x-osa: 313

y-osa: 111

Objekat događaja tastature, između ostalih, sadrži sledeće atribute:

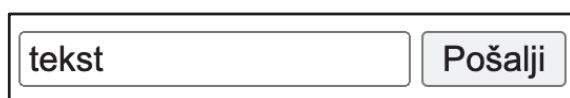
- *key* – string identifikator tastera tastature (alfanumerički i funkcionalni tasteri),
- *keyDown* – pritisak na taster tastature,
- *keyUp* – otpuštanje tastera tastature,
- *keyPress* – produžen pritisak tastera tastature.

Primer u nastavku pokazuje korišćenje događaja forme. Forma sadrži tekstualno polje i dugme. Vrednosti elemenata forme kao DOM čvorova se dodeljuju varijablama *forma* i *tekstPolje*, respektivno. Osluškivač se definiše kao anonimna funkcija i pridružuje formi za događaja slanja podataka forme (*submit*). Funkcija u predefinisanim JavaScript *alert* prozoru za obaveštenje korisniku prikazuje sadržaj tekstualnog polja forme.

```
<form id="forma">
<input type="text" id="tekst" name="ime" value="tekst"/>
<input type="submit" name="submit" value="Pošalji" />
</form>
```

```
var forma = document.getElementById("forma");
var tekstPolje = document.getElementById("tekst");

forma.addEventListener("submit", function(event) {
    alert("text: " + tekstPolje.value);
});
```

A screenshot of a web form. It consists of a single-line text input field with the text 'tekst' inside. To the right of the input field is a button labeled 'Pošalji'.

An embedded page at cdpn.io says

text: tekst

OK

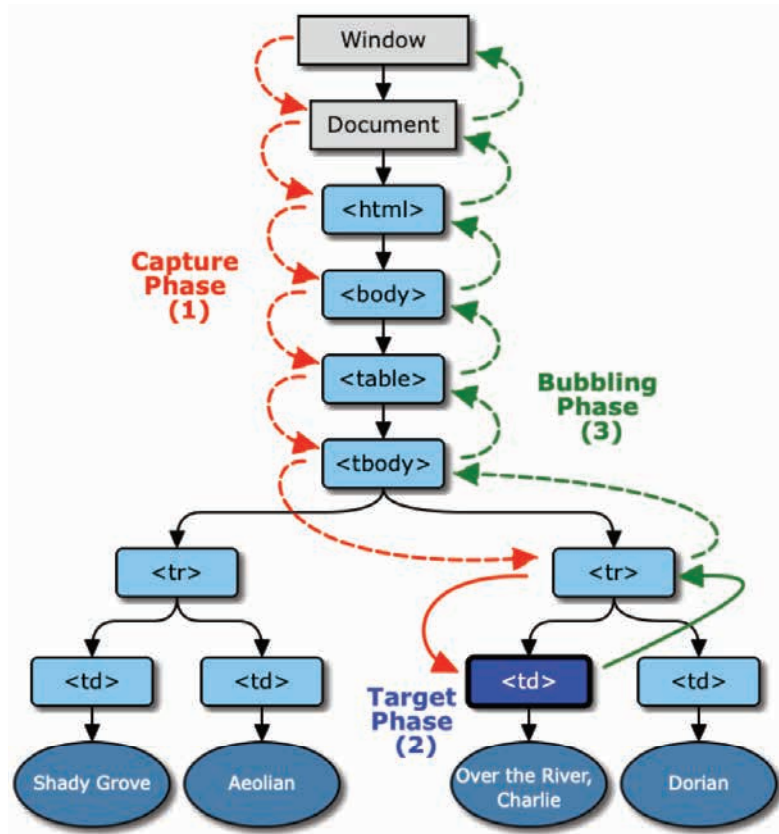
Veći broj HTML elemenata može imati registrovane osluškivače za konkretan događaj. Pored toga, ti elementi mogu sadržavati jedni druge. Tada je potrebno definisati koji elementi obrađuju događaj kada on nastane. To mogu biti roditeljski elementi ili elementi deca u hijerarhiji DOM stabla.

DOM standard [65] predviđa različite načine na koji se događaj kreće kroz stablo objekata GKI kao faze:

- *Capture* faza – objekat događaj putuje u dubinu elementa, ka elementina deci.
- *Target* faza – objekat događaj stiže na cilj, tj. u element koji ima registrovan osluškivač.

- *Bubble* faza – objekat događaj putuje od elementa izvora ka višim nivoima hijerarhije. Predstavlja podrazumevan smer putovanja događaja.

Slika 4.6. ilustruje faze kao načine na koje događaj putuje kroz DOM stablo. *De facto* način koji realizuje većina Veb pretraživača je *bubble*.



Slika 4.6. Faze kao načini prosleđivanja događaja kroz DOM stablo. Izvor: [65].

Bubble označava kretanje događaja od elemenata dece ka vrhu hijerarhije, dok *capture* znači kretanje od roditeljskog elementa ili vrha hijerarhije u dubinu ka elementima deci. *Bubble* je podrazumevano kretanje koje se može kontrolisati pomoću naredbi u nastavku.

```
event.stopPropagation();
//zaustavlja dalje prosleđivanje događaja event nezavisno
//od faze
```

```
element.addEventListener(tipDogađaja, imeOsluškivača, {
capture: true });
//definiše capture prosleđivanje događaja koje počinje od
//najvišeg elementa u hijerarhiji ka elementima deci, do
//elementa u kojem je svojstvo capture true
```

Primeri u nastavku ilustruju *bubble* i *capture* prosleđivanje događaja u DOM stablu, respektivno.

HTML stranica sadrži dva *div* elementa, roditeljski element (*roditelj*) i element dete (*dete*). Elementi menjaju boju pozadine radi ilustracije kretanja događaja. Dugme služi za vraćanje početne boje elemenata.

Skup pravila stilova u nastavku definišu margine i boje pozadine *div* elemenata. Za svaki *div* element se definiše početna boja pozadine (*roditelj*, *dete*) i boja pozadine selektovanog elementa (*roditelj.selected*, *dete.selected*).

Funkcije osluškivači menjaju boju pozadine elemenata i prikazuju odgovorajuću poruku kada korisnik klinke na svaki od elemenata. Funkcija *početnoStanje()* vraća početnu boju pozadine *div* elemenata tako što uklanja prethodno dodeljene boje selektovanih elemenata.

Na kraju se elementima HTML stranice kao čvorovima DOM stabla dodaju kreirani osluškivači događaja klika dugmeta.

Slika sa leve strane pokazuje početni izgled HTML stranice.

Klikom na element dete menja se boja deteta i roditelja jer oba elementa osluškuju taj događaj i događaj se prosleđuje od deteta ka roditelju (*bubble*). Redosled poziva je ilustrovan porukama u funkcijama osluškivača na slici u sredini.

Slika na desnoj strani prikazuje slučaj klika na roditeljski element. Sa obzirom da se može prosleđivati samo elementima na višim nivoima hijrerhije, događaj se izvršava u tekućem elementu, ne i u elementu detetu. Zbog toga je promenjena samo boja pozadine roditeljskog elementa i ispisana poruka iz njemu pridruženog osluškivača.

```
<html>
<head>
<title>JS Događaji: Dva osluškivača</title>
</head>
<body>
<div id="roditelj"> Klik na roditelja !
<div id="dete"> Klik na dete !
</div>
</div>
<button>Početno stanje</button>
```

```

</body>
</html>
div {
  padding: 20px;
  margin: 20px;
}
#roditelj {
  background-color: lightgrey;
}
#roditelj.selected {
  background-color: pink;
}
#dete {
  background-color: lavender;
}
#dete.selected {
  background-color: orange;
}
function klikRoditelj() {
  roditelj.classList.add('selected');
  console.log('Izabran roditelj!');
}
function klikDete() {
  dete.classList.add('selected');
  console.log('Izabrano dete!');
}
function početnoStanje() {
  dete.classList.remove('selected');
  roditelj.classList.remove('selected');
}
const roditelj = document.querySelector('#roditelj');
const dete = document.querySelector('#dete');
roditelj.addEventListener('click', klikRoditelj);
dete.addEventListener('click', klikDete);
const početno = document.querySelector('button');
početno.addEventListener('click', početnoStanje);

```

Console

```
"Izabrano dete!"
```

```
"Izabran roditelj!"
```

Console

```
"Izabran roditelj!"
```




Naredni primer modifikuje prethodni tako da prikaže *capture* prosleđivanje događaja. Prilikom dodele osluškivača roditeljskom *div* elementu svojstvu *capture* se dodeljuje vrednost *true*.

Srednja slika ilustruje *capture* princip - klikom na element dete, događaj će najpre biti obrađen u elementu roditelju i zatim prosleđen detetu.

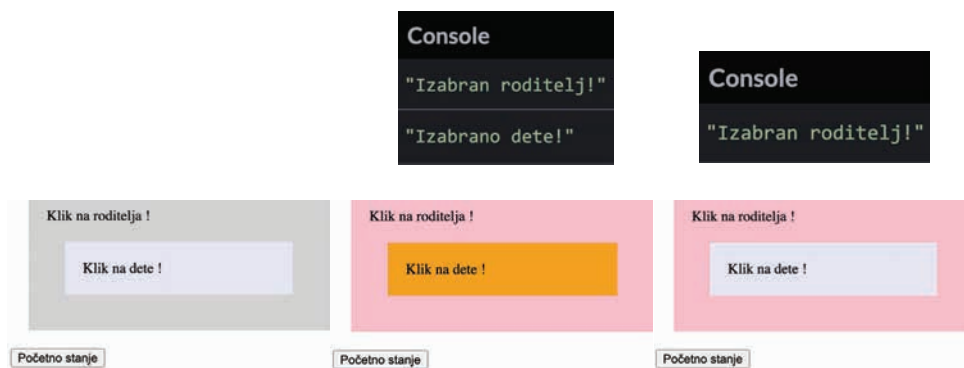
Kada se klikne na roditeljski element, događaj se pokušava proslediti ka višim nivoima hijerarhija. Ukoliko oni ne postoje, događaj se izvršava u elementu izvoru događaja i ne prosleđuje dalje, kao na slici desno.

```
<html>
<head>
<title>JS Događaji: Dva osluškivača</title>
</head>
<body>
<div id="roditelj"> Klik na roditelja !
<div id="dete"> Klik na dete !
</div>
</div>
<button>Početno stanje</button>
</body>
</html>
div {
  padding: 20px;
  margin: 20px;
}
#roditelj {
  background-color: lightgrey;
}
#roditelj.selected {
  background-color: pink;
}
#dete {
  background-color: lavender;
}
#dete.selected {
  background-color: orange;
}
```

```

function klikRoditelj() {
  roditelj.classList.add('selected');
  console.log('Izabran roditelj!');
}
function klikDete() {
  dete.classList.add('selected');
  console.log('Izabrano dete!');
}
function početnoStanje() {
  dete.classList.remove('selected');
  roditelj.classList.remove('selected');
}
const roditelj = document.querySelector('#roditelj');
const dete = document.querySelector('#dete');
roditelj.addEventListener('click', klikRoditelj, {
  capture: true } );
dete.addEventListener('click', klikDete);
const početno = document.querySelector('button');
početno.addEventListener('click', početnoStanje);

```



JavaScript jezik ne implementira odvojene niti (eng. *thread*) kao veći broj paralelnih tokova izvršavanja programa. U skladu sa tim, događaji se obrađuju sekvencijalno i procesiraju po jedan u datom trenutku. Obrada svakog događaja se odvija kao nezavisan tok kontrole. Zbog ovoga funkcije obrađivača događaja zahtevaju pažljiv dizajn. Pre svega u pogledu složenosti obrade kako bi odgovorile na zahtev korisnika u vremenu od nekoliko sekundi. Potrebno je distribuirati obradu interakcije sa korisnikom kroz neophodan broj obrađivača koji će za relativno kratko vreme izvršiti potrebne obrade.

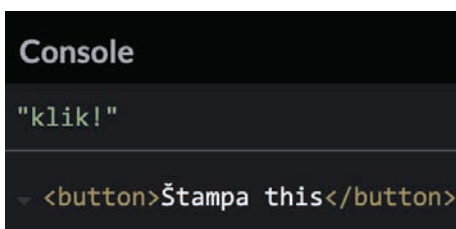
U Sekciji 4.2.1. je opisano korišćenjem predefinisaniog pokazivača *this*. U kontekstu funkcije, *this* može ukazivati na različite objekte u zavisnosti od

toga kako se funkcija poziva. Ako se funkcija koristi kao osluškivač događaja tada *this* pokazuje na element izvor događaja čiji nastanak osluškuje.

```
<html>
<head>
<title>this u osluškivaču događaja</title>
</head>
<body>
<button>Štampa this</button>
</body>
</html>

function klikDugmeta() {
  console.log('klik!');
  console.log(this);
}

const dugme = document.querySelector('button');
dugme.addEventListener('click', klikDugmeta);
```



Kada se *this* koristi u funkciji koja se ne poziva kao obrađivač događaja, tada ukazuje na objekat iz kojeg se se funkcija poziva. Ovo je ilustrovano u primeru ispod.

```
function pozdrav() {
  console.log(this.name + ' kaže zdravo');
}

var osoba = {
  name: 'osoba',
  hobbies: ['hodanje', 'trčanje', 'plivanje'],
  poruka: pozdrav
};

osoba.poruka();
```

Console

"osoba kaže zdravo"

Više informacija o događajima u JavaScript jeziku se može naći u [64].

4.5. TypeScript jezik

TypeScript jezik implementira sve elemente JavaScript jezika i dodaje nove elemente uz postojeće. Ovde se prvenstveno misli na ograničenje sistem tipova. JavaScript sadrži tipove kao što su *number* i *string*, ali ne vrši proveru tipova prilikom dodele vrednosti varijabli. Tip varijable je uvek određen tipom poslednje dodeljene vrednosti. Ovakav pristup sa jedne strane obezbeđuje fleksibilnost u izradi programskog koda. Sa druge strane, u toku izvršavanja programa mogu nastupiti neočekivane situacije i proizvesti greške. Na primer, pokušaj aritmetičke operacije sa vrednošću koja nije broj.

TypeScript uvodi striktna pravila u vezi sa tipovima. Tipovi varijabli se moraju unapred definisati i moraju se podudarati tipovi varijabli i tipovi vrednosti koje im se dodeljuju. Jednom definisana varijabla ne može menjati tipove kao u JavaScript jeziku. Na ovaj način se može izbeći neočekivano izvršavanje programa i sprečiti nastanak grešaka.

Pored toga, TypeScript realizuje objektno-orijentisane elementa nalik na postojeće objektno-orijentisanje jezik kao što je C#.

Povezanost dva jezika ogleda se i u tome što se TypeScript kod prilikom izvršavanja najpre prevodi u JavaScript kod koji se dalje izvršava.

Podrška za TypeScript se može instalirati lokalno, podrazumevano u okviru razvojnog okruženja, ili globalno na nivou operativnog sistema računara.

4.5.1. Osnovni tipovi

TypeScript implementira osnovne JavaScript tipove predstavljene u Sekciji 4.1.4, uključujući i tip funkcije i objekat koji su opisani u Sekciji 4.2.1.

Definicije varijabli koriste identične modifikatore vidljivosti kao u JavaScript jeziku. Obavezno je navesti tip varijable. Sintaksa za definisanje varijable je kao u nastavku.

```
modifikator_vidljivosti imeVarijable: ime_tipa =  
vrednost;
```

Primeri ispod ilustruju korišćenje tipova.

```
//Boolean
let isDone: boolean = false;

//Number (floating point broj)
let decimal: number = 6;

//String
let fullName: string = 'Mladen Jovanović';
let sentence: string = 'Hello, my name is ${ fullName }';

//Nizovi
let list: number[] = [1, 2, 3];
let list: Array<number> = [1, 2, 3];
```

Any je tip za varijable koje mogu primiti vrednosti čiji tipovi nisu unapred poznati i čiji se tip može menjati nakon definisanja. Primer u nastavku ilustruje korišćenje varijable tipa *any*.

```
let anyVarijabla: any = 4;
anyVarijabla = "string vrednost";
anyVarijabla = false;
```

TypeScript omogućava definisanje varijable bez navođenja tipa. U tom slučaju će varijabli biti određen tip vrednosti koja joj se dodeljuje (eng. *type by inference*) i dalje joj se mogu dodeljivati samo vrednosti tog tipa.

```
let poruka = "Pozdrav!";
//identično sa let poruka: string = "Pozdrav!";

let broj = 10;
//identično sa let broj: number = 10;
```

Unija (eng. *union*) je korisnički definisan tip koji se definiše kao skup dozvoljenih vrednosti, najčešće tipa broj ili string. Iza ključne reči *type* navodi se ime tipa i skup vrednosti koje može imati odvojene simbolom uspravne crte (eng. *vertical bar*). Primeri ispod ilustruju unije.

```
type stanjeProzora = "otvoreno" | "zatvoreno" |
"minimalno" | "maksimalno";
```

```
type pristupFajlu = "odobren" | "neodobren";  
type neparniBrojevi = 1 | 3 | 5 | 7 | 9;
```

Generički tipovi (eng. *generics*) omogućavaju varijacije unutar tipa, najčešće niza. Primeri ispod bliže opisuju tip elemenata za svaki niz.

```
type nizString = Array<string>;  
type nizBroj = Array<number>;  
type objekatNiz = Array< { ime: string } >;
```

4.5.2. Interfejsi i klase

Strukturni tipovi (eng. *structural type system*) definišu objekte složenijih struktura u odnosu na osnovne tipove. Interfejs tip (eng. *Interface*) služi za definisanje struktura programa koje se sastoje od skupa svojstava. Sintaksa korišćenja interfejsa je u nastavku.

```
interface ImeInterfejsa {  
    svojstvo1: Tip1;  
    svojstvo2: Tip2;  
    svojstvo3: Tip3;  
}
```

Opciona svojstva, tj. ona koja nije obavezno navesti pri kreiranju instanci tipa interfejsa se označavaju korišćenjem sufiksa ?. U primeru ispod, prilikom kreiranja objekata tipa Tačka boja se ne mora navoditi.

```
interface Tačka {  
    x: number;  
    y: number;  
    boja?: string;  
}
```

Ukoliko dva objekta ima identičnu strukturu ili oblik (eng. *shape*) smatraju se istog tipa. U sledećem primeru varijabla *tačka* nije definisana kao tip *Tačka*. Međutim struktura varijable je identična strukturi interfejsa te se smatraju istog tipa. Zbog toga se varijabla može proslediti funkciji *logTačka()* koja prihvata parametar tipa *Tačka*.

```
interface Tačka {  
    x: number;
```

```

    y: number;
}

function logTačka(t: Tačka) {
    console.log(`${t.x}, ${t.y}`);
}

const tačka = { x: 15, y: 29 };
logTačka(tačka);

```

Console

"15, 29"

Strukture interfejsa se mogu nasleđivati. Primer u nastavku kreira strukturu *Oblik* kao interfejs sa jednim svojstvom. Interfejs *Pravougaonik* nasleđuje *Oblik* i dodaje svojstvo. Objekat izvedenog interfejsa se dodeljuje varijabli *pravougaonik*. Vrednosti svojstava objekta se naknadno definišu. Na kraju se ispisuje sadržaj objekta kao vrednost varijable.

```

interface Oblik {
    boja: string;
}

interface Pravougaonik extends Oblik {
    duzinaStranice: number;
}

var pravougaonik: Pravougaonik = {};
pravougaonik.boja = "crvena";
pravougaonik.duzinaStranice = 20;
console.log(pravougaonik);

```

Console

```
// [object Object]
{
  "boja": "crvena",
  "duzinaStranice": 20
}
```

Koncept klase predstavlja apstraktni tip podataka i preuzet je iz objektno-orijentisanih jezika. Može se koristiti za definisanje strukturnih tipova kao alternativa tipa interfejsa. Primer u nastavku predstavlja strukturu tačke kao tip klase.

```
class Tačka {
  x: number;
  y: number;

  constructor(x: number, y: number) {
    this.x = x;
    this.y = y;
  }
}

function logTačka(t: Tačka) {
  console.log(`${t.x}, ${t.y}`);
}

const novaTačka = new Tačka(20, 30);
logTačka(novaTačka);
```

Console

```
"20, 30"
```

Elementi klasa su slični kao u objektno-orijentisanim jezicima.

Klase se mogu nasleđivati pomoću ključne reči *extends*. Izvedene klase moraju inicijalizovati instancu roditeljske klase korišćenjem funkcije *super()* pre korišćenja nasleđenih atributa, uključujući i *this*. U primeru u nastavku roditeljskoj varijabli *var1* se može pristupiti nakon što joj je dodeljena vrednost pozivom metoda *super()* u izvedenoj klasi. Metoda može imati parametre kao vrednosti koje se dodeljuju atributima roditelja.

```
class Osnovna {
    var1 = 10;
}

class Izvedena extends Osnovna {
    constructor() {
        super();
        console.log(this.var1);
    }
}
```

Klasa može implementirati interfejs pomoću ključne reči *implements*. Tada mora implementirati metode deklarisanе interfejsom kao u primeru ispod.

```
interface Poruka{
    poruka(): void;
}

class Greška implements Poruka {
    ping() {
        console.log("Nastupila je greška !");
    }
}

var greška = new Greška();
greška.ping();
```

Console

"Nastupila je greška !"

This ukazuje na objekat klase.

Ključna reč *readonly* označava atribut klase koji se može inicijalizovati samo u konstruktoru i čija se vrednost naknadno ne može menjati.

Modifikatori vidljivosti se odnose na atribut i metode i uključuju:

- *public* – može se pristupiti bilo gde iz instance klase (podrazumevana vidljivost ukoliko se ne navede);
- *private* – vidljivost unutar tela klase u kojoj se atribut ili svojstvo definiše, isključujući izvedene klase;
- *protected* – vidljivost unutar tela klase u kojoj se atribut ili svojstvo definiše, uključujući izvedene klase.

Pristupne (eng. *accessor*) metode obezbeđuju kontrolisan pristup atributima klase. Omogućavaju čitanje (*get*) i postavljanje (*set*) vrednosti atributa. Primenjuju se sledeća pravila:

- Ukoliko se definiše samo *get* metoda, atribut je podrazumevano *readonly*.
- Ako se ne navede tip parametra *set* metode, on je određen tipom povratne vrednosti *get* metode.
- *Get* i *set* metoda moraju imati isti modifikator vidljivosti.

Primeru ispod pokazuje da se nazivima atributa ne dodaje prefiks simbol `_` kao u jeziku JavaScript.

```
class Klasa {
  dužina = 0;
  get dužina() {
    return this.dužina;
  }
  set dužina(vrednost: number) {
    this.dužina = vrednost;
  }
}
```

Statički atributi i metode se označavaju korišćenjem ključne reči *static*. U tom slučaju postoji jedan atribut ili metoda za sve instance klase. Pristupa im se preko imena klase, kao u primeru u nastavku.

```
class Klasa {
  static var1 = 0;
  static printVar1() {
    console.log(Klasa.var1);
  }
}
console.log(Klasa.var1);
Klasa.printVar1();
```

Console

100

100

Klasa u nastavku opisuje tip profila korisnika. Sadrži *readonly* atribut *ime*, privatni atribut *fotoUrl* i javni atribut *statusi*. Par pristupnih metoda je definisan za atribut *fotoUrl*. Metoda *dodajStatus()* ažurira vrednost odgovarajućeg atributa klase. Na kraju se kreira i prikazuje instanca klase kao vrednost varijable *osoba*.

```
class Profil {
    readonly ime: string;
    private fotoUrl: string;
    statusi: string[];
    constructor(ime: string, foto: string) {
        this.ime = ime;
        this.fotoUrl = foto;
        this.statusi = [];
    }
    getFoto(): string {
        return this.fotoUrl;
    }
    setFoto(noviFotoUrl: string): void {
        this.fotoUrl = noviFotoUrl;
    }
    dodajStatus(noviStatus: string): void {
        this.statusi.push(noviStatus);
    }
}
let osoba = new Profil("Mladjan Jovanovic",
    "https://image.url/");
console.log(osoba);
```

Console

```
// [object Object]
{
  "ime": "Mladjan Jovanovic",
  "fotoUrl": "https://image.url/",
  "statusi": []
}
```

Više informacija o klasama u TypeScript jeziku se može naći u [66].

4.5.3. Funkcije

TypeScript funkcije navode tipove argumenata i tip povratne vrednosti. Posle imena argumenta i simbola dvotačke (:) navodi se ime tipa. Nakon deklaracije argumenata u običnim zagradama i simbola dvotačke a pre tela funkcije navodi se tip povratne vrednosti funkcije.

U nastavku su najpre dati primeri definicija imenovane funkcije, sa opcionim i podrazumevanim argumentima.

Opcioni argumenti se označavaju pomoću prefiksa ? i preporučljivo je proveriti da li je prosleđena vrednost argumenta kod poziva funkcije (uslovna naredba).

Podrazumevani argumenti se definišu kao dodela vrednosti u deklaraciji argumenata funkcije.

Anonimna funkcija se definiše na sličan način kao u jeziku JavaScript, uz navođenje tipova argumenata i povratne vrednosti.

Ukoliko funkcija ne vraća vrednost, tip povratne vrednosti se mora navesti kao ključna reč `void`.

```
//Imenovana funkcija
function sabiranje(a: number, b: number): number {
    return a + b;
}

//Imenovana funkcija sa opcionim argumentima
function imePrezime(ime: string, prezime?: string) :
string {
```

```

    if (prezime)
        return ime + " " + prezime;
    else
        return ime;
}

//Imenovana funkcija sa podrazumevanim argumentima
function imePrezime(ime: string, prezime = "Jovanovic") :
string {
    return ime + " " + prezime;
}

//Anonimna funkcija
let sabiranje = function(a: number, b: number): number {
return a + b; };

//Void funkcija - funkcija koja ne vraća vrednost
function upozorenje(): void {
    console.log("Ovo je upozorenje !");
}

```

Definicija *Arrow* funkcije se proširuje na sličan način kao kod prethodno opisanih funkcija. Sledeći primer ilustruje sintaksu sa više argumenata. Za svaki argument i povratnu vrednost mora se navesti tip.

```

var suma = (a: number, b: number): number => {
    return a + b;
}

console.log(suma(35, 45));

```

Console

80

Ukoliko telo funkcije sadrži jednu naredbu ne moraju se koristiti vitičaste zagrade.

```

var suma = (a: number, b: number): number => a + b;
console.log(suma(35, 45));

```

Console

80

Arrow funkcija bez parametara može izgledati kao u nastavku.

```
let Stampa = () => console.log("Pozdrav svima !");  
Stampa();
```

Console

"Pozdrav svima !"

Klase mogu imati metode kao *arrow* funkcije.

```
class Korisnik {  
  id: number;  
  ime: string;  
  
  constructor(id: number, ime: string) {  
    this.id = id;  
    this.ime = ime;  
  }  
  
  prikaz = () => console.log(this.id + ' ' + this.ime);  
}  
  
let korisnik = new Korisnik(1, 'Mladjan Jovanovic');  
korisnik.prikaz();
```

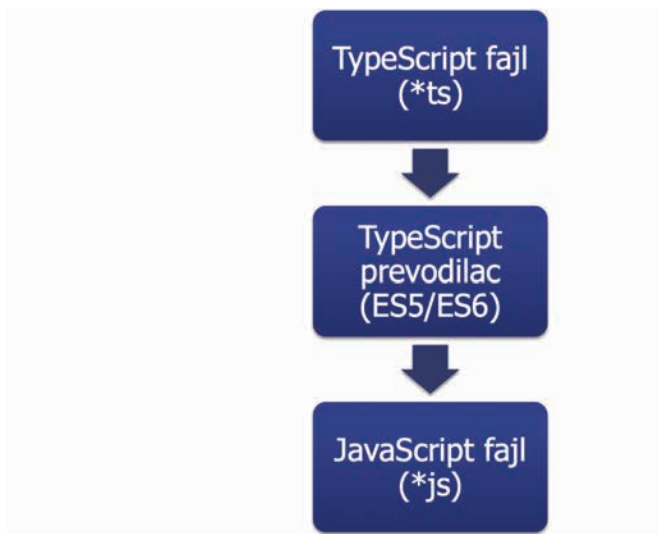
Console

"1 Mladjan Jovanovic"

4.5.4. Prevođenje

Prilikom izvršavanja aplikacija napisanih u TypeScript jeziku, datoteke sa ekstenzijom `.ts` se najpre prevode u JavaScript fajlove sa ekstenzijom `.js` koje se zatim izvršavaju. Prevodilac se pokreće `tsc` naredbom (eng. *TypeScript compiler*). Konfiguracioni fajl, `tsconfig.json`, je neophodan kako bi se kreirali validni izlazi fajlovi na odgovarajućoj lokaciji.

Više informacija o TypeScript programskom prevođenju se može naći u [67].



Slika 4.7. TypeScript prevodilac generiše JavaScript datoteke koje se dalje izvršavaju.

Prevođenje je ilustrovano sa nekoliko primera u nastavku. Na levoj strani je prikazan TypeScript kod, dok je na desnoj prikazan rezultat prevođenja kao JavaScript kod. Ispod izvornog i prevedenog koda je rezultat izvršavanja.

Klasa se transformiše u funkciju. Konstruktor je preslikan u identičan skup naredbi koje inicijalizuju istoimene atribute pomoću *this* pokazivača. Metode se prevode u anonimne funkcije kao vrednosti atributa čiji nazivi odgovaraju nazivima metoda klase. Atributi su definisani na nivou korenog objekta prototipa funkcije. Objekat funkcije se kreira korišćenjem ključne reči *new*, identično instanciranju klase.

<pre>//TypeScript class Profil { readonly ime: string; private fotoUrl: string; statusi: string[]; constructor(ime: string, foto: string) { this.ime = ime; this.fotoUrl = foto; this.statusi = []; } getFoto(): string { return this.fotoUrl;} setFoto(noviFotoUrl: string): void { this.fotoUrl = noviFotoUrl; } addStatus(noviStatus: string): void { this.statusi.push(noviStatus) ; }} var osoba = new Profil("Mladjan Jovanovic", "https://image.url/"); console.log(osoba);</pre>	<pre>//JavaScript var Profil = (function () { function Profil(ime, foto) { this.ime = ime; this.fotoUrl = foto; this.statusi = []; } Profil.prototype.getFoto = function () { return this.fotoUrl; }; Profil.prototype.setFoto = function (noviFotoUrl) { this.fotoUrl = noviFotoUrl; }; Profil.prototype.addStatus = function (noviStatus) { this.statusi.push(noviStatus); }; return Profil; }()); var osoba = new Profil("Mladjan Jovanovic", "https://image.url/"); console.log(osoba);</pre>
Console <pre>// [object Object] { "ime": "Mladjan Jovanovic", "fotoUrl": "https://image.url/", "statusi": [] }</pre>	

Funkcija se prevodi u funkciju. Prevedena funkcija ne navodi tipove argumenata i povratne vrednosti.

<pre>//TypeScript function getMesec(mesec: number): string { switch (mesec) { case 1: return "Januar"; case 2: return "Februar"; case 3: return "Mart"; case 4: return "April"; case 5: return "Maj"; case 6: return "Jun"; case 7: return "Jul"; case 8: return "Avgust"; case 9: return "Septembar"; case 10: return "Oktobar"; case 11: return "Novembar"; case 12: return "Decembar"; default: return "Ne postoji mesec !"; } } function logMesec(mesec: string): void { console.log(getMesec(parseInt(mesec))); } logMesec(12);</pre>	<pre>//JavaScript function getMesec(mesec) { switch (mesec) { case 1: return "Januar"; case 2: return "Februar"; case 3: return "Mart"; case 4: return "April"; case 5: return "Maj"; case 6: return "Jun"; case 7: return "Jul"; case 8: return "Avgust"; case 9: return "Septembar"; case 10: return "Oktobar"; case 11: return "Novembar"; case 12: return "Decembar"; default: return "Ne postoji mesec !"; } } function logMesec(mesec) { console.log(getMesec(parseInt(mesec))); } logMesec(12);</pre>

Kod prevođenja interfejsa JavaScript ne prepoznaje i ne preslikava tip kao takav. Struktura interfejsa se preslikava u strukturu tipa objekat, kao što se može videti na mestima korišćenja interfejsa u prevedenoj funkciji. Funkcija se preslikava u funkciju kao u prethodnom primeru.

//TypeScript

```
interface
KonfiguracijaObjekta {
širina: number;
visina: number;
boja: string;
}
interface Pravougaonik
extends KonfiguracijaObjekta
{
površina: number;
obim: number;
}
function
kreirajPravougaonik(konfig:
KonfiguracijaObjekta):
Pravougaonik {
return {
širina: konfig.širina,
visina: konfig.visina,
boja: konfig.boja,
površina: konfig.širina *
konfig.visina,
obim: konfig.širina*2 +
konfig.visina*2
}
}
var p =
kreirajPravougaonik({širina:
5, visina: 5, boja:
"Zelena"});
console.log(p);
```

//JavaScript

```
function
kreirajPravougaonik(konfig) {
return {
širina: konfig.širina,
visina: konfig.visina,
boja: konfig.boja,
površina: konfig.širina *
konfig.visina,
obim: konfig.širina*2 +
konfig.visina*2
};
}
var p = kreirajPravougaonik({
širina: 5, visina: 5, boja:
"Zelena" });
console.log(p);
```

Console

```
// [object Object]
{
  "širina": 5,
  "visina": 5,
  "boja": "Zelena",
  "površina": 25,
  "obim": 20
}
```

5. Angular okvir za razvoj korisničkog interfejsa

Početni Veb GKI su se sastojali od HTML stranica sa formama za unos podataka.

U cilju podrške složenoj interakciji sa korisnicima najpre se razvijaju biblioteke koje su realizovale specifične funkcionalnosti kao što su rukovanje URL adresiranjem, generisanje HTML sadržaja, upravljanje sesijama i komunikacija sa bazom podataka.

Povećanje kompleksnosti i broja korisnika interfejsa dovelo je do pojave okvira za razvoj GKI. Okviri diktiraju arhitekturu interfejsa kao skupa elemenata i relacija između njih. Ovo doprinosi performansama, unapređuje proces razvoja, i olakšava održavanje interfejsa. Prvi okviri su dodavali funkcionalnost vizuelnim elementima kao HTML tagovima. Takva HTML stranica se naziva templejt (eng. *template*) zbog toga što je njen sadržaj nije fiksne strukture već dinamički i promenljiv u vreme izvršavanja aplikacije, odnosno prikaza stranice u Veb pretraživaču. Tagovi imaju deklarativno značenje i govore šta i kako treba prikazati, dok naredbe tagova kreiraju sadržaj koji se prikazuje.

Ubrzani razvoj tehnologija zasnovanih na jeziku JavaScript dovodi do usložnjavanja arhitekture okvira tako da se deo serverske ili poslovne logike premešta u GKI. Jedan takav okvir jeste Angular [68] čija su struktura i funkcionalnosti opisani u ovom i narednim poglavljima. Serverska logika je implementirana u JavaScript jeziku i odvojena od logike interfejsa kao Node.js izvršno okruženje [69].

Ovde je važno razdvojiti primenu MVC obrasca na dva nivoa:

1. Vertikalno, na nivou čitave aplikacije kao sloj GKI (prikaz), sloj poslovne logike (kontroler) i sloj baze podataka (baza podataka). GKI može biti realizovan korišćenjem Angular okvira, poslovna logika kao Node.js i relaciona ili nerelaciona baza podataka.
2. Nivo korisničkog interfejsa, kao što je opisano u Poglavlju 2.

Arhitektura Angular okvira prati MVC obrazac. Pored toga, obrazac je implementiran na nivou strukture osnovnog gradivnog elementa Angular aplikacije ili komponente. Osnovni razlog jeste raspodela odgovornosti unutar korisničkog interfejsa radi bržeg i kvalitetnijeg razvoja i održavanja. Ovo postaje kritično sa povećanjem kompleksnosti GKI u pogledu podržanih funkcionalnosti i broja korisnika.

Model obuhvata podatke GKI, kao što su TypeScript objekti sa atributima. Prikaz čine vizuelni, tj. vidljivi elementi sadržani u HTML dokumentima i uređeni CSS pravilima. Kontroler upravlja ponašanjem GKI, odnosno načinom na koji prihvata zahteve korisnika i odgovara na iste. Ovo uglavnom podrazumeva upravljanje modelom, kako lokalnim podacima tako i komunikacija sa serverskom logikom (Node.js), i ažuriranje prikaza kao upravljanje interakcijom sa korisnikom preko događaja i osluškivača.

U vezi sa prikazom, Angular koristi mehanizam templejta. Templejt je HTML dokument čija se struktura i sadržaj definišu u vreme pisanja programa, ali se predviđaju delovi koji se mogu menjati u toku prikaza stranice, tj. izvršavanja programa. Ovi dinamički delovi su programske naredbe koje rade sa podacima modela. Njihovim izvršavanjem se menja izgled HTML stranice. Prednost ovakvog pristupa u odnosu na tehnike modifikacije DOM opisane u Sekciji 4.4.1 je što su promenjivi delovi predviđeni rano, u fazi dizajna, i vidljivi u templejtu. Kao takvi se prepoznaju u vreme prevođenja aplikacije i obezbeđuju efikasnije izvršavanje promena u templejtu.

Osnovna namena kontrolera je povezivanje prikaza i modela. Ovo podrazumeva komunikaciju sa serverskom stranom, obradu pristiglih podataka i ažuriranje prikaza na osnovu rezultata obrade. Na ovaj način upravlja dinamičkim delovima templejta. Pored toga, zadužen je za obradu interakcije sa korisnikom kroz registrovanje i obradu događaja nad elementima GKI.

Na nivou Angular okvira ili GKI, model obuhvata lokalne podatke potrebne za rad kontrolera i generisanje prikaza. Tu spadaju podaci koji postoje za vreme korišćenja aplikacije (ili korisničke sesije) kao što su događaji kao objekti, strukture podataka (objekti, nizovi, liste), instance klasa sa vrednostima atributa i podaci dobijeni od serverske logike (tekstualne ili binarne datoteke), Perzistentni podaci kao što je baza podataka ili datoteke se ne razmatraju kao deo modela GKI, već čitave aplikacije vertikalno po slojevima.

Osnovna ideja Angular okvira je da je GKI kolekcija *komponenti* kao osnovnih gradivnih elemenata [70]. Komponente mogu biti organizovane po *modulima*. Moduli predstavljaju mehanizam grupisanja komponenti koje pripadaju istom kontekstu ili rade sa istim entitetima aplikacije.

Akcenat je na unapređenju procesa razvoja i održavanja složenih Veb GKI kroz modularan dizajn koda i ponovnu upotrebljivost. Modularan dizajn podrazumeva upotrebu modula. Ponovna upotrebljivost podržana kreiranjem komponenti kao delova aplikacije koje imaju zaseban izgled i ponašanje i kao takvi se mogu koristiti na više mesta unutar jednog GKI ili u različitim aplikacijama.

Angular može koristiti JavaScript ili TypeScript za implementaciju programske logike GKI. TypeScript ima prioritet zbog implementacije objektno-orijentisanih principa (apstrakcija, enkapsulacija i ponovna upotrebljivosti) koji pogoduju prirodi okvira.

5.1. Gradivni elementi Angular okvira

Tabela 5.1. prikazuje osnovne elemente Angular okvira.

Modul predstavlja grupu komponenti slične namene. Pored komponenti, mogu grupisati servise i druge module. Analogija može biti paket (eng. *package*) u Java jeziku ili prostor imena (eng. *namespace*) u jeziku C#. Koncept modula potiče iz jezika TypeScript.

Komponenta je osnovna funkcionalna jedinica koja ima svoj prikaz tako da se svaki GKI može posmatrati kao skup komponenti.

Templejt je prikaz komponente u okviru kojeg se mogu navoditi direktive kao naredbe koje menjaju sadržaj i izgled.

Kontroler implementira interaktivnu logiku kao ponašanje komponente tokom interakcije sa korisnikom. *Model* je obično sastavni deo kontrolera kao skup podataka koje komponenta koristi i prikazuje.

Servisi realizuju delove GKI koji predstavljaju funkcionalnosti i nemaju izgled, tj. prikaz kao kod komponente. To mogu biti funkcije za komunikaciju sa pozadinskom logikom aplikacije, kao i funkcije koje koristi veći broj komponenti interfejsa.

Navigacija i orijentacija su od ključnog značaja za pravilno korišćenje Veb GKI [39]. Navigacija označava kretanje kroz GKI, dok orijentacija podrazumeva jasnu i preciznu informaciju za korisnika u vezi sa tim gde se trenutno nalazi u korišćenju GKI. *Rutiranje* omogućava definisanje navigacije i orijentacije u interfejsu.

Umetanje zavisnosti je uspostavljanje veze između servisa i delova aplikacije tako da se u vreme izvršavanja kreira samo neophodan broj instanci servisa. Samim tim omogućava optimalno zauzeće memorije što se može odraziti na performanse interfejsa.

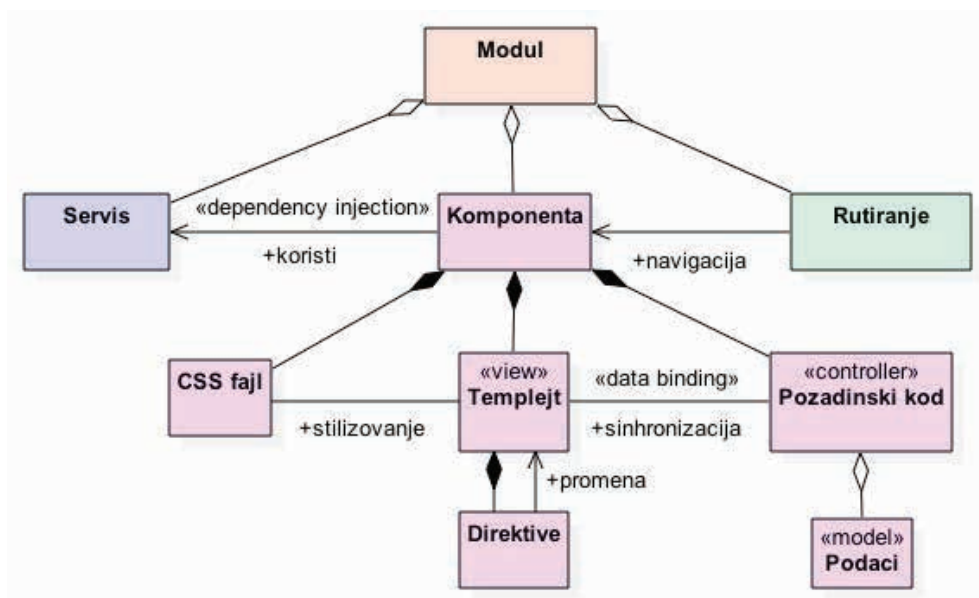
Komponenta se sastoji iz odvojenih datoteka koje je potrebno povezati kako bi predstavljali celinu. U tom pogledu povezivanje znači mogućnost deljenja podataka. Ovo je omogućeno različitim načinima *povezivanja podataka* datoteka. Na primer, kada korisnik inicira na templejtu komponente objekat događaja se mora proslediti kontroleru. Kao posledica izvršenja osluškivača kontroler podatke za prikaz čini dostupnim templejtu.

Tabela 5.1. Pregled i opis gradivnih elemenata Angular okvira.

Koncept	Opis (Realizacija)
Modul (eng. <i>Module</i>)	Kontejner, tj. mehanizam grupisanja komponenti, servisa i rutiranja (TypeScript datoteka)
Komponenta (eng. <i>Component</i>)	Osnovni gradivni element aplikacije (HTML dokument, CSS datoteka, TypeScript datoteka)
Templejt (eng. <i>Template</i>)	HTML dokument čiji se sadržaj može menjati dinamički i koji je deo komponente
Direktiva (eng. <i>Directive</i>)	Naredbe templejta koje omogućavaju dinamičku promenu sadržaja
Kontroler (eng. <i>Controller</i>)	Deo komponente koji čuva njene podatke i definiše ponašanje (TypeScript datoteka)
Servis (eng. <i>Service</i>)	Izdvojen skup funkcionalnosti bez prikaza (TypeScript datoteka)
Rutiranje (eng. <i>Routing</i>)	Mehanizam definisanja putanja do komponenti kako bi bile prikazane, tj. navigacije i orijentacije u GKI (TypeScript datoteka)
Umetanje zavisnosti (eng. <i>Dependency Injection</i>)	Mehanizam kojim funkcije servisa postaju dostupne (vidljive) komponenti, modulu ili interfejsu
Povezivanje podataka (eng. <i>Data binding</i>)	Automatizovani mehanizam sinhronizacije kao razmena podataka između templejta i kontrolera komponente

Dijagram ispod daje vizuelni prikaz strukture Angular GKI i veza između gradivnih elemenata. Aplikacija može imati veći broj modula. Jedan je obavezan, naziva se koreni modul (eng. *root module*) i služi za inicijalizaciju aplikacije. Ostali moduli se dodaju u zavisnosti od konkretne aplikacije.

Moduli grupišu komponente i servise po funkcionalnim celinama. Na taj način je olakšano dodavanje, uklanjanje i izmena komponenti tako da ne utiče na druge delove GKI. Rutiranje je implementirano kao zaseban i jedan modul za aplikaciju. Navigacija se definiše do postojećih komponenti. Na ovaj način se korisnici mogu kretati između različitih delova aplikacije. Servisi se mogu registrovati na nivou modula kako bi bili vidljivi svim komponentama. Komponente mogu koristiti metode servisa po potrebi. Aplikacija sadrži jednu korenu komponentu (eng. *root component*) preko koje se prikazuje sadržaji drugih komponenti i komponente koje se definišu u skladu sa zahtevima GKI. Svaka komponenta sadrži templejt kao prikaz sa pridruženim stilom, pozadinski kod kao klasu kontrolera i podatke modela. Delovi zajedno čine izgled i funkcionalnost komponente. Templejt integriše direktive kao naredbe koje mogu menjati strukturu i sadržaj HTML dokumenta.



Slika 5.1. Logička šema Angular GKI.

5.2. Modul

Modul je mehanizam grupisanja Angular okvira. Logički grupiše i definiše zajednički kontekst korišćenja povezanih komponenti ili servisa. U zavisnosti od složenosti, GKI se može posmatrati kao skup modula funkcionalnih blokova aplikacije. Definiše kao TypeScript datoteka i klasa u njoj označena ili anotirana specifičnim skupom metapodataka kao svojstvima za koja se navode odgovarajuće vrednosti. Na ovaj način se određuje kako će se klasa koristiti u vreme prevođenja i izvršavanja aplikacije. Skup metapodataka je poznat pod nazivom dekorator (eng. *decorator*). Naziv asocira na dekoraciju

klase metapodacima kao dodatnim informacijama. Dekorator ima sledeću strukturu.

```
@ImeDekoratora({
  svojstvo1: [vrednost11, vrednost12, ...],
  svojstvo2: [vrednost21, vrednost22, ...],
  ...
  svojstvoN: [vrednostN1, vrednostN2, ...]
})
```

Osnovni razlog uvođenja dekoratora jeste pravilna interpretacija i korišćenje TypeScript datoteke i klase u različitim kontekstima. Na primer, kao klase kontrolera komponente, klase modula ili klase servisa (videti Tabelu 5.1). Angular definiše specifične dekoratore sa svojstvima kako bi razlikovao navedene slučajeve. U ovoj sekciji će biti opisan dekorator modula, dok će u sekcijama u nastavku biti predstavljeni drugi dekoratori zajedno sa elementima na koje se primenjuju.

Svaka aplikacija sadrži:

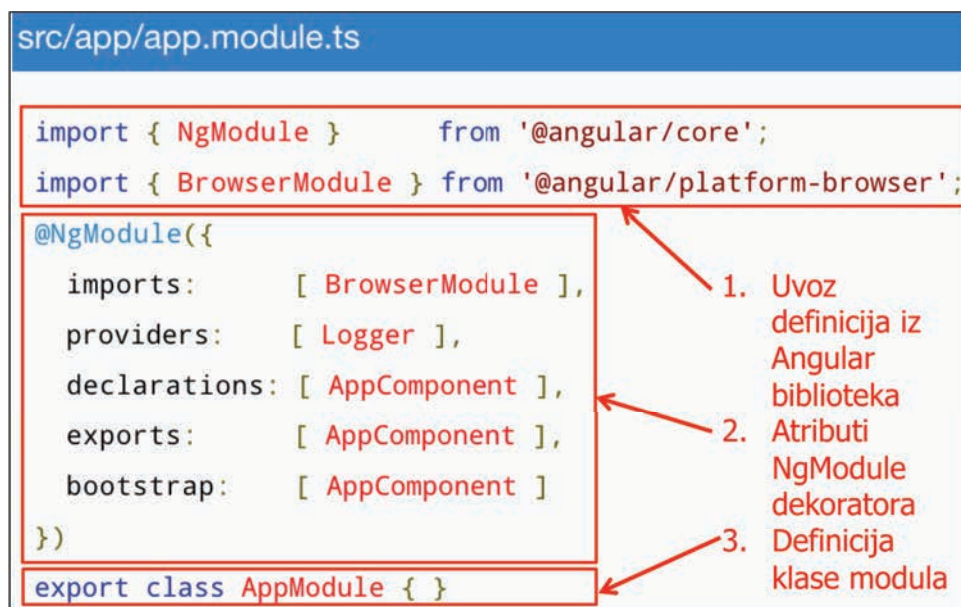
- Koreni moduli koji omogućava pokretanje i inicijalizaciju aplikacije. Ima predefinisani naziv *app.module.ts* i generiše se automatski pri kreiranju Angular projekta (videti Poglavlje 8).
- Module koje definišu programeri u zavisnosti od korisničkih zahteva GKI. Poznati su pod nazivom *feature modules* i služe organizaciji programskog koda.

Moduli koji grupišu odgovarajuće komponente ili servise će se učítavati samo onda kada se sadržani elementi koriste, tj. pozivaju u aplikaciji. Princip je poznat pod nazivom učítavanje na zahtev (eng. *lazy loading*). Na ovaj način se optimizuje memorijsko zauzeće što može doprineti boljim performansama aplikacije. Moduli mogu koristiti ili sadržati druge module.

Modul je predstavljen kao TypeScript klasa sa dekoratorom *@NgModule*. Dekorator sadrži sledeća svojstva gde svako svojstvo ima kao vrednost niz:

- *declarations* – navodi nazive komponenti koje pripadaju tekućem modulu;
- *exports* – nazivi komponenti modula koje su vidljive drugim modulima;
- *imports* – moduli čije će komponente biti korišćene u komponentama tekućeg modula;
- *providers* – imena servisa koje će tekući modul instancirati i mogu se koristiti u aplikaciji;
- *bootstrap* – navodi se samo za koreni modul i to ime korene komponente (*AppComponent*).

Slika u nastavku ilustruje koreni modul aplikacije. Dekorator sa svojstvima i njihovim vrednostima se navodi pre definicije klase modula. Ključna reč *export* u definicije klase znači da je ona vidljiva i van modula u kojem je definisana. Ovo se odnosi i na definicije klase u komponenti ili servisu.



Slika 5.2. Struktura korenog modula aplikacije (*app.module.ts*).

Angular definiše module i kao skupove funkcionalnosti tipične za Veb GKI. Na slici u nastavku su opisani neki od često korišćenih kao modul koji komunicira sa Veb pretraživačem u kojem se aplikacija izvršava (*BrowserModule*), modul sa funkcijama rutiranja (*RouterModule*) i modul za komunikaciju sa drugim aplikacijama ili Veb serverom putem HTTP (eng. *Hypertext Transfer Protocol*) (*HttpClientModule*).

NgModule	Import it from	Why you use it
BrowserModule	@angular/platform-browser	When you want to run your app in a browser
CommonModule	@angular/common	When you want to use NgIf, NgFor
FormsModule	@angular/forms	When you want to build template drive forms (includes NgModel)
ReactiveFormsModule	@angular/forms	When you want to build reactive forms
RouterModule	@angular/router	When you want to use RouterLink, .forRoot(), and .forChild()
HttpClientModule	@angular/common/http	When you want to talk to a server

Slika 5.3. Predefinisani Angular moduli sa opisom funkcionalnosti. Izvor: [70].

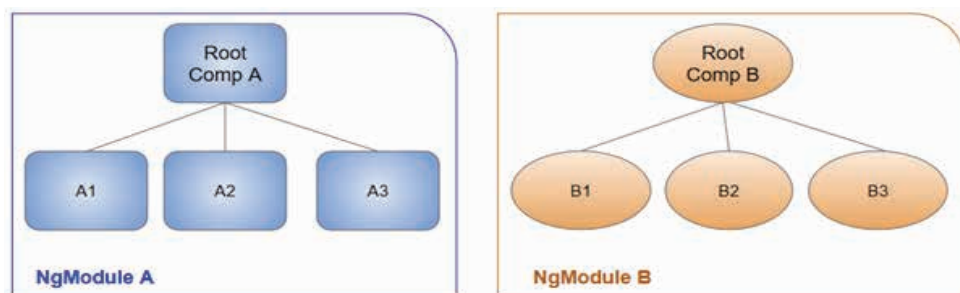
Moduli se mogu kreirati pomoću interfejsa komandne linije (eng. *CLI - Command Line Interface*) pomoću sledeće naredbe nastavku. Prva naredba koristi pune nazive komandi, dok druga koristi skraćeni oblik.

```
ng generate module ime_modula
ng g m ime_modula
```

Naredbe iznad kreiraju modul kao TypeScript datoteku sa dekoratorom `@NgModule` i nazivom koji prati konvenciju imenovanja kao ispod.

```
ime_modula.module.ts
```

Slika 5.4. daje pojednostavljen prikaz grupisanja komponenti aplikacije po modulima.



Slika 5.4. Organizacija komponenti korišćenjem modula. Izvor: [70].

Više informacija o modulima se može naći u [70].

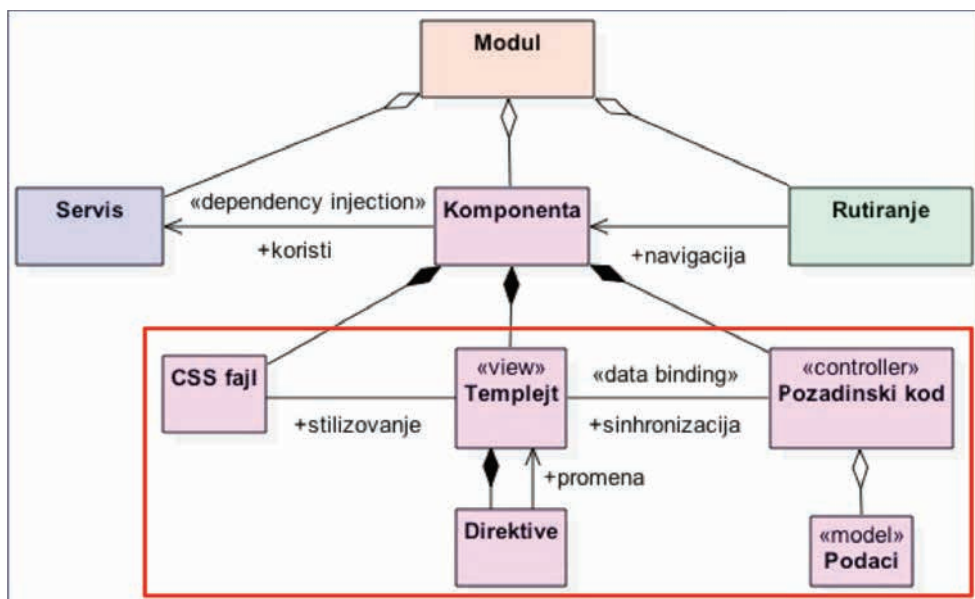
5.3. Komponenta

Komponente su osnovni delovi Angular aplikacije koji definišu izgled i ponašanje korisničkog interfejsa.

5.3.1. Struktura

Struktura komponente prati MVC obrazac i sastoji se iz sledećih delova:

1. Templejt (HTML dokument) kao *prikaz*,
2. CSS datoteka za stilizovanje templejta,
3. Pozadinski kod (TypeScript klasa) kao *kontroler* koji realizuje funkcije koje podržavaju prikaz, i upravlja i sadrži podatke komponente kao *model*.



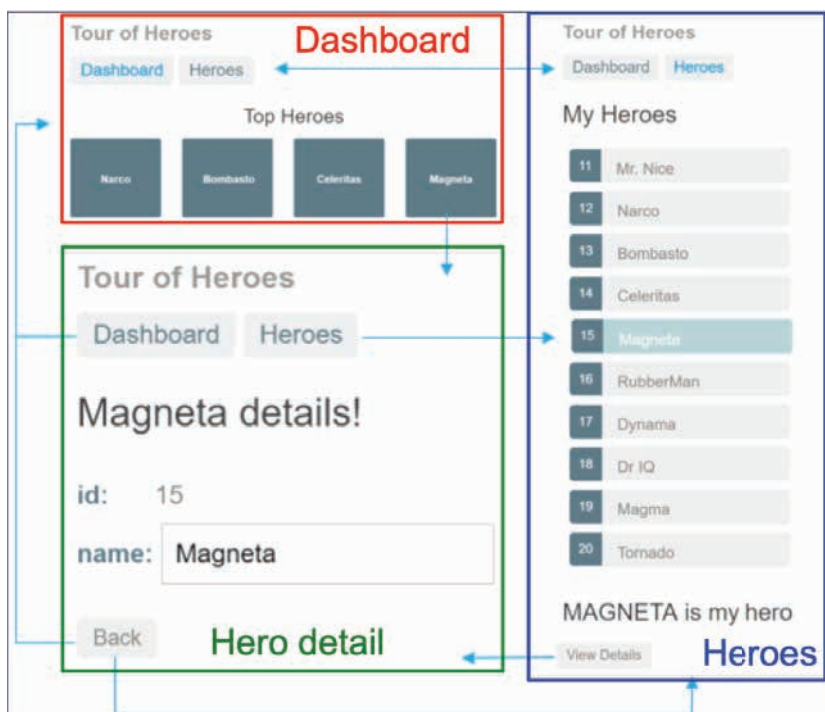
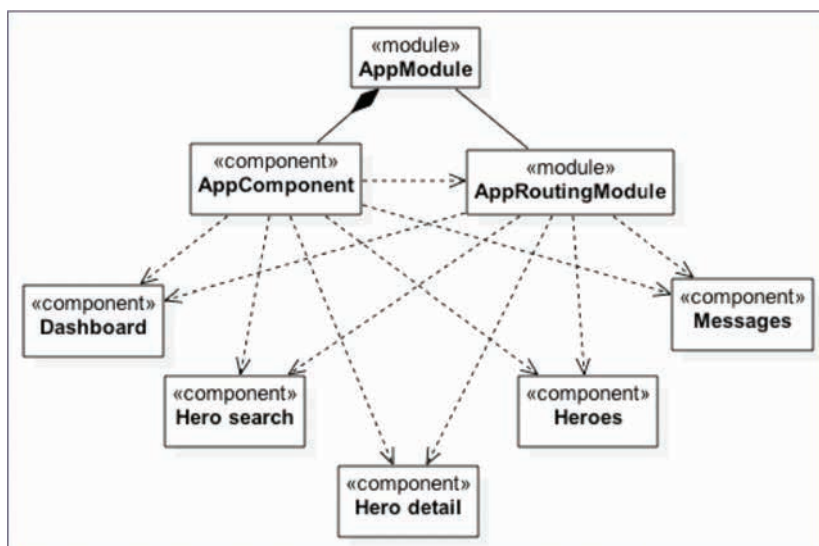
Slika 5.5. Struktura i uloga delova komponente.

Prilikom kreiranja komponente, generišu se delovi iznad kao zasebne datoteke koje su logički povezane.

Svaka aplikacija se sastoji od:

1. Korene komponente koja je vrh DOM hijerarhije komponenti aplikacije (*AppComponent*). Kreira se automatski po generisanju projekta. Njena namena je prikaz drugih komponenti.
2. Komponenti koje definiše programer i koje implementiraju specifične funkcije aplikacije.

Na ovaj način Angular aplikacija je DOM stablo prikaza komponenti u vreme izvršavanja. Slika 5.6. ilustruje strukturu aplikacije pomoću dijagrama klasa i izgled GKI sastavljenog od prikaza (templejta) komponenti.



Slika 5.6. Dijagram koji ilustruje strukturu i relacije između elemenata Angular aplikacije (gore), i prikaz odgovarajućih komponenti u GKI (dole).

Komponente objedinjuju templejt (prikaz) i pozadinski kod (kontroler) entiteta aplikacije ili korisničkog interfejsa. Mogu koristiti servise koji definišu specifične funkcionalnosti. Korišćenje servisa je opisano u Sekciji 5.4.

Templejt je HTML datoteka sa dinamičkim sadržajem. Kombinuje HTML tagove sa naredbama koje mogu modifikovati strukturu i sadržaj tagova. Naredbe se zovu *direktive* i opisane su u Sekciji 5.3.4. Direktive rade se podacima komponente, odnosno kontrolera. Kako bi razmenjivali podatke sa kontrolerom definisani su specifični načini komunikacije poznati pod nazivom povezivanje podataka (eng. *data binding*). Povezivanje podataka je objašnjeno u Sekciji 5.3.3. Prilikom prikazivanja templejta izvršavaju se sve navedene direktive i povezivanja podataka kako bi korisnik imao ažuran pregled komponente.

Kontroler komponente je TypeScript klasa sa dekoratorom `@Component` koja implementira programsku logiku i ima pridružen templejt kao prikaz.

```
ng generate component ime_komponente
ng g c ime_komponente
```

Naredbe iznad kreiraju delove komponente sa nazivima po šablonu u nastavku.

```
ime_komponente.component.ts
```

Naredbe iznad kreiraju komponentu u direktorijumu sa nazivom *ime_komponente* koji sadrži delove koji uključuju HTML templejt kao prikaza, CSS datoteku za stilizovanje templejta, i TypeScript datoteku sa dekoratorom `@Component` kao kontroler.

```
ime_komponente (direktorijum):
ime_komponente.component.html (templejt)
ime_komponente.component.css (stil)
ime_komponente.component.ts (kontroler)
```

Dekorator `@Component` ima dvostruku ulogu - označava TypeScript klasu kao pozadinski kod komponente i povezuje delove komponente. Sadrži sledeće svojstva:

- *selector* – CSS selektor taga koji govori Angular okruženju da kreira i insertuje instancu komponente u slučaju da pronade tag u HTML templejtu ili stranici u aplikaciji. Predstavlja tag identifikator komponente. Na primer, tag `<trening-lista></trening-lista>` znači da će templejt komponente biti prikazan kao sadržaj taga. Vrednost se navodi kao string.

- *templateUrl* – relativna putanja do templejta komponente. Definicija templejta se može navesti i kao ugrađena (eng. *inline*), u obliku string vrednost.
- *providers* – nazivi servisa koje koristi tekuća komponenta. Vrednost se navodi kao niz.
- *styleUrls* – nazivi CSS datoteka čiji stilovi se mogu koristiti za uređivanje izgleda templejta. Vrednost se navodi kao niz.

Slika 5.7. ilustruje strukturu *@Component* dekoratora. Ukoliko komponenta ne koristi servise, svojstvo *providers* se ne mora navoditi.

```
@Component ({
  selector: 'naziv-selektora',
  templateUrl: './putanja-do-templejt-datoteke',
  styleUrls: ['./putanja-do-stil-datoteke']
})
```

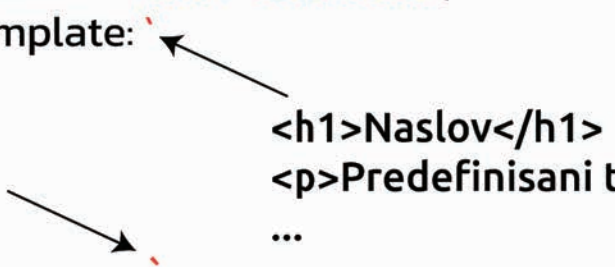
Slika 5.7. Struktura dekoratora *@Component* koji označava kontroler komponente i povezuje delove komponente

Slika 5.8. daje primer ugrađene definicije templejta komponente. Vrednost svojstva *template* se navodi u okviru *back-tick* simbola (```). Ovaj način definisanja se koristi za templejte jednostavnijeg sadržaja i strukture.

```
@Component ({
  selector: 'naziv-selektora',
  template: `

# Naslov</h1> <p>Predefinisani tekst</p> ...` styleUrls: ['./putanja-do-stil-datoteke'] })


```



Slika 5.8. Ugrađena definicija templejta komponente u *@Component* dekoratoru.

Svaka aplikacija ima jednu korenu komponentu čiji delovi imaju predefinisane nazive kao:

- *app.component.html* – templejt korene komponente.
- *app.component.css* – stil templejta korene komponente.
- *app.component.ts* – kontroler korene komponente.

U nastavku je dat izgled kontrolera korene komponente. Komponenta ima podrazumevanu vrednost *selector* svojstva (*app-root*) i prethodno navedena imena templejta i stila.

```
//Kontroler korene komponente
import { Component } from '@angular/core';

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent {
  title = 'component-overview';
}
```

Templejt korene komponente sadrži tag čije ime predstavlja vrednost *selector* atributa nove komponente koju kreira programer. U vreme izvršavanja aplikacije korisnik vidi templejt ove komponente kroz templejt korene komponente koja se podrazumevano prikazuje.

```
//Templejt korene komponente
<app-component-overview></app-component-overview>
```

U kodu ispod se može videti da vrednost *selector* atributa u kontroleru dodate komponente odgovara nazivu taga iznad.

```
//Kontroler komponente
import { Component } from '@angular/core';

@Component({
  selector: 'app-component-overview',
  templateUrl: './component-overview.component.html',
  styleUrls: ['./component-overview.component.css']
})
export class ComponentOverviewComponent {}
```


Templejt dodate komponente sadrži samo paragraf sa kraćim tekstom radi demonstracije prikaza komponente.

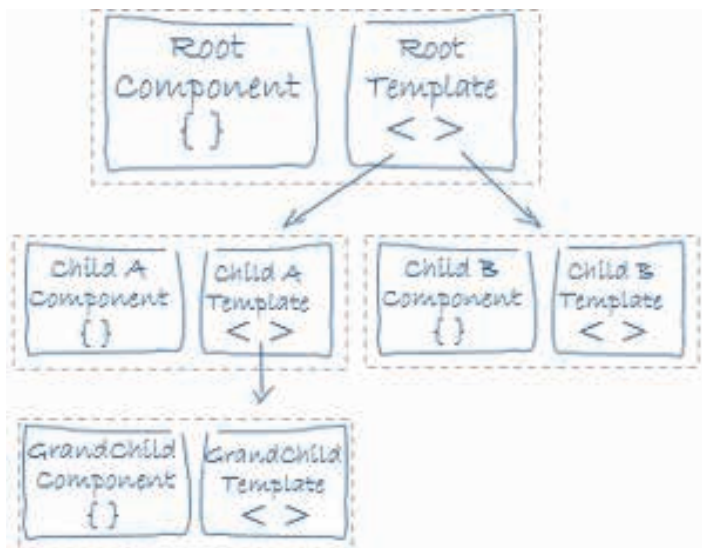
```
//Templejt komponente
<p>component-overview works!</p>
```

Na kraju je dat prikaz *index.html* stranice kroz koju se prikazuje sadržaj korisničkog interfejsa aplikacije. U okviru stranice se prikazuje korena komponenta kao *app-root* tag. Templejt korene komponente dalje prikazuje drugu komponentu (*app-component-overview* iznad). Na ovaj način se generiše prikaz svake aplikacije. Opisani mehanizam je poznat pod nazivom *Single Page Application* (SPA). Jedna ili veći broj izabranih komponenti u datom trenutku se prikazuju kao tag ili skup tagova u okviru jedne, tzv. *single page* stranice. To je *index.html* stranica u kojoj se posredstvom taga korene komponente prikazuju ostale komponente aplikacije. SPA unapređuje korišćenje GKI tako što se prilikom promene tekućeg prikaza ažurira, tj. osvežava samo deo stranice kao sadržaj taga koji prikazuje komponentu. Ovo eliminiše treperenje (eng. *flickering*) stranice u slučaju složenijih sadržaja. Mehanizam koji omogućava prikaz različitih komponenti u toku korišćenja interfejsa je opisan u Sekciji 5.5.

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

component-overview works!

SPA princip je ilustrovan šemom u nastavku. Komponente se dalje mogu ugnježdavati i kreirati hijerarhije.



Slika 5.9. SPA princip i ugnježdavanje komponenti. Izvor: [71].

Više informacija o komponentama se može naći u [71].

5.3.2. Životni ciklus komponente

U prethodnoj sekciji opisana je struktura komponente kao šablona za kreiranje instanci komponenti u vreme izvršavanja aplikacije.

Instanca komponente ima životni ciklus koji počinje kreiranjem komponente i njenim prikazivanjem, uključujući i ugnježdene komponente ukoliko postoje. Ciklus se nastavlja kao detektovanje promena i ažuriranje templejta komponente i kontrolera sa podacima. Na kraju ciklusa se instanca dealocira i uklanja iz DOM stabla aplikacije.

Angular omogućava pristup instanci komponente u tipičnim momentima u toku životnog ciklusa u cilju izvršenja potrebnih obrada.

Pristup se omogućava tako što kontroler komponente implementira odgovarajući interfejs i metodu koju taj interfejs propisuje. Na taj način će se naredbe navedene u metodi izvršiti kada instanca komponente bude u odgovarajućoj fazi životnog ciklusa. Zbog toga su ove metode poznate i pod nazivom *hook methods*.

Interfejsi prate faze životnog ciklusa komponente i imaju sledeće metode:

- *ngOnChanges()* – poziva se prva i kod svake promene podataka (modela) instance komponente;
- *ngOnInit()* – poziva se prilikom inicijalizacije sadržaja (podataka) i izgleda instance komponente;

- *ngDoCheck()* – poziva se u slučaju detekcije promene i uvek nakon prethodnih;
- *ngOnDestroy()* – poziva se pre uklanjanja instance komponente, na primer otkazivanje registrovanih osluškivača događaja.

Tabela 5.2. sumira faze životnog ciklusa instance komponente i interfejsa sa metodama koje treba implementirati radi obrade u odgovarajućoj fazi.

Tabela 5.2. Faze životnog ciklusa instance komponente i odgovarajuće metode sa interfejsima.

Faza životnog ciklusa	Metoda - Interfejs
Kreiranje instance	ngOnInit() - OnInit
Prikaz instance	ngOnInit() - OnInit
Kreiranje i prikazivanje ugnježđenih instanci	ngOnInit() - OnInit
Ažuriranje instance	ngOnChanges() - OnChanges
Uklanjanje instance iz DOM stabla	ngOnDestroy() - OnDestroy

Primer u nastavku pokazuje implementaciju interfejsa *OnInit* i metode interfejsa *ngOnInit()*. Čest je slučaj da se potrebne radnje moraju obaviti po kreiranju komponente kako bi korisnik mogao da interaguje sa komponentom. Zbog toga prilikom kreiranja komponente kontroler podrazumevano implementira ovaj interfejs. U primeru se u metodi interfejsa poziva metoda servisa koji dobavlja podatke komponente.

```
export class HeroListComponent implements OnInit {
  heroes: Hero[] = [];
  selectedHero: Hero | undefined;

  constructor(private service: HeroService) { }

  ngOnInit() {
    this.heroes = this.service.getHeroes();
  }
  selectHero(hero: Hero) { this.selectedHero = hero; }
```

Više informacija o upravljanju životnim ciklusom komponente se može naći u [72].

5.3.3. Povezivanje podataka

Komponentu čine fizički razdvojene datoteke koje su povezane kako bi predstavljale celinu koja ima izgled i ponašanje. Vezu obezbeđuje mehanizam komunikacije kao razmene podataka između kontrolera i modela komponente, i njenog templejta kako bi delovi bili sinhronizovani. Kao posledica promene templejta se reflektuju na kontroler i model. Na primer, ulaz koji prosleđuje korisnik. I obrnuto, promene u kontroleru i modelu se reflektuju na templejt, tj. izgled. Na primer, izlaz kao rezultat obrade ulaza korisnika. Pored toga, podaci se istovremeno mogu prosleđivati u oba smera.

Povezivanje se svodi na deklarativno kreiranje veza između elemenata kontrolera i templejta komponente. Naredbe koje kreiraju ove veze se navode u templejtu (eng. *binding markup*).

Opšte posmatrano, mogu se razmenjivati dve vrste podataka:

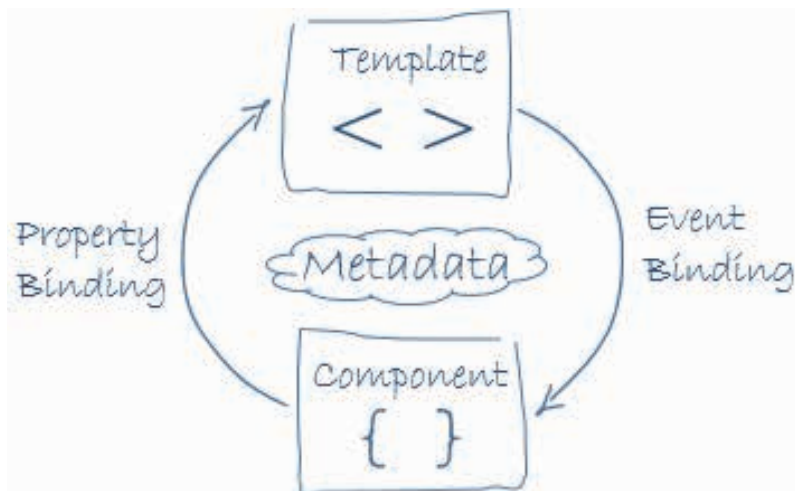
- Objekti događaja kod obrade i odgovaranja na ulaz korisnika, gde su događaji su programski objekti aplikacije.
- Podaci iz programske logike komponente i modela.

Tabela u nastavku daje pregled načina povezivanja podataka templejta i kontrolera komponente. Načini odgovaraju tipičnim zahtevima razmene podataka u korišćenju Veb GKI. Prvi način prosleđuje privremenu vrednost kontrolera kao lokalnu varijablu ili izraz ka templejtu radi prikaza. Kod sledećeg načina vrednost atributa kontrolera se prikazuje u templejtu. Kreiranje programskog objekta događaja u templejtu i prosleđivanje do registrovanog osluškivača je zaseban način prosleđivanja. Dvosmerno povezivanje (eng. *two-way data binding*) omogućava ažuriranje vrednosti taga odgovarajućom vrednošću atributa ili varijable kontrolera, i obrnuto.

Tabela 5.3. Povezivanje podataka – vrsta i smer razmene podataka.

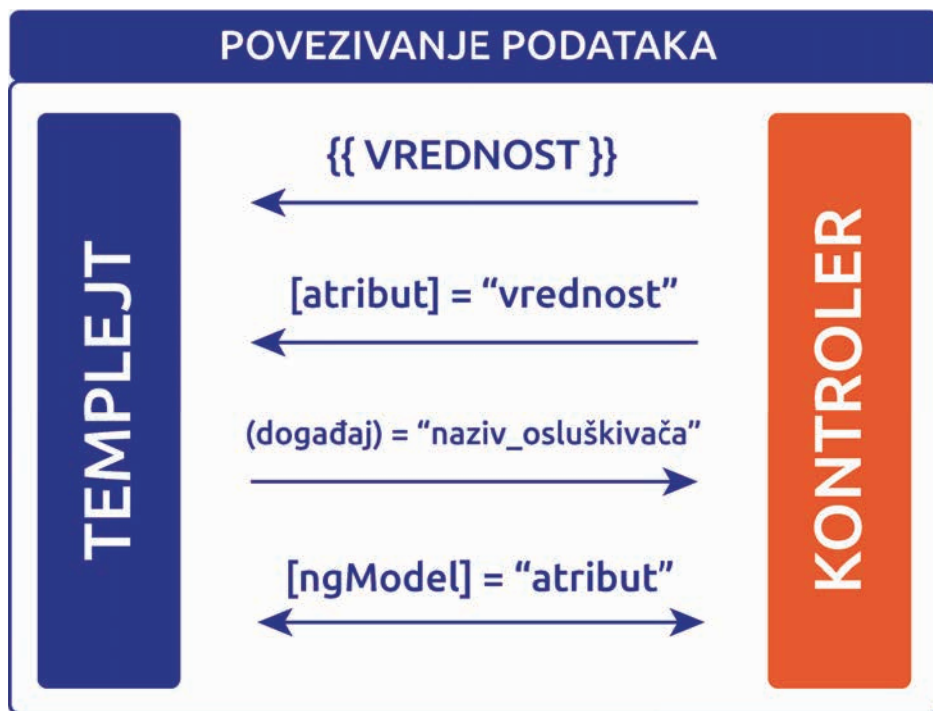
Vrsta podatka	Smer razmene	Sintaksa
Vrednost kontrolera (lokalna varijabla ili vrednost izraza)	Kontroler => Templejt	{{vrednost_izraza}}
Atribut kontrolera	Kontroler => Templejt	[ime_atributa_templejta] = "ime_atributa_kontrolera"
Objekat događaja	Templejt => Kontroler	(ime_događaja) = "ime_metode_oslušivača"
Atribut kontrolera	Templejt <=> Kontroler	([ngModel]) = "ime_atributa_kontrolera"

Slika 5.10. ilustruje povezivanje svojstava (eng. *property binding*) i povezivanje događaja (eng. *event binding*) na nivou komponente. Naziv *Component* označava kontroler komponente.



Slika 5.10. Logička šema povezivanja događaja i svojstva između templejta i kontrolera komponente. Izvor [70].

U vreme izvršavanja aplikacije, sva povezivanja navedena u komponentama će se izvršiti za svaku komponentu i ugnježdene komponente ukoliko postoje. Šema u nastavku daje uporedni prikaz načina povezivanja podataka komponente. Prvi prenosi vrednost iz kontrolera u templejt komponente i poznat je pod nazivom interpolacija (eng. *interpolation*). Povezivanje osobina (eng. *property binding*) kreira vezu između povezuje atributa templejta i atributa kontrolera komponente. Povezivanje događaja (eng. *event binding*) omogućava prenošenje objekta događaja iz templejta u pozadinsku logiku komponente na obradu. Obostrano povezivanje (eng. *two-way data binding*) međusobno sinhronizuje vrednost u templejtu sa vrednošću u kontroleru.



Slika 5.11. Uporedni prikaz načina povezivanja podataka. Podaci se mogu prosledivati od kontrolera ka templejtu, obrnuto ili u oba smera.

Više informacija u vezi sa povezivanjem podataka se može naći u [73].

U nastavku je dat opis svakog načina povezivanja podataka.

5.3.3.1. Interpolacija

Vrednost lokalne varijable ili izraza se prosleđuje templejtu iz kontrolera. Tada se navodi unutar operatora interpolacije kao para duplih vitičastih

zagrada, kao ispod. Vrednost se izračunava i prikazuje kao sadržaj taga templejta.

```
<HTML tag>{{ vrednost }}</HTML tag>
```

Vrednost može biti izraz koji sadrži tekst, operatore i varijable postojećih tipova. Nazivi varijabli moraju biti isti kao oni navedeni u kontroleru. Prepoznavanje i pristup vrednostima varijabli po nazivima je omogućeno zahvaljujući svojstvu *template* dekoratora *@Component* koje povezuje kontroler i templejt komponente.

U primeru ispod, vrednost varijable *korisničkoIme* iz klase kontrolera se prikazuje u templejtu komponente kao sadržaj taga zaglavlja (*h1*).

```
<!-- Templejt komponente -->
<h1>Korisničko ime: {{ korisničkoIme }}</h1>

//Kontroler komponente
class KlasaKomponente {
    korisničkoIme = 'administrator'; }
```

Primeri u nastavku prikazuju interpolaciju vrednosti istoimenih atributa kontrolera različitih tipova.

```
//Numeričke vrednosti
<p>Trošak: {{ cena * količina }}</p>

//String
<p>Zdravo {{ student.ime + " " + student.prezime }}!</p>

//TypeScript objekat
<p> Broj upisa: {{ student.upis }}</p>

//Niz
<p>Ocene(HCI): {{ ocene[3] }}</p>
```

Vrednosti se mogu prikazivati i u željenom formatu korišćenjem sintakse u nastavku.

```
{{ vrednost | format }}
```

Vrednosti tipa Datum se na ovaj način mogu prezentovati u različitim formatima kao na slici ispod.

```
<!-- Default format: output 'Jun 15, 2015'-->
<p>Today is {{today | date}}</p>

<!-- fullDate format: output 'Monday, June 15, 2015'-->
<p>The date is {{today | date:'fullDate'}}</p>

<!-- shortTime format: output '9:43 AM'-->
<p>The time is {{today | date:'shortTime'}}</p>

export class DatePipeComponent {
  today: number = Date.now();
}
```

Slika 5.12. Različiti formati prikaza vrednosti datuma. Templejt (gore) prikazuje vrednost atributa kontrolera (dole). Izvor: <https://angular.io/api/common/DatePipe>.

5.3.3.2. Povezivanje atributa

Kod ovog načina povezivanja uparuju se atribut kontrolera i atribut taga templejta. Rezultat je da se promena atributa kontrolera odražava na atribut-par u tagu templejta koji dobija istu vrednost. Uparivanje se vrši pomoću sintakse u nastavku.

```
[ime_atributa_templejta] = "ime_atributa_kontrolera"
```

Kao posledica, promenom vrednosti atributa kontrolera se menja vrednost njegovog para, tj. atributa templejta. Obrnuto ne važi, tj. promena se prenosi samo od kontrolera ka templejtu.

Za razliku od interpolacije koja menja sadržaj taga, ovaj način utiče na vrednost atributa taga i može menjati detalje izgleda elementa.

Ovaj način se uglavnom koristi u elementima GKI čiji se izgled ili stanje mogu kontrolisati iz programa.

U prvom primeru ispod, *src* atribut *img* taga templejta se povezuje sa atributom *heroImageUrl* kontrolera komponente. Nova vrednost *heroImageUrl* će automatski biti i vrednost *src* atributa. Na ovaj način se sadržaj taga, tj. slika koja se prikazuje može menjati programski. Ovakav način promene izgleda templejta je jednostavniji i efikasniji u poređenju sa pristupom DOM stablu iz JavaScript programa (videti Sekciju 4.4.1).

Drugi primer povezuje *disabled* atribut dugmeta templejta sa atributom *isUnchanged* kontrolera. Na ovaj način se iz programa može menjati stanje dugmeta. Na primer omogućeno dugme za korisnika sa odgovarajućim pravima pristupa funkcijama aplikacije.

```
<img [src]="heroImageUrl">  
<button [disabled]="isUnchanged">Cancel is disabled</button>
```

Slika 5.13. Povezivanje atributa kontrolera sa atributom taga templejta komponente na primeru standardnih elemenata GKI - slika (gore) i dugme (dole). Izvor: [74].

5.3.3.3. Povezivanje događaja

Ovaj način prosleđuje podatke od templejta ka kontroleru kao objekte događaja. Tako osluškivač događaja u kontroleru može vršiti obradu vezanu za nastanak događaja. Izvor događaja i osluškivač se povezuju korišćenjem sintakse u nastavku. Nazivi događaja su predefinisani, kao u Tabeli 1.1.

```
<HTML tag (ime_događaja)="ime_osluškivača()">  
</HTML tag>
```

Događaj se generiše u templejtu, dok se metoda obrađivača definiše i izvršava u kontroleru komponente. Primer u nastavku prikazuje tag dugmeta u templejtu komponente i kod kontrolera. Dugme se proglašava izvorom događaja tako što reaguje na klik korisnika (*click*) i poziva odgovarajuću metodu osluškivača koja će obraditi događaj (*onClickMe()*). Metoda atributu *clickMessage* dodaje tekst kao vrednost. Templejt komponente je definisan kao ugrađen. Ispod dugmeta se vrednost atributa ispisuje korišćenjem interpolacije. Klikom na dugme vrednost atributa će biti ispisana ispod dugmeta.

```
<button (click)="onClickMe()">Click me!</button>
```

```
@Component({
  selector: 'app-click-me',
  template: `
    <button (click)="onClickMe()">Click me!</button>
    {{clickMessage}}`
})
export class ClickMeComponent {
  clickMessage = '';

  onClickMe() {
    this.clickMessage = 'You are my hero!';
  }
}
```

Slika 5.14. Povezivanje događaja između templejta i kontrolera komponente.
Izvor: [74].

5.3.3.4. Obostrano povezivanje podataka

Kod ovog načina se podaci razmenjuju u oba smera, od templejta do kontrolera i obrnuto. Može se posmatrati kao kombinacija povezivanja atributa i povezivanja događaja. Stoga uglaste zagrade `[]` označavaju razmenu podatka od kontrolera ka templejtu, slično povezivanju atributa. Obične zagrade `()` impliciraju razmenu od templejta ka kontroleru kao kod povezivanja događaja. Pored toga, atribut kontrolera se može povezati sa atributom taga templejta (prva varijanta sintakse) ili sa sadržajem taga (druga varijanta sintakse). U drugom slučaju se koristi naredba ili direktiva *ngModel* koja definiše obostrano povezivanje elementa HTML forme i atributa kontrolera. Više reči u vezi sa direktivama će biti u narednoj sekciji.

```
[(ime_atributa_templejta)] =  
"ime_atributa_kontrolera"
```

```
<HTML tag [(ngModel)] =  
"ime_atributa_kontrolera"></HTML tag>
```

Tekstualno polje (*input*) se korišćenjem *ngModel* direktive povezuje sa atributom kontrolera *korIme* u donjem primeru. Direktiva kreira par element templejta - atribut kontrolera tako da se promene jednog dela odražavaju na drugi deo. Na taj način će njihove vrednosti biti identične (ili sinhronizovane) u toku korišćenja interfejsa.

Kada korisnik unese vrednost u tekstualno polje, atribut kontrolera dobija istu vrednost. Ovo je slično kao kod povezivanja događaja sa tim što se kontroleru prosleđuje vrednost polja forme.

U slučaju da se vrednost atributa kontrolera promeni programski, kao posledica izvršavanja funkcije ili naredbi, sadržaj tekstualnog polja će automatski biti promenjen na novu vrednost. Princip je sličan povezivanju atributa, ali u ovom slučaju se u templejtu menja sadržaj polja forme.

```
//Kontroler komponente
import { Component, OnInit } from '@angular/core';

@Component({
  selector: 'app-pozdrav',
  template: `Korisničko ime: <input type="text"
[(ngModel)]="korIme"><br/>
            {{korIme}}`
})

export class Pozdrav implements OnInit {

  constructor() {}

  private korIme: string = "mladen jovanović";

  get korIme(): string {
    return this.korIme;
  }

  set korIme(vrednost: string) {
    this.korIme = vrednost;
  }

  ngOnInit(): void {
  }
}
```

5.3.4. Direktive

U prethodnoj sekciji su opisani mehanizmi razmene podataka između templejta i kontrolera komponente. Direktiva je naredba templejta koja koristi razmenjene podatke kako bi promenila, tj. prilagodila prikaz komponente. Naredbe se navode kao instrukcije u tagovima templejta. U toku prikazivanja komponenti DOM stabla, sve navedene naredbe se izvršavaju i utiču na generisani prikaz.

U zavisnosti od njihovog efekta, postoje dve vrste direktiva:

- ***Strukturne** direktive koje menjaju strukturu i sadržaj templejta, i
- Direktive **[atributa]** koje menjaju izgled ili stanje pojedinih tagova templejta.

U narednim sekcijama je dat opis često korišćenih direktiva. Više informacija u vezi sa direktivama može se naći u [75].

5.3.4.1. Strukturne direktive

Ova vrsta direktiva modifikuje templejt komponente tako što menja, dodaje ili uklanja tagove. Promena izgleda je posledica obrade u kontroleru kao rezultat ulazne akcije korisnika. Direktive odgovaraju upravljačkim naredbama u programskim jezicima i mogu biti:

- **ngIf* – naredba uslovnog grananja,
- **ngFor* – naredba ciklusa ili petlje, i
- **ngSwitch* – naredba višestrukog uslovnog grananja.

NgIf direktiva uslovno prikazuje ili uklanja delove prikaza templejta. Opšti oblik sintakse je kao u nastavku. Tag može biti i selektor komponente. Ukoliko *izraz* vraća vrednost *true* tag i sadržaj će postojati u DOM stablu i biti prikazani. Kada je vrednost *false*, tag i podređeni tagovi neće postojati u DOM stablu i prikazivati se. Ovo doprinosi optimalnom korišćenju memorije Veb pretraživača.

```
<HTML tag *ng-if="izraz">sadržaj_tag</HTML tag>
```

Primer u nastavku ilustruje opisane efekte korišćenja *ngIf* direktive.

```

<p *ngIf="true">
  Expression is true and ngIf is true.
  This paragraph is in the DOM.
</p>
<p *ngIf="false">
  Expression is false and ngIf is false.
  This paragraph is not in the DOM.
</p>

```

Slika 5.15. Princip rada *ngIf* direktive na primeru HTML paragrafa.

U narednom primeru prvi *div* tag templejta će prikazati ime tekućeg korisnika samo ako je vrednost atributa *imeKorisnika* kontrolera različita od *null* ili *undefined*, odnosno definisana. U suprotnom slučaju, biće prikazan sadržaj sledećeg taga sa odgovarajućom porukom.

```

<div *ngIf="imeKorisnika">
<span>{{imeKorisnika}}</span>
</div>

<div *ngIf="!imeKorisnika">
<span>Ime korisnika nije uneto.</span>
</div>

```

Sledeći primer prati sličan princip, ali sa atributom tipa niza (*mojNiz*). Ako niz nema elemenata (početni *div*) prikazuje se odgovarajuća poruka. U slučaju da niz nije prazan (naredni *div*) u poruci korisniku je ispisana dužina niza korišćenjem interpolacije.

```

<div *ngIf="!mojNiz.length">
<span>Niz je prazan.</span>
</div>
<div *ngIf="mojNiz.length">
<span>Niz sadrži {{ mojNiz.length }} elemenata.</span>
</div>

```

NgFor direktiva ponavlja DOM čvor u skladu sa brojem elemenata u kolekciji čiji se članovi prikazuju. Kod korišćenja ove direktive najpre se bira tag, tj. čvor koji će prikazivati sadržaj. Zatim se tag ponavlja za svaki element kolekcije, kao u nastavku. *Kolekcija* je atribut kontrolera tipa kolekcije (niz i druge).

```
<HTML tag *ngFor="let element of kolekcija">
sadržaj_taga
</HTML tag>
```

Primer u nastavku definiše atribut tipa niz u kontroleru komponente (*kolekcija*). Zatim se u templejtu komponente vrednosti elemenata niza prikazuju kao stavke (*li*) neuređene liste (*ul*). Sadržaj stavke se dobavlja korišćenjem interpolacije.

```
//Kontroler komponente
import { Component } from '@angular/core';

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
})
export class AppComponent {
  kolekcija = ['element1', 'element2', 'element3',
'element4'];
}

<!-- Templejt komponente -->
<ul>
  <li *ngFor='let element of kolekcija'> {{element}}
</li>
</ul>
```

NgSwitch kombinuje skup direktiva od kojih svaka prikazuje alternative sadržaja za dati deo templejta. Sastoji se od tri direktive:

- **ngSwitch* – definiše i proverava referentni skup vrednosti (eng. *switch value*) za definisanje alternativa izgleda.
- **ngSwitchCase* – za svaku alternativu dodaje DOM čvor koji prikazuje odgovarajući sadržaj u slučaju da pridružena vrednost direktive odgovara vrednosti iz referentnog skupa.
- **ngSwitchDefault* – dodaje DOM čvor kao podrazumevani sadržaj ukoliko nije izvršena nijedna *ngSwitchCase* direktiva.

Primer u nastavku ilustruje korišćenje pojedinačnih direktiva. *NgSwitch* direktiva se povezuje sa atributom kontrolera *radniDan* (povezivanje atributa) čija vrednost postaje referentna. Ova direktiva je direktiva atributa (videti narednu sekciju) zbog toga što ne menja sadržaj templejta, već definiše specifično stanje koje utiče na promenu sadržaja. *NgSwitchCase* konsultuje referentnu vrednost i prikazuje odgovarajući sadržaj. Ovde je to naziv dana u nedelji. Ukoliko atribut *radniDan* ima vrednost koja ne pripada referentnom skupu ([1,2,3,4,5]), tada se prikazuje sadržaj naveden u *ngSwitchDefault* direktivi. Za vrednost atributa kao u primeru, biće prikazan *div* element sa tekstom *Utorak*.

```
//Kontroler komponente
import { Component } from '@angular/core';

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent {
  radniDan: number = 2;
}

<!-- Templejt komponente -->
<div [ngSwitch]="radniDan">
  <div *ngSwitchCase="'1'">Ponedeljak</div>
  <div *ngSwitchCase="'2'">Utorak</div>
  <div *ngSwitchCase="'3'">Sreda</div>
  <div *ngSwitchCase="'4'">Četvrtak</div>
  <div *ngSwitchCase="'5'">Petak</div>
  <div *ngSwitchDefault>Ne postoji dan za dati
  broj.</div>
</div>
```

5.3.4.2. Direktive atributa

Ovaj tip direktiva menja izgled ili stanje tagova templejta. Navode se slično kao atributi taga i najčešće se koriste sledeće:

- *[ngModel]* – omogućava obostrano povezivanje podataka.
- *[ngClass]* – dodaje ili uklanja eksterne CSS klase koje se primenjuju na tag.
- *[ngStyle]* – dodaje ili uklanja ugrađene CSS stilove koji se primenjuju na tag.

U praksi se češće koristi *ngModel* direktiva. Direktiva omogućava sinhronizaciju elementa templejta (na primer, tekstualno polje forme) i atributa kontrolera komponente. Kada korisnik unese vrednost u tekstualno polje templejta, ta vrednost će automatski biti dodeljena atributu kontrolera. Važi i obrnuto, tj. kada se vrednost atributa kontrolera promeni (na primer, selekcija objekta korisnika iz pozadinske logike), tekstualno polje će prikazivati tu vrednost.

Primer u nastavku ilustruje opisani mehanizam. Kontroler komponente sadrži atribut *podaci* čija je vrednost instanca klase *Podaci*. Korišćenjem direktive tekstualno polje forme templejta se povezuje sa atributom *ime* instance klase. Na ovaj način će vrednost atributa *ime* navedena prilikom instanciranja objekta klase biti prikazana u tekstualnom polju templejta. Ukoliko korisnik unese vrednost u tekstualno polje, vrednost atributa *podaci* kontrolera će biti ažurirana na unetu vrednost.

```
//Podaci.ts
export class Podaci {
  ime: string;
  constructor(ime: String) {
    this.ime = ime;
  }
}

//App.component.ts
import { Component } from '@angular/core';

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})

export class AppComponent {
  naslov: string = 'Aplikacija';
  podaci: Podaci = new Podaci('Mladen Jovanović');
}

<!-- App.component.html -->
<input type="text" [(ngModel)]="podaci.ime" />
```


5.4. Servisi

Servis je gradivni element sa skupom funkcionalnosti ili vrednosti koje mogu koristiti komponente i moduli, pa čak i različite aplikacije. Za razliku od komponente, nema izgled i zbog toga je realizovan kao TypeScript klasa sa specifičnim dekoratorom. Svrha servisa je slična interfejsu u drugim objektno-orijentisanim jezicima, sa tim što servis može sadržati i atribute.

Koncept servisa podržava ponovnu upotrebljivost i modularnost. *Ponovna upotrebljivost* se ogleda u izdvajanju zajedničkih funkcionalnosti i vrednosti tako da ih mogu koristiti različiti moduli, komponente i aplikacije. *Modularnost* podrazumeva da je glavna uloga komponenta interaktivna logika sa korisnikom, dok servisi implementiraju funkcionalnosti koje podržavaju ovu logiku. Komponenta koristi funkcionalnosti servisa po potrebi. Ovo olakšava razvoj i održavanje aplikacije.

Primeri funkcionalnosti servisa mogu biti komunikacija sa lokalnom ili udaljenom serverskom logikom, obrada podataka pristiglih od serverske strane, čuvanje i obrada akcija korisnika, provera unetih podataka od strane korisnika i operacije nad modelom podataka aplikacije.

Servis je TypeScript klasa sa dekoratorom *@Injectable* koja implementira skup funkcija.

```
ng generate service ime_servisa  
ng g s ime_servisa
```

Naredba iznad kreira datoteku servisa sa nazivom kao u nastavku.

```
ime_servisa.service.ts
```

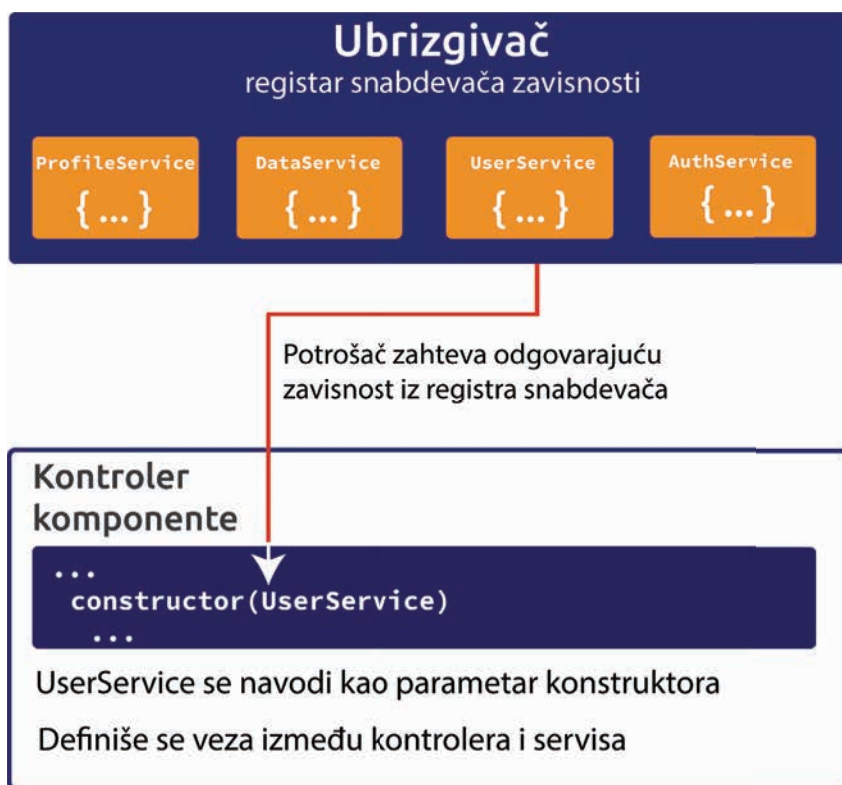
Umetanje zavisnosti (eng. *Dependency Injection*) je mehanizam uspostavljanje veza između između Angular klasa sa dekoratorima. Ove veze se nazivaju zavisnosti zbog toga što jedna klasa koristi podatke ili funkcije druge klase, tj. zavisi od nje.

Veza definiše dve uloge - potrošač zavisnosti (eng. *dependency consumer*) i snabdevač zavisnosti (eng. *dependency provider*). Potrošač koristi podatke i funkcije kao usluge koje pruža snabdevač.

Za uspostavljanje i održavanje veza je zadužen predefinisani Angular objekat pod nazivom *Ubrizgivač* (eng. *Injector*). Objekat sadrži registar (eng. *registry*) kreiranih zavisnosti. Kada potrošač zahteva zavisnost uslugu, objekat proverava da li u registru postoji veza sa odgovarajućim snabdevačem,

odnosno da li je instanca snabdevača na raspolaganju ili kreirana. Ukoliko instanca ne postoji, nova se kreira i čuva u registru. Svaka aplikacija ima instancu objekta i u većini slučajeva nema potreba za dodatnim instancama. Dodatni objekti se mogu kreirati programski, po potrebi.

Mehanizam se često koristi za definisanje veze između kontrolera komponente i servisa. Servis se navodi kao parametar konstruktora kontrolera. Prilikom kreiranja instance komponente, analiziraju se parametri konstruktora i proverava da li instance pronađenih servisa postoje. Ukoliko ne postoje, one se kreiraju i prosleđuju kao parametri konstruktora i registruju u objektu *Ubrizgivač*.



Slika 5.16. Ubrizgivač predstavlja registar snabdevača zavisnosti. Snabdevači su najčešće servisi.

U primeru iznad, zavisnost se umeće između komponente (kontrolera) i servisa. Umetanje zavisnosti ili registracija servisa se može definisati na tri nivoa:

1. Nivo aplikacije – jedna instanca servisa se kreira za aplikaciju.
2. Nivo modula – jedna instanca servisa se kreira za dati modul.
3. Nivo komponente – jedna instanca servisa se kreira za datu komponentu.

Kod registracije na nivou aplikacije postoji jedna instanca servisa se pravi za čitavu aplikaciju. Drugim rečima, sve komponente dele jednu instancu. U dekoratoru *@Injectable* klase servisa sa kao vrednost svojstva *providedIn* navodi vrednost *root*, kao u primeru ispod.

```
@Injectable ({
    providedIn: 'root'
})
```

Registracija na nivou modula kreira jednu instancu servisa za sve komponente i module koji pripadaju datom modulu. Drugim rečima, komponente datog modula dele jednu instancu servisa. U *@NgModule* dekoratoru klase modula se kao vrednost svojstva *providers* navodi ime servisa. Svojstvo je tipa niz tako da se proizvoljan broj različitih servisa može registrovati za dati modul.

```
@NgModule ({
    providers: ['UserService',
'BackendService']
})
```

Registracija na nivou komponente kreira instancu servisa za svaku kreiranu instancu date komponente. U *@NgComponent* dekoratoru klase, tj. kontrolera komponente se kao vrednost svojstva *providers* navodi ime servisa. Svojstvo je tipa niz tako da se proizvoljan broj različitih servisa može registrovati za datu komponentu.

```
@Component({
    selector: 'user',
    templateUrl: './user.component.html',
    styleUrls: ['./user.component.css'],
    providers: [UserService]
})
```

U nastavku je dat primer servisa. Servis *UserService* radi sa tipom *User* koji je definisan kao interfejs. Tip opisuje korisnika skupom svojstava. Servis ima dva atributa - prvi ima vrednost tekućeg korisnika (*currentUser*) i drugi ima vrednost niza objekata korisnika (*dummyUserList*). Drugi atribut simulira podatke koji bi se inače dobavili iz pozadinske logike aplikacije, a kako bi GKI imao podatke za prikaz i rad. Metode servisa vraćaju ime korisnika iz

objekta korisnika i objekat korisnika na osnovu identifikatora korisnika, respektivno.

```
import { Injectable } from '@angular/core';

export interface User {
  id: number;
  email: string;
  password: string;
  firstName: string;
  lastName: string;
  phone: string;
  address: string;
  type: 'driver' | 'passenger';
  rating: number;
}

@Injectable()
export class UserService {
  currentUser: User = UserService.dummyUserList[0];
  static dummyUserList: Array<User> = [
    {
      id: 1,
      email: "test1@test.com",
      password: "test123",
      firstName: "Test",
      lastName: "Test",
      phone: "+3816711223344",
      address: "Test Adresa 1",
      type: "driver",
      rating: 4.5,
    },
    {
      id: 2,
      email: "test2@test.com",
      password: "test123",
      firstName: "Mladjan Test",
      lastName: "Jovanovic Test",
    }
  ]
}
```

```

        phone: "+3816711223344",
        address: "Test Adresa 2",
        type: "passenger",
        rating: 4.9,
    }
]

getUsersFullName(user: User): string {
    return user.firstName + ' ' + user.lastName
}

getUserById(id: number): User {
    var foundUser: User;
    UserService.dummyUserList.forEach(user => {
        if (user.id === id) {
            foundUser = user;
        }
    });
    return foundUser;
}
}

```

Sledeći primer koristi prethodno kreirani interfejs *User* i servis *UserService* za funkcije prijave i registracije korisnika za korišćenje aplikacije. Prva metoda vraća objekat korisnika na osnovu adrese elektronske pošte kao korisničkog imena za prijavu. Druga metoda proverava ispravnost unete lozinke. Treća metoda dodaje novog korisnika.

```

import { Injectable } from '@angular/core';
import { User, UserService } from './user.service';

@Injectable()
export class AuthService {
    registeredUsers: Array<User> =
        UserService.dummyUserList;

    getUser(email: string): string {
        return this.registeredUsers.find(userToFind =>

```

```

        userToFind.email === userEmail);
    }

    isPasswordCorrect(userEmail: string, password:
    string): boolean {
        return this.registeredUsers.find(userToFind =>
(userToFind.email === userEmail &&
    userToFind.password === password)) !=
    undefined;
    }

    registerUser(email: string, password: string,
    firstName: string,
    lastName: string, phone: string, address:
    string, type): User {
        var maxId: number = 0;
        this.registeredUsers.forEach(user => {
            if (maxId < user.id) {
                maxId = user.id;
            }
        });
        var id = ++maxId;
        var user: User = {id, email, password,
firstName, lastName, phone, address, type,
rating: 0};
        this.registeredUsers.push(user);
        UserService.dummyUserList.push(user);

        return user;
    }
}

```

Primeri servisa ilustruju pogodan način izrade interaktivnog prototipa GKI bez pozadinske logike koja sadrži prave podatke aplikacije. Koraci su sledeći:

1. Kreiranje interfejsa koji definišu strukturu objekta aplikacije sa testnim podacima.
2. Definisanje servisa koji implementiraju funkcije čitanja, izmene, dodavanja i brisanja objekata tipa interfejsa.

Testni objekti se kreiraju kao kolekcija objekata interfejsa (na primer, TypeScript Array).

Više informacija u vezi sa korišćenjem servisa se može naći u [76].

5.5. Rutiranje

Tipičan model navigacije Veb aplikacije podrazumeva da korisnik bira ili unosi URL adresu i pretraživač otvara željenu stranicu. Birajući linkove ka stranicama, pretraživač dalje upućuje korisnika na nove stranice. Standardna *Forward* i *Back* dugmad pretraživača se mogu koristiti za kretanje kroz istorijat posećenih stranica.

Okvir realizuje kretanje kroz GKI na sličan način, sa tim što navigacija vodi do komponenti čiji se templejt prikazuje.

Rutiranje ili ruter je mehanizam koji implementira navigaciju tako što definiše putanje do komponenti aplikacije i omogućava njihovo prikazivanje.

U tabeli u nastavku dat je opis Angular modula koji implementiraju specifične elementa rutiranja na nivou okvira.

Tabela 5.4. Predefinisani moduli rutiranja.

Element	Opis
<i>RouterModule</i>	Implementacija funkcionalnosti rutiranja, tj. navigacije do komponenti.
<i>Routes</i>	Struktura zapisa ruta koja se navodi kao kolekcija elemenata <i>putanja_do_komponente : ime_komponente</i> .
<i>RouterOutlet</i>	HTML direktiva (<code><router-outlet></code>) koja definiše mesto prikazivanja templejta tekuće komponente, tj. one do koje vodi trenutna ruta.
<i>RouterLink</i>	Atribut HTML taga koji sadrži rutu do komponente.

Implementacija rutiranja obuhvata sledeće korake:

1. Učitavanje potrebnih modula,
2. Definisane ruta,
3. Određivanje mesta prikazivanja komponenti na koje ukazuju rute,
4. Kreiranje linkova ka rutama za korisnika.

Ruter je modul koji se može kreirati prilikom generisanja Angular aplikacije, tj. projekta ili naknadno (videti Sekciju 8.1).

Pre definisanja ruta do komponenti aplikacije, potrebno je učitati odgovarajuće module. Moduli implementiraju mehanizam rutiranja (*RouterModule*) i strukturu podataka rute (*Routes*).

```
import { NgModule } from '@angular/core';
import { Routes, RouterModule } from '@angular/router';
```

Na nivou Angular aplikacije se kreira jedna instanca rutera (eng. *singleton*). Rute su predstavljene kao parovi u nastavku. Naziv komponente odgovara imenu koje se navodi prilikom kreiranja komponente.

```
putanja_do_komponente : naziv_komponente
```

Primer u nastavku ilustruje tipičnu strukturu i sadržaj rutera. Varijabla *rute* sadrži niz objekata ruta. Svaka se navodi kao objekat sa dva svojstva. Prvo svojstvo (*path*) sadrži relativnu putanju do direktorijuma komponente bez simbola '/' (eng. *leading slash*). Prazna putanja se navodi za komponentu koja se podrazumevamo prikazuje kada korisnik pokrene aplikaciju (eng. *home page*). Sledeće svojstvo (*component*) kao vrednost prihvata naziv komponente do koje vodi putanja. Svojstva *@NgModule* dekoratora imaju specifične vrednosti. *Imports* svojstvo kao vrednost dobija *RouterModule* instancu koja je inicijalizovana prethodno definisanim rutama pomoću metode *forRoot()*. Metoda konfiguriše ruter za konkretnu aplikaciju. Navođenje *RouterModule* kao vrednost *exports* svojstva čini ruter vidljivim u čitavoj aplikaciji. Ruter identifikuje komponentu koja se prikazuje kao vrednost *component* svojstva na osnovu vrednosti *path* svojstva. Vrednost *path* svojstva odgovara putanji koju korisnik zadaje u toku korišćenja GKI.

```
import { NgModule } from '@angular/core';
import { Routes, RouterModule } from '@angular/router';

//Varijabla čija vrednost sadrži rute
const rute: Routes = [
  {
    path: '',
    component: HomeComponent
  },
  {
```



```

        path: 'registracija',
        component: RegisterComponent
    },
    {
        path: 'prijava',
        component: LoginComponent
    }
];

//Konfigurisanje dekoratora
@NgModule({
    imports: [RouterModule.forRoot(rute)],
    exports: [RouterModule]
})

export class AppRoutingModule {}

```

Nakon kreiranja rutera i definisanja ruta, određuje se mesto u GKI na kojem će se prikazivati tekuća komponenta, tj. ona na koju ukazuje trenutna ruta. *Router-outlet* tag označava mesto na kojem se prikazuje sadržaj komponente do koje vodi trenutna ruta. Najčešće se navodi u templejtu korene komponente (*app.component.html*).

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

Na kraju je potrebno kreirati linkove ka rutama za korisnika, odnosno definisati deo za navigaciju. Mogu se navoditi kao HTML linkovi (*<a>* tag) sa direktivom *routerLink* koja ukazuje na konkretnu rutu. Vrednosti direktiva moraju odgovarati onima koje su navedene u definiciji ruta (*path* svojstva). Za razliku od vrednosti *path* svojstva, putanje se navode sa '/' prefiksom.

```

<!-- Navigacioni meni sa linkovima ka rutama -->
<nav>
  <ul>
    <li>
      <a routerLink="/registracija">
        Registracija
      </a>
    </li>
    <li>
      <a routerLink="/prijava">
        Prijava

```

```
        </a>
    </li>
</ul>
</nav>
```

6. Angular Material elementi korisničkog interfejsa

Angular Material (AM) je biblioteka standardnih elemenata GKI [78]. Termin komponenta se koristi za svaki element. Značenje termina se razlikuje od Angular komponente kao gradivnog elementa aplikacije. Biblioteka je zasnovana je na *Google Material Design (GMD)* preporukama [79]. Preporuke su smernice dizajna i implementacije korisnih i upotrebljivih GKI i obuhvataju elemente kao što su GKI kontrole, boje, oblici, simboli, vizuelni raspored i drugi.

AM komponente se na osnovu uloge koju obavljaju mogu podeliti u sledeće kategorije:

- Komponente *forme* predstavljaju deo logički povezane grupe podataka koji se prosleđuju i obrađuju kao celina. Na primer, tekstualno polje, radio dugme, polje za potvrdu i druge.
- Komponente *navigacije* omogućavaju kretanje i orijentaciju u GKI, poput vertikalne i horizontalne linije alata.
- Komponente *akcija* i *indikatora* služe za zadavanje instrukcija GKI i pružanje povratne informacije korisniku, kao što su različite vrste dugmadi i progres element.
- Komponente *dijaloga* korisnika obaveštavaju o značajnijim radnjama i promenama u GKI preko prozora sa porukama i akcijama.
- Komponente *tabele* služe za kreiranje tabela sa pratećim funkcionalnostima kao što su sortiranje, straničenje i filtriranje.

Mogu se posmatrati i kao unapređene verzije standardnih HTML tagova za elemente GKI. Detalji o nameni i upotrebi elemenata GKI se mogu naći u [39].

Pravilna upotreba AM komponenti može unaprediti razvoj Web korisničkih interfejsa na dva načina. Prvi predstavlja povećanje *efikasnosti* korišćenjem predefinisanih elemenata GKI koji ispunjavaju kriterijume upotrebljivosti i korisnosti interfejsa. Efikasnost se ogleda u optimalnom korišćenju vremena i dostupnih ljudskih resursa. Sledeći se odnosi na podizanje *efektivnosti* (ili kvaliteta) kreiranog GKI kroz implementaciju prakse dobrog dizajna interfejsa (GMD smernice).

Zvanična dokumentacija AM biblioteke [78] opisuje GKI elemente kao komponente (odjeljak *Components*), i zajedničke funkcionalnosti komponenti (odjeljak *Component Development Kit*) poput prevlačenja, skrolovanja, straničenja i drugih.

AM komponente se dodaju Angular komponentama kao odgovarajuće naredbe templejta i kontrolera. Korišćenje AM je opciono i tada se mora instalirati biblioteka kao dodatak aplikaciji. U suprotnom se mogu koristiti i standardni HTML elementi GKI.

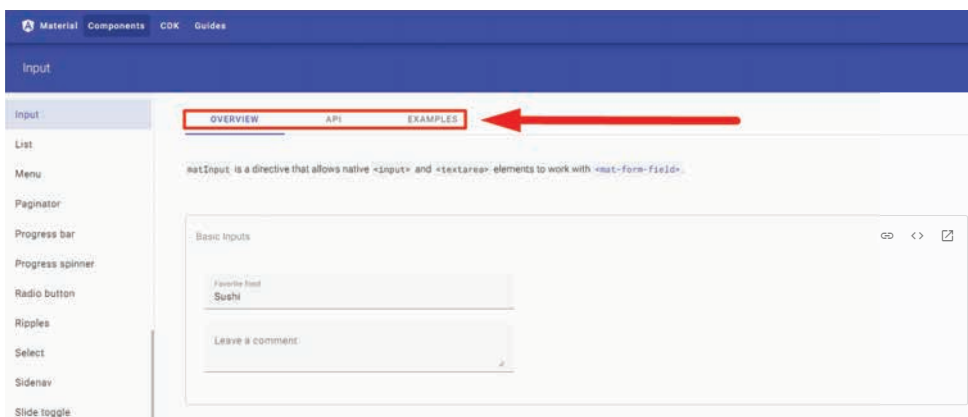
Slika 6.1. prikazuje kolekciju AM komponenti kao elemenata GKI.



Slika 6.1. Kolekcija AM komponenti kao elemenata GKI. Izvor: [78].

Dokumentacija svake komponente je organizovana u tri sekcije:

1. Pregled (eng. *Overview*),
2. API,
3. Primeri (eng. *Examples*).



Slika 6.2. Dokumentacija AM komponente. Izvor: [80].

Sekcija *Pregled* daje opis komponente sa svojstvima i metodama. API sekcija prikazuje načine programskog pristupa i korišćenja svojstava i metoda. Sekcija *Primeri* daje konkretne slučajeve korišćenja komponente, sa izvornim programskim kodom, kao na slici u nastavku.



Slika 6.3. Primeri korišćenja AM komponente. Izvor: [80].

6.1. Instalacija

Naredba u nastavku instalira biblioteku.

```
ng add @angular/material
```

Detalji instalacije su prikazani na Slici 6.4.

```
i Using package manager: npm
✓ Found compatible package version: @angular/material@13.3.9.
✓ Package information loaded.

The package @angular/material@13.3.9 will be installed and executed.
Would you like to proceed? Yes
✓ Package successfully installed.
? Choose a prebuilt theme name, or "custom" for a custom theme: Indigo/Pink
  view: https://material.angular.io?theme=indigo-pink ]
? Set up global Angular Material typography styles? Yes
? Set up browser animations for Angular Material? Yes
UPDATE package.json (1137 bytes)
✓ Packages installed successfully.
UPDATE src/app/app.module.ts (423 bytes)
UPDATE angular.json (3215 bytes)
UPDATE src/index.html (576 bytes)
UPDATE src/styles.css (181 bytes)
```

Slika 6.4. Proces instalacije AM biblioteke komponenti GKI.

Nakon uspešne instalacije, u projektu se ažurira datoteka *package.json*, koja sadrži nazive i verzije instaliranih biblioteka. Kaže se da aplikacija zavisi od instaliranih biblioteka tako što koristi njihove funkcionalnosti. Zbog toga se za instalacije koristi i naziv zavisnosti (eng. *dependencies*). AM biblioteka se instalira pod nazivom *@angular/material*, kao na slici u nastavku.

```
"dependencies": {  
  "@angular/animations": "^15.0.0",  
  "@angular/common": "^15.0.0",  
  "@angular/compiler": "^15.0.0",  
  "@angular/core": "^15.0.0",  
  "@angular/forms": "^15.0.0",  
  "@angular/material": "^15.0.4",  
  "@angular/platform-browser": "^15.0.0",  
  "@angular/platform-browser-dynamic": "^15.0.0",  
  "@angular/router": "^15.0.0",  
  "rxjs": "~7.5.0",  
  "tslib": "^2.3.0",  
  "zone.js": "~0.12.0"  
},
```

Slika 6.5. AM instalacija (*@angular/material*) u *package.json* datoteci.

6.2. Grupisanje AM komponenti

Svaka AM komponenta ima svoju implementaciju kao modul. Ova vrsta modula ima tipičan naziv koji se sastoji iz prefiksa *Mat*, imena elementa GKI i sufiksa *Module*.

MatIme_elementa_GKIModule

Na taj način nazivi modula za odgovarajuće elemente GKI su *MatButtonModule* (dugme), *MatIconModule* (ikona), *MatListModule* (lista) i drugi. Svi moduli se nalaze u *@angular/material* biblioteci.

Definicije svih komponenti nalaze u instaliranoj biblioteci. Međutim, komponente koje se koriste u aplikaciji se moraju eksplicitno navesti u odgovarajućem modulu. To može biti koreni modul aplikacije ili poseban modul. Preporuka je imati modul koji evidentira korišćene komponente na sledeći način:

1. Kreiranje modul za registrovanje AM komponenti u projektu (na primer, *material.module.ts*).
2. Registracija modula u korenom modulu aplikacije (*app.module.ts*).
3. Prilikom prvog korišćenja AM komponente, navesti modul komponente u kreiranom modulu.

AM komponenta registrovana na gore opisan način je dostupna svim komponentama aplikacije.

Primer u nastavku prikazuje modul za registrovanje AM komponenti (*material.module.ts*). Modul registruje AM komponentu dugme. Ime modula koji implementira se navodi na odgovarajućim mestima, tj. u *import* naredbi i kao vrednost svojstva *imports* i *exports* dekoratora *@NgModule*.

```
//Modul za registraciju AM komponenti
import { NgModule } from '@angular/core';
import { MatButtonModule } from '@angular/material/button';

@NgModule({
  imports: [MatButtonModule],
  exports: [MatButtonModule]
})

export class MaterialModule {}
```

Kreirani modul se zatim registruje u korenom modulu aplikacije kao u nastavku.

```
//Koreni modul aplikacije
import { MaterialModule } from './material.module';
@NgModule({
  declarations: [
    AppComponent
  ],
  imports: [
    BrowserModule,
    BrowserAnimationsModule,
    MaterialModule
  ],
  providers: [],
  bootstrap: [AppComponent]
})
```

```
export class AppModule { }
```

6.3. Teme

AM tema je skup boja koje se primenjuju na AM komponentama GKI. Tema se sastoji od pet paleta boja, koje se koriste za vizuelizaciju tipičnih stanja AM komponenti. U skladu sa tim, palete boja mogu biti:

- Primarna (eng. *primary*) paleta predstavlja podrazumevanu boju komponente.
- Naglašena (eng. *accent*) paleta se koristi za interaktivne komponente.
- Paleta upozorenja (eng. *warn*) sadrži boje za kritična stanja poput grešaka.
- Frontalna (eng. *foreground*) paleta nudi boje za tekst i ikone, tj. elemente komponente u prvom planu.
- Pozadinska (eng. *background*) paleta obezbeđuje boje za pozadine elemenata.

Korišćenje tema obezbeđuje konzistentnost GKI zbog toga što komponente imaju istu boju u različitim delovima interfejsa. Pored toga, naglašavanje različitih stanja drugačijim bojama korisniku pruža bolju povratnu informaciju o stanju elementa interfejsa.

Instalacija AM sadrži četiri predefinisane teme, kao u nastavku.

```
deeppurple-amber.css  
indigo-pink.css  
pink-bluegrey.css  
purple-green.css
```

Definicije predefinisanih teme se nalaze u direktorijumu *node_modules/@angular/material/prebuilt-themes* izvornog koda projekta. Tema koja se primenjuje na interfejs se mora eksplicitno navesti u datoteci stila aplikacije (*styles.css*).

```
@import '~@angular/material/prebuilt-  
themes/indigopink.css';
```

Nazivi i sadržaj predefinisanih tema odgovaraju nazivima paleta boja GMD preporuka. Slika ispod prikazuje palete boja tema *deeppurple-amber.css* i *indigo-pink.css*.

Deep purple 50	#EDE7F6	Amber 50	#FFF8E1
100	#D1C4E9	100	#FFECB3
200	#B39DDB	200	#FFE082
300	#9575CD	300	#FFD54F
400	#7E57C2	400	#FFCA28
500	#673AB7	500	#FFC107
600	#5E35B1	600	#FFB300
700	#512DA8	700	#FFA000
800	#4527A0	800	#FF8F00
900	#311B92	900	#FF6F00
A100	#B388FF	A100	#FFE57F
A200	#7C4DFF	A200	#FFD740
A400	#651FFF	A400	#FFC400
A700	#6200EA	A700	#FFAB00

Indigo 50	#E8EAF6	Pink 50	#FCE4EC
100	#C5CAE9	100	#F8BBD0
200	#9FA8DA	200	#F48FB1
300	#7986CB	300	#F06292
400	#5C6BC0	400	#EC407A
500	#3F51B5	500	#E91E63
600	#3949AB	600	#D81B60
700	#303F9F	700	#C2185B
800	#283593	800	#AD1457
900	#1A237E	900	#880E4F
A100	#8C9EFF	A100	#FF80AB
A200	#536DFF	A200	#FF4081
A400	#3D5AFE	A400	#F50057
A700	#304FFE	A700	#C51162

Slika 6.6. Predefinisane palete boja tema *deppurple-amber.css* (levo) i *indigo-pink.cs* (desno). Izvor: [79].

Slika 6.7. prikazuje palete boja tema *pink-bluegrey.css* i *purple-green.css*.

Pink 50	#FCE4EC		
100	#F8BBD0		
200	#F48FB1	Blue Gray 50	#ECEFF1
300	#F06292	100	#CFD8DC
400	#EC407A	200	#B0BEC5
500	#E91E63	300	#90A4AE
600	#D81B60	400	#78909C
700	#C2185B	500	#607D8B
800	#AD1457	600	#546E7A
900	#880E4F	700	#455A64
A100	#FF80AB	800	#37474F
A200	#FF4081	900	#263238
A400	#F50057		
A700	#C51162		

Purple 50	#F3E5F5	Green 50	#E8F5E9
100	#E1BEE7	100	#C8E6C9
200	#CE93D8	200	#A5D6A7
300	#BA68C8	300	#81C784
400	#AB47BC	400	#66BB6A
500	#9C27B0	500	#4CAF50
600	#8E24AA	600	#43A047
700	#7B1FA2	700	#388E3C
800	#6A1B9A	800	#2E7D32
900	#4A148C	900	#1B5E20
A100	#EA80FC	A100	#B9F6CA
A200	#E040FB	A200	#69F0AE
A400	#D500F9	A400	#00E676
A700	#AA00FF	A700	#00C853

Slika 6.7. Predefinisane palete boja tema *pink-bluegrey.css* (levo) i *purple-green.css* (desno). Izvor [79].

Paleta se mogu eksplicitno navoditi u okviru AM komponenti. U primeru u nastavku, *mat-button* direktiva označava dugme kao AM komponentu i navodi se u okviru HTML taga dugmeta (*button*). Atribut *color* kao vrednost prima naziv palete boja važeće teme (*deeppurple-amber.css*).

```
<button mat-button color="primary">Primary</button>
<button mat-button color="accent">Accent</button>
<button mat-button color="warn">Warn</button>
```



Pored predefinisanih tema, moguće je kreirati temu za potrebe konkretne aplikacije. Kreiranje ovakve teme je opisano u Sekciji 8.9.2.

Više informacija u vezi sa temama se može naći u [83].

6.4. Animacije AM komponenti

AM omogućava korišćenje unapred definisanih animacija kao različitih oblika tranzicija (suptilni pomeraji i promene boja, oblika i pozicija). Animacije su definisane kao modul pod nazivom *BrowserAnimationsModule*. Da bi se koristile, tj, primenjivale na AM komponente, modul se mora registrovati u korenom modulu aplikacije (*app.module.ts*).

```
//Koreni modul aplikacije
import { NgModule } from '@angular/core';
import { BrowserModule } from '@angular/platform-
browser';
import
{ BrowserAnimationsModule } from '@angular/platform-
browser/animations';
import { AppComponent } from './app.component';
import { MapComponent } from
'./filter/map/map.component';
import { TravelComponent } from
'./travel/travel.component';
import { ProfileComponent } from
'./auth/profile/profile.component';

@NgModule({
```

```

declarations: [
  AppComponent,
  MapComponent,
  TravelComponent,
  ProfileComponent
],
imports: [
  BrowserModule,
  BrowserAnimationsModule,
  MaterialModule,
  RoutingModule,
  FlexLayoutModule,
  FormsModule,
],
providers: [TravelService, UserService],
bootstrap: [AppComponent]
})
export class AppModule { }

```

U nastavku je dat opis često korišćenih AM komponenti. Više informacija u vezi sa svakom AM komponentom se može naći u [80], odakle se može pristupiti dokumentaciji komponente.

6.5. Dugme

AM dugme (eng. *button*) se navodi pomoću *mat-button* direktive u okviru HTML *button* taga.

```
<button mat-button>naziv_dugmeta</button>
```

Direktiva ukazuje da je dugme AM tip dugmeta. Dugme se najpre mora navesti u modulu za registraciju AM komponenti (*material.module.ts*).

```
import { MatButtonModule } from
 '@angular/material/button';
```

U istom modulu implementacija AM dugmeta (*MatButtonModule*) se dodaje kao vrednost *imports* i u *exports* svojstava *@NgModule* dekoratora.

AM sadrži varijante dugmeta koje se navode korišćenjem različitih direktiva u okviru *button* taga. Varijante na različite načine ističu dugme u odnosu na druge elemente GKI i mogu biti:

- Istaknuto dugme (eng. *raised button*),
- Oivičeno dugme (eng. *stroked button*),
- Ravno dugme (eng. *flat button*),
- Dugme ikona (eng. *icon button*),
- Kružno dugme (eng. *FAB button*), i
- Manje kružno dugme (eng. *mini FAB button*).

Istaknuto dugme se kreira pomoću direktivu *mat-raised-button*. Dugme kreira vizuelni efekat senke u pozadini. Primer u nastavku prikazuje deo templejta komponente u kojem se navodi dugme i rezultat za različite palete boja izabrane teme (vrednost *color* atributa taga) i onemogućeno (eng. *disabled*) stanje.

```
<button mat-raised-button>Basic</button>
<button mat-raised-button color="primary">Primary</button>
<button mat-raised-button color="accent">Accent</button>
<button mat-raised-button color="warn">Warn</button>
<button mat-raised-button disabled>Disabled</button>
```



Oivičeno dugme koristi direktivu *mat-stroked-button* i ističe element granicom dugmeta. Primer pokazuje izgled dugmadi za različite palete tema.

```
<button mat-stroked-button>Basic</button>
<button mat-stroked-button color="primary">Primary</button>
<button mat-stroked-button color="accent">Accent</button>
<button mat-stroked-button color="warn">Warn</button>
<button mat-stroked-button disabled>Disabled</button>
```



Ravno ili obično dugme se kreira direktivom *mat-flat-button*, kao u primeru ispod.

```

<button mat-flat-button>Basic</button>
<button mat-flat-button color="primary">Primary</button>
<button mat-flat-button color="accent">Accent</button>
<button mat-flat-button color="warn">Warn</button>
<button mat-flat-button disabled>Disabled</button>

```

Basic

Primary

Accent

Warn

Disabled

Dugme ikona podrazumeva *mat-icon-button* direktivu u okviru taga *button*. Kao sadržaj taga *button* navodi se ime simbola ikone. Ime se smešta kao sadržaj *mat-icon* taga. Naziv simbola je ime objekta koji definiše simbola. Simbol ističe dugme i potrebno ga je pažljivo odabrati tako da značenje simbola asocira na akciju koja se pokreće klikom na dugme. U primerima u nastavku, nazivi simbola odgovaraju onima u GMD kolekciji ikona. Povezivanje aplikacije sa kolekcijom je opisano u narednoj sekciji. Simboli se mogu bojiti korišćenjem odgovarajuće palete boja.

```

<button mat-icon-button>
  <mat-icon>more_vert</mat-icon>
</button>
<button mat-icon-button color="primary">
  <mat-icon>home</mat-icon>
</button>
<button mat-icon-button color="accent">
  <mat-icon>menu</mat-icon>
</button>
<button mat-icon-button>
  <mat-icon>favorite</mat-icon>
</button>
<button mat-icon-button>
  <mat-icon>open_in_new</mat-icon>
</button>

```



Kružno dugme koristi *mat-fab* direktivu, kao i *mat-mini-fab* direktivu za dugme manjih dimenzija. Koristi se kod GKI ekrana osjetljivih na dodir sa

obzirom da oblik dugmeta odgovara obliku vrha prsta korisnika. U zavisnosti od veličine ekrana može se koristiti *mat-fab* varijanta (tablet) ili *mat-mini-fab* varijanta (mobilni telefon). Dugme može tekstualno sa labelom ili imati simbol poput dugmeta ikone, kao u nastavku.

```
<button mat-fab>Basic</button>
<button mat-fab color="primary">Primary</button>
<button mat-fab color="accent">Accent</button>
<button mat-fab color="warn">Warn</button>
<button mat-fab disabled>Disabled</button>
<button mat-fab>
  <mat-icon>favorite</mat-icon>
</button>
```



Za sve varijante dugmeta umesto *button* taga može se koristiti i *a* tag, kao u narednom primeru.

```
<button mat-fab color="accent">
  <mat-icon>home</mat-icon>
</button>
```



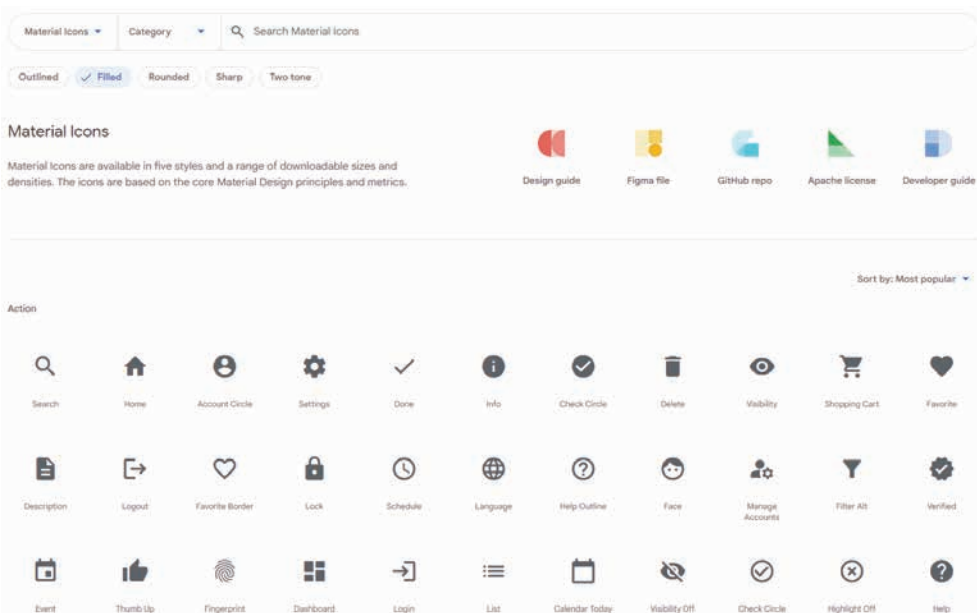
6.6. Ikona

AM ikone (eng. *icons*) su grafički simboli predstavljeni kao vektorski objekti.

```
<mat-icon color='ime_palette_teme'>ime_simbola
</mat icon>
```

Simboli se mogu instalirati lokalno ili se mogu referencirati direktno iz GMD kolekcija ikona [84]. Kolekcija je rezultat prakse dizajna, implementacije i

testiranja GKI. Nazivi simbola su odabrani tako da asociraju na akciju ili objekat koje simbol predstavlja. Samim tim obezbeđuje širok izbor simbola za ikone GKI.



Slika 6.8. GDM kolekcija ikona omogućava pretragu po različitim kriterijumima. Izvor: [84].

Ikone u GKI mogu imati sledeće uloge:

- Dekorativna – pruža korisniku informaciju i nije interaktivna (na primer, legenda).
- Interaktivna – dugme ikona koja pokreće akciju u pozadini.
- Indikator – deo poruke ili statusa izvršenja akcije i obično se navodi uz tekst.

Korišćenje podrazumeva registraciju u odgovarajuće modulu (*material.module.ts*) i navođenje *MatIconModule* u *imports* i *exports* svojstvima *@NgModule* dekoratora.

```
import { MatIconModule } from  
'@angular/material/icon';
```

Interaktivna ikona se navodi podređeni element *button* ili *a* taga. Ikona indikator se često koristi u *span* tagu kao deo teksta poruke.

GMD kolekcija ikona se može koristiti u okviru aplikacije.

Jedan način podrazumeva dobavljanje (eng. *download*) i dodavanje aplikaciji lokalno, kao deo fajl strukture projekta.

Drugi način je direktno korišćenje kolekcije. Za to je potrebno referencirati kolekciju u *index.html* stranici projekta kao u primeru u nastavku. Na ovaj način simboli ikona postaju dostupni svim komponentama aplikacije.

```
<link  
href="https://fonts.googleapis.com/icon?family=Material+Icons" rel="stylesheet">
```

Nakon što se referenciraju, simboli ikona se mogu koristiti kao u primeru u nastavku, tj. navođenjem naziva simbola ikone onako kako je definisano u GMD kolekciji ikona. Ukoliko se GMD kolekcija ikona pretraži po ključnoj reči *home* dobiće se simbol u primeru.

```
<button mat-fab color="accent">  
    <mat-icon>home</mat-icon>  
</button>
```



6.7. Polje za unos

Tekstualno polje za unos (eng. *input*) se kreira pomoću direktive *matInput* u okviru HTML *input* taga. Po pravilu se navodi kao polje forme, tj. u okviru taga *mat-form-field*.

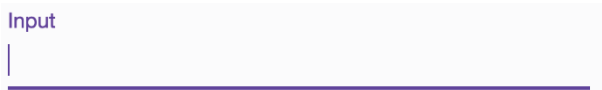
```
<mat-form-field> <input matInput type="text"> </mat-form-field>
```

Komponenta se mora registrovati u odgovarajućem modulu (*material.module.ts*) i njena implementacija (*MatIconModule*) navesti u *imports* i *exports* svojstvima *@NgModule* dekoratora.

```
import { MatInputModule } from  
'@angular/material/input';
```

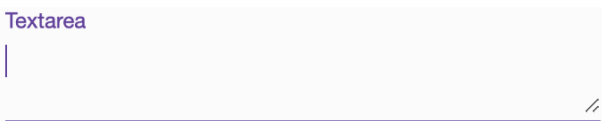

Primer u nastavku pokazuje jednolinijsko polje za unos teksta. Atribut *placeholder* definiše naziv polja koji će se prikazivati korisniku.

```
<mat-form-field>
  <input matInput placeholder="Input">
</mat-form-field>
```



Naredni primer prikazuje višelinijnsko polje za unos teksta. Direktiva *matInput* se navodi u *textarea* tagu, slično kao u HTML.

```
<mat-form-field>
  <textarea matInput placeholder="Textarea"></textarea>
</mat-form-field>
```

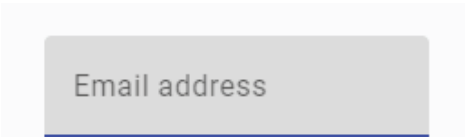


Polje može biti specifičnog formata sadržaja. Format se definiše pomoću *type* atributa *input* taga. Tipovi mogu biti:

- Datum (eng. *date*),
- Elektronska pošta (eng. *email*),
- Lozinka (eng. *password*),
- Tekst (eng. *text*),
- Broj (eng. *number*).

Naredni primer prikazuje polje za unos adrese elektronske pošte. Atribut *type* dobija vrednost *email*.

```
<mat-form-field>
  <input type="email" matInput placeholder="Email address">
</mat-form-field>
```



Polje datuma se kreira tako što se atributu *type* dodeljuje vrednost *date*.

```
<mat-form-field>  
  <input matInput type="date" placeholder="Date">  
</mat-form-field>
```

Date

dd/mm/yyyy

April 2019

Mon	Tue	Wed	Thu	Fri	Sat	Sun
1	2	3	4	5	6	7
8	9	10	11	12	13	14
15	16	17	18	19	20	21
22	23	24	25	26	27	28
29	30	1	2	3	4	5

Naredni primer prikazuje polje lozinke kod kojeg *type* atribut ima vrednost *password*.

```
<mat-form-field>  
  <input matInput type="password" placeholder="Password">  
</mat-form-field>
```

Password

.....

Ukoliko atribut *type* dobije vrednost *text*, u polje se može unositi tekst u slobodnoj formi.

```
<mat-form-field>  
  <input matInput type="text" placeholder="Text">  
</mat-form-field>
```

Text

simple text

6.8. Padajuća lista

AM padajuća lista (eng. *select*) omogućava izbor stavke, tj. vrednosti iz skupa navedenih stavki. Navodi se u sklopu forme. Lista se kreira kao *mat-select* tag, dok se svaka stavka ubacuje kao *mat-option* tag. Stavka ima internu vrednost (atribut *value*) i tekst koji se prikazuje (sadržaj taga). Stavke se mogu unapred selektovati (*selected*) ili onemogućiti (*disabled*).

```
<mat-select placeholder="placeholder_tekst">
<mat-option value="vrednost1" [selected]>Stavka1</mat-option>
<mat-option value="vrednost2" [disabled]>Stavka2</mat-option>
<mat-option value="vrednost3">Stavka3</mat-option>
...
<mat-option value="vrednostN">StavkaN</mat-option>
<mat-select>
```

Komponenta se registruje u odgovarajućem modulu (*material.module.ts*) i njena implementacija (*MatSelectModule*) navodi u *imports* i *exports* svojstvima *@NgModule* dekoratora.

```
import { MatSelectModule } from
'@angular/material/select';
```

Sledeći primer koristi padajuću listu za obostrano povezivanja podataka komponente. Atribut *value* *mat-select* taga ima vrednost trenutno selektovane stavke liste.

Atribut se povezuje sa *selected* atributom kontrolera. Sa obzirom da atribut kontrolera ima vrednost *option1*, stavka liste sa tom vrednošću će biti inicijalno selektovana. Tada se vrednost prosleđuje od kontrolera ka templejtu. Paragraf u nastavku koristi interpolaciju za ispisivanje vrednosti atributa kontrolera. Po izboru stavke liste njena vrednost postaje vrednost atributa kontrolera koji se dalje prikazuje u paragrafu.

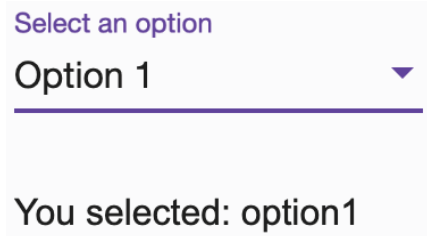
```

<!-- Templejt komponente -->
<mat-form-field>
  <mat-select [(value)]="selected" placeholder="Select an option">
    <mat-option>None</mat-option>
    <mat-option value="option1">Option 1</mat-option>
    <mat-option value="option2">Option 2</mat-option>
  </mat-select>
</mat-form-field>

<p>You selected: {{selected}}</p>

//Kontroler komponente
export class SelectAnOption {
  selected = option1;
}

```



Select an option

Option 1 ▼

You selected: option1

Naredni primer proširuje prethodni tako što kreira stavke liste dinamički, iz kontrolera komponente. Direktiva atributa *ngModel* definiše obostrano povezivanje trenutne vrednosti liste i atributa *selectedCar* kontrolera. Strukturna direktiva **ngFor* kreira stavke liste iz podataka kontrolera. Interfejs *Car* definiše strukturu podataka za listu tako da odgovara vrednosti (*value*) i sadržaju stavke (*viewValue*). Kontroler sadrži niz vrednosti tipa interfejsa (*cars*) koji obezbeđuje podatke za **ngFor* direktivu, tj. popunjavanje liste. Rezultat je u nastavku koda.

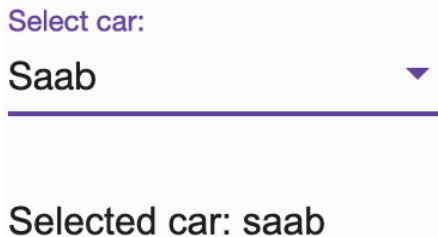
```

<!-- Templejt komponente -->
<mat-form-field>
  <mat-select [(ngModel)]="selectedCar" placeholder="Select car:">
    <mat-option *ngFor="let car of cars" [value]="car.value">
      {{car.viewValue}}
    </mat-option>
  </mat-select>
</mat-form-field>
<p> Selected car: {{selectedCar}} </p>

//Kontroler komponente
interface Car {
  value: string;
  viewValue: string;
}

export class SelectForm {
  selectedCar: string;
  cars: Car[] = [
    {value: 'volvo', viewValue: 'Volvo'},
    {value: 'saab', viewValue: 'Saab'},
    {value: 'mercedes', viewValue: 'Mercedes'}];
}

```



Select car:

Saab ▼

Selected car: saab

6.9. Polje za potvrdu

Polje za potvrdu (eng. *checkbox*) predstavlja AM alternativu HTML elementu (`<input type="checkbox">`) sa unapređenom funkcijom i izgledom.

```
<mat-checkbox>ime_polja</mat-checkbox>
```

Registruje se u odgovarajućem modulu (*material.module.ts*) i implementacija (*MatCheckboxModule*) navodi u *imports* i *exports* svojstvima *@NgModule* dekoratora.

```
import { MatCheckboxModule } from
'@angular/material/checkbox';
```

Komponenta se može nalaziti u jednom od četiri stanja:

1. Označeno (eng. *checked*) polje,
2. Neoznačeno (eng. *unchecked*) polje,
3. Neodređeno (eng. *indeterminate*) polje, i
4. Onemogućeno (eng. *disabled*) polje.

Primer prikazuje označeno polje.

```
<mat-checkbox>Check me!</mat-checkbox>
```



Check me!

Naredni primer ilustruje izgled polja za potvrdu u različitim stanjima.

Svako od prva tri polja je obostrano povezano sa odgovarajućim atributom kontrolera komponente. Vrednosti atributa su inicijalno *false* što znači da će tri polja na početku biti u stanju neoznačeno. Promena vrednosti atributa kontrolera će dovesti svako polje u specifično stanje - prvo u označeno, drugo u neodređeno i treće u onemogućeno.

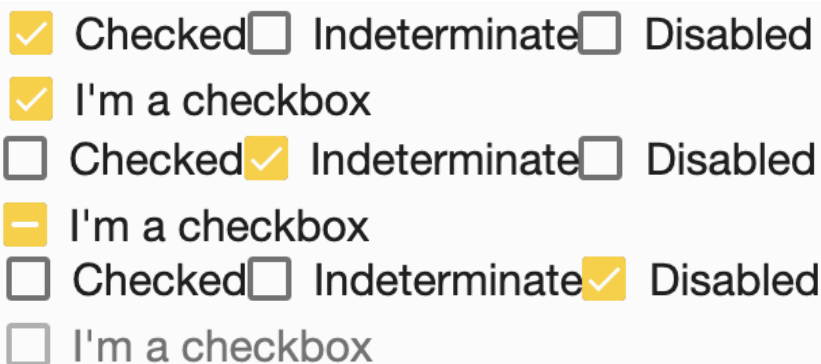
Poslednje polje se obostrano povezuje sa sva tri atributa kontrolera tako da promena stanja polja menja vrednost atributa pridruženog tom stanju.

Na ovaj način će promena stanja poslednjeg polja uzrokovati promenu vrednosti atributa koja se dalje odražava na stanje i izgled odgovarajućeg polja iznad. Rezultat promena je prikazan na slici u nastavku koda.

```
<!-- Templatejt komponente -->
<mat-checkbox [(ngModel)]="checked">Checked</mat-checkbox>
<mat-checkbox [(ngModel)]="indeterminate">Indeterminate</mat-checkbox>
<mat-checkbox [(ngModel)]="disabled">Disabled</mat-checkbox>

<mat-checkbox
  [checked]="checked"
  [indeterminate]="indeterminate"
  [disabled]="disabled">
  I'm a checkbox
</mat-checkbox>

//Kontroler komponente
export class CheckBoxPrimer {
  checked = false;
  indeterminate = false;
  disabled = false;
}
```



6.10. Forma

AM komponenta forma (eng. *form*) je složena kontrola GKI za prihvati i obradu skupa podataka korisnika. Podaci obično predstavljaju logičku celinu, na primer profil korisnika ili opis proizvoda. Funkcije AM forme obuhvataju:

1. Prihvati i validaciju ulaza korisnika u templejtu komponente.
2. Kreiranje modela podataka forme u pozadini, tj. kontroleru komponente.
3. Upravljanje modelom i ažuriranje templejta iz kontrolera.

Forma predstavlja kombinaciju dve komponente - forme kao celine (*FormsModule*) i većeg broja polja forme (*MatFormFieldModule*).

Implementacija polja forme se navodi u modulu za registraciju AM komponenti.

```
//Material.module.ts
import { MatFormFieldModule } from
'@angular/material/form-field';

@NgModule({
  imports: [
    ...
    MatFormFieldModule
  ],
  exports: [
    ...
    MatFormFieldModule
  ],
})
```

Implementacija forme se navodi u korenom modulu aplikacije, kao u nastavku.

```
//app.module.ts
import { FormsModule } from '@angular/forms';

@NgModule({
  ...
  imports: [
    ...
    FormsModule
  ],
```

Naredni primer prikazuje formu sa dva polja, korisničkim imenom i lozinkom. Forma je kreirana kao *form* element koji sadrži dva polja forme (*mat-form-field*). Polje forme ima tekstualno polje za unos (*input*) imena i lozinke korisnika, respektivno.

```
<form>
  <mat-form-field>
    <input type="text" matInput placeholder="Your name">
  </mat-form-field>
  <mat-form-field>
    <input type="password" matInput placeholder="Your password">
  </mat-form-field>
</form>
```

Your name

Your password

U sledećem primeru se pristupa poljima forme iz kontrolera komponente.

U tagu forme se definiše lokalna varijabla tipa objekta forme (*#f = ngForm*) i prosleđuje metodi osluškivača (*onSubmit()*) koja reaguje na događaj slanja podataka forme (*ngSubmit*). Sadržaju polja forme se može pristupiti iz kontrolera. Za ovo je potrebno proglasiti polje delom modela forme (*ngModel*) i dodeliti naziv polju (*name*) kao identifikator.

Kontroler komponente implementira metodu osluškivača koja čita sadržaje polja forme i dodeljuje kao vrednosti varijablama *email* i *password*. Nakon toga ispisuje sadržaj objekta forme (*form*). U nastavku se kod i rezultat izvršavanja metode.


```

<!-- Templejt komponente -->
<form fxLayout="column" fxLayoutGap="10px" fxLayoutAlign="center center"
#f="ngForm" (ngSubmit)="onSubmit(f)">
  <mat-form-field>
    <input type="email" matInput placeholder="Your name"
    ngModel name="email">
  </mat-form-field>
  <mat-form-field>
    <input type="email" matInput placeholder="Your password"
    ngModel ame="password">
  </mat-form-field>
  <button type="submit" mat-raised-button color="primary">Submit</button>
</form>

//Kontroler komponente
onSubmit(form: NgForm) {
  var email = form.value.email;
  var password = form.value.password;
  console.log(form);
}

```

```

name: "email"
path: (...)
pending: (...)
pristine: (...)
status: (...)
statusChanges: (...)
touched: (...)
untouched: (...)
► update: EventEmitter {_isScalar: false }
valid: (...)
validator: (...)
value: "marko.markovic.22@singimail.rs"

```

Validacija ili provera ispravnosti unetih podataka forme se vrši u templejtu komponente. Definiše se lokalna varijabla templejta (*#emailInput*) u kojoj se čuva sadržaj polja za unos adrese elektronske pošte. Poruke o greškama se prikazuju ukoliko se u polje za unos adrese elektronske pošte ne unese sadržaj ili ukoliko se unese sadržaj u neodgovarajućem formatu. *Mat-error* tag ispisuje poruku o grešci. Za svaki slučaj greške se navodi specifična poruka kao tag koji se uslovno prikazuje korišćenjem direktive *ngIf*.

```

<mat-form-field>
  <input type="email" matInput placeholder="Your email"
    ngModel name="email" email required #emailInput="ngModel">
  <mat-error *ngIf="emailInput.hasError('required')">
    Field should not be empty
  </mat-error>
  <mat-error *ngIf="!emailInput.hasError('required')">
    Please use the correct email format.
  </mat-error>
</mat-form-field>

```

Your name *

korisnik

Please use the correct email format.

Your password *

.....

Please enter minimum 6 characters.

11 / 6

Submit

6.11. Kalendar

AM kalendar (eng. *DatePicker*) omogućava unos i rad sa sadržajem tipa datuma. Datum se može uneti direktno ili odabrati iz objekta kalendara. Najčešće se koristi kao deo forme.

Komponenta se registruje u odgovarajućem modulu (*material.module.ts*) i njena implementacija (*MatDatepickerModule*) navodi u *imports* i *exports* svojstvima *@NgModule* dekoratora.

```

import { MatDatepickerModule } from
 '@angular/material/datepicker';

```

Kalendar se sastoji od tri elementa:

1. Tekstualnog polja za unos datuma (*input*),
2. Objekta kalendara (*mat-datepicker*),
3. Ikone koja prikazuje ili sakriva kalendar (*mat-datepicker-toggle*).

Tri elementa se kombinuju na specifičan način opisan u narednom primeru. Tekstualno polje i kalendar se povezuju pomoću *matDatePicker* direktive. Definiše se lokalna varijabla (*picker*) čija je vrednost objekat kalendar (*mat-datepicker*). Varijabla se prosleđuje kao parametar polju za unos i prikaz datuma (*input*) i ikoni kalendara (*mat-datepicker-toggle*). *MatSuffix* direktiva ukazuje da se ikona prikazuje nakon sadržaja datuma (kao sufiks).

```
<mat-form-field>
  <input
    matInput
    placeholder="Your date of birth."
    [matDatePicker]="picker"
    ngModel
    name="birthDate"
    required
  >
  <mat-datepicker-toggle matSuffix [for]="picker"></mat-datepicker-toggle>
  <mat-datepicker #picker></mat-datepicker>
</mat-form-field>
```

Your name *

korisnik@singimail.rs

Your password *

.....

Please enter minimum 6 characters.

8 / 6

Your date of birth. *

12/10/2022



Submit

6.12. Vertikalna linija alata

Komponenta predstavlja vertikalni kontejner elemenata. Alati su često korišćene funkcije aplikacije u vidu dugmadi ili ikona. Najčešće su to opcije za navigaciju te otuda i naziv *sidenav*. Može se koristiti kao deo GKI smešten uz levu stranu prikaza koji se prikazuje po potrebi. Primer je tzv. klizeći meni (eng. *sliding menu*) kod interfejsa mobilnog uređaja.

Kombinuje se sa horizontalnom linijom (opisana u narednoj sekciji) tako da se izgled GKI dinamički prilagođava dimenzijama ekrana. U slučaju manje širine

prikazuje se vertikalna linija, dok se kod veće širine prikazuje horizontalna linija. Tehnika je detaljnije opisana u Sekciji 8.4.

Komponenta se registruje u odgovarajućem modulu (*material.module.ts*) i implementacija (*MatSidenavModule*) navodi u *imports* i *exports* svojstvima *@NgModule* dekoratora.

```
import { MatSidenavModule } from
 '@angular/material/sidenav';
```

Vertikalna linija alata se sastoji iz sledećih elemenata:

1. Koreni element komponente (*mat-sidenav-container*),
2. Zaglavlje koje kontroliše vidljivost sadržaja (*mat-sidenav*),
3. Sadržaj koji se prikazuje (*mat-sidenav-content*).

Primer u nastavku ilustruje strukturu komponente.

Vrednost *true* atributa *opened* ukazuje da je vertikalna linija alata automatski vidljiva po učitavanju i prikazivanju komponente. Definisana je i lokalna varijabla *#sidenav* koja ukazuje na element zaglavlja. Sadržaj može biti i komponenta do koje vodi trenutna ruta (*router-outlet* tag). Korišćenje varijable je opisano u Sekciji 8.4.1.

```
<mat-sidenav-container>
  <mat-sidenav #sidenav opened="true">
    <p>sidenavigacija</p>
  </mat-sidenav>
  <mat-sidenav-content>
    <router-outlet></router-outlet>
  </mat-sidenav-content>
</mat-sidenav-container>
```

6.13. Horizontalna linija alata

AM komponenta horizontalna linija alata ima sličnu namenu kao i vertikalna linija alata. Kreira se pomoću *mat-toolbar* taga u koji se smešta željeni sadržaj. Registruje u odgovarajućem modulu (*material.module.ts*) i njena implementacija (*MatToolbarModule*) navodi u *imports* i *exports* svojstvima *@NgModule* dekoratora.

```
import { MatToolbarModule } from
 '@angular/material/toolbar';
```

Kombinovanje sa vertikalnom linijom alata je prikazano u narednom primeru. Horizontalna linija alata se dodaje kao element sadržaja vertikalne linije alata (*mat-sidenav-content* tag). Kompletan primer sa objašnjenjima se nalazi u Sekciji 8.4.2.

```
<mat-sidenav-container>
  <mat-sidenav #sidenav opened="true">
    <p>sidenavigacija</p>
  </mat-sidenav>
  <mat-sidenav-content>
    <mat-toolbar>
      <div>
        <button mat-button (click)="sidenav.toggle()">
          Toggle sidenav
        </button>
      </div>
      <div>Logo</div>
      <div>
        <a routerLink="/signup">Signup</a>
        <a routerLink="/login">Login</a>
        <a routerLink="/training">Training</a>
      </div>
    </mat-toolbar>
    <router-outlet></router-outlet>
  </mat-sidenav-content>
</mat-sidenav-container>
```

6.14. Lista

AM komponenta lista koristi se kao element nabiranja, slično HTML *list* tagu. Za razliku od padajuće liste, sve stavke date liste su vidljive.

AM komponentu je potrebno registrovati u modulu *material.module.ts* i potrebno je navesti implementaciju (*MatListModule*) u *imports* i *exports* svojstvima *@NgModule* dekoratora.

```
import { MatListModule } from
 '@angular/material/list';
```

Postoji više varijanti liste gde svaka ima specifičnu namenu.

Kod obične liste, u okviru *mat-list* taga svaka stavka se navodi kao *mat-list-item* tag.

```
<mat-list>
  <mat-list-item>Item 1</mat-list-item>
  <mat-list-item>Item 2</mat-list-item>
  <mat-list-item>Item 2</mat-list-item>
</mat-list>
```

Item 1

Item 2

Item 3

Navigaciona lista (*mat-nav-list*) definiše elemente za navigaciju, tj. kretanje kroz GKI. U ovom slučaju stavka se navodi kao *a* tag sa *mat-list-item* direktivom.

```
<mat-sidenav #sidenav opened="true">
  <mat-nav-list>
    <a mat-list-item routerLink="/signup">Signup</a>
    <a mat-list-item routerLink="/login">Login</a>
    <a mat-list-item routerLink="/training">Training</a>
    <a mat-list-item routerLink="/">Logout</a>
  </mat-nav-list>
</mat-sidenav>
```

Akciona lista (*mat-action-list*) grupiše stavke koje predstavljaju funkcije aplikacije. Ovde se stavka navodi kao *button* tag sa *mat-list-item* direktivom.

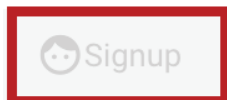
```
<mat-action-list>
  <button mat-list-item (click)="save()">Save</button>
  <button mat-list-item (click)="undo()">Undo</button>
</mat-action-list>
```

Stavke navigacione liste mogu sadržati ikone (*mat-icon*) i tekst.

```

<mat-nav-list>
  <a mat-list-item routerLink="/signup">
    <mat-icon>person</mat-icon>
    Signup
  </a>
  <a mat-list-item routerLink="/login">
    <mat-icon>login</mat-icon>
    Login
  </a>
  <a mat-list-item routerLink="/training">
    <mat-icon>fitness_center</mat-icon>
    Training
  </a>
  <a mat-list-item routerLink="/">
    <mat-icon>logout</mat-icon>
    Logout
  </a>
</mat-nav-list>

```



⇒ Login

↗ Training

⇒ Logout

6.15. Višestruki pogledi

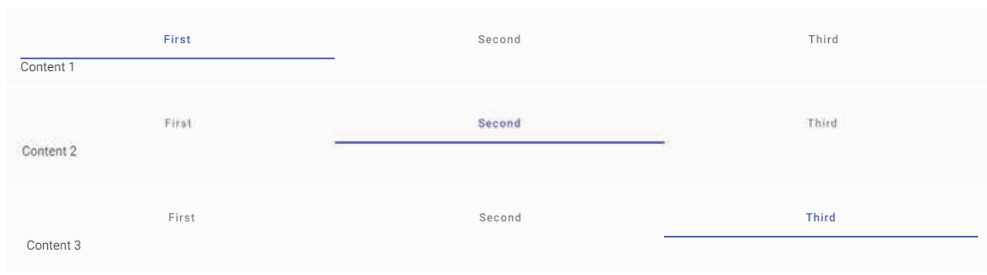
Komponenta višestruki pogledi (eng. *tab*) struktura odvojene delove sadržaja tako da se u datom trenutku vidi samo jedan deo. Deo koji se želi videti se bira pomoću jezička (*tab*) na vrhu. Element podseća na fizičke fascikle u kojima se sadržaj uređuje na sličan način.

Registruje se u odgovarajućem modulu (*material.module.ts*) i implementacija (*MatTabsModule*) navodi u *imports* i *exports* svojstvima *@NgModule* dekoratora.

```
import { MatTabsModule } from
 '@angular/material/tabs';
```

Mat-tab-group je kontejnerski tag, dok se odvojeni pogledi smeštaju u okviru *mat-tab* taga. Sadržaj *mat-tab* taga je proizvoljan. Svaki pogled je opisan imenom ili labelom (*label* atribut). Primer u nastavku ilustruje višestruke poglede sa pojednostavljenim sadržajem.

```
<mat-tab-group>
  <mat-tab label="First"> Content 1 </mat-tab>
  <mat-tab label="Second"> Content 2 </mat-tab>
  <mat-tab label="Third"> Content 3 </mat-tab>
</mat-tab-group>
```



Sadržaj pogleda može biti i komponenta. U sledećem primeru selektori komponenti su navedeni u okviru tagova pogleda.

```
<mat-tab-group>
  <mat-tab label="Current training">
    <app-current-training></app-current-training>
  </mat-tab>
  <mat-tab label="New training">
    <app-new-training></app-new-training>
  </mat-tab>
  <mat-tab label="Past training">
    <app-previous-training></app-previous-training>
  </mat-tab>
</mat-tab-group>
```

Višestruki pogledi su pogodni za sadržaje bogate informacijama koji se ne mogu prikazati odjednom i gde postoje jednostavniji delovi koje je moguće nezavisno pregledati i koristiti. U gornjem primeru, sadržaj je fizički trening,

dok su delovi prošli trening (istorijat), tekući trening (aktivnosti u toku) i budući trening (plan vežbi).

6.16. Kartica

Komponenta kartica (eng. *card*) organizuje sadržaj po odeljcima kao zaglavlje, sadržaj, akcije i podnožje. Sadržaj je organizovan po principu dokumenta i kartica je pogodna za prikaz instanci entiteta aplikacije (na primer, korisnik, proizvod, porudžbina i slično).

Registruje se u odgovarajućem modulu (*material.module.ts*) i implementacija (*MatCardModule*) navodi u *imports* i *exports* svojstvima *@NgModule* dekoratora.

```
import { MatCardModule } from  
'@angular/material/card';
```

U skladu sa namenom, kartica prikazuje sadržaj po sledećim odeljcima:

- Kartica kao grupa (eng. *mat-card*),
- Naslov kartice (eng. *mat-card-title*),
- Glavni sadržaj kartice (eng. *mat-card-content*),
- Akcije nad sadržajem (eng. *mat-card-actions*), i
- Podnožje kartice (eng. *mat-card-footer*) za dodatne informacije.

Primer prikazuje strukturu kartice sa jednostavnim sadržajem odeljaka. U ovom slučaju podnožje nije navedeno.

```

<section>
  <mat-card>
    <!-- Naslov kartice -->
    <mat-card-title>
      Start training
    </mat-card-title>
    <!-- Sadržaj kartice -->
    <mat-card-content>
      Choose training:
    </mat-card-content>
    <!-- Akcije kartice -->
    <mat-card-actions>
      <button type="submit" mat-raised-button color="primary">
        Start
      </button>
    </mat-card-actions>
  </mat-card>
</section>

```

Naredni primer proširuje odeljak glavnog sadržaja kartice u odnosu na prethodni. Dodaje se padajuća lista kao polje forme (*mat-select*) sa tri stavke za različite vrste fizičke aktivnosti. Izgled kartice je prikazan u nastavku.

```

<mat-card-content>
  <mat-form-field>
    <mat-select placeholder="Choose training:">
      <!-- Definisanje opcija -->
      <mat-option value="walking">
        Walking
      </mat-option>
      <mat-option value="running">
        Running
      </mat-option>
      <mat-option value="squats">
        Squats
      </mat-option>
    </mat-select>
  </mat-form-field>
</mat-card-content>

```



6.17. Tabela

AM tabela (eng. *table*) omogućava tabelarni pregled i upravljanje podacima. Koriste se za prikaz i rukovanje složenijim entitetima aplikacije, gde kolone odgovaraju atributima objekta dok redovi definišu instance objekata kao skupove vrednosti atributa.

Tabela je podržana funkcionalnostima poput sortiranja, straničenja i filtriranja podataka. Ove funkcionalnosti nisu deo tabele po definiciji i dodaju se kao zasebni elementi.

Na ovaj način se tabela može prilagoditi promenama zahteva korisnika u vreme korišćenja GKI u cilju optimizacije kvaliteta upotrebe i performansi aplikacije. Ukoliko korisnik radi sa manjom količinom podataka koja se može prikazati na jednoj strani tabele, ne postoji potreba za dodatnim funkcionalnostima. Ovo može unaprediti performanse aplikacije sa obzirom da moduli koji implementiraju funkcionalnosti neće biti učitani i izvršavani u memoriji Veb pretraživača zajedno sa funkcionalnostima. U slučaju veće količine podataka, funkcionalnosti će biti deo izvršavanja programa.

Tabelu je potrebno registrovati u odgovarajućem modulu (*material.module.ts*) i implementaciju (*MatTableModule*) navesti u *imports* i *exports* svojstvima *@NgModule* dekoratora.

```
import { MatTableModule } from
'@angular/material/table';
```

6.17.1. Kreiranje tabele

Obrazac za definisanje tabele je sledeći:

- Tabela (eng. *mat-table*) u okviru koje se navodi izvor podataka (eng. *dataSource*).
- Kolone se definišu kao zaglavlja ćelija kolone (eng. *mat-header-cell*) i vrednosti ćelija kolone (eng. *mat-cell*).

- Redovi se sastoje iz reda sa zaglavljima kolona (eng. *mat-header-row*) i redova sa vrednostima podataka, odnosno vrednostima ćelija kolona (eng. *mat-row*).

Primer u nastavku ilustruje kreiranje tabele u templejtu komponente.

Tabela opisuje periodični sistem elemenata. U tagu tabele atribut izvora podataka (*dataSource*) se povezuje sa istoimenim atributom kontrolera (povezivanje atributa).

Zatim se definišu kolone tabele. Svaka kolona počinje definicijom kolone (*matColumnDef*) u okviru koje se dodaje ćelija zaglavlja (*mat-header-cell*) pomoću strukturne direktive **matHeaderCellDef*. Zatim se navodi ćelija podataka (*mat-cell*) pozivom direktive **matCellDef*. Direktiva prikazuje podatak u ćeliji. Podaci koji se prikazuju su povezani sa izvorom navedenim u korenom elementu tabele i vrednost ćelije se ispisuje korišćenjem interpolacije. *Ng-container* je specijalan tag koji sadrži strukturne direktive bez eksplicitnog navođenja novih elemenata u dizajn stranice. Drugim rečima, sadržaj ovog taga se generiše dinamički u vreme izvršavanja aplikacije. Ovaj mehanizam je značajan za performanse aplikacije. Na primer, ukoliko podaci za određene kolone ne postoje, kolone neće postojati u DOM stablu aplikacije. Na kraju se kreiraju redovi tako što se najpre navodi prvi red zaglavlja tabele (*mat-header-row*) pomoću direktive **matHeaderRowDef*. Nazivi zaglavlja se dobavljaju iz varijable kontrolera tipa niz (*displayedColumns*). Nakon toga se generišu redovi podataka tabele (*mat-row*) korišćenjem direktive **matRowDef*.

```

<mat-table [dataSource]="dataSource">

  <ng-container matColumnDef="position">
    <mat-header-cell *matHeaderCellDef> No. </mat-header-cell>
    <mat-cell *matCellDef="let element"> {{element.position}} </mat-cell>
  </ng-container>

  <ng-container matColumnDef="name">
    <mat-header-cell *matHeaderCellDef> Name </mat-header-cell>
    <mat-cell *matCellDef="let element"> {{element.name}} </mat-cell>
  </ng-container>

  <ng-container matColumnDef="weight">
    <mat-header-cell *matHeaderCellDef> Weight </mat-header-cell>
    <mat-cell *matCellDef="let element"> {{element.weight}} </mat-cell>
  </ng-container>

  <ng-container matColumnDef="symbol">
    <mat-header-cell *matHeaderCellDef> Symbol </mat-header-cell>
    <mat-cell *matCellDef="let element"> {{element.symbol}} </mat-cell>
  </ng-container>

  <mat-header-row *matHeaderRowDef="displayedColumns"></mat-header-row>
  <mat-row *matRowDef="let row; columns: displayedColumns;"></mat-row>

</mat-table>

```

Kontroler komponente obezbeđuje podatke za prikaz u tabeli. Podaci su tipa interfejsa *PeriodicElement* i definisani su kao niz predefinisanih vrednosti koji se kao vrednost dodeljuje atributu kontrolera *dataSource*. Atribut je povezan sa istoimenim atributom izvora podataka tabele. Nakon povezivanja, svojstvima podataka interfejsa se pristupa po nazivu kod definisanja ćelija podataka tabele. Pored toga, definisan je i atribut čija vrednost je niz sa nazivima kolona tabele (*displayedColumns*). Ovaj atribut koristi direktiva koja kreira prvi red zaglavlja tabele. Slika u nastavku daje izgled tabele.

```

export interface PeriodicElement {
  name: string;
  position: number;
  weight: number;
  symbol: string;
}

const ELEMENT_DATA: PeriodicElement[] = [
  { position: 1, name: 'Hydrogen', weight: 1.0079, symbol: 'H' },
  { position: 2, name: 'Helium', weight: 4.0026, symbol: 'He' },
  { position: 3, name: 'Lithium', weight: 6.941, symbol: 'Li' },
  { position: 4, name: 'Beryllium', weight: 9.0122, symbol: 'Be' },
  { position: 5, name: 'Boron', weight: 10.811, symbol: 'B' }
];

export class TableBasicExample {
  displayedColumns: string[] = ['position', 'name', 'weight', 'symbol'];
  dataSource = ELEMENT_DATA;
}

```

No.	Name	Weight	Symbol
1	Hydrogen	1.0079	H
2	Helium	4.0026	He
3	Lithium	6.941	Li
4	Beryllium	9.0122	Be
5	Boron	10.811	B

Tabela u sledećem primeru se kreira na sličan način. Razlika u odnosu na prethodni primer je sadržaj, odnosno podaci koje tabela prikazuje. Isti primer je detaljnije opisan u studiji slučaja u Sekciji 8.7.

```

<mat-table [dataSource]="exerciseSource">
  <ng-container matColumnDef="date">
    <mat-header-cell *matHeaderCellDef>Date</mat-header-cell>
    <mat-cell *matCellDef="let element">{{ element.date | date }}</mat-cell>
  </ng-container>

  <ng-container matColumnDef="name">
    <mat-header-cell *matHeaderCellDef>Name</mat-header-cell>
    <mat-cell *matCellDef="let element">{{ element.name }}</mat-cell>
  </ng-container>

  <ng-container matColumnDef="duration">
    <mat-header-cell *matHeaderCellDef>
      Duration
    </mat-header-cell>
    <mat-cell *matCellDef="let element">
      {{ element.duration | number }}
    </mat-cell>
  </ng-container>

  <ng-container matColumnDef="calories">
    <mat-header-cell *matHeaderCellDef>
      Calories
    </mat-header-cell>
    <mat-cell *matCellDef="let element">
      {{ element.calories | number }}
    </mat-cell>
  </ng-container>

  <ng-container matColumnDef="status">
    <mat-header-cell *matHeaderCellDef>
      Status
    </mat-header-cell>
    <mat-cell *matCellDef="let element">
      {{ element.status }}
    </mat-cell>
  </ng-container>

  <mat-header-row *matHeaderRowDef="displayedColumns"></mat-header-row>
  <mat-row *matRowDef="let rows; columns: displayedColumns"></mat-row>
</mat-table>

```

Kontroler komponente podatke dobija pomoću servisa koji se navodi kao argument konstruktora. Kontroler sadrži dva atributa. Prvi definiše nazive kolona tabele kao niz (*displayedColumns*). Drugi atribut kreira izvor podataka za tabelu kao instancu generičke klase *MatTableDataSource* koja prihvata argument tipa kolekcije. U ovom slučaju je to predefinisani niz objekata tipa interfejsa *Exercise*. U toku inicijalizacije komponente atribut *date* kreirane instance dobija vrednost pozivom odgovarajuće metode servisa. Preporuka je da komponente koje prikazuju tabelu implementiraju interfejs *AfterViewInit* kako bi podaci koje komponenta treba da prikaže bili dobavljeni u vreme inicijalizacije i prikaza komponente.

```

import { Component, OnInit } from '@angular/core';
import { MatTableDataSource } from '@angular/material/table';
import { Exercise } from '../exercise.model';
import { TrainingService } from '../training.service';

@Component({
  selector: 'app-previous-training',
  templateUrl: './previous-training.component.html',
  styleUrls: ['./previous-training.component.css']
})
export class PreviousTrainingComponent implements OnInit, AfterViewInit {

  displayedColumns = ["date", "name", "duration", "calories", "status"];
  exerciseSource = new MatTableDataSource<Exercise>();

  constructor(private trainingService : TrainingService) { }

  ngOnInit() {
    this.exerciseSource.data = this.trainingService.getExercises();
  }
}

```

Interfejs *Exercise* definiše strukturu podataka koje tabela treba da prikaže.

```

export interface Exercise {
  id: string;
  name: string;
  duration: number;
  calories: number;
  date?: Date;
  status?: 'done' | 'canceled' | null;
}

```

Servis koji obezbeđuje podatke za popunjavanje tabele sadrži atribut čija je vrednost niz predefinisanih instanci tipa interfejsa *Exercise* (*exercise*) i metodu koja vraća vrednost atributa (*getExercises()*).


```
import { Exercise } from './exercise.model';

export class TrainingService {
  private exercises: Exercise[] = [
    {
      id: 'running', name: 'Morning running',
      duration: 15, calories: 100, date: new Date(), status: 'canceled',
    },
    {
      id: 'squats', name: 'Midday squats',
      duration: 5, calories: 200, date: new Date(), status: 'done',
    },
    {
      id: 'walking', name: 'Afternoon walking',
      duration: 20, calories: 50, date: new Date(), status: 'done',
    },
    {
      id: 'running', name: 'Evening running',
      duration: 15, calories: 300, date: new Date(), status: 'canceled',
    },
  ];
  getExercises(): Exercise[] {
    return this.exercises;
  }
}
```

Rezultat je tabela kao na slici u nastavku.

New training		Past training
Date	Name	Duration
Dec 2, 2018	Morning running	15
Dec 2, 2018	Midday squats	5
Dec 2, 2018	Afternoon walking	20
Dec 2, 2018	Evening running	15

Slika 6.9. Izgled AM tabele.

6.17.2. Sortiranje tabele

Funkcija sortiranja tabele je implementirana i dodaje se kao odvojena komponenta, odnosno modul.

Sortiranje kao modul je potrebno registrovati u odgovarajućem modulu (*material.module.ts*) i implementaciju (*MatSortModule*) navesti u *imports* i *exports* svojstvima *@NgModule* dekoratora.

```
import { MatTableModule } from
 '@angular/material/sort';
```

Funkcija sortiranja se dodaje postojećoj tabeli u templejtu i kontroleru komponente.

U tabelu u templejtu komponente se dodaju dve direktiva za sortiranje prikaza:

- *matSort* direktiva se navodi korenom elemente table (*mat-table*), i
- *mat-sort-header* direktiva se smešta u zaglavlje kolone table (*mat-header-cell*).

Direktiva *matSort* definiše tabelu čije se kolone mogu sortirati. Direktiva *mat-sort-header* označava da je kolona u čijem se zaglavlju navodi sortirajuća. Na ovaj način se može sortirati po svim ili izabranim kolonama.

Primeri templejta i kontrolera komponente u nastavku proširuju tabelu iz prethodne sekcije.

MatSort direktiva se dodaje tabeli u templejtu komponente, dok se u zaglavlje svake kolone koja se želi sortirati ubacuje direktiva *mat-sort-header*.

```
<mat-table [dataSource]="exerciseSource" matSort>
  <ng-container matColumnDef="date">
    <mat-header-cell *matHeaderCellDef mat-sort-header>Date</mat-header-cell>
    <mat-cell *matCellDef="let element">{{ element.date | date }}</mat-cell>
  </ng-container>
  <ng-container matColumnDef="name">
    <mat-header-cell *matHeaderCellDef mat-sort-header>Name</mat-header-cell>
    <mat-cell *matCellDef="let element">{{ element.name }}</mat-cell>
  </ng-container>
  <!-- ... -->
  <mat-header-row *matHeaderRowDef="displayedColumns"></mat-header-row>
  <mat-row *matRowDef="let rows; columns: displayedColumns"></mat-row>
</mat-table>
```

Potrebno je još izvršiti izmene u kontroleru komponente kroz sledeće korake:

1. Definisanje objekta tipa *MatSort* za sortiranje podataka table.
2. Dodavanje funkcije sortiranja izvoru podataka table.
3. Implementacija interfejsa *AfterViewInit* kako bi se tabela mogla sortirati nakon što podaci budu učitanu u komponentu.

Prva dva koraka su neophodna, dok je treći korak preporuka u slučaju da tabela može prikazivati veće količine podataka.

U skladu sa koracima, kontroler komponente se modifikuje kao u nastavku. Objekat tipa *MatSort* se kreira i dodeljuje *sort* svojstvu izvora podataka *exerciseDataSource*.

Dekorator *@ViewChild* detektuje promene prvog elementa ili direktive u DOM stablu čiji selektor odgovara elementu kontrolera uz koji se navodi, i ažurira dati element. U primeru ispod element uz koji se navodi je *MatSort* instanca *sort*, dok je element čije se promene prate *matSort* direktiva u tabeli u templejtu.

Na ovaj način se sortiranje tabele detektuje u kontroleru, vrše odgovarajuće promene u podacima, i prikazuju izvršene promene.

Drugim rečima, kada korisnik sortira tabelu promena u templejtu se detektuje i u izvoru podataka u kontroleru, redosled prikaza podataka ažurira, i osvežava prikaz tabele.

```
//Kontroler komponente
import { Component, OnInit, ViewChild, AfterViewInit } from '@angular/core';
import { MatSort } from '@angular/material/sort';
import { MatTableDataSource } from '@angular/material/table';
import { Exercise } from '../exercise.model';
import { TrainingService } from '../training.service';

@Component({
  selector: 'app-previous-training',
  templateUrl: './previous-training.component.html',
  styleUrls: ['./previous-training.component.css']
})
export class PreviousTrainingComponent implements OnInit, AfterViewInit {

  displayedColumns = ["date", "name", "duration", "calories", "status"];
  exerciseSource = new MatTableDataSource<Exercise>();

  @ViewChild(MatSort) sort: MatSort;

  constructor(private trainingService: TrainingService) { }

  ngOnInit(){
    this.exerciseSource.data = this.trainingService.getExercises();
  }

  ngAfterViewInit(){
    this.exerciseSource.sort = this.sort;
  }
}
```

Izgled tabele koja se može sortirati je prikazan na slici ispod.

Past training		
Name	Duration	Calories ↓
Morning running	15	100
Evening running	15	100
Afternoon walking	20	80
Midday squats	5	70

Slika 6.10. AM tabela sa funkcijom sortiranja.

6.17.3. Straničenje tabele

Ukoliko tabela ne može prikazati sve podatke u jednom pogledu, onda je potrebno podatke prikazivati postupno ili po stranicama koje sadrže određen broj redova.

Funkcija straničenja tabele je implementirana i dodaje se kao odvojena komponenta, odnosno modul.

Straničenje kao modul je potrebno registrovati u odgovarajućem modulu (*material.module.ts*) i implementaciju (*MatPaginatorModule*) navesti u *imports* i *exports* svojstvima *@NgModule* dekoratora.

```
import { MatPaginatorModule } from
 '@angular/material/paginator';
```

Funkcija straničenja se dodaje postojećoj tabeli u templejtu i kontroleru komponente.

U tabelu u templejtu komponente se dodaju sledeći elementi straničenja:

- *Mat-paginator* tag kao dodatak u nastavku tabele.
- *PageSize* kao direktiva atributa taga čija je vrednost broj redova u stranici.
- *PageSizeOptions* kao direktiva atributa taga čije je vrednost niz za definisanje padajuće liste sa opcijama za promenu veličine stranice.

Primeri templejta i kontrolera komponente u nastavku proširuju tabelu iz prethodne sekcije.

Mat-paginator tag se dodaje nakon definicije tabele, sa dve direktive atributa kao u primeru ispod.

```

<mat-table [dataSource]="exerciseSource" matSort>

  <ng-container matColumnDef="date">
    <mat-header-cell *matHeaderCellDef mat-sort-header>Date</mat-header-cell>
    <mat-cell *matCellDef="let element">{{ element.date | date }}</mat-cell>
  </ng-container>

  <ng-container matColumnDef="name">
    <mat-header-cell *matHeaderCellDef mat-sort-header>Name</mat-header-cell>
    <mat-cell *matCellDef="let element">{{ element.name }}</mat-cell>
  </ng-container>

  <!-- ... -->

  <mat-header-row *matHeaderRowDef="displayedColumns"></mat-header-row>
  <mat-row *matRowDef="let rows; columns: displayedColumns"></mat-row>
</mat-table>

<mat-paginator
[pageSize]="1"
[pageSizeOptions]="[1, 2, 4, 6, 10]">
</mat-paginator>

```

Potrebno je još izvršiti izmene u kontroleru komponente kroz sledeće korake:

4. Definisanje objekta tipa *MatPaginator* za straničenje podataka tabele.
5. Dodavanje funkcije straničenja izvoru podataka tabele.
6. Implementacija interfejsa *AfterViewInit* kako bi se tabela mogla straničiti nakon što podaci budu učitani u komponentu.

Prva dva koraka su neophodna, dok se treći korak preporučuje kada tabela može prikazivati veće količine podataka.

U skladu sa koracima, kontroler komponente se modifikuje kao u nastavku.

Objekat tipa *MatPaginator* se kreira i dodeljuje *paginator* svojstvu izvora podataka *exerciseDataSource*.

Dekorator *@ViewChild* detektuje promene prvog elementa ili direktive u DOM stablu čiji selektor odgovara elementu kontrolera uz koji se navodi, i ažurira dati element. U primeru ispod element uz koji se navodi je *MatPaginator* instanca *paginator*, dok je element čije se promene prate *matPaginator* tag pridružen tabeli u templejtu.

Ovim se straničenje tabele detektuje u kontroleru, pripremaju nove stranice podataka za prikaz, i prikazuju u tabeli.

Drugim rečima, kada se korisnik kreće kroz stranice tabele promena u templejtu se detektuje i u izvoru podataka u kontroleru, podaci za naredne stranice pripremaju, i osvežava prikaz tabele.

```

import { Component, OnInit, ViewChild, AfterViewInit } from '@angular/core';
import { MatSort } from '@angular/material/sort';
import { MatPaginator } from '@angular/material/paginator';
import { MatTableDataSource } from '@angular/material/table';
import { Exercise } from '../exercise.model';
import { TrainingService } from '../training.service';

@Component({
  selector: 'app-previous-training',
  templateUrl: './previous-training.component.html',
  styleUrls: ['./previous-training.component.css']
})
export class PreviousTrainingComponent implements OnInit, AfterViewInit {

  displayedColumns = ["date", "name", "duration", "calories", "status"];
  exerciseSource = new MatTableDataSource<Exercise>();

  @ViewChild(MatSort) sort: MatSort;
  @ViewChild(MatPaginator) paginator: MatPaginator;

  constructor(private trainingService: TrainingService) { }

  ngOnInit(){
    this.exerciseSource.data = this.trainingService.getExercises();
  }

  ngAfterViewInit(){
    this.exerciseSource.sort = this.sort;
    this.exerciseSource.paginator = this.paginator;
  }
}

```

Tabela sa stranicenjem je prikazana na slici u nastavku.

Duration	Calories	Status
15	100	done
5	70	done

Items per page: 2 1 - 2 of 4 < >

Slika 6.11. AM tabela sa funkcijom stranicenja.

6.17.4. Filtriranje tabele

Stranicenje omogućava ručnu pretragu redova tabele. Tabela se može pretraživati i na osnovu vrednosti specifičnog polja podataka, odnosno vrednosti ćelije. Ovakva vrsta pretrage se naziva filtriranje ili filter podataka tabele.

Filter ne postoji kao zasebna AM komponenta. Filtriranje tabele se definiše tako što se dodaje tekstualno polje za unos forme u kojem korisnik zadaje kriterijum za pretragu tabele. Polje se obično navodi pre definicije tabele.

U templejt komponente, filter se dodaje kao polje za unos u nastavku. *MatInput* direktiva tipa *text* označava tekstualno polje za unos teksta u slobodnoj formi. Na događaj otpuštanja tastera tastature (*keyup*) poziva se funkcija osluškivača koja filtrira podatke tabele (*doFilter()*). Argument metode osluškivača predstavlja sadržaj tekstualnog polja kao izvora događaja - *\$any(\$event.target).value*. Eksplicitna konverzija (eng. *cast*) u tip *any* se dodaje u cilju pretrage ćelija tabele sa različitim tipovima vrednosti na osnovu sadržaja koji korisnik unese u polje za pretragu.

```
<mat-form-field>
  <input
    matInput
    type="text"
    (keyup)="doFilter($any($event.target).value)"
    placeholder="Filter">
</mat-form-field>
```

Metoda osluškivača u kontroleru komponente je implementirana kao u nastavku. Izvor podataka tabele ima *filter* svojstvo koje po definiciji filtrira podatke tabele na osnovu zadate vrednosti-kriterijuma. Filtrirani podaci se automatski prikazuju u tabeli. Vrednost je sadržaj koji je korisnik uneo u tekstualno polje, a koji se u templejtu iznad prosleđuje kao parametar metodi osluškivača. *Filter* svojstvo vrši pretragu podataka redova kao vrednosti tipa *String*. Radi brže i preciznije pretrage, vrednost se najpre parsira tako da se uklanjaju specijalni znaci na početku i kraju koje korisnik može slučajno uneti (*trim()*). Zatim se rezultujući string transformiše u mala slova ili znakove (*toLowerCase()*). Izgled filtera je kao na slici u nastavku, za kriterijum pretrage *Morning running*.

```
doFilter(filterValue: string){
  this.exerciseSource.filter = filterValue.trim().toLowerCase();
}
```

Date	Name	Duration	Calories	Status
Jan 30, 2023	Morning running	20	100	canceled

6.18. Dijalog

AM dijalog (eng. *dialog*) omogućava specifičnu interakciju sa korisnikom kao izdvojen pogled koji se pojavljuje u prvom planu u odnosu na pozadinu interfejsa.

Dijalog kao modul je potrebno registrovati u odgovarajućem modulu (*material.module.ts*) i implementaciju (*MatDialogModule*) navesti u *imports* i *exports* svojstvima *@NgModule* dekoratora.

```
import { MatDialogModule } from
'@angular/material/dialog';
```

Struktura dijaloga podseća na karticu (videti Sekciju 6.16.) i sadrži sledeće elemente:

- Naslov dijaloga (eng. *mat-dialog-title*) koji se dodaje kao direktiva HTML tagovima zaglavlja (*h1*, *h2*, *h3* i drugi).
- Glavni sadržaj dijaloga (eng. *mat-dialog-content*) kao tag, i
- Akcije dijaloga (eng. *mat-dialog-actions*) kao tag.

Dijalog se dodaje u templejt i kontroler komponente. Direktiva *mat-dialog-title* se navodi u zaglavlju prvog nivoa (*h1*). Sadržaj (*mat-dialog-content*) je paragraf teksta. Kao akcije (*mat-dialog-actions*) se navode dva AM dugmeta. Vrednost direktive atributa [*mat-dialog-close*] govori da li se na klik dugmeta dijalog zatvara (*false*) ili ostaje otvoren (*true*).


```

<section>
  <h1 mat-dialog-title>Stop training</h1>
  <mat-dialog-content>
    <p>Are you sure?</p>
  </mat-dialog-content>

  <mat-dialog-actions>
    <button mat-button [mat-dialog-close]="true">Yes</button>
    <button mat-button [mat-dialog-close]="false">No</button>
  </mat-dialog-actions>
</section>

```

Dijalog spada u kategoriju tzv. ulaznih komponenti (eng. *entry components*). Ovakve komponente se učitavaju na zahtev u vreme izvršavanja aplikacije. U cilju optimizacije performansi aplikacije, ulazne komponente, odnosno komponente koje ih sadrže se navode kao vrednost *entryComponents* svojstva *@Component* dekoratora korenog ili drugog modula aplikacije. Primer u nastavku ilustruje dodavanje komponente koja sadrži AM dijalog kao vrednost svojstva u korenom modulu aplikacije.

```

imports: [
  BrowserModule,
  MaterialModule,
  RoutingModule,
  BrowserAnimationsModule,
  FlexLayoutModule,
  FormsModule
],
providers: [TrainingService],
bootstrap: [AppComponent],
entryComponents: [StopTrainingComponent]

```

Dijalogom se upravlja programski iz kontrolera komponente kroz sledeće korake:

1. Dodavanje instance servisa *MatDialog* za upravljanje dijalogom.
2. Navođenje servisa kao argumenta konstruktora.
3. Otvaranje i zatvaranje dijaloga pomoću servisa.

Kod korišćenja dijaloga obično dve Angular komponente - komponenta koja poziva drugu komponentu koja sadrži dijalog (*pozivajuća komponenta*), i komponenta koja se poziva i sadrži dijalog (*komponenta dijaloga*).

U primeru u nastavku pozivajuća komponenta opisuje tekuću vežbu ili trening korisnika i prikazuje progres treninga. Korisnik može zaustaviti trening i tada se otvara druga komponenta kao dijalog koji traži potvrdu korisnika za

zaustavljanje treninga (komponenta dijaloga). Primer je preuzet iz studije slučaja u Sekciji 8.

Primer u nastavku ilustruje upravljanje dijalogom iz kontrolera pozivajuće komponente. Instanca servisa (*dialog*) se navodi kao argument konstruktora klase. Kada korisnik klikne na dugme za zaustavljanje treninga, poziva se metoda *onStop()*. Metoda otvara dijalog (*open()*) iz instance servisa. Metoda ima povratnu vrednost koja predstavlja pokazivač na objekat dijaloga. Prvi argument metode je naziv komponente dijaloga, dok se preko drugog argumenta dijalogu mogu proslediti podaci. Argument je tipa *any* i podaci koji se prosleđuju se navode kao parovi *ime:vrednost*, kao što je prikazano ispod. Izgled dijaloga je prikazan na slici u nastavku koda.

```

export class CurrentTrainingComponent implements OnInit {
  progress = 0;
  timer: any;
  constructor(private dialog: MatDialog) { }
  ngOnInit(): void {
    this.timer = setInterval(() => {
      this.progress += 5;
      if (this.progress >= 100) {
        clearInterval(this.timer);
      }
    }, 1000);
  }
  onStop() {
    clearInterval(this.timer);
    const dialogRef = this.dialog.open(StopTrainingComponent, {
      data: {
        progress: this.progress
      }
    });
    dialogRef.afterClosed().subscribe(result => {
      if (result) {
        clearInterval(this.timer);
      } else {
        this.onContinue();
      }
    });
  }
  onContinue() {
    this.timer = setInterval(() => {
      this.progress += 5;
      if (this.progress >= 100) {
        clearInterval(this.timer);
      }
    }, 1000);
  }
}

```

Stop training

Are you sure?

Yes

No

U kontroleru komponente dijaloga vrši se prihvatanje podataka koje prosleđuje pozivajuća komponenta. `MAT_DIALOG_DATA` je predefinisana simbolička konstanta koja omogućava pristup podacima koji se prosleđuju dijalogu u okviru `open()` metode. `@Inject()` direktiva se koristi za umetanje vrednosti argumenta (`MAT_DIALOG_DATA`) u navedenu varijablu (`receivedData`). Varijabla je tipa *any* kako bi čuvala vrednosti proizvoljnih tipova. Na ovaj način se kreira zavisnost argumenta i varijable tako da podaci argumenta bivaju prosleđivani kao vrednost varijable.

```
import { Component, Inject, OnInit } from '@angular/core';
import { MAT_DIALOG_DATA } from '@angular/material/dialog';

@Component({
  selector: 'app-stop-training',
  templateUrl: './stop-training.component.html',
  styleUrls: ['./stop-training.component.css']
})
export class StopTrainingComponent implements OnInit {

  constructor(@Inject(MAT_DIALOG_DATA) public receivedData: any ) { }

  ngOnInit(): void {
  }
}
```

Prihvaćeni podaci se prikazuju u templejtu komponente dijaloga korišćenjem interpolacije. Slika u nastavku koda ilustruje dijalog u prvom planu i pozivajuću komponentu u pozadini.

```
<section>
  <h1 mat-dialog-title>Stop training</h1>
  <mat-dialog-content>
    <p>Are you sure?</p>
    <p>Your progress is: {{ receivedData.progress }}</p>
  </mat-dialog-content>
  <mat-dialog-actions>
    <button mat-button [mat-dialog-close]="true">Yes</button>
    <button mat-button [mat-dialog-close]="false">No</button>
  </mat-dialog-actions>
</section>
```



Mehanizam prosleđivanja podataka od pozivajuće ka komponenti dijaloga je ilustrovan na Slici 6.12.



Slika 6.12. Šema prosleđivanja podataka od pozivajuće komponente ka dijalogu.

6.19. Korak

AM korak (eng. *stepper*) daje vizuelni prikaz složenije operacije GKI kao skupa koraka. Pogodna je za tzv. vizard (eng. *wizard*) procese koji se mogu raščlaniti na jednostavnije korake.

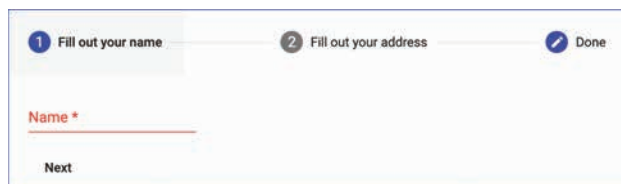
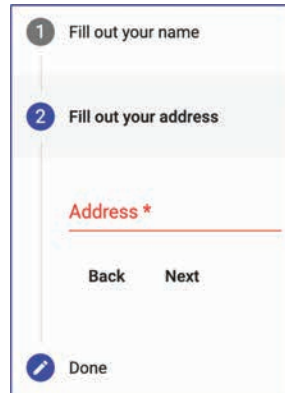
Korak kao modul je potrebno registrovati u odgovarajućem modulu (*material.module.ts*) i implementaciju (*MatStepperModule*) navesti u *imports* i *exports* svojstvima *@NgModule* dekoratora.

```
import { MatStepperModule } from
 '@angular/material/dialog';
```

Korak može biti orijentisan vertikalno (*mat-vertical-stepper*) ili horizontalno (*mat-horizontal-stepper*).

Struktura koraka je sledeća:

- Kontejnerski tag za horizontalni (*mat-horizontal-stepper*) ili vertikalni (*mat-vertical-stepper*) korak procesa.
- Veći broj elementarnih koraka procesa (*mat-step*).

A horizontal stepper interface. At the top, there are three steps: '1 Fill out your name' (active, blue circle), '2 Fill out your address' (inactive, grey circle), and 'Done' (inactive, blue circle with a checkmark). Below the first step, there is a text input field labeled 'Name *' with a red underline. At the bottom, there is a 'Next' button.A vertical stepper interface. On the left, there are two steps: '1 Fill out your name' (inactive, grey circle) and '2 Fill out your address' (active, blue circle). To the right of step 2, there is a text input field labeled 'Address *' with a red underline. Below the input field, there are 'Back' and 'Next' buttons. At the bottom, there is a 'Done' button with a blue circle and a checkmark.

Slika 6.13. Horizontalni (levo) i vertikalni (desno) korak.

Struktura elementarnog koraka je prikazana u nastavku. Direktiva atributa *[stepControl]* predstavlja kontrolu nad grupom operacija ili akcija koje korak sadrži. Njena vrednost je ime grupe. Grupa se kreira kao forma (*form*) sa skupom polja forme čiji sadržaj zavisi od funkcije konkretnog koraka. Direktiva atributa *[formGroup]* kao vrednost ima naziv grupe elemenata forme. Vrednosti dve direktive moraju biti identične jer na ovaj način korak objedinjuje sve operacije elemenata forme. Na kraju svakog koraka se nalazi

dugmad za navigaciju. Definiše se dugme za povratak na prethodni korak pomoću direktive *matStepperPrevious* i dugme za prelazak na sledeći korak korišćenjem direktive *matStepperNext*.

```
<mat-step [stepControl]="groupName">
  <form [formGroup]="groupName">
    <mat-form-field>...</mat-form-field> ...
    <button mat-button matStepperPrevious>Back</button>
    <button mat-button matStepperNext>Next</button>
  </form>
</mat-step>
```

Korak sadrži veći broj elementarnih koraka, kao u šemi ispod.

```
<mat-vertical-stepper linear>
  <mat-step [stepControl]="formGroup1">
    <form [formGroup]="formGroup1">
      <!-- ... -->
    </form>
  </mat-step>
  <mat-step [stepControl]="formGroup2">
    <form [formGroup]="formGroup2">
      <!-- ... -->
    </form>
  </mat-step>
</mat-vertical-stepper>
```

Primer u nastavku ilustruje vertikalni korak za registraciju korisnika aplikacije koji se sastoji iz tri elementarna koraka. Atribut *isLinear* određuje da li je korak lineran (*true*) ili nelinearan (*false*). Linearni korak podrazumeva da se elementarni koraci izvršavaju sekvencijalno, odnosno ispunjenje tekućeg koraka zahteva kompletiranje svih prethodnih koraka. U suprotnom je korak nelinearan. U primeru se linearnost kontroliše programski tako što se atribut povezuje sa atributom kontrolera *isLinear* (povezivanje atributa). Lokalna varijabla templejta *#stepper* ukazuje na korak i koristi se u nastavku za resetovanje (eng. *reset*) sadržaja čitavog elementa.

Prvi korak omogućava unos punog imena korisnika i sadrži dugme za prelazak na sledeći korak. U drugom koraku korisnik unosi adresu i može preći u naredni korak ili se vratiti u prethodni korak. Kroz treći korak korisnik se može ili vratiti u drugi korak ili resetovati sadržaj čitavog elementa klikom na dugme *Reset*.

Ng-template je generički tag koji se prikazuje samo ako se u okviru taga navede odgovarajuća naredba ili direktiva. U slučaju ispod, *matStepLabel* direktiva govori da je sadržaj taga naziv koraka.

Slike u nastavku koda prikazuju izgled svakog koraka.


```

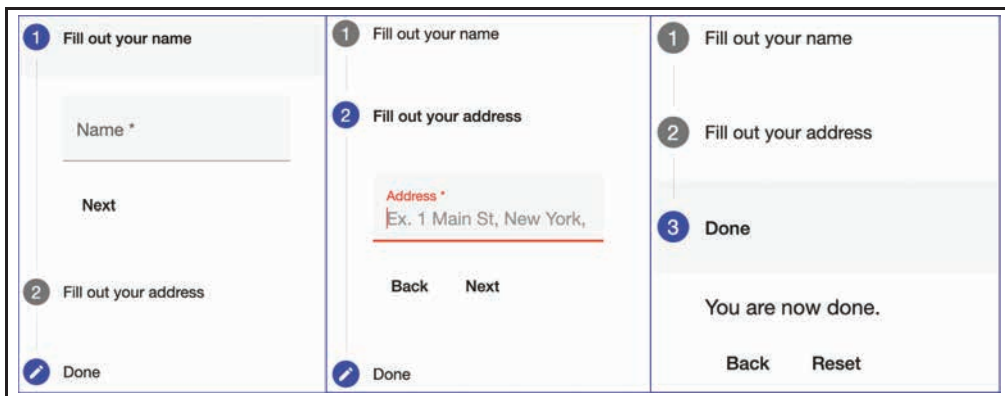
<mat-vertical-stepper [linear]="isLinear" #stepper>

  <!-- Korak 1 -->
  <mat-step [stepControl]="firstFormGroup">
    <form [formGroup]="firstFormGroup">
      <ng-template matStepLabel>Fill out your name</ng-template>
      <mat-form-field>
        <mat-label>Name</mat-label>
        <input
          matInput
          placeholder="Last name, First name"
          formControlName="firstCtrl"
          required>
        </mat-form-field>
      <div>
        <button mat-button matStepperNext>Next</button>
      </div>
    </form>
  </mat-step>

  <!-- Korak 2 -->
  <mat-step [stepControl]="secondFormGroup">
    <form [formGroup]="secondFormGroup">
      <ng-template matStepLabel>Fill out your address</ng-template>
      <mat-form-field>
        <mat-label>Address</mat-label>
        <input
          matInput
          placeholder="Address"
          formControlName="secondCtrl"
          required>
        </mat-form-field>
      <div>
        <button mat-button matStepperPrevious>Back</button>
        <button mat-button matStepperNext>Next</button>
      </div>
    </form>
  </mat-step>

  <!-- Korak 3 -->
  <mat-step>
    <ng-template matStepLabel>Done</ng-template>
    You are now done.
    <div>
      <button mat-button matStepperPrevious>Back</button>
      <button mat-button (click)="stepper.reset()">Reset</button>
    </div>
  </mat-step>
</mat-vertical-stepper>

```



Primer u nastavku proširuje prethodni i ilustruje promenu linearnosti koraka pomoću dugmeta. Na događaj klika dugmeta menja se vrednost atributa kontrolera logičkog tipa *isLinear*. U zavisnosti od vrednosti atributa prikazuje se odgovarajuća labela ili naziv dugmeta korišćenjem interpolacije. Korak je inicijalno nelinearan i dugme izgleda kao na slici levo. Klikom na dugme vrednost atributa *isLinear* postaje *true*, korak postaje linearan i dugme dobija naziv kao na slici desno.

```

<!-- Templejt komponente -->
<button mat-raised-button (click)="isLinear = !isLinear" id="toggle-linear">
  {{!isLinear ? 'Enable linear mode' : 'Disable linear mode'}}
</button>

//Pozadinska logika - kontroler komponente
export class StepperVerticalExample {
  isLinear = false;
  ...
}

```

Enable linear mode

1

Fill out your name

Disable linear mode

1

Fill out your name

Opcioni korak, tj. onaj koji se može preskočiti i ne mora kompletirati se navodi korišćenjem atributa *optional*. U prvom primeru se u templejtu definiše da je korak opcion, dok se u drugom primeru definiše programski tako što se atribut povezuje sa atributom kontrolera *isOptional*.

```
<mat-step [stepControl]="secondFormGroup" optional="true">
<mat-step [stepControl]="secondFormGroup" [optional]="isOptional">
```

Nazivima koraka se mogu pridružiti i simboli, odnosno ikone. U ovom slučaju se naziv koraka definiše kao vrednost atributa *label*. Koraku se dodaje atribut *state* čija je vrednost naziv ikone. Ikone koje se prikazuju se dodaju na kraju koraka korišćenjem *ng-template* taga sa atributom *matStepperIcon* čija je vrednost naziv ikone i ikonom kao sadržajem (*mat-icon*). Povezivanje ikone sa korakom se vrši na osnovu vrednosti dva atributa koje moraju biti identične. Važno je napomenuti da vrednosti ova dva atributa ne moraju odgovarati imenima u GMD kolekciji (videti 6.6). Sadržaj *mat-icon* taga mora identičan nazivu željene ikone iz GMD kolekcije.

Slike u nastavku koda ilustruju izgled prvog i drugog koraka uz čije nazive su dodate ikone.

```
<mat-stepper>
  <mat-step label="Step 1" state="phone">
    <p>Put down your phones.</p>
    <div>
      <button mat-button matStepperNext>Next</button>
    </div>
  </mat-step>
  <mat-step label="Step 2" state="chat">
    <p>Socialize with each other.</p>
    <div>
      <button mat-button matStepperPrevious>Back</button>
      <button mat-button matStepperNext>Next</button>
    </div>
  </mat-step>
  <mat-step label="Step 3">
    <p>You're welcome.</p>
  </mat-step>
  <ng-template matStepperIcon="phone">
    <mat-icon>call_end</mat-icon>
  </ng-template>
  <ng-template matStepperIcon="chat">
    <mat-icon>forum</mat-icon>
  </ng-template>
</mat-stepper>
```



6.20. Klizač

AM klizač (eng. *slider*) omogućava izbor vrednosti iz datog opsega, korišćenjem miša, dodira ili tastature. Element podseća na fizički klizač koji se koristi za definisanje vrednosti signala uređaja.

Klizač kao modul je potrebno registrovati u odgovarajućem modulu (*material.module.ts*) i implementaciju (*MatSliderModule*) navesti u *imports* i *exports* svojstvima *@NgModule* dekoratora.

```
import { MatSliderModule } from
 '@angular/material/slider';
```

Klizač se definiše i smešta u okviru taga *mat-slider*. Prilikom definisanja klizača navode se opseg, pomeraj i vrednost skale, kao u primeru u nastavku. Vrednosti atributa *min* i *max* određuju opseg vrednosti klizača. Atribut *step* definiše veličinu pomeraja klizača, dok *value* označava trenutnu izabranu vrednost. Ručica klizača (eng. *slider thumb*) se definiše pomoću direktive *matSliderThumb* koja se navodi u *input* tagu.

```
<mat-slider min="100" max="1000" step="10" value="250">
  <input matSliderThumb />
</mat-slider>
```



Sledeći primer ilustruje klizač opsega (eng. *range slider*) koji se zadaje pomoću dve ručice. Leva ili početna vrednost opsega se zadaje pomoću početne ručice (*matSlideStartThumb* direktiva), dok se desna ili krajnja vrednost opsega bira korišćenjem krajnje ručice (*matSliderEndThumb* direktiva).

```
<mat-slider min="200" max="500">  
  <input value="300" matSliderStartThumb>  
  <input value="400" matSliderEndThumb>  
</mat-slider>
```



7. Tehnologije za dizajn korisničkog interfejsa

Nakon dodavanja AM komponenti komponentama Angular aplikacije, potrebno je urediti sadržaj interfejsa. AM biblioteka implementira principe dizajna po definiciji, ali na nivou pojedinačnih komponenti. Potrebno je još prikazati sve komponente kao skladnu celinu tako da korisnik može lakše koristiti GKI. Ovde se pre svega misli na definisanje vizuelnog rasporeda komponenti u cilju prikladnog grupisanja u interfejsu.

U sekcijama u nastavku opisani su principi korišćenja dve tehnologije koje omogućavaju efikasno i efektivno raspoređivanje komponenti u interfejsu. FlexBox [81] biblioteka nudi naredbe za definisanje rasporeda sadržaja u CSS datoteci komponente ili globalnoj datoteci stila aplikacije (*styles.css*), dok FlexLayout biblioteka [82] obezbeđuje naredbe (ili direktive) koje se navode u templejtu komponente (HTML datoteka).

Za FlexBox biblioteku sintakse naredbi su opisane kao šematske definicije CSS klase stila, dok se FlexLayout direktive opisuju sa skupom mogućih vrednosti odvojenih simbolom uspravne crte (|) ili tipom vrednosti koji se navodi unutar simbola manje (<) i veće (>).

Iako u trenutku pisanja udžbenika postoje najave da Angular u budućnosti možda neće podržavati novije verzije FlexLayout biblioteke, ona je opisana kao alternativa FlexBox biblioteke.

7.1. FlexBox biblioteka

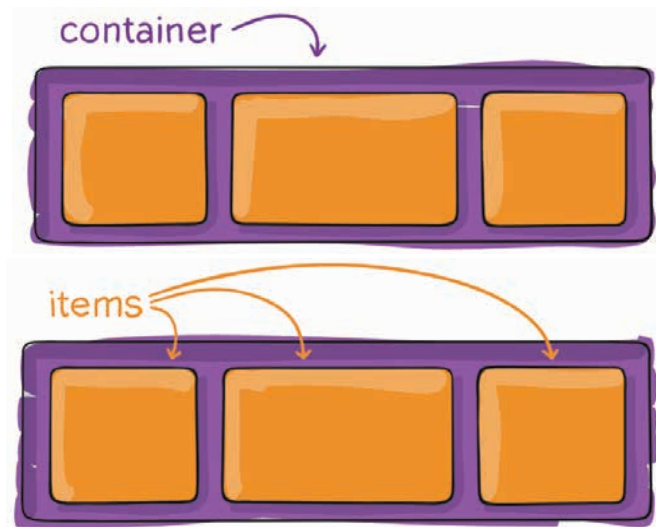
FlexBox [81] biblioteka je zasnovana na CSS jeziku. Biblioteka nasleđuje CSS model kutije (eng. *box model*) u kojem svaki element GKI zauzima pravougaoni region stranice sa graničnim prostorom sa spoljašnje strane elementa kao marginama (eng. *margin*) i unutrašnje strane kao dopunama (eng. *padding*). Omogućava fleksibilno raspoređivanje elemenata GKI kada njihova veličina nije unapred poznata ili se menja u toku korišćenja. Na ovaj način se optimalno koristi raspoloživ prostor ekrana.

Tehnologija polazi od dva tipa elemenata GKI za koje se definiše raspored:

- Roditelj (eng. *container*), i
- Dete (eng. *item*).

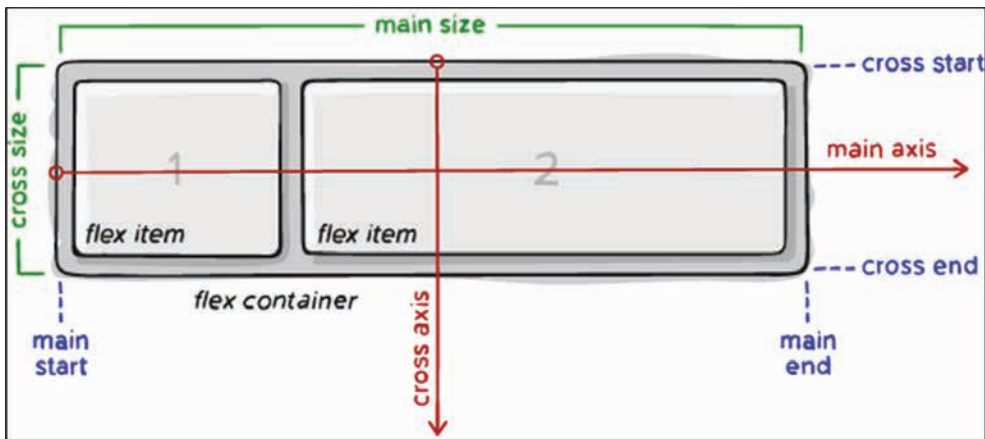
Roditelj je sadržavajući element ili kontejner. Često je to HTML tag grupisanja (na primer, *div* ili *section*) Roditelj sadrži elemente decu, kao što su

komponente. U zavisnosti od potreba aplikacije, raspoređivanje se može definisati na nivou roditelja za sve elemente decu, kao i na nivou deteta za pojedinačne elemente.



Slika 7.1. FlexBox elementi roditelji (eng. *container*) i deca (eng. *items*).
Izvor: [85].

Raspoređivanje se vrši u dvodimenzionalnom koordinatnom sistemu GKI, u kojem je glavna osa (eng. *main axis*) horizontalna, dok je upravna osa (eng. *cross axis*) uvek normalna na horizontalnu. Gornji levi ugao prikaza je podrazumevano koordinatni početak, glavna osa orijentisana sa leva na desno i upravna odozgo nadole. Orijentacija i smer dve ose se može menjati programski. Elementi deca se podrazumevano raspoređuju po glavnoj i upravnoj osi. Specifičan raspored se definiše pomoć CSS naredbi koje se opisane u nastavku.



Slika 7.2. FlexBox koordinatni sistem za raspoređivanje elemenata. Izvor: [85].

7.1.2. Direktive rasporeda roditeljskih elemenata

Da bi se mogle koristiti naredbe za specifičan raspored dece, element sa najpre mora eksplicitno proglasiti roditeljem. Elementu sa pridružuje CSS svojstvom *display* sa vrednošću *flex*.

```
.container {
    display: flex;
}
```

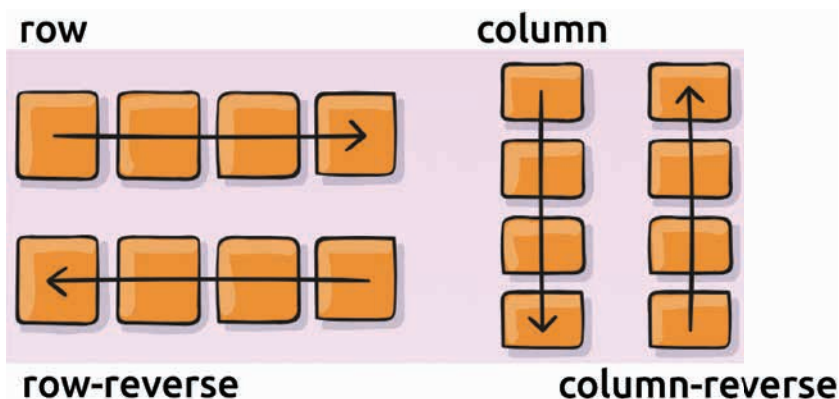
Rezultat je da se elementi deca pokušavaju smestiti u jedan red sa leva na desno, sa mogućim preklapanjem ukoliko ne postoji dovoljno prostora po glavnoj osi. Deca ne povećavaju veličinu po glavnoj osi (širina elementa), i mogu povećavati veličinu po upravnoj osi (visina elementa) ukoliko postoji raspoloživ prostor po visini prikaza.

Flex-direction definiše orijentaciju i smer glavne ose na osnovu kojih se raspoređuju elementi deca. Naredba može imati jednu od sledećih vrednosti:

- *Row* je podrazumevani smer (raspoređivanje dece sa leva na desno).
- *Row-reverse* orijentiše osu i raspoređuje decu sa desna na levo.
- *Column* postavlja glavnu osu vertikalno, sa orijentacijom i rasporedom dece odozgo nadole.
- *Column-reverse* rotira glavnu osu vertikalno, sa orijentacijom i rasporedom dece odozdo nagore.


```
.container {
  flex-direction: row |
  row-reverse |
  column |
  column-reverse;
}
```

Slika u nastavku ilustruje raspoređivanje elemenata dece za različite orijentacije glavne ose roditelja.



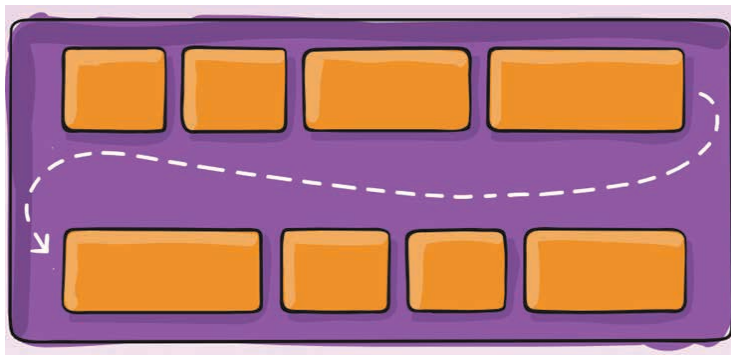
Slika 7.3. Orijentacije glavne ose roditelja i raspoređivanje dece za različite vrednosti flex-direction naredbe. Izvor: [85].

Flex-wrap definiše horizontalni raspored dece, da li se smeštaju u jedan ili više redova. Naredba može imati jednu od sledećih vrednosti:

- *Nowrap* je podrazumevana vrednost (deca se smeštaju u jedan red).
- *Wrap* omogućava raspoređivanje dece u više redova.
- *Wrap-reverse* označava smeštanje dece u više redova, ali u obrnutom redosledu (od podnožja roditelja ka vrhu).

```
.container {
  flex-wrap:
    nowrap |
    wrap |
    wrap-reverse;
}
```

Naredba se često koristi za prikazivanje elemenata čiji broj i veličina nisu unapred poznati. Slika u nastavku ilustruje raspored za *wrap* vrednost naredbe.



Slika 7.4. Flex-wrap naredba - raspoređivanje elemenata dece u više redova.
Izvor: [85].

Nakon utvrđivanja orijentacija osa i prelamanja redova, koriste se naredbe za tipična ravnjanja elemenata dece. Svako od ravnjanja zavisi od konkretnog sadržaj koji se prikazuje i ima za cilj da olakša i ubrza vizuelno skeniranje GKI i pronalaženje željenih informacija.

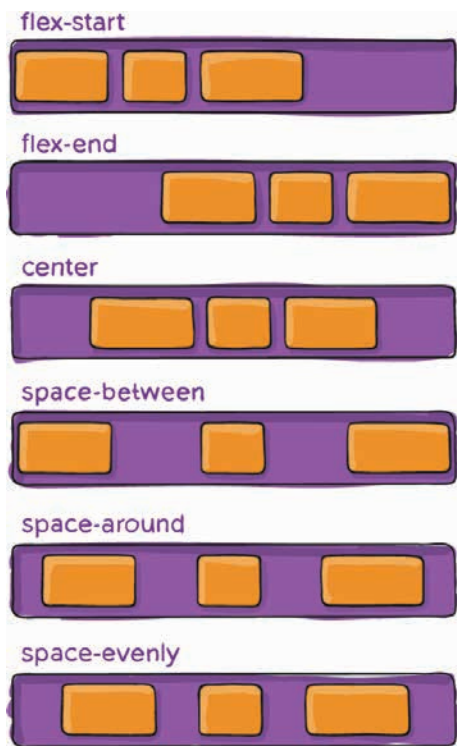
Justify-content definiše ravnjanje elemenata dece po glavnoj osi kao raspodelu prostora između elemenata dece. Može imati jednu od sledećih vrednosti:

- *Flex-start* ravna elemente decu uz početak glavne ose.
- *Flex-end* ravna elemente uz završetak glavne ose.
- *Center* ravna elemente po sredini glavne ose.
- *Space-between* raspoređuje decu sa podjednakim prostorom između elemenata tako da prvi i poslednji element budu poravnati uz početak i završetak granice roditelja.
- *Space-around* razmešta elementa kao u prethodnom slučaju, sa tim što su prvi i poslednji odvojeni od granice roditelja.
- *Space-evenly* definiše raspored u kojem je prostor po glavnoj osi ravnomerno raspoređen između elemenata, uključujući i prostor između granice roditelja i početnog i poslednjeg elementa.

```
.container {  
  justify-content:  
  flex-start |  
  flex-end |  
  center |  
  space-between |
```

```
space-around |  
space-evenly;  
}
```

Slika 7.5. ilustruje ravnanje elemenata dece po glavnoj osi za različite vrednosti naredbe *justify-content* roditelja.



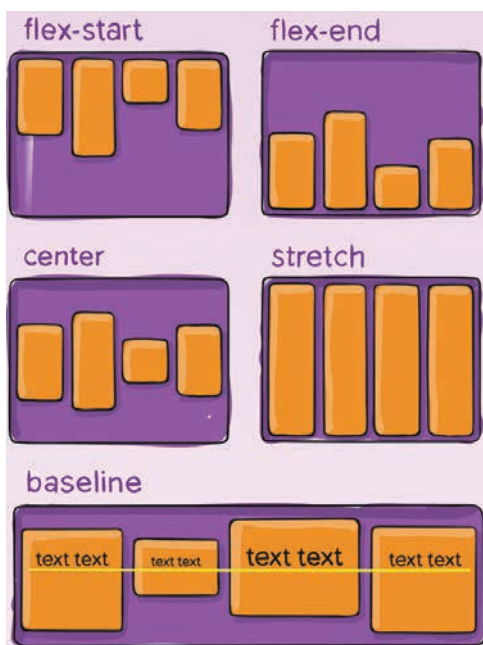
Slika 7.5. FlexBox varijante ravnanja po glavnoj osi i raspodela prostora između elemenata dece. Izvor: [85].

Align-items definiše ravnanje po upravnoj osi i raspodelu slobodnog prostora između elemenata dece. Vrednosti naredbe uključuju:

- *Flex-start* ravna elemente decu uz početak upravne ose.
- *Flex-end* ravna elemente decu uz kraj upravne ose.
- *Center* ravna elemente po sredini upravne ose.
- *Stretch* raspoređuje elemente tako da zauzimaju čitav prostor raspoloživ po upravnoj osi (obično vertikalno).
- *Baseline* ravna elemente u odnosu na osu koja predstavlja osnovnu liniju teksta dece.

```
.container {
  align-items:
    flex-start |
    flex-end |
    center |
    stretch |
    baseline;
}
```

Slika 7.6. prikazuje ravnanje elemenata dece po upravnoj osi za različite vrednosti naredbe *align-items* roditelja.



Slika 7.6. FlexBox varijante ravnanja po upravnoj osi i raspodela prostora između elemenata dece. Izvor: [85].

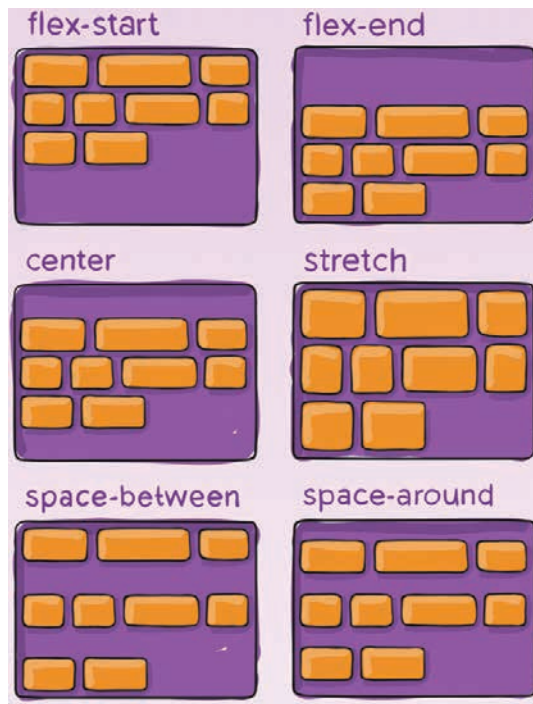
Align-content raspoređuje elemente decu duž upravne ose ukoliko se ne mogu smestiti u jedan nego u više redova.

- *Flex-start* raspoređuje redove dece tako da se ravnaju uz početak upravne ose.
- *Flex-end* raspoređuje redove dece tako da se ravnaju uz kraj upravne ose.
- *Center* raspoređuje redove dece tako da se ravnaju u odnosu na sredinu upravne ose.

- *Space-between* obezbeđuje ravnomeran međuprostor između redova dece po upravnoj osi, sa ravnanjem uz granice roditelja.
- *Space-around* obezbeđuje ravnomeran međuprostor između redova dece po upravnoj osi, uključujući i granice.
- *Stretch* označava identične visine redova dece tako da zauzimaju raspoloživ prostor po upravnoj osi roditelja (podrazumevano ponašanje).

```
.container {
  align-content:
    flex-start |
    flex-end |
    center |
    space-between |
    space-around |
    stretch; }
```

Slika 7.7 ilustruje ravnanje redova elemenata dece po upravnoj osi za različite vrednosti naredbe *align-content* roditelja.



Slika 7.7. FlexBox varijante ravnanja redove dece po upravnoj osi i raspodela prostora između redova. Izvor: [85].

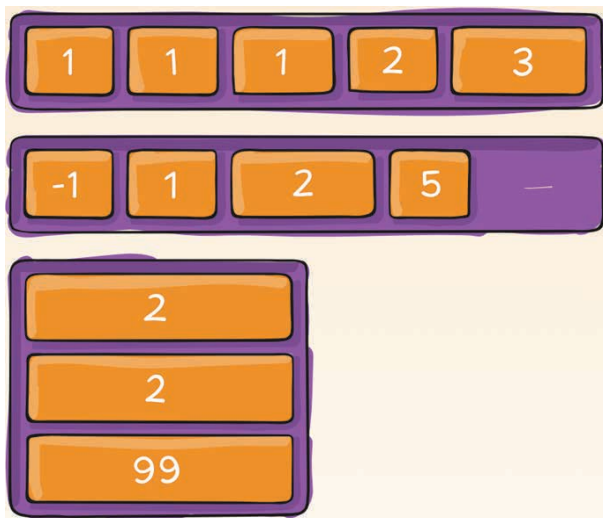
7.1.2. Raspoređivanje elemenata dece

Sadržani elementi se mogu dodatno urediti nakon definisanja rasporeda roditelja.

Order naredba koristi se za definisanje redosleda prikazivanja deteta u roditelju. Na ovaj način se može menjati podrazumevani način prikazivanja elemenata dece, redosledom kojim su definisana u HTML templejta. Vrednost je ceo broj tako da niža vrednost podrazumeva veći prioritet, odnosno element se prikazuje pre elemenata sa višom vrednošću naredbe. Podrazumevana vrednost je nula za svaki element.

```
.item { order: <integer>; /* podrazumevano = 0 */ }
```

Vrednosti se tretiraju relativno u odnosu na vrednosti drugih elemenata, kao što je prikazano na Slici 7.8.

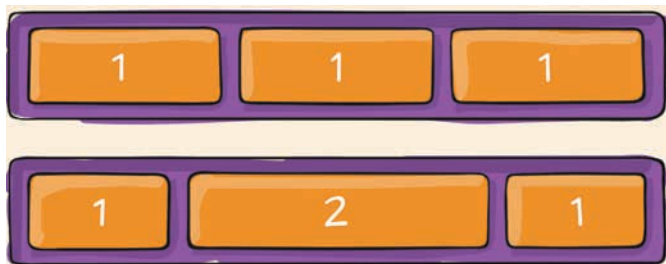


Slika 7.8. FlexBox redosled prikazivanja elemenata dece za različite vrednosti naredbe `order`. Izvor: [85].

Flex-grow određuje na koji način se povećava veličina elementa deteta po glavnoj osi u zavisnosti od trenutno raspoloživog prostora. Vrednost je broj sa značenjem proporcije, tj. udelom slobodnog prostora koji element može zauzeti u odnosu na druge elemente. Veća vrednost znači da će element zauzimati srazmerno više prostora po glavnoj osi. Podrazumevana vrednost je nula za svaki element.

```
.item { flex-grow: <number>; /* podrazumevano = 0 */ }
```

Slika 7.9. ilustruje korišćenje *flex-grow* naredbe.



Slika 7.9. FlexBox povećanje veličine elemenata dece po glavnoj osi pomoću naredbe *flex-grow*. Izvor: [85].

Flex-shrink definiše način na koji se smanjuje veličina elementa deteta po glavnoj osi u zavisnosti od trenutno raspoloživog prostora. Vrednost je broj sa značenjem proporcije, tj. udelom smanjenja veličine elementa u odnosu na druge elemente. Veća vrednost znači da će se element srazmerno više smanjivati po glavnoj osi. Podrazumevana vrednost je nula za svaki element (bez smanjivanja). Preporuka je zadavati identične vrednosti kako bi se elementi smanjivali ujednačeno.

```
.item { flex-shrink: <number>; /* podrazumevano = 0 */ }
```

Flex-basis određuje podrazumevanu veličinu elementa deteta po glavnoj osi. Vrednost može biti broj ili ključna reč. Poželjno je definisati podrazumevanu veličinu ukoliko će se veličina elemenata povećavati (*flex-grow*) ili smanjivati (*flex-shrink*). Podrazumevana vrednost je označena ključnom rečju *auto* koja ukazuje da će veličina deteta biti ona koja je navedena u odgovarajućoj CSS naredbi.

```
.item { flex-basis: <number> | auto; }
```

Flex direktiva kombinuje prethodne naredbe kao povećanje veličine (*flex-grow*), smanjenje veličine (*flex-shrink*) i podrazumevanu veličinu (*flex-basis*), u navedenom redosledu. Kao vrednosti naredbe se često koriste sledeće simboličke konstante sa specifičnim značenjima:

- *Flex: initial* je ekvivalentno sa *flex: 0 1 auto*. Prva vrednost (*flex-grow*) znači da element ne povećava veličinu po glavnoj osi. Druga (*flex-shrink*) ukazuje da element smanjuje veličinu po glavnoj osi. Treća vrednost (*flex-basis*) govori da će početna veličina deteta biti ona koja je unapred definisana za taj element. Ujedno i podrazumevane vrednosti.
- *Flex: auto* je identično kao *flex: 1 1 auto*. Prva i druga vrednost govore da će se veličina elementa po glavnoj osi povećavati i smanjivati u zavisnosti od raspoloživog prostora. Treća vrednost znači isto kao u prethodnom slučaju.
- *Flex: none* odgovara *flex: 0 0 auto*. Element ne menja veličinu po glavnoj osi. Treća vrednost daje isti rezultat kao u prethodnim kombinacijama.

7.2. FlexLayout biblioteka

FlexLayout [82] biblioteka koja sadrži HTML naredbe ili direktive za definisanje rasporeda elemenata u Angular komponentama. Slično FlexBox naredbama, postoje direktive templejta za roditeljske elemente i direktive koje se primenjuju na elemente decu. Biblioteka je nastala iz potreba programera za jednostavnijim raspoređivanjem elemenata GKI direktno u templejtu komponente, umesto korišćenja CSS naredbi.

Fleksibilan koordinatni sistem za raspoređivanje elemenata (glavna i upravna osa) i tipovi elemenata (roditelj i dete) su identični kao u FlexBox biblioteci. Podrška za FlexLayout se mora instalirati u postojeći Angular projekat pomoću naredbe u nastavku.

```
npm install @angular/flex-layout --save
```

Instalacija se može videti u *package.json* datoteci projekta.

```
"dependencies": {
  "@angular/animations": "^14.2.4",
  "@angular/cdk": "^14.2.3",
  "@angular/common": "^14.2.0",
  "@angular/compiler": "^14.2.0",
  "@angular/core": "^14.2.0",
  "@angular/flex-layout": "^14.0.0-beta.40",
  "@angular/forms": "^14.2.0",
```

Slika 7.10. Instalacija FlexLayout biblioteke u *package.json* datoteci Angular projekta.

7.2.1. Raspoređivanje roditeljskih elemenata

Podrazumevani raspored definiše da se elementi deca pokušavaju smestiti u jedan red sa leva na desno, sa mogućim preklapanjem ukoliko ne postoji dovoljno prostora po glavnoj osi. Deca ne povećavaju veličinu po glavnoj osi (širina elementa), i mogu povećavati veličinu po upravnoj osi (visina elementa) ukoliko postoji raspoloživ prostor po visini prikaza.

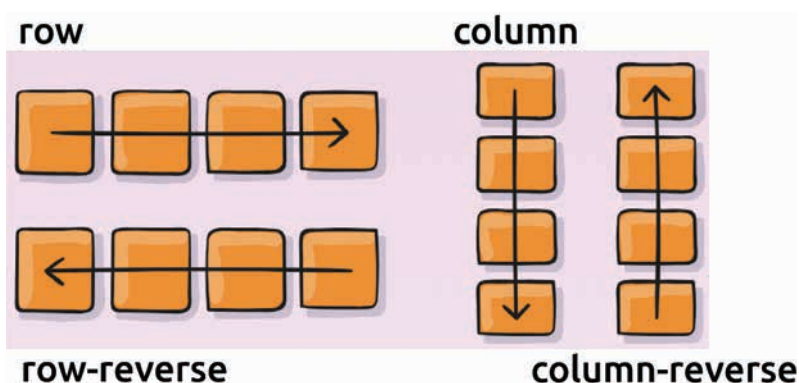
Direktive u nastavku mogu prilagoditi raspored zahtevima konkretnog GKI.

FlexLayout direktiva definiše orijentaciju i glavni smer glavne ose. *FlexBox* ekvivalent je naredba *flex-direction*. Naredba može imati jednu od sledećih vrednosti:

- *Row* je podrazumevani smer (raspoređivanje dece sa leva na desno).
- *Row-reverse* orijentiše osu i raspoređuje decu sa desna na levo.
- *Column* postavlja glavnu osu vertikalno, sa orijentacijom i rasporedom dece odozgo nadole.
- *Column-reverse* rotira glavnu osu vertikalno, sa orijentacijom i rasporedom dece odozdo nagore.

```
<div fxLayout="row | row-reverse | column | column-reverse">
```

Slika 7.11 prikazuje rasporede elemenata dece za različite orijentacije i smerova glavne ose roditelja.



Slika 7.11. Orijentacije glavne ose roditelja i raspoređivanje dece za različite vrednosti *fxLayout* direktive. Izvor: [85].

Primer u nastavku ilustruje korišćenje *fxLayout* direktive sa vrednošću *row* tako da se elementi deca raspoređuju horizontalno prema podrazumevanoj orijentaciji glavne ose.

```
<div fxLayout="row">
  <div>1. One</div>
  <div>2. Two</div>
  <div>3. Three</div>
</div>
```



Sledeći primer glavnu osu postavlja vertikalno tako što je rotira za 90 stepeni u smeru kazaljke na satu. Zbog toga se elementi deca raspoređuju vertikalno.

```
<div fxLayout="column">
  <div>1. One</div>
  <div>2. Two</div>
  <div>3. Three</div>
</div>
```

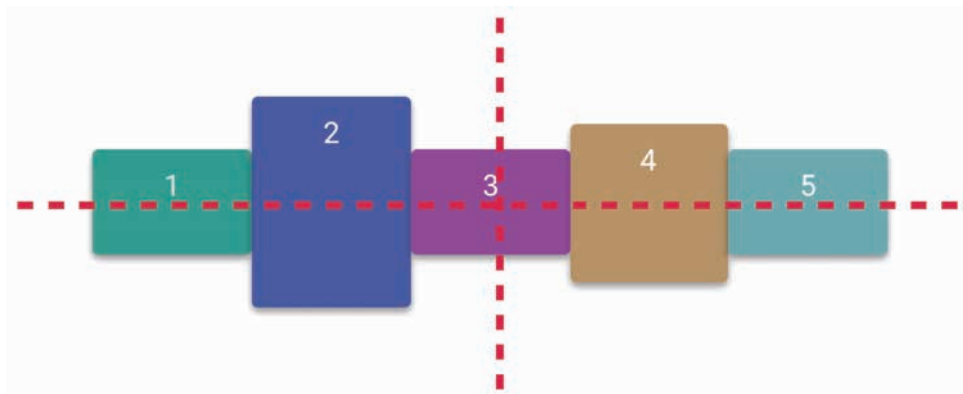


FxLayoutAlign direktiva definiše ravnanje elemenata dece prema glavnoj osi i, opciono, prema upravnoj osi. Prva navedena vrednost predstavlja ravnanje prema glavnoj osi, dok druga vrednost definiše ravnanje po upravnoj osi.

Primer ispod raspoređuje elemente po glavnoj osi (*row*) i zatim ravna po sredini u odnosu na glavnu (prva *center* vrednost) i upravnu osu (druga *center* vrednost).

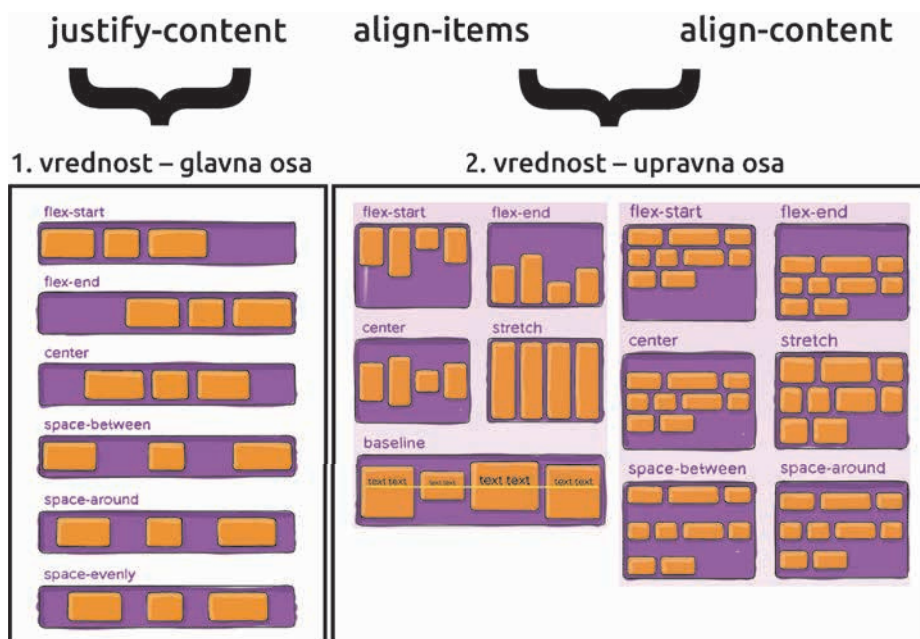
```
<div fxLayout="row" fxLayoutAlign="center center">...</div>
```

Rezultat rasporeda može biti kao na Slici 7.12.



Slika 7.12. `fxLayout` direktiva vrši ravnanje po glavnoj i upravnoj osi. Izvor [86].

Sa obzirom da omogućava raspoređivanje po obe ose, direktiva `fxLayoutAlign` se može posmatrati i kao kombinacija dve `FlexBox` naredbe kao što je prikazano na Slici 7.13. Prva vrednost odgovara naredbi `justify-content` koja raspoređuje elemente po glavnoj osi. Druga vrednost može biti `align-items` ili `align-content` koje ravnaaju elemente po upravnoj osi.



Slika 7.13. FxLayoutAlign direktiva objedinjuje više FlexBox naredbi. Izvor: [85].

FxLayoutGap direktiva omogućava dodavanje margina elementima deci. U zavisnosti od orijentacije glavne ose, dodaju se margine na određenim stranama. Ukoliko je glavna osa podrazumevano orijentisana, tj. horizontalno, dodaju se horizontalne margine. Ako je vertikalna, margine se elementima dodaju po vertikali.

Naredba ima vrednosti postojećih CSS jedinica mera. Preporuka je koristiti relativne jedinice (procenti), umesto apsolutnih vrednosti (pikseli). Kao relativne jedinice mere koriste se *vw* (eng. *viewport width*) i *vh* (eng. *viewport height*). Vrednosti su skalabilne tako da 1 *vw* zauzima 1% trenutne širine prozora Veb pretraživača, dok 1 *vh* zauzima 1% trenutne visine prozora pretraživača.

Primer u nastavku ilustruje dodavanje horizontalnih margina elementima deci. Margine su date u pikselima.

```
<div fxLayout="row" fxLayoutGap="20px">
  <div>1. One</div>
  <div>2. Two</div>
  <div>3. Three</div>
</div>
```



Sledeći primer glavnu osu orijentiše vertikalno. Zbog toga se elementima deca dodaju vertikalne margine izražene u pikselima.

```

<div fxLayout="column" fxLayoutGap="20px">
  <div>1. One</div>
  <div>2. Two</div>
  <div>3. Three</div>
</div>

```

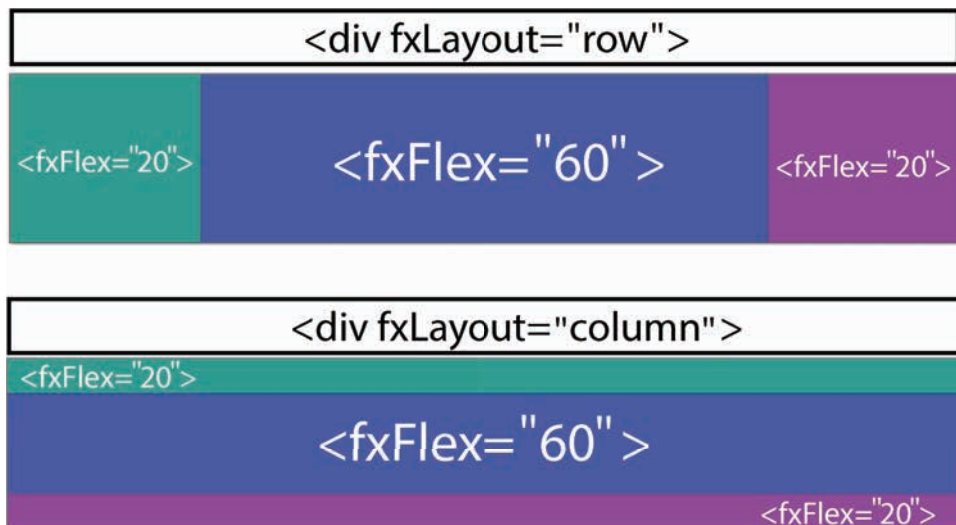
7.2.2. Direktive rasporeda elemenata dece

Elementi deca se mogu dodatno urediti nakon definisanja rasporeda na nivou roditelja.

Flex definiše promenu veličine elementa deteta u odnosu na glavnu osu roditelja. Promena se navodi kao podrazumevana veličina, i opcije za povećanje i smanjenje veličine u zavisnosti od trenutno raspoloživog prostora. Tri vrednosti se navode redom za povećanje (*grow*), smanjenje (*shrink*) i podrazumevanu veličinu (*basis*) elementa. Preporuka je korišćenje relativnih jedinica mere (procenti ili skalabilne vrednosti).

Može se navoditi i jedna vrednost koja označava podrazumevanu veličinu elementa po glavnoj osi (*basis*).

U primerima u nastavku, roditeljski *div* element sadrži tri *div* elementa deteta. U prvom primeru, elementi se raspoređuju horizontalno (podrazumevana orijentacija glavne ose), dok se u drugom raspoređuju vertikalno. Vrednosti direktive *flex* za elemente decu definišu podrazumevane veličine po glavnoj osi. Elementi deca zauzimaju prostor srazmeran odnosu između navedenih vrednosti direktive.



Slika 7.14. *flex* direktiva određuje povećavanje, smanjenje i podrazumevanu veličinu elementa deteta u *flexLayout* roditelju. Izvor: [86].

U nastavku je dat pregled *flex* simboličkih konstanti (alias) i značenje odgovarajuće *flex* CSS naredbe. Prva vrednost se odnosi povećanje veličine, sledeća na smanjenje i poslednja na podrazumevanu veličinu.

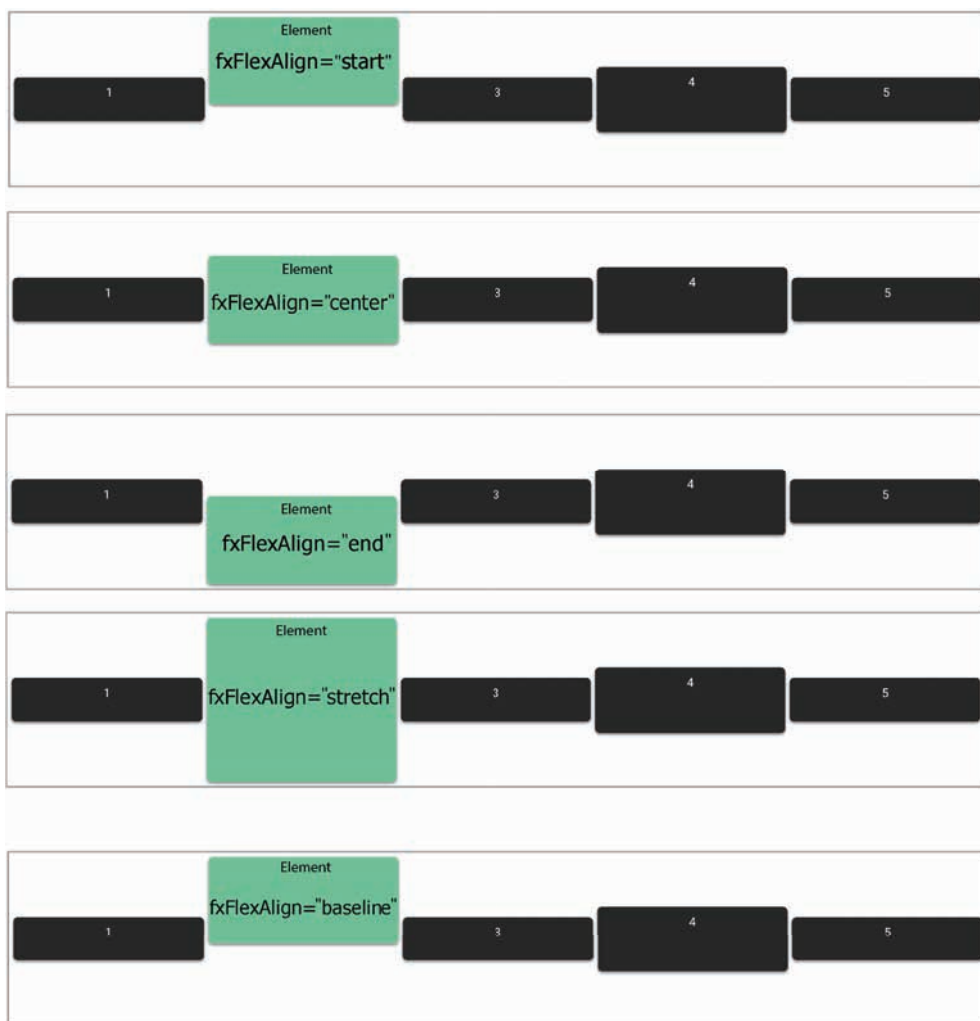
Tabela 7.1. FxFlex simboličke konstante sa opisom.

Alias	FlexBox značenje	Opis
<i>grow</i>	<i>flex: 1 1 100%;</i>	Element se povećava i smanjuje i zauzima raspoloživ prostor po glavnoj osi.
<i>none</i>	<i>flex: 0 0 auto;</i>	Element se ne povećava niti smanjuje i ima veličinu kao po definiciji (<i>width</i> i <i>height</i> atributi) i zauzima raspoloživ prostor po glavnoj osi.
<i>nogrow</i>	<i>flex: 0 1 auto;</i>	Element se ne povećava ali se smanjuje i ima veličinu kao po definiciji (<i>width</i> i <i>height</i> atributi) i zauzima raspoloživ prostor po glavnoj osi.
<i>noshrink</i>	<i>flex: 1 0 auto;</i>	Element se povećava ali se ne smanjuje i ima veličinu kao po definiciji (<i>width</i> i <i>height</i> atributi) i zauzima raspoloživ prostor po glavnoj osi.

FxFlexAlign direktiva redefiniše vrednost ravnanja na nivou roditelja po upravnoj osi. Na ovaj način se element dete može istaći u odnosu na ostatak elemenata, odnosno realizovati kontrast kao princip dizajna GKI. Vrednosti naredbe su sledeće:

- *Start* ravna element dete uz početak upravne ose roditelja.
- *Center* raspoređuje element dete po sredini upravne ose roditelja.
- *End* postavlja element dete uz kraj upravne ose roditelja.
- *Baseline* ravna element dete u odnosu na osu koja predstavlja osnovnu liniju teksta elemenata dece.
- *Stretch* povećava veličinu deteta tako da zauzima čitav raspoloživ prostor po upravnoj osi.

Slika 7.15. ilustruje poziciju elementa deteta za moguće vrednosti *fxFlexAlign* direktive. Upravna osa je orijentisana vertikalno.



Slika 7.15. `fxFlexAlign` direktiva omogućava raspoređivanje pojedinačnih elemenata dece po upravnoj osi roditelja. Izvor: [86].

8. Studija slučaja Angular aplikacije

Studija slučaja daje primer Angular aplikacije koja koristi prethodno opisane tehnologije. Izrada aplikacije se organizovana po sekcijama koje prate tipičan proces izrade Veb GKI u Angular tehnologijama. Svaka sekcija počinje sažetkom koji sumira korake razvoja aplikacije u datoj sekciji.

U studiji slučaja se kreira pojednostavljen primer GKI Angular aplikacije za simulaciju fizičkog treninga. Korisnik se može prijaviti postojećim nalogom ili registrovati nalog. Može se birati i pokretati neki od predefinisanih treninga i pratiti trening u toku. Korisnik može pregledati završene treninge i vršiti pretragu na osnovu zadatog kriterijuma. Interfejsa se prilagođava različitim dimenzijama ekrana.

Prilikom dodavanja gradivnih elemenata aplikacije, čitalac se upućuje na odgovarajuću sekciju u kojoj je korišćenje datog elementa detaljnije opisano.

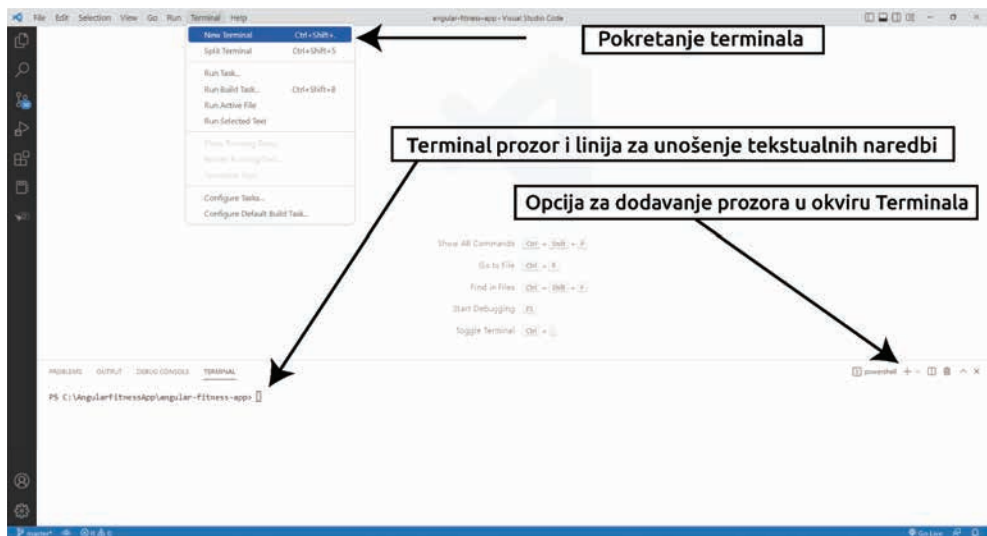
Razvojno okruženje korišćeno za izradu projekta je Visual Studio Code (VSC). Okruženje je dostupno u [87].

8.1. Uvod u strukturu projekta i instalacija potrebnih biblioteka

Sažetak: Uvodni deo prikazuje razvojno okruženje i Angular CLI naredbe za instalaciju potrebnih biblioteka i programsko dodavanje gradivnih elemenata aplikacije. Zatim se dodaju komponente kao entiteti aplikacije, modul za registrovanje AM komponenti i modul za rutiranje.

8.1.1. Razvojno okruženje i kreiranje projekta

Slika 8.1. prikazuje deo razvojnog okruženja VSC za korišćenje Angular CLI. Interfejs komandne linije je dostupan u *Terminal* prozoru okruženja.



Slika 8.1. Terminal prozor kao interfejs komandne linije razvojnog okruženja Visual Studio Code.

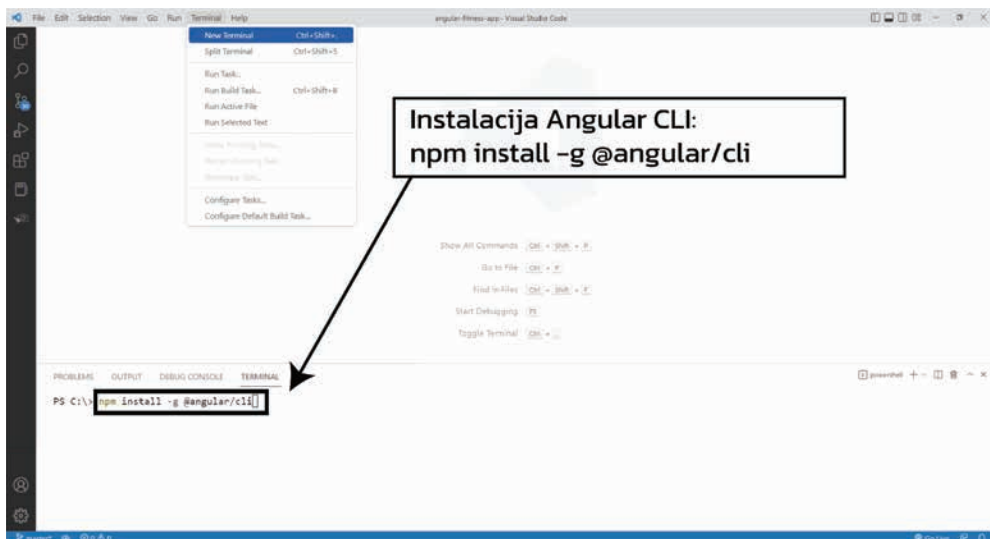
Najpre je potrebno instalirati Node.js izvršno okruženje i interfejse komandne linije (*npm - Node Package Manager*) korišćenjem sledeće naredbe.

```
npm install -g npm
```

Angular CLI se zatim instalira kao zaseban alat u *Terminal* prozoru pomoću naredbe u nastavku.

```
npm install -g @angular/cli
```

Slika 8.2. prikazuje naredbu u *Terminal* prozoru VSC.



Slika 8.2. Instalacija Angular interfejsa komandne linije (CLI) u Terminal prozoru razvojnog okruženja.

Naredba u nastavku kreira Angular projekat pod nazivom *angular-fitness-app*. Kreirani projekat se smešta u istoimeni direktorijum. Projekat se kreira bez modula za rutiranje koji se dodaje naknadno u Sekciji 8.1.8. Potrebno je izabrati CSS kao jezik za vizuelni dizajn komponenti aplikacije.

```
ng new angular-fitness-app

? Would you like to add Angular routing? No
? Which stylesheet format would you like to use? (Use arrow
keys)
> CSS
  SCSS  [ https://sass-lang.com/documentation/syntax#scss
        ]
  Sass  [ https://sass-lang.com/documentation/syntax#the-indented-syntax ]
  Less  [ http://lesscss.org
        ]
```

Direktorijumu kreiranog projekta pristupa se putem *Terminal* prozora pomoću naredbe u nastavku.

```
cd angular-fitness-app
```

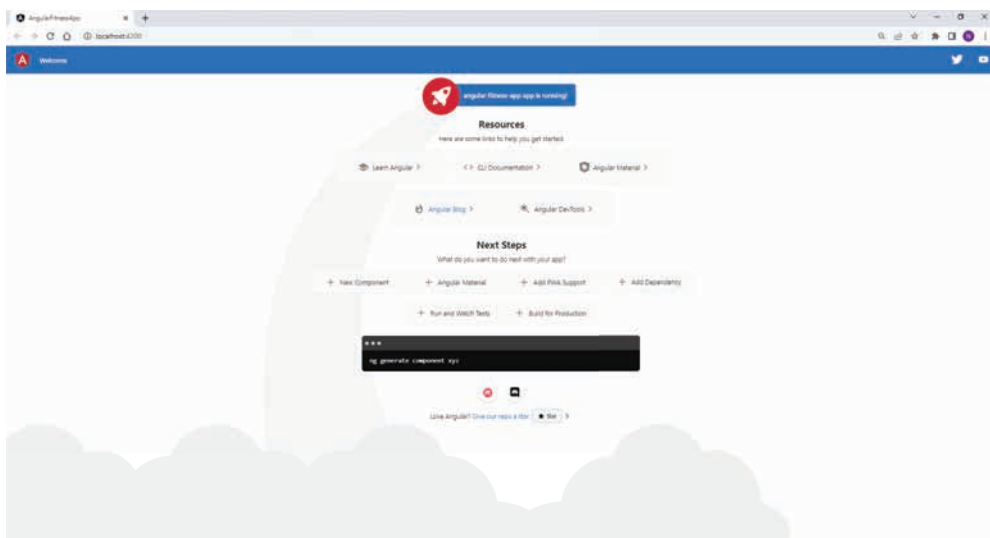
Slika 8.3. prikazuje hijerarhijsku strukturu Angular projekta u *Explorer* prozoru razvojnog okruženja.


```
| 210.09 kB |
| 17.63 kB |
| 6.52 kB |
```

```
2 node
2 powershell
```

Slika 8.4. Terminal prozor u razvojnom okruženju.

Nakon pokretanja razvojnog servera u pretraživaču se može videti Angular aplikacija sa predefinisanim sadržajem. Adresa lokalnog razvojnog servera i standardni port Angular aplikacije u Veb pretraživaču je *localhost:4200*.



Slika 8.5. Izgled kreiranog Angular projekta u Veb pretraživaču.

8.1.3. Kreiranje komponenti

Angular komponente se mogu kreirati programski u *Terminal* prozoru. Komponente su detaljnije opisane u Sekciji 5.3. Naredba za kreiranje komponente je kao u nastavku.

```
ng generate component ime_komponente
```

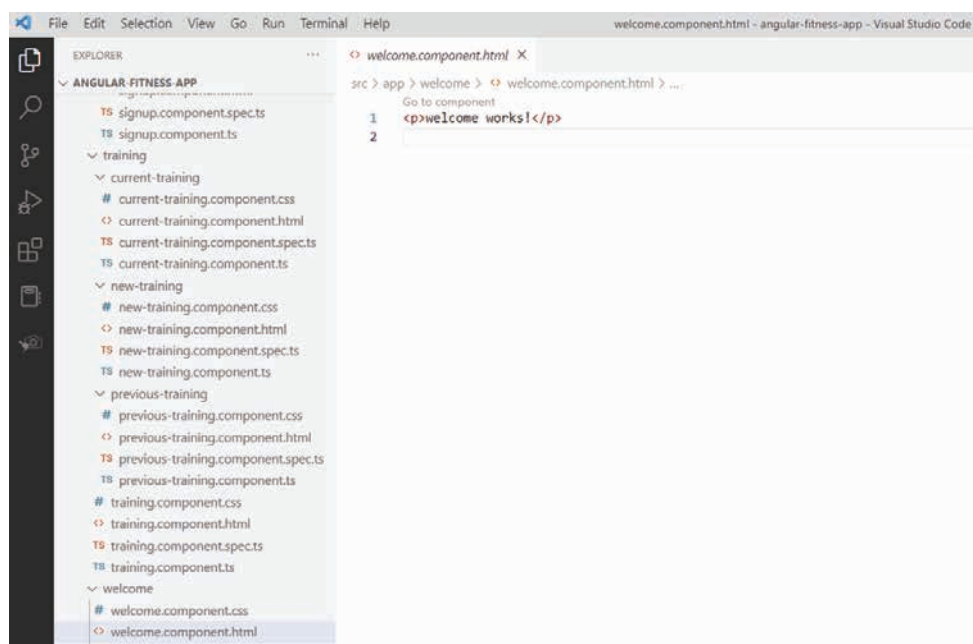
Kreirane komponente se smeštaju u *src* direktorijum projekta. U okviru ovog direktorijuma se mogu smeštati u poddirektorijumu pomoću naredbe ispod.

```
ng generate component direktorijum/ime_komponente
```

Kreirati komponente pomoću sledećih naredbi:

- *ng g c auth/signup* - komponenta za registraciju korisnika,
- *ng g c auth/login* - komponenta za prijavu korisnika,
- *ng g c welcome* - komponenta početna stranica Angular aplikacije,
- *ng g c training* - komponenta koja integriše delove treninga kao sadržane komponente,
- *ng g c training/previous-training* - komponenta istorijata treninga,
- *ng g c training/current-training* - komponenta tekućeg treninga,
- *ng g c training/new-training* - komponenta za kreiranje novog treninga.

Kreirane komponente se mogu videti u strukturi projekta kao što je prikazano na Slici 8.6.



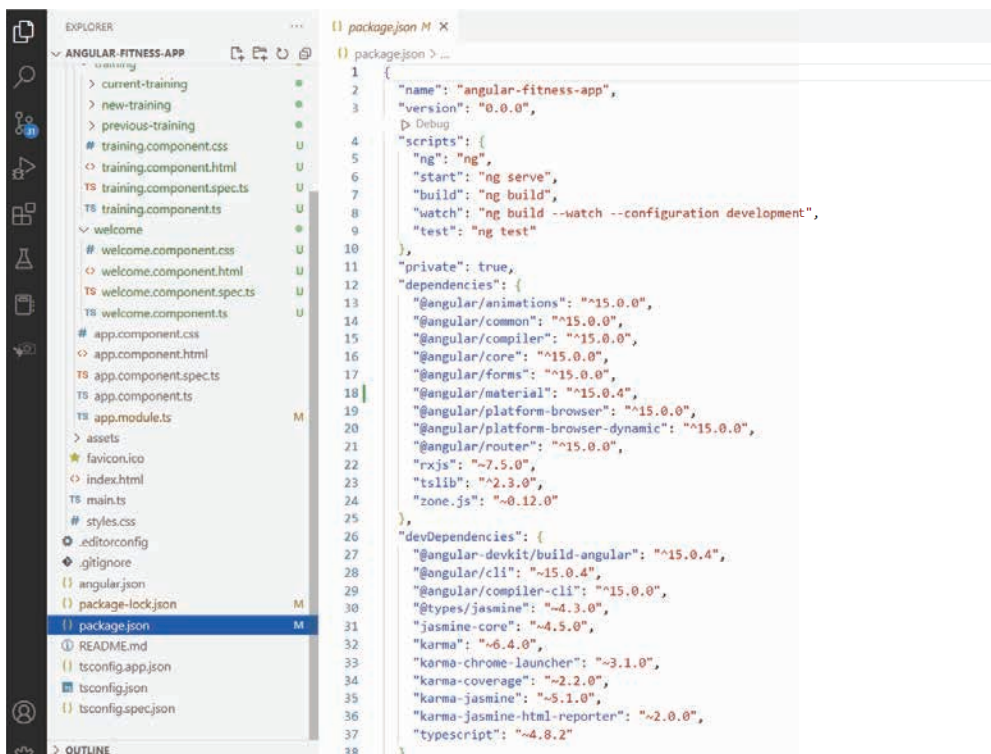
Slika 8.6. Pregled komponenti aplikacije u razvojnom okruženju.

8.1.4. Instalacija AM biblioteke

Naredba u nastavku instalira biblioteke neophodne za korišćenje AM komponenti.

```
npm install @angular/material
```

Verzije instaliranih biblioteka se mogu videti u *package.json* datoteci, kao na slici ispod.



Slika 8.7. Verzije instaliranih biblioteka u datoteci *package.json* Angular projekta.

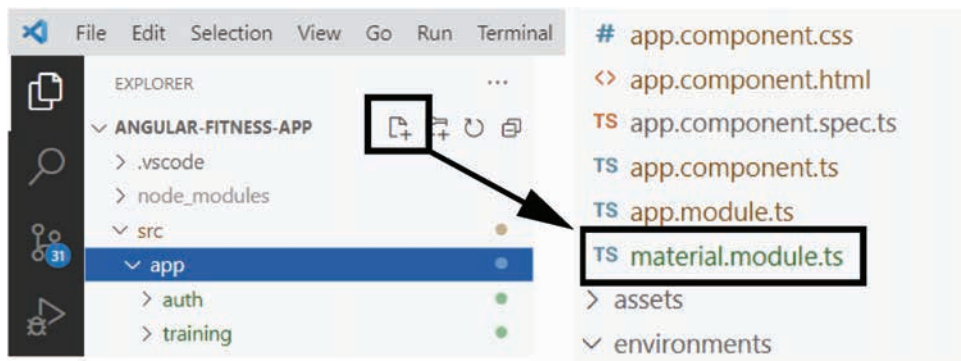
Instalacija AM komponenti donosi i predefinisane teme. Više informacija u vezi sa temama se može naći u Sekciji 6.3. Tema se definiše, tj. primenjuje na nivou projekta tako što se navodi u datoteci stila projekta (*styles.css*) smeštenoj u *src* direktorijumu.

```
@import '@angular/material/prebuilt-themes/indigo-pink.css';
```

8.1.5. Kreiranje modula za registraciju AM komponenti

U *src/app* direktorijumu projekta (iz *Explorer* prozora) potrebno je kreirati modul za registraciju AM komponenti pod nazivom *material.module.ts*. Moduli su opisani u Sekciji 5.2, dok je kreiranje i korišćenje navedenog modula opisano u Sekciji 6.2.

Slika u nastavku ilustruje kreiranje modula pomoću odgovarajuće ikone razvojnog okruženja.



Slika 8.8. Kreiranje datoteke za registraciju AM komponenti u src/app direktorijumu projekta.

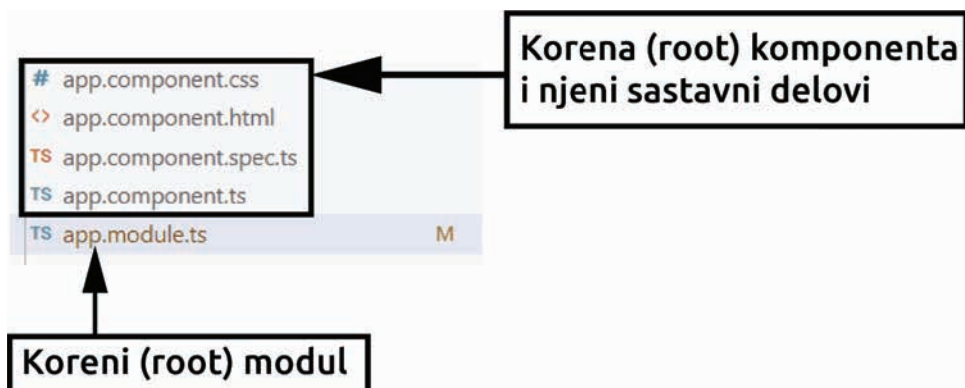
Slika u nastavku prikazuje registraciju AM komponente dugme (*MatButtonModule*) u kreiranom modulu.

```
import { NgModule } from '@angular/core';
import { MatButtonModule } from '@angular/material/button';

@NgModule({
  imports: [MatButtonModule],
  exports: [MatButtonModule],
})
export class MaterialModule {}
```

8.1.6. Navođenje modula za registraciju AM komponenti u korenom modulu

Slika 8.9. prikazuje korenu komponentu i koreni modul aplikacije u projektu.



Slika 8.9. Korena komponenta i koreni modul aplikacije.

Prethodno kreirani modul za registraciju AM komponenti se navodi u korenom modulu aplikacije, kao u kodu u nastavku. Postupak je detaljnije opisan u Sekciji 6.2.

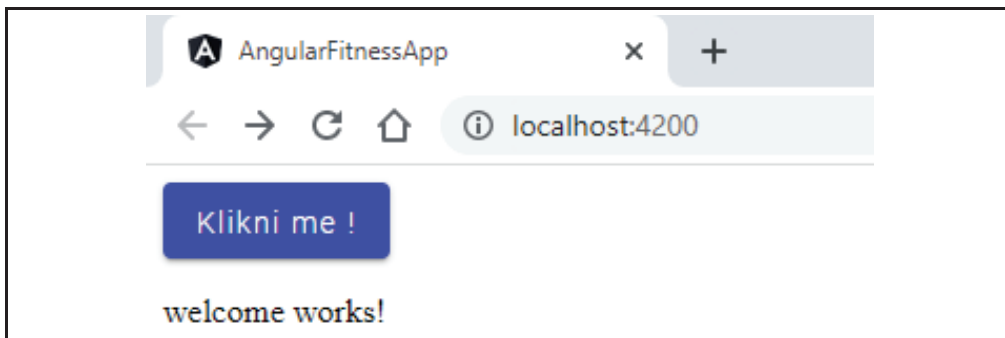
```
import { MaterialModule } from './material.module';

imports: [
  BrowserModule,
  MaterialModule
],
```

8.1.7. Dodavanje dugmeta u templejt komponente

U nastavku se dodaje AM komponenta dugme u templejt korene komponente aplikacije (`app.component.html`). Sačuvati promene i pogledati rezultat u prozoru pretraživača kao na slici ispod.

```
<section>
  <button mat-raised-button color="primary">Klikni me !</button>
</section>
```



Zadatak: Na sličan način dodati dugme u svaku kreiranu komponentu aplikacije.

8.1.8. Implementacija rutiranja

Modul za rutiranje ili ruter se kreira pomoću naredbe u nastavku. Naredba kreira ruter kao datoteku sa nazivom *routing.module.ts* u *src/app* direktorijumu projekta.

```
ng generate module --routing
```

Napomena: Pored generisanja pomoću naredbe iznad, ruter se može kreirati u procesu pravljenja Angular projekta potvrdom opcije za kreiranje modula za rutiranje.

Rad sa ruterom je detaljnije opisan u Sekciji 5.5. Kod u nastavku definiše rute do komponenti aplikacije u klasi modula za rutiranje.

```

import { NgModule } from '@angular/core';
import { RouterModule, Routes } from '@angular/router';
import { LoginComponent } from './auth/login/login.component';
import { SignupComponent } from './auth/signup/signup.component';
import { TrainingComponent } from './training/training.component';
import { WelcomeComponent } from './welcome/welcome.component';

const route: Routes = [
  { path: '', component: WelcomeComponent },
  { path: 'signup', component: SignupComponent },
  { path: 'login', component: LoginComponent },
  { path: 'training', component: TrainingComponent }
];

@NgModule({
  imports: [RouterModule.forRoot(route)],
  exports: [RouterModule]
});
export class RoutingModule {}

```

Ruter je potrebno registrovati u korenom modulu aplikacije kao u nastavku.

```

import { RoutingModule } from './routing.module';

imports: [
  BrowserModule,
  MaterialModule,
  RoutingModule
]

```

Ukloniti postojeći sadržaja templejta korene komponente (*app.component.html*) i navesti *router-outlet* tag koji označava mesto na kojem se prikazuje sadržaj komponente do koje vodi trenutna ruta.

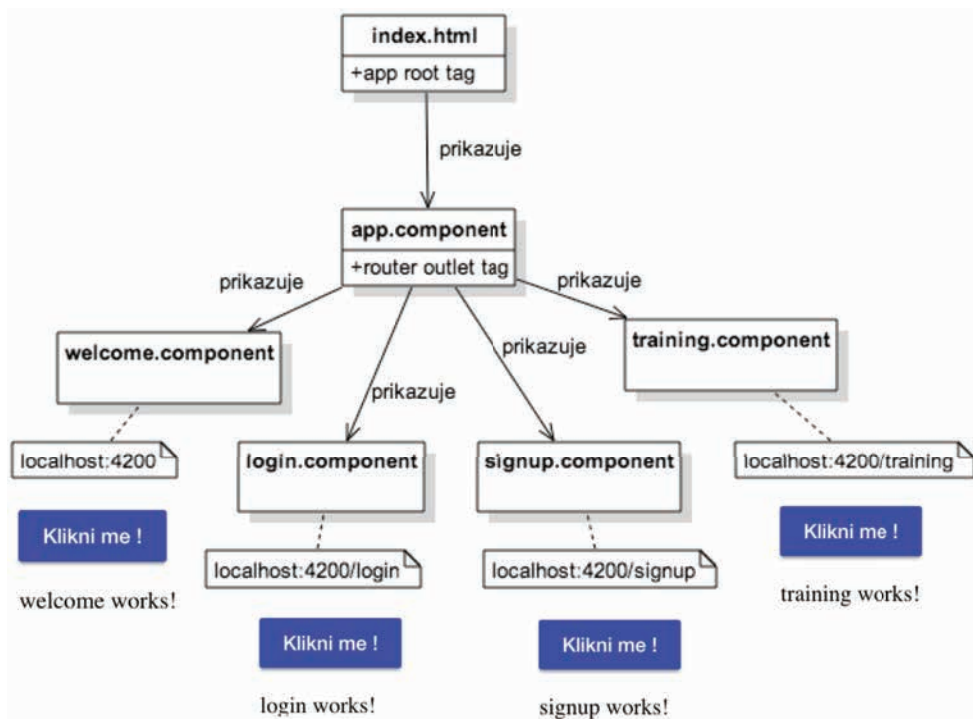
```
<router-outlet></router-outlet>
```

Pokrenuti razvojni server i testirati rute do komponenti navođenjem sledećih adresa u adresnoj liniji Veb pretraživača:

- *localhost:4200*,
- *localhost:4200/signup*,
- *localhost:4200/login*,

- `localhost:4200/training`.

Slici 8.10. prikazuje strukturu rutiranja. Selektor korene komponente aplikacije (*app-root*) se navodi u *index.html* datoteci projekta. *Router-outlet* tag se zatim smešta u templejt korene komponente. Na ovaj način će se sadržaj, tj. templejt svake komponente aplikacije za koju je definisana ruta prikazivati na jednom mestu. Na ovaj način je implementiran SPA princip (videti Sekciju 5.3.1).



Slika 8.10. Implementacija SPA principa u studiji slučaja.

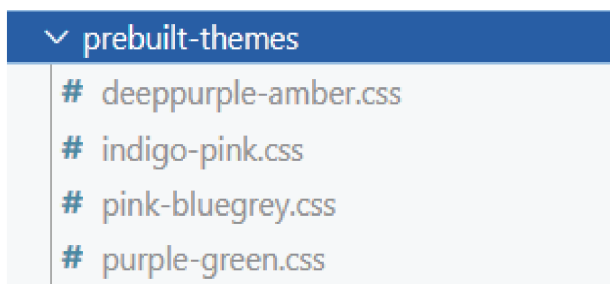
8.2. Prikaz i raspored elemenata GKI aplikacije

Sažetak: Sekcija dodaje elemente GKI kreiranoj aplikaciji. Najpre se opisuju predefinisane teme i animacije elemenata GKI koje dolaze sa instalacijom AM biblioteke. Zatim se dodaje podrška za ekrane osetljive na dodir. Nakon toga se kreiraju ikone sa simbolima, instalira *FlexLayout* biblioteka za raspored elemenata interfejsa i implementira forma za unos podataka.

8.2.1. Korišćenje predefinisanih tema i animacija

Teme su detaljnije opisane u Sekciji 5.3. Tema koja se želi primenjivati na sve komponente aplikacije se navodi u datoteci stila projekta (*styles.css*).

Može se koristiti postojeća, tj. predefinisana tema ili kreirati nova tema. Kreiranje teme je opisano u Sekciji 8.9.2. Definicije predefinisanih tema se mogu naći u direktorijumu *@angular/material/prebuilt-themes* kao na slici u nastavku.



Slika 8.11. Predefinisane teme koje dolaze sa instalacijom AM biblioteke.

AM omogućava korišćenje animacija kao suptilnih tranzicija elemenata GKI (kao što su pomeraji i promene boja, oblika i pozicija). Primer animacije je vizuelni efekat koncentričnog talasa (eng. *ripple effect*) koji se javlja klikom na AM dugme. Da bi se animacije primenjivale na elemente potrebno je registrovati modul koji ih implementira (*BrowserAnimationsModule*) u korenom modulu aplikacije (*app.module.ts*) kao u nastavku.

```
import { BrowserAnimationsModule } from '@angular/platform-browser/animations';

imports: [
    BrowserModule,
    MaterialModule,
    RoutingModule,
    BrowserAnimationsModule,
    FlexLayoutModule],
```

8.2.2. Podrška za ekrane osjetljive na dodir

Hammer.js je JavaScript biblioteka koja omogućava korišćenje aplikacije na ekranima osjetljivim na dodir. Više informacija u vezi sa bibliotekom se može naći u [89].

Biblioteka se instalira u *Terminal* prozoru razvojnog okruženja tako što se pozicionira u koreni direktorijum projekta i pokrene naredba u nastavku.

```
npm install hammerjs
```

Nakon instalacije biblioteku je potrebno registrovati u *main.ts* datoteci smeštenoj u *src* direktorijumu projekta.

```
import 'hammerjs';
```

8.2.3. Dodavanje ikona

Kolekcija GMD simbola koji predstavljaju tipične akcije GKI se nalazi u [84]. Simboli se mogu koristiti uspostavljanjem veze sa kolekcijom u *index.html* datoteci projekta. U zaglavlju datoteke (*head* tag) se navodi link ka kolekciji kao u nastavku.

```
<link href="https://fonts.googleapis.com/icon?family=Material+Icons" rel="stylesheet">
```

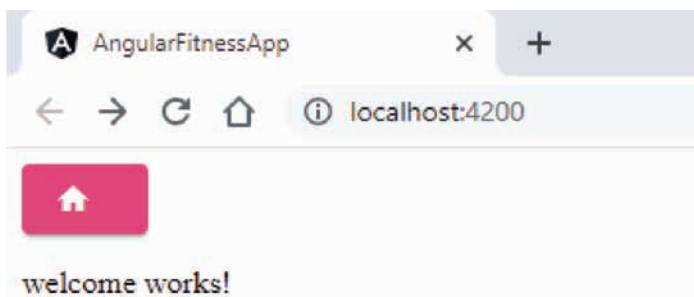
Na ovaj način se simboli ikona mogu koristiti na osnovu naziva u GMD kolekciji.

Dugme ikona je predstavljena kao modul *MatIconModule*. Radi korišćenja se mora registrovati u *material.module.ts*, slično dugmetu u Sekciji 8.1.5.

Dugme se dodaje u templejt korene komponente (*app.component.html*) kao u nastavku.

```
<section>
  <button mat-raised-button color="accent">
    <mat-icon>home</mat-icon>
  </button>
</section>
```

Menjati vrednost *color* atributa dugmeta sa nazivima paleta izabrane teme (*primary*, *warn*, *accent*) i pogledati rezultat u Veb pretraživaču uz pokrenut server. Rezultat pretrage GMD kolekcije ikona po ključnoj reči *home* daće simbol kao na slici u nastavku.



Slika 8.12. Dugme ikona u templejtu korene komponente aplikacije.

8.2.4. Instalacija FlexLayout biblioteke

Instalacija i korišćenje biblioteke su opisani u Sekciji 7.2. Biblioteka se instalira u okviru tekućeg projekta.

Zaustaviti razvojni server ukoliko je pokrenut i u *Terminal* prozoru pokrenuti sledeću naredbu.

```
npm install @angular/flex-layout --save
```

Instalacija se može videti u *package.json* datoteci projekta.

Nakon uspešno izvršene naredbe pokrenuti razvojni server. Ukoliko se pojave problemi prilikom instalacije ili tokom pokretanja servera, moguće je da instalirane verzije Angular biblioteka nisu usaglašene. U tom slučaju se verzije instaliranih biblioteka ažuriraju naredbom u nastavku.

```
ng update --all
```

Funkcionalnosti su implementirane kao modul (*FlexLayoutModule*) koji se radi radi korišćenja mora registrovati u korenom modulu projekta (*app.module.ts*).

Napomena: U trenutku pisanja udžbenika postoje najave da Angular u budućnosti možda neće podržavati novije verzije FlexLayout biblioteke. Korišćenje je opisano kao alternativa FlexBox biblioteke. U slučaju da čitalac kreira projekat u vreme kada FlexLayout nije podržan, nadalje se mogu koristiti FlexBox naredbe rasporeda. Naredbe se navode u datoteci stila odgovarajuće komponente umesto u njenom templejtu (videti Sekciju 7.1).

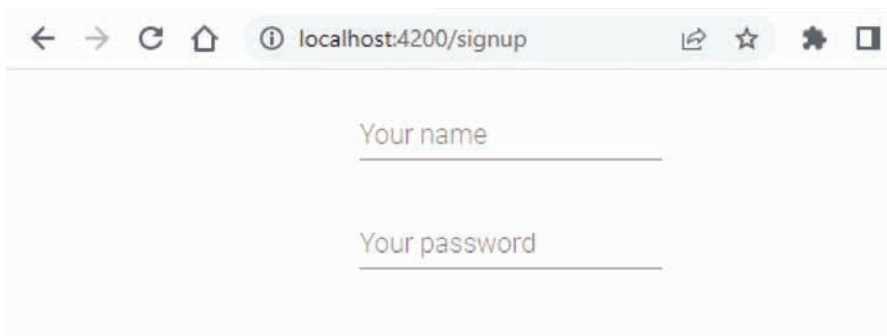
8.2.5. Dodavanje forme

Forma se dodaje u templejt komponente za registraciju korisnika aplikacije (*signup.component.html*). Struktura i korišćenje forme su opisani u Sekciji 6.10. Forma sadrži polja za korisničko ime i lozinku, kao u nastavku.

```
<section>
  <form>
    <mat-form-field>
      <input type="email" matInput placeholder="Your name">
    </mat-form-field>
    <mat-form-field>
      <input type="password" matInput placeholder="Your password">
    </mat-form-field>
  </form>
</section>
```

Sadržaj forme se orijentiše vertikalno (*fxLayout*) i centrira po visini i širini templejta (*fxLayoutAlign*) kao u kodu u nastavku.

```
<section>
  <form fxLayout="column" fxLayoutAlign="center center">
    <mat-form-field>
      <input type="email" matInput placeholder="Your name">
    </mat-form-field>
    <mat-form-field>
      <input type="password" matInput placeholder="Your password">
    </mat-form-field>
  </form>
</section>
```



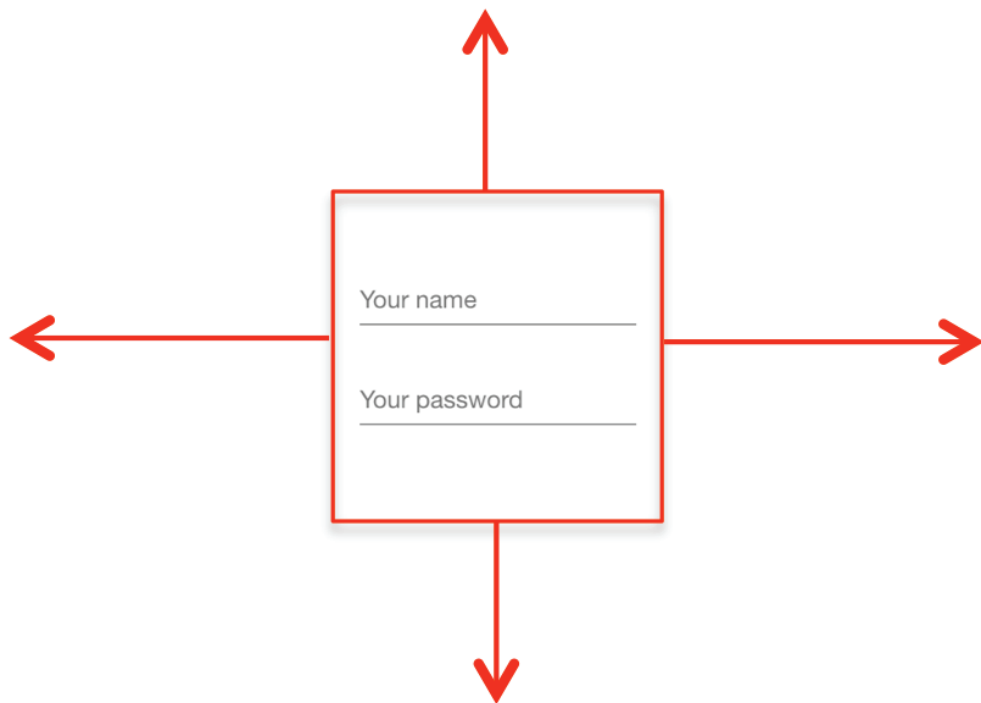
Direktiva *fxLayoutGap* definiše prostor između elemenata dece roditelja (*form* tag). Detalji korišćenja direktive mogu se videti u Sekciji 7.2.1. Probati

direktivu sa različitim jedinicama mere i vrednostima i videti kako se menja rastojanje između polja forme.

U datoteci stila komponente (*signup.component.css*) definiše se izgled *form* elementa kao u kodu ispod.

```
form {  
    height: 500px;  
    border: 1px solid ■ #000000;  
}
```

Sada je potrebno kombinovati različite vrednosti *fxLayoutAlign* direktive i videti kako se pozicija forme menja na osnovu navedenih vrednosti, kao što je šematski prikazano na Slici 8.13. Vrednosti su opisane u Sekciji 7.2.1.



Slika 8.13. Promena pozicije forme korišćenjem *fxLayoutAlign* direktive.

8.3. Uređivanje forme

Sažetak: Sekcija uređuje formu za registraciju korisnika aplikacije. Dodaje se dugme za slanje podataka forme, implementira provera, čuvanje i pristup unetim podacima, kreiraju kalendar i polje za potvrdu, i definiše raspored elemenata forme.

8.3.1. Dugme za slanje podataka forme

Dugme se dodaje u templejt komponente za registraciju korisnika (*signup.component.html*) u nastavku polja forme.

```
<section>
  <form fxLayout="column" fxLayoutAlign="center center">
    <mat-form-field>
      <input type="email" matInput placeholder="Your name">
    </mat-form-field>
    <mat-form-field>
      <input type="password" matInput placeholder="Your password">
    </mat-form-field>
    <button type="submit" mat-raised-button color="primary">Submit</button>
  </form>
</section>
```

Modul koji implementira formu (*FormsModule*) je potrebno registrovati u korenom modulu aplikacije (*app.module.ts*), kao što je opisano u Sekciji 6.10. Na ovaj način forma kao element GKI postaje dostupna svim komponentama aplikacije i mogu se koristiti funkcionalnosti forme vezane za pristup i upravljanje podacima.

8.3.2. Pristup i korišćenje podataka forme

Primer u nastavku je opisan u Sekciji 6.10. Lokalna varijabla tipa objekta forme (*#f = ngForm*) se prosleđuje metodi osluškivača (*onSubmit()*) događaja slanja podataka forme (*ngSubmit*). Da bi se sadržaju polja forme moglo pristupiti iz kontrolera, ono se mora proglasiti delom modela forme (*ngModel*) i imati naziv (*name*).

```

<section>
  <form fxLayout="column" fxLayoutAlign="center center" #f="ngForm" (ngSubmit)="onSubmit(f)">
    <mat-form-field>
      <input type="email" matInput placeholder="Your name" ngModel name="email">
    </mat-form-field>
    <mat-form-field>
      <input type="password" matInput placeholder="Your password" ngModel name="password">
    </mat-form-field>
    <button type="submit" mat-raised-button color="primary">Submit</button>
  </form>
</section>

```

Kontroler komponente za registraciju korisnika implementira metodu osluškivača koja prihvata i ispisuje sadržaj objekta forme (*form*).

```

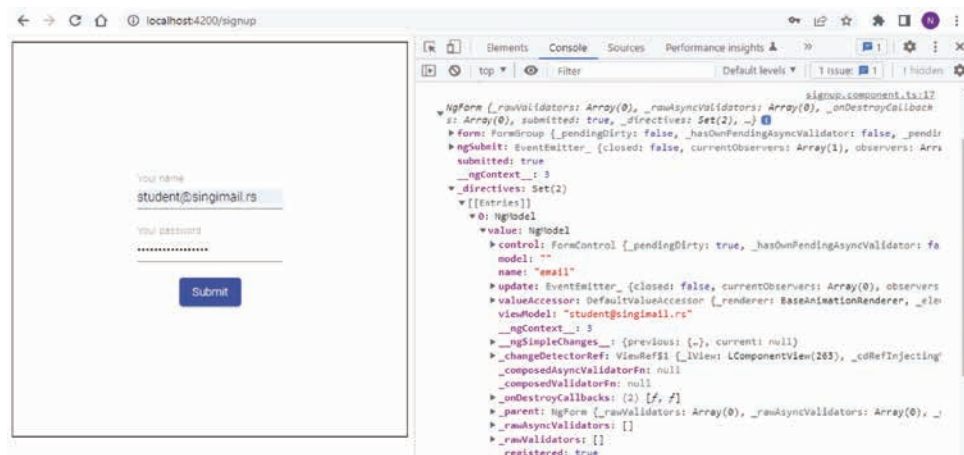
import { NgForm } from '@angular/forms';

onSubmit(form: NgForm){
  console.log(form);
}

```

Metoda ispisuje sadržaj objekta forme u konzoli prozora Veb pretraživača (*Developer View*). Potrebno je uneti vrednosti u polja forme, kliknuti da dugme za slanje podataka forme, i pronaći vrednosti u *NgForm* objektu ispisanom u konzoli. Izabрати *NgForm* -> *_directives* -> *NgModel* -> *value* (...) kao na slici u nastavku.

Objekti tipa *NgForm* i *NgModel* se kreiraju kao rezultat izvršavanja direktiva *ngForm* i *ngModel* u formi templejta komponente. *NgForm* se definiše za formu kao celinu, dok se *ngModel* navodi za svako polje forme kojem se želi pristupiti programski iz kontrolera komponente.



Slika 8.14. Ispisivanje sadržaja forme u komponenti za registraciju korisnika.

U nastavku je prikazana tipična obrada podataka forme, poput smernica (eng. *hints*) za pravilan unos podataka korisnika, obrade grešaka i validacije unosa.

8.3.3. Smernice za korišćenje polja forme

Smernice su uputstva, tj. informacije u vezi sa pravilnim unosom podataka u polje forme. U primeru u nastavku se dodaje smernica kao tekst ili labela (*hintLabel* atribut) koja govori da korisnik mora uneti najmanje 8 karaktera lozinke. Ovo podrazumeva da je u polju forme specificirano odgovarajuće ograničenje (*minLength*). Tag *mat-hint* ispisuje broj unetih karaktera u toku unosa korisnika. Trenutni broj karaktera se računa u okviru izraza (interpolacije) tako što je prethodno definisana varijabla *passInput* koja čuva vrednost polja lozinke (*value*). Izraz najpre proverava da u polju postoji sadržaj (?) i zatim prikazuje broj trenutno unetih karaktera (*length*).

```
<mat-form-field hintLabel="Please enter minimum 8 characters.">
  <input type="password" matInput placeholder="Your password" ngModel name="password"
    required minlength="8" #passInput="ngModel">
  <mat-hint align="end">{{ passInput.value?.length }} / 8</mat-hint>
</mat-form-field>
```

U datoteci stila komponente za registraciju korisnika (*signup.component.css*) se definiše širina polja forme tako da je tekst smernice ispisan u jednom redu, kao u primeru ispod.

```
mat-form-field {
  width: 300px;
}
```



The screenshot shows a web form with two input fields. The first field is labeled 'Your name' and contains the text 'student@singimail.rs'. The second field is labeled 'Your password *' and contains masked characters '.....'. Below the password field, a hint message 'Please enter minimum 8 characters.' is displayed, followed by a character count '10 / 8'. A blue 'Submit' button is located at the bottom of the form.

8.3.4. Obrada grešaka i validacija unosa u polje forme

U slučaju da korisnik ne unese korisničko ime ili unese podatke u neispravnom formatu potrebno je prikazati odgovarajuću poruku greške.

Lokalna varijabla *emailInput* čuva sadržaj polja za unos korisničkog imena u formatu adrese elektronske pošte. Svaki *mat-error* tag vrši proveru tipa greške tako što ispituje vrednost varijable i ispisuje odgovarajuću poruku. Provera se vrši ispitivanjem uslova pomoću *ngIf* strukturne direktive. Korišćenje strukturnih direktiva je opisano u Sekciji 5.3.4.1.

```
<mat-form-field>
  <input type="email" matInput placeholder="Your email" ngModel name="email" email
  required #emailInput="ngModel">
  <mat-error *ngIf="emailInput.hasError('required')">Field should not be empty</mat-error>
  <mat-error *ngIf="!emailInput.hasError('required')">Please use the correct email format.
  </mat-error>
</mat-form-field>
```

8.3.5. Dodavanje kalendara

Korišćenje kalendara je opisano u Sekciji 6.11. Svaki kalendar čine tekstualno polje za unos datuma (*input*), objekat kalendar (*mat-datepicker*), i ikona koja prikazuje ili sakriva objekat kalendara (*mat-datepicker-toggle*). Tekstualno polje i objekat kalendar se povezuju pomoću *matDatePicker* direktive atributa. Lokalna varijabla (*picker*) čuva vrednost objekta kalendara (izabrani datum). Varijabla se prosleđuje kao vrednost polju za unos i prikaz datuma (*input*) i ikoni kalendara (*mat-datepicker-toggle*). *MatSuffix* direktiva ukazuje da se ikona prikazuje nakon sadržaja datuma.

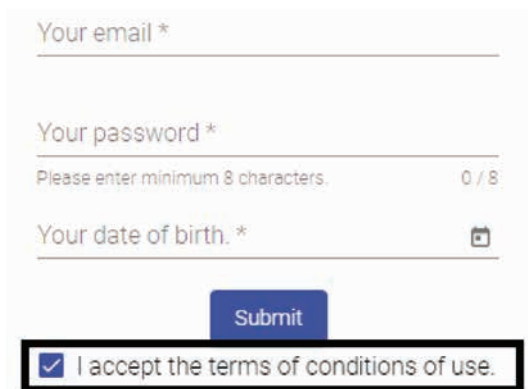
```
<mat-form-field>
  <input matInput placeholder="Your date of birth." [matDatePicker]="picker" ngModel name="birthDate" required>
  <mat-datepicker-toggle matSuffix [for]="picker"></mat-datepicker-toggle>
  <mat-datepicker #picker></mat-datepicker>
</mat-form-field>
```

The screenshot displays a form with three input fields. The first field is labeled 'Your email *' and contains the text 'student@singimail.rs'. The second field is labeled 'Your password *' and shows masked characters '.....'. Below the password field is a feedback message: 'Please enter minimum 8 characters.' followed by a character count '10 / 8'. The third field is labeled 'Your date of birth *' and contains the date '1/3/2023'. To the right of the date is a calendar icon. A blue 'Submit' button is located at the bottom of the form.

8.3.6. Korišćenje polja za potvrdu

Polje za potvrdu služi davanju saglasnosti korisnika sa uslovima korišćenja aplikacije. Korišćenje polja za potvrdu je opisano u Sekciji 6.9. Polje se dodaje u okviru forme, ispod dugmeta.

```
<mat-checkbox color="primary" ngModel name="acceptance" required>  
  I accept the terms of conditions of use.  
</mat-checkbox>
```



8.3.7. Dodatne funkcije forme

Potrebno je povećati vertikalni razmak između polja forme. Razmak se može povećati korišćenjem *fxLayoutGap* direktive za roditeljski element formu (videti Sekciju 7.2.1). Lokalna varijabla *f* čuva tekući objekat ili sadržaj forme. Varijable se koristi za proveru validnosti unetih podataka forme. Atribut *invalid* je logičkog tipa i govori da li je sadržaj validan. Validnost se utvrđuje na osnovu pravila za unos podataka polja forme definisanih u Sekcijama 8.3.3. i 8.3.4. Na primer, ako su vrednosti polja forme unete u skladu se ograničenjima i u odgovarajućem formatu, atribut ima vrednost *false*. U suprotnom dobija vrednost *true* i dugme je onemogućeno (*disabled* atribut). Atribut *disabled* dugmeta se povezuje sa vrednošću atributa *invalid* korišćenjem direktive atributa (videti Sekciju 5.3.4.2).

```
<form fxLayout="column" fxLayoutAlign="center center" fxLayoutGap="15px"  
  #f="ngForm" (ngSubmit)="onSubmit(f)">  
  
  <button type="submit" mat-raised-button color="primary" [disabled]="f.invalid">Submit</button>
```

The image shows a registration form with the following elements:

- Your email ***: A text input field containing the email address `student@singimail.rs`.
- Your password ***: A password input field with masked characters (dots). Below the field, there is a feedback message: "Please enter minimum 8 characters." and a character count "10 / 8".
- Your date of birth. ***: A date selection field with a calendar icon.
- Submit**: A light gray button with the text "Submit".
- Terms of Use**: A checkbox followed by the text "I accept the terms of conditions of use."

Zadatak: Na osnovu primera forme komponente za registraciju korisnika, samostalno implementirati komponentu za prijavu korisnika (*login*). Komponenta sadrži sledeća polja i funkcionalnosti:

1. Polja za unos korisničke šifre i lozinke,
2. Validacija sadržaja polja kao u primeru,
3. Prikaz grešaka i smernica kao u primeru, i
4. Dugme za slanje podataka forme (*submit*) koje je onemogućeno ukoliko sadržaj forme nije validan.

Izgled komponente je prikazan na slici u nastavku.

The image shows a login form with the following elements:

- Your email ***: A text input field.
- Your password ***: A password input field. Below the field, there is a feedback message: "Please enter minimum 8 characters." and a character count "0 / 8".
- Submit**: A light gray button with the text "Submit".

Slika 8.15. Izgled forme komponente za prijavu korisnika aplikacije.

8.4. Dodavanje navigacije

Sažetak: Sekcija dodaje i uređuje elemente navigacije GKI u korenoj komponenti aplikacije, uključujući vertikalnu liniju alata (eng. *sidenav*) i horizontalnu liniju alata (eng. *toolbar*). Elementi se kombinuju tako da se izgled GKI može prilagoditi dimenzijama prozora Veb pretraživača.

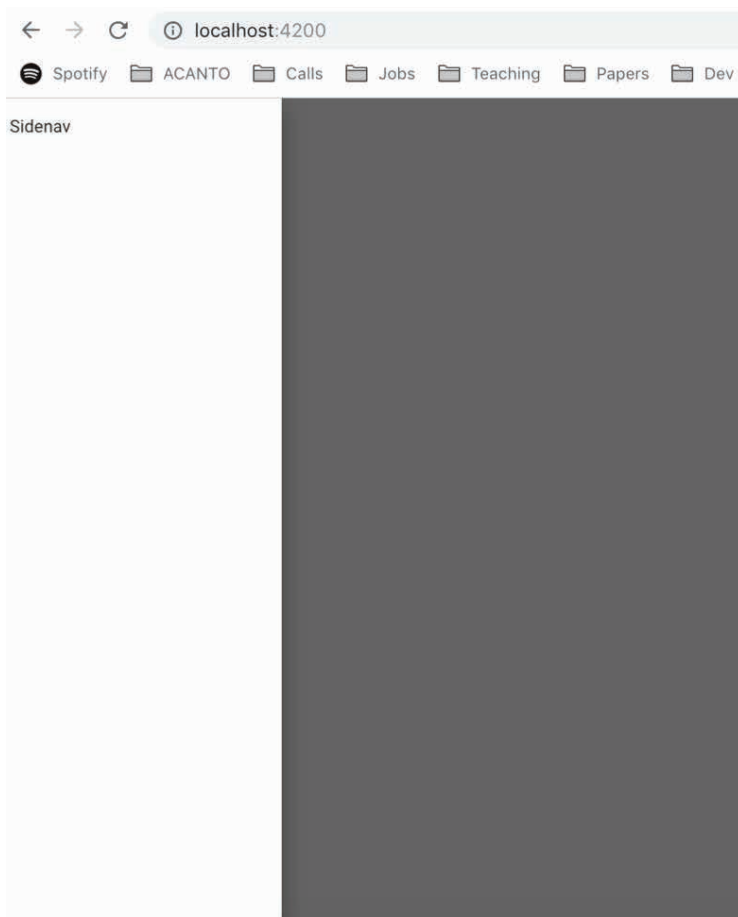
8.4.1. Vertikalna linija alata za navigaciju

Korišćenje AM komponente vertikalna linija alata je opisano u Sekciji 6.12. Vertikalna linija alata služi navigaciji, tj. omogućava pristup komponentama aplikacije iz GKI. Zbog toga se smešta u templejt korene komponente (*app.component.html*). Postojeći sadržaj templejta, tj. *router-outlet* tag se navodi u delu u kojem će se prikazivati sadržaj linije kao komponenta do koje vodi tekuća ruta.

```
<mat-sidenav-container>
  <mat-sidenav #sidenav opened="true">
    <p>Sidenav</p>
  </mat-sidenav>

  <mat-sidenav-content>
    <router-outlet></router-outlet>
  </mat-sidenav-content>
</mat-sidenav-container>
```

Slika 8.16. prikazuje izgled templejta korene komponente u pretraživaču. Vertikalna linija za navigaciju je inicijalno prikazana. Kada se klikne na bilo koji deo stranice van linije, ona se više ne prikazuje.



Slika 8.16. Početni izgled vertikalne linije za navigaciju aplikacije.

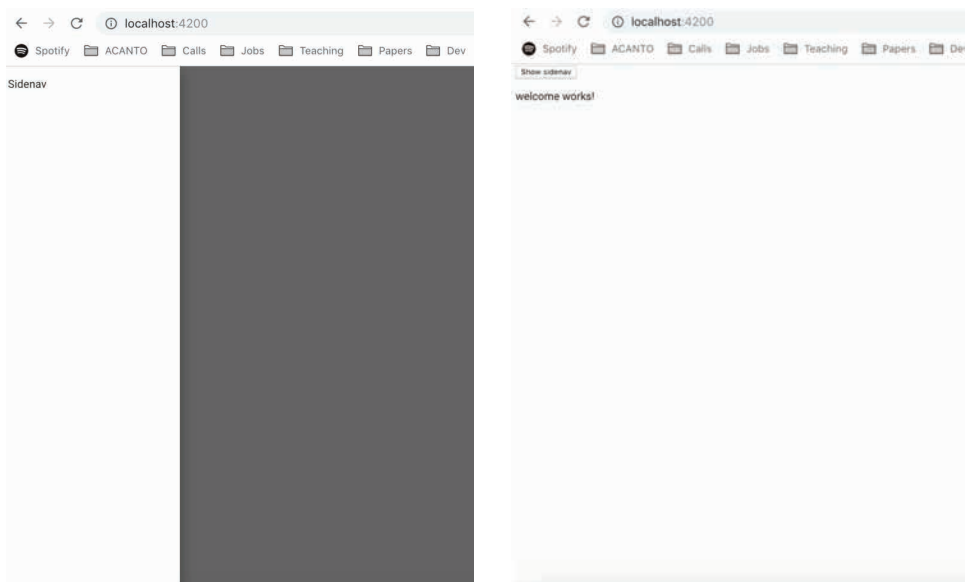
Lokalna varijabla *sidenav* ukazuje na zaglavlje vertikalne linije. U sadržaj linije se dodaje dugme koje osluškuje događaj klika (*click*) i poziva ugrađenu metodu linije iz definisane varijable (*sidenav.toggle()*). Metoda prikazuje liniju ako je sakrivena i obrnuto.

```

<mat-sidenav-container>
  <mat-sidenav #sidenav opened="true">
    <p>Sidenav</p>
  </mat-sidenav>
  <mat-sidenav-content>
    <button (click)="sidenav.toggle()"> Show sidenav </button>
    <router-outlet></router-outlet>
  </mat-sidenav-content>
</mat-sidenav-container>

```

Slika 8.17. ilustruje izgled i ponašanje linije nakon izvršenih promena.



Slika 8.17. Vertikalna linija za navigaciju. Klikom na dugme linija se naizmenično prikazuje i sakriva.

Primer ilustruje tipičan scenario u koje se ponašanje AM komponente može kontrolisati putem postojećih metoda. Svaka AM komponenta ima skup ugrađenih atributa i metoda koje se mogu pronaći u API sekciji opisa. Obrazac korišćenja ovih metoda se može definisati na sledeći način:

1. Definisati lokalnu varijablu koja ukazuje na instancu komponente. Ime lokalne varijable uvek počinje simbolom #.
2. Promene nastaju kao rezultat akcije korisnika, te se u trenutku nastanka odgovarajućeg događaja pristupa varijabli i poziva željena metoda. U primeru iznad, na događaj klika dugmeta se otvara ili zatvara vertikalna linija alata.

U nastavku se vertikalna linija alata kombinuje sa horizontalnom linijom alata.

8.4.2. Horizontalna linija alata za navigaciju

Korišćenje AM komponente horizontalna linija alata je opisano u Sekciji 6.13. Horizontalna linija alata se dodaje kao sadržaj komponente vertikalne linije alata (*mat-sidenav-content* tag). U horizontalnu liniju se kao AM dugme smešta postojeće dugme koje kontroliše prikaz vertikalne linije i dodeljuje mu se nov naziv *Toogle sidenav*. Zatim se rezerviše prostor za logo aplikacije. Na kraju se dodaju linkovi ka komponentama aplikacije koji odgovaraju prethodno definisanim rutama (videti Sekciju 8.1.8).

```
<mat-sidenav-content>
  <mat-toolbar color="primary">
    <div>
      <button mat-button (click)="sidenav.toggle()">Toggle sidenav</button>
    </div>
    <div>Logo</div>
    <div>
      <a routerLink="/signup">Signup</a>
      <a routerLink="/login">Login</a>
      <a routerLink="/training">Training</a>
    </div>
  </mat-toolbar>
  <router-outlet></router-outlet>
</mat-sidenav-content>
```

Slika 8.18. prikazuje deo korene komponente nakon promena. Klikom na dugme *Toogle sidenav* vertikalna linija se prikazuje (desno) dok se klikom na bilo koji deo stranice van vertikalne linije ona sakriva (levo).



Slika 8.18. Horizontalna linija alata sa linkovima ka komponentama aplikacije i dugmetom za kontrolu prikaza vertikalne linije alata.

8.4.3. Uređivanje izgleda navigacije

Stil vertikalne linije alata se uređuje u datoteci stila korene komponente (*app.component.css*). Linija, tj. njeni sastavni delovi zauzimaju raspoloživ prostor po visini stranice i širine je 250 piksela.

```
mat-sidenav-container, mat-sidenav-content, mat-sidenav {
  height: 100%;
}

mat-sidenav {
  width: 250px;
}
```

U datoteci stila aplikacije (*styles.css*) se definiše vrsta fonta i visina za kontejnerske HTML tagove.

```
html, body {
  font-family: Roboto;
  height: 100%;
}
```

Navedeni font se mora registrovati na nivou aplikacije (*head* tag *index.html* datoteke) kako bi njegova definicija bila dostupna. Na ovaj način svaka komponenta koja se prikazuje kroz *index.html* stranicu može koristiti font (videti Sekciju 8.1.8, Slika 8.10).

```
<link href="https://fonts.googleapis.com/css?family=Roboto" rel="stylesheet">
```

Vertikalna linija alata je inicijalno prikazana. Klik na deo površine stranice van linije zatvara, tj. sakriva liniju. Ukoliko se zatim klikne na neki od linkova prikazuje se komponenta do koje vodi odgovarajuća ruta. Klik na dugme *Toggle sidenav* prikazuje vertikalnu liniju alata.

Horizontalna linija alata se modifikuje tako da se umesto tekstualnog dugmeta koristi dugme ikona za prikaz vertikalne linije alata. Naziv ikone mora biti identičan nazivu izabranog simbola u GMD kolekciji ikona [84].

```

<mat-toolbar color="primary">
  <div>
    <button mat-button (click)="sidenav.toggle()">
      <mat-icon>menu</mat-icon>
    </button>
  </div>
  <div>Logo</div>
  <div>
    <a routerLink="/signup">Signup</a>
    <a routerLink="/login">Login</a>
    <a routerLink="/training">Training</a>
  </div>
</mat-toolbar>

```

U datoteci stila korene komponente (*app.component.css*) se definišu stilovi za linkove ka komponentama. Definisani stilovi će biti primenjeni na sve komponente sa obzirom da se one prikazuju kao *router-outlet* tag u templejtu korene komponente.

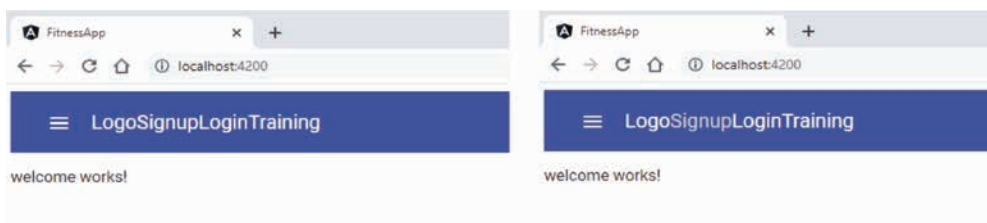
```

a {
  color: #ffffff;
  text-decoration: none;
}

a:hover, a:active {
  color: #d0cece;
}

```

Nakon izvršenih promena korena komponenta će izgledati kao na Slici 8.19.



Slika 8.19. Izgled navigacije u korenoj komponenti aplikacije.

U nastavku se logo pretvara u link ka komponenti početnoj stranici aplikacije (*welcome*).

```
<mat-toolbar color="primary">
  <div>
    <button mat-button (click)="sidenav.toggle()">
      <mat-icon>menu</mat-icon>
    </button>

    <div>
      <a routerLink="/">Logo</a>
    </div>
  </div>
```

Na kraju je potrebno urediti izgled linkova ka komponentama aplikacije, tj. poravnati ih po desnoj strani interfejsa i dodati prostor između njih.

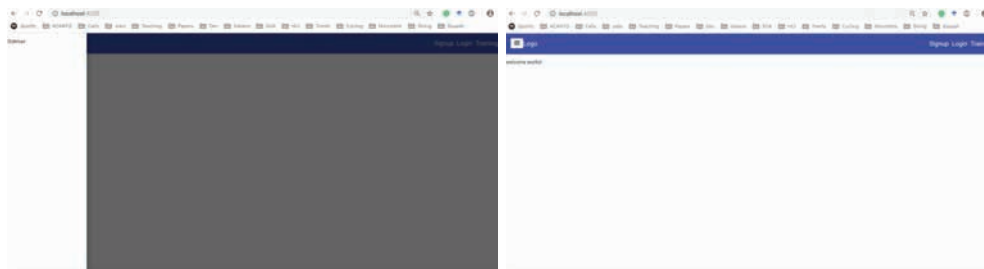
Flex direktiva definiše pravila za promenu veličine elementa deteta u okviru elementa roditelja. Korišćenje direktive je opisano u Sekciji 7.2.2. U konkretnom slučaju horizontalna linija alata (*mat-toolbar*) je roditelj koji sadrži tri deteta - *div* element koji sadrži dugme ikonu, *div* element logo i *div* element za linkove ka komponentama aplikacije u kojem se navodi direktiva.

FlexAlign direktiva definiše ravnanje elemenata dece u elementu roditelju (videti Sekciju 7.2.1). Vrednost *end* znači da će elementi biti poravnati uz desnu stranu interfejsa.

Horizontalan prazan prostor između elemenata dece, tj. linkova se dodaje pomoću direktive *FlexGap*.

```
<div fxFlex fxLayoutAlign="end" fxLayoutGap="10px">
  <a routerLink="/signup">Signup</a>
  <a routerLink="/login">Login</a>
  <a routerLink="/training">Training</a>
</div>
```

Izgled korene komponente nakon izvršenih promena je prikazan na Slici 8.20.



Slika 8.20. Konačan izgled navigacije u korenoj komponenti aplikacije.

Zadatak: Dodati i menjati vrednosti *fxFlex* direktive za sve *div* elemente horizontalne linije alata i videti promene izgleda i razmeštaja elemenata dece.

8.5. Prilagođavanje interfejsa različitim dimenzijama prikaza

Sažetak: Sekcija implementira izgled aplikacije koji se prilagođava različitim dimenzijama, tj. širinama prozora pretraživača. Pored toga, dodaju se element za organizaciju delova treninga (višestruki pogledi), navigaciona lista u vertikalnu liniju alata i kartica za sadržaj komponente za kreiranje novog treninga.

8.5.1. Dodavanje podrške za različite veličine ekrana

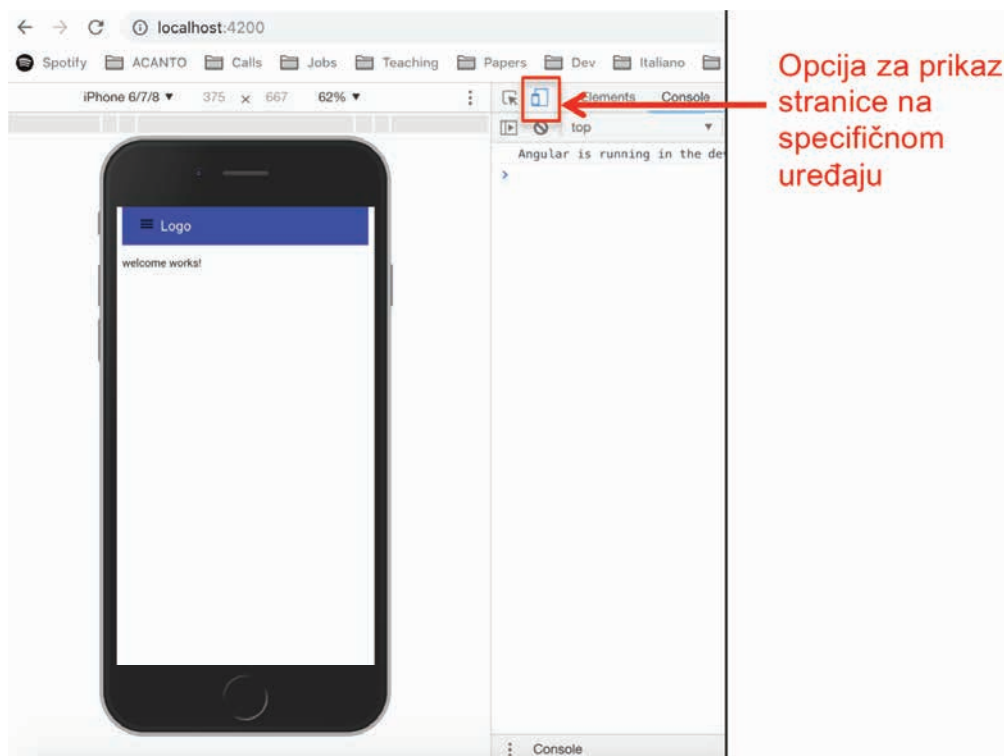
Podrška se svodi na sakrivanje grupe elemenata za konkretnu vrednost širine prozora pretraživača. Odgovarajuće pravilo, tj. direktiva se definiše u roditeljskom elementu grupe koja se prikazuje za širine veće od granične vrednosti. Primer u nastavku ilustruje primenu pravila.

Direktiva *fxHide.xs* se navodi u *div* elementu koji grupiše linkove ka komponentama u horizontalnoj liniji alata. Direktiva sadrži ključnu reč *fxHide* kao naredbu koja sakriva element (ili tag) u kojem je navedena, i simboličku

konstantu `xs` koja označava vrednost od 600 piksela. Rezultat naredbe formulisane na ovaj način je da se dati element neće prikazivati za širinu prikaza, tj. komponente manju od 600 piksela.

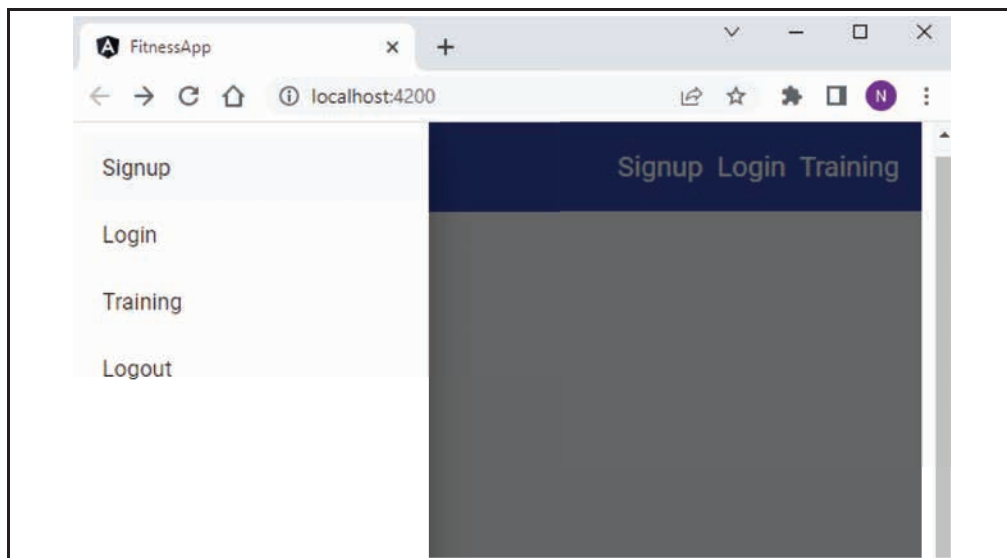
```
<div fxFlex fxLayoutAlign="end" fxLayoutGap="10px" fxHide.xs>
  <a routerLink="/signup">Signup</a>
  <a routerLink="/login">Login</a>
  <a routerLink="/training">Training</a>
</div>
```

Slika u nastavku daje slučaj širine prikaza manje od 600 piksela.



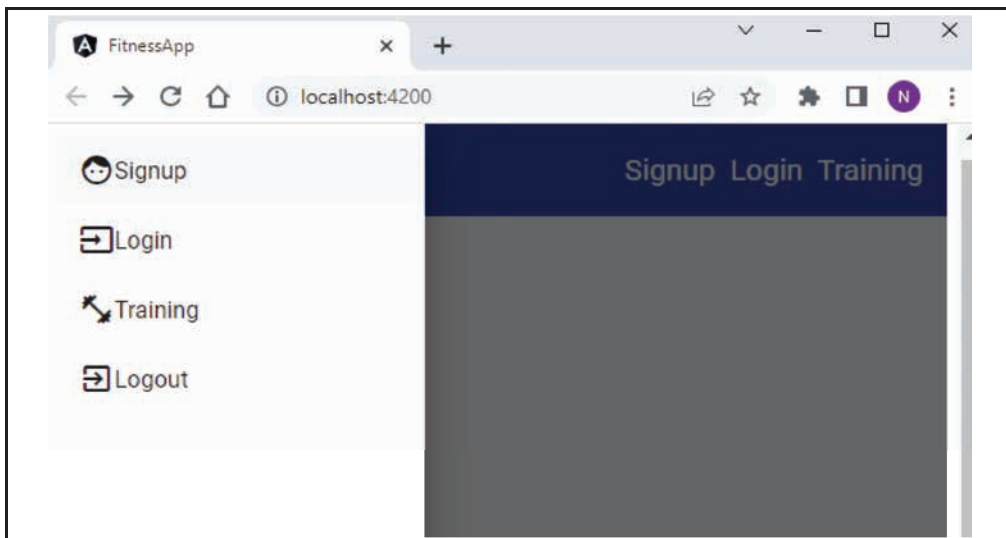
Slika 8.21. Sakrivanje linkova ka komponentama u horizontalnoj liniji alata za prikaze širine manje od 600 piksela.

Na sličan način se može definisati da se ikona za otvaranje vertikalne linije alata vidi samo za manje širine prikaza. U primeru u nastavku ikona će biti sakrivena na prikazima širim od 600 piksela. Više informacija u vezi sa prilagodljivim dizajnom se može u naći u [90].



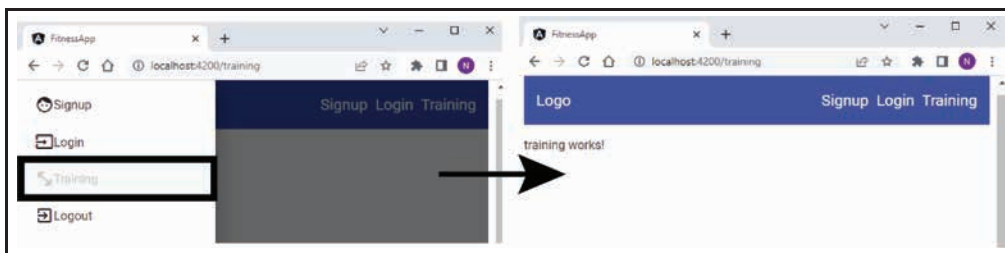
Svakom elementu, tj. nazivu linka navigacione liste dodaje se simbol ikone. Koristiti nazive simbola iz GMD kolekcije [84] (*face*, *input*, *fitness_center* i *exit_to_app* u primeru koji sledi).

```
<mat-nav-list>
  <a mat-list-item routerLink="/signup">
    <mat-icon>face</mat-icon>
    Signup
  </a>
  <a mat-list-item routerLink="/login">
    <mat-icon>input</mat-icon>
    Login
  </a>
  <a mat-list-item routerLink="/training">
    <mat-icon>fitness_center</mat-icon>
    Training
  </a>
  <a mat-list-item routerLink="/">
    <mat-icon>exit_to_app</mat-icon>
    Logout
  </a>
</mat-nav-list>
```



Klikom na link liste u pozadini se prikazuje sadržaj odgovarajuće komponente. Sadržaj se vidi delimično zbog toga što vertikalna linija alata ostaje prikazana. Metoda `close()` varijable `sidenav` koja ukazuje na zaglavlje vertikalne linije sakriva liniju i izvršava se na događaj klika na link (`click`).

```
<mat-nav-list>
  <a mat-list-item routerLink="/signup" (click)="sidenav.close()">
    <mat-icon>face</mat-icon>
    Signup
  </a>
  <a mat-list-item routerLink="/login" (click)="sidenav.close()">
    <mat-icon>input</mat-icon>
    Login
  </a>
  <a mat-list-item routerLink="/training" (click)="sidenav.close()">
    <mat-icon>fitness_center</mat-icon>
    Training
  </a>
  <a mat-list-item routerLink="/" (click)="sidenav.close()">
    <mat-icon>exit_to_app</mat-icon>
    Logout
  </a>
</mat-nav-list>
```



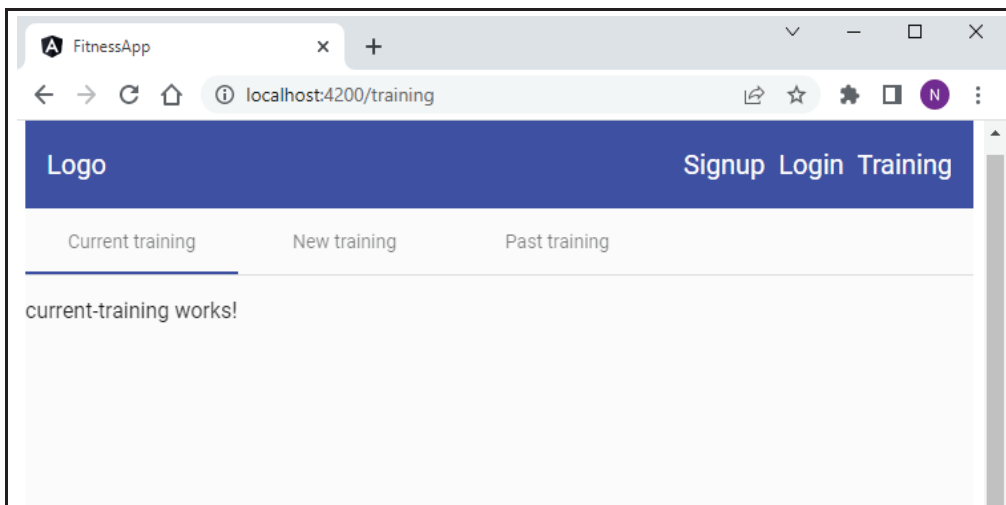
Kada je širina prikaza veća 600 piksela aplikacija će prikazati samo horizontalnu liniju alata, u suprotnom će biti prikazana vertikalna linija alata.

8.5.3. Dodavanje višestrukih pogleda

AM komponenta se koristi za implementaciju navigacije na nivou komponente treninga, tj. kretanje između prethodnog, tekućeg i novog treninga. Korišćenje komponente je opisano u Sekciji 6.15.

U tri odvojena pogleda (*mat-tab*) se smeštaju odgovarajuće komponente, tj. navode njihovi selektori kao tagovi identifikatori. Svakom pogledu se dodeljuje i naslov (*label*). Primer u nastavku dodaje poglede u templejt komponente treninga (*training.component.html*).

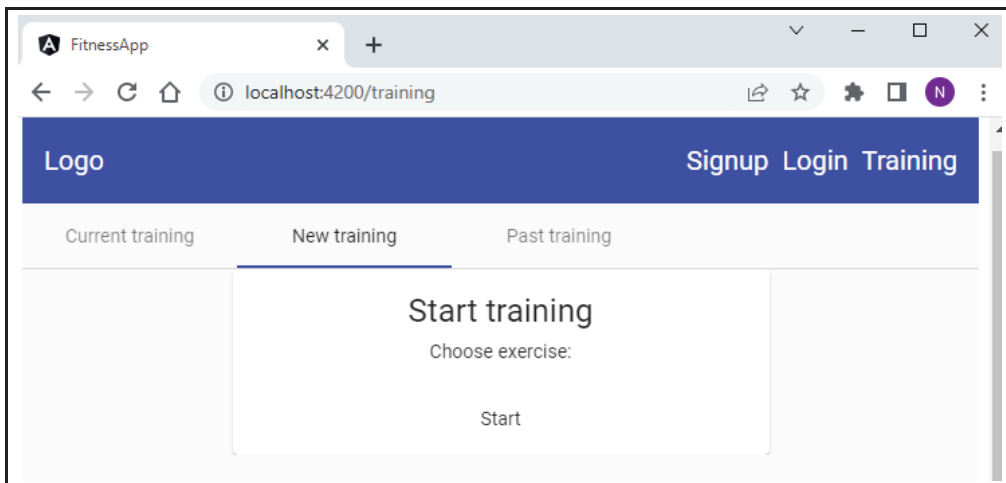
```
<mat-tab-group>
  <mat-tab label="Current training">
    <app-current-training></app-current-training>
  </mat-tab>
  <mat-tab label="New training">
    <app-new-training></app-new-training>
  </mat-tab>
  <mat-tab label="Past training">
    <app-previous-training></app-previous-training>
  </mat-tab>
</mat-tab-group>
```



8.5.4. Dodavanje kartice

Trening korisnika se kreira kao kartica sa naslovom, sadržajem i akcijama. Korišćenje kartice je opisano u Sekciji 6.16. Kartica će omogućiti izbor vežbe i pokretanje treninga. U templatejtu komponente za kreiranje novog treninga (*new-training.component.html*) definisati početnu verziju kartice novog treninga kao u nastavku.

```
<section fxLayout fxLayoutAlign="center">
  <mat-card fxFlex="400px">
    <mat-card-title fxLayoutAlign="center">
      Start training
    </mat-card-title>
    <mat-card-content fxLayoutAlign="center">
      Choose exercise:
    </mat-card-content>
    <mat-card-actions fxLayoutAlign="center">
      <button type="submit" mat-button>Start</button>
    </mat-card-actions>
  </mat-card>
</section>
```



8.6. Razmena podataka između komponenti

Sažetak: Sekcija uređuje komponente treninga. Prvo se dodaje padajući za izbor vežbe treninga. Zatim se povezuju komponenta treninga i komponenta novog treninga tako da razmenjuju podatke. Na ovaj način se može kreirati trening i pratiti status (ili progres) vežbe. Na kraju se dodaje element progres koji prikazuje status treninga u toku, zajedno sa pratećom pozadinskom logikom.

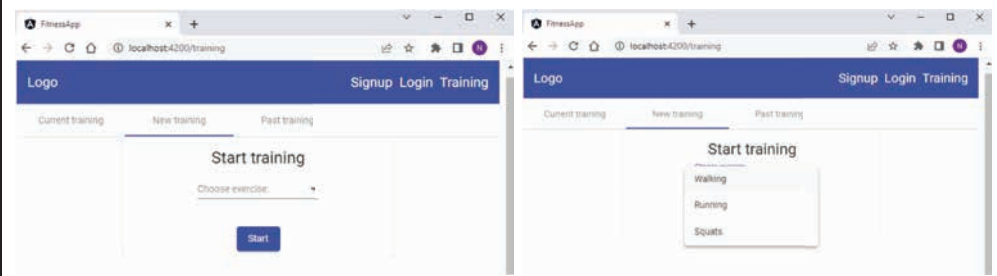
8.6.1. Dodavanje padajuće liste

Korišćenje padajuće liste je opisano u Sekciji 6.8. Lista se dodaje kao spisak vežbi potreban za kreiranje novog treninga. tipova vežbi u okviru kartice novog treninga. Lista se smešta u odeljak sadržaja kartice (*mat-card-content* tag) u templejtu komponente novog treninga (*new-training.component.html*). U odeljku akcija kartice (*mat-card-content* tag) se navodi dugme koje pokreće izabranu vežbu, tj. trening.

```

<mat-card-content fxLayoutAlign="center" fxLayoutGap="20px">
  Choose exercise:
  <mat-form-field>
    <mat-select placeholder="Choose training:">
      <mat-option value="walking">
        Walking
      </mat-option>
      <mat-option value="running">
        Running
      </mat-option>
      <mat-option value="squats">
        Squats
      </mat-option>
    </mat-select>
  </mat-form-field>
</mat-card-content>
<mat-card-actions fxLayoutAlign="center">
  <button type="submit" mat-raised-button color="primary">Start</button>
</mat-card-actions>

```



8.6.2. Povezivanje komponenti treninga

Pristup glavnoj komponenti treninga daje prikaz delova novog i prethodnog treninga. Kada se u novom treningu izabere vežba i trening pokrene klikom na dugme *Start*, vidi se samo trening u toku, tj. tekući trening.

Templejt glavne komponente treninga (*training.component.html*) se modifikuje tako da se komponenta tekućeg treninga prikazuje van višestrukih pogleda (*mat-tab-group* tag).


```

<mat-tab-group>
  <mat-tab label="New training">
    <app-new-training></app-new-training>
  </mat-tab>
  <mat-tab label="Past training">
    <app-previous-training></app-previous-training>
  </mat-tab>
</mat-tab-group>
<app-current-training></app-current-training>

```

U kontroler trening komponente (*training.component.ts*) se dodaje atribut *ongoingTraining*, kao u nastavku. Atribut govori da li postoji trening u toku, tj. tekući trening.

```

export class TrainingComponent implements OnInit {
  ongoingTraining = false;

  constructor() {}

  ngOnInit(): void {}
}

```

Kontroleru komponente novog treninga (*new-training.component.ts*) dodaje se atribut kao programski izvor događaja (*trainingStart*) i funkcija osluškivača koja eksplicitno generiše objekat događaja pozivom *emit()* metode varijable. Funkcija dekorator *@Output()* ukazuje da atribut prosleđuje podatke (kao objekte događaja) komponente deteta ka roditeljskoj komponenti. U ovom slučaju, novi trening (dete) preko objekta događaja obaveštava glavnu komponentu treninga (roditelj) da je počeo novi trening za izabranom vežbom. Događaj se generiše programski i prosleđuje u metodi *onStartTraining()*.

```

import { Component, OnInit, EventEmitter, Output } from '@angular/core';

```



```
export class NewTrainingComponent implements OnInit {
  @Output() trainingStart = new EventEmitter<void>();

  constructor() {}

  ngOnInit(): void {}

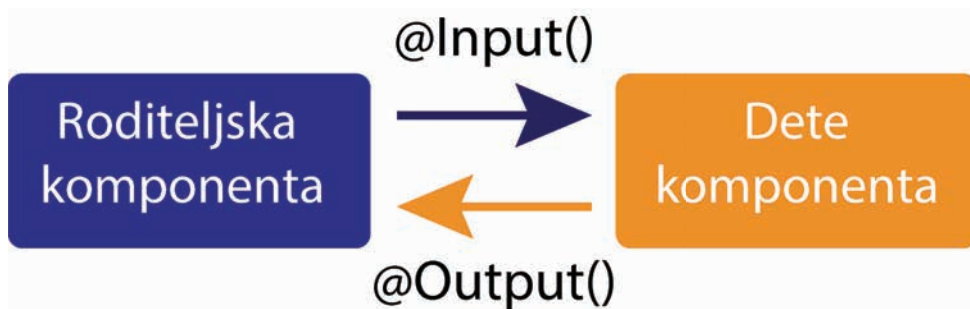
  onStartTraining() {
    this.trainingStart.emit();
  }
}
```

Klikom na dugme *Start* poziva se metoda *onStartTraining()* kao osluškivač događaja. Na ovaj način korisnik inicira programsko generisanje događaja u okviru date metode pomoću kojeg se glavna komponenta treninga obaveštava o pokretanju novog treninga.

```
<mat-card-actions fxLayoutAlign="center">
  <button type="submit" mat-raised-button color="primary"
    (click)="onStartTraining()">Start</button>
</mat-card-actions>
```

Primer ilustruje mehanizam razmene podataka između komponenti aplikacije. Objekti razmene mogu različite vrste podataka, podaci, kao što su događaji, vrednosti atributa, naredbe i funkcije. Princip korišćenja je sledeći:

1. Definiše se atribut sa odgovarajućim dekoratorom funkcijom *@Input* (prosleđivanje od roditelja ka detetu) ili *@Output* (prosleđivanje od deteta ko roditelju, kao u datom slučaju). Atribut kao vrednost prima instancu klase generatora događaja generičkog tipa (*EventEmitter*). Objekat generisanog događaja može imati attribute različitih tipova.
2. Definiše se metoda u okviru koje atribut kreira generički događaj (poziv metode *emit()*).



Slika 8.22. Razmena podataka između roditeljske komponente i komponente deteta. U tekućem primeru, roditelj je komponenta treninga, dok je dete komponenta novog treninga.

U nastavku se u templejtu komponente treninga (*training.component.html*) konsultuje vrednost *ongoingTraining* atributa kontrolera komponente.

Kada trening nije u toku (*ongoingTraining* ima vrednost *false*), prikazivaće se komponente prethodnih i novog treninga. Kada je trening u toku (*ongoingTraining* ima vrednost *true*), biće prikazana samo komponenta tekućeg treninga.

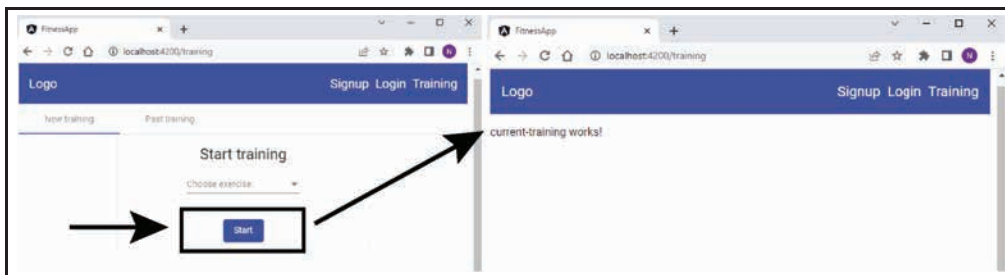
Prikaz odgovarajućih komponenti se kontroliše pomoću strukturne direktive *ngIf* (videti Sekciju 5.3.4.1).

Na ovaj način će kada korisnik pokrene komponentu treninga biti prikazane komponente novog i prethodnog treninga kao sadržani elementi (deca). Kada se pokrene novi trening klikom na dugme *Start*, aplikacija ulazi u režim rada tekućeg treninga koji se tada prikazuje. U komponenti novog treninga kao deteta menja se vrednost *ongoingTraining* atributa kontrolera treninga pomoću *trainingStart* atributa programskog izvor događaja. Nova vrednost *ongoingTraining* atributa (*true*) će usloviti da bude prikazana samo komponenta tekućeg treninga (*current-training*).

Kod i primer u nastavku ilustruju opisani mehanizam.

```
<mat-tab-group *ngIf="!ongoingTraining">
  <mat-tab label="New training">
    <app-new-training (trainingStart)="ongoingTraining=true"> </app-new-training>
  </mat-tab>

  <mat-tab label="Past training">
    <app-previous-training></app-previous-training>
  </mat-tab>
</mat-tab-group>
<app-current-training *ngIf="ongoingTraining"> </app-current-training>
```



8.6.3. Dodavanje progres elementa

U komponentu tekućeg treninga se dodaju progres kao element GKI, motivacioni tekst i dugme za zaustavljanje treninga.

AM biblioteka sadrži dve komponente za prikaza progresu akcija u GKI - kružni progres (eng. *progress spinner*) i linearni progres (eng. *progress bar*). Više informacija u vezi sa korišćenjem komponenti se može naći u [80].

Korišćenje AM komponente kružni progres podrazumeva registraciju odgovarajućeg modula (*MatProgressSpinnerModule*) u modulu za registraciju AM komponenti (*material.module.ts*) i njegovo navođenje u *import* naredbi u zaglavlju kontrolera komponente tekućeg treninga. Postupak je sličan kao za AM komponente opisane u Sekciji 6.

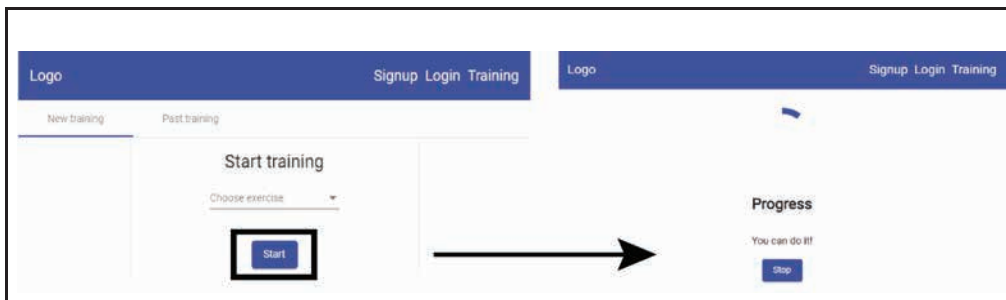
U templejtu komponente tekućeg treninga (*current-training.component.html*) se uklanja postojeći sadržaj i dodaje kružni progres (*mat-progress-spinner*) i sadržaj u nastavku taga.

U cilju razdvajanja sadržaja *section* taga od horizontalne linije alata korene komponente aplikaciju, datoteka stila komponente tekućeg treninga (*current-training.component.css*) definiše pravilo koje dodaje prazan prostor sa unutrašnje strane granice *section* taga (*padding*).

Kod i izgled komponente tekućeg treninga su prikazani ispod.

```
<section fxLayout="column" fxLayoutAlign="center center" fxLayoutGap="20px">
  <mat-progress-spinner mode="determinate" [value]="10"></mat-progress-spinner>
  <h2>Progress</h2>
  <p>You can do it!</p>
  <button mat-raised-button color="primary">Stop</button>
</section>
```

```
section {
  padding: 30px;
}
```



8.6.4. Uređivanje progres elementa

Tajmer (eng. *timer*) je opšti naziv za mehanizam koji izvršava grupu naredbi ili funkciju) u regularnim vremenskim intervalima. U aplikaciji se tajmer koristi za simulaciju statusa izvršenja vežbe u toku tako što inkrementira vrednost kružnog progressa.

U kontroleru komponente tekućeg treninga (*current-training.component.ts*) se dodaje atribut *progress* koji se koristi kao trenutna vrednost kružnog progressa. *SetInterval()* je Angular funkcija tajmera koja omogućava kontinuirano pozivanje proizvoljne funkcije nakon isteka zadatog vremenskog intervala. Kao parametre prihvata funkciju i vrednost intervala u milisekundama nakon čijeg isteka se poziva funkcija. U primeru u nastavku svake sekunde se inkrementira vrednost *progress* atributa. Tajmer se pokreće prilikom inicijalizacije komponente tekućeg treninga.

```
export class CurrentTrainingComponent implements OnInit {
    progress = 0;

    constructor() { }

    ngOnInit(): void {
        setInterval(() => { this.progress += 5; }, 1000);
    }
}
```

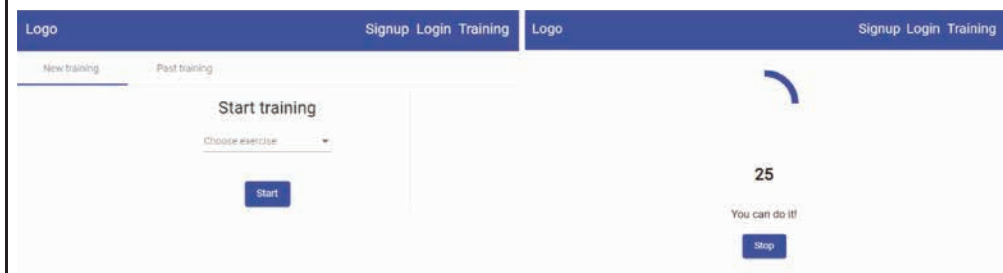
U templejtu komponente tekućeg treninga (*current-training.component.html*) vrednost kružnog progressa (*value*) se povezuje sa vrednošću *progress* atributa (videti povezivanje atributa u Sekciji 5.3.3.2). U nastavku se ispisuje vrednost kružnog progressa korišćenjem izraza interpolacije (videti Sekciju 5.3.3.1) i motivaciona poruka za korisnika.

Rezultat je kao na slici ispod.

```

<section fxLayout="column" fxLayoutAlign="center center" fxLayoutGap="20px">
  <mat-progress-spinner mode="determinate" [value]="progress"></mat-progress-spinner>
  <h2>{{ progress }}</h2>
  <p>You can do it!</p>
  <button mat-raised-button color="primary">Stop</button>
</section>

```



Progres se svake sekunde inkrementira za vrednost 5. Potrebno je omogućiti korisniku da zaustavi tekući trening klikom na *Stop* dugme, kao i da se element progres zaustavi kada dostigne maksimalnu vrednost (100).

Metoda *setInterval()* kao povratnu vrednost vraća instancu tajmera. Atribut *timer* će čuvati povratnu vrednost metode. Metoda se proširuje kao u nastavku. Kada vrednost *progress* atributa bude maksimalna tajmer, tj. izvršavanje metode se zaustavlja pomoću *clearInterval()* metode koja kao parametar prihvata vrednost *timer* atributa. Na ovaj način se dve metode kombinuju tako da upravljaju mehanizmom tajmera. Tajmer se eksplicitno zaustavlja klikom na *Stop* dugme i pozivom metode osluškiavača *onStop()*.

```
export class CurrentTrainingComponent implements OnInit {

  progress = 0;
  timer: any;

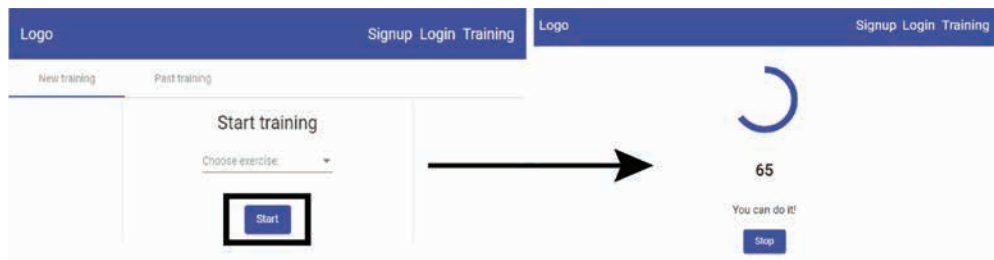
  constructor() { }

  ngOnInit(): void {
    this.timer = setInterval(() => {
      this.progress += 5;
      if(this.progress >= 100){
        clearInterval(this.timer);
      }
    }, 1000);
  }

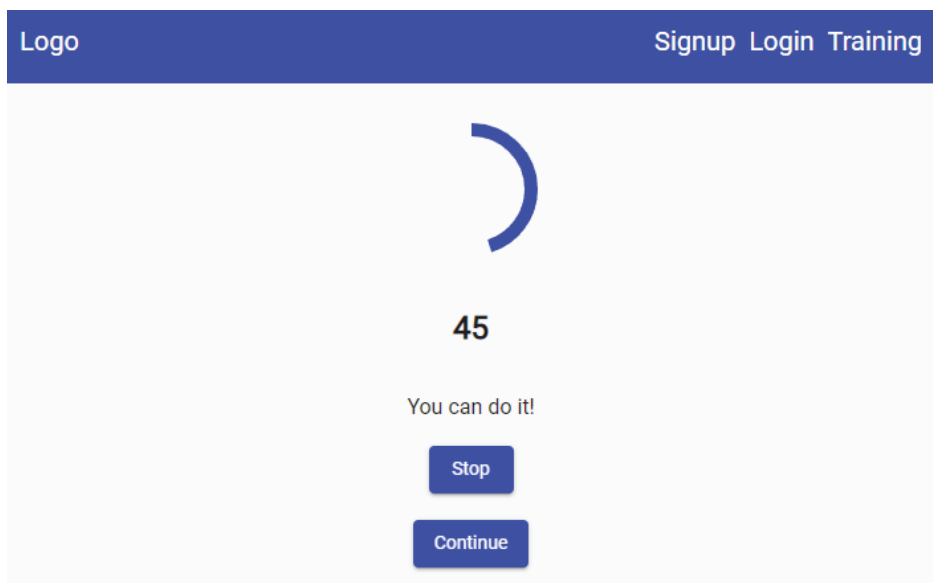
  onStop(){
    clearInterval(this.timer);
  }
}
```

U templejtu komponente tekućeg treninga (*current-training.component.html*) registruje se osluškivač događaja klika *Stop* dugmeta. Rezultat je prikazan na slici u nastavku.

```
<mat-progress-spinner mode="determinate" [value]="progress"></mat-progress-spinner>
<h2>{{ progress }}</h2>
<p>You can do it!</p>
<button mat-raised-button color="primary" (click)="onStop()">Stop</button>
```



Zadatak: Dodati dugme *Continue* koje će nastaviti izvršavanje treninga nakon što se zaustavi pomoću *Stop* dugmeta tako da komponenta tekućeg treninga izgleda kao na slici u nastavku.



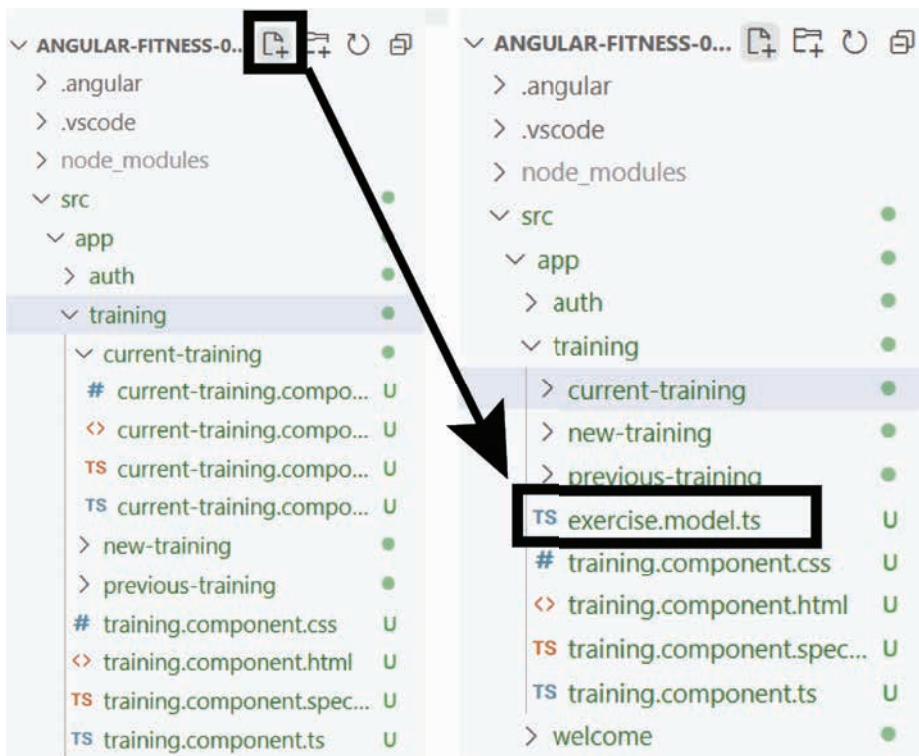
Slika 8.23. Dugme Continue nastavlja tekući trening ukoliko ga korisnik prethodno zaustavi klikom na dugme Stop.

8.7. Dodavanje tabele sa simulacijom pozadinske logike interfejsa

Sažetak: Sekcija dodaje tabelu u komponentu prethodnog treninga ili istorijata. Najpre se kreira tabelarni prikaz završenih treninga. Nakon toga se dodaju funkcionalnosti sortiranja i straničenja. Primer ilustruje korišćenje servisa za kreiranje i pristup podacima koji se prikazuju u tabeli.

8.7.1. Priprema podataka tabele korišćenjem servisa

Najpre se definiše struktura podataka treninga koji će se prikazati tabelarno. Programski se definiše struktura podataka, a podaci strukture se prikazuju tabelarno. Datoteka pod nazivom *exercise.model.ts* se u kreira u *training* direktorijumu projekta. Reč *model* govori da kreirana klasa definiše strukturu podataka vežbe treninga.



Slika 8.24. Kreiranje datoteke `exercise.model.ts` koja definiše strukturu podataka vežbe treninga.

U datoteci se navodi se interfejs *Exercise* koji definiše attribute strukture podataka treninga. Atributi *date* i *status* su opcioni i atribut *status* može imati jednu od navedenih vrednosti.

```
export interface Exercise {
  id: string;
  name: string;
  duration: number;
  calories: number;
  date?: Date;
  status?: 'done' | 'canceled' | null;
}
```

Servis se kreira kao datoteka `training.service.ts` u `training` direktorijumu projekta. Servis sadrži atribut *exercises* tipa niza instanci interfejsa *Exercise*. `Get()` metoda vraća elemente niza.

Na ovaj način se mogu simulirati pozadinska logika i podaci aplikacije. Pored `get()` metoda koja čita elemente niza, mogu se implementirati metode koje dodaju nove elemente niza i menjaju ili brišu postojeće elemente.

```
import { Exercise } from '../exercise.model';

export class TrainingService {

  private exercises: Exercise[] = [
    { id: 'running', name: 'Morning running', duration: 20, calories: 100, date: new Date(), status: 'canceled' },
    { id: 'squats', name: 'Midday squats', duration: 5, calories: 200, date: new Date(), status: 'done' },
    { id: 'walking', name: 'Afternoon walking', duration: 60, calories: 50, date: new Date(), status: 'done' },
    { id: 'running', name: 'Evening running', duration: 100, calories: 300, date: new Date(), status: 'canceled' }
  ];

  getExercises() {
    return this.exercises;
  }
}
```

Kako bi servis dostupan svim komponentama aplikacije, mora se registrovati u korenom modulu (*app.module.ts*).

```
import { TrainingService } from '../training/training.service';

imports: [
  BrowserModule,
  MaterialModule,
  RoutingModule,
  BrowserAnimationsModule,
  FlexLayoutModule,
  FormsModule
],
providers: [TrainingService],
bootstrap: [AppComponent]
```

8.7.2. Kreiranje prikaza tabele

Korišćenje i kreiranje AM tabele je opisano u Sekcijama 6.17 (uvod) i 6.17.1, respektivno.

U nastavku se u templejtu komponente istorijata treninga (*previous-training.component.html*) kreira tabela za prikaz prethodnih treninga. Kolone tabele odgovaraju atributima interfejsa *Exercise*.

Ključna reč *let* koristi se za deklarisanje lokalnih varijabli templejta. Opseg vidljivosti ovakve varijable je tag templejta u kojem je deklararisana.

Globalna varijabla templejta se navodi sa prefiksom #. Varijabla može biti referenca na komponentu, njen opseg vidljivosti je templejt komponente u kojoj je deklarirana.

```
<mat-table [dataSource]="exerciseSource">
  <ng-container matColumnDef="date">
    <mat-header-cell *matHeaderCellDef>Date</mat-header-cell>
    <mat-cell *matCellDef="let element">{{ element.date | date }}</mat-cell>
  </ng-container>
  <ng-container matColumnDef="name">
    <mat-header-cell *matHeaderCellDef>Name</mat-header-cell>
    <mat-cell *matCellDef="let element">{{ element.name }}</mat-cell>
  </ng-container>
  <ng-container matColumnDef="duration">
    <mat-header-cell *matHeaderCellDef>Duration</mat-header-cell>
    <mat-cell *matCellDef="let element">{{ element.duration | number }}</mat-cell>
  </ng-container>
  <ng-container matColumnDef="calories">
    <mat-header-cell *matHeaderCellDef>Calories</mat-header-cell>
    <mat-cell *matCellDef="let element">{{ element.calories | number }}</mat-cell>
  </ng-container>
  <ng-container matColumnDef="status">
    <mat-header-cell *matHeaderCellDef>Status</mat-header-cell>
    <mat-cell *matCellDef="let element">{{ element.status }}</mat-cell>
  </ng-container>
  <mat-header-row *matHeaderRowDef="displayedColumns"></mat-header-row>
  <mat-row *matRowDef="let rows; columns: displayedColumns"></mat-row>
</mat-table>
```

8.7.3. Popunjavanje tabele

U kontroleru komponente prethodnog treninga (*previous-training.component.ts*) se definišu atributi *displayedColumns* koji sadrži nazive zaglavlja kolona tabele, i *dataSource* kao instanca izvora tabele (*MatTableDataSource*) koji sadrži objekte tipa interfejsa *Exercise*.

Servis se navodi kao parametar konstruktora. Time se instanca servisa kreira zajedno sa instancom komponente. Prilikom inicijalizacije komponente (*ngOnInit()*), atributu izvora podataka (*dataSource*) se dodeljuje vrednost niza završenih treninga koju vraća metoda servisa (*getExercises()*). Na ovaj način tabela ima na raspolaganju podatke za prikaz.

```

import { Component, OnInit } from '@angular/core';
import { MatTableDataSource } from '@angular/material/table';
import { Exercise } from '../exercise.model';
import { TrainingService } from '../training.service';

@Component({
  selector: 'app-previous-training',
  templateUrl: './previous-training.component.html',
  styleUrls: ['./previous-training.component.css']
})
export class PreviousTrainingComponent implements OnInit {

  displayedColumns = ["date", "name", "duration", "calories", "status"];
  exerciseSource = new MatTableDataSource<Exercise>();

  constructor(private trainingService: TrainingService) { }

  ngOnInit(): void {
    this.exerciseSource.data = this.trainingService.getExercises();
  }
}

```

Komponenta istorijata treninga sada izgleda kao na slici u nastavku.

Logo		Signup Login Training		
New training		Past training		
Date	Name	Duration	Calories	Status
Jan 14, 2023	Morning running	20	100	canceled
Jan 14, 2023	Midday squats	5	200	done
Jan 14, 2023	Afternoon walking	60	50	done
Jan 14, 2023	Evening running	100	300	canceled

Slika 8.25. Tabelarni prikaz prethodnih treninga.

8.7.4. Sortiranje tabele istorijata treninga

Sortiranje tabele je opisano u Sekciji 6.17.2.

Templejt komponente istorijata treninga (*previous-training.component.html*) se proširuje direktivama kao u nastavku. *MatSort* direktiva uvodi sortiranje na nivou tabele, dok direktiva *mat-sort-header* definiše kolonu po kojoj se mogu sortirati redovi tabele.

```

<mat-table [dataSource]="exerciseSource" matSort>
  <ng-container matColumnDef="date">
    <mat-header-cell *matHeaderCellDef mat-sort-header> Date</mat-header-cell>
    <mat-cell *matCellDef="let element">{{ element.date | date }}</mat-cell>
  </ng-container>
  <ng-container matColumnDef="name">
    <mat-header-cell *matHeaderCellDef mat-sort-header> Name</mat-header-cell>
    <mat-cell *matCellDef="let element">{{ element.name }}</mat-cell>
  </ng-container>
  <ng-container matColumnDef="duration">
    <mat-header-cell *matHeaderCellDef mat-sort-header> Duration</mat-header-cell>
    <mat-cell *matCellDef="let element">{{ element.duration | number }}</mat-cell>
  </ng-container>
  <ng-container matColumnDef="calories">
    <mat-header-cell *matHeaderCellDef mat-sort-header> Calories</mat-header-cell>
    <mat-cell *matCellDef="let element">{{ element.calories | number }}</mat-cell>
  </ng-container>
  <ng-container matColumnDef="status">
    <mat-header-cell *matHeaderCellDef mat-sort-header> Status</mat-header-cell>
    <mat-cell *matCellDef="let element">{{ element.status }}</mat-cell>
  </ng-container>

  <mat-header-row *matHeaderRowDef="displayedColumns"></mat-header-row>
  <mat-row *matRowDef="let rows; columns: displayedColumns"></mat-row>
</mat-table>

```

Zatim se kontroler komponente istorijata treninga (*previous-training.component.ts*) proširuje kao što je naznačeno u nastavku.

Sort svojstvu izvora podataka *exerciseDataSource* se dodaje vrednost *sort* atributa kontrolera. Dodela vrednosti se vrši nakon što podaci za prikaz budu dostupni tabeli (*ngAfterViewInit()*). Dekorator *@ViewChild* prati promene elementa ili direktive DOM stabla (*matSort* direktiva templejta) čiji selektor odgovara atributu kontrolera (*sort*) i ažurira DOM stablo. Na ovaj način se korisnikova akcija sortiranja tabele registruje i u izvoru podataka tabele u kontroleru, ažurira redosled prikaza podataka, i osvežava izgled tabele.

Slika ispod koda prikazuje kolonu po kojoj se mogu sortirati redovi tabele.

```

import { Component, OnInit, ViewChild, AfterViewInit } from '@angular/core';
import { MatSort } from '@angular/material/sort';
import { MatTableDataSource } from '@angular/material/table';
import { Exercise } from '../exercise.model';
import { TrainingService } from '../training.service';

@Component({
  selector: 'app-previous-training',
  templateUrl: './previous-training.component.html',
  styleUrls: ['./previous-training.component.css']
})
export class PreviousTrainingComponent implements OnInit, AfterViewInit {
  displayedColumns = ['date', 'name', 'duration', 'calories', 'status'];
  exerciseSource = new MatTableDataSource<Exercise>();

  @ViewChild(MatSort) sort : MatSort;

  constructor(private trainingService: TrainingService) { }

  ngOnInit(): void {
    this.exerciseSource.data = this.trainingService.getExercises();
  }

  ngAfterViewInit() {
    this.exerciseSource.sort = this.sort;
  }
}

```

Logo					Signup Login Training
New training		Past training			
Date	Name	Duration	Calories	Status ↓	
Jan 14, 2023	Midday squats	5	200	done	
Jan 14, 2023	Afternoon walking	60	50	done	
Jan 14, 2023	Morning running	20	100	canceled	
Jan 14, 2023	Evening running	100	300	canceled	

8.7.5. Straničenje tabele istorijata treninga

Straničenje tabele je opisano u Sekciji 6.17.3.

U templejtu komponente istorijata treninga (*previous-training.component.html*) se navodi *mat-paginator* tag sa *pageSize* direktivom koja definiše tekući broj redova u jednoj stranici tabele, i *pageSizeOptions* direktivom koja dodaje padajuću listu za izbor broja redova, tj. veličine stranice. Ponuđene vrednosti su one navedene u nizu.

```
</ng-container>
<mat-header-row *matHeaderRowDef="displayedColumns"></mat-header-row>
<mat-row *matRowDef="let rows; columns: displayedColumns"></mat-row>
</mat-table>

<mat-paginator [pageSize]="1" [pageSizeOptions]="[1, 2, 4, 6, 10]"></mat-paginator>
```

Zatim se kontroler komponente istorijata treninga (*previous-training.component.ts*) proširuje kao u nastavku.

Paginator svojstvu izvora podataka *exerciseDataSource* se dodaje vrednost *paginator* atributa kontrolera. Dodela vrednosti se vrši nakon što podaci za prikaz budu dostupni tabeli (*ngAfterViewInit()*). Dekorator *@ViewChild* prati promene elementa ili direktive DOM stabla (*mat-paginator* tag) čiji selektor odgovara atributu kontrolera (*paginator*) i ažurira DOM stablo. Na ovaj način se korisnikova akcija straničenja tabele registruje i u izvoru podataka tabele u kontroleru, ažuriraju, tj. selektuju podaci koji se prikazuju na stranici, i osvežava izgled tabele.


```

import { Component, OnInit, ViewChild, AfterViewInit } from '@angular/core';
import { MatPaginator } from '@angular/material/paginator';
import { MatSort } from '@angular/material/sort';
import { MatTableDataSource } from '@angular/material/table';
import { Exercise } from '../exercise.model';
import { TrainingService } from '../training.service';

@Component({
  selector: 'app-previous-training',
  templateUrl: './previous-training.component.html',
  styleUrls: ['./previous-training.component.css']
})
export class PreviousTrainingComponent implements OnInit, AfterViewInit {

  displayedColumns = ["date", "name", "duration", "calories", "status"];
  exerciseSource = new MatTableDataSource<Exercise>();

  @ViewChild(MatSort) sort : MatSort;
  @ViewChild(MatPaginator) paginator : MatPaginator;

  constructor(private trainigService: TrainingService) { }

  ngOnInit(): void {
    this.exerciseSource.data = this.trainigService.getExercises()
  }
  ngAfterViewInit(){
    this.exerciseSource.sort = this.sort;
    this.exerciseSource.paginator = this.paginator;
  }
}

```

Dodate opcije za straničenje tabele su prikazane na slici u nastavku.

Logo					Signup Login Training
New training		Past training			
Date	Name	Duration	Calories	Status	
Jan 14, 2023	Morning running	20	100	canceled	
Jan 14, 2023	Midday squats	5	200	done	
Items per page: 2 1 - 2 of 4 < >					

Slika 8.26. Straničenje tabele prethodnih treninga.

8.8. Filtriranje tabele i dodavanje dijaloga

Sažetak: Sekcija dodaje funkciju filtriranja, odnosno pretrage podataka tabele. Zatim se AM komponenta dijaloga implementira u komponenti tekućeg treninga aplikacije. Opisano je prosleđivanje podataka od pozivajuće komponente ka dijalogu.

8.8.1. Filtriranje tabele istorijata treninga

Filtriranje tabele je opisano u Sektiji 6.17.4.

U templejt komponente istorijata treninga (*previous-training.component.html*) filter se dodaje kao tekstualno polje za unos pre definicije tabele. Kada korisnik unosi sadržaj za pretragu, poziva se funkcija koja izdvaja podatke tabele (*doFilter()*). Argument metode osluškivača je sadržaj tekstualnog polja. Eksplicitna konverzija u tip *any* omogućava pretragu podataka tabele sa različitim tipovima vrednosti.

```
<mat-form-field>
  <input matInput type="text" (keyup)="doFilter($any($event.target).value)" placeholder="Filter">
</mat-form-field>
```

Metoda osluškivača se implementira u kontroleru komponente istorijata treninga (*previous-training.component.ts*). *Filter* svojstvo izvor podataka tabele izdvaja podatke tabele na osnovu zadate vrednosti tako da se samo filtrirani podaci prikazuju u tabeli. Vrednost je sadržaj koji korisnik unosi u tekstualno polje u templejtu komponente. Sadržaj se obrađuje tako što se uklanjaju specijalni znaci na početku i kraju koje korisnik može slučajno uneti (*trim()*). Zatim se rezultujući string pretvara u mala slova ili znakove (*toLowerCase()*). Slika u nastavku koda ilustruje korišćenje filtera tabele istorijata treninga.

```
doFilter(filterValue: string){
  this.exerciseSource.filter = filterValue.trim().toLowerCase();
}
```


Logo

Signup Login Training

New training

Past training

Filter

done

Date	Name	Duration	Calories	Status
Jan 14, 2023	Midday squats	5	200	done
Jan 14, 2023	Afternoon walking	60	50	done

Items per page: 4

1 – 2 of 2

<

>

Na kraju se polje filtera pozicionira po sredini prikaza komponente istorijata treninga korišćenjem `fxLayoutAlign` direktive (videti Sekciju 7.2.1). Širina polja zauzima 35 procenata trenutne širine roditeljskog `div` taga, tj. templejta komponente. Za korišćenje `fxFlex` direktive videti Sekciju 7.2.2.

```

<div fxLayoutAlign="center center">
  <mat-form-field fxFlex="35%">
    <input matInput type="text" (keyup)="doFilter($any($event.target).value)" placeholder="Filter">
  </mat-form-field>
</div>

```

Logo

Signup Login Training

New training

Past training

Filter

Date	Name	Duration	Calories	Status
Jan 14, 2023	Morning running	20	100	canceled
Jan 14, 2023	Midday squats	5	200	done
Jan 14, 2023	Afternoon walking	60	50	done
Jan 14, 2023	Evening running	100	300	canceled

Items per page: 4

1 – 4 of 4

<

>

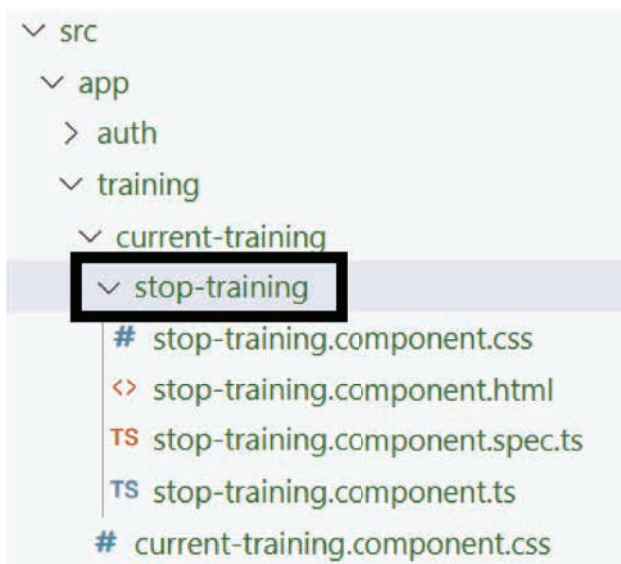
8.8.2. Dodavanje dijaloga za rad sa tekućim treningom

Korišćenje dijaloga je opisano u Sekciji 6.18. Dijalog se koristi za potvrdu prekida tekućeg treninga aplikacije.

Najpre se kreira komponenta za zaustavljanje tekućeg treninga. U interfejsu komandne linije *Terminal* prozora (iz korenog direktorijuma projekta *angular-fitness-app*) komponenta se kreira u okviru komponente tekućeg treninga naredbom u nastavku.

```
ng g c training/current-training/stop-training --  
module app.module
```

Opcija *--module app.module* naredbe znači da će kreirana komponenta biti registrovana u korenom modulu aplikacije (*app.module.ts*).



Slika 8.27. Struktura komponente za zaustavljanje treninga u hijerarhiji komponenti projekta.

Primer dijaloga u nastavku je detaljnije opisan u Sekciji 6.18.

U templejtu kreirane komponente (*stop-training.component.html*) se navodi sadržaj dijaloga kao u nastavku.

```

<section>
  <h1 mat-dialog-title>Stop training</h1>
  <mat-dialog-content>Are you sure?</mat-dialog-content>

  <mat-dialog-actions>
    <button mat-button [mat-dialog-close]="true">Yes</button>
    <button mat-button [mat-dialog-close]="false">No</button>
  </mat-dialog-actions>
</section>

```

Pozivajuća komponenta je komponenta tekućeg treninga, dok se dijalog prikazuje u okviru komponente koja se poziva (komponenta za zaustavljanje treninga).

U kontroleru pozivajuće komponente (*current-training.component.ts*) instanca servisa (*dialog*) se navodi kao argument konstruktora. Klik na dugme za zaustavljanje treninga poziva metodu *onStop()* koja stopira tajmer koji inkrementira element progres i otvara dijalog (*open()*) iz instance servisa.

```

import { Component, OnInit } from '@angular/core';
import { MatDialog } from '@angular/material/dialog';
import { StopTrainingComponent } from './stop-training/stop-training.component';

@Component({
  selector: 'app-current-training',
  templateUrl: './current-training.component.html',
  styleUrls: ['./current-training.component.css']
})
export class CurrentTrainingComponent implements OnInit {
  progress = 0;
  timer: any;

  constructor(private dialog: MatDialog) { }

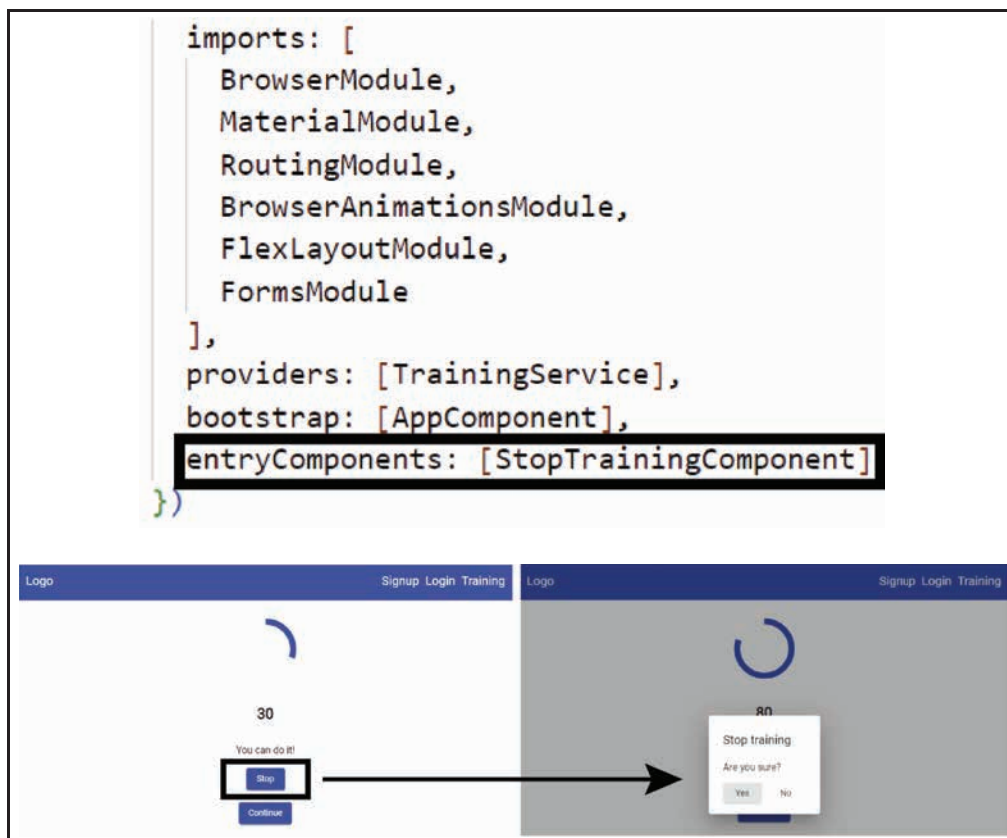
  ngOnInit(): void {
    this.timer = setInterval(() => {
      this.progress += 5;
      if(this.progress >= 100){
        clearInterval(this.timer);
      }
    }, 1000);
  }

  onStop() {
    clearInterval(this.timer);
    this.dialog.open(StopTrainingComponent);
  }
}

```

Servis *MatDialog* kreira dijalog dinamički, u toku izvršavanja Angular aplikacije. Radi optimizacije performansi aplikacije, komponente koje se generišu dinamički se moraju eksplicitno deklarirati u vreme pisanja programa. Komponenta za zaustavljanje treninga sadrži dijalog i navodi se kao element *entryComponents* niza *@Component* dekoratora korenog modula aplikacije (*app.module.ts*).

Slika u nastavku koda prikazuje izgled komponente za zaustavljanje treninga, odnosno dijaloga prilikom zaustavljanja treninga u toku.



8.8.3. Prosleđivanje podataka dijalogu

Povratna vrednost metode *open()* je pokazivač na objekat dijaloga. Metoda sadrži dva argumenta. Prvi argument metode je naziv komponente dijaloga. Drugi argument služi za prosleđivanje podataka dijalogu. Argument je tipa *any* i podaci se navode u formatu *ime:vrednost*. U nastavku se metoda proširuje tako što se navodi drugi parametar kao objekat *data* koji sadrži svojstvo *progress* koje dobija vrednost istoimenog atributa kontrolera komponente (*current-training.component.ts*).

```

onStop() {
  clearInterval(this.timer);
  this.dialog.open(StopTrainingComponent, {
    data: {
      progress: this.progress
    }
  });
}

```

U kontroleru komponente za zaustavljanje treninga (*stop-training.component.ts*) se prihvataju podaci koje prosleđuje pozivajuća komponenta. *MAT_DIALOG_DATA* predefinisani objekat omogućava pristup podacima koji se prosleđuju u *open()* metodi. *@Inject()* direktiva se koristi za umetanje vrednosti objekta (*MAT_DIALOG_DATA*) u navedenu varijablu (*receivedData*). Varijabla je tipa *any* kako bi se čuvale vrednosti proizvoljnih tipova. Na ovaj način se kreira zavisnost objekta i varijable tako da podaci objekta budu prosleđeni kao vrednost varijable.

```

import { Component, Inject, OnInit } from '@angular/core';
import { MAT_DIALOG_DATA } from '@angular/material/dialog';

@Component({
  selector: 'app-stop-training',
  templateUrl: './stop-training.component.html',
  styleUrls: ['./stop-training.component.css']
})
export class StopTrainingComponent implements OnInit {
  constructor(@Inject(MAT_DIALOG_DATA) public receivedData: any) {}

  ngOnInit(): void {}
}

```

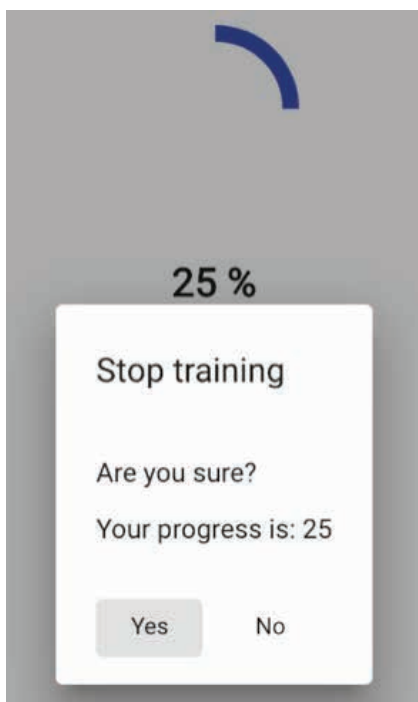
Korišćenjem interpolacije se vrednost atributa *receivedData*, odnosno prosleđena vrednost *progres* prikazuje u dijalogu za zaustavljanje treninga.

```

<section>
  <h1 mat-dialog-title>Stop training</h1>
  <mat-dialog-content>Are you sure?</mat-dialog-content>
  <p>Your progress is: {{ receivedData.progress }}</p>

  <mat-dialog-actions>
    <button mat-button [mat-dialog-close]="true">Yes</button>
    <button mat-button [mat-dialog-close]="false">No</button>
  </mat-dialog-actions>
</section>

```



8.9. Upravljanje dijalogom i dodavanje teme

Sažetak: Sekcija demonstrira prosleđivanje podataka od dijaloga ka pozivajućoj komponenti. U zavisnosti izabrane opcije dijaloga, trening se nastavlja ili prekida. Zatim je opisano kreiranje i primena korisničke teme. Na kraju se dodatno uređuje komponenta početna stranica aplikacije.

8.9.1. Prosleđivanje podataka od dijaloga ka pozivajućoj komponenti

U nastavku se izbor opcije dijaloga kao vrednost prosleđuje pozivajućoj komponenti tekućeg treninga i prikazuje u konzoli pretraživača. Metoda `onStop()` se proširuje tako što se prilikom otvaranja dijaloga čuva referenca na objekat otvorenog dijaloga (`dialogRef`). Referenca registruje osluškivač događaja zatvaranja dijaloga (`afterClosed()`). Metoda osluškivača ispisujeadržaj dijaloga u konzoli. Sadržaj koji se prosleđuje od komponente dijaloga ka komponenti treninga je vrednost `mat-dialog-close` atributa (videti povezivanje atributa u Sekciji 5.3.3.2). Na taj način se u konzoli ispisuje vrednost atributa `mat-dialog-close` templejta. Ukoliko se dijalog zatvori izborom opcije *Yes*, prikazuje se *true*. U suprotnom se prikazuje *false*.



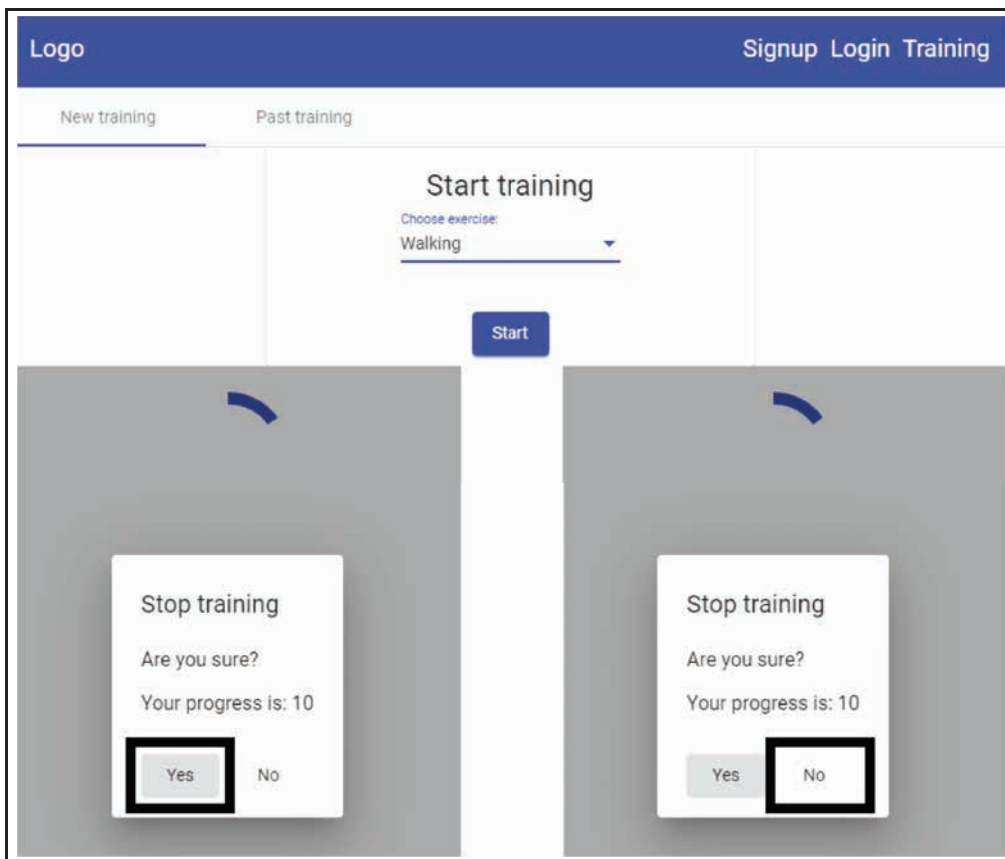
U zavisnosti od toga da li korisnik bira *Yes* ili *No* dugme, tekući trening se zaustavlja ili nastavlja. Proširuje se metoda osluškivača događaja (`afterClosed()`) tako što se vrši provera vrednosti prosleđene od dijaloga, tj. da li je zatvoren izborom opcije *Yes* ili *No*. Opcija *Yes* vraća vrednost *true* i poziva se metoda koja zaustavlja tajmer, odnosno dalji progres treninga (`clearInterval()`). Opcija *No* vraća vrednost *false* i poziva metodu koja nastavlja trening (`onResume()`). Metoda ponovo pokreće tajmer (`setInterval()`) koji inkrementira postojeću vrednost `progres` atributa kontrolera povezanog sa

atributom *value* progres elementa (videti Sekcije 8.6.3 i 8.6.4). Na ovaj način se nastavlja progres treninga u toku.

```
onStop() {
  clearInterval(this.timer);
  const dialogRef = this.dialog.open(StopTrainingComponent, {
    data: {
      progress: this.progress
    }
  });

  dialogRef.afterClosed().subscribe(result => {
    if (result){
      clearInterval(this.timer)
    }
    else {
      this.onResume()
    }
  });
}

onResume(){
  this.timer = setInterval(() => {
    this.progress += 5;
    if (this.progress >= 100) {
      clearInterval(this.timer);
    }
  }, 1000);
}
```

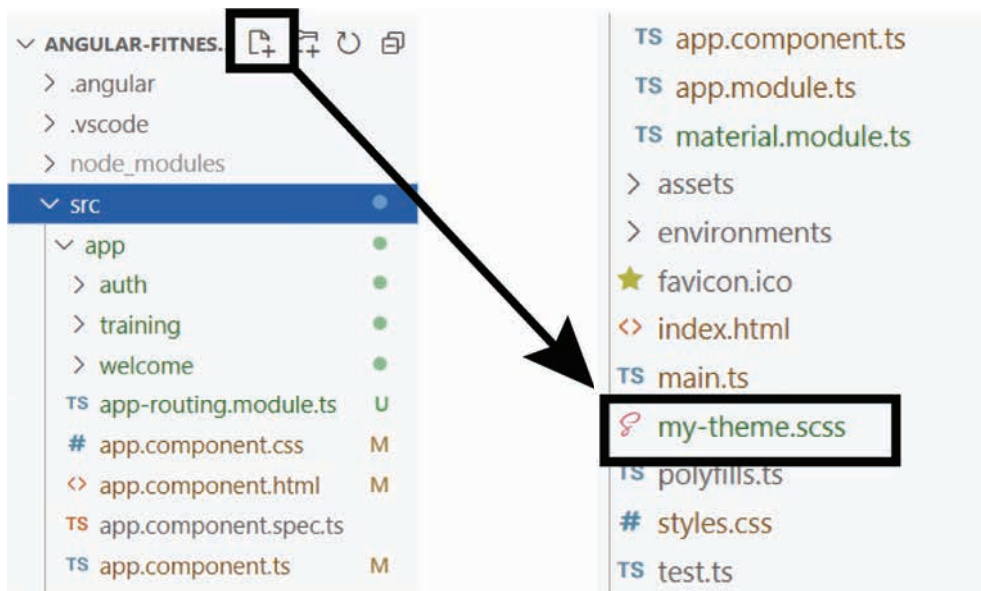



8.9.2. Kreiranje teme

Korišćenje tema je opisano u Sekciji 6.3.

Teme predstavljaju palete boja koje se primenjuju na templejte komponenti aplikacije. Izbor palete teme se vrši tako tako što se *color* atributu elementa templejta dodeli vrednost koja odgovara nazivu palete (*primary*, *accent* ili *warn*).

U *src* direktorijumu projekta kreirati datoteku pod nazivom *my-theme.scss*, kao na Slici 8.28.



Slika 8.28. Kreiranje datoteke teme my-theme.scss.

Ekstenzija `.scss` potiče od SASS jezika (eng. *Syntactically Awesome Style Sheets*) kao nadogradnje CSS jezika [88] koja služi za definisanje tema. Naredbe SASS jezika se prevode CSS naredbe. Samim tim se definicije predefinisanih tema (videti Sekciju 6.3) nalaze u `.css` datotekama.

Dok se detalji SASS mogu naći u [88], u nastavku se koriste naredbe potrebne za definisanje teme.

Najpre se referenciraju biblioteke (`@import` naredba) i ugrađene funkcije (`@include` naredba) potrebne za definisanje teme.

Zatim se definiše svaka paleta teme kao varijabla (prefiksom `$`). Varijabli se dodeljuje povratna vrednost funkcije paleta boja (eng. *mat-palette*) koja kao parametar prihvata naziv željene boje (`$mat-ime_boje`). Za svaku boju mogu se dodati nijanse - podrazumevana (*A200*), svetlija (*A100*) i tamnija (*A400*).

Kada se definišu palete i njihove boje varijablama, tema, kao objekat sastavljen od paleta kreira se pomoću Funkcije `mat-light-theme()` i `mat-dark-theme()` kreiraju temu pri čemu kao parametre prihvataju iznad kreirane varijable koje predstavljaju palete. `Mat-light-theme()` funkcija kreira temu svetlije pozadine i tamnije frontalne boje (tekst), dok `mat-dark-theme()` radi obrnuto.

Kreirana tema se prilikom pokretanja aplikacije učitava, prevodi i primenjuje pomoću funkcije `angular-material-theme()`.

Opisani mehanizam je obrazac za kreiranje korisničkih tema.

```

@import '~@angular/material/theming';

@include mat-core();

$my-theme-primary: mat-palette($mat-orange);
$my-theme-accent: mat-palette($mat-blue, A200, A100, A400);
$my-theme-warn: mat-palette($mat-red);

$my-theme: mat-dark-theme($my-theme-primary,
                          $my-theme-accent,
                          $my-theme-warn);

@include angular-material-theme($my-theme);

```

Kako je prethodno odabrana predefinisana tema, istu je potrebno ukloniti ili komentarisati u datoteci stila projekta (*styles.css*) kako bi se kreirana tema primenjivala na komponente aplikacije.

```

/* @import '@angular/material/prebuilt-themes/indigo-pink.css'; */

```

Kako bi tema bila vidljiva na nivou aplikaciji, mora se navesti u *styles* nizu datoteke *angular.json*.

Sačuvati promene i pokrenuti aplikaciju.

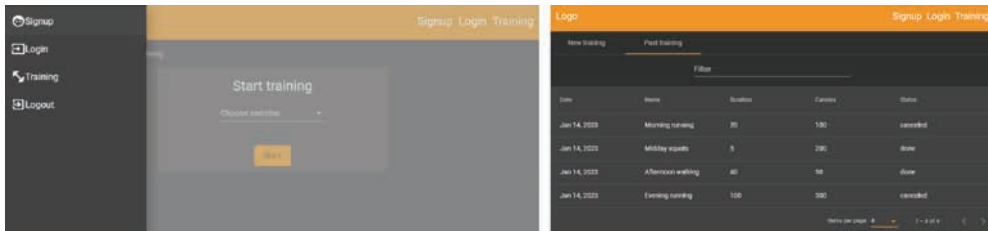
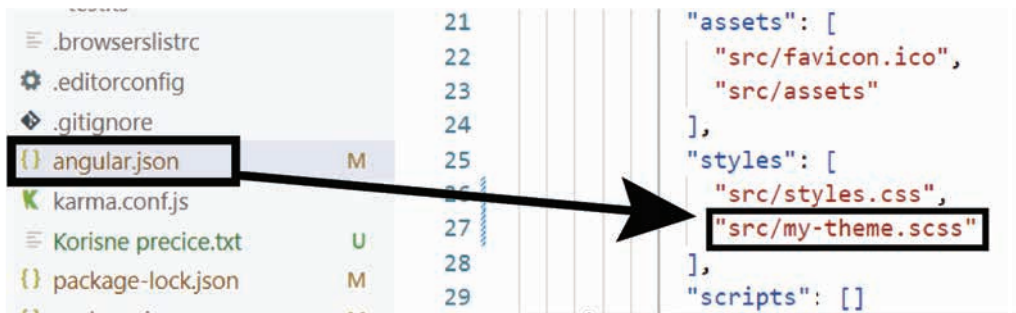
U slučaju greške, u *Terminal* prozoru iz korenog direktorijuma projekta (*angular-fitness-app*) pokrenuti naredbu kao u nastavku.

```

npm rebuild node-sass

```

Nakon izvršenja naredbe zaustaviti i pokrenuti razvojni server i pogledati promene izgleda aplikacije u pretraživaču.



Slika 8.29. Izgled aplikacije nakon primene kreirane teme my-theme.scss.

Zadatak: Promeniti funkciju za kreiranje teme. Koristiti funkciju *mat-light-theme()* umesto funkcije *mat-dark-theme()*. Aplikacija izgleda kao na slici u nastavku.



Slika 8.30. Izgled aplikacije nakon primene funkcije mat-light-theme().

8.9.3. Uređivanje početne stranice

Postojeći sadržaj templejta komponente početne stranice aplikacije (*welcome.component.html*) se uklanja i dodaju motivacione poruke za korisnike. Elementi se raspoređuju po glavnoj i upravnoj osi, kao u kodu i na slici u nastavku.

```
<div fxLayout="column" fxLayoutAlign="center center">
  <h1>Activities!</h1>
  <p>Be active and healthy</p>

  <h1>Community / forum</h1>
  <p>meet new people</p>
</div>
```

Logo

Signup Login Training

Activities!

Be active and healthy

Community / forum

meet new people

8.10. Simulacija pozadinske logike

Sažetak: Sekcija kreira servis za simulaciju pozadinske logike rada sa korisničkim nalogima aplikacije. Servis obezbeđuje podatke i implementira metode za prijavu i registraciju korisnika. Nakon toga se menjaju komponente za prijavu i registraciju korisnika kako bi koristile podatke i metode servisa.

8.10.1. Kreiranje servisa za rad sa korisničkim nalogima

Rad sa servisima je opisan u Sekciji 5.4.

Datoteka servisa *user.service.ts* se kreira u *src/app/auth* direktorijumu projekta. Zatim se servis registruje u korenom modulu aplikacije (*app.module.ts*). Servis može biti automatski registrovan u korenom modulu ako je generisan naredbom za kreiranje servisa.

```

imports: [
  BrowserModule,
  MaterialModule,
  RoutingModule,
  BrowserAnimationsModule,
  FlexLayoutModule,
  FormsModule
],
providers: [TrainingService, UserService],
bootstrap: [AppComponent],

```

U kreiranom servisu se najpre definiše se interfejs za opisivanje profila korisnika. Svaki korisnik je opisan identifikatorom (*id*), korisničkim imenom (*email*), lozinkom (*password*) i datumom rođenja (*date*).

```

import { Injectable } from "@angular/core";

export interface User {
  id: number;
  email: string;
  password: string;
  date: Date;
}

```

Klasa servisa sadrži statički atribut *dummyUserList* čija je vrednost niz objekata interfejsa *User*. Servisu se dodaju metoda koja vraća korisničko ime iz objekta korisnika (*getUserName()*) i metoda koja vraća objekat korisnika na osnovu na osnovu identifikatora (*getUserById()*).

```

@Injectables()
export class UserService {
    static dummyUserList: Array<User> = [
        {
            id: 0,
            email: "admin@mail.rs",
            password: "876543210",
            date: new Date("2022-3-18 16:00")
        },
        {
            id: 1,
            email: "user@mail.rs",
            password: "012345678",
            date: new Date("2023-04-23 14:00")
        }
    ];
    getUsername(user: User): string {
        return user.email;
    }
    getUserById(id: number): User {
        var foundUser : User;
        UserService.dummyUserList.forEach(user => {
            if (user.id == id) {
                foundUser = user;
            }
        });
        return foundUser;
    }
}

```

Zatim se u klasi servisa definišu sledeće metode:

- *GetUser()* - na osnovu korisničkog imena vraća objekat korisnika;
- *IsPasswordCorrect()* - proverava ispravnost unosa korisničkog imena i lozinke;
- *RegisterUser()* - registruje novog korisnika tako što kreira i dodaje objekat korisnika na kraj niza korisnika. Prikazuje kreirani objekat korisnika u konzoli pretraživača.


```

getUser(email: string): User {
    return UserService.dummyUserList.find(userToFind => userToFind.email == email);
}
isPasswordCorrect(email: string, password: string): boolean {
    return UserService.dummyUserList.find(userToFind =>
        (userToFind.email == email && userToFind.password == password)) != undefined;
}
registerUser(email: string, password: string, date: Date): User {
    var maxId: number = 0;
    UserService.dummyUserList.forEach(user => {
        if (maxId < user.id) {
            maxId = user.id;
        }
    });
    var id = ++maxId;
    var user: User = {
        id,
        email,
        password,
        date
    };
    UserService.dummyUserList.push(user);
    console.log(user);
    return user;
}

```

8.10.2. Izmene komponente za registraciju korisnika

Komponentu je potrebno prilagoditi kako bi koristila metode kreiranog servisa. U kontroleru komponente za registraciju (*signup.component.ts*) se navode atributi *errorExists* i *errorTest*. Prvi atribut se koristi za verifikaciju podataka forme za registraciju korisnika, dok se drugi koristi za kreiranje i prikaz greške u toku registracije.

U konstruktoru se kao argumenti navode servis za rad korisničkim nalogima (*userService*) i ruter objekat (*router*) koji se koristi za preusmeravanje korisnika na početnu stranicu kod uspešne registracije.

Logika registracije korisnika se implementira u metodi osluškivača (*onSubmit()*). Najpre se proverava da li korisnik sa datim imenom već postoji. Ukoliko ne postoji, vrednost *errorExists* atributa se postavlja na *false*. Zatim se kreira objekat korisnika i dodaje u niz korisnika pomoću odgovarajuće metode servisa (*registerUser()*). Metodi se kao parametri prosleđuju uneti podaci forme. Korisnik se zatim preusmerava na komponentu početnu stranicu aplikacije pozivom odgovarajuće metode *router* objekta (*navigate()*). Kao parametar se navodi vrednost *path* atributa rute *welcome* komponente (videti Sekciju 8.1.8).

U slučaju da korisnik sa datim imenom već postoji, dodeljuju se odgovarajuće vrednosti atributima za *errorExists* i *errorTest*.


```

export class SignupComponent implements OnInit {

  errorExists = false;
  errorText = "";

  constructor(private userService: UserService, private router: Router) { }

  ngOnInit(): void {
  }
  onSubmit(form: NgForm){
    if (!this.userService.getUser(form.value.email)) {
      this.errorExists = false;
      var newUser = this.userService.registerUser(form.value.email,
                                                    form.value.password,
                                                    form.value.birthDate);
      this.router.navigate(['']);
    } else {
      this.errorExists = true;
      this.errorText = "User with this email already exists.";
    }
  }
}

```

U templejtu komponente za registraciju korisnika (*signup.component.html*) se dodaje labela (*mat-label*) za prikaz poruke o grešci. Labela se navodi ispod dugmeta za slanje podataka forme (*Submit*). *NgIf* strukturna direktiva prikazuje labelu ukoliko atribut *errorExists* ima vrednosti *true*. Labela prikazuje sadržaj atributa *errorText*. *False* vrednost atributa *errorExists* označava uspešnu registraciju korisnika i tada se poruka, tj. labela ne prikazuje. U nastavku se dodaje link ka komponenti za prijavu korisnika (*login*).

```

<mat-label *ngIf="errorExists"> {{ errorText }} </mat-label>
<a routerLink="/login">Go back to login.</a>
</form>
</section>

```

8.10.3. Izmene komponente za prijavu korisnika

Na sličan način se modifikuje komponenta za prijavu korisnika (*login*).

U kontroler komponente se dodaju atributi *errorExists* i *errorText*. U konstruktoru klase kontrolera kao argumenti se navode servis za rad korisničkim nalogima (*userService*) i ruter objekat (*router*) za preusmeravanje korisnika na početnu stranicu u slučaju uspešne prijave.

Logika prijave korisnika se implementira u metodi osluškivača (*onSubmit()*). Najpre se proverava da li korisnik sa datim imenom već postoji. Najpre se preuzimaju podaci forme koje je korisnik eksplicitno uneo. Na osnovu unetih podataka forme, proverava se da li je korisnik registrovan korisnik. Ukoliko nije registrovan, *errorExists* atribut dobija vrednost *true*, kreira se odgovarajuća poruka o grešci i završava sa izvršavanjem metode.

Zatim se proverava ispravnost unete lozinke pozivom odgovarajuće metode servisa za rad sa korisničkim nalogima (*isPasswordCorrect()*). U slučaju neispravne lozinke, postavljaju se odgovarajuće vrednosti dva atributa i izlazi iz metode.

U slučaju ispravnog unosa imena i lozinke, korisnik se preusmerava na komponentu početnu stranicu aplikacije pozivom odgovarajuće metode *router* objekta (*navigate()*).

```
export class LoginComponent implements OnInit {

  errorExists = false;
  errorText = "";
  constructor(private userService : UserService, private router : Router) {}

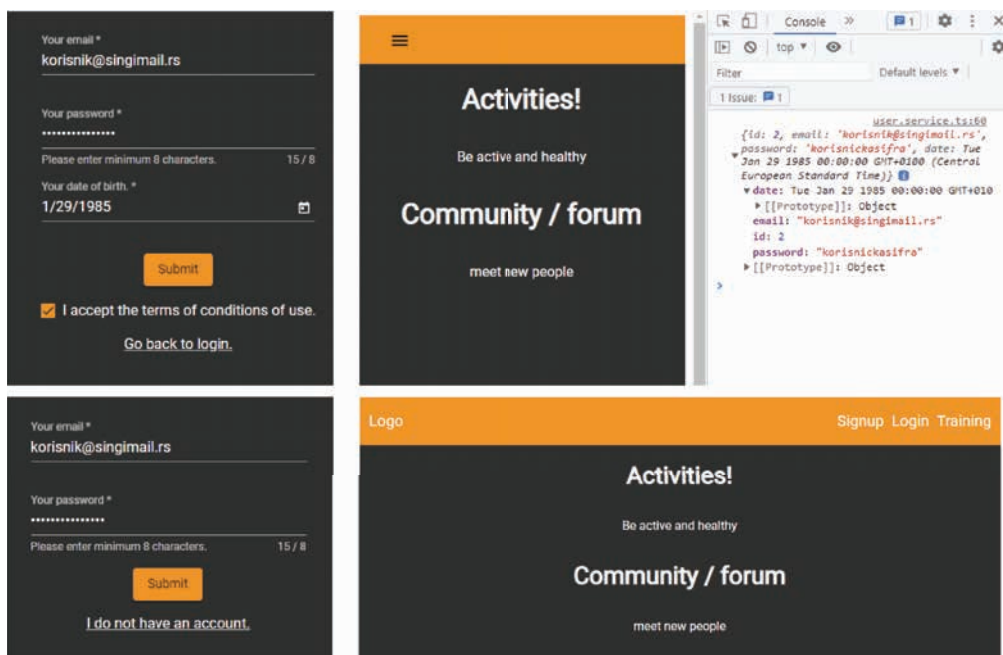
  ngOnInit() {}
  onSubmit (form : NgForm) {
    var email = form.value.email;
    var password = form.value.password;
    var user = this.userService.getUser(email);
    if (!user) {
      this.errorExists = true;
      this.errorText = "There is no registered user with the email " + email;
      return;
    }
    var isValid = this.userService.isPasswordCorrect(email, password);
    if (!isValid) {
      this.errorExists = true;
      this.errorText = "Password is incorrect";
      return;
    }
    this.errorExists = false;
    this.router.navigate(['']);
  }
}
```

U templateju komponente za prijavu korisnika (*login.component.html*) se dodaje labela (*mat-label*) za prikaz poruke o grešci. Labela se navodi ispod dugmeta za slanje podataka forme (*Submit*). *NgIf* strukturna direktiva prikazuje labelu ukoliko atribut *errorExists* ima vrednosti *true*. Labela prikazuje sadržaj atributa *errorText*. *False* vrednost atributa *errorExists*

označava uspešnu prijavu korisnika i tada se poruka, tj. labela ne prikazuje. U nastavku se dodaje link ka komponenti za registraciju korisnika (*signup*).

```
<mat-label *ngIf="errorExists">{{errorText}}</mat-label>
<a routerLink="/signup">I don't have an account.</a>
</form>
</section>
```

Registracija i prijava korisnika izgledaju kao na Slici 8.31.



Slika 8.31. Registracija korisnika (gore) i prijava registrovanog korisnika (dole).

8.11. Ažuriranje profila korisnika

Sažetak: Sekcija proširuje servis za simulaciju pozadinske logike rada sa korisničkim nalogima aplikacije. Dodaju se metode za izmenu profila korisnika. Kreira se komponenta za izmenu podataka profila korisnika pomoću metoda servisa.

8.11.1. Proširenje servisa za rad sa korisničkim nalogima

Interfejs za definisanje strukture korisnika (*User*) se proširuje opcionim atributom adrese korisnika. U klasi servisa se dodaje atribut *currentUser* čija je vrednost trenutno prijavljen korisnik. Atribut inicijalno ukazuje na prvog korisnika u nizu korisnika (*dummyUserList*).

```
export interface User {  
    id: number;  
    email: string;  
    password: string;  
    date: Date;  
    address?: string;  
}  
  
@Injectable()  
export class UserService {  
    currentUser: User = UserService.dummyUserList[0];  
}
```

Metode servisa se proširuju tako da svaka ažurira vrednost atributa tekućeg korisnika kao u nastavku.

```

    getUserById(id: number): User {
        var foundUser : User;
        UserService.dummyUserList.forEach((user) => {
            if (user.id == id) {
                foundUser = user;
            }
        });

        this.currentUser = foundUser;
        return foundUser;
    }

    getUser(userEmail: string): User {
        this.currentUser = UserService.dummyUserList.find(
            (userToFind) => userToFind.email == userEmail
        );
        return this.currentUser;
    }

    registerUser(email: string, password: string, date: Date): User {
        var maxId: number = 0;
        UserService.dummyUserList.forEach((user) => {
            if (maxId < user.id) {
                maxId = user.id;
            }
        });
        var id = ++maxId;
        var user: User = {
            id,
            email,
            password,
            date
        };
        UserService.dummyUserList.push(user);

        this.currentUser = user;

        console.log(user);
        return user;
    }
}

```

8.11.2. Kreiranje komponente profila korisnika

Komponenta omogućava izmenu podataka profila korisnika. Kreira se navođenjem naredbe u nastavku u *Terminal* prozoru razvojnog okruženja.

```
ng g c auth/profile --module app.module.ts
```

Naredba kreira *profile* komponentu u direktorijumu *src/app/auth/profile* projekta. Komponenta je registrovana u korenom modulu aplikacije (*app.module.ts*).

Radi optimizacije performansi aplikacije, komponente koje se generišu dinamički se eksplicitno deklarišu u vreme pisanja programa. Profil komponenta sadrži dijalog koji omogućava pregled i izmene podataka korisnika. Zbog toga se komponenta dodaje kao element *entryComponents* niza *@Component* dekoratora korenog modula aplikacije.

```
imports: [  
  BrowserModule,  
  MaterialModule,  
  RoutingModule,  
  BrowserAnimationsModule,  
  FlexLayoutModule,  
  FormsModule  
],  
providers: [TrainingService, UserService],  
bootstrap: [AppComponent],  
entryComponents: [StopTrainingComponent, ProfileComponent]
```

U zaglavlju templejta komponente profila (*profile.component.html*) se smeštaju ikone za zatvaranje dijaloga (*close*) i za prelazak u režim izmene podataka (*edit*). Ikona za izmenu se prikazuje samo ako je korisnik prijavljen (*ngIf* direktiva).


```

<div fxLayoutAlign="space-between">
  <button mat-button [mat-dialog-close]="true">
    <mat-icon matListIcon>close</mat-icon>
  </button>
  <button mat-icon-button>
    <mat-icon matListIcon
      *ngIf="this.profileForInput.id == this.userService.currentUser.id"
      (click)="this.isEditing = !this.isEditing">edit</mat-icon>
  </button>
</div>

```

Glavni deo templejta su elementi za prikaz i ažuriranje podataka profila korisnika. Sekcija je organizovana kao dijalog (*mat-dialog-content*) u koji se u okviru forme (*form*) smeštaju tri kartice (*mat-card*) za ime, lozinku i adresu korisnika.

Svaka kartica ima dva dela koja se naizmenično prikazuju - jedan u režimu prikaza, drugi u režimu izmene.

Tehnika dizajna kartice je sledeća:

- U kontroleru komponente profila se definiše atribut *isEditing* koji označava režim pregleda (*false*) ili izmene podataka (*true*). Početna vrednost atributa je *false*.
- Vrednost atributa se naizmenično menja klikom na *edit* ikonu iznad (izraz koji se izvršava na nastanak *click* događaja). Vrednost se proverava u svakoj kartici, jednom za svaki režim pomoću *ngIf* direktive.
- *False* vrednost atributa aktivira režim prikaza u kojem se podatak ne može menjati (**ngIf="!this.isEditing"*). *True* vrednost aktivira režim koji omogućava izmenu podatka kao polja forme (**ngIf="this.isEditing"*).
- U režimu izmene polja forme se obostrano povezuju sa odgovarajućim atributima objekta tekućeg korisnika u kontroleru komponente. Videti Sektiju 5.3.3.4 za obostrano povezivanje. Na ovaj način će promene polja templejta biti sačuvane u objektu korisnika u kontroleru.

U skladu sa prethodno opisanom tehnikom, templejt komponente profila izgleda kao u nastavku. Metoda *finishEditing()* kontrolera se poziva prilikom slanja podataka forme (*ngSubmit*), odnosno izmene podataka korisnika.

```

<mat-dialog-content>

  <form fxLayout="column" fxLayoutGap="2vh" #editForm="ngForm" (ngSubmit)="finishEditing(editForm)">

    <!-- Email -->
    <mat-card>
      <div fxLayout="row" fxLayoutAlign="start center" fxLayoutGap="1vw">
        <mat-icon>mail</mat-icon>

        <div fxLayout="column" *ngIf="!this.isEditing">
          <mat-card-subtitle>Email</mat-card-subtitle>
          <span>{{ this.profileForInput.email }}</span>
        </div>

        <mat-form-field *ngIf="this.isEditing">
          <input matInput type="email" placeholder="Your email" ngModel name="email"
            [(ngModel)]=this.profileForInput.email email required #emailInput="ngModel">

          <mat-error *ngIf="emailInput.hasError('required')">Email is required</mat-error>
          <mat-error *ngIf="!emailInput.hasError('required')">Email has incorrect format</mat-error>

        </mat-form-field>

      </div>
    </mat-card>

    <!-- Password -->
    <mat-card *ngIf="this.isEditing">
      <div fxLayout="row" fxLayoutAlign="start center" fxLayoutGap="1vw">

        <mat-icon>lock</mat-icon>
        <mat-form-field hintLabel="Minimum 8 characters">
          <input matInput type="password" placeholder="Your password" ngModel name="password" minlength="8"
            [(ngModel)]=this.profileForInput.password password required #passwordInput="ngModel">

          <mat-hint align="end">{{ passwordInput.value?.length }} / 8</mat-hint>
        </mat-form-field>
      </div>
    </mat-card>

    <!-- Address -->
    <mat-card>
      <div fxLayout="row" fxLayoutAlign="start center" fxLayoutGap="1vw">
        <mat-icon>near_me</mat-icon>

        <div fxLayout="column" *ngIf="!this.isEditing">
          <mat-card-subtitle>Address</mat-card-subtitle>
          <span>{{ this.profileForInput.address }}</span>
        </div>

        <mat-form-field *ngIf="this.isEditing">
          <input matInput type="text" placeholder="Address" ngModel name="address"
            [(ngModel)]=this.profileForInput.address required #addressInput="ngModel">
          <mat-error *ngIf="addressInput.hasError('required')">Please write your full address.</mat-error>
        </mat-form-field>
      </div>
    </mat-card>

    <button *ngIf="this.isEditing" mat-raised-button type="submit" color="primary">Save user</button>
  </form>
</mat-dialog-content>

```

Kontroler komponente profila (*profile.component.ts*) implementira logiku za korišćenje dijaloga i obradu podataka korisnika koji se menjaju. Atribut *profileForInput* tipa *User* čuva izmene napravljene u dijalogu templejta komponente. Prilikom inicijalizacije (*ngOnInit()*), odnosno

otvaranja komponente kao dijaloga atribut dobija vrednost tekućeg korisnika čiji podaci su prosleđeni dijalogu (*data* argument konstruktora).

U toku izmena podataka u dijalogu templejta, vrednosti polja atributa se automatski se ažuriraju na osnovu unetih vrednosti polja templejta sa kojima su obostrano povezani. Pre čuvanja promena (*submit* forme) u kontroleru će postojati dva objekta (skupa podataka) tekućeg korisnika:

- Atribut *data* - predstavlja objekat korisnika prosleđen dijalogu prilikom otvaranja. Predstavlja izvorni, tj. originalni objekat iz niza korisnika pre ažuriranja vrednosti.
- Atribut *profileForInput* - predstavlja lokalnu kopiju objekta korisnika vezanu za komponentu. Predstavlja izmenjeni objekat nakon ažuriranja vrednosti.

Nakon promena u templejtu poziva se metoda osluškivača *finishEditing()*. Metoda ažurira originalnu vrednost objekta prosleđenu dijalogu (*data*) vrednostima polja lokalnog objekta korisnika koji sadrži ažurirane podatke (*profileForInput*). Prikazuju se vrednosti promenjenog originala i niza korisnika, respektivno. Na kraju se izlazi iz režima izmene podataka profila tako što atribut *isEditing* dobija vrednost *false*.

Kod u nastavku realizuje prethodno opisanu logiku.

```

export class ProfileComponent implements OnInit {
  isEditing: boolean = false;
  profileForInput : User;

  constructor(
    @Inject(MAT_DIALOG_DATA) public data: any,
    public userService: UserService
  ) {}

  ngOnInit(): void {
    this.profileForInput = {
      id: this.data.user.id,
      email: this.data.user.email,
      date: this.data.date,
      password: this.data.user.password,
      address: this.data.user.address
    };
  }

  finishEditing(form: NgForm) {
    this.data.user.email = this.profileForInput.email;
    this.data.user.password = this.profileForInput.password;
    this.data.user.address = this.profileForInput.address;
    console.log(this.data.user);
    console.log(UserService.dummyUserList);
    this.isEditing = false;
  }
}

```

U datoteci stila komponente profila (*profile.component.css*) definišu se pravila za izgled dijaloga.

```

mat-form-field {
  width: 100%;
}

mat-card-subtitle {
  margin-bottom: 2vh;
  font-size: 0.7em;
}

```

8.11.3. Rad sa komponentom profila korisnika

U horizontalnoj liniji alata za navigaciju templejta korene komponente (*app.component.html*) i u nastavku linkova ruta ka komponentama aplikacije se dodaje dugme za prikaz komponente profila. Klikom na dugme se poziva metoda koja prikazuje komponentu kao dijalog (*openProfile()*). Kao argument se prosleđuje identifikator *id* trenutno prijavljenog korisnika. Naziv dugmeta je korisničko ime kao adresa elektronske pošte.

```

<div fxFlex fxLayoutAlign="end" fxLayoutGap="15px" fxHide.xs>
  <a routerLink="/signup">Signup</a>
  <a routerLink="/login">Login</a>
  <a routerLink="/training">Training</a>
  <button mat-raised-button color="warn" (click)="openProfile(this.userService.currentUser.id)">
    {{ this.userService.currentUser.email }}
  </button>
</div>
</mat-toolbar>

```

Metoda oslušivača (*openProfile()*) se implementira u kontroleru korene komponente (*app.component.ts*). Metoda dodeljuje atributu *profileOpened* vrednost *true* i otvara komponentu profila kao dijalog (*open()*). Atribut *disableClose* određuje da li će se dijalog zatvoriti kada korisnik klikne izvan polja dijaloga. Dijalogu se prosleđuje objekat korisnik (*user*) kreiran na osnovu vrednosti identifikatora *userId* kao argumenta metode. Nakon zatvaranja dijaloga (*afterClosed()*) atribut *profileOpened* dobija vrednost *false*. Rad sa komponentom profila izgleda kao na slici u nastavku koda.

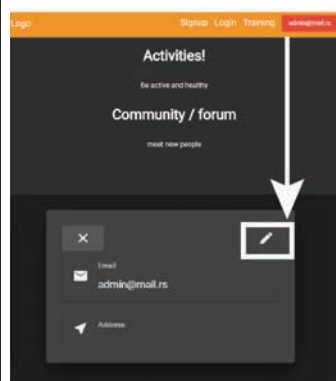
```
export class AppComponent {

  profileOpened: boolean = false;

  constructor(public userService: UserService, private dialog: MatDialog) {}

  openProfile(userId: number){
    this.profileOpened = true;
    const profileDialog = this.dialog.open(ProfileComponent, {
      disableClose: true,
      width: "25vw",
      data: { user: this.userService.getUserById(userId)}
    });

    profileDialog.afterClosed().subscribe(result => {
      this.profileOpened = false;
    });
  }
}
```



9. Literatura

1. Grudin, J. (1990). The computer reaches out: the historical continuity of interface design. In Proceedings of CHI'90, pp. 261-268.
2. Gentner R., Grudin, J. (1990). Why good engineers (sometimes) create bad interfaces. Conference on Human Factors and Computing Systems, Seattle, Washington, United States, ACM Press, NY, USA.
3. Wexelblat, R.L. (Ed.) (1978). Proceedings of ACM SIGPLAN History of Programming Languages Conference. SIGPLAN Notices, 13, 8.
4. Fitts, P. M. (1954). The information capacity of the human motor system controlling the amplitude of movement. J. Exper. Psych. 47, 6, 381–391.
5. McGuffin, M.J., Balakrishnan, R. (2005). Fitts' Law and Expanding Targets: Experimental Studies and Designs for User Interfaces. ACM Transactions on Computer-Human Interaction, Vol. 12, No. 4, 388-422.
6. Shneiderman, B. (1986). Seven plus or minus two central issues in human-computer interaction. Proceedings CHI'86 Human Factors in Computing Systems.
7. Myers, B. (1998). A Brief History of Human-Computer Interaction Technology. Interactions, ACM Press, 1998, 44-54.
8. Turk, M., Robertson, G. (2000). Perceptual user interfaces (introduction). Communications of the ACM, Vol. 43, No. 3, 33-35.
9. Vertegaal, R. (2000). Attentive User Interfaces. Communications of the ACM, Vol. 46, No. 3, 31-33.
10. Allanson, J. (2002). Electrophysiologically Interactive Computer Systems. IEEE Computer, 60-65.
11. Ishii, H. (2008). The Tangible UserInterface and It's Evolution. Communications of the ACM, Vol. 51, No. 6, 32-37.
12. McTear, M. F. (2002). Spoken dialogue technology: enabling the conversational user interface. ACM Computing Surveys, 34(1), 90-169.
13. Chen, W., Shan, Y., Wu, Y., Yan, Z., Li, X. (2021). Design and Evaluation of a Distance-Driven User Interface for Asynchronous Collaborative Exhibit Browsing in an Augmented Reality Museum. IEEE Access, 9, 73948-73962.
14. Scheifler, R.W., Gettys, J. (1986). The X Window System. ACM Transactions on Graphics 5, 2, 79–109.
15. Microsoft Azure Kinect uređaj: <https://azure.microsoft.com/en-us/products/kinect-dk/> (Pristupljeno 20.12.2022).
16. Mane, R., Chouhan, T., Guan, C. (2020). BCI for stroke rehabilitation: motor and beyond. Journal of Neural Engineering, 17(4), 041001.
17. Ishii, H. (2008). The Tangible UserInterface and It's Evolution. Communications of the ACM, Vol. 51, No. 6, 32-37.
18. Muender, T., Reinschluessel, A. V., Drewes, S., Wenig, D., Döring, T., Malaka, R. (2019). Does it feel real? using tangibles with different

- fidelities to build and explore scenes in virtual reality. In Proceedings of the 2019 CHI Conference on Human Factors in Computing Systems, 1-12.
19. Jovanović, M., Baez, M., Casati, F. (2020). Chatbots as conversational healthcare services. *IEEE Internet Computing*, 25(3), 44-51.
 20. Weiser, M. (1994). The world is not a desktop. *ACM Interactions* 1(1), 7-8.
 21. Johnson, J. (2020). *Designing with the Mind in Mind: Simple Guide to Understanding User Interface Design Guidelines*. 3rd edition. Morgan Kaufmann Publishing.
 22. Smith, D.K., & Alexander, R.C. (1988). *Fumbling the Future: How Xerox Invented, Then Ignored, the First Personal Computer*.
 23. Olsen, D. R. (1998). *Developing user interfaces*. Morgan Kaufmann.
 24. Gamma, E., Helm, R., Johnson, R., Vlissides, J. (1995). *Design patterns: elements of reusable object-oriented software*. Pearson.
 25. MODELS - VIEWS - CONTROLLERS, Xerox PARC technical note (1979). <https://folk.universitetetioslo.no/trygver/themes/mvc/mvc-index.html> (Pristupljeno 20.12.2022).
 26. Krasner, G. E., Pope, S. T. (1988). A cookbook for using the model-view-controller user interface paradigm in smalltalk-80. *Journal of Object Oriented Programming*, 1(3), 26-49.
 27. Fowler, M. (2006). *GUI Architectures*. <https://martinfowler.com/eaDev/uiArchs.html> (Pristupljeno 20.12.2022).
 28. How to Use Text Fields, The Java Tutorials (2022). <https://docs.oracle.com/javase/tutorial/uiswing/components/textfield.html> (Pristupljeno 20.12.2022).
 29. René, L. (2013). The MVC pattern implemented with java swing, <https://link-intersystems.com/blog/2013/07/20/the-mvc-pattern-implemented-with-java-swing/> (Pristupljeno 20.12.2022).
 30. Fowler, M. (2002). *Patterns of Enterprise Application Architecture*. Addison-Wesley Professional.
 31. Winograd, T. (1975). Frame representations and the declarative/procedural controversy. In *Representation and understanding* (pp. 185-210). Morgan Kaufmann.
 32. StatCounter Global Stats, Desktop vs Mobile Market Share Worldwide (2022). <https://gs.statcounter.com/platform-market-share/desktop-mobile/worldwide> (Pristupljeno 20.11.2022).
 33. Statista, Percentage of mobile device website traffic worldwide from 1st quarter 2015 to 2nd quarter 2022. <https://www.statista.com/statistics/277125/share-of-website-traffic-coming-from-mobile-devices/> (Pristupljeno 20.11.2022).
 34. McVicar, E. *Designing for Mobile, Part 1: Information Architecture* (2012). <https://www.uxbooth.com/articles/designing-for-mobile-part-1-information-architecture/> (Pristupljeno 20.12.2022).

35. Nielsen Norman Group, Flat vs. Deep Website Hierarchies (2013). <https://www.nngroup.com/articles/flat-vs-deep-hierarchy/> (Pristupljeno 20.12.2022).
36. Nielsen Norman Group, Customer-Service Information on Websites: The Hub-and-Spoke Model (2018). <https://www.nngroup.com/articles/customer-service-model/> (Pristupljeno 20.12.2022).
37. Nielsen Norman Group, Tabs, Used Right (2016). <https://www.nngroup.com/articles/tabs-used-right/> (Pristupljeno 20.12.2022).
38. Marschner, S., Shirley, P. (2021). Fundamentals of Computer Graphics: International Student Edition, 5th Edition. A K Peters/CRC Press.
39. Jovanović, M., Jevremović, A. (2020). Interakcija Čovek-Računar. <https://singipedia.singidunum.ac.rs/izdanje/43280-interakcija-covek-racunar>. Univerzitet Singidunum.
40. Mullet, K., Sano, D. (1994). Designing Visual Interfaces: Communication Oriented Techniques. Prentice Hall.
41. Interaction Design Foundation, Simplicity in Design: 4 Ways to Achieve Simplicity in Your Designs (2020). <https://www.interaction-design.org/literature/article/simplicity-in-design-4-ways-to-achieve-simplicity-in-your-designs> (Pristupljeno 20.12.2022).
42. Bertin, J. (1981). Graphics and Graphic Information Processing. De Gruyter.
43. Krug, S. (2013). Don't Make Me Think, Revisited: A Common Sense Approach to Web Usability (3rd Edition). New Riders.
44. Nielsen Norman Group, Using Grids in Interface Designs (2022). <https://www.nngroup.com/articles/using-grids-in-interface-designs/> (Pristupljeno 20.12.2022).
45. Nielsen Norman Group, Proximity Principle in Visual Design (2020). <https://www.nngroup.com/articles/gestalt-proximity/> (Pristupljeno 20.12.2022).
46. Nielsen Norman Group, Using Color to Enhance Your Design (2021). <https://www.nngroup.com/articles/color-enhance-design/> (Pristupljeno 20.12.2022).
47. Sawyer, B. D., Dobres, J., Chahine, N., Reimer, B. (2017). The Cost of Cool: Typographic Style Legibility in Reading at a Glance. In Proceedings of the Human Factors and Ergonomics Society Annual Meeting, 61(1), 833-837. SAGE Publications.
48. Rabinowitz, T. (2015). Exploring Typography, 2nd Edition. Cengage Learning.
49. Statista, Most used programming languages among developers worldwide as of 2022. <https://www.statista.com/statistics/793628/worldwide-developer-survey-most-used-languages/> (Pristupljeno 20.12.2022).

50. Wikipedia, Brendan Eich. https://en.wikipedia.org/wiki/Brendan_Eich (Pristupljeno 20.12.2022).
51. JavaScript, Mozilla Developer Network (2022). <https://developer.mozilla.org/en-US/docs/Web/JavaScript> (Pristupljeno 20.12.2022).
52. V8 JavaScript Engine (2022). <https://v8.dev/> (Pristupljeno 20.12.2022).
53. SpiderMonkey Mozilla JavaScript Engine (2022). <https://spidermonkey.dev/> (Pristupljeno 20.12.2022).
54. JavaScript Functions Hoisting, Mozilla Developer Network (2022). https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Functions#function_hoisting (Pristupljeno 20.12.2022).
55. JavaScript Hoisting, Mozilla Developer Network (2022). <https://developer.mozilla.org/en-US/docs/Glossary/Hoisting> (Pristupljeno 20.12.2022).
56. JavaScript Equality comparisons and sameness, Mozilla Developer Network (2022). https://developer.mozilla.org/en-US/docs/Web/JavaScript/Equality_comparisons_and_sameness (Pristupljeno 20.12.2022).
57. JavaScript Working with objects, Mozilla Developer Network (2022). https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Working_with_Objects (Pristupljeno 20.12.2022).
58. JavaScript Indexed collections, Mozilla Developer Network (2022). https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Indexed_collections (Pristupljeno 20.12.2022).
59. JavaScript Object prototypes, Mozilla Developer Network (2022). https://developer.mozilla.org/en-US/docs/Learn/JavaScript/Objects/Object_prototypes (Pristupljeno 20.12.2022).
60. Classes in JavaScript, Mozilla Developer Network (2022). https://developer.mozilla.org/en-US/docs/Learn/JavaScript/Objects/Classes_in_JavaScript (Pristupljeno 20.12.2022).
61. JavaScript Arrow function expressions, Mozilla Developer Network (2022). https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Functions/Arrow_functions (Pristupljeno 20.12.2022).
62. JavaScript Introduction to the DOM, Mozilla Developer Network (2022). https://developer.mozilla.org/en-US/docs/Web/API/Document_Object_Model/Introduction (Pristupljeno 20.12.2022).

63. DOM standard (2022). <https://dom.spec.whatwg.org/> (Pristupljeno 20.12.2022).
64. JavaScript Introduction to events, Mozilla Developer Network (2022). https://developer.mozilla.org/en-US/docs/Learn/JavaScript/Building_blocks/Events (Pristupljeno 20.12.2022).
65. UI Events, W3C Working Draft (2016). <https://www.w3.org/TR/DOM-Level-3-Events/> (Pristupljeno 20.12.2022).
66. TypeScript: Documentation - Classes (2022). <https://www.typescriptlang.org/docs/handbook/2/classes.html> (Pristupljeno 04.01.2023).
67. TypeScript: Documentation - tsc CLI options (2022). <https://www.typescriptlang.org/docs/handbook/compiler-options.html> (Pristupljeno 06.01.2023).
68. Learning Angular: A no-nonsense guide to start building web applications with Angular, 4th Edition (2023)
69. Node.js Guide (2022). <https://nodejs.org/en/docs/guides/> (Pristupljeno 06.01.2023).
70. Angular - Introduction to modules (2023). <https://angular.io/guide/architecture-modules> (Pristupljeno 06.01.2023).
71. Angular - Introduction to components and templates (2023). <https://angular.io/guide/architecture-components> (Pristupljeno 08.01.2023).
72. Angular - Lifecycle hooks (2022). <https://angular.io/guide/lifecycle-hooks> (Pristupljeno 08.01.2023).
73. Angular - Understanding binding (2023). <https://angular.io/guide/binding-overview> (Pristupljeno 10.01.2023).
74. Angular - Tour of Heroes application and tutorial (2022). <https://angular.io/tutorial/tour-of-heroes> (Pristupljeno 12.01.2023).
75. Angular - Built-in directives (2022). <https://angular.io/guide/built-in-directives> (Pristupljeno 13.01.2023).
76. Angular - Creating an injectable service (2022). <https://angular.io/guide/creating-injectable-service> (Pristupljeno 17.01.2023).
77. Angular - Using Angular routes in a single page application (2022). <https://angular.io/guide/router-tutorial> (Pristupljeno 18.01.2023).
78. Angular Material (2023). <https://material.angular.io/> (Pristupljeno 25.01.2023).
79. Material Design (2023). <https://m3.material.io/> (Pristupljeno 25.01.2023).
80. Angular Material: Documentation - Categories (2022). <https://material.angular.io/components/categories> (Pristupljeno 20.11.2022).

81. Flexbox, Mozilla Developer Network (2023). https://developer.mozilla.org/en-US/docs/Learn/CSS/CSS_layout/Flexbox (Pristupljeno 01.02.2023).
82. Angular Flex-Layout (2023). <https://github.com/angular/flex-layout> (Pristupljeno 01.02.2023).
83. Theming Angular Material (2023). <https://material.angular.io/guide/theming> (Pristupljeno 25.01.2023).
84. Material Icons (2023). <https://fonts.google.com/icons?selected=Material+Icons> (Pristupljeno 04.01.2023).
85. A Complete Guide to Flexbox (2013). <https://css-tricks.com/snippets/css/a-guide-to-flexbox/> (Pristupljeno 03.01.2023).
86. Angular Flex-Layout Demos: Version: 7.0.0-beta.23 (2022). <https://tburleson-layouts-demos.firebaseio.com/#/docs> (Pristupljeno 04.01.2023).
87. Visual Studio Code Installation (2023). <https://code.visualstudio.com/> (Pristupljeno 07.02.2023).
88. SASS: Documentation (2023). <https://sass-lang.com/documentation/> (Pristupljeno 04.02.2023).
89. HammerJS (2023). <https://hammerjs.github.io/> (Pristupljeno 07.02.2023).
90. Responsive API - angular/flex-layout Wiki (2023). <https://github.com/angular/flex-layout/wiki/Responsive-API> (Pristupljeno 01.03.2023).

Spisak slika i tabela

Slika 1.1. Fitts-ov zakon kaže da vreme pokreta ka cilju zavisi od veličine cilja i udaljenosti do cilja. Na slici, prosečna udaljenost od naziva menija do stavke menija zavisi od tipa menija: linearni meni je manje efikasan od pravougaonog menija, koji je dalje manje efikasan od kružnog menija. Izvor: https://www.nngroup.com/articles/fitts-law/ .	9
Slika 1.2. Xerox PARC radna stanica iz 1973. godine. Izvor: https://www.computerhistory.org/revolution/input-output/14/347 .	10
Slika 1.3. Primer korisničkog interfejsa proširene stvarnosti. Izvor [13].	12
Slika 2.1. Elementi stabla prikaza u GKI. Izvor [23].	16
Slika 2.2. Osluškiivači događaja se dodeljuju čvorovima - vidljivim elementima stabla prikaza GKI koji predstavljaju izvore događaja. Dugmad imaju osluškivač registrovan za klik događaj, dok graničnik (eng. <i>splitter</i>) generiše i registrovanom osluškivaču prosleđuje događaj prevlačenja (eng. <i>drag</i>). Izvor [23].	17
Slika 2.3. Raspodela odgovornosti elemenata model-prikaz-kontroler obrasca. Izvor: RegisFrey - Own work, Public Domain, https://commons.wikimedia.org/w/index.php?curid=10298177 .	19
Slika 2.4. Model-prikaz-kontroler (a) šema i (b) implementacija u Java Swing biblioteci za GKI. Korisnik: 1) unosi kriterijum za pretragu (model) u tekstualno polje (prikaz); 2) aktivira pretragu klikom na dugme (prikaz) koje generiše događaj i inicira funkcije pretrage u osluškivaču (kontroler); i 3) interfejs prikazuje rezultat pretrage (model) u višelinijskom tekstualnom polju (prikaz). Izvor [29].	20
Slika 2.5. Vremenski redosled izvršavanja elemenata model-prikaz-kontroler obrasca u dijalogu za pretragu. Izvor [29].	21
Slika 2.6. Uporedni prikaz deklarativnog pristupa u jeziku HTML (levo)	23
Slika 2.7. Primer deklarativne specifikacije korisničkog interfejsa pomoću direktne manipulacije.	24
Slika 2.8. Kombinacija deklarativnog i proceduralnog pristupa u Angular okviru za dizajn i programiranje korisničkih interfejsa.	24
Slika 2.9. Hijerarhijska arhitektura GKI (gore), izvor [34], i tipovi hijerarhije (ravna dole levo, duboka dole desno), izvor [35].	26
Slika 2.10. Primer ravne hijerarhije - prvi nivo (gore) prikazuje kategorije proizvoda, drugi nivo (dole) nudi proizvode izabrane kategorije. Izvor: https://optikica.com/ .	27

Slika 2.11. Centralizovana arhitektura GKI (gore) i primena u klijent-servis scenariju (dole). Izvor [34].	28
Slika 2.12. Ugnježdena arhitektura GKI (gore), izvor [34], i hijerarhijski meni kao često korišćen element za navigaciju (dole).	29
Slika 2.13. Kartična arhitektura GKI (gore), izvor [34], i primer linije alata (dole), izvor: https://www.booking.com .	30
Slika 2.14. Arhitektura GKI tipa Komandna tabla (gore), izvor [34], i primer komandne table (dole), izvor: https://www.freepik.com/free-vector/gradient-elements-collection-dashboard-template_5500787.htm Image by pikisuperstar on Freepik.	31
Slika 2.15. Filter arhitektura GKI (gore), izvor [34], i primer filtera za pretragu (dole), izvor: https://www.booking.com .	33
Slika 2.16. Red događaja skladišti događaje po redosledu njihovog pojavljivanja. Po istom redosledu događaje procesiraju osluškivači. Izvor: https://circuitcellar.com/resources/event-queues/ .	35
Slika 2.17. Transformacije između nivoa prikaza GKI. Komponente GKI se sastoje od geometrijskih primitiva koje se prikazuju kao dvodimenzionalni niz piksela na ekranu.	37
Slika 2.18. Ilustracija iscrtavanja elemenata stabla prikaza (levo) kao skupa primitiva (sredina) i naredbi za crtanje primitiva (desno).	38
Slika 2.19. Duplo baferovanje razdvaja brzinu iscrtavanja stabla prikaza od frekvencije osvežavanja ekrana. Prikazuje frontalni bafer (front), iscrtava u pozadinski bafer (back), i zamenjuje uloge dva bafera. Izvor [38].	39
Slika 3.1. Princip jednostavnosti je univerzalan i nalaže onu količinu sadržaja neophodnu za funkcije sistema ili uređaja. Izvor [40].	41
Slika 3.2. Princip jednostavnosti se posmatra u kontekstu namene GKI i ciljnih korisnika. Interfejs Miro kolaborativnog alata za dijagrame i projektne timove (gore), izvor: https://miro.com , i interfejs Google Search alata za pretragu namenjenog korisniku Interneta (dole), izvor: https://www.google.com .	42
Slika 3.3. Primena tehnike redukcije na primeru poznatih aplikacija, Apple Music (gore) i Instagram (dole). Izvor: https://medium.com/slate-teams/complexion-reduction-a-new-trend-in-mobile-design-cef033a0b978 .	44

Slika 3.4. Primer regularnosti predstavlja mrežni raspored (eng. <i>grid</i>) čije ćelije imaju uniformne dimenzije, kao i strukturu sadržaja svake ćelije. Izvor: https://flipboard.com .	45
Slika 3.5. Šablon za opis komandi koje koriste pokrete prstiju i šake korisnika (eng. <i>gesture</i>) je primer regularnosti. Izvor: https://www.apple.com/macbook-pro/ .	46
Slika 3.6. Mrvice omogućavaju korisnicima kretanje kroz GKI (funkcija navigacije) i pružaju informaciju o trenutnoj poziciji korisnika u GKI (funkcija orijentacije). Izvor: https://www.justinmind.com/blog/breadcrumb-website-navigation/ .	47
Slika 3.7. Primeri vizuelnih varijabli kao zasebnih promena (sa leva desno, odozgo na dole): boje, nijanse, oblika, pozicije, orijentacije i veličine.	48
Slika 3.8. Kontrast korišćenjem veličine i pozicije (levo), i veličine, pozicije, oblika i boje (desno). Izvor: [43].	48
Slika 3.9. Točak boja organizuje boje na taj način da su dijametralno raspoređene boje komplementarne, dok susedne boje nisu. Izvor: https://www.canva.com/colors/color-wheel/ .	49
Slika 3.10. Dobar kontrast se često postiže kombinacijom boje, pozicije i veličine elemenata GKI. Izvor: https://gss.rs (Gorska Služba Spasavanja Srbije).	49
Slika 3.11. Ponavljajuće forme u dizajnu GKI doprinose utisku celine.	50
Slika 3.12. Varijante mreže mogu definisati ravnanje po kolonama (levo), modularno kao uniformne ćelije (sredina), i kao hijerarhiju sa ćelijama različitih dimenzija i raspored (desno). Hijerarhija može istaći određene delove prikaza u odnosu na ostatak sadržaja. Izvor: [44].	51
Slika 3.13. Zbog principa blizine se u skupu levo opažaju redovi, dok se u skupu desno opažaju kolone čak i kada su izgledi elemenata različiti. Izvor: [45].	51
Slika 3.14. Paragrafi teksta predstavljaju primer principa blizine. Izvor [45].	52
Slika 3.15. RGB kocka boja omogućava definisanje boja u trodimenzionalnom referentnom sistemu čije ose predstavljaju osnovne boje (crvena, zelena i plava). Izvor: [38].	53
Slika 3.16. Pravilo 60-30-10 preporučuje balans boja u pogledu relativnih udela. Primarna boja je bela, sekundarna ljubičasta i akcentovana je siva (poruka u fokusu). Izvor: https://www.slack.com .	54

Slika 4.1. Poređenje JavaScript vrednosti korišćenjem metoda slabe i striktne jednakosti. Izvor [56].	66
Slika 4.2. Šematski prikaz objekata funkcije sa vrednostima svojstava u lancu prototipa.	80
Slika 4.3. DOM reprezentacija Veb stranice. Izvor: [62].	90
Slika 4.4. DOM stablo HTML paraagrafa. Izvor: [63].	91
Slika 4.5. Interakcija korisnika i interfejsa odvija se putem objekata događaja.	92
Slika 4.6. Faze kao načini prosleđivanja događaja kroz DOM stablo. Izvor: [65].	102
Slika 4.7. TypeScript prevodilac generiše JavaScript datoteke koje se dalje izvršavaju.	119
Slika 5.1. Logička šema Angular GKI.	127
Slika 5.2. Struktura korenog modula aplikacije (<i>app.module.ts</i>).	129
Slika 5.3. Predefinisani Angular moduli sa opisom funkcionalnosti. Izvor: [70].	130
Slika 5.4. Organizacija komponenti korišćenjem modula. Izvor: [70].	130
Slika 5.5. Struktura i uloga delova komponente.	132
Slika 5.6. Dijagram koji ilustruje strukturu i relacije između elemenata Angular aplikacije (gore), i prikaz odgovarajućih komponenti u GKI (dole).	133
Slika 5.7. Struktura dekoratora <i>@Component</i> koji označava kontroler komponente i povezuje delove komponente.	135
Slika 5.8. Ugrađena definicija templejta komponente u <i>@Component</i> dekoratoru.	135
Slika 5.9. SPA princip i ugnježdavanje komponenti. Izvor: [71].	138
Slika 5.10. Logička šema povezivanja događaja i svojstva između templejta i kontrolera komponente. Izvor [70].	141
Slika 5.11. Uporedni prikaz načina povezivanja podataka. Podaci se mogu prosleđivati od kontrolera ka templejtu, obrnuto ili u oba smera.	142
Slika 5.12. Različiti formati prikaza vrednosti datuma. Templejt (gore) prikazuje vrednost atributa kontrolera (dole). Izvor: https://angular.io/api/common/DatePipe .	144

Slika 5.13. Povezivanje atributa kontrolera sa atributom taga templejta komponente na primeru standardnih elemenata GKI - slika (gore) i dugme (dole). Izvor: [74].	145
Slika 5.14. Povezivanje događaja između templejta i kontrolera komponente. Izvor: [74].	146
Slika 5.15. Princip rada <i>ngIf</i> direktive na primeru HTML paragrafa.	149
Slika 5.16. Ubrizgivač predstavlja registar snabdevača zavisnosti. Snabdevači su najčešće servisi.	154
Slika 6.1. Kolekcija AM komponenti kao elemenata GKI. Izvor: [78].	164
Slika 6.2. Dokumentacija AM komponente. Izvor: [80].	164
Slika 6.3. Primeri korišćenja AM komponente. Izvor: [80].	165
Slika 6.4. Proces instalacije AM biblioteke komponenti GKI.	165
Slika 6.5. AM instalacija (<i>@angular/material</i>) u <i>package.json</i> datoteci.	166
Slika 6.6. Predefinisane palete boja tema <i>deeppurple-amber.css</i> (levo) i <i>indigo-pink.cs</i> (desno). Izvor: [79].	169
Slika 6.7. Predefinisane palete boja tema <i>pink-bluegrey.css</i> (levo) i <i>purple-green.css</i> (desno). Izvor [79].	169
Slika 6.8. GDM kolekcija ikona omogućava pretragu po različitim kriterijumima. Izvor: [84].	175
Slika 6.9. Izgled AM tabele.	201
Slika 6.10. AM tabela sa funkcijom sortiranja.	204
Slika 6.11. AM tabela sa funkcijom straničenja.	206
Slika 6.12. Šema prosleđivanja podataka od pozivajuće komponente ka dijalogu.	213
Slika 6.13. Horizontalni (levo) i vertikalni (desno) korak.	214
Slika 7.1. FlexBox elementi roditelji (eng. <i>container</i>) i deca (eng. <i>items</i>). Izvor: [85].	223
Slika 7.2. FlexBox koordinatni sistem za raspoređivanje elemenata. Izvor: [85].	224
Slika 7.3. Orijentacije glavne ose roditelja i raspoređivanje dece za različite vrednosti <i>flex-direction</i> naredbe. Izvor: [85].	225
Slika 7.4. Flex-wrap naredba - raspoređivanje elemenata dece u više redova. Izvor: [85].	226

Slika 7.5. FlexBox varijante ravnanja po glavnoj osi i raspodela prostora između elemenata dece. Izvor: [85].	227
Slika 7.6. FlexBox varijante ravnanja po upravnoj osi i raspodela prostora između elemenata dece. Izvor: [85].	228
Slika 7.7. FlexBox varijante ravnanja redove dece po upravnoj osi i raspodela prostora između redova. Izvor: [85].	229
Slika 7.8. FlexBox redosled prikazivanja elemenata dece za različite vrednosti naredbe order. Izvor: [85].	230
Slika 7.9. FlexBox povećanje veličine elemenata dece po glavnoj osi pomoću naredbe flex-grow. Izvor: [85].	231
Slika 7.10. Instalacija FlexLayout biblioteke u package.json datoteci Angular projekta.	232
Slika 7.11. Orijentacije glavne ose roditelja i raspoređivanje dece za različite vrednost fxLayout direktive. Izvor: [85].	233
Slika 7.12. FxLayout direktiva vrši ravnanje po glavnoj i upravnoj osi. Izvor [86].	235
Slika 7.13. FxLayoutAlign direktiva objedinjuje više FlexBox naredbi. Izvor: [85].	236
Slika 7.14. FxFlex direktiva određuje povećavanje, smanjenje i podrazumevanu veličinu elementa deteta u fxLayout roditelju. Izvor: [86].	238
Slika 7.15. FxFlexAlign direktiva omogućava raspoređivanje pojedinačnih elemenata dece po upravnoj osi roditelja. Izvor: [86].	240
Slika 8.1. Terminal prozor kao interfejs komandne linije razvojnog okruženja Visual Studio Code.	242
Slika 8.2. Instalacija Angular interfejsa komandne linije (CLI) u Terminal prozoru razvojnog okruženja.	243
Slika 8.3. Hijerarhijska struktura projekta u razvojnem okruženju.	244
Slika 8.4. Otvorena dva Terminal prozora u razvojnem okruženju.	245
Slika 8.5. Izgled kreiranog Angular projekta u Veb pretraživaču.	245
Slika 8.6. Pregled komponenti aplikacije u razvojnem okruženju.	246
Slika 8.7. Verzije instaliranih biblioteka u datoteci package.json Angular projekta.	247

Slika 8.8. Kreiranje datoteke za registraciju AM komponenti u src/app direktorijumu projekta.	248
Slika 8.9. Korena komponenta i koreni modul aplikacije.	249
Slika 8.10. Implementacija SPA principa u studiji slučaja.	252
Slika 8.11. Predefinisane teme koje dolaze sa instalacijom AM biblioteke.	253
Slika 8.12. Dugme ikona u templejtu korene komponente aplikacije.	255
Slika 8.13. Promena pozicije forme korišćenjem <code>fxLayoutAlign</code> direktive.	257
Slika 8.14. Ispisivanje sadržaja forme u komponenti za registraciju korisnika.	259
Slika 8.15. Izgled forme komponente za prijavu korisnika aplikacije.	263
Slika 8.16. Početni izgled vertikalne linije za navigaciju aplikacije.	265
Slika 8.17. Vertikalna linija za navigaciju. Klikom na dugme linija se naizmenično prikazuje i sakriva.	266
Slika 8.18. Horizontalna linija alata sa linkovima ka komponentama aplikacije i dugmetom za kontrolu prikaza vertikalne linije alata.	267
Slika 8.19. Izgled navigacije u korenoj komponenti aplikacije.	270
Slika 8.20. Konačan izgled navigacije u korenoj komponenti aplikacije.	271
Slika 8.21. Sakrivanje linkova ka komponentama u horizontalnoj liniji alata za prikaze širine manje od 600 piksela.	272
Slika 8.22. Razmena podataka između roditeljske komponente i komponente deteta. U tekućem primeru, roditelj je komponenta treninga, dok je dete komponenta novog treninga.	282
Slika 8.23. Dugme Continue nastavlja tekući trening ukoliko ga korisnik prethodno zaustavi klikom na dugme Stop.	287
Slika 8.24. Kreiranje datoteke <code>exercise.model.ts</code> koja definiše strukturu podataka vežbe treninga.	288
Slika 8.25. Tabela prikaz prethodnih treninga.	291
Slika 8.26. Straničenje tabele prethodnih treninga.	295
Slika 8.27. Struktura komponente za zaustavljanje treninga u hijerarhiji komponenti projekta.	298

Slika 8.28. Kreiranja datoteke teme my-theme.scss.	306
Slika 8.29. Izgled aplikacije nakon primene kreirane teme my-theme.scss.	308
Slika 8.30. Izgled aplikacije nakon primene funkcije mat-light-theme().	308
Slika 8.31. Registracija korisnika (gore) i prijava registrovanog korisnika (dole).	315
Tabela 2.1. Vrste događaja u GKI.	34
Tabela 5.1. Pregled i opis gradivnih elemenata Angular okvira.	126
Tabela 5.2. Faze životnog ciklusa instance komponente i odgovarajuće metode sa interfejsima.	139
Tabela 5.3. Povezivanje podataka – vrsta i smer razmene podataka.	141
Tabela 5.4. Predefinisani moduli rutiranja.	159
Tabela 7.1. FxFlex simboličke konstante sa opisom.	239

Spisak skraćenica

AM	<i>Angular Material.</i>
API	<i>Application Programming Interface.</i>
AR	<i>Augmented reality.</i>
AUI	<i>Attentive user interface.</i>
CDK	<i>Component development kit.</i>
CLI	<i>Command line interface.</i>
CRT	<i>Cathode ray tube.</i>
CSS	<i>Cascading Style Sheets.</i>
CUI	<i>Conversational user interface.</i>
DOM	<i>Document Object Model.</i>
GKI	<i>Grafički korisnički interfejs.</i>
GMD	<i>Google Material Design.</i>
GUI	<i>Graphical user interface.</i>
HMTL	<i>HyperText Markup Language.</i>
HTTP	<i>Hypertext Transfer Protocol.</i>
JIT	<i>Just-in-Time.</i>
MVC	<i>Model-view-controller.</i>
NL	<i>Natural language.</i>
RGB	<i>Red Green Blue.</i>
SASS	<i>Syntactically Awesome Style Sheets.</i>
SPA	<i>Single Page Application.</i>
TUI	<i>Tangible user interface.</i>
VR	<i>Virtual reality.</i>
VSC	<i>Visual Studio Code.</i>
WIMP	<i>Windows, icons, menus, pointer.</i>
W3C	<i>World Wide Web Consortium.</i>

CIP - Каталогизација у публикацији
Народна библиотека Србије, Београд

004.415.2:004.5(075.8)
004.42:004.5(075.8)

ЈОВАНОВИЋ, Млађан, 1981-

Dizajn i programiranje korisničkog interfejsa / Mlađan Jovanović, Nikola
Savanović. - 1. izd. - Beograd : Univerzitet Singidunum, 2023 (Beograd : Birograf). -
339 str. : ilustr. ; 24 cm

Tiraž 800. - Bibliografija: str. 325-330.

ISBN 978-86-7912-792-1

1. Савановић, Никола, 1988- [аутор]

а) Софтвер -- Кориснички интерфејс -- Дизајн б) Кориснички интерфејс --
Програмирање

COBISS.SR-ID 115233801

© 2023

Sva prava zadržana. Nijedan deo ove publikacije ne može biti reprodukovан u bilo
kom vidu i putem bilo kog medija, u delovima ili celini bez prethodne pismene saglasnosti
izdavača.



Dizajn i programiranje korisničkog interfejsa

Mlađan Jovanović
Nikola Savanović

Udžbenik je rezultat višegodišnje saradnje sa studentima 3. godine smjera Računarske Nauke na predmetu Programiranje i dizajn korisničkih interfejsa i studentima 3. godine smjera Softversko i Informaciono Inženjerstvo na predmetu Klijentske Veb aplikacije, Univerziteta Singidunum. Namenjen je i široj publici zainteresovanoj za oblasti dizajna i programiranja korisničkog interfejsa.

Udžbenik najpre daje širi kontekst oblasti korisničkog interfejsa kroz istorijski razvoj i vrste. Zatim opisuje grafički korisnički interfejs (GKI) kao najzastupljeniju vrstu korisničkog interfejsa. Prikazana je arhitektura sa obrascima, pristupi izradi, i ulazni i izlazni mehanizmi GKI. Implementacija principa dizajna u izradi GKI čini korisnički interfejs estetski privlačnijim i lakšim za korišćenje te je njima posvećena posebna sekcija.

Sekcije u nastavku daju pregled programskih jezika za izradu Veb GKI i tehnologija zasnovanih na ovim jezicima. Prikazani su osnovni principi i mehanizmi JavaScript i TypeScript jezika čije je poznavanje neophodno za implementaciju Veb GKI. Jezici predstavljaju osnovu Angular okvira za razvoj korisničkog interfejsa. Angular Material je biblioteka standardnih elemenata GKI sa unapređenim izgledom i funkcijama. Pored toga, opisane su i tehnologije za vizuelni dizajn Veb GKI. Na kraju je korišćenje navedenih tehnologija demonstrirano kroz prototip aplikacije.

