



IT255 - VEB SISTEMI 1

Veb sistemi i HTML

Lekcija 01

PRIRUČNIK ZA STUDENTE

IT255 - VEB SISTEMI 1

Lekcija 01

WEB SISTEMI I HTML

- ✓ Veb sistemi i HTML
- ✓ Poglavlje 1: Obavezni elementi strukture HTML dokumenta
- ✓ Poglavlje 2: Sadržaj HTML dokumenata
- ✓ Poglavlje 3: Liste u HTML-u
- ✓ Poglavlje 4: Tabele i okviri u HTML-u
- ✓ Poglavlje 5: HTML forme
- ✓ Poglavlje 6: Pokazna vežba - raspoređivanje elemenata u HTML-u
- ✓ Poglavlje 7: Individualna vežba
- ✓ Poglavlje 8: Domaći zadatak 1
- ✓ Zaključak

Copyright © 2017 – UNIVERZITET METROPOLITAN, Beograd. Sva prava zadržana. Bez prethodne pismene dozvole od strane Univerziteta METROPOLITAN zabranjena je reprodukcija, transfer, distribucija ili memorisanje nekog dela ili čitavih sadržaja ovog dokumenta., kopiranjem, snimanjem, elektronskim putem, skeniranjem ili na bilo koji drugi način.

Copyright © 2017 BELGRADE METROPOLITAN UNIVERSITY. All rights reserved. No part of this publication may be reproduced, stored in a retrieval system or transmitted in any form or by any means, electronic, mechanical, photocopying, recording, scanning or otherwise, without the prior written permission of Belgrade Metropolitan University.

❖ Uvod

UVOD

HTML je standardni jezik za kreiranje veb stranica.

HTML je standardni jezik za kreiranje veb stranica. Ovaj jezik se koristi za pisanje koda na osnovu kojeg veb stranice postaju funkcionalne. Međutim, ovaj jezik često nije dovoljan da omogući prikazivanje bogatog, stilizovanog i dinamičkog ponašanja neke veb stranice. Iz navedenog razloga se često kombinuje sa drugim jezicima:

- *CSS* - jezik stilova za HTML;
- *JavaScript* - skripting jezik za kreiranje dinamičkog HTML sadržaja.

Za razvoj i održavanje HTML je zadužen konzorcijum za veb (*W3C, World Wide Web Consortium*). Ovaj jezik je prošao kroz više iteracija razvoja i trenutno je aktuelna verzija *HTML5* koja je osnov savremenog veb razvoja i nalazi se u konstantnom razvoju i unapređenju.

HTML je nastao kroz dalji razvoj i pojednostavljinje definicije jezika *SGML (Standard Generalized Markup Language)*, standardizovani uopšteni jezik za označavanje). SGML je definisan kao standard sa zadatkom opisivanja dokumenta koji se objavljuju na vebu.

Sami počeci HTML jezika bili su vezani za ograničen set funkcionalnosti, pružao je uglavnom elementarne stvari za:

- označavanje i
- formatiranje teksta (paragrafi, naslovi, citati itd.).

Veb se veoma brzo našao u rapidnom razvoju tako da je brzo došlo do potrebe za bogatijim i dinamičnjim veb sadržajem. Iz navedenog razloga bilo je neophodno razviti i HTML standard. Tada su standardu dodati elementi za opis tabela, slika, slojeva, napredno formatiranje teksta itd.

Istorijski gledano, sledeće HTML specifikacije su veoma bitne:

- HTML2 - novembar 1995. (link specifikacije: <https://web.archive.org/web/20050305063804/http://www.ietf.org/rfc/rfc1866.txt>);
- HTML3.2 - januar 1997. (link specifikacije: <https://www.w3.org/TR/2018/SPSD-html32-20180315/>);
- HTML4.0 - decembar 1997. (link specifikacije: <https://www.w3.org/TR/REC-html40-971218/>);
- HTML4.0.1 - decembar 1999. (link specifikacije: <https://www.w3.org/TR/html401/>);
- ISO/IEC 15445:2000 - maj 2000. (link specifikacije: <https://www.scss.tcd.ie/misc/15445-15445.HTML>)
- HTML5 - maj 2008 i u daljem je razvoju (link specifikacije: <https://html.spec.whatwg.org/multipage/>).

PRIMENA HTML JEZIKA

HTML se široko koristi za formatiranje veb stranica uz pomoć različitih oznaka (tagova).

Izvorno je HTML razvijen sa namerom da definiše strukturu dokumenata kao što su: naslovi, paragrafi, liste i tako dalje sa ciljem lakše razmene naučnih informacija između istraživača.

Sada se HTML široko koristi za formatiranje veb stranica uz pomoć različitih oznaka (tagova) dostupnih na HTML jeziku.

HTML danas predstavlja obaveznu temu izučavanja da bi studenti, kao budući profesionalci, postali odlični softverski inženjeri, posebno kada rade u domeni veb razvoja.

Slede neke od ključnih prednosti učenja HTML-a:

- **kreiranje veb sajtova** - moguće je kreirati vlastiti ili redizajnirati postojeći veb sajt primenom znanja stečenog savladavanjem HTML jezika;
- **angajovanje na poslovima veb dizajnera** - znanje stečeno savladavanjem HTML jezika i jezika stilova (CSS) omogućava studentima lakše zaposlenje u veb industriji;
- **razumevanje veba** - znanje stečeno savladavanjem HTML jezika može biti odličan osnov za razumevanje koncepta optimizacije veb sajtova, podizanja nivoa brzine i pouzdanosti izvršavanja istih;
- **učenje drugih jezika** - HTML često nije dovoljan pa programer stiče dodatnu širinu izučavanjem drugih jezika, poput: *JavaScript, php, Angular (TypeScript)* i tako dalje.

Kao što je već istaknuto, HTML je široko primenljiv jezik na vebu. Sledi nekoliko teza kojima je istaknuta česta upotreba ovog jezika:

- **razvoj veb strana** - skoro svaka stranica sadrži HTML tagove koje interpretira veb pregledač sa ciljem prikazivanja odgovarajućeg sadržaja;
- **Internet navigacija** - HTML obezbeđuje tagove preko kojih je omogućena navigacija sa jedne veb stranice na drugu;
- **responzivan UI** - HTML podjednako dobro funkcioniše na različitim platformama: veb, mobilni uređaji, tableti, desktop, bazirajući se na konceptima responzivnog dizajna;
- **razvoj igara** - HTML5 je nezaobilazan standard koji je našao veliku primenu i u industriji igara.

Postoje još brojne primene ali će o njima biti diskutovano kako dalje bude odmicalo izlaganje predmeta koji se bave izučavanjem veb sistema.

UVODNI VIDEO

Trajanje video snimka: 5min 58sek

Ova lekcija sadrži video materijal. Ukoliko želite da pogledate ovaj video morate da otvorite LAMS lekciju.

▼ Poglavlje 1

Obavezni elementi strukture HTML dokumenta

VIDEO PREDAVANJE ZA OBJEKAT "OBAVEZNI ELEMENTI STRUKTURE HTML DOKUMENTA"

Trajanje video snimka: 16min 9sek

Ova lekcija sadrži video materijal. Ukoliko želite da pogledate ovaj video morate da otvorite LAMS lekciju.

TAGOVI I ATRIBUTI

Tagovi i atributi su osnovni elementi HTML jezika.

Nakon uvodnog razmatranja, moguće je preći na konkretno izučavanja elemenata i funkcionalnosti jezika HTML. Kada se počne diskusija u vezi sa navedenim jezikom neophodno je istaći pre sve svega osnovne elemente jezika. U konkretnom slučaju to su tagovi i atributi.

Ova dva elementa funkcionišu zajedno, pa ipak, imaju značajno različitu ulogu unutar jednog HTML dokumenta:

- *HTML tagovi* - predstavljaju oznake za početak svakog HTML elementa u formi naziva elementa smeštenog u okviru uglastih zagrada. Primer jedne takve oznake može biti tag za isticanje nekog teksta `<h1>` (h je skraćenica od *heading*). Većina tagova mora biti zatvorena na kraju elementa kojeg opisuje. Završni tag za element `<h1>` glasi `</h1>`.
- *HTML atributi* - sadrže dodatne informacije unutar tagova. Atributi se nalaze u početnim tagovima i na taj način zaokružuju definiciju taga.

Primer dodavanja atributa, u neki početni HTML tag, može biti ilustrovan sledećim malim HTML listingom.

```

```

Ovde je moguće primetiti da su u tag `` za dodavanje slike u HTML stranicu dodata dva atributa:

- `src` - označava lokaciju sa koje se slika učitava;

- *alt* - označava alternativni tekst za učitanu sliku.

VAŽNO ZA PAMĆENJE!!!

1. Većina tagova mora da bude otvorena (<tag>), a zatim i zatvorena (</tag>);
2. Unutar tagova za otvaranje i zatvaranje nalaze se informacije o elementu, na primer malo teksta dodatog između ova dva taga;
3. Kada se koristi veći broj tagova, tagovi se zatvaraju obrnutim redosledom od otvaranja. Navedeno može biti ilistrovano na sledeći način:

```
<spoljasni><utrasnji>Ovo je veoma važno!!!!</unutrasnji></spoljasni>
```

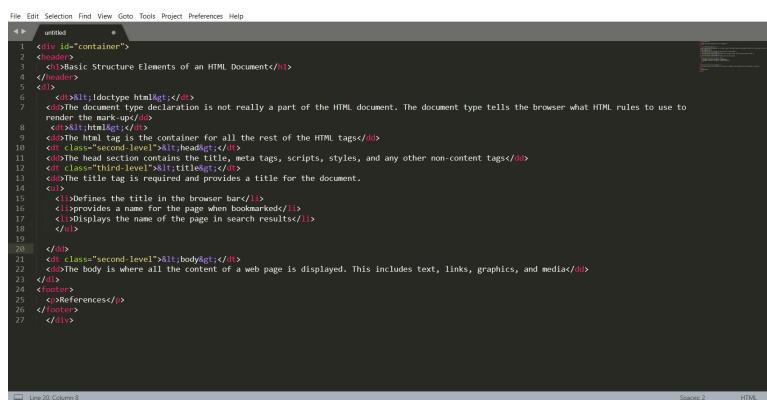
EDITORI ZA PISANJE HTML KODA

Postoje brojni tekst alati koji omogućavaju pisanje i kasniju primenu HTML koda.

Postoje brojni tekst alati koji omogućavaju pisanje i kasniju primenu *HTML* koda. Svi imaju neke svoje prednosti i nedostatke, a takođe postoje preporuke za izbegavanje tekst procesora, kao što je *Microsoft Word*, za pisanje, izmenu i čuvanje koda HTML stranica.

U sledećem izlaganju će biti navedeni neki tekst editori pomoću kojih je moguće započeti HTML razvoj:

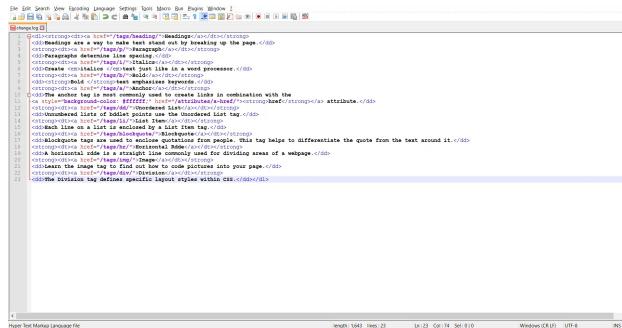
- 1. *Sublime Text 3* (stranica za preuzimanje : <https://www.sublimetext.com/>) - Ovo je besplatan tekst editor dostupan na različitim platformama: Windows, Mac ili Linux. Prilagodljiv je, odličan za početnike, sadrži veliki broj šema i boja. Nedostaci: nema dostupne trake sa alatkama ili kontrolne table.



Slika 1.1 Sublime Text 3 tekst editor

- 2. *Notepad ++* (stranica za preuzimanje: <https://notepad-plus-plus.org/downloads/>) - Predstavlja čest izbor za pisanje, izmenu i čuvanje koda HTML stranica. Veoma je jednostavan, sadrži alatku za automatsko kompletiranje teksta, moguće ga je proširiti

brojnim korisnim dodacima. Nedostaci: može biti težak za primenu kada su početnici u pitanju, nema podršku za Mac.



Slika 1.2 Notepad ++ tekst editor

Ovo su samo neki predlozi. Moguće je koristiti i "običan" *Notepad* (koji će biti i korišćen za pisanje pokaznih primera i vežbi) koji dolazi instaliran sa *Windows* operativnim sistemom. Takođe, preporučuje se i primena tekstu editora *Komodo* (link za preuzimanje: <https://www.activestate.com/products/komodo-ide/downloads/edit/>), uglavnom, stvar je navike i preferencija HTML programera.

ORGANIZACIJA HTML STRANICE

Kreiranje koda koji je obavezan za svaku HTML stranicu.

Analizu i diskusiju vezanu za osnovnu strukturu HTML dokumenta započećemo otvaranjem izabranog tekstu editora i dodavanjem koda koji je obavezan za svaku *HTML* stranicu. Sledećom listom su prikazani obavezni elementi svake HTML stranice:

1. **<!DOCTYPE html>** - Ovaj tag ukazuje na jezik kojim će stranice niti pisane. U konkretnom slučaju to je [HTML5](#) ;
2. **<html>** — Ovaj tag ukazuje na početak HTML koda;
3. **<head>** — Ovaj tag predstavlja početak lokacije u HTML dokumentu unutar koje se dodaju meta podaci — uglavnom namenjene pretraživačima i drugim programima;
4. **<body>** - Ovaj tag otvara lokaciju u *HTML* kodu u kojoj se kodira sadržaj *HTML* stranice.

Veoma je bitno napomenuti da kada dođe trenutak za zatvaranje navedenih tagova, oni se zatvaraju u obrnutom poretku od navedenog: **</body>** , **</head>** i konačno **</html>** .

Osnovna struktura HTML dokumenta može biti ilustrovana sledećom slikom.



Slika 1.3 Šematski prikaz organizacije HTML stranice

Sada kada je sve ilustrovano moguće je pristupiti i kodiranju jednostavne HTML stranice, upravo na osnovu priložene šeme.

KREIRANJE PRVE HTML STRANICE

Pristupa se kodiranju jednostavne HTML stranice, upravo na osnovu priložene šeme.

Pristupa se kodiranju jednostavne HTML stranice, upravo na osnovu priložene šeme, a sa ciljem demonstracije dodavanja i primene dopunskih HTML elemenata.

Sledećim listingom je data inicijalna struktura jednostavne veb stranice:

```
<!DOCTYPE html>
<html>
<head>
<title>Page Title</title>
</head>
<body>

<h1>Ovo je zaglavlje</h1>
<p>Ovo je paragraf</p>

</body>
</html>
```

Kao što je moguće primetiti, kod sadrži tag `<!DOCTYPE html>` koji ukazuje da se za kodiranje stranice koristi **HTML5**. U nastavku, tagom `<html>` je ukazano na početak pisanja HTML koda.

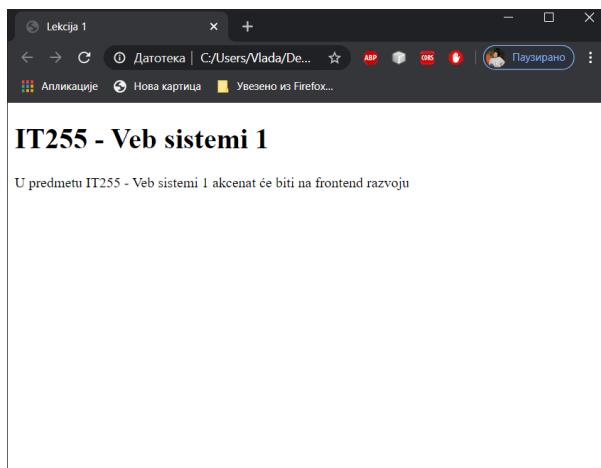
Prva korekciju koju je moguće napraviti jeste izmena naziva veb stranice koja se kreira. Ova informacija se nalazi u tagu `<title>` koji je ugnježden u tag `<head>`. Sledećim listingom je prikazana učinjena korekcija prema navedenim smernicama:

```
<title>Lekcija 1</title>
```

Sada se dolazi do dela koda kojim su određeni detalji koji su vidljivi krajnjem korisniku. Vidljivi deo stranice se kodira pod tagom `<body>`. Za početak, sadržaj je veoma jednostavan, predstavlja dve linije teksta prikazane u formi istaknutog teksta (header) i paragrafa. Sada je moguće napraviti korekcije i umesto početnog teksta dodati željeni sadržaj. Nakon korekcije tagovi `<h1>` i `<p>` dobijaju sledeću formu.

```
<h1>IT255 - Veb sistemi 1</h1>
<p>U predmetu IT255 - Veb sistemi 1 akcenat će biti na frontend razvoju</p>
```

Učitavanjem kreirane stanice u veb pregledač, dobija se sledeći rezultat:



Slika 1.4 Izgled prve HTML stranice

KORIŠĆENJE OBAVEZNIH HTML TAGOVA

Moguće je dodati meta podatke unutar posmatranog taga head.

U prethodnom izlaganju je prikazana osnovna struktura HTML dokumenta zajedno sa dodatim sadržajem.

Kao prvi korak uvek je moguće istaći formiranje naziva stranice unutar taga `<title>` na način na koji je to opisano u prethodnom izlaganju. Nakon toga moguće je dodati meta podatke unutar posmatranog taga `<head>`. Ovo se postiže ugnježdavanjem taga `<meta>`. Na ovom mestu se čuvaju informacije koje su direktno u vezi sa dokumentom: standard kodiranja

karaktera (osobina *charset*), naziv autora (meta informacije - osobina *author*) i odgovarajući opis(osobina *description*).

Proširuje se prethodni, inicijalni HTML kod, sledećom dopunom:

```
<head>
<title>Lekcija 1</title>
<meta charset="UTF-8">
<meta name="description" content="Ovo je stranica uvodnog primera! Sledi dalja razrada HTML koda.">
<meta name="author" content="Vladimir Milićević"
</head>
```

Vrši se analiza priloženog koda:

- Linijom koda 3 definisan je standard za kodiranje karaktera *UTF-8* ;
- Linijom koda 4 dat je kratak opis namene stranice. Ovde je bitno da se ne preteruje, obično opis ima formu do dva reda;
- Linijom koda 5 istaknute su informacije u vezi sa autorom stranice.

Sledećim video materijalom moguće je zaokružiti razmatranje u vezi sa kreiranjem HTML stranica sa osnovnim elementima.

Ova lekcija sadrži video materijal. Ukoliko želite da pogledate ovaj video morate da otvorite LAMS lekciju.

▼ Poglavlje 2

Sadržaj HTML dokumenata

ELEMENTI TAGA BODY

Detalji koji su vidljivi učitavanjem stranice, kodiraju se unutar taga BODY.

U nastavku izlaganja, akcenat će biti na sadržaju koji je vidljiv kada veb pregledač učita i prikaže sadržaj veb stranice. Detalji koji su vidljivi učitavanjem HTML stranice, kodiraju se unutar taga `<body>`.

Upravo iz navedenog razloga ovaj objekat učenja će se baviti sledećim temama:

- dodavanje i isticanje teksta;
- rad sa slikama u HTML stranicama;
- rad sa linkovima u HTML stranicama;
- rad sa listama u HTML stranicama (zbog svoje važnosti, biće obrađeno kao poseban objekat učenja);
- rad sa formama u HTML stranicama (zbog svoje važnosti, biće obrađeno kao poseban objekat učenja).

Jedan od osnovnih elementata HTML sintakse je komentar. To je kod kojeg ignoriše veb pregledač i piše se na sledeći način:

`<!-- Ovo je HTML komentar -->`.

ISTICANJE TEKSTA (HEADINGS) U HTML STRANICAMA

Isticanje teksta u HTML-u omogućeno je primenom tagova koji se jednim nazivom zovu headings.

Isticanje (naglašavanje) teksta u HTML - u omogućeno je primenom tagova (njih ukupno 6) koji se jednim nazivom zovu headings. Isticanje teksta je moguće ako se upotrebi neki od tagova koji se nalazi na sledećoj slici.

- `<h1>`
- `<h2>`
- `<h3>`
- `<h4>`
- `<h5>`
- `<h6>`

Slika 2.1 Skup headings tagova

Veličina teksta, koji se prikazuje primenom navedenih tagova, određen je od najvećeg ka najmanjem krećući se od vrha slike ka njenom dnu. Otuda je veoma lako izvući zaključak da se najznačajniji naslovi prikazuju tagovima `<h1>` i `<h2>`, a podnaslovi ili manje bitni naslovi mogu da se prikažu primenom preostalih headings tagova.

Bot pretraživači koriste navedeni redosled da bi otkrili koji tekst ima najveći značaj.

Sve tagove headings tagove na delu možemo da vidimo ukoliko ih angažujemo na način prikazan sledećim listingom:

```
<!DOCTYPE html>
<html>
<body>

<h1>Heading 1</h1>
<h2>Heading 2</h2>
<h3>Heading 3</h3>
<h4>Heading 4</h4>
<h5>Heading 5</h5>
<h6>Heading 6</h6>

</body>
</html>
```

Učitavanjem ovakvog koda, u veb pregledač, dobija se sledeći rezultat:

Heading 1

Heading 2

Heading 3

Heading 4

Heading 5

Heading 6

Slika 2.2 Prikaz teksta pomoću headings tagova

DODAVANJE NASLOVA U VLASTITI HTML PRIMER

Dodavanje naslova u vlastiti HTML primer započet u prethodnom izlaganju.

Kada je naučeno da se rukuje tagovima za isticanje teksta, moguće je pristupiti njihovoj primeni tokom razvoja veb stranice koja je bila predmet diskusije u prethodnom objektu učenja.

Sve što je potrebno jeste korišćenjem teksta editora ponovo otvoriti datoteku index.html, pronaći lokaciju taga `<body>` i unutar njega dodati sledeći kod:

```
<h1>IT255 - Veb sistemi 1</h1>
<h2>Prvi pokazni primer</h2>
```

Na ovaj način postavljen je naslov stranice "IT255 - Veb sistemi 1", kao i slabije istaknut tekst u formi podnaslova "Prvi pokazni primer". Sada je potrebno snimiti stranicu i ponovo je učitati u veb pregledač. Rezultat će biti identičan kao na sledećoj slici.

IT255 - Veb sistemi 1

Prvi pokazni primer

Slika 2.3 Primena headings tagova u pokaznom primeru

Na ovaj način je demonstrirana jednostavna primena tagova `<h1>` i `<h2>` za isticanje teksta. Međutim, ovo je sam početak rada sa tekstom u [HTML](#) stranicama i postoji još mnogo korisnih tagova koje je moguće primeniti na tekst. U nastavku će biti više reči o njima.

DODAVANJE TEKSTA U HTML DOKUMENT

Dodavanje teksta u HTML dokument omogućeno je primenom elementa koji započinje otvorenim paragraf tagom.

Dodavanje teksta u HTML dokument omogućeno je primenom elementa koji započinje otvorenim paragraf tagom `<p>`. Tekst koji predstavlja regularni sadržaj HTML stranice piše se unutar ovakvog elementa.

Sledećim proširenjem započetog primera moguće je na jednostavan način demonstrirati dodavanje teksta u HTML dokument.

```
|| <!DOCTYPE html>
<html>
<head>
<title>Lekcija 1</title>
<meta charset="UTF-8">
<meta name="description" content="Polje sadrži informacije o stranici, obično u dva reda.">
<meta name="author" content="Vladimir Milićević"
</head>

<body>
<h1>IT255 - Veb sistemi 1</h1>
<h2>Prvi pokazni primer</h2>
<p>U predmetu IT255 - Veb sistemi 1 akcenat će biti na frontend razvoju</p>
<p>U predmetu IT355 - Veb sistemi 2 akcenat će biti na backend razvoju</p>
</body>
</html>
```

Slika 2.4 Dodavanje teksta kao novog paragrafa u HTML dokument

Iz priloženog listinga je moguće primetiti da je obavljeno dodavanje dva paragrafa teksta (istaknuto crvenom bojom) u postojeći primer preko kojih su u kratkim crtama opisani predmeti koji se bave izučavanjem veb sistema.

Nakon što je obavljena ilustrovana modifikacija koda stranice, neophodno je ponovo snimiti promene, a nakon toga obaviti ponovno učitavanje stranice u veb pregledač. Rezultat učitavanja je ilustrovan sledećom slikom.

IT255 - Veb sistemi 1

Prvi pokazni primer

U predmetu IT255 - Veb sistemi 1 akcenat će biti na frontend razvoju

U predmetu IT355 - Veb sistemi 2 akcenat će biti na backend razvoju

Slika 2.5 Dodavanje teksta u HTML dokument

Kada je naučeno kako je moguće obaviti jednostavno dodavanje teksta u HTML dokument, neophodno je pozabaviti se mehanizmima njegovog formatiranja, a to će upravo biti predmet diskusije u narednom izlaganju.

FORMATIRANJE TEKSTA U HTML DOKUMENTIMA

Postoje brojne HTML alatke za formatiranje teksta.

U prethodnom izlaganju je naučeno kako je moguće obaviti jednostavno dodavanje teksta u HTML dokument. Sada se diskusija pomera dalje i neophodno je pozabaviti se mehanizmima njegovog formatiranja. Postoje brojne HTML alatke koje je moguće primeniti na dodati tekst sa ciljem njegove kvalitetnije prezentacije.

Sledi lista najčešće korišćenih tagova za formatiranje teksta u HTML stranicama:

- **** - značenje ovog taga je *BOLD* (podebljano - istaknuto). Ovaj tag se koristi za isticanje veoma bitnih informacija unutar teksta;
- **** - značenje ovog taga je *STRONG* (snažno - istaknuto, podebljano). Tag se koristi na sličan način kao i prethodni - za isticanje pojedinih detalja u tekstu;
- **<i>** - značenje ovog taga je *ITALIC* (ukošeno). Tag se koristi za obeležavanje teksta koji će biti prikazan u ukošenom formatu;
- **** - značenje ovog taga je *Emphasised Text* (istaknuti tekst). Efekat na tekst je sličan kao kod primene prethodnog taga.
- **<mark>** - značenje ovog taga je *Marked Text* (markirani tekst). Označavanjem teksta, primenom ovog taga, postavlja se pozadina za tekst (u žutoj boji po osnovnim podešavanjima);
- **<small>** - značenje ovog taga je *Small Text* (mali tekst). Označavanjem teksta, primenom ovog taga, obavlja se skupljanje teksta, u smislu smanjivanja veličine fonta kojim je prikazan;
- **<strike>** - značenje ovog taga je *STRIKED OUT TEXT* (precrtni tekst). Ovaj tag se koristi za postavljanje linije preko teksta kojeg obuhvata;
- **<u>** - značenje ovog taga je *UNDERLINED TEXT* (podvučeni tekst). Ovaj tag se koristi za postavljanje linije ispod teksta kojeg obuhvata;
- **<ins>** - značenje ovog taga je *INSERTED TEXT* (umetnuti tekst). Ovaj tag se koristi za postavljanje linije ispod teksta kojeg obuhvata za isticanje da se radi o umetnutom tekstu;
- **<sub>** - značenje ovog taga je *SUBSCRIPT TEXT* (indeksni tekst). Ovaj tag se koristi za postavljanje teksta kojeg obuhvata u formi indeksa;
- **<sup>** - značenje ovog taga je *SUPERSCRIPT TEXT* (eksponencijalni tekst). Ovaj tag se koristi za postavljanje teksta kojeg obuhvata u formi eksponenta;

VAŽNO!!!

Svaki od navedenih tagova mora da bude zatvoren na odgovarajućem mestu u HTML kodu.

PRIMER FORMATIRANJA TEKSTA

Na postojeći tekst biće primjenjeni izvesni tagovi za njegovo formatiranje.

Nastavlja se razvoj započetog primera. Na postojeći tekst, dodat u prethodnom izlaganju preko zaglavlja i dva paragrafa, biće primjenjeni izvesni tagovi za njegovo formatiranje.

Analizira se sledeći listing:

```
***  
<body>  
<h1><mark>IT255 - Veb sistemi 1</mark></h1>  
<h2><u>Prvi pokazni primer</u></h2>  
<p>U predmetu <strong>IT255 - Veb sistemi 1</strong> akcenat će biti na  
<em>frontend</em> razvoju</p>  
<p>U predmetu <strong>IT355 - Veb sistemi 2</strong> akcenat će biti na  
<em>backend</em> razvoju</p>  
  
</body>  
****
```

Prvo što je moguće primetiti jeste da je zaglavlj (naslov) stranice obeležen tagom `<mark>` (linija koda 3). Ovo znači da će ispod teksta naziva stranice biti postavljena pozadina - tekst će biti "markiran".

U nastavku, na podnaslov (tekst unutar taga `<h2>`), primjenjen je tag `<u>` što znači da će taj tekst biti podvučen (linija koda 4).

U linijama koda 5 i 6 na tekst su primjenjeni tagovi `` za podebljavanje i `` za ukošavanje.

Nakon što je obavljena ilustrovana modifikacija koda stranice, sa ciljem formatiranja dodatog teksta, neophodno je ponovo snimiti promene, a nakon toga obaviti ponovno učitavanje stranice u veb pregledač. Rezultat učitavanja je ilustrovan sledećom slikom.

IT255 - Veb sistemi 1

Prvi pokazni primer

U predmetu **IT255 - Veb sistemi 1** akcenat će biti na *frontend* razvoju

U predmetu **IT355 - Veb sistemi 2** akcenat će biti na *backend* razvoju

Slika 2.6 Primer formatiranja teksta

Pošto tekst predstavlja najzastupljeniji sadržaj svake prezentacije, pa i prezentacije na vebu, veoma je važno uspešno savladati mehanizme njegovog formatiranja sa ciljem prikazivanja u željenoj formi.

Kada je to urađeno na uspešan način moguće je preći na dodavanje naprednjeg sadržaja u HTML dokument, poput linkova, slika i ostalog.

PRIMENA LINKOVA U HTML DOKUMENTIMA

Brojne internet prezenatacije su sastavljene od velikog broja linkova.

Kada se pogleda struktura većine veb stranica moguće je primetiti veliku količinu linkova koji se nalaze na njima. **Link** je kontrola pomoću koje se vrši navigacija ka drugim stranicama ili detaljima unutar iste stranice. Kada se govori o navigaciji ka drugim stranicama ona može biti unutar istog veb sajta (vеб aplikације) ili ka stranicama koje pripadaju eksternim veb sistemima.

Linkovi su uključeni u formi atributa početnog taga **<a>** (značenje ovog taga je **ANCHOR** - sidro). U trenutnoj diskusiji ovo je prvi element koji koristi atribut i po tome se razlikuje od svih prethodno navedenih i korišćenih tagova.

Početni **<a>** tag ima sledeći opšti format:

```
<a href="Ovde dodajete adresu linka">Ovde dodajete tekst vašeg linka </a>
```

Ako se pogleda pažljivo priloženi **HTML** listing moguće je primetiti da se pod početnim tagom javlja atribut **href** koji ukazuje na adresu stranice koja će biti učitana nakon klika na link. Između početnog **<a>** i završnog taga **** ovog elementa nalazi se tekst koji će biti vidljiv korisnicima i preko kojeg će biti moguće obaviti akciju "klik".

Kao što je već istaknuto, linkovima je moguća navigacija:

- unutar iste veb aplikacije;
- prema stranicama eksterne veb aplikacije.

Upravo prema navedenim smernicama će teći naredno izlaganje i demonstracija primene navedenih tipova navigacije,

NAVIGACIJA IZMEĐU STRANICA ISTOG VEB SISTEMA

Linkovima je omogućena navigacija između stranica istog veb sistema.

Linkovima je omogućena navigacija između stranica istog veb sistema. Za početak je neophodno kreirati još jednu jednostavnu veb stranicu koja će biti učitana nakon obavljenog klika na odgovarajući link iz početne veb stranice. Sledi listing nove stranice **druga.html**:

```
<!DOCTYPE html>
<html>
```

```
<head>
<title>Stranica 2</title>
</head>

<body>
<h1><mark>Ovo je druga veb stranica</mark></h1>

</body>
</html>
```

Nakon kreiranja stranice, ka kojoj će teći navigacija, neophodno je obaviti kreiranje linka u početnoj stranici (index.html) koji će omogućiti navigaciju ka stranici čiji je kod upravo prikazan.

Opšti oblik linka za navigaciju unutar istog veb sistema može biti prikazan na sledeći način:

```
<a href="naziv_foldera/naziv_stranice.html">Stranica2</a>
```

Atributom *href* je određena putanja, ka odgovarajućem folderu u kojem se čuva datoteka stranice na koju želimo da pređemo, a tekstrom ""Stranica2" je određen tekst linka koji će biti vidljiv u početnoj stranici.

Pošto je detaljno objašnjena upotreba unutrašnjih linkova i procedura njihovog kreiranja, moguće je sada pristupiti njihovom kreiranju i dodavanju u već započeti veb projekat. U tekućem pokaznom primeru, neophodno je dodati sledeći kod u okviru postojećeg:

```
***  
<p>U predmetu <strong>IT355 - Veb sistemi 2</strong> akcenat će biti na  
<em>backend</em> razvoju</p>  
  
<a href = "druga.html"> Otvorite sledeću stranicu</a>  
  
</body>  
****
```

Razlog zašto je naveden samo naziv stranice, bez foldera, jeste taj što se obe stranice čuvaju unutar istog foldera. Sve je spremno za demonstraciju navigacije između stranica istog veb sistema.

DEMONSTRACIJA NAVIGACIJE IZMEĐU STRANICA ISTOG VEB SISTEMA

Sledi demonstracija navigacije između stranica istog veb sistema.

Sledi demonstracija navigacije između stranica istog veb sistema. Nakon što je obavljena ilustrovana modifikacija koda stranice, sa ciljem dodavanja odgovarajućeg navigacionog linka, neophodno je ponovo snimiti promene, a nakon toga obaviti ponovno učitavanje stranice u veb pregledač. Rezultat učitavanja je ilustrovan sledećom slikom.

IT255 - Veb sistemi 1

Prvi pokazni primer

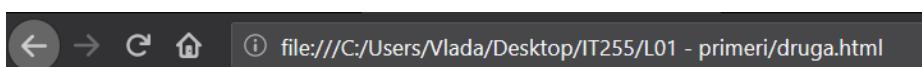
U predmetu **IT255 - Veb sistemi 1** akcenat će biti na *frontend* razvoju

U predmetu **IT355 - Veb sistemi 2** akcenat će biti na *backend* razvoju

[Otvorite sledeću stranicu](#) ovo je link

Slika 2.7 Link koji je dodat u početnu stranicu

Za testiranje funkcionalnosti dodatog linka dovoljno je obaviti klik mišem direktno na tekst linka. Rezultat navedene aktivnosti je prikazan sledećom slikom:



Ovo je druga veb stranica

Slika 2.8 Prelazak na drugu stranicu nakon klika na link

Kroz prikazanu demonstraciju potvrđuje se da je zadatak navigacije između stranica istog veb sistema uspešno obavljen. Sledi novi zadatak - navigacija ka stranicama eksternog sistema, kao veoma bitna HTML funkcionalnost. Upravo će to biti predmet diskusije u narednom izlaganju.

NAVIGACIJA IZMEĐU STRANICA RAZLIČITIH VEB SISTEMA

Linkovima je omogućena navigacija između stranica različitih veb sistema.

Linkovima je, takođe, omogućena i navigacija između stranica različitih veb sistema. Postavlja se sledeći zahtev za dopunom aktuelnog primera - u stranicu *index.html* neophodno je dodati link koji omogućava navigaciju ka početnoj stranici veb sajta našeg Univerziteta.

Opšti oblik linka za navigaciju između stranica različitih veb sistema može biti prikazan na sledeći način:

```
<a href="link_ka_stranici_drugog_vez_sistema">Tekst na koji treba kliknuti</a>
```

Pošto je detaljno objašnjena upotreba spoljašnjih linkova i procedura njihovog kreiranja, moguće je sada pristupiti njihovom kreiranju i dodavanju u već započeti veb projekat. U tekućem pokaznom primeru, neophodno je dodati sledeći kod u okviru postojećeg:

```
***
```

```
<p>U predmetu <strong>IT355 - Veb sistemi 2</strong> akcenat će biti na  
<em>backend</em> razvoju</p>

<a href = "druga.html"> Otvorite sledeću stranicu</a> <br/>
<a href="https://www.metropolitan.ac.rs/">Sajt Univerziteta Metropolitan</a>

</body>
```

```
****
```

Iz priloženog listinga je moguće primetiti kako je sada vrednost atributa *href* eksterni link koji omogućava navigaciju ka početnoj stranici veb sajta našeg Univerziteta

Primećuje se primena još jednog taga

. Ovaj tag nije potrebno zatvoriti iako ga je moguće pisati i u obliku *
* . Značenje ovog taga je *BREAK LINE* i njegov zadatak je da omogući prikazivanje teksta u novom redu.

DEMONSTRACIJA NAVIGACIJE IZMEĐU STRANICA RAZLIČITIH VEB SISTEMA

Sledi demonstracija navigacije između stranica različitih veb sistema.

Sledi demonstracija navigacije između stranica različitih veb sistema. Nakon što je obavljena ilustrovana modifikacija koda stranice, sa ciljem dodavanja odgovarajućeg navigacionog linka ka stranici eksternog veb sistema, neophodno je ponovo snimiti promene, a nakon toga obaviti i ponovno učitavanje stranice u veb pregledač. Rezultat učitavanja je ilustrovan sledećom slikom.

IT255 - Veb sistemi 1

Prvi pokazni primer

U predmetu **IT255 - Veb sistemi 1** akcenat će biti na *frontend* razvoju

U predmetu **IT355 - Veb sistemi 2** akcenat će biti na *backend* razvoju

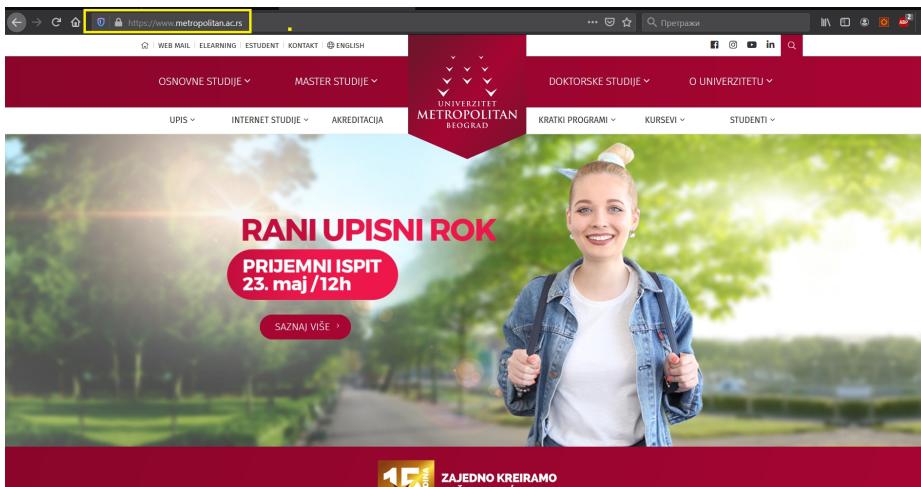
Otvorite sledeću stranicu

Sajt Univerziteta Metropolitan

link ka stranici eksternog
vеб sistema

Slika 2.9 Link koji je dodat u početnu stranicu za navigaciju ka eksternoj stranici

Za testiranje funkcionalnosti dodatog linka dovoljno je obaviti klik mišem direktno na tekst linka i, ukoliko je sve urađeno na pravi način, u veb pregledaču će ubrzo biti učitana početna stranica sajta našeg Univerziteta. Navedeno je moguće ilustrovati sledećom slikom.



Slika 2.10 Početna stranica sajta Univerziteta Metropolitan

Sada je moguće zaključiti da je izučavanje upotrebe linkova u HTML stranicama uspešno završeno.

FORMAT SLIKA U HTML DOKUMENTU

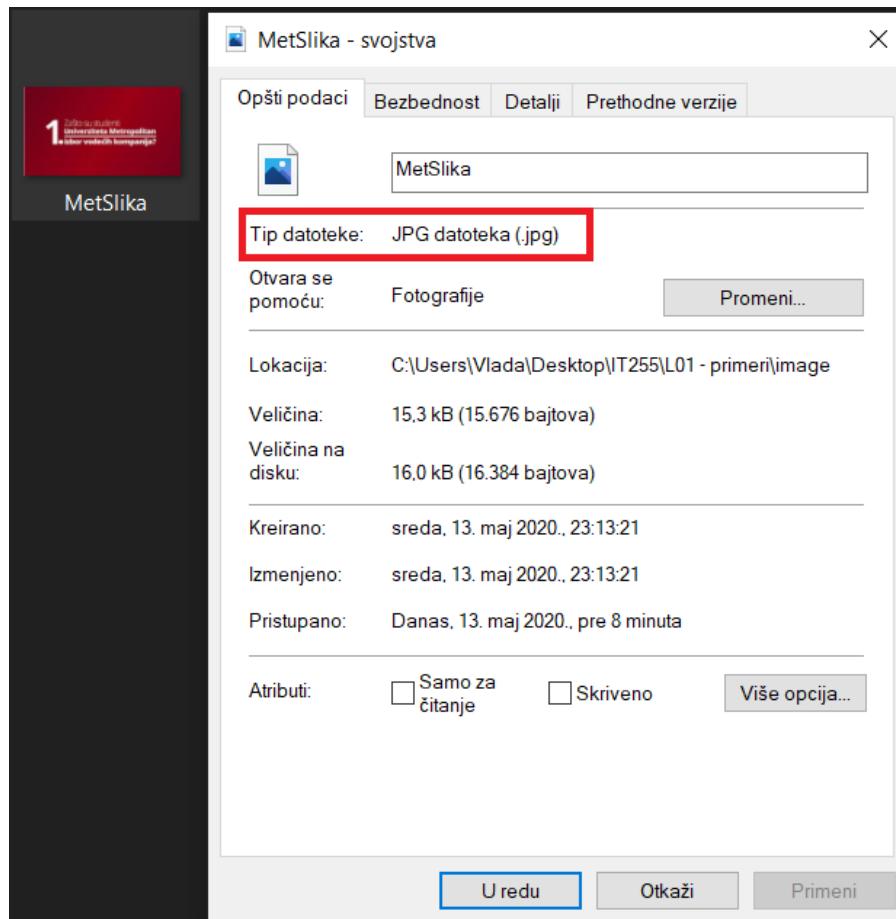
U savremenom digitalnom okruženju primena slika je važan segment.

Smatra se da je savremeni razvoj veb sistema jedan od elemenata na koji se oslanja savremeni digitalni svet. **U savremenom digitalnom okruženju primena slika je važan segment.** Otuda, sledeća tema za diskusiju i demonstraciju predstavlja dodavanje i upotreba slika u HTML stranicama.

Za dodavanje slika u HTML stranicu je zadužen poseban tag koji se naziva ``. Značenje ovog taga je *IMAGE* (slika) i, kao što je to bio slučaj sa tagom `<a>`, ovaj tag takođe poseduje atribut. Atribut sadrži informacije koje se odnose na: izvor, visinu, širinu i alternativni tekst slike.

Pre dodavanja željene slike u stranicu neophodno je proveriti njen format. Generalno, u veb stranicama se koriste sledeći formati slika: `.jpg`, `.png` i (sve manje i manje) `.gif`. Provera formata se obavlja na veoma jednostavan način - desnim klikom na sliku otvorice se prozor u kojem se bira stavka *Svojstva (Properties)* - slika 11.

Kada je utvrđeno da slika može da bude upotrebljena u HTML kodu, pristupa se njenom dodavanju u listing.



Slika 2.11 Provera formata slike

DODAVANJE SLIKA U HTML DOKUMENT

Pristupa se dodavanju izabrane slike na odgovarajuću poziciju unutar HTML stranice.

U prethodnom izlaganju je opisano koji formati fotografija su poželjni u HTML stranicama, kao i način provere da li fotografija pripada odgovarajućem formatu. Kada je sve to obavljeno, pristupa se dodavanju izabrane slike na odgovarajuću poziciju unutar HTML stranice.

Opšti oblik pisanja taga `` glasi:

```

```

Atributom *src* navodi se lokacija i datoteka slike, atribut *alt* predstavlja alternativni naziv slike značajan u situacijama kada se vrši rangiranje sajta u uslovima pretrage ili kada slika ne može da bude prikazana, atributima *height* i *width* definisana je visina i širina slike koja se prikazuje, respektivno.

Sada je moguće obaviti i konkretan zadatak. U folderu projekta kreiran je podfolder *images* i unutar njega je sačuvana datoteka *MetSlika.jpg* koja odgovara slici koju je potrebno dodati u stranicu *index.html*. U postojeći kod stranice dodaje se sledeća linija, odmah ispod poslednjeg dodatog linka:

```

```

Iz koda se primećuje koja slika, i sa koje lokacije, je upotrebljena, njen alternativni tekst, kao i dimenzije slike.

Sledi demonstracija prikazivanja slike u HTML stranici. Nakon što je obavljena ilustrovana modifikacija koda stranice, sa ciljem dodavanja izabrane slike, neophodno je ponovo snimiti promene, a nakon toga obaviti i ponovno učitavanje stranice u veb pregledač. Rezultat učitavanja je ilustrovan sledećom slikom.

IT255 - Veb sistemi 1

Prvi pokazni primer

U predmetu **IT255 - Veb sistemi 1** akcenat će biti na *frontend* razvoju

U predmetu **IT355 - Veb sistemi 2** akcenat će biti na *backend* razvoju

[Otvorite sledeću stranicu](#)
[Sajt Univerziteta Metropolitan](#)



Slika 2.12 Dodavanje slike u HTML dokument

▼ Poglavlje 3

Liste u HTML-u

VRSTE LISTI U HTML JEZIKU

HTML poseduje bogatu podršku za rad sa listama.

HTML poseduje bogatu podršku za rad sa listama i podržava primenu različitih tipova listi. Generalno, u veb dizajnu su od posebnog značaja sledeća tri tipa listi:

- uređene ili liste sa redosledom (ordered lists);
- neuređene ili liste sa tačkama navođenja (unordered list ili bullet point list);
- liste sa definicijama (definition lists).

U narednom izlaganju će biti obrađena sva tri navedena tipa listi, nakon čega će biti demonstrirana i njihova primena u HTML stranicama.

UREĐENE LISTE

Uređene liste predstavljaju način prikazivanja stavki u formi stavki poredanih po rednim brojevima.

Uređene liste predstavljaju način prikazivanja stavki u formi stavki poredanih po rednim brojevima. Sledećim redosledom je određena jedna uređena lista:

1. **Stavka1**
2. **Stavka2**
3. **Stavka3**
4. **Stavka4**

Svaka uređena lista predstavlja HTML element koji je smešten između početnog taga i završnog taga . Značenje ovih tagova je uređena lista (eng. ORDERED LIST). Svaka stavka koja se nalazi u uređenoj listi smeštena je između tagova i .

Sada je moguće dodati jednu uređenu listu u aktuelni primer. U nastavku koda stranice index.html dodaje se sledećih nekoliko linija koda:

```
<!-- Primena uređenih listi-->
<p>Na našem Univerzitetu postoje sledeći fakulteti:</p>
<ol>
<li>Fakultet informacionih tehnologija </li>
<li>Fakultet za menadžment</li>
```

```
<li>Fakultet digitalnih umetnosti</li>  
</ol>
```

Ovaj skup naredbi započinje odgovarajućim komentarom, a zatim sledi početni tag uređene liste (linija koda 3), odmah iza teksta koji opisuje sadržaj liste. Kao posebne stavke liste (linije koda 4 - 6) navedeni su fakulteti našeg Univerziteta. Konačno, nakon navođenja poslednje stavke, uređena lista je zatvorena (linija koda 7).

Sledi demonstracija prikazivanja uređene liste u stranici. Nakon što je obavljena ilustrovana modifikacija koda stranice, sa ciljem dodavanja uređene liste sa stavkama, neophodno je ponovo snimiti promene, a nakon toga obaviti i ponovno učitavanje stranice u veb pregledač. Rezultat učitavanja je ilustrovan sledećom slikom.

IT255 - Veb sistemi 1

Prvi pokazni primer

U predmetu IT255 - Veb sistemi 1 akcenat će biti na *frontend* razvoju

U predmetu IT355 - Veb sistemi 2 akcenat će biti na *backend* razvoju

Otvorite sledeću stranicu
Sajt Univerziteta Metropolitan



Slika 3.1 Primena uređene liste u HTML stranici

NEUREĐENE LISTE

*Neuređene liste predstavljaju način prikazivanja stavki u formi stavki poređanih po tačkama (eng. *bullet points*). Na sledeći način je određena jedna neuređena lista:*

- **Stavka1**
- **Stavka2**
- **Stavka3**
- **Stavka4**

Svaka neuređena lista predstavlja HTML element koji je smešten između početnog taga `` i završnog taga ``. Značenje ovih tagova je uređena lista (eng. *UNORDERED LIST*). Svaka stavka koja se nalazi u neuređenoj listi smeštena je, kao i u prethodnom slučaju, između tagova `` i ``.

Sada je moguće dodati jednu neuređenu listu u aktuelni primer. U nastavku koda stranice `index.html` dodaje se sledećih nekoliko linija koda:

```
<!-- Primena neuređenih listi-->
<p>Smerovi na FIT-u su:</p>
<ul>
<li>Softversko inženjerstvo </li>
<li>Informacione tehnologije</li>
<li>Računarske igre</li>
</ul>
```

Priloženi listing započinje odgovarajućim komentarom, a zatim sledi početni tag neuređene liste (linija koda 3), a odmah iza teksta koji opisuje sadržaj liste. Kao posebne stavke liste (linije koda 4 - 6) navedeni su smerovi *Fakulteta informacionih tehnologija*. Konačno, nakon navođenja poslednje stavke, neuređena lista je zatvorena (linija koda 7).

Sledi demonstracija prikazivanja neuređene liste u stranici. Nakon što je obavljena ilustrovana modifikacija koda stranice, a sa ciljem dodavanja neuređene liste sa stawkama, neophodno je ponovo snimiti promene, a nakon toga obaviti i ponovno učitavanje stranice u veb pregledač. Rezultat učitavanja je ilustrovan sledećom slikom.



Na našem Univerzitetu postoje sledeći fakulteti:

1. Fakultet informacionih tehnologija
2. Fakultet za menadžment
3. Fakultet digitalnih umetnosti

Smerovi na FIT-u su:

- Softversko inženjerstvo
- Informacione tehnologije
- Računarske igre

Slika 3.2 Primena neuređene liste u HTML stranici

LISTE SA DEFINICIJAMA (OPISIMA)

Liste sa definicijama sadrže stavku kao termin i njenu definiciju.

Liste sa definicijama (ili liste sa opisima, opisne liste) sadrže stavku kao termin i odgovarajuću definiciju (opis) tog termina. Svaka lista sa definicijama predstavlja HTML element koji se smešten između početnog taga `<dl>` i završnog taga `</dl>`. Značenje ovih tagova je lista sa definicijama ili lista sa opisima (eng. *DEFINITION / DESCRIPTION LIST*). Svaka stavka koja se nalazi u ovakvoj listi smeštena je između tagova `<dt>` i `</dt>` (drugi karakter **t** označava - *term* ili pojam). Definicija (opis) stavke je smešten između tagova `<dd>` i `</dd>` (drugi karakter **d** označava - *definition / description* ili u prevodu definiciju / opis).

Sada je moguće dodati jednu opisnu listu u aktuelni primer. U nastavku koda stranice index.html dodaje se sledećih nekoliko linija koda:

```
<!-- Primena listi sa definicijama (opisna lista) -->
<p>Predmeti za izučavanje veb razvoja su:</p>
```

```
<dl>
<dt>Veb sistemi 1</dt>
<dd>Predmet na 3. godini u kojem se izučavaju frontend tehnologije!!!</dd>
<dt>Veb sistemi 2</dt>
<dd>Predmet na 3. godini u kojem se izučavaju backend tehnologije!!!</dd>
</dl>
```

Priloženi listing započinje odgovarajućim komentarom, a zatim sledi početni tag opisne liste (linija koda 3), a odmah iza teksta koji opisuje sadržaj liste. Kao posebne stavke liste (linije koda 4 i 6) navedeni su predmeti na kojima se izučavaju veb tehnologije na **Fakultetu informacionih tehnologija**. U linijama koda 5 i 7 uvedeni su opisi navedenih stavki opisne liste.

Sledi demonstracija prikazivanja opisne liste u stranici. Nakon što je obavljena ilustrovana modifikacija koda stranice, a sa ciljem dodavanja prikazane opisne liste (videti kod), neophodno je ponovo snimiti promene, a nakon toga obaviti i ponovno učitavanje stranice u veb pregledač. Rezultat učitavanja je ilustrovan sledećom slikom.



Slika 3.3 Primena opisne liste u HTML stranici

▼ Poglavlje 4

Tabele i okviri u HTML-u

ELEMENTI HTML TABLE

Primena tabela je još jedan način da kreirane veb stranice izgledaju pregledno i sređeno.

Primena tabela je još jedan način da kreirane veb stranice izgledaju pregledno i sređeno. U HTML stranicama pomoću tabela je moguće na veoma jednostavan način organizovati određeni sadržaj te stranice. HTML jezik koristi četiri taga sa kreiranje tabela i dodavanje odgovarajućeg sadržaja unutar njih:

- <table> - početni tag tabele.
- <th> - značenje taga je TABLE HEADER (zaglavlje tabele). Ovaj tag se koristi u prvom redu (vrsti) za definisanje naziva kolona;
- <tr> - značenje taga je TABLE ROW (red u tabeli): Tag se koristi za dodavanje novog reda u tabelu;
- <td> - značenje taga je TABLE DATA (podaci u tabeli). Tag se koristi za dodavanje odgovarajućih vrednosti unutar prethodno kreiranog reda u tabeli.

VAŽNO!!!

Svaki od navedenih tagova mora da bute zatvoren na odgovarajućem mestu u HTML kodu.

Pre nego što kreiramo tabelu, unutar tagova <head> i </head> biće ugrađen sledeći kod:

```
<!-- Kreiranje linije tabele - o stilovima više u sledećoj lekciji -->
<style>
table, th, td {
    border: 1px solid black;
}
</style>
```

O ovakvom kodu će biti detaljno govora u okviru sledeće lekcije kada se bude izučavao jezik stilova CSS. Ovde je dovoljno da studenti primete da će kreiranje linije tabele biti u formi punih crnih linija, debljine od jednog piksela.

KREIRANJE HTML TABELE

Ispunjeni su uslovi za dodavanje tabele unutar započetog projekta.

Nakon što su opisani elementi pomoću kojih se tabele kreiraju i dobijaju elemente, moguće je pristupiti njihovom dodavanju unutar započetog projekta.

U navedenom svetu, a za dalju diskusiju, odmah iza poslednje dodatog elementa, u okviru tekućeg primera, dodaje se sledeći kod:

```
<!-- Primena tabela u HTML kodu -->
<p><strong>Tabela pokazuje jezike koji se uče na sledećim predmetima</strong></p>
<table>
<tr>
<th>CS230 - Distribuirani sistemi</td>
<th>IT255 - Veb sistemi 1</td>
<th>IT355 - Veb sistemi 2</td>
</tr>
<tr>
<td>Java EE</td>
<td>HTML, CSS, JS, Angular</td>
<td>Spring, Spring Boot</td>
</tr>
</table>
```

Ako se napravi analiza prethodnog listinga moguće je voditi sledeću diskusiju:

- linijama koda 3 i 15 je kreiran, odnosno zatvoren, HTML element koji odgovara tabeli;
- linijama koda 4 i 8, a zatim i 9 i 14, kreirana su dva reda u tabeli;
- prvi red (linije koda 5 - 7) predstavlja red sa naslovima kolona;
- drugi red (linije koda 11 - 14) predstavlja red sa vrednostima kolona.

Sledi demonstracija prikazivanja kreirane tabele u stranici. Nakon što je obavljena ilustrovana modifikacija koda stranice, a sa ciljem dodavanja tabele u HTML dokument (videti kod), neophodno je ponovo snimiti promene, a nakon toga obaviti i ponovno učitavanje stranice u veb pregledač. Rezultat učitavanja je ilustrovan sledećom slikom.

Tabela pokazuje jezike koji se uče na sledećim predmetima

CS230 - Distribuirani sistemi	IT255 - Veb sistemi 1	IT355 - Veb sistemi 2
Java EE	HTML, CSS, JS, Angular	Spring, Spring Boot

Slika 4.1 Tabela na dnu aktuelne HTML stranice

OKVIRI U HTML-U

Okvir (frame) je HTML element kojim je moguće prikazati stranicu u stranici.

Okvir (frame) je HTML element kojim je moguće prikazati "stranicu u stranici". Ovakvu integraciju, unutar jednog HTML dokumenta, moguće je obaviti na veoma jednostavan način primenom taga <iframe>. Ovaj tag takođe može da koristi atribute, a najčešće se preko atributa podešava veličina prozora u kojem se prikazuje ugnježdena stranica.

Posmatra se sledeći kod (integrisan je kao nastavak aktuelnog primera):

```
<p><strong>/>Stranica u stranici:</strong></p>
<iframe src="https://www.metropolitan.ac.rs/" width="450" height="300"></iframe>
```

Listing pokazuje kako je moguće umetnuti sadržaj neke druge stranice, u konkretnom slučaju početne stranice sajta našeg Univerziteta, unutar stranice koja predstavlja osnovu pokaznog primera.

Sledi demonstracija prikazivanja kreiranog okvira u stranici. Nakon što je obavljena ilustrovana modifikacija koda stranice, a sa ciljem dodavanja okvira u HTML dokument (videti kod), neophodno je ponovo snimiti promene, a nakon toga obaviti i ponovno učitavanje stranice u veb pregledač. Rezultat učitavanja je ilustrovan sledećom slikom.

Predmeti za izučavanje veb razvoja su:

Veb sistemi 1

Pređet na 3. godini u kojem se izučavaju frontend tehnologije!!!

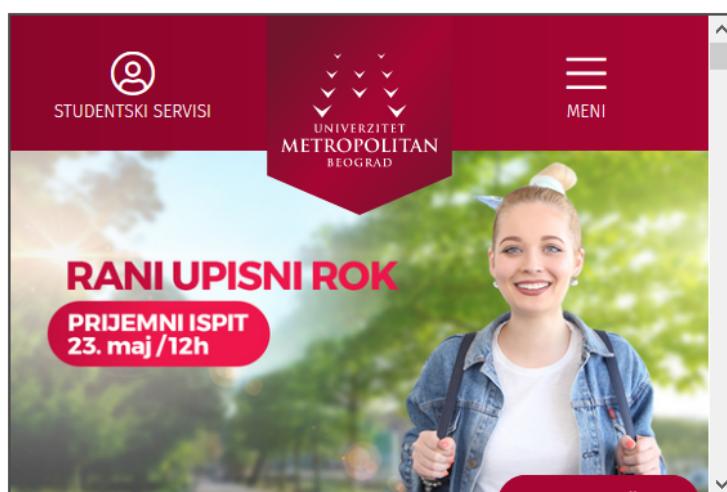
Veb sistemi 2

Pređet na 3. godini u kojem se izučavaju backend tehnologije!!!

Tabela pokazuje jezike koji se uče na sledećim predmetima

CS230 - Distribuirani sistemi	IT255 - Veb sistemi 1	IT355 - Veb sistemi 2
Java EE	HTML, CSS, JS, Angular	Spring, Spring Boot

Stranica u stranici:



Slika 4.2 Okviri u HTML-u

TABELE U HTML-U (VIDEO MATERIJAL)

HTML tutorijal - Tabele 8

Ova lekcija sadrži video materijal. Ukoliko želite da pogledate ovaj video morate da otvorite LAMS lekciju.

▼ Poglavlje 5

HTML forme

ELEMENTI FORME

U HTML stranicama forma predstavlja složeni element za preuzimanje korisničkih podataka.

U HTML stranicama forma predstavlja složeni element za preuzimanje korisničkih podataka. Primer jednostavne HTML forme moguće je ilustrovati sledećom slikom:

HTML forme - primer

Ime:

Prezime:

Ukoliko kliknete na "Submit" dugme, podaci sa forme se upućuju na stranicu "/druga.html".

Slika 5.1 Primer jednostavne HTML forme

Sa slike je moguće primetiti da forma sadrži dve labele za isticanje informacija koje je neophodno uneti u dva polja za unos teksta. Nakon što korisnik obavi unos podataka, klikom na dugme tipa *Submit* sa oznakom "Potvrđi" inicira se akcija kojom se korisnički podaci prosleđuju iz forme dalje na obradu. U realnim aplikacijama čest je scenario da se ovakvi podaci prosleđuju nekoj backend klasi koja se naziva kontrolerom. Ovakva klasa obraduje prosleđene podatke i kreira adekvatan izlaz koji se vraća klijentu i prezentuje njegovim korisnicima.

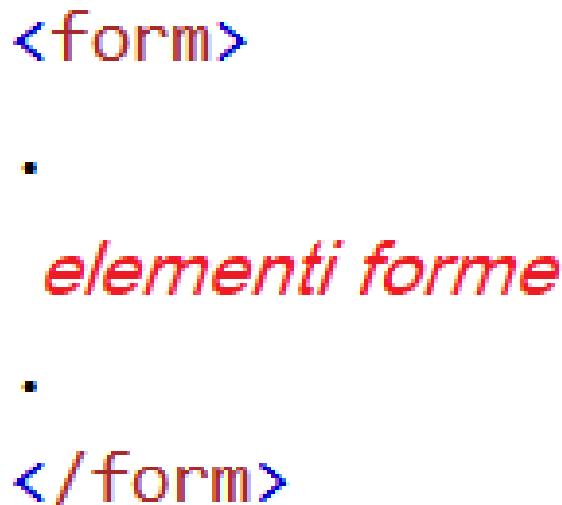
Da bi HTML forma bila kreirana, dostupni su sledeći HTML (elementi) tagovi:

- **<form>** - početni tag forme. Element kojim se kreira forma;
- **<input>** - najznačajniji element forme. Javlja se u više oblika i funkcionalnosti u zavisnosti od vrednosti atributa *type*;
- **<label>** - ističe labelu za elemente forme;
- čitav niz atributa koji se koriste u kombinaciji sa navedenim tagovima.

UPOTREBA TAGA FORM

Upotrebom taga FORM kreira se forma.

Element `<form>` definiše HTML element koji se naziva formom i čiji je zadatak prikupljanje korisničkih podataka sa stranice. Ovo je početni tag forme, odnosno, element kojim se forma kreira. Sledećom slikom je ilustrovana upotreba navedenog taga:



Slika 5.2 Ilustracija upotrebe taga

Svaka forma je izgrađena od elemenata forme. Elementi forme su različiti tipovi ulaznih (`input`) elemenata, poput:

- **polja za unos teksta** (*text fields*);
- **polja za izbor** (*checkboxes*),
- **radio dugmadi** (*radio buttons*);
- **dugmadi za potvrdu forme** (*submit buttons*)
- i drugih.

Upravo u narednom izlaganju će biti analizirani, diskutovani i demonstrirani navedeni `input` elementi forme.

UPOTREBA TAGA INPUT

Ovo je najznačajniji element svake HTML forme.

Element `<input>` je najznačajniji element HTML forme. Javlja se u nekoliko različitih oblika u zavisnosti od vrednosti atributa `type`. Sledećom tabelom su ilustrovani oblici u kojima ovaj element može da se javi:

TIP	OPIS
<code><input type="text"></code>	Definiše polje za unos teksta u jednom redu
<code><input type="radio"></code>	Definiše radio dugme (za izbor jedne od više ponuđenih opcija)
<code><input type="submit"></code>	Definiše dugme za potvrđivanje forme
<code><input type = "checkbox"></code>	Definiše polje za izbor (checkbox) jedne ili više ponuđenih opcija

Slika 5.3 Neki od oblika INPUT elementa

U narednom izlaganju biće razmotren i demonstriran svaki od navedenih oblika pojedinačno. Posmatramo sledeći listing:

```
<form action="druga.html">
    <!-- Unos preko polja za unos teksta-->
    <label for="fname">Ime:</label><br/>
    <input type="text" id="fname" name="fname" value="Vladimir"><br/>
    <label for="lname">Prezime:</label><br/>
    <input type="text" id="lname" name="lname" value="Milićević"><br/><br/>

    <!-- Unos preko polja za radio dugmadi-->
    <p><strong>Izaberite pol: </strong></p>
    <input type="radio" id="muski" name="pol" value="muski">
    <label for="male">Muški</label><br/>
    <input type="radio" id="female" name="pol" value="zenski">
    <label for="zenski">Ženski</label><br/>

    <!-- Unos preko polja za checkbox - a-->
    <p><strong>Izaberite predmet: </strong></p>
    <input type="checkbox" id="it255" name="it255" value="VS1">
    <label for="it255"> Veb sistemi 1</label><br/>
    <input type="checkbox" id="it355" name="it355" value="VS2">
    <label for="it355"> Veb sistemi 2</label><br/>
    <input type="checkbox" id="cs230" name="cs230" value="DS">
    <label for="cs230"> Distribuirani sistemi</label><br/>

    <!-- Potvrda forme-->
    <input type="submit" value="Potvrdi">
</form>
```

Razmatramo navedene oblike taga `<input>`:

- `<input type="text">` - Preko polja za unos teksta unose se vrednosti za ime i prezime. Svako od polje za unos teksta poseduje osobinu `id` preko koje je povezan sa odgovarajućom labelom. Na labelama je prikazan tekst koji ukazuje na vrednosti koje je neophodno uneti u polja za unos teksta (linije koda 1-6);
- `<input type="radio">` - Preko radio dugmadi unose se vrednosti za pol: muški ili ženski. Svako od radio dugmadi poseduje osobinu `id` preko koje je povezan sa odgovarajućom labelom. Na labelama je prikazan tekst koji ukazuje na vrednosti koje se biraju preko ovih kontrola. Moguće je izabrati samo jednu od ponuđenih opcija (linije koda 10-13);
- `<input type="checkbox">` - Preko `checkbox` kontrole unose se vrednosti za izbor predmeta. Svaki od njih poseduje osobinu `id` preko koje je povezan sa odgovarajućom labelom. Na labelama je prikazan tekst koji ukazuje na vrednosti koje se biraju preko ovih kontrola. Moguće je izabrati jednu ili više od ponuđenih opcija (linije koda 17-22);
- `<input type="submit">` - Element predstavlja dugme preko kojeg se potvrđuju uneti podaci u formi i inicira odgovarajuća akcija (linija koda 26).

UPOTREBA FORME - DEMONSTRACIJA

Sledi demo koda kreirane forme.

Izvršena je korekcija početnog primera, na osnovu koda iz prethodnog izlaganja, i sada on ima sledeću formu:

```
<!DOCTYPE html>
<html>
<body>

<h2>HTML forme - primer</h2>

<form action="druga.html">
    <!-- Unos preko polja za unos teksta-->
    <label for="fname">Ime:</label><br/>
    <input type="text" id="fname" name="fname" value="Vladimir"><br/>
    <label for="lname">Prezime:</label><br/>
    <input type="text" id="lname" name="lname" value="Milićević"><br/><br/>

    <!-- Unos preko polja za radio dugmadi-->
    <p><strong>Izaberite pol: </strong></p>
    <input type="radio" id="muski" name="pol" value="muski">
    <label for="male">Muški</label><br/>
    <input type="radio" id="female" name="pol" value="zenski">
    <label for="zenski">Ženski</label><br/>

    <!-- Unos preko polja za checkbox - a-->
    <p><strong>Izaberite predmet: </strong></p>
    <input type="checkbox" id="it255" name="it255" value="VS1">
    <label for="it255"> Veb sistemi 1</label><br/>
    <input type="checkbox" id="it355" name="it355" value="VS2">
    <label for="it355"> Veb sistemi 2</label><br/>
    <input type="checkbox" id="cs230" name="cs230" value="DS">
    <label for="cs230"> Distribuirani sistemi</label><br/>

    <!-- Potvrda forme-->
    <input type="submit" value="Potvrdi">
</form>

<p>Ukoliko kliknete na "Submit" dugme, podaci sa forme se upućuju na stranicu "/druga.html".</p>

</body>
</html>
```

Neophodno se ponovo snimiti datoteku. Ponovnim učitavanjem ove datoteke u veb pregledač dobija se sledeći izlaz:

HTML forme - primer

Ime:

Prezime:

Izaberite pol:

- Muški
 Ženski

Izaberite predmet:

- Veb sistemi 1
 Veb sistemi 2
 Distribuirani sistemi

Ukoliko kliknete na "Submit" dugme, podaci sa forme se upućuju na stranicu "/druga.html".

Slika 5.4 Upotreba forme - demonstracija

Ova lekcija sadrži video materijal. Ukoliko želite da pogledate ovaj video morate da otvorite LAMS lekciju.

▼ Poglavlje 6

Pokazna vežba - raspoređivanje elemenata u HTML-u

RASPOREĐIVANJE ELEMENATA U HTML-U

Veoma je važno da stranica bude lepo organizovana iz više razloga.

Elementi u HTML stranicama su uvek uređeni po nekom rasporedu (layout) koji je vizuelno dopadljiv i koji je rezultat mašte i ukusa dizajnera. Veoma je važno da stranica bude lepo organizovana iz više razloga:

- lepši izgled privlači korisnike;
- preglednije stranice omogućavaju da se lakše dođe do sadržaja;
- oba razloga povlače veće zadovoljstvo korisnika.

U prethodnom izlaganju fokus je bio na učenju HTML-a pa je ovaj segment dizajna bio u drugom planu. Sada je vreme da se konačno stvari postave na svoje mesto i da se ovde zaokruži izučavanje HTML-a.

Postoji nekoliko HTML elemenata raspoređivanja, kao:

- `<header>` - Definiše zaglavje dokumenta ili sekcije;
- `<nav>` - Definiše kontejner za navigacione linkove
- `<section>` - Definiše sekciju u dokumentu
- `<article>` - Definiše nezavisan članak
- `<aside>` - Definiše bočnu traku (sidebar) za prikazivanje sadržaja;
- `<footer>` - Definiše podnožje (footer) dokumenta ili sekcije;
- `<details>` - Definiše dodatne detalje;
- `<summary>` - Definiše zaglavje elementa `<details>`.

Navedene elemente je moguće ilustrovati slikom 1 (sledeća sekcija).

Postoji i nekoliko tehnika raspoređivanja HTML elemenata. Tehnike se baziraju na CSS jeziku, a on je predmet bavljenja sledeće lekcije. Opšte tehnike raspoređivanja HTML elemenata su:

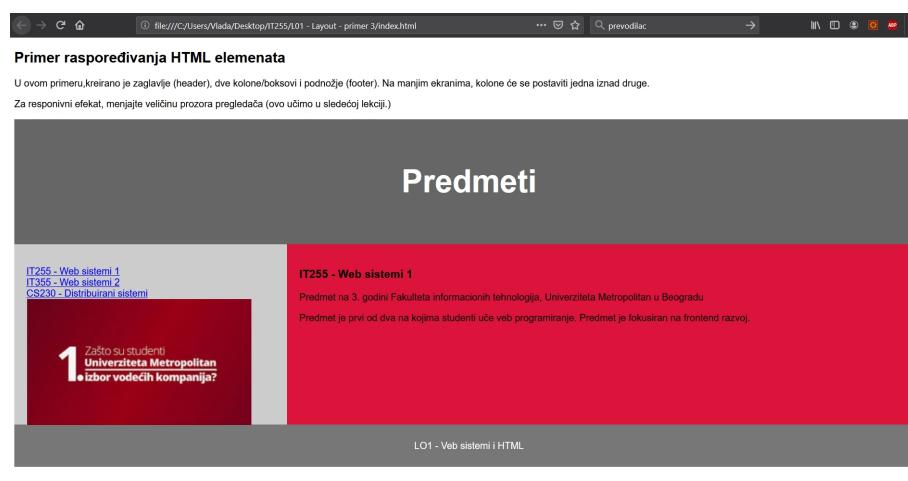
- *CSS radni okvir* (framework);
- *CSS float* osobina;
- *CSS flexbox*;
- *CSS grid*.

FLOAT MENADŽER RASPOREDA - POKAZNA VEŽBA 1 (15 MINUTA)

Česta praksa u veb dizajnu jeste kreiranje veb stranica koje koriste osobinu float.

Česta praksa u veb dizajnu jeste kreiranje veb stranica koje koriste osobinu *float*. Ovakav dizajn je jednostavno naučiti ali ima jedan nedostatak - "lebdenje" elemenata vezuje elemente za tok elementa, a to može da ima posledice na fleksibilnost HTML stranice.

Sledećom slikom je prikazano rešenje pokazne vežbe 1.



Slika 6.1 Rešenje pokazne vežbe 1

Svi elementi ove stranice su detaljno opisani u prethodnoj sekciji i oni su ovde u fokusu. Primjenjeni stilovi su korisni za lepši izgled i biće detaljno objašnjeni u sledećoj lekciji.

Sledećim listingom su implementirani elementi detaljno opisani u prethodnoj sekciji. Primećuje se u kodu da je na svaki od njih primjenjen stil. Ovde je cilj da uočite kako se to radi, a detaljno o tome će biti govora u sledećoj lekciji.

```
<!DOCTYPE html>
<html lang="en">
<head>
<title>CSS šabloni</title>
<meta charset="utf-8">
<meta name="viewport" content="width=device-width, initial-scale=1">
<style>
* {
  box-sizing: border-box;
}

body {
  font-family: Arial, Helvetica, sans-serif;
}
```

```
/* stilizuje zahglavlje (header) */
header {
    background-color: #666;
    padding: 30px;
    text-align: center;
    font-size: 35px;
    color: white;
}

/* Kreira dve kolone (boksa) koji plutaju jedan pored drugog */
nav {
    float: left;
    width: 30%;
    height: 300px; /* only for demonstration, should be removed */
    background: #ccc;
    padding: 20px;
}

/* stilizuje listu u meniju */
nav ul {
    list-style-type: none;
    padding: 0;
}

article {
    float: left;
    padding: 20px;
    width: 70%;
    background-color: #DC143C;
    height: 300px; /* only for demonstration, should be removed */
}

section:after {
    content: "";
    display: table;
    clear: both;
}

/* stil podnožja (footer) */
footer {
    background-color: #777;
    padding: 10px;
    text-align: center;
    color: white;
}

/* Responzivni raspored - kolone (boksovi) će se namestiti jedna iznad druge na
manjim ekranima */
@media (max-width: 600px) {
    nav, article {
        width: 100%;
        height: auto;
    }
}
```

```
}

</style>
</head>
<body>

<h2>Primer raspoređivanja HTML elemenata</h2>
<p>U ovom primeru, kreirano je zagлавље (header), dve kolone/boksovi i podnožje (footer). Na manjim ekranima, kolone će se postaviti jedna iznad druge.</p>
<p>Za responivni efekat, menjajte veličinu prozora pregledača (ovo učimo u sledećoj lekciji.)</p>

<header>
    <h2>Predmeti</h2>
</header>

<section>
    <nav>
        <ul>
            <li><a href="#">IT255 - Web sistemi 1</a></li>
            <li><a href="#">IT355 - Web sistemi 2</a></li>
            <li><a href="#">CS230 - Distribuirani sistemi</a></li>
        <li>
        </li>
        </ul>
    </nav>

    <article>
        <h1>IT255 - Web sistemi 1</h1>
        <p>Predmet na 3. godini Fakulteta informacionih tehnologija, Univerziteta Metropolitan u Beogradu</p>
        <p>Predmet je prvi od dva na kojima studenti uče veb programiranje. Predmet je fokusiran na frontend razvoj.</p>
    </article>

</section>

<footer>
    <p>L01 - Veb sistemi i HTML</p>
    <a href = "flexbox.html"> Otvorite sledeću stranicu</a> <br/>
</footer>

</body>
</html>
```

Više o Float menadžeru rasporeda možete pogledati na sledećem linku:
https://www.w3schools.com/cssref/pr_class_float.asp.

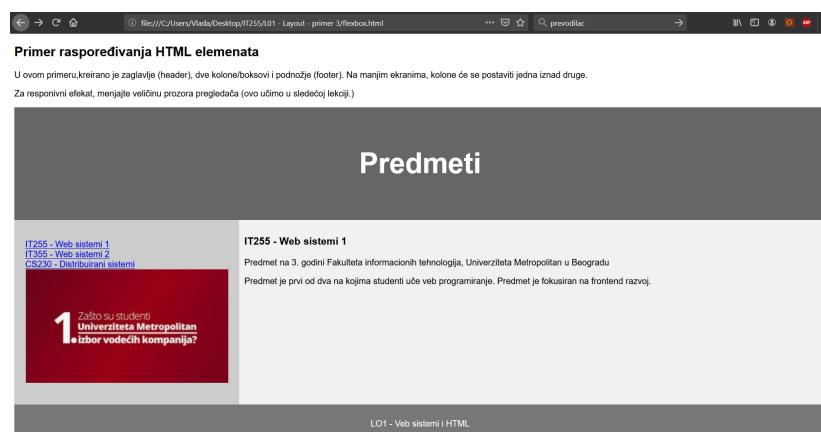
FLEXBOX MENADŽER RASPOREDA - POKAZNA VEŽBA 2 (15 MINUTA)

Flexbox menadžer rasporeda se takođe često koristi u HTML stranicama.

Flexbox menadžer rasporeda je novi menadžer rasporeda u CSS3. još jednom, stilovi su tema za sledeću lekciju. Ovde je cilj da još malo obnovimo i provežbamo HTML.

Primenom Flexbox menadžera rasporeda obezbeđuje se da se elementi ponašaju na predvidljiv način kada raspored elemenata na stranici mora da se prilagodi različitim veličinama ekrana i različitim ekranima na uređajima.

Sledećom slikom je prikazano rešenje pokazne vežbe 2.



Slika 6.2 Rešenje pokazne vežbe 2

Flexbox menadžer rasporeda nije podržan u Internet Explorer 10 i ranijim verzijama.

Sledećim listingom su implementirani elementi detaljno opisani u prethodnom izlaganju. Primećuje se u kodu da je na svaki od njih primjenjen stil. Ovde je cilj da uočite kako se to radi, a detaljno o tome će biti govora u sledećoj lekciji.

```
<!DOCTYPE html>
<html lang="en">
<head>
<title>CSS Template</title>
<meta charset="utf-8">
<meta name="viewport" content="width=device-width, initial-scale=1">
<style>
* {
  box-sizing: border-box;
}

body {
  font-family: Arial, Helvetica, sans-serif;
```

```
}
```

```
/* stilizuje header */
```

```
header {
```

```
    background-color: #666;
```

```
    padding: 30px;
```

```
    text-align: center;
```

```
    font-size: 35px;
```

```
    color: white;
```

```
}
```

```
/* Kontejner za flexbox-ove */
```

```
section {
```

```
    display: -webkit-flex;
```

```
    display: flex;
```

```
}
```

```
/* stil navigacionog menija */
```

```
nav {
```

```
    -webkit-flex: 1;
```

```
    -ms-flex: 1;
```

```
    flex: 1;
```

```
    background: #ccc;
```

```
    padding: 20px;
```

```
}
```

```
/* stil liste iz menija */
```

```
nav ul {
```

```
    list-style-type: none;
```

```
    padding: 0;
```

```
}
```

```
/* stil sadržaja */
```

```
article {
```

```
    -webkit-flex: 3;
```

```
    -ms-flex: 3;
```

```
    flex: 3;
```

```
    background-color: #f1f1f1;
```

```
    padding: 10px;
```

```
}
```

```
/* stil footer-a */
```

```
footer {
```

```
    background-color: #777;
```

```
    padding: 10px;
```

```
    text-align: center;
```

```
    color: white;
```

```
}
```

```
/* Responzivni raspored*/
```

```
@media (max-width: 600px) {
```

```
    section {
```

```
        -webkit-flex-direction: column;
```

```
        flex-direction: column;
    }
}
</style>
</head>
<body>

<h2>Primer raspoređivanja HTML elemenata</h2>
<p>U ovom primeru, kreirano je zaglavje (header), dve kolone/boksovi i podnožje (footer). Na manjim ekranima, kolone će se postaviti jedna iznad druge.</p>
<p>Za responivni efekat, menjajte veličinu prozora pregledača (ovo učimo u sledećoj lekciji.)</p>

<header>
    <h2>Predmeti</h2>
</header>

<section>
    <nav>
        <ul>
            <li><a href="#">IT255 - Web sistemi 1</a></li>
            <li><a href="#">IT355 - Web sistemi 2</a></li>
            <li><a href="#">CS230 - Distribuirani sistemi</a></li>
        <li>
            </li>
        </ul>
    </nav>

    <article>
        <h1>IT255 - Web sistemi 1</h1>
        <p>Predmet na 3. godini Fakulteta informacionih tehnologija, Univerziteta Metropolitan u Beogradu</p>
        <p>Predmet je prvi od dva na kojima studenti uče veb programiranje. Predmet je fokusiran na frontend razvoj.</p>
    </article>

</section>

<footer>
    <p>L01 - Veb sistemi i HTML</p>
    <a href = "grid.html"> Otvorite sledeću stranicu</a> <br/>
</footer>

</body>
</html>
```

Više o Flexbox menadžeru rasporeda možete pogledati na sledećem linku: https://www.w3schools.com/css/css3_flexbox.asp.

GRID MENADŽER RASPOREDA - POKAZNA VEŽBA 3 (15 MINUTA)

Grid menadžer rasporeda postavlja HTML elemente u formi matrice.

Grid menadžer rasporeda postavlja HTML elemente u formi matrice. Još jednom napominjemo da CSS ovde nije u prvom planu, to će biti u sledećoj lekciji. I dalje vežbamo da primenjujemo naučene HTML elemente.

Stil je ovde od značaja u toj meri što određuje primenjeni izgled (layout). Sledećim listingom je kreirana HTML stranica sa grid menadžerom rasporeda

```
<!DOCTYPE html>
<html>
<head>
<meta name="viewport" content="width=device-width, initial-scale=1.0">
<style>
* {
  box-sizing: border-box;
}

.row::after {
  content: "";
  clear: both;
  display: table;
}

[class*="col-"] {
  float: left;
  padding: 15px;
}

.col-1 {width: 8.33%;}
.col-2 {width: 16.66%;}
.col-3 {width: 25%;}
.col-4 {width: 33.33%;}
.col-5 {width: 41.66%;}
.col-6 {width: 50%;}
.col-7 {width: 58.33%;}
.col-8 {width: 66.66%;}
.col-9 {width: 75%;}
.col-10 {width: 83.33%;}
.col-11 {width: 91.66%;}
.col-12 {width: 100%;}

html {
  font-family: "Lucida Sans", sans-serif;
}

.header {
  background-color: #9933cc;
```

```
color: #ffffff;
padding: 15px;
}

.menu ul {
list-style-type: none;
margin: 0;
padding: 0;
}

.menu li {
padding: 8px;
margin-bottom: 7px;
background-color: #33b5e5;
color: #ffffff;
box-shadow: 0 1px 3px rgba(0,0,0,0.12), 0 1px 2px rgba(0,0,0,0.24);
}

.menu li:hover {
background-color: #0099cc;
}
</style>
</head>
<body>

<div class="header">
<h1>Fakulteti</h1>
</div>

<div class="row">
<div class="col-3 menu">
<ul>
<li>Fakultet informacionih tehnologija</li>
<li>Fakultet za menadžment</li>
<li>Fakultet digitalnih umetnosti</li>
</ul>
</div>

<div class="col-9">
<h1>FIT</h1>
<p>Naš fakultet je Fakultet informacionih tehnologija, Univerziteta  
Metropolitan u Beogradu.</p>
<p>Za responivni efekat, menjajte veličinu prozora pregledača (ovo učimo u  
sledećoj lekciji.).</p>
</div>
</div>

</body>
</html>
```

Grid menadžer rasporeda poseduje redove (linija koda 69) i kolone (linije koda 70 i 78). Za realizaciju ovakvog menadžera rasporeda je zadužen HTML5 tag `<div>`. Ovo je tag koji

je zadužen za definisanje nekog elementa na kojeg je primenjen neki CSS stil (detaljnije: https://www.w3schools.com/tags/tag_div.ASP).

Sledećom slikom je prikazano rešenje pokazne vežbe 3.



Slika 6.3 Rešenje pokazne vežbe 3

Pokazni primer1 1, 2 i 3 su povezani i integrisani u jedinstvenu veb HTMLaplikaciju koju možete da preuzmete i testirate odmah iza ovog objekta učenja.

Više o Grid menadžeru rasporeda možete pogledati na sledećem linku: https://www.w3schools.com/css/css_grid.asp.

RASPOREĐIVANJE ELEMENATA U HTML-U (VIDEO MATERIJAL)

Raspoređivanje elemenata u HTML-u rezime kroz video materijale.

Build an HTML + CSS Layout with Flexbox in just a few lines of code (trajanje 7:04).

Ova lekcija sadrži video materijal. Ukoliko želite da pogledate ovaj video morate da otvorite LAMS lekciju.

CSS Grid Layout Crash Course (trajanje 27:54)

Ova lekcija sadrži video materijal. Ukoliko želite da pogledate ovaj video morate da otvorite LAMS lekciju.

✓ Poglavlje 7

Individualna vežba

ZAHTEVI VEŽBE (TRAJANJE: 90 MINUTA)

Koristeći raspored sa slike napravite vlastitu HTML stranicu.

Zadatak 1:

1. Koristeći raspored sa slike napravite vlastitu HTML stranicu;
2. Možete da koristite CSS iz individualne vežbe 1 - pokušajte da ga malo modifikujete;
3. Poželjno je ali nije obavezno da dodate stilove u HTML;
4. Svaki element sa slike mora da sadrži ogovarajući sadržaj;
5. Studenti imaju slobodu da odrede konačan izgled stranice ali moraju da implementiraju: slike, linkove i formu;
6. Ukoliko imate poteškoća pri radu, kontaktirajte predmetnog asistenta.
7. Kada završite zadatak, pozovite asistenta i prodiskutujte.



Slika 7.1 Layout zadatka individualne vežbe

▼ Poglavlje 8

Domaći zadatak 1

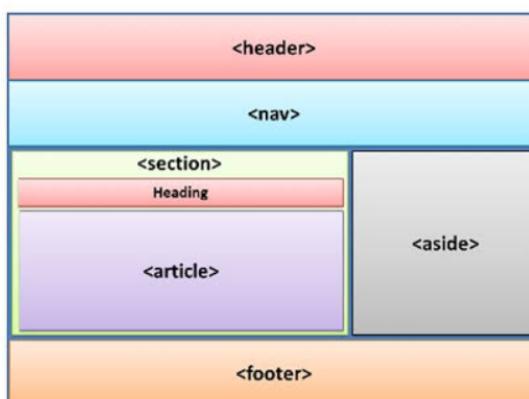
DOMAĆI ZADATAK - ZAHTEVI (PREDVIĐENO VREME 180 MINUTA)

Samostalno rešavanje domaćeg zadatka.

Zadatak 1:

1. Koristeći raspored sa slike napravite vlastitu HTML stranicu;
2. Poželjno je, **ali nije obavezno**, da dodate vlastit stilove u HTML - možete da koristite stilove iz pokaznih vežbi;
3. Svaki element sa slike mora da sadrži ogovarajući sadržaj;
4. Studenti imaju slobodu da odrede konačan izgled stranice ali moraju da implementiraju: slike, linkove i formu;
5. Urađen rad poslati predmetnom asistentu na mail;
6. Spremiti dokumentaciju za usmenu odbranu DZ.

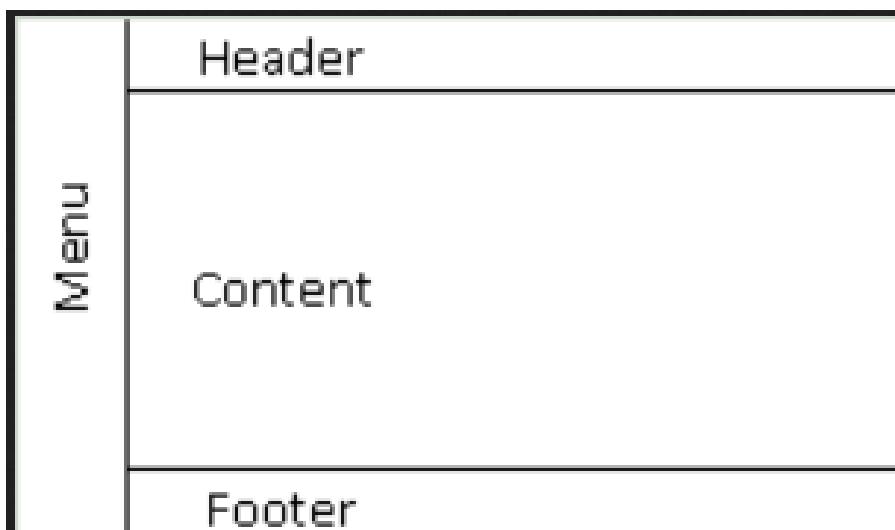
HTML5 Layout



Slika 8.1 Layout domaćeg zadatka 1

Zadatak 2

Pod istim zahtevima kao i 1. zadatku, uraditi domaći zadatak na osnovu sledeće slike.



Slika 8.2 Layout domaćeg zadatka 2

▼ Poglavlje 9

Zaključak

ZAKLJUČAK

Lekcija se bavila izučavanje osnova veb jezika HTML.

Lekcija se bavila izučavanjem osnova veb jezika *HTML*. U uvodnom izlaganju je predstavljena:

- definicija ovog jezika,
- generacije kroz vremenske intervale pojavljivanja, kao i
- organizacija *W3C* koja je zadužena za razvoj i standardizaciju ovog jezika.

Diskusija je nastavljana kroz navođenje osnovnih koncepata i elemenata jezika *HTML*, praćena kroz ilustraciju adekvatnim pokaznim primerima. Kada je student naučio da kreira osnovnu strukturu nekog *HTML* dokumenta, lekcija je prešla na izučavanje složenijih koncepata od kojih je svakako najznačajniji kreiranje i upravljanje formama u *HTML* jeziku, kao i primena menadžera rasporeda koji je u direktnoj vezi sa CSS jezikom koji se izučava u sledećoj lekciji.

Već je istaknuto da se *HTML* oslanja na intenzivnu podršku drugih jezika, kada je reč o stilizaciji i kreiranju i prikazivanju dinamičkog sadržaja u *HTML* stranicama. Upravo iz navedenog razloga ovaj predmet će u nastavku staviti fokus na izučavanje jezika *CSS* i *JavaScript*.

LITERATURA

Za pripremu lekcije je korišćena savremena štampana i veb literatura.

Obavezna literatura:

1. Jennifer Niederst Robbins, Learning Web Design - Fifth Edition, by , Copyright ©; 2018 O'Reilly Media, Inc.
2. Roxane Anquetil, Fundamental Concepts for Web Development: HTML5, CSS3, JavaScript and much more! For complete beginners!, 2019,

Dopunska literatura:

1. Daniel Bell, HTML & CSS: A Step-by-Step Guide for Beginners2, Guzzler Media, 2019.

Web sadržaj:

1. <https://www.w3schools.com/html/default.asp>
2. https://html.com/#What_is_HTM
3. <https://web.archive.org/web/20050305063804/http://www.ietf.org/rfc/rfc1866.txt>
4. <https://www.w3.org/TR/2018/SPSD-html32-20180315/>
5. <https://www.w3.org/TR/REC-html40-971218/>
6. <https://www.w3.org/TR/html401/>
7. <https://www.scss.tcd.ie/misc/15445/15445.HTML>
8. <https://html.spec.whatwg.org/multipage/>
9. <https://sr.wikipedia.org/sr-ec/HTML>



IT255 - VEB SISTEMI 1

CSS i Bootstrap

Lekcija 02

PRIRUČNIK ZA STUDENTE

IT255 - VEB SISTEMI 1

Lekcija 02

CSS I BOOTSTRAP

- ✓ CSS i Bootstrap
- ✓ Poglavlje 1: Karakteristike, istorija i razvoj CSS-a
- ✓ Poglavlje 2: CSS po generacijama
- ✓ Poglavlje 3: Sintaksa CSS-a
- ✓ Poglavlje 4: CSS3 - Pokazna vežba 1
- ✓ Poglavlje 5: CSS3 - Individualna vežba 1
- ✓ Poglavlje 6: Bootstrap - osnove
- ✓ Poglavlje 7: Primena Bootstrap okvira
- ✓ Poglavlje 8: Dodatna literatura za rad
- ✓ Poglavlje 9: Bootstrap - Pokazna vežba 2
- ✓ Poglavlje 10: Bootstrap - Individualna vežba 2
- ✓ Poglavlje 11: Domaći zadatak
- ✓ Zaključak

Copyright © 2017 – UNIVERZITET METROPOLITAN, Beograd. Sva prava zadržana. Bez prethodne pismene dozvole od strane Univerziteta METROPOLITAN zabranjena je reprodukcija, transfer, distribucija ili memorisanje nekog dela ili čitavih sadržaja ovog dokumenta., kopiranjem, snimanjem, elektronskim putem, skeniranjem ili na bilo koji drugi način.

Copyright © 2017 BELGRADE METROPOLITAN UNIVERSITY. All rights reserved. No part of this publication may be reproduced, stored in a retrieval system or transmitted in any form or by any means, electronic, mechanical, photocopying, recording, scanning or otherwise, without the prior written permission of Belgrade Metropolitan University.

❖ Uvod

UVOD

CSS je jezik formatiranja pomoću kog se definiše izgled elemenata veb-stranice

CSS (engl. *Cascading Style Sheets*) je jezik formatiranja pomoću kog se definiše izgled elemenata veb-stranice. Prvobitno, *HTML* je služio da definiše kompletan izgled, strukturu i sadržaj veb-stranice, ali je od verzije 4.0 HTML-a uveden CSS koji bi definisao konkretni izgled, dok je HTML ostao u funkciji definisanja strukture i sadržaja.

CSS je u određenoj formi postojao još u začecima SGML-a (*Standard Generalized Markup Language*) 1970ih godina.

Kako je HTML postajao komplikovaniji, davao je sve više mogućnosti za definiciju izgleda elemenata, ali je istovremeno postajao nečitljiviji i teži za održavanje. Različiti veb pregledači (engl. *web browsers*) su prikazivali dokumente na različite načine, i postojala je potreba za doslednjom tehnikom definisanja prikaza elemenata na stranici.

Da bi se ovo postiglo, devet različitih metoda je predloženo na zvaničnom forumu W3C-a. Od devet, dve metode su izabrane kao temelj onoga što je kasnije postalo CSS: *CHSS* (engl. *Cascading HTML Style Sheets*) i *SSP* (engl. *Stream-based Style Sheet Proposal*). Prvo je *Hakon Vium Li* (koji je sada šef tehničke službe kompanije Opera) predložio *CHSS* u oktobru 1994, jezik koji je imao dosta sličnosti sa današnjim CSS-om. Bert Bos je radio na pregledaču po nazivu *Argo*, koji je imao sopstveni način definisanja stilova, *SSP*. **Li i Bos su radili zajedno da bi osnovali CSS standard** (slovo H je izbačeno iz skraćenice CHSS jer se CSS mogao odnositi i na druge jezike pored HTML-a).

Za razliku od postojećih jezika poput *DSSSL*-a i *FOSI*-a, *CSS* je dozvoljavao da više opisa utiče na dokument, tj. jedna definicija stilova je mogla naslediti osobine od druge.

Lijev predlog je postavljen na konferenciji "Veb mozaik" u Čikagu 1994. godine, i ponovo sa Bosovim predlogom 1995. Otpriklje u ovo vreme je osnovan *W3C*, koji je preuzeo funkciju razvoja CSS-a. Do kraja 1996, CSS je bio spremjan da se objavi kao standard, i CSS1 je objavljen u decembru.

Razvoj HTML-a, CSS-a i DOM-a se odvijao u jednoj istoj grupi, *HTML Editorial Review Board* (ERB). Početkom 1997. grupa ERB se podelila na tri radne grupe: radna grupa za HTML, kojom je upravljao Den Konoli iz W3C-a, radna grupa za DOM, kojom je upravljao Loren Vud iz kompanije Softkvod, i radna grupa za CSS, kojom je upravljao Kris Lili iz W3C-a.

Radna grupa za CSS je počela da radi na problemima koji nisu bili obuhvaćeni CSS-om verzije 1, koji se tako razvio u CSS2, 4. novembra 1997; objavljen je kao zvanična verzija 12. maja 1998. CSS3, čiji je razvoj započet 1998. se još uvek razvija.

U međuvremenu je razvijeno dosta radnih okvira za kvalitetniji i brži razvoj veb stranica primenom CSS-a. Jedan od najpopularnijih je *Bootstrap*.

UVODNI VIDEO

Trajanje video snimka: 5min 30sek

Ova lekcija sadrži video materijal. Ukoliko želite da pogledate ovaj video morate da otvorite LAMS lekciju.

▼ Poglavlje 1

Karakteristike, istorija i razvoj CSS-a

VIDEO PREDAVANJE ZA OBJEKAT "KARAKTERISTIKE, ISTORIJA I RAZVOJ CSS-A"

Trajanje video snimka: 15min 59sek

Ova lekcija sadrži video materijal. Ukoliko želite da pogledate ovaj video morate da otvorite LAMS lekciju.

STILSKI JEZIK WEB-A

Stilski jezik opisuje kako se dokumenti prezentuju na ekranu.

Stilski jezik opisuje kako se dokumenti prezentuju na ekranu, u štampi, ili čak i kako se izgovaraju.

W3C aktivno promoviše upotrebu stilskih fajlova na Web-u od kako je Konzorcijum osnovan 1994-te.

U samom početku W3C konzorcijum je kreirao nekoliko W3C preporuka:

- CSS1,
- CSS2,
- XPath,
- XSLT.

Ideja je bila da CSS bude široko implementiran u aktuelnim veb pregledačima.

Sledećim video materijalom je izložena kratka istorija razvoja CSS i HTML.

Ova lekcija sadrži video materijal. Ukoliko želite da pogledate ovaj video morate da otvorite LAMS lekciju.

Pojavom CSS -a veb razvoj dobija jednu novu dimenziju. Priključivanjem stilskih fajlova strukturiranim dokumentima na veb-u (npr. HTML), autori i čitaoci mogu da:

- utiču na prezentaciju dokumenta ;
- ne žrtvujući nezavisnost od platforme i
- ne dodajući nove HTML elemente.

Istorijat stilizacije na vebu, na samom početku, oslanjao se na razvoj i unapređenje koji je tekao u dva pravca (stilska jezika):

- **CSS = (Cascade) Style Sheets for HTML**
- **XSL = (Extensible) Style Sheets for XML.**

DVA JEZIKA STILIZACIJE

Istorijski gledano, stilizacija na vebu je tekla u dva pravca, razvojem i primenom jezika: CSS i XLS.

Kao što je istaknuto u prethodnom izlaganju, istorijski gledano, stilizacija na vebu je tekla u dva pravca, razvojem i primenom jezika:

- **CSS = (Cascade) Style Sheets for HTML**
- **XSL (T) = (Extensible) Style Sheets for XML.**

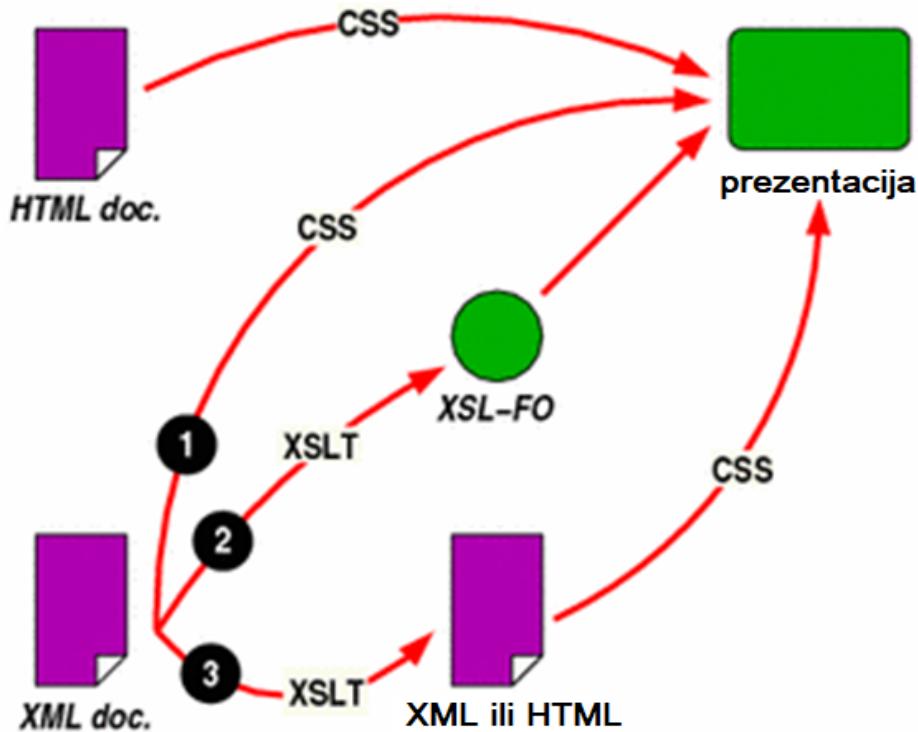
Generalno gledano, sledećim tezama je moguće na veoma jednostavan način povući paralele između primena navedenih jezika.

CSS:

- Jednostavniji za upotrebu;
- Lakši za učenje;
- Lakši i jeftiniji za održavanje.

XSL(T):

- Primenljiv kada se koriste transformacije (više o ovom jeziku možete da pogledate na linku https://www.w3schools.com/xml/xsl_languages.asp).



Slika 1.1 Paralele između primena jezika CSS i XSL [izvor: https://www.w3schools.com/xml/xsl_languages.asp]

CSS I XSL(T) - POREĐENJE

CSS i XSL(T) mogu biti komplementarni.

Ako se detaljno posveti pažnja prethodnoj slici, moguće je uočiti suštinske razlike u primeni jezika CSS i XSL(T). Jedinstvene karakteristike su te da:

1. CSS može da se koristi za stilizovanje HTML i XML dokumenata.
2. XSL, suprotno tome, može samo da transformiše XML dokumenta.

Na primer, XSL može da se koristi za transformaciju XML podataka u HTML / CSS dokumenta na Web serveru. Na ovaj način, dva jezika se komplementiraju, i mogu se zajedno koristiti. Oba jezika mogu da se koriste za stilizovanje XML dokumenata.

Navedena diskusija, u svrhu lakšeg pamćenja, može biti zaokružena sledećom slikom:

	CSS	XSL
Može da se koristi sa HTML?	da	ne
Može da se koristi sa XML?	da	da
Jezik transformacije?	ne	da

Slika 1.2 CSS i XSL(T) - poređenje [izvor: autor]

OPŠTE KARAKTERISTIKE JEZIKA CSS

Jezik CSS je veoma jednostavan za korišćenje - definisanje izgleda veb stranice.

Jezik CSS je veoma jednostavan za korišćenje i njegove osobine, u HTML kontekstu, je moguće izdvojiti kroz sledeće teze:

1. *CSS* predstavlja skraćenicu engleskih reči *Cascading Style Sheets*;
2. *HTML* obezbeđuje strukturu stranice;
3. *CSS* je jezik formatiranja koji omogućava definisanje izgleda veb stranice.
4. *CSS* je dizajniran iz razloga da razdvoji sadržaj dokumenta od načina na koji će biti predstavljen (font, boja, veličina...).
5. *CSS* može biti direktno integriran u HTML;
6. *CSS* može biti eksterni tekstualni dokument koji sadrži instrukcije za formatiranje.
7. Navedene instrukcije koriste veb pregledači / pretraživači.

NAPOMENA: CSS je dizajniran iz razloga da razdvoji sadržaj dokumenta od načina na koji će biti predstavljen (font, boja, veličina...).

CSS sintaksa se sastoji od opisa izgleda elemenata u dokumentu.

Opis može da definiše izgled više elemenata, i više opisa može da definiše jedan element. Na taj način se opisi slažu jedan preko drugog da bi definisali konačni izgled određenog elementa.

Otuda naziv kaskadni (engl. cascade - kaskada) - slaganje jednog stila preko drugog u definisanju konačnog izgleda nekog HTML elementa.

Kao što je već rečeno, i jasno istaknuto, CSS je dizajniran iz razloga da razdvoji sadržaj dokumenta od načina na koji će biti predstavljen.

Ovo razdvajanje doprinosi:

- većoj fleksibilnosti i kontroli specifikacije karakteristika prezentacije, smanjuje kompleksnost i
- omogućava da veći broj HTML strana bude formatiran na isti način.

OPIS HTML ELEMENATA U CSS - U

CSS sintaksa se sastoji od opisa izgleda elemenata u dokumentu ili ugnježdenom CSS kodu.

Kao što je već diskutovano, CSS sintaksa se sastoji od opisa izgleda elemenata u eksternom dokumentu ili ugnježdenom CSS kodu unutar HTML-a. Opis može da definiše izgled više

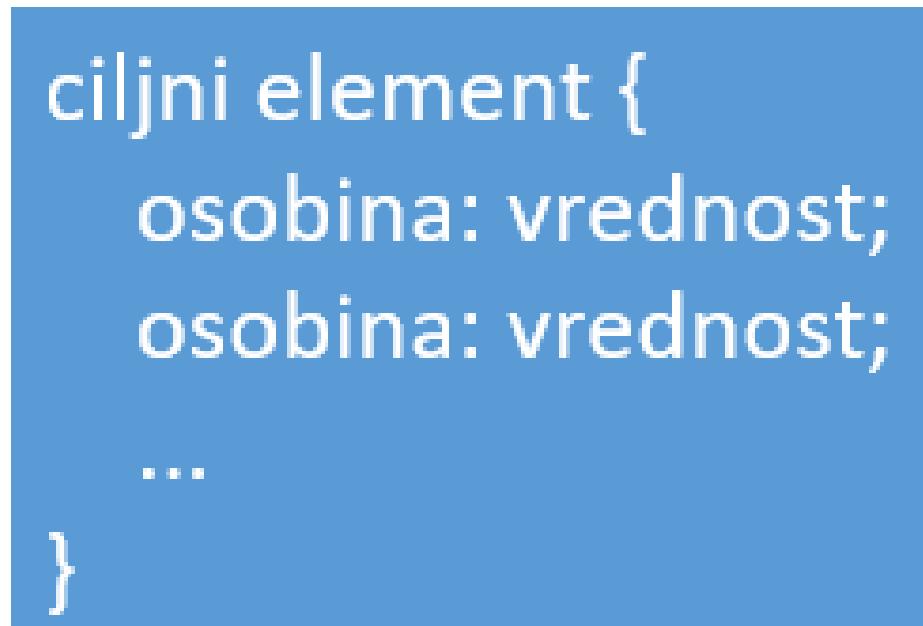
elemenata, i više opisa može da definiše jedan element. Na taj način se opisi slažu jedan preko drugog da bi definisali konačni izgled određenog elementa - kaskadni izgled.

Svaki opis se sastoji od tri elementa:

1. definicija ciljnih elemenata (na koje se opis primjenjuje);
2. osobine
3. vrednosti

Definišemo ciljne elemente, tj. elemente na koje će se trenutni opis odnositi, a potom nizom parova svojstvo-vrednost definisemo izgled svakog ciljnog elementa.

Sintaksa koja se pri tome definiše je sledećeg oblika:



Slika 1.3 Osnovna sintaksa CSS opisa HTML elementa [izvor: autor]

KARAKTERISTIKE CSS - KASKADNI NIVOI

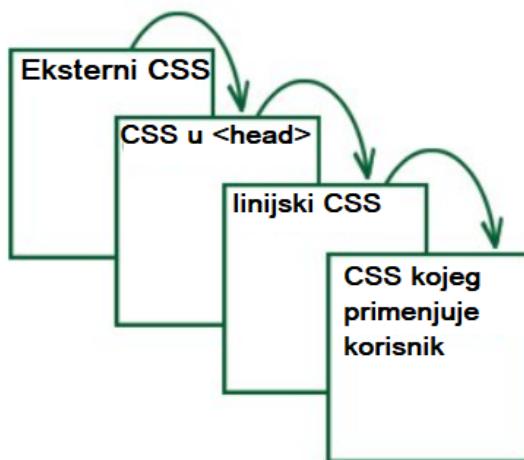
CSS formatiranje se kaskadno pomera nadole po prioritetu.

CSS Instrukcije se primenjuju na elemente koji su obuhvaćeni skupom instrukcija, pri čemu nije bitno gde se ti elementi pojavljuju na veb strani.

CSS određuje šemu prioriteta koja odlučuje koja pravila za stilove važe ukoliko se desi da postoji više pravila za jedan element. **Ovo su CSS kaskade.**

Instrukcije u vezi sa jednim slojem stavljuju se van snage u sloju koji je jedan korak niži. Iz ovog razloga, formatiranje se kaskadno pomera nadole po prioritetu.

Sledećom slikom su ilustrovani CSS kaskadni nivoi.



Slika 1.4 kaskadni nivoi [izvor: autor]

Za našu diskusiju je od posebnog značaja CSS formatiranje na tri različita nivoa.

1. *Prvi nivo* predstavlja eksternu listu stilova - zasebni tekstualni dokument koji se nalazi na serveru mreže i na tu listu se HTML datoteke pozivaju.
2. *Drugi nivo* postavlja instrukcije za formatiranje u elemente `<style>` u okviru odeljka `<head>`. Ovaj nivo radi na stranama u kojima su uključeni stilovi. **Ukoliko dođe do konflikta, staviće van snage instrukcije iz eksternih lista stilova.**
3. *Treći nivo*, najniži, jeste redni stil ili stil za redove. Stilovi se nalaze u atributu `style` elementa kojeg taj atribut formatira.

Upravo navedenim redosledom veb pregledač čita i tumači stlove primenjene na određeni HTML dokument i njegove elemente.

▼ Poglavlje 2

CSS po generacijama

CSS - 1. GENERACIJA

Prva CSS generacija (ili kraće CSS1) predstavlja specifikaciju iz decembra 1996.

Prva CSS generacija (ili kraće CSS1) predstavlja specifikaciju iz decembra 1996. Uvodi brojne mogućnosti za formatiranje i stilizovanje HTML stranica i elemenata, među kojima su najznačajnije:

1. boja teksta, pozadina i drugih elemenata;
2. svojstva fonta kao što su tipografsko pismo i naglašavanje;
3. tekstualni atributi kao što su razmak između reči, slova i linije teksta
poravnanje teksta, slika i drugih elemenata
4. margine, granice, "padding" (prostor između granice elementa i sadržaja elementa) i pozicioniranje većine elemenata;
5. jedinstvena identifikacija i generička klasifikacija grupa atributa.

CSS - 2. GENERACIJA

Druga CSS generacija predstavlja specifikaciju iz maja 1998, nadograđenu 2011.

Druga CSS generacija (ili kraće CSS2) predstavlja specifikaciju iz maja 1998.

Ovde je uključen i broj novih mogućnosti kao što su:

1. apsolutno, relativno i fiksno pozicioniranje elemenata i z-index (redosled elementa u steku),
2. koncepti medijskih tipova,
3. podrška za zvučne liste stilova i dvosmeran tekst i
4. nova svojstva fonta kao što su senke

Unapređenje CSS 2.1 je dugo razvijano i konačno zvanično objavljeno 7. juna 2011. Ovom nadogradnjom su ispravljene greške identifikovane u generaciji CSS2, a koje se odnose na:

1. uklonjene slabo podržane ili ne u potpunosti interoperabilne karakteristike i
2. dodate su već implementirane ekstenzije pretraživača, da bi CSS bio u skladu sa W3C procesom za standardizovanje tehničkih specifikacija na vebu.

CSS - 3. GENERACIJA

Rad na CSS3 započeo je još u vreme objavljivanja CSS2.

Rad na CSS3 započeo je još u vreme objavljivanja CSS2. Najraniji CSS3 nacrti bili su objavljeni juna 1999.

Ono što je specifično za CSS3 jeste način na koji su specifikacije ponuđene i raspoložive proizvođačima pretraživača. Dok je u ranijim verzijama bila neophodna cela specifikacija za odobrenje, u CSS3 ponuđeni su moduli. Oni predstavljaju nekoliko odvojenih dokumenata.

Svaki modul dodaje nove funkcije ili proširuje svojstva definisana u CSS2.

Novembra 2011. već postoji preko 50 modula objavljenih od strane CSS radne grupe.

Druge promene obuhvataju veću fleksibilnost pozadina i okvira, kolona sa sadržajem i veću mogućnost prilikom štampanja multimedijalnih strana.

SPECIFIČNOSTI 3. GENERACIJE CSS - A

CSS3 je razdvojen u module.

Kao što je već diskutovano, CSS3 je razdvojen u module. Neki od najbitnijih CSS3 modula su:

- *selektori*;
- *model oblasti*;
- *pozadine i okviri*;
- *tekstualni efekti*.

Sa verzijom CSS3 mogu se:

- kreirati zaobljene ivice,
- dodavati senke okvirima i
- koristiti slike kao okviri.

U tabeli prikazana je podrška čitača za razne atribute slike.

Атрибут	Подршка претраживача				
border-radius					
box-shadow					
border-image					

Slika 2.1 Neke funkcionalnosti CSS3 i podrška veb čitača [izvor: <http://www.nerdcorelabs.com/2012/03/css-3-borders-border-radius.html>]

NAPOMENA: CSS3 je postavljen kao konačni CSS standard.

Sledećim video materijalom je ilustrovana evolucija CSS jezika za stilizaciju HTML dokumenata.

Ova lekcija sadrži video materijal. Ukoliko želite da pogledate ovaj video morate da otvorite LAMS lekciju.

▼ Poglavlje 3

Sintaksa CSS-a

KOMENTARI I SELEKTORI U CSS JEZIKU

Komentare ignoriše pregledač, selektor je HTML element koji želimo da formatiramo.

Diskusiju u vezi sa **CSS** sintaksom započinjemo izučavanjem pisanja komentara u ovom jeziku. Komentari su CSS instrukcije koje pretraživač potpuno ignoriše. Označavaju se na sledeći način:

- **/* */**, između zvezdica nalazi se komentar (slika 1).

```
*C:\Users\User\Desktop\wiki.html - Notepad++
File Edit Search View Encoding Language Settings Macro Run Plugins Window ?
wiki.html [3]
1  <!DOCTYPE html>
2  <html>
3  <head>
4  <style>
5  p {
6      color: #93B874; /*You can create comments at the end of lines */
7      /* Or you can create comments between lines */
8      font-family: "times";
9  }
10
11 /* You can create multi-line comments.
12 Everything between the two comment tags will
13 not be processed when displaying the page */
14 </style>
15 </head>
16 <body>
17
18 <p> The comments above will not appear on the page. </p>
19
20 </body>
21 </html>
```

Slika 3.1 Pisanje komentara u CSS jeziku [izvor: <https://www.wikihow.com/Comment-in-CSS>]

Sledeći važan element sintakse prestavlja selektor. CSS se sastoji od dva dela:

- selektora i
- od jedne ili više deklaracija.

Selektor je **HTML** element koji želimo da formatiramo. Deklaracija se sastoji od atributa koji menjamo i odgovarajuće vrednosti. Navedeno je ilustrovano slikom 2.



Slika 3.2 Selektor i deklaracije u CSS jeziku [izvor: autor]

OSNOVE SINTAKSE CSS-A

U CSS-u svaki opis se sastoji od tri elementa: ciljni element, osobine i vrednosti.

U CSS-u svaki opis se sastoji od tri elementa:

1. definicija ciljnih elemenata;
2. osobine
3. vrednosti.

Ilustracija navedenog je data kroz CSS kod sa sledeće slike.

```
body {  
    font-size: 14px;  
    font-family: "Times New Roman", Times, serif;  
}
```

Slika 3.3 Opis elementa u CSS-u [izvor: autor]

Za dodatnu ilustraciju moguće je uvesti sledeći primer:

1. Veličina fonta dodeljena elementu `<body>` biće nasleđena u celokupnom tekstu html dokumenta (veb stranice).
2. Izuzetak je tekst ugnježdenog elementa naslova `<h1>` - veličine 28px,
3. Tekst paragrafa `<p>` će biti isписан словимa veličine 16px.
4. Tekst koji se ne nalazi u naslovima `<h1>` ili paragrafima `<p>` će naslediti osobinu veličine fonta elementa `<body>`.

Rešenje primera može biti ilustrovano CSS kodom sa sledeće slike.

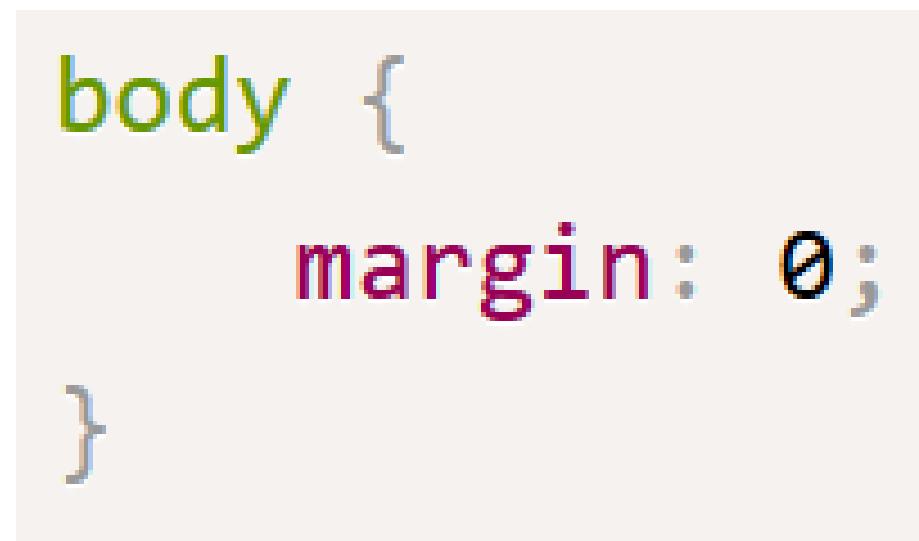
```
body {  
    font-size: 14px;  
}  
  
h1 {  
    font-size: 28px;  
}  
  
p {  
    font-size: 16px;  
}
```

Slika 3.4 CSS listing kao rešenje postavljenog problema stilizacije [izvor: autor]

OSNOVE SINTAKSE CSS-A - DOPUNSKA DISKUSIJA

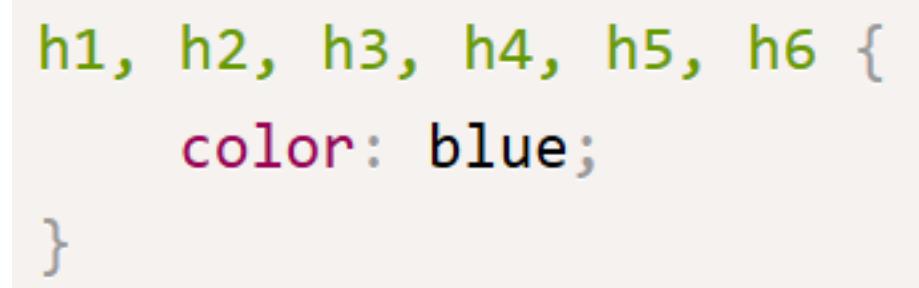
Postoje osobine čije se vrednosti ne nasleđuju u ugnježdenim elementima.

Postoje osobine čije se vrednosti ne nasleđuju u ugnježdenim elementima. Jednu od takvih osobina moguće je ilustrovati na primeru postavljanja vrednosti margine u elementu `<body>`, kao na sledećoj slici.



Slika 3.5 Postavljanje margine je CSS osobina koja se ne nasleđuje [izvor: autor]

Ukoliko se neka osobina odnosi na veći broj različitih HTML elemenata, u CSS - u se to rešava na veoma jednostavan način. Za istovremeno određivanje osobina za više elemenata - navodimo ih jedan iza drugoga odvojene zarezom. Navedeno je ilustrovano sledećom slikom.



Slika 3.6 Definisanje osobine za više HTML elemenata u CSS [izvor: autor]

SINTAKSA – CLASS I ID

Atribut id se naziva identifikatorom selektora.

Atribut *id* se naziva identifikatorom selektora. **Identifikator(ski) selektor koristimo pri definisanju svojstava samo jednog elementa.** Identifikator selektora se odnosi na HTML element kojem je vrednost *id* osobine jednaka nazivu selektora. Primer dodeljivanja osobine identifikatora moguće je ilustrovati na sledeći način:

```
<div id="sidebar">  
    <!-- Sve što želimo da se nalazi unutar ovog elementa -->  
</div>elementa
```

Slika 3.7 Primer dodeljivanja osobine identifikatora [izvor: autor]

Sledeći *CSS* kod primenjuje se na HTML element koji ima postavljenu osobinu *id="siderbar"* (na element *<div>*). Oznaka selektora počinje znakom „#“:

```
#sidebar {  
    text-align: center;  
    color: blue;  
}
```

Slika 3.8 Definisanje selektora za odgovarajući id [izvor: autor]

Za razliku od identifikatora, osobinu klasa (`class`) koristimo za definisanje osobina više elementa unutar HTML dokumenta. Pomoću klasnog selektora definisemo sve HTML elemente kojima je vrednost osobine `class` jednaka njegovom imenu. Primer HTML koda koji ima podešenu osobinu klase na "podvuceno" je prikazan sledećim listingom.

```
<h1 class="podvuceno">  
    Naslov je podvučen.  
</h1>  
<p class="podvuceno">  
    I tekst je takođe podvučen.  
</p>
```

Slika 3.9 Primena klasnog selektora [izvor: autor]

Sledeći CSS kod primenjuje se na sve HTML elemente koji imaju podešenu osobinu `class="podvuceno"` (na elemente `<h1>` i `<p>`). Oznaka selektora klase počinje s znakom tačka '.'.

```
.podvuceno {  
    text-decoration: underline;  
    color: green;  
}
```

Slika 3.10 Definisanje klasnog selektora [izvor: autor]

CSS TEXT - BOJA I PORAVNANJE

CSS osobine teksta se nasleđuju s roditelja na potomke.

U nastavku izlaganja, poseban akcenat se stavlja na izučavanje CSS osobina za tekst. **CSS osobine teksta se nasleđuju s roditelja na potomke.**

Boja se podešava po sledećem obrascu: `color: vrednost;`

Moguće vrednosti su:

- **heksadekadna RGB vrednost:** npr. #FFFF00, #00CC00...
- *RGB kod boje* – npr. rgb(0, 255, 0), rgb(0, 128, 128)
- **primena naziva boje** (engleski nazivi): npr. *green, yellow, blue* i tako dalje.

Osobina *text-align* određuje način poravnjanja teksta. Moguća poravnjanja teksta su levo, desno, središnje ili obostrano (engl. justify).

Koristimo sledeći obrazac: *text-align: vrednost;*

Moguće vrednosti su:

- *left*
- *right*
- *center*
- *justify.*

CSS TEXT - DEKORACIJA, UVLAČENJE I RAZMAK TEKSTA

Osobina text-decoration omogućava povlačenje linije u određenom položaju u odnosu na tekst.

Osobina *text-decoration* omogućava postavljanje crte ispod, iznad ili preko teksta, odnosno isključuje potcrtavanje teksta. Vrednost *blink* postavlja treperenje teksta (ne radi u Internet Explorer - u).

Obrazac koji se koristi za podešavanje osobine je:

- *text-decoration: vrednost;*

Moguće vrednosti su:

- *none* - ne koristi dekoraciju;
- *underline* - podvlači tekst;
- *overline* - postavlja liniju iznad teksta;
- *line-through* - postavlja liniju preko teksta;
- *blink* - podešava treperenje teksta.

Dodatna CSS podešavanja teksta, poput uvlačenja i razmaka, moguće je definisati na neki od sledećih načina:

- Osobina *text-indent* definiše uvlačenje prvog reda teksta na opšti način: *text-indent: vrednost;*
- Definisanje razmaka između slova se definiše podešavanjem: *letter-spacing: vrednost;*
- Definisanje razmaka između reči se definiše podešavanjem: *word-spacing: vrednost;*

Sledećom slikom je demonstrirana primena CSS - a na tekst.

O v a s u s l o v a r a z m a k n u t a 8 p x .

Ove reči su razmaknute 12px.

Slika 3.11 Primena CSS - a na tekst. [izvor: autor]

CSS TEXT – TRANSFORMISANJE TEKSTA I FONT

CSS omogućava transformacije teksta i primenu fontova.

Osobina **text-transform** određuje veličinu (u smislu velikih i malih slova) slova u HTML elementu. Obrazac je: **text-transform: vrednost;**

Moguće vrednosti su:

- **none** - ne vrši transformaciju teksta;
- **capitalize** - podešava početno veliko slovo;
- **lowercase** - transformiše tekst u prikaz malim slovima u celosti;
- **uppercase** - transformiše tekst u prikaz velikim slovima u celosti;

CSS osobine fonta se nasleđuju s roditeljskih elemenata na elemente potomke. Osobinom **font** se podešava više osobina:

- **font-style** - podešava jednu od dve vrednosti **normal** ili **italic**;
- **font-weight** - podešava font na neku od sledećih vrednosti: **bold** (podebljan - boldiran), **bolder** (definiše "deblja" slova), **lighter** (definiše "tanja" slova), **normal** (ne primenjuje boldiranje), **initial** (podešava ovu osobinu na podrazumevanu vrednost), **inherit** (nasleđuje osobinu iz roditeljskih HTML elemenata).
- **font-size** - podešava veličinu fonta u veličinama px (pikseli), em (pixels/16=em) ili % (procentima);
- **font-family** - podešava se familija fonta (Times New Roman, Arial, Verdana...);

Sledećom slikom je moguće ilustrovati podešavanje fonta primenom CSS jezika.

```
font: italic bold normal small 1.4em Verdana, sans-serif;
```

Ovaj primer automatski podešava sledeće osobine:

```
font-style: italic;
font-weight: bold;
font-variant: normal;
font-size: small;
line-height: 1.4em;
font-family: Verdana, sans-serif;
```

Slika 3.12 Podešavanje fonta primenom CSS jezika [izvor: autor]

CSS - LINKOVI

Pomoću pseudo klase linkova određujemo ponašanje linka u određenom stanju.

Pomoću pseudo klase linkova određujemo ponašanje linka u određenom stanju. Stanja pseudo klasa su:

- prethodna posećenost ili neposećenost web stranice,
- položaj miša iznad linka ili
- stanje držanja pritisnutog tastera miša na linku.

1. *a:link* - Opisuje stanje linka koji još nije posećen od strane veb pregledača;
2. *a:visited* – Pokazuje na stranicu koja je posećena i nalazi se u lokalnoj memoriji veb pregledača;
3. *a:hover* - Opisuje stanje linka kada se pokazivač miša nalazi na njemu;
4. *a:active* - Opisuje stanje linka kada se desi klik miša na link, a taster još uvek nije otpušten.

Pseudo klase linkova moramo navoditi zadatim redosledom.

```
a:link {  
    text-decoration: none;  
}  
a:visited {  
    text-decoration: none;  
}  
a:hover {  
    text-decoration: underline;  
}  
a:active {  
    text-decoration: underline;  
}
```

Slika 3.13 Redosled navođenja pseudo klasa linkova [izvor: autor]

CSS POZADINA

Osobine CSS pozadine se ne nasleđuju s roditeljskih elemenata na potomke.

Osobine CSS pozadine se ne nasleđuju s roditeljskih elemenata na potomke.

Osobinom *background* možemo postaviti više osobina pozadine HTML elementa istovremeno. Redosled navođenja nije obavezujući. Moguće je podesiti sledeće osobine:

- *background-attachment* – Podešava sliku pozadine (vrednosti: *fixed* ili *scroll*);
- *background-color* - Podešava boju pozadine (vrednosti: ime ili *RGB* ili *transparent*);
- *background-image* - Podešava sliku kao pozadinu elementa (vrednosti: *url* ili *none*);
- *background-position* - Podešava položaj pojavljivanja pozadinske slike (vrednosti po x i y : *top*, *center*, *bottom*, *left*, *right*, % ili vrednost položaja);
- *background-repeat* – Ponavljanje pozadinske slike po x i y osi (vrednosti: *repeat-x*, *repeat-y*, *repeat*, *no-repeat*).

CSS pozadina - primer

```
background: #ffffff url('putanja_do_slike') top left no-repeat fixed;
```

Ovom instrukcijom je automatski podešeno sledeće:

```
background-color: #ffffff;
background-image: url('putanja_do_slike');
background-position: top left;
background-repeat: no-repeat;
background-attachment: fixed;
```

Slika 3.14 Primer CSS podešavanja pozadine [izvor: autor]

CSS RAZMACI

U CSS-u razmaci se ne nasleđuju se sa roditelja na potomke.

U CSS-u razmaci se ne nasleđuju se sa roditelja na potomke. Osobinom *padding* se određuje razmak između HTML sadržaja i ivica stranice. Vrednosti mogu biti :

- dužina ili
- procenat;

Podešavaju se sledeće osobine:

- *padding-top*,
- *padding-right*,
- *padding-bottom* i
- *padding-left*.

Obrazac podešavanja osobina razmaka u CSS - u je sledeći:

```
padding-top: vrednost;  
padding-right: vrednost;  
padding-bottom: vrednost:  
padding-left: vrednost;
```

Slika 3.15 Obrazac podešavanja osobina razmaka u CSS [izvor: autor]

CSS razmaci - primer

- Primer unosa sva četiri rastojanja od ivica:

```
padding: 20px 15px 10px 5px;
```

Slika 3.16 Primer unosa sva četiri rastojanja od ivica [izvor: autor]

- Primer unosa tri rastojanja od ivica – izostavljeni se uzima kao njemu suprotni:

```
padding: 20px 15px 10px; Levi razmak = desni razmak = 15 px
```

Slika 3.17 Primer unosa tri rastojanja od ivica [izvor: autor]

- Primer unosa dva rastojanja od ivica – izostavljeni se uzima kao njemu suprotni:

```
padding: 20px 15px; Levi razmak = desni razmak = 15 px, gornji = donji = 20 px
```

Slika 3.18 Primer unosa dva rastojanja od ivica [izvor: autor]

CSS GRANICE - BOJE I STILOVI

Osobinom border možemo postaviti boju, stil i širinu ivica za stranice.

Osobinom `border` možemo postaviti boju, stil i širinu ivica za sve četiri stranice elementa odjednom. Nije obavezno navesti sve vrednosti.

Navedeno je moguće ilustrovati sledećim primerom:

```
border: 2px dotted #ff0000; isto kao  
border-width: 2px;  
border-style: dotted;  
border-color: #ff0000;
```

Slika 3.19 CSS podešavanje osobina granica [izvor: autor]

Podešavanje boja ivica (granica) nekog HTML elementa može biti po nekom od sledećih scenarija:

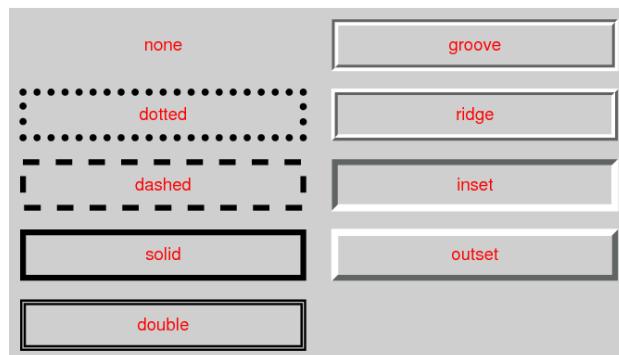
- Podešavaju se sve 4 granice;
- Podešavaju se gornja, desna, donja (leva dobija vrednost desne);
- Podešavaju se gornja i desna (donja i leva dobijaju identične vrednosti, respektivno);
- Podešava se gornja (sve ostale dobijaju identičnu vrednost).

```
border-color: red, yellow, blue, green;  
border-color: red, yellow, blue;  
border-color: red, yellow;  
border-color: purple;
```

Slika 3.20 Podešavanje boja ivica (granica) nekog HTML elementa - scenariji [izvor: autor]

Takođe, moguće je podesiti 1 - 4, različitih stilova prikaza linija granica. Navedeno se obavlja podešavanjem osobine `border-style` na neku od sledećih vrednosti (videti sliku 21):

- `dashed`
- `dotted`
- `double`
- `groove`
- `hidden`
- `inset`
- `none`
- `outset`
- `ridge`
- `solid`.



Slika 3.21 Demonstracija različitih stilova prikaza linija granica [izvor: autor]

CSS GRANICE – ŠIRINA

Koristeći osobinu `border-width` postavljamo širinu ivice za sve četiri stranice.

Koristeći osobinu *border-width* postavljamo širinu ivice za sve četiri stranice (gornju, desnu, donju i levu) – vrednosti : dužina, *thin*, *medium*, *thick*.

Obrazac za podešavanje ove osobine glasi: *border-width = vrednost;*

Podešavanje osobina pojedinačnih ivica obavlja se preko sledećih atributa:

- *border-top*;
- *border-right*:
- *border-bottom*;
- *border-left*.

Navedeno je moguće ilustrovati sledećim primerom.

The diagram illustrates the decomposition of the *border-top* property into its individual components. On the left, a box contains the declaration `border-top: 1px solid #0000ff;`. An arrow labeled "isto kao" (which means "as" or "the same as") points from this box to the right, where another box contains the expanded form: `border-top-width: 1px; border-top-style: solid; border-top-color: #0000ff;`.

Slika 3.22 Podešavanje osobina pojedinačnih ivica [izvor: autor]

CSS ŠIRINA I VISINA

CSS osobine širine i visine se ne nasleđuju.

CSS osobine širine i visine se ne nasleđuju:

1. *width* služi za određivanje širine elementa (auto, vrednost, % roditeljske širine);
2. *max-width* služi za određivanje maksimalne širine elementa;
3. *min-width* služi za određivanje minimalne širine elementa;
4. *height* služi za određivanje visine elementa (auto, vrednost, % roditeljske visine);
5. *line-height* služi određivanju visine između redova;
6. *max-height* služi za određivanje maksimalne visine elementa;
7. *min-height* služi za određivanje minimalne visine elementa;

CSS KLASIFIKACIJA (CLASSIFICATION)

Klasifikacija obuhvata set različitih CSS podešavanja.

Klasifikacija obuhvata set različitih CSS podešavanja koja se **ne prenose na potomke**:

- *float* - služi da promeni način na koji su tekst (ili slike) prikazani;
- *clear* - omogućena je kontrola nad elementima sa osobinom float
- *display* - menja način rasporeda elemenata na ekranu;
- *overflow* - služi za određivanje ponašanja delova elementa kada njegov sadržaj izlazi iz zadanih granica;
- *visibility* - određuje hoće li element biti prikazan na web stranici;
- *z-index* - određuje redosled pozicioniranja elemenata u sloju.
- *cursor* - određuje izgled pokazivača miša koji će se koristiti unutar elementa.

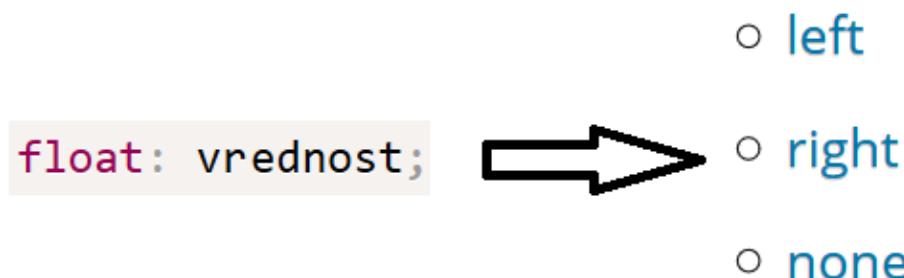
CSS KLASIFIKACIJA - FLOAT I CLEAR

Osobina float služi da promeni način na koji su tekst (ili slike) prikazani, clear to poništava.

Osobina `float` služi da promeni način na koji su tekst (ili slike) prikazani.

Kada dodelimo osobini vrednost `left` ili `right`, tekst (odnosno slika) će biti prikazani uz levi, odnosno desni rub roditeljskog elementa. U slučaju da dodelimo vrednost osobini `none` neće biti promene u prikazu teksta ili slike.

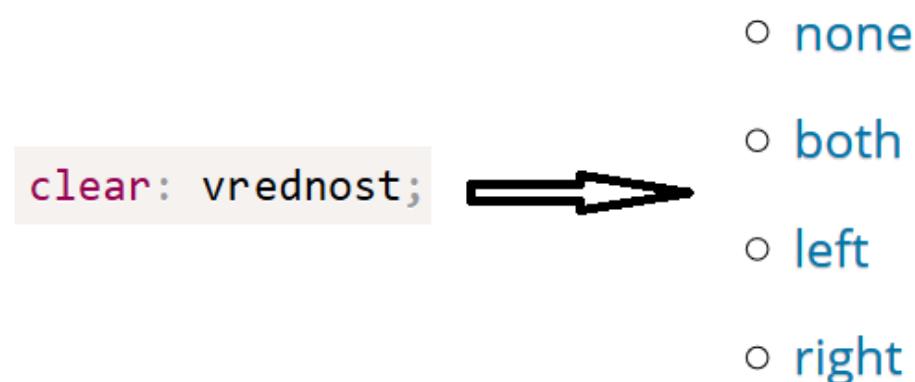
Navedeno može biti ilustrovano sledećom slikom:



Slika 3.23 Moguće vrednosti CSS osobine float [izvor: autor]

Osobinom `clear` omogućena je kontrola nad elementima sa osobinom `float`. Delovanjem osobine `float` možemo poništiti pomoću osobine `clear`.

Sledećom slikom su prikazane moguće vrednosti za ovu osobinu:



Slika 3.24 Moguće vrednosti CSS osobine clear [izvor: autor]

CSS KLASIFIKACIJA - DISPLAY I OVERFLOW

Osobinom display se menja način rasporeda elemenata na ekranu, overflow određuje njihovo ponašanje kada izlaze izvan zadatih granica.

Osobinom `display` se menja način rasporeda elemenata na ekranu. U HTML - u razlikujemo unapred određene elemente:

- *blok tipa* nakon kojih veb pregledač automatski prelazi u novi red i
- *linijskog (inline) tipa* elemenata nakon kojih veb pregledač ostaje u istom redu.

Primeri blok elemenata u html-u su: `<p>`, `<h1>...<h6>`, `<table>`...

Primeri linijskih (inline) elemenata su `<a>`, ``, ``...

Osobina `display` se primenjuje prema sledećem šablonu: `display: vrednost;`

Vrednosti mogu biti:

- `block`,
- `inline`,
- `list-item` ili
- `none`.

Osobina `overflow` služi za određivanje ponašanja delova elementa kada njegov sadržaj izlazi iz zadanih granica. Ukoliko sadržaj elementa prelazi granice širine ili visine, osobinom `overflow` možemo odrediti šta se događa sa sadržajem izvan područja elementa.

Sledećom slikom su prikazane moguće vrednosti ove osobine:

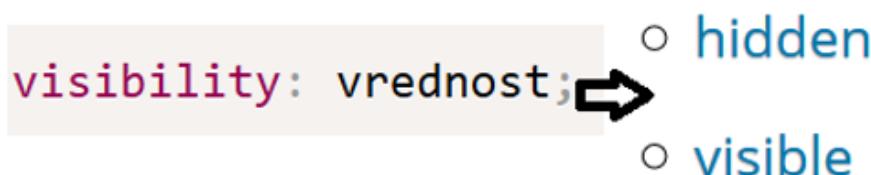


Slika 3.25 Vrednosti osobine overflow [izvor: autor]

CSS KLASIFIKACIJA – VISIBILITY, Z-INDEX I CURSOR

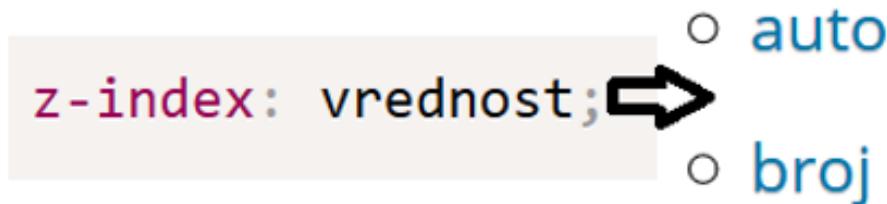
CSS klasifikacija obuhvata i osobine: visibility, z-index i cursor.

Osobina `visibility` određuje hoće li element biti prikazan na web stranici.



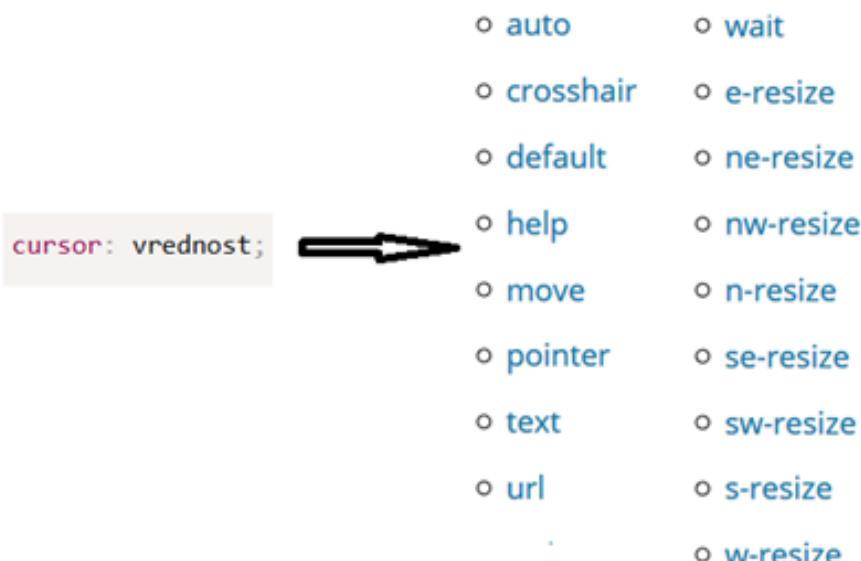
Slika 3.26 Osobina visibility - moguće vrednosti [izvor: autor]

Osobina **z-index** određuje redosled pozicioniranja elemenata u sloju.



Slika 3.27 Osobina z-index - moguće vrednosti [izvor: autor]

Osobinom **cursor** određujemo izgled pokazivača miša koji će se koristiti unutar elementa.



Slika 3.28 sobina cursor - moguće vrednosti [izvor: autor]

Ako se postavi slobodno odabrana slika kao pokazivač miša (primenom vrednosti osobine **url**), dobra je praksa deklarisati uz njega i neku od zadatih vrednosti:

```
cursor: url('slika.cur'), default;
```

Slika 3.29 Definisanje vlastitog igleda pokazivača [izvor: autor]

CSS POZICIONIRANJE

Osobinu position koristimo kada smo želimo da promenimo zadati položaj html elementa.

Osobinu **position** koristimo kada smo želimo da promenimo zadati položaj HTML elementa. Šablon po kojem se osobina podešava glasi:

position : vrednost;

Dozvoljene vrednosti su:

- *Static* - podrazumevana - obezbeđuje zadati raspored prikazivanja elementa;
- *Relative* - menja položaj po scenarijima *left*, *right*, *top* ili *bottom* u odnosu na zadati;
- *Absolute* - Element se pojavljuje u odnosu na gornji levi (*top left*) položaj najbližeg roditeljskog elementa;
- *Fixed* - Fiksno pozicioniranje u stranici ;

Primer CSS pozicioniranja može biti ilustrovan na sledeći način:

```
position: fixed;  
top: 10px;  
right: 10px;
```

Slika 3.30 Primer CSS pozicioniranja [izvor: autor]

CSS PSEUDO ELEMENTI

Sintaksa za pseudo elemente se nešto razlikuje u odnosu na uobičajeni kod CSS-a.

CSS pseudo elementi su komponente CSS sintakse koji se ne nasleđuje se sa roditelja na potomke. Sintaksa za pseudo elemente se nešto razlikuje u odnosu na uobičajeni kod CSS-a:

```
selector:pseudo-element {  
    svojstvo: vrednost;  
}
```

Slika 3.31 Sintaksa za pseudo elemente - osnovni oblik [izvor: autor]

Koristeći navedeni obrazac možemo dodeliti sledeće pseudo elemente:

- *first-line;*
- *first-letter*

```
p {  
    font-size: small;  
}  
p:first-line {  
    font-size: medium;  
    color: green;  
}  
  
p {  
    font-size: small;  
}  
p:first-letter {  
    font-size: medium;  
    color: red;  
}
```

Slika 3.32 Definisanje CSS pseudoelemenata [izvor: autor]

Sledećim video materijalom je moguće zaokružiti diskusiju o CSS3 osnovama.

Ova lekcija sadrži video materijal. Ukoliko želite da pogledate ovaj video morate da otvorite LAMS lekciju.

DODATNI CSS3 MODULI

CSS3 je podeljen u dva dela: stari CCS i dodatni moduli.

CSS3 je podeljen u dva dela:

- stari CSS - CSS3 podržava sve stavke starijih verzija CSS-a;
- dodatni moduli.

Dodatni moduli su:

- *Selectors*
- *Box Model*
- *Backgrounds and Borders*
- *Image Values and Replaced Content*
- *Text Effects*
- *2D/3D Transformations*
- *Animations*
- *Multiple Column Layout*
- *User Interface*

▼ Poglavlje 4

CSS3 - Pokazna vežba 1

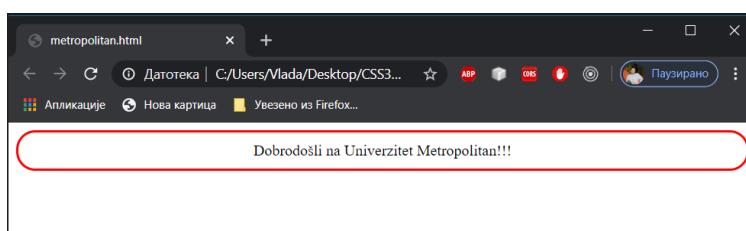
CSS3 FUNKCIONALNOSTI - PRIMER 1 I 2 (TRAJANJE 10 MIN)

Primer zaobljenih ivica i box-shadow.

Sledećim **HTML** listingom prikazan je kod unutar kojeg je pod tagom **<style>** ugrađen CSS kod za formatiranje **<div>** tagova (linije koda 5 - 10). Ideja je da tekst koji se nalazi unutar **<div>** taga bude ovičen punom linijom, crvene boje.

```
<!DOCTYPE html>
<html>
<head>
<style>
div{
    border: 2px solid red;
    padding: 10px 40px;
    border-radius: 20px;
    text-align: center;
}
</style>
</head>
<body>
    <div> Dobrodošli na Univerzitet Metropolitan!!!</div>
</body>
</html>
```

Kada se prikazani listing pokrene u veb pregledaču, dobija se sledeći prikaz:



Slika 4.1 Primer CSS stila - tekst sa zaobljenim granicama [izvor: autor]

Sledeći primer prikazuje sliku koje je učitana u osenčen boks (**CSS** osobina **box-shadow**) dimenzija 300px * 300px sa podešenom bojom pozadine i razmacima (linije koda 4 -11).

```
<!DOCTYPE html>
<html>
```

```
<head>
<style>
  div{
    width: 300px;
    height: 300px;
    background-color: #a70532;
    box-shadow: 10px 10px 5px #888888;
  }
</style>
</head>
<body>
  <div><img src = "https://fakulteti.edukacija.rs/wp-content/uploads/2015/10/
metropolitan-logo.png"/></div>
</body>
</html>
```



Slika 4.2 CSS - slika učitana u osenčen box [izvor: autor]

CSS3 FUNKCIONALNOSTI - PRIMER 3 (TRAJANJE 10 MIN)

Primer osenčenog teksta

Primer demonstrira primenu stila na tag `<H1>`. Tekst koji se nalazi unutar navedenog taga biće osenčen crvenom bojom (linije koda 4 - 9).

```
<!DOCTYPE html>
<html>
<head>
<style>
  h1{
    text-shadow: 5px 5px 5px #FF0000;
  }
</style>
</head>
<body>
  <H1>Univerzitet Metropolitan Beograd</H1>
```

```
</body>  
</html>
```

Univerzitet Metropolitan Beograd

Slika 4.3 Primer osenčenog teksta [izvor: autor]

✓ Poglavlje 5

CSS3 - Individualna vežba 1

INDIVIDUALNA VEŽBA 1 (TRAJANJE 20 MIN)

Samostalno kreiranje stilizovanog HTML dokumenta.

Zahtevi zadatka:

1. Detaljno prostudirajte primere sa pokaznih vežbi;
2. Kreirajte HTML dokument;
3. Unutar HTML dokumenta dodajte odgovarajući CSS;
4. Stilizovani HTML dokument mora da izgleda kao na sledećoj slici.



Slika 5.1 Rešenje individualne vežbe 1 [izvor: autor]

▼ Poglavlje 6

Bootstrap - osnove

UVOD U BOOTSTRAP

Bootstrap je frontend radni okvir za brz razvoj responzivnih veb sajtova.

Bootstrap je **frontend** radni okvir za brz razvoj responzivnih veb sajtova. Početak razvoja i korišćenja ovog okvira vezan je za kompaniju **Twitter** dok se danas koristi za širok spektar problema razvoja veb sistema, od razvoja veb aplikacija pa sve do kreiranja tema za predefinisane izglede veb stranica. Veoma je bitno naglasiti, iz perspektive veb programera i dizajnera, **Bootstrap** je besplatan za korišćenje, široko primenljiv i intuitivan.

Primenom Bootstrap - a moguće je na veoma jednostavan način kreirati složene veb stranice, oslanjajući se na standardni **HTML**, prilagođavajući ih konkretnim potrebama i zahtevima. Bootstrap se takođe isporučuje sa nizom **jQuery** dodataka koji mogu pružiti dodatne funkcije poput karusela (alat za prolaz kroz elemente), dugmadi, opisom alata i još mnogo toga.

Bootstrap daje, takođe, veliki broj prečica za kreiranje veb stranica štedeći na taj način vreme i trud veb dizajnera i programera. Primena ovog radnog okvira zahteva osnovno poznavanje **HTML** - a i **CSS** - a za kreiranje responzivnih veb stranica, kompatibilnih sa svim savremenim veb pregledačima.

Sledećim video materijalom je dat brz pregled razvoja responzivnih veb stranica primenom ovog frontend radnog okvira.

Ova lekcija sadrži video materijal. Ukoliko želite da pogledate ovaj video morate da otvorite LAMS lekciju.

▼ 6.1 Podešavanje Bootstrap okvira

INTEGRACIJA BOOTSTRAP-A

Neophodno je obaviti, na samom početku, integrisanje Bootstap-a u radno okruženje.

Sa ciljem upotrebe radnog okvira **Bootstrap**, neophodno je obaviti, na samom početku, njegovo integrisanje u radno okruženje, odnosno u konkretnu veb stranicu. Navedeno je moguće obaviti na primenom dva različita scenarija:

- učitavanjem Bootstrap -a sa udaljene lokacije;
- preuzimanjem i lokalnim instaliranjem Bootstrap-a.

U oba slučaja, potreban je resurs u koji će Bootstrap biti učitan, tj. konkretna veb stranica.

KREIRANJE HTML STRANICE

Kreira se radni folder sa HTML stranom za kasniju upotrebu Bootstrap-a.

U prvom koraku, neophodno je obaviti kreiranje jednostavne **HTML** stranice kao osnove za upotrebu **Bootstrap** radnog okvira za njenu stilizaciju i bogatiji izgled i funkcionalnosti. Kreira se radni folder na računaru ili serveru u kojem će biti čuvane datoteke projekta. U konkretnom slučaju, datoteka je nazvana "Primer 1" i nakon ovog izlaganja možete da preuzmete njoj odgovarajuću ZIP arhivu. Prva datoteka koja će biti kreirana unutar ovog foldera naziva se index.html i sledećim listingom je ilustrovan njen sadržaj.

```
<!DOCTYPE html>
<html lang="sr">
<head>
  <title>Bootstrap - uvodni primer</title>
  <meta charset="utf-8">
  <meta name="viewport" content="width=device-width, initial-scale=1">
</head>
<body>
</body>
</html>
```

UČITAVANJE BOOTSTRAP-A PREKO CDN

Moguće je koristiti CDN za daljinsko učitavanje Bootstrap-a u konkretne veb stranice.

Potrebno je istaći da se Bootstrap sastoji od stilova i skriptova koji mogu jednostavno biti učitani u zaglavlja (**header**) i podnožja (**footer**) veb stranica kao i bilo koji drugi CSS alati. Radni okvir koristi **CDN** (*Content Delivery Network*) pristupnu putanju za daljinsko učitavanje Bootstrap-a u konkretne veb stranice. Ovaj link možete pronaći na Bootstrap stranici za preuzimanje (link: <https://getbootstrap.com/docs/4.2/getting-started/download/>).

Za uključivanje Bootstrap-a u konkretnu veb stranicu, neophodno je dodati sledeći kod unutar **<head>** HTML taga:

```
<link rel="stylesheet"
      href="https://stackpath.bootstrapcdn.com/bootstrap/4.1.3/css/bootstrap.min.css"
      integrity="sha384-MCw98/SFnGE8fJT3GXwEOngsV7Zt27NXFoaoApmYm81iuXoPkFOJwJ8ERdknLPM0"
      crossorigin="anonymous">
```

Dodavanjem priloženog koda u kreiranu veb stranicu biće omogućeno da veb pregledač, po automatizmu, učita sve Bootstrap alate prilikom otvaranja stranice.

Upotreba prikazane udaljene metode angažovanja *Bootstrap*-a je odlična ideja koja omogućava da brojni korisnici imaju keširan radni okvir *Bootstrap* unutar veb pregledača. Na ovaj način nije potrebno njegovo ponovno učitavanje prilikom svakog poziva konkretnog veb sajta. Ovo za posledicu ima brže vreme učitavanja veb stranica. Navedeni pristup predstavlja metodu koja se predlaže u slučaju "živih" veb sajtova.

LOKALNO ANGAŽOVANJE BOOTSTRAP-A

Alternativno, Bootstrap-a se podešava preuzimanjem i raspakivanje, unutar kreiranog projekta.

Sa druge strane, za potrebe eksperimentisanja i razvoja, ili ako programeri i dizajneri ne žele da zavise od dostupnosti Internet konekcije, moguće je preuzeti na razvojni računar kopiju Bootstrap-a. Ovo će, takođe, biti analizirano i diskutovano budući da zahteva manju količinu koda kojeg je neophodno dodati.

Alternativni način podešavanja Bootstrap-a predstavlja njegovo preuzimanje, sa linka navedenog u prethodnom izlaganju, i raspakivanje unutar kreiranog foldera projekta. Ovde je neophodno preuzeti kompajlirane *CSS* i *JavaScript* datoteke - nije potreban izvorni kod.

Kada je sve ovo obavljeno moguće je učitati *Bootstrap CSS* alate u aktuelni projekat na sledeći način, unutar *<head>* taga *index.html* stranice:

```
<link rel="stylesheet" href="bootstrap/css/bootstrap.min.css">
```

Moguće je primetiti da kod uključuje putanju na kojoj je moguće pronaći Bootstrap datoteku. Neophodno je obaviti dodatnu proveru ta putanja odgovara aktuelnim podešavanjima budući da nazivi foldera mogu da se razlikuju u zavisnosti od preuzete verzije Bootstrap-a.

UKLJUČIVANJE JQUERY BIBLIOTEKE

Za potpune funkcionalnosti Bootstrap-a neophodno je obaviti učitavanje jQuery biblioteke.

Sa ciljem dobijanja potpune funkcionalnosti radnog okvira *Bootstrap*, neophodno je obaviti učitavanje *jQuery* biblioteke. Takođe, navedeno je moguće obaviti po udaljenom scenariju ili lokalnim angažovanjem nakon preuzimanja (sa linka: <https://code.jquery.com/>) i raspakivanja *jQuery* biblioteke unutar radnog foldera projekta.

Za udaljeno preuzimanje *jQuery* biblioteke neophodni je unutar *<body>* taga dodati sledeći kod:

```
<script src="https://code.jquery.com/jquery-3.3.1.slim.min.js"
integrity="sha384-q8i/X+965Dz00rT7abK41JStQIAqVgRVzbzo5smXKp4YfRvH+8abtTE1Pi6jizo"
crossorigin="anonymous"></script>
```

Alternativno, nakon preuzimanja i raspakivanja *jQuery* biblioteke, unutar radnog foldera, moguće je obaviti njeno dodavanje *<body>* taga na sledeći način:

```
<script src="jquery-3.3.1.min.js"></script>
```

Takođe, neophodno je obaviti dodatnu proveru aktuelnosti podešavanjima budući da nazivimogu da se razlikuju u zavisnosti od preuzete verzije jQuery biblioteke.

UKLJUČIVANJE BOOTSTRAP JAVASCRIPT BIBLIOTEKE

Poslednji korak podešavanja Bootstrap-a predstavlja učitavanje Bootstrap JavaScript biblioteke.

Poslednji korak podešavanja Bootstrap-a predstavlja učitavanje *Bootstrap JavaScript* biblioteke. Ova biblioteka je sadržana unutar preuzete verzije Bootstrap radnog okvira, kao i preko udaljenog pristupa. Međutim, razlikuje se mesto u stranici na kojem se angažuje. Umesto *<head>* taga, Bootstrap JavaScript biblioteka se uključuje u okviru *<body>* taga, odmah iza prethodno uključene *jQuery* biblioteke.

Koristeći udaljeni scenario angažovanja *Bootstrap JavaScript* biblioteke, neophodno je na pomenutu lokaciju dodati sledeći kod:

```
<script src="https://stackpath.bootstrapcdn.com/bootstrap/4.1.3/js/
bootstrap.min.js"
integrity="sha384-ChfqxuZUCnJSK3+MXmPNIyE6ZbWh2IMqE241rYiqJxyMiZ60W/JmZQ5stwEULTy"
crossorigin="anonymous"></script>
```

Koristeći lokalni scenario angažovanja Bootstrap JavaScript biblioteke, neophodno je na pomenutu lokaciju dodati sledeći kod:

```
<script src="bootstrap/js/bootstrap.min.js"></script>
```

Konačno, moguće je sve objediniti u jedinstveni listing. Sledećim listingom je prikazan kod kreiran po udaljenom scenariju:

```
<!DOCTYPE html>
<html lang="sr">
<head>
  <title>Bootstrap - uvodni primer</title>
  <meta charset="utf-8">
  <meta name="viewport" content="width=device-width, initial-scale=1">
  <link rel="stylesheet" href="https://stackpath.bootstrapcdn.com/bootstrap/4.1.3/css/
```

```
bootstrap.min.css" integrity="sha384-MCw98/  
SFnGE8fJT3GXwE0ngsV7Zt27NXFoaoApmYm81iuXoPkF0JwJ8ERdknLPM0" crossorigin="anonymous">  
</head>  
<body>  
  
<script src="https://code.jquery.com/jquery-3.3.1.slim.min.js"  
integrity="sha384-q8i/X+965Dz00rT7abK41JStQIAqVgRVzbzo5smXKp4YfRvH+8abTE1Pi6jizo"  
crossorigin="anonymous"></script>  
  
<script src="https://stackpath.bootstrapcdn.com/bootstrap/4.1.3/js/  
bootstrap.min.js"  
integrity="sha384-ChfqqxuZUCnJSK3+MXmPNIyE6ZbWh2IMqE241rYiqJxyMiZ60W/JmZQ5stwEULTy"  
crossorigin="anonymous"></script>  
  
</body>  
</html>
```

Primenom lokalne implementacije Bootstrap okvira, dobija se sledeći listing:

```
<!DOCTYPE html>  
<html lang="sr">  
  <head>  
    <title>Bootstrap - uvodni primer</title>  
    <meta charset="utf-8">  
    <meta name="viewport" content="width=device-width, initial-scale=1">  
    <link rel="stylesheet" href="bootstrap/css/bootstrap.min.css">  
  </head>  
  <body>  
    <script src="jquery-3.3.1.min.js"></script>  
    <script src="bootstrap/js/bootstrap.min.js"></script>  
  </body>  
</html>
```

▼ Poglavlje 7

Primena Bootstrap okvira

DIZAJNIRANJE VEB STRANICE

Akcenat će biti na primeni Bootstrap-a na dizajnu kreirane stranice.

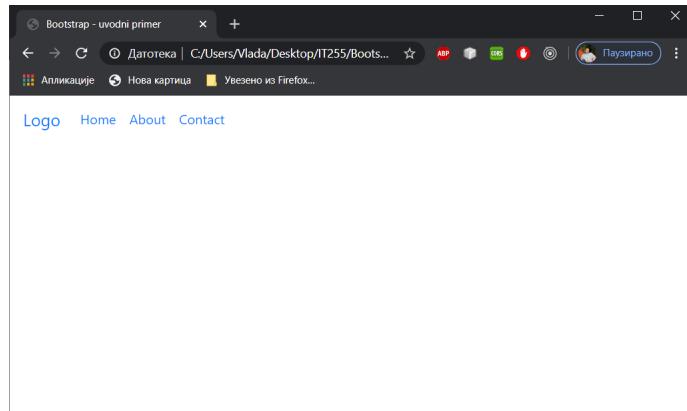
U prethodnom izlaganju objašnjeno je i demonstrirano podešavanje Bootstrap okvira kao preduslov za njegovu primenu u daljem veb razvoju. U nastavku, akcenat će biti na njegovoj primeni na dizajnu kreirane stranice *index.html* koja je za sada prazna.

Kao prvi korak u dizajnu navedene stranice moguće je dodati navigacioni bar na vrh stranice. Preko ovog menija posetnici stranice će moći da obavljaju navigaciju iz stranice i otkrivanje preostalog dela sajta. Za realizaciju navedenog biće upotrebljena klasa pod nazivom *navbar*. Ova klasa je jedan od osnovnih *Bootstrap* elemenata. Njenom primenom kreira se navigacioni element koji je responzivan po osnovnim podešavanjima i automatski će se prilagoditi manjim ekranima. Takođe, poseduje ugrađenu podršku za dodavanje: brendiranja, šema boja, rastojanja, kao i ostalih komponenata.

Dodavanje navigacionog bara je moguće uključivanjem sledećeg koda u *<body>* tag posmatrane HTML stranice:

```
<nav class="navbar navbar-expand-md">
  <a class="navbar-brand" href="#">Logo</a>
  <button class="navbar-toggler navbar-dark" type="button" data-toggle="collapse" data-target="#main-navigation">
    <span class="navbar-toggler-icon"></span>
  </button>
  <div class="collapse navbar-collapse" id="main-navigation">
    <ul class="navbar-nav">
      <li class="nav-item">
        <a class="nav-link" href="#">Home</a>
      </li>
      <li class="nav-item">
        <a class="nav-link" href="#">About</a>
      </li>
      <li class="nav-item">
        <a class="nav-link" href="#">Contact</a>
      </li>
    </ul>
  </div>
</nav>
```

Navedeni listing će dati rezultat prikazan sledećom stranicom.



Slika 7.1 Kreiran navigacioni bar primenom Bootstrap-a [izvor: autor]

NAVIGACIONI BAR - POJAŠNJENJA

Prethodno priloženi kod neophodno je precizno pojasniti.

Prethodno priloženi kod neophodno je precizno pojasniti. Korišćene su sledeće CSS klase:

- **navbar-expand-md** - Ovo označava u kom se trenutku navigaciona traka širi od vertikalne ili "hamburger" ikone (slika 2) do horizontalne trake pune veličine (slika 3). U ovom slučaju podešena je na srednje ekrane, što je u Bootstrapu znači širinu veću od 768px.
- **navbar-brand** - Koristi se za brendiranje veb sajta. Ovde je moguće uključiti datoteku slike sa logoom sajta;
- **navbar-toggler** - Označava dugme prekidač (*toggle button*) za prikazivanje i skrivanje menija dobijenog skupljanjem navigacione trake (bara). Detalj *data-toggle="collapse"* ukazuje da će skupljanjem ekrana navigaciona traka prikazivati "hamburger" ikonicu, umesto padajućeg menija, koji je druga opcija koja se dobija kada se klikne na ovu ikonicu. Veoma je važno da se obavi definisanje *data-target* koji sadrži *id* (označen sa *#*) stvarnog *navbar* elementa obuhvaćenog *<div>* tagom;
- **navbar-toggler-icon** - Kreira ikonu menija na koju korisnik treba da klikne za pristup kreiranim opcijama kada je stranica učitana u manje ekrane;
- **navbar-nav** - klasa za kreiranje liste sa stavkama primenom elemenata *nav-item* u kojima su integrисани elementi *nav-link*.



Slika 7.2 Hamburger ikona i meni u menjem prozoru [izvor: autor]



Slika 7.3 Navigaciona traka u većem prozoru [izvor: autor]

Moguće je primetiti da stranica trenutno ima drugačiji izgled nego što je prikazano slikom 1. Razlog je što je pristupljeno dodavanju dodatnog CSS. Upravo je diskusija koja predstoji u narednom izlaganju.

DODAVANJE CSS PUTEM EKSTERNE DATOTEKE

Moguće je koristiti vlastite CSS datoteke kojima je moguće preklopiti već primenjene stilove.

Rada sa Bootstrap-om odlikuje se velikom fleksibilnošću. Tako je na sadržaj kreirane stranice moguće dodati dodatni CSS za njen još atraktivniji izgled. Ukoliko je cilj promena podrazumevanog Bootstrap stila, nije potrebno koristiti neku veliku biblioteku stilova i ručno obaviti željene izmene. Umesto toga, moguće je koristiti vlastite CSS datoteke kojima je moguće preklopiti već primenjene stilove.

Da bi navedeno bilo moguće realizovati, u radnom folderu biće kreiran podfolder **CSS** i unutar njega datoteka **main.css**. Sadržaj ove datoteke je prikazan sledećim listingom:

```
body {  
    padding: 0;  
    margin: 0;  
    background: #f2f6e9;  
}  
  
/*--- navigation bar ---*/  
  
.navbar {  
    background:#6ab446;  
}  
  
.nav-link,  
.navbar-brand {  
    color: #fff;  
    cursor: pointer;  
}  
  
.nav-link {  
    margin-right: 1em !important;  
}
```

```
.nav-link:hover {  
    color: #000;  
}  
  
.navbar-collapse {  
    justify-content: flex-end;  
}
```

Da bi stilovi definisani ovom datotekom mogli da budu primenjeni na posmatranu stranicu *index.html*, neophodno je u *<head>* tagu ove stranice dodati sledeći kod:

```
<link rel="stylesheet" type="text/css" href="CSS/main.css">
```

Rezultat primene dodatnih stilova je prikazan slikama 2 i 3.

KREIRANJE KONTEJNERA SADRŽAJA STRANICE

Neophodno je pristupiti kreiranju kontejnera u kojem će biti pakovan sadržaj stranice.

Nakon kreiranja navigacione trake neophodno je pristupiti kreiranju kontejnera u kojem će biti pakovan sadržaj stranice. Navedeno je veoma jednostavno postići u Bootstrap radnom okviru. Sve što je neophodno jeste dodavanje sledećeg koda, odmah ispod taga *<nav>*:

```
***  
</nav>  
<header class="page-header header container-fluid">  
  
</header>  
***
```

Iz priloženog listinga je moguće primetiti upotrebu klase *container-fluid*. Ovo je još jedna od osnovnih Bootstrap klasa koja automatski na odgovarajuće *<div>* elemente primenjuje veliki broj CSS podešavanja.

Deo *-fluid* ukazuje da se kontejner proteže celokupnom širinom ekrana. Takođe, postoji i mogućnost podešavanja kontejnera sa fiksnom širinom tako da je, u tom slučaju, moguće koristiti prostor na oba dela ekrana.

Ako se, nakon obavljenih izmena, ponovo učita stranica, neće biti moguće primetiti nikakav dodatni sadržaj. Razlog ovome je što je kreiran prezan HTML element. Ovo znači da je neophodno krenuti dalje sa razvojem.

DODAVANJE SLIKE POZADIJE I JAVASCRIPT-A

Sa malo jQuery koda je omogućeno rastezanje slike preko celokupnog ekrana.

Kao sledeći korak, primene Bootstrap okvira, u razvoju konkretne veb stranice može biti dodavanje pozadinske slike preko celokupne veličine ekrana unutar prethodno kreiranog kontejnera (`<header>` taga). U ovu svrhu se koristi malo `jQuery` koda kojim je omogućeno "rastezanje" slike preko celokupnog ekrana.

Ovo se obavlja na potpuno isti način kao i dodavanje korisničkog CSS-a. Prvo, kreira se tekstualna datoteka pod nazivom `main.js` koja se čuva unutar foldera projekta. Ova datoteka se poziva pre zatvaranja taga `</body>` na način prikazan sledećim listingom:

```
***  
</header>  
  
<script src="main.js"></script>  
</body>  
***
```

Kada je ovo obavljeno, neophodno je u kreiranu datoteku dodati malo koda na način prikazan sledećim listingom:

```
$(document).ready(function(){  
    $('.header').height($(window).height());  
})
```

Jedina preostala stvar jeste dodavanje pozadinske slike. To se veoma jednostavno radi dodavanjem sledećeg koda unutar datoteke main.css.

```
.header {  
    background-image: url('images/header-background.jpg');  
    background-size: cover;  
    background-position: center;  
    position: relative;  
}
```

Važno je napomenuti da je slika `header-background.jpg` preuzeta na vreme i postavljena unutar podfoldera `images`, foldera `CSS`.

Ako se sada obavi učitavanje stranice index.html, dobija se sledeći rezultat:



Slika 7.4 Stranica sa podešenom slikom pozadine [izvor: autor]

PREKRIVANJE (OVERLAY) POZADINE

Na sliku pozadine moguće je primeniti prekrivanje.

Na sliku pozadine moguće je primeniti prekrivanje ([overlay](#)). Na taj način će pozadina biti dodatno stilizovana. Ovo se takođe, primenom Bootstrap-a postiže na jednostavan način. Sve što je potrebno jeste dodavanje novog `<div>` taga unutar prethodno kreiranog `<header>`, a to je prikazano sledećim listingom:

```
***  
<header class="page-header header container-fluid">  
  
<div class="overlay"></div>  
  
</header>  
<script src="main.js"></script>  
***
```

Sada je potrebno dovršiti podešavanje dodavanjem podešavanja klase [overlay](#) unutar datoteke [main.css](#):

```
.overlay {  
    position: absolute;  
    min-height: 100%;  
    min-width: 100%;  
    left: 0;  
    top: 0;  
    background: rgba(0, 0, 0, 0.6);  
}
```

Kao što je moguće primetiti, definisan je nivo prozirnosti prekrivanja pozadinske slike.

Ako se ponovo učita stranica, dobija je još stilizovaniji rezultat prikazan sledećom slikom:



Slika 7.5 Prekrivanje slike sa definisanim nivoom prozirnosti [izvor: autor]

DODAVANJE NASLOVA STRANICE I TEKSTA

Postavljaju se dva nova zahteva: dodavanje naslova stranice i teksta.

Primenom Bootstrap-a moguće je dalje nastaviti razvoj započete veb stranice. Postavljaju se dva nova zahteva:

- kreiranje naslova stranice;
- dodavanje teksta u stranicu (unutar <body> taga).

Kada se ovo obavi, posetioci sajta će odmah znati na kojem se sajtu nalaze i koje informacije i aktivnosti mogu na njemu da očekuju.

Da bi navedeno bilo realizovano, neophodno je dodati sledeći kod unutar kreiranog kontejnera, podešenog u prethodnom slučaju, odmah ispod dela za prekrivanje (overlay) pozadine.

```
****  
<header class="page-header header container-fluid">  
  
<div class="overlay"></div>  
  
<div class="description">  
  <h1>Dobrodošli na Univerzitet Metropolitan!</h1>  
  <p>Prvi po zadovoljstvu studenata!!!</p>  
</div>  
  
</header>  
****
```

Moguće je primetiti da je dodati `<div>` element stilizovan CSS klasom pod nazivom `description`. To znači da se sada moguće dodati kompletну definiciju ove klase u dokument `main.css`, baš kao što je to urađeno na sledeći način:

```
.description {  
    left: 50%;  
    position: absolute;  
    top: 45%;  
    transform: translate(-50%, -55%);  
    text-align: center;  
}  
.description h1 {  
    color: #6ab446;  
}  
.description p {  
    color: #fff;  
    font-size: 1.3rem;  
    line-height: 1.5;  
}
```



Slika 7.6 Dodavanje naslova stranice i teksta [izvor: autor]

DEFINISANJE CTA DUGMETA

Uglavnom, CTA se realizuje putem dugmeta.

Kada se razvija savremena veb stranica, pogotovo početna, nije moguće zamisliti nepostojanje **CTA** (*Call To Action*) kontrole. Uglavnom, **CTA se realizuje putem dugmeta** (ili odgovarajućeg linka). Iz navedenog razloga, prelazi se na uključivanje jedne takve kontrole u aktuelnu stranicu.

Bootstrap radni okvir omogućava realizovanje navedenog na brojne načine kroz postojanje brojnih alata za kreiranje dugmadi na brz i lak način. U konkretnom slučaju, ovo je obavljeno dodavanjem sledećeg koda u nastavku kreiranog kontejnera za prikazivanje sadržaja:

```
***  
<div class="description">  
    <h1>Dobrodošli na Univerzitet Metropolitan!</h1>  
    <p>Prvi po zadovoljstvu studenata!!!</p>
```

```
<button class="btn btn-outline-secondary btn-lg">Više o Univerzitetu!</button>  
</div>  
***
```

Sada je moguće u *main.css* proširiti primenu CSS klase *description* na kreirano dugme, kao što je to urađeno na sledeći način:

```
.description button {  
    border:1px solid #6ab446;  
    background:#6ab446;  
    border-radius: 0;  
    color:#fff;  
}  
.description button:hover {  
    border:1px solid #fff;  
    background:#fff;  
    color:#000;  
}
```

Ponovnim učitavanjem stranice dobija se sledeći rezultat:



Slika 7.7 Definisanje CTA dugmeta [izvor: autor]

PRIKAZIVANJE SADRŽAJA U VIŠE KOLONA

U nastavku, cilj je prikazivanje dodatnog sadržaja u tri kolone.

Korak po korak, stranica dobija sve bogatiji izgled. U nastavku, cilj je prikazivanje dodatnog sadržaja u tri kolone, odmah ispod glavnog sadržaja, koji je bio predmet do sadašnjeg razvoja. Ova aktivnost predstavlja pravu "specijalnost" radnog okvira Bootstrap i oslanja se na kreiranje matrične (grid) strukture (više na linku: <https://getbootstrap.com/docs/4.0/layout/grid/>).

Za dodavanje dodatnog sadržaja u stranicu, u formi tri kolone, neophodno je dodati sledeći kod odmah ispod glavnog sadržaja (*<header>* taga).

```
</header>

<div class="container features">
  <div class="row">
    <div class="col-lg-4 col-md-4 col-sm-12">
      <h3 class="feature-title">Centar u Beogradu</h3>
      
      <p>Univerzitet Metropolitan - Centar u Beogradu</p>
    </div>
    <div class="col-lg-4 col-md-4 col-sm-12">
      <h3 class="feature-title">Centar u Nišu</h3>
      
      <p>Univerzitet Metropolitan - Centar u Nišu</p>
    </div>
    <div class="col-lg-4 col-md-4 col-sm-12">
    </div>
  </div>
</div>

<script src="main.js"></script>
```

Prva stvar koja se primećuje je postojanje elementa *row*. Ovo je potrebno svaki put kada se kreiraju kolone koje vrše ulogu kontejnera za matricu (*grid*).

Kolone su stilizovane različitim klasama: *col-lg-4*, *col-md-4* i *col-sm-12*. ovo ukazuje da se radi o kolonama i koje će veličine imati kada se prikazuju u različitim veličinama.

Da biste razumeli napisano morate da razumete da *Bootstrap* u mreži (grid-u) sve kolone u jednom redu dodaje do broja 12. Dajući im gore navedene klase znači da će one zauzeti trećinu ekrana, kada se stranica učitava u veliki ili srednji ekran ($12 : 3 = 4$) ili ceo ekran kada se stranica učitava u ekran male širine ($12 : 1 = 12$).

Dalje, primećuje se da su u kolone uključene slike i da je dodata klasa *.image-fluid* za njih. Ovo će ih učini responzivnim i prilagodljivim u odnosu na ekran u kojem se stranica učitava.

DODAVANJE FORME U KOLONU

U treću kolonu će biti dodata forma preko koje će korisnici moći da dodaju određene podatke.

Da bi prethodni posao stilizovali, neophodno je pozabaviti se stilom određenim klasom *features* (.

```
.features {
  margin: 4em auto;
  padding: 1em;
  position: relative;
}
.feature-title {
  color: #333;
```

```
font-size: 1.3rem;  
font-weight: 700;  
margin-bottom: 20px;  
text-transform: uppercase;  
}  
.features img {  
-webkit-box-shadow: 1px 1px 4px rgba(0, 0, 0, 0.4);  
-moz-box-shadow: 1px 1px 4px rgba(0, 0, 0, 0.4);  
box-shadow: 1px 1px 4px rgba(0, 0, 0, 0.4);  
margin-bottom: 16px;  
}
```



Slika 7.8 Prikazivanje sadržaja po kolonama [izvor: autor]

Primećuje se sa slike da nedostaje treća kolona. U ovu kolonu će biti dodata forma preko koje će korisnici moći da dodaju određene podatke. Tada će stranica dobiti sledeći izgled:



Slika 7.9 Dodavanje forme u kolonu [izvor: autor]

Da bi ovo bilo realizovano opet će biti zatražena pomoć radnog okvira Bootstrap.

KODIRANJE FORME

Kodiranjem forme dodaje se i treća kolona u grid.

Sa prethodne slike je moguće sagledati kako će izgledati stranica kada se doda odgovarajući sadržaj u treću, sada praznu, kolonu. Za početne stranice veb aplikacije (sajta) je česta praksa da postoji određeno mesto preko kojeg posetioci mogu da budu u kontaktu i da ostave vlastite podatke.

Kreiranje kontakt forme u Bootstrap-u je veoma jednostavno i dovoljno je da se u ostavljenu praznu `<div>` sekciju doda sledeći kod:

```
<div class="col-lg-4 col-md-4 col-sm-12">

    <h3 class="feature-title">Budimo u kontaktu!</h3>
    <div class="form-group">
        <input type="text" class="form-control" placeholder="Ime" name="">
    </div>
    <div class="form-group">
        <input type="email" class="form-control" placeholder="email adresa" name="email">
    </div>
    <div class="form-group">
        <textarea class="form-control" rows="4"></textarea>
    </div>
    <input type="submit" class="btn btn-secondary btn-block" value="Pošalji" name="">

</div>
```

Ovde je potrebno dodati malo dodatnog objašnjenja. Korišćene su posebne klase:

- **form-group** — Postavlja omotač oko polja forme za formatiranje.
- **form-control** — Označava polja forme kao ulaze, polja za unos teksta i slično.

Mnogo toga još je moguće uraditi sa formama, za dodatno istraživanje možete da konsultujete sledeću dokumentaciju: <https://getbootstrap.com/docs/4.0/components/forms/>. Za uvodna razmatranja i sticanje predstave o primeni Bootstrap-a za dodavanje formi u stranice, obrađeno je sasvim dovoljno. Rad na dodavanju forme, kao elementa treće kolone, završava se dodavanjem odgovarajućeg stila u *main.css*:

```
.features .form-control,
.features input {
    border-radius: 0;
}
.features .btn {
    background-color: #589b37;
    border: 1px solid #589b37;
    color: #fff;
    margin-top: 20px;
}
.features .btn:hover {
    background-color: #333;
    border: 1px solid #333;
}
```

DODAVANJE PODNOŽJA (FOOTER-A)

Karakterističan detalj za svaku početnu stranicu nekog veb sajta jeste sekcija podnožja.

Diskusija je veoma blizu završnim razmatranjima. Poslednji detalj koji je karakterističan za svaku početnu stranicu nekog veb sajta jeste sekcija podnožja (footer) sa dodatnim informacijama. Ideja je da se ova sekcija takođe podeli na dve kolone, a to se veoma lako rešava primenom radnog okvira Bootstrap.

Odmah ispod prethodne sekcije, a ispred JavaScript `<script>` taga, neophodno je dodati sledeći kod (spakovan kao sadržaj taga `<footer>`) unutar datoteke `index.html`:

```
</div>

<footer class="page-footer">
  <div class="container">
    <div class="row">
      <div class="col-lg-8 col-md-8 col-sm-12">
        <h6 class="text-uppercase font-weight-bold">Dodatne informacije</h6>
        <p>Fakultet informacionih tehnologija
          <br/>Fakultet digitalnih umetnosti
          <br/>Fakultet za menadžment</p>
      </div>
      <div class="col-lg-4 col-md-4 col-sm-12">
        <h6 class="text-uppercase font-weight-bold">Kontakt</h6>
        <p>Tadeuša Košćuška 63, Beograd, Srbija
          <br/>Svetog Cara Konstantina 80-86, Niš, Srbija
          <br/>info@metropolitan.ac.rs
          <br/>011 203 088 5
          <br/>018 551 000</p>
      </div>
    </div>
    <div class="footer-copyright text-center">© 2020 Copyright: IT255 - Veb sistemi
  1</div>
</footer>

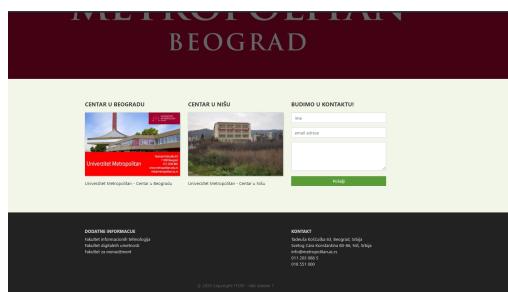
<script src="main.js"></script>
```

Pored deljenja sadržaja podnožja stranice u dve kolone, ova sekcija ističe dodatna podešavanja za modifikaciju tipografije primenom Bootstrap-a:

- `text-uppercase` - prevodenje slova u velika slova;
- `font-weight-bold` - boldiranje teksta;
- `text-center` - centriranje teksta;

Diskusija se završava dodavanjem stila za podnožje u `main.css`.

```
.page-footer {  
background-color: #222;  
color: #ccc;  
padding: 60px 0 30px;  
}  
.footer-copyright {  
color: #666;  
padding: 40px 0;  
}
```



Slika 7.10 Dodavanje podnožja (footer-a) [izvor: autor]

VIDEO MATERIJAL

Responsive Bootstrap Website Start To Finish with Bootstrap 4, HTML5 & CSS3

Ova lekcija sadrži video materijal. Ukoliko želite da pogledate ovaj video morate da otvorite LAMS lekciju.

✓ Poglavlje 8

Dodatna literatura za rad

DODATNA LITERATURA

Proširite vaše znanje posetom sledećih linkova.

Proširite vaše znanje posetom sledećih linkova:

1. <https://www.w3schools.com/css/>
2. <https://www.tutorialspoint.com/css/index.htm>
3. <https://www.javatpoint.com/css-tutorial>
4. <https://www.w3schools.com/bootstrap/>
5. <https://www.w3schools.com/bootstrap4/>
6. <https://www.tutorialspoint.com/bootstrap/index.htm>

✓ Poglavlje 9

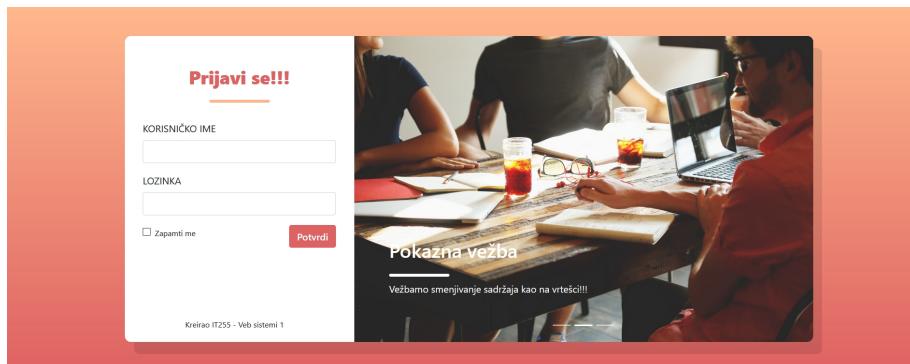
Bootstrap - Pokazna vežba 2

BOOTSTRAP - POKAZNA VEŽBA (TRAJANJE: 25 MINUTA)

Bootstrap vežbanje

Zadatak:

1. Kreirajte HTML stranicu čije učitavanje odgovara rešenju sa slike 1;
2. Koristeći Bootstrap, stilizujte stranicu na način prikazan slikom 1;
3. Koristite slike po želji;
4. Kreirajte CSS datoteku main.css da biste dobili rešenje kao na slici1.



Slika 9.1 Rešenje pokazne vežbe [izvor: autor]

Program pored obrađenih elemenata demonstrira primenu Bootstrap funkcionalnosti *carousel* (vrteška) unutar koje se tri slajda (*carousel-item*) ciklično smenjuju po zadatoj animaciji. Aplikacija koristi grid sa dve kolone: forma i vrteška. Takođe, dodat je *carouselExampleIndicators* za pokazivanje na kojem se trenutno slajdu u vrtešci nalazite.

U sekciji Shared Resources, odmah iza ovog objekta učenja, imate priloženo rešenje koje možete da preuzmete i testirate.

BOOTSTRAP - POKAZNA VEŽBA - LISTINZI

Rešenje je realizovano kroz dve datoteke: index.html i main.css.

Rešenje je realizovano kroz dve datoteke: index.html i main.css.

Sledi listing datoteke index.html.

```
<!DOCTYPE html>
<html lang="sr">
<head>
    <title>Bootstrap - uvodni primer</title>
    <meta charset="utf-8">
    <meta name="viewport" content="width=device-width, initial-scale=1">
<link rel="stylesheet" href="https://stackpath.bootstrapcdn.com/bootstrap/4.1.3/css/
bootstrap.min.css" integrity="sha384-MCw98/
SFnGE8fJT3GXwE0ngsV7Zt27NXFoaoApmYm81iuXoPkF0JwJ8ERdknLPM0" crossorigin="anonymous">

<link rel="stylesheet" type="text/css" href="CSS/main.css">
</head>
<body>

<script src="https://code.jquery.com/jquery-3.3.1.slim.min.js"
integrity="sha384-q8i/X+965Dz00rT7abK41JStQIAqVgRVzbzo5smXKp4YfRvH+8abTE1Pi6jizo"
crossorigin="anonymous"></script>

<script src="https://stackpath.bootstrapcdn.com/bootstrap/4.1.3/js/
bootstrap.min.js"
integrity="sha384-ChfqxuZUCnJSK3+MXmPNIyE6ZbWh2IMqE241rYiqJxyMiZ60W/JmZQ5stwEULTy"
crossorigin="anonymous"></script>

<section class="login-block">
    <div class="container">
        <div class="row">
            <div class="col-md-4 login-sec">
                <h2 class="text-center">Prijava se!!!</h2>
                <form class="login-form">
                    <div class="form-group">
                        <label for="exampleInputEmail1" class="text-uppercase">Korisničko ime</label>
                        <input type="text" class="form-control" placeholder="">
                    </div>
                    <div class="form-group">
                        <label for="exampleInputPassword1" class="text-uppercase">Lozinka</label>
                        <input type="password" class="form-control" placeholder="">
                    </div>

                    <div class="form-check">
                        <label class="form-check-label">
                            <input type="checkbox" class="form-check-input">
                            <small>Zapamti me</small>
                        </label>
                        <button type="submit" class="btn btn-login float-right">Potvrdi</button>
                    </div>
                </form>
                <div class="copy-text">Kreirao <i class="fa fa-heart"></i> IT255 - Veb sistemi
1</div>
    </div>
</div>
```

```
</div>
<div class="col-md-8 banner-sec">
    <div id="carouselExampleIndicators" class="carousel slide"
data-ride="carousel">
        <ol class="carousel-indicators">
            <li data-target="#carouselExampleIndicators" data-slide-to="0"
class="active"></li>
            <li data-target="#carouselExampleIndicators"
data-slide-to="1"></li>
            <li data-target="#carouselExampleIndicators"
data-slide-to="2"></li>
        </ol>
        <div class="carousel-inner" role="listbox">
            <div class="carousel-item active">
                
                <div class="carousel-caption d-none d-md-block">
                    <div class="banner-text">
                        <h2>Pokazna vežba</h2>
                        <p>Šta je karusel?</p>
                    </div>
                </div>
            </div>
            <div class="carousel-item">
                
                <div class="carousel-caption d-none d-md-block">
                    <div class="banner-text">
                        <h2>Pokazna vežba</h2>
                        <p>Vežbamo smenjivanje sadržaja kao na vrtešci!!!</p>
                    </div>
                </div>
            </div>
            <div class="carousel-item">
                
                <div class="carousel-caption d-none d-md-block">
                    <div class="banner-text">
                        <h2>Pokazna vežba</h2>
                        <p>Poslednja slika u karuselu</p>
                    </div>
                </div>
            </div>
        </div>
    </div>
</div>
</section>

<script src="main.js"></script>
```

```
</body>  
</html>
```

Sledi listing datoteke main.css.

```
@import url("//netdna.bootstrapcdn.com/font-awesome/4.0.3/css/font-awesome.css");  
.login-block{  
    background: #DE6262; /* fallback for old browsers */  
background: -webkit-linear-gradient(to bottom, #FFB88C, #DE6262); /* Chrome 10-25,  
Safari 5.1-6 */  
background: linear-gradient(to bottom, #FFB88C, #DE6262); /* W3C, IE 10+/ Edge,  
Firefox 16+, Chrome 26+, Opera 12+, Safari 7+ */  
float:left;  
width:100%;  
padding : 50px 0;  
}  
.banner-sec{background:url(https://static.pexels.com/photos/33972/  
pexels-photo.jpg) no-repeat left bottom; background-size:cover; min-height:500px;  
border-radius: 0 10px 10px 0; padding:0;}  
.container{background:#fff; border-radius: 10px; box-shadow:15px 20px 0px  
rgba(0,0,0,0.1);}  
.carousel-inner{border-radius:0 10px 10px 0;}  
.carousel-caption{text-align:left; left:5%;}  
.login-sec{padding: 50px 30px; position:relative;}  
.login-sec .copy-text{position:absolute; width:80%; bottom:20px; font-size:13px;  
text-align:center;}  
.login-sec .copy-text i{color:#FEB58A;}  
.login-sec .copy-text a{color:#E36262;}  
.login-sec h2{margin-bottom:30px; font-weight:800; font-size:30px; color: #DE6262;}  
.login-sec h2:after{content:" "; width:100px; height:5px; background:#FEB58A;  
display:block; margin-top:20px; border-radius:3px;  
margin-left:auto; margin-right:auto}  
.btn-login{background: #DE6262; color:#fff; font-weight:600;}  
.banner-text{width:70%; position:absolute; bottom:40px; padding-left:20px;}  
.banner-text h2{color:#fff; font-weight:600;}  
.banner-text h2:after{content:" "; width:100px; height:5px; background:#FFF;  
display:block; margin-top:20px; border-radius:3px;}  
.banner-text p{color:#fff;}
```

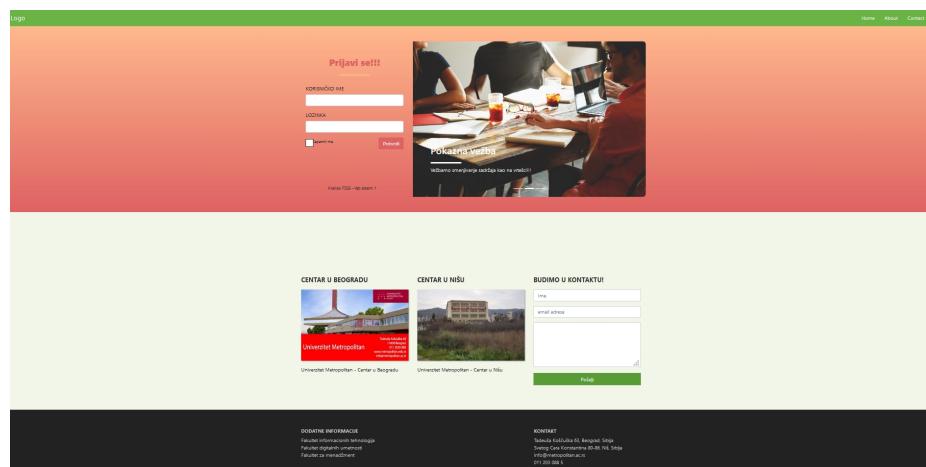
✓ Poglavlje 10

Bootstrap - Individualna vežba 2

INDIVIDUALNA VEŽBA 2 (TRAJANJE: 70 MINUTA)

Na osnovu predavanja i pokaznih vežbi radite samostalno na sledećem zadatku.

1. Koristite pokazne primere sa predavanja i vežbi;
2. Uradite samostalno zadatak čije je rešenje prikazano slikom 2;
3. U slučaju nejasnoća i zastoja u radu kontaktirajte asistenta.
4. Kada uradite zadatak, pozovite asistenta da proveri Vaš rad.



Slika 10.1 Rešenje individualne vežbe [izvor: autor]

✓ Poglavlje 11

Domaći zadatak

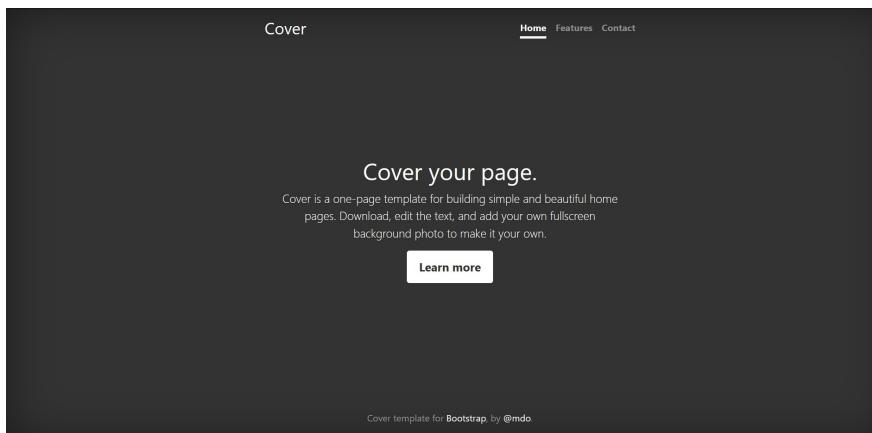
ZADATAK (PREDVIĐENO VREME 180 MINUTA)

Samostalna izrada DZ iz oblasti CSS i Bootstrap okvira.

Koristeći [CSS](#) i [Bootstrap](#) znanje stečeno na predavanjima, pokaznim i individualnim vežbama kreirajte vaše stranice po sledećim zahtevima:

1. sadržaj stranica - tekst i slike, prilagodite vašim željama i potrebama;
2. boje - prilagodite vašim željama i potrebama;
3. formati stranica moraju da odgovaraju stranicama priloženim slikama 1 i 2;
4. **domaći zadatak dokumentovati, poslati vašem predmetnom asistentu i spremiti usmenu odbranu.**

Domaći zadatak 1 uraditi po uzoru na sledeću sliku:



Slika 11.1 Model za izradu domaćeg zadatka 1 [izvor: autor]

Domaći zadatak 2 uraditi po uzoru na sledeću sliku:

The screenshot shows a pricing table template. At the top left is a placeholder 'Company name'. On the right are links for 'Features', 'Enterprise', 'Support', 'Pricing', and a blue 'Sign up' button. Below this is a title 'Pricing'.

The main section is titled 'Pricing' with a subtitle: 'Quickly build an effective pricing table for your potential customers with this Bootstrap example. It's built with default Bootstrap components and utilities with little customization.' The table has three columns: 'Free', 'Pro', and 'Enterprise'. Each column contains a price, features, and a call-to-action button.

Free	Pro	Enterprise
\$0 / mo	\$15 / mo	\$29 / mo
10 users included 2 GB of storage Email support Help center access	20 users included 10 GB of storage Priority email support Help center access	50 users included 15 GB of storage Phone and email support Help center access
Sign up for free	Get started	Contact us

At the bottom, there are footer sections for 'Features', 'Resources', and 'About' with various links like 'Cool stuff', 'Random feature', 'Team', 'Locations', etc.

Slika 11.2 Model za izradu domaćeg zadatka 1 [izvor: autor]

▼ Poglavlje 12

Zaključak

ZAKLJUČAK

Lekcija se bavila nadogradnjom diskusije o HTML - stilizacijom.

Lekcija se bavila nadogradnjom diskusije o HTML - stilizacijom. Poseban akcenat je stavljen na izučavanje:

1. Stilskog jezika CSS sa istorijskim osvrtom na njegov razvoj;
2. Elemenata stilskog jezika CSS;
3. Primene stilskog jezika CSS kroz adekvatne pokazne i individualne zadate;
4. Radnog okvira za primenu stilova Bootstrap;
5. Primene radnog okvira za primenu stilova Bootstrap kroz adekvatne pokazne i individualne zadate.

U nastavku predmet će se baviti izučavanjem skripting jezika JavaScript kao preduslova za razvoj frontend aplikacija primenom radnog okvira Angular.

LITERATURA

Za pripremu lekcije 2 korišćena je aktuelna štampana i veb literatura.

Obavezna literatura:

1. Jennifer Niederst Robbins, Learning Web Design - Fifth Edition, by , Copyright © 2018 O'Reilly Media, Inc.
2. Roxane Anquetil, Fundamental Concepts for Web Development: HTML5, CSS3, JavaScript and much more! For complete beginners!, 2019,

Dopunska literatura:

1. Daniel Bell, HTML & CSS: A Step-by-Step Guide for Beginners2, Guzzler Media, 2019.

Veb lokacije:

1. <http://taligarsiel.com/Projects/howbrowserswork1.htm>
2. <http://www.w3schools.com/>
3. <https://www.packtpub.com/>
4. <https://sh.wikipedia.org/wiki/CSS>
5. <https://www.w3.org/TR/CSS1/>

6. <https://www.w3.org/TR/CSS2/>
7. <https://www.w3.org/TR/CSS/#css>
8. https://www.w3schools.com/xml/xsl_languages.asp
9. <https://www.mojwebdizajn.net/>
10. <https://websitesetup.org/bootstrap-tutorial-for-beginners/>



IT255 - VEB SISTEMI 1

Osnove JavaScript ECMAScript v6

Lekcija 03

PRIRUČNIK ZA STUDENTE

IT255 - VEB SISTEMI 1

Lekcija 03

OSNOVE JAVASCRIPT ECMASCRIPT V6

- ▼ Osnove JavaScript ECMAScript v6
- ▼ Poglavlje 1: JavaScript standardizacija
- ▼ Poglavlje 2: JavaScript i veb pregledači
- ▼ Poglavlje 3: JavaScript i DOM
- ▼ Poglavlje 4: Elementi osnovne sintakse JavaScript jezika
- ▼ Poglavlje 5: JavaScript funkcije
- ▼ Poglavlje 6: HTML i JavaScript - Pokazna vežba 3
- ▼ Poglavlje 7: Individualne vežbe 3
- ▼ Poglavlje 8: Domaći zadatak 3
- ▼ Zaključak

Copyright © 2017 – UNIVERZITET METROPOLITAN, Beograd. Sva prava zadržana. Bez prethodne pismene dozvole od strane Univerziteta METROPOLITAN zabranjena je reprodukcija, transfer, distribucija ili memorisanje nekog dela ili čitavih sadržaja ovog dokumenta., kopiranjem, snimanjem, elektronskim putem, skeniranjem ili na bilo koji drugi način.

Copyright © 2017 BELGRADE METROPOLITAN UNIVERSITY. All rights reserved. No part of this publication may be reproduced, stored in a retrieval system or transmitted in any form or by any means, electronic, mechanical, photocopying, recording, scanning or otherwise, without the prior written permission of Belgrade Metropolitan University.

▼ Uvod

UVOD

JavaScript je verovatno najpopularniji skripting jezik.

Kada se pogledaju oglasi na IT tržištu moguće je sagledati veliku potrebu za dobro obučenim i sposobljenim stručnjacima koji barataju tehnologijama i alatima poput: *HTML*, *CSS* i skripting jezicima. *JavaScript* je verovatno najpopularniji skripting jezik.

JavaScript je jednostavan, interpretirani skripting jezik koji ima veliku primenu u razvoju interaktivnih HTML stranica. Pored HTML-a i CSS-a, JavaScript predstavlja osnovu koda kojeg razumeju savremeni veb pregledači. Upravo iz navedenog razloga JavaScript je standardizovan i integriran u sve savremene veb pregledače:

- *Internet Explorer;*
- *Google Chrome;*
- *Mozilla Firefox;*
- *Opera;*
- *Safari*, i drugi.

Primena JavaScript jezika, u klijentima veb aplikacija, je veoma značajna i podrazumeva sledeće:

- implementaciju elemenata za interakciju sa korisnikom;
- promenu osobina prozora veb pregledača;
- dodavanje dinamičkog sadržaja u HTML stranicu.

Postoji zabluda da je *JavaScript* pojednostavljena verzija programskog jezika Java. **Ovo apsolutno nije tačno.** Izuzev par sličnih sintaksnih elemenata i slučajeva upotrebe u okviru veb pregledača, ova dva jezika su potpuno različita i po nameni i po sintaksi. JavaScript je skripting jezik (**ne prevodi se**) koji ima primenu u razvoju lakih klijenata - programa koji funkcionišu u veb pregledačima, dok je **Java pravi programski jezik** sa moćnim funkcionalnostima za razvoj savremenih softverskih komponenata na serverskoj strani.

Naziv *JavaScript* ima uglavnom komercijalne korene sa željom da se ubrza usvajanje i primena novog skripting jezika, koji je originalno trebalo da se naziva *LiveScript*. Od tog perioda *JavaScript* kao jezik je značajno unapređen.

UVODNI VIDEO

Trajanje video snimka: 5min 58sek

Ova lekcija sadrži video materijal. Ukoliko želite da pogledate ovaj video morate da otvorite LAMS lekciju.

▼ Poglavlje 1

JavaScript standardizacija

KRATAK OSVRT NA ISTORIJAT JS JEZIKA

Od 1995, JavaScript je prošao kroz brojna unapređenja.

Prve verzije **JavaSrcipt** (kraće JS) jezika datiraju još iz 1995. godine kada je kompanija **NetScape** predstavila prvih nekoliko specifikacija ovog jezika. Prepoznajući potencijalni značaj novog jezika, kompanija **Microsoft**, nedugo zatim, izbacuje svoju verziju **JScript** jezika, koja je bila veoma slična **JavaScript** jeziku. Od tog perioda, do danas JavaScript je prošao kroz veliki broj razvojnih iteracija i stekao veliku popularnost. Zahvaljujući toj popularnosti **JavaSrcipt** jezik je integrisan u sve savremene veb pregledače. Upravo iz navedenog razloga, ovaj jezik je morao da prođe kroz standardizaciju.

Danas je za standardizaciju skriptnih jezika, pa tako i JavaScript - a, zadužena organizacija pod nazivom **ECMA** (<https://www.ecma-international.org/>). Otkako je jezik standardizovan, prošao je kroz 6 glavnih iteracija i aktuelni standard predstavlja JavaScript ECMA v6, koji je i predmet izučavanja ove lekcije.

U savremenoj literaturi pojam JavaScript označava bilo koju implementaciju jezika, uključujući i Microsoft-ov JScript.

Sledećim video materijalom je prezentovan istorijski razvoj JavaScript jezika.

Ova lekcija sadrži video materijal. Ukoliko želite da pogledate ovaj video morate da otvorite LAMS lekciju.

PREGLED JS VERZIJA

JavaScript je standardizovan skripting jezik.

U prethodnom izlaganju je istaknuto da je JS jezik standardizovan i da je prošao je kroz 6 glavnih iteracija, te da aktuelni standard predstavlja **JavaScript ECMA v6**, koji je i predmet izučavanja ove lekcije. Međutim, put do trenutno aktuelnog standarda je bio dugačak i kretao se konstantnom uzlaznom putanjom po sledećim koracima:

- **JavaScript 1.0** - Prva verzija sa velikim brojem grešaka. Ova verzija je bila implementirana u veb pregledač pod nazivom **NetScape 2**;
- **JavaScript 1.1** - Sadrži značajne ispravke sa dodatom funkcionalnošću za rad sa **Array** objektima. Ova verzija je bila implementirana u veb pregledač pod nazivom **NetScape 3**;

- *JavaScript 1.2* - Dalji rad na otklanjanju grešaka. Prvi put je implementiran operator *switch* u jezik. Ovo je prva verzija koja poštuje *ECMA v1* standard uz određene nekompatibilnosti. Ova verzija je bila implementirana u veb pregledač pod nazivom *NetScape 4*;
- *JavaScript 1.3* - Ispravljene su nekompatibilnosti prethodne verzije i obavljeno je dalje usklađivanje sa standardom *ECMA v1*. Ova verzija je bila implementirana u veb pregledač pod nazivom *NetScape 4.5*;
- *JavaScript 1.4* - Dalja unapređenja vezana za podršku *NetScape* pregledaču;
- *JavaScript 1.5* - Uvodi rukovanje izuzecima. Poštuje standard *ECMA v3* i ugrađen je u pregledače *NetScape 6* i *Mozilla Firefox*.

Sledi pregled verzija jezika *JScript* kompanije *Microsoft*:

- *JScript 1.0* - Veoma sličan standardnoj verziji *JavaScript 1.0* i integrisan je u prve verzije pregledača Internet *Explorer 3*;
- *JScript 2.0* - Veoma sličan standardnoj verziji *JavaScript 1.1* i integrisan je u naredne verzije pregledača *Internet Explorer 3*;
- *JScript 3.0* - Veoma sličan standardnoj verziji *JavaScript 1.3*. Poštuje standard *ECMA v1* i integrisan je u prve verzije pregledača *Internet Explorer 4*;
- *JScript 4.0* - Test verzija. Nije implementiran ni u jedan veb pregledač;
- *JScript 5.0* - Verzija koja parcijalno poštuje standard *ECMA v3*. Implementiran je u veb pregledač *Internet Explorer 5*;
- *JScript 5.5* - Ekvivalentan standardnoj verziji *JavaScript 1.5*. Uvodi potpuno poštovanje standarda *ECMA v3*. Implementiran je u veb pregledač *Internet Explorer 5.5* i *6*;

Dalje verzije *JavaScript* jezika odnose se na *ECMA* standarde v4, v5 i v6 o čemu će biti govora u narednom izlaganju.

PREGLED JS STANDARDA

JS jezik standardizovan jezik koji je prošao je kroz 6 glavnih iteracija standardizacije.

Konačno, nakon isticanja različitih verzija JS jezika, sledi i navođenje standarda na osnovu kojih je ovaj jezik i razvijan:

- *ECMA v1* - Prva verzija JS standarda. Standardizuje osnove jezika *JavaScript 1.1* i predviđa uvođenje dodatnih funkcionalnosti. Implementacije jezika koje su u potpunosti izgrađene na osnovu ovog standarda su *JavaScript 1.3* i *JScript 3.0*;
- *ECMA v2* - Standard se nije fokusirao na nove funkcionalnosti već na otklanjanje dvosmislenosti uočenih u prethodnom standardu;
- *ECMA v3* - Standardizuje primenu naredbe *switch*, rukovanje izuzecima i primenu regularnih izraza (*regex* - *regular expression*). Implementacije jezika koje su u potpunosti izgrađene na osnovu ovog standarda su *JavaScript 1.5* i *JScript 5.0*;
- *ECMA v4* - Standard nikada zvanično nije primjenjen usled nesporazuma radnih grupa koje su radile na njegovom razvoju i implementaciji;
- *ECMA v5* - Standard uvodi primenu *JSON* (*JavaScript Object Notation*) notacije i uklanja nedostatke iz *ECMA v3* standarda;

- *ECMA v5.1* - Unapređenje prethodnog standarda - nema noviteta;
- *ECMA v6* - Uveden 2015 i unapređivan do današnjeg dana. Iako postoje i v7 - v10 njih možemo posmatrati kao verzijama ovog standarda. **Predstavlja osnov za Angular / TypeScript jezik.**

▼ Poglavlje 2

JavaScript i veb pregledači

JS INTERPRETATOR

JS interpretator je integriran u sve savremene veb pregledače.

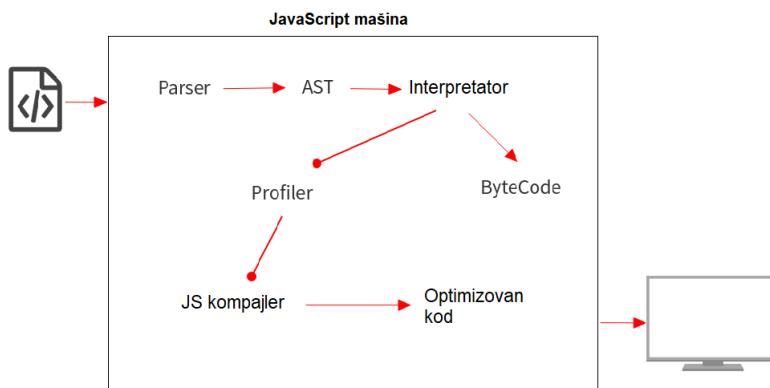
JavaScript interpretator je integriran u sve savremene veb pregledače, za nas su značajne njihove poslednje verzije:

- *Internet Explorer;*
- *MS Edge;*
- *Mozilla Firefox;*
- *Google Chrome;*
- *Safari;*
- *Opera;*
- i drugi.

Budući da je *JavaScript* podrška obavezna za sve savremene veb pregledače, proizvođači pregledača se više ne takmiče na polju *JavaScript* integracije već po pitanju brzine kojom se određeni algoritmi izvršavaju u pregledačima. Takođe, proizvođači veb pregledača se takođe nadmeću u podršci za lakši razvoj veb aplikacija koje na klijent strani sadrže JavaScript kod. **Otuda savremeni veb pregledači (Google Chrome, Mozilla Firefox i Internet Explorer - koji su najpopularniji) poseduju razvojne alate (development tools)** do kojih se lako dolazi klikom na taster F12 na tastaturi.

Zbog svoje jednostavnosti *JavaScript* je stekao veliku popularnost i podršku i otuda, kao što je navedeno, svi savremeni pregledači imaju podršku za interpretaciju JavaScript koda. Interpretator je tip prevodioca koji čita linije koda, jednu po jednu i prevodi ih u mašinski jezik i odmah izvršava. Ovde nemamo kao u programskim jezicima prevedenu verziju koda (Java bytecode ili exe u C++ / C# kao posebne datoteke koje se kreiraju i čuvaju na disku kada se program prevede primenom prevodioca pod nazivom kompjajler).

Jezici koji služe za kreiranje koda koji se na opisani način interpretira, nazivaju se zajedničkim imenom skripting jezici. *JavaScript* kod se parsira, prevodi (interpretira) liniju po liniju u kod razumljiv računaru. Ovaj kod preuzima JS kompjajler koji ga optimizuje, nakon čega se izvršava.



Slika 2.1 Interpretacija JS koda

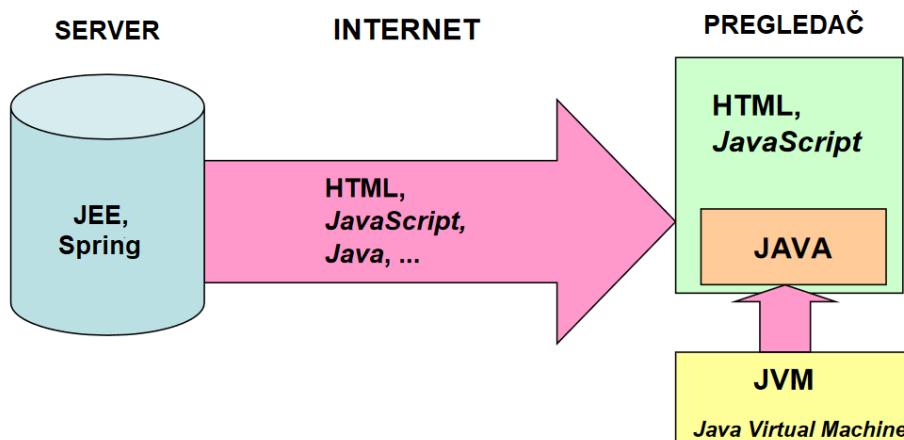
SCENARIO UPOTREBE JS KODA

JS kod je integriran u HTML stranicu sa kojom se zajedno izvršava u veb pregledaču.

JS kod je integriran u HTML stranicu sa kojom se zajedno izvršava u veb pregledaču. Ovde je posebno značajan sledeći scenario:

1. klijent sa HTML i JS kodom šalje zahtev serveru;
2. server (izgrađen na primer Java tehnologijama JEE ili Spring) obrađuje zahtev i vraća odgovor klijentu;
3. odgovor može da sadrži i dinamički sadržaj kojeg klijent sa HTML i JS kodom prihvata i prezentuje korisniku.

Sledećom slikom je moguće prikazati tok podataka od servera do klijenta.



Slika 2.2 Tok podataka od servera do klijenta

✓ Poglavlje 3

JavaScript i DOM

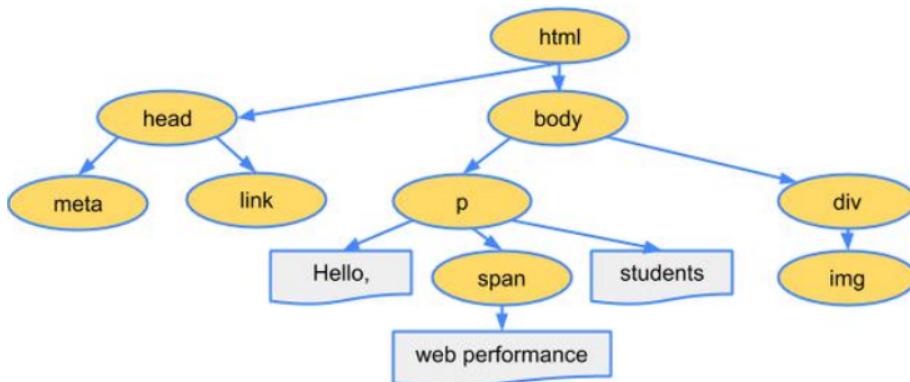
HTML DOM

DOM koristi veb pregledač za definisanje logičke strukture veb stranice i prikazivanje elemenata.

U ulozi veb programera često ćete imati potrebu da manipulišete **DOM** (Document Object Model) kojeg koristi veb pregledač za definisanje logičke strukture veb stranice i na osnovu nje poređa **HTML** elemente na stranici.

HTML definiše podrazumevanu DOM strukturu. U velikom broju slučajeva neophodno je programsko manipulisanje **DOM** strukturom primenom **JavaScript** jezika. Ovo se obično radi iz razloga što je potrebno dodati dopunske funkcionalnosti u stranicu.

HTML DOM model je izgrađen u formi stabla objekata (sledeća slika - izvor: <https://www.hongkiat.com/blog/dom-manipulation-javascript-methods/>).



Slika 3.1 HTML DOM

Svakom HTML objektu ili osobini pristupa se kroz model, tj. **document** je osnovni objekat preko kojeg se pristupa svim drugim objektima HTML dokumenta.

Upravo u sledećem izlaganju će biti govora kako se koristi **JavaScript** jezik za manipulisanje **DOM** strukturom neke **HTML** stranice.

Upotrebom objektnog modela, JavaScript dobija mogućnost za kreiranje dinamičkog HTML-a:

- JavaScript može da menja sve HTML elemente na stranici;
- JavaScript može da menja sve HTML atribute na stranici;
- JavaScript može da menja sve CSS stilove na stranici;
- JavaScript može da briše sve HTML elemente i atribute na stranici;

- JavaScript može da dodaje nove HTML elemente i atribute na stranici;
- JavaScript može da reaguje na sve događaje na stranici;
- JavaScript kreira nov događaj na stranici;

JAVASCRIPT - HTML DOM METODE

HTML DOM metode predstavljaju akcije koju je moguće obaviti nad HTML elementima.

U prvom koraku je bitno da se razvoje:

- *HTML DOM metode* - predstavljaju akcije koje je moguće obaviti nad HTML elementima;
- *HTML DOM osobine* - vrednost HTML elementa koji je moguće dodeliti ili promeniti.

JavaScript jezik veoma jednostavno pristupa i manipuliše DOM elementima. U DOM-u, svi elementi su definisani kao objekti. Pod pojmom DOM interfejsa podrazumevamo sve osobine i metode koje pripadaju skupu navedenih objekata. Upravo metode koriste osobine za obavljanje određenih akcija (na primer, dodavanje ili brisanje HTML elementa).

Posmatrajmo sledeći listing:

```
<html>
<body>

<p id="univerzitet"></p>
<p id="adresa"></p>

<script>
    document.getElementById("univerzitet").innerHTML = "Univerzitet Metropolitan
Beograd";
    document.getElementById("adresa").innerHTML = "Tadeuša Košćuška 63";
</script>

</body>
</html>
```

Listingom je dat primer promene sadržaja (*innerHTML*) elementa *<p>* korišćenjem osobine *id* za navedeni element. U DOM kontekstu *getElementById()* je metoda, dok *innerHTML* predstavlja osobinu.

Navedeno je verovatno najlakši način za pristupanje DOM elementima. Svaki element je jednostavno pronaći na osnovu njegovog identifikatora, a *innerHTML* je osobina koju je moguće upotrebiti za izmenu bilo kojeg HTML elementa, računajući i *<html>* i *<body>*.

Sledećom slikom je prikazan rezultat učitavanja stranice sa priloženim listingom u veb pregledač:

Univerzitet Metropolitan Beograd

Tadeuša Košćuška 63

Slika 3.2 Primena getElementById() metode i osobine innerHTML

OBJEKAT DOCUMENT

Objekat HTML DOM document je vlasnik svih ostalih objekata na vašoj HTML stranici.

Objekat HTML DOM **document** je vlasnik svih ostalih objekata na vašoj HTML stranici. Ovaj objekta, zapravo, reprezentuje veb stranicu. Ukoliko programer želi da pristupi bilo kojem HTML elementu, moraće prvo da se obrati **document** objektu, kao što je bio slučaj sa elementom **<p>** u prethodnom primeru.

Sledećom diskusijom je istaknuto kako je moguće upotrebiti **document** objekat za pristup i manipulaciju HTML elementima.

Pronalaženje HTML elemenata:

1. **document.getElementById(id)** - pronalaženje HTML elementa na osnovu njegovog identifikatora;
2. **document.getElementsByTagName(name)** - pronalaženje HTML elementa na osnovu naziva taga;
3. **document.getElementsByClassName(name)** - pronalaženje HTML elementa na osnovu naziva klase.

Izmena HTML elemenata:

1. **element.innerHTML = new html content** - menja unutrašnji sadržaj HTML elementa;
2. **element.setAttribute(attribute, value)** - menja vrednost atributa HTML elementa;
3. **element.style.property = new style** - menja stil HTML elementa;
4. **element.removeAttribute(attribute)** - menja vrednost atributa HTML elementa.

Dodavanje i brisanje elemenata:

1. **document.createElement(element)** - Kreira novi HTML element;
2. **document.removeChild(element)** - Briše postojeći HTML element;
3. **document.appendChild(element)** - Dodaje HTML element;
4. **document.replaceChild(new, old)** - Menja stari HTML element novim;
5. **document.write(text)** - Piše tekst u HTML odlazni tok.

DODAVANJE OSLUŠKIVAČA DOGAĐAJA I PRONALAŽENJE HTML OBJEKATA

Dodavanje osluškivača događaja i pronalaženje HTML objekata se jednostavno realizuje preko objekta document.

Dodavanje osluškivača događaja i pronalaženje HTML objekata se jednostavno realizuje preko objekta document.

Na sledeći način je prezentovano opšte obraćanje `document` objektu kada je neophodno obaviti registrovanje osluškivača događaja:

```
document.getElementById(id).onclick = function(){kod}
```

Ovde je prikazano kako se jednostavno klikom na HTML element, čiji je identifikator argument metode `getElementById()`, izvršava funkcija sa odgovarajućim kodom.

Po [HTML5](#) standardu postoji veliki broj objekata, osobina i kolekcija kojima je moguće manipulisati. Sledi lista najčešće korišćenih metoda za pronalaženje HTML elemenata, pri čemu su izostavljene one koje su po HTML5 standardu zastarele:

- 1. `document.anchors` - Vraća sve `<a>` elemente koji poseduju `name` atribut;
- 2. `document.baseURI` - Vraća absolutni osnovni URI dokumenta;
- 3. `document.body` - Vraća `<body>` element;
- 4. `document.cookie` - Vraća "kolačiće" dokumenta;
- 5. `document.doctype` - Vraća tip dokumenta (doctype);
- 6. `document.documentElement` - Vraća `<html>` element;
- 7. `document.documentElementURI` - Vraća URI dokumenta;
- 8. `document.domain` - Vraća domenski naziv servera dokumenta;
- 9. `document.forms` - Vraća sve `<form>` elemente;
- 10. `document.head` - Vraća `<head>` element;
- 11. `document.images` - Vraća sve `` elemente;
- 12. `document.links` - Vraća sve `<area>` i `<a>` elemente koji poseduju `href` atribut;
- 13. `document.scripts` - Vraća sve `<script>` elemente;
- 14. `document.title` - Vraća `<title>` element;
- 15. `document.URL` - Vraća kompletan URL dokumenta.

Postoji još metoda ali su izabrane one koje su aktuelne i češće se koriste.

PRIMER PRONALAŽENJA ELEMENATA

Primer pronalaženja HTML elementa na osnovu identifikatora i naziva taga.

Najlakši način za pronalaženje HTML elementa jeste na osnovu njegovog identifikatora ili naziva taga. Posmatramo sledeći listing:

```
<html>
<body>

<p id="univerzitet"></p>
<p id="adresa"></p>
<p id="info"></p>

<script>
    //pronalaženje elemenata po id
    document.getElementById("univerzitet").innerHTML = "Univerzitet Metropolitan
Beograd";
    document.getElementById("adresa").innerHTML = "Tadeuša Košćuška 63";

    //pronalaženje elementa po nazivu taga
    var x = document.getElementsByTagName("p");
    document.getElementById("info").innerHTML = "Sadržaj prvog paragrafa krije naziv
našeg Univerziteta : " + x[0].innerHTML;

</script>

</body>
</html>
```

Iz priloženog listinga je moguće primetiti kako se koristi JavaScript kod za pronalaženje elemenata na osnovu identifikatora (linije koda 10, 11 i 15) i na osnovu naziva taga (linija koda 14). U varijabli x čuva se niz koji odgovara vraćenim tagovima `<p>`. Izrazom `x[0].innerHTML` uzima se sadržaj prvog `<p>` taga (0 je indeks prvog elementa u nizu). Ova vrednost će biti pridružena trećem tagu `<p>` čiji je id "info" (linija koda 15).

Učitavanjem kreiranog HTML dokumenta u veb pregledač, dobija se izlaz kao na sledećoj slici:

```
Univerzitet Metropolitan Beograd
Tadeuša Košćuška 63
Sadržaj prvog paragrafa krije naziv našeg Univerziteta : Univerzitet Metropolitan Beograd
```

Slika 3.3 Pronalaženja HTML elementa na osnovu identifikatora i naziva taga.

PRIMER IZMENE HTML-A

Primer upisivanja u HTML tok i izmene stilova.

Sledi demonstracija upisivanja podataka u HTML tok primenom `document` objekta, kao i kako taj objekat može da bude upotrebljen za korekciju stilova unutar HTML dokumenta. Sledećim listingom je pokazano kako jezik JavaScript obavlja navedene zadatke:

```
<html>
<body>

<p id="univerzitet"></p>
<p id="adresa"></p>
<p id="info"></p>

<script>
    //pronalaženje elemenata po id
    document.getElementById("univerzitet").innerHTML = "Univerzitet Metropolitan
Beograd";
    document.getElementById("adresa").innerHTML = "Tadeuša Košćuška 63";
    //pronalaženje elemenata po nazivu taga
    var x = document.getElementsByTagName("p");
    document.getElementById("info").innerHTML = "Sadržaj prvog paragrafa krije naziv
našeg Univerziteta : " + x[0].innerHTML;

    //izmena stila
    document.getElementById("univerzitet").style.color = "blue";
    document.getElementById("adresa").style.fontFamily = "Arial";
    document.getElementById("info").style.fontSize = "larger";

    //upisivanje u tok
    document.write(Date());

</script>

</body>
</html>
```

Sledi analiza priloženog listinga. Linijama koda 16 - 19 je demonstrirano kako je nad paragrafima izvršena promena stilova, i to promena: boje teksta, familije fonta i veličine fonta, respektivno.

Linijom koda 22 je obavljeno upisivanje vrednosti trenutnog datuma u odlazi HTML tok.

Učitavanjem kreiranog HTML dokumenta u veb pregledač, dobija se izlaz kao na sledećoj slici:

Univerzitet Metropolitan Beograd
Tadeuša Košćuška 63
Sadržaj prvog paragrafa krije naziv našeg Univerziteta : Univerzitet Metropolitan Beograd
Mon May 18 2020 19:42:49 GMT+0200 (Средњевропско летње време)

Slika 3.4 Primer upisivanja u HTML tok i izmene stilova.

Nikada ne koristite `document.write()` nakon učitavanja dokumenta. Prebrisaće dokument!!!

REAGOVANJE NA DOGAĐAJE KROZ KLIK NA TEKST

Primer pisanja JavaScript koda koji reaguje na događaje.

Sledi demonstracija reagovanja na događaje u HTML dokumentu. Sledećim listingom je pokazano kako jezik *JavaScript* obavlja navedene zadatke:

```
<html>
<body>

<p id="univerzitet"></p>
<p id="adresa"></p>
<p id="info"></p>
<p id="marketing" onclick="changeText(this)">Klikni na tekst!!!</p>

<script>
    //pronalaženje elemenata po id
    document.getElementById("univerzitet").innerHTML = "Univerzitet Metropolitan
Beograd";
    document.getElementById("adresa").innerHTML = "Tadeuša Košćuška 63";
    //pronalaženje elemenata po nazivu taga
    var x = document.getElementsByTagName("p");
    document.getElementById("info").innerHTML = "Sadržaj prvog paragrafa krije naziv
našeg Univerziteta : " + x[0].innerHTML;

    //izmena stila
    document.getElementById("univerzitet").style.color = "blue";
    document.getElementById("adresa").style.fontFamily = "Arial";
    document.getElementById("info").style.fontSize = "larger";

    //upisivanje u tok
    document.write(Date());

    //obrada događaja
    function changeText(id) {
        id.innerHTML = "Broj 1 po zadovoljstvu studenata!!!";
    }

</script>

</body>
</html>
```

Na paragraf, u liniji koda 7, postavljen je osluškivač koji čeka događaj tipa *onClick*. Kada korisnik obavi klik na tekst prikazan ovim paragrafom poziva se metoda *changeText()* koja obrađuje događaj tako što će u paragrafu zameniti izvorni tekst novim.

Sledećom slikom je prikazan izgled HTML stranice pre javljanja događaja.

Univerzitet Metropolitan Beograd
Tadeuša Košćuška 63
Sadržaj prvog paragrafa krije naziv našeg Univerziteta : Univerzitet Metropolitan Beograd
Broj 1 po zadovoljstvu studenata!!!
Mon May 18 2020 20:00:34 GMT+0200 (Средњеевропско летње време)

Slika 3.5 Izgled HTML stranice pre javljanja događaja.

Sledećom slikom je prikazan izgled stranice nakon obrađenog događaja.

Univerzitet Metropolitan Beograd
Tadeuša Košćuška 63
Sadržaj prvog paragrafa krije naziv našeg Univerziteta : Univerzitet Metropolitan Beograd
Broj 1 po zadovoljstvu studenata!!!
Mon May 18 2020 20:00:34 GMT+0200 (Средњеевропско летње време)

Slika 3.6 Izgled stranice nakon obrađenog događaja

REAGOVANJE NA DOGAĐAJE KROZ KLIK NA DUGME

Kreiranje dugmeta i pisanje JS koda koji obrađuje click događaj.

Kreiranje dugmeta i pisanje JS koda koji obrađuje click događaj je sledeća funkcionalnost koja će biti istražena. U fokusu je sledeći listing:

```
<body>

<p id="univerzitet"></p>
<p id="adresa"></p>
<p id="info"></p>
<p id="marketing" onclick="changeText(this)">Klikni na tekst!!!</p>
<button id="myBtn">Promeni pozadinu</button> <br/>

<script>
    //pronalaženje elemenata po id
    document.getElementById("univerzitet").innerHTML = "Univerzitet Metropolitan
Beograd";
    document.getElementById("adresa").innerHTML = "Tadeuša Košćuška 63";
    //pronalaženje elemenata po nazivu taga
    var x = document.getElementsByTagName("p");
    document.getElementById("info").innerHTML = "Sadržaj prvog paragrafa krije naziv
našeg Univerziteta : " + x[0].innerHTML;

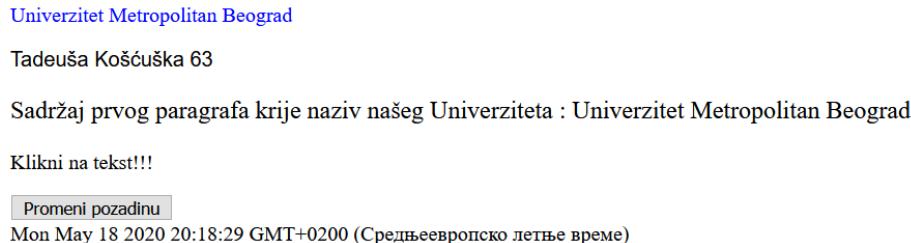
    //izmena stila
    document.getElementById("univerzitet").style.color = "blue";
    document.getElementById("adresa").style.fontFamily = "Arial";
    document.getElementById("info").style.fontSize = "larger";

    //upisivanje u tok
```

```
document.write(Date());  
  
//obrada događaja  
function changeText(id) {  
    id.innerHTML = "Broj 1 po zadovoljstvu studenata!!!";  
}  
  
document.getElementById("myBtn").addEventListener("click", function() {  
    document.body.style.background = "#aaf3f3";  
});  
  
</script>  
  
</body>  
</html>
```

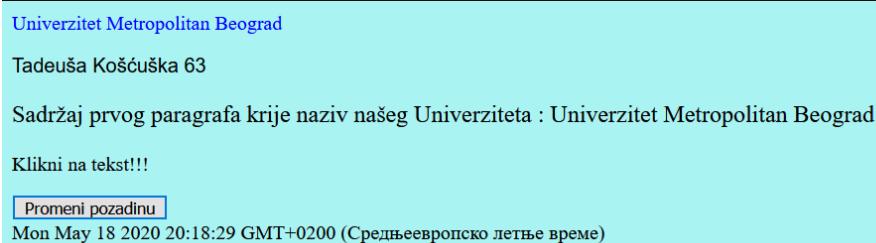
Ako se detaljno pogleda priloženi listing primećuje se kontrola dugme sa identifikatorom "myBtn" (linija koda 7). U liniji koda 30, na ovaj HTML element je postavljen osluškivač. Klikom na dugme izvršava se linija koda 31 u kojoj objekat `document` vraća element `<body>` i na njega primenjuje nov stil tako što mu menja boju pozadine.

Sledećom slikom je prikazan izgled stranice pre klika na dugme.



Slika 3.7 Izgled stranice pre klika na dugme

Sledećom slikom je prikazan izgled stranice nakon klika na dugme.



Slika 3.8 Izgled stranice nakon klika na dugme

JAVASCRIPT I DOM (VIDEO MATERIJAL)

JavaScript i DOM diskusiju je moguće zaokružiti odgovarajućim video materijalom.

Ova lekcija sadrži video materijal. Ukoliko želite da pogledate ovaj video morate da otvorite LAMS lekciju.

Primeri koji su pratili izlaganje ovog objekta učenja, dostupni su za preuzimanje u sekciji **Shared Resources** odmah po završetku ovog objekta učenja. Preuzmite primere, proanalizirajte ih i pokušajte sami da ugradite vaše vlastite modifikacije u skladu sa naučenim gradivom.

▼ Poglavlje 4

Elementi osnovne sintakse JavaScript jezika

JAVASCRIPT – VARIJABLE

Nije neophodno deklarisati promenljivu pre prve dodele vrednosti (automatski će se izvršiti deklarisanje), ali je i predeklarisanje je dozvoljeno.

Nazivi promenljivih - Imena promenljivih mogu da sadrže brojeve i slova engleske abecede, ali prvi znak mora da bude slovo engleske abecede ili simbol “_” , pored toga:

- Ne mogu se koristiti prazna mesta u okviru imena.
- Ne mogu se koristiti rezervisane reči kao imena promenljivih.

Promenljiva ili varijabla – opisno rečeno je memoriska lokacija, “ostava” za određenu informaciju / podatak ili vrednost.

Koristi se za deklarisanje promenljive (deklaracija je kreiranje promenljive, a definicija znači i inicijalizaciju - postavljanje početne vrednosti), čija sintaksa je:

`var imePromenljive;`

Opciono moguće je izvršiti i njenu inicijalizaciju.

```
var imePromenljive = vrednost;  
  
var ime_promenljive1 = vrednost1, ime_promenljive2 = vrednost2,  
...;  
  
var evro; //deklaracija promenljiva  
  
var dinar = 95; //definicija int promenljive
```

Nije neophodno deklarisati promenljivu pre prve dodele vrednosti (automatski će se izvršiti deklarisanje). Predeklarisanje je dozvoljeno. Više primera je na veb stranici:

https://www.w3schools.com/js/js_variables.asp

Deklaracija varijabli u JavaScriptu se vrši na sledeći način:

- `var x;`

Sledeće četiri sekvene imaju isti efekat:

- `var x; x=8;`
- `x=8; var x;`
- `var x=8;`
- `x=8;`

Još malo primera deklarisanja promenljivih:

- `var x=5; var ime="Ana";`

Na ovaj način su deklarisane dve promenljive, prva celobrojnog tipa sa vrednošću 5, a druga tipa string sa vrednošću "Ana".

NUMERIČI TIPOVI PODATAKA

Tip podataka definiše i vrste operacija koje se mogu izvršiti sa tom promenljivom.

Case sensitive

JavaScript je *case sensitive* jezik, što znači da se velika i mala slova razlikuju, pa je promenljiva Aaa različita promenljiva od promenljive AAA.

Takođe se ključne reči (for, if, else, class, int,...) ne mogu koristiti u imenu promenljivih.

Tipovi podataka

Informacija koja se sadrži u promenljivoj određuje tip varijable.

Postoje: celobrojni brojevi, racionalni brojevi, stringovi (niz karaktera), logički tip (true/false).

Tip podataka definiše i vrste operacija koje se mogu izvršiti sa tom promenljivom.

Celobrojni brojevi

Mogu se koristiti sa brojnom osnovom 10, sa osnovom 8 i osnovom 16.

Uobičajena je predstava pomoću osnove 10. Ovakvi brojevi imaju cifre od 0 - 9, s tim da početna cifra ne sme biti 0.

Brojevi prikazani u oktalnom brojnom sistemu sa osnovom 8 moraju počinjati sa cifrom 0, a ostale cifre su od 0 - 7.

Brojevi prikazani u heksadecimalnom brojnom sistemu sa osnovom 16 moraju počinjati sa 0x ili 0X, a ostale cifre su od 0 - 15, s tim da se cifre 10 - 15 prikazuju slovima A - F.

Racionalni brojevi

Mogu se prikazati na dva načina:

- pomoću decimalne tačke, na primer 3.14
- pomoću eksponencijalne prezentacije, na primer 314E-2 ili 314e-2

OSTALI TIPOVI PODATAKA

Velika količina obrade zahteva se u ranim fazama odvija u obliku (stil softverske arhitekture) proširive cevne obrade.

String

String predstavlja proizvoljan niz karaktera između navodnika ("neki tekst") ili između apostrofa ('neki tekst'). U stringovima se mogu koristiti i specijalni karakteri.

Specijalni karakteri

- `\b` = jedno mesto levo (backspace)
- `\f` = jedan red nadole (form feed)
- `\n` = početak novog reda (new line character)
- `\r` = return (carriage return)
- `\t` = tabulator (tab)

Konverzija u string - primer

```
<script>
x=2+4;
document.write(x); document.write("<br/>");
x="2"+4";
document.write(x); document.write("<br/>");
x=2+"4";
document.write(x); document.write("<br/>");
x="2"+4;
document.write(x); document.write("<br/>");
</script>
```

Rešenje nakon **izvršenja tog koda je:**

6
24
24
24

Zaključak:

Integer se uvek konvertuje u string pri spajanju sa stringom.

Logički tip

Logički tip podataka obuhvata dve vrednosti true (tačno) i false (netačno).

Prilikom rada ako je potrebno može se izvršiti konverzija logičke vrednosti true u broj 1 i vrednosti false u broj 0.

KONVERZIJA PODATAKA U JAVASCRIPT JEZIKU

JavaScript je jezik koji automatski izvršava promenu jednog tipa u drugi.

Konverzija podataka

JavaScript je jezik koji automatski izvršava promenu jednog tipa u drugi, jer se dozvoljava da promenljiva ima različite tipove podataka u različito vreme izvršavanja programa.

Primer promene jednog tipa u drugi u Javascript jeziku može da bude ilustrovan kroz sledećih nekoliko slučajeva:

- `a = 5;` //a je sada celobrojni podatak
- `b = 8;` //b je sada celobrojni podatak
- `b = "broj " + a;` //b je sada string podatak, zato što se na string "broj" nadovezuje ceo broj, pa se dobija string!

PRIMENA NULL I UNDEFINED

Nepostojeće vrednosti imaju značajnu primenu u mnogim programskim jezicima.

Nepostojeće vrednosti imaju značajnu primenu u mnogim programskim jezicima. *JavaScript* podržava primenu dva tipa nepostojećih vrednosti:

- `null` i
- `undefined`.

Tradicionalno, primena ovakvih koncepta u programiranju kod početnika često izaziva zabune. Iz navedenih razloga, potrebno je posebno razjasniti ove koncepte.

null (prazno)

Vrednost `null` se koristi za označavanje da je varijabla prazna. Obično se `null` inicijalno dodeli varijabli koja kasnije dobija pravu vrednost. Vrednost `null` je tipa objekat:

```
typeof null
"object"
```

undefined (nedefinisano)

Ako pokušamo da upotrebimo nepostojeću promenljivu, dobićemo grešku:

```
x
ReferenceError: x is not defined
```

Ali ako upotrebimo `typeof` operator na nepostojeću promenljivu, dobićemo " `undefined`" (string):

```
typeof x  
"undefined"
```

Nedefinisana vrednost se podrazumevano dodeljuje varijablama bez vrednosti. Ako deklarišemo promenljivu bez dodele vrednosti, *JavaScript* je inicijalizuje pomoću vrednosti *undefined* (bez navodnika):

```
let x  
x  
undefined
```

Kada ispitamo tip promenljive koja ima vrednost *undefined*, dobićemo string " *undefined*":

```
let x  
typeof x  
"undefined"
```

Takođe, ako ispitamo samu primitivnu vrednost *undefined* dobićemo isti rezultat:

```
typeof undefined  
"undefined"
```

Primer: Poređenje sa " *undefined* " možemo koristiti da proverimo je li promenljiva definisana:

```
if (typeof x !== "undefined") {  
    // uradi nesto  
}
```

Ovo jednako važi ako promenljiva *x* ne postoji, ili ako je deklarisana bez dodele vrednosti.

NUMERIČKI OPERATORI

Operatori su specijalni karakteri, koji definišu operaciju koja treba da se izvrši nad operandima, koji mogu biti promenljive, izrazi ili konstante.

Operatori

Operatori su specijalni karakteri, koji definišu operaciju koja treba da se izvrši nad operandima, koji mogu biti promenljive, izrazi ili konstante.

Razlikujemo sledeće tipove operatora:

- **Numerički operatori;**
- **Operatori sa bitovima;**
- **Logički operatori**
- **Relacioni operatori.**

Aritmetički operatori

Koriste se za matematičke operacije. Ukoliko je jedan od operanada tipa String za sve operatore, osim za sabiranje, pokušaće se da se izvede konverzija Stringa u broj i da se tako izvrši definisana operacija. Ako se ne uspe kao rezultat se dobija specijalna vrednost NaN (Not A Number).

Izuzetak kod sabiranja: podatak koji nije tipa String konvertuje se u String i izvršava se sabiranje dva Stringa. Primer je:

```
a=24; b = "broj " + a; //dobija se da je b: broj 24
```

Pregled aritmetičkih operatora je prikazan na Slici 1.

Operator	Opis	Operator	Opis
+	sabiranje	+=	sabiranje dodela
-	oduzimanje	-=	oduzimanje dodela
*	množenje	*=	množenje dodela
/	deljenje	/=	deljenje dodela
%	moduo	%=	moduo dodela
++	inkrement (x=x+1)	--	dekrement (x=x-1)

Slika 4.1 Aritmetički operatori - pregled

LOGIČKI OPERATORI

Logički operatori se primenjuju za građenje izraza koji imaju vrednosti tačno ili netačno.

Logički operatori se primenjuju za građenje izraza koji imaju vrednosti:

- `true` ili
- `false`

Ovi operatori imaju veliku primenu u okviru kontrolama toka. Primer upotrebe navedenih operatora je:

```

a = true;
b = false;
c = a || b;
d = a && b;
f = (!a && b) || (a && !b);
g = !a;
document.write( " a = " + a + "<BR>" );
document.write ( " b = " + b + "<BR> " );
document.write ( " c = " + c + "<BR> " );
document.write ( " d = " + d + "<BR> " );
document.write ( " f = " + f + "<BR> " );
document.write ( " g = " + g );

```

Pregled logičkih operatora dat je sledećom slikom zajedno sa odgovarajućim opisom.

Operator		Opis
I (&&)	izraz1 && izraz2	Rezultat je TRUE, jedino ako su oba izraza TRUE, u ostalim slučajevima FALSE.
ILI ()	izraz1 izraz2	Rezultat je TRUE, ako je bar jedan izraz TRUE, ako su oba FALSE, rezultat je FALSE.
NE (!)	! izraz	Rezultat daje komplement: ako je izraz TRUE rezultat je FALSE i obrnuto.

Slika 4.2 Logički operatori - pregled

OPERATORI NAD BITOVIMA

Operatori na nivou bita obavljaju operacije nad celobrojnim brojevima, i to dužine 32 bita.

Operatori iz ove grupe obavljaju operacije nad celobrojnim brojevima, i to dužine 32 bita. Ukoliko neki od operanada nije celobrojni broj dužine 32 bita, pokušaće se izvršiti konverzija u traženi tip, pa tek onda primeniti operaciju.

Primeri su:

- **13 & 8** daje 8 ($1101 \& 1000 = 1000$)
- **13 | 8** daje 13 ($1101 | 1000 = 1101$)
- **13 ^ 8** daje 5 ($1101 ^ 1000 = 0101$)

Detaljan pregled operatora nad bitovima je priložen sledećom slikom, zajedno sa odgovarajućim opisom svakog operatora.

Operator		Opis
Logičko I (and)	a & b	Rezultat je 1, samo ako su oba bita 1.
Logičko ILI (or)	a b	Rezultat je 0, samo ako su oba bita 0.
Logičko eksluzivno ILI (xor)	a ^ b	Rezultat je 1, samo ako je jedan bit 1, a drugi 0.
Logičko NE (not)	~ a	Komplementira bit 0->1, 1->0.
Pomeranje ulevo	a << b	Pomera binarni sadržaj operanda a za b mesta ulevo. Prazna mesta popunjava nulama.
Pomeranje udesno sa znakom	a >> b	Pomera binarni sadržaj operanda a za b mesta udesno. Prazna mesta popunjava vrednošću najstarijeg bita.
Pomeranje udesno sa nulama	a >>> b	Pomera binarni sadržaj operanda a za b mesta udesno. Prazna mesta popunjava sa vrednošću 0.

Slika 4.3 Tabela operatori nad bitovima

OPERATORI POREĐENJA

Operatori poređenja obavljaju poređenje dve vrednosti i kao rezultat vraćaju vrednost logičkog tipa true ili false.

Operatori poređenja

Obavljaju poređenje dve vrednosti i kao rezultat vraćaju vrednost logičkog tipa true ili false. Svaki dozvoljeni tip podataka, celobrojan, racionalni, karakter, String i logički tip može se upoređivati koristeći operatore == i !=. Samo numerički tipovi koriste ostale operatore.

Tabela operatora za poređenje je prikazana na Slici 4.

Operator	Upotreba	Opis
Jednakost	x == y	Rezultat je TRUE, ako su operandi x i y jednaki
Nejednakost	x != y	Rezultat je TRUE, ako su operandi x i y različiti
Veće	x > y	Rezultat je TRUE, ako je x veće od y
Veće ili jednako	x >= y	Rezultat je TRUE, ako je x veće ili jednako y
Manje	x < y	Rezultat je TRUE, ako je x manje od y
Manje ili jednako	x <= y	Rezultat je TRUE, ako je x manje ili jednako y
Jednakost (bez konverzije tipova)	x === y	Rezultat je TRUE, ako su x i y jednaki, ali bez konverzije tipova (moraju biti istog tipa!)
Različito (bez konverzije tipova)	x !== y	Rezultat je TRUE, ako su x i y različiti, ali bez konverzije tipova

Slika 4.4 Tabela operatora za poređenje

Razlika između == i ===

Operatori == i != obavljaju potrebnu konverziju podataka pre poređenja, ukoliko su operandi različitog tipa.

Znači za ove operatore vrednosti 5 (integer) i "5" (string) su iste, pa će posle njihovog poređenja rezultat sa operatorom == biti *TRUE*, a sa operatorom != *FALSE*.

S druge strane operatori === i !== ne obavljaju potrebnu konverziju podataka pre poređenja, ukoliko su operandi različitog tipa. Znači za ove operatore vrednosti 5 (ceo broj) i "5" (string) su različite, pa će posle njihovog poređenja rezultat sa operatorom === biti *FALSE*, a sa operatorom !== *TRUE*.

Primer koji to ilustruje priložen je sledećim listingom:

```
a = 4;  
b = 1;  
c = a < b;  
d = a == b;  
  
document.write( " c = " + c + "<BR>" );  
document.write ( " d = " + d );
```

Rezultat izvršavanja prethodnog primera je:

```
c = false  
d = false
```

KONSTRUKCIJE NAREDBI ZA KONTROLU TOKA IZVRŠAVANJA PROGRAMA

Konstrukcije naredbi za kontrolu toka izvršavanja programa u JavaScript jeziku su slične kao i u ostalim programskim jezicima.

Naredba if - else

Ova konstrukcija omogućava izvršenje određenog bloka instrukcija ako je uslov konstrukcije ispunjen. Opšti oblik konstrukcije je:

```
if (boolean_izraz) blok1;
```

```
[else blok2;]
```

svaki od blokova, bilo u if ili u else delu može biti nova if-else konstrukcija. Primer upotrebe ove konstrukcije je:

```
if (x == 8) {  
y=x;  
} else {  
z=x;  
y=y*x  
}
```

Naredba if - then - else kao uslovni operator

Forma ovog operatora je:

```
expression ? statement1 : statement2
```

gde je izraz expression bilo koji izraz čiji rezultat je vrednost logičkog tipa (na primer: a>b).

Ako je rezultat izraza true, onda se izvršava statement1, u suprotnom statement2. Primer:

```
(x%2==0) ? document.write("paran broj") : document.write("neparan broj");
```

Izgled složene if - else konstrukcije:

```
if (mesec == 1)  
ime_meseca = "Januar"  
else if (mesec == 2)  
ime_meseca = "Februar"  
else if (mesec == 3)  
ime_meseca = "Mart"  
else if (mesec == 4)  
ime_meseca = "Maj"  
else
```

```
....  
else if (mesec == 12)  
ime_meseca = "Decembar"
```

Naredba **switch** se koristi na sledeći način:

```
switch(mesec) {  
case 1: ime_meseca = "Januar"; break;  
case 3: ime_meseca = " Mart"; break;  
case 5: ime_meseca = "Maj"; break;  
case 7: ime_meseca = "Jul"; break;  
case 8: ime_meseca = "Avgust"; break;  
case 10: ime_meseca = "Oktobar"; break;  
case 12: ime_meseca = "Decembar"; break;  
case 4: ime_meseca = " April "; break;  
case 6: ime_meseca = "Jun"; break;  
case 9: ime_meseca = "Septembar"; break;  
case 11: ime_meseca = "Novembar"; break;  
case 2: ime_meseca = " Februar ";  
default: ime_meseca = " Nije naveden mesec";  
}
```

IZVRŠAVANJE WHILE PETLJE U JAVASCRIPT JEZIKU

U JavaScript jeziku petlja se koristi za deo programa koji se izvršava nijednom, jednom ili više puta.

Ukoliko se vrednost izraza mesec, u prethodnom prikeru, ne nalazi medju vrednostima case 1,..., N, tada se izvršava blok naredbi default;

Naredba while petlja

while petlja funkcioniše na taj način što se blok instrukcija unutar nje ponovljeno izvršava sve dok je uslov za ostanak u petlji, koji se nalazi na ulasku u petlju, ispunjen. Opšti oblik petlje izgleda ovako:

```
while(uslov_ostanka) {  
telo_petlje;  
}  
Jednostavan primer:  
i=1  
while(i<=10){  
document.writeln(i);  
i=i+1;  
}
```

Izvršavanje while petlje

Nakon izvršavanja ovog primera dobiće se prikazani brojevi od 1 do 10. Treba napomenuti da će se u slučaju da uslov petlje nije ispunjen kada se prvi put ispituje uslov petlje, telo petlje neće izvršiti nijednom.

Izvršavanje while petlje

Nakon izvršavanja ovog primera dobiće se prikazani brojevi od 1 do 10. Treba napomenuti da će se u slučaju da uslov petlje nije ispunjen kada se prvi put ispituje uslov petlje, telo petlje neće izvršiti nijednom. Dakle, ovo je petlja koja se izvršava nijednom, jednom ili više puta.

Izvršavanje do - while petlja

Za razliku od prethodne petlje koja je imala uslov na svom početku, do-while petlja ima uslov na kraju. Prema tome, telo petlje će se sigurno izvršiti bar jednom kao u sledećem primeru.

```
do {  
    telo_petlje  
    [iteracija]  
} while (uslov);  
i=1  
do {  
    document.writeln(i);  
    i++; //i=i+1  
} while(i<=10)
```

IZVRŠAVANJE FOR PETLJE; BREAK NAREDBA

Naredba return se koristi za povratak iz funkcije na mesto poziva.

Izvršavanje for petlje

Opšti oblik for petlje izgleda ovako:

```
for( inicijalizacija; uslov; iteracija){  
    telo_petlje;  
}  
for(i=0; i<10; i++){  
    document.writeln(i);  
}
```

Promenljiva i je privremena (lokalna) promenljiva, a blok u kome je definisana je blok naredbi u kome se nalazi *for* petlja.

NAREDBE BREAK I CONTINUE

Upravljanje iteracijama je omogućeno primenom naredbi break i continue

Naredba break

Naredba `break` se koristi za skok na kraj bloka koji je označen labelom uz `break` ili na kraj bloka u kome se `break` nalazi, ako `break` stoji bez labele. Labele, pomoću kojih se označavaju blokovi, se formiraju kao i svi ostali identifikatori s tim što iza njih mora stajati dvotačka (:). Na primer, sledeći kod:

```
a: {  
b: {  
c: {  
document.writeln("pre break-a"); //ovo se izvrsava!  
break b;  
document.writeln("ovo nece biti prikazano"); //ovo se ne izvrsava!  
}  
} // ovde izlazi iz bloka kada uradi break b!  
document.writeln("posle break-a"); //ovo se izvrsava!  
}
```

Naredba `continue`

Prelaz na sledeću iteraciju petlje, a da se deo koda pre njenog kraja ne izvrši. Za takve situacije se koristi `continue`.

```
for( i=0; i<10; i++) {  
  
document.write(i+ " ");  
  
if (i%2 == 0) continue; /*kada je broj paran  
preskace sve naredbe  
  
do kraja petlje */  
  
document.writeln(" "  
");  
}
```

Zahvaljujući `continue` naredbi nakon izvršavanja ovog primera dobija se:

```
0 1  
2 3  
4 5  
6 7  
8
```

NAREDBA RETURN

Naredba return se koristi za povratak iz funkcije na mesto poziva i sadrži povratnu vrednost.

Naredba return

Naredba `return` se koristi za povratak iz funkcije na mesto poziva. Ukoliko funkcija vraća neku vrednost tada `return` mora slediti izraz čiji je tip kompatibilan sa povratnim tipom funkcije. Primer korišćenja ove naredbe je prikazan sledećim listingom:

```
//kreiranje funkcije
function kvadratBroja( x ){
    return x * x;
}

x = kvadratBroja(5);

/* poziv funkcije */

document.write("Kvadrat od 5 je " + x);
```

Kao rezultat poziva funkcije dobija se:

Kvadrat od 5 je 25

ELEMENTI SINTAKSE JAVASCRIPT JEZIKA (VIDEO MATERIJALI)

Izučavanje osnovne sintakse JavaScript jezika može biti zaokruženo video materijalima.

Izučavanje osnovne sintakse JavaScript jezika može biti zaokruženo video materijalima.

Javascript Tutorial - 2 - Basic Syntax (trajanje: 5:00):

Ova lekcija sadrži video materijal. Ukoliko želite da pogledate ovaj video morate da otvorite LAMS lekciju.

Beginner JavaScript Tutorial - 3 - Variables (trajanje: 6:42)

Ova lekcija sadrži video materijal. Ukoliko želite da pogledate ovaj video morate da otvorite LAMS lekciju.

Beginner JavaScript Tutorial - 4 - Different Types of Variables (trajanje: 6:25):

Ova lekcija sadrži video materijal. Ukoliko želite da pogledate ovaj video morate da otvorite LAMS lekciju.

▼ Poglavlje 5

JavaScript funkcije

DRFINISANJE FUNKCIJE

Funkcija je konstrukcija u JavaScript - u pomoću koje je moguće grupisati veći broj naredbi.

Funkcija je konstrukcija u JavaScript - u pomoću koje je moguće grupisati veći broj naredbi koje se izvršavaju pomoću navođenja naziva funkcije. Na ovaj način je omogućeno značajno skraćivanje količine pisanog koda, kao i poboljšanje njegove preglednosti i naknadnog održavanja (npr. ceo program može da se sastoji od poziva funkcija proizvoljnog imena te tako pokazuje logiku programa, što bi inače bilo realizovano kroz veliki broj naredbi, uglavnom ponovljenih). Najčešći način definisanja funkcije jeste primena ključne reči function.

Sledećim listingom je dat opšti oblik funkcije u JavaScript jeziku:

```
function <naziv_funkcije>(<argument 1>, <argument2>, ...){  
    // naredbe (telo funkcije)  
}
```

Spisak argumenata funkcije nije obavezan ali zato male (okrugle) zagrade jesu.

Telo funkcije predstavlja kod koji je spakovanu unutar velikih (vitičastih) zagrada {...}. Primenom bloka naredbi omogućeno je da se nekoliko naredbi poveže u jednu celinu.

POVRATNE VREDNOSTI JAVASCRIPT FUNKCIJA

JavaScript funkcija može, ali i ne mora, da vrati neku vrednost.

Funkcija u JavaScript jeziku može da vrati vrednost naredbom return, ali i ne mora. Upravo prisustvo naredbe return, na kraju tela funkcije, je kriterijum po kojem se funkcije i razlikuju po tome da li:

- vraćaju vrednost;
- ne vraćaju vrednost.

Sledeća funkcija ne vraća nikakvu vrednost i za nju često kažemo da je i procedura:

```
// Funkcija koja ništa ne vraća  
function odgovor(sPoruka) {
```

```
    document.write(sPoruka, '<br/>');
}
```

Sledeći primer pokazuje *JavaScript* funkciju koja vraća vrednost. Za ovakve funkcije kažemo često da su "prave" funkcije:

```
// Funkcija koja vraća rastojanje dveju tačaka
function fUdaljenost(x1, y1, x2, y2) {
    var fDx,
    fDy,
    fRezultat;
    fDx = x2 - x1;
    fDy = y2 - y1;
    fRezultat = Math.sqrt(fDx * fDx + fDy * fDy);
    return fRezultat;
}
```

JavaScript podržava i rad sa rekurzivnim funkcijama. Sledi primer rekurzivne funkcije koja vraća vrednost.

```
// Rekurzivna funkcija (poziva samu sebe) koja računa faktorijel
// Za podsećanje:  $x! = x \cdot (x-1) \cdot (x-2) \cdots \cdot 3 \cdot 2 \cdot 1$ 

function faktorijel(x){
    if (x <= 1){
        return 1;
    }
    return x * faktorijel(x-1);
}
```

POZIV JAVASCRIPT FUNKCIJA

JavaScript funkcija se poziva tako da se navede njen naziv sa argumentima u okruglim zagradama.

JavaScript funkcija se poziva tako da se navede njen naziv, a njeni argumenti funkcije u okruglim zagradama. Ako funkcija nema argumenata, ne navodi se ništa, ali zagrada su obavezne. Ako se funkciji prosledi manje argumenata nego ih sadrži njena definicija, argumenti koji nisu navedeni će dobiti vrednost *undefined*. Na primer, funkcije definisane u prethodnom izlaganju, pozivaju se na sledeći način:

```
odgovor("Kako si, " + ime);

odgovor("Pozdrav svima!");

ukupno = fUdaljenost(0,0,2,1) + fUdaljenost(2,1,3,5);

odgovor("Verovatnoća je: " + faktorijel(39)/faktorijel(52));
```

OPSEG PROMENLJIVIH

U JavaScript-u vrednost je promenljive dostupna na dva načina: lokalno i globalno.

U JavaScript-u vrednost je **promenljive** dostupna na dva načina:

- samo unutar određene funkcije ili
- u celom programu.

Promenljive koje su dostupne u celom programu nazivamo *globalnim promenljivim*, dok promenljive koje su dostupne samo unutar neke funkcije nazivamo *lokalnim promenljivim* za tu funkciju.

Kada se u programu koriste promenljive, poželjno je da one uvek budu deklarisane primenom ključne reči **var**. Razmatraju se sledeći scenariji primene promenljivih u *JavaScript* programima:

- Ako je promenljiva deklarisana pomoću ključne reči **var** ili joj je samo dodeljena vrednost u glavnom programu (izvan svih funkcija), promenljiva je globalna;
- Ako je promenljiva deklarisana unutar određene funkcije pomoću ključne reči **var**, promenljiva je lokalna;
- Ako promenljiva nije deklarisana, nego joj je samo dodeljena neka vrednost unutar određene funkcije, promenljiva je globalna.

Sledećim primerom je ilustrovana primena lokalnih i globalnih promenljivih u *JavaScript* programima:

```
var iGlobalna1 = 10;
iGlobalna2 = 34;
// dostupne su iGlobalna1 i iGlobalna2

function funkcija1(){
var iLokalna1 = 4;
iGlobalna3 = 15;
// dostupne su iGlobalna1, iGlobalna2 i iLokalna1
}

// dostupne su iGlobalna1 i iGlobalna2
function funkcija2(){
var iLokalna2 = 7;
// dostupne su iGlobalna1, iGlobalna2 i iLokalna2
}
```

JAVASCRIPT FUNKCIJE (VIDEO MATERIJAL)

Izučavanje primene funckija u JavaScript jeziku može biti zaokruženo video materijalima.

JavaScript Tutorial For Beginners - #3 JavaScript Functions Tutorial (trajanje 13:48)

Ova lekcija sadrži video materijal. Ukoliko želite da pogledate ovaj video morate da otvorite LAMS lekciju.

Modern JavaScript Tutorial #4 - Functions (trajanje 43:33):

Ova lekcija sadrži video materijal. Ukoliko želite da pogledate ovaj video morate da otvorite LAMS lekciju.

✓ Poglavlje 6

HTML i JavaScript - Pokazna vežba 3

REZIME VAŽNIH HTML ELEMENATA.

Pre prelaska na izučavanje okvira neophodno je napraviti pregled najvažnijih HTML elemenata.

Pre prelaska na izučavanje okvira, čije se stranice baziraju na HTML standardima, neophodno je napraviti kraći pregled najvažnijih HTML elemenata, neophodnih za dalji rad, kroz teme:

1. Liste i rad sa listama;
2. Rad sa linkovima i tabelama;
3. HTML5 - okviri i forme

ZADATAK 1 (TRAJANJE: 20 MINUTA)

Samostalno vežbanje dodavanja JS koda u HTML - tabele.

Dat je sledeći primer, koji primenom JS omogućava prikaz tablice množenja u HTML dokumentu. Na početku se preko dijaloga zadaju dimenzije tablice, nakon čega se ona prikazuje u HTML stranici (slika 1).

```
<html>
<head>
    <title>Tablica množenja</title>
    <script type="text/javascript">
        var rows = prompt("Koliko vrsta ima tablica množenja?");
        var cols = prompt("Koliko kolona ima tablica množenja?");
        if(rows == "" || rows == null)
            rows = 10;
        if(cols== "" || cols== null)
            cols = 10;
        createTable(rows, cols);
        function createTable(rows, cols)
        {
            var j=1;
            var output = "<table border='1' width='500' cellspacing='0' cellpadding='5'>";
            for(i=1;i<=rows;i++)
            {
                output = output + "<tr>";
                while(j<=cols)
                {
                    output = output + "<td>" + (i-1)*cols+j + "</td>";
                    j++;
                }
                output = output + "</tr>";
            }
            document.write(output);
        }
    </script>
</head>
<body>
</body>

```

```
        output = output + "<td>" + i*j + "</td>";
        j = j+1;
    }
    output = output + "</tr>";
    j = 1;
}
output = output + "</table>";
document.write(output);
}

</script>
</head>
<body>
</body>
</html>
```

The screenshot shows a web browser window with the address bar containing "file:///C:/Users/Vlada/Desktop/Primer1.html". The main content area displays a 10x10 multiplication table. The table has 10 rows and 10 columns, with each cell containing the product of its row and column indices. The numbers are color-coded: the first row and column are black, while the other cells contain either orange or blue text.

1	2	3	4	5	6	7	8	9	10
2	4	6	8	10	12	14	16	18	20
3	6	9	12	15	18	21	24	27	30
4	8	12	16	20	24	28	32	36	40
5	10	15	20	25	30	35	40	45	50
6	12	18	24	30	36	42	48	54	60
7	14	21	28	35	42	49	56	63	70
8	16	24	32	40	48	56	64	72	80
9	18	27	36	45	54	63	72	81	90
10	20	30	40	50	60	70	80	90	100

Slika 6.1 Tablica množenja u HTML dokumentu

ZADATAK 2 (TRAJANJE: 25 MINUTA)

Samostalno vežbanje dodavanja JS koda u HTML - forme.

Sledećim primerom je omogućen unos korisničkih podataka putem forme i validacija polja email (slika 2):

```
<html>
<head>
    <title>Form Validation</title>
    <script type="text/javascript">
        var divs = new Array();
        divs[0] = "errFirst";
        divs[1] = "errLast";
        divs[2] = "errEmail";
```

```
divs[3] = "errUid";
divs[4] = "errPassword";
divs[5] = "errConfirm";
function validate()
{
    var inputs = new Array();
    inputs[0] = document.getElementById('first').value;
    inputs[1] = document.getElementById('last').value;
    inputs[2] = document.getElementById('email').value;
    inputs[3] = document.getElementById('uid').value;
    inputs[4] = document.getElementById('password').value;
    inputs[5] = document.getElementById('confirm').value;
    var errors = new Array();
    errors[0] = "<span style='color:red'>Please enter your first name!</span>";
    errors[1] = "<span style='color:red'>Please enter your last name!</span>";
    errors[2] = "<span style='color:red'>Please enter your email!</span>";
    errors[3] = "<span style='color:red'>Please enter your user id!</span>";
    errors[4] = "<span style='color:red'>Please enter your password!</span>";
    errors[5] = "<span style='color:red'>Please confirm your password!</span>";
    for (i in inputs)
    {
        var errMessage = errors[i];
        var div = divs[i];
        if (inputs[i] == "")
            document.getElementById(div).innerHTML = errMessage;
        else if (i==2)
        {
            var atpos=inputs[i].indexOf("@");
            var dotpos=inputs[i].lastIndexOf(".");
            if (atpos<1 || dotpos<atpos+2 || dotpos+2>=inputs[i].length)
                document.getElementById('errEmail').innerHTML = "<span style='color:red'>Enter a valid email address!</span>";
            else
                document.getElementById(div).innerHTML = "OK!";
        }
        else if (i==5)
        {
            var first = document.getElementById('password').value;
            var second = document.getElementById('confirm').value;
            if (second != first)
                document.getElementById('errConfirm').innerHTML = "<span style='color:red'>Your passwords don't match!</span>";
            else
                document.getElementById(div).innerHTML = "OK!";
        }
        else
            document.getElementById(div).innerHTML = "OK!";
    }
}
function finalValidate()
{
    var count = 0;
    for(i=0;i<6;i++)
```

```
{  
    var div = divs[i];  
    if(document.getElementById(div).innerHTML == "OK!")  
        count = count + 1;  
    }  
    if(count == 6)  
        document.getElementById("errFinal").innerHTML = "All the data you  
entered is correct!!!";  
    }  
</script>  
</head>  
<body>  
    <table id="table1">  
        <tr>  
            <td>First Name:</td>  
            <td><input type="text" id="first" onkeyup="validate();"/></td>  
            <td><div id="errFirst"></div></td>  
        </tr>  
        <tr>  
            <td>Last Name:</td>  
            <td><input type="text" id="last" onkeyup="validate();"/></td>  
            <td><div id="errLast"></div></td>  
        </tr>  
        <tr>  
            <td>Email:</td>  
            <td><input type="text" id="email" onkeyup="validate();"/></td>  
            <td><div id="errEmail"></div></td>  
        </tr>  
        <tr>  
            <td>User Id:</td>  
            <td><input type="text" id="uid" onkeyup="validate();"/></td>  
            <td><div id="errUid"></div></td>  
        </tr>  
        <tr>  
            <td>Password:</td>  
            <td><input type="password" id="password" onkeyup="validate();"/></td>  
            <td><div id="errPassword"></div></td>  
        </tr>  
        <tr>  
            <td>Confirm Password:</td>  
            <td><input type="password" id="confirm" onkeyup="validate();"/></td>  
            <td><div id="errConfirm"></div></td>  
        </tr>  
        <tr>  
            <td><input type="button" id="create" value="Create"  
onclick="validate();finalValidate();"/></td>  
            <td><div id="errFinal"></div></td>  
        </tr>  
    </table>  
</body>  
</html>
```

First Name: OK!

Last Name: OK!

Email: OK!

User Id: OK!

Password: OK!

Confirm Password: OK!

All the data you entered is correct!!!

Slika 6.2 Primer JS forme

✓ Poglavlje 7

Individualne vežbe 3

ZADATAK 1 (TRAJANJE: 45 MINUTA)

Samostalno modifukujte 1. zadatak sa pokaznih vežbi.

Zadatak:

1. Samostalno implementirajte pokazani primer 1;
2. Omogućite na sličan način prikazivanje korena i kvadrata prvih 10 brojeva u novoj tabeli.
3. dodajte link za odlazak na veb sajt našeg univerziteta.

ZADATAK 2 (TRAJANJE: 45 MINUTA)

Samostalno modifukujte 2. zadatak sa pokaznih vežbi.

Zadatak:

1. Samostalno implementirajte pokazani primer 2;
2. Kreirajte vlastitu formu za unos i proveru unetih podataka o automobilu sa poljima: marka, tip, snaga, kubikaža, boja;
3. snaga i kubikaža ne smeju da budu negativne vrednosti.

pomoći link: <https://stackoverflow.com/questions/3571717/javascript-negative-number>

✓ Poglavlje 8

Domaći zadatak 3

DOMAĆI ZADATAK BROJ 3 - 1 (PREDVIĐENO VREME 90 MINUTA)

Cilj domaćeg zadatka je da student provežba naučeno na vežbama

Za domaći zadatak broj 3 potrebno je da student kombinuje svoj rad iz DZ1 i DZ2, da primeni odgovarajući layout i stilove po želji, kao i da se osloni na primere urađene na individualnim vežbama.

- kreirajte HTML dokument sa jednom tabelom ili formom i primetite stil po izboru;
- dodajte JavaScript kod koji manipuliše podacima iz tabele ili forme;
- omogućiti prikazivanje rezultata manipulacije podacima primenom JavaScript-a.

Urađeni domaći zadatak pošaljite predmetnom asistentu na mail i adekvatno ga dokumentujte.

DOMAĆI ZADATAK BROJ 3 - 2 (PREDVIĐENO VREME 90 MINUTA)

Samostalno dodavanje JS koda u HTML

Zadatak 2: Učitavanje eksterne JavaScript datoteke u HTML

1. Napisati primer JavaScript fajla koji se učitava u okviru HTML strane kao eksterni fajl;
2. Izgled HTML strane je definisan sledećom slikom;
3. JavaScript fajl sadrži funkcije za pronalaženje rezultata i za obradu forme sa slike.

VAŽNO!!! Rešenja preuzeta sa Interneta se neće bodovati!!!

Urađeni domaći zadatak pošaljite predmetnom asistentu na mail i adekvatno ga dokumentujte.

KALKULATOR

Unesite prvi broj:

Unesite drugi broj:

Rezultat :

Slika 8.1 HTML forma za DZ

▼ Poglavlje 9

Zaključak

ZAKLJUČAK

Lekcija se bavila osnovnim konceptima i principima JavaSrcipt jezika.

Lekcija se bavila osnovnim konceptima i principima *JavaSrcipt* jezika. U fokusu lekcije su bile teme:

- standardizacija jezika;
- osnove sintakse;
- promenljive i objekti;
- operatori i funkcije;
- upravljanje kontrolom toka;
- ostali koncepti osnovne sintakse *JavaSrcipt* jezika.

Savladavanjem ove lekcije, student je osposobljen da koristi osnovne koncepte *JavaSrcipt* jezika i da pređe na izučavanje naprednih kao osnova za savladavanje *Angular / TypeScript* jezika koji predstavlja proširenje standardnog *JavaSrcipt* jezika.

LITERATURA

Za pripremu lekcije je korišćena aktuelna pisana i veb literatura.

Obavezna literatura:

1. Jennifer Niederst Robbins, Learning Web Design - Fifth Edition, by , Copyright ©; 2018 O'Reilly Media, Inc.
2. Roxane Anquetil, Fundamental Concepts for Web Development: HTML5, CSS3, JavaScript and much more! For complete beginners!, 2019,

Dopunska literatura:

1. Daniel Bell, HTML & CSS: A Step-by-Step Guide for Beginners2, Guzzler Media, 2019.

Veb literatura:

1. <https://www.ecma-international.org/>
2. https://www.srce.unizg.hr/files/srce/docs/edu/osnovni-tecajevi/c501_polaznik.pdf
3. <https://skolakoda.org/javascript-null-i-undefined>



IT255 - VEB SISTEMI 1

**HTML5 i Javascript jezik
(napredni koncepti)**

Lekcija 04

PRIRUČNIK ZA STUDENTE

IT255 - VEB SISTEMI 1

Lekcija 04

HTML5 I JAVASCRIPT JEZIK (NAPREDNI KONCEPTI)

- ✓ HTML5 i Javascript jezik (napredni koncepti)
- ✓ Poglavlje 1: Rad sa objektima
- ✓ Poglavlje 2: Napredni rad sa funkcijama
- ✓ Poglavlje 3: Podešavanje osobina objekata u JavaScript - u
- ✓ Poglavlje 4: Prototipovi, nasleđivanje
- ✓ Poglavlje 5: JavaScript klase
- ✓ Poglavlje 6: Rukovanje greškama
- ✓ Poglavlje 7: Objekti tipa Promise - async/await sintaksa
- ✓ Poglavlje 8: Materijali za dodatni rad
- ✓ Poglavlje 9: Pokazna vežba - HTML, CSS i JS
- ✓ Poglavlje 10: Individualna vežba 4
- ✓ Poglavlje 11: Domaći zadatak 4
- ✓ Zaključak

Copyright © 2017 – UNIVERZITET METROPOLITAN, Beograd. Sva prava zadržana. Bez prethodne pismene dozvole od strane Univerziteta METROPOLITAN zabranjena je reprodukcija, transfer, distribucija ili memorisanje nekog dela ili čitavih sadržaja ovog dokumenta., kopiranjem, snimanjem, elektronskim putem, skeniranjem ili na bilo koji drugi način.

Copyright © 2017 BELGRADE METROPOLITAN UNIVERSITY. All rights reserved. No part of this publication may be reproduced, stored in a retrieval system or transmitted in any form or by any means, electronic, mechanical, photocopying, recording, scanning or otherwise, without the prior written permission of Belgrade Metropolitan University.

✓ Uvod

UVOD

Lekcija se bavi izučavanjem naprednih JS koncepata.

Lekcija se bavi izučavanjem naprednih JS koncepata i na taj način zaokružuje pripremu za učenje razvoja frontend komponenata veb sistema primenom jezičke nadogradnje za *JavaScript ECMA 6* pod nazivom *TypeScript*. Jezik *TypeScript* je osnov svega što će biti rađeno u nastavku primenom radnog okvira *Angular*.

Lekcija je fokusirana na sledeće teme, držeći se snažno pokaznih primera kao podrške za analizu, diskusiju i demonstraciju:

1. Napredni rad sa funkcijama u *JavaScript* jeziku;
2. Podešavanje objektnih osobina u *JavaScript* jeziku;
3. Prototipovi, nasleđivanje;
4. Klase;
5. Rukovanje greškama;
6. Koncepti Promises, *async / await*;
7. Generatori, napredne iteracije;
8. Moduli;
9. Ostali napredni JavaScript koncepti kao uvod u Angular okvir.

Savladavanjem ove lekcije student će biti u potpunosti pripremljen na izučavanje frontend radnog okvira *Angular*.

UVODNI VIDEO

Trajanje video snimka: 5min 58sek

Ova lekcija sadrži video materijal. Ukoliko želite da pogledate ovaj video morate da otvorite LAMS lekciju.

✓ Poglavlje 1

Rad sa objektima

VIDEO PREDAVANJE ZA OBJEKAT "RAD SA OBJEKTIMA"

Trajanje video snimka: 12min 56sek

Ova lekcija sadrži video materijal. Ukoliko želite da pogledate ovaj video morate da otvorite LAMS lekciju.

OBJEKTI U JS

Za razliku od primitivnih tipova podataka, objekti sadrže kolekciju podataka različitih tipova.

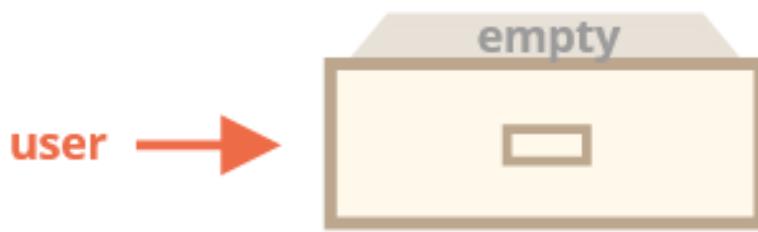
U prethodnoj lekciji je razmatran koncept primitivnih tipova podataka, u JavaScript jeziku, koji mogu sadržati samo jednu vrednost.

Za razliku od primitivnih tipova podataka, objekti sadrže kolekciju podataka različitih tipova. U JavaScript jeziku primena objekata se proteže na skoro sve aspekte jezika. Iz navedenog razloga je bitno dobro razumevanje objekata pre upuštanja u dalji rad sa naprednim konceptima jezika.

U JavaScript-u, objekat je moguće kreirati primenom velikih (vitičastih) zagrada sa opcionalnom listom osobina. Osobine objekata u JS su definisane u formi parova (*ključ, vrednost*) pri čemu je ključ naziv osobine i tipa je *string*, dok vrednost može biti bilo kojeg tipa. Objekat je moguće kreirati i primenom konstruktora odgovarajuće klase.

Prazan objekat je moguće kreirati upotrebom sledeće JS sintakse:

```
let user = new Object(); // kreiranje objekta konstruktorom  
let user = {};// sintaksa objektnog literala
```



Slika 1.1 Ilustracija praznog objekta [izvor: javascript.info]

LITERALI I OSOBINE JS OBJEKATA

Najkraći način za definisanje objekata je upotrebom stila sintaksa objektnog literalja.

Prilikom definisanja objekata moguće je postupiti po scenariju da je trenutno moguće kreirati niz parova (*ključ, vrednost*) unutar zagrade `{...}`. Ovakav stil kreiranja objekata u JavaScript jeziku naziva se sintaksa objektnog literalja.

Ovaj stil kreiranja objekata je moguće predstaviti sledećim listingom:

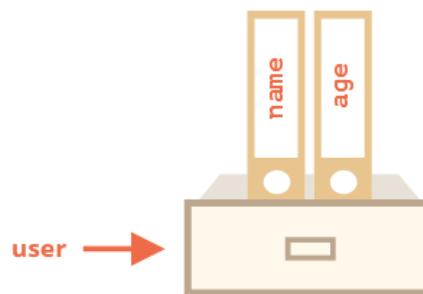
```
let user = {      // objekat
    name: "Petar", // u ključu "name" čuva se vrednost "Petar"
    age: 30        // u ključu "age" čuva se vrednost 30
};
```

Svaka osobina objekta sadrži ključ (poznat pod nazivom "ime", "naziv" ili "identifikator") iza kojeg sledi simbol "dvotačka" (:), a zatim i vrednost pridružena navedenoj osobini.

U konkretnom slučaju, kreiran je objekat `user` koji poseduje dve osobine:

- `name` sa vrednošću "Petar";
- `age` sa vrednošću 30.

Sledećom slikom može biti vizuelno ilustrovan kreirani objekat - kao ostava (fioka) u kojoj se čuvaju dve fascikle obeležene sa "`name`" i "`age`" u kojima se čuvaju odgovarajuće informacije - vrednosti osobina objekata.

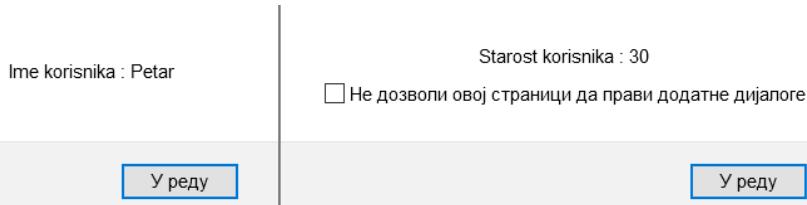


Slika 1.2 Vizuelna ilustracija objekta user [izvor: javascript.info]

Dodavanjem koda u **HTML** stranicu, u tagu **<script>** i pozivanjem naredbe za prikazivanje informacija o objektu moguće je demonstrirati najjednostavniji način primene objekata u JS programu. Ispod kreiranog objekta dodaje se još malo koda:

```
alert("Ime korisnika : " + user.name );
alert("Starost korisnika : " + user.age);
```

Učitavanjem kreirane stranice sa prikazanim JS kodom u pregledač, dobija se izlaz kao na sledećoj slici:



Slika 1.3 Prikaz osobina objekata u JS programu [izvor: autor]

DODAVANJE I UKLANJANJE OSOBINA OBJEKATA

JavaScript nudi fleksibilan pristup modifikovanju kreiranih objekata.

JavaScript nudi fleksibilan pristup modifikovanju kreiranih objekata. Moguće je naknadno:

- dodavanje novih osobina objekata;
- brisanje postojećih osobina objekata.

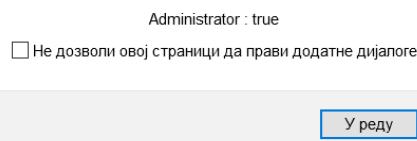
Dodavanje novih osobina je veoma jednostavno. Navodi se naziv objekta, a onda primenom sintakse "tačka" uvodi se nova promenljiva (objektna osobina) kojoj je moguće dodeliti vrednost. Sledećim listingom je ilustrovano dodavanje nove osobine objektu **user** (linija koda 6):

```
let user = {      // objekat
    name: "Petar", // u ključu "name" čuva se vrednost "Petar"
    age: 30        // u ključu "age" čuva se vrednost 30
};

user.isAdmin = true;
```

Sada je moguće dodati još jednu JavaScript naredbu za proveru da li objekat **user** poseduje novu osobinu:

```
alert("Administrator : " + user.isAdmin);
```



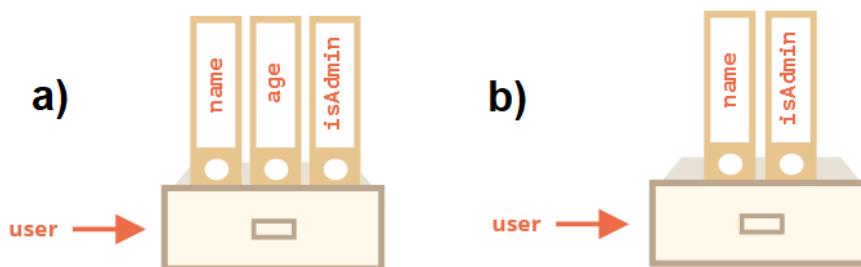
Slika 1.4 Prikaz nove osobine objekta [izvor:autor]

Prethodnom slikom je prikazana vrednost za novu osobinu objekta `user` pod nazivom `isAdmin`. Kao što je već rečeno, **vrednost može biti bilo kojeg tipa, dok su ključevi tipa string**.

Kao što je obavljeno dodavanje nove objektne osobine, `JavaScript` dozvoljava uklanjanje postojećih osobina iz kreiranog objekta. Za navedenu funkcionalnost je zadužena `JS` naredba `delete`:

```
delete user.age;
```

Sadržaj objekta nakon dodavanja nove osobine prikazan je slikom 5 - a, a nakon brisanja postojeće, prikazan je slikom 5 - b.



Slika 1.5 Sadržaj objekta nakon manipulacije njegovim osobinama [izvor: javascript.info]

DODATNO RAZMATRANJE U VEZI SA OSOBINAMA OBJEKATA

Naziv osobine objekta može biti izgrađen iz više reči.

Naziv osobine objekta, za razliku od nekih drugih skripting i programskega jezika, može biti izgrađen iz više reči. U ovom slučaju, naziv osobine se piše pod navodnicima, a to je ilustrovano na sledeći način:

```
let user = {      // objekat
    name: "Petar", // u ključu "name" čuva se vrednost "Petar"
    age: 30,        // u ključu "age" čuva se vrednost 30
    "studira informatiku": true // osobina sa nazivom od više reči mora da bude pod
                                navodnicima
};
```

Takođe, osobine objekta koji je definisan kao konstanta moguće je menjati. Neophodno je pogledati sledeći listing:

```
const user = {
    name: "Petar"
};

user.name = "Marko"; // (*)

alert(user.name); // Marko
```

Na prvi pogled moguće je zaključiti da će linija koda obeležena sa (*) izazvati grešku u programu, ali neće. Rezervisana reč `const` fiksira vrednosti za objekat `user`, ali ne i za njegov sadržaj.

Primena rezervisane reči `const` proizvešće grešku samo u slučaju pokušaja izmene objekta `user` u celini.

PRIMENA UGLASTIH ZAGRADA

Za osobine sa složenim nazivom, sintaksa tačke nije funkcionalna.

Za osobine sa složenim nazivom (iz više reči), sintaksa tačke nije funkcionalna. Sledeći kod bi rezultovao greškom u programu:

```
user.studira informatiku = true
```

`JavaScript` ne bi razumeo ovu naredbu na pravi način. Jezik bi smatrao da je pokušan pristup osobini `user.studira`, a ostatak koda bi kreirao sintaksnu grešku.

`JavaScript` poseduje rešenje za ovakav vid problema u formi upotrebe uglastih zagrada za obuhvatanje naziva osobine koji je sastavljen iz više reči. Ovakva sintaksa je primenljiva na bilo koji `string`:

```
let user = {};  
  
// set  
user["studira informatiku"] = true;  
  
// get  
alert(user["studira informatiku"]); // true  
  
// delete  
delete user["studira informatiku"];
```

Ukoliko se postupi na prikazani način, neće biti sintaksne greške u kodu.

Moguće je postupiti i na sledeći način. Primena sintakse uglastih zagrada obezbeđuje način dobijanja naziva osobine kao rezultata bilo kojeg izraza ili promenljive. Moguće je postupiti na sledeći način:

```
let key = "studira informatiku";  
  
// potpuno isto kao user["studira informatiku"] = true;  
user[key] = true;
```

Varijabla "`key`" može biti određena (izračunata) i tokom vremena izvršavanja programa. Posmatra se sledeći listing:

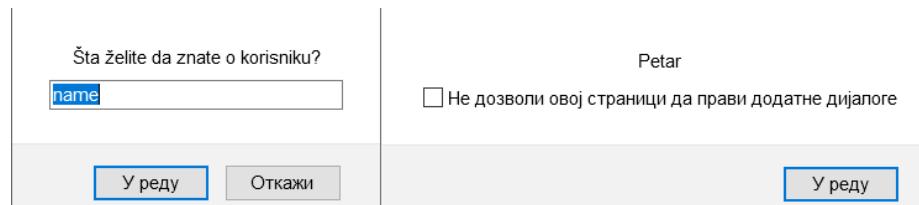
```
let user = { // objekat  
    name: "Petar", // u ključu "name" čuva se vrednost "Petar"
```

```
age: 30          // u ključu "age" čuva se vrednost 30
//studira informatiku": true // osobina sa nazivom od više reči mora da bude pod
navodnicima
};

let key = prompt("Šta želite da zнате о кориснику?", "name");

// pristup promenljivoj
alert( user[key] ); // Petar (ukoliko se unese "name")
```

Rešenje listinga je moguće prikazati sledećom slikom



Slika 1.6 Računanje vrednosti varijable tokom vremena izvršavanja programa [izvor: autor]

PROGRAMSKA MANIPULACIJA OSOBINAMA OBJEKATA

U realnom kodu često se koriste postojeće promenljive kao vrednosti za objektne osobine.

U realnom kodu često se koriste postojeće promenljive kao vrednosti za objektne osobine.

Na primer, posmatra se sledeći *JavaScript* listing:

```
let user = {      // objekat
    name: "Petar", // u ključu "name" čuva se vrednost "Petar"
    age: 30        // u ključu "age" čuva se vrednost 30
//studira informatiku": true // osobina sa nazivom od više reči mora da bude pod
navodnicima
};

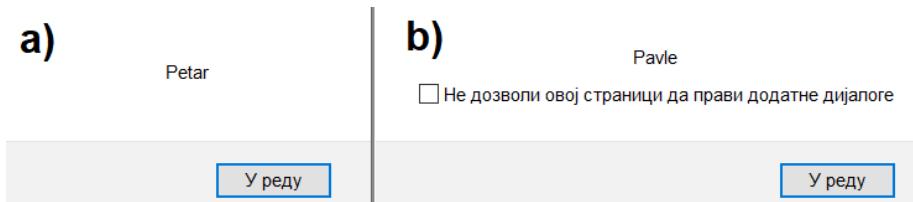
alert(user.name); // Petar

function makeUser(name, age) {
    return {
        name: name,
        age: age,
        // ...ostale osobine ako postoje
    };
}

user = makeUser("Pavle", 28);
alert(user.name); // Pavle
```

U prvom koraku, inicijalno je kreiran objekat `user` sa unapred zadatim osobinama (linije koda 1 - 6). Takav objekat je i prikazan na ekranu (slika 7 - a) preko svoje osobine `user.name` (linija koda 7).

U nastavku je kreirana JavaScript funkcija `makeUser()` koja uzima dva argumenta `name` i `age` (linije koda 9 - 15). Povratna vrednost funkcije je objekat `user` čije osobine odgovaraju argumentima kreirane metode. Takav objekat je i prikazan na ekranu (slika 7 - b) preko svoje osobine `user.name` (linija koda 7 - b), nakon što je ponovo kreiran (linija koda 17).



Slika 1.7 Programska manipulacija osobinama objekata [izvor: autor]

Ovo je odlično mesto da se ponovo vratimo na primenu rezervisane reči `const`. Ako bi objekat `user`, u liniji koda 1, bio deklarisan sa `const`, umesto sa `let`, linija koda 17, u kojoj se menja objekat u celini, izazvala bi grešku u programu. Navedenu grešku je moguće identifikovati u konzoli veb pregledača (klikom na F12) i prikazana je sledećom slikom.

Slika 1.8 Greška - dodata vrednosti konstantnom objektu [izvor: autor]

OGRANIČENJA NAZIVA OBJEKTNIH OSOBINA

Ne postoje značajna ograničenja na polju davanja naziva objektnim osobinama.

U prethodnoj lekciji je već diskutovano da JS promenljive ne mogu da imaju nazive koji odgovaraju rezervisanim rečima jezika, poput: `var`, `let`, `const`, `for` i tako dalje.

Međutim, kada su u pitanju nazivi objektnih osobina, **takva ograničenja ne postoje**. Navedeno je moguće ilustrovati sledećim listingom:

```
// moguće je koristiti sledeće objektne osobine
let obj = {
    for: 1,
    let: 2,
    return: 3
};

alert( obj.for + obj.let + obj.return ); // rešenje je 6
```

PROVERA POSTOJANJA OSOBINE U OBJEKTU

JavaScript može da pristupi osobini objekta čak i kada ona nije definisana.

Bitna funkcionalnost JavaScript jezika, za razliku od nekih drugih programskih i skripting jezika, jeste mogućnost pristupa bilo kojoj osobini objekta. Neće se pojaviti greška čak i u slučaju kada osobina ne postoji.

Čitanje nepostojeće osobine će kao rezultat jednostavno vratiti `undefined`. Veoma je jednostavno obaviti testiranje da li je neka osobina nepostojeća, kao na sledeći način:

```
let user = {};  
  
alert( user.nepostojacaOsobina === undefined ); // true znači "ne postoji osobina"
```

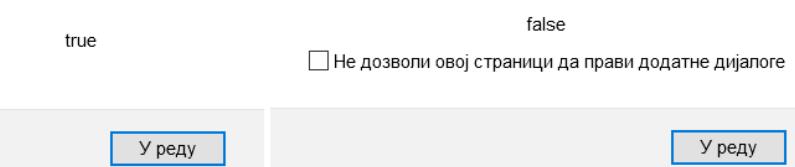
Za proveru postojanja osobine u objektu, JavaScript je predviđao primenu operatora `in`. Osnovna sintaksa upotrebe ovog operatora glasi:

```
"key" in object
```

Primena operatora `in` je veoma jednostavna i može biti ilustrovana sledećim kodom:

```
let user = { name: "Petar", age: 30 };  
  
alert( "age" in user ); // true, user.age postoji  
alert( "nekaOsobina" in user ); // false, user.nekaOsobina ne postoji
```

Prethodni kod je testiran unutar odgovarajuće HTML stranice i dao je sledeći izlaz nakon učitavanja stranice u veb pregledač.



Slika 1.9 Provera postojanja osobine u objektu [izvor:autor]

ITERACIJA PREKO KLJUČEVA OBJEKTA I SORTIRANJE PO KLJUČEVIMA

Za iteraciju preko ključeva koristi se petlja.

Za iteraciju preko ključeva koristi se petlja. Ovo je potpuno drugačija logika od standardne petlje `for(; ;)` koja je izučavana u prethodnoj lekciji.

Opšti oblik petlje `for`, koja obavlja navedeni zadatak, glasi:

```
for (let key in object) {  
    // izvršava telo petlje za svaki ključ  
    // koji odgovara objektnim osobinama  
}
```

U konkretnom primeru, iteracija preko ključeva može biti ilustrovana na sledeći način:

```
let user = {  
    name: "Petar",  
    age: 30,  
    isAdmin: true  
};  
  
for (let key in user) {  
    // ključevi  
    alert( key ); // vraća : name, age, isAdmin  
    // vrednosti za ključeve  
    alert( user[key] ); // vraća: Petar, 30, true  
}
```

Iz listinga, a i opšteg oblika petlje, moguće je sagledati da promenljiva koja iterira kroz ključeve mora da bude deklarisana sa *let*. Iteracijom je moguće vratiti ključeve (linija koda 9) ali i njima odgovarajuće vrednosti (linija koda 11).

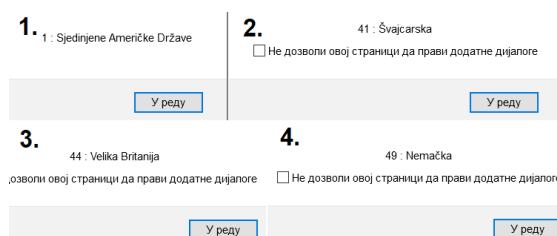
Moguće je, na ovom nivou, postaviti pitanje: "Da li je sadržaj objekata sortiran"? Sortiranje ovog tipa je posebna funkcionalnost *JavaScript* jezika:

- celi brojevi su sortirani po vrednosti;
- ostale osobine se pojavljuju po redosledu dodavanja.

Pogledajmo sledeći primer gde se kao ključevi koriste stringovi u koje su upisani celi brojevi:

```
let codes = {  
    "49": "Nemačka",  
    "41": "Švajcarska",  
    "44": "Velika Britanija",  
    "381": "Srbija",  
  
    // ...  
    "1": "Sjedinjene Američke Države"  
};  
  
for (let code in codes) {  
    alert(code + " : " + codes[code]); // 1, 41, 44, 49, 381  
}
```

Budući da su ključevi dati u formi celih brojeva, kroz iteraciju će biti sortirani, a to je prikazano sledećom slikom:



Slika 1.10 Sortiranje po ključevima [izvor: autor]

RAD SA OBJEKTIMA U JS JEZIKU (VIDEO MATERIJAL)

Diskusija rada sa objektima u JS jeziku zaokružena je preko adekvatnog video materijala.

What Are Objects in JavaScript | How to Create an Object in JavaScript | JavaScript Tutorial (trajanje: 11:34).

Ova lekcija sadrži video materijal. Ukoliko želite da pogledate ovaj video morate da otvorite LAMS lekciju.

▼ Poglavlje 2

Napredni rad sa funkcijama

REKURZIJA VS ITERACIJA

Rekurzija je prva napredna primena JS funkcija koja će biti razmatrana.

U nastavku je cilj da se znanje stečeno u prethodnoj lekciji dodatno produbi. U prvom delu akcenat će biti na JavaScript konceptu funkcija sa čijim osnovnim elementima je prethodna lekcija završila izlaganje,

Sledi izučavanje naprednih primena funkcija u JavaScript programima. Rekurzija je prva napredna primena JS funkcija koja će biti razmatrana.

Budući da nije prvi put da se studenti sreću sa rekurzijom, koncept im je od ranije poznat. Ovde će u fokusu biti pisanje i primena rekurzivnih funkcija u JavaScript jeziku.

Za obnavljanje, pod rekurzijom se podrazumeva šablon programiranja koji je veoma pogodan kada neki složeni problem može da se razloži na jednostavnije potprobleme istog tipa i sve tako dok se ne stigne do najjednostavnijeg slučaja za koji je rešenje problema poznato. Vraćanjem unazad, rešavaju se svi ostali slučajevi, zaključno sa početnim.

U programiranju ovakav problem je često rešen mogućnošću neke funkcije (osnovni - početni problem) da poziva samu sebe (za jednostavnije potprobleme). Za funkciju mora da postoji osnovni slučaj (uslov zaustavljanja rekurzije) za koji je poznata njena vrednost.

Uopšteno - sposobnost funkcije da poziva samu sebe naziva se rekurzija.

Uporedimo dva načina razmišljanja u JavaScript jeziku: rešavanje problema primenom iteracija i rešavanje identičnog problema primenom rekurzije. Posmatraju se sledeći pozivi funkcije pow():

```
pow(2, 2) = 4  
pow(2, 3) = 8  
pow(2, 4) = 16
```

Očigledno je da se radi o funkciji koja stepenuje prvi argument drugim i vraća odgovarajuće rešenje. Sledećim listingom bi ovakva funkcija bila kodirana na tradicionalni - iterativni način:

```
function pow(x, n) {  
    let result = 1;  
  
    // množi rezultat sa x n puta u petlji  
    for (let i = 0; i < n; i++) {
```

```
    result *= x;  
}  
  
return result;  
}  
  
alert( pow(2, 3) ); // rešenje je 8
```

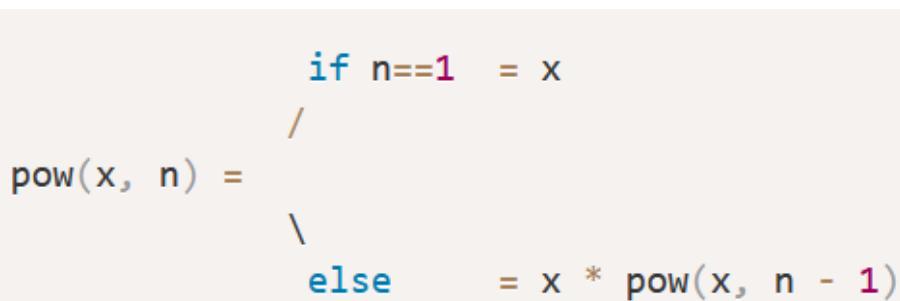
REKURZIVNO REŠENJE PROBLEMA

Prethodni problem može biti elegantnije rešen ako se uključi rekurzivni pristup.

Ako se pogleda prethodni listing vidi se tradicionalni pristup kreiranju metoda u kojima se izvršava višestruko množenje - primenom iteracija (petlje). Međutim ovakav problem može biti elegantnije rešen ako se uključi rekurzivni pristup:

1. Uoči se osnovni slučaj, koji se još naziva i uslovom za zaustavljanje rekurzije, a to može biti slučaj da svaki broj stepenovan jedinicom daje rešenje identično samom sebi.
2. Razbijanje na jednostavnije potprobleme može biti po scenariju da je vrednost dobijena stepenovanjem broja x brojem n jednaka proizvodu broja x i stepenu x na n-1. Na ovaj način je početni primer umanjio svoju složenost;
3. Ovaj postupak se ponavlja sve do uslova za zaustavljanje.

Navedeno može biti ilustrovano sledećom slikom:



```
if n==1 = x
/
pow(x, n) =
 \
else      = x * pow(x, n - 1)
```

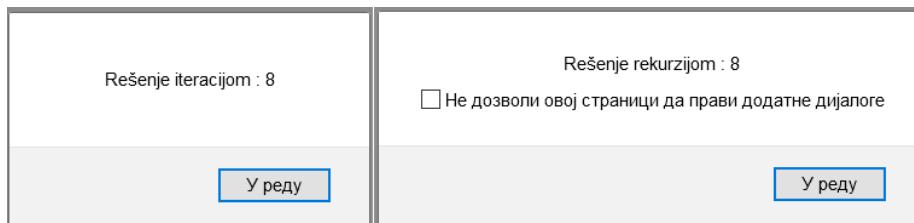
Slika 2.1 Šablon rekurzivnog rešenja [izvor: javascript.info]

Po šablonu prikazanom na slici moguće je napraviti adekvatno rekurzivno rešenje kao zamena za prethodno prikazano iterativno. Sledi listing:

```
function powRek(x, n) {  
    if (n == 1) {  
        return x;  
    } else {  
        return x * powRek(x, n - 1); //rekurzivni poziv  
    }  
}
```

```
alert( "Rešenje rekurzijom : " + powRek(2, 3) ); // rezultat je 8
```

Kao što je već analizirano, sada imamo i kompletну implementaciju. U liniji koda 2 javlja se osnovni slučaj - uslov za zaustavljanje rekurzije, a u liniji koda imamo rekurzivni poziv - poziv funkcije za jednostavniji slučaj. Sledećom slikom je prikazano rešenje poziva ove metode u oba obrađena slučaja:



Slika 2.2 Rešenja poziva iterativne i rekurzivne metode [izvor: autor]

PROMENLJIVA LISTA ARGUMENATA FUNKCIJE

JavaScript podržava rad sa promenljivom listom argumenata za kreirane funkcije.

JavaScript podržava rad sa promenljivom listom argumenata za kreirane funkcije, ali i za bojne ugrađene. Na primer:

- `Math.max(arg1, arg2, ..., argN)` – kao rezultat vraća najveći od argumenata;
- `Object.assign(dest, src1, ..., srcN)` – kopira osobine iz objekata src1 ... srcN u obejkat dest;

Posmatrajmo sledeći slučaj:

```
function sum(a, b) {
    return a + b;
}
alert( "Funkcija vraća rešenje: " + sum(1, 2, 3, 4, 5) );
```

Funkcija je definisana sa dva argumenta, međutim, u njenom pozivu učestvuje njih 5. U JavaScript - u funkcija može biti pozvana sa različitim brojem argumenata, bez obzira na način na koji je definisana i navedeni poziv neće izazvati grešku. Međutim, u rezultatu će učestvovati samo prva dva argumenta.

Funkcija vraća rešenje: 3

Не дозволи овој страници да прави додатне дијалоге

Уреди

Slika 2.3 Poziv funkcije sa većim brojem argumenata od deklarisanog [izvor: autor]

Postavlja se pitanje - kako je moguće uključiti preostale argumente u obradu funkcije?

Ostale argumente funkcije je moguće uključiti u definiciju funkcije primenom sintakse "tri tačke" (*tree dots syntax*) iza kojih sledi naziv niza koji sadrži navedene argumente. Tri tačke imaju bukvalno značenje - **dodaj preostale argumente u niz**. Na primer, dodavanje preostalih argumenata u niz, u prethodnom primeru, moglo bi da se napiše na sledeći način:

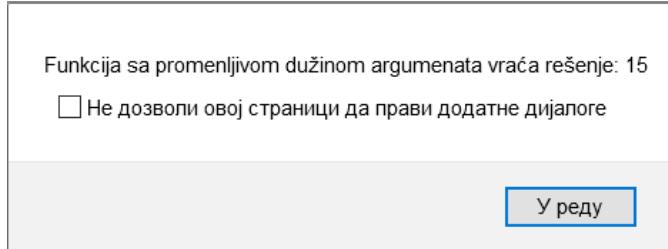
```
function sumAll(...args) { // args je naziv za niz
    let sum = 0;

    for (let arg of args) sum += arg;

    return sum;
}

alert( "Funkcija sa promenljivom dužinom argumenata vraća rešenje: " + sumAll(1, 2,
3, 4, 5) );
```

Nakon dodavanja argumenata, iz linije koda 9, u niz, unutar petlje argumenti se dodaju na zbir. Na ovaj način će funkcija moći da obradi proizvoljan niz argumenata.



Slika 2.4 Promenljiva lista argumenata funkcije - primena [izvor: autor]

PROMENLJIVA LISTA ARGUMENATA FUNKCIJE - DODATNO RAZMATRANJE

Scenario: prvih nekoliko atributa su stalni, a preostali predstavljaju niz različite dužine.

Moguće je razmišljati i na sledeći način kada je u pitanju rad sa promenljivom listom argumenata u funkcijama *JavaScript* jezika - prvih nekoliko atributa su stalni, a preostali predstavljaju niz različite dužine.

Navedeno je moguće ilustrovati sledećim listingom:

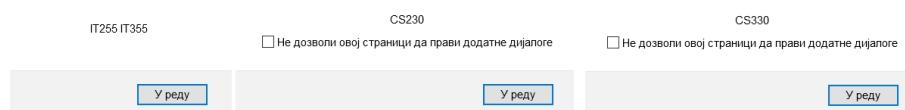
```
function showName(predmet1, predmet2, ...predmeti) {
    alert( predmet1 + ' ' + predmet2 ); // IT255 IT355

    // preostali se čuvaju u nizu argumenata
    // npr. predmeti = ["CS230", "CS330"]
    alert( predmeti[0] ); // CS230
    alert( predmeti[1] ); // CS330
    alert( predmeti.length ); // 2
}
```

```
showName( "IT255", "IT355", "CS230", "CS230" );
```

Iz listinga je moguće primetiti da su atributi *predmet1* i *predmet2* stalni, a ukoliko postoje preostali, koji se navode iza stalnih atributa oni će biti sačuvani u nizu atributa *predmeti*.

Listing daje izlaz kao na sledećoj slici:



Slika 2.5 Rad sa nizovima argumenata [izvor: autor]

Pa ukoliko se pozove funkcija kao u liniji koda 11, prikazivaće rezultate na sledeći način:

- prikazuje se string kojeg čine atributi *predmet1* i *predmet2* (linija koda dva; rezultat je prvi segment slike 5);
- prikazuje se string koji prikazuje prvi argument u nizu (linija koda 6; rezultat je drugi segment slike 5);
- prikazuje se string koji prikazuje drugi argument u nizu (linija koda 7; rezultat je treći segment slike 5);
- prikazuje dužinu (broj argumenata u nizu) niza argumenata (linija koda 8).

VAŽNO:

Niz "preostalih" argumenata funkcije, tj. svih onih koji nisu stalni mora da se piše na kraju liste argumenata!!! Sledеći kod bi proizveo grešku u programu:

function f(arg1, ...ostali, arg2) { // arg2 je iza...ostali ?!

// greška!!!}

SINTAKSA ŠIRENJA

Moguće je dobiti i potrebne argumente iz niza kao ulaz za konkretne funkcije.

U prethodnom izlaganju je pokazano kako je moguće formirati niz na osnovu dostupne liste argumenata. Ponekad je, zapravo, potrebno uraditi obrnuto.

Na primer, posmatra se ugrađena *JavaScript* funkcija *Math.max()* koja vraća najveći broj iz liste brojeva. Sledećim listingom je demonstriran poziv navedene funkcije:

```
alert( Math.max(3, 5, 1) ); // vraћа rezultat 5
```

Posmatrajmo sada numerički niz koji odgovara navedenoj listi argumenata: [3, 5, 1]. Kako je moguće primenom ove funkcije odrediti njegov najveći član?

Prosleđivanje niza kao argumenta funkciji neće pomoći prilikom rešavanja ovog problema jer funkcija kao argumente očekuje listu brojeva, a ne jedinstveni niz. Sledeći listing bi proizveo grešku:

```
let arr = [3, 5, 1];  
  
alert( Math.max(arr) ); // NaN
```

Moguće rešenje predstavlja manuelno dodavanje članova niza kao argumenata funkcije:

```
Math.max(arr[0], arr[1], arr[2])
```

Međutim, ovo svakako nije dobro rešenje budući da, uglavnom, nije poznato koliko niz ima članova. Kako se kreirani skript izvršava, moguće je da postoji njih puno, a moguće je i da je niz prazan.

Kao odlično rešenje za ovakav tip problema nameće se **sintaksa širenja** (*spread syntax*). Veoma je slična sintaksi opisanoj u prethodnom izlaganju, ali deluje u suprotnom smeru.

kada se koristi ...arr unutar poziva funkcije, on "proširuje" iterabilni niz objekata u listu argumenata funkcije. Na primer, za `Math.max()` moguće je primeniti sledeće rešenje:

```
let arr = [3, 5, 1];  
  
alert( Math.max(...arr) ); // 5 (sintaksa prevodi niz u listu argumenata)
```

SINTAKSA ŠIRENJA - RAZLIČITI SCENARIJI PRIMENE

Sintaksa širenja je primenljiva na različite scenarije primene i pokazuje visoku fleksibilnost.

Sintaksa širenja je primenljiva na različite scenarije primene i pokazuje visoku fleksibilnost. Posmatrajmo sledeće scenarije upotrebe sintakse širenja:

- prosleđivanje većeg broja različitih nizova funkciji;
- kombinovanje niza sa običnim argumentima;
- spajanje nizova;
- dobijanje nove kopije niza / objekta.

Prethodni primer može biti rezonovan i na drugačiji način - neophodno je prosleđivanje većeg broja različitih nizova funkciji `Math.max()`. Prilikom pozivanja funkcije, oslanjajući se na sintaksu širenja, to je moguće učiniti na sledeći način:

```
let arr1 = [1, -2, 3, 4]; //prvi niz  
let arr2 = [8, 3, -8, 1]; //drugi niz  
  
//poziv funkcije primenom sintakse širenja  
alert( Math.max(...arr1, ...arr2) ); // rešenje je 8
```

Takođe, moguće je i kombinovanje nizova sa običnim vrednostima unutar listi sa argumentima funkcije. Navedeni scenario je veoma jednostavno opisati sledećim listingom:

```
let arr1 = [1, -2, 3, 4];
let arr2 = [8, 3, -8, 1];

alert( Math.max(1, ...arr1, 2, ...arr2, 25) ); // rešenje je 25
```

Konačno, sledi i poslednji od izabranih scenarija primene sintakse širenja. U JavaScript jeziku primenom ove sintakse je moguće na veoma jednostavan način obaviti spajanje nizova. Sve je prikazano i opisano unutar sledećeg listinga:

```
let arr = [3, 5, 1];
let arr2 = [8, 9, 15];

//spajanje nizova primenom sintakse širenja
let merged = [0, ...arr, 2, ...arr2];

alert(merged); // 0,3,5,1,2,8,9,15 (0, sledi arr, sledi 2, sledi arr2)
```

Postoji još mnogo načina za primenu navedene sintakse poziva JavaScript funkcija, ovde su izabrani neki karakteristični slučajevi

DOBIJANJE NOVE KOPIJE NIZA / OBJEKTA

Dobijanje nove kopije niza / objekta je funkcionalnost koju je moguće rešiti sintaksom širenja.

Dobijanje nove kopije niza / objekta je funkcionalnost koju je moguće rešiti sintaksom širenja na alternativni ali i veoma jednostavan način. Kopiranje jednog objekta u drugi, što je u osnovi zadatak metode `Object.assign()`, može biti obavljen na sledeći način primenom sintakse širenja:

```
let arr = [1, 2, 3];
let arrCopy = [...arr]; //proširuje niz u listu parametara
                        // zatim čuva rezulatat u novom nizu

// Da li nizovi imaju isti sadržaj?
alert(JSON.stringify(arr) === JSON.stringify(arrCopy)); // true

// Da li su nizovi jednaki?
alert(arr === arrCopy); // false (nisu iste reference)

// modifikacija originalnog niza ne modifikuje kopiju:
arr.push(4);
alert(arr); // 1, 2, 3, 4
alert(arrCopy); // 1, 2, 3
```

Potpuno isti scenario je moguće primeniti i na objekte.

```
let obj = { a: 1, b: 2, c: 3 };
let objCopy = { ...obj }; // proširuje objekat u listu parametara
                        // zatim vraća rezultat u nov objekat

// Da li objekti imaju isti sadržaj?
alert(JSON.stringify(obj) === JSON.stringify(objCopy)); // true

// Da li su objekti isti?
alert(obj === objCopy); // false (nisu iste reference)

// modifikacija originalnog objekta ne modifičuje kopiju
obj.d = 4;
alert(JSON.stringify(obj)); // {"a":1,"b":2,"c":3,"d":4}
alert(JSON.stringify(objCopy)); // {"a":1,"b":2,"c":3}
```

Ovakav vid kopiranja objekata je značajno kraći od sledećih oblika:

```
let objCopy = Object.assign({}, obj); //za objekte
let arrCopy = Object.assign([], arr); //za nizove
```

pa je poželjno da se koristi kada god je to moguće.

DEKLARISANJE PROMENLJIVIH - REZIME

Deklarisanje promenljivih primenom ključnih reči var, let i const.

U prethodnoj lekciji pokazano je kako se deklarišu varijable primenom ključne reči `var`. Generalno, postoje tri načina za deklariranje varijabli u JavaScript jeziku:

1. `var`;
2. `let`;
3. `const`.

Ključne reči `let` i `const` poseduju identično ponašanje sa razlikom što je `const` zadužen za deklarisanje promenljivih kojima kada se jednom dodeli vrednost to nije moguće uraditi naknadno.

Generalno `var` datira iz ranijih verzija *JavaScript* jezika i otuda je ovoj rezervisanoj reči prvo posvećena pažnja. Generalno, u novijim skriptovima se sve manje ili veoma retko pojavljuje. Međutim, bitna je iz razloga koji se u savremenim računarskim, i srodnim, naukama naziva kompatibilnost unazad.

Prva razlika između `var` i `let` jeste u oblasti primene / važenja. Umesto fokusiranja na funkcije, rezervisana reč `let` je fokusirana na blokove. To praktično znači da je promenljiva kreirana primenom rezervisane reči `let` dostupna unutar bloka, u kojem je kreirana, kao i unutar svih ugnezdenih blokova unutar pomenutog bloka. Kada se kaže "blok" misli se na kod koji je obuhvaćen vitičastim zagradama `{}` - kao u petljama ili `if` iskazima.

Posmatra se scenario deklarisanja promenljivih priložen sledećom slikom:

```
function discountPrices (prices, discount) {
  var discounted = []

  for (var i = 0; i < prices.length; i++) {
    var discountedPrice = prices[i] * (1 - discount)
    var finalPrice = Math.round(discountedPrice * 100) / 100
    discounted.push(finalPrice)
  }

  console.log(i) // 3
  console.log(discountedPrice) // 150
  console.log(finalPrice) // 150

  return discounted
}

function discountPrices (prices, discount) {
  let discounted = []

  for (let i = 0; i < prices.length; i++) {
    let discountedPrice = prices[i] * (1 - discount)
    let finalPrice = Math.round(discountedPrice * 100) / 100
    discounted.push(finalPrice)
  }

  console.log(i) // 3
  console.log(discountedPrice) // 150
  console.log(finalPrice) // 150

  return discounted
}

discountPrices([100, 200, 300], .5) // ✘ ReferenceError: i is not defined
```

Slika 2.6 Dva scenarija deklarisanja promenljivih [izvor: autor]

Zašto se u drugom slučaju javlja greška tipa: "**Reference Error: i is not defined**"? Razlog je u tome što se promenljiva definisana primenom ključne reči *let* (promenljiva *i* je upravo tako definisana u petlji *for* drugog primera) nije vidljiva izvan bloka u kojem je kreirana.

Postoji još jedna razlika između *var* i *let* i o tome upravo govori naredno izlaganje.

DEKLARISANJE PROMENLJIVIH - REZIME (ZAVRŠNA RAZMATRANJA)

Promenljivoj deklarisanoj sa const nije moguće pridružiti novu vrednost.

Posmatra se sledeći scenario. Deklarisana je promenljiva *discounted* primenom ključne reči *var*. Međutim, ako se pogleda prvi listing sa slike 7, poziv ovakve promenljive pre sopstvene deklaracije će vratiti rezultat *undefined*.

Ako se diskutuje iz perspektive savremene programerske prakse, **teško je zamisliti situaciju u kojoj se promenljiva poziva pre nego što je deklarisana**. Otuda bi bilo bolje da umesto *undefined*, ovakav poziv vrati "*ReferenceError*". Upravo to radi rezervisana reč *let*, a to je pokazano drugim listingom sa slike 7.

```
function discountPrices (prices, discount) {
  console.log(discounted) // undefined

  var discounted = []

  for (var i = 0; i < prices.length; i++) {
    var discountedPrice = prices[i] * (1 - discount)
    var finalPrice = Math.round(discountedPrice * 100) / 100
    discounted.push(finalPrice)
  }

  console.log(i) // 3
  console.log(discountedPrice) // 150
  console.log(finalPrice) // 150

  return discounted
}

function discountPrices (prices, discount) {
  console.log(discounted) // ✘ ReferenceError

  let discounted = []

  for (let i = 0; i < prices.length; i++) {
    let discountedPrice = prices[i] * (1 - discount)
    let finalPrice = Math.round(discountedPrice * 100) / 100
    discounted.push(finalPrice)
  }

  console.log(i) // 3
  console.log(discountedPrice) // 150
  console.log(finalPrice) // 150

  return discounted
}
```

Slika 2.7 Dodatno poređenje var i let [izvor: autor]

Konačno, potrebno je zaokružiti i diskusiju koja se tiče razlike između *let* i *const*. Ključne reči kada definišu promenljive pokazuju potpuno identično ponašanje sve do slučaja kada je promenljivoj deklarisanoj sa *const* potrebno pridružiti novu vrednost. **Ovo nije moguće!**

Razlika između *let* i *const* je ilustrovana sledećom slikom.

```
let name = 'Petar'  
const handle = 'petarpetrovic'  
  
name = 'Petar Petrovic' // ✓  
handle = '@petarpetrovic' // ✗ TypeError: Assignment to constant variable.
```

Slika 2.8 Razlika između let i const - ilustracija [izvor: autor]

Ova lekcija sadrži video materijal. Ukoliko želite da pogledate ovaj video morate da otvorite LAMS lekciju.

NAPREDNI RAD SA FUNKCIJAMA (VIDEO MATERIJAL)

Diskusija rada sa funkcijama u JS jeziku zaokružena je preko adekvatnog video materijala.

Advanced JavaScript - Module 01 - Part 07 - Array functions: Push, Pop, Shift & Unshift (trajanje 19:30).

Ova lekcija sadrži video materijal. Ukoliko želite da pogledate ovaj video morate da otvorite LAMS lekciju.

Advanced JavaScript - Module 02 - Part 01 - Functions, Anonymous, FE, IIFE, Callbacks (trajanje: 18:41).

Ova lekcija sadrži video materijal. Ukoliko želite da pogledate ovaj video morate da otvorite LAMS lekciju.

Advanced JavaScript - Module 02 - Part 02 - Function Expressions (FE) demonstration (trajanje: 5:22).

Ova lekcija sadrži video materijal. Ukoliko želite da pogledate ovaj video morate da otvorite LAMS lekciju.

▼ Poglavlje 3

Podešavanje osobina objekata u JavaScript - u

NAPREDNI KONCEPTI PODEŠAVANJA OBJEKATA U JAVASCIPT JEZIKU

Proširivanje osnovnog znanja rada sa objektima u JS.

U nastavku lekcije fokus će biti na širenju znanja u vezi sa objektima JavaScript jezika, stečenog u prethodnoj i uvodnim delovima ove lekcije. Ovaj deo lekcije će biti podeljen na dva dela:

1. oznake (flags) i deskriptori osobina objekata;
2. seteri i geteri osobina objekata.

Upravo ove teme će biti razmatrane u nastavku kao posebne celine u lekciji.

▼ 3.1 Oznake (flags) i deskriptori osobina objekata

OZNAKE OBJEKATA (FLAGS)

Zadatak objekata je da čuvaju osobine koje čine neku logičku celinu.

Kao što je više puta istaknuto, zadatak objekata je da čuvaju osobine koje čine neku logičku celinu. Sve do sada, osobine objekata su razmatrane u formi parova (ključ, vrednost). Međutim, objektna osobina je značajno fleksibilniji i moćniji koncept.

Cilj ovog dela lekcije jeste analiza, diskusija i demonstracija dopunskih opcija za podešavanje objekata. Prva od njih predstavlja oznaka (flag) objektne osobine.

Objektna osobina, pored vrednosti (value) može da poseduje i jednu od sledeće tri oznake (specijalne atribute):

- writable – ukoliko je podešena na true, vrednost može da bude promenjena, u suprotnom je moguće samo čitanje vrednosti (read-only);
- enumerable – ukoliko je podešena na true, osobina je dodata u listu i moguće je pronaći je kroz iteraciju osobina, u suprotnom nije dodata u listu osobina objekta;

- **configurable** – ukoliko je podešena na true, osobina može biti brisana ili modifikovana, u suprotnom to nije moguće.

DESKRIPTOR

Deskriptor omogućava dobijanje potpune informacije u vezi sa nekom objektnom osobinom.

Navedene oznake nisu još razmatrane, budući da se, praktično, nisu ni pojavljivale. Kada se kreira osobina na "uobičajeni način", svaka od njih je podešena na **true**. Međutim, ove vrednosti je moguće promeniti u bilo kojem trenutku..

Za početak, neophodno je pokazati kako je moguće doći do navedenih oznaka. Metoda **Object.getOwnPropertyDescriptor()** omogućava kreiranje upita za dobijanje potpune informacije u vezi sa nekom objektnom osobinom.

Ovakva informacija se čuva u promenljivoj koja se naziva **deskriptor** (descriptor) i njena osnovna sintaksa glasi:

```
let descriptor = Object.getOwnPropertyDescriptor(obj, propertyName);
```

Iz definicije deskriptora neophodno je izolovati sledeće:

- **obj** - objekat iz kojeg se dobija informacija;
- **propertyName** - naziv osobine za koju se traži informacija.

Deskriptor, kao povratna vrednost navedene funkcije, sadrži vrednost i sve oznake vezane za datu osobinu.

Pokažimo to na sledećem primeru:

```
let user = {  
    name: "Petar"  
};  
  
let descriptor = Object.getOwnPropertyDescriptor(user, 'name');  
  
alert( JSON.stringify(descriptor, null, 2 ) );  
/* deskriptor osobine:  
{  
    "value": "Petar",  
    "writable": true,  
    "enumerable": true,  
    "configurable": true  
}  
*/
```

MODIFIKOVANJE OZNAKA OSOBINA

Modifikovanje oznaka osobina moguće je rešiti programski ukoliko je to neophodno.

U prethodnom izlaganju je pokazano kako se dodeljuju oznake kreiranim osobinama nekog objekta. Ukoliko je cilj izmena vrednosti oznake moguće je upotrebiti sintaksu:

```
Object.defineProperty(obj, propertyName, descriptor)
```

Ukoliko osobina postoji, metoda `defineProperty()` ažurira sve njene oznake. U suprotnom, kreiraće novu osobinu sa datom vrednošću i oznakom. U ovom slučaju, ukoliko vrednost oznake nije zadata, podrazumevano će biti `false`.

Na primer, na sledeći način se objektu `user` dodaje osobina `name` čije će sve oznake biti podešene na `false`:

```
let user = {};  
  
Object.defineProperty(user, "name", {  
    value: "Petar"  
});  
  
let descriptor = Object.getOwnPropertyDescriptor(user, 'name');  
  
alert( JSON.stringify(descriptor, null, 2) );  
/*  
{  
    "value": "Petar",  
    "writable": false,  
    "enumerable": false,  
    "configurable": false  
}  
*/
```

U odnosu na "normalno" kreiranu osobinu (prethodno izlaganje) sve osobine u ovom slučaju su podešene na `false`. Ukoliko to nije ono što želimo, neophodno je pokazati kako je to moguće promeniti.

EFEKTI PRIMENE OZNAKA : NON - WRITABLE I NON - ENUMERABLE

Sledi diskusija u vezi sa efektima oznaka na osobinu kojoj pripadaju.

Sledi diskusija u vezi sa efektima oznaka na osobinu kojoj pripadaju.

Non - writable:

Podešavanjem osobine na `writable` na `false` sprečava se promena vrednosti osobine. To je u JavaScript - u moguće učiniti na sledeći način:

```
let user = {
    name: "Petar"
};

Object.defineProperty(user, "name", {
    writable: false
});

user.name = "Marko"; // GREŠKA! Nije moguća dodata za "read only" osobinu 'name'
```

Non - enumerable:

Dodaje se nova osobina o objekat `user` na sledeći način:

```
let user = {
    name: "Petar",
    toString() {
        return this.name;
    }
};

// Po osnovnim podešavanjima, obe osobine su dodate u listu:
for (let key in user) alert(key); // name, toString
```

Ukoliko se postupi na sledeći način:

```
let user = {
    name: "Petar",
    toString() {
        return this.name;
    }
};

Object.defineProperty(user, "toString", {
    enumerable: false
});

// Osobina toString izbačena je iz liste osobina:
for (let key in user) alert(key); // name
```

Podešavanjem označe `enumerable : false` dodata osobina je izbačena iz liste osobina i iteracijom nije moguće doći do nje. Takođe, to je slučaj i sa njenom vrednošću:

```
alert(Object.keys(user)); // name
```

EFEKTI PRIMENE OZNAKA : NON - CONFIGURABLE

Neizmenljiva oznaka je često podešena za ugrađene objekte i osobine.

Neizmenljiva oznaka je često podešena za ugrađene objekte i osobine. **Ovako podešenu osobinu nije moguće obrisati.**

Na primer, `Math.PI` nije upisiva (*non-writable*), nabrojiva (*non-enumerable*) i podešiva (*non-configurable*).

```
let descriptor = Object.getOwnPropertyDescriptor(Math, 'PI');

alert( JSON.stringify(descriptor, null, 2) );
/*
{
  "value": 3.141592653589793,
  "writable": false,
  "enumerable": false,
  "configurable": false
}
*/
```

Ukoliko programer pokuša sledeću izmenu:

```
Math.PI = 3; // GREŠKA!!!

// brisanje (delete) Math.PI takođe nije moguće
```

desiće se greška. Razlog je što nije moguće promeniti vrednost osobine čija je oznaka *configurable* podešena na `false`.

Obeležavanje osobine kao nepromenljive (*non-configurable*) je jednosmerna aktivnost i nisu dozvoljene promene unazad sa `defineProperty()`.

Za rezime, podešavanjem oznake `configurable : false` dobijaju se sledeća ograničenja za metodu `defineProperty()`:

1. Nije moguće izmeniti oznaku *configurable*;
2. Nije moguće izmeniti oznaku enumerable *flag*;
3. Nije moguće izmeniti oznaku writable: `false` ili `true`;
4. Nije moguće izmeniti `get/set` za pristupajuću osobinu (ali ih je moguće dodeliti ukoliko ne postoje).

Na sledeći način je osobina `user.name` zauvek "zapečaćena" kao konstantna:

```
let user = { };

Object.defineProperty(user, "name", {
  value: "Petar",
  writable: false,
  configurable: false
});

// nije moguće menjanje osobine user.name ili njenih oznaka
// sve navedeno NE RADI!!!!:
//   user.name = "Marko"
//   delete user.name
//   defineProperty(user, "name", { value: "Marko" })
Object.defineProperty(user, "name", {writable: true}); // GREŠKA!!!
```

3.2 Seteri i geteri osobina objekata

NAČINI PRISTUPA OSOBINAMA

Postoje dva načina pristupa osobinama - direktno obraćanje osobini i preko set / get funkcija.

Postoje dva načina pristupa osobinama objekata:

- direktni, primenom sintakse "tačka";
- pristup preko `set` i `get` funkcija.

Drugi način podrazumeva postojanje dveju ključnih funkcija čiji je zadatak preuzimanje vrednosti (`get`) iz promenljive i dodela nove vrednosti promenljivoj (`set`).

Ovakav način rada sa osobinama moguće je predstaviti u sledećem opštem obliku:

```
let obj = {
    get propName() {
        // getter, vraća vrednost za obj.propName
    },
    set propName(value) {
        // setter, dodeljuje novu vrednost za obj.propName = value
    }
};
```

PRIMENA FUNKCIJA SET I GET U OBJEKTIMA

Funkcija get čita vrednost osobine, a set je podešava.

Jednostavno rečeno, funkcija `get` čita vrednost osobine, a `set` je podešava. Za ilustraciju moguće je angažovati postojeći primer. Posmatra se sledeći kod:

```
let user = {
    name: "Petar",
    surname: "Petrovic"
};
```

Kreiran je objekat `user` sa osobinama `name` i `surname` kojima su pridružene vrednosti.

Sada je cilj dodavanje nove objektne osobine `fullName` čija će vrednost biti kombinacija osobina `name` i `surname` (u konkretnom slučaju "Petar Petrović").

Prethodni kod se modifikuje na sledeći način:

```
let user = {
    name: "Petar",
```

```
surname: "Petrovic",

get fullName() {
    return `${this.name} ${this.surname}`;
}

alert(user.fullName); // REŠENJE: Petar Petrovic
```

Na ovaj način osobina ima pridruženu vrednost koja može biti prikazana na ekranu. Međutim, pokušajmo sada da promenimo vrednost osobini *fullName*.

```
let user = {
    name: "Petar",
    surname: "Petrovic",

    get fullName() {
        return `${this.name} ${this.surname}`;
    }
};

user.fullName = "Marko"; // GREŠKA!!! (osobina poseduje samo getter)
```

Bez funkcije za podešavanje vrednosti osobine, ovakvoj osobini nije moguće promeniti vrednost i javlja se greška u liniji koda 10. Neophodno je dodati *set* funkciju na sledeći način i greška se neće javiti:

```
let user = {
    name: "Petar",
    surname: "Petrovic",

    get fullName() {
        return `${this.name} ${this.surname}`;
    },

    set fullName(value) {
        [this.name, this.surname] = value.split(" ");
    }
};

user.fullName = "Marko Markovic";

alert(user.name); // Marko
alert(user.surname); // Markovic
```

▼ Poglavlje 4

Prototipovi, nasleđivanje

PROTOTIPSKO NASLEĐIVANJE

Prototipsko nasleđivanje je mehanizam koji omogućava izgradnju novog objekta na vrhu postojećeg.

U programiranju je često potrebno preuzeti neki osnovni koncept i proširiti ga ([extend](#)). Na primer, posmatra se dobro poznati objekat `user` da svojim osobinama i metodama. Cilj je kreiranje specijalnih i blago modifikovanih varijanti ovog objekta `admin` i `guest`. Dobra praksa je višestruko iskorišćenje onoga što je već kreirano, a to znači, u konkretnom slučaju, izgradnja novog objekta na vrhu postojećeg, bez kopiranja i ponovnog implementiranja postojećih metoda.

U [JavaScript](#) jeziku prototipsko nasleđivanje je mehanizam koji omogućava navedeno.

`[[Prototype]]`

U [JavaScript](#) jeziku objekti poseduju posebnu skrivenu osobinu `[[Prototype]]` koja može biti `null` ili da referencira na drugi objekat. Taj objekat se naziva prototipom ([prototype](#)).



Slika 4.1 Prototipsko nasleđivanje [izvor:javascript.info]

U [JavaScript](#) jeziku ne postoji klasično nasleđivanje, kao što je slučaj sa programskim jezikom [Java](#). Ovde to funkcioniše na malo drugačiji način. Kada je potrebno pročitati osobinu objekta, a ona nedostaje, [JavaScript](#) će automatski preuzeti tu osobinu iz prototipa. U programiranju ovo je poznato kao [prototipsko nasleđivanje](#).

Osobina `[[Prototype]]` je unutrašnja i skrivena, međutim postoje načini da se ona podesi. Jedan od njih je primena specijalnog naziva `__proto__`, na sledeći način:

```
<script>  
let user = {
```

```
name : "",  
password : ""  
};  
let admin = {  
    access: true  
};  
  
admin.name = "admin";  
admin.password = "admin";  
  
// sve tri osobine sada postoje u objektu admin:  
alerts("Username : " + admin.name ); // admin (**)  
alert("Password : " + admin.password ); // admin  
alert("Admin role : " + admin.access ); //true  
admin.__proto__ = user;  
  
</script>
```

PODEŠAVANJE PROTOTIPA

Objekat user je prototip za admin objekat.

Ako se pogleda prethodni listing tada je moguće reći da je `user` prototip za `admin` objekat. To je podešeno kodom koji je na sledećoj slici obeležen plavom bojom.

```
let user = {  
    name : "",  
    password : ""  
};  
let admin = {  
    access: true  
};  
  
admin.name = "admin";  
admin.password = "admin";  
  
// sve tri osobine sada postoje u objektu admin:  
alerts("Username : " + admin.name ); // admin (**)  
alert("Password : " + admin.password ); // admir(**)  
alert("Admin role : " + admin.access ); //true  
admin.__proto__ = user;
```

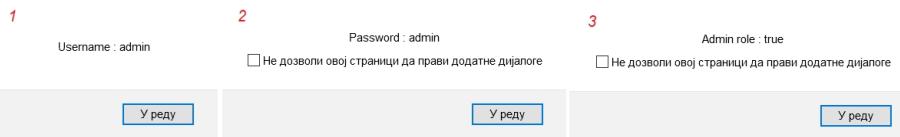
Slika 4.2 Podešavanje prototipa [izvor: autor]

Takođe, kada metoda `alert()` pokuša da čita osobine `name` i `password`, koje nisu definisane u objektu `admin` (istaknuto crvenom bojom na slici), `JavaScript` će pratiti `[[Prototype]]` referencu i pronaći ove osobine u objektu `user`. Vizuelno to može biti pokazano na sledeći način:



Slika 4.3 Referenciranje prototipa [izvor: autor]

Učitavanjem HTML stranice, sa priloženim skriptom, unutar veb pregledača dobija se izlaz kao na sledećoj slici:



Slika 4.4 Demonstracija prototipskog nasleđivanja [izvor: autor]

PODEŠAVANJE PROTOTIPA U OBJEKTU

Ukoliko prototip poseduje korisne osobine i metode, one automatski postaju dostupne u nasledniku.

Ukoliko prototip poseduje brojne i korisne osobine i metode, one automatski postaju dostupne u nasledniku. Ovakve osobine nazivamo nasleđenim osobinama.

Ukoliko `user` poseduje metodu (linija koda 4 - 7) ona automatski može biti pozvana `admin` objektom (linija koda 18).

```

let user = {
    name: "",
    password: "",
    prikazi(){
        alert("Username : " + this.name );
        alert("Password : " + this.password );
    }
};

let admin = {
    access: true,
    __proto__: user
};

admin.name = "admin";
admin.password = "admin";
    
```

```
admin.prikazi();
alert("Admin role : " + admin.access ); //true
```

Takođe, listing pokazuje još jednu mogućnost podešavanja prototipa unutar objekta koji ga nasleđuje (linija koda 12). Na ovaj način je omogućeno **prototipsko ulančavanje** jer na potpuno isti način i admin može da ima naslednika. Objekat `admin2` direktno nasleđuje prototip `admin` i indirektno njegov prototip `user`.

```
let user = {
  name: "",
  password: "",
  prikazi(){
    alert("Username : " + this.name );
    alert("Password : " + this.password );
  }
};

//user je prototip za admin
let admin = {
  access: true,
  __proto__: user
};

admin.name = "admin";
admin.password = "admin";

//admin je prototip za admin2
let admin2= {
  deleteUsers: true,
  __proto__: admin
};
```

OGRANIČENJA PROTOTIPSKOG NASLEĐIVANJA

Prototipske reference ne mogu biti ciklične.

Iako veoma korisna JavaScript funkcionalnost, prototipsko nasleđivanje ima i određena ograničenja po pitanju upotrebe:

- 1. Reference ne mogu biti ciklične.** Ukoliko bi neki objekat u prototipskom lancu postao prototip za objekat koji je na višem nivou u prototipskoj hijerarhiji, *JavaScript* bi izbacio grešku;
2. Vrednost za `__proto__` je ili `null` ili je objekat. Ostale vrednosti se ignorisu;
- 3. Postoji samo jedan prototip** - objekti ne mogu da nasleđuju iz dva i više prototipova;
4. Operacije `write / delete` rade direktno sa objektima i ne koriste prototip.

Posmatra se sledeći kod:

```

let user = {
    name: "",
    password: "",
    prikazi(){
        //ovu metodu neće koristiti admin
    }
};

let admin = {
    access: true,
    __proto__: user
};

admin.name = "admin";
admin.password = "admin";

//admin dobija vlastitu f-ju prikazi()
admin.prikazi = function(){

    alert("Username : " + admin.name );
    alert("Password : " + admin.password );
    alert("Admin role : " + admin.access ); //true
};

admin.prikazi();

```

Iz listinga se primećuje da je *admin* dobio vlastitu metodu *prikazi()*. Ova metoda se poziva u liniji koda 26. Ovakav poziv nadjačava poziv prototipske metode i poziva se direktno iz objekta bez upotrebe prototipa.

Osobine *set* i *get* su izuzetak, budući da je dodata vrednosti upravljana *setter* metodom. Otuda, upisivanje u ovaku osobinu je identično pozivu odgovarajuće metode.

Iz navedenog razloga sledeći kod funkcioniše ispravno:

```

let user = {
    name: "",
    password: "",
    prikazi(){
        //ovu metodu neće koristiti admin
    },
    set fullName(value) {
        [this.name, this.password] = value.split(" ");
    },
    get fullName() {
        return `${this.name} ${this.password}`;
    }
};

let admin = {
    access: true,
    __proto__: user
};

admin.name = "admin";
admin.password = "admin";

admin.prikazi = function(){

    alert("Username : " + this.name );
    alert("Password : " + this.password );
    alert("Admin role : " + admin.access ); //true
    alert("Username & Password : " + admin.fullName);
};

admin.prikazi();
admin.fullName = "user 1234";
admin.prikazi();

```

Slika 4.5 Upisivanje u osobinu preko prototipskog setter-a [izvor: autor]

Rezervisanom rečju this se zamjenjuje objekat koji poziva metodu.

PROTOTIPSKO NASLEĐIVANJE (VIDEO MATERIJAL)

JavaScript Prototypal Inheritance (trajanje 10:48)

Ova lekcija sadrži video materijal. Ukoliko želite da pogledate ovaj video morate da otvorite LAMS lekciju.

▼ Poglavlje 5

JavaScript klase

SINTAKSA KLASE U JAVASCRIPT-U

U praksi je često potrebno kreirati veći broj objekata istog tipa.

U praksi je često potrebno kreirati veći broj objekata istog tipa. U programiranju, definisanje ovakvih objekata je u domenu koncepta koji se naziva klasa (class).

U najjednostavnijem slučaju, funkcijom koja se naziva konstruktor i primenom operatora new jednostavno se kreira objekat neke klase. U savremenom JavaScript jeziku postoje naprednije uloge koje konstruktori mogu da obave.

Osnovna sintaksa klase u JavaScript jeziku prikazana je sledećim listingom:

```
class MyClass {  
    // metode klase  
    constructor() { ... }  
    method1() { ... }  
    method2() { ... }  
    method3() { ... }  
    ...  
}
```

Vodeći se osnovnom sintaksom, sada je moguće kreirati konkretnu klasu i njene objekte. Metoda constructor() će automatski biti pozvana operatorom new pa na ovom mestu objekat može da bude inicijalizovan.

Na primer:

```
class User {  
  
    constructor(name) {  
        this.name = name;  
    }  
  
    pozdrav() {  
        alert("Pozdrav " + this.name);  
    }  
  
}  
  
// Usage:  
let user = new User("Petar");  
user.pozdrav();
```

Klasom `User` moguće je sada kreirati veći broj objekata koji mogu po potrebi da pozivaju metodu `pozdrav()` koja je definisana u klasi. Poziv konstruktora i kreiranje novog objekta desio se u liniji koda 15.

Klasa je šablon za kreiranje objekara u JavaScript-u.

SPECIFIČNOSTI JAVASCRIPT KLASA

JavaScript klase su, u suštini, funkcije.

Klasa je potpuno nov `JavaScript` koncept i prisutan je od uvođenja standarda `ES6`. JavaScript, u suštini, tumači klasu kao funkciju sa razlikom što ne koristi rezervisanu reč `function`, za njenu inicijalizaciju, već reč `class`. Osobine klase su dodeljene unutar metode `constructor()`.

Konstruktor je specijalna metoda u kojoj se objekat inicijalizuje. Automatski se poziva kada se kreira objekat klase i mora da poseduje tačno određen naziv "`constructor`". Zapravo, čak i da nije eksplicitno naveden u definiciji klase, JavaScript će dodati nevidljivu i praznu konstruktorsku metodu.

UPOTREBA STATIČKIH METODA

Statička metoda je definisana za klasu i nije primenljiva za prototip.

Statička metoda je definisana za klasu i nije primenljiva za prototip. To praktično znači da nije moguće pozvati statičku metodu za objekat (`user`). Za njen poziv je zadužena direktno klasa. Za deklarisanje statičke metode, koristi se rezervisana reč `static`.

Navedeno je ilustrovano sledećim primerom:

```
class User {  
  
    constructor(name) {  
        this.name = name;  
    }  
  
    static pozdrav() {  
        alert("Pozdrav!!!");  
    }  
  
}  
  
// Usage:  
let user = new User("Petar");  
user.pozdrav(); // Greška - objekat ne može da pozove statičku metodu  
User.pozdrav(); // Ispravno - klasa poziva statičku metodu
```

U liniji koda 7 je definisana metoda `pozdrav()` klase `User` kao statička. U linijama koda 14 i 15 je obavljen poziv ove metode. U prvom slučaju će poziv rezultovati greškom iz razloga što je nemoguće objektom pozvati statičku metodu. Drugi poziv je korektan.

NASLEĐIVANJE KLASA

Nasleđivanje u JavaScript jeziku omogućava nasleđivanje metoda iz neke druge klase.

Nasleđivanje je veoma bitan koncept objektno - orijentisanog programiranja. Iako ima limite u odnosu na nasleđivanje kod pravih objektno - orijentisanih programske jezika, poput Java, nasleđivanje u JavaScript jeziku je značajno iz razloga što omogućava nasleđivanje metoda iz neke druge klase - roditeljske klase.

Za implementaciju nasleđivanja koristi se rezervisana reč `extends`. Na postojećem primeru biće urađena modifikacija sa ciljem demonstracije nasleđivanja klasa u jeziku `JavaScript`. Konstruktor superklase je određen rezervisanim rečju `super`.

```
<script>

class User {
    constructor(name, password) {
        this.name = name;
        this.password = password;
    }
    prikazi() {
        return 'Podaci : ' + this.name + " " + this.password;
    }
}

class Admin extends User {
    constructor(name, password, isAdmin) {
        super(name, password);
        this.isAdmin = isAdmin;
    }
    prikaziAdmin() {
        return this.prikazi() + ', administrator : ' + this.isAdmin;
    }
}

admin = new Admin("admin", "admin", true);
document.getElementById("demo").innerHTML = admin.prikaziAdmin();

</script>
```

Klasa `Admin` nasleđuje iz klase `User`. Kroz nasleđivanje dobija konstruktor (`super`) roditeljske klase (linija koda 15), kao i metodu `prikazi()`. U liniji koda 19, potomak klasa koristi roditeljsku metodu u okviru definicije vlastite metode `prikaziAdmin()` (linije koda 18 - 21).

Kreiranje objekta admin i poziv njegove metode desio se u liniji koda 24. Da bi ovaj rezultat mogao da bude prikazan kreiran je HTML element `<p>` čiji id mora da odgovara reči "*demo*". Dodate su još neke HTML linije za dodatne informacije:

```
<h2>Primer nasleđivanja klasa</h2>

<p>Rezervisana reč "extends" se koristi za nasleđivanje metoda iz superklase.</p>
<p>Rezervisana reč "super" predstavlja konstruktor roditeljske klase.</p>

<p id="demo"></p>
```

Učitavanjem kreirane stranice u pregledač, dobija se sledeći izlaz:

Primer nasleđivanja klasa

Rezervisana reč "extends" se koristi za nasleđivanje metoda iz superklase.

Rezervisana reč "super" predstavlja konstruktor roditeljske klase.

Podaci :admin admin, administrator : true

Slika 5.1 Nasleđivanje klasa - demonstracija [izvor: autor]

SETTERI I GETTERI KLASA

Klase dozvoljavaju primenu set i get metoda.

Klase dozvoljavaju primenu `set` i `get` metoda. Ovo je veoma dobro u slučajevima kada je neophodno uraditi nešto sa vrednostima osobina pre njihovog vraćanja ili podešavanja.

Za dodavanje navedenih metoda u klasu koriste se ključne reči `set` i `get`. To može biti ilustrovano sledećim primerom:

```
class User {
    constructor(name, password, status) {
        this.name = name;
        this.password = password;
        this.status = status;
    }

    get usStatus() {
        return this.status;
    }
    set usStatus(x) {
        this.status = x;
    }

    prikazi() {
```

```
        return 'Podaci : ' + this.name + " " + this.password + " " + this.usStatus;
    }
}

class Admin extends User {
    constructor(name, password, status, isAdmin) {
        super(name, password, status);
        this.usStatus = "";
        this.st = this.status;
        this.isAdmin = isAdmin;
    }
    prikaziAdmin() {
        return this.prikazi() + ', administrator : ' + this.isAdmin;
    }
}

admin = new Admin("admin", "admin", "neaktivna", true);
admin.status = "aktivna";
document.getElementById("demo").innerHTML = admin.prikaziAdmin();
```

Primerom je demonstrirana upotreba set i get metoda za manipulaciju osobinom status. Metode su definisane u roditeljskoj klasi *User* u linijama koda 10 i 13, respektivno.

Klasa Admin nasleđuje ove metode i primenjuje ih redom u linijama 25 i 35 (*set*), kao i liniji koda 18 (*get*).

Sledećom slikom je prikazan izgled stranice u čijem kreiranju je primenjen prethodni JS kod.

Primer nasleđivanja klasa

Rezervisana reč "extends" se koristi za nasleđivanje metoda iz superklase.

Rezervisana reč "super" predstavlja konstruktor roditeljske klase.

Podaci :admin admin aktivna, administrator : true

Slika 5.2 Setteri i Getteri klasa - demonstracija [izvor: autor]

VAŽNO!!! Za razliku od funkcija i ostalih deklaracija, u JavaScript jeziku, klase moraju da budu deklarisane pre njihove prve upotrebe.

JAVASCRIPT KLASE (VIDEO MATERIJAL)

JavaScript ES6 / ES2015 - [04] Classes and Inheritance (trajanje 7:36)

Ova lekcija sadrži video materijal. Ukoliko želite da pogledate ovaj video morate da otvorite LAMS lekciju.

▼ Poglavlje 6

Rukovanje greškama

RAZLOZI JAVLJANJA GREŠAKA U SKRIPTOVIMA

Greške u skriptovima mogu biti rezultat brojnih očekivanih ili neočekivanih razloga,

Bez obzira koliko je neko iskusan i dobar programer, greške u skriptovima se ponekad dešavaju. **Greška** može biti rezultat:

- ljudskog faktora - greška u kodu sintaksnog ili logičkog tipa;
- neočekivanih ulaznih vrednosti;
- grešaka u odgovoru servera;
- brojnih drugih očekivanih ili neočekivanih razloga.

U najvećem broju slučajeva **skript** se u potpunosti zaustavi prilikom javljanja greške i odgovarajuća poruka o grešci se prikazuje u konzoli veb pregledača.

Kada je rad sa greškama u pitanju **JavaScript** predlaže upotrebu konstrukcije **try..catch** koja omogućava hvatanje greške i njenu obradu na način da skript uradi nešto odgovarajuće, umesto da otkaže.

SINTAKSA TRY / CATCH U JAVASCRIPT JEZIKU

JavaScript koristi konstrukciju try..catch koja omogućava hvatanje greške i njenu obradu.

Kao što je istaknuto u prethodnom izlaganju, **JavaScript** predlaže upotrebu konstrukcije **try..catch** koja omogućava hvatanje greške i njenu obradu na način da skript uradi nešto odgovarajuće, umesto da otkaže.

Sledećim listingom je prikazan opšti oblik ove naredbe:

```
try {  
    // kod...  
}  
    catch (err) {  
        // rukovanje greškama
```

}

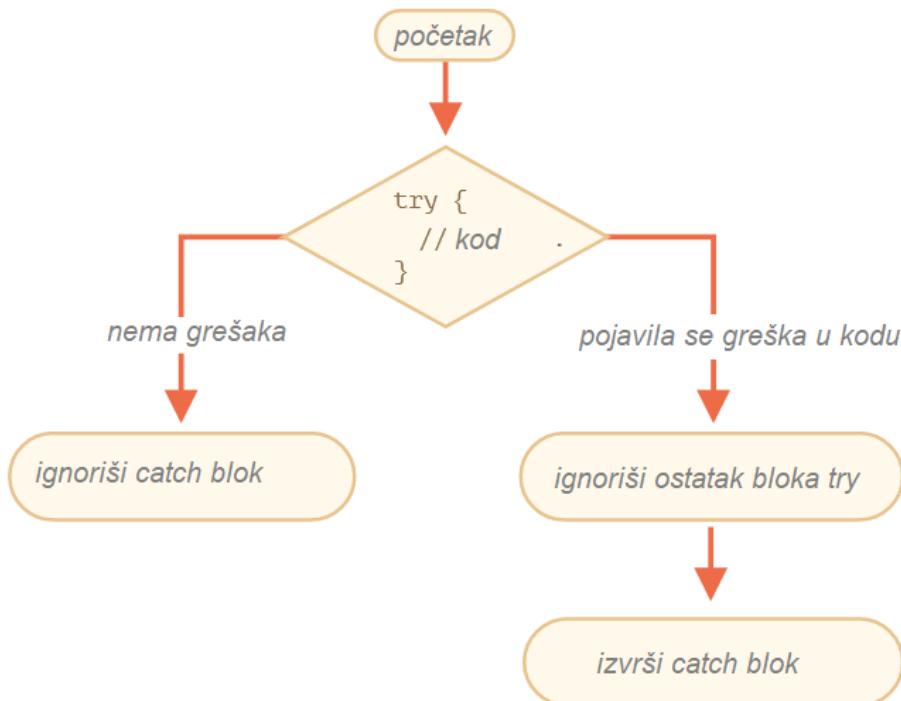
Navedeni kod funkcioniše po sledećem scenariju:

1. izvršava se kod u `try{...}` bloku;
2. ukoliko u kodu nema grešaka, `catch(err)` se ignoriše i izvršavanje koda se nastavlja kodom koji se nalazi odmah iza ovog bloka;
3. ukoliko se u `try{...}` bloku javi greška, ostatak koda iz ovog bloka se ignoriše i odmah se prelazi na izvršavanje bloka `catch(err)` za obradu greške.

Promenljiva `err` (moguće je koristiti bilo koji drugi naziv) sadrži objekat greške sa svim njegovim detaljima.

Na ovaj način je realizovana odlična savremena programerska praksa po kojoj se detekcija greške razdvaja od njene obrade.

Navedeno je moguće ilustrovati sledećom slikom.



Slika 6.1 Sintaksa try / catch u JavaScript jezik - funkcionisanje [izvor: autor]

SINTAKSA TRY / CATCH - PRIMER

Try blok može da objavi veći broj različitih tipova grešaka.

Za demonstraciju obrade greške u JavaScript kodu prilaže se sledeći listing:

```
<!DOCTYPE html>
<html>
<body>
```

```
<p>Unesite broj između 5 i 10:</p>

<input id="demo" type="text">
<button type="button" onclick="myFunction()">Test unosa</button>
<p id="greska"></p>

<script>
function myFunction() {
    var message, x;
    message = document.getElementById("greska");
    message.innerHTML = "";
    x = document.getElementById("demo").value;
    try {
        if(x == "") throw "prazno";
        if(isNaN(x)) throw "nije broj";
        x = Number(x);
        if(x < 5) throw "vrednost manja od dozvoljene";
        if(x > 10) throw "vrednost veća od dozvoljene";
    }
    catch(err) {
        message.innerHTML = "Greška je : " + err;
    }
}
</script>

</body>
</html>
```

Try blok može da objavi veći broj različitih tipova grešaka. Svaki od njih se hvata, kao objekat greške i obrađuje u *catch* bloku. Za objavljuvanje greške je, u *JavaScript* jeziku, zadužena naredba *throw* kojom se u objekat greške upisuje i dodatna informacija o grešci (linije koda 18 - 22). Navedena dodatna informacija može biti prikazana kao *message* objekat (linija koda 25).

Sledećom slikom je demonstrirano prikazivanje navedenog objekta greške na ekranu, u slučaju kada se javi tokom izvršavanja programa.

Unesite broj između 5 i 10:

25

Greška je : vrednost veća od dozvoljene

Slika 6.2 Sintaksa try / catch - primer [izvor: autor]

▼ Poglavlje 7

Objekti tipa Promise - async/await sintaksa

KONCEPT OBEĆANJA (PROMISES OBJEKATA)

Svako obećanje se sastoji iz tri stanja (states).

Koncept obećanja (promises) u JavaScript jeziku nije težak za primenu, međutim, u početku može biti poteškoća u njegovom razumevanju. Upravo iz navedenog razloga, cilj ovog dela lekcije jeste da studentima približi ovaj koncept na što lakši način. Veoma je bitan kao uvodno razmatranje za napredne Angular koncepte jer se na njima bazira nov koncept Observables karakterističan za TypeScript jezik u Angular okviru.

Za razumevanje pravi se paralela sa realnim životnim scenarijem: "Zamislite da ste dete. Roditelji su vam obećali nov mobilni telefon sledeće nedelje. Ne znate da li ćete dobiti taj telefon do sledeće nedelje. Roditelj može zaista kupiti potpuno novi telefon, ili neće, jer nije zadovoljan kako obavljate vaše obaveze".

Upravo ovo predstavlja obećanje. Svako obećanje se sastoji iz tri stanja (states). Navedena stanja su:

- **prosleđeno, nerešeno (pending)** - ne znate da li ćete dobiti telefon;
- **rešeno, ispunjeno (fullfilled)** - roditelji su zadovoljni, kupuju vam nov telefon;
- **odbijeno (rejected)** - ne ispunjavate vaše obaveze; roditelji su nezadovoljni i nećete dobiti nov telefon.

Kodiranje obećanja:

Ako se prethodna definicija prevede u JavaScript dobija se sledeći kod:

```
// Primer upotrebe Promise objekata

var roditeljZadovoljan = false;

// Promise
var dobicuTelefon = new Promise(
    function (resolve, reject) {
        if (roditeljZadovoljan) {
            var phone = {
                brand: 'Samsung',
                color: 'black'
            };
            resolve(phone); // rešeno
        } else {
```

```
        var reason = new Error('roditelji nisu zadovoljni');
        reject(reason); // odbijeno
    }

};

)
```

Kod sam po sebi sve govori. Iz listinga je moguće sagledati u kojem opštem obliku se javlja svaki *Promise* objekat:

```
new Promise(function (resolve, reject) { ... } );
```

UPOTREBA PROMISE OBJEKATA

Obećanje može biti rešeno ili odbijeno.

Nakon demonstracije kreiranja *Promise* objekata, neophodno je objasniti kako se oni koriste u *JavaScript* programima. Kao osnov za diskusiju biće upotrebljen primer započet u prethodnom izlaganju.

Na dnu već kreiranog skript koda dodaje se kod iz sledećeg listinga:

```
// poziv obećanja (Promise objekta)
var pitajRoditelja = function () {
    dobicuTelefon
        .then(function (fulfilled) {
            // da, dobićeš nov telefon
            console.log(fulfilled);
            // output: { brand: 'Samsung', color: 'black' }
        })
        .catch(function (error) {
            // nećeš dobiti nov telefon
            console.log(error.message);
            // output: 'roditelji nisu zadovoljni'
        });
};

pitajRoditelja();
```

Samo rešavanje problema obećanja zavisi od vrednosti promenljive *roditeljZadovoljan* koja može biti *true* ili *false*. U prvom slučaju, obećanje će biti rešeno (*fullfilled*) i u konzoli kao izlaz će biti prikazan objekat telefona kojeg će roditelji kupiti (slika 1). U suprotnom, obećanje će biti odbijeno (*reject*) i u konzoli će biti prikazana greška (razlog odbijanja - slika 2).



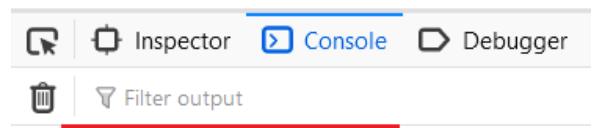
Pogledajte rezultat u konzoli pregledača !!!



Slika 7.1 Rešeno obećanje [izvor: autor]



Pogledajte rezultat u konzoli pregledača !!!



Slika 7.2 Odbijeno obećanje [izvor: autor]

ULANČAVANJE OBEĆANJA

Često se *Promise* objekti vezuju u tzv. lance obećanja.

Često se *Promise* objekti vezuju u tzv. lance obećanja. Da bi se ulančavanje pokazalo na primeru, neophodno je prethodni kod modifikovati po sledećem scenariju:

- opet se nalazite u ulozi deteta;
- pokazaćete drugu telefon ukoliko vam roditelj kupi.

```
// 2. obećanje
var pokazatiTelefon = function (phone) {
    return new Promise(
        function (resolve, reject) {
            var message = 'Imam novi telefon: ' +
                phone.color + ' ' + phone.brand + ' phone';
            resolve(message);
    }
}
```

```
 );  
};
```

Ovo je kraće moguće zapisati na sledeći način:

```
// 2nd promise  
var pokazatiTelefon = function (phone) {  
    var message = 'Imam novi telefon: ' +  
        phone.color + ' ' + phone.brand + ' phone';  
  
    return Promise.resolve(message);  
};
```

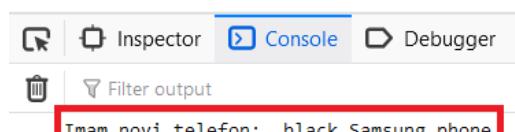
Sada je moguće obaviti ulančavanje kreiranih obećanja. Drugo obećanje se navodi odmah iza prvog, upravo kao na sledećoj slici:

```
// poziv obećanja (Promise objekta)  
var pitajRoditelja = function () {  
    dobicuTelefon  
        .then(pokazatiTelefon) // ulančavanje  
        .then(function (fulfilled) {  
            // da, dobićeš nov telefon  
            console.log(fulfilled);  
            // output: { brand: 'Samsung', color: 'black' }  
        })  
        .catch(function (error) {  
            // nećeš dobiti nov telefon  
            console.log(error.message);  
            // output: 'roditelji nisu zadovoljni'  
        });  
};
```

Slika 7.3 Ulančavanje obećanja [izvor: autor]

Nakon modifikacije koda, proveravamo urađeno osvežavanjem stranice u pregledaču i izlaz može biti kao na sledećoj slici.

Pogledajte rezultat u konzoli pregledača !!!



Slika 7.4 Ulančavanje obećanja - demonstracija [izvor: autor]

ASINHRONI RAD SA OBEĆANJIMA

Obećanja se izvršavaju asinhrono.

Obećanja se izvršavaju *asinhrono*. Šta to praktično znači? Za početak, razmatrajmo malu modifikaciju prethodno kreirane metode sa ulančanim obećanjima. Modifikacija je istaknuta crvenom bojom na sledećoj slici:

```
// poziv obećanja (Promise objekta)
var pitajRoditelja = function () {
    console.log('Pre nego što pita roditelja :'); // log pre
    dobituTelefon
        .then(pokazatiTelefon) // ulančavanje
        .then(function (fulfilled) {
            // da, dobiceš nov telefon
            console.log(fulfilled);
            // output: { brand: 'Samsung', color: 'black' }
        })
        .catch(function (error) {
            // nećeš dobiti nov telefon
            console.log(error.message);
            // output: 'roditelji nisu zadovoljni'
        });
    console.log('Nakon što pita roditelja :'); // log posle
}:
```

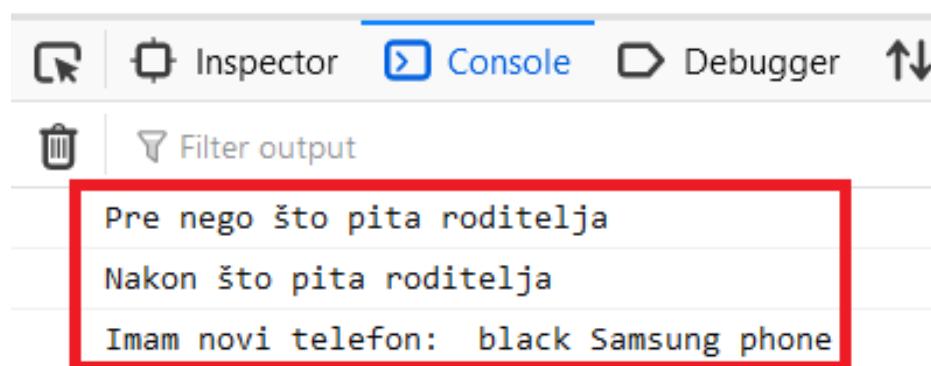
Slika 7.5 Izlaz naredbe za praćenje izvršavanja obećanja [izvor: autor]

Kada bi ovakav kod bio pokrenut, očekivani rezultat bi trebalo da bude u formi sledeće sekvence:

1. Pre nego što pita roditelja;
2. Imam novi telefon: black Samsung phone;
3. Nakon što pita roditelja.

Međutim, u realnosti rezultat izvršavanja programa je drugačiji i izgleda kao na sledećoj slici.

Pogledajte rezutat u konzoli pregledača !!!



Slika 7.6 Asinhrono izvršavanje lanca obećanja [izvor: autor]

Razlog navedenog je asinhrono izvršavanje koda koji sadrži obećanje. Asinhrono izvršavanje znači da se kod izvršava bez blokiranja i čekanja nekog rezultata. Sve što je potrebno da se sačeka jeste izvršavanje koda koji je spakovan u poziv `.then()`, a to predstavlja nastavak izvršavanja obećanja.

SINTAKSA ASYNC / AWAIT

Za standardom ES7 uvedena je nova sintaksa poznata pod nazivom **async / await**.

Za standardom **ES7** uvedena je nova sintaksa poznata pod nazivom **async / await** čijom primenom asinhrona sintaksa postaje još preglednija, lepša i lakša za razumevanje. Ovom sintaksom su izbačene iz upotrebe klauzule **.then** i **.catch**.

Sledi inoviran kod, izložen u prethodnom izlaganju, primenom stila **async / await**:

```
// Primer upotrebe Promise objekata

const roditeljZadovoljan = true;

// Promise
var dobicuTelefon = new Promise(
    function (resolve, reject) {
        if (roditeljZadovoljan) {
            var phone = {
                brand: 'Samsung',
                color: 'black'
            };
            resolve(phone); // rešeno
        } else {
            var reason = new Error('roditelji nisu zadovoljni');
            reject(reason); // odbijeno
        }
    }
);

// 2. obećanje
var pokazatiTelefon = function (phone) {
    return new Promise(
        function (resolve, reject) {
            var message = 'Imam novi telefon: ' +
                phone.color + ' ' + phone.brand + ' phone';

            resolve(message);
        }
    );
};

// poziv obećanja preko ES7 async await sintakse

async function pitajRoditelja() {
    try {
        console.log('Pre nego što pita roditelja'); // log pre

        let phone = await dobicuTelefon;
```

```
let message = await pokazatiTelefon(phone);

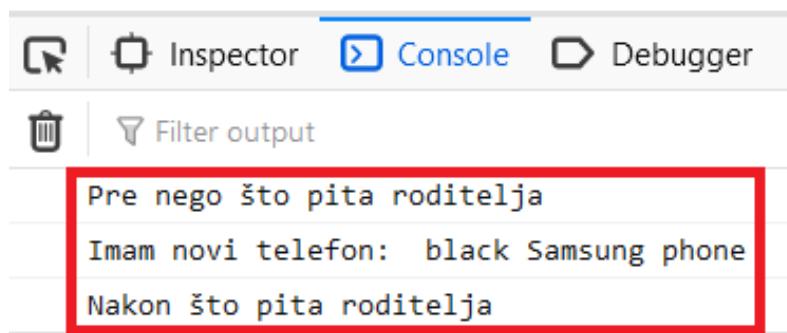
console.log(message);
console.log('Nakon što pita roditelja'); // log posle
}
catch (error) {
    console.log(error.message);
}
}

// async await je takođe i ovde
(async () => {
    await pitajRoditelja();
})();
```

Reč “`async`” pre naziva funkcije znači jednu stvar - funkcija uvek vraća obećanje (`Promise` objekat). Sa druge strane, primenom ključne reči `await` ukazuje na to da `JavaScript` kod čeka dok se obećanje ne podesi i ne vrati rezultat.

Rezultat je sinhronizovan kod čije izvršavanje je prikazano sledećom slikom.

Pogledajte rezutat u konzoli pregledača !!!



Slika 7.7 Rezultat primene sintakse `async / await` [izvor: autor]

OBIĆNE ILI ASINHRONE FUNKCIJE

Poređenje i scenariji primene običnih i asinhronih funkcija.

Poređenje i scenariji primene običnih i asinhronih funkcija su teme kojima će biti zaokružena diskusija u vezi sa ovom naprednom JS funkcionalnošću.

Biće analizirana dva primera. Oba primera obavljaju jednostavnu operaciju sabiranja dva broja. U prvom slučaju će to uraditi obična funkcija, a u drugom funkcija kojoj se pristupa preko udaljenog poziva.

Sledećim listingom je prikazan prvi scenario - primena "obične" funkcije:

```
// sabiranje brojeva na uobičajen način

function add (num1, num2) {
    return num1 + num2;
}

const result = add(1, 2); // dobija se trenutno result = 3
```

U ovom slučaju rezultat poziva metode će biti trenutno dostupan.

Razmatra se drugi scenario po kojem je prethodni zadatak obavljen preko udaljenog poziva metode. Ovde nije moguće dobiti trenutno rešenje i na izvršavanje funkcije je neophodno sačekati.

```
// udaljeno sabira dva broja

// dobija rezultat API pozivom
const result = getAddResultFromServer('http://www.example.com?num1=1&num2=2');
// dobija se result = "undefined"
```

Zaključak:

1. Ukoliko se radi o jednostavnim pozivima, koji moraju odmah da daju rezultat, koriste se obične funkcije;
2. API pozivi, preuzimanje datoteka, čitanje datoteka predstavljaju neke od brojnih asinhronih operacija koje nije moguće obaviti preko običnih funkcija

OBJEKTI TIPO PROMISE - ASYNC/AWAIT SINTAKSA (VIDEO MATERIJAL)

Async JS Crash Course - Callbacks, Promises, Async Await (trajanje 24:30)

Ova lekcija sadrži video materijal. Ukoliko želite da pogledate ovaj video morate da otvorite LAMS lekciju.

✓ Poglavlje 8

Materijali za dodatni rad

DODATNI MATERIJALI

Proširite vaše znanje posetom sledećim linkovima.

Proširite vaše znanje posetom sledećim linkovima:

1. <https://www.w3schools.com/js/>
2. <https://javascript.info/>
3. <https://www.tutorialspoint.com/javascript/index.htm>
4. <https://www.guru99.com/interactive-javascript-tutorials.html>

▼ Poglavlje 9

Pokazna vežba - HTML, CSS i JS

KREIRANJE MALOG SAJTA SA SLIKAMA (TRAJANJE: 45 MINUTA)

Kreiranje malog sajta sa slikama za demonstraciju povezivanja obrađenih tehnologija.

Zadatak vežbe jeste kreiranje HTML stranice koja:

- prikazuje datum asinhronim pozivom JavaScript koda;
- prikazuje četiri slučajno izabrane slike sa sajta za preuzimanje slika unsplash.com;
- koristi eksterne CSS i JS datoteke za svoje funkcionisanje;
- daje izlaz kao na sledećoj slici.



Slika 9.1 Predloženi izgled stranice [izvor: autor]

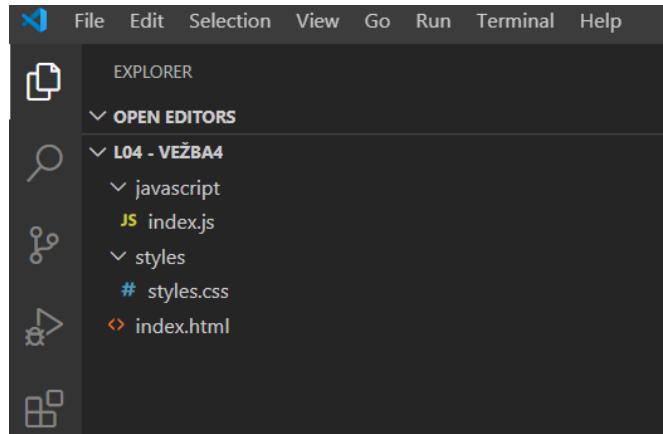
Za razvoj može da se upotrebi i razvojno okruženje *Visual Studio Code* kojeg ćemo intenzivno koristiti za razvoj Angular aplikacija od sledeće lekcije pa sve do kraja.

U razvojnog okruženju će biti kreiran folder *L04-Vežba4* i u njemu dva podfoldera *styles* i *javascript*. Pre kodiranja, napravimo i par datoteka:

- *index.html* direktno u folderu *L04-Vežba4*;
- *styles.css* u folderu *styles*;

- *index.js* u folderu *javascript*.

Struktura projektnog foldera je prikazana sledećom slikom.



Slika 9.2 Struktura projektnog foldera [izvor: autor]

KREIRANJE HTML-A

Prvi korak jeste kreiranje HTML koda.

Prvi korak jeste kreiranje HTML koda kojim će stranica biti i kreirana nakon interpretaciju u veb pregledaču. Sledećim listingom je priložen HTML kod stranice index.html.

```
<!DOCTYPE html>
<html lang="sr">
  <head>
    <title>Vežba4!</title>
    <link rel="stylesheet" href="styles/styles.css" />
    <script async src=".//javascript/index.js"></script>
  </head>
  <body>
    <h1>Vežba 4 - Malo asinhronog JavaScript-a!</h1>
    <h4 id='date'></h4>

    <div class="image-section">
      <div class="section-style">
        
        <p>Slučajno izabrana slika sa sajta unsplash.com.</p>
      </div>

      <div class="section-style">
        
        <p>Slučajno izabrana slika sa sajta unsplash.com.</p>
      </div>
    </div>

    <div class="image-section">
      <div class="section-style">
        
```

```
<p>Slučajno izabrana slika sa sajta unsplash.com.</p>
</div>

<div class="section-style">
    
    <p>Slučajno izabrana slika sa sajta unsplash.com.</p>
</div>
</div>
</body>
</html>
```

Osim osnovnih elemenata, koje svaka HTML stranica mora da poseduje, u fokusu vežbe su:

- asinhrono učitavanje JavaScript koda iz eksterne datoteke [/javascript/index.js..](#) Zadatak koda je kreiranje objekta *Date* koji prikazuje trenutno vreme kada je stranica učitana u okviru taga: `<h4 id='date'></h4>;`
- kreiranje `<div>` tagova za učitavanje *CSS* klase *image-section* i *section-style* za stilizovanje i prikazivanje slučajno izabranih slika sa sajta splash.com;
- takođe, podešava se i pozadina odgovarajućim stilom.

CSS I JAVASCRIPT

Datoteka stilova i JavaScript-om završava se definisija mini sajta.

Kao što je rečeno u prethodnom izlaganju, u folderu *styles* je kreirana datoteka stilova *styles.css*, koja je u listingu sa prethodne strane učitana u liniji koda 5. Datoteka podešava:

- pozadinu i izgled stranice;
- način prikazivanja slika na stranici.

Sledi listing navedene datoteke stilova:

```
body {
    text-align: center;
    background-color: #f0e8c5;
}

div {
    margin-top: 15px;
}

.image-section {
    display: flex;
    justify-content: center;
}

.section-style {
    margin-right: 25px;
    margin-left: 25px;
```

```
background-color: white;  
}
```

Konačno, datoteka index.js sadrži malo JavaScript koda čiji je zadatak da prikaže trenutno vreme u HTML elementu čiji je identifikator "*date*".

Sledi listing navedene *JavaScript* datoteke:

```
document.getElementById('date').innerHTML = new Date().toLocaleDateString();
```

✓ Poglavlje 10

Individualna vežba 4

VEŽBA 1 (TRAJANJE: 45 MINUTA)

Vežbanje naprednog rada sa JS funkcijama.

1. Preuzmите delimično urađen primer iz sekcije Shared Resources objekta učenja "**Napredni rad sa funkcijama**";
2. Implementirajte sintaksu širenja na osnovu svih razmatranih scenarija:

- predaja niza funkciji;
- predaja više različitih nizova funkciji;
- kombinovanje niza i običnih argumenata funkcije;
- spajanje više različitih nizova u jedan;

Celokupan kod integrisati u HTML stranicu. Pozvati predmetnog asistenta nakon urađenog zadatka.

VEŽBA 2 (TRAJANJE: 45 MINUTA)

Kreirajte samostalno objekat i primenite funckiju na njega.

1. Možete da se oslonite na pokazni primer iz objekta učenja "Rad sa objektima";
2. Kreirajte objekat *osoba* klase sa osobinama:*ime, prezime*;
3. Kreirajte objekat *student* sa osobinama: *brojIndeksa, prosek*;
4. Osobina prosek mora da bude unutar vrednosti 6.0 i 10.0, u suprotnom se javlja izuzetak kojeg je neophodno obraditi;
5. Poruka o grešci za prosek se prikazuje u HTML stranici na odgovarajućem mestu;
6. Objekat *student* nasleđuje osobine iz prototipa *osoba*;
7. Kreirajte funkciju *krerajStudenta()* koja kao vrednost vraća objekat student sa vrednostima vlastitih osobina koje odgovaraju argumentima kreirane funkcije;
8. Ugradite kod u odgovarajuću HTML stranicu;
9. Pozvati predmetnog asistenta nakon urađenog zadatka.

✓ Poglavlje 11

Domaći zadatak 4

DOMAĆI ZADATAK (PREDVIĐENO VREME 180 MINUTA)

Demonstracija savladavanja naprednih JS koncepata u formi domaćeg zadatka.

Zadatak 1:

U fokusu domaćeg zadatka je primer sa korisnicima. Postoje dva tipa korisnika:

- admin;
- user;

1. Na stranici *index.html* se nalazi link ka stranci *login.html*;
2. Korisnici se na ovoj stranici prijavljuju na aplikaciju;
3. U slučaju uspešnog prijavljivanja korisnici se preusmeravaju na stranicu *uspeh.html*;
4. Sve stranice moraju da budu pregledne, organizovana CSS stilovima i rasporedima;
5. Izbor stila je po želji i slobodi studenta;
6. Admin daje obećanje običnom korisniku da će mu omogućiti prikazivanje trenutnog datuma na stranici *uspeh.html* ukoliko je običan korisnik prijavljen lozinkom dužine 6 - 10 karaktera. U suprotnom prikazuje mu poruku da bi trebalo da dužinu lozinke prilagodi zahtevima;

Zadatak 2:

Koristite primer iz OU 7 "*Objekti tipa Promise - async/await sintaksa*"

1. Roditelj se prijavljuje na stranici *index.html* sa svojim podacima za korisničko ime i lozinku (mama - 1234 ili tata - 9876);
2. U slučaju uspešnog prijavljivanja sa stranici *potvrda.html* nalazi se kontrola gde roditelj bira da li kupuje ili ne (*RadioButton* ili *ComboBox* - po izboru studenta);
3. Ovako roditelj iskazuje zadovoljstvo detetovim zalaganjem;
4. Na ovoj stranici se nalazi i mogućnost izbora od tri ponuđena telefona, ukoliko se roditelj odluči za kupovinu;
5. Rezultate izvršavanja odgovarajućeg obećanja prikazati na stranici *potvrda.html* nakon klika na dugme kojim se potvrđuje izbor roditelja;
6. Sve stranice moraju da budu stilizovane iz eksterne CSS datoteke.

Student bira jedan od ponuđenih domaćih zadataka. U slučaju da student uradi oba i uspešno ih javno odbrani, nagrađuje se sa dodatnim poenom za zalaganje.

▼ Poglavlje 12

Zaključak

ZAKLJUČAK

Lekcija se bavila izučavanjem naprednih JS koncepta kao osnova za izučavanje okvira Angular.

U fokusu lekcije je bilo izučavanje naprednih JS koncepta i na taj način je zaokružena priprema za učenje razvoja frontend komponenata veb sistema primenom jezičke nadogradnje za JavaScript ECMA 6 pod nazivom *TypeScript*. Posebno je istaknuto da će jezik *TypeScript* biti osnov svega što će biti rađeno u nastavku primenom radnog okvira *Angular*.

Lekcija je pažljivo birala teme i držala se snažno pokaznih primera kao podrške za analizu, diskusiju i demonstraciju. Lekcija se bavila sledećim naprednim temama jezika *JavaScript*:

1. Napredni rad sa funkcijama u *JavaScript* jeziku;
2. Podešavanje objektnih osobina u *JavaScript* jeziku;
3. Prototipovi, nasleđivanje;
4. Klase;
5. Rukovanje greškama;
6. Koncepti Promises, *async / await*;
7. Generatori, napredne iteracije;
8. Moduli;
9. Ostali napredni *JavaScript* koncepti kao uvod u *Angular* okvir.

Savladavanjem ove lekcije student je u potpunosti pripremljen na izučavanje frontend radnog okvira Angular i vlada *HTML*, *JS* i *CSS* konceptima i principima.

LITERATURA

Za pripremu lekcije 4 korišćena je aktuelna štampana i veb literatura.

Obavezna literatura:

1. Jennifer Niederst Robbins, Learning Web Design - Fifth Edition, by , Copyright ©; 2018 O'Reilly Media, Inc.
2. Roxane Anquetil, Fundamental Concepts for Web Development: HTML5, CSS3, JavaScript and much more! For complete beginners!, 2019,

Dopunska literatura:

1. Daniel Bell, HTML & CSS: A Step-by-Step Guide for Beginners2, Guzzler Media, 2019.

Veb izbori:

1. <https://javascript.info/>
2. <https://tylermcginnis.com/var-let-const/>



IT255-VEB SISTEMI 1

NodeJS kao radno okruženje

Lekcija 05

PRIRUČNIK ZA STUDENTE

IT255-VEB SISTEMI 1

Lekcija 05

NODEJS KAO RADNO OKRUŽENJE

- ✓ NodeJS kao radno okruženje
- ✓ Poglavlje 1: Uvod u Node.js
- ✓ Poglavlje 2: Podešavanje Node.js okruženja
- ✓ Poglavlje 3: Node Package Manager
- ✓ Poglavlje 4: Funkcije povratnog poziva i događaji u Node.js
- ✓ Poglavlje 5: Klasa EventEmitter
- ✓ Poglavlje 6: Baferi i tokovi u Node.js
- ✓ Poglavlje 7: Web modul
- ✓ Poglavlje 8: Node.js RESTful API
- ✓ Poglavlje 9: Pokazna vežba (predviđeno vreme 45 min)
- ✓ Poglavlje 10: Individualna vežba (predviđeno vreme 90 min)
- ✓ Poglavlje 11: Domaći zadatak (predviđeno vreme 120 min)
- ✓ Zaključak

Copyright © 2017 - UNIVERZITET METROPOLITAN, Beograd. Sva prava zadržana. Bez prethodne pismene dozvole od strane Univerziteta METROPOLITAN zabranjena je reprodukcija, transfer, distribucija ili memorisanje nekog dela ili čitavih sadržaja ovog dokumenta., kopiranjem, snimanjem, elektronskim putem, skeniranjem ili na bilo koji drugi način.

Copyright © 2017 BELGRADE METROPOLITAN UNIVERSITY. All rights reserved. No part of this publication may be reproduced, stored in a retrieval system or transmitted in any form or by any means, electronic, mechanical, photocopying, recording, scanning or otherwise, without the prior written permission of Belgrade Metropolitan University.

✓ Uvod

UVOD

Moćna platforma bazirana na JS i Google Chrome podršci.

Node.js je izuzetno moćna platforma bazirana na jeziku *JavaScript*, izgrađnena na *Google Chrom JavaScript V8* podršci. Ova platforma je namenjena za razvoj savremenih veb aplikacija, poput:

- sajtovi za striming video zapisa;
- aplikacija "jedne stranice" (*single -page application*);
- brojnih drugih veb aplikacija.

Node.js pripada konceptu otvorenog koda (eng. *open source*), potpuno je besplatan i koristi ga ogromna programerska zajednica širom sveta. Upravo iz navedenog razloga, lekcija ima za cilj da omogući studentima savladavanje i razumevanje osnovnih *Node.js* koncepta, kao i arhitekture same platforme, kao osnova za izučavanje razvoja savremenih front-end aplikacija korišćenjem radnog okvira *Angular*.

▼ Poglavlje 1

Uvod u Node.js

O NODE.JS PLATFORMI

Node.js predstavlja serversku platformu izgrađenu na "mašini" Google Chrome's JavaScript Engine.

[Node.js](#) predstavlja serversku platformu izgrađenu na "mašini" *Google Chrome's JavaScript Engine (V8)*. Platformu je razvio [Ryan Dahl](#) tokom 2009. i njena trenutna verzija glasi [v16.14.2](#). Sama definicija platforme [Node.js](#) je podržana vlastitom zvaničnom dokumentacijom na linku: <https://nodejs.org/en/>.



Slika 1.1 Node.js je podržana vlastitom zvaničnom dokumentacijom[izvor: autor]

[Node.js](#) predstavlja platformu izgrađenu sa ciljem davanja podrške jednostavnom građenju brzih i skalabilnih mrežnih aplikacija. Platforma koristi neblokirajući (eng. *non - blocking*) i vođen događajima (eng. *event - driven*) model koji je čini laganom i efikasnom, skoro savršenom za primenu na aplikacijama sa velikom količinom podataka, koje funkcionišu u realnom vremenu preko distribuiranih uređaja.

Node.js predstavlja okruženje za izvršavanje aplikacija (eng. *runtime environment*) u realnom vremenu koje pripada konceptu otvorenog koda eng. [open source](#)). Aplikacije za [Node.js](#) pisane su u jeziku [JavaScript](#) i mogu biti pokrenute na različitim operativnim sistemima, poput: OS X, Microsoft Windows, kao i Linux.

[Node.js](#), takođe, obezbeđuje bogatu biblioteku [JavaScript](#) modula kojima se u velikoj meri pojednostavljuje razvoj veb aplikacija.

Sledećom slikom je ilustrovana pojednostavljena organizacija [Node.js](#) platforme.



Slika 1.2 Organizacija Node.js platforme [izvor: autor]

KARAKTERISTIKE PLATFORME NODE.JS

Važne karakteristike koje čine Node.js prvim izborom od strane velikog broja arhitekata softvera.

U narednom izlaganju neophodno je istaći najvažnije karakteristike koje čine *Node.js* prvim izborom od strane velikog broja arhitekata softvera.

- **Asinhronost i vođenje događajima** - *Node.js* biblioteka sadrži veliki broj API koji su svi asinhroni, a to znači da grade neblokirajuća softverska rešenja. U osnovi to znači da server **baziran na Node.js** nikada neće čekati da API vrati rezultat. Server se fokusira na sledeći API poziv i pomoću notifikacija mehanizma događaja dobiće odgovor za prethodni API poziv.
- **Velika brzina** - zahvačujući činjenici da je izgrađena na "mašini" *Google Chrome's V8 JavaScript Engine*, *Node.js* biblioteka je jako brza po pitanju izvršavanja koda.
- **Jedna nit ali visok stepen skalabilnosti** - *Node.js* koristi jednonitni model sa ciklusima događaja. Mehanizam događaja omogućava serveru da odgovara bez blokiranja, čineći server visoko skalabilnim u odnosu na tradicionalne servere koji kreiraju ograničene niti za rukovanje zahtevima. Node.js koristi jednonitne programe koji mogu da obezbede obradu mnogo većeg broja zahteva u odnosu na tradicionalne servere, poput *Apache HTTP* servera.
- **Ne čuva podatke u privremenim skladištima** – *Node.js* aplikacije nikada ne baferuju podatke. Ove aplikacije jednostavno šalju podatke u delovima.
- **Licenca** – *Node.js* je objavljen pod MIT licencom (<https://raw.githubusercontent.com/joyent/node/v0.12.0/LICENSE>).

Trenutno, brojne kompanije koriste vlastite aplikacije koje koriste *Node.js*:

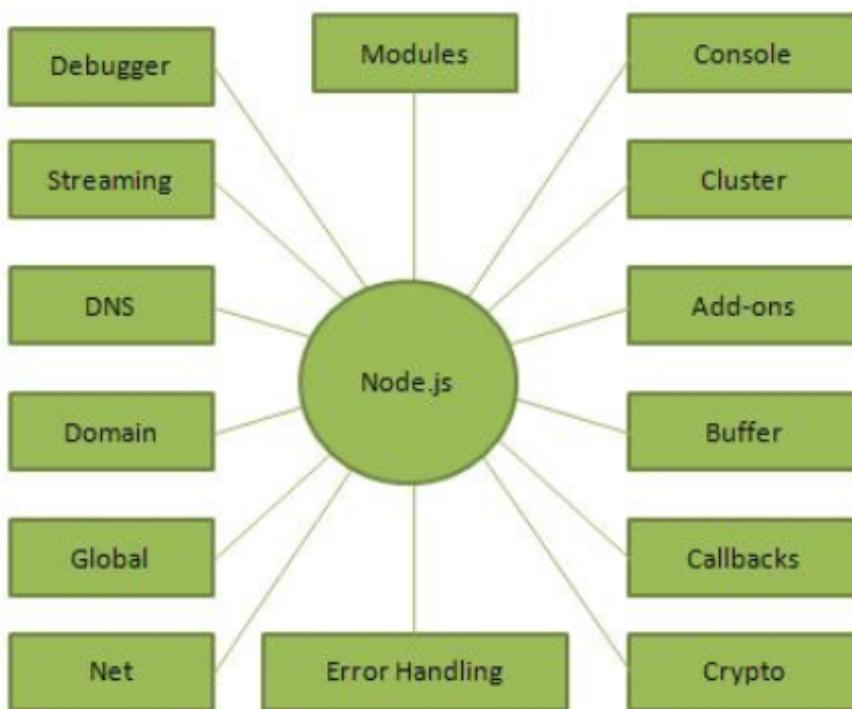
- *eBay*,
- *General Electric*,
- *GoDaddy*,
- *Microsoft*,
- *PayPal*,

- *Uber*,
- *Wikipedia*,
- *Yahoo!*,
- kao i brojne druge.

NODE.JS KONCEPT

Demonstracija komponenata Node.js platforme.

Sledeći dijagram prikazuje neke važne delove *Node.js* platforme o kojima ćemo detaljno raspravljati u narednim objektima učenja. Komponente se upravo tako i nazivaju i, upravo iz navedenog razloga, na sledećoj slici neće biti prevedeni.



Slika 1.3 Demonstracija komponenata Node.js platforme[izvor: autor]

Kada bi trebalo korititi Node.js:

- I/O vezane aplikacije;
- Aplikacije sa tokovima podataka;
- Aplikacije sa velikom količinom podataka u realnom vremenu (*Data Intensive Real-time Applications (DIRT)*)
- Aplikacije bazirane na *JSON API*;
- Aplikacije jedne stranice (eng. *Single Page Applications*)

Kada ne bi trebalo korititi Node.js:

- Nije preporučljivo koristiti *Node.js* za procesorski intenzivne aplikacije.

✓ Poglavlje 2

Podešavanje Node.js okruženja

PREUZIMANJE NODE.JS INSTALERA

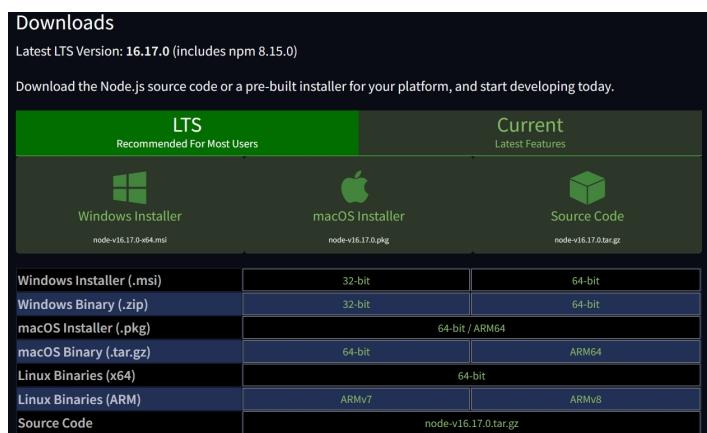
Instalacija se vrši jednostavno iz preuzetog Node.js instalera.

Da bi korisnik mogao da koristi [Node.js](#) neophodno je da ima dve stvari:

- instaliran [Node.js](#);
- pogodan tekst editor za kodiranje;

Po pitanju tekst editora, nema ograničenja. Moguće je kodiranje u svakom poznatom tekstu editoru. Preporuka je da studenti koriste razvojno okruženje [Visual Studio Code](#) i njegov tekst editor u daljem radu.

Na veb lokaciji <https://nodejs.org/en/download/> moguće je preuzeti odgovarajući instalacioni paket, u zavisnosti od operativnog sistema na kojem se instalacija obavlja.



Slika 2.1 Link za preuzimanje instalacionog paketa[izvor: autor]

Za [Windows](#) operativni sistem neophodno je preuzeti odgovarajući instalator koji omogućava jednostavnu instalaciju [Node.js](#) i [NPM](#) (*Node Package Manager*) putem čarobnjaka (eng. *wizzard*). U trenutku pisanja ovih materijala najnovija verzija glasi: [node-v16.17.0-x64](#).

Nakon nekoliko klikova i praćenja instalacije, potrebno je proveriti da li je instalacija uspešno obavljena. Naredbama:

```
node -v  
npm -v
```

koje se kucaju u **MS DOS** Prompt-u ili terminalu razvojnog okruženja **Visual Studio Code**, dobijaju se informacije o trenutnim instaliranim verzijama.

The screenshot shows a terminal window with tabs: TERMINAL, PROBLEMS, OUTPUT, and DEBUG CONSOLE. The TERMINAL tab is selected. The command `node -v` is run, resulting in the output `v16.14.2`. The command `npm -v` is run, resulting in the output `8.6.0`. Both outputs are highlighted with yellow boxes.

Slika 2.2 Provera instalacije za Node.js i NPM [izvor: autor]

AŽURIRANJE NODE.JS I NPM

Preporuka je da uvek imate poslednje stabilne verzije.

Preporuka je da uvek imate poslednje stabilne verzije. U poslednjem izlaganju (slika 2) pokazano je kako je moguće proveriti poslednje instalirane verzije za **Node.js** i **NPM**. Ukoliko nastanu promene u verzijama, na jednostavan način je moguće obaviti njihovo ažuriranje.

Za **Node.js**, najjednostavnije je preuzeti poslednju verziju instalera i ponoviti proceduru koja je prethodno objašnjena.

Što se tiče paketa **NPM**, njega je moguće ažurirati jednostavno sa komandne linije. Za početak potrebno je obrisati **NPM** keš navođenjem naredbe:

```
npm cache clean -f
```

The screenshot shows a terminal window with tabs: TERMINAL, PROBLEMS, OUTPUT, and DEBUG CONSOLE. The TERMINAL tab is selected. The command `npm cache clean -f` is run. The output shows a warning about using --force, followed by notices about a new minor version of npm available (8.6.0 to 8.18.0), the changelog URL (<https://github.com/npm/cli/releases/tag/v8.18.0>), and a suggestion to run `npm install -g npm@8.18.0` to update. The output is highlighted with yellow boxes.

Slika 2.3 Brisanje NPM keša [izvor: autor]

Na slici je moguće primetiti da je u terminalu ukazano na najnoviju dostupnu verziju i kako je moguće obaviti njeno instaliranje.

Navođenjem naredbe:

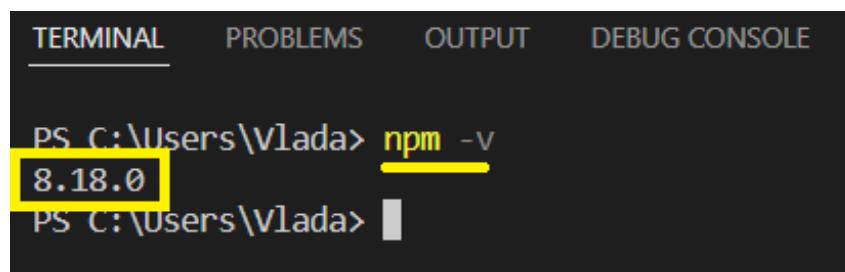
```
npm install -g npm@8.18.0
```

biće obavljeno ažuriranje **NPM** paketa sa verzije 8.6.0 na verziju 8.18.0. Navedeno je ilustrovano sledećom slikom.

```
PS C:\Users\Vlada> npm install -g npm@8.18.0
removed 6 packages, changed 72 packages, and audited 206 packages in 5s
11 packages are looking for funding
  run `npm fund` for details
```

Slika 2.4 Ažuriranje NPM paketa [izvor: autor]

Ponovo će u terminalu biti proverena trenutna NPM verzija sa ciljem provere uspešnosti urađenog posla.



Slika 2.5 Provera ažurirane NPM verzije [izvor: autor]

Sada je moguće smatrati da je računar podešen kao radna stanica za ravoj i pokretanje aplikacija primenom serverske platforme **Node.js**.

▼ Poglavlje 3

Node Package Manager

NPM FUNKCIONALNOSTI

Node Package Manager (NPM) obezbeđuje dve glavne funkcionalnosti.

Node Package Manager (NPM) obezbeđuje dve glavne funkcionalnosti:

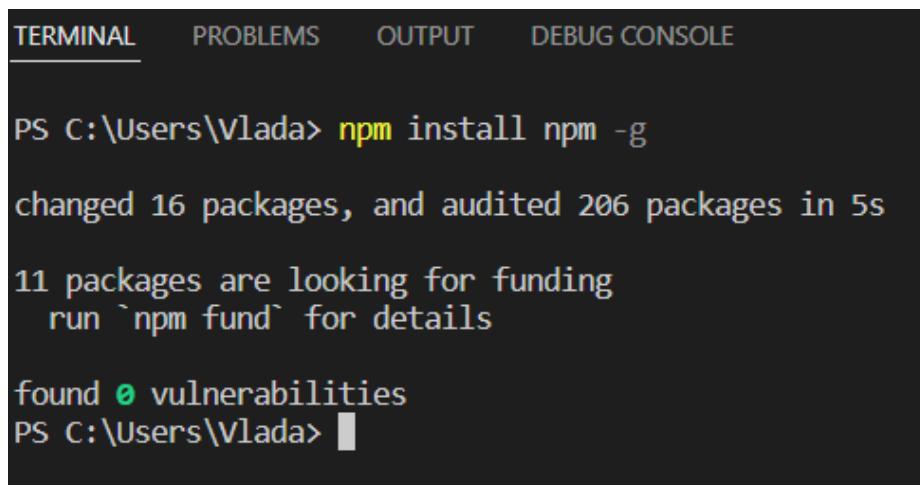
- online repozitorijumi za pronalaženje i instaliranje Node.js paketa i modula koje je moguće manuelno pretražiti na linku search.nodejs.org;
- alati komandne linije za instaliranje Node.js paketa, rukovanje verzijama i zavisnostima Node.js paketa.

NPM dolazi kao sastavni deo Node.js instalacije i kao takav dolazi u određenoj verziji. Ukoliko je, u međuvremenu, objavljena nova NPM verzija, u prethodnom izlaganju je pokazano kako ona može da bude ažurirana.

Sledećom naredbom je pokazano kako, alternativno, NPM može da bude ažuriran iz globalnog repozitorijuma (oznčeno sa g):

```
npm install npm -g
```

Sledećom slikom je prikazan rezultat ažuriranja NPM na prethodno opisani način.

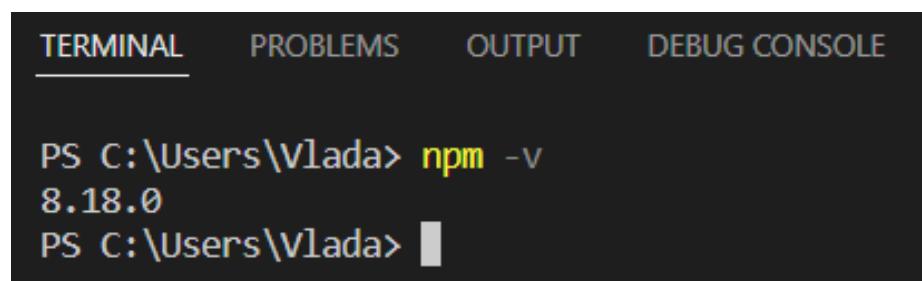


The screenshot shows a terminal window with tabs for TERMINAL, PROBLEMS, OUTPUT, and DEBUG CONSOLE. The TERMINAL tab is active, displaying the command 'npm install npm -g' and its output. The output shows that 16 packages were changed and 206 packages were audited in 5 seconds. It also indicates that 11 packages are looking for funding and provides a command to run 'npm fund' for details. Finally, it states that 0 vulnerabilities were found. The prompt 'PS C:\Users\Vlada>' is visible at the end of the session.

```
PS C:\Users\Vlada> npm install npm -g
changed 16 packages, and audited 206 packages in 5s
11 packages are looking for funding
  run `npm fund` for details
found 0 vulnerabilities
PS C:\Users\Vlada>
```

Slika 3.1 Rezultat ažuriranja NPM [izvor: autor]

Ponovo, prikazuje se provera trenutne NPM verzije.



The screenshot shows a terminal window with a dark background and light-colored text. At the top, there are four tabs: TERMINAL (underlined), PROBLEMS, OUTPUT, and DEBUG CONSOLE. Below the tabs, the terminal prompt is PS C:\Users\Vlada>. The user then types 'npm -v' and presses Enter. The output shows '8.18.0' in green text. A small gray rectangular input field is visible at the bottom right of the terminal area.

```
PS C:\Users\Vlada> npm -v
8.18.0
PS C:\Users\Vlada>
```

Slika 3.2 Provera trenutne NPM verzije [izvor: autor]

▼ Poglavlje 4

Funkcije povratnog poziva i događaji u Node.js

FUNKCIJE POVRATNOG POZIVA

Funkcija povratnog poziva je asinhroni ekvivalent običnoj funkciji.

Funkcija povratnog poziva, ili kraće, *povratna funkcija* (eng. callback) je asinhroni ekvivalent običnoj funkciji. Ovakva vrsta funkcije se poziva nakon završetka datog zadatka. Node.js daje snažnu podršku korišćenju funkcija povratnog poziva. Drugim rečima, svi Node.js API su napisani tako da podržavaju primenu navedene vrste funkcija.

Posmatra se konkretni primer - funkcija za čitanje datoteke može da počne da čita datoteku i da, odmah, vratи kontrolu u okruženje za izvršavanje kako bi sledeća instrukcija mogla da se izvrši. Kada je čitanje / upisivanje datoteke završeno, okruženje će pozvati *povratnu funkciju* dok joj prosleđuje drugu *povratnu funkciju* sa sadržajem datoteke kao parametrom. Odavde sledi da ne dolazi do blokiranja ili čekanja da se završi operacija čitanje / upisivanje datoteke. Na ovaj način, Node.js je veoma skalabilan i može da obradi veliki broj zahteva bez potrebe za čekanjem da bilo koja funkcija vrati rezultat.

POKAZNI PRIMER BLOKIRAJUĆEG KODA

Uvodi se odgovarajući pokazni primer blokirajućeg - sinhronog koda.

Da bi koncept funkcije povratnog poziva bio jasniji, uvodi se odgovarajući pokazni primer.

JavaScript kod, iz datoteke **main.js**, na sinhron način poziva metodu koja čita iz datoteke koja će biti nazvana **input.txt**. To znači da sve operacije koje slede iza ovog poziva, moraju da sačekaju da metoda vrati rezultat. Ovo ima za posledicu blokiranje ostatka koda do okončanja pomenutog zadatka.

Sledi listing datoteke **input.txt**:

Prvi primer pokazuje primenu blokirajućeg / neblokirajućeg koda u Node.js okruženju.

Sledi listing datoteke main.js:

```
var fs = require("fs");
var data = fs.readFileSync('input.txt');
```

```
console.log(data.toString());  
console.log("Kraj programa!!!");
```

U razvojnom oruženju **Visual Studio Code** biće pokrenut terminal i u njemu je potrebno otkucati sledeću naredbu, kojom se pokreće kreirani program:

```
node main.js
```

Sledeća slika pokazuje rezultat izvršavanja. Svi naknadni pozivi (linije koda 4 i 5) su bili na čekanju dok nije završen sinhroni poziv (linija koda 2).

```
TERMINAL PROBLEMS OUTPUT DEBUG CONSOLE  
Windows PowerShell  
Copyright (C) Microsoft Corporation. All rights reserved.  
Install the latest PowerShell for new features and improvements! https://aka.ms/PSWindows  
PS C:\Users\Vlada\Desktop\Node.js - primeri\Primer 1 - 003> node main.js  
Prvi primer pokazuje primenu blokirajućeg / neblokirajućeg koda u Node.js okruženju.  
Kraj programa!!!  
PS C:\Users\Vlada\Desktop\Node.js - primeri\Primer 1 - 003>
```

Slika 4.1 Pokazni primer blokirajućeg koda [izvor: autor]

POKAZNI PRIMER NEBLOKIRAJUĆEG KODA

Primena poziva funkcije povratnog poziva.

Uvode se modifikacije prethodnog primera. **JavaScript** kod, iz datoteke **main.js**, na asinhron način poziva metodu koja čita iz datoteke koja će biti nazvana **input.txt**. To znači da sve operacije koje slede iza ovog poziva, ne moraju da sačekaju da metoda vrati rezultat. Ovo ima za posledicu izvršavanje ostatka koda bez potrebe čekanja okončanja pomenutog zadatka.

Upravo je ovde na delu primena poziva funkcije povratnog poziva (linije koda 3 - 6).

```
var fs = require("fs");  
  
fs.readFile('input.txt', function (err, data) {  
    if (err) return console.error(err);  
    console.log(data.toString());  
});  
  
console.log("Kraj programa!!!");
```

Asinhrona funkcija **readFile()** kao parametar preuzima drugu funkciju povratnog poziva kojoj je prosleđen parametar data koji preuzima sadržaj pročitane datoteke.

Ovo ima za posledicu da poziv koji sledi (linija koda 8) ne mora da čeka da pomenuti lanac poziva vrati vrednosti.

Ponovo, u razvojnom oruženju **Visual Studio Code** biće pokrenut terminal i u njemu je potrebno otkucati sledeću naredbu, kojom se pokreće kreirani program:

node main.js

Sledećom slikom je prikazano izvršavanje kreiranog **neblokirajućeg** koda.

```
TERMINAL PROBLEMS OUTPUT DEBUG CONSOLE
PS C:\Users\Vlada\Desktop\Node.js - primeri\Primer 1 - 003\neblokirajući> node main.js
Kraj programa!!!
Prvi primer pokazuje primenu blokirajućeg / neblokirajućeg koda u Node.js okruženju.
PS C:\Users\Vlada\Desktop\Node.js - primeri\Primer 1 - 003\neblokirajući>
```

Slika 4.2 Pokazni primer neblokirajućeg koda [izvor: autor]

REZIME VEZAN ZA PRIMENU FUNKCIJE POVRATNOG POZIVA

Kada neblokirajući program treba da koristi podatak za obradu, treba ga zadržati u istom bloku.

Prethodna dva primera su ilustrovala primenu blokirajućih i neblokirajućih poziva.

Prvi primer je pokazao kako dolazi do blokiranja programskih blokova dok se zadatak čitanja datoteke ne kopletira. Nakon toga dolazi do nastavka izvršavanja ostalog dela koda.

U drugom primeru, ostale linije koda ne čekaju da se obavi čitanje datoteke do kraja i normalno se izvršavaju.

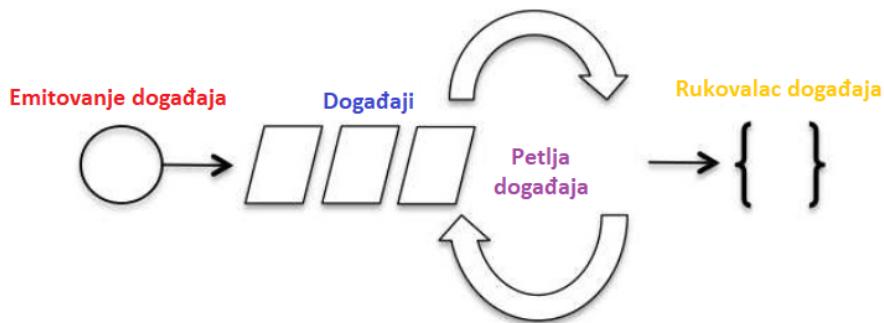
Dakle, program sa blokirajućim kodom izvršava se redom onako kako su naredbe pisane. Sa tačke gledišta programiranja, lakše je implementirati ovakvu logiku, jer se neblokirajući programi ne izvršavaju u nizu. **U slučaju da neblokirajući program treba da koristi bilo koji podatak za obradu, treba ga zadržati u istom bloku da bi se izvršio sekvencialno.**

PETLJE DOGAĐAJA

Moguće obezrediti konkurentnost kroz koncepte događaja funkcija povratnog poziva u Node.js.

Aplikacije koje se izvršavaju u okruženju [Node.js](#) su jednonitne. Međutim, ipak je moguće obezrediti konkurentnost kroz koncepte događaja ([event](#)) i funkcija povratnog poziva ([callbacks](#)). Svaki Node.js API je asinhron i jednonitan, a to znači da koristi asinhronne funkcije za rukovanje konkurentnošću. [Node.js](#) koristi tzv. šablon osmatranja ([observer pattern](#)). Jedina nit sadrži događaje koji se izvršavaju u petljama i svaki put, kada se neki zadatak obavi, *okida se odgovarajući događaj koji daje signal funkciji za osluškivanje događaja ([event-listener](#)) da bude izvršena* (sledeća slika).

[Node.js](#) se u velikoj meri oslanja na rukovanje događajima i to čini na veoma brz način u poređenju sa sličnim tehnologijama. Čim pokrene vlastiti server, [Node.js](#) inicira svoje promenljive, deklariše funkcije i, jednostavno, čeka na javljanje odgovarajućih događaja.



Slika 4.3 Rukovanje događajima u Node.js[izvor: autor]

Iako rukovanje događajima veoma podseća na upotrebu funkcija povratnog poziva ipak, postoji razlika. Funkcija povratnog poziva se poziva kada asinhrona funkcija vrati vlastiti rezultat. Rukovanje događajima se bazira na šablonu osmatranja. Funkcije koje osluškuju i obrađuju događaje ponašaju se kao osmatrači. Kada kod se javi neki događaj, odgovarajuća osluškivačka funkcija će početi sa svojim izvršavanjem. [Node.js](#) poseduje brojne ugrađene module događaja. Jedna od njih je klasa [EventEmitter](#) koja se koristi za povezivanje događaja sa vlastitim osluškivačem.

Sledeći listing pokazuje: učitavanje modula događaja (linije koda 1 - 5), povezivanje događaja sa rukovaocem (linija koda 8) i pokretanje događaja (linija koda 11).

```
// učitavanje modula događaja
var events = require('events');

// Kreiranje eventEmitter objekta
var eventEmitter = new events.EventEmitter();

// povezivanje događaja sa obrađivačem
eventEmitter.on('eventName', eventHandler);

// Pokretanje događaja
eventEmitter.emit('eventName');
```

POKAZNI PRIMER RUKOVANJA DOGAĐAJEM U NODE.JS.

Primena osluškivanja događana konceptom osmatrača.

Biće kreiran [JavaScript](#) kod pod nazivom **main.js** sa sledećim listingom:

```
// Učitavanje modula događaja
var events = require('events');

// Kreiranje eventEmitter objekta
var eventEmitter = new events.EventEmitter();

// kreiranje rukovaoca događajem
```

```
var connectHandler = function connected() {
    console.log('uspešna konekcija.');

    // Ispaljivanje događaja
    eventEmitter.emit('data_received');
}

// povezivanje događaja konekcije sa rukovaocem
eventEmitter.on('connection', connectHandler);

// povezivanje događaja data_received sa anonimnom funkcijom
eventEmitter.on('data_received', function() {
    console.log('uspešan prijem podataka.');
});

// Ispaljivanje connection događaja
eventEmitter.emit('connection');

console.log("Kraj programa.");
```

Prethodni listing pokazuje primenu osluškivanja događana konceptom osmatrača. Za početak, ispaljen je događaj pod nazivom *connection* (linija koda 24). Ovaj događaj je osmatran i obrađen funkcijom koja se naziva *connectHandler()*. Ako se pažljivo pogleda kod navedene funkcije, nakon informacije o uspešnoj konekciji, javlja se ispaljivanje novog događaja pod nazivom *data_received* (linija koda 12). Ovaj događaj je osmatran i obrađen anonimnom funkcijom (linija koda 19 - 21).

U razvojnog oruženju *Visual Studio Code* biće pokrenut terminal i u njemu je potrebno otkucati sledeću naredbu, kojom se pokreće kreirani program:

```
node main.js
```

```
TERMINAL JUPYTER PROBLEMS OUTPUT DEBUG CONSOLE

PS C:\Users\Vlada\Desktop\Node.js - primeri\Primer 2 - OU4> node main.js
uspešna konekcija.
uspešan prijem podataka.
Kraj programa.

PS C:\Users\Vlada\Desktop\Node.js - primeri\Primer 2 - OU4>
```

Slika 4.4 Pokazni primer rukovanja događajem u Node.js [izvor: autor]

▼ Poglavlje 5

Klasa EventEmitter

EMITOVANJE DOGAĐAJA

Brojni objekti emituju događaje.

Mnogi objekti u [Node.js](#) emituju događaje. na primer, `fs.readStream` emituje događaj svaki put kada je odgovarajuća datoteka otvorena. Svi objekti koji emituju događaje su instance klase [EventEmitter](#).

Kao što je prikazano u prethodnom izlaganju, klasa `EventEmitter` pripada modulu pod nazivom `events` i pristupa joj se na način prikazan sledećim listingom:

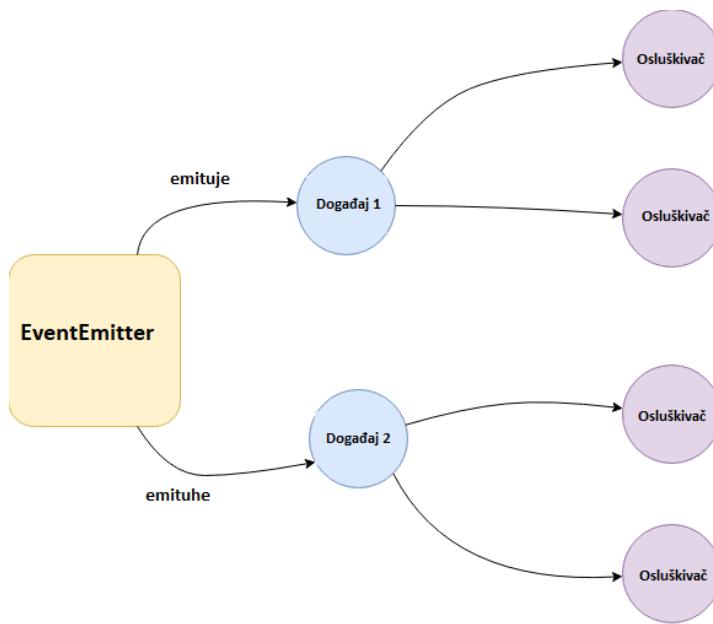
```
// Učitava events modul
var events = require('events');

// Kreira eventEmitter objekat
var eventEmitter = new events.EventEmitter();
```

Ukoliko se instanca tipa `EventEmitter` suoči sa nekom greškom, emitovaće događaj pod nazivom `error`. Kada se registruje nov osluškivač događaja, događaj pod nazivom `newListener` je "ispaljen". U slučaju uklanjanja osluškivača, emituje se događaj pod nazivom `removeListener`.

`EventEmitter` obezbeđuje veći broj osobina, kao što su `on` i `emit`, kojima se vrši povezivanje sa funkcijom sa događajem i emitovanje (ispaljivanje) događaja, respektivno.

Sledećeom slikom je ilustrovan mehanizam emitovanja i osluškivanja događaja u [Node.js](#).



Slika 5.1 Emitovanje događaja [izvor: autor]

METODE KLASE EVENTEMITTER

Klasa sadrži više objektnih i jednu klasnu metodu.

Klasa `EventEmitter` sadrži više objektnih i jednu klasnu metodu.

Sledi lista objektnih metoda zajedno sa njihovim opisima:

- `addListener(event, listener)` - Dodaje osluškivač na kraj niza slušalaca za navedeni događaj. Ne proverava se da li je osluškivač već dodat. Višestruki pozivi koji prosleđuju istu kombinaciju događaja i osluškivača dovešće do toga da se osluškivač dodaje više puta. Vraća emiter, tako da se pozivi mogu povezati.
- `on(event, listener)` - potpuno ista uloga kao u slučaju prethodne metode.
- `once(event, listener)` - Dodaje jednokratnog osluškivača za događaj. Ovaj osluškivač se poziva samo jednom - sledeći put kada se događaj pokrene, nakon čega se uklanja. Vraća emiter, tako da se pozivi mogu povezati.
- `removeListener(event, listener)` - Uklanja osluškivač iz niza osluškivača za navedeni događaj. **Oprez** - menja indeks niza u nizu iza obrisanog osluškivača. Poziv metode `removeListener()` će ukloniti, najviše, jednu instancu osluškivača iz niza. Ako je bilo koji pojedinačni osluškivač dodat više puta u niz za neki događaj, metoda `removeListener()` mora biti pozvana više puta da bi bila uklonjena svaka instanca. Vraća emiter, tako da se pozivi mogu povezati.
- `removeAllListeners([event])` - Briše sve osluškivače za posmatrani događaj.
- `setMaxListeners(n)` - Podrazumevano, EventEmitter će štampati upozorenje ako se za određeni događaj doda više od 10 osluškivača. Ovo je korisna podrazumevana vrednost koja pomaže u pronalaženju "curenja" memorije. Očigledno je da svi emiteri ne bi trebalo

da budu ograničeni na 10. Ova funkcija omogućava da se to poveća. **Vrednost nula znači neograničeno.**

- *listeners(event)* - vraća niz osluškivača koji su registrovani za posmatrani događaj.
- *emit(event, [arg1], [arg2], [...])* - Izvršava sve osluškivače u po redosledu koji je određen navedenim argumentima. Vraća **true** ako je događaj imao osluškivače, a inače **false**.

Klasna metoda klase *Event Emitter* je:

- *listenerCount(emitter, event)* - vraća broj osluškivača za posmatrani događaj.

POKAZNI PRIMER RADA SA OSLUŠKIVAČIMA I DOGAĐAJIMA

Demostracija kreiranja i uklanjanja osluškivača za date događaje.

Kao pokazni primer biće kreiranje JavaScript datoteka sa sledećim listingom:

```
//učitavanje modula događaja i kreiranje emitera
var events = require('events');
var eventEmitter = new events.EventEmitter();

// osluškivač #1
var listner1 = function listner1() {
    console.log('Prvi osluškivač je izvršen.');
}

// osluškivač #2
var listner2 = function listner2() {
    console.log('Drugi osluškivač je izvršen.');
}

// povezivanje događaja connection sa metodom prvog osluškivača
eventEmitter.addListener('connection', listner1);

// povezivanje događaja connection sa metodom drugom osluškivača
eventEmitter.on('connection', listner2);

var eventListeners = require('events').EventEmitter.listenerCount
    (eventEmitter,'connection');
console.log(eventListeners + " osluškivača osluškuje connection događaj");

// Emitovanje connection događaja
eventEmitter.emit('connection');

// uklanjanje prvog osluškivača
eventEmitter.removeListener('connection', listner1);
console.log("Prvi osluškivač je uklonjen.");

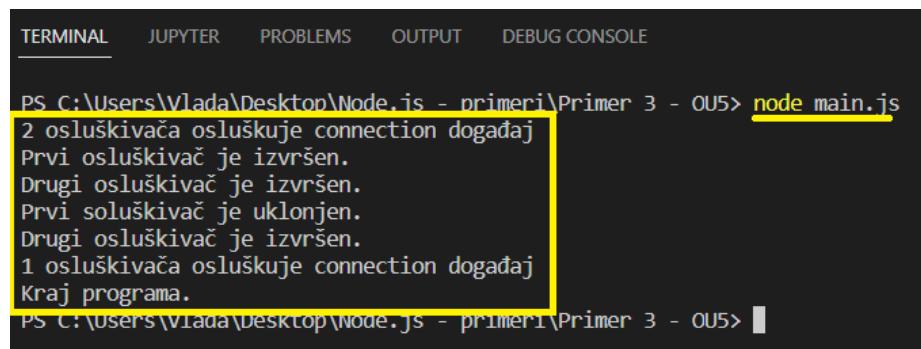
// Emitovanje connection događaja
eventEmitter.emit('connection');
```

```
eventListeners =  
require('events').EventEmitter.listenerCount(eventEmitter,'connection');  
console.log(eventListeners + " osluškivača osluškuje connection događaj");  
  
console.log("Kraj programa.");
```

Krećući se redom po listingu primećuju se sledeće aktivnosti:

- kreiranje emitera događaja (linije koda 2 i 3);
- kreiranje dve osluškivačke metode (linije koda 5 - 13);
- povezivanje događaja sa osluškivačima (linije koda 15 - 23);
- ispaljivanje (emitovanje) događaja (linija koda 26);
- uklanjanje prvog registrovanog osluškivača (linija koda 29);
- ponovno ispaljivanje (emitovanje) događaja (linija koda 33);
- prebrojavanje i vraćanje broja aktivnih osluškivača (linije koda 21 i 35);

Sledećom slikom je demonstrirana funkcionalnost kreiranog koda.



```
TERMINAL JUPYTER PROBLEMS OUTPUT DEBUG CONSOLE  
PS C:\Users\Vlada\Desktop\Node.js - primeri\Primer 3 - 0U5> node main.js  
2 osluškivača osluškuje connection događaj  
Prvi osluškivač je izvršen.  
Drugi osluškivač je izvršen.  
Prvi soluškivač je uklonjen.  
Drugi osluškivač je izvršen.  
1 osluškivača osluškuje connection događaj  
Kraj programa.  
PS C:\Users\Vlada\Desktop\Node.js - primeri\Primer 3 - 0U5>
```

Slika 5.2 Pokazni primer emitovanja događaja [izvor: autor]

▼ Poglavlje 6

Baferi i tokovi u Node.js

KREIRANJE BAFERA

Node.js obezbeđuje klasu Buffer za rad sa objektima koji skladište sirove podatke.

Čist [JavaScript](#) se jasno oslanja na [Unicode](#) podršku, osim u slučaju binarnih podataka. Tokom rada sa [TCP](#) tokovima ili datotečnim sistemom, neophodno je rukovati oktetnim tokovima (<https://isotropic.co/what-is-octet-stream/>). [Node.js](#) obezbeđuje klasu [Buffer](#) za rad sa objektima koji skladište sirove podatke slične nizovima celih brojeva ali odgovaraju alokaciji sirove memorije izvan [V8](#) dinamičke memorije ([heap](#)).

Klasa [Buffer](#) je globalna i moguće joj je pristupiti u aplikaciji bez potrebe za uključivanjem odgovarajućeg modula.

Bafer može biti kreiran na više načina:

1. Sledi sintaksa za kreiranje neiniciranog bafera od 10 okteta:

```
var buf = new Buffer(10);
```

2. Sledi sintaksa za kreiranje bafera iz datog niza:

```
var buf = new Buffer([10, 20, 30, 40, 50]);
```

3. Sledi sintaksa za kreiranje bafera od datog stringa i, opcionalno, tipa kodiranja:

```
var buf = new Buffer("IT255 - Veb sistemi 1", "utf-8");
```

Iako je [utf8](#) podrazumevani sistem kodiranja, moguće je koristiti bilo koji od sledećih kodnih standarda:

- [ascii](#);
- [utf8](#);
- [utf16le](#);
- [ucs2](#);
- [base64](#);
- [hex](#).

NAPOMENA: Ukoliko imate problema sa zastarelošću konstruktora [Buffer\(\)](#), koristiti novu sintaksu: [Buffer.alloc \(broj\)](#)

UPISIVANJE U BAFER

Pregled sintakse metode za upisivanje u bafer.

Sledi sintaksa metode za upisivanje podataka u bafer:

```
buf.write(string[, offset][, length][, encoding])
```

Ako se pogleda prethodni kod, neophodno je dati pojašnjenja u vezi sa primenjenim parametrima:

- *string* – string podatak koji će biti upisan u bafer.
- *offset* – indeks bufera na kojem se počinje upisivanje. Podrazumevano je 0.
- *length* – broj bajtova koji se upisuju. Podrazumevano je *buffer.length*.
- *encoding* – Šema kodiranja. Podrazumevano je *utf8*.

Ova metoda vraća broj napisanih okteta. Ako u baferu nema dovoljno prostora da stane ceo string, on će upisati deo stringa.

Sledi primer koda za upisivanje stringa u bafer:

```
buf = new Buffer.alloc (256);
len = buf.write("IT255 - Veb sistemi 1");

console.log("Broj upisanih okteta : "+ len);
```

Sledećom slikom je prikazan rezultat izvršavanja kreiranog koda:

```
TERMINAL JUPYTER PROBLEMS OUTPUT DEBUG CONSOLE

PS C:\Users\Vlada\Desktop\Node.js - primeri> node main.js
Broj upisanih okteta : 21
PS C:\Users\Vlada\Desktop\Node.js - primeri>
```

Slika 6.1 Upisivanje u bafer [izvor: autor]

ČITANJE IZ BAFERA

Pregled sintakse metode za čitanje iz bafera.

Sledi sintaksa metode za čitanje podataka iz bafera:

```
buf.toString([encoding][, start][, end])
```

Ako se pogleda prethodni kod, neophodno je dati pojašnjenja u vezi sa primenjenim parametrima:

- *encoding* – Šema kodiranja. Podrazumevano je *utf8*.

- *start* - početni indeks čitanja. Podrazumevano je 0,
- *end* - krajnji indeks čitanja. Podrazumevano je ceo bafer.

Ovaj metoda dekodira i vraća string iz podataka bafera kodiranih korišćenjem specificiranog standarda kodiranja.

Sledi primer koda za upisivanje stringa u bafer:

```
buf = new Buffer.alloc (256);
len = buf.write("IT255 - Veb sistemi 1");

console.log("Broj upisanih okteta : "+ len);

//čitanje iz bafera
console.log( buf.toString('ascii'));
console.log( buf.toString('ascii',0,5));
console.log( buf.toString('utf8',0,5));
console.log( buf.toString(undefined,0,5));
```

Sledećom slikom je prikazan rezultat izvršavanja kreiranog koda:

Ukoliko se doda još malo koda:

```
var json = buf.toJSON(buf.toString('ascii'));
console.log(json);
```

```
PS C:\Users\Vlada\Desktop\Node.js - primeri> node main.js
Broj upisanih okteta : 21
IT255 - Veb sistemi 1
IT255
IT255
IT255
PS C:\Users\Vlada\Desktop\Node.js - primeri>
```

Slika 6.2 Čitanje iz bafera [izvor: autor]

Sadržaj bafera može biti prikazan kao JSON objekat:

```
{
  type: 'Buffer',
  data: [
    73, 84, 50, 53, 53, 32, 45,
    32, 86, 101, 98, 32, 115, 105,
    115, 116, 101, 109, 105, 32, 49
  ]
}
```

Slika 6.3 Sadržaj bafera u JSON objektu [izvor: autor]

SPAJANJE, KOPIRANJE I POREĐENJE BAFERA

Spajanje, kopiranje i poređenje bafera su operacije koje će dodatno biti razmatrane.

Spajanje, kopiranje i poređenje bafera su operacije koje će dodatno biti razmatrane.
Sledi sintaksa metode za spajanje bafera:

```
Buffer.concat(list[, totalLength])
```

Ako se pogleda prethodni kod, neophodno je dati pojašnjenja u vezi sa primenjenim parametrima:

- *list* – lista bufer objekata koji se spajaju sa posmatranim baferom.
- *totalLength* - ukupna dužina spojenih bafera.

Metoda vraća objekat tipa *Buffer*.

Sledi sintaksa metode za kopiranje bafera:

```
buf.copy(targetBuffer[, targetStart][, sourceStart][, sourceEnd])
```

Ako se pogleda prethodni kod, neophodno je dati pojašnjenja u vezi sa primenjenim parametrima:

- *targetBuffer* – bafer objekat koji se kopira.
- *targetStart* - opcioni broj, podrazumevano je 0.
- *sourceStart* - opcioni broj, podrazumevano je 0.
- *sourceEnd* - opcioni broj, podrazumevano je buffer.length.

Metoda *copy()* nema povratnu vrednost.

Sledi sintaksa metode za poređenje bafera:

```
buf.compare(otherBuffer);
```

Jedini parametar metode označava bafer koji će biti upoređen sa baferom koji poziva metodu (*buf*).

Metoda vraća broj koji pokazuje da li bafer *baf* dolazi pre, posle ili je isti kao i *otherBuffer* po redosledu sortiranja.

OBJEKTI TOKOVA

U Node.js postoje 4 tipa tokova.

Tokovi (eng. *streams*) predstavljaju objekte koji omogućavaju čitanje podataka iz nekog izvora ili upisivanje istih, na određenu destinaciju, kontinuirano. U *Node.js*, postoje 4 tipa tokova:

- *Readable* – tok koji se koristi za operaciju čitanja.
- *Writable* – tok koji se koristi za operaciju upisivanja.
- *Duplex* – tok koji se koristi istovremeno za operacije čitanja i upisivanja.
- *Transform* – tip duplex toka gde je izlaz izračunat na osnovu vrednosti ulaza.

Svaki od navedenih tipova tokova predstavlja instancu tipa *EventEmitter* i u različitim vremenskim okvirima emituju nekoliko tipova događaja. Najčešće korišćeni događaji su:

- *data* – emituje se kada su podaci dostupni za čitanje.
- *end* – emituje se kada više nema podataka za čitanje.
- *error* – emituje se tokom javljanja greške prilikom čitanja ili upisivanja podataka.
- *finish* – emituje se kada su svi podaci prosleđeni (isprani, eng. *flush*) u posmatrani sistem.

ČITANJE IZ TOKA

Demonstracija primene čitljivog toka.

Za demonstraciju *Node.js* funkcionalnosti čitanja iz tokova uvodi se odgovarajući prateći primer.

Kreira se prvo jedna tekstualna datoteka pod nazivom *input.txt* sa sledećim sadržajem.

```
Predmet IT-255 Veb sistemi 1  
Na predmetu se uči frontend programiranje...
```

U nastavku se gradi konkretna programska logika u formi datoteke *main.js*. Sledi listing ove datoteke.

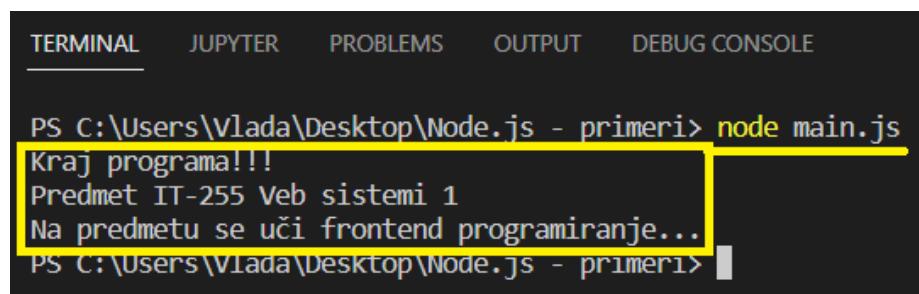
```
var fs = require("fs");  
var data = '';  
  
// kreiranje čitljivog toka  
var readerStream = fs.createReadStream('input.txt');  
  
// podešavanje kodiranja na utf8.  
readerStream.setEncoding('UTF8');  
  
// rukovanje događajima toka --> data, end i error  
readerStream.on('data', function(chunk) {  
    data += chunk;  
});  
  
readerStream.on('end', function() {  
    console.log(data);  
});  
  
readerStream.on('error', function(err) {  
    console.log(err.stack);  
});
```

```
console.log("Kraj programa!!!");
```

Pokazni primer pokazuje detaljno, a na veoma jednostavan način, čitanje podataka iz toka. Nakon učitavanja odgovarajućeg modula i inicijalizacije glavne promenljive (linije koda 1 i 2) pristupljeno je kreiranju toka i podešavanju sistema kodiranja (linije koda 5 i 8). Dalje, pokazano je kako se čita it toka uz obradu specifičnih događaja *data*, *end* i *err*.

Potrebno je napomenuti da se rukovanje navedenim događajima odigrava asinhrono.

Sledećom slikom je prikazano izvršavanje kreiranog pokaznog primera.



```
TERMINAL JUPYTER PROBLEMS OUTPUT DEBUG CONSOLE

PS C:\Users\Vlada\Desktop\Node.js - primeri> node main.js
Kraj programa!!!
Predmet IT-255 Veb sistemi 1
Na predmetu se uči frontend programiranje...
PS C:\Users\Vlada\Desktop\Node.js - primeri>
```

Slika 6.4 Čitanje iz toka (demonstracija) [izvor: autor]

UPISIVANJE U TOK

Demonstracija primene toka za upisivanje podataka.

Za demonstraciju *Node.js* funkcionalnosti upisivanja u tokove uvodi se odgovarajući prateći primer.

U nastavku se gradi konkretna programska logika u formi datoteke *main.js*. Sledi listing ove datoteke.

```
var fs = require("fs");
var data = 'IT255 - Veb sistemi 1';

// kreiranje toka za upisivanje
var writerStream = fs.createWriteStream('output.txt');

// podešavanje kodiranja za upisivanje u tok na utf8
writerStream.write(data,'UTF8');

// oznaka kraja datoteke
writerStream.end();

// rukovanje događajima toka --> finish i error
writerStream.on('finish', function() {
    console.log("Završeno upisivanje.");
});

writerStream.on('error', function(err) {
    console.log(err.stack);
```

```
});  
  
console.log("Kraj programa!!!");
```

Pokazni primer pokazuje detaljno, a na veoma jednostavan način, upisivanje podataka u tok. Nakon učitavanja odgovarajućeg modula i inicijalizacije glavne promenljive (linije koda 1 i 2) pristupljeno je kreiranju toka i podešavanju sistema kodiranja (linije koda 5 i 8). Dalje, pokazano je kako se upisuje u tok uz obradu specifičnih događaja *finish* i *err*.

Potrebno je napomenuti da se rukovanje navedenim događajima odigrava asinhrono.

Sledećom slikom je prikazano izvršavanje kreiranog pokaznog primera.

```
TERMINAL JUPYTER PROBLEMS OUTPUT DEBUG CONSOLE  
  
PS C:\Users\Vlada\Desktop\Node.js - primeri> node main.js  
Kraj programa!!!  
Završeno upisivanje.  
PS C:\Users\Vlada\Desktop\Node.js - primeri>
```

Slika 6.5 Upisivanje u tok (demonstracija) [izvor: autor]

POVEZIVANJE TOKOVA KROZ CEVOVOD (PAJP)

Povezivanje tokova je moguće uraditi na više načina.

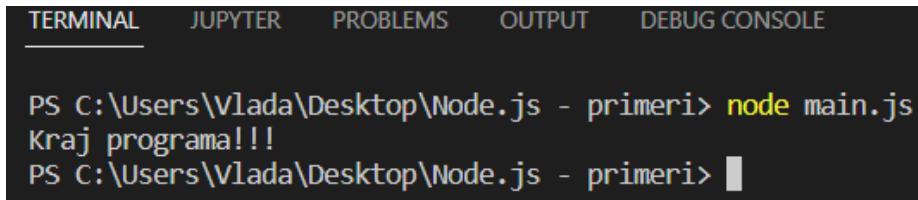
Povezivanje tokova je moguće uraditi na više načina:

- koncept cevovoda (eng. [pipe](#));
- ulančavanje tokova.

Princip cevovoda (postoji odomaćen izraz [piping](#) - eng. [piping](#)) odnosi se na primenu mehanizma u kome se obezbeđuje izlaz jednog toka kao ulaz za drugi tok. Obično se koristi za dobijanje podataka iz jednog toka i za prosleđivanje izlaza tog toka u drugi tok. Nema ograničenja u radu cevovoda. Sledi primer cevovoda za čitanje iz jedne datoteke i pisanje u drugu datoteku.

```
var fs = require("fs");  
  
// kreiranje toka za čitanje  
var readerStream = fs.createReadStream('input.txt');  
  
// kreiranje toka za upisivanje  
var writerStream = fs.createWriteStream('output.txt');  
  
// operacije čitanja i upisivanja putem pajpa  
// čita datoteku input.txt i upisuje podatke u output.txt  
readerStream.pipe(writerStream);  
  
console.log("Kraj programa!!!");
```

Sledećom slikom je prikazano izvršavanje kreiranog pokaznog primera.

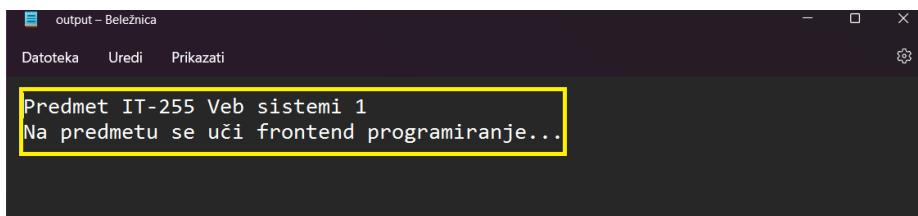


```
TERMINAL JUPYTER PROBLEMS OUTPUT DEBUG CONSOLE

PS C:\Users\Vlada\Desktop\Node.js - primeri> node main.js
Kraj programa!!!
PS C:\Users\Vlada\Desktop\Node.js - primeri>
```

Slika 6.6 Povezivanje tokova putem pajpa [izvor: autor]

Dodatno, proverava se da li je zaista prikazani zadatak uspešno obavljen, odnosno, da li je sadržaj datoteke *output.txt* identičan sadržaju datoteke *input.txt*.



Slika 6.7 Sadržaj datoteke output.txt [izvor: autor]

ULANČAVANJE TOKOVA

Mehanizam za stvaranje lanca višestrukih operacija toka.

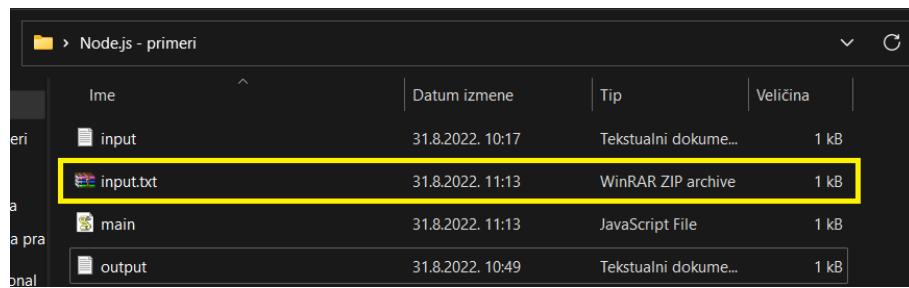
Ulančavanje predstavlja mehanizam za povezivanje izlaza jednog toka sa drugim tokom i stvaranje lanca višestrukih operacija toka. Obično se koristi za operacije cevovoda. Sledi primer primene cevovoda i ulančavanja. Za početak je potrebno komprimovati datoteku (prvi listing), a zatim obaviti njeno raspakivanje (drugi listing).

```
var fs = require("fs");
var zlib = require('zlib');

// komprimovanje datoteke input.txt u input.txt.gz
fs.createReadStream('input.txt')
  .pipe(zlib.createGzip())
  .pipe(fs.createWriteStream('input.txt.zip'));

console.log("ZIP je kreiran.");
```

Nakon učitavanja dodatnog modula (linija koda 2) i ulančavanja (linije koda 6 i 7) primer je moguće pokrenuti i rezultat je prikazan sledećom slikom.



Slika 6.8 Ulančavanje tokova (prvi deo) [izvor: autor]

Sledi drugi deo koda za raspakivanje kreirane ZIP arhive.

```
var fs = require("fs");
var zlib = require('zlib');

// raspakivanje datoteke input.txt.zip to input.txt
fs.createReadStream('input.txt.zip')
  .pipe(zlib.createGunzip())
  .pipe(fs.createWriteStream('input.txt'));

console.log("Datoteka je raspakovana.");
```

Sa poslednjim listingom je zaokružena diskusija u vezi rada sa tokovima u [Node.js](#) okruženju.

✓ Poglavlje 7

Web modul

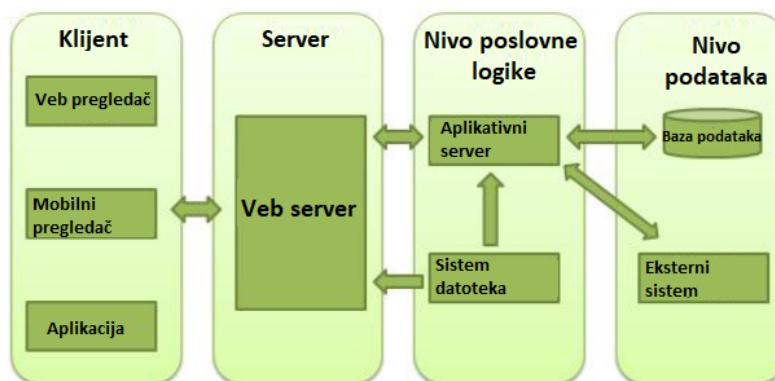
NODE.JS VEB SERVER

Veb server je aplikacija koja rukuje HTTP zahtevima sa ciljem kreiranja odgovora.

Veb server predstavlja računarski program koji rukuje HTTP zahtevima prosleđenim od strane HTTP klijenta, poput veb pregledača, i vraća veb stranice kao odgovor klijentu. Po običaju, veb server klijentu dostavlja HTML dokument zajedno sa slikama, stilovima i skriptovima.

Većina veb servera podržava primenu serverskih skriptova, kreiranih nekim od poznatih skripting jezika, ili vrši preusmeravanje zadataka ka aplikativnim serverima koji preuzimaju podatke iz baze podataka i izvode kompleksnu logiku nakon čega šalju rezultate HTTP klijentima preko veb servera. Apache web server je jedan od najčešće korišćenih veb servera koji, takođe, predstavlja projekat otvorenog koda.

Sledećom slikom je prikazan sistem veb aplikacije.



Slika 7.1 Sistem veb aplikacije[izvor: autor]

- **Klijent** – Nivo sadrži veb i mobilne pregledače, kao i druge aplikacije koje mogu da upute HTTP zahteve ka veb serveru.
- **Server** – Veb server koji presreće zahteve i prosleđuje odgovore klijentima.
- **Poslovna logika** – Nivo sadrži aplikativni server kojeg angažuje veb server za obradu. Ovaj nivo interaguje sa nivoom podataka koristeći bazu podataka ili neki eksterni softver.
- **Podaci** – Baze podataka i/ili drugi izvori informacija.

KREIRANJE VEB SERVERA POMOĆU NODE.JS

Node.js obezbeđuje modul pod nazivom http za kreiranje HTTP klijenta servera.

Node.js obezbeđuje modul pod nazivom `http` koji može jednostavno da bude upotrebljen za kreiranje `HTTP` klijenta servera. Za ilustraciju biće kreiran `HTTP` server sa minimalnom strukturom koji osluškuje na portu 8080. Server odgovara datoteci `server.js` sa sledećim listingom:

```
var http = require('http');
var fs = require('fs');
var url = require('url');

// Kreira server
http.createServer( function (request, response) {
    // Parsira zahtev koji sadrži ime datoteke
    var pathname = url.parse(request.url).pathname;

    // Štampa naziv fajla za kojim je zahtev napravljen.
    console.log("Zahtev za: " + pathname + " je primljen.");

    // Čita sadržaj traženog fajla iz fajl sistema
    fs.readFile(pathname.substr(1), function (err, data) {
        if (err) {
            console.log(err);

            // HTTP Status: 404 : NOT FOUND
            // Content Type: text/plain
            response.writeHead(404, {'Content-Type': 'text/html'});
        } else {
            //Pronađena stranica
            // HTTP Status: 200 : OK
            // Content Type: text/plain
            response.writeHead(200, {'Content-Type': 'text/html'});

            // štampa sadržaj fakla kao telo odgovora
            response.write(data.toString());
        }

        // Šalje telo odgovora
        response.end();
    });
}).listen(8080);

// Poruka u konzoli
console.log('Server je pokrenut na http://127.0.0.1:8080/');
```

Linije koda 1 - 3 uključuju potrebne module. Objekat (varijabla) `http` kreira server i prosleđuje mu zahtev. Zahtev se parsira i sadržaj se čuva u promenljivoj `fs` (iz modula `fs` - FILE SYSTEM).

Tokom čitanja fajla, proverava se da li takva datoteka postoji ili ne. U skladu sa navedenim, server vraća status *200 : OK* ili *404 : NOT FOUND*, tim redosledom. Ako fajl postoji, kreira se telo odgovora i server odgovara klijentu. Konačno, server osluškuje na portu 8080.

Sledi listing klijenta:

```
<html>
  <head>
    <title>Primer Node.js servera</title>
  </head>

  <body>
    Prva web aplikacija!
  </body>
</html>
```

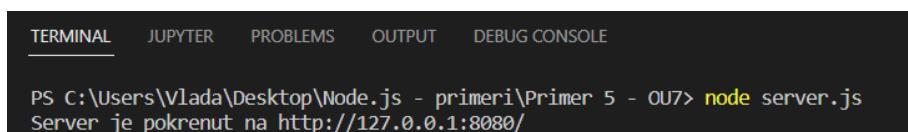
POKRETANJE SERVERA I PROVERA REZULTATA

U Node.js pokreće se kreirani serverski skript.

U Node.js pokreće se kreirani serverski skript navođenjem naredbe:

```
node server.js
```

U terminalu se pojavljuje sledeći izlaz:



```
TERMINAL JUPYTER PROBLEMS OUTPUT DEBUG CONSOLE
PS C:\Users\Vlada\Desktop\Node.js - primeri\Primer 5 - OUT> node server.js
Server je pokrenut na http://127.0.0.1:8080/
```

Slika 7.2 Server je pokrenut [izvor: autor]

Server je pokrenut i osluškuje na portu 8080.

Dalje, u veb pregledaču upućuje se zahtev za stranicom index.html na sledeći način:

```
http://localhost:8081/index.html
```

Odgovor servera je moguće pratiti na dva načina:

- putem veb pregledača;
- u konzoli.



Prva web aplikacija!

Slika 7.3 Odgovor servera u veb pregledaču [izvor: autor]

```
TERMINAL JUPYTER PROBLEMS OUTPUT DEBUG CONSOLE

PS C:\Users\Vlada\Desktop\Node.js - primeri\Primer 5 - OUT> node server.js
Server je pokrenut na http://127.0.0.1:8080/
Zahtev za: /index.html je primljen.
```

Slika 7.4 Odgovor servera u konzoli [izvor: autor]

KREIRANJE VEB KLIJENTA U NODE.JS

Primer demonstrira kreiranje veb klijenta primenom http modula.

Prethodno diskutovani primer biće proširen novom datotekom pod nazivom *client.js*. Sledi njen listing:

```
var http = require('http');

// opcije koje se koriste u zahtevu
var options = {
  host: 'localhost',
  port: '8080',
  path: '/index.html'
};

// Funkcija povratnog poziva koja rukuje zahtevom
var callback = function(response) {
  // Kontinuirano ažurira tok podataka
  var body = '';
  response.on('data', function(data) {
    body += data;
  });

  response.on('end', function() {
    // Podaci su u potpunosti primljeni.
    console.log(body);
  });
}

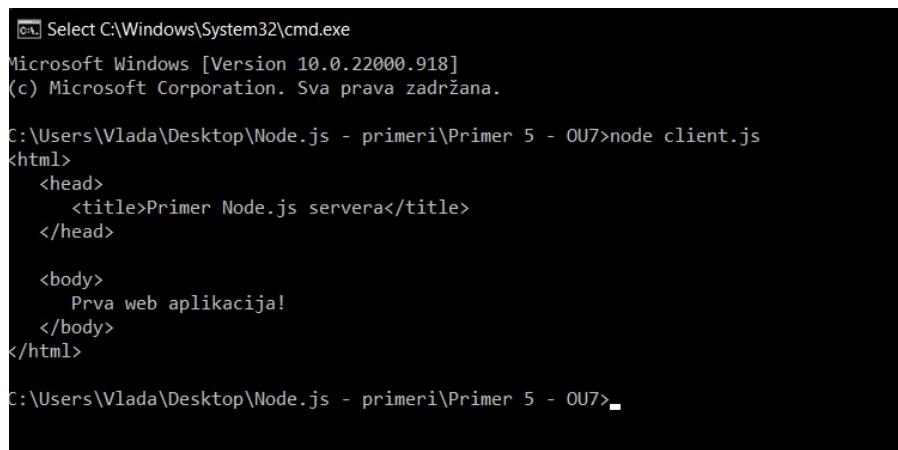
// Kreira odgovor ka serveru
var req = http.request(options, callback);
req.end();
```

Primer demonstrira kreiranje veb klijenta primenom *http* modula.

Server će biti pokrenut na potpuno isti način kao u prethodnom slučaju, a preko druge komandne linije, npr iz *Command Prompt*-a pokreće se klijent komandom:

```
node client.js
```

Klijent produkuje izlaz kao na sledećoj slici:



```
PS C:\Windows\System32\cmd.exe
Microsoft Windows [Version 10.0.22000.918]
(c) Microsoft Corporation. Sva prava zadržana.

C:\Users\Vlada\Desktop\Node.js - primeri\Primer 5 - 0U7>node client.js
<html>
  <head>
    <title>Primer Node.js servera</title>
  </head>

  <body>
    Prva web aplikacija!
  </body>
</html>

C:\Users\Vlada\Desktop\Node.js - primeri\Primer 5 - 0U7>
```

Slika 7.5 Izlaz klijenta [izvor: autor]

Server osluškuje klijenta na portu 8080, sledeća slika:



```
TERMINAL JUPYTER PROBLEMS OUTPUT DEBUG CONSOLE

Windows PowerShell
Copyright (C) Microsoft Corporation. All rights reserved.

Install the latest PowerShell for new features and improvements! https://aka.ms/PSWindows

PS C:\Users\Vlada\Desktop\Node.js - primeri\Primer 5 - 0U7> node server.js
Server je pokrenut na http://127.0.0.1:8080/
Zahtev za: /index.html je primljen.
```

Slika 7.6 Izlaz servera [izvor: autor]

▼ Poglavlje 8

Node.js RESTful API

REST ARHITEKTURA

REST predstavlja arhitekturu baziranu na veb standardima.

REST predstavlja skraćenicu za *Representational State Transfer* i radi se o arhitekturi baziranoj na veb standardima. REST koristi HTTP protokol. Kompletna ideja primene REST servisa vrati se oko resursa gde je svaka komponenta resurs kojem se pristupa preko zajedničkog interfejsa koristeći HTTP standardne metode.

REST server pruža pristup resursima, a REST klijent pristupa i modificuje resurse koristeći HTTP protokol. Ovde je svaki resurs identifikovan pomoću URI linkova. REST koristi različite reprezentacije za predstavljanje resursa kao što je tekst, JSON ili XML. Danas je, svakako, JSON je najpopularniji.

Sledeće četiri HTTP metode su najzastupljenije u implementaciji REST servisa:

- GET – pružanje pristupa resursu samo za čitanje.
- PUT – kreiranje novog resursa.
- DELETE – uklanjanje resursa.
- POST – ažuriranje postojećeg ili kreiranje novog resursa.

KREIRANJE API ZA RESTFUL VEB SERVISE

Veb servisi zasnovani na REST arhitekturi poznati su kao RESTful veb servisi.

Veb usluga, ili servis, predstavlja skup otvorenih protokola i standarda koji se koriste za razmenu podataka između aplikacija i/ili sistema. Veb servisi zasnovani na REST arhitekturi poznati su kao RESTful veb servisi. Ovi veb servisi koriste HTTP metode za implementaciju koncepta REST arhitekture. RESTful veb usluga obično definiše URI, jedinstveni identifikator resursa, koja obezbeđuje reprezentaciju resursa kao što je JSON i skup HTTP metoda.

Danas se RESTful veb servisi obično smatraju posrednikom između klijenta i izvora podataka (npr. baze podataka) koji se nalazi na serverskoj strani aplikacije (backend). U navedenom svetu, izlaganje će početi demonstraciju primene RESTful servisa od samog izvora podataka. Neka to bude sledeća baza podataka bazirana na JSON notaciji.

```
{  
  "user1" : {
```

```

    "name" : "Vlada",
    "password" : "1234",
    "profession" : "profesor",
    "id": 1
  },

  "user2" : {
    "name" : "Milos",
    "password" : "4321",
    "profession" : "asistent",
    "id": 2
  },

  "user3" : {
    "name" : "Pera",
    "password" : "0000",
    "profession" : "student",
    "id": 3
  }
}

```

Na osnovu strukture samog izvora podataka vrši se kreiranje programskog interfejsa (API). U konkretnom slučaju API je definisan sledećom tabelom.

Redni broj	URI	HTTP metoda	Atribut body	Rezultat
1	/listUsers	GET	prazan	Lista svih User objekata
2	/addUser	POST	JSON string	Dodat novi User objekat
3	/deleteUser	DELETE	JSON string	Obrisani User objekat
4	:id	GET	prazan	User objekat koji odgovara traženom id

Slika 8.1 RESTful API [izvor: autor]

IMPLEMENTACIJA RESTFUL SERVISA U NODE.JS PRIMENOM GET METODE

GET metoda vraća sve korisnike iz baze podataka.

Na osnovu kreirane tabele, prethodna slika, moguće je početi sa implementacijom **RESTful** servisa i neka to za početak bude servis koji vraća sve korisnike iz baze podataka. Ovaj servis koristi metodu **GET** i prvi **URI** iz prethodne tabele (linije koda 5 - 10).

Server koji rukuje **REST GET** zahtevom prikazan je sledećim listingom:

```

var express = require('express');
var app = express();
var fs = require("fs");

app.get('/listUsers', function (req, res) {
  fs.readFile( __dirname + "/" + "users.json", 'utf8', function (err, data) {
    console.log( data );
    res.end( data );
  });
})

```

```
var server = app.listen(8080, function () {  
    var host = server.address().address  
    var port = server.address().port  
    console.log("Primer slušanja na http://%s:%s", host, port)  
})
```

Linije koda 12 - 16 ukazuju da server osluškuje zahteve na portu 8080. Na dobro poznat način, u [Node.js](#) (slika 2), biće pokrenut kreirani server, a nakon toga upućen zahtev primenom URI linka:

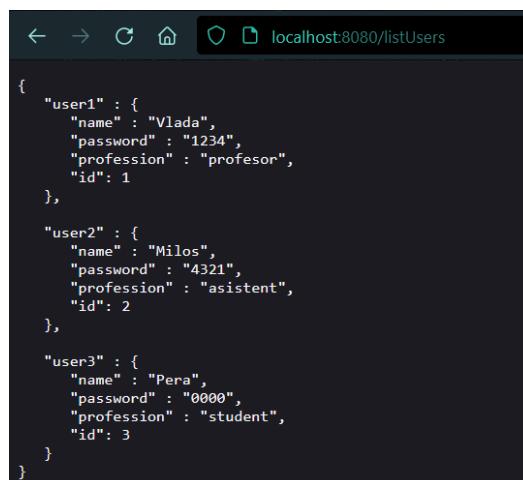
```
http://127.0.0.1:8080/listUsers  
ili  
http://localhost:8080/listUsers
```

```
http://127.0.0.1:8080/listUsers  
ili  
http://localhost:8080/listUsers
```



```
TERMINAL JUPYTER PROBLEMS OUTPUT DEBUG CONSOLE  
Windows PowerShell  
Copyright (c) Microsoft Corporation. All rights reserved.  
Install the latest PowerShell for new features and improvements! https://aka.ms/PSWindows  
PS C:\Users\Vlada\Desktop\Node.js - primeri\Primer 6 - 0U8> node server.js  
Primer slušanja na http://:::8080
```

Slika 8.2 Pokretanje servera na Node.js [izvor: autor]



Slika 8.3 Rezultat GET zahteva [izvor: autor]

PRIMENA POST METODE

POST metodom se dodaju novi resursi u bazu podataka.

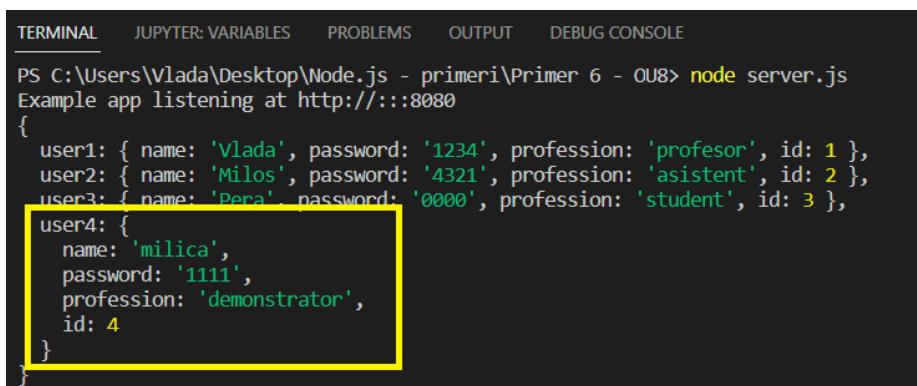
POST metodom se dodaju novi resursi u bazu podataka.

```
var user = {  
    "user4" : {  
        "name" : "mohit",  
        "password" : "password4",  
        "profession" : "teacher",  
        "id": 4  
    }  
}  
  
app.post('/addUser', function (req, res) {  
    // Čitanje postojećih korisnika.  
    fs.readFile( __dirname + "/" + "users.json", 'utf8', function (err, data) {  
        data = JSON.parse( data );  
        data["user4"] = user["user4"];  
        console.log( data );  
        res.end( JSON.stringify(data));  
    });  
})
```

Prethodnim listingom objekat pod nazivom user4 (linije koda 1 - 7) metodom POST (linije koda 10 -17) se dodaje u bazu podataka REST pozivom:

```
http://localhost:8080/addUser
```

Poznato je da server osluškuje zahteve na portu 8080. Na dobro poznat način, u [Node.js](#) (slika 4), biće pokrenut kreirani server, a nakon toga upućen zahtev primenom prethodnog URI linka. Sledi rezultat:



```
TERMINAL JUPYTER: VARIABLES PROBLEMS OUTPUT DEBUG CONSOLE  
PS C:\Users\Vlada\Desktop\Node.js - primeri\Primer 6 - 0U8> node server.js  
Example app listening at http://:8080  
{  
  user1: { name: 'Vlada', password: '1234', profession: 'profesor', id: 1 },  
  user2: { name: 'Milos', password: '4321', profession: 'asistent', id: 2 },  
  user3: { name: 'Pera', password: '0000', profession: 'student', id: 3 },  
  user4: {  
    name: 'milica',  
    password: '1111',  
    profession: 'demonstrator',  
    id: 4  
}
```

Slika 8.4 Primena POST metode [izvor: autor]

PRIMENA DELETE METODE

DELETE metodom se brišu postojeći resursi iz baze podataka.

Implementacija **REST** poziva **DELETE** je veoma slična pozivu **GET** (odgovara **URI** linku **/addUser**). Dešava se prijem ulaznih podataka putem atributa **body** (zahtev) nakon čega se obavlja brisanje konkretnog objekta iz baze podataka na osnovu njegovog identifikatora (**ID**). Za potrebe primera pretpostavlja se da je potrebno obrisati korisnika čiji je ID 2.

U navedenom svetu datoteka **server.js** biće dopunjena sledećim listingom:

```
app.delete('/deleteUser', function (req, res) {
  // Prvo čita postojeće korisnike.
  fs.readFile( __dirname + "/" + "users.json", 'utf8', function (err, data) {
    data = JSON.parse( data );
    delete data["user" + 2];

    console.log( data );
    res.end( JSON.stringify(data));
  });
})
```

Na početku je dfinisana promenljiva koja odgovara identifikatoru korisnika koji će biti obrisan. Linije koda 3 - 11 deinišu metodu koja kao argumente uzima URI i funkciju povratnog poziva za brisanje željenog korisnika iz baze podataka.

Poznato je da server osluškuje zahteve na portu 8080. Na dobro poznat način, u **Node.js** (slika 4), biće pokrenut kreirani server, a nakon toga upućen zahtev primenom **/deleteUser** URI linka. Sledi rezultat:

```
TERMINAL JUPYTER PROBLEMS OUTPUT DEBUG CONSOLE
PS C:\Users\Vlada\Desktop\Node.js - primeri\Primer 6 - 0U8> node server.js
Primer slušanja na http://:::8080
{
  user1: { name: 'Vlada', password: '1234', profession: 'profesor', id: 1 },
  user3: { name: 'Pera', password: '0000', profession: 'student', id: 3 }
}
```

Slika 8.5 Primena DELETE metode [izvor: autor]

VRAĆANJE POJEDINAČNIH RESURSA IZ BAZE PODATAKA

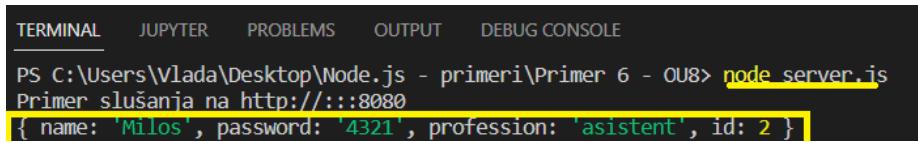
*Vraćanje pojedinačnih resursa iz baze podataka je u domenu **GET** metode koja koristi ID resursa.*

Vraćanje pojedinačnih resursa iz baze podataka je u domenu **GET** metode koja koristi ID resursa. U navedenom svetu datoteka **server.js** biće dopunjena sledećim listingom:

```
app.get('/:id', function (req, res) {
  // First read existing users.
  fs.readFile( __dirname + "/" + "users.json", 'utf8', function (err, data) {
    var users = JSON.parse( data );
    var user = users["user" + req.params.id]
    console.log( user );
    res.end( JSON.stringify(user));
})
```

```
});  
})
```

Poznato je da server osluškuje zahteve na portu 8080. Na dobro poznat način, u [Node.js](#) (slika 4), biće pokrenut kreirani server, a nakon toga upućen zahtev primenom `localhost:8080/2` URI linka. Sledi rezultat:



The screenshot shows a terminal window with the following text:
TERMINAL JUPYTER PROBLEMS OUTPUT DEBUG CONSOLE
PS C:\Users\Vlada\Desktop\Node.js - primeri\Primer 6 - 0U8> `node_server.js`
Primer slušanja na <http://:::8080>
`{ name: 'Milos', password: '4321', profession: 'asistent', id: 2 }`

Slika 8.6 Primena GET metode za pojedinačne rezultate [izvor: autor]

▼ Poglavlje 9

Pokazna vežba (predviđeno vreme 45 min)

BAZA PODATAKA I API

Baza podataka predstavlja udaljeni JSON dokument.

Baza podataka će biti predstavljena **JSON** dokumentom pod nazivom **products.json**. Predviđeno je da studenti kreiraju folder PokaznaVezba5 i unjemu da kreiraju pomenuti fajl. Sledi njegov listing:

```
[{  
    "id": 1,  
    "product_name": "Hard Disk",  
    "supplier": "Win Win",  
    "quantity": 261,  
    "unit_cost": "5000"  
, {  
    "id": 2,  
    "product_name": "Tastatura",  
    "supplier": "Gigatron",  
    "quantity": 292,  
    "unit_cost": "1200"  
, {  
    "id": 3,  
    "product_name": "Miš",  
    "supplier": "Tehnomedia",  
    "quantity": 211,  
    "unit_cost": "950"  
}]
```

Biće definisan API sa odgovarajućim URI linkovima na sledeći način:

- **/productList (GET)** - vraća listu svih dostupnih proizvoda;
- **/id (GET)** - vraća konkretni proizvod na osnovu ID;
- **/deleteProduct (DELETE)** - briše željeni proizvod iz baze podataka;
- **/addProduct (POST)** - dodaje nov proizvod u bazu podataka.

KREIRANJE SERVERA

Serverski skript odgovara datoteci server.js

Serverski skript odgovara datoteci *server.js*. Za početak je predviđeno da server uključi nekoliko modula:

- **express** - lagan i veoma brz veb radni okvir za *Node.js*;
- **fs** - modul za rukovanje datotekama (*File System*).

Upravo iz navedenog razloga dodajemo u kreiranu datoteku, sledeći listing:

```
var express = require('express');
var app = express();
var fs = require("fs");

//ovde ide ostatak koda

var server = app.listen(8080, function () {
    var port = server.address().port
    console.log("Primer slušanja na portu %s", port)
})
```

Linije koda 7 - 10 definišu funkciju povratnog poziva koja omogućava da server osluškuje zahteve na konkretnom portu - 8080.

GET ZAHTEVI

GET zahtevi su realizovani na dva načina.

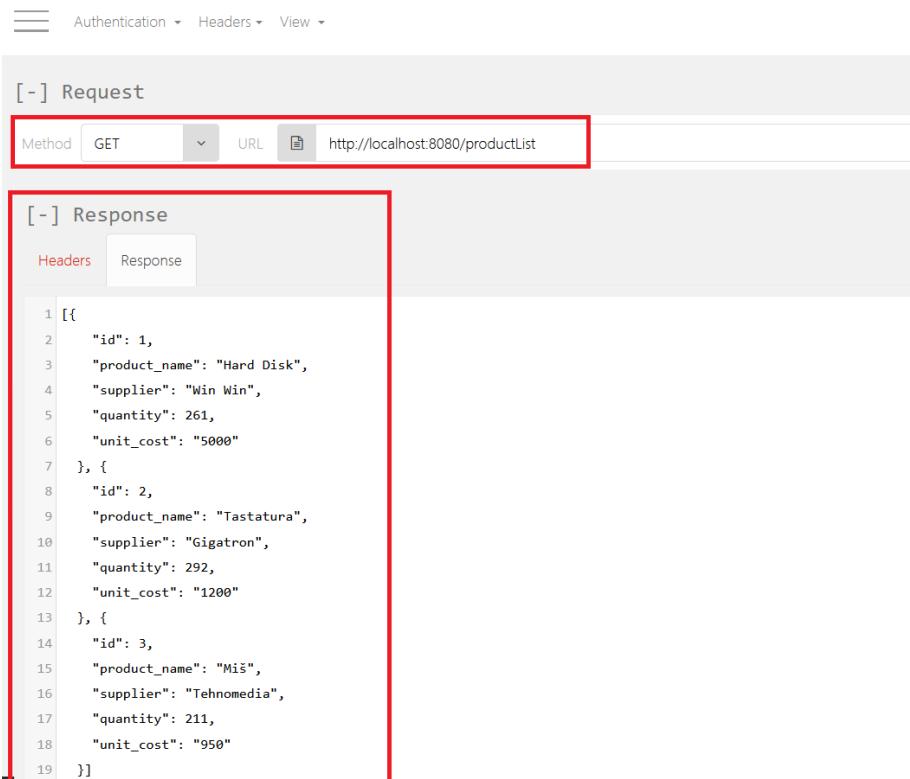
GET zahtevi su realizovani na dva načina:

1. vraćanje svih rezultata iz baze podataka;
2. vraćanje pojedinačnih rezultata, na osnovu ID proizvoda.

Biće proširen listing datoteke *server.js*, sa ciljem omogućavanja poziva *GET* za vraćanje liste svih dostupnih proizvoda:

```
app.get('/productList', function (req, res) {
  fs.readFile( __dirname + "/" + "products.json", 'utf8', function (err, data) {
    console.log( data );
    res.end( data );
  });
})
```

Pokretanjem servera i pozivom linka <http://localhost:8080/productList>, metoda vraća JSON reprezentaciju svih proizvoda iz baze podataka. Za demonstraciju biće upotrebljen REST klijent dodatak za veb pregledač.



Slika 9.1 Lista svih dostupnih proizvoda [izvor: autor]

VRAĆANJE POJEDINAČNIH PROIZVODA

Vraćanje pojedinačnih rezultata, na osnovu ID proizvoda

Vraćanje pojedinačnih rezultata, na osnovu ID proizvoda moguće je dodavanjem sledeće metode u datoteku servera:

```
app.get('/:id', function (req, res) {  
  // Prvo čita postojeće proizvode.  
  fs.readFile( __dirname + "/" + "products.json", 'utf8', function (err, data) {  
    var products = JSON.parse( data );  
    var product = products[req.params.id]  
    console.log( product );  
    res.end( JSON.stringify(product));  
  });  
})
```

Pokretanjem servera i pozivom linka <http://localhost:8080/1>, metoda vraća JSON reprezentaciju proizvoda, iz baze podataka čiji je, ID 1. Za demonstraciju biće upotrebljen REST klijent dodatak za veb pregledač.

The screenshot shows a REST client interface. At the top, there's a 'Request' section with a 'Method' dropdown set to 'GET', a 'URL' input field containing 'http://localhost:8080/1', and a large red box highlighting both the method and URL fields. Below this is a 'Body' section labeled 'Request Body' with a text area containing 'Request Body'. Underneath is a 'Response' section with tabs for 'Headers' and 'Response'. A red box highlights the 'Response' tab. The response body contains the JSON data:

```
{"id":2,"product_name":"Tastatura","supplier":"Gigatron","quantity":292,"unit_cost":1200}
```

.

Slika 9.2 Vraćanje proizvoda na osnovu ID [izvor: autor]

DODAVANJE NOVOG PROIZVODA

POST pozivom se vršni dodavanje novog proizvoda u bazu podataka

POST pozivom se vršni dodavanje novog proizvoda u bazu podataka. Nephodno je u datoteku servera dodati novu metodu:

```
app.post('/addProduct', function (req, res) {
  // Prvo čita postojeće proizvode.
  fs.readFile( __dirname + "/" + "products.json", 'utf8', function (err, data) {
    data = JSON.parse( data );
    data[product.id-1] = product
    console.log( data );
    res.end( JSON.stringify(data));
  });
})
```

Pokretanjem servera i pozivom linka <http://localhost:8080/addProduct>, metoda vraća JSON reprezentaciju svih proizvoda, iz baze podataka, uključujući i dodati proizvod. Za demonstraciju biće upotrebljen REST klijent dodatak za veb pregledač.

The screenshot shows a REST client interface. At the top, there's a 'Request' section with a 'Method' dropdown set to 'POST', a 'URL' input field containing 'http://localhost:8080/addProduct', and a large red box highlighting both the method and URL fields. Below this is a 'Body' section labeled 'Request Body' with a text area containing 'Request Body'. Underneath is a 'Response' section with tabs for 'Headers' and 'Response'. A red box highlights the 'Response' tab. The response body contains the JSON data:

```
[{"id":1,"product_name":"Hard Disk","supplier":"Win Win","quantity":261,"unit_cost":5000}, {"id":2,"product_name":"Tastatura","supplier":"Gigatron","quantity":292,"unit_cost":1200}, {"id":3,"product_name":"Miš","supplier":"Tehnmedia","quantity":211,"unit_cost":950}, {"id":4,"product_name":"SSD","supplier":"ALIT","quantity":100,"unit_cost":15000}]
```

.

Slika 9.3 Dodavanje novog proizvoda [izvor: autor]

BRISANJE POSTOJEĆEG PROIZVODA

*Vežba zaključuje diskusiju kreiranjem metode za **DELETE REST** poziv.*

Vežba zaključuje diskusiju kreiranjem metode za **DELETE REST** poziv. Nephodno je u datoteku servera dodati novu metodu:

```
var id = 2;

app.delete('/deleteProduct', function (req, res) {
    // Prvo čita postojećeg korisnika.
    fs.readFile( __dirname + "/" + "products.json", 'utf8', function (err, data) {
        data = JSON.parse( data );
        delete data["" + (id-1)];

        console.log( data );
        res.end( JSON.stringify(data));
    });
})
```

Pokretanjem servera i pozivom linka <http://localhost:8080/deleteProduct>, metoda vraća **JSON** reprezentaciju svih proizvoda, iz baze podataka, izuzev obrisanog proizvoda. Za demonstraciju biće upotrebljen **REST** klijent dodatak za veb pregledač.



Slika 9.4 Brisanje postojećeg proizvoda [izvor: autor]

▼ Poglavlje 10

Individualna vežba (predviđeno vreme 90 min)

SAMOSTALNA IZRADA ZADATKA

Primena znanja stečenog na predavanjima i pokaznim vežbama.

Na osnovu znanja stečenog na predavanjima i pokaznim vežbama, pokušajte samostalno da obavite sledeće zadatke:

1. Kreirajte JSON bazu podataka na osnovu priložene šeme;
2. Kreirajte serverski skript koji koristi fajl sistem i express radni okvir;
3. Kreirajte API za CRUD REST operacije nad bazom podataka;
4. Pokrenite kreirani server
5. Testirajte REST pozive.

Ukoliko imate poteškoća sa izradom primera, kontaktirajte vašeg predmetnog asistenta.

Sledećim listingom je data šema baze podataka.

```
0
userId 1
id 1
title "sunt aut facere repellat provident occaecati excepturi optio reprehenderit"
body "quia et suscipit\nsuscipit recusandae consequuntur expedita et cum\
nreprehenderit molestiae ut ut quas totam\nnostrum rerum est autem sunt rem eveniet
architecto"

1
userId 1
id 2
title "qui est esse"
body "est rerum tempore vitae\nsequi sint nihil reprehenderit dolor beatae ea
dolores neque\nfugiat blanditiis voluptate porro vel nihil molestiae ut reiciendis\
nqui aperiam non debitis possimus qui neque nisi nulla"

2
userId 1
id 3
title "ea molestias quasi exercitationem repellat qui ipsa sit aut"
body "et iusto sed quo iure\nvoluptatem occaecati omnis eligendi aut ad\
nvoluptatem doloribus vel accusantium quis pariatur\nmolestiae porro eius odio et
labore et velit aut"
```

```
3
userId 1
id 4
title "eum et est occaecati"
body "ullam et saepe reiciendis voluptatem adipisci\nsit amet autem assumenda
provident rerum culpa\nquis hic commodi nesciunt rem tenetur doloremque ipsam iure\
nquis sunt voluptatem rerum illo velit"

4
userId 1
id 5
title "nesciunt quas odio"
body "repudiandae veniam quaerat sunt sed\nalias aut fugiat sit autem sed est\
nvoluptatem omnis possimus esse voluptatibus quis\nest aut tenetur dolor neque"

5
userId 1
id 6
title "dolorem eum magni eos aperiam quia"
body "ut aspernatur corporis harum nihil quis provident sequi\nmollitia nobis
aliquid molestiae\nperspiciatis et ea nemo ab reprehenderit accusantium quas\
nvoluptate dolores velit et doloremque molestiae"
```

✓ Poglavlje 11

Domaći zadatak (predviđeno vreme 120 min)

POSTAVKA

Samostalna izrada domaćeg zadatka.

Na osnovu znanja stečenog na predavanjima, individualnim i pokaznim vežbama, pokušajte samostalno da obavite sledeće zadatke:

1. Kreirajte JSON bazu podataka na osnovu priložene šeme;
2. Kreirajte serverski skript koji koristi fajl sistem i express radni okvir;
3. Kreirajte API za CRUD REST operacije nad bazom podataka;
4. Pokrenite kreirani server
5. Testirajte REST pozive.

Studenti imaju različite zadatke. Do vašeg zadatka dolazite na sledeći način:

1. **Podelite vaš broj indeksa sa 5;**
2. **Ostatak pri deljenju uvećajte za 1;**
3. **Broj koji dobijete odgovara vašem zadatku.**

ZADATAK 1

Prvi domaći zadatak

Primenite prethodno definisani postavku na sledeću šemu baze podataka:

```
0
id 1
name      "Leanne Graham"
username   "Bret"
email     "Sincere@april.biz"
address
street    "Kulas Light"
suite     "Apt. 556"
city      "Gwenborough"
zipcode   "92998-3874"
geo
lat      "-37.3159"
lng      "81.1496"
```

```
phone "1-770-736-8031 x56442"
website "hildegard.org"
company
name "Romaguera-Crona"
catchPhrase "Multi-layered client-server neural-net"
bs "harness real-time e-markets"

1
id 2
name "Ervin Howell"
username "Antonette"
email "Shanna@melissa.tv"
address
street "Victor Plains"
suite "Suite 879"
city "Wisokyburgh"
zipcode "90566-7771"
geo
lat "-43.9509"
lng "-34.4618"
phone "010-692-6593 x09125"
website "anastasia.net"
company
name "Deckow-Crist"
catchPhrase "Proactive didactic contingency"
bs "synergize scalable supply-chains"
```

ZADATAK 2

Drugi domaći zadatak

Primenite prethodno definisani postavku na sledeću šemu baze podataka:

```
0
userId 1
id 1
title "delectus aut autem"
completed false

1
userId 1
id 2
title "quis ut nam facilis et officia qui"
completed false

2
userId 1
id 3
title "fugiat veniam minus"
completed false
```

```
3
userId 1
id 4
title "et porro tempora"
completed true
```

ZADATAK 3

Treći domaći zadatak

Primenite prethodno definisani postavku na sledeću šemu baze podataka:

```
[
  {
    "albumId": 1,
    "id": 1,
    "title": "accusamus beatae ad facilis cum similique qui sunt",
    "url": "https://via.placeholder.com/600/92c952",
    "thumbnailUrl": "https://via.placeholder.com/150/92c952"
  },
  {
    "albumId": 1,
    "id": 2,
    "title": "rehenderit est deserunt velit ipsam",
    "url": "https://via.placeholder.com/600/771796",
    "thumbnailUrl": "https://via.placeholder.com/150/771796"
  },
  {
    "albumId": 1,
    "id": 3,
    "title": "officia porro iure quia iusto qui ipsa ut modi",
    "url": "https://via.placeholder.com/600/24f355",
    "thumbnailUrl": "https://via.placeholder.com/150/24f355"
  },
  {
    "albumId": 1,
    "id": 4,
    "title": "culpa odio esse rerum omnis laboriosam voluptate repudiandae",
    "url": "https://via.placeholder.com/600/d32776",
    "thumbnailUrl": "https://via.placeholder.com/150/d32776"
  },
  {
    "albumId": 1,
    "id": 5,
    "title": "natus nisi omnis corporis facere molestiae rerum in",
    "url": "https://via.placeholder.com/600/f66b97",
    "thumbnailUrl": "https://via.placeholder.com/150/f66b97"
  }
]
```

ZADATAK 4

Četvrti domaći zadatak

Primenite prethodno definisanu postavku na sledeću šemu baze podataka:

```
0
userId 1
id 1
title "quidem molestiae enim"

1
userId 1
id 2
title "sunt qui excepturi placeat culpa"

2
userId 1
id 3
title "omnis laborum odio"

3
userId 1
id 4
title "non esse culpa molestiae omnis sed optio"
```

ZADATAK 5

Peti domaći zadatak

Primenite prethodno definisanu postavku na sledeću šemu baze podataka:

```
[
  {
    "postId": 1,
    "id": 1,
    "name": "id labore ex et quam laborum",
    "email": "Eliseo@gardner.biz",
    "body": "laudantium enim quasi est quidem magnam voluptate ipsam eos\\ntempora
    quo necessitatibus\\ndolor quam autem quasi\\nreiciendis et nam sapiente accusantium"
  },
  {
    "postId": 1,
    "id": 2,
    "name": "quo vero reiciendis velit similique earum",
    "email": "Jayne_Kuhic@sydney.com",
    "body": "est natus enim nihil est dolore omnis voluptatem numquam\\net omnis
    occaecati quod ullam at\\nvoluptatem error expedita pariatur\\nnihil sint nostrum
    voluptatem reiciendis et"
```

```
},
{
  "postId": 1,
  "id": 3,
  "name": "odio adipisci rerum aut animi",
  "email": "Nikita@garfield.biz",
  "body": "quia molestiae reprehenderit quasi aspernatur\naut expedita occaecati
  aliquam eveniet laudantium\nomnis quibusdam delectus saepe quia accusamus maiores
  nam est\nCum et ducimus et vero voluptates excepturi deleniti ratione"
},
{
  "postId": 1,
  "id": 4,
  "name": "alias odio sit",
  "email": "Lew@alysha.tv",
  "body": "non et atque\noccaecati deserunt quas accusantium unde odit nobis qui
  voluptatem\nquia voluptas consequuntur itaque dolor\net qui rerum deleniti ut
  occaecati"
}
]
```

▼ Poglavlje 12

Zaključak

ZAKLJUČAK

Lekcija je pokrila izučavanje Node.js kao izuzetno moćne platforme bazirane na jeziku JavaScript

Lekcija je pokrila izučavanje [Node.js](#) kao izuzetno moćne platforme bazirane na jeziku [JavaScript](#), izgrađena na [Google Chrom JavaScript V8](#) podršci. Ova platforma je, kao što je diskutovano, namenjena za razvoj savremenih veb aplikacija, poput:

- sajtovi za striming video zapisa;
- aplikacija "jedne stranice" (*single -page application*);
- brojnih drugih veb aplikacija.

Posebno je istaknuto da [Node.js](#) pripada konceptu otvorenog koda (eng. [open source](#)), potpuno je besplatan i koristi ga ogromna programerska zajednica širom sveta. Upravo iz navedenog razloga, lekcija je imala za cilj da omogući studentima savladavanje i razumevanje osnovnih [Node.js](#) koncepcata, kao i arhitekture same platforme, kao osnova za izučavanje razvoja savremenih front-end aplikacija korišćenjem radnog okvira [Angular](#).

Lekcija se fokusirala na najznačajnije [Node.js](#) funkcionalnosti, koje će biti u fokusu u narednim lekcijama. U tom svetlu, u prethodnim objektima učenja, istaknute su sledeće značajne teme:

- uvodna razmatranja;
- podešavanje Node.js okruženja;
- [NPM - Node Package Manager](#);
- funkcije povratnog poziva i rad sa događajima;
- emitovanje događaja;
- baferi i tokovi podataka u [Node.js](#);
- [Web](#) modul;
- [Node.js RESTful API](#).

Ukoliko studenti žele da steknu šira znanja u vezi sa primenom platforme [Node.js](#), mogu da angažuju literaturu navedenu u sledećoj sekciji.

LITERATURA

Za pripremu lekcije korišćena je savremena štampana i veb literatura.

1. <https://nodejs.org/en/>

2. <https://www.tutorialspoint.com/nodejs/>
3. https://www.w3schools.com/nodejs/nodejs_get_started.asp
4. <https://phoenixnap.com/kb/install-node-js-npm-on-windows>
5. <https://jsonplaceholder.typicode.com/>



IT255 - VEB SISTEMI 1

Uvod u Angular veb aplikacije

Lekcija 06

PRIRUČNIK ZA STUDENTE

IT255 - VEB SISTEMI 1

Lekcija 06

UVOD U ANGULAR VEB APLIKACIJE

- ✓ Uvod u Angular veb aplikacije
- ✓ Poglavlje 1: Prva Angular aplikacija - priprema
- ✓ Poglavlje 2: Instrukcije za Windows korisnike
- ✓ Poglavlje 3: Pisanje koda Angular aplikacije
- ✓ Poglavlje 4: Rad sa nizovima
- ✓ Poglavlje 5: Komponenta potomak
- ✓ Poglavlje 6: Komponente Angular aplikacije
- ✓ Poglavlje 7: Pokazni primer - proširenje aplikacije
- ✓ Poglavlje 8: Objavljivanje aplikacije
- ✓ Poglavlje 9: Uvod u Angular dodatni materijali
- ✓ Poglavlje 10: Pokazne vežbe 6
- ✓ Poglavlje 11: Individualna vežba 6
- ✓ Poglavlje 12: Domaći zadatak 5
- ✓ Zaključak

Copyright © 2017 – UNIVERZITET METROPOLITAN, Beograd. Sva prava zadržana. Bez prethodne pismene dozvole od strane Univerziteta METROPOLITAN zabranjena je reprodukcija, transfer, distribucija ili memorisanje nekog dela ili čitavih sadržaja ovog dokumenta., kopiranjem, snimanjem, elektronskim putem, skeniranjem ili na bilo koji drugi način.

Copyright © 2017 BELGRADE METROPOLITAN UNIVERSITY. All rights reserved. No part of this publication may be reproduced, stored in a retrieval system or transmitted in any form or by any means, electronic, mechanical, photocopying, recording, scanning or otherwise, without the prior written permission of Belgrade Metropolitan University.

✓ Uvod

UVOD

AngularJS je JavaScript MVC framework napravljen od strane Google-a.

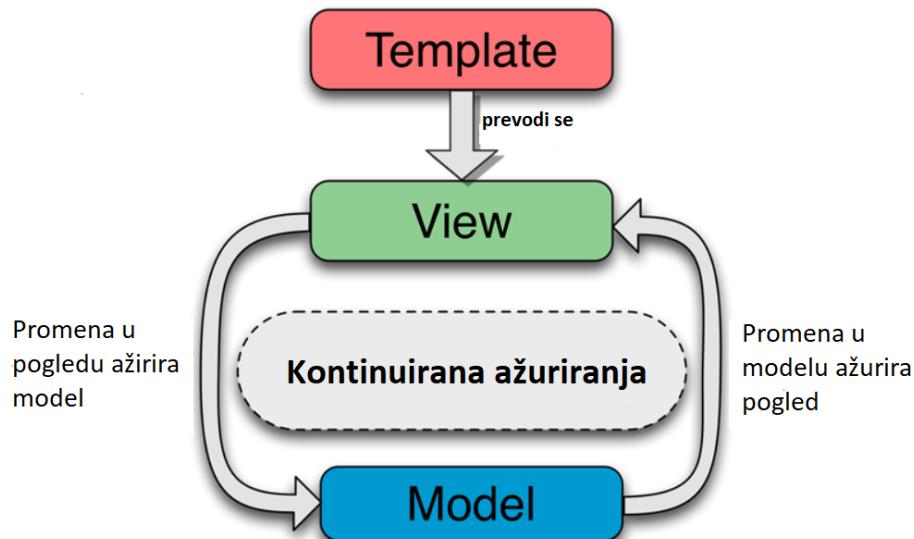
AngularJS je JavaScript MVC okvir (framework) napravljen od strane Google-a koji omogućava veb programerima kreiranje dobro strukturiranih, lакih za testiranje i održavanje front-end aplikacija.

AngularJS je MVC framework koji definiše brojne koncepte za pravilno organizovanje veb aplikacije koja se kreira.

Aplikacija je definisana uz pomoć modula koji mogu biti zavisni jedni od drugih. AngularJS proširuje HTML pružajući direktive oznakama pomoću kojih je moguće praviti moćne i dinamičke stranice. Takođe moguće je napraviti sopstvene direktive pomoću kojih će se vršiti DOM manipulacije. Takođe implementira dvosmerno vezivanje podataka, vezujući HTML (poglede) sa JavaScript objektima (modelom) na veoma lak način.

Jednostavno rečeno to znači da će se bilo kakva promena na modelu smesta reflektovati na view strani bez potrebe za bilo kakvom DOM manipulacijom ili rukovanjem događaja.

Angular je vrlo fleksibilan što se tiče serverske komunikacije. Kao i većina JavaScript okvira dopušta vam rad sa bilo kojom server-side tehnologijom dokle god može da opslužuje aplikaciju preko RESTful API-ja.



Slika-1 Dvosmerno povezivanje podataka [izvor: autor]

UVODNI VIDEO

Trajanje video snimka: 6min 28sek

Ova lekcija sadrži video materijal. Ukoliko želite da pogledate ovaj video morate da otvorite LAMS lekciju.

▼ Poglavlje 1

Prva Angular aplikacija - priprema

VIDEO PREDAVANJE ZA OBJEKAT "PRVA ANGULAR APLIKACIJA - PRIPREMA"

Trajanje video snimka: 31min 51sek

Ova lekcija sadrži video materijal. Ukoliko želite da pogledate ovaj video morate da otvorite LAMS lekciju.

KREIRANJE JEDNOSTAVNOG REDDIT KLONA

Lekcija je opredeljena da najznačajnije koncepte pokazuje na konkretnim primerima.

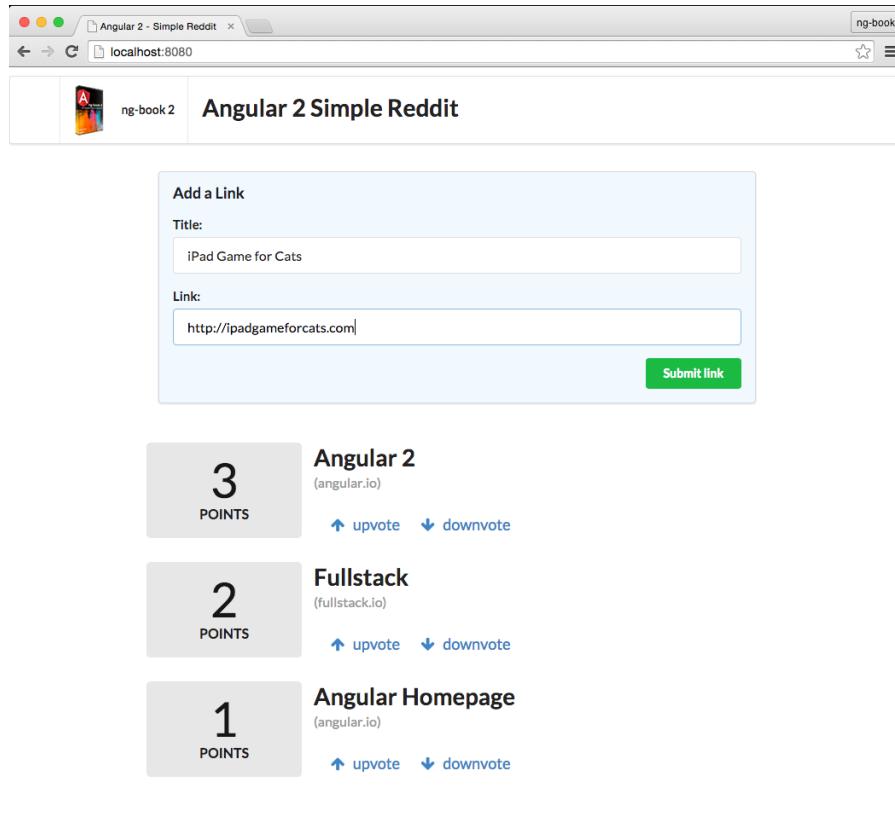
U ovoj lekciji cilj će biti kreiranje aplikacije koja će omogućiti korisnicima postave članak (zajedno sa nazivom i linkom) i da, nakon toga, mogu da glasaju (pozitivno ili negativno) za ostale postove. Aplikacija će biti pojednostavljena verzija dobro poznatih veb aplikacija poput *Reddit* ili *Product Hunt*.

Lekcija je opredeljena da najznačajnije koncepte pokazuje na konkretnim primerima. Otuda, u ovoj jednostavnoj aplikaciji biće neophodno pokriti najznačajnije Angular koncepte, poput:

- izgradnje vlastitih komponenata;
- kreiranje formi i prihvatanje unosa sa istih;
- kreiranje pogleda iz liste objekata;
- presretanje interakcije korisnika i GUI komponenata i rukovanje njima;
- angažovanje aplikacije na veb serveru.

Ideja je da kada student završi izučavanje ove lekcije moći će da iskoristi prazan folder, kreira osnovnu Angular aplikaciju, a zatim je angažuje na serveru i pokrene. Upravo ovakva aplikacija će predstavljati osnov za razvoj iskustva iz kojeg će student moći da kreira veći broj različitih front-end aplikacija koristeći okvir *Angular*.

Sledećom slikom je prezentovan mogući izgled aplikacije čije su osnove izložene u prethodnom izlaganju.

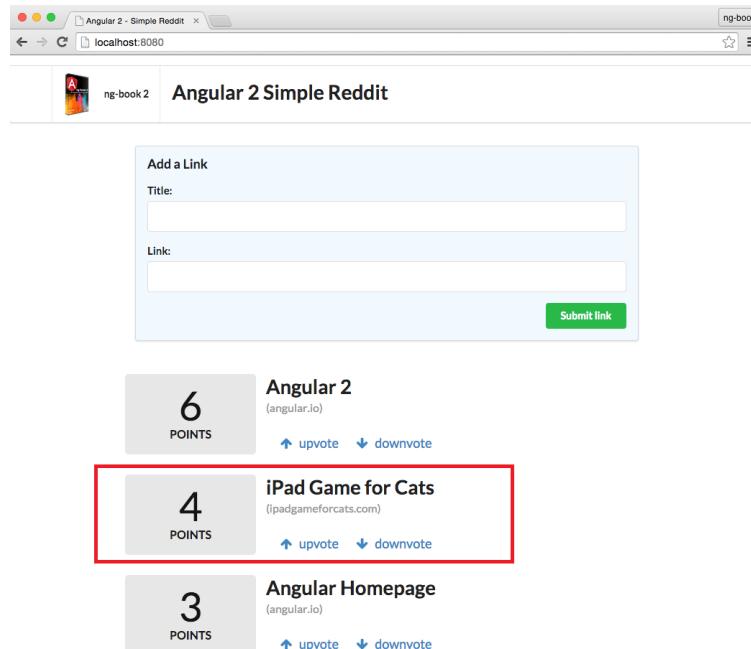


Slika 1.1 Predloženi izgled Angular aplikacije [izvor: autor]

FORMA I TYPESCRIPT

TypeScript predstavlja nadskup za JavaScript ES6 sa dodatkom tipova podataka.

U aplikaciji se pojavljuje koncept bez kojeg nije moguće zamisliti nijednu ozbiljnu veb aplikaciju - forma. Preko forme će korisnik moći da doda u aplikaciju novi link za kojeg će ostali korisnici moći da glasaju (pozitivno ili negativno). Pored svakog linka će stajati ocena koja predstavlja utisak korisnika o korisnosti sadržaja koji se krije iza linka. Dodati link je prikazan sledećom slikom.



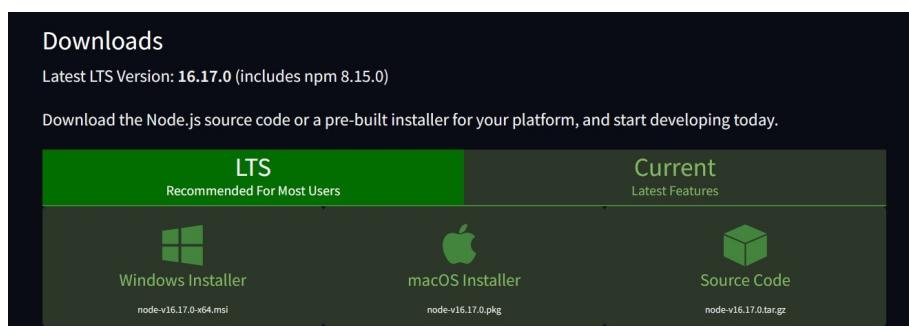
Slika 1.2 Dodavanje podataka preko forme [izvor: autor]

Kroz ovaj projekat, a i u daljem izlaganju u ovom predmetu, biće korišćen [TypeScript](#). [TypeScript](#) predstavlja nadskup za JavaScript ES6 sa dodatkom tipova podataka. O ovoj notaciji će biti detaljno diskutovano u lekcijama koje slede. Studenti će bez problema moći da prate lekciju ukoliko su savladali sintaksu baznog JavaScript jezika.

NODE.JS I NPM

Neophodno je instaliranje podrške izvršavanju JavaScript instrukcija u formi Node.js.

Da bi bilo moguće raditi bilo šta sa Angular okvirom, za početak, neophodno je instaliranje podrške izvršavanju JavaScript instrukcija u formi serverskog okruženja [Node.js](#). Postoji nekoliko različitih načina da se instalira Node.js i svi su detaljno objašnjeni na sajtu <https://nodejs.org/en/>.



Slika 1.3 Sajt za preuzimanje instalacije [izvor: <https://nodejs.org/en/>]

Uvek bi bilo dobro preuzeti i instalirati poslednju stabilnu verziju za Node.js i preporuka je da to svakako bude verzija 8.9.0 ili neka novija. Sa slike je moguće videti da je poslednja stabilna verzija 16.17.0.

Dalje, za rad je takođe potreban i [npm](#) (*Node Package Manager*) koji bi trebalo da bude instaliran kao deo Node.js paketa. Da bi bila ovavljena provera uspešnog instaliranja [npm](#) - a kao dela razvojnog okruženja, neophodno je otvoriti *command prompt* i u terminalu otkucati sledeću naredbu:

```
$ npm -v
```

Ukoliko nije prikazan broj verzije ili je primljena greška, neophodno je proveriti da li je instaliran Node.js koji uključuje npm. Verzija bi trebalo da bude 5.6.0 ili novija (poslednja 8.19.0).

Ova lekcija sadrži video materijal. Ukoliko želite da pogledate ovaj video morate da otvorite LAMS lekciju.

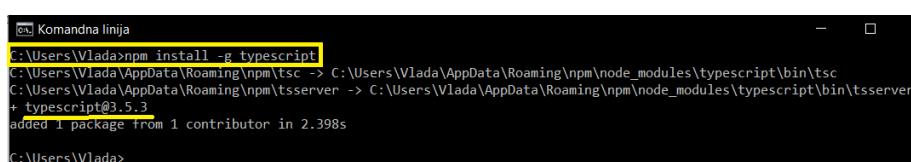
TYPESCRIPT I PREGLEDAČ - PODEŠAVANJE

Za dalji rad je neophodno instalirati TypeScript podršku na radni računar i izabrati pregledač.

Za dalji rad je neophodno instalirati TypeScript podršku na radni računar. Ukoliko je sve prethodno navedeno ispoštovano (instaliran Node.js), u command promt-u, na komandnoj liniji, neophodno je uneti sledeću naredbu:

```
npm install -g typescript
```

Ukoliko je dobijen izlaz kao na sledećoj slici, sve je dobro obavljen.



```
C:\ Komandna linija
C:\Users\Vlada>npm install -g typescript
C:\Users\Vlada\AppData\Roaming\npm\tsc -> C:\Users\Vlada\AppData\Roaming\npm\node_modules\typescript\bin\tsc
C:\Users\Vlada\AppData\Roaming\npm\tsserver -> C:\Users\Vlada\AppData\Roaming\npm\node_modules\typescript\bin\tsserver
+ typescript@3.5.3
added 1 package from 1 contributor in 2.398s
C:\Users\Vlada>
```

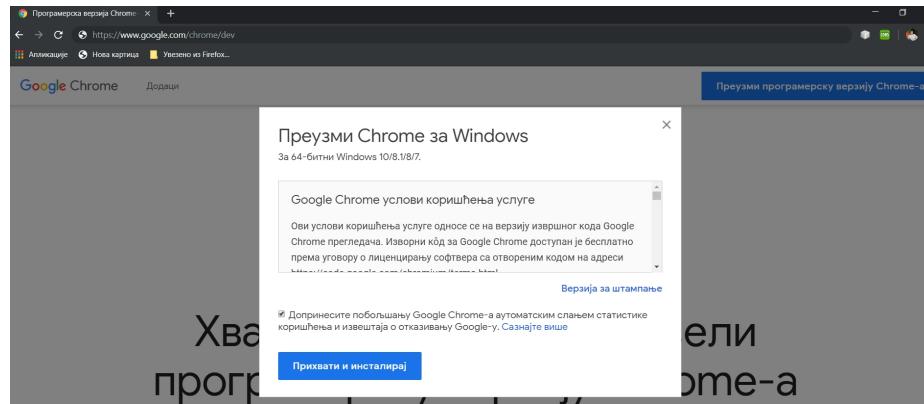
Slika 1.4 Instalacija TypeScript podrške [izvor: autor]

Takođe, ovde je neophodno voditi računa o verziji, morala bi da bude 2.1 ili novija (ovde je konkretno 3.5.3).

Za pokretanje i testiranje kreiranih veb aplikacija pomoću Angular okvira, Google preporučuje vlastiti Google Chrome veb pregledač. Takođe, preporuka je instaliranje programerske verzije ovog pregledača, a to je omogućeno ako u standardnom pregledaču navedete link:

```
https://www.google.com/chrome/dev
```

Sve što je potrebno u nastavku jeste da pratite uputstva koja vam nudi čarobnjak za instalaciju prikazan na sledećoj slici.



Slika 1.5 Chrome developer toolkit instalacija [izvor: autor]

▼ Poglavlje 2

Instrukcije za Windows korisnike

ANGULAR CLI

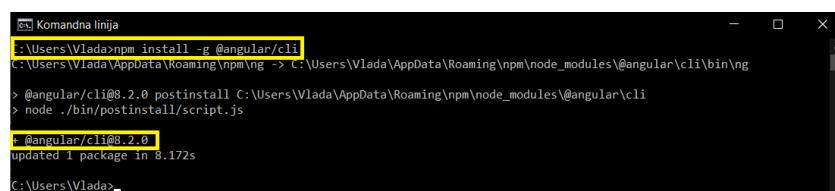
Angular CLI dozvoljavaju korisnicima kreiranje i upravljanje projektima sa komandne linije.

Angular obezbeđuje alate koji dozvoljavaju korisnicima kreiranje i upravljanje projektima sa komandne linije. Ovi alati su univerzalni za različite platforme i nazivaju se [Angular CLI](#). Zapravo kreiranje novih projekata, dodavanje kontrolera i ostalih komponenata Angular aplikacije primenom Angular CLI alata predstavlja jako dobru ideju. Ovi alati omogućavaju, takođe, kreiranje i održavanje različitih šabloni koji se koriste u aplikaciji.

Za instaliranje alata [Angular CLI](#) dovoljno je u komandnoj liniji Command Prompt - a navesti naredbu:

```
npm install -g @angular/cli
```

Ako je rezultat izvršavanja navedene naredbe identičan kao na sledećoj slici, sav posao instalacije Angular CLI je uspešno obavljen.



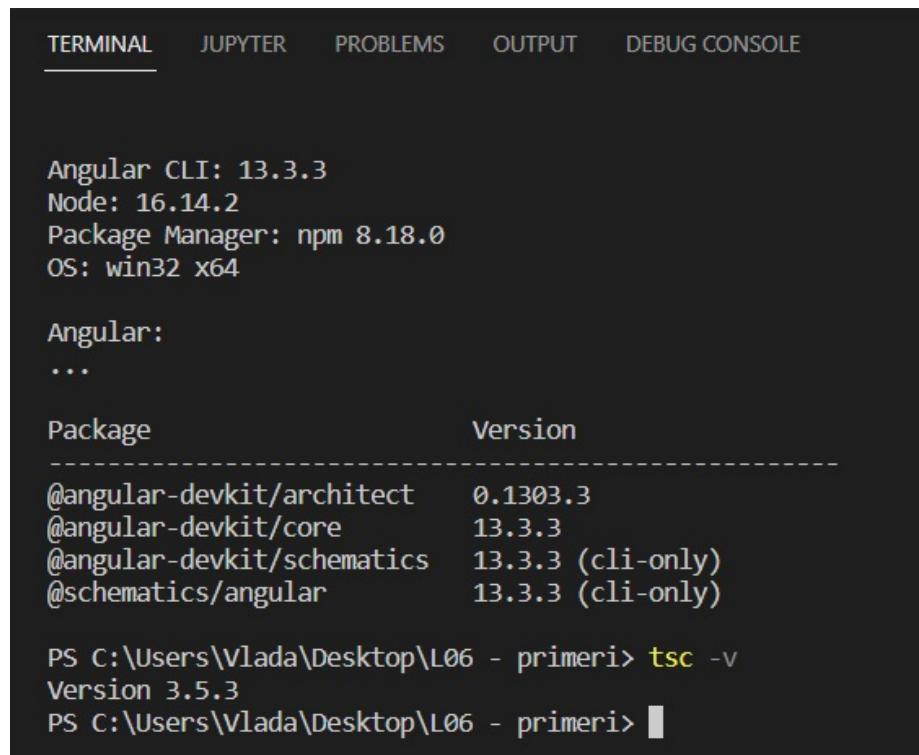
```
Administrator: Komandna linija
C:\Users\Vlada>npm install -g @angular/cli
C:\Users\Vlada\AppData\Roaming\npm\ng -> C:\Users\Vlada\AppData\Roaming\npm\node_modules\@angular\cli\bin\ng
> @angular/cli@8.2.0 postinstall C:\Users\Vlada\AppData\Roaming\npm\node_modules\@angular\cli
> node ./bin/postinstall/script.js
+ @angular/cli@8.2.0
updated 1 package in 8.172s
C:\Users\Vlada>
```

Slika 2.1 Angular CLI instalacija za Windows operativni sistem [izvor: autor]

Nakon uspešne instalacije, programeri imaju mogućnost korišćenja naredbe [ng](#) na komandnoj liniji. pozivom ove naredbe dobija se detaljan izlaz sa podacima u vezi sa Angular CLI. Na primer, pozivom:

```
ng --version
```

dobija se izlaz kao na sledećoj slici.



The screenshot shows a terminal window with the following output:

```
Angular CLI: 13.3.3
Node: 16.14.2
Package Manager: npm 8.18.0
OS: win32 x64

Angular:
...
Package          Version
-----
@angular-devkit/architect    0.1303.3
@angular-devkit/core         13.3.3
@angular-devkit/schematics   13.3.3 (cli-only)
@schematics/angular          13.3.3 (cli-only)

PS C:\Users\Vlada\Desktop\L06 - primeri> tsc -v
Version 3.5.3
PS C:\Users\Vlada\Desktop\L06 - primeri>
```

Slika 2.2 Instalirana Angular CLI verzija [izvor: autor]

Takođe, pozivom: `ng --help` moguće je dobiti i dodatne informacije.

PRIMER JEDNOSTAVNOG ANGULAR PROJEKTA

Moguće je pristupiti kreiranju prve Angular aplikacije.

Budući da su sva podešavanja za rad sa Angular okvirom uspešno obavljena, moguće je pristupiti kreiranju prve Angular aplikacije. U tu svrhu je moguće koristiti novu `ng` naredbu koju je neophodno navesti u terminalu razvojnog okruženja (preporuka je Visual Studio Code). Konkretno, naredba glasi:

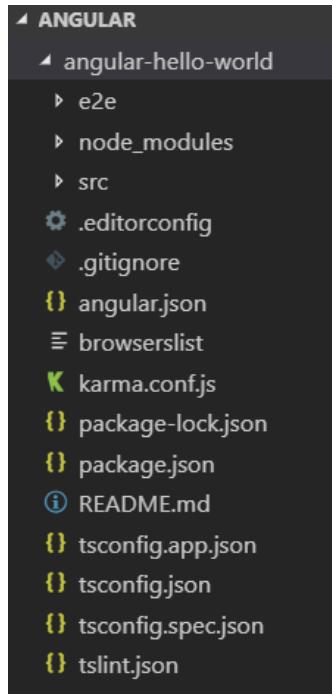
```
ng new angular-hello-world
```

Izvršavanjem naredbe kreira se izlaz prikazan sledećom slikom.

```
PS C:\Users\vlada\Documents\Angular> ng new angular-hello-world
? Would you like to add Angular routing? Yes
? Which stylesheet format would you like to use? stylus [ http://stylus-lang.com ]
CREATE angular-hello-world/angular.json (3785 bytes)
CREATE angular-hello-world/package.json (1293 bytes)
CREATE angular-hello-world/README.md (1034 bytes)
CREATE angular-hello-world/tsconfig.json (543 bytes)
CREATE angular-hello-world/tslint.json (1988 bytes)
CREATE angular-hello-world/.editorconfig (246 bytes)
CREATE angular-hello-world/.gitignore (631 bytes)
CREATE angular-hello-world/browserslist (429 bytes)
CREATE angular-hello-world/karma.conf.js (1031 bytes)
CREATE angular-hello-world/tsconfig.app.json (270 bytes)
CREATE angular-hello-world/tsconfig.spec.json (270 bytes)
CREATE angular-hello-world/src/favicon.ico (5430 bytes)
CREATE angular-hello-world/src/index.html (304 bytes)
CREATE angular-hello-world/src/main.ts (372 bytes)
CREATE angular-hello-world/src/polyfills.ts (2838 bytes)
CREATE angular-hello-world/src/styles.styl (80 bytes)
CREATE angular-hello-world/src/test.ts (642 bytes)
CREATE angular-hello-world/src/assets/.gitkeep (0 bytes)
CREATE angular-hello-world/src/environments/environment.prod.ts (51 bytes)
CREATE angular-hello-world/src/environments/environment.ts (662 bytes)
CREATE angular-hello-world/src/app/app-routing.module.ts (246 bytes)
CREATE angular-hello-world/src/app/app.module.ts (393 bytes)
CREATE angular-hello-world/src/app/app.component.html (1152 bytes)
CREATE angular-hello-world/src/app/app.component.spec.ts (1134 bytes)
CREATE angular-hello-world/src/app/app.component.ts (224 bytes)
CREATE angular-hello-world/src/app/app.component.styl (0 bytes)
CREATE angular-hello-world/e2e/protractor.conf.js (810 bytes)
CREATE angular-hello-world/e2e/tsconfig.json (214 bytes)
CREATE angular-hello-world/e2e/src/app.e2e-spec.ts (648 bytes)
CREATE angular-hello-world/e2e/src/app.po.ts (251 bytes)
```

Slika 2.3 Kreiranje Angular projekta [izvor: autor]

Navedeni proces će teći neko vreme dok se sve zavisnosti ne učitaju i neophodni fajlovi kreiraju. U ovom trenutku ne bi trebalo da brine studenta što trenutno ne razume ulogu pojedinih fajlova u aplikaciji - svi će biti detaljno objašnjeni u nastavku. Navedena naredba je kreirala projekat sa sledećom strukturu.

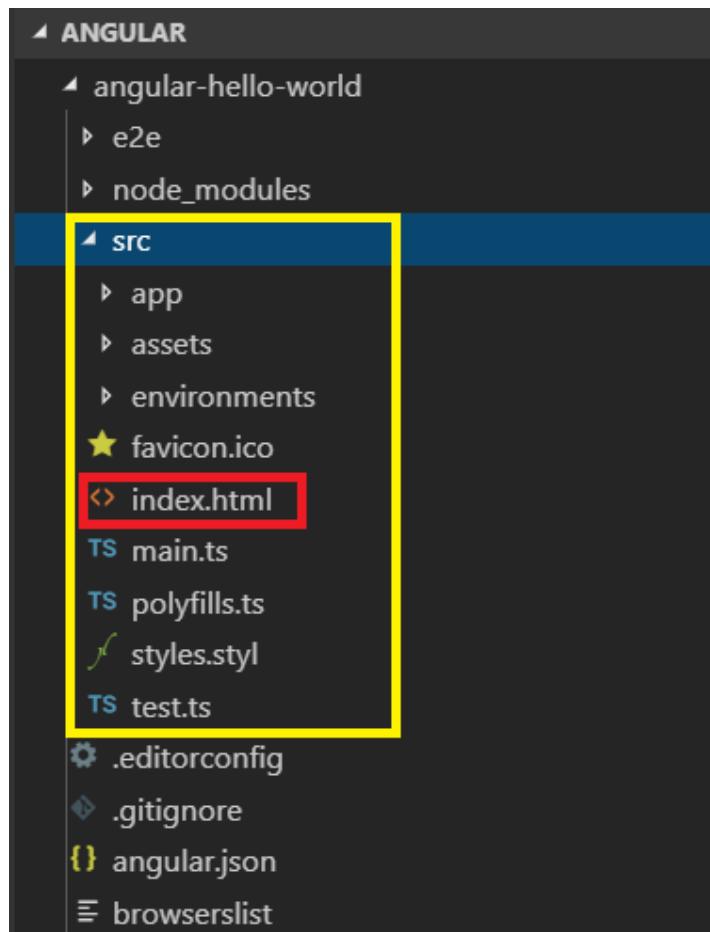


Slika 2.4 Struktura kreiranog projekta [izvor: autor]

GENERISANI KOD KREIRANOG ANGULAR PROJEKTA

Sve programske datoteke moguće je naći pod čvorom src u stablu projekta.

Kao što je istaknuto u prethodnom izlaganju, prilikom kreiranja aktuelnog projekta generisane su brojne datoteke. Među njima se nalaze i datoteke koje u sebi čuvaju programski kod: **JavaScript**, **HTML** i slično. Sve programske datoteke moguće je naći pod čvorom **src** u stablu projekta. Navedeno je prikazano sledećom slikom.



Slika 2.5 Lokacija programskih datoteka u strukturi projekta [izvor: autor]

ANALIZA GENERISANE DATOTEKE POGLEDA

Analiza kreirane datoteke index.html.

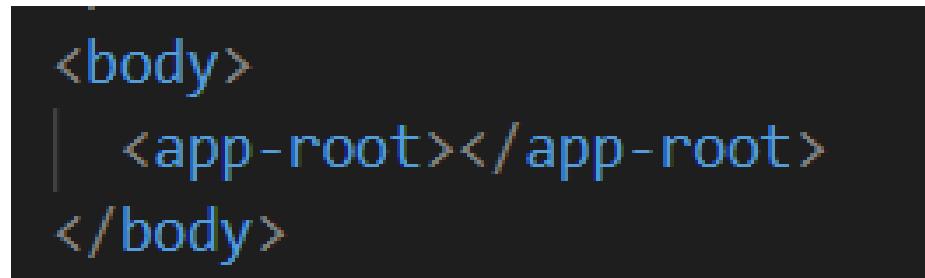
Na prethodnoj slici, crvenom bojom, posebno je istaknuta jedna datoteka. Radi se o datoteci *index.html* koja predstavlja pogled (veb stranicu) u kreiranoj aplikaciji i data je sledećim listingom.

```
<!doctype html>
<html lang="en">
<head>
  <meta charset="utf-8">
  <title>AngularHelloWorld</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>
```

Prvi deo koda je poprilično jasan za sve koji su imalo upoznati sa [HTML](#) - om. Definisan je set karaktera, naslov i osnovni link za učitavanje pogleda. Posebnu pažnju bi trebalo obratiti na deo koda koji je prikazan sledećom slikom.



Slika 2.6 Segment datoteke index.html. [izvor: autor]

Prikazani tag `<app-root>` određuje mesto gde će aplikacija biti kreirana na način koji je prezentovan krajnjem korisniku. Ovim tagom je određena komponenta koja je definisana Angular aplikacijom. U Angularu je moguće definisati vlastite HTML tagove i njihove funkcionalnosti. Navedeni tag `<app-root>` će predstavljati "ulaznu tačku" aplikacije na stranici index.html.

Ova lekcija sadrži video materijal. Ukoliko želite da pogledate ovaj video morate da otvorite LAMS lekciju.

▼ Poglavlje 3

Pisanje koda Angular aplikacije

POKRETANJE KREIRANE APLIKACIJE

Neophodno je izvršiti demonstraciju prevođenja i pokretanja Angular aplikacije.

Pre bilo kakve intervencije nad kreiranim datotekama i dodavanja vlastitog programskog koda, neophodno je izvršiti demonstraciju prevođenja i pokretanja Angular aplikacije. U tu svrhu može da posluži i kreirana aplikacija sa osnovnim generisanim kodom. Za kreirani projekat, u terminalu razvojnog okruženja, neophodno je navesti naredbu:

```
cd angular-hello-world  
ng -serve
```

Naredbom **ng - serve** aplikacija se prevodi i angažuje na serveru, a rezultati prevođenja su prikazani sledećom slikom.

```
PS C:\Users\Vlada\Documents\Angular\angular-hello-world> ng serve  
1% building 3/3 modules (active) [wds]: Project is running at http://localhost:4200/webpack-dev-server/  
i [wds]: webpack output is served from /  
i [wds]: 404s will fallback to //index.html  
  
chunk {main} main.js, main.js.map (main) 11.6 kB [initial] [rendered]  
chunk {polyfills} polyfills.js, polyfills.js.map (polyfills) 251 kB [initial] [rendered]  
chunk {runtime} runtime.js, runtime.js.map (runtime) 6.09 kB [entry] [rendered]  
chunk {styles} styles.js, styles.js.map (styles) 16.8 kB [initial] [rendered]  
chunk {vendor} vendor.js, vendor.js.map (vendor) 4.02 MB [initial] [rendered]  
Date: 2019-08-03T09:52:33.869Z - Hash: 57c859b3f22d699e9ff4 - Time: 7426ms  
** Angular Live Development Server is listening on localhost:4200, open your browser on http://localhost:4200/ **  
i [wcm]: Compiled successfully.
```

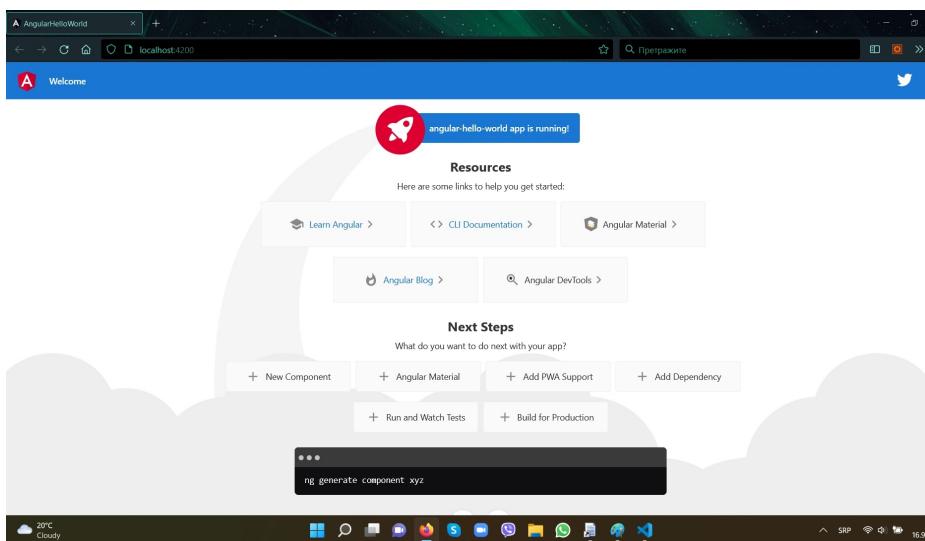
Slika 3.1 Prevođenje i angažovanje Angular aplikacije [izvor: autor]

Po podrazumevanim vrednostima, a to je moguće sagledati i sa slike, **Angular aplikacije se izvršavaju na portu 4200 na lokalnom računaru (*localhost*)**. Ukoliko je ovaj port zauzet i aplikaciju nije moguće pokrenuti na njemu, moguće je postupiti drugačiji. Prilikom prevođenja aplikacije navodi se i alternativni port preko kojeg je moguće pokrenuti aplikaciju. To je moguće obaviti sledećom naredbom.

```
ng serve --port 9001
```

Na ovaj način je omogućeno funkcionisanje aplikacije na portu 9001 lokalnog računara.

Sada je moguće otvoriti veb pregledač (*Google Chrome* je predloženi) i unosom linka <http://localhost:4200/> pokreće se kreirana aplikacija, tačnije učitava stranica *index.html*. Navedeno je prikazano sledećom slikom.



Slika 3.2 Početna strana kreirane Angular aplikacije [izvor: autor]

KREIRANJE KOMPONENTE

Jedna od velikih ideja razvoja Angular okvira je rad sa komponentama.

Jedna od velikih ideja razvoja Angular okvira je rad sa komponentama. U Angular aplikacijama programeri pišu HTML kod koji aplikaciju čini interaktivnom. Međutim, veb pregledač razume ograničen broj HTML tagova. Ugrađeni tagovi, poput <select>, <form> ili <video> poseduju funkcionalnosti koje je definisao proizvođač veb pregledača.

Postavlja se novo pitanje. Da li je moguće naučiti pregledač da izvršava nov tag? Na primer, ukoliko programer želi da primeni tag <weather> za prikazivanje podataka o vremenskim prilikama, ili <login> za prikazivanje forme za prijavljivanje korisnika, da li će pregledač moći da nauči da koristi nove funkcionalnosti?

Da bi sve bilo jasnije neophodno je pristupiti kreiranju prve Angular komponente. Komponenta će biti dostupna i u HTML dokumentu putem taga:

```
<app-hello-world></app-hello-world>
```

Za kreiranje nove komponente putem Angular CLI koristi se naredba generate, na primer u sledećem obliku:

```
ng generate component hello-world
```

Rezultat izvršavanja ove naredbe moguće je prezentovati sledećom slikom.

```
PS C:\Users\Vlada\Documents\Angular\angular-hello-world> ng generate component hello-world
CREATE src/app/hello-world/hello-world.component.html (26 bytes)
CREATE src/app/hello-world/hello-world.component.spec.ts (657 bytes)
CREATE src/app/hello-world/hello-world.component.ts (289 bytes)
CREATE src/app/hello-world/hello-world.component.styl (0 bytes)
UPDATE src/app/app.module.ts (493 bytes)
PS C:\Users\Vlada\Documents\Angular\angular-hello-world>
```

Slika 3.3 Kreiranje nove Angular komponente [izvor: autor]

DELOVI KOMPONENTE

Svaka komponenta je sastavljena iz dva dela.

Svaka komponenta je sastavljena iz dva dela:

- dekorator *Component*;
- *klasa definicije komponente*.

Za početak moguće je pogledati inicijalni kod komponente. Komponenta se nalazi u projektu na lokaciji *src/app/hello-world/hello-world.component.ts* i sledećim listingom je priložen njen inicijalni *TypeScript* kod:

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

@Component({
  selector: 'app-hello-world',
  templateUrl: './hello-world.component.html',
  styleUrls: ['./hello-world.component.style']
})
export class HelloWorldComponent implements OnInit {

  constructor() { }

  ngOnInit() {
  }
}
```

Studentima kod na prvi pogled može da bude zastrašujući. Bez brige, sve će biti analizirano detaljno i korak po korak.

Moguće je primetiti da je ekstenzija ove datoteke .ts umesto .js. Problem je u tome što veb pregledač ne zna kako da interpretira TypeScript fajlove. Ovaj problem je jednostavno rešen tako što komandom ng - serve vrši se automatska konverzija .ts fajlova u .js fajlove.

DODAVANJE ZAVISNOSTI

Iskaz Import ukazuje na module koji se koriste prilikom pisanja koda.

Iskaz *Import* ukazuje na module koji se koriste prilikom pisanja koda. Ako se pogleda listing na prethodnoj sekciji moguće je primetiti da se dodaju dve zavisnosti: *Component* i *OnInit*. Uključuje se zavisnost *Component* iz modula *@angular/core*. Ovaj modul ukazuje programu na lokaciju na kojoj može da pronađe traženu zavisnost. U konkretnom slučaju, ukazano

je kompjajleru da `@angular/core` definiše i eksportuje dva *JavaScript/TypeScript* objekta `Component` i `OnInit`.

Na sličan način, iz istog modula se dodaje komponenta `OnInit`. Ova komponenta će biti posebna tema u nekom od narednih izlaganja i njen zadatak je da pokrene kod prilikom inicijalizovanja komponente.

Neka je dalji predmet analize struktura `Import` naredbi koja može biti prikazana u opštem obliku:

```
import { zavisnost } from modul
```

U delu `{ zavisnost }` obavlja se osobina *destrukturiranja*. Destrukturiranje je osobina obezbeđena putem JavaScript ES6 i TypeScript i o njoj će detaljno biti govora u narednoj lekciji.

COMPONENT DEKORATORI

Nakon dodavanja zavisnosti neophodno je obaviti deklarisanje komponente.

Nakon dodavanja zavisnosti neophodno je obaviti deklarisanje komponente `hello-world.component.ts`. Neka je dat deo koda ove datoteke:

```
@Component({
  selector: 'app-hello-world',
  templateUrl: './hello-world.component.html',
  styleUrls: ['./hello-world.component.styl']
})
```

Šta se ovde dešava? Ovaj blok se naziva dekoratorom. Dekorator je moguće razumeti kao metapodatke dodate u kod. Kada se klasa, na primer `HelloWorld`, obeleži anotacijom `@Component` može se reći da je kasa dekorisana kao komponenta.

Sada je moguće vratiti se na postavljeni problem korišćenja komponente u HTML kodu putem vlastitog taga. U ovom slučaju primena taga `<app-hello-world>` omogućena je definisanjem selektora unutar dekoratora. Navedeno je prikazano sledećom slikom.

```
@Component({
  selector: 'app-hello-world',
  templateUrl: './hello-world.component.html',
  styleUrls: ['./hello-world.component.styl']
})
```

Slika 3.4 Definisanje selektora [izvor: autor]

Sintaksa Angular selektora je slična CSS selektorima iako postoje neke specifičnosti u sintaksi Angular selektora o kojima će detaljno biti govora kasnije. Za sada, dovoljno je shvatiti da se selektorom definiše novi tag kojeg je moguće koristiti u HTML kodu.

Osobine selektor ovde pokazuje koji će DOM element koristiti komponenta. U konkretnom slučaju, tagovi `<app-hello-world></app-hello-world>` se pojavljuju unutar šablonu i biće prevedeni preko klase `HelloWorldComponent` i posedovaće bilo koju pridruženu funkcionalnost.

DODAVANJE ŠABLONA PREKO TEMPLATEURL I TEMPLATE

U aktuelnoj komponenti specificiran je šablon pod ključem templateUrl.

U aktuelnoj komponenti specificiran je šablon pod ključem `templateUrl` koji odgovara linku `./hello-world.component.html`. To znači da će šablon biti učitan iz fajla `hello-world.component.html` u istom direktorijumu kao i komponenta. Sledećim listingom priložen je kod koji se nalazi u navedenom fajlu:

```
<p>hello-world works!</p>
```

Na ovaj način definisan je `<p>` tag sa nekim osnovnim tekstom. Kada Angular učita komponentu, takođe čita i iz ovog fajla i koristi ga kao šablon za komponentu.

Šablon može biti specificiran i na drugi način. U ovom slučaju kao ključ u dekorateru se koristi reč `template`, umesto prethodno korišćenje `templateUrl`. To može biti učinjeno na sledeći način:

```
@Component({
  selector: 'app-hello-world',
  template: `
    <p>
      hello-world works inline!
    </p>
  `
})
```

Iz listinga je moguće primetiti da je sting šablon smešten između znakova apostrofa. Ovo je nova Angular funkcionalnost koja omogućava pisanje šablonu u formi stringova sa više linija. Na ovaj način se šabloni lakše dodaju u kod.

DODAVANJE CSS PREKO STYLEURLS

Angular koristi koncept poznat pod nazivom enkapsulacija stilova.

Ako se još jednom baci pogled na kod dekoratera `@Component` moguće je primetiti još jedan par (ključ, vrednost) prikazan sledećim kodom:

```
styleUrls: ['./hello-world.component.css']
```

Ovaj deo koda ukazuje da se za komponentu koriste CSS stilovi definisani u datoteci *hello-world.component.css*. Angular koristi koncept poznat pod nazivom *enkapsulacija stilova* što praktično znači da stilovi specificirani za određenu komponentu mogu biti primenjeni samo na tu komponentu. O ovome će biti detaljno govora u posebnoj lekciji. Za sada, neće biti korišćeni nikakvi lokalni stilovi za komponente i ovaj fajl neće biti menjan (ili ključ može biti obrisan u potpunosti).

UČITAVANJE KOMPONENTE

Kada je kompletiran kod komponente, moguće je obaviti njeno učitavanje u konkretnoj stranici.

Sada, kada je kompletiran kod komponente, moguće je obaviti njeno učitavanje u konkretnoj HTML stranici.

Ukoliko se ponovo pokrene aplikacija u pregledaču, neće biti moguće primetiti bilo kakvu promenu. Ovo je iz razloga što je komponenta kreirana ali nije upotrebljena. Sa ciljem realizacije navedenog, neophodno je dodati tag komponente u šablon koji je već kreiran. Za ovo je neophodno otvoriti datoteku *src/app/app.component.html*. Iz razloga što je komponenta *HelloWorldComponent* podešena preko selektora *app-helloworld* može biti korišćena u šablonu. Neka bude dodat tag *<app-hello-world>* u fajl *app.component.html*. Pre toga biće obrisan celokupan inicijalni kod ove datoteke. Novi listing datoteke *app.component.html* izgleda ovako:

```
<h1>  
  {{title}}  
  
  <app-hello-world></app-hello-world>  
</h1>
```

Sada je moguće osvežiti stranicu *index.html* i biće prikazan sadržaj kao na sledećoj slici.



angular-hello-world

hello-world works!

Slika 3.5 Osvežena stranica sa novim sadržajem [izvor: autor]

DODAVANJE PODATAKA U KOMPONENTU

Aplikacija dobija mogućnost prikazivanja liste korisnika i prikazuju se njihova imena.

Aplikacija, koja predstavlja pokazni primer za trenutno izlaganje, koristi komponente koje kreiraju statičke šablove, a to i nije preterano zanimljivo.

U nastavku, razmišljanje može da ide u drugom pravcu. Aplikacija dobija mogućnost prikazivanja liste korisnika i prikazuju se njihova imena. Pre kreiranja navedene liste neophodno je obaviti kreiranje pojedinačnih korisnika. U tu svrhu biće kreirana komponenta čiji će zadatak biti prikazivanje imena korisnika.

Da bi navedeno bilo obavljeno, u terminalu je neophodno navesti naredbu:

```
ng generate component user-item
```

Rezultat izvršavanja naredbe je prikazan sledećom slikom.

```
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL
PS C:\Users\Vlada\Documents\Angular\angular-hello-world> ng generate component user-item
CREATE src/app/user-item/user-item.component.html (24 bytes)
CREATE src/app/user-item/user-item.component.spec.ts (643 bytes)
CREATE src/app/user-item/user-item.component.ts (281 bytes)
CREATE src/app/user-item/user-item.component.styl (0 bytes)
UPDATE src/app/app.module.ts (585 bytes)
PS C:\Users\Vlada\Documents\Angular\angular-hello-world>
```

Slika 3.6 Kreiranje komponente user-item [izvor: autor]

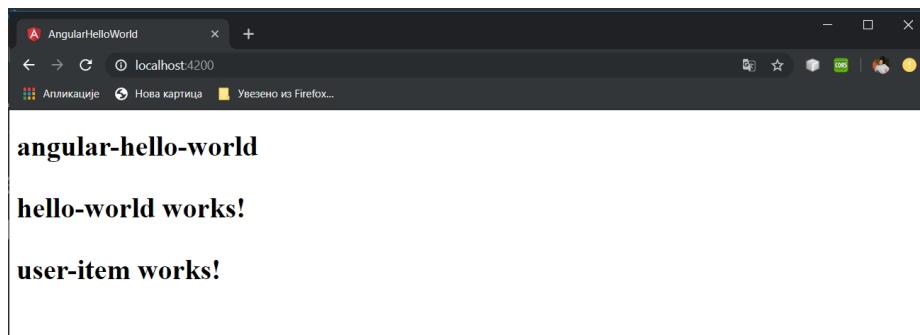
Da bi kreirana komponenta postala vidljiva, neophodno je obaviti njeno dodavanje u šablon. Ovo se obavlja dodavanjem taga `<app-user-item>` u šablon `app.component.html`. Izmenjen sadržaj datoteke prikazan je sledećim listingom:

```
<h1>
  {{title}}
</h1>

<app-hello-world></app-hello-world>

<app-user-item></app-user-item>
</h1>
```

Osvežavanjem početne stranice proverava se da li je sve do sada obavljeno kako treba. Ako jeste, dobiće se izlaz kao na sledećoj slici.



Slika 3.7 Nova komponenta u šablonu [izvor: autor]

KODIRANJE KLASE KOMPONENTE

Moguće je pristupiti kodiranju klase komponente.

Pošto je sve spremno za da dalji rad, moguće je pristupiti kodiranju klase komponente. Cilj je da komponenta *UserItemComponent* prikazuje ime konkretnog korisnika.

Za klasu će biti uvedena nova osobina pod nazivom *name*. Dodavanjem ove osobine biće moguće višestruko koristiti komponentu za različite korisnike (uz zadržavanje istog HTML koda, logike i stilova). Sa ciljem realizovanja navedenog, klasa *UserItemComponent* je proširena kodom koji dodaje navedenu promenljivu i ona sada izgleda ovako:

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

@Component({
  selector: 'app-user-item',
  templateUrl: './user-item.component.html',
  styleUrls: ['./user-item.component.styles']
})
export class UserItemComponent implements OnInit {

  name: string; // dodata nova osobina

  constructor() {
    this.name = 'Vlada';
  }

  ngOnInit() {
  }
}
```

Moguće je primetiti da su u klasi načinjene dve izmene:

1. **dodata je nova osobina** : Ovo je način kako *TypeScript* uvodi nove osobine u klase. Na ovaj način je obezbeđeno da će svaki kreirani objekat klase *UserItemComponent* imati osobinu *name*. Podešavanjem ove osobine na tip *string* ukazano je kompjajleru

da podatak pridružen ovoj promenljivoj mora biti tipa *string* i da će se u suprotnom javiti greška;

2. dodat je nov kod u konstruktor klase: *TypeScript* klase, takođe, poseduju konstruktore - funkcije koje se pozivaju kada se kreiraju novi objekti klase. U konstruktoru je moguće, na primer, veoma jednostavno dodeliti konkretnu vrednost promenljivoj:

```
constructor() {  
    this.name = 'Vlada'; //dodata vrednosti promenljivoj  
}
```

KREIRANJE ŠABLONA ZA PRIKAZIVANJE KORISNIKA

Osobinu je moguće prikazati u šablonu primenom para vitičastih zagrada.

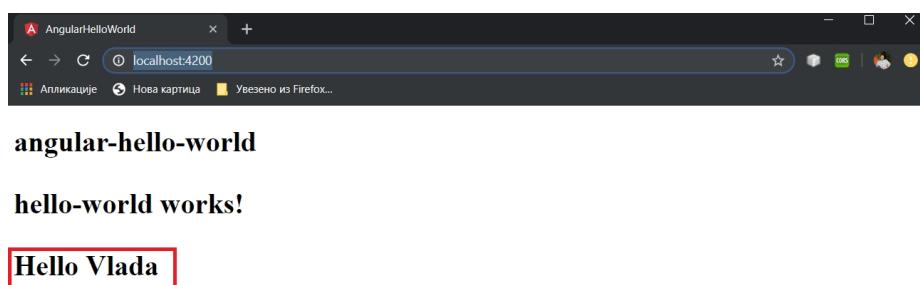
Komponenta poseduje osobinu. Osobinu je moguće prikazati u šablonu primenom para vitičastih zagrada *{}{ }{ }*. U konkretnom slučaju to može biti urađeno izmenom sadržaja datoteke *user-item-component.html* na sledeći način:

```
<p>  
    Hello {{name}}  
</p>
```

Još jednom neophodno je istaći da je u šablonu dodat kod koji predstavlja sintaksu koja do sada nije bila razmatrana - *{}{ name }{ }*. Par vitičastih zagrada predstavlja segment koji se naziva *šablonskim tagom* (template tag), po nekad se, u literaturi, šaljivo naziva i *brkovima* (mustaches).

Šta god da se nalazi između navedenih zagrada predstavlja *izraz*. Budući da je šablon povezan sa komponentom, vrednost izraza *name* odgovara vrednosti *this.name* iz konstruktoru klase komponente.

Sada je moguće proveriti urađeni posao. Osvežava se stranica i, ako je sve urađeno kako treba, dobija se sledeći izlaz:



Slika 3.8 Prikazivanje korisnika [izvor: autor]

✓ Poglavlje 4

Rad sa nizovima

PETLJA NGFOR

Ideja je da se proširi primer tako da može da manipuliše kolekcijom objekata.

Pokazni primer funkcioniše tako što je moguće uputiti pozdravnu poruku samo jednom korisniku. U nastavku, ideja je da se proširi ova funkcionalnost tako da može da manipuliše kolekcijom objekata, odnosno da svaki od korisnika dobije pozdravnu poruku.

Angular omogućava iteraciju preko liste objekata primenom naredbe `*ngFor`. Takođe, ideja je da se HTML kod ostavi nepromjenjenim u najvećoj mogućoj meri i kada se radi sa kolekcijom objekata (biće dodata samo petlja umesto pojedinačnih objekata).

Ukoliko ste radili sa starijom verzijom AngularJS 1.x moguće je da ste koristili naredbu (direktivu) ng-repeat. Ova naredba radi identično kao ngFor.

KOMPONENTA ZA KOLEKCIJE

Neophodno je kreirati novu komponentu i šablon za prikazivanje liste korisnika

Da bi kolekcija korisnika mogla da bude prikazana, neophodno je kreirati novu komponentu i šablon za prikazivanje liste korisnika. U terminalu se navodi sledeća instrukcija:

```
ng generate component user-list
```

Takođe, u datoteci `app.component.html` neophodno je zameniti tag za prikazivanje pojedinačnih korisnika, tagom koji prikazuje informacije koje se odnose na kolekciju korisnika. To je prikazano sledećim listingom:

```
<h1>
  {{title}}
<app-hello-world></app-hello-world>

<app-user-list></app-user-list>
</h1>
```

Za razliku od klase `UserItemComponent`, klasa `UserListComponent` bi trebalo da ima mogućnost rada sa nizom objekata tipa `string`. Iz navedenog razloga, biće kreirana njena osobina `names` koja će predstavljati niz stringova, odnosno imena korisnika.

Dalje, u Angularu nizovi se predstavljaju uglastim zagradama - `[]`. Ovo može biti iskorišćeno da se u konstruktoru klase kreira konkretan niz koji odgovara osobini `names` klase komponente `UserListComponent`.

Sledećim listingom je demonstriran kod klase `UserListComponent` sa navedenim izmenama:

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

@Component({
  selector: 'app-user-list',
  templateUrl: './user-list.component.html',
  styleUrls: ['./user-list.component.css']
})
export class UserListComponent implements OnInit {

  names: string[];
  name:string;

  constructor() {
    this.name = "";
    this.names = ['Vlada', 'Tina', 'Laza', 'Sofija', 'Isidora'];
  }

  ngOnInit(): void {
  }

}
```

ŠABLON ZA KOLEKCIJE OBJEKATA

Neophodno je obaviti ažuriranje šablon za prikazivanje liste imena

Da bi celokupan posao oko prikazivanja podataka, sa nizom korisnika, bio uspešno okončan, neophodno je obaviti ažuriranje šablon za prikazivanje liste imena. U ovom kontekstu biće korišćena petlja `*ngFor` čiji će zadatak biti da:

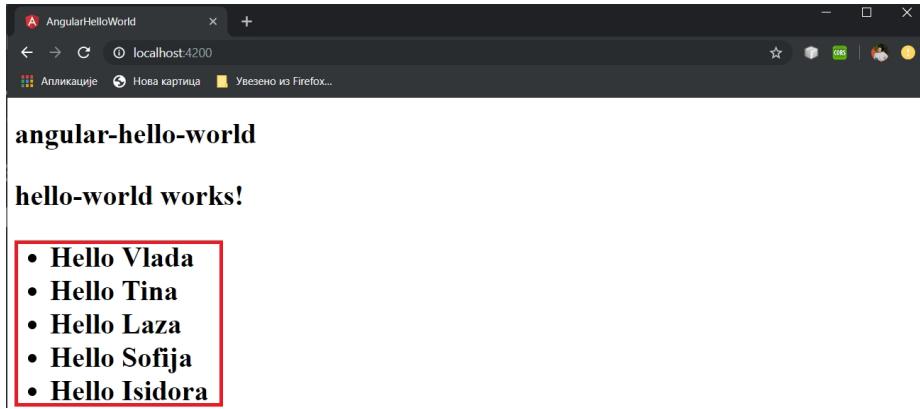
- prođe preko liste objekata;
- generiše novi tag za svakog od njih.

Ažurirani šablon `user-list.component.html` je prikazan sledećim listingom:

```
<ul>
  <li *ngFor="let name of names">Hello {{ name }}</li>
</ul>
```

I priloženog listinga može se primetiti da se u svakoj iteraciji dodaje nov član (tag ``) u neuređenu listu (tag ``). Kako funkcioniše petlja `*ngFor`? U principu, radi se o klasičnoj `for-each` petlji. Deo koda ove petlje "`let name of names`" ukazuje da petlja kreira lokalnu promenljivu `name` u kojoj se čuva vrednost člana niza `names` iz tekuće iteracije petlje.

Sada je moguće učitati početnu stranicu aplikacije ponovo i, ako je sve urađeno na pravi način, trebalo bi da bude dobijen sledeći izlaz:



Slika 4.1 Pozdravne poruke za niz imena [izvor: autor]

▼ Poglavlje 5

Komponenta potomak

NOVA ULOGA KOMPONENTE USERITEMCOMPONENT

Namera je da se iskoristi komponenta UserItemComponent kao komponenta potomak

Za novo razmatranje biće ponovo aktivirana komponenta *UserItemComponent*. Umesto dodavanja pojedinačnih imena primenom komponente *UserListComponent*, namera je da se iskoristi komponenta *UserItemComponent* kao komponenta potomak. To znači da umesto direktnog kreiranja pozdravne poruke za članove niza korisnika biće omogućeno komponenti *UserItemComponent* da specificira šablon (i funkcionalnosti) za svaku stavku u listi. Da bi ovo bilo moguće neophodne su tri stvari:

- Podešavanje komponente *UserListComponent* da koristi komponentu *UserItemComponent* (u šablonu);
- Podešavanje komponente *UserItemComponent* da prihvata vrednosti za promenljivu *name* sa ulaza;
- Podešavanje šablona *UserListComponent* da prosleđuje vrednost *name* ka *UserItemComponent*.

U nastavku, neophodno je izvesti i objasniti navedene korake.

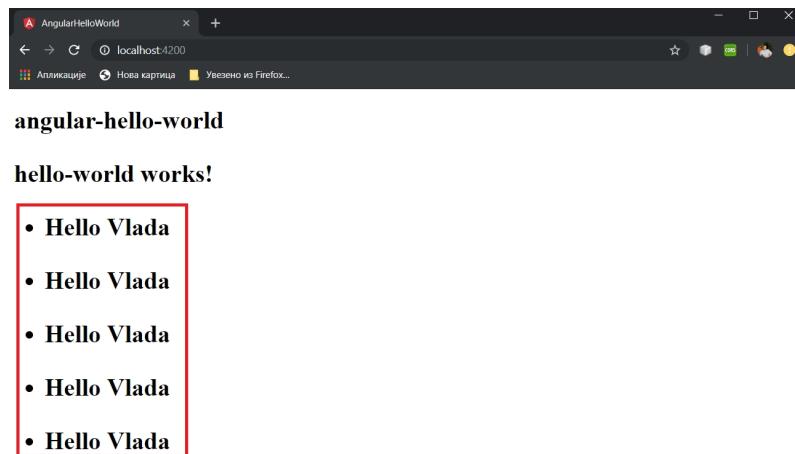
DODAVANJE POTOMAK KOMPONENTE USERITEMCOMPONENT

Dodaje se tag potomka u šablon roditeljske komponente.

Komponenta *UserItemComponent* specificira selektor *app-user-item* kojim je određen njen tag. Sledeći zadatak jeste dodavanje navedenog taga u šablon *user-list.component.html* roditeljske komponente. To može biti urađeno na sledeći način:

```
<ul>
  <li *ngFor = "let name of names">
    <app-user-item></app-user-item>
  </li>
</ul>
```

Moguće je primetiti da je linija *Hello {{name}}* zamenjena tagom *<app-user-item>*. Ako se u veb pregledaču osveži stranica biće dobijen sledeći izlaz:



Slika 5.1 Izlaz nakon dodavanja potomka u šablon roditelja [izvor: autor]

Petljia radi svoj posao, ali umesto niza razlicitih imena uvek se pojavljuje isto ime. Problem je sto trenutno ne postoji mehanizam u aplikaciji za prosledivanje podataka komponenti potomka. Srecom, [Angular](#) poseduje rešenje za navedeni problem primenom dekoratora (anotacije) [@Input](#).

PODEŠAVANJE KOMPONENTE DA PRIHVATA ULAZ

Angular poseduje rešenje za ovaj problem primenom dekoratora (anotacije) Input

Kao što je istaknuto, Angular poseduje rešenje za ovaj problem primenom dekoratora (anotacije) [@Input](#). Za početak neophodno je prisetiti se da u konstruktoru klase [UserItemComponent](#) postoji podešena osobina [name](#) na način: `this.name = 'Vlada'`.

Za ostvarivanje zacrtanih ciljeva, prvo bi trebalo redefinisati kod ove klase na sledeći način:

```
import { Component,
  OnInit,
  Input // ovo je dodato
} from '@angular/core';

@Component({
  selector: 'app-user-item',
  templateUrl: './user-item.component.html',
  styleUrls: ['./user-item.component.style']
})
export class UserItemComponent implements OnInit {

  @Input() name: string; // dodata anotacija

  constructor() {
    //obrisano podešavanje osobine
  }
}
```

```
ngOnInit() {  
}  
}
```

Iz priloženog listinga je moguće primetiti promenu na osobini `name` koja je obeležena dekoratorom `@Input`. O ulazima i izlazima u Angular aplikacijama biće više govora u narednoj lekciji. Za sada, dovoljno je da se zna da ova sintaksa omogućava prosleđivanje vrednosti iz roditeljskog šablona.

Da mi ovaj dekorator mogao da bude korišćen, neophodno je obaviti i njegovo dodavanje u `import` delu koda klase komponente.

Konačno, odustaje se od zadavanja podrazumevane vrednosti za ime i ovaj deo koda se briše iz konstruktora.

Sada se postavlja novo pitanje. Kako se koristi promenljiva koja je označena kao ulazna vrednost?

UKOLIKO NOVE ANGULAR VERZIJE VRAĆAJU GREŠKU: *Property 'name' has no initializer and is not definitely assigned in the constructor,* potrebno je da u datoteci `tsconfig.json`, dodate sledeću liniju:

"strictPropertyInitialization": false

PROSLEĐIVANJE ULAZNE VREDNOSTI

Za prosleđivanje vrednosti komponenti koristi se sintaksa uglaste zagrade u okviru šablona

Za prosleđivanje vrednosti komponenti koristi se sintaksa uglaste zagrade [] u okviru šablona. Ažurirani šablon `user-list.component.html` ima sledeću formu:

```
<ul>  
    <li *ngFor="let name of names">  
        <app-user-item [name]="name"></app-user-item>  
    </li>  
</ul>
```

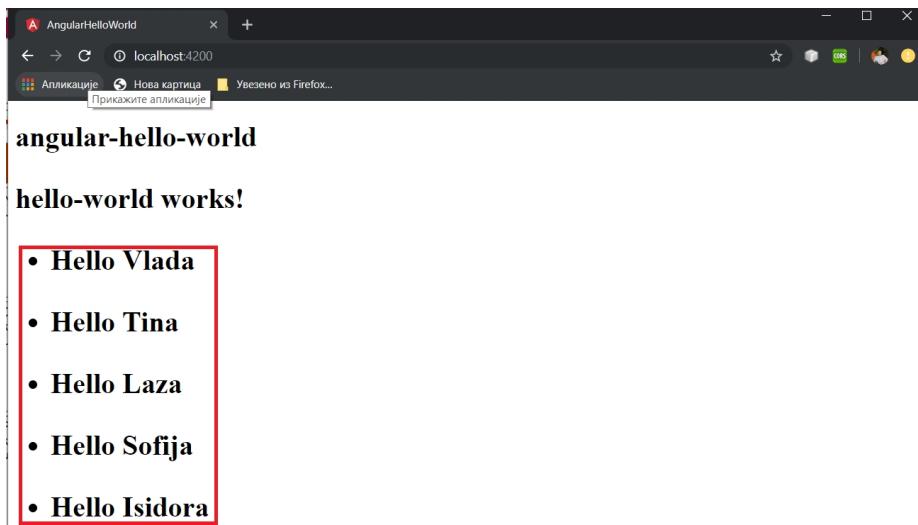
Iz priloženog listinga moguće je sagledati da je problem prosleđivanja ulazne vrednosti rešen na veoma jednostavan način dodavanjem novog atributa unutra taga `<app-user-item tag>` - `[name] = "name"`.

Kroz jednostavan primer urađene su veoma bitne stvari koje studenti u ovom trenutnu moraju da razumeju:

- iteracija preko niza (imena);
- kreiranje nove komponente (`UserItemComponent`) za svaki član niza imena;

- prosleđivanje vrednosti (imena) u ulaznu osobinu (`name`) komponente potomka (`UserItemComponent`).

Nakon urađenih korekcija, program ponovo funkcioniše kako se od njega očekuje, a to je prikazano sledećom slikom.



Slika 5.2 Primer prosleđivanja vrednosti potomak komponenti [izvor: autor]

Ova lekcija sadrži video materijal. Ukoliko želite da pogledate ovaj video morate da otvorite LAMS lekciju.

▼ Poglavlje 6

Komponente Angular aplikacije

ULAZNA TAČKA APLIKACIJE

Svaka aplikacija poseduje ulaznu tačku.

Svaka aplikacija poseduje ulaznu tačku. Aplikacija koja je kreirana i analizirana, u ovoj lekciji, prevedena je primenom alata [Angular CLI](#) koji se bazira na alatu poznatom pod nazivom [Webpack](#). Aplikacija se prevodi i pokreće pozivom komande:

```
ng serve
```

Naredba `ng` će pretražiti datoteku `angular.json` sa ciljem pronalaženja ulazne tačke aplikacije. Sledećim koracima je prikazano kako naredba `ng` pronađi kreirane komponente aplikacije:

- `angular.json` specificira glavni fajl aplikacije - u ovom slučaju to je `main.ts`;
- `main.ts` je ulazna tačka za aplikaciju i pokretanje aplikacije;
- pokretački proces pokreće `Angular modul` - o modulima će detaljno biti govora veoma brzo;
- `appModule.ts` specificira koja komponenta predstavlja komponentu najvišeg nivoa. U konkretnom slučaju to je `appComponent.ts`;
- navedena komponenta poseduje tag `<app-user-list>` i kreira listu svih korisnika.

SISTEM ANGULAR MODULA

Neophodno je staviti veći fokus na sistem Angular modula.

Kao što je navedeno u prethodnom izlaganju, neophodno je staviti veći fokus na [sistemu Angular modula](#) - [NgModule](#).

Angular neguje moćan sistem modula. Kada se podiže Angular aplikacija, ne pokreće se komponenta direktno. Umesto toga kreira se `NgModule` koji ukazuje na komponentu koju bi trebalo učitati.

Sledećim listingom je priložen delimičan kod datoteke `app.module.ts`.

```
@NgModule({
  declarations: [
    AppComponent,
    HelloWorldComponent,
    UserItemComponent,
    UserListComponent
```

```
],
imports: [
  BrowserModule,
  AppRoutingModule
],
providers: [],
bootstrap: [AppComponent]
})
export class AppModule { }
```

Prvo što je moguće primetiti iz listinga jeste dekorator `@NgModule`. Kao i svi dekoratori on ima zadatak da doda meta podatke u klasu. U konkretnom slučaju podeljuje četiri ključa:

- `declarations`;
- `imports`;
- `providers`;
- `bootstrap`.

DEKLARACIJE

Ključem declarations su specificirane komponente.

Ključem `declarations` su označene komponente koje definiše ovaj modul. Ovo predstavlja veoma važnu ideju Angular okvira. Neophodno je deklarisati komponente pre nogo što budu upotrebljene u šablonima.

`NgModule` može biti razmatran i kao paket, a deklaracije ukazuju na komponente koje modul poseduje. Takođe, moguće je primetiti da kada se kreira komponenta, primenom naredbe:

```
ng generate
```

ovaj `ng` alat će automatski dodati komponentu u listu deklarisanih komponenata. Ideja je upravo da kada se kreira nova komponenta, `ng` alat podrazumeva da ona mora da pripada aktuelnom `NgModule`.

IMPORTS KLJUČ

Ključ imports opisuje koje zavisnosti modul poseduje.

Ključ `imports` opisuje koje zavisnosti modul poseduje. Budući da je kreirana aplikacija koja za izvršavanje zahteva veb pregledač, neophodno je dodati modul `BrowserModule`. Ukoliko modul zavisi od još nekih modula i njih je potrebno dodati na ovu listu.

VAŽNO!!!

Neophodno je napraviti razliku između `import` i `imports`. `Import` se koristi u klasama, a `imports` u modulima. Najjednostavnije objašnjenje bi moglo biti da se pod ključem `imports` dekoratora `@NgModule` čuvaju elementi koji se koriste u `HTML` šablonima ili prilikom `umetanja zavisnosti`

(*dependency injection*). O konceptu umetanja zavisnosti biće detaljno govora u ovom predmetu.

KLJUČEVI PROVIDERS I BOOTSTRAP

Provajderi se koriste prilikom umetanja zavisnosti. Ključ bootstrap omogućava učitavanje AppComponent komponente.

Provajderi se koriste prilikom umetanja zavisnosti. Klasa obeležena dekoratorom `@NgModule` omogućava umetanje konkretnih servisa tokom izvršavanja aplikacije tako što ih deklariše pod ključem *decorators*.

Ključ `bootstrap` kazuje Angularu da kada se ovaj modul koristi za pokretanje aplikacije, neophodno je obaviti učitavanje `AppComponent` komponente kao komponente najvišeg nivoa.

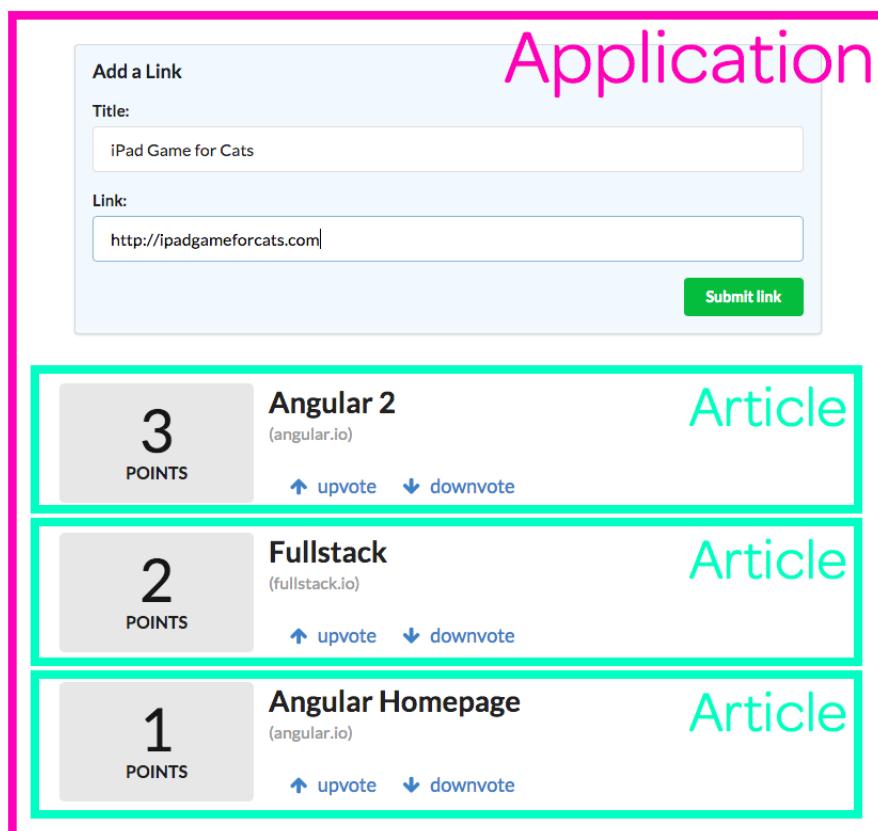
✓ Poglavlje 7

Pokazni primer - proširenje aplikacije

BUDUĆI IZGLED APLIKACIJE

Aplikacija bi trebalo da bude pojednostavljena kopija aplikacije Reddit.

Kao što je istaknuto u prethodnom izlaganju, konačni cilj je kreiranje aplikacije koja bi trebalo da bude pojednostavljena kopija aplikacije [Reddit](#). Sledećom slikom je priložena ideja samog izgleda aplikacije koja će u nastavku biti kreirana po principu korak - po - korak proširenjem inicijalnog koda.



Slika 7.1.1 Predloženi izgled aplikacije [izvor: autor]

KOMPONENTE APLIKACIJE

Neophodno je dodati dve nove komponente u aplikaciju.

Aplikaciju će činiti dve komponente:

1. celokupna aplikacija koja će sadržati formu za dodavanje novih proizvoda (na slici je istaknuto ljubičastom bojom);
2. pojedinačni proizvodi (istaknuti su zelenom mint bojom).

U velikim, realnim aplikacijama, forma za dodavanje novih proizvoda bi verovatno predstavljala novu komponentu. Međutim, postojanje nove komponente bi u ovom trenutku učinilo prosleđivanje podataka kompleksnijim. Za sada, da bi bolje i lakše shvatili ovu problematiku, akcenat će biti na jednostavnosti.

Sada je moguće početi sa izradom konkretne aplikacije. U terminalu razvojnog okruženja, primenom [Angular CLI](#), kucamo naredbu za kreiranje novog projekta pod nazivom [angular-reddit](#):

```
ng new angular-reddit
```

Ubrzo započinje kreiranje novog projekta što rezultuje sledećim izlazom u terminalu.

```
Windows PowerShell
Copyright (C) Microsoft Corporation. All rights reserved.

Try the new cross-platform PowerShell https://aka.ms/pscore6

PS C:\Users\Vlada\Documents\Angular> ng new angular-reddit
? would you like to add Angular routing? Yes
? which stylesheet format would you like to use? CSS
CREATE angular-reddit/angular.json (3657 bytes)
CREATE angular-reddit/package.json (1288 bytes)
CREATE angular-reddit/README.md (1038 bytes)
CREATE angular-reddit/tsconfig.json (543 bytes)
CREATE angular-reddit/tslint.json (1988 bytes)
CREATE angular-reddit/.editorconfig (246 bytes)
CREATE angular-reddit/.gitignore (631 bytes)
CREATE angular-reddit/browserslist (429 bytes)
CREATE angular-reddit/karma.conf.js (1026 bytes)
CREATE angular-reddit/tsconfig.app.json (270 bytes)
CREATE angular-reddit/tsconfig.spec.json (270 bytes)
CREATE angular-reddit/src/favicon.ico (5430 bytes)
CREATE angular-reddit/src/index.html (300 bytes)
CREATE angular-reddit/src/main.ts (372 bytes)
CREATE angular-reddit/src/polyfills.ts (2838 bytes)
CREATE angular-reddit/src/styles.css (80 bytes)
CREATE angular-reddit/src/test.ts (642 bytes)
CREATE angular-reddit/src/assets/.gitkeep (0 bytes)
CREATE angular-reddit/src/environments/environment.prod.ts (51 bytes)
CREATE angular-reddit/src/environments/environment.ts (662 bytes)
CREATE angular-reddit/src/app/app-routing.module.ts (246 bytes)
CREATE angular-reddit/src/app/app.module.ts (393 bytes)
CREATE angular-reddit/src/app/app.component.html (1152 bytes)
CREATE angular-reddit/src/app/app.component.spec.ts (1119 bytes)
CREATE angular-reddit/src/app/app.component.ts (218 bytes)
CREATE angular-reddit/src/app/app.component.css (0 bytes)
CREATE angular-reddit/e2e/protractor.conf.js (810 bytes)
CREATE angular-reddit/e2e/tsconfig.json (214 bytes)
CREATE angular-reddit/e2e/src/app.e2e-spec.ts (643 bytes)
CREATE angular-reddit/e2e/src/app.po.ts (251 bytes)
```

Slika 7.1.2 Kreiranje novog Angular projekta [izvor: autor]

CSS

Pre bavljenja glavnom komponentom aplikacije cilj je da se izgled aplikacije malo ulepša.

Pre bavljenja glavnom komponentom aplikacije cilj je da se izgled aplikacije malo ulepša. Ukoliko pravite aplikaciju iz početka, možete se poslužiti urađenim primerom kojeg možete preuzeti odmah nakon ovog objekta učenja. U vaš projekat prekopirajte sledeće datoteke i foldere u folder sa vašom aplikacijom:

- *src/index.html*
- *src/styles.css*

- *src/app/vendor*
- *src/assets/images*.

Za ulepšavanje projekta korišćen je *Semantic-UI* (više na linku <https://semantic-ui.com/>).

GLAVNA KOMPONENTA APLIKACIJE

*Glavna komponenta aplikacije je određena datotekom *src/app/app.component.ts**

Sada je moguće pristupiti kreiranju nove komponente čiji će zadatak biti da:

1. čuva listu postojećih proizvoda;
2. sadrži formu za dodavanje novih proizvoda.

Glavna komponenta aplikacije je određena datotekom *src/app/app.component.ts*. Otvara se datoteka i u njoj se trenutno nalazi dobro poznat inicijalni kod:

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

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

Moguće je primetiti da osobina se *title* automatski dodaje u klasu *AppComponent*. **Ovu liniju je moguće slobodno obrisati iz razloga što nigde u aplikaciji neće biti korišćen naslov komponente.**

Sada je moguće načiniti izmene i u šablonu glavne komponente kreiranjem forme za dodavanje novih linkova. Biće korišćeni stilovi iz paketa *semantic-ui* za lepši izgled stranice. Sledećim listingom data je redefinisana datoteka *app.component.html*.

```
<form class="ui large form segment">
  <h3 class="ui header">Add a Link</h3>

  <div class="field">
    <label for="title">Title:</label>
    <input name="title">
  </div>
  <div class="field">
    <label for="link">Link:</label>
    <input name="link">
  </div>
</form>
```

Kreiran je šablon koji sadrži dva taga za unos (*input*): jedan za unos naziva proizvoda, a drugi za odgovarajući link. Sada je moguće pokrenuti aplikaciju sa ciljem sagledavanja do sada postignutih rezultata.



Slika 7.1.3 Kreirana forma na stranici Angular aplikacije [izvor: autor]

DODAVANJE INTERAKCIJE

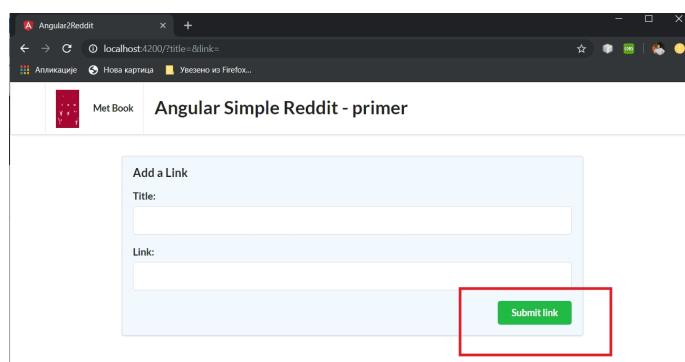
Neophodno je kreirati mehanizam interakcije u formi dugmeta za potvrđivanje unosa sa forme.

U dosadašnjem radu je kreirana **forma** koja poseduje tagove za unos podataka. Međutim, ne postoji mehanizam za potvrđivanje i slanje unetih podataka na dalju obradu. U tom svetu, neophodno je kreirati mehanizam interakcije u formi dugmeta za potvrđivanje unosa sa forme.

Kada je unos na formi potvrđen, neophodno je pozvati funkciju koja kreira i dodaje link. Ovo je moguće obaviti dodavanjem interaktivnog događaja za GUI element (dugme). Na primer, dodavanje funkcije, za događaj dugmeta tipa *onClick*, moguće je obaviti na sledeći način u kodu šablona razmatranog u prethodnom izlaganju:

```
<button (click)="addArticle()" class="ui positive right floated button">  
    Submit link  
</button>
```

Stranica će dobiti izgled kao na sledećoj slici.



Slika 7.1.4 Dodato dugme za interakciju na formi [izvor: autor]

Klikom na dugme se još uvek ne dešava ništa. Neophodno je definisati funkciju `addArticle()`, u okviru klase `AppComponent` koja se poziva i izvršava klikom na kreirano dugme. To je moguće obaviti na sledeći način:

```
export class AppComponent {  
    addArticle(title: HTMLInputElement, link: HTMLInputElement): boolean {  
        console.log(`Adding article title: ${title.value} and link: ${link.value}`);  
        return false;  
    }  
}
```

Iz koda je moguće primetiti da funkcija `addArticle()` uzima dva argumenta: `title` i `link`. Neophodno je napraviti izmene u šablonu da bi dugme moglo da prosledi ove dve vrednosti u poziv metode `addArticle()`.

```
<form class="ui large form segment">  
    <h3 class="ui header">Add a Link</h3>  
  
    <div class="field">  
        <label for="title">Title:</label>  
        <input name="title" id="title" #newtitle> <!-- promenjeno -->  
    </div>  
    <div class="field">  
        <label for="link">Link:</label>  
        <input name="link" id="link" #newlink> <!-- promenjeno -->  
    </div>  
  
    <!-- dugme prosleđuje metodi parametre newtitle i newlink-->  
    <button (click)="addArticle(newtitle, newlink)" class="ui positive right floated button">  
        Submit link  
    </button>  
  
</form>
```

DODAVANJE INTERAKCIJE - ANALIZA

Primenom specijalnog znaka "hash" ukazuje se Angularu da pridruži tagove lokalnim promenljivim.

Iz prethodno priloženog listinga moguće je primetiti primenu specijalnog znaka `#` preko kojeg se ukazuje Angularu da pridruži tagove lokalnim promenljivim. Dodavanjem, parametara `#newtitle` i `#newlink` odgovarajućim `<input/>` elementima, moguće je njihovo prosleđivanje kao promenljivih u metodu `addArticle()` nakon klika na kreirano dugme.

Napravljene su sledeće korekcije:

1. Kreirano je dugme u HTML kodu koji navodi korisnika gde treba da klikne;
2. Kreirana je funkcija `addArticle()` kojom je definisano šta bi trebalo obaviti kada se klikne na dugme;

3. Za dugme je dodat atribut (`click`) koji ukazuje da se pozove funkcija `addArticle()` kada se klikne na dugme;
4. Dodati su atributi `#newtitle` i `#newlink` u odgovarajuće `<input>` tagove.

Neophodno je dati kratko objašnjenje za svaki od navedenih koraka.

POVEZIVANJE ULAZA SA VREDNOSTIMA

Povezivanje ulaza sa vrednostima ima za efekat dostupnost promenljive izrazima unutar pogleda.

Neophodno je obratiti pažnju na prvi `<input>` tag:

```
<input name="title" #newtitle>
```

navedeni kod ukazuje okviru Angular da poveže vrednost unetu u polju `<input>` sa promenljivom `newtitle`. Ova sintaksa se još naziva "rešavanjem" (`resolve`). Ona ima za efekat dostupnost promenljive izrazima unutar pogleda (šablona, veb stranice). Varijablom `newtitle` je sada određen objekat koji reprezentuje ulazni DOM (Document Object Model) element (posebno, tip je `HTMLInputElement`). Pošto je `newtitle` objekat, vrednost `<input>` taga se dobija pozivom `newtitle.value`.

Potpuno identična priča važi za promenljivu `newlink` i neće biti ponavljana.

POVEZIVANJE AKCIJA SA DOGAĐAJIMA

Atributom (`click`) je definisana akcija koja se izvršava kada se klikne na dugme.

Ako je još jednom pogleda poslednji priloženi HTML listing moguće je primetiti da je tagu `<button>` dodat atribut (`click`) kojim je definisana akcija koja se izvršava kada se klikne na dugme. Kada se desi navedeni događaj, poziva se funkcija `addArticle()` sa dva argumenta: `newtitle`

i `newlink`. Odakle se javljaju funkcija i navedeni argumenti?

1. `addArticle()` je funkcija definisana u klasi komponente `AppComponent`;
2. `newtitle` se dobija "rešavanjem" izraza (`#newtitle`) koji pripada `<input>` tagu pod nazivom `title`;
3. `newlink` se dobija "rešavanjem" izraza (`#newlink`) koji pripada `<input>` tagu pod nazivom `link`;

Sve zajedno moguće je priložiti kroz sledeći listing.

```
<!-- dugme prosleđuje metodi parametre newtitle i newlink-->
<button (click)="addArticle(newtitle, newlink)"
         class="ui positive right floated button">
```

```
Submit link  
</button>
```

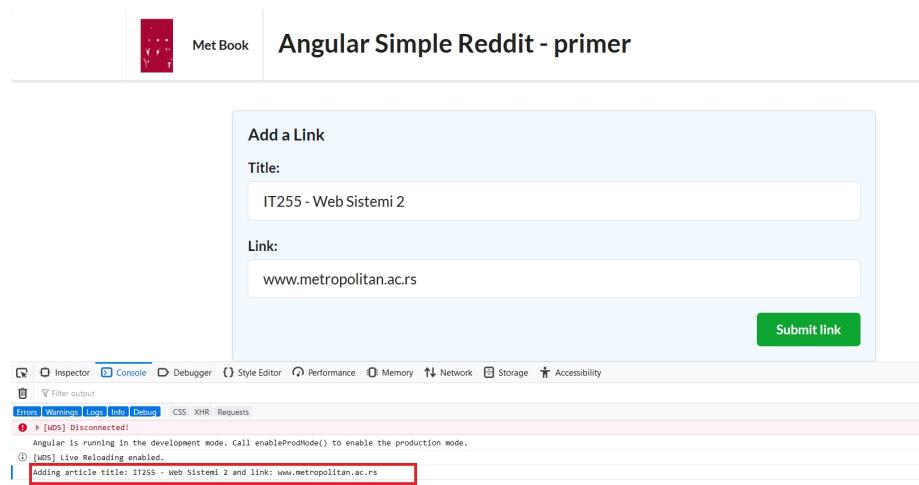
DEFINISANJE LOGIKE AKCIJE

Argumenti metode koja rukuje događajem su objekti tipa `HTMLInputElement` i ne predstavljaju direktno unete vrednosti.

Za klasu `AppComponent` definisana je funkcija pod nazivom `addArticle()`. Funkcija, kao što je već istaknuto, uzima dva argumenta: `title` i `link`. Opet, važno je razumeti da su oba argumenta objekti tipa `HTMLInputElement` i ne predstavljaju direktno unete vrednosti. Da bi se dobila uneta vrednost sa ulaza, na primer za objekat `title`, neophodno je obaviti poziv izraza `title.value`. Za sada će biti dovoljno prikazati izlaz, nastao pozivom ove metode, u konzoli. Ovo je omogućeno dodavanjem sledećeg koda u metodu `addArticle()` klase komponente `AppComponent`.

```
export class AppComponent {  
  addArticle(title: HTMLInputElement, link: HTMLInputElement): boolean {  
    console.log(`Adding article title: ${title.value} and link: ${link.value}`);  
    return false;  
  }  
}
```

Sada je moguće obaviti testiranje urađenog posla. Učitava se ponovo stranica, popunjava se forma i u konzoli se prate rezultati nakon klika na dugme za potvrdu forme. Ako je sve u redu, rezultat može biti kao na sledećoj slici.



Slika 7.1.5 Prikaz rezultata obrade forme u konzoli veb pregledača [izvor: autor]

DODAVANJE KOMPONENTE ARTICLE

Javlja se potreba za kreiranjem nove komponente koja će rukovati dodavanjem članaka na stranicu.

Očigledno je da posao oko izrade željene aplikacije nije priveden kraju. Postoji forma za dodavanje proizvoda ali se onii ne prikazuju nigde. Iz razloga što se svaki proizvod, koji je unet putem forme, prikazuje u listi na odgovarajućoj stranici, javlja se potreba za kreiranjem nove komponente koja će rukovati ovim zadatkom.

Pa neka u sledećem koraku bude kreirana tražena komponenta koja će predstavljati pojedinačne postavljene proizvode (kao na slici 6).

ng generate component article



Slika 7.1.6 Izgled dodatog članka na stranici [izvor: autor]

Za kreiranu komponentu je potrebno obaviti posebnu brigu oko krajnjih definicija za sledeće delove:

1. Definisanje *ArticleComponent* pogleda u šablonu;
2. Definisanje *ArticleComponent* osobina obeležavanjem klase dekoratorom *@Component*
3. Definisanje *ArticleComponent* klase komponente koja čuva programsku logiku navedene komponente.

U nastavku sledi diskusija u vezi sa istaknutim zadacima.

KREIRANJE ŠABLONA ARTICLECOMPONENT

Šablon će biti kreiran preko datoteke article.component.html.

Šablon će biti kreiran preko datoteke article.component.html.

```
<div class="four wide column center aligned votes">
  <div class="ui statistic">
    <div class="value">
      {{ votes }}
    </div>
    <div class="label">
      Points
    </div>
  </div>
</div>
<div class="twelve wide column">
  <a class="ui large header" href="{{ link }}">
    {{ title }}
  </a>
  <ul class="ui big horizontal list voters">
```

```

<li class="item">
    <a href (click)="voteUp()">
        <i class="arrow up icon"></i>
        upvote
    </a>
</li>
<li class="item">
    <a href (click)="voteDown()">
        <i class="arrow down icon"></i>
        downvote
    </a>
</li>
</ul>
</div>

```

Šablonom su određene dve kolone:

1. broj glasova za proizvod, na levoj strani;
2. informacije o proizvodu, na desnoj strani.

Kolone su ulepšane CSS klasama: *four wide column* i *twelve wide column*, respektivno (rečeno je na početku da će biti korišćen *SemanticUI's CSS*).

Glasovi i naslovi se prikazuju u šablonu primenom izraza `{{ votes }}` i `{{ title }}`. Vrednosti se dobijaju preko osobina `vote` i `title` klase komponente `ArticleComponent` koja će ubrzo biti definisana.

Takođe, moguće je koristiti šablonske stringove u vrednostima atributa, kao u slučaju: `href="{{link}}"`. U ovom slučaju, vrednost za `href` biće dodata dinamički preko vrednosti linka iz klase komponente.

Za linkove `upvote` i `downvote` neophodno je definisati akcije. Biće upotrebljen `(click)` za povezivanje metoda `voteUp()` i `voteDown()` sa odgovarajućim kontrolama na stranici. Kada se klikne na link `upvote`, biće pozvana metoda `voteUp()` klase komponente `ArticleComponent`. Identičan slučaj je sa linkom `downvote` i odgovarajućom metodom `voteDown()`.

DEKORATOR KLASE ARTICLECOMPONENT

Obeležavanje dekoratorom omogućava primenu novog taga za prikazivanje članaka.

U nastavku, neophodno je obaviti obeležavanje klase komponente odgovarajućim dekoratorom `@Component`. Posebno je bitna vrednost ključa `selector` kojom je određen naziv taga koji će omogućiti primenu komponente u HTML stranici.

```

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

```

Iz priloženog listinga je moguće primetiti da je definisan tag `<app-article> </app-article>` koji odgovara aktuelnoj komponenti.

ZAVRŠNA DEFINICIJA KLASE ARTICLECOMPONENT

Kreira se programska logika klase komponente ArticleComponent.

Konačno, na red dolazi i završna definicija klase komponente `ArticleComponent`. Neophodno je otvoriti datoteku `article.component.ts` i dodati kod priložen sledećim listingom:

```
export class ArticleComponent implements OnInit {  
  
    @HostBinding('attr.class') cssClass = 'row';  
    votes: number;  
    title: string;  
    link: string;  
  
    constructor() {  
        this.title = 'IT255 - Veb sistemi 1';  
        this.link = 'http://www.metropolitan.ac.rs';  
        this.votes = 10;  
    }  
  
    voteUp() {  
        this.votes += 1;  
    }  
  
    voteDown() {  
        this.votes -= 1;  
    }  
  
    ngOnInit() {  
    }  
}
```

Kreirane su četiri osobine klase `ArticleComponent`:

1. `cssClass` - predstavlja CSS klasu koja će biti primenjena na "hosta" ove komponente;
2. `votes` - broj koji prestavlja zbir glasova vraćen nakon svih poziva metoda `voteUp()` i `voteDown()`, respektivno;
3. `title` - string koji označava naziv proizvoda;
4. `link` - string koji označava link proizvoda.

Ovde je cilj da svaki novi proizvod bude dodat u poseban red na stranici. Za stilizaciju će biti upotrebljen, kao i do sada, `Semantic-UI` koji obezbeđuje `CSS` klasu za redove pod nazivom `row`.

U Angularu, komponenta `host` je određena elementom sa kojim je komponenta povezana. Moguće je podesiti osobinu host elementa primenom dekoratora `@HostBinding()`. U ovom slučaju, traži se od `Angulara` da vrednost klase host elemenata bude u sinhronizaciji sa svojstvom `cssClass`.

Da bi dekorator `@HostBinding()`, uopšte mogao da bude korišćen na ovakav način, neophodno je proširiti `import` sekciju klase na sledeći način:

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

KLASA ARTICLECOMPONENT - DODATNA RAZMATRANJA

Nastavlja se analiza kreirane klase komponente.

Nastavlja se analiza kreirane klase komponente. Sada je pažnja usmerena ka konstruktoru gde su podešene podrazumevane vrednosti za atribute na sledeći način:

```
constructor() {
    this.title = 'IT255 - Veb sistemi 1';
    this.link = 'http://www.metropolitan.ac.rs';
    this.votes = 10;
}
```

Konačno, definicija klase je završena metodama koje omogućavaju korisnicima glasanje za dati proizvod: `voteUp()` i `voteDown()`.

```
voteUp() {
    this.votes += 1;
}

voteDown() {
    this.votes -= 1;
}
```

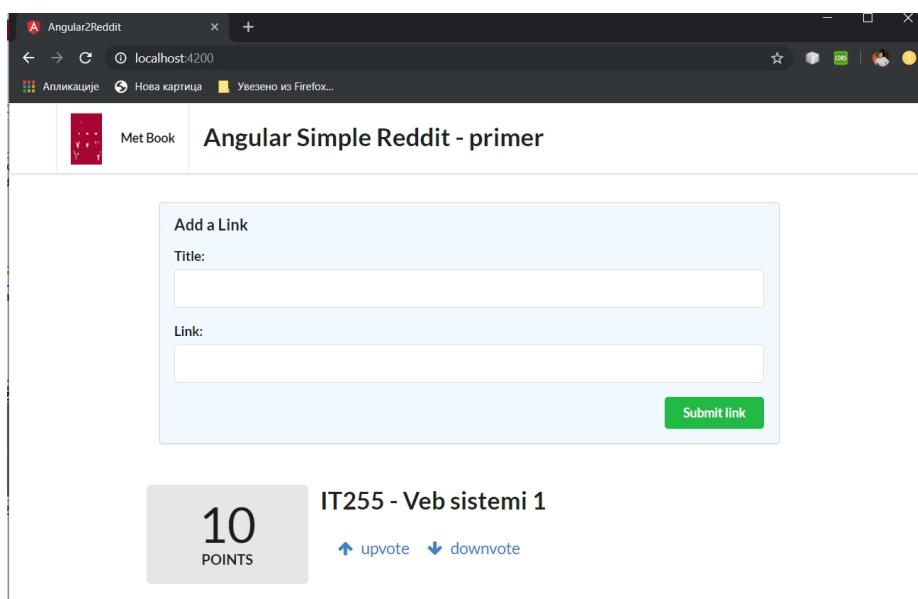
UPOTREBA KOMPONENTE APP-ARTICLE

Neophodno je dodati tag komponente na odgovarajuće mesto u HTML kodu.

Sa ciljem korišćenja kreirane komponente i činjenjem vidljivih podataka kojima ona manipuliše, neophodno je dodati tag komponente na odgovarajuće mesto u HTML kodu. To mesto je u šablonu komponente `AppComponent` (u datoteci `app.component.html`) odmah iza završnog taga `</form>`. Sledećim, delimičnim, listingom ove datoteke, prikazana je opisana aktivnost.

```
***  
<button (click)="addArticle(newtitle, newlink)" class="ui positive right floated button">  
    Submit link  
</button>  
</form>  
  
<div class="ui grid posts">  
    <app-article>  
    </app-article>  
</div>
```

Sada je moguće testirati urađenu aplikaciju ponovnim učitavanjem stranice. Ukoliko se dobije izlaz kao na slici 7, sve je obavljeno na pravi način.



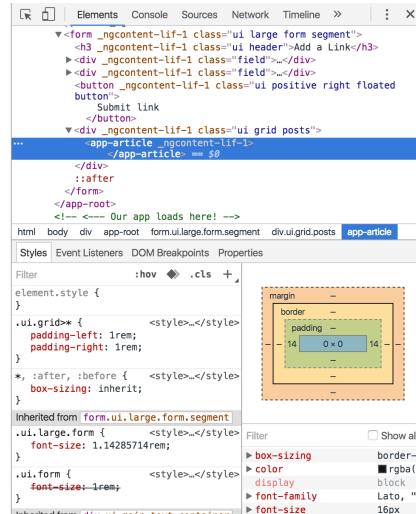
Slika 7.1.7 Izgled urađene aplikacije [izvor: autor]

DISKUSIJA O PREDNOSTI UPOTREBE ANGULAR CLI

Angular CLI automatski registruje kreiranu komponentu.

Ukoliko je komponenta *ArticleComponent* kreirana primenom Angular CLI, primenom instrukcije *ng generate*, po osnovnim podešavanjima Angular će biti upoznat sa postojanjem ove komponente jer će ona biti automatski dodata u datoteku *app.module.ts*. Aplikacija će biti uspešno prevedena i izvršena na način prikazan prethodnom slikom.

Međutim, ukoliko je izbegnut ovaj alat prilikom kreiranja komponente, prilikom osvežavanja veb pregledača moguće je primetiti da navedeni tag nije preveden, a to je moguće dodatno proveriti iz konzole veb pregledača (sledeća slika).



Slika 7.1.8 Greška prilikom interpretacije taga komponente [izvor: autor]

Problem je u tome da glavna komponenta `AppComponent` još uvek nije upoznata sa komponentom `ArticleComponent`.

Da bi navedeni problem bio rešen, neophodno je manuelno obaviti registrovanje komponente u datoteci `app.module.ts` na način prikazan sledećom slikom.

```
angular-reddit > src > app > ts app.module.ts > ...
1 import { BrowserModule } from '@angular/platform-browser';
2 import { NgModule } from '@angular/core';
3
4 import { AppRoutingModule } from './app-routing.module';
5 import { AppComponent } from './app.component';
6 import { ArticleComponent } from './article/article.component';
7
8 @NgModule({
9   declarations: [
10     AppComponent,
11     ArticleComponent
12   ],
13   imports: [
14     BrowserModule,
15     AppRoutingModule
16   ],
17   providers: [],
18   bootstrap: [AppComponent]
19 })
20 export class AppModule { }
```

Slika 7.1.9 Manuelno registrovanje komponente u datoteci `app.module.ts` [izvor: autor]

PROBLEM OSVEŽAVANJA STRANICA

Problem: Klikom na linkove za glasanje vrši se ponovno učitavanje stranice, a ne ažuriranje liste.

Prilikom testiranja izvršavanja kreirane aplikacije, moguće je uočiti dodatni problem. Klikom na linkove za glasanje vrši se ponovno učitavanje stranice umesto prostog ažuriranja liste proizvoda. **JavaScript**, po osnovnim podešavanjima, širi klik događaj prema svim roditeljskim komponentama. Upravo zbog ovoga, veb pregledač pokušava da prati prazan link koji mu ukazuje na ponovno učitavanje sadržaja.

Da bi ovaj problem bio rešen neophodno je dodati malo koda u metodu koja rukuje događajem klika na link. Metoda mora da vrati vrednost **false** koja će osigurati da veb pregledač neće pokušati, nakon poziva metode, da osveži stranicu.

Jasno je da su metode koje rukuju događajima, u našem primeru, **voteUp()** i **voteDown()**. Sledi listing sa njihovim korigovanim kodom:

```
voteUp() {  
    this.votes += 1;  
    return false;  
}  
  
voteDown() {  
    this.votes -= 1;  
    return false;  
}
```

Sada je sve u redu sa linkovima i glasanje za proizvod ide glatko i nije praćeno ponovnim učitavanjima stranice.

✓ 7.1 Obrada više redova

PROBLEM DODAVANJA NOVIH PROIZVODA

Postoji samo jedan proizvod na stranici i ne postoji mehanizam za dodavanje novih

Prilikom testiranja izvršavanja aplikacije moguće je uočiti još jedan problem. Trenutno, **postoji samo jedan proizvod na stranici i ne postoji mehanizam za dodavanje novih**, ukoliko ne bude dodat još jedan tag. Međutim, ako ovo bude urađeno, svi proizvodi će imati identičan sadržaj, a to i nije ono što se očekuje od ove aplikacije.

KREIRANJE MODELSKE KLASE ARTICLE

Dobru praksu predstavlja pisanje koda kojim se razdvaja struktura podataka od koda komponente.

Dobru praksu predstavlja pisanje Angular kodakojim se razdvaja struktura podataka koja se koristi od koda komponente. Za realizovanje ovog zadatka, neophodno je kreirati strukturu

podataka koja predstavlja pojedinačne proizvode. Ovakva struktura podataka je poznata pod nazivom modelska klasa ili model u MVC (*Model - View - Controller*) šablonima.

U folderu `/article` aktuelnog projekta biće kreirana datoteka `article.model.ts` kojom će biti definisana modelska klasa `Article`. Sledеćim listingom je data njena definicija:

```
export class Article {  
    title: string;  
    link: string;  
    votes: number;  
  
    constructor(title: string, link: string, votes?: number) {  
        this.title = title;  
        this.link = link;  
        this.votes = votes || 0;  
    }  
}
```

Kao što je moguće primetiti iz koda, radi se o običnoj (*plain*) klasi, a ne o Angular komponenti. Svaki proizvod ima: naslov, link i broj glasova.

Kada se kreira nov objekat za proizvod, osobine `title` i `link` su obavezne. Parametar `vote` je opcionalan (obezbeđeno znakom ? na kraju naziva parametra) sa podrazumevanom vrednošću 0.

PRIDRUŽIVANJE MODELA KOMPONENTI

Neophodno je ažuriranje klase komponente da bi mogla da koristi model.

Neophodno je ažuriranje klase komponente `ArticleComponent` da bi mogla da koristi modelsku klasu `Article`. Umesto direktno čuvanja osobina u klasi komponente `ArticleComponent`, osobine će biti čuvane u instanci klase `Article`.

Prvi korak jeste kreiranje `import` instrukcije u klasi komponente `ArticleComponent` za klasu `Article`.

```
import { Article } from './article.model';
```

Nakon ovoga, moguće je upotrebiti objekat `Article` u `ArticleComponent` klasi komponente na sledeći način:

```
export class ArticleComponent implements OnInit {  
  
    @HostBinding('attr.class') cssClass = 'row';  
    article: Article;  
  
    constructor() {
```

```

this.article = new Article(
    'IT255 - Veb sistemi 1',
    'http://www.mwtropolitan.ac.rs',
    10);
}

voteUp() {
    this.article.votes += 1;
    return false;
}

voteDown() {
    this.article.votes -= 1;
    return false;
}

ngOnInit() {
}

}

```

Iz koda je moguće primetiti šta je promenjeno: umesto čuvanja naziva, linka i broja glasova u odgovarajućim osobinama komponente, čuva se referenca na proizvod. Kada se vrši glasanje, glasovi se na ažuriraju za komponentu već za *article* objekat.

KOREKCIJE ŠABLONA

Neophodno obaviti izmene na HTML šablonima da bi mogli da prihvataju vrednosti promenljivih sa pravih lokacija.

Navedene promene u kodu uvode i dodatne promene koje je neophodno obaviti na *HTML* šablonima da bi mogli da prihvataju vrednosti promenljivih sa pravih lokacija. Da bi ovo bilo moguće, neophodno je obaviti promenu tagova šablonu da čitaju iz *article* objekata. To znači, da bi izraz `{{ votes }}` trebalo da bude zamenjen izrazom `{{ article.votes }}`. Potpuno isto važi i za osobine *title* i *link*.

```

<div class="four wide column center aligned votes">
    <div class="ui statistic">
        <div class="value">
            {{ article.votes }}
        </div>
        <div class="label">
            Points
        </div>
    </div>
</div>
<div class="twelve wide column">
    <a class="ui large header" href="{{ article.link }}">
        {{ article.title }}
    </a>
</div>

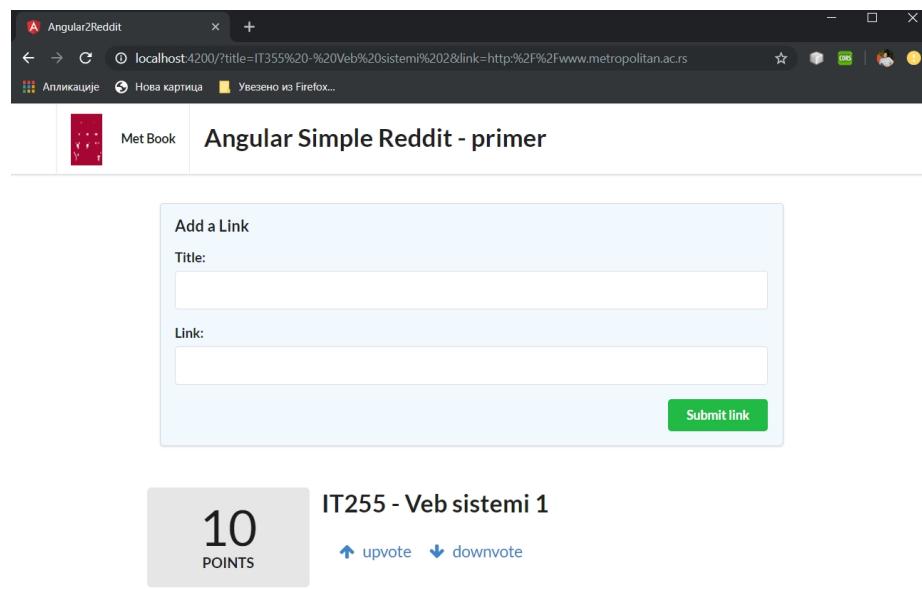
```

```

</a>
<ul class="ui big horizontal list voters">
    <li class="item">
        <a href (click)="voteUp()">
            <i class="arrow up icon"></i>
            upvote
        </a>
    </li>
    <li class="item">
        <a href (click)="voteDown()">
            <i class="arrow down icon"></i>
            downvote
        </a>
    </li>
</ul>
</div>

```

Sada je aplikacija ponovo vraćena na nivo funkcionalnosti na kojem je bila pre korekcija. Sve radi osim dodavanja novih proizvoda.



Slika 7.2.1 Aplikacija sa delimičnim funkcionalnostima [izvor: autor]

DODATNI PROBLEM SA ENKAPSULACIJOM

Metode `voteUp()` i `voteDown()` narušavaju enkapsulaciju klase `Article`.

Trenutna situacija sa aplikacijom je nešto bolja od polazne ali i dalje ima nedostataka. Metode `voteUp()` i `voteDown()` narušavaju enkapsulaciju klase `Article` budući da menjaju direktno unutrašnje osobine njenih objekata. Ovde je narušen, konkretno, *Demeterov zakon* (https://en.wikipedia.org/wiki/Law_of_Demeter) koji govori da posmatrani objekat trebalo da ima što manje saznanja o strukturi drugih objekata.

Ovde je problem što klasa komponente `ArticleComponent` ima puno saznanja o unutrašnjoj implementaciji klase `Article`. Da bi ovaj problem bio prevaziđen, metode `voteUp()` i

`voteDown()` će biti dodate u klasu `Article`. Takođe, biće dodata i funkcija `domain()` o kojoj će uskoro biti govora.

```
export class Article {
    title: string;
    link: string;
    votes: number;

    constructor(title: string, link: string, votes?: number) {
        this.title = title;
        this.link = link;
        this.votes = votes || 0;
    }

    voteUp(): void {
        this.votes += 1;
    }

    voteDown(): void {
        this.votes -= 1;
    }

    // domain() je funkcija koja izdvaja
    // domen iz URL, biće uskoro objašnjena
    domain(): string | null{
        try {
            // e.g. http://foo.com/path/to/bar
            const domainAndPath: string = this.link.split('/')[1];
            // e.g. foo.com/path/to/bar
            return domainAndPath.split('/')[0];
        } catch (err) {
            return null;
        }
    }
}
```

Sada je neophodno obaviti i izmene u klasi `ArticleComponent` da bi navedene komponente mogle neometano da budu pozivane.

```
export class ArticleComponent implements OnInit {

    @HostBinding('attr.class') cssClass = 'row';
    article: Article;

    constructor() {
        this.article = new Article(
            'IT255 - Veb sistemi 1',
            'http://www.metropolitan.ac.rs',
            10);
    }

    voteUp() {
        this.article.voteUp();
    }
}
```

```
    return false;
}

voteDown() {
    this.article.voteDown();
    return false;
}

ngOnInit() {
}

}
```

U suštini, sve je sada isto kao pre sa razlikom što je dobijen čistiji i jasniji kod.

ČUVANJE KOLEKCIJE PROIZVODA

Sada je moguće kreirati kod koji omogućava rad sa listom objekata.

Sada je moguće kreirati kod koji omogućava rad sa listom objekata klase *Articles*. Prvi korak je korekcija klase *AppComponent* koja će sada imati mogućnost čuvanja kolekcije proizvoda.

```
import { Component } from '@angular/core';
import { Article } from './article/article.model'; //dodata je ova linija

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

export class AppComponent {
  articles: Article[]; //osobina komponente

  constructor() {
    this.articles = [
      new Article('IT255 - Veb sistemi 1', 'http://www.metropolitan.ac.rs', 3),
      new Article('Fullstack', 'http://fullstack.io', 2),
      new Article('Angular Homepage', 'http://angular.io', 1),
    ];
  }
}
```

Prva stvar koju je moguće primetiti jeste da klasa poseduje liniju koda:

```
articles: Article[];
```

Na ovaj način je rečeno da je osobina `articles`, zapravo, niz objekata klase `Article`. Ovo je moguće zapisati na još jedan način `Array<Article>`. Radi se o primeni generičkih tipova - `Array` je kolekcija u kojoj je moguće čuvati samo `Article` objekte.

Da bi bilo moguće pristupati klasi `Article` neophodno je prvo izvršiti njeno importovanje (linija koda 2). Kolekciju je moguće napuniti preko konstruktora komponente (linije koda 13 - 18).

PODEŠAVANE KLASE ARTICLECOMPONENT DEKORATOROM INPUT

Glavno pitanje je kako obaviti njihovo prosleđivanje liste proizvoda ka komponenti.

Sada postoji lista objekata `Articles` i glavno pitanje je kako obaviti njihovo prosleđivanje ka komponenti `ArticlesComponent`?

Ponovo na scenu stupa dekorator `@Input`. Na početku, klasa `ArticleComponent` ima sledeći oblik:

```
export class ArticleComponent implements OnInit {  
  
    @HostBinding('attr.class') cssClass = 'row';  
    article: Article;  
  
    constructor() {  
        this.article = new Article(  
            'IT255 - Veb sistemi 1',  
            'http://www.metropolitan.ac.rs',  
            10);  
    }  
}
```

Ovde je glavni problem što postoji "čvrsto" kodiran konkretan proizvod u konstruktoru. Poenta kreiranja komponenata nije samo enkapsulacija, već i višestruka upotrebljivost.

Ono što je zapravo cilj jeste pokušaj podešavanja proizvoda kojeg bi trebalo prikazati na stranici. Ako, na primer, postoje dva proizvoda `article1` i `article2`, trebalo bi da postoji mogućnost ponovne upotrebe komponente `app-article` prosleđivanjem proizvoda kao parametra komponenti, na sledeći način:

```
<app-article [article]="article1"></app-article>  
<app-article [article]="article2"></app-article>
```

Angular poseduje veoma jednostavno rešenje za ovaj problem - primenjuje dekorator `@Input` na osobinu komponente.

```
class ArticleComponent {  
    @Input() article: Article;
```

```
// ...
```

Sada, ukoliko bi postojao proizvod čuvan u varijabli *myArticle*, mogao bi iz pogleda da bude prosleđen komponenti ArticleComponent na sledeći način:

```
<app-article [article]="myArticle"></app-article>
```

Dalje, a ovo je veoma važno, vrednost *this.article* instance *ArticleComponent* biti podešena sa *myArticle*. O varijabli *myArticle* može se razmišljati kao o prosleđenom parametru (ulazu, inputu) komponenti.

LISTING KLASE ARTICLECOMPONENT

Izmenjeni kod klase ArticleComponent sadrži dekorator Inputs

Nakon detaljne analize, moguće je sada priložiti izmenjeni kod klase *ArticleComponent* u koji je ugrađena primena dekoratora *@Input* na polje tipa *Article*.

```
import { Component,
         OnInit,
         HostBinding,
         Input // dodato je ovo
     } from '@angular/core';
import { Article } from './article.model';

@Component({
  selector: 'app-article',
  templateUrl: './article.component.html',
  styleUrls: ['./article.component.css']
})
export class ArticleComponent implements OnInit {

  @HostBinding('attr.class') cssClass = 'row';
  @Input() article: Article;

  constructor() {
    // ovako se više ne zadaje proizvod
    //zadaje se preko Input - a
  }

  voteUp() {
    this.article.voteUp();
    return false;
  }

  voteDown() {
    this.article.voteDown();
    return false;
  }
}
```

```
ngOnInit() {  
}  
}
```

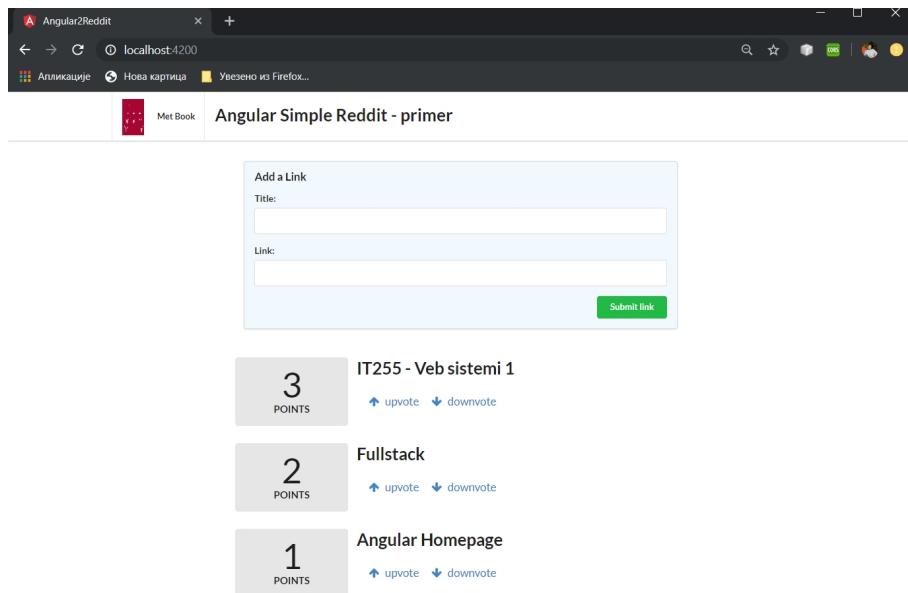
PRIKAZIVANJE LISTE PROIZVODA

Podešava se AppComponent da bi mogla da omogući prikazivanje svih proizvoda.

U ranijem izlaganju podešena je komponenta *AppComponent* da čuva niz proizvoda. Dalje, podešava se *AppComponent* da bi mogla da omogući prikazivanje svih proizvoda. Da bi ovo bilo omogućeno, biće korišćena *ngFor* direktiva za iteraciju preko liste proizvoda i da prikaže tag *<app-article>* za svakog od njih. Upravo će navedeno biti dodato u šablon komponente *AppComponent* odmah ispod taga *<form>*:

```
...  
Submit link  
</button>  
</form>  
  
<!-- početak dodavanja ovde -->  
<div class="ui grid posts">  
  <app-article *ngFor="let article of articles" [article]="article">  
    </app-article>  
</div>  
<!-- kraj dodavanja ovde -->
```

Ponovnim učitavanjem stranice, u veb pregledaču, dobijeni su inovirani rezultati našeg rada na projektu. To je prikazano sledećom slikom.



Slika 7.2.2 Inovirana aplikacija [izvor: autor]

▼ 7.2 Korišćenje forme

DODAVANJE NOVIH PROIZVODA

Dodavanje novih proizvoda je rešeno posebnom metodom klase AppComponent.

Sada je potreno posvetiti pažnju metodi `addArticle()` kojom će biti omogućeno dodavanje novih proizvoda u kada se klikne na dugme za potvrdu forme.

Metoda može da bude smeštena na kraj sadržaja datoteke `app.component.ts` ima sledeći listing:

```
addArticle(title: HTMLInputElement, link: HTMLInputElement): boolean {
    console.log(`Adding article title: ${title.value} and link: ${link.value}`);
    this.articles.push(new Article(title.value, link.value, 0));
    title.value = '';
    link.value = '';
    return false;
}
```

Izvršavanjem metode se postiže:

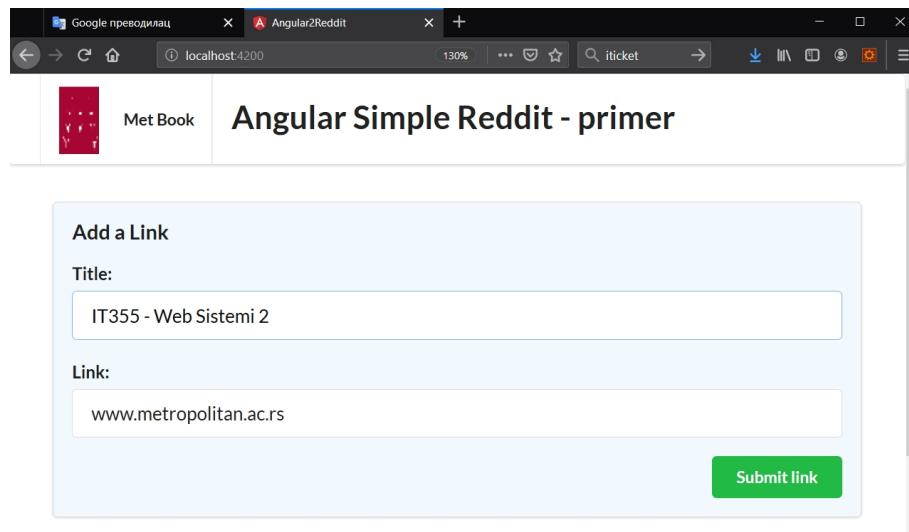
1. kreiranje novog `Article` objekta sa prosleđenim naslovom i linkom;
2. dodaje kreirani objekat u postojeći niz objekata;
3. resetuje polja za unos podataka za nov `Article` objekat.

Sada je aplikacija dostigla željeni nivo zrelosti i može biti demonstrirana.

DEMONSTRACIJA KOMPLETIRANE APLIKACIJE

Slede izgledi ekrana koji demonstriraju izvršavanje aplikacije.

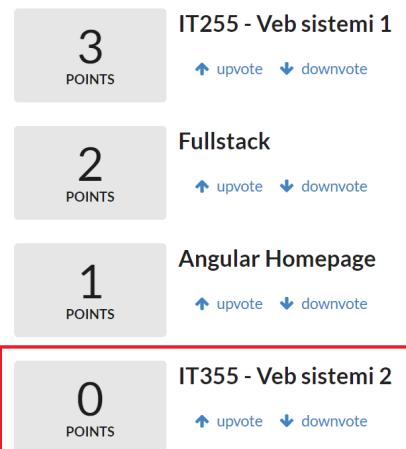
Ponovo je potrebno osvežiti veb pregledač. Učitava se stranica sa formom u kojoj je moguće uneti nov proizvod. To može biti dokumentovano sledećom slikom.



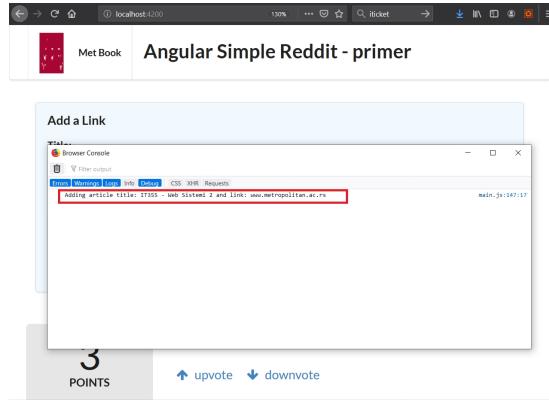
Slika 7.3.1 Dodavanje novog objekta putem forme [izvor: autor]

Klikom na dugme *Submit link* poziva se metoda *addArticle()* glavne komponente *AppComponent*. Metoda pokazuje rezultate na dva načina:

1. dodaje nov objekat u kolekciju i on biva vidljiv zajedno sa ostalim objektima (slika 2);
2. šalje rezultat u konzolu veb pregledača (slika 3).



Slika 7.3.2 Nov objekat u listi objekata [izvor: autor]



Slika 7.3.3 Rezultat slanja objekta sa forme u logu veb pregledača. [izvor: autor]

PRIKAZIVANJE DOMENA PROIZVODA

Pored naziva linka, prikazuje se i njegov domen.

Kao zanimljiv dodatak moguće je dodati napomenu pored linka u formi domena na koji će korisnik biti preusmeren ukoliko klikne na link. Kod ove metode je već viđen u klasi Article, a ovde zbog preglednosti može biti sagledan izolovano:

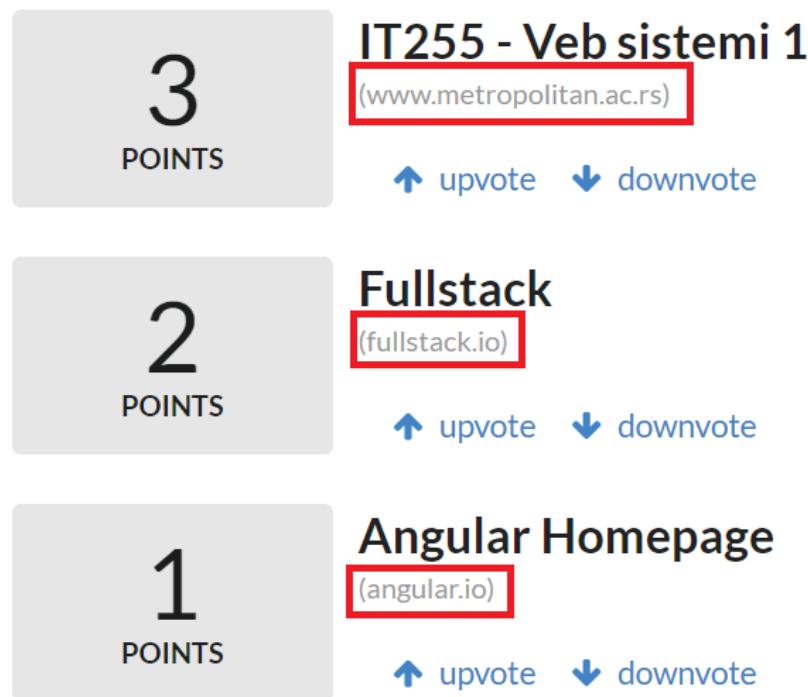
```
// domain() je funkcija koja izdvaja
// domen iz URL, biće uskoro objašnjena
domain(): string {
    try {
        // e.g. http://foo.com/path/to/bar
        const domainAndPath: string = this.link.split('/')[1];
        // e.g. foo.com/path/to/bar
        return domainAndPath.split('/')[0];
    } catch (err) {
        return null;
    }
}
```

Sada ide mala izmena u šablonu *ArticleComponent*.

```
<div class="twelve wide column">
    <a class="ui large header" href="{{ article.link }}"
       {{ article.title }}>
    </a>
    <!-- domen proizvoda -->
    <div class="meta">{{ article.domain() }}</div>
    <ul class="ui big horizontal list voters">
        <li class="item">
            <a href (click)="voteUp()">
                <i class="arrow up icon"></i>
                upvote
            </a>
        </li>
    </ul>
</div>
```

Slika 7.3.4 Dodavanje domena uz link u šablonu [izvor: autor]

Konačno, rezultat rada je moguće sagledati na sledećoj slici.



Slika 7.3.5 Domen i link na istom mestu [izvor: autor]

▼ Poglavlje 8

Objavljivanje aplikacije

PROCES OBJAVLJIVANJA APLIKACIJE

Objavljivanje aplikacije (deploying) predstavlja čin postavljanja kreiranog koda na server.

Objavljivanje aplikacije (*deploying*) predstavlja čin postavljanja kreiranog koda na server odakle mogu da joj pristupe razni korisnici. Šire rečeno, ideja je u izvođenju sledećih koraka:

- kompajliranje celokupnog TypeScript koda u JavaScript kojeg veb pregledač razume;
- pakovanje celokupnog JavaScript koda u jedan ili dva fajla;
- podizanje (*upload*) celokupnog JavaScript koda, HTML, CSS i slika na server.

Konačno, Angular aplikacija je HTML datoteka koja učitava JavaScript kod. Cilj je podizanje kreiranog koda na nekom računaru na Internetu.

KREIRANJE VERZIJE PROGRAMA ZA PRODUKCIJU

Angular CLI je upotrebljen za generisanje verzije kreirane aplikacije koja je spremna za produkciju

Angular CLI je alat koji može, između ostalog, da bude upotrebljen za generisanje verzije kreirane aplikacije koja je spremna za produkciju. Sama procedura je veoma jednostavna. Neophodno je ući u radni folder aplikacije, u ovom slučaju to je Angular\angular-reddit. Nakon toga se navodi naredba, ovde data u opštem obliku:

```
ng build --base-href /naziv-foldera/ --prod
```

U konkretnom slučaju to je urađeno kao na sledećoj slici.

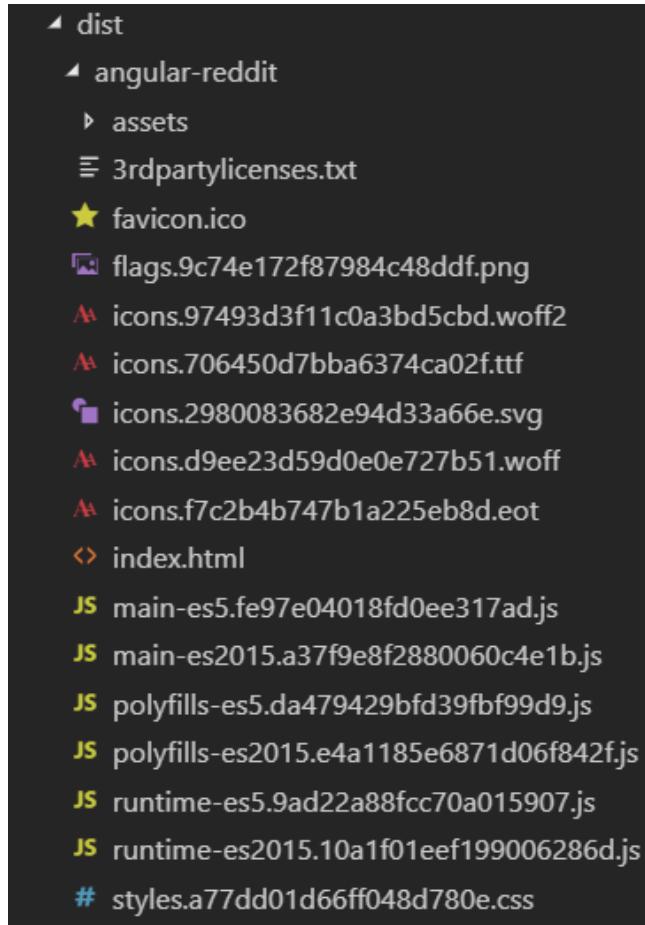
```
C:\Users\Vlada\Documents\Angular\angular-reddit>ng build --base-href /prvaapp/ --prod
chunk {0} runtime-es2015.10a1f01eef199006286d.js (runtime) 1.41 kB [entry] [rendered]
chunk {1} main-es2015.a37f9e8f2880060c4e1b.js (main) 215 kB [initial] [rendered]
chunk {2} polyfills-es2015.e4a1185e6871d06f842f.js (polyfills) 36.4 kB [initial] [rendered]
chunk {3} styles.a77dd01d66ff048d780e.css (styles) 469 kB [initial] [rendered]
Date: 2019-08-05T19:54:38.705Z - Hash: 83e21855452f6cce70c9 - Time: 31676ms

chunk {0} runtime-es5.9ad22a88fcc70a015907.js (runtime) 1.41 kB [entry] [rendered]
chunk {1} main-es5.fe97e04018fd0ee317ad.js (main) 249 kB [initial] [rendered]
chunk {2} polyfills-es5.da479429bfd39fbf99d9.js (polyfills) 113 kB [initial] [rendered]
Date: 2019-08-05T19:55:05.067Z - Hash: f07b727bffc8fc0912fc - Time: 26248ms

C:\Users\Vlada\Documents\Angular\angular-reddit>
```

Slika 8.1 Kreiranje verzije programa za produkciju [izvor: autor]

Naredba će za izvršavanje trebati malo vremena ali će uskoro, kao rezultat njenog izvršavanja, biti kreiran folder *dist* u okviru radnog foldera i njegov sadržaj je prikazan sledećom slikom. Datoteke unutar ovog foldera predstavljaju potpun rezultat prevođenja kreirane aplikacije.



Slika 8.2 Potpun rezultat prevođenja kreirane aplikacije [izvor: autor]

POSTAVLJANJE APLIKACIJE NA SERVER

Postoji puno načina da se hostuje HTML i JavaScript.

Postoji puno načina da se *hostuje* HTML i JavaScript. Za ovaj primer je možda najlakši za upotrebu server *now* dostupan na linku: <https://vercel.com/home>.

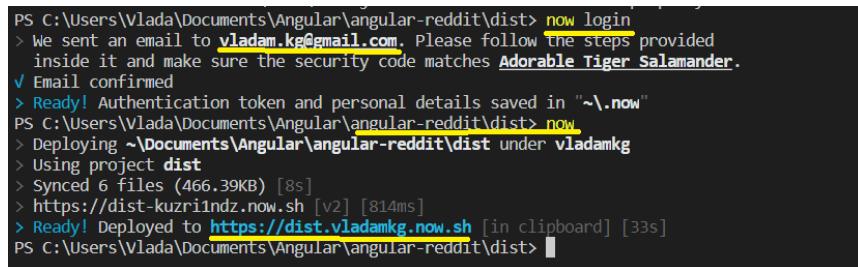
Instaliranje podrške za podizanje aplikacije na ovom serveru ide primenom *npm* naredbe:

```
npm install -g now
```

Zatim je, kada je ova podrška dostupna neophodno otići u folder *dist* i pokrenuti naredbu *now* kojom se aplikacija postavlja na server:

```
cd dist # change into the dist folder
now
```

Komanda će ubrzo pitati i za neke korisničke informacije, poput email adrese kojom je kreiran *now* nalog. Nakon potvrđivanja naloga započinje postavljanje aplikacije na serveru. Navedeno je prikazano sledećom slikom.



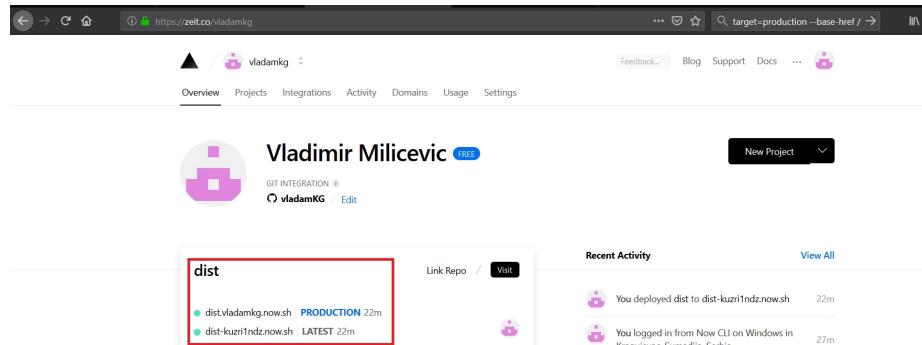
```

PS C:\Users\Vlada\Documents\Angular\angular-reddit\dist> now login
> We sent an email to vladam.kg@gmail.com. Please follow the steps provided
inside it and make sure the security code matches Adorable Tiger Salamander.
✓ Email confirmed
> Ready! Authentication token and personal details saved in "~\.now"
PS C:\Users\Vlada\Documents\Angular\angular-reddit\dist> now
> Deploying ~\Documents\Angular\angular-reddit\dist under vladamkg
> Using project dist
> Synced 6 files (466.39KB) [8s]
> https://dist-kuzri1ndz.now.sh [v2] [814ms]
> Ready! Deployed to https://dist.vladamkg.now.sh [in clipboard] [33s]
PS C:\Users\Vlada\Documents\Angular\angular-reddit\dist>

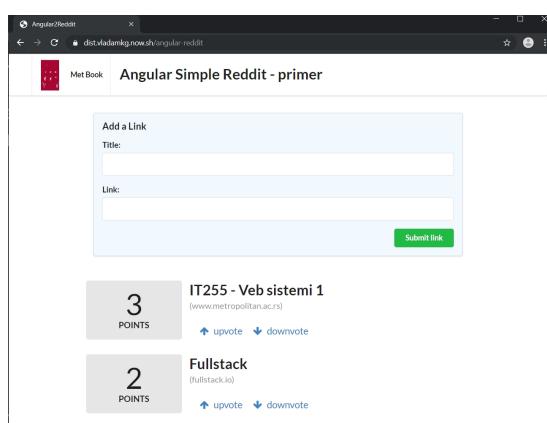
```

Slika 8.3 Postavljanje aplikacije na serveru [izvor: autor]

Sa slike je moguće videti i URL na kojem je dostupna aplikacija. Pozivom tog URL u veb pregledaču, pokreće se aplikacija sa servera.



Slika 8.4 Podignuta aplikacija na now serveru [izvor: autor]



Slika 8.5 Pokrenuta aplikacija sa servera [izvor: autor]

✓ Poglavlje 9

Uvod u Angular dodatni materijali

DODATNI MATERIJALI

Proširivanje naučenog na predavanju.

Proučite dodatnu literaturu:

1. <https://angular.io/>
2. <https://angular.io/tutorial>
3. <https://www.w3schools.com/angular/>
4. <https://www.tutorialspoint.com/angular4/>
5. <https://nodejs.org/en/>
6. <https://code.visualstudio.com/>

▼ Poglavlje 10

Pokazne vežbe 6

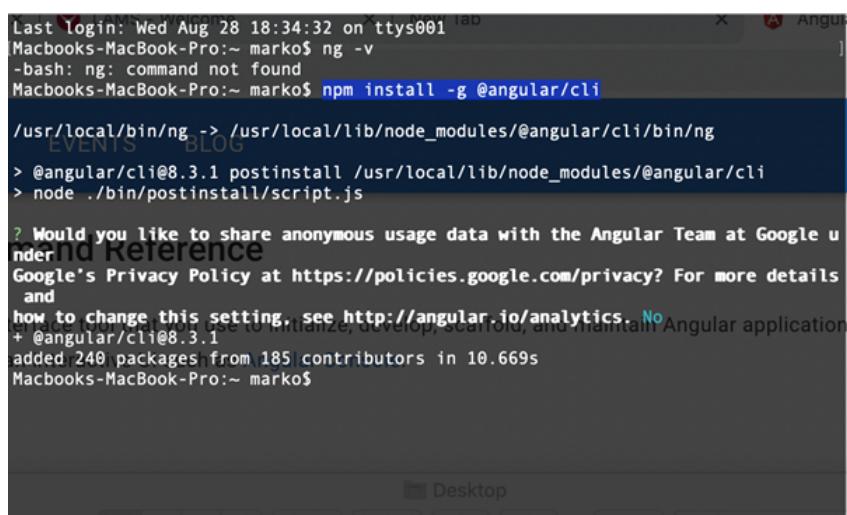
PRIPREMA ANGULAR CLI - POKAZNA VEŽBA (TRAJANJE: 45 MINUTA)

Cilj ove lekcije predstavlja kreiranje projekta kroz Angular CLI

Ukoliko to već nije učinjeno, prvi korak je preuzimanje i instaliranje [NodeJS](#) instalacionog paketa.

U nastavku je neophodno instalirati [Angular CLI](#) (*Angular Command Line Interface*). To možete učiniti navođenjem sledeće naredbe u terminalu vašeg razvojnog okruženja *Visual Studio Code*:

```
npm install -g @angular/cli
```



```
Last login: Wed Aug 28 18:34:32 on ttys001 100
Macbooks-MacBook-Pro:~ marko$ ng -v
-bash: ng: command not found
Macbooks-MacBook-Pro:~ marko$ npm install -g @angular/cli
/usr/local/bin/ng -> /usr/local/lib/node_modules/@angular/cli/bin/ng
> @angular/cli@8.3.1 postinstall /usr/local/lib/node_modules/@angular/cli
> node ./bin/postinstall/script.js

? Would you like to share anonymous usage data with the Angular Team at Google under the Reference
Google's Privacy Policy at https://policies.google.com/privacy? For more details and
how to change this setting, see http://angular.io/analytics. No
+ @angular/cli@8.3.1
added 240 packages from 185 contributors in 10.669s
Macbooks-MacBook-Pro:~ marko$
```

Slika 10.1 Instaliranje Angular CLI [izvor: autor]

Po uspešnoj instalaciji, pokretanjem komande [*ng help*](#), možemo videti da je instalacija uspešna kao na slici 2.

```

added 240 packages from 103 contributors in 10.005s
[Macbooks-MacBook-Pro:~ marko$ ng -v
Available Commands:
  add Adds support for an external library to your project, you use to initialize, develop, si
  analytics Configures the gathering of Angular CLI usage metrics. See https://v8.angular.i
o/cli/usage-analytics-gathering.
  build (b) Compiles an Angular app into an output directory named dist/ at the given outpu
t path. Must be executed from within a workspace directory.
  >deploy (d) Invokes the deploy builder for a specified project or for the default project
in the workspace.
  config Retrieves or sets Angular configuration values in the angular.json file for the wo
rkspace.
  doc (d) Opens the official Angular documentation (angular.io) in a browser, and searches
for a given keyword.
  >e2e (e) Builds and serves an Angular app, then runs end-to-end tests using Protractor.
  generate (g) Generates and/or modifies files based on a schematic.
  help Lists available commands and their short descriptions.
  >lint (l) Runs linting tools on Angular app code in a given project folder.
  new (n) Creates a new workspace and an initial Angular app.
  run Runs an Architect target with an optional custom builder configuration defined in you
r project.
  >serve (s) Builds and serves your app, rebuilding on file changes.
  test (t) Runs unit tests in a project.
  update Updates your application and its dependencies. See https://update.angular.io/
  >version (v) Outputs Angular CLI version.
  xi18n Extracts i18n messages from source code.

For more detailed help run "ng [command name] --help"
Macbooks-MacBook-Pro:~ marko$
```

Slika 10.2 Provera instalacije Angular CLI [izvor: autor]

KREIRANJE PROJEKTA

Nakon uspešne instalacije, pristupamo kreiranju projekta.

Nakon uspešne instalacije, pristupamo kreiranju projekta sledećom komandom:

```
ng new IT255
```

Prethodno navedena komanda će kreirati folder, sa svim propratnim kodom neophodnim za start određene aplikacije. Na slici 3 možemo videti da je neophodno izabrati Angular rutiranje, kao i SCSS stylesheet format, kako bismo dobili komajpler za SCSS o kom je bilo reči u nekoj od prethodnih lekcija.

```

Macbooks-MacBook-Pro:~/Desktop marko$ cd Workspace/
Macbooks-MacBook-Pro:Workspace marko$ ng new it255
? Would you like to add Angular routing? Yes
? Which stylesheet format would you like to use? SCSS  [ https://sass-lang.com/documentation/on-syntax#scss ]
```

Slika 10.3 Angular rutiranje - izbor [izvor: autor]

Po uspešno izvršenoj komandi, unutar direktorijuma pokrećemo sledeću komandu: „npm start“ za pokretanje projekta.

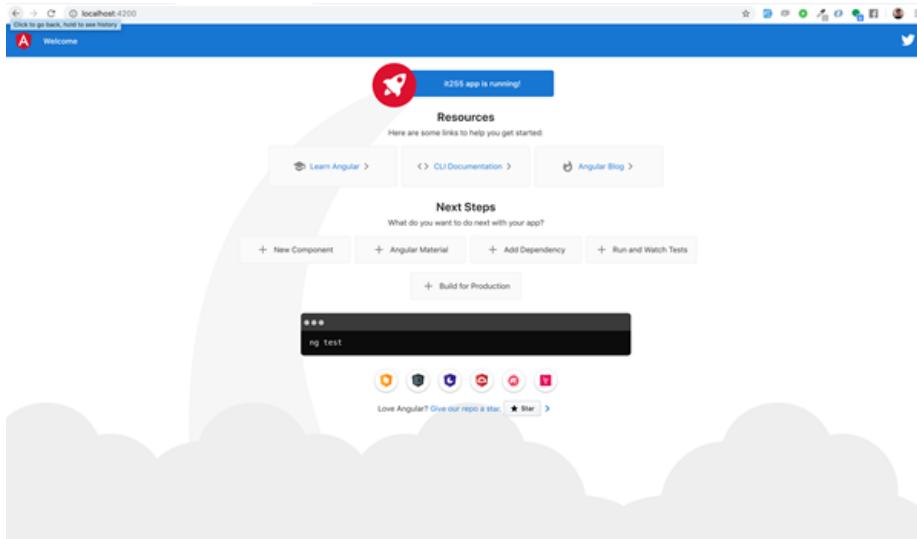
```

Macbooks-MacBook-Pro:it255 marko$ npm start
> it255@0.0.0 start /Users/marko/Workspace/it255
> ng serve
+ New Component + Angular Material + Build for Production
10% building 3/3 modules 0 active [wds]: Project is running at http://localhost:4200/webpack-dev-server/
[ wds]: webpack output is served from /
[ wds]: 404s will fallback to //index.html
chunk {main} main.js, main.js.map (main) 49.4 kB [initial] [rendered]
chunk {polyfills} polyfills.js, polyfills.js.map (polyfills) 264 kB [initial] [rendered]
chunk {runtime} runtime.js, runtime.js.map (runtime) 6.15 kB [entry] [rendered]
chunk {styles} styles.js, styles.js.map (styles) 10 kB [initial] [rendered]
chunk {vendor} vendor.js, vendor.js.map (vendor) 4.08 MB [initial] [rendered]
Date: 2019-08-28T17:07:24.458Z - Hash: 725a4b4889ed54d8d86b - Time: 7014ms
** Angular Live Development Server is listening on localhost:4200, open your browser on http://localhost:4200/ **
[ wdm]: Compiled successfully.
```

Slika 10.4 Prevođenje i pokretanje projekta [izvor: autor]

Vidimo da je naša aplikacija dostupna na sledećoj adresi:<http://localhost:4200/>.

Navođenjem navedene adrese u veb pregledač, dobija se ekran kao na sledećoj slici:



Slika 10.5 Inicijalni izgled kreiranog Angular projekta [izvor: autor]

KREIRANJE PRVOG VLASTITOG ANGULAR (TYPESCRIPT) KODA

Izmenom inicijalnog i dodavanjem novog koda prilagođavamo aplikaciju vlastitim zahtevima.

Pošto je uspešno, samostalno, kreiran prvi Angular projekat, napravićemo određene modifikacije nad datotekom [app.component.ts](#):

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

//dekorator
@Component({}

  selector: 'my-app',
  template: '<h1>{{name}}</h1>'
})

export class AppComponent {
  name: string = "Angular - IT255"
}
```

Listing pokazuje nekoliko značajnih elemenata:

- dekorator [@Component](#) koji govori da se radi o klasi komponente (glavne);
- [selector](#) - definiše tag pod kojim će biti prikazan sadržaj komponente;
- [template](#) - za razliku od prethodnog scenarija, kada je html dokument komponente bio posebna datoteka, na ovaj način možemo, takođe, da definišemo šablon komponente.

Takođe, inicijalni sadržaj datoteke index.html biće zamenjen sledećim:

```
<!DOCTYPE html>
<html>
  <head>
    <title>Angular QuickStart</title>
    <base href="/">
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1">
    <link rel="stylesheet" href="styles.css">

    <!-- Dodavanje skriptova -->
    <script src="/node_modules/core-js/client/shim.min.js"></script>

    <script src="/node_modules/zone.js/dist/zone.js"></script>
    <script src="/node_modules/systemjs/dist/system.src.js"></script>

    <script src="systemjs.config.js"></script>
    <script>
      System.import('main.js').catch(function(err){ console.error(err); });
    </script>
  </head>

  <body>

    <!--mapiran selektor-->
    <my-app>Učitavanje sadržaja komponente ...</my-app>

  </body>
</html>
```



Angular - IT255

Slika 10.6 Trenutni izgled ekrana [izvor: autor]

EKSTERNA DATOTEKA ŠABLONA

Alternativni pristup izradi prethodnog koda.

U prethodnom izlaganju je pokazano kako je moguće dodati unutar same klase komponente sadržaj šablona direktno. Međutim, to je moguće uraditi i na način koji je obrađivan na predavanjima.

Sledi listing klase komponente:

```
import { Component } from "@angular/core";

@Component({
    selector: 'my-app',
    templateUrl: 'app/app.component.html'
})

export class AppComponent {
    name: string = "Angular - IT255"
}
```

U ovom slučaju, selektorom se učitava i prikazuje sadržaj iz datoteke [app.component.html](#):

```
<!DOCTYPE html>
<html>
<head>
    <title></title>
    <meta charset="utf-8" />
</head>
<body>
    <h1>This is {{name}} Home Page</h1>
</body>
</html>
```

Izlaz je identičan kao i u prethodnom slučaju.

KREIRANJE KOMPONENTE

Potom ćemo koristeći komandu za generisanje komponenti, kreirati novu komponentu.

U nastavku će biti kreirana komponenta pod nazivom [StudentComponent](#). U terminalu razvojnog okruženja ulazimo u folder projekta i na [CLI](#) kucamo:

```
ng generate component studentcomponent
```

Definišemo klasu kreirane komponente na sledeći način:

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

@Component({
    selector: 'app-studentcomponent',
    templateUrl: './studentcomponent.component.html',
    styleUrls: ['./studentcomponent.component.css']
})
//Kreiranje klase
export class StudentComponent {
    firstName: string = "Vladimir";
```

```
lastName: string = "Milićević";
title: string = "Profesor";
qualification: string = "PHD";
}
```

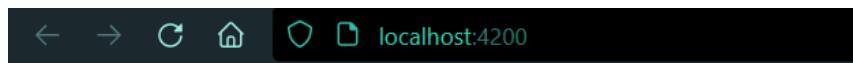
Sada je moguće modifikovati i klasu glavne aktivnosti:

```
import { Component } from "@angular/core";

@Component({
    selector: 'my-App',
    template: `
        <div>
            <h1>{{pageheader}}</h1>
        </div>

        <app-studentcomponent></app-studentcomponent>
    `
})

export class AppComponent {
    pageheader: string = "Detalji o studentu ili profesoru"
}
```



Detalji o studentu ili profesoru

studentcomponent works!

Slika 10.7 Izgled ekrana nakon kreirane komponente [izvor: autor]

ŠABLON KOMPONENTE I DEMO APLIKACIJE

Prikativanje konkretnih podataka šablonom kreirane komponente.

U datoteci `studentcomponent.component.css` dodaćemo malo stilizacije na sledeći način:

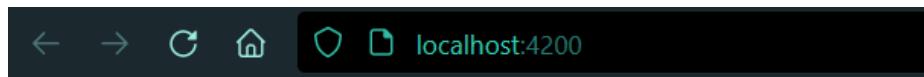
```
table, th, td {
    border: 1px solid;
}
```

Konačno, šablon dobija definiciju kroz implementaciju tabele za prikaz podataka na sledeći način:

```
<table>
  <tr>
    <th>Ime</th>
    <th>Prezime</th>
    <th>Titula</th>
    <th>Obrazovanje</th>

  </tr>
  <tr>
    <td> {{ firstName }} </td>
    <td> {{ lastName }} </td>
    <td> {{ title }} </td>
    <td> {{ qualification }} </td>
  </tr>
</table>
```

Krajnji rezultat aplikacije prikazan je na slici ispod:



Detalji o studentu ili profesoru

Ime	Prezime	Titula	Obrazovanje
Vladimir	Milićević	Profesor	PHD

Slika 10.8 Demo aplikacije [izvor: autor]

✓ Poglavlje 11

Individualna vežba 6

INDIVIDUALNA VEŽBA (TRAJANJE: 90 MINUTA)

Samostalna dopuna zadatka sa pokaznih vežbi.

Otvorite zadatak koji ste radili na pokaznim vežbama, a koji se nalazi u LAMS materijalima u prethodom objektu učenja.

Dodajte nove funkcionalnosti u kreirani *Angular* projekat po sledećim zahtevima:

1. Kreirati formu kroz koju će biti uneti podaci o novom profesoru ili studentu;
2. Kreirati dugme "OBRADI PODATKE"
3. Klikom na dugme uneti podaci se prikazuju u prethodno kreiranoj tabeli.

✓ Poglavlje 12

Domaći zadatak 5

DOMAĆI ZADATAK (PREDVIĐENO VREME 120 MIN)

Samostalna izrada domaćeg zadatka.

Uradite Angular projekat na osnovu sledećih zahteva:

1. Kreirati projekat pod imenom *MetHotels*, koji će na početnoj strani prikazivati listu smeštaja u stilizovanoj tabeli;
2. Neophodno je implementirati formu koja je prikazana u predavanju za dodavanje novih soba.
3. Projekat mora da sadrži komponente (razbiti aplikaciju po uzoru na predavanja i vežbe) i modelsku klasu za sobu

Domaći zadatak postaviti na lični *Github* nalog ili neki drugi drajv, gde naziv *commit*-a treba da bude „*IT255-DZ06*“. Link do zadatka poslati predmetnom asistentu na mail.

▼ Poglavlje 13

Zaključak

ZAKLJUČAK

U lekciju su postavljene osnove za dalji razvoj Angular aplikacija.

U lekciji je bilo detaljno govora o postavljanju ozbiljnih osnova za dalji razvoj složenijih Angular aplikacija. Naučeno je kako se:

1. deli aplikacija na komponente;
2. kreiraju pogledi;
3. definiše model;
4. prikazuje model;
5. dodaje i koristi interakcija.

Sve vreme lekcija je insistirala na kvalitetnim pokaznim primerima za ilustraciju i lakše savladavanje navedene problematike.

LITERATURA

Za pripremu lekcije korišćena je aktuelna pisana i elektronska literatura.

Obavezna literatura:

1. Jennifer Niederst Robbins, Learning Web Design - Fifth Edition, by , Copyright ©; 2018 O'Reilly Media, Inc.
2. Roxane Anquetil, Fundamental Concepts for Web Development: HTML5, CSS3, JavaScript and much more! For complete beginners!, 2019,

Dopunska literatura:

1. Daniel Bell, HTML & CSS: A Step-by-Step Guide for Beginners2, Guzzler Media, 2019.

Elektronska literatura:

4. <https://angular.io/>
5. <https://angular.io/tutorial>
6. <https://www.w3schools.com/angular/>
7. <https://www.tutorialspoint.com/angular4/>
8. <https://nodejs.org/en/>
9. <https://code.visualstudio.com/>



IT255 - VEB SISTEMI 1

TypeScript i Angular

Lekcija 07

PRIRUČNIK ZA STUDENTE

IT255 - VEB SISTEMI 1

Lekcija 07

TYPESCRIPT I ANGULAR

- ✓ TypeScript i Angular
- ✓ Poglavlje 1: TypeScript
- ✓ Poglavlje 2: Tipovi podataka u TypeScript - u
- ✓ Poglavlje 3: Klase
- ✓ Poglavlje 4: Programske alatne
- ✓ Poglavlje 5: Funkcionisanje Angular okvira
- ✓ Poglavlje 6: Analiza – arhitektura podataka
- ✓ Poglavlje 7: Uvod u Angular i TypeScript dodatni materijali
- ✓ Poglavlje 8: Pokazna vežba 7 (TRAJANJE VEŽBE: 45 minuta)
- ✓ Poglavlje 9: Individualna vežba 7
- ✓ Poglavlje 10: Domaći zadatak 7
- ✓ Zaključak

Copyright © 2017 – UNIVERZITET METROPOLITAN, Beograd. Sva prava zadržana. Bez prethodne pismene dozvole od strane Univerziteta METROPOLITAN zabranjena je reprodukcija, transfer, distribucija ili memorisanje nekog dela ili čitavih sadržaja ovog dokumenta., kopiranjem, snimanjem, elektronskim putem, skeniranjem ili na bilo koji drugi način.

Copyright © 2017 BELGRADE METROPOLITAN UNIVERSITY. All rights reserved. No part of this publication may be reproduced, stored in a retrieval system or transmitted in any form or by any means, electronic, mechanical, photocopying, recording, scanning or otherwise, without the prior written permission of Belgrade Metropolitan University.

✓ Uvod

UVOD

Lekcija je podeljena u dva dela: TypeScript diskusija i funkcionisanje Angular okvira.

Lekcija je podeljena u dva dela:

- *TypeScript* diskusija i
- funkcionisanje *Angular* okvira.

U prvom delu će posebno biti govora o migraciji Angular okvira sa standardnog JavaScript jezika ka njegovom proširenju u formi TypeScript notacije. Posebno će biti diskusije o:

- prednostima primene *TypeScript* okvira;
- primeni tipova podataka u *TypeScript* okviru;
- primeni objektno - orijentisanih koncepcata *TypeScript* okvira;
- programskim alatima koji olakšavaju kodiranje u ovom okviru.

Drugi deo lekcije je rezervisan za analizu funkcionisanja Angular okvira sa akcentom na sledećim temama:

- aplikacija kao "top - level" komponenta;
- modeli u Angular aplikacijama;
- kreiranje i primena komponenata u Angular aplikacijama;
- analiza arhitekture podataka u Angular okviru.

Ova lekcija sadrži video materijal. Ukoliko želite da pogledate ovaj video morate da otvorite LAMS lekciju.

▼ Poglavlje 1

TypeScript

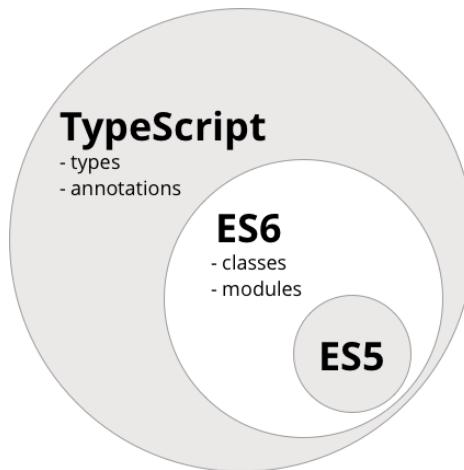
ANGULAR I JEZICI

Angular je baziran na jeziku TypeScript.

Angular je baziran na jeziku **TypeScript** koji predstavlja proširenje dobro poznatog jezika **JavaScript**.

Za nekoga može biti problematično korišćenje novog jezika samo za okvir kao što je Angular, postoje brojni razlozi zašto je dobro koristiti TypeScript umesto čistog JavaScript jezika.

Posebno je važno napomenuti da TypeScript nije potpuno nov jezik, On predstavlja proširenje **JavaScript ES6** koda. Ukoliko se koristi čist JavaScript ES6 kod on je savršeno kompatibilan sa TypeScript kodom. Sledećom slikom prikazan je odnos između navedenih jezika:



Slika 1.1 Odnos između povezanih skupinjenih jezika [izvor: www.typescriptlang.org]

TypeScript predstavlja rezultat zvanične saradnje kompanija **Microsoft** i **Google**. Sa ovako snažnom podrškom sasvim je jasno da će jezik biti dugo vremena aktuelan. Oba partnera su u obavezi da unapređuju veb pa samim tim programeri koji koriste Angular i TypeScript su na velikom dobitku.

Kod ovakvih jezika (*transpilers*) je odlično što relativno mali timovi mogu da kreiraju unapređenja jezika, a da ne moraju od proizvođača da traže da prilagode i unaprede veb pregledače.

Ova lekcija sadrži video materijal. Ukoliko želite da pogledate ovaj video morate da otvorite LAMS lekciju.

TYPESCRIPT PREDNOSTI U ODNOSU NA ČIST JAVASCRIPT

Postoji pet velikih unapređenja u odnosu na čist JavaScript

TypeScript donosi pet velikih unapređenja u odnosu na čist *JavaScript*:

- podrška za tipove podataka;
- podrška za klase;
- primena dekoratora;
- primena import instrukcija;
- primena raznih jezičkih alata, poput destrukcije.

O navedenim unapređenjima će u nastavku biti detaljno razmatrano.

▼ Poglavlje 2

Tipovi podataka u TypeScript - u

UVODENJE TIPOVA PODATAKA U JS KOD

Primena tipova podataka je opcionalna u TypeScript - u.

Glavno unapređenje TypeScript jezika, u odnosu na čist JavaScript, a po čemu je jezik i dobio ime, je uvođenje sistema tipova podataka.

Neki programeri su smatrali prednošću primene JavaScript jezika izbegavanje primene tipova podataka prilikom kodiranja. Međutim, pristup koji podrazumeva primenu tipova u ovakvom skripting jeziku, sigurno zaslužuje šansu ili bar vredno diskutovanja.

Međutim, velika prednost primene koncepta tipa dodataka sastoji se u:

1. pomoći prilikom pisanja koda jer sprečava pojavu grešaka (bug - ova) tokom vremena prevođenja programa;
2. pomoći prilikom čitanja koda jer jasnije pokazuje namere programera.

Takođe, vredno je napomenuti, primena tipova podataka je opcionalna u TypeScript - u. To znači, ako programer želi brzo da kreira određenu količinu koda može da preskoči navođenje tipova podataka. Kasnije, kada kod dostigne određeni nivo zrelosti, programer može da se vrati i da, na odgovarajućim mestima u kodu, doda tipove podataka.

TypeScript osnovni tipovi podataka u potpunosti odgovaraju tipovima koje se "implicitno" koriste u čistom JavaScript kodu:

- *string*;
- *number*;
- *boolean*, itd.

Sve do *JavaScript ES5*, varijable su bile definisane preko ključne reči *var*, na primer:

```
var ime;
```

Nova TypeScript sintaksa predstavlja prirodnu evoluciju *JavaScript ES5*. I dalje se koristi ključna reč *var* ali sada je moguće, opcionalno, obezbediti i tip podataka na sledeći način:

```
var ime : string;
```

Ova lekcija sadrži video materijal. Ukoliko želite da pogledate ovaj video morate da otvorite LAMS lekciju.

POVRATNE VREDNOSTI I ARGUMENTI FUNKCIJA

TypeScript omogućava primenu tipova podataka na argumente i povratne vrednosti funkcija.

Nova TypeScript sintaksa omogućava primenu tipova podataka na argumente i povratne vrednosti funkcija. Dat je sledeći izolovani listing funkcije `greetText()`

```
function greetText(name: string): string {
    return "Hello " + name;
}
```

U konkretnom primeru, definisana je nova funkcija pod nazivom `greetText()` koja uzima jedan argument, tipa `string`, naziva `name`. Studentima, na ovom nivou školovanja, trebalo bi da bude jasno da kod neće biti preveden ukoliko se obavi poziv ove funkcije preko argumenta koji ne pripada tipu podataka `string` i da će prevodilac javiti grešku. Ovo, naravno, predstavlja dobru stvar jer bi, u suprotnom, program radio neispravno.

Takođe, u definiciji funkcije se na još jednom mestu koristi tip podataka. Neophodno je pogledati istaknuti deo na sledećoj slici.

```
function greetText(name: string): string {
    return "Hello " + name;
}
```

Slika 2.1.1 Povratni tip funkcije [izvor: autor]

Na slici, crvenom bojom je istaknut *povratni tip podataka funkcije*, odnosno, tip podataka kojem pripada rezultat izvršavanja kreirane metode. Ova funkcionalnost je, takođe, veoma korisna jer ukoliko, nekim slučajem, funkcija vrati rezultat koji ne odgovara tipu podatak `string`, kompjajler će, ponovo, ukazati na grešku. Posebna korist od navedenog se ogleda u činjenici da će programer, koji koristi u svom radu navedenu funkciju, precizno znati koju će tip vrednosti dobiti kada iskoristi funkciju.

GREŠKE U RADU SA TIPOVIMA PODATAKA

Ukoliko se koristi neodgovarajući tip podataka, kompjajler će javiti grešku.

U prethodnom izlaganju je jasno istaknuto da ukoliko se koristi neodgovarajući tip podataka, u TypeScript kodu, za argumente ili povratne tipove funkcija, kompjajler će javiti grešku. U nastavku, neophodno je obaviti demonstraciju šta se dešava ukoliko se u kodu javi nepodudaranje tipova podataka koje je navedeno i opisano. Posmatra se sledeći izolovani kod metode:

```
function greetText(name: string): string {  
    return 12;  
}
```

Pristupa se prevođenju programa, međutim, vrlo brzo će kompjajler javiti izveštaj sa greškom u obliku prikazanom na sledećoj slici.

```
$ tsc compile-error.ts  
compile-error.ts(2,12): error TS2322: Type 'number' is not assignable to type 'string'.
```

Slika 2.1.2 Nepodudaranje tipova - greška u prevođenju [izvor: autor]

Veoma je važno u potpunosti sagledati šta je se, upravo, ovde desilo. U metodi je pokušan poziv rezultata 12, a jasno je naglašeno u kodu da je povratni tip metode `string`. Kompajler nije dozvolio prevođenje ovakvog koda.

Sa ciljem ispravke navedene greške, moguće je postupiti na sledeći način:

```
function greetText(name: string): number {  
    return 12;  
}
```

Povratni tip metode, kao što je moguće primetiti, promenjen je u numerički tip podataka `number`. Ako se sada pristupi prevođenju programa moguće je primetiti da je sada sve u redu i da je program očišćen od prethodno uočene greške.

Iz prethodnog izlaganja je jasno moguće sagledati u kolikoj meri primena tipova podataka može olakšati rukovanje greškama u `TypeScript` kodu.

✓ 2.1 Ugrađeni tipovi podataka

PRIMENA UGRAĐENIH TIPOVA PODATAKA

TypeScript daje podršku za nekoliko standardnih tipova podataka.

TypeScript daje podršku za nekoliko standardnih tipova podataka koji imaju veliku primenu u Angular aplikacijama. To su:

- `string`;
- `number`;
- `boolean`;
- `Array`
- `enum`;
- `any`;
- `void`.

Sledi kratko predstavljanje ovih veoma često korišćenih tipova podataka.

STRING I NUMERIČKI TIPOVI PODATAKA

TypeScript koristi string i number tipove podataka.

TypeScript koristi `string` i `number` tipove podataka za rad sa nizovima karaktera i brojevima.

Tip `string` omogućava deklarisanje i čuvanje tekstualnih podataka na način prikazan sledećim kodom:

```
var fullName: string = 'Vladimir Milicevic';
```

U TypeScript - u ne postoje posebni tipovi podataka za cele i realne brojeve. Koristi se jedan opšti tip označen ključnom rečju `number`. Sledеćim kodom je prikazan način na koji je moguće deklarisati numeričku promenljivu primenom `TypeScript` jezika:

```
var age: number = 45;
```

LOGIČKI TIP PODATAKA I NIZOVI

TypeScript koristi boolean i Array tipove podataka.

TypeScript koristi `boolean` i `Array` ključne reči za označavanje logičkog tipa podataka i nizova, respektivno.

Kao i u ostalim jezicima, programskim ili skripting, logička vrednost, deklarisana tipom podataka `boolean`, može imati vrednost `true` ili `false`. Na sledeći način je moguće ilustrovati deklarisanje logičke promenljive primenom `TypeScript` jezika:

```
var married: boolean = true;
```

Nizovi u `TypeScript` jeziku se deklarišu primenom tipa podataka `Array`. Međutim, budući da je `Array` kolekcija, takođe je neophodno odrediti i tip podataka kojem pripadaju elementi kolekcije. Imajući u vidu navedeno, tip podataka kojem pripadaju elementi kolekcije moguće je specificirati na jedna od sledeća dva načina:

- `Array<tip>`
- `tip[]`

Navedeno je moguće ilustrovati sledećim listinzima:

```
var jobs: Array<string> = ['IBM', 'Microsoft', 'Google'];
var jobs: string[] = ['Apple', 'Dell', 'HP'];
```

Analogno, za numerički tip podataka moguće je kreirati nizove na sledeći način:

```
var chickens: Array<number> = [1, 2, 3];
var chickens: number[] = [4, 5, 6];
```

NABROJIVI TIP PODATAKA

Nabrojivi tip podataka zasniva se na dodeli naziva numeričkim vrednostima.

Nabrojivi tip podataka ili enumeracija zasniva se na dodeli naziva numeričkim vrednostima. Ovaj tip podataka je označen rezervisanim rečju `enum`. Na primer, ukoliko je cilj da se u programu kreira fiksna lista uloga, koju može da ima neka obaveza, moguće je postupiti na sledeći način:

```
enum Role {Employee, Manager, Admin};  
var role: Role = Role.Employee;
```

Podrazumevana početna vrednost za ovaj tip podataka je 0. Međutim, ovo može biti promenjeno tako što početna vrednost može biti eksplicitno zadatka kao što je to urađeno u sledećem listingu:

```
enum Role {Employee = 3, Manager, Admin};  
var role: Role = Role.Employee;
```

U prethodnom kodu urađeno je sledeće:

1. umesto da uloga Employee započinje vrednošću 0, počinje sa 3;
2. vrednost `enum` liste se uvećava za 1, za svaku narednu stavku, ali počinje sa 3;
3. to znači da su ulogama Manager i Admin pridružene vrednosti 4 i 5, tim redom.

Moguće je modifikovati ove vrednosti i drugačije. Moguće je svakoj ulozi eksplicitno dodeliti drugačiju numeričku vrednost. To je moguće ilustrovati sledećim listingom:

```
enum Role {Employee = 3, Manager = 5, Admin = 7};  
var role: Role = Role.Employee;
```

Prikazivanje naziva, pridruženih vrednostima `enum` tipa, je takođe veoma jednostavno. U tom slučaju je moguće, na najjednostavniji način, kreirati kod priložen sledećim listingom:

```
enum Role {Employee, Manager, Admin};  
console.log('Roles: ', Role[0], ', ', Role[1], 'and', Role[2]);
```

TIPOVI PODATAKA ANY I VOID

Tip podataka `any` označava univerzalni tip podataka. Tip `void` obeležava metodu bez povratne vrednosti.

Tip podataka `any`, u TypeScript jeziku, označava univerzalni i podrazumevani tip podataka. Promenljiva obeležena ovim tipom podataka može da primi bilo koju vrednost. Navedeno je moguće ilustrovati sledećim listingom:

```
var something: any = 'as string';
something = 1;
something = [1, 2, 3];
```

Tip podataka `void` ukazuje da se ne očekuje nijedan od navedenih tipova podataka. U praksi, tipom void je deklarisana metoda bez povratne vrednosti. Navedeno je moguće ilustrovati sledećim listingom:

```
function setName(name: string): void {
    this.fullName = name;
}
```

▼ Poglavlje 3

Klase

PRIMENA KLASA U JAVASCRIPT BAZIRANIM JEZICIMA

Pojavom generacije JavaScript ES6 klase su postale sastavni deo JavaScript jezika.

Ako se uzme u obzir istorijat razvoja **JavaScript** jezika, prva pojava objektno - orijentisanog programiranja bazirala se na objektima baziranim na konceptu prototipa (prototype-based objects). Ovo je bilo karakteristično za generaciju JavaScript ES5 koja nije u kodiranju koristila koncept klasične klase, već se oslanjala na prototipove.

Kako je vreme odmicalo, brojna dobra iskustva su usvojene od strane JavaScript zajednice sa ciljem kompenzovanja nedostatka klasa u kodiranju. Dobri rezimei primene navedenih dobrih praksi mogu biti sagledani izučavanjem sadržaja koji su ponuđeni na sledećim linkovima:

- <https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide>;
- <https://developer.mozilla.org/en-US/docs/Learn/JavaScript/Objects>.

Konačno, **pojavom generacije JavaScript ES6 klase su postale sastavni deo JavaScript jezika.**

U **JavaScript** jeziku, **klasa** se kreira na način prikazan sledećim listingom:

```
class NazivKlase {  
    // sadržaj klase  
}
```

Klasa je izgrađena od sledećih komponenata:

- *osobine (properties);*
- *metode (ili funkcije kako se u JavaScript žargonu obično nazivaju);*
- *konstruktori.*

OSOBINE KLASA

Osobina definiše podatak koji će biti pridružen kreiranom objektu klase.

Osobina definiše podatak koji će biti pridružen kreiranom objektu klase. Na primer, klasa pod nazivom **Person** može imati osobine:

- *first_name;*

- *last_name*;
- *age*;

Svaka od navedenih osobina može, opcionalno, biti deklarisana primenom određenog tipa podataka. Na primer, osobine *first_name* i *last_name* mogu biti deklarisane primenom tipa podataka *string*, a osobina *age* može biti deklarisana primenom tipa podataka *number*. Ako se navedeno pretoči u programski kod, on može biti ilustrovan sledećim listingom:

```
class Person {  
    first_name: string;  
    last_name: string;  
    age: number;  
}
```

METODE KLASA

Metode ili funkcije predstavljaju potprograme koji se izvršavaju u kontekstu objekta klase.

Metode ili funkcije predstavljaju potprograme koji se izvršavaju u kontekstu objekta klase. Da bi metoda bila izvršena, neophodno je prvo kreirati objekat date klase koji će, potom, pozvati posmatranu metodu i omogućiti njeno izvršavanje.

Ako se posmatra klasa, kreirana u prethodnoj sekciji, i ukoliko je cilj kreiranje metode koja će omogućiti prikazivanje poruke za objekat klase *Person*, moguće je kod klase proširiti na sledeći način:

```
class Person {  
    first_name: string;  
    last_name: string;  
    age: number;  
  
    // metoda  
    greet() {  
        console.log("Hello", this.first_name);  
    }  
}
```

Iz priloženog listinga moguće je sagledati nekoliko stvari:

- pristup individualnoj osobini *first_name*, za objekat koji poziva metodu, omogućen je primenom ključne reči *this*;
- nije deklarisan povratni tip metode.

Kada metoda ne poseduje jasno naveden povratni tip i vrednost, podrazumeva se da vraća bilo šta (određeno tipom podataka *any*). Međutim, u ovom konkretnom slučaju, metoda ne poseduje naredbu *return* pa je sasvim jasno da će njen povratni tip odgovarati tipu *void*.

Ukoliko je u programskom kodu neophodno obaviti poziv kreirane metode, to je moguće učiniti na sledeći način:

```
// deklarisanje promenljive tipa Person
var p: Person;

// poziv konstruktora - kreiranje objekta tipa Person
p = new Person();

// dodela vrednosti osobini klase
p.first_name = 'Vladimir';

// poziv metode kreiranim objektom klase
p.greet();
```

Kreiranje objekta klase je moguće obaviti i u jednoj liniji koda, na sledeći način:

```
var p: Person = new Person();
```

METODE KOJE VRAĆAJU VREDNOST

Ukoliko metoda poseduje naredbu return, mora da vrati vrednost odgovarajućeg tipa podataka.

Ukoliko metoda poseduje naredbu **return**, mora da vrati vrednost odgovarajućeg tipa podataka. Tip podataka se jasno navodi prilikom deklarisanja metode.

Da bi navedeno bilo ilustrovano, neophodno je kreirati novu metodu klase **Person** koja će vratiti broj koji odgovara starosti osobe, koju predstavlja konkretan objekat ove klase, nakon perioda određenog zadatim brojem godina. Upravo navedeno je realizovano kodom iz sledećeg listinga:

```
class Person {
    first_name: string;
    last_name: string;
    age: number;

    greet() {
        console.log("Hello", this.first_name);
    }

    ageInYears(years: number): number {
        return this.age + years;
    }
}
```

Poziv ove metode je moguće, na veoma jednostavan način, ilustrovati sledećim listingom:

```
// kreiranje objekta klase Person
var p: Person = new Person();

// podešavanje početne vrednosti za osobinu age
```

```
p.age = 6;  
  
// starost za 12 godina  
p.ageInYears(12);  
  
// rezultat -> 18
```

KONSTRUKTORI KLASA

Konstruktor je posebna metoda koju je neophodno izvršiti da bi bila kreirana instanca klase.

Konstruktor je posebna metoda koju je neophodno izvršiti da bi bila kreirana instanca klase. Obično, konstruktorm se obavljaju inicijalna podešavanja objekata.

Po podrazumevanim podešavanjima, metoda konstruktora mora da nosi naziv **constructor**. Ove metode, optionalno, mogu da poseduju parametre ali ne mogu da vraćaju vrednosti budući da se pozivaju kada se kreira objekat klase.

Sa ciljem kreiranja objekta klase neophodno je pozvati konstruktor metodu primenom naziva klase, kao na sledeći način:

```
new ClassName()
```

Ukoliko klasa ne poseduje eksplisitno deklarisan konstruktor, on će automatski biti **kreiran**. To praktično znači da je kod:

```
class Vehicle {  
}  
  
...  
var v = new Vehicle();  
...
```

ekvivalentan kodu:

```
class Vehicle {  
    constructor() {  
    }  
}  
  
...  
var v = new Vehicle();  
...
```

U **TypeScript jeziku može da postoji samo jedan konstruktor po klasi.** Ovo predstavlja odstupanje od standardnog JavaScript jezika gde je moguće preklapanje konstruktora sve dok imaju različite argumente.

KONSTRUKTORI SA PARAMETRIMA

Konstruktor može da poseduje i parametre koji se koriste prilikom kreiranja objekata.

Konstruktor može biti podrazumevani, kao u prethodnom listingu, a može da poseduje i parametre koji se koriste prilikom kreiranja objekata. Na primer, klasa `Person` poseduje konstruktor kojim je moguće inicijalizovati podatke:

```
class Person {  
    first_name: string;  
    last_name: string;  
    age: number;  
  
    constructor(first_name: string, last_name: string, age: number) {  
        this.first_name = first_name;  
        this.last_name = last_name;  
        this.age = age;  
    }  
  
    greet() {  
        console.log("Hello", this.first_name);  
    }  
  
    ageInYears(years: number): number {  
        return this.age + years;  
    }  
}
```

Na ovaj način je kreiranje objekata dodatno pojednostavljeno, u odnosu na prethodni slučaj i može biti realizovano na sledeći način:

```
var p: Person = new Person('Vladimir', 'Milicevic', 45);  
p.greet();
```

Na ovaj način su osobine objekta automatski podešene prilikom njegovog kreiranja.

NASLEDIVANJE

Važan aspekt objektno - orijentisanog programiranja predstavlja nasleđivanje.

Kao što je dobro poznato, veoma važan aspekt objektno - orijentisanog programiranja predstavlja nadleđivanje. Nasleđivanje predstavlja mehanizam preko kojeg neka klasa (potomak) prima ponašanje druge (roditeljske) klase. Na ovaj način je moguće zameniti, modifikovati ili proširiti ponašanje nove klase.

Jezik TypeScript u potpunosti podržava koncept nasleđivanja klasa koji je ugrađen u jezgro jezika. Nasleđivanje se postiže primenom ključne reči *extends*.

Za ilustrovanje navedenog, kreira se klasa pod nazivom *Report*.

```
class Report {  
    data: Array<string>;  
  
    constructor(data: Array<string>) {  
        this.data = data;  
    }  
  
    run() {  
        this.data.forEach(function(line) { console.log(line); });  
    }  
}
```

Iz listinga je moguće primetiti da klasa poseduje osobinu *data* koja odgovara tipu podataka *Array<string>*. Pozivom metode *run()* izvršava se petlja nad svim elementima niza i vrši se njihovo štampanje u konzoli.

Kreiranje objekta i poziv metode *run()* moguće je postići sledećim listingom:

```
var r: Report = new Report(['First line', 'Second line']);  
r.run();
```

Rezultat izvršavanja ovog koda je sledeći:

```
First line  
Second line
```

Sada je moguće tražiti postojanje još jednog izveštaja koji koristi zaglavljive i neke podatke, ali je i dalje bitno korišćenje prikazivanje podataka definisano prethodnom klasom. Nasleđivanje ovakvog ponašanja klase *Report* moguće je postići kreiranjem nove klase, na sledeći način:

```
class TabbedReport extends Report {  
    headers: Array<string>;  
  
    constructor(headers: string[], values: string[]) {  
        super(values)  
        this.headers = headers;  
    }  
  
    run() {  
        console.log(this.headers);  
        super.run();  
    }  
}
```

Primena klase naslednice je opisana sledećim listingom:

```
var headers: string[] = ['Name'];
var data: string[] = ['Alice Green', 'Paul Pfifer', 'Louis Blakeship'];
var r: TabbedReport = new TabbedReport(headers, data)
r.run();
```

ANGULAR KLASE (VIDEO MATERIJAL)

EP 2.10 - Angular / ES6 & TypeScript / Classes & Interface (trajanje: 17:41)

Ova lekcija sadrži video materijal. Ukoliko želite da pogledate ovaj video morate da otvorite LAMS lekciju.

▼ Poglavlje 4

Programski alati

VRSTE TYPESCRIPT PROGRAMSKIH ALATA

TypeScript obezbeđuje brojne sintaksne alate za olakšavanje programiranja.

JavaScript ES6 i proširenje TypeScript obezbeđuju brojne sintaksne alate za olakšavanje programiranja. Dva najvažnija alata su:

1. funkcionalna sintaksa "debele linije" (*fat arrow function syntax*);
2. šabloni stringova.

U sledećem izlaganju sledi objašnjenje primene navedenih alata.

FAT ARROW NOTACIJA

Notacija "Fat Arrow" omogućava pojednostavljeni pisanje koda funkcija.

Notacija "Fat Arrow" omogućava pojednostavljeni pisanje koda funkcija. Ovo pojednostavljenje je moguće uporediti sa primenom "lambda" izraza u Java jeziku.

Od generacije JavaScript ES5, kada god je neophodno koristiti funkciju kao argument, neophodno je upotrebiti ključnu reč *function* zajedno sa zagradama na sledeći način:

```
//JavaScript ES5 primer
var data = ['Vladimir Milicevic', 'Marko Rajevic', 'Nikola Dimitrijevic'];
data.forEach(function(line) { console.log(line); });
```

Primenom "Fat Arrow" notacije, ovaj kod je moguće lakše zapisati na sledeći način:

```
// Typescript primer
1 var data: string[] = ['Vladimir Milicevic', 'Marko Rajevic', 'Nikola
Dimitrijevic'];
2 data.forEach( line => console.log(line) );
```

Veoma važna stavka => sintakse da koristi isti *this* objekat kao i kod koji je okružuje. Ovo je veoma važno i razlikuje se od uobičajenog rada sa funkcijama u JavaScript - u. Kada se piše funkcija u JavaScript - u, njoj je dat vlastiti *this*. Ponekad, u JavaScript - u moguće je sresti sledeći kod,

```
var nate = {
  name: "Vlada",
  guitars: ["Gibson", "Martin", "Taylor"],
  printGuitars: function() {
    var self = this;
    this.guitars.forEach(function(g) {
      // this.name je nedefinisano pa se koristi self.name
      console.log(self.name + " plays a " + g);
    });
  }
};
```

Pošto sintaksa "fat arrow" deli *this* sa okružujućim kodom, moguće je napisati:

```
var nate = {
  name: "Vlada",
  guitars: ["Gibson", "Martin", "Taylor"],
  printGuitars: function() {
    this.guitars.forEach( (g) => {
      console.log(this.name + " plays a " + g);
    });
  }
};
```

ŠABLONI STRINGOVA

JavaScript ES6 uvodi nove šablone za rukovanje stringovima.

JavaScript ES6 uvodi nove šablone za rukovanje stringovima. Dva najznačajnija alata ovog tipa su:

- *varijable unutar stringova* (bez potrebe za povezivanjem putem operatora +);
- *višelinjski (multi-line) stringovi*.

Varijabla unutar stringa ili interpolacija stringova može da se ugradi na sledeći način:

```
var firstName = "Vladimir";
var lastName = "Milićević";

// interpolacija stringa
var greeting = `Hello ${firstName} ${lastName}`;

console.log(greeting);
```

Višelinjski stringovi su sledeći odličan alat za manipulaciju stringovima kada ih je potrebno dodati u kod. Evo primera:

```
var template = `
<div>
<h1>Hello</h1>
```

```
<p>This is a great website</p>
</div>
`  
  
// uraditi nešto sa stringom `template`
```

OSTALI TYPESCRIPT ALATI

Postoje brojni i korisni TypeScript / ES6 alati

U ovom delu lekcije je neophodno je navesti još neke korisne *TypeScript / ES6* alate o kojima će biti diskusije, detaljno, kako analiza i razumevanje Angular problematike bude odmicalo dalje:

- **interfejsi;**
- **generički tipovi podataka;**
- **importovanje i eksportovanje modula;**
- **dekoratori;**
- **destrukcija.**

▼ Poglavlje 5

Funkcionisanje Angular okvira

UVODNO RAZMATRANJE

Akcentat će biti na Angular konceptima visokog nivoa i njihovom funkcionisanju kao celine.

U ovom delu lekcije, akcentat će biti na Angular konceptima visokog nivoa (high - level). Za analizu i diskusiju biće neophodno vratiti se jedna korak unazad sa ciljem sagledavanja mehanizama funkcionisanja navedenih delova u formi celine. Za svaki od koncepata biće dat detaljan pregled i objašnjenje njegove funkcionalnosti.

Prvi značajan koncept predstavlja komponenta. Kao što je već rečeno, primenom komponenata Angular uči veb pregledač da koristi novi HTML tag. Ako je neko od studenata pokušao da samostalno uči AngularJS 1.x, savremeni koncept komponenata odgovara starom konceptu direktiva. Ono što je bitno, za ovaj deo razmatranja, Angular komponente poseduju značajne prednosti u odnosu na starije direktive o čemu će, takođe, biti detaljno govora kada u fokus dođu ovi koncepti.

U daljem izlaganju, biće započeta diskusija od samog vrha - koncepta poznatog pod nazivom aplikacija.

▼ 5.1 Aplikacija

STABLO KOMPONENTA

Angular aplikacija ne predstavlja ništa drugo do stablo komponenata.

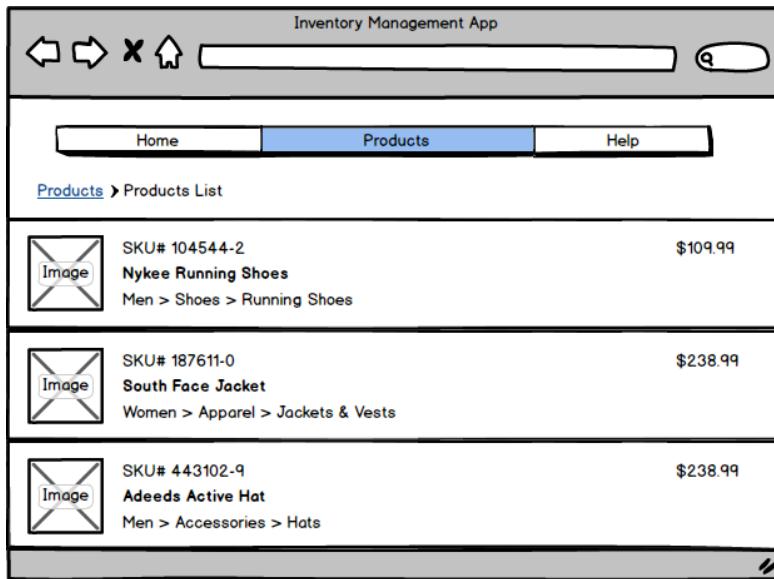
Angular aplikacija ne predstavlja ništa drugo do stabla komponenata. U korenu stabla, kao komponenta najvišeg nivoa, javlja se upravo aplikacija. Upravo njom je određen sadržaj koji će veb pregledač prikazati kada pokreće aplikaciju.

Odlična stvar, vezana za komponente, jeste neograničena mogućnost njihovog povezivanja. To znači da je kombinovanjem sitnijih komponenata moguće kreirati složene komponente. Jednostavno rečeno, aplikacija je komponenta koja kreira druge komponente.

Budući da su sve komponente strukturirane u stablu roditelj / potomak, kada se neka komponenta kreira, ona rekursivno kreira i vlastite potomke.

Da bi diskusija tekla jednostavnije i razumljivije, moguće je osloniti se na adekvatan primer i neka to bude aplikacija za upravljanje zalikama.

Sledećom slikom je moguće opisati idejni izgled same aplikacije.



Slika 5.1.1 Idejni izgled prateće Angular aplikacije [izvor: autor]

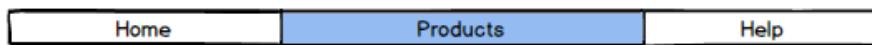
Na osnovu priložene slike, ideja može biti da se za potrebe realizacija aplikacije razviju tri posebne komponente:

- *navigaciona komponenta;*
- *komponenta zaprikaz detalja;*
- *komponenta za kreiranje i prikaz liste proizvoda.*

NAVIGACIONA KOMPONENTA

Zadatak navigacione komponente jeste kreiranje sekcije za navigaciju.

Zadatak navigacione komponente jeste kreiranje sekcije za navigaciju. Da bi bilo jasnije, iz prethodne slike je potrebno izolovati baš ovaj deo i to je prikazano sledećom slikom.



Slika 5.1.2 Navigaciona komponenta aplikacije [izvor: autor]

KOMPONENTA ZA PRIKAZ DETALJA

Komponenta za izbor detalja smeštena je neposredno ispod sekcije za navigaciju.

Komponenta za prikaz detalja, ako se pogleda slika 1, smeštena je neposredno ispod sekcije za navigaciju. Ova komponenta kreira prikaz hijerarhijske reprezentacije mesta u aplikaciji

u kojem se korisnik trenutno nalazi. Ovakve komponente se često u literaturi nazivaju simbolično "mrvicama hleba" (*Breadcrumbs*) pa će ta praksa biti zadržana i u aktuelnom primeru.

Sledećom slikom je izolovana komponenta za prikaz detalja.

Products ➔ Products List

Slika 5.1.3 Komponenta za prikaz detalja [izvor: autor]

KOMPONENTA ZA KREIRANJE I PRIKAZ LISTE PROIZVODA

Komponenta kojom je omogućeno reprezentovanje kolekcije dostupnih proizvoda.

Komponenta za kreiranje i prikaz liste proizvoda predstavlja komponentu kojom je omogućeno reprezentovanje kolekcije dostupnih proizvoda. Sledеćom slikom je ilustrovana navedena komponenta.

	SKU# 104544-2 Nykee Running Shoes Men > Shoes > Running Shoes	\$109.99
	SKU# 187611-0 South Face Jacket Women > Apparel > Jackets & Vests	\$238.99
	SKU# 443102-9 Adeeds Active Hat Men > Accessories > Hats	\$238.99

Slika 5.1.4 Komponenta za kreiranje i prikaz liste proizvoda [izvor: autor]

Razlaganjem posmatrane komponente na sledeći nivo jednostavnijih komponenata, moguće je istaći da će lista proizvoda biti dobijena kombinovanjem jednog ili više redova koji odgovaraju pojedinačnim proizvodima.

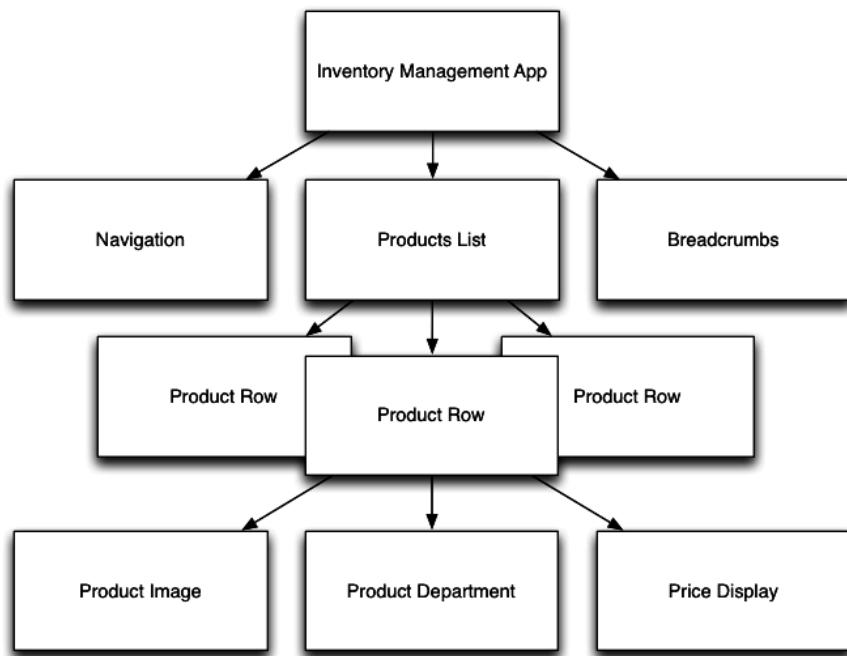
Dekompozicija može ići i dalje pa je moguće reći da svaki red, koji odgovara pojedinačnim proizvodima, čine:

- *slika proizvoda* - posebna komponenta biće angažovana za vizualni prikaz proizvoda iz liste;
- *odeljenje proizvoda* - komponenta će da kreira stablo odeljenja, na primer: *Men > Shoes > Running Shoes*;
- *prikaz cene* - posebna komponenta biće angažovana za prikaz cene proizvoda.

OBJEDINJENI PRIKAZ KOMPONENTA APLIKACIJE

Povezivanjem navedenih komponenata aplikacije u celinu, dobija se stablo aplikacije.

Povezivanjem navedenih komponenata aplikacije u celinu, dobija se stablo aplikacije. U konkretnom slučaju, stablo aplikacije može biti ilustrovano sledećom slikom:



Slika 5.1.5 Stablo Angular aplikacije [izvor: autor]

Na vrhu se nalazi **Inventory Management App** što upravo predstavlja aplikaciju koja će biti kreirana. Ispod se nalaze komponente: **Navigation**, **Breadcrumb** i **Products List**. Komponenta **Products List** sadrži **Product Row** za svaki od proizvoda. Svaki red **Product Row** koristi tri komponente za prikazivanje: slike, odeljenja i cene proizvoda.

Upravo na ovim postavkama biće kreirana nova Angular aplikacija, kao na sledećoj slici.

The screenshot displays a user interface for an inventory application. At the top left is a logo for 'MET Store'. To its right, the title 'Angular 2 Inventory App' is centered. Below this, there are three product cards arranged vertically. The first card shows a black running shoe with the text 'Black Running Shoes', 'SKU #MYSHOES', and a link 'Men > Shoes > Running Shoes'. The second card shows a blue jacket with the text 'Blue Jacket', 'SKU #NEATOJACKET', and a link 'Women > Apparel > Jackets & Vests'. The third card shows a black baseball cap with the text 'A Nice Black Hat', 'SKU #NICEHAT', and a link 'Men > Accessories > Hats'. Each card also includes a price: \$109.99 for the shoe, \$238.99 for the jacket, and \$29.99 for the hat.

Slika 5.1.6 Budući izgled apliakcije [izvor: autor]

VAŽNE SMERNICE

Slede važne napomene u vezi sa izradom aplikacije.

Slede važne napomene u vezi sa izradom aplikacije:

- koristi se *Angular CLI*;
- kreiranje *aplikacije*: *ng new NazivAplikacije*;
- kreiranje komponenata: *ng generate component NazivKomponente*;
- pokretanje *aplikacije*: *ng serve*;

Posebnu pažnju bi trebalo obratiti na:

- Kako se *aplikacija* razbija na komponente?
- Kako se komponente višestruko koriste primenom ulaza (inputs)?
- Kako se rukuje korisničkim interakcijama?

✓ 5.2 Modeli u aplikaciji

MODEL PROIZVODA

Angular je veoma fleksibilan okvir i podržava različite tipove modela i arhitekture podataka.

Jedna od ključnih stvari koje je neophodno razumeti u vezi sa okvirom *Angular* da on ne propisuje primenu neke specijalne biblioteke modela. *Angular je veoma fleksibilan okvir i*

podržava različite tipove modela i arhitekture podataka. To praktično znači, da je programeru ostavljeno na volju da odluči kako će da implementira navedene koncepte.

Što se tiče arhitekture podataka u narednim izlaganjima će se ukazati potreba da se veoma detaljno diskutuje na tu temu. Za sada, akcenat će biti na implementaciji modela u formi čistih [JavaScript](#) objekata.

U podfolderu, kreiranog projekta, [src/app/](#) kreira se datoteka [product.model.ts](#) koja će čuvati kod modelske klase Product. Sledеćim listingom je priložen kod ove klase:

```
/**  
 * Obezbeđuje objekte tipa `Product`  
 */  
export class Product {  
    constructor(  
        public sku: string,  
        public name: string,  
        public imageUrl: string,  
        public department: string[],  
        public price: number) {}  
}
```

Iz priloženog listinga je moguće primetiti da klasa [Product](#) sadrži konstruktor sa 5 argumenata. Klasa takođe ne sadrži nikakve Angular zavisnosti, ona je osmišljena kao modelska klasa koja će biti dalje korišćena u aplikaciji.

▼ 5.3 Komponente u Angular aplikaciji

KOMPONENTE

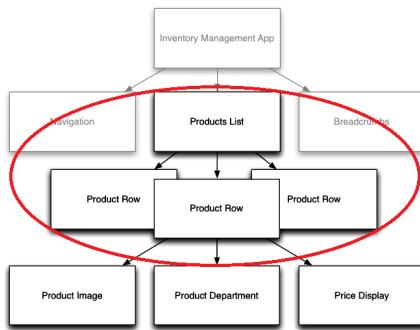
Komponenta je osnovni gradivni blok Angular aplikacije.

Kao što je već napomenuto, [komponenta](#) je osnovni gradivni blok [Angular](#) aplikacije. Aplikacija, sama po sebi, je komponenta najvišeg nivoa. U daljem razvoju aplikacija se razbija na manje komponente - [potomke](#).

Svaka komponenta je izgrađena iz tri dela:

- [dekorator](#) komponente;
- [pogled](#);
- [kontroler](#).

Za ilustraciju ključnih koncepata u Angularu, nepogodno je dobro razumeti komponente. Izlaganje započinje sa najvišeg nivoa aplikacije [InventoryManagementApp](#), a zatim će fokus biti na komponenti [ProductListComponent](#) i komponentama potomcima (sledeća slika).



Slika 5.2.1 ProductListComponent i komponente potomci [izvor: autor]

Za početak, prilaže se listing centralne komponente *AppComponent* koja za sada izgleda ovako:

```

import { Component } from '@angular/core';

@Component({
  selector: 'inventory-app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent {
  //ovde dolazi logika
}
  
```

Anotacija *@Component* se naziva dekoratorom i preko nje se dodaju meta-podaci u klasu. Dekorator specificira:

- selektor, koji govori Angularu koji element da traži;
- šablon (*template*, *templateUrl*), kojim se definiše pogled (HTML izlaz).

Kontroler komponente je definisan klasom, *AppComponent* u konkretnom slučaju.

SELEKTOR I ŠABLON

Selektor ukazuje se na komponentu koja će biti prepoznata kada se bude koristila u šablonu.

Primenom ključa *selector* ukazuje se na komponentu koja će biti prepoznata kada se bude koristila u šablonu. Iz listinga, prethodno prikazane klase *AppComponent*, primećuje se da je selektorem definisan tag *inventory-app-root* kojim je komponenta predstavljena u odgovarajućem pogledu. Postoje dva načina da se ovaj tag angažuje u HTML kodu:

```
<inventory-app-root></inventory-app-root>
```

Ili na drugi način:

```
<div inventory-app-root></div>
```

Šablon je vidljivi deo komponente. Primenom ključa `template`, unutar dekoratora `@Component`, deklariše se HTML šablon kojeg će komponenta koristiti:

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

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent {
  //ovde dolazi logika
}
```

Slika 5.2.2 Definisanje šablona [izvor: autor]

DODAVANJE PROIZVODA

Da bi proizvod bio dodat, neophodno je prvo importovati njegovu klasu u centralnu komponentu.

Aplikacija nije preterano zanimljiva ako ne prikazuje proizvode, a sada je to slučaj. Biće iskorišćena centralna komponenta i u njoj će biti dodat jedan proizvod.

Da bi proizvod bio dodat, neophodno je prvo importovati njegovu klasu u centralnu komponentu, a onda u okviru njenog konstruktora obaviti dodavanje konkretnog objekta, primenom definisanog konstruktora sa 5 argumenata. Sledi listing:

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

//importovanje klase Product
import { Product } from './product.model';

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

  constructor() {
    //kreiranje novog proizvoda
    let newProduct = new Product(
      'NICEHAT', //sku
      'A Nice Black Hat', //name
      '/assets/images/products/black-hat.jpg', //imageUrl
      ['Men', 'Accessories', 'Hats'], //department
    )
  }
}
```

```
    29.99); //price
    this.product = newProduct;
}
}
```

PRIKAZIVANJE PROIZVODA PREKO POVEZIVANJA POGLEDA

Koristi se varijabla sa ciljem kreiranja pogleda kojim će biti moguće prikazati kreirani proizvod.

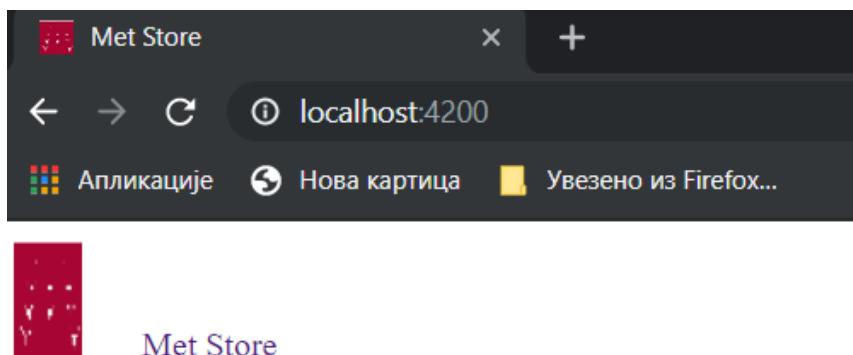
U prethodnom izlaganju je obavljeno dodavanje proizvoda u centralnu komponentu *AppComponent*. Sada je moguće ovu varijablu iskoristiti sa ciljem kreiranja pogleda kojim će biti moguće prikazati kreirani proizvod.

Neophodno je otvoriti datoteku *app.component.html* i u nju dodati sledeći kod:

```
<div class="inventory-app">
  <h1>{{ product.name }}</h1>
  <span>{{ product.sku }}</span>
</div>
```

Ukoliko je sve u redu, prevođenjem i pokretanjem ove aplikacije biće prikazan pogled koji prikazuje osobine: *sku* i *name*, kreiranog objekta proizvoda.

Trenutni izgled aplikacije je moguće prikazati sledećom slikom:



Angular Inventory App

A Nice Black Hat

NICEHAT

Slika 5.2.3 Prikazivanje proizvoda preko povezivanja pogleda [izvor: autor]

DODAVANJE LISTE PROIZVODA

Hajde sada da jedan proizvod bude zamenjen nizom proizvoda.

Kod koji je do sad kreiran, odličan je osnov za dalji rad i unapređivanje. Hajde sada da jedan proizvod bude zamenjen nizom proizvoda. Prvi korak je zamena osobine `product: Product`, osobinom `products: Product[]`. Dalje, u konstruktoru centralne komponente biće kreirano još par proizvoda koji će potom svi zajedno biti dodati u kreirani niz `products`.

Navedene izmene je moguće ilustrovati sledećim listingom:

```
export class AppComponent {  
    //pojedinačni proizvod je zamenjen nizom proizvoda  
    products: Product[];  
  
    constructor() {  
  
        //kreiranje i dodavanje proizvoda u listu  
        this.products = [  
            new Product(  
                'MYSHOES',  
                'Black Running Shoes',  
                '/assets/images/products/black-shoes.jpg',  
                ['Men', 'Shoes', 'Running Shoes'],  
                109.99),  
            new Product(  
                'JACKET',  
                'Blue Jacket',  
                '/assets/images/products/blue-jacket.jpg',  
                ['Women', 'Apparel', 'Jackets & Vests'],  
                238.99),  
            new Product(  
                'NICEHAT',  
                'A Nice Black Hat',  
                '/assets/images/products/black-hat.jpg',  
                ['Men', 'Accessories', 'Hats'],  
                29.99)  
        ];  
    }  
}
```

IZBOR PROIZVODA

Cilj je dodavanje podrške interakciji korisnika sa aplikacijom.

Moguće je dalje igrati se idejama i vršiti modifikacije na aplikaciji. Cilj je dodavanje podrške interakciji korisnika sa aplikacijom. Na primer, korisnik može da: izabere konkretan proizvod sa ciljem dobijanja dodatnih informacija u vezi sa proizvodom, doda proizvod u korpu i još mnogo toga.

Kada je istaknut nov zahteve aplikacije, moguće je dodati nove funkcionalnosti u centralnu komponentu `AppComponent` za rukovanje događajem koji je rezultat korisnikovog izbora iz liste. Prethodni listing biće proširen metodom `productWasSelected()`:

```
productWasSelected(product: Product): void {  
  console.log('Product clicked: ', product);  
}
```

Iz listinga je moguće primetiti da funkcija uzima jedan argument tipa `Product` koji će biti prikazan u logu veb pregledača. Još malo pa će funkcija moći u punoj mjeri da se koristi.

PRIKAZIVANJE PROIZVODA PRIMENOM TAGA PRODUCTS-LIST

Za uvedenu novu funkcionalnost više nije dovoljna samo top - level komponenta.

Za uvedenu novu funkcionalnost - prikazivanje liste dostupnih proizvoda, više nije dovoljna samo top - level komponenta pod nazivom `AppComponent`. Dakle, neophodno je kreiranje nove komponente `ProductsList` koja će biti zadužena da omogući prikazivanje liste proizvoda. Neka je selektorom nove komponente definisan tag `<products-list>` koji će omogućiti podudaranje sa implementacijom liste proizvoda.

Pre ulaska u implementacione detalje nove komponente, sledećim listingom prikazana je primena nove komponente u šablonu:

```
<div class="inventory-app">  
  
  <products-list  
    [productList]="products"  
    (onProductSelected)="productWasSelected($event)">  
  
  </products-list>  
</div>
```

Iz listinga je moguće sagledati prisustvo nove sintakse o kojoj sledi diskusija u narednom izlaganju.

ULAZ / IZLAZ

Primena ključnih Angular koncepata: ulaza (input) i izlaza (output).

Iz prethodnog listinga nazire se **primena ključnih Angular koncepata: ulaza (input) i izlaza (output)**. Problematiku je bolje i lakše sagledati vizuelno sa sledeće slike:

```
<products-list  
  [productList]="products"  
  (onProductSelected)="productWasSelected($event)"> <!-- input -->  
</products-list>
```

Slika 5.2.4 Primena kocepata ulaza i izlaza [izvor: autor]

Ako se osmotri priloženi kod sa slike moguće je primetiti dve stvari:

- uglastim zagradama [] prosleđuje se ulaz;
- malim zagradama () je istaknuta sintaksa u kojoj se rukuje izlazom.

Podaci se kreću ka komponenti preko povezivanja ulaza , a događaji izlaze iz komponente preko povezivanja izlaza. Podešavanje povezivanja ulaza / izlaza može se rezonovati kao definisanje javnog (*public*) API - ja za posmatranu komponentu.

O radu sa ulaznim vrednostima je već bilo diskusije u prethodnoj lekciji. Sada je neophodno posvetiti više pažnje rukovanju izlaznim vrednostima.

U Angularu, slanje podataka iz komponente moguće je obaviti na sledeći način:

```
<products-list  
  ...  
  (onProductSelected)="productWasSelected($event)">
```

Na ovaj način se vrši osluškivanje *onProductSelected* izlaza iz komponente *ProductsList* . Kod funkcioniše na sledeći način:

- (*onProductSelected*), na levoj strani izraza, je naziv izlaza koji se osluškuje;
- "*productWasSelected*", na desnoj strani izraza, je naziv metode koja se poziva kada je neka nova vrednost prosleđena izlazu;
- \$event* označava specijalnu promenljivu koja reprezentuje veličinu koja se emituje na izlazu.

Za sada nije bilo detaljno govora o definisanju ulaza i izlaza za vlastite komponente ali će to ubrzo doći na red čim se pristupi kreiranju *ProductsList* komponente. Za sada naučeno je slanje podataka ka potomku i prijem podataka iz komponente potomka.

ZAOKRUŽENA DEFINICIJA CENTRALNE KOMPONENTE

Prilažu se puni listinzi klase i pogleda komponente AppComponent.

Sada je moguće zaokružiti definiciju centralne komponente aplikacije. Sledi listing klase *AppComponent*:

```
import { Component } from '@angular/core';  
import { Product } from './product.model';  
  
@Component({  
  selector: 'inventory-app-root',
```

```
templateUrl: './app.component.html',
styleUrls: ['./app.component.css']
})
export class AppComponent {
//pojedinačni proizvod je zamenjen nizom proizvoda
products: Product[];

constructor() {

    //kreiranje i dodavanje proizvoda u listu
    this.products = [
        new Product(
            'MYSHOES',
            'Black Running Shoes',
            '/assets/images/products/black-shoes.jpg',
            ['Men', 'Shoes', 'Running Shoes'],
            109.99),
        new Product(
            'JACKET',
            'Blue Jacket',
            '/assets/images/products/blue-jacket.jpg',
            ['Women', 'Apparel', 'Jackets & Vests'],
            238.99),
        new Product(
            'NICEHAT',
            'A Nice Black Hat',
            '/assets/images/products/black-hat.jpg',
            ['Men', 'Accessories', 'Hats'],
            29.99)
    ];
}

productWasSelected(product: Product): void {
    console.log('Product clicked: ', product);
}
}
```

```
<div class="inventory-app">
    <products-list
        [productList]="products"
        (onProductSelected)="productWasSelected($event)">
    </products-list>
</div>
```

Konačno, sledi i listing datoteke [app.compat.html](#):

```
<div class="inventory-app">
```

```
<products-list  
  [productList]="products"  
  (onProductSelected)="productWasSelected($event)">  
</products-list>  
</div>
```

KOMPONENTA PRODUCTSLISTCOMPONENT

Kreiranje komponente `ProductsListComponent` za kreiranje i prikazivanje redova liste proizvoda.

Sada kada aplikacija poseduje komponentu najvišeg nivoa, moguće je pristupiti kreiranju komponente `ProductsListComponent` čiji je zadatak kreiranje i prikazivanje redova liste proizvoda.

Takođe, cilj je omogućavanje korisniku da izabere jedan proizvod i da se čuva trag u vezi sa izabranim proizvodom. Komponenta `ProductsListComponent` je idealno mesto za implementiranje navedenih funkcionalnosti zato što će "znati" sve `Product` objekte istovremeno.

Komponenta će biti kreirana prateći sledeća tri koraka:

1. podešavanje `ProductsListComponent @Component` opcija;
2. pisanje `ProductsListComponent` kontrolerske klase;
3. pisanje `ProductsListComponent` šablona pogleda.

PODEŠAVANJE DEKORATORA PRODUCTSLISTCOMPONENT KOMPONENTE

Definicija i analiza konfiguracije dekoratora Component komponente `ProductsListComponent`.

Data je konfiguracija dekoratora `@Component` komponente `ProductsListComponent`.

```
import {  
  Component,  
  OnInit,  
  EventEmitter,  
  Input,  
  Output }  
from '@angular/core';  
  
import { Product } from '../product.model';  
  
@Component({
```

```
    selector: 'products-list',
    templateUrl: './product-list.component.html',
    styleUrls: ['./product-list.component.css']
})
export class ProductListComponent implements OnInit {

  @Input() productList: Product[];

  @Output() onProductSelected: EventEmitter<Product>;

  ...
}
```

Iz listinga je moguće jasno uočiti sledeće stvari:

- selektorom je određen tak komponente `<products-list>`;
- definisane su dve osobine: `productList` i `onProductSelected`;
- `productList` poseduje anotaciju `@Input()` koja ukazuje da se radi o ulaznoj vrednosti;
- `onProductSelected` poseduje anotaciju `@Output()` koja ukazuje da se radi o izlazu.

ULAZI KOMPONENTE

Veličina označena kao ulaz predstavlja osobinu komponente dekoratorom Input().

Ulazima je specificirano koje parametre komponenta može da primi. Da bi veličina bila označena kao ulaz, neophodno je obeležiti odgovarajuću osobinu komponente dekoratorom `@Input()`. Kada je cilj da se istakne da komponenta prihvata ulaz, klasa komponente mora da poseduje promenljivu u kojoj će ulazni podaci biti čuvani. Na primer, to je moguće ilustrovati sledećim kodom:

```
import { Component, Input } from '@angular/core';

@Component({
  selector: 'my-component',
})
class MyComponent {
  @Input() name: string;
  @Input() age: number;
}
```

Slika 5.2.5 Varijabla za prijem ulaza [izvor: autor]

PROSLEĐIVANJE PROIZVODA PREKO ULAZA

Prosleđivanje proizvoda preko ulaza primenom taga šablonu i izraza ulaza.

U ovom delu izlaganja, dovoljno je prisetiti se da je u kodu prosleđena lista proizvoda za prikazivanje, u okviru taga `<products-list>` primenom izraza ulaza: `[productList]`. Navedeno je ilustrovano sledećom slikom.

```
<div class="inventory-app">
  <products-list
    [productList]="products"
    (onProductSelected)="productWasSelected($event)">
  </products-list>
</div>
```

Slika 5.2.6 Prosleđivanje proizvoda preko ulaza [izvor: autor]

Primena ovakve sintakse je veoma jednostavna, prosleđuje se vrednost `this.products()` na komponentu `AppComponent` putem unosa sa komponente `ProductsListComponent`.

IZLAZI KOMPONENTA

Kada je neophodno prosleđivanje podataka komponenti koristi se koncept povezivanja izlaza.

Kada je neophodno prosleđivanje podataka komponenti, iz njenog okruženja, koristi se koncept povezivanja izlaza. Na primer, komponenta koja se kreira poseduje dugme i nešto se dešava kada korisnik klikne da to dugme. Kao posledica ove akcije korisnika, vrši se povezivanje izlaza dugmeta sa metodom koja je deklarisana u odgovarajućem kontroleru. Za ovo se koristi notacija u opštem obliku:

```
(output)="action"
```

Sledećom slikom je prikazano prosleđivanje izlaza ka metodi kontrolera

```
<div class="inventory-app">
  <products-list
    [productList]="products"
    (onProductSelected)="productWasSelected($event)">
  </products-list>
</div>
```

Slika 5.2.7 Prosleđivanje izlaza ka metodi kontrolera [izvor: autor]

EMITOVANJE KREIRANIH DOGAĐAJA

Kreira se komponenta koja emituje kreirani događaj.

Neka se sada razmatra sledeći scenario - kreira se komponenta koja emituje kreirani događaj, poput događaja: `click` ili `mousedown`. Za kreiranje izlaza neophodno je obaviti tri stvari:

1. specifikacija izlaza unutar `@Component` podešavanja;

2. povezivanje *EventEmitter* sa izlaznom osobinom;
3. emitovanje događaja putem *EventEmitter* u odgovarajućem trenutku.

Navedeno je u potpunosti ilustrovano sledećom slikom.

```
import {
  Component,
  OnInit,
  EventEmitter,
  Input,
  Output }
from '@angular/core';

import { Product } from '../product.model';

@Component({
  selector: 'products-list',
  templateUrl: './product-list.component.html',
  styleUrls: ['./product-list.component.css']
})
export class ProductListComponent implements OnInit {

  @Input() productList: Product[];

  // 2. povezivanje EventEmitter sa izlaznom osobinom
  @Output() onProductSelected: EventEmitter<Product>;

  private currentProduct: Product;

  constructor() {
    // 3. emitovanje događaja putem EventEmitter
    this.onProductSelected = new EventEmitter();
  }
}
```

Slika 5.2.8 Emitovanje kreiranih događaja [izvor: autor]

KREIRANJE KONTROLERSKE KLASE PRODUCTSLISTCOMPONENT

Kontrolerska klasa ProductsListComponent dobija tri objektne varijable.

Nastavlja se dalja analiza i kreiranje pratećeg primera. Kontrolerska klasa *ProductsListComponent* dobija tri objektne varijable:

- prva čuva listu proizvoda;
- druga predstavlja izlazni događaj;
- treća označava referencu na trenutno izabrani proizvod.

Kod koji uvažava navedene zahteve prikazan je sledećim listingom:

```
export class ProductListComponent implements OnInit {

  @Input() productList: Product[];

  // 2. povezivanje EventEmitter sa izlaznom osobinom
```

```
@Output() onProductSelected: EventEmitter<Product>;  
  
private currentProduct: Product;  
  
constructor() {  
    //3. emitovanje događaja putem EventEmitter  
    this.onProductSelected = new EventEmitter();  
}  
  
}
```

PISANJE ŠABLONA PRODUCTSLISTCOMPONENT

Konačna definicija komponente se dobija kodiranjem odgovarajućeg šablona.

Konačna definicija komponente se dobija kodiranjem odgovarajućeg šablona. Sledećim listingom je moguće dati konačan listing šablona komponente [šablona ProductsListComponent](#). Listing odgovara sadržaju datoteke: [src/app/products-list/products-list.component.html](#).

```
<div class="ui items">  
    <product-row  
        *ngFor="let myProduct of productList"  
        [product]="myProduct"  
        (click)='clicked(myProduct)'  
        [class.selected]="isSelected(myProduct)">  
    </product-row>  
</div>
```

Iz listinga je moguće primetiti da je neophodno obaviti kreiranje nove komponente koja će odgovarati primjenjom tagu [`<product-row>`](#) za prikazivanje svakog proizvoda, iz liste, u vlastitom redu.

Primenom petlje [`ngFor`](#) prolazi se kroz listu proizvoda i vrši generisanje lokalne promenljive [`myProduct`](#) za svaki proizvod iz liste.

Dalje, linija [`\[product\] = "myProduct"`](#) ukazuje da bi trebalo proslediti kreiranu lokalnu varijablu [`myProduct`](#) kao ulaznu vrednost [`product`](#) unutar taga [`<product-row>`](#).

U liniji [`\(click\)='clicked\(myProduct\)'`](#) koristi se ugrađeni događaj [`click`](#) koji poziva funkciju [`clicked\(\)`](#) komponente [`ProductsListComponent`](#) svaki put kada korisnik obavi klik na odgovarajući element iz liste.

Linija koda [`\[class.selected\] = "isSelected\(myProduct\)"`](#) bazira se na činjenici da Angular dozvoljava, primenom ove sintakse, podešavanje klase za element uslovno. Konkretno, neophodno je dodati [`CSS`](#) klasu [`selected`](#) ukoliko poziv [`isSelected\(myProduct\)`](#) vrati vrednost [`true`](#).

PROŠIRENJE KONTROLERA NOVIM FUNKCIONALNOSTIMA

Proširenje kontrolera novim funkcionalnostima za servisiranje koda šablonu.

Iz prethodnog izlaganja je moguće zaključiti da funkcije `clicked()` i `isSelected()` još nisu definisani. To je moguće uraditi izmenom koda kontrolera `src/app/products-list/products-list.component.ts` na sledeći način:

```
clicked(product: Product): void {
  this.currentProduct = product;
  this.onProductSelected.emit(product);
}

isSelected(product: Product): boolean {
  if (!product || !this.currentProduct) {
    return false;
  }
  return product.sku === this.currentProduct.sku;
}
```

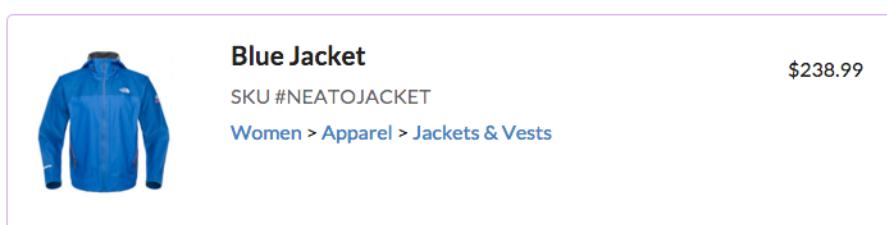
Kod govori sledeće:

- metoda `clicked()` - promenljivoj `this.currentProduct` je pridružen proizvod koji je prosleđen i vrši se slanje izabranog proizvoda na izlaz;
- metoda `isSelected()` - metoda prihvata proizvod i vraća vrednost `true` ukoliko osobina `product.sku` odgovara osobini `currentProduct.sku`. U suprotnom, vratiće `false`.

KOMPONENTA PRODUCTROWCOMPONENT

Aplikaciji nedostaje komponenta za prikazivanje sadržaja određenog tagom `<product-row>`.

Kao što je bilo moguće sagledati iz prethodnog izlaganja, aplikaciji nedostaje komponenta koja će omogućiti prikazivanje sadržaja određenog tagom `<product-row>`. Primer takvog sadržaja je moguće ilustrovati sledećom slikom:



Slika 5.2.9 Prikaz pojedinačnih proizvoda iz liste proizvoda [izvor: autor]

U razvojnog okruženju se kuca `Angular CLI` naredba:

```
ng generate component ProductRow
```

Ubrzo će biti kreirana željena komponenta i sada se postavljaju dva nova izazova:

- definisanje kontrolera komponente;
- definisanje pogleda (šablon) komponente.

Ako se obrati detaljnija pažnja na sliku 9 moguće je primetiti i postojanje: slike predmeta, informacija o odeljenju na kojem se predmet nalazi i cenu predmeta. Iz ovoga je moguće izvući zaključak da je ovu komponentu neophodno dodatno razložiti na sledeće tri komponente:

- *ProductImageComponent* - za prikazivanje slika;
- *ProductDepartmentComponent* - za prikazivanje informacija o odeljenju (detalji ili "mrvice hleba");
- *PriceDisplayComponent* - komponenta koja omogućava prikazivanje cena pojedinačnih proizvoda.

KONTROLER KOMPONENTE PRODUCTROWCOMPONENT

Kontroler komponente ProductRowComponent ima zadatak definisanje taga za šablon.

Kontroler komponente *ProductRowComponent* ima zadatak definisanje taga za šablon. Navedeno je urađeno u liniji koda 12, pod selektorom, a tag *<product-row>* već je upotrebljen unutar šablonu *src/app/products-list/products-list.component.html*.

Dalje, moguće je primetiti (iz listinga koji sledi) da klasa ima dve osobine:

- *product* - ova promenljiva će biti podešena za tip podataka *Product* onog trenutka kada proizvod bude prosleđen iz roditeljske komponente;
- *cssClass* - ova osobina je obeležena dekoratorom *@HostBinding* i ukazuje na CSS klasu *attr.class* i njen element "*item*" kojeg je neophodno povezati sa host elementom, odnosno konkretnim objektom klase *ProductRowComponent*.

```
import {  
  Component,  
  Input,  
  HostBinding  
} from '@angular/core';  
import { Product } from '../product.model';  
  
/**  
 * @ProductRow: komponenta za prikazivanje pojedinačnih objekata  
 */  
@Component({  
  selector: 'product-row',  
  templateUrl: './product-row.component.html',  
})
```

```
export class ProductRowComponent {  
  @Input() product: Product;  
  @HostBinding('attr.class') cssClass = 'item';  
}
```

ŠABLON KOMPONENTE PRODUCTROWCOMPONENT

Šablon koriti tagove novih potkomponenata.

Šablon komponente `ProductRowComponent` određen je datotekom `src/app/product-row/product-row.component.html` čiji je sadržaj prikazan sledećim listingom:

```
<product-image [product]="product"></product-image>  
<div class="content">  
  <div class="header">{{ product.name }}</div>  
  <div class="meta">  
    <div class="product-sku">SKU #{{ product.sku }}</div>  
  </div>  
  <div class="description">  
    <product-department [product]="product"></product-department>  
  </div>  
</div>  
<price-display [price]="product.price"></price-display>
```

Koncepcijski, šablon ne sadrži ništa novo ali je moguće primetiti da insistira na podeli tekuće komponente na tri nove potkomponente koje su određene tagovima: `<product-image>`, `<product-department>` i `<price-display>`.

KOMPONENTA ZA PRIKAZIVANJE SLIKE PROIZVODA

Komponenta za prikazivanje slike proizvoda u potpunosti se oslanja na vlastiti kontroler.

Komponenta za prikazivanje slike proizvoda je definisana tagom `<product-image>` (videti selektor) i u potpunosti se oslanja na vlastiti kontroler definisan `TypeScript` klasom `ProductImageComponent`.

Ovde je uvedeno malo odstupanje od dosadašnjeg izlaganja. Umesto definisanja šablona komponente u zasebnoj datoteci, on je realizovan unutar dekoratora `@Component` pod ključem `template` (linija koda 13-14).

Konačno, klasa uzima kao ulaz objekat klase `Product` i koristi CSS klasu `ui small image` koja se koristi za stilizovano prikazivanje slika pojedinačnih proizvoda.

```
import {  
  Component,  
  Input,  
  HostBinding
```

```
} from '@angular/core';
import { Product } from '../product.model';

/**
 * @ProductImage: komponenta koja prikazuje sliku proizvoda
 */
@Component({
  selector: 'product-image',
  template: `
    <img class="product-image" [src]="product.imageUrl">
  `
})
export class ProductImageComponent {
  @Input() product: Product;
  @HostBinding('attr.class') cssClass = 'ui small image';
}
```

KOMPONENTA ZA PRIKAZIVANJE CENE PROIZVODA

Komponenta za prikazivanje slike proizvoda definiše šablon unutar kontrolera.

Komponenta za prikazivanje slike proizvoda, takođe, definiše šablon unutar kontrolera. Koristi jedan `${{ price }}` (linija koda 12) izraz koji zapravo prikazuje vrednost koju ova klasa prima na ulazu (linija koda 16)

```
import {
  Component,
  Input
} from '@angular/core';

/**
 * @PriceDisplay: komponenta prikazuje cenu pojedinačnog proizvoda
 */
@Component({
  selector: 'price-display',
  template: `
    <div class="price-display">\${{ price }}</div>
  `
})
export class PriceDisplayComponent {
  @Input() price: number;
}
```

KOMPONENTA ZA PRIKAZIVANJE ODELJENJA PROIZVODA

Završna definicija aplikacije je komponenta za prikazivanje odeljenja proizvoda.

Sledećim listingom je dat kontroler ove komponente koji kao ulaz uzima objekat klase *Product*.

```
import {  
  Component,  
  Input  
} from '@angular/core';  
import { Product } from '../product.model';  
  
/**  
 * @ProductDepartment: prikazuje detalje odeljenja sa kojeg  
 * se uzima proizvod  
 */  
@Component({  
  selector: 'product-department',  
  templateUrl: './product-department.component.html'  
})  
export class ProductDepartmentComponent {  
  @Input() product: Product;  
}
```

Detaljniju pažnju je potrebno posvetiti šablonu komponente određenog datotekom *src/app/product-department/product-department.component.html*

```
<div class="product-department">  
  <span *ngFor="let name of product.department; let i=index">  
    <a href="#">{{ name }}</a>  
    <span>{{i < (product.department.length-1) ? '>' : ''}}</span>  
  </span>  
</div>
```

Petlja *ngFor* prelazi preko svih *product.department* i pridružuje svaki *department* string osobini *name*. Izraz: *let i=index* određuje broj iteracije za petlju *ngFor*.

U tagu ** upotrebljena je promenljiva *i* kojom je određeno prikazivanje simbola *>*. Cilj je prikazivanje odeljenja, kojem proizvod pripada, u formi stringa:

```
Women > Apparel > Jackets & Vests
```

Izraz *{{i < (product.department.length-1) ? '>' : ''}}* ukazuje da je moguće prikazati znak *>* ukoliko se ne radi o poslednjem odeljenu. U suprotnom, prikazuje se prazan string.

NGMODULE I POKRETANJE APLIKACIJE

Neophodno je proveriti da li su sve komponente registrovanje u NgModule klasi.

Neophodno je proveriti da li su sve komponente registrovanje u *NgModule* klasi. Budući da je prilikom izrade projekta korišćen *Angular CLI*, sve komponente su automatski registrovane. Takođe, komponenta kojom se pokreće aplikacija bi trebalo da bude komponenta najvišeg

nivoa *AppComponent*. Ukoliko klasa sa lokacije *src/app/app.module.ts* i pod nazivom *AppModule* (NgModule klasa projekta) poseduje sledeći sadržaj, prethodni posao je dobro obavljen.

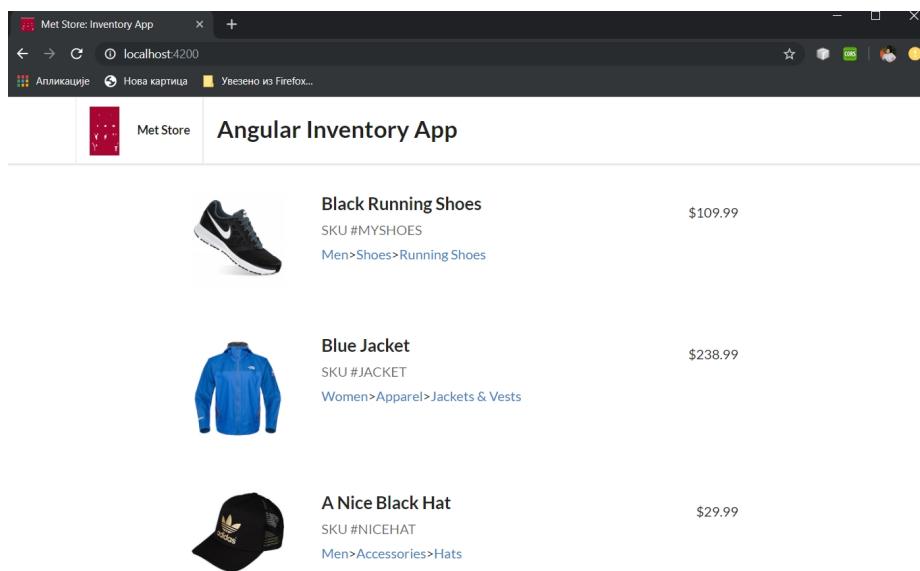
```
import { BrowserModule } from '@angular/platform-browser';
import { NgModule } from '@angular/core';

import { AppRoutingModule } from './app-routing.module';
import { AppComponent } from './app.component';
import { ProductListComponent } from './product-list/product-list.component';
import { ProductRowComponent } from './product-row/product-row.component';
import { ProductImageComponent } from './product-image/product-image.component';
import { ProductDepartmentComponent } from './product-department/product-department.component';
import { PriceDisplayComponent } from './price-display/price-display.component';

@NgModule({
  declarations: [
    AppComponent,
    ProductListComponent,
    ProductRowComponent,
    ProductImageComponent,
    ProductDepartmentComponent,
    PriceDisplayComponent
  ],
  imports: [
    BrowserModule,
    AppRoutingModule
  ],
  providers: [],
  bootstrap: [AppComponent]
})
export class AppModule { }
```

Slika 5.2.10 Klasa AppModule (NgModule klasa projekta) [izvor: autor]

Konačno, moguće je prevesti aplikaciju, instrukcijom *ng serve* i u web pregledaču pozivom: *localhost:4200* pogledati rezultat izvršavanja aplikacije:



Slika 5.2.11 Konačan izgled aplikacije [izvor: autor]

Kompletno urađen primer možete preuzeti odmah nakon ovog objekta učenja!

VIDEO MATERIJAL

Angular 2 Tutorial - 2 - Architecture - trajanje 4:54 minuta.

Ova lekcija sadrži video materijal. Ukoliko želite da pogledate ovaj video morate da otvorite LAMS lekciju.

▼ Poglavlje 6

Analiza – arhitektura podataka

O ANGULAR ARHITEKTURI PODATAKA

Problem upravljanja tokovima podataka ukoliko se počne sa dodavanjem novih funkcionalnosti.

Sa ovim nivoom znanja, studenti mogu da se zapitaju kako je moguće upravljanje tokovima podataka ukoliko se počne sa dodavanjem novih funkcionalnosti. Na primer, kreira se nov pogled koji odgovara potrošačkoj korpi tako da će u budućnosti aplikacija imati mogućnost dodavanja novih proizvoda u korpu.

Kako je moguće rešiti ovaj problem?

Jedini alat o kojem je do sada bilo diskusije jeste emitovanje izlaznih događaja. Kada bi bio izvršen klik na link ili dugme `add-to-cart` vrši se kreiranje događaja `addedToCart` i rukovanje događajem u korenskoj komponenti. Ovo može biti nezgodno.

Arhitektura podataka je velika tema sa brojnim opcijama. Angular je, po ovom pitanju, veoma fleksibilan i omogućava rukovanje širokim spektrom arhitektura podataka. Ostavlja se programeru da odluči koju želi da koristi.

ARHITEKTURE PODATAKA I VERZIJE ANGULAR OKVIRA

U okviru Angular 1, podrazumevana opcija je bila dvosmerno povezivanje podataka.

U okviru Angular 1, podrazumevana opcija je bila dvosmerno povezivanje podataka (*two-way data binding*). Za početak ovo je bila odlična opcija:

- kontroleri imaju podatke;
- forme direktno manipulišu podacima;
- pogledi prikazuju podatke.

Problem se javio kada je ovaj pristup počeo da rezultuje kaskadnim efektima u aplikaciji pri čemu je praćenje tokova podataka postao otežan, kako je projekat rastao.

Sledeći problem koji se javio kod dvosmernog povezivanja podataka jeste da kada se prosledi podatak "naniže" kroz komponente često se forsira čvrsto podudaranje stabla organizacije

podataka (*data layout tree*) sa stablom pogleda (*dom view tree*). U praksi, ove stvari moraju da se razdvoje.

Jedan od načina bavljenja navedenim scenarijem je kreiranje *ShoppingCartService*, kao *singlon*-a koji bi držao listu trenutnih stavki u potrošačkoj korpi. Ova usluga može obavestiti sve relevantne objekte kada je stavka u korpi promenjena.

Savremeni veb okviri, i nove verzije Angular-a, predlažu nešto drugačiji pristup - šablon jednosmernog povezivanja podataka (*one-way data binding*). U ovom slučaju, tokovi podataka teku isključivo naniže kroz komponente. Ukoliko je potrebno napraviti promenu, emituje se događaj koji izaziva promene koje se sa vrha prelivaju naniže. Na ovaj način su izbegnute brojne komplikacije i olakšano je praćenje tokova podataka u aplikaciji.

Srećom, postoje dva glavna kandidata za upravljanje arhitekturom podataka, o kojima će detaljno biti kasnije govora:

- primena *Observables* bazirane arhitekture, poput *RxJS*;
- primena *Flux* bazirane arhitekture.

▼ Poglavlje 7

Uvod u Angular i TypeScript dodatni materijali

DODATNI MATERIJALI

Proširivanje naučenog na predavanju.

Proučite dodatnu literaturu:

1. <https://angular.io/>
2. <https://angular.io/tutorial>
3. <https://www.w3schools.com/angular/>
4. <https://www.tutorialspoint.com/angular4/>
5. <https://nodejs.org/en/>
6. <https://code.visualstudio.com/>

▼ Poglavlje 8

Pokazna vežba 7 (TRAJANJE VEŽBE: 45 minuta)

POKAZNA VEŽBA I NASTAVAK RADA SA PRIMEROM IZ VEŽBE 6

Proširuju se Angular funkcionalnosti kroz nastavak rada sa primerom iz Vežbe 6.

Nastavljamo rad na primeru sa prethodnih pokaznih vežbi. Preuzmite iz nastavnih materijala urađen primer pokazne vežbe iz prethodne nedelje.

Uvode se nove funkcionalnosti:

1. Iznad tabele, kreirane u prethodnom primeru, dodati navigacioni bar sa linkovima: *Metropolitan, Facebook* (stranica Univerziteta), *Fakulteti*;
2. Markiranjem opcije Fakulteti, dobija se padajući meni sa stavkama: *FIT, FAM, FDU*;
3. Svaka od izabranih stavki obavlja navigaciju ka odgovarajućoj veb stranici;
4. Ispod tabele biće kreiran bar sa dugmićima: *OAS, MAS i DS*.
5. Klikom na odgovarajuće dugme popunjava se nova tabela sa podacima o izabranom tipu studiranja.

Ideja je da se provežba:

- kreiranje i primena komponenata;
- kreiranje metoda;
- primena interakcije korisnik - aplikacija;
- primena CSS stilova u Angular aplikaciji

Konačno rešenje bi trebalo da ima ovakav izgled:



Slika 8.1 Novi izgled primera pokazne vežbe [izvor: autor]

KOMPONENTA NAVBARCOMPONENT, KREIRANJE I ŠABLON

Komponenta navbarcomponent ima zadatak prikaz i primenu navigacionog bara.

Komponenta navbarcomponent ima zadatak prikaz i primenu navigacionog bara. Vizuelni prikaz komponente, zajedno sa pripadajućim padajućim menijem, je dat sledećom slikom:



Slika 8.2 Komponenta navbarcomponent [izvor: autor]

Da bi komponenta bila kreirana, u terminalu razvojnog okruženja, ulazimo u folder aplikacije i navodimo komandu:

```
ng generate component navbarcomponent
```

Prva datoteka koja će biti modifikovana jeste šablon. Šablon prikazuje bar sa linkovima i padajućim menijem. Sledi Isiting ove datoteke:

```
<div class="navbar">
    <a href="https://www.metropolitan.ac.rs/">Univerzitet Metropolitan</a>
    <a href="https://www.facebook.com/UniverzitetMetropolitan/">Facebook</a>
    <div class="dropdown">
        <button class="dropbtn">Fakulteti
            <i class="fa fa-caret-down"></i>
        </button>
        <div class="dropdown-content">
            <a href="https://www.metropolitan.ac.rs/osnovne-studije/
fakultet-informacionih-tehnologija/">FIT</a>
            <a href="https://www.metropolitan.ac.rs/osnovne-studije/
fakultet-za-menadzment/">FAM</a>
            <a href="https://www.metropolitan.ac.rs/osnovne-studije/
fakultet-za-menadzment/">FDU</a>
        </div>
    </div>
</div>
```

Ako se pogleda listing moguće je sagledati da je kod stilizovan CSS klasama *navbar* (linija koda 1 - 13) i *dropdown* (linije koda 4 - 12), kao i pomoćnim CSS klasama *dropbtn* i *dropdown-content* za stilizovanje dugmeta na baru i sadržaja padajućeg menija, respektivno. Sledi sadržaj CSS datoteke kreirane komponente:

```
/* Navbar kontejner */
.navbar {
    overflow: hidden;
    background-color: darkred;
    font-family: Arial;
}

/* linkovi u komponenti navbar */
.navbar a {
    float: left;
    font-size: 16px;
    color: white;
    text-align: center;
    padding: 14px 16px;
    text-decoration: none;
}

/* padajući kontejner */
.dropdown {
    float: left;
    overflow: hidden;
}

/* dugmići */
.dropdown .dropbtn {
    font-size: 16px;
    border: none;
    outline: none;
    color: white;
```

```
padding: 14px 16px;
background-color: inherit;
font-family: inherit; /* važno vertikalno poravnanje za mobilne telefone */
margin: 0; /* važno vertikalno poravnanje za mobilne telefone */
}

/* plava pozadina obleženog linka */
.navbar a:hover, .dropdown:hover .dropbtn {
    background-color: blue;
}

/* sadržaj padajućeg menija */
.dropdown-content {
    display: none;
    position: absolute;
    background-color: #f9f9f9;
    min-width: 160px;
    box-shadow: 0px 8px 16px 0px rgba(0,0,0,0.2);
    z-index: 1;
}

/* Linkovi */
.dropdown-content a {
    float: none;
    color: darkred;
    padding: 12px 16px;
    text-decoration: none;
    display: block;
    text-align: left;
}

.dropdown-content a:hover {
    background-color: #ddd;
}

/* prikazivanje menija kada se markira mišem */
.dropdown:hover .dropdown-content {
    display: block;
}
```

KLASA KOMPONENTE NAVBARCOMPONENT

Klasa komponente navbarcomponent definiše selektor komponente.

Klasa komponente navbarcomponent definiše selektor komponente i to je trenutno jedini njen zadatak.

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

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

  constructor() { }

  ngOnInit(): void {
  }

}
```

Neophodno je da se vratimo korak unazad i da ovi komponentu dodamo u roditeljsku komponentu [appcomponent.ts](#), na vrh šablona definisanog ključem template (linija koda 8 u sledećem listingu).

```
import { style } from "@angular/animations";
import { Component } from "@angular/core";

@Component({
  selector: 'my-app',
  template: `
    <app-navbarcomponent></app-navbarcomponent>
    <div>
      <h1>{{pageheader}}</h1>
    </div>

    <app-studentcomponent></app-studentcomponent>
    <app-buttongroupcomponent></app-buttongroupcomponent>
  `

})

export class AppComponent {
  pageheader: string = "Detalji o studentu ili profesoru"
}
```

TESTIRANJE KOMPONENTE NAVBARCOMPONENT

Prevodimo i testiramo aplikaciju.

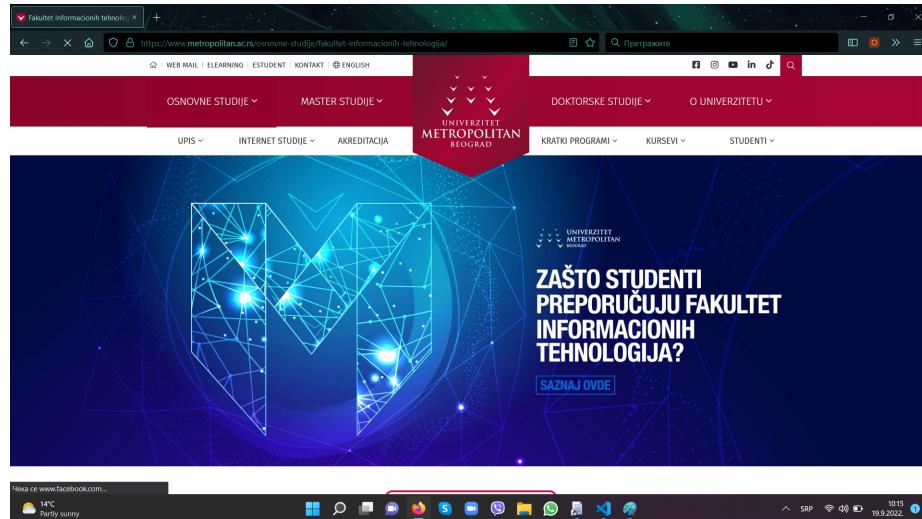
U terminalu kucamo naredbu:

```
ng serve
```

Čekamo da kreirani kod bude preveden i ako je pokrenemo na linku <localhost:4200/dobicemo> izlaz kao na slici 1.

Za proveru valjanosti urađenog posla, izaberimo jednu od opcija, na primer opciju **FIT** u padajućem meniju **Fakulteti**. Markiranjem mišem opcije fakulteti, dugme iz bara menjaju boju, posledica primene CSS stila, otvaraju se opcije od kojih biramo FIT (videti sliku 2).

Izborom navedene opcije, aplikacija prelazi na prezentaciju Fakulteta informacionih tehnologija, kao na sledećoj slici.



Slika 8.3 Izbor opcije FIT [izvor: autor]

KOMPONENTA BUTTONGROUPCOMPONENT, KREIRANJE, MODEL I ŠABLON

Komponenta buttongroupcomponent prikazuje i primenjuje navigacioni bar sa dugmićima.

Komponenta buttongroupcomponent prikazuje i primenjuje navigacioni bar sa dugmićima: OAS, MAS, DS.

Cilj je da izborom opcije, u posebno kreiranoj tabeli, priažu se podaci vezani za izabrani nivo studiranja. Da bi ovakvi pdaci mogli da budu sačuvani i primenjeni, neophodno je da bude kreirana modelska klasa **studije.ts** u posebno kreiranom folderu **app/model**. Sledi listing klase:

```
export class Studije {
  naziv: string;
  trajanje: number;
  espb: number;

}
```

Da nove verzije Angulara okvira, ne bi javljale grešku da osobine kreirane klase nisu inicijalizovane, neophodno je u datoteci tsconfig.json dodati sledeću liniju koda:

"strictPropertyInitialization": false

Kucanjem naredbe

```
ng generate component navbarcomponent
```

kreiramo novu komponentu, nakon čega pristupamo modifikaciji njenog šablona. Navedeno je prikazano sledećim listingom.

```
<br/>
<div class="btn-group">
  <button (click)="on0as()">0AS</button>
  <button (click)="onMas()">MAS</button>
  <button (click)="onDs()">DS</button>
</div>

<br/>

<table>
  <tr>
    <th>Studije</th>
    <th>Trajanje</th>
    <th>ESPB</th>
  </tr>
  <tr>
    <td> {{ this.studije.naziv }} </td>
    <td> {{ this.studije.trajanje }} </td>
    <td> {{ this.studije.espb }} </td>
  </tr>
</table>
```

Šablon kreira tri dugmeta koja proizvode događaje tipa `click` za koje su predviđene tri metode za njihovu obradu (linije koda 3 - 5). Linije kora 10 - 22 grade tabelu u kojoj će biti prikazani podaci u zavisnosti od izabranog dugmeta.

KLASA KOMPONENTE BUTTONGROUPCOMPONENT

Klasa komponente `buttongroupcomponent` definiše selektor komponente i metode obrađivače događaja.

Klasa komponente `buttongroupcomponent` definiše selektor komponente i metode obrađivače događaja. Sledi njen listing:

```
import { Component, OnInit } from '@angular/core';
import { Studije } from '../model/studije';

@Component({
  selector: 'app-buttongroupcomponent',
```

```
templateUrl: './buttongroupcomponent.component.html',
styleUrls: ['./buttongroupcomponent.component.css']
})
export class ButtongroupcomponentComponent implements OnInit {

studije: Studije = new Studije();

constructor() { }

ngOnInit(): void {
}

onOas(){
  this.studije.naziv = "Osnovne akademske studije";
  this.studije.trajanje = 4;
  this.studije.espb = 240;
  console.log("OAS");

}

onMas(){
  this.studije.naziv = "Master akademske studije";
  this.studije.trajanje = 1;
  this.studije.espb = 60;
  console.log("MAS");

}

onDs(){
  this.studije.naziv = "Doktorskse akademske studije";
  this.studije.trajanje = 3;
  this.studije.espb = 180;
  console.log("DS");

}

}
```

Iz listinga se primećuje uključivanje modelske klase *Studije* (linija koda 2) i kreiranje njenog objekta (linija koda 11).

Klasa implementira tri metode, definisane za prethodno kreiranje dugmiće u šablonu, kojima se obrađuju događaji klika na odgovarajuće dugme. Svaka od metoda dodeljuje odgovarajuće vernosti osobinama kreiranog objekta, koje se potom, prikazuju u tabeli šablonu (linije koda 18 - 40).

Neophodno je da se vratimo korak unazad i da ovi komponentu dodamo u roditeljsku komponentu *appcomponent.ts* (linija koda 14 u sledećem listingu).

```
import { style } from "@angular/animations";
import { Component } from "@angular/core";
```

```
@Component({  
  
    selector: 'my-app',  
    template: `  
        <app-navbarcomponent></app-navbarcomponent>  
        <div>  
            <h1>{{pageheader}}</h1>  
        </div>  
  
        <app-studentcomponent></app-studentcomponent>  
        <app-buttongroupcomponent></app-buttongroupcomponent>  
        `  
  
    })  
  
export class AppComponent {  
    pageheader: string = "Detalji o studentu ili profesoru"  
}
```

TESTIRANJE KOMPONENTE BUTTONGROUPCOMPONEN

Prevodimo i testiramo konačnu verziju aplikaciju za ovu vežbu.

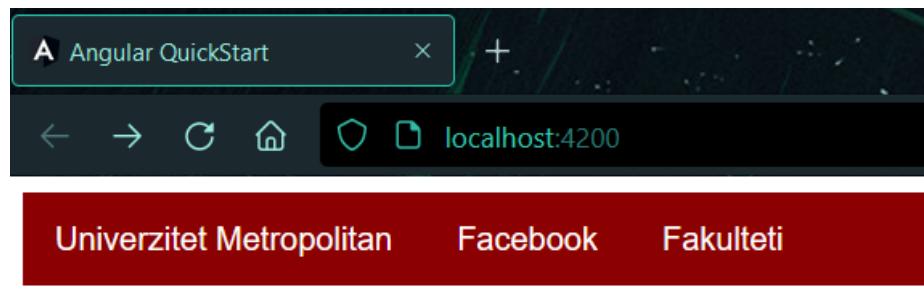
U terminalu kucamo naredbu:

```
ng serve
```

Čekamo da kreirani kod bude preveden i ako je pokrenemo na linku localhost:4200 dobijemo izlaz kao na slici 1.

Za proveru valjanosti urađenog posla, izaberimo jednu od opcija na baru sa dugmićima, na primer opciju [OAS](#).

Izborom navedene opcije, aplikacija prikazuje novu tabelu sa odgovarajućim sadržajem, kao na sledećoj slici.



Detalji o studentu ili profesoru

Ime	Prezime	Titula	Obrazovanje
Vladimir	Milićević	Profesor	PHD

OAS	MAS	DS
Studije	Trajanje	ESPB
Osnovne akademske studije	4	240

Slika 8.4 Testiranje komponente buttongroupcomponent [izvor: autor]

✓ Poglavlje 9

Individualna vežba 7

INDIVIDUALNA VEŽBA (TRAJANJE: 90 MINUTA)

Samostalna dopuna zadatka sa pokaznih vežbi.

1. Kreirati komponentu listaProfesora i Studenata;
2. Kreirati bar sa dugmatima PROFESORI i STUDENTI;
3. Klikom na odgovarajuće dugme, u listi se (tabeli) se primenjuju svi profesori ili studenti.
4. Stilizovati šablon.
5. Ukoliko budete imali problem, konsultujte predmetnog asistenta.

✓ Poglavlje 10

Domaći zadatak 7

DOMAĆI ZADATAK 7 (PREDVIĐENO VREME 120 MIN)

Samostalna izrada domaćeg zadatka.

Na projekat započet prethodnim domaćim zadatkom dodati sledeće funkcionalnosti:

1. Dodati komponentu (bar) sa stavkama: ponuda, preporuka, o nama.
2. Za svaku od opcija kreirati odgovarajuću komponentu;
3. Izborom opcije prikazuje se: lista sa ponudama, lista sa preporukama (na primer za obrok, izlazak i sl.), kao i podaci o MetHotel.
4. Dodati CSS po želji.

Domaći zadatak je potrebno dostaviti kao *Github commit* na prethodni, pod nazivom „*IT255-DZ07*“. Link do zadatka poslati predmetnom asistentu na mail.

NAPOMENA: Domaći zadatak dodati kao commit na prethodni zadatak, a ne kreirati novi repozitorijum.

▼ Poglavlje 11

Zaključak

ZAKLJUČAK

Lekcija je bazirala svoje izlaganje kroz dve celine: TypeScript i funkcionisanje Angular okvira.

Kao što je već istaknuto, lekcija je podeljena u dva dela:

- *TypeScript* diskusija i
- funkcionisanje *Angular* okvira.

U prvom delu je posebno bilo govora o migraciji Angular okvira sa standardnog JavaScript jezika ka njegovom proširenju u formi TypeScript notacije. Lekcija je kroz svoje objekte učenja i sekcije posebno diskutovala o:

- prednostima primene *TypeScript* okvira;
- primeni tipova podataka u *TypeScript* okviru;
- primeni objektno - orijentisanih koncepcata *TypeScript* okvira;
- programskim alatima koji olakšavaju kodiranje u ovom okviru.

Drugi deo lekcije bio je rezervisan za analizu funkcionisanja Angular okvira sa akcentom na sledećim temama:

- aplikacija kao "top - level" komponenta;
- modeli u Angular aplikacijama;
- kreiranje i primena komponenata u Angular aplikacijama;
- analiza arhitekture podataka u Angular okviru.

LITERATURA

Za pripremu lekcije korišćena je aktuelna pisana i elektronska literatura.

Pisana literatura:

1. Nate Murray, Felipe Coury, Ari Lerner, Carlos Taborda, ng-Book – The Complete Book on Angular 6, Fullstack.io, 2018
2. Cody Lindley, Frontend Handbook – 2017, Frontend masters, 2017
3. Sandeep Panda, AngularJS – From Novice to Ninja, SitePoint Pty. Ltd, 2014

Elektronska literatura:

4. <https://angular.io/>
5. <https://angular.io/tutorial>
6. <https://www.w3schools.com/angular/>
7. <https://www.tutorialspoint.com/angular4/>
8. <https://nodejs.org/en/>
9. <https://code.visualstudio.com/>



IT255 - VEB SISTEMI 1

Ugrađene directive i rad sa formama

Lekcija 08

PRIRUČNIK ZA STUDENTE

IT255 - VEB SISTEMI 1

Lekcija 08

UGRAĐENE DIRECTIVE I RAD SA FORMAMA

- ✓ Ugrađene directive i rad sa formama
- ✓ Poglavlje 1: Direktive grananja
- ✓ Poglavlje 2: Direktiva stila u Angularu
- ✓ Poglavlje 3: Direktiva klasa u Angularu
- ✓ Poglavlje 4: Direktive petlji u Angularu
- ✓ Poglavlje 5: Direktiva ngNonBindable
- ✓ Poglavlje 6: Forme u Angular okviru
- ✓ Poglavlje 7: Dodatni materijali za rad
- ✓ Poglavlje 8: Pokazna vežba 8 (TRAJANJE: 45 minuta)
- ✓ Poglavlje 9: Individualna vežba 8
- ✓ Poglavlje 10: Domaći zadatak 8
- ✓ Zaključak

Copyright © 2017 – UNIVERZITET METROPOLITAN, Beograd. Sva prava zadržana. Bez prethodne pismene dozvole od strane Univerziteta METROPOLITAN zabranjena je reprodukcija, transfer, distribucija ili memorisanje nekog dela ili čitavih sadržaja ovog dokumenta., kopiranjem, snimanjem, elektronskim putem, skeniranjem ili na bilo koji drugi način.

Copyright © 2017 BELGRADE METROPOLITAN UNIVERSITY. All rights reserved. No part of this publication may be reproduced, stored in a retrieval system or transmitted in any form or by any means, electronic, mechanical, photocopying, recording, scanning or otherwise, without the prior written permission of Belgrade Metropolitan University.

✓ Uvod

UVOD

Direktive predstavljaju atribute koji se dodaju u HTML elemente i daju dinamičko ponašanje.

Angular okvir obezbeđuje veliki broj ugrađenih direktiva, koje predstavljaju atribute koji se dodaju u HTML elemente i na taj način kreiraju dinamičko ponašanje Angular veb aplikacije. U ovoj lekciji je, upravo, cilj pokrivanje najčešće korišćenih ugrađenih direktiva kroz detaljnu analizu i diskusiju vođenu primenom adekvatnog primera.

Savladavanjem ove lekcije, student će biti sposoban da koristi osnovne ugrađene Angular direktive i da prvi put kreira dinamičku Angular veb aplikaciju.

Prateći primer, koji objedinjuje primenu navedenih direktiva, priložen je na kraju lekcije. Studenti mogu na ga preuzmu, otvore u razvojnog okruženju i kucanjem naredbi:

```
npm install  
npm start
```

mogu da pokrenu i testiraju urađenu aplikaciju.

PREPORUKA:

Pokušajte sami da uradite aplikaciju, prateći izlaganje lekcije, pre nego što testirate već urađen program. Iz programa samo preuzmите slike i stilove.

UVODNI VIDEO

Trajanje video snimka: 5min 44sek

Ova lekcija sadrži video materijal. Ukoliko želite da pogledate ovaj video morate da otvorite LAMS lekciju.

▼ Poglavlje 1

Direktive grananja

DIREKTIVA NGIF

Direktiva je naredba koja je umetnuta u HTML kod i obavlja neke specifične zadatke.

U ovom delu lekcije cilj je učenje primene grananja u [HTML](#) sekcijama [Angular](#) aplikacije. [Direktiva](#) je nije ništa drugo do naredba koja je umetnuta u HTML kod i obavlja neke specifične zadatke. Zadaci grananja, u Angular aplikacijama, realizuju se primenom jedne od sledeće dve direktive:

- [ngIf](#);
- [ngSwitch](#).

Prva na red za razmatranje dolazi direktiva [ngIf](#). Navedena direktiva se koristi kada je neophodno prikazati ili sakriti neki sadržaj u zavisnosti od ispunjenosti određenog logičkog uslova. Uslov je određen rezultatom nekog logičkog izraza koji se prosleđuje u direktivu.

Ukoliko izraz vrati vrednost [true](#), određeni element (ili sadržaj) biće prikazan na veb stranici. U suprotnom, ukoliko izraz vrati vrednost [false](#), navedeni element će biti uklonjen iz [DOM](#) (*Document Object Model*) - https://www.w3schools.com/whatis/whatis_htmldom.asp.

Sledećim listingom je data ilustracija primene direktive [ngIf](#).

```
<div *ngIf="false"></div> <!-- ne prikazuje se nikada -->
<div *ngIf="a > b"></div> <!-- prikazuje se ako je a veće od b -->
<div *ngIf="str == 'yes'"></div> <!-- prikazuje se ako str ima vrednost "yes" -->
<div *ngIf="myFunc()"></div> <!-- prikazuje se ako myFunc vraća true -->
```

Prethodnim listingom je prikazano nekoliko različitih scenarija upotrebe ove direktive i velika je šansa da ćete prilikom kreiranja vaših veb aplikacija, upravo, koristiti ovu direktivu u nekom od navedenih oblika.

Ova lekcija sadrži video materijal. Ukoliko želite da pogledate ovaj video morate da otvorite LAMS lekciju.

DIREKTIVA NGSWITCH

Preko ngSwitch je moguće prikazati neke elemente u zavisnosti od konkretne vrednosti nekog izraza

Naredbe *if* i *switch* su već rađene u predmetima *CS101* i *CS102*. Logika direktiva *ngIf* i *ngSwitch* potpuno je identična. Ponekad je, u HTML pogledima veb aplikacije, neophodno prikazati neke elemente u zavisnosti od konkretne vrednosti nekog izraza. Posmatra se primer priložen sledećim listingom:

```
<div class="container">
<div *ngIf="myVar == 'A'">Var je A</div>
<div *ngIf="myVar == 'B'">Var je B</div>
<div *ngIf="myVar != 'A' && myVar != 'B'">Var je nešto treće</div>
</div>
```

Kao što je moguće primetiti, ovde je na delu primer višestrukog grananja. Program se različito ponaša ukoliko je vrednost varijable *myVar*: A, B ili nešto treće. Kompleksnost višestrukog grananja se povećava ukoliko je neophodno rukovati novom vrednošću, na primer C. Sa ciljem realizacije navedenog, nije dovoljno samo dodati nov element u *ngIf*, već je potrebno i poslednji slučaj (*case*) proširiti. Navedeno je moguće ilustrovati sledećim listingom:

```
<div class="container">
<div *ngIf="myVar == 'A'">Var je A</div>
<div *ngIf="myVar == 'B'">Var je B</div>
<div *ngIf="myVar == 'C'">Var je C</div>
<div *ngIf="myVar != 'A' && myVar != 'B' && myVar != 'C'">Var je nešto četvrto</div>
</div>
```

Za slučajeve kao što je ovaj, Angular omogućava primenu direktive *ngSwitch*. Prednost primene ove direktive ogleda se u činjenici da se vrednost izraza proverava samo jedanput, a potom se prikazuje element na osnovu vrednosti koja je rezultat tačne provere.

Kada je dobijen rezultat izraza, moguće je postupiti na sledeći način:

- izvršava se neka od direktiva *ngSwitchCase* koja odgovara vrednosti izračunatog izraza;
- izvršava se *ngSwitchDefault*, ukoliko nijedan *ngSwitchCase* nije uparen sa izračunatom vrednošću izraza.

Na lakši, i lepši način, moguće je napisati kod koji je ekvivalentan kodu iz prethodnog listinga.

```
<div class="container" [ngSwitch]="myVar">
<div *ngSwitchCase="'A'">Var je A</div>
<div *ngSwitchCase="'B'">Var je B</div>
<div *ngSwitchCase="'C'">Var je C</div>
<div *ngSwitchDefault>Var je nešto četvrto</div>
</div>
```

Kao što je bio slučaj u programskom jeziku Java, *ngSwitchDefault* je opcionalan element. Ukoliko je izostavljen, ništa neće biti prikazano ukoliko *myVar* ne bude uparena sa odgovarajućom vrednošću iz neke od *ngSwitchCase* naredbe.

ALTERNATIVNA PRIMENA DIREKTIVE NGSWITCHCASE

Primenu naredbe ngSwitchCase moguće je vezati i za vrednosti različitih elemenata.

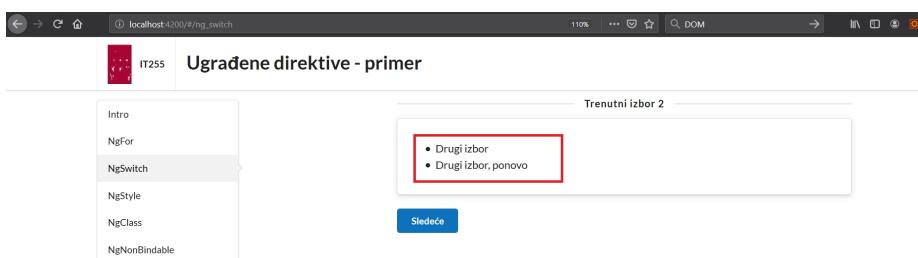
Primenu naredbe *ngSwitchCase moguće je vezati i za vrednosti različitih elemenata tako da ona nije ograničena na proveru podudaranja samo jednom. Navedeno je moguće ilustrovati sledećim listingom:

```
<h4 class="ui horizontal divider header">
  Trenutni izbor {{ choice }}
</h4>

<div class="ui raised segment">
  <ul [ngSwitch]="choice">
    <li *ngSwitchCase="1">Prvi izbor</li>
    <li *ngSwitchCase="2">Drugi izbor</li>
    <li *ngSwitchCase="3">Treći izbor</li>
    <li *ngSwitchCase="4">Četvrti izbor</li>
    <li *ngSwitchCase="2">Drugi izbor, ponovo</li>
    <li *ngSwitchDefault>Podrazumevani izbor</li>
  </ul>
</div>

<div style="margin-top: 20px;">
  <button class="ui primary button" (click)="nextChoice()">
    Sledeće
  </button>
</div>
```

U priloženom kodu neophodno je, posebno, pogledati linije koda 8 i 11. Budući da se odnose na istu vrednost (choice == 2) u slučaju pozitivnog podudaranja oba elementa će biti prikazana. To je moguće videti na sledećoj slici koja predstavlja rezultat izvršavanja Angular aplikacije koja će biti kreirana kao rezultat rada na ovoj lekciji.



Slika 1.1 Dvostruko podudaranje sa ngSwitchCase [izvor: autor]

▼ Poglavlje 2

Direktiva stila u Angularu

DIREKTIVA NGSTYLE

Direktivom `ngStyle` omogućeno je podešavanje CSS osobine DOM elementa iz Angular izraza

Primenom direktive `ngStyle` omogućeno je podešavanje `CSS` osobine `DOM` elementa iz `Angular` izraza.

Najjednostavniji način da se upotrebi ova direktiva jeste pisanje izraza u sledećem opštem obliku:

```
[style.<cssproperty>]="value"
```

Na primer:

```
<div [style.background-color]="'yellow'">  
    Koristi fiksnu žutu pozadinu  
</div>
```

Navedeni listing koristi direktivu `ngStyle` za podešavanje `CSS` osobine `background-color` preko string literalja "`yellow`".

Postoje i drugi način za podešavanje fiksnih vrednosti, na primer, primenom atributa direktive `ngStyle` i parova (ključ, vrednost) za svaku od osobina koju je neophodno podesiti. Navedeno je moguće realizovati na sledeći način:

```
<div [ngStyle]="{color: 'white', 'background-color': 'blue'}">  
    Koristi fiksni beli tekst na plavoj pozadini  
</div>
```

Iz priloženog listinga je moguće zaključiti da je obavljeno istovremeno podešavanje osobina: `color` i `background-color`, respektivno. Međutim, najveća snaga primene direktive `ngStyle` leži u radu sa dinamičkim HTML elementima.

PRIMENA DINAMIČKIH VREDNOSTI

Najveća snaga primene direktive `ngStyle` leži u radu sa dinamičkim HTML elementima.

Kao što je upravo konstatovano, najveća snaga primene direktive `ngStyle` leži u radu sa dinamičkim HTML elementima. U aktuelnom primeru uvodi se dodatna funkcionalnost. Postoje dva polja za unos teksta, u kojima se unose željena boja i veličina fonta, i dugme kojim se potvrđuju nova podešavanja. U priloženom primeru, ovaj listing, kao i prethodni, uzet je iz datoteke: `src/app/ng-style-example/ng-style-example.component.html`.

```
<div class="ui input">
  <input type="text" name="color" value="{{color}}" #colorinput>
</div>

<div class="ui input">
  <input type="text" name="fontSize" value="{{fontSize}}" #fontinput>
</div>

<button class="ui primary button" (click)="apply(colorinput.value,
fontinput.value)">
  Primeni podešavanja
</button>
```

Klikom na kreirano dugme menjaju se CSS osobine navedenih za boju u veličinu fonta. Ove osobine su bile inicijalno podešene sledećim kodom:

```
<div>
  <span [ngStyle]="{color: 'red'}" [style.fontSize.px]="fontSize">
    crveni tekst
  </span>
</div>
```

Važno je napomenuti da je obavezno specificiranje veličina gde je to potrebno. Na primer, nije pravilno podesiti CSS osobinu `font-size` na vrednost 12. **Ona mora biti izražena u jedinicama poput 12px ili 1.2em.** Angular obezbeđuje jednostavnu sintaksu za rešavanje navedenog problema iskazivanja veličine fonta u nekoj od podržanih jedinica:

- primenom notacije `[style.fontSize.px]` veličina fonta se iskazuje u pikselima;
- primenom notacije `[style.fontSize.em]` veličina fonta se iskazuje u `em` vrednostima (`1em = 12pt, 2 em = 24 pt...`);
- primenom notacije `[style.fontSize.%]` veličina fonta se iskazuje u procentima.

DINAMIČKO PODEŠAVANJE BOJE TEKSTA I POZADINE

Sledeća dva elementa koriste color osobinu kao ulaz za podešavanje boje teksta i pozadine.

Primer može biti dodatno proširen, sledeća dva elementa koriste `color` osobinu kao ulaz za podešavanje boje teksta i pozadine:

```
<h4 class="ui horizontal divider header">
  ngStyle sa objektnom osobinom iz promenljive
```

```
</h4>

<div>
  <span [ngStyle]="{color: color}">
    {{ color }} tekst
  </span>
</div>

<h4 class="ui horizontal divider header">
  Stil iz promenljive
</h4>

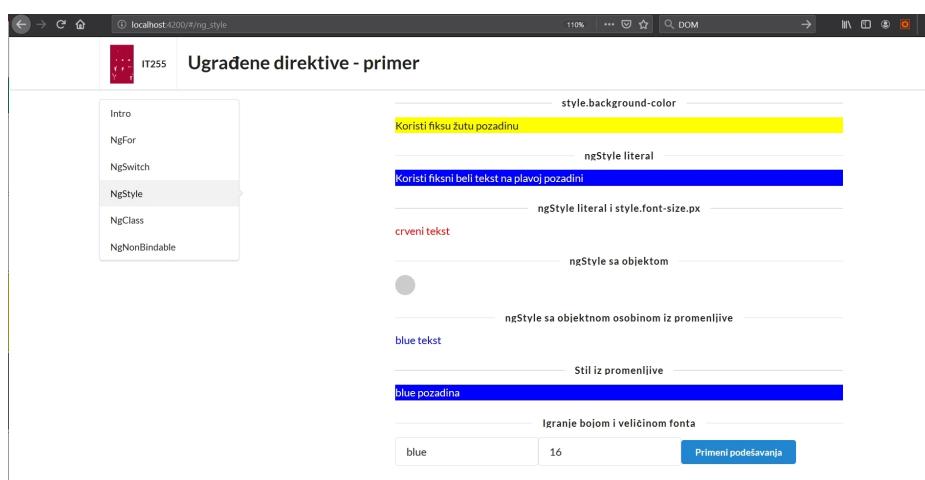
<div [style.background-color]="color"
      style="color: white;">
  {{ color }} pozadina
</div>
```

Na ovaj način, klikom na već kreirano dugme "**Primeni podešavanja**", poziva se sledeća metoda ([src/app/ng-style-example/ng-style-example.component.ts](#)) koja podešava nove vrednosti:

```
apply(color: string, fontSize:string): void {
  const priv: number = parseInt(fontSize);
  this.color = color;
  this.fontSize = priv;
}
```

Na ovaj način, boja i veličina fonta biće primenjeni na odgovarajuće elemente primenom direktive [ngStyle](#).

Sledećom slikom je prikazan tekući primer koji angažuje komponentu zaduženu za demonstraciju primene direktive [ngStyle](#).



Slika 2.1 Dinamičko podešavanje boje teksta i pozadine [izvor: autor]

▼ Poglavlje 3

Direktiva klasa u Angularu

DIREKTIVA NGCLASS

Angular direktiva `ngClass` omogućava dinamičko podešavanje i menjanje CSS klase za neki element

Angular direktiva `ngClass` je predstavljena istoimenim atributom u HTML šablonu i omogućava dinamičko podešavanje i menjanje CSS klase za dati DOM element.

Postoji više načina implementacije navedene direktive, prvi od njih može biti prosleđivanje objektnog literalja. Objekat očekuje da poseduje ključeve koji odgovaraju nazivima klase i vrednostima koje mogu biti `true` ili `false` i koje ukazuju da li će navedena CSS klasa biti primenjena ili ne.

Neka važi prepostavka da postoji `CSS` klasa pod nazivom `bordered` čiji je zadatak da doda isprekidanu crnu liniju kao ivicu oko elementa. Sledeći CSS listing, uzet iz datoteke `styles.css`, detaljnije opisuje navedenu klasu.

```
.bordered {  
    border: 1px dashed black;  
    background-color: #eee; }
```

Sada je moguće dodati dva `<div>` elementa:

- prvi primenjuje navedenu `bordered` klasu i, otuda, je uvek oivičen;
- drugu ne primenjuje ovu klasu pa nije ni istaknut na navedeni način.

Navedeni kod je sastavni deo datoteke tekućeg primera: `src/app/ng-class-example/ng-class-example.component.html`.

```
<div [ngClass]="{bordered: false}">Nikada nema ivice</div>  
<div [ngClass]="{bordered: true}">Uvek ima ivice</div>
```

```
<div [ngClass]="{bordered: false}">Nikada nema ivice</div>  
<div [ngClass]="{bordered: true}">Uvek ima ivice</div>
```

Ovaj kod će kreirati sledeći HTML izlaz:

Nikada nema ivice

Uvek ima ivice

Slika 3.1 Jednostavna primena direktive ngClass [izvor: autor]

DINAMIČKA PRIMENA DIREKTIVE NGCLASS

U praksi je mnogo značajnije znati kako je moguće ovoj direktivi dodeliti dinamičke zadatke

Međutim, **u praksi je mnogo značajnije znati kako je moguće ovoj direktivi dodeliti dinamičke zadatke.** Da bi navedeno bilo moguće, neophodno je iskoristiti neku promenljivu kao vrednost konkretnе objektne osobine. Na primer, dat je sledeći listing kao nastavak tekućeg primera (kod *opet pripada datoteci src/app/ng-class-example/ng-class-example.component.html*):

```
<div [ngClass]="{bordered: isBordered}">  
    Primena objektnog literalja. Border {{ isBordered ? "ON" : "OFF" }}  
</div>
```

Alternativno, moguće je definisati *classesObj* unutar klase komponente (u konkretnom slučaju radi se o klasi *NgClassExampleComponent* čiji se sledeći kod čuva u datoteci *src/app/ng-class-example/ng-class-example.component.ts*.

```
@Component({  
  selector: 'app-ng-class-example',  
  templateUrl: './ng-class-example.component.html'  
)  
export class NgClassExampleComponent implements OnInit {  
  isBordered: boolean;  
  classesObj: Object;  
  classList: string[];  
  
  constructor() {}  
  
  ngOnInit() {  
    this.isBordered = true;  
    this.classList = ['blue', 'round'];  
    this.toggleBorder();  
  }  
  
  toggleBorder(): void {  
    this.isBordered = !this.isBordered;  
    this.classesObj = {  
      bordered: this.isBordered  
    };  
  }  
}
```

Sada je moguće direktno koristiti objekat na sledeći način:

```
<div [ngClass]="classesObj">  
    Primena objektne varijable. Border {{ classesObj.bordered ? "ON" : "OFF" }}  
</div>
```

Nakon dodatnog proširenja primera, za sada imamo HTML šablon koji daje sledeći izlaz:

Nikada nema ivice

Uvek ima ivice

Primena objektnog literala. Border OFF

Primena objektne varijable. Border OFF

Slika 3.2 Dinamička primena direkтиve ngClass [izvor: autor]

PRIMENA LISTE KLASA

Lista sa nazivima klasa za određivanje koja će SCSS klasa biti pridružena nekom elementu.

Posebna pogodnost, koju nudi okvir *Angular*, jeste mogućnost primene liste sa nazivima klasa za određivanje koja će, konkretno, CSS klasa biti pridružena nekom elementu.

Na primer, dat je sledeći niz koji ima dva konkretna člana (literalna):

```
<div class="base" [ngClass]=["'blue', 'round']">  
    Ovde će uvek biti plava pozadina i  
    zaobljene ivice.  
</div>
```

Takođe, moguće je dodeliti niz vrednosti nekoj osobini komponente, na primer na sledeći način:

```
this.classList = ['blue', 'round'];
```

Nakon ovoga je moguće proslediti ovu osobinu u šablon na prikazivanje:

```
<div class="base" [ngClass]= "classList">  
    Ovo je {{ classList.indexOf('blue') > -1 ? "" : "NOT" }} plavo  
    i {{ classList.indexOf('round') > -1 ? "" : "NOT" }} zaobljeno  
</div>
```

U poslednjem primeru, dodata vrednosti za *[ngClass]* funkcioniše uporedo sa dodelom postojećih vrednosti za HTML atribut *class*. CSS klase koje se vezuju za konkretan element, uvek će predstavljati skup klasa obezbeđenih upotrebom standardnog HTML atributa *class* i rezultatom provere direktive *[ngClass]*. U navedenom svetu, neophodno je izvršiti fokusiranje na sledeću sliku i prvi listing na ovoj stranici.



Slika 3.3 Dobijene klase iz atributa i direkutive [izvor: autor]

Kao što je moguće primetiti, element će posedovati tri klase:

- *base* - iz HTML *class* atributa;
- *blue* - iz direktive *[ngClass]*;
- *round* - iz direktive *[ngClass]*.

DODAVANJE INTERAKCIJE

Uvođenjem GUI kontrola postiže se interaktivnost tekuće komponente.

Aktuelni primer je moguće proširiti sledećom logikom:

- dodaje se novo dugme u šablon koje će omogućiti korisniku da, nakon klika, doda ivicu postojećim stringovima u šablonu;
- dodaju se dve *checkbox* kontrole u šablon za izbor da li će pozadina izabranih stringova biti obojena i predstavljena zaobljenim ivicama, respektivno.

Uvođenjem navedenih GUI kontrola postiže se interaktivnost tekuće komponente. Kontrole su realizovane sledećim listingom koji predstavlja proširenje koda šablona tekuće komponente:

```
<button (click)="toggleBorder()">Klikni</button>

<div class="selectors">
  <div>
    <input type="checkbox"
      [checked]="classList.indexOf('blue') > -1"
      (click)="toggleClass('blue')">
    <span>Plavo</span>
  </div>

  <div>
    <input type="checkbox"
      [checked]="classList.indexOf('round') > -1"
      (click)="toggleClass('round')">
    <span>Zaobljeno</span>
  </div>
</div>
```

Da bi interakcija korisnika sa kreiranim kontrolama zaživila, neophodno je, a i vidi se iz priloženog listinga, proširiti kod klase komponente metodama:

- *toggleBorder()* - dodaje stilizovanu ivicu oko elementa nakon klika na dugme;
- *toggleClass()* - pridružuje elementu CSS klasu nakon čekiranja određene checkbox kontrole.

```
ngOnInit() {
    this.isBordered = true;
    this.classList = ['blue', 'round'];
    this.toggleBorder();
}

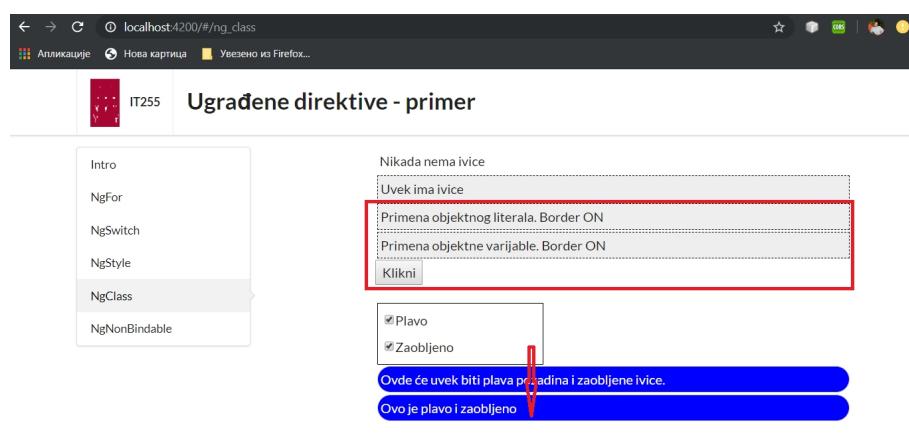
toggleBorder(): void {
    this.isBordered = !this.isBordered;
    this.classesObj = {
        bordered: this.isBordered
    };
}

toggleClass(cssClass: string): void {
    const pos: number = this.classList.indexOf(cssClass);
    if (pos > -1) {
        this.classList.splice(pos, 1);
    } else {
        this.classList.push(cssClass);
    }
}
```

DIREKTIVA NGCLASS I INTERAKCIJA - DEMO

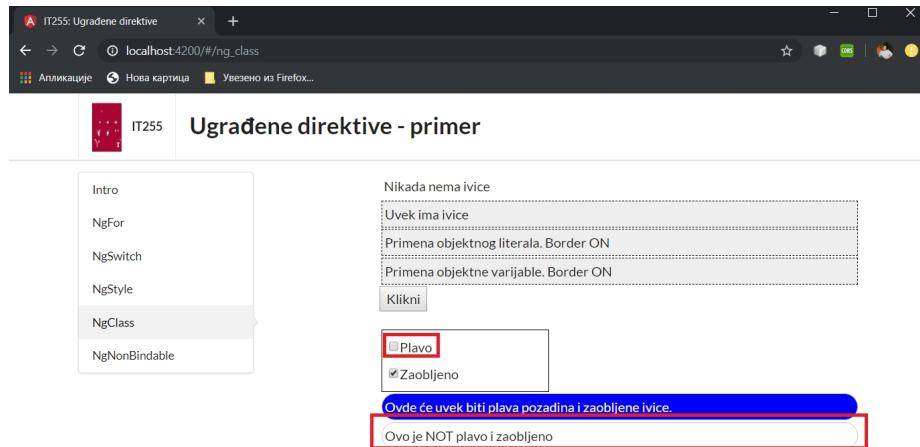
Testiranje izvršavanja kompletirane komponente.

Sledećom slikom je prikazan rezultat klika na kreirano dugme i selekcije oba kreirana *checkbox* - a.



Slika 3.4 Rezultat klika na kreirano dugme i selekcije oba kreirana checkbox - a [izvor: autor]

Za demonstraciju interaktivnosti aplikacije isključuje se jedna *checkbox* kontrola, na primer "Plavo". Rezultat izvršavanja aplikacije moguće je ilustrovati sledećom slikom. Promene su istaknute crvenom bojom.



Slika 3.5 Demonstracija interaktivnosti aplikacije isključivanjem jedne kontrole [izvor: autor]

▼ Poglavlje 4

Direktive petlji u Angularu

DIREKTIVA NGFOR

Uloga Angular direktive ngFor je obezbeđivanje ponavljanja izvesnog DOM elementa.

Uloga Angular direktive ngFor je obezbeđivanje ponavljanja izvesnog DOM elementa (ili kolekcije DOM elemenata) i prosleđivanje elementa niza za svaku od iteracija. Osnovna sintaksa direktive ngFor je data sledećim iskazom:

```
*ngFor="let item of items".
```

Iz priloženog formata direktive ngFor moguće je primetiti sledeće:

- sintaksa let item specificira varijablu HTML šablona koja prihvata svaki od elemenata niza;
- sa items je označen niz elemenata dobijen od kontrolera.

Da bi primena navedene direktive bila lakše usvojena, neophodno je obaviti dodatna proširenja aktuelnog primera. Kreira se nova komponenta ng-for-example i u njenoj kontrolerskoj klasi postoji sledeći kod kojim se deklariše niz gradova (stringova):

```
this.cities = ['Kragujevac', 'Beograd', 'Niš'];
```

Da bi korisnik mogao da vidi sadržaj kreiranog niza, neophodno je dodati malo HTML koda u šablon nove komponente:

```
<h4 class="ui horizontal divider header">
    Jednostavna lista stringova
</h4>

<div class="ui list" *ngFor="let c of cities">
    <div class="item">{{ c }}</div>
</div>
```

Rezultat izvršavanja kreiranog koda bi mogao da bude ilustrovan sledećom slikom.

Jednostavna lista stringova

Kragujevac

Beograd

Niš

Slika 4.1 Rezultat primene direktive ngFor [izvor: autor]

ITERACIJA KROZ NIZ OBJEKATA

Cilj je demonstracija mehanizma iteracije kroz niz objekata u Angularu.

Primer i izlaganje može biti prošireno u drugom smeru. Cilj je demonstracija mehanizma iteracije kroz niz objekata u *Angularu*. U kontrolerskoj klasi [src/app/ng-for-example/ng-for-example.component.ts](#) javlja se nov kod priložen sledećim listingom:

```
this.people = [
  { name: 'Vladimir', age: 45, city: 'Kragujevac' },
  { name: 'Marko', age: 24, city: 'Beograd' },
  { name: 'Nikola', age: 35, city: 'Niš' }
];
```

Da bi prikazani niz objekata mogao da bude prikazan na ekranu, neophodno je proširiti i HTML kod šablona komponente, dodavanjem sledećih linija koda:

```
<h4 class="ui horizontal divider header">
  Lista objekata
</h4>






```

Ako se pogleda linija koda 13, poslednjeg listinga, primećuje se da petlja jednostavno prolazi kroz niz *people* objekata i omogućava njihovo prikazivanje na ekranu u formi tabele. Navedeno može biti ilustrovano sledećom slikom.

Lista objekata		
Ime	Starost	Grad
Vladimir	45	Kragujevac
Marko	24	Beograd
Nikola	35	Niš

Slika 4.2 Iteracija kroz niz objekata [izvor: autor]

OBRADA UGNJEŽDENIH NIZOVA

Na jednostavan način je moguće vršiti obradu ugnježdenih (nested) nizova.

Na jednostavan način je moguće vršiti obradu ugnježdenih (*nested*) nizova. Ukoliko želimo da koristimo i prikazujemo slične podatke kao u prethodnom slučaju, ali razvrstane prema gradovima, moguće je deklarisati nov niz objekata, na sledeći način:

```
this.peopleByCity = [
  {
    city: 'Kragujevac',
    people: [
      { name: 'Lazar', age: 11 },
      { name: 'Isidora', age: 9 }
    ]
  },
  {
    city: 'Niš',
    people: [
      { name: 'Jovana', age: 25 },
      { name: 'Zoran', age: 36 }
    ]
  }
];
```

Listing pokazuje ugnježdene nizove *people* objekata unutar niza *city* objekata.

Sada, prema usvojenoj dobroj praksi, vrši se dogradnja HTML koda šablona tekuće komponente sa ciljem prikazivanja podataka iz prethodnog listinga.

```
<h4 class="ui horizontal divider header">
  Ugnježdeni podaci
</h4>

<div *ngFor="let item of peopleByCity">
  <h2 class="ui header">{{ item.city }}</h2>
```

```

<table class="ui celled table">
  <thead>
    <tr>
      <th>Ime</th>
      <th>Starost</th>
    </tr>
  </thead>
  <tr *ngFor="let p of item.people">
    <td>{{ p.name }}</td>
    <td>{{ p.age }}</td>
  </tr>
</table>
</div>

```

Problem je rešen, kao što se vidi iz priloženog listinga, ugnježdavanjem `ngFor` petlji. Spoljašnja petlja (linija koda 5) prolazi kroz listu gradova, a unutrašnja petlja (linija koda 15) za svaki od gradova prolazi kroz listu osoba. Sledećom slikom je prikazan rezultat izvršavanja prikazanog koda.

Ugnježdeni podaci	
Kragujevac	
Ime	Starost
Lazar	11
Isidora	9
Niš	
Ime	Starost
Jovana	25
Zoran	36

Slika 4.3 Obrada ugnježdenih nizova [izvor: autor]

DOBIJANJE INDEKSA

U brojnim situacijama je veoma značajno dobijanje vrednosti indeksa za tekući element u iteraciji.

Do sada je posmatrana `ngFor` petlja koja se ponašala po poznatom šablonu `for - each`. Drugim rečima, prolazila je kroz niz i omogućavala vraćanje članova niza ali nije davala podršku za rad sa indeksima niza. U brojnim situacijama je veoma značajno dobijanje vrednosti indeksa za tekući element u iteraciji.

Dobijanje indeksa moguće je obaviti na veoma jednostavan način primenom proširene sintakse direktive `ngFor`. Proširenje se ogleda na dodavanje sintakse:

```
let idx = index
```

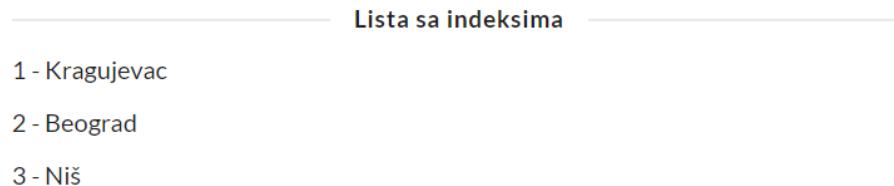
na vrednost `ngFor` direktive od koje je odvojena znakom "tačka - zarez" (;). Na ovaj način, `Angular` vrši pridruživanje vrednosti indeksa, za tekuću iteraciju petlje, promenljivoj `idx`.

Budući da je obavljena sva potrebna analiza, moguće je obaviti dodatno proširenje aktuelnog primera tako da korisnici mogu da dobiju informaciju o rednom broju grada, izvedenu na osnovu indeksa odgovarajućeg člana niza gradova. Sledećim listingom je proširen HTML kod šablonu [src/app/ng-for-example/ng-for-example.component.html](#) tekuće komponente.

```
<h4 class="ui horizontal divider header">
  Lista sa indeksima
</h4>

<div class="ui list" *ngFor="let c of cities; let num = index">
  <div class="item">{{ num+1 }} - {{ c }}</div>
</div>
```

Petlja (linija koda 5) prolazi kroz niz gradova, uzima indeks i uvećava ga za 1 (linija koda 6), jer svaka petlja broji od 0 i, konačno, omogućava prikazivanje grada sa rednim brojem pojavljivanja na ekranu. Navedeno je prikazano sledećom slikom.



Slika 4.4 Dobijanje indeksa iz ngFor direktive [izvor: autor]

✓ Poglavlje 5

Direktiva ngNonBindable

PRIMENA DIREKTIVE NGNONBINDABLE

Direktiva se koristi kada je neophodno ukazati Angularu okviru da ne prevodi ili povezuje izvesnu sekciju na stranici.

Poslednja u nizu direktiva, koje obrađuje ova lekcija, jeste [Angular direktiva ngNonBindable](#). Ova direktiva se koristi kada je neophodno ukazati Angular okviru da ne prevodi ili povezuje izvesnu sekciju na stranici.

Na primer, cilj je prikazivanje teksta putem izraza `{{ content }}` ugrađenog u posmatrani HTML šablon. U normalnim okolnostima tekst bi bio povezan sa vrednošću varijable `content` i prikazan na ekranu.

Šta ako je potrebno prikazati tekst koji je identičan navedenom izrazu: `{{ content }}`? U tom slučaju je neophodno sprečiti obradu izraza primenom direktive `ngNonBindable`.

Neka je ideja primena taga `<div>` za prikazivanje sadržaja kojeg čine:

- vrednost neke promenljive;
- leva strelica;
- izraz koji omogućava prikazivanje vrednosti navedene promenljive.

Navedeno kada se prevede u HTML kod izgleda kao u sledećem slučaju:

```
<div class='ngNonBindableDemo'>
  <span class="bordered">{{ content }}</span>
  <span class="pre" ngNonBindable>
    &larr; Ovo je {{ content }} kreirao
  </span>
</div>
```

Izraz `{{ content }}`, iz linije koda 2, biće rešen i njegova vrednost (varijabla `content`) će biti prikazana na ekranu. U slučaju linije koda 4, identičan izraz je obuhvaćen direktivom `ngNonBindable`, neće biti obrađen i kao takav će biti prikazan kao običan tekst.

Još je neophodno specificirati vrednost promenljive `content` u kontroleru komponente:

```
constructor() {
  this.content = 'Neki tekst';
}
```

Rezultat primene direktive `ngNonBindable` dat je sledećom slikom.



Slika 5.1 Primena direktive ngNonBindable [izvor: autor]

ANGULAR DIREKTIVE KROZ VIDEO MATERIJAL

Directives in Angular Applications - trajanje 25:12

Ova lekcija sadrži video materijal. Ukoliko želite da pogledate ovaj video morate da otvorite LAMS lekciju.

Kompletno uređen i proveren pokazni primer možete preuzeti iz aktivnosti Shared Resources odmah iza ovog objekta učenja.

▼ Poglavlje 6

Forme u Angular okviru

FORMA U VEB APLIKACIJI

Forma je verovatno najznačajniji aspekt svake veb aplikacije.

Forma je verovatno najznačajniji aspekt svake veb aplikacije. U velikom broju slučajeva, primenom formi je obezbeđeno prikupljanje širokog spektra podataka koje korisnik unese u aplikaciji.

Primena formi u veb aplikacijama, pa tako i Angular aplikacijama, često deluje dosta jednostavna i pravolinijska:

- kreiraju se ulazni (input) tagovi za podatke;
- korisnik popuni polja podacima;
- korisnik klikne na dugme za posleđivanje unetih podataka.

Međutim, nije sve tako jednostavno i rad sa formama može biti veoma komplikovan:

- ulazi formi su predviđeni da menjaju podatke, kako na stranici, tako i na serveru;
- promene na formama se često reflektuju i na ostatak stranice;
- često je potrebno implementirati validaciju (proveru ispravnosti) za pojedina polja forme;
- korisnički interfejs bi trebalo jasno da prikaže uputstva i greške rada sa formama, ukoliko ih ima;
- zavisna polja forme mogu da poseduju kompleksnu logiku;
- često je neophodno obezbediti testiranje forme, bez oslanjanja na DOM selektore.

Zbog svega navedenog, veb programeri često koriste Angular okvir za prevazilaženje uočenih problema kompleksnosti rada sa formama. Angular kao rešenje nudi sledeće alate:

- FormControl - enkapsulira sve ulaze sa forme i grupiše ih objekat koji se dalje obrađuje u aplikaciji;
- Validator - proverava da li obavezno polje forme popunjeno, da li ima odgovarajući format, da li je unet odgovarajući tip podataka u polje forme i slično;
- Observer - posmatra (osluškuje) sve izmene na formi i reaguje na odgovarajući način.

U nastavku lekcije, cilj je učenje kreiranja formi po principu "korak - po - korak".

✓ 6.1 FormControl i FormGroup

FORMCONTROL

FormControl je jedan od dva osnovna objekta u Angular formama.

Dva osnovna objekta u Angular formama su:

- FormControl i
- FormGroup.

FormControl predstavlja jedno polje za unos podataka i predstavlja najmanju jedinicu Angular forme. *FormControl* enkapsulira vrednost koja je uneta u polje i daje neko od sledećih stanja:

- ispravno (*valid*);
- izmenjeno (*dirty* ili *changed*);
- sadrži greške (*errors*).

Na primer, sledećim listingom je prikazano kako se koristi *FormControl* u TypeScript - u:

```
// kreira nov FormControl objekat sa vrednošću "Vlada"
let nameControl = new FormControl("Vlada");

let name = nameControl.value; // -> Nate

// sada je moguće izvršiti upit nad kontrolom za doijanje vrednosti:
nameControl.errors // -> StringMap<string, any> za greške
nameControl.dirty // -> false
nameControl.valid // -> true
// i tako dalje
```

Za građenje forme kreiran je *FormControl* objekat kojem su, nakon toga, dodeljeni meta podaci i logika. Kao i za većinu ostalih stvari, a o nekima je već bilo diskutovano, Angular obezbeđuje konkretnu klasu *FormControl* koja se povezuje za DOM sa atributom (*formControl*, u ovom slučaju). Na primer u formi može da postoji sledeći segment:

```
<!-- deo neke veće forme -->
<input type="text" [formControl]="name" />
```

Navedeni kod će u kontekstu forme kreirati novi *FormControl* objekat. Naknadno će biti više diskusije u vezi sa funkcionisanjem navedenog.

U novim verzijama Angular okvira insistira se na konstantnoj proveri tipova podataka. Otuda, program može da javlja grešku prilikom povezivanja tipa podataka sa *formControl* objektom. Da bi kompjajler ignorisao proveru, prethodni listing može biti proširen primenom pseudo metode *\$any()* na sledeći način:

```
<!-- deo neke veće forme -->
<input type="text" [formControl]="$any(name)" />
```

FORMGROUP

U praksi, forma najčešće sadrži višegrupisanih polja za unos podataka.

U praksi, **forma** najčešće sadrži više od jednog polja za unos podataka. To znači da je neophodno obezbediti upravljanje većim brojem **FormControl** objekata. Ukoliko je cilj provera ispravnosti kreirane forme bilo bi preteško vršiti iteracije preko niza **FormControl** objekata i proveravati validnost svakog od njih, pojedinačno. Angular klasa **FormGroup** rešava ovaj problem na veoma jednostavan način. Klasa obezbeđuje omotački interfejs oko kolekcije **FormControl** objekata.

Sada je moguće izvršiti kreiranje jednog **FormGroup** objekta. Neka je to urađeno na sledeći način:

```
let personInfo = new FormGroup({  
  firstName: new FormControl("Vladimir"),  
  lastName: new FormControl("Milićević"),  
  zip: new FormControl("34000")  
})
```

Klase **FormGroup** i **FormControl** poseduju zajedničku roditeljsku klasu **AbstractControl** (<https://angular.io/api/forms/AbstractControl>). To praktično znači da status i vrednost **FormGroup** može biti proverena na jednostavan način, kao što je bio slučaj sa **FormControl** objektom.

```
personInfo.value; // -> {  
  // firstName: "Vladimir",  
  // lastName: "Milićević",  
  // zip: "34000"  
  // }  
  
// sada je moguće izvršiti upit nad grupom kontrola, koje imaju osetljive  
// vrednosti u zavisnosti od vrednosti potomaka FormControl:  
personInfo.errors // -> StringMap<string, any> za greške  
personInfo.dirty // -> false  
personInfo.valid // -> true  
// i tako dalje
```

Iz priloženog listinga je moguće primetiti da je pokušano dobijanje vrednosti iz **FormGroup** objekta prijemom objekata u formi (*ključ, vrednost*). Ovo je veoma pogodno za pribavljanje kompletног skupa vrednosti iz forme bez potrebe za primenom iteracija preko pojedinačnih **FormControl** objekata.

❖ 6.2 Kreiranje prve forme

IZGLED PRVE FORME

Postoje brojni koraci koje veb programeri izvode kada kreiraju forme.

Postoje brojni koraci koje veb programeri izvode kada kreiraju forme, a dobar broj njih još uvek nije razmatran. U tu svrhu biće razvijan potpuno nov primer za lakše razumevanje problematike rada sa formama u [Angular](#) - u.

Potpuno uređeni primer možete preuzeti na kraju ovog dela lekcije, iz kategorije, Shared resources ali je preporuka da pokušate samostalno, prateći izlaganje u ovom delu lekcije, kreirate vašu aplikaciju.

Sledećom slikom je dato idejno rešenje izgleda forme na kojem će biti rađeno tokom kreiranja pratećeg primera.

The screenshot shows a light gray rectangular form with a title 'Demo forma: Sku' at the top. Below the title is a label 'SKU' followed by a white input field with a thin gray border. Inside the input field, the word 'SKU' is written in a light blue-gray color. At the bottom of the form is a dark gray rectangular button with the white text 'Pošalji' (Send).

Slika 6.1.1 Idejno rešenje izgleda forme [izvor: autor]

SKU je skraćenica za „stockkeeping unit - jedinica zaliha“. To je izraz za jedinstveni id proizvoda koji će se pratiti u zalihamama. Kada se govori o SKU-u, govori seo ljudski čitljivom ID-u predmeta.

Predložena forma je veoma jednostavna: postoji polje za unos (zajedno sa odgovarajućom labelom) i dugme za slanje podataka nakon što [forma](#) bude popunjena. Da bi navedeno bilo realizovano, neophodno je kreiranje komponente koja implementira ovu formu.

Kao i u prethodnim slučajevima, tri dela čine komponentu:

- podešavanje [@Component\(\)](#) dekoratora;
- kreiranje šablona komponente;
- implementacija funkcionalnosti putem klase komponente.

UČITAVANJE FORMSMODULE PODRŠKE

Neophodno je učitavanje odgovarajućih biblioteka u klasu obeleženu dekoratorom `NgModule`.

Sa ciljem korišćenja najnovijih biblioteka za rad sa formama neophodno je obaviti učitavanje odgovarajućih biblioteka u klasu obeleženu dekoratorom `@NgModule`.

Postoje dva načina korišćenja formi podržana Angular okvirom:

- primena `FormsModule` biblioteke;
- primena `ReactiveFormsModule` biblioteke.

Budući da će obe biblioteke biti korišćene u radu na konkretnom primeru, biće obe učitane (importovane) u navedeni modul. Da bi navedeno bilo realizovano neophodno je u datoteku centralne komponente aplikacije `app.module.ts` dodati sledeći kod:

```
import {  
    FormsModule,  
    ReactiveFormsModule  
} from '@angular/forms';  
  
// dalje idu dodatni importi  
  
@NgModule({  
    declarations: [  
        FormsDemoApp,  
        DemoFormSkuComponent,  
        // ... ove idu deklaracije  
    ],  
    imports: [  
        BrowserModule,  
        FormsModule,           // dodati ovo  
        ReactiveFormsModule // dodati ovo  
    ],  
    bootstrap: [ FormsDemoApp ]  
})  
class AppModule {}
```

Slika 6.1.2 Učitavanje FormsModule podrške [izvor: autor]

Listingom prikazanim prethodnom slikom postignuta je mogućnost korišćenja direktiva formi u HTML pogledima aplikacije, koji će tek biti kreirani kako budu dodatne komponente uvođene u aplikaciju. `FormsModule` obezbeđuje šablone bazirane na direktivama na poput:

- `ngModel` i
- `NgForm`.

`ReactiveFormsModule`, sa druge strane, obezbeđuje šablone bazirane na direktivama na poput:

- `FormControl` i
- `ngFormGroup`.

Do sada nije bilo govora o tome šta koja direktiva radi ili kako se koristi ali će biti uskoro. Za sada, dovoljno je da se zna da dodavanjem `FormsModule` i `ReactiveFormsModule` u `NgModule` obezbeđeno je korišćenje direktiva formi u šablonima ili umetanje odgovarajućih provajdera direktiva u komponente.

JEDNOSTAVNA FORMA: @COMPONENT DECORATOR

Kreiran je nov tag app-demo-form-sku koji će omogućiti prikazivanje HTML sadržaja šablona.

Sada kada je napravljen detalja uvod, moguće je posvetiti posebnu pažnju kreiranju forme koja je ilustrovana slikom 1. Za početak, neophodno je obaviti kreiranje odgovarajuće komponente: [src/app/demo-form-sku/demo-form-sku.component.ts](#). Sledećim listingom je dat njen dekorator @Component sa definisanim selektorom i šablonom.

```
@Component({  
  selector: 'app-demo-form-sku',  
  templateUrl: './demo-form-sku.component.html',  
  // ... ostatak će kasnije biti definisan
```

Ako se pogleda kod, bitno je još jednom napomenuti da je kreiran nov HTML tag [app-demo-form-sku](#) koji će omogućiti prikazivanje HTML sadržaja šablona [demo-form-sku.component.html](#) navođenjem:

```
<app-demo-form-sku></app-demo-form-sku>
```

Sledeći korak predstavlja upravo definisanje šablona, odnosno dela aplikacije koji će biti vidljiv krajnjem korisniku. Kod šablona se nalazi u datoteci [src/app/demo-form-sku/demo-form-sku.component.html](#) i priložen je sledećim listingom.

```
<div class="ui raised segment">  
  <h2 class="ui header">Demo forma: Sku</h2>  
  <form #f="ngForm"  
    (ngSubmit)="onSubmit(f.value)"  
    class="ui form">  
  
    <div class="field">  
      <label for="skuInput">SKU</label>  
      <input type="text"  
        id="skuInput"  
        placeholder="SKU"  
        name="sku" ngModel>  
    </div>  
  
    <button type="submit" class="ui button">Pošalji</button>  
  </form>  
</div>
```

U narednom izlaganju sledi detaljna analiza priloženog koda šablona.

PRIMENA FORM I NGFORM

Uključena biblioteka FormsModule omogućava primenu direktive ngForm

Kako izučavanje problematike odmiče, tako ona postaje sve zanimljivija. Iz razloga što je uključena biblioteka **FormsModule**, omogućena je primena direktive **NgForm**. Takođe, trebalo bi imati na umu, da kada se direktiva učini dostupnom u pogledu, ona će se vezati za bilo koji element koji odgovara selektoru.

Primenom direktive **NgForm** omogućene su neke veoma korisne stvari koje nisu očigledne: uključuje tag forme u vlastiti selektor (umesto zahteva za eksplisitnim dodavanjem **NgForm** kao atributa). Ovo praktično znači da ukoliko se obavi dodavanje biblioteke **FormsModule**, direktiva **NgForm** će automatski biti povezana sa svakim **<form>** tagom koji se nalazi u pogledu. Ovo je veoma korisno, ali može biti i zbumujuće jer se dešava u pozadini.

Postoje dva ključna detalja funkcionalnosti direktive **NgForm**:

- kreiranje **FormGroup** objekta pod nazivom **ngForm**;
- kreiranje (**ngSubmit**) izlaza.

Primena navedenog u okviru **<form>** taga data je sledećim listingom:

```
<form #f="ngForm"
      (ngSubmit)="onSubmit(f.value)"
```

Posebno je neophodno, u ovom trenutnu, dati značenje sintakse: **#f="ngForm"**. Na ovaj način kreirana je lokalna promenljiva **f** koja će biti korišćena u pogledu i predstavlja pseudonim za **ngForm**. Dalje, **ngForm** objekat je tipa **FormGroup** i dolazi iz direktive **NgForm**. Konačno, promenljiva **f** se koristi u pogledu kao **FormGroup** (videti liniju koda 2 (**ngSubmit**) izlaz).

Povezivanje akcije **ngSubmit**, forme koja se kreira, obavlja se primenom sintakse:

```
(ngSubmit)="onSubmit(f.value)"
```

Ovde je značajno par stvari:

- **(ngSubmit)** - dobija se iz **NgForm**;
- **onSubmit()** - funkcija koja će biti implementirana u klasi komponente;
- **f.value** - **f** je prethodno definisan objekat tipa **FormGroup** pri čemu atribut **value** vraća vrednost u formi (ključ / vrednost) za ovaj **FormGroup**.

Kada se sve ovo poveže u celinu može biti interpretirano na sledeći način: **kada se popuni i potvrди forma (klikom na dugme), poziva se funkcija onSubmit() klase komponente kojoj se, kao argument, prosleđuje vrednost forme.**

PRIEMENA INPUT & NGMODEL

input tag poseduje detalje koje je neophodno razumenti pre doticanja NgModel direktive.

Korišćeni input tag poseduje par zanačajnih stvari koje su vredne diskusije pre doticanja **NgModel** direktive.

```
<form #f="ngForm"
      (ngSubmit)="onSubmit(f.value)"
      class="ui form">

  <div class="field">
    <label for="skuInput">SKU</label>
    <input type="text"
           id="skuInput"
           placeholder="SKU"
           name="sku" ngModel>
  </div>
  <button type="submit" class="ui button">Pošalji</button>
</form>
```

- iskazi `class="ui form"` i `class="field"` su opcionalni i definišu CSS klase koje se koriste za stilizovanje forme i kontrola koje se prikazuju u pogledu;
- atribut labele `for` mora da se podudara sa atributom `id` polja za unos podataka po `W3C` standardu. Jednostavno, labela se odnosi na input polje;
- iskaz `placeholder="SKU"` predstavlja napomenu za korisnika koja postoji kada je input polje ne popunjeno.

Direktiva NgModel specificira selektor za ngModel. To znači da je moguće primeniti je u input tagu dodavanjem sledeće vrste atributa:

`ngModel="bilo šta".`

U ovom slučaju (pogledati priloženi kod), specificiran je `ngModel` bez vrednosti atributa.

Postoji nekoliko različitih načina da se specificira `ngModel` u šablonima i ovo je prvi od njih. Kada se koristi `ngModel` bez vrednosti atributa, specificira se sledeće:

- jednosmerno povezivanje podataka (one-way data binding);
- kreiranje `FormControl` objekta sa nazivom `sku` (zbog atributa `name` `input` taga).

`NgModel` kreira nov `FormControl` objekat koji je automatski dodat roditelju `FormGroup` (u ovom slučaju, na formu) i potom povezuje `DOM` element sa kreiranim `FormControl` objektom. To znači, podešena je asocijacija između `input` taga iz pogleda sa objektom tipa `FormControl`. A podudaranje u asocijaciji je obezbeđeno preko naziva (atribut `name`), a to je u ovom slučaju "sku".

JEDNOSTAVNA FORMA: DEFINICIJA KLASE KOMPONENTE

Ovaj deo primera je zaokružen definicijom klase odgovarajuće komponente.

Ovaj deo primera mora da bude zaokružen definicijom klase odgovarajuće komponente. U datoteku `src/app/demo-form-sku/demo-form-sku.component.tsneophodno` je dodati kod koji odgovara sledećem listingu:

```
export class DemoFormSkuComponent implements OnInit {  
  
    constructor() { }  
  
    ngOnInit() {  
    }  
  
    onSubmit(form: any): void {  
        console.log('Poslali ste vrednost:', form);  
    }  
  
}
```

Klasa definiše jednu metodu `onSubmit()` koja će biti izvršena kada korisnik klikne na dugme sa forme. Metoda će u konzoli veb pregledača prikazati podatke unete putem forme.

JEDNOSTAVNA FORMA: DEMO PRIMERA

Kod je objedinjen i može da bude isprobан.

Za lakše snalaženje studenata, prilikom izrade ovog primera, sledi objedinjavanje koda komponente na jednom mestu.

Listing klase komponente:

```
import { Component, OnInit } from '@angular/core';  
  
@Component({  
    selector: 'app-demo-form-sku',  
    templateUrl: './demo-form-sku.component.html',  
    styles: []  
})  
export class DemoFormSkuComponent implements OnInit {  
  
    constructor() { }  
  
    ngOnInit() {  
    }  
  
    onSubmit(form: any): void {  
        console.log('Poslali ste vrednost:', form);  
    }  
  
}
```

Listing šablona komponente:

```
<div class="ui raised segment">  
    <h2 class="ui header">Demo forma: Sku</h2>  
    <form #f="ngForm"  
          (ngSubmit)="onSubmit(f.value)"
```

```

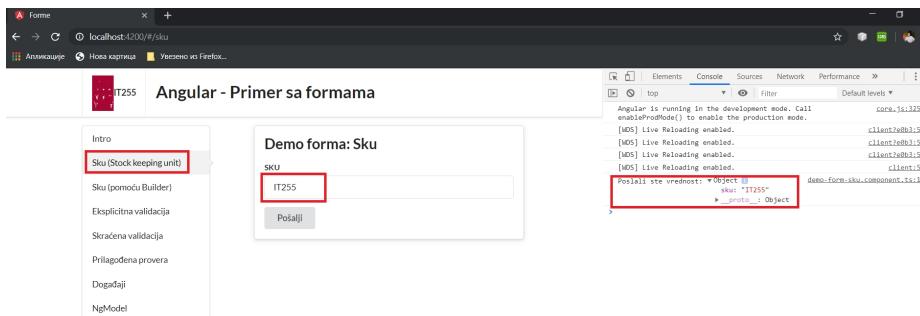
class="ui form">

<div class="field">
    <label for="skuInput">SKU</label>
    <input type="text"
        id="skuInput"
        placeholder="SKU"
        name="sku" ngModel>
</div>

    <button type="submit" class="ui button">Pošalji</button>
</form>
</div>

```

Izgled ekrana generisanog izvršavanjem koda komponente:



Slika 6.1.3 Jednostavna forma: demo primera [izvor: autor]

▼ 6.3 Korišćenje klase FormBuilder

POMOĆNA KLASA FORMBUILDER

Građenje FormControl i FormGroup primenom ngForm je pogodno ali ne dozvoljava veća podešavanja.

Implicitno građenje FormControl i FormGroup primenom ngForm je veoma pogodno ali ne dozvoljava veća podešavanja. Fleksibilniji i opšti način za kreiranje formi u Angular - u predstavlja primena FormBuilder - a.

FormBuilder je pomoćna klasa koja pomaže prilikom kreiranja formi. Kao što je do sad naučeno, forme su napravljene od *FormControl* i *FormGroup* objekata, a *FormBuilder* pomaže prilikom njihovog kreiranja.

Sada će biti diskusija preneta na aktuelni primer u kojem će biti kreirana forma primenom klase *FormBuilder*. Posebno je potrebno obratiti pažnju na sledeće stvari:

- kako se koristi *FormBuilder* u klasi komponente;
- kako se kreira prilagođen *FormGroup* na formi u pogledu.

REAKTIVNE FORME SA FORMBUILDER IMPLEMENTACIJOM

Tokom umetanja zavisnosti, objekat klase `FormBuilder` će biti kreiran i pridružen varijabli.

Primer se dalje proširuje kreiranjem potpuno nove komponente. U ovoj komponenti će biti korišćene direktive `formGroup` i `formControl`, a to znači da je neophodno uključiti odgovarajuće klase. To je prikazano sledećim listingom (sadržaj datoteke: `src/app/demo-form-sku-with-builder/demo-form-sku-with-builder.component.ts`):

```
import { Component, OnInit } from '@angular/core';
import {
  FormBuilder,
  FormGroup
} from '@angular/forms';
```

Umetanje (*injection*) objekta tipa `FormBuilder` obavlja se kreiranjem argumenta u konstruktoru klase nove komponente.

```
import { Component, OnInit } from '@angular/core';
import {
  FormBuilder,
  FormGroup
} from '@angular/forms';

@Component({
  selector: 'app-demo-form-sku-with-builder',
  templateUrl: './demo-form-sku-with-builder.component.html',
  styles: []
})
export class DemoFormSkuWithBuilderComponent implements OnInit {
  myForm: FormGroup;

  constructor(fb: FormBuilder) {
    this.myForm = fb.group({
      'sku': ['ABC123']
    });
  }

  ngOnInit() {}

  onSubmit(value: string): void {
    console.log('Poslali ste vrednost: ', value);
  }
}
```

Tokom *umetanja zavisnosti*, objekat klase `FormBuilder` će biti kreiran i pridružen kreiranoj promenljivoj `fb` (u konstruktoru).

Nad ovim objektom će biti korišćene dve glavne funkcije:

- *control* - kreira nov *FormControl*;
- *group* - kreira nov *FormGroup*.

Dalje, moguće je primetiti da je podešena nova objektna promenljiva *myForm* klase komponente kao tip *FormGroup*. Objekat, navedenog tipa podataka, kreira se pozivom *fb.group()* (linija koda 16). Funkcija *group()* uzima objekat parova (ključ, vrednost) koji specificiraju *FormControl*s objekte u ovoj grupi.

U ovom slučaju, podešena je kontrola sa nazivom "*sku*" i njena vrednost je `["ABC123"]` - ovo ukazuje da je podrazumevana vrednost za kontrolu "ABC123". Ovde je, na osnovu zagrade, moguće primetiti da se radi o nizu, to je zbog toga što će naknadno dodatne konfiguracione opcije biti dodate.

Sada, pošto je kreiran *myForm* objekat, on mora da bude iskorišćen u pogledu komponente.

UPOTREBA MYFORM OBJEKTA U ŠABLONU KOMPONENTE

Ideja je da se sadržaj taga form izmeni tako da koristi myForm objekat.

Ako se pogleda početni primer ovde je neophodno uvesti neka odstupanja. Ideja je da se sadržaj taga *<form>* izmeni tako da koristi myForm objekat kreiran u klasi nove komponente. U prošlom izlaganju je konstatovano da je *ngForm* direktiva automatski bila primenjena kada se koristi *FormsModule*. Takođe, *ngForm* kreira vlastitu *FormGroup*.

U ovom slučaju upotreba spoljašnje implementacije *FormGroup* klase biće zamenjena upotrebom objektne promenljive *myForm*, koja je kreirana pomoću *FormBuilder* objekta.

Angular nudi dodatnu direktivu koja se koristi kada je na raspolaganju postojeća *FormGroup* promenljiva. Direktiva poseduje naziv *formGroup* i koristi se na sledeći način:

```
<form [formGroup]="myForm"  
       (ngSubmit)="onSubmit(myForm.value)"
```

Na ovaj način je ukazano Angular okviru da se promenljiva *myForm* koristi kao *FormGroup* na ovoj formi. Takođe, bilo je neophodno promeniti poziv *onSubmit()* tako da koristi promenljivu *myForm* umesto prethodne *f*. To je bitno iz razloga što promenljiva *myForm* sada poseduje urađena podešavanja i vrednosti.

Još jednu stvar je neophodno obaviti da bi nov primer postao funkcionalan. Potrebno je obaviti povezivanje *FormControl* objekta sa input tagom. Sada je moguće prisetiti se da direktiva *ngControl* kreira nov *FormControl* objekat i povezuje ga sa roditeljskom klasom *FormGroup*. U ovom slučaju je upotrebljen *FormBuilder* za kreiranje vlastitog *FormControl* objekta.

Dakle, kada je cilj povezivanje postojećeg *FormControl* objekta koristi se direktiva *formControl*:

```
<div class="field">  
    <label for="skuInput">SKU</label>  
    <input type="text"
```

```
        id="skuInput"
        placeholder="SKU"
        [FormControl]="$any(myForm.controls['sku'])">
    </div>
```

CELOKUPAN LISTINGKLASE FORMBUILDER

Kod je proširen, objedinjen i može da bude isprobani.

Za lakše snalaženje studenata, prilikom izrade ovog primera, sledi objedinjavanje koda nove komponente na jednom mestu.

Listing klase komponente:

```
import { Component, OnInit } from '@angular/core';
import {
  FormBuilder,
  FormGroup
} from '@angular/forms';

@Component({
  selector: 'app-demo-form-sku-with-builder',
  templateUrl: './demo-form-sku-with-builder.component.html',
  styles: []
})
export class DemoFormSkuWithBuilderComponent implements OnInit {
  myForm: FormGroup;

  constructor(fb: FormBuilder) {
    this.myForm = fb.group({
      'sku': ['ABC123']
    });
  }

  ngOnInit() {}

  onSubmit(value: string): void {
    console.log('Poslali ste vrednost: ', value);
  }
}
```

Listing šablonu komponente:

```
<div class="ui raised segment">
  <h2 class="ui header">Demo Form: Sku koristeći Builder</h2>
  <form [FormGroup]="myForm"
        (ngSubmit)="onSubmit(myForm.value)"
        class="ui form">
```

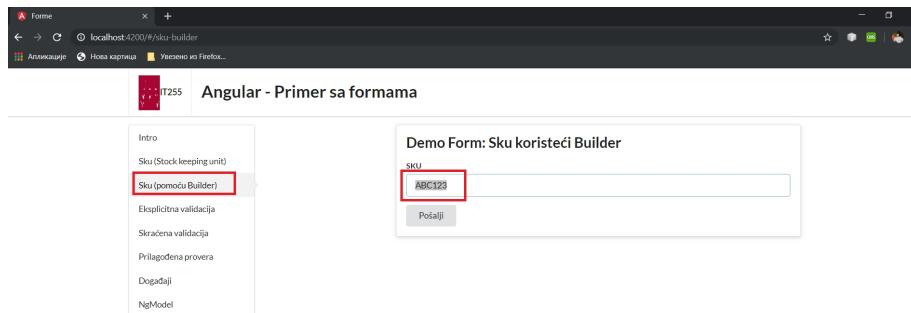
```
<div class="field">
  <label for="skuInput">SKU</label>
  <input type="text"
    id="skuInput"
    placeholder="SKU"
    [FormControl]="$any(myForm.controls['sku'])">
</div>

<button type="submit" class="ui button">Pošalji</button>
</form>
</div>
```

DEMO KORIŠĆENJA KLASE FORMBUILDER

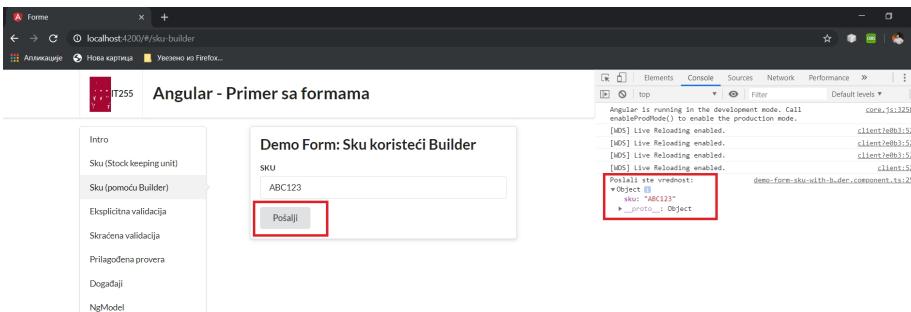
Izgled ekrana nastao kao rezultat izvršavanja koda nove komponente.

Pokazni primer se ponovo prevodi, iz menija se bira opcija koja odgovara novom primeru i na stanicu se pojavljuje forma koja u svojoj kontroli ima podrazumevanu vrednost "ABC123" (slika 1)



Slika 6.2.1 Forma koja u svojoj kontroli ima podrazumevanu vrednost [izvor: autor]

Klik na dugme forme poziva metodu `onSubmit()` komponente, a rezultat se vidi na sledećoj slici.



Slika 6.2.2 Izvršavanje metode komponente nakon klika za potvrdu forme [izvor: autor]

❖ 6.4 Validacija

VALIDATOR

Često se dešava da korisnici unesu podatke u formu koji nisu u traženom formatu.

Iz prethodnog izlaganja studenti su stekli osnovnu predstavu u vezi sa kreiranjem [Angular](#) formi, prikupljanjem i slanjem podataka iz forme i, konačno njihovom jednostavnom obradom (prikazivanje u konzoli pregledača). Međutim, u velikom broju slučajeva nije sve tako jednostavno. Često se dešava da korisnici unesu podatke u formu koji nisu u traženom formatu. Da bi bilo sprečeno da ovakav podatak ode na obradu koriste se validatori. [Validator](#) je funkcija koja obrađuje [FormControl](#) ili kolekciju kontrola i obezbeđena je preko modula [Validators](#).

Najjednostavniji validator je [Validators.required](#) kojim se zahteva da konkretno polje za unos podataka mora da bude popunjeno. U suprotnom, [FormControl](#) objekat će se smatrati nevalidnim (*invalid*).

Da bi u Angular veb aplikaciji validator mogao da bude upotrebljen, neophodno je uraditi sledeće:

1. pridružiti validator [FormControl](#) objektu;
2. proveriti status validatora u pogledu i preuzeti određenu akciju.

Pridruživanje validatora konkretnoj kontroli ([FormControl](#) objektu) obavlja se jednostavno njegovim prosleđivanjem u obliku drugog argumenta konstruktora [FormControl](#).

```
let control = new FormControl('sku', Validators.required);
```

U slučaju aktuelnog primera, budući da se sada koristi [FormBuilder](#) biće upotrebljena sledeća sintaksa:

```
constructor(fb: FormBuilder) {
  this.myForm = fb.group({
    'sku': ['', Validators.required]
  });

  this.sku = this.myForm.controls['sku'];
}
```

U nastavku, neophodno je primeniti validaciju u pogledu komponente. Postoje dva načina da se pristupi vrednosti validacije u HTML pogledima:

1. moguće je eksplicitno pridružiti kontrolu *sku* objektnoj promenljivoj klase komponente koju je moguće više puta upotrebiti i daje jednostavna pristup [FormControl](#) objektima u šablonima (pogledima);

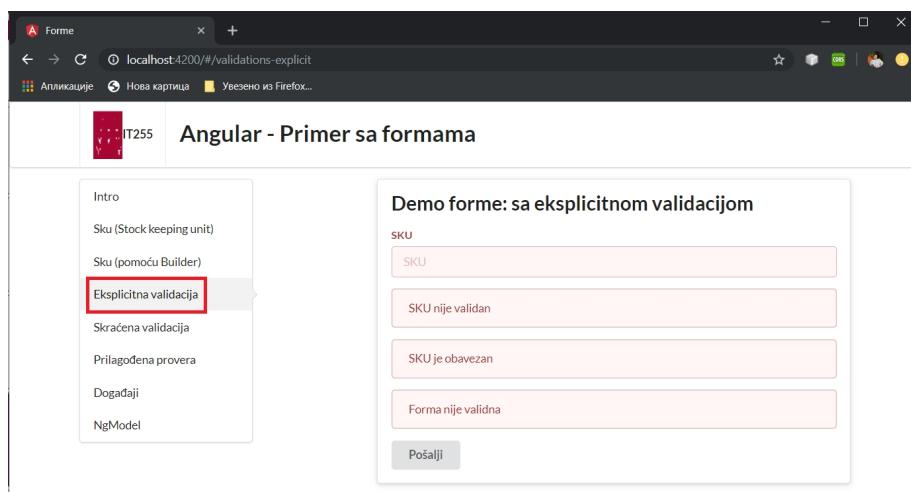
2. *FormControl* objekat *sku* moguće je potražiti iz varijable *myForm* unutar pogleda. To zahteva manje rada sa klasom komponente, ali je kompleksnije kada je druga strana komponente u pitanju - pogled.

Da bi navedeni scenariji bili jasniji, neophodno je obaviti njihovu pojedinačnu analizu u narednom izlaganju.

EKSPLICITNO PODEŠAVANJE KONTROLE KAO OBJEKTNE PROMENLJIVE

Najfleksibilniji način rada sa pojedinačnim kontrolama je njihovo podešavanje preko varijabli.

Sledećom slikom je prikazan izgled forme sa primenjenom validacijom.



Slika 6.3.1 Izgled forme sa primenjenom validacijom [izvor: autor]

Najfleksibilniji način rada sa pojedinačnim kontrolama forme u šablonima jeste podešavanje svakog *FormControl* objekta kao objektne promenljive klase odgovarajuće komponente. Poznato polje za unos podataka "*sku*", u klasi komponente, može biti podešeno na sledeći način:

```
export class DemoFormWithValidationsExplicitComponent {  
    myForm: FormGroup;  
    sku: AbstractControl;  
  
    constructor(fb: FormBuilder) {  
        this.myForm = fb.group({  
            'sku': ['', Validators.required]  
        });  
  
        this.sku = this.myForm.controls['sku'];  
    }  
  
    onSubmit(value: string): void {  
        console.log('you submitted value: ', value);  
    }  
}
```

Slika 6.3.2 Podešavanje FormControl objekta kao objektne promenljive [izvor: autor]

U nastavku sedi kraća analiza priloženog koda.

ANALIZA KODA KLASE KOMPONENTE

Moguće referencirati promenljivu klase bilo gde u pogledu komponente.

Ukoliko se pogleda kod koji je priložen prethodnom slikom, moguće je primetiti sledeće:

1. specificirana je promenljiva `sku` tipa `AbstractControl`;
2. `this.sku` dobija vrednost nakon kreiranja `myForm` preko `FormBuilder-a`.

Ovo je veoma značajno iz razloga što je moguće referencirati `sku` bilo gde u pogledu komponente. Nedostatak ovog pristupa bi bila neophodnost podešavanja objektne promenljive za svako polje forme. Za složene velike forme ovo može biti veoma naporno i kompleksno.

DODATNI NAČINI VALIDACIJE

Proširenje problematike validacije dodatnim načinima primene validatora u pogledima.

Sada kada je postavljen validator na polje `sku`, moguće je obaviti proširenje ove problematike dodatnim načinima primene validatora u pogledima:

1. provera ispravnosti forme u celini i prikazivanje poruke;
2. provera ispravnosti pojedinačnih polja forme i prikazivanje poruke;

3. provera ispravnosti pojedinačnih polja forme i njihovo bojenje crvenom bojom u slučaju greške;
4. provera ispravnosti pojedinačnih polja forme za određeni zahtev i prikazivanje poruke.

PORUKE FORME I POLJA

Greške prilikom unosa podataka u formu ili pojedinačna polja prikazuju se porukama validacije.

Greške prilikom unosa podataka u formu ili pojedinačna polja prikazuju se porukama validacije.

Poruke forme

Dat je sledeći deo listinga šablona komponente: [src/app/demo-form-with-validations-explicit/demo-form-with-validations-explicit.component.html](#):

```
<div *ngIf="!myForm.valid"
      class="ui error message">Forma nije validna</div>
```

Ovde je važno zapamtiti, myForm je objekat tipa **FormGroup** koji je validan samo ako su i svi potomci - **FormControl** objekti, takođe, validni.

Poruke polja

Takođe, moguće je prikazati i poruku validacije sa pojedinačna polja u slučaju da njihovi **FormControl** objekti nisu validni. U istu datoteku, kao u prethodnom slučaju, moguće je dodati sledeći kod:

```
<div *ngIf="!sku.valid"
      class="ui error message">SKU nije validan</div>
```

BOJENJE POLJA

Za isticanje grešaka unosa podataka u polja forme moguće je koristiti i specijalne CSS klase.

Bojenje polja forme

Za isticanje grešaka unosa podataka u polja forme moguće je koristiti i specijalne CSS klase. U konkretnom primeru, dostupno je u urađenom primeru na kraju ovog objekta učenja, koristi se CSS klasa **error**, koja pripada CSS okviru **Semantic UI**. Ova klasa, kada je pridružena **input** tagu, u slučaju greške prikazaće ovaj tag obojen u crveno.

```
<div class="field"
      [class.error]="!sku.valid && sku.touched">
```

Ovde je moguće primetiti da postoje dva uslova za podešavanje **.error** klase:

- proverava se tačnost izraza `!sku.valid`;
- proverava se tačnost izraza `sku.touched`.

Ovde je ideja da se greška ispoljava samo ako je korisnik pokušao unos u polje forme i ako to nije uradio na ispravan način.

Ovo je moguće isprobati tako se unosi neki podatak u polje za unos, a potom se odmah obriše. Rezultat će biti kao na sledećoj slici.

Demo forme: sa eksplisitnom validacijom

SKU

SKU

SKU nije validan

SKU je obavezan

Forma nije validna

Pošalji

Slika 6.3.3 Obojena polja zbog grešaka unosa [izvor: autor]

SPECIFIČNA VALIDACIJA

Polje forme može da bude nevalidno iz različitih razloga.

Polje forme može da bude nevalidno iz brojnih razloga. Često je potrebno prikazati različite poruke u zavisnosti od razloga zbog kojeg je "pala" validacija.

Za demonstraciju specifičnog "pada" validacije, moguće je upotrebiti metodu `hasError`. Nastavlja se rad na istom primeru i u datoteku [src/app/demo-form-with-validations-explicit/demo-form-with-validations-explicit.component.html](#) se dodaje sledeći kod:

```
<div *ngIf="sku.hasError('required')"  
      class="ui error message">SKU je obavezan</div>
```

Metoda `hasError` je definisana istovremeno za `FormControl` i `FormGroup`, a to znači da je primenjiva na pojedinačne kontrole ali i na formu u celini. Imajući ovo u vidu, moguće je imati i sledeći kod:

```
<div *ngIf="myForm.hasError('required', 'sku')"  
      class="error">SKU je obavezan</div>
```

VALIDACIJA KROZ CELOKUPAN LISTING

Kod je proširen validacijom, objedinjen i može da bude isprobani.

Sledi celokupan listing klase komponente koji se nalazi u datoteci [src/app/demo-form-with-validations-explicit/demo-form-with-validations-explicit.component.ts](#).

```
import { Component } from '@angular/core';
import {
  FormBuilder,
  FormGroup,
  Validators,
  AbstractControl
} from '@angular/forms';

@Component({
  selector: 'app-demo-form-with-validations-explicit',
  templateUrl: './demo-form-with-validations-explicit.component.html',
  styles: []
})
export class DemoFormWithValidationsExplicitComponent {
  myForm: FormGroup;
  sku: AbstractControl;

  constructor(fb: FormBuilder) {
    this.myForm = fb.group({
      'sku': ['', Validators.required]
    });

    this.sku = this.myForm.controls['sku'];
  }

  onSubmit(value: string): void {
    console.log('Poslali ste vrednost: ', value);
  }
}
```

Sledi celokupan listing pogleda komponente koji se nalazi u datoteci [src/app/demo-form-with-validations-explicit/demo-form-with-validations-explicit.component.html](#).

```
<div class="ui raised segment">
  <h2 class="ui header">Demo forme: sa eksplisitnom validacijom</h2>
  <form [formGroup]="myForm"
        (ngSubmit)="onSubmit(myForm.value)"
        class="ui form"
        [class.error]="!myForm.valid && myForm.touched">

    <div class="field"
        [class.error]="!sku.valid && sku.touched">
      <label for="skuInput">SKU</label>
```

```

<input type="text"
       id="skuInput"
       placeholder="SKU"
       [FormControl]="$any(sku)">
<div *ngIf="!sku.valid"
      class="ui error message">SKU nije validan</div>
<div *ngIf="sku.hasError('required')"
      class="ui error message">SKU je obavezan</div>
</div>

<div *ngIf="!myForm.valid"
      class="ui error message">Forma nije validna</div>

<button type="submit" class="ui button">Pošalji</button>
</form>
</div>

```

UKLANJANJE OBJEKTNE PROMENLJIVE

Primenom osobine myForm.controls je moguće obaviti referenciranje FormControl objekata.

U prethodnom primeru podešena je objektna promenljiva `sku` sa tipom podataka `AbstractControl`. U praksi se često izbegava kreiranje ovakvih objekata i postavlja se pitanje kako je moguće obaviti referenciranje `FormControl` objekata bez postojanja odgovarajućih objektnih promenljivih.

Odgovor leži u primeni osobine `myForm.controls`, na sledeći način:

```

<label for="skuInput">SKU</label>
<input type="text"
       id="skuInput"
       placeholder="SKU"
       [FormControl]="$any(myForm.controls['sku'])">
<div *ngIf="!myForm.controls['sku'].valid"
      class="ui error message">SKU nije validan</div>
<div *ngIf="myForm.controls['sku'].hasError('required')"
      class="ui error message">SKU je obavezan</div>
</div>

```

Ovaj pristup se često naziva skraćenom validacijom (*shorthanded validation*). Omogućio je pristup `sku` kontroli bez uslovljavanja za dodavanjem njoj odgovarajuće varijable u klasu komponente.

```

export class DemoFormWithValidationsShorthandComponent implements OnInit {
  myForm: FormGroup;

  ngOnInit() {
  }
}

```

```
constructor(fb: FormBuilder) {
  this.myForm = fb.group({
    'sku': ['', Validators.required]
  });
}

onSubmit(value: any): void {
  console.log('Poslali ste vrednost:', value.sku);
}

}
```

Za demonstraciju ovog pristupa, u projektu je kreirana nova komponenta `demo-form-with-validations-shorthand`. Kompletan kod, klase i šablona komponente je dostupan na kraju objekta učenja u sekciji **Shared Resources**.

PRILAGOĐENA VALIDACIJA

Često je potrebno neko nestandardno pravilo validacije.

Još jedan slučaj se veoma često javlja kada veb programeri kreiraju forme u svojim veb aplikacijama - definiše se neko nestandardno pravilo validacije. Ovde je cilj da se upravo pokaže kako se rukuje ovakvim pravilima i kako se ona koriste prilikom provere ispravnosti forme i njenih polja.

Da bi bilo moguće sagledavanje implementacije samih validatora, prilaže se definicija najjednostavnijeg validatora `Validators.required` iz izvornog Angular koda:

```
export class Validators {
  static required(c: FormControl): StringMap<string, boolean> {
    return isBlank(c.value) || c.value == "" ? {"required": true} : null;
  }
}
```

Iz priloženog listinga se vidi da metoda validatora, kao argument, uzima objekat `FormControl` (kontrolu), a kao rezultat vraća objekat tipa `StringMap<string, boolean>` gde stringu odgovara kod greške, a logička vrednost je `true` ukoliko je validacija "pala".

PISANJE VLASTITOG VALIDATORA

Uvodi se novo pravilo za validaciju unosa u kontrolu sku.

Uvodi se novo pravilo za validaciju unosa u kontrolu **sku**: `sku` string mora sa počne sa "123". Da bi bila obezbeđena provera, po novom kriterijumu, neophodno je da programer napiše novi - prilagođeni validator. Sledi njegov listing:

```
function skuValidator(control: FormControl): { [s: string]: boolean } {
  if (!control.value.match(/^123/)) {
```

```
    return {invalidSku: true};  
}  
}
```

Ovaj validator će vratiti kod greške `invalidSku` ukoliko ulaz (`control.value`) ne počinje stringom "123".

Za demonstraciju ovog pristupa, u projektu je kreirana nova komponenta `demo-form-with-custom-validation`. Kompletan kod, klase i šabloni komponente je dostupan na kraju objekta učenja u sekciji Shared Resources.

PRIDRUŽIVANJE VALIDATORA FORMCONTROL OBJEKTU

Novi problem - već postoji validator za kontrolu sku i kako je moguće dodati još jedan

Ako se malo detaljnije analizira trenutno stanje pokaznog primera moguće je uočiti jedan problem - već postoji validator za kontrolu `sku` i kako je moguće dodati još jedan?

Rešenje predstavlja primena metode `Validators.compose()`:

```
constructor(fb: FormBuilder) {  
  this.myForm = fb.group({  
    'sku': ['', Validators.compose([  
      Validators.required, skuValidator])]  
  });  
  
  this.sku = this.myForm.controls['sku'];  
}
```

Metoda `Validators.compose()` kao ulaz uzima niz validatora koji se primenjuju na neki `FormControl` objekat. Ovaj objekat nije validan ukoliko nije uspešno obavljena validacija po svim validatorima.

Sada je moguće primeniti kreirani validator u pogledu komponente:

```
<div *ngIf="sku.hasError('invalidSku')"  
      class="ui error message">SKU mora da počne sa <span>123</span></div>
```

Sada je moguće proveriti šta je do sada urađeno. Aplikacija se ponovo prevodi, da bi nova komponenta bila angažovana. Postoje dva slučaja:

- u kontrolu se unosi vrednost koja ne počinje stringom "123" - slika 4;
- u kontrolu se unosi vrednost koja počinje stringom "123" - slika 5.

Demo forme: sa prilagođenom validacijom

SKU
aa

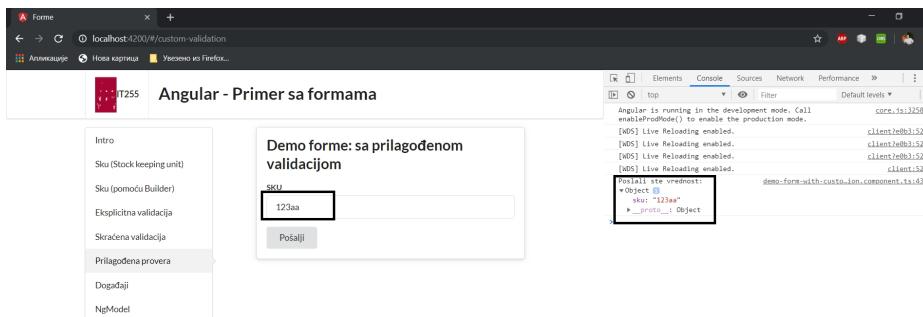
SKU nije validan

SKU mora da počne sa 123

Form nije validna

Pošalji

Slika 6.3.4 Primena prilagođenog validatora - validacija je "pala" [izvor: autor]



Slika 6.3.5 Primjena prilagođenog validatora - validacija je "prošla" [izvor: autor]

PROMENE NA FORMI

Često je neophodno sagledati bilo koju promenu koja se dešava u nekoj kontroli.

U dosadašnjem izlaganju, dobijana je vrednost iz forme pozivom metode onSubmit() nakon klika na dugme za potvrđivanje forme. Često je neophodno sagledati bilo koju promenu koja se dešava u nekoj kontroli.

Obe klase, Both *FormGroup* i *FormControl* poseduju *EventEmitter* pa je na pojednostavljen način omogućeno praćenje promena.

Za praćenje promena u kontroli neophodno je sledeće:

- dobijanje pristupa objektu `EventEmitter` pozivom `control.valueChanges`;
 - dodavanje osluškivača primenom `.subscribe()` metode.

Navedeno je realizovano sledećim listingom:

```
constructor(fb: FormBuilder) {
  this.myForm = fb.group({
    'sku': ['', Validators.required]
  });

  this.sku = this.myForm.controls['sku'];

  this.sku.valueChanges.subscribe(
    (value: string) => {
      console.log('sku je promenjen u:', value);
    }
  );

  this.myForm.valueChanges.subscribe(
    (form: any) => {
      console.log('forma je promenjena u:', form);
    }
  );
}
```

Za demonstaciju ovog pristupa, u projektu je kreirana nova komponenta **demo-form-with-events**. Kompletan kod, klase i šablona komponente je dostupan na kraju objekta učenja u sekciji **Shared Resources**.

PRAĆENJE PROMENA NA FORMI

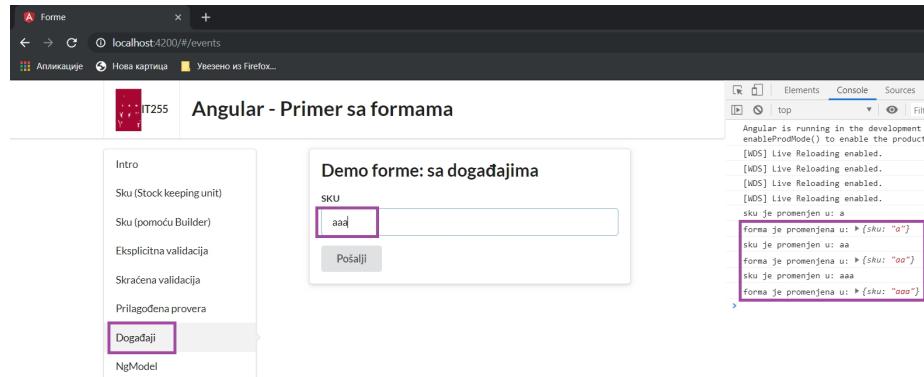
Osluškuju se događaji forme i kontrola.

Ovde sada postoji slučaj osluškivanja dva tipa razdvojenih događaja:

- događaji polja `sku`;
- događaj forme u celini.

Kada se postavi fokus na polje za unos, klik na taster tastature obavlja unos stringa u polje i osluškivač za ovu kontrolu je pokrenut. Kada kontrola promeni svoje stanje, osluškivač za formu je pokrenut.

Ako se pokrene tekući primer, praćenje promena na formi je moguće pratiti u konzoli veb pregledača, a to je demonstrirano sledećom slikom.



Slika 6.3.6 Praćenje promena na formi [izvor: autor]

▼ 6.5 Direktiva ngModel

NGMODEL

Direktiva ngModel predstavlja posebnu direktivu za povezivanje modela sa formom.

Direktiva ngModel predstavlja posebnu direktivu za povezivanje modela sa formom. Ovom Angular direktivom je omogućeno dvosmerno povezivanje podataka (*two-way data binding*). Dvosmerno povezivanje podataka je skoro uvek kompleksnije i teže za razumevanje nego što je jednosmerno koje je do sada bilo u fokusu. Okvir Angular, je u osnovi, osmišljen da daje podršku jednosmernom povezivanju podataka - od vrha ka dnu (*top-down*). Međutim, kada na scenu stupi rad sa formama, postoje slučajevi u kojima je lakše primeniti dvosmerno povezivanje podataka.

Neka forma doživi blagu izmenu - u formu se unosi string koji predstavlja naziv proizvoda (osobina *productName* klase komponente). Nakon izmene forme biće primenjena direktiva *ngModel* za održavanje sinhronizacije između kreirane objektne promenljive klase i pogleda (šablonu komponente).

U prvom koraku će biti kreirana nova komponenta za demonstraciju primene direktive *ngModel*. Sledi listing njene klase:

```
export class DemoFormNgModelComponent {
    myForm: FormGroup;
    productName: string;

    constructor(fb: FormBuilder) {
        this.myForm = fb.group({
            'productName': ['', Validators.required]
        });
    }

    onSubmit(value: string): void {
    }
}
```

```
    console.log('Poslali ste vrednost: ', value);
}
}
```

Iz listinga je moguće primetiti postojanje nove objektne promenljive *productName*, tipa string.

Za demonstraciju ovog pristupa, u projektu je kreirana nova komponenta demo-form-ng-model. Kompletan kod, klase i šablonu komponente je dostupan na kraju objekta učenja u sekciji Shared Resources.

NGMODEL I POGLEDI

Nakon kreiranja klase komponente, ngModel direktiva se primenjuje u odgovarajućem šablonu.

Nakon što je kreirana klasa komponente, *ngModel* direktivu je moguće primeniti u odgovarajućem šablonu.

```
<label for="productNameInput">Naziv proizvoda</label>
<input type="text"
       id="productNameInput"
       placeholder="Product Name"
       [FormControl]="myForm.get('productName')"
       [(ngModel)]="productName">
```

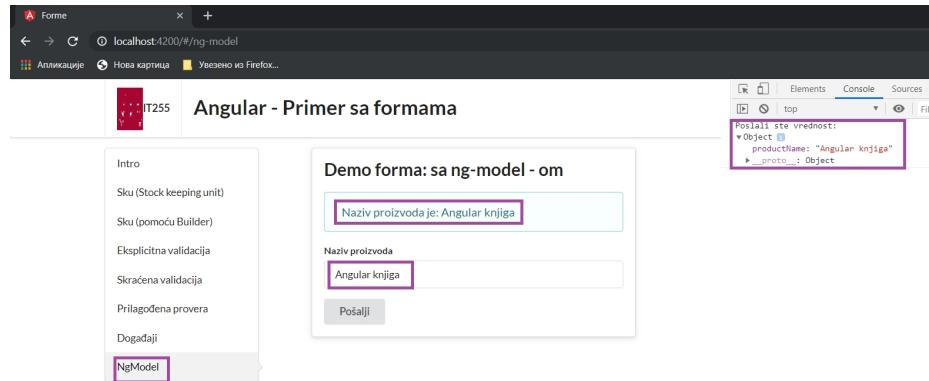
Ovde je sada moguće primetiti malo čudnu sintaksu. Istovremeno su i uglaste i male zagrade primenjene na atribut *ngModel*. Uglaste zagrade predstavljaju ulaz, a male izlaz. Sve ovo ukazuje na dvosmerno povezivanje podataka.

Takođe, moguće je primetiti još nešto: još uvek se koristi *FormControl* za specificiranje da ovaj ulaz mora da bude u vezi sa *FormControl* objektom na formi. Ovo je neophodno uraditi iz razloga što *ngModel* samo povezuje objektnu promenljivu klase komponente od koje je *FormControl* u potpunosti odvojen. Međutim, pošto je i dalje potrebno koristiti validaciju ove vrednosti i obaviti njeno prosleđivanje kao dela forme, direktiva *FormControl* je morala da bude zadržana.

Prikazivanje osobine klase *productName* u pogledu komponente može biti obavljen na sledeći način:

```
<div class="ui info message">
  Naziv proizvoda je: {{productName}}
</div>
```

Sada je samo potrebno prevođenje primera, da bi poslednja komponenta mogla da bude angažovana. Rezultat izvršavanja je prikazan sledećom slikom.



Slika 6.4.1 Primena direktive ngModel [izvor: autor]

ANGULAR FORME KROZ VIDEO MATERIJAL

Building Forms in Angular Apps; Reactive Forms - The Basics - video materijali

Building Forms in Angular Apps | Mosh - trajanje 25:03.

Ova lekcija sadrži video materijal. Ukoliko želite da pogledate ovaj video morate da otvorite LAMS lekciju.

Reactive Forms - The Basics - trajanje 15:47

Ova lekcija sadrži video materijal. Ukoliko želite da pogledate ovaj video morate da otvorite LAMS lekciju.

✓ Poglavlje 7

Dodatni materijali za rad

DODATNI MATERIJALI

Proširivanje znanja stečenog na predavanjima.

Proučite dodatnu literaturu:

1. <https://angular.io/>
2. <https://angular.io/tutorial>
3. <https://www.w3schools.com/angular/>
4. <https://www.tutorialspoint.com/angular4/>
5. <https://nodejs.org/en/>
6. <https://code.visualstudio.com/>

✓ Poglavlje 8

Pokazna vežba 8 (TRAJANJE: 45 minuta)

NOVA KOMPONENTA

Cilj vežbe je upoznavanje korisnika sa kreiranjem formi i validacijom polja.

Cilj vežbe je upoznavanje korisnika sa kreiranjem formi, kao i validacijom polja.

Neophodno je da prvo kreiramo novu komponentu, u primeru iz prošle lekcije, koja će sadržati i formu. Već poznatom komandom kreiramo komponentu *forma*:

```
ng generate component forma
```

Da bi aktuelna aplikacija dobila podršku za rad sa formama, neophodno je da glavni konfiguracioni fajl *app.module.ts* bude proširen sledećim kodom:

```
// app.module.ts

import { ReactiveFormsModule } from '@angular/forms';

imports: [
  BrowserModule, ReactiveFormsModule
],
```

ŠABLON KOMPONENTE

Šablon komponente će biti kreiran sa ciljem prikazivanja konkretne forme.

Šablon komponente će biti kreiran sa ciljem prikazivanja konkretne forme. Sledi njegov listing:

```
<div style="text-align:left">
  <h1>
    Primer forme
  </h1>
  <table>
    <tr>
      <th>
```

```
<form [FormGroup]="angForm" novalidate>
    <div class="form-group">
        <label class="center-block">Naziv:
            <input class="form-control" formControlName="name">
        </label>
    </div>
    <div *ngIf="angForm.controls['name'].invalid &&
(angForm.controls['name'].dirty || angForm.controls['name'].touched)"
         class="alert alert-danger">
        <div *ngIf="$any(angForm.controls['name']).errors.required">
            Naziv je obavezan!!!
        </div>
    </div>
</form>
</th>
<th>
    <p>Vrednost forme: {{ angForm.value | json }}</p>
    <p>Status forme: {{ angForm.status | json }}</p>
</th>
</tr>
</table>
</div>
```

Kao što je moguće primetiti, iz priloženog listinga, koriste se obrađeni Angular objekti **FormGroup** (forma) i **formControl** (input polja u formi).

Dalje, moguće je primetiti da u linijama koda 11 - 15, u priloženom šablonu, vrši se validacija unosa u formu. Da bi validacija mogla uspešno da bude obavljena, neophodno je da, u sledećem koraku, bude kreirana i klasa komponente.

KLASA KOMPONENTE

Da bi validacija mogla uspešno da bude obavljena, neophodno je da bude kreirana i klasa komponente.

Kao što je istaknuto, u prethodnom izlaganju, da bi validacija mogla uspešno da bude obavljena, neophodno je da, u sledećem koraku, bude kreirana i klasa komponente. Sledi njen listing:

```
import { Component } from '@angular/core';
import { FormGroup, FormBuilder, Validators } from '@angular/forms';

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

```
angForm: FormGroup;
constructor(private fb: FormBuilder) {
  this.createForm();
}
createForm() {
  this.angForm = this.fb.group({
    name: ['', Validators.required ]
  });
}
```

Dakle, klasa je u potpunosti u službi prethodno kreiranog šablonu. Forma se kreira na osnovu konstruktora klase i metode `createForm()` (linije koda 12 -20). U liniji 18 se primećuje da je za polje forme, pod nazivom `name`, uključen validator koji insistira da navedeno polje mora da bude popunjeno.

Konačno, neophodno je dodati kreiranu komponentu u roditeljsku, na sledeći način (linija koda 12):

```
@Component({
  selector: 'my-app',
  template: `
    <app-navbarcomponentnew></app-navbarcomponentnew>
    <div>
      <h1>{{pageheader}}</h1>
    </div>

    <app-studentcomponent></app-studentcomponent>
    <app-buttongroupcomponent></app-buttongroupcomponent>
    <app-forma></app-forma>
  `
})

export class AppComponent {
```

DODAVANJE STILA NA FORMU

Koristeći Bootstrap ili Semantic UI moguće je stilizovati formu.

Koristeći Bootstrap ili Semantic UI moguće je stilizovati formu.

Pokažimo u sledećem primeru kako je moguće na jednostavan način primenom CSS okvira stilizovati prikaz forme.

U terminalu, unutar foldera projekta, kucamo naredbu:

```
npm install semantic-ui-css --save
```

Na ovaj način, biblioteka će biti sačuvana u folderu [node_modules](#) projekta i uspešno je obavljeno instaliranje CSS radnog okvira. Da bi projekat mogao, globalno da koristi sve predefinisane stilove, neophodno je da napravimo još jednu malu modifikaciju.

Otvaramo fajl pod nazivom [angular.json](#) i u osobini [styles](#) neophodno je navesti putanju do instaliranog radnog okvira [Semantic UI](#), na sledeći način:

```
"styles": [  
    "src/styles.css",  
    "node_modules/semantic-ui-css/semantic.min.css"  
],
```

Kao u pokaznim primerima sa predavanja, sada možemo da koristimo gotove stilove za formu, kontrole i td.

Sa druge strane, možemo napisati i vlastiti CSS skript. Na primer, za kreiranu komponenu, u njenoj datoteci za stilove ([forma.component.css](#)) dodaćemo malo koda na sledeći način:

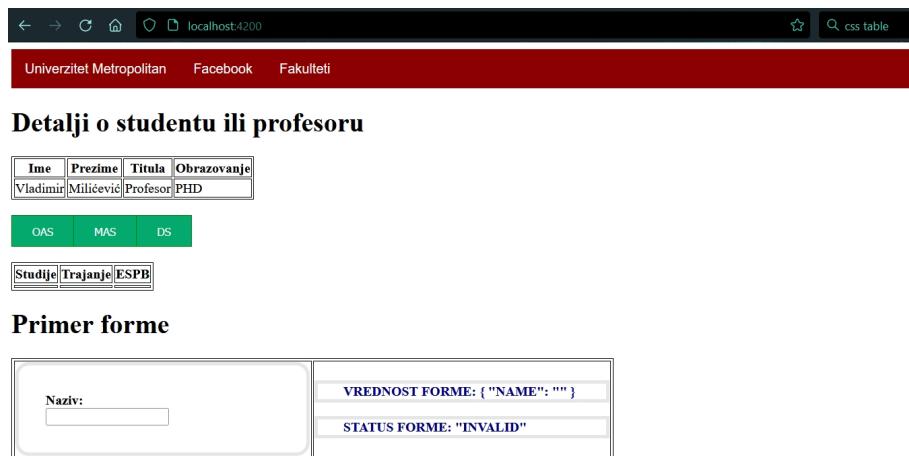
```
form {  
    box-sizing: border-box;  
    padding: 2rem;  
    border-radius: 1rem;  
    background-color: hsl(0, 0%, 100%);  
    border: 4px solid hsl(0, 0%, 90%);  
    display: grid;  
    grid-template-columns: 1fr 1fr;  
    gap: 2rem;  
    width: 350px;  
}  
  
p {  
    box-sizing: border-box;  
    background-color: hsl(0, 0%, 100%);  
    border: 4px solid hsl(0, 0%, 90%);  
    display: grid;  
    color: navy;  
    text-indent: 30px;  
    text-transform: uppercase;  
    width: 350px;  
}  
  
table, th, td {  
    border: 1px solid;  
}
```

DEMONSTRACIJA PRIMERA

Pokrećemo urađeni primer

Pokrećemo urađeni primer sa ciljem demonstracije urađenog posla.

Sledećom slikom je prikazan početni izgled forme.



The screenshot shows a web browser window with the URL `localhost:4200`. The page title is "Univerzitet Metropolitan". A navigation bar at the top includes links for "Facebook" and "Fakulteti". Below the navigation bar, the main content area has a heading "Detalji o studentu ili profesoru". Under this heading is a table with four columns: "Ime", "Prezime", "Titula", and "Obrazovanje". The first row contains the values "Vladimir", "Milićević", "Profesor", and "PHD". Below the table are three buttons labeled "OAS", "MAS", and "DS". At the bottom of the form are three buttons labeled "Studije", "Trajanje", and "ESPB".

Primer forme

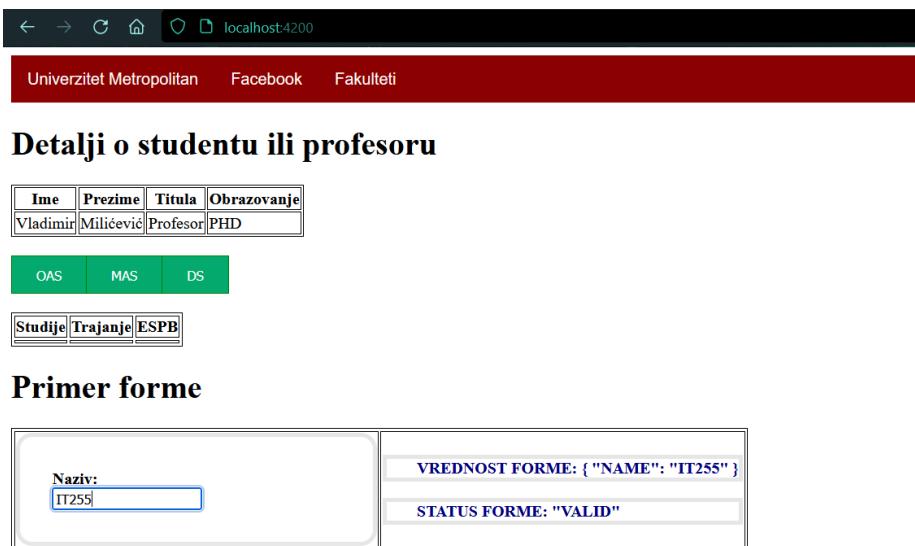
VREDNOST FORME: { "NAME": "" }
STATUS FORME: "INVALID"

Slika 8.1 Početni izgled forme [izvor: autor]

Iz priloženog je moguće videti da je forma "pala" validaciju jer njen polje `name` nije popunjeno, a obavezno je.

Dodaćemo neku vrednost u navedeno polje, kao na slici 2.

Sada je sve u redu i forma je prošla validaciju.



The screenshot shows the same web browser window and page structure as in Slika 8.1. The "Name" input field now contains the value "IT255". The validation message "STATUS FORME: 'VALID'" is displayed below the input field.

Slika 8.2 Forma je prošla validaciju

Kompletno urađen primer možete preuzeti kao dodatni materijal ovog objekta učenja.

✓ Poglavlje 9

Individualna vežba 8

INDIVIDUALNA VEŽBA (TRAJANJE: 90 MINUTA)

Samostalna dopuna zadatka sa pokaznih vežbi.

Otvorite zadatak koji ste radili na pokaznim vežbama:

Dodajte nove funkcionalnosti u kreirani Angular projekat po sledećim zahtevima:

1. Kreirajte formu za unos imena, prezimena i broja indeksa.
2. Dodati osmatrača vrednosti polja `ime`, koji će na svaku promenu ispisati određenu poruku unutar konzole.
3. Polje za unos broja indeksa ne sme da bude ne popunjeno i dozvoljene vrednosti su 1000 - 1000.

✓ Poglavlje 10

Domaći zadatak 8

DOMAĆI ZADATAK (PREDVIĐENO VREME 120 MIN)

Samostalna izrada domaćeg zadatka.

Na projekat započet prethodnim domaćim zadatkom dodati sledeće funkcionalnosti:

1. Na domaći zadatak *MetHotels*, dodati validaciju formi tako da nije moguće dodati nepravilno popunjene sobe.
2. Integrисати посматране вредности, тако да уколико је дужина вредности мања од 6 карактера, да исписује одређену поруку унутар конзоле.
3. Dodajte у форми *checkbox* контрола додатне услуге: клима, мини бар, сауна...
4. Implementацијом nglf, у зависности од изабране додатне услуге, kreirajte одговарајући излаз - нпр. "Ваша рачуна је увећана за XXX динара"

Domaći zadatak je potrebno dostaviti kao *Github commit* na prethodni, pod nazivom „*IT255-DZ08*“. Link do zadatka poslati predmetnom asistentu na mail.

NAPOMENA: Domaći zadatak dodati kao *commit* na prethodni zadatak, a ne kreirati novi repozitorijum.

▼ Poglavlje 11

Zaključak

ZAKLJUČAK

Ova lekcija je pokrila primenu ugrađenih Angular direktiva u veb aplikacijama.

Kao što je istaknuto kroz izlaganje ove lekcije, *Angular* okvir obezbeđuje veliki broj ugrađenih direktiva, koje predstavljaju atributte koji se dodaju u *HTML* elemente i na taj način kreiraju dinamičko ponašanje *Angular* veb aplikacije. Lekcija je detaljno ispunila vlastiti cilj, a to je pokrivanje najčešće korišćenih ugrađenih direktiva kroz detaljnu analizu i diskusiju vođenu primenom adekvatnog primera.

Ukoliko je detaljno pratio uputstva lekcije, student je sada sposobljen da koristi osnovne ugrađene *Angular direktive* i da prvi put kreira dinamičku *Angular* veb aplikaciju.

Prateći primer, koji objedinjuje primenu navedenih direktiva, priložen je na kraju lekcije. Možete da ga preuzmete iz sekcije *Shared Resources*.

LITERATURA

Za pripremu lekcije korišćena je aktuelna pisana i elektronska literatura.

Pisana literatura:

1. Nate Murray, Felipe Coury, Ari Lerner, Carlos Taborda, ng-Book – The Complete Book on Angular 6, Fullstack.io, 2018
2. Cody Lindley, Frontend Handbook – 2017, Frontend masters, 2017
3. Sandeep Panda, AngularJS – From Novice to Ninja, SitePoint Pty. Ltd, 2014

Elektronska literatura:

4. <https://angular.io/>
5. <https://angular.io/tutorial>
6. <https://www.w3schools.com/angular/>
7. <https://www.tutorialspoint.com/angular4/>
8. <https://nodejs.org/en/>
9. <https://code.visualstudio.com/>
10. https://www.w3schools.com/whatis/whatis_htmldom.asp



IT255 - VEB SISTEMI 1

Umetanje zavisnosti

Lekcija 09

PRIRUČNIK ZA STUDENTE

IT255 - VEB SISTEMI 1

Lekcija 09

UMETANJE ZAVISNOSTI

- ✓ Umetanje zavisnosti
- ✓ Poglavlje 1: Uvodni primer
- ✓ Poglavlje 2: Segmenti umetanja zavisnosti
- ✓ Poglavlje 3: Obezbeđivanje zavisnosti sa ngModule
- ✓ Poglavlje 4: Provajderi
- ✓ Poglavlje 5: Umetanje zavisnosti u Angular aplikaciji
- ✓ Poglavlje 6: Materijali za dodatni rad
- ✓ Poglavlje 7: Pokazne vežbe 9
- ✓ Poglavlje 8: Individualna vežba 9
- ✓ Poglavlje 9: Domaći zadatak 9
- ✓ Zaključak

Copyright © 2017 – UNIVERZITET METROPOLITAN, Beograd. Sva prava zadržana. Bez prethodne pismene dozvole od strane Univerziteta METROPOLITAN zabranjena je reprodukcija, transfer, distribucija ili memorisanje nekog dela ili čitavih sadržaja ovog dokumenta., kopiranjem, snimanjem, elektronskim putem, skeniranjem ili na bilo koji drugi način.

Copyright © 2017 BELGRADE METROPOLITAN UNIVERSITY. All rights reserved. No part of this publication may be reproduced, stored in a retrieval system or transmitted in any form or by any means, electronic, mechanical, photocopying, recording, scanning or otherwise, without the prior written permission of Belgrade Metropolitan University.

▼ Uvod

UVOD

Lekcija će se fokusirati na veoma korišćeni koncept umetanja zavisnosti u savremenim aplikacijama.

Lekcija će se fokusirati na veoma korišćeni koncept umetanja zavisnosti u savremenim aplikacijama. O ovom konceptu će uskoro biti detaljno diskutovano.

Kako napreduje gradivo ovog predmeta i kako se aktuelnim lekcijama uvode novi koncepti, programi, koji prate izlaganje, biće sve veći i kompleksniji. Tako će se ubrzo javiti potreba da komponente komuniciraju sa ostalim modulima. **U situaciji kada je za izvršavanje modula A neophodan modul B, kaže se da je B zavisnost za A.**

Jedan od opštih načina za pristupanje zavisnostima predstavlja importovanje odgovarajuće datoteke. Na primer, ovo je deo listinga nekog modula *A.ts*:

```
//deo koda klase A.ts

import {B} from 'B'; // ovo je zavisnost

B.funkcija(); // koristi B
```

U brojnim slučajevima dovoljan je prikazani uvoz koda. Međutim, postoje i situacije kada je neophodno obezbediti zavisnosti na sofisticiraniji način. Na primer, možda je cilj da se u aplikaciji primeni neki od sledećih scenarija:

- zamena klase B klasom *MockB* tokom izvođenja testiranja;
- deljenje jedinstvene instance klase B preko cele aplikacije (*Singleton šablon - pattern*);
- kreiranje nove instance klase B svaki put kada se koristi (*Producioni šablon - Factory pattern*).

Navedene probleme je moguće rešiti primenom koncepta umetanja zavisnosti. **Umetanje zavisnosti (dependency injection - DI)** prestavlja sistem koji omogućava da delovi nekog programa budu dostupni ostalim delovima tog programa, a programeri podešavaju načine kako se to izvodi.

Izraz "umetanje zavisnosti" se istovremeno upotrebljava za opisivanje dizajn šablona (kojeg koriste brojni radni okviri) i specifične implementacije za *DI* ugrađene u *Angular* okvir.

Glavna korist primene umetanja zavisnosti ogleda se u činjenici da klijent komponenta ne mora da bude upućena u način kreiranja same zavisnosti. Sve što klijent komponenta mora da zna jeste kako da komunicira sa zavisnošću.

Za sada, studentima navedeno može biti apstraktno i nerazumljivo. Međutim, brzo će biti, kao i u ostalim lekcijama, uveden pokazni primer za lakše razumevanje ove važne problematike veb programiranja.

OPŠTI POGLED NA UMETANJE ZAVISNOSTI

Izlaganje u vezi sa umetanjem zavisnosti može biti apstraktno i nerazumljivo - neophodno ga je približiti na jednostavniji način.

Upravo iz razloga što studentima navedeno izlaganje u vezi sa umetanjem zavisnosti može biti apstraktno i nerazumljivo i pre uvođenja pokaznog primera za lakše razumevanje ove važne problematike veb programiranja, moguće je napraviti paralele sa realnim životnim situacijama.

Kako bi DI bilo primenjeno u stvarnom životu?

Uzmimo za primer da ste neko ko voli da putuje i za to angažujete agenciju, ili da ste na poziciji menadžera. Da biste organizovali putanje imali biste otprilike sledeće korake:

- kontaktirali agenciju za prodaju avio karata, tj. avio - prevoznika i rezervisali karte;
- da biste kupili karte morate fizički da odete do agenciju (u slučaju da nemaju online prodaju) i podignite karte;
- ukoliko agencija ima online prodaju, karte rezervišete, platite i onda ih morate preuzeti (osim ako ih nema na aerodromu);
- pozvali biste taksi i rezervisali vožnju do aerodroma u određeno vreme.

Naravno, sve ove korake morali biste da uradite i za koleginice, odnosno kolege koje putuju sa vama što je vremenski zahtevan posao.

Sa druge strane sve ove obaveze može za vas da uradi posebno odeljenje. Odeljenje koje unapred zna termine putovanja i vi dobijete kupljenu kartu na sto.

Oba ova scenarije su interakcija između vas kao klijenta i agencije kao pružaoca usluge, jedino što u drugom scenariju vas i vašu kompaniju mnogo manje košta vremena i novca a rezultat je isti. Vi imate zakazano putovanje i kupljene karte.

Ovim primerom želeti smo da predstavimo koncept umetanja zavisnosti u realnom životu, jer je karta koja je vama potrebna kreirana eksterno i vama dostavljena bez da ste vi pokrenuli neku akciju.

Šta je onda DI u softverskom domenu?

Skraćena verzija odgovora: davanje (ili dodeljivanje) instance objekta varijabli nekog drugog objekta. I to je to!

Ništa više od toga, ništa manje.

UVODNI VIDEO

Trajanje video snimka: 6min 5sek

Ova lekcija sadrži video materijal. Ukoliko želite da pogledate ovaj video morate da otvorite LAMS lekciju.

▼ Poglavlje 1

Uvodni primer

VIDEO PREDAVANJE ZA OBJEKAT "UVODNI PRIMER PRICESERVICE"

Trajanje video snimka: 19min 35sek

Ova lekcija sadrži video materijal. Ukoliko želite da pogledate ovaj video morate da otvorite LAMS lekciju.

DEFINICIJA KLASE PRICESERVICE

Izlaganje u vezi sa umetanjem zavisnosti može biti apstraktno i nerazumljivo - uvodi se primer.

Kao što je istaknuto u uvodnom izlaganju, studentima prikazano izlaganje u vezi sa umetanjem zavisnosti može biti apstraktno i nerazumljivo. Međutim, ovde će biti, kao i u ostalim lekcijama, uveden pokazni primer za lakše razumevanje ove važne problematike veb programiranja.

Na primer, kreira se [Angular aplikacija](#) prodavnice koja poseduje proizvode ([Products](#)) i jedan od njenih zadataka je računanje konačne cene proizvoda nakon uračunate takse. Sa ciljem računanja pune cene proizvoda neophodna je klasa [PriceService](#) koja poseduje adekvatnu metodu koja kao ulaz uzima:

- osnovnu cenu proizvoda (objekta [Product](#));
- državu ([state](#)) u koju se proizvod prodaje;

a nakon toga vraća rezultat u formi konačne cene, računajući i taksu.

Sledećim listingom je prikazan kod ove klase. Potpuno uređen i proveren primer možete preuzeti na kraju izlaganja lekcije u aktivnosti [Shared Resources](#). U uređenom primeru, ovaj listing se nalazi na lokaciji: [src/app/price-service-demo/price.service.1.ts](#)

```
export class PriceService {  
    constructor() { }  
  
    calculateTotalPrice(basePrice: number, state: string) {  
        // npr. zamislite da se u realnoj aplikaciji pristupa bazi podataka  
        // sa podacima o iznosu poreza  
        const tax = Math.random();  
    }  
}
```

```
    return basePrice + tax;  
}  
  
}
```

U kreiranoj servisnoj klasi, funkcija `calculateTotalPrice()` preuzima osnovnu cenu (parametar `basePrice`) proizvoda i državu (parametar `state`) u koju se kupljeni proizvod šalje i vraća ukupnu cenu proizvoda.

PRIMENA SERVISA U MODELU

Servis je neophodno primeniti na konkretan objekat.

Servis, sam po sebi, ne radi ništa ukoliko nije povezan sa konkretnim objektom koji ga primenjuje. U konkretnom slučaju, kreirani servis se primenjuje na objekte modelske klase Product. Za početak neophodno je pokazati kako je to moguće uraditi bez umetanja zavisnosti. Sledi listing iz datoteke primera: `src/app/price-service-demo/product.model.1.ts`:

```
import { PriceService } from './price.service';  
  
export class Product {  
  service: PriceService;  
  basePrice: number;  
  
  constructor(basePrice: number) {  
    this.service = new PriceService(); // <-- direktno kreirano ("hardcoded")  
    this.basePrice = basePrice;  
  }  
  
  totalPrice(state: string) {  
    return this.service.calculateTotalPrice(this.basePrice, state);  
  }  
}
```

Neka je sledeća zamisao da je potrebno kreirati test za upravo demonstriranu klasu i to je moguće ilustrovati sledećim listingom:

```
import { Product } from './product.model';  
  
describe('Product', () => {  
  let product;  
  
  beforeEach(() => {  
    product = new Product(11);  
  });  
  
  describe('price', () => {  
    it('izračunato na osnovu osnovne cene i države', () => {  
      expect(product.totalPrice('RS')).toBe(11.66); // -> ???  
    });  
  });  
});
```

```
});  
});  
});
```

Problem ovog testa je u tome što ne postoji saznanje o tačnoj vrednosti takse za slanje kupljenog proizvoda ka Republici Srbiji ("RS"). Čak i ako `PriceService` bude implementiran na realan način `API` pozivom ili pozivom baze podataka, postojaće problem:

- `API` mora da bude dostupan ili pokrenuta baza podataka;
- Neophodna je informacija o tačnom iznosu takse za slanje kupljenog proizvoda ka Republici Srbiji u trenutku pisanja testa.

KREIRANJE INTERFEJSA

Ukoliko se testira metoda iz spoljašnjeg resursa moguće je simulirati (mock-ovati) servis.

U ovom delu izlaganja postavlja se ključno pitanje: Šta je neophodno učiniti ukoliko se testira metoda `product.totalPrice()` oslanjajući se na spoljašnji resurs? U ovakvim situacijama simulira se (*mock*) `PriceService`. Ukoliko je poznat interfejs za `PriceService`, moguće je napisati klasu `MockPriceService` koja će uvek obaviti predvidljivo izračunavanje i neće se oslanjati na bazu podataka ili API.

Neka se traženi interfejs naziva `IPriceService` i neka je njegova lokacija u priloženom urađenom projektu: `src/app/price-service-demo/price-service.interface.ts`. Sledi njegov listing:

```
export interface IPriceService {  
    calculateTotalPrice(basePrice: number, state: string): number;  
}
```

Posmatrani interfejs definiše samo jednu funkciju pod nazivom `calculateTotalPrice()`. Sada je moguće kreirati pomenutu klasu `MockPriceService` koja je zapravo implementaciona klasa kreiranog interfejsa. Sledi njen listing koji se u projektu nalazi u datoteci: `src/app/price-service-demo/price.service.mock.ts`.

```
import { IPriceService } from './price-service.interface';  
  
export class MockPriceService implements IPriceService {  
    calculateTotalPrice(basePrice: number, state: string) {  
        if (state === 'RS') {  
            return basePrice + 0.66; // uvek je 66 centi!  
        }  
  
        return basePrice;  
    }  
}
```

Klasom je konkretizovana metoda i definisana je visina takse za slanje proizvoda koji je kupljen.

U nastavku izlaganja je neophodno posebnu pažnju posvetiti modifikaciji modelske klase [Product](#).

MODIFIKOVANJE MODELA

Da bi Product objekti mogli da koriste inoviran servis, neophodno je modifiovati ovu klasu.

Na programu je neophodan dodatni rad. Iako je uspešno kreirana klasa [MockPriceService](#) klasa [Product](#) je još uvek ne koristi. Da bi [Product](#) objekti mogli da koriste ovakav servis, neophodno je obaviti modifikaciju modelske klase [Product](#). Sledi modifikovan listing navedene klase iz datoteke [src/app/price-service-demo/product.model.ts](#):

```
import { IPriceService } from './price-service.interface';

export class Product {
    service: IPriceService;
    basePrice: number;

    constructor(service: IPriceService, basePrice: number) {
        this.service = service; // <- prosleđeno kao argument!
        this.basePrice = basePrice;
    }

    totalPrice(state: string) {
        return this.service.calculateTotalPrice(this.basePrice, state);
    }
}
```

Sada, prilikom kreiranja objekta [Product](#), klijent koristeći klasu [Product](#) postaje odgovoran za odlučivanje koja konkretna implementacija servisa [PriceService](#) će biti data novoj instanci.

Nakon izvršene modifikacije modela, moguće je sada popraviti test i osloboediti se zavisnosti od nepredvidljivog [PriceService](#)-a (datoteka: [/src/app/price-service-demo/product.spec.ts](#)):

```
import { Product } from './product.model';
import { MockPriceService } from './price.service.mock';

describe('Product', () => {
    let product;

    beforeEach(() => {
        const service = new MockPriceService();
        product = new Product(service, 11.00);
    });

    describe('price', () => {
```

```
it('izračunato na osnovu osnovne cene i države', () => {
    expect(product.totalPrice('RS')).toBe(11.66);
});
});
```

U nastavku izlaganje, akcenat se stavlja na izučavanje novog modela po kojem se implementacija zavisnosti može menjati tokom vremena izvršavanja aplikacije. U literaturi, ovaj model je poznat pod nazivom model sistema Angular umetanja zavisnosti.

MODEL SISTEMA ANGULAR DI

Po ovom modelu implementacija zavisnosti može menjati tokom vremena izvršavanja aplikacije.

Ako se dodatno posveti pažnja prethodnom izlaganju moguće je zaključiti da sigurno postoje dobre strane testiranja modelskih klasa u izolaciji. To, pre svega, znači da sa sigurnošću može da se tvrdi da će klasa pravilno funkcionisati sa predvidljivim zavisnostima.

Iako je predvidljivost, u određenim situacijama, sasvim u redu, postavlja se nov problem. Šta se dešava sa prosleđivanjem konkretnе implementacije servisa svaki put kada se kreira nov objekat klase *Product*? Angular DI biblioteka u velikoj meri pomaže prilikom rešavanja navedenog problema.

Unutar Angular DI sistema, umesto direktnog uvoza (*import*) i kreiranja nove instance klase, moguće je postupiti na sledeći način:

- registrovanje zavisnosti pomoću *Angular*-a;
- opisivanje kako će zavisnost biti umetnuta;
- umetanje zavisnosti.

Jedna prednost ovog modela je ta što se implementacija zavisnosti može menjati tokom vremena izvršavanja aplikacije (kao u prethodnom slučaju sa *mock* klasom). Međutim, značajnija prednost primene ovog modela je u činjenici da programer sam može da podesi kako će zavisnost biti kreirana.

Ovo je veoma čest slučaj u programima koji su servisno orijentisani - pri čemu se insistira na postojanju jedne instance koja je označena kao *Singleton* (jedinac). Primenom umetanja zavisnosti ovakvi objekti se lako podešavaju.

Treći slučaj primene umetanja zavisnosti predstavlja podešavanje aplikacije ili specifičnih promenljivih okruženja. Na primer, moguće je definisati *API_URL* kao konstantu, a potom obaviti umetanje različitih vrednosti u različitim fazama životnog ciklusa razvoja aplikacije: produkcija (*production*) ili razvoj (*development*).

Nakon obavljenog uvoda, najvažniji zadatak jeste da studenti nauče da kreiraju vlastite servise i različite načine kako da obave njihovo umetanje. Rešenje se nalazi u nastavku lekcije.

▼ Poglavlje 2

Segmenti umetanja zavisnosti

TOKEN I DELOVI ZAVISNOSTI

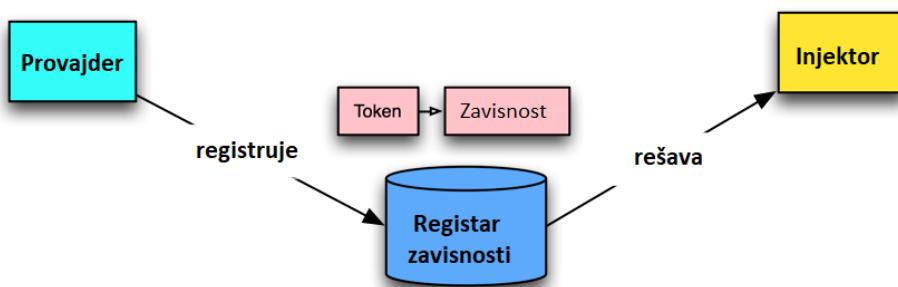
Za registrovanje zavisnosti neophodno je povezati zavisnost sa objektom koji će je identifikovati.

U [Angular](#) okviru, za registrovanje zavisnosti neophodno je povezati zavisnost sa nekim objektom koji će je identifikovati. Ova identifikacija je poznata kao [token zavisnosti](#) (*dependency token*). Na primer, neophodno je registrovati URL za API i tada je moguće koristiti string [API_URL](#) kao token. Na sličan način, ukoliko se registruje klasa moguće je koristiti upravo klasu kao vlastiti token.

Umetanje zavisnosti u Angular - u čine tri dela:

- [provajder \(provider\)](#) - povezuje token (koji može biti string ili klasa) sa listom zavisnosti. Navodi Angular kako da kreira objekat na osnovu tokena;
- [umetač ili injektor \(injector\)](#) - sadrži skup povezivanja (binding) i odgovoran je za rešavanje zavisnosti i njihovo umetanje prilikom kreiranja objekata;
- [zavisnost \(dependency\)](#) - vrednost koja je umetnuta.

Sledećom slikom je moguće ilustrovati uloge svakog navedenog dela umetanja zavisnosti u [Angular](#) okviru.



Slika 2.1.1 Angular - umetanje zavisnosti [izvor: autor]

Slika daje sledeće tumačenje: kada se podešava umetanje zavisnosti, specificira se šta je umetnuto (*inject*) i kako će biti rešeno (*resolve*).

Značaj i primenu umetača za implementaciju koncepta umetanja zavisnosti u [Angular](#) aplikacijama je posebno potrebno skenirati u narednom izlaganju.

✓ 2.1 Rad sa umetačima (Injector)

ULOGA UMETAČA

Angular koristi injektore za rešavanje zavisnosti i kreiranje promenljivih.

Umetač ili **injektor** (injector) je veoma važan koncept **Angular DI** modela kojeg je neophodno detaljno izložiti u narednom izlaganju.

U prethodnoj analizi, prilikom rada sa klasama **Product** i **PriceService**, instanca za **PriceService** je manuelno kreirana primenom operatora **new**. Šta Angular, zapravo, radi? Angular koristi injektore za rešavanje zavisnosti i kreiranje promenljivih. Ovo se sve dešava u pozadini. Međutim, za vežbu je odlično istražiti i pokazati šta se dešava. U tom kontekstu, moguće je obaviti manuelnu primenu injektora sa ciljem pokazivanja šta to, zapravo, Angular radi u pozadini.

Proširuje se tekući primer kreiranjem nove komponente **services**. Komponenta će posedovati funkcionalnosti koje omogućavaju manuelnu upotrebu injektora za rešavanje (izvršavanje) i kreiranje servisa.

Jedan od opštih načina primene servisa podrazumeva postojanje globalne jedinac (singleton) instance. Na primer, postroji klasa **UserService** koja sadrži informaciju u vezi sa trenutno prijavljenim korisnikom. Veliki broj različitih komponenata može da zahteva logiku baziranu na tekućem korisniku. Zbog navedenog, ovo je odličan primer servisa.

Sledećim listingom je data jednostavna **UserService** klasa koja poseduje objekat **user** kao osobinu. U projektu, klasa se nalazi na lokaciji **/src/app/services/user.service.ts**:

```
import { Injectable } from '@angular/core';

@Injectable()
export class UserService {
  user: any;

  setUser(newUser) {
    this.user = newUser;
  }

  getUser(): any {
    return this.user;
  }
}
```

PRIMENA SERVISA U DRUGOJ KOMPONENTI

*Proširuje se tekući primer kreiranjem nove komponente **user-demo**.*

U drugoj komponenti (*user-demo*) neophodno je kreirati i odgovarajući šablon koji će omogućiti rad sa servisima korisnika. Neka šablon odgovara najjednostavnijoj mogućoj formi za prijavljivanje korisnika (videti sliku) koja sadrži samo jedno dugme "Prijavljanje". Sledećim listingom je dat HTML kod opisanog šablona (datoteka: *src/app/user-demo/user-demo.component.html*):

```

<div>
  <p
    *ngIf="userName"
    class="welcome">
    Dobrodošli: {{ userName }}!
  </p>
  <button
    (click)="signIn()"
    class="ui button">
    Prijavljanje
  </button>
</div>

```

Iz listinga je moguće primetiti da će klikom na dugme "Prijavljanje" biti omogućeno prikazivanje pozdravne poruke za korisnika, baš kao na sledećoj slici:



Slika 2.2.1 Jednostavna forma za prijavljenog korisnika [izvor: autor]

DIREKTNA PRIMENA INJEKTORA

Funkcionalnost servisa je neophodno implementirati direktnom primenom injektora.

Funkcionalnost, prikazana prethodnom slikom, još uvek nije postignuta u trenutnom stanju primera, koji prati izlaganje lekcije. Traženu funkcionalnost je neophodno implementirati direktnom primenom injektora. Za tekuću komponentu *user-demo* kreira se klasa komponente *UserDemoInjectorComponent* na lokaciji u projektu: *src/app/user-demo/user-demo.injector.component.ts*:

```

import {
  Component,
  ReflectiveInjector
} from '@angular/core';

```

```
import { UserService } from '../services/user.service';

@Component({
  selector: 'app-injector-demo',
  templateUrl: './user-demo.component.html',
  styleUrls: ['./user-demo.component.css']
})
export class UserDemoInjectorComponent {
  userName: string;
  userService: UserService;

  constructor() {
    // kreira injektor za kreiranje i rešavanje UserService
    const injector: any = ReflectiveInjector.resolveAndCreate([UserService]);

    // koristi injektor za dobijanje UserService instance
    this.userService = injector.get(UserService);
  }

  signIn(): void {
    // sa prijavljivanjem, podešava se korisnik

    this.userService.setUser({
      name: 'Vladimir Milićević'
    });

    // čitanje korisničkog imena iz servisa
    this.userName = this.userService.getUser().name;
    console.log('Korisničko ime je: ', this.userName);
  }
}
```

Kod započinje kao kod osnovne komponente: tu je selektor, šablon i CSS. Zatim, postoje dve osobine: `userName`, koja čuva naziv trenutno prijavljenog korisnika i `userService` koja čuva referencu ka `UserService`.

Posebni pažnju bi trebalo posvetiti konstruktoru. Iz koda se primećuje da on sadrži statičku metodu `resolveAndCreate()` klase `ReflectiveInjector`. Ova metoda je odgovorna za kreiranje novog injektora. Parametar koji se metodi prosleđuje predstavlja informacije koje injektor koristi za umetanje zavisnosti. U konkretnom slučaju, ukazuje se injektoru na klasu `UserService`.

Klasa `ReflectiveInjector` je konkretna implementacija apstraktne klase `Injector` koja koristi refleksiju za pretragu odgovarajućih tipova parametara.

Dobrodošli: Vladimir Milićević!

Prijavljanje

Slika 2.2.2 Poruka prijavljenom korisniku [izvor: autor]

▼ Poglavlje 3

Obezbeđivanje zavisnosti sa NgModule

TIPIČAN NAČIN PRIMENE INJEKTORA

Direktna primena injektora ne predstavlja tipičan način njihove primene u Angular okviru.

U prethodnom izlaganju da direktna primena injektora može biti veoma zanimljiva. Pa, ipak, ovo ne predstavlja tipičan način njihove primene u **Angular** okviru. Umesto toga, obično se radi sledeće:

- koristi se **NgModule** za registrovanje šta će biti umetnuto;
- koriste se dekoratori (obično za konstruktore) za specifikaciju šta će biti umetnuto.

Izvođenjem navedenih koraka Angular upravlja kreiranjem injektora i rešavanjem zavisnosti. Neophodno je krenuti od prvog koraka, prepostavka je da je klasa **UserService** konvertovana u oblik da može biti umetnuta kao jedinac (singleton) na više pogodnih mesta u aplikaciji. Zbog toga je potrebno dodati je u **NgModule** klasu pod ključem providers. To je prikazano sledećim listingom (datoteka: */src/app/user-demo/user-demo.module.ts*)

```
import { NgModule } from '@angular/core';
import { CommonModule } from '@angular/common';

// uvde se uvozi UserService
import { UserService } from './services/user.service';

@NgModule({
  imports: [
    CommonModule
  ],
  providers: [
    UserService // <-- registrovan pod ključem providers
  ],
  declarations: []
})
export class UserDemoModule { }
```

Sada je moguće umetnuti servis **UserService** u komponentu na pogodan način. To je prikazano sledećim listingom (datoteka: */src/app/user-demo/user-demo.component.ts*):

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

```
import { UserService } from './services/user.service';

@Component({
  selector: 'app-user-demo',
  templateUrl: './user-demo.component.html',
  styleUrls: ['./user-demo.component.css']
})
export class UserDemoComponent {
  userName: string;
  // uklonjen `userService` zbog dole skraćene def. konstruktora

  // Angular će umetnuti singleton instancu `UserService` ovde.
  // Biće podešena kao osobina `private`.
  constructor(private userService: UserService) {
    // prazno jer ovde nije ništa potrebno!
  }

  // ostaje isto...
  signIn(): void {
    // prilikom prijavljivanja, podešava se user
    this.userService.setUser({
      name: 'Vladimir Milićević'
    );

    // čitanje korisničkog imena iz servisa
    this.userName = this.userService.getUser().name;
    console.log('Korisničko ime je: ', this.userName);
  }
}
```

Prvo što je moguće primetiti iz listinga jeste da je unutar konstruktora klase *UserDemoComponent*, kao argument, dodata vrednost *userService: UserService*. Kada se ova komponenta kreira u odgovarajućoj stranici, Angular će rešiti i umetnuti *UserService singleton* objekat. Ono što je odlično, Angular upravlja instancom, nije potrebno da to uradi sam programer. Svaka klasa koja ubrizgava *UserService* dobiće isti singleton.

PROVAJDERI KAO KLJUČEVİ

UserService je registrovan pod ključem providers u klasi NgModules.

Važno je znati da kada se dodaje *UserService* u konstruktor komponente *UserDemoComponent*, Angular će znati šta i kako treba da umetne. Ovo je moguće iz razloga što je *UserService* registrovan pod ključem providers u klasi *NgModules*.

Angular ne obavlja umetanje proizvoljnih klasa. Veoma je bitno podešavanje *NgModule* klase za omogućavanje funkcionisanja umetanja zavisnosti. Iako je već bilo dosta govora o singleton servisima neophodno je imati na umu da je njihovo umetanje u komponente Angular aplikacije moguće obaviti na više različitih načina. Upravi će o tome biti reči u nastavku lekcije.

▼ Poglavlje 4

Provajderi

NAČINI REŠAVANJA UMETNUTIH ZAVISNOSTI

Postoji nekoliko načina na koje je moguće podesiti rešavanje umetnutih zavisnosti u radnom okviru Angular.

Postoji nekoliko načina na koje je moguće podesiti rešavanje umetnutih zavisnosti u radnom okviru [Angular](#). Na primer, moguće je sledeće:

- umetnuti ([singleton](#)) instancu klase (pokazano je u prethodnom izlaganju);
- umetnuti vrednost;
- obaviti poziv funkcije i umetnuti vrednost posmatrane funkcije.

Neophodno je sagledati detalje za svaku prethodno navedenu stavku.

PRIMENA KLASE

Umetanje singleton instance klase je verovatno najčešći tip umetanja zavisnosti

Kao što je već diskutovano, umetanje singleton instance klase je verovatno najčešći tip umetanja zavisnosti. Tada se klasa dodaje u listu provajdera. To je moguće uraditi na sledeći način:

```
providers: [ UserService ]
```

Na ovaj način je ukazano Angular radnom okviru da je neophodno obezbediti singleton instancu klase [UserService](#) svaki put kada je ovaj servis injektovan. Zbog toga što je ovaj šablon opšti, klasa sama po sebi predstavlja skraćenu notaciju za sledeće, ekvivalentno podešavanje:

```
providers: [
  { provide: UserService, useClass: UserService }
]
```

Ono što je važno napomenuti jeste da podešavanje objekta, u drugom slučaju, zahteva primenu dva ključa: [provide](#) i [useClass](#). Ključem [provide](#) je označen [token](#) koji se koristi za identifikovanje umetanja zavisnosti, a drugim ključem, [useClass](#), određeno je kako i šta bi trebalo umetnuti.

U konkretnom slučaju vrši se mapiranje klase `UserService` sa tokenom `UserService`. Kao što se vidi, u ovom slučaju, token i naziv klase savršeno odgovaraju jedno drugom. Ove je opšti slučaj, ali je takođe bitno napomenuti da token i umetnuta klasa ne moraju, nužno, da imaju iste nazive.

Kao što je već diskutovano, injektor će kreirati singleton objekat u pozadini i vratiti istu instancu svaki put kada se vrši njeno umetanje. Naravno, kada je prvi put umetnuta, singleton instanca još nije kreirana. Zbog navedenog, kada se prvi put kreira `UserService` instance, Angular DI sistem će pokrenuti konstruktor klase.

PRIMENA VREDNOSTI

Sledeći način umetanja zavisnosti predstavlja obezbeđivanje vrednosti

Kao što je već diskutovano, umetanje singleton instance klase je verovatno najčešći tip umetanja zavisnosti. Sledeci način, vredan diskusije, umetanja zavisnosti predstavlja *obezbeđivanje vrednosti*. Navedeno je veoma slično upotrebi globalnih promenljivih. Na primer, moguće je podešavanje URL-a krajne API tačke (`API Endpoint URL`) u zavisnosti od okruženja. Da bi navedeno bilo realizovano, pod globalnim `NgModule` ključem `providers`, koristi se ključ `useValue`, na primer, na sledeći način:

```
providers: [
  { provide: 'API_URL', useValue: 'http://my.api.com/v1' }
]
```

Iz priloženog koda se primećuje da je obezbeđen token u formi stringa `API_URL`. Ukoliko se koristi string za obezbeđivanje vrednosti, Angular ne može da odluči koju zavisnost je neophodno rešiti po tipu podataka. Na primer, nije dozvoljena upotreba ovakvog koda.

```
// ovo ne radi - anti-primer
2 export class AnalyticsDemoComponent {
3   constructor(apiUrl: 'API_URL') { // <--- ovo nije tip podataka, samo običan string
4     // ako stavimo `string` to je dvosmisленo
5   }
6 }
```

Postavlja se pitanje: Šta se čini u ovakvim situacijama? Odgovor leži u primeni dekoratora `@Inject()`, na primer, na sledeći način:

```
import { Inject } from '@angular/core';

export class AnalyticsDemoComponent {
  constructor(@Inject('API_URL') apiUrl: string) {
    // radi! uradi nešto w/ apiUrl
  }
}
```

Sada, pošto je naučeno upravljanje jednostavnim vrednostima preko *useValue* i singleton klasama preko *useClass*, moguće je posvetiti se naprednim konceptima. Jedan od njih je, svakako, pisanje podesivih servisa primenom fabrikacije ili produkcije (*factory*).

PODESIVI SERVISI

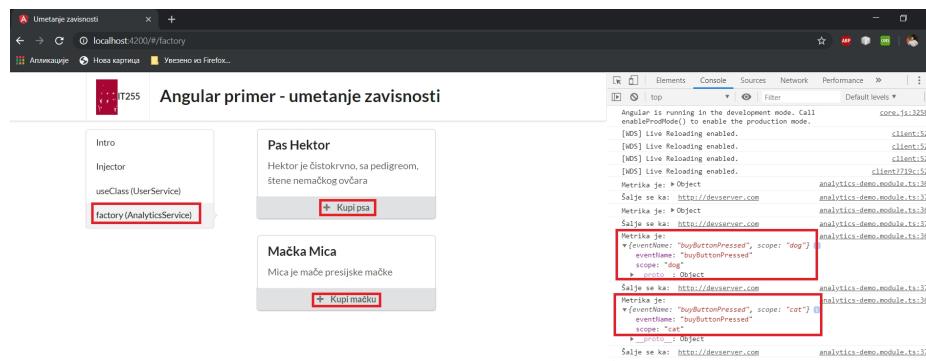
Podesivi servisi predstavljaju napredan koncept kojem je neophodno posvetiti posebnu pažnju.

Podesivi servisi (*Configurable Services*), kao što je istaknuto u prethodnom izlaganju, predstavljaju napredan koncept kojem je neophodno posvetiti posebnu pažnju.

U slučaju servisa *UserService*, nijedan argument nije potreban konstruktoru. Međutim, šta se dešava ukoliko konstruktor servisa zahteva neke argumete? Ovaj problem je moguće rešiti primenom **produkione funkcije (fabrikacije)** koja predstavlja funkciju koja vraća bilo koji **umetnuti objekat**.

Na primer, kreira se biblioteka za beleženje analitike korisnika (čuvanje zapisa o događajima akcija koje je korisnik obavio na stranici). U ovom slučaju, neophodno je kreiranje servisne klase *AnalyticsService* koja definiše interfejs za čuvanje događaja ali ne i implementaciju za rukovanje događajima.

Sledećom slikom je prikazana funkcionalnost aktuelnog primera za praćenje analitike događaja



Slika 4.1 Praćenje analitike događaja [izvor: autor]

Korisnici aplikacije bi možda želeli da šalju i čuvaju vlastite metrike na Google Analytics (<https://analytics.google.com/analytics/web/provision/?authuser=0#/provision>) ili da koriste platformu poput *Optimizely* (<https://www.optimizely.com/>) ili neku drugu soluciju.

SERVIS ZA PRAĆENJE ANALITIKE

Servisne metode će biti definisane preko dva konkretna interfejsa koje će konkretizovati klasa servisa AnalyticsService.

U nastavku cilj je kreiranje servisa kojeg će biti moguće umetnuti i koji će definisati metode koje će konkretnizovati implementaciona klasa servisa [AnalyticsService](#). Servisne metode će biti definisane preko dva konkretna interfejsa, od kojih i započinje izlaganje ovog dela lekcije.

Prvo sledi definicija interfejsa [Metric](#) (datoteka: [/src/app/analytics-demo/analytics-demo.interface.ts](#)):

```
/**  
 * Definiše jednostavnu metriku  
 **/  
export interface Metric {  
  eventName: string;  
  scope: string;  
}
```

Kreirani interfejs će poslužiti za čuvanje osobina [eventName](#) i [scope](#). Za primer može da posluži sledeće, kada se korisnik "Vlada" poveže na aplikaciju, osobina [eventName](#) može da dobije vrednost [loggedIn](#), a osobina scope vrednost "Vlada".

```
// ovo je samo primer  
let metric: Metric = {  
  eventName: 'loggedIn',  
  scope: 'Vlada'  
}
```

Na ovaj način je moguće, na primer, prebrojati korisnička povezivanja na aplikaciju preko osobine [eventName](#) i vrednosti "[loggedIn](#)", kao i povezivanje specifičnih korisnika (vrednost "Vlada") na aplikaciju.

U nastavku je, takođe, potrebno definisati implementaciju analitike putem interfejsa [AnalyticsImplementation](#) (datoteka:[/src/app/analytics-demo/analytics-demo.interface.ts](#)):

```
/**  
 * Definiše šta bi implementacija sačuvane metrike trebalo da bude  
 **/  
export interface AnalyticsImplementation {  
  recordEvent(metric: Metric): void;  
}
```

KLASA SERVISA ZA PRAĆENJE ANALITIKE

Neophodno je obaviti kreiranje implementacione klase za interfejs koja će koristiti koncept umetanja zavisnosti.

U prethodnom izlaganju je kreiran interfejs [AnalyticsImplementation](#) koji definiše jednu funkciju [recordEvent\(\)](#). Navedena funkcija uzima objekat tipa [Metrics](#) kao argument. U nastavku je neophodno kreiranoj metodi dati konkretno zaduženje. To se postiže kreiranjem implementacione klase za interfejs koja će koristiti koncept umetanja zavisnosti.

Klasa se naziva `AnalyticsService` i u priloženom projektu se nalazi na lokaciji `/src/app/services/analytics.service.ts`:

```
import { Injectable, Inject } from '@angular/core';
import {
  Metric,
  AnalyticsImplementation
} from '../analytics-demo/analytics-demo.interface';

@Injectable()
export class AnalyticsService {
  constructor(@Inject(AnalyticsService) private implementation:
  AnalyticsImplementation) {
  }

  record(metric: Metric): void {
    this.implementation.recordEvent(metric);
  }
}
```

Klasa definiše jednu metodu `record()` koja kao parametar uzima objekat tipa `Metric` i prosleđuje ga objektu tipa `AnalyticsImplementation`.

Takođe, zanimljivo je za primetiti kako konstruktor uzima izraz kao parametar. Ukoliko bi bilo pokušana regularna primena mehanizma umetanja `useClass`, u veb pregledaču bi se pojavila greška poput:

Cannot resolve all parameters for AnalyticsService.

Ovo se dešava iz razloga što nije obezbeđen injektor sa implementacijom neophodnom za konstruktor. Sa ciljem rešavanja navedenog problema, neophodno je podesiti provajdera da koristi fabrikaciju (`factory`).

PRIMENA FABRIKACIJE

Da bi servis AnalyticsService mogao da bude korišćen neophodno ga je registrovati u listi provajdera preko produkcione metode useFactory()

Ukratko rečeno, da bi servis `AnalyticsService` mogao da bude korišćen, neophodno je uraditi sledeće:

- kreirati implementaciju koja odgovara interfejsu `AnalyticsImplementation`;
- dodati servis u listu provajdera primenom produkcione metode (fabrikacije) `useFactory()`.

Sledećim listingom je prikazano kako se to radi (datoteka: `src/app/analytics-demo/analytics-demo.module.1.ts`):

```
import { NgModule } from '@angular/core';
import { CommonModule } from '@angular/common';
import {
```

```
Metric,  
AnalyticsImplementation  
} from './analytics-demo.interface';  
import { AnalyticsService } from '../services/analytics.service';  
  
@NgModule({  
  imports: [  
    CommonModule  
  ],  
  providers: [  
    {  
      // `AnalyticsService` je token koji se koristi za umetanje  
      // tapamtit, token je klasa, ali se koristi kao identifikator!  
      provide: AnalyticsService,  
  
      // useFactory je funkcija - svaki vraćeni rezultat funkcije  
      // biće umetnut  
      useFactory() {  
  
        // kreiranje implementacije za log prikazivane događaja  
        const loggingImplementation: AnalyticsImplementation = {  
          recordEvent: (metric: Metric): void => {  
            console.log('Metrika je:', metric);  
          }  
        };  
  
        // kreiranje novog `AnalyticsService` sa implementacijom  
        return new AnalyticsService(loggingImplementation);  
      }  
    },  
    declarations: [ ]  
  })  
export class AnalyticsDemoModule { }
```

DODATNO O SINTAKSI PROVAJDERA KROZ PRIMENU FABRIKACIJE

Neophodno je izolovati jedan detalj sintakse provajdera.

Ako se pogleda detaljno prethodni listing, neophodno je izolovati jedan detalj sintakse provajdera.

```
providers: [  
  { provide: AnalyticsService, useFactory: () => ... }  
]
```

useFactory kao ključ preuzima funkciju i šta god da ta funkcija vrati biće injektovano.

Takođe, primećuje se da je obezbeđena i primena klase `AnalyticsService`. Ponovo, kada se koristi ključ provide na ovaj način, klasa `AnalyticsService` se koristi kao token za identifikaciju šta će biti ubrizgano (umetnuto ili injektovano).

Primenom `useFactory` kreira se objekat tipa `AnalyticsImplementation` koji poseduje jednu funkciju: `recordEvent()`. Preko ovoj funkcije moguće je, na primer, podesiti šta će se desiti kada je događaj sačuvan.

Još jednom je moguće napomenuti da bi ovakva aplikacija verovatno, u realnim uslovima, poslala događaj na platformu Google Analytics ili neki drugi softver za vođenje i obradu evidencije događaja (*event logging software*).

Konačno, obavlja se instanciranje i vraćanje `AnalyticsService` objekta.

ZAVISNOSTI FABRIKACIJE

Ponekad će za funkcionisanje produkcionih metoda morati da postoje dodatne zavisnosti.

Primena produkcionih metoda (fabrikacija) je moćan način za implementaciju koncepta umetanja zavisnosti u `Angular` okviru jer je veoma širok spektar funkcionalnosti koje mogu da budu ugrađene u ovu funkciju. Ponekad će za funkcionisanje produkcionih metoda morati da postoje dodatne zavisnosti. Na primer, neophodno je podesiti klasu Ponekad će za funkcionisanje produkcionih metoda morati da postoje `AnalyticsImplementation` da upućuje HTTP zahteve za određeni URL. Sa ciljem realizovanja navedenog, neophodno je posedovati sledeće:

- Angulat Http klijent;
- Konkretan URL.

U klasi `AnalyticsDemoModule`, koja se čuva u tekućem projektu na lokaciji `/src/app/analytics-demo/analytics-demo.module`, moguće je obaviti sledeća podešavanja u skladu sa navedenim smernicama:

```
import { NgModule } from '@angular/core';
import { CommonModule } from '@angular/common';
import {
  Metric,
  AnalyticsImplementation
} from './analytics-demo.interface';
import { AnalyticsService } from '../services/analytics.service';

// dodato je sledeće ->
import {
  HttpClientModule,
  HttpClient
} from '@angular/common/http';

@NgModule({
  imports: [
```

```
CommonModule,  
HttpClientModule, // <-- dodato  
],  
providers: [  
    // dodaje API_URL provajdera  
    { provide: 'API_URL', useValue: 'http://devserver.com' },  
    {  
        provide: AnalyticsService,  
  
        // dodaje zavisnosti za specifikaciju produpcionih (factory) zavisnosti  
        deps: [ HttpClient, 'API_URL' ],  
  
        // ovde su dodati argumenti  
        // redosled odgovara redosledu zavisnosti  
        useFactory(http: HttpClient, apiUrl: string) {  
  
            // kreiranje implementacije za prikazivanje događaja u logu  
            const loggingImplementation: AnalyticsImplementation = {  
                recordEvent: (metric: Metric): void => {  
                    console.log('Metrika je:', metric);  
                    console.log('Šalje se ka:', apiUrl);  
                    // ... slanje metrike primenom http ...  
                }  
            };  
  
            // kreiranje nove `AnalyticsService` sa implementacijom  
            return new AnalyticsService(loggingImplementation);  
        }  
    },  
],  
declarations: [ ]  
}  
export class AnalyticsDemoModule { }
```

U priloženom listingu je obavljeno dodavanje [HttpModule](#), istovremeno putem [ES6](#) import naredbe (omogućava primene konstanti klase) u [NgModul](#) dekoratoru (omogućava da ova klasa bude dostupna za umetanje zavisnosti).

Dalje, kao provajder je dodat [API_URL](#), a potom i [AnalyticsService](#). Posebna pažnja bi trebalo da se posveti drugom navedenom provajderu. Ovde je dodat nov ključ: [deps](#) koji predstavlja niz tokena umetanja zavisnosti koji će biti rešeni (upareni) i prosleđeni kao argumenti produkcionim funkcijama (fabrikacijama).

VIDEO MATERIJAL 1

[*Angular Dependency Injection - Understanding Providers and Injection Tokens - trajanje 9:34*](#)

Ova lekcija sadrži video materijal. Ukoliko želite da pogledate ovaj video morate da otvorite LAMS lekciju.

▼ Poglavlje 5

Umetanje zavisnoti u Angular aplikaciji

KORACI IZVOĐENJA UMETANJA ZAVISNOSTI

Postoje tri koraka koje je neophodno preći sa ciljem izvođenja umetanja zavisnosti.

Ako se detaljno prođe kroz objekte učenja ove lekcije, moguće je izvući sledeći zadatak - postoje tri koraka koje je neophodno preći sa ciljem izvođenja umetanja zavisnosti u Angular radnom okviru:

1. kreiranje zavisnosti (na primer, kreiranje servisne klase);
2. podešavanje umetanja zavisnosti (registrovanje u Angularu preko NgModule);
3. deklarisanje zavisnosti u komponenti koja je primenjuje.

Prva stvar koju je neophodno učiniti jeste kreiranje servisne klase. Ova klasa ispoljava neko ponašanje koje je neophodno upotrebiti u ostalim komponentama programa. Kreirana servisna klasa je obeležena dekoratorom @Injectable() zato što predstavlja resurs koji je dostupan ostalim komponentama aplikacije preko umetanja zavisnosti.

U zavisnosti od terminologije, provajderi obezbeđuju (kreiraju, instanciraju i tako dalje) servise za umetanje zavisnosti. U Angular okviru kada je neophodno pristupiti servisu, obavlja se umetanje zavisnosti u neku funkciju (često konstruktor) nakon čega će Angular DI okvir locirati i obezbediti traženu zavisnost.

Konačno, kao rezime lekcije, moguće je istaći da koncept umetanja zavisnosti obezbeđuje moćan način upravljanja zavisnostima u Angular veb aplikacijama.

Konačno, možete preuzeti i testirati prateći primer. Potpuno urađen i proveren primer možete preuzeti na kraju ovog objekta učenja u aktivnosti Shared Resources.

DODATNI POGLED NA KONCEPT UMETANJA ZAVISNOSTI

Angular preporučuje da komponente sadrže samo podatke specifične za šablove.

Angular preporučuje da komponente sadrže samo podatke specifične za šablove. Biznis logika i obrada podataka treba da se nalazi u servisima. U ovom slučaju komponente su korisnici servisa i služe da proslede šablonu podatke iz servisa za prikaz. Na neki način komponente samo delegiraju posao servisima i koriste njihove usluge. Da bi se klasa definisala kao servis ona mora da bude opisana dekoratorom `@Injectable()`. Servisi mogu da imaju zavisnost i od drugih servisa, funkcija ili vrednosti pa je tako Česta upotreba servisa koji zavise od servisa HttpClient koji služi za slanje zahteva ka zadatim adresama. Kada je kreirana instanca neke klase, obezbeđivanje svih elemenata od kojih ta klasa zavisi se zove umetanje zavisnosti (eng. `dependency injection`).

U Angularu - u, "umetač" (eng. `injector`) održava skladište servisa i podataka od kojih zavisi funkcionisanje svakog pojedinačnog servisa. Ako neka komponenta zahteva korišćenje nekog servisa, umetač prvo proverava da li postoji instanca tog servisa i ako ne postoji kreira je, a zatim instancu servisa prosleđuje kroz provajdere (eng. `providers`) nazad komponenti na korišćenje. Umotač na osnovu konstruktora komponente određuje koji servisi su toj komponenti potrebni i u koju promenljivu da ih stavi, jer se servisi koji se koriste posleduju kao argumenti konstruktora. U tom slučaju bi kreiranje instance komponente zavisilo i od servisa.

Ovim bi bilo završeno teorijsko izlaganje na temu primene i značaja koncepta umetanja zavisnosti u *Angular* aplikacijama i fokus se prebacuje na izradu konkretnih zadataka kroz pokazne i individualne vežbe.

VIDEO MATERIJAL 2

Angular 8 Tutorial - 18 - Dependency Injection - trajanje 9:24

Ova lekcija sadrži video materijal. Ukoliko želite da pogledate ovaj video morate da otvorite LAMS lekciju.

VIDEO MATERIJAL 3

EP 10.6 - Angular / Dependency Injection & Providers /Providers and viewProviders - 15:01

Ova lekcija sadrži video materijal. Ukoliko želite da pogledate ovaj video morate da otvorite LAMS lekciju.

✓ Poglavlje 6

Materijali za dodatni rad

DODATNI MATERIJALI

Proširite znanje sa predavanja.

1. <https://angular.io/>
2. <https://angular.io/tutorial>
3. <https://www.w3schools.com/angular/>
4. <https://www.tutorialspoint.com/angular4/>
5. <https://nodejs.org/en/>
6. <https://code.visualstudio.com/>

7. https://www.w3schools.com/whatis/whatis_htmldom.asp
8. <https://angular.io/guide/dependency-injection>
9. <http://www.ucim-programiranje.com/2013/02/dependency-injection-inversion-of-control/>
10. <https://www.codingame.com/playgrounds/8003/angular-dependency-injection-tutorial>

✓ Poglavlje 7

Pokazne vežbe 9

UMETANJE ZAVISNOSTI - POKAZNA VEŽBA (TRAJANJE 45 MINUTA)

Cilj vežbe je da upozna studenta kreiranjem servisa i primenom umetanja zavisnosti.

Cilj ove pokazne vežbe jeste vežbanje studenata da koriste koncept umetanja zavisnosti (*dependency injection*) te da se znanje zaokruži nakon teorijskog izlaganja prezentovanih u prethodnim objektima učenja. Izradom individualne vežbe, a posebno domaćeg zadatka, student će steći osnove za samostalno rukovanje ovom veoma važnom programerskom problematikom.

Vežba započinje kreiranjem odgovarajuće komponente koja će poslužiti kao osnov za implementaciju koncepta umetanja zavisnosti u pokaznoj vežbi. Ali pre toga, biće kreiran folder *services* unutar foldera *app* tekućeg projekta (nastavljamo sa projektom kojeg razvijamo iz nedelje u nedelju).

U ovom folderu, biće kreirana datoteka pod nazivom *metServices.ts*. Sledi njen listing:

```
import { Injectable } from "@angular/core";
import { Studije } from "../model/studije";

@Injectable()
export class MetService {

    //umetanje zavisnosti - objekat Studije je umetnut putem konstruktora
    constructor (private studije : Studije){
        this.studije = studije;
    }

    getStudije(): Studije {
        this.studije.naziv = "OAS";
        this.studije.trajanje = 4;
        this.studije.espb = 240;
        return this.studije;
    }
}
```

Ovde je moguće primetiti da je klasa obeležena anotacijom *@Injectable()* što znači da je dostupna ostalim komponentama aplikacije putem koncepta umetanja zavisnosti. Takođe, na umetanju objekta tipa *Studije*, prvi put smo samostalno primenili DI (linija koda 8).

REGISTROVANJE PROVAJDERA UMETANJA ZAVISNOSTI

Registrovanje provajdera umetanja zavisnosti unutar klase AppModule.ts.

Registrovanje provajdera umetanja zavisnosti unutar klase `AppModule.ts` predstavlja korak bez kog koncept umetanja zavisnosti ne bi mogao da funkcioniše.

Sledi detalj navedene klase ku kojem su registrovani provajderi umetanja zavisnosti:

```
//ostatak koda
imports: [
  BrowserModule, ReactiveFormsModule
],
providers: [Studije, MetService],
bootstrap: [AppComponent]
})
export class AppModule { }
```

KREIRANJE NOVE KOMPONENTE

Kreiranje nove komponente koja koristi servis.

Kreiranje nove komponente koja koristi servis je ključni deo tekuće diskusije. Navođenjem instrukcije

```
ng generate component DI
```

kreiramo novu komponentu. Sledi listing klase komponente:

```
import { Component, OnInit } from '@angular/core';
import { Studije } from '../model/studije';
import { MetService } from '../services/metService';

@Component({
  selector: 'app-di',
  templateUrl: './di.component.html',
  styleUrls: ['./di.component.css']
})
export class DIComponent implements OnInit {

  //umetanje zavisnosti putem konstruktora
  constructor(private metService : MetService) {
    this.metService = metService;
  }

  st : Studije = this.metService.getStudije();
```

```
ngOnInit(): void {  
}  
}
```

Klasa injektuje servisni objekat, putem konstruktora (linija koda 13), a zatim koristi njegovu metodu `getStudije()`. Sve informacije iz servisa, na ovaj način, direktno su dostupne klasi kreirane komponente.

Da bi one mogle da budu prikazane, neophodno je dodati odgovarajući kod u šablon komponente, kao na sledeći način:

```
<div style="text-align:left">  
    <h1>  
        Primer umetanja zavisnosti  
    </h1>  
    <table>  
        <tr>  
            <td>  
                {{ st.naziv}}  
            </td>  
        </tr>  
        <tr>  
            <td>  
                {{ st.trajanje }}  
            </td>  
        </tr>  
        <tr>  
            <td>  
                {{ st.espb}}  
            </td>  
        </tr>  
    </table>  
</div>
```

DODAVANJE KOMPONENTE, STILOVI I DEMO

Selektor komponente mora da bude ugrađen u šablon roditeljske komponente.

Da bi sadržaj kreirane komponente mogao da bude prikazan, njen selektor mora da bude ugrađen u šablon roditeljske komponente, baš kao na sledeći način:

```
@Component({  
  
    selector: 'my-app',  
    template: `<app-navbarcomponentnew></app-navbarcomponentnew>  
    <div>
```

```

<h1>{{pageheader}}</h1>
</div>

<app-studentcomponent></app-studentcomponent>
<app-buttongroupcomponent></app-buttongroupcomponent>
<app-forma></app-forma>
<app-di></app-di>
` 

})

export class AppComponent //sledi ostatak koda

```

Da bi tabela kreirane komponente bila stilizovana, u *CSS* datoteku komponente *DI* dodajmo još malo koda:

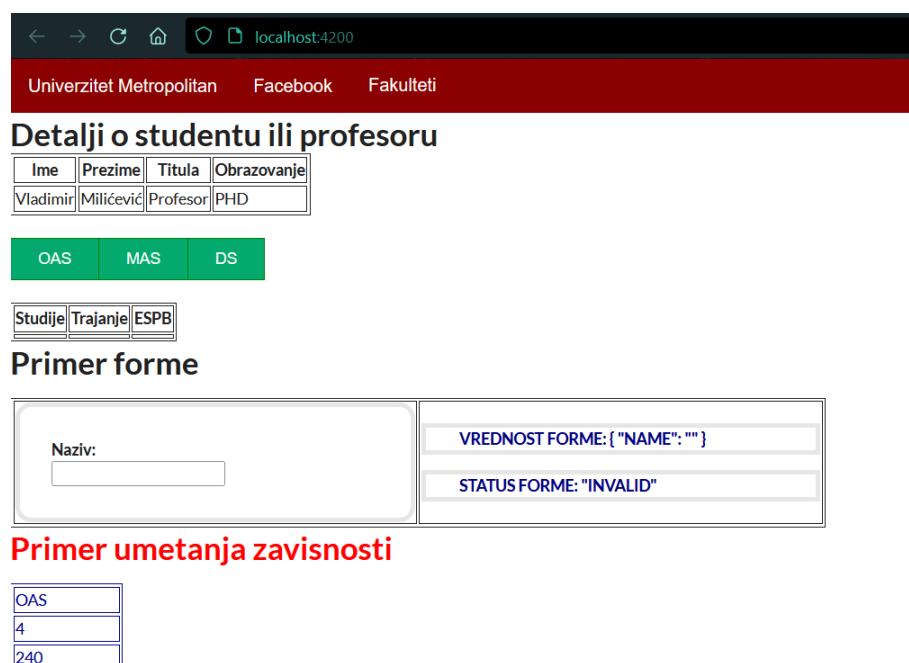
```

H1 {
    color: red;
}

table, th, td {
    color: darkblue;
    border: 1px solid;
    width: 100px;
}

```

Sada je moguće obaviti prevođenje aplikacije i njeno ponovno pokretanje kao što je prikazano sledećom slikom:



Slika 7.1 Demonstracija primene DI [izvor: autor]

✓ Poglavlje 8

Individualna vežba 9

INDIVIDUALNA VEŽBA (TRAJANJE 90 MINUTA)

Na osnovu date ideje samostalno kreirajte i pokrenite Angular DI aplikaciju.

Za izradu individualne vežbe može poslužiti primer urađene aplikacije koja je demonstrirana na sledećem linku: <https://www.codingame.com/playgrounds/8003/angular-dependency-injection-tutorial>. Idejno rešenje aplikacije, može biti prikazano sledećom slikom:



Slika 8.1 Idejno rešenje aplikacije [izvor: autor]

Zadatak za samostalni rad:

1. **Istražite rešenje koje ne primenjuje koncept umetanja zavisnosti;**
2. **Kreirajterešenje koje prilikom kodiranja u potpunosti primenjuje i uvažava koncept umetanja zavisnosti;**
3. **Napravite paralele i obrazložite ukratko prednosti drugog pristupa.**

PRIMER BEZ UMETANJA ZAVISNOSTI

Doradite aplikaciju u skladu sa gradivom obrađenim u predavanjima i pokaznoj vežbi

Na osnovu inicijalnog primera koji ne uvažava koncept umetanja zavisnosti, napravite korekcije i doradite aplikaciju u skladu sa gradivom obrađenim u predavanjima i pokaznoj vežbi.

Zadatak:

Kompjuter čine sledeće komponente:

- *Monitor*;
- *CPU*;
- *KeyBoard*.

Da bi kompjuter softverski bio realizovan, neophodno je kreirati sledeće klase:

- *Computer*;
- *Monitor*;
- *CPU*;
- *Keyboard*.

Vaš prvi zadatak je kreiranje navedenih modelskih klasa na odgovarajućim lokacijama u projektu ([src/app](#) direktorijum).

Predlog:

1. Klasa *Monitor* (datoteka *Monitor.ts*) ima jednu javnu osobinu koja može inicijalno biti podešena na sledeći način:

public monitorNo = 2;

2. Klasa *CPU* (datoteka *CPU.ts*) ima jednu javnu osobinu koja može inicijalno biti podešena na sledeći način:

public cpuNo = 3;

3. Klasa *Keyboard* (datoteka *Keyboard.ts*) ima jednu javnu osobinu koja može inicijalno biti podešena na sledeći način:

public keyboardNo = 1;

PREDLOG IZGLEDA KLASE COMPUTER

Klasa Computer je agregatna klasa koja sadrži objekte prethodnih klasa.

Klasa *Computer* (datoteka: *Computer.ds*) je agregatna klasa koja sadrži objekte prethodnih klasa. Njen konstruktor može da ima sledeći oblik:

```
constructor() {
    this.monitor = new Monitor();
    this.cpu = new CPU();
    this.keyboard = new Keyboard();
}
```

Predlaže se dodavanje sledeće metode klase *Computer* odmah iza konstruktora.

```
public description = 'This Matrix computer';
complete() {
    return `${this.description} has ` +
        `${this.monitor.monitorNo} monitors, ${this.cpu.cpuNo} cpus and,
${this.keyboard.keyboardNo} keyboard.`;
}
```

Za kompletiranje komponente *Computer* neophodno je importovanje kreiranih klasa *Monitor*, *CPU* i *KeyBoard* sa ciljem kreiranja potpuno kompletног *Matrix* računara. Na osnovu konstruktora klase *Computer*, koji je priložen prvim listingom, vidi se da su kreirane tri instance klasa čiji je predlog dat u prethodnoj sekciji.

Ovde je bitno navesti da, po ovom scenariju, klasa *Computer* će biti u potpunosti zavisna od ostalih predloženih klasa: *Monitor*, *CPU* i *KeyBoard*. Budući da će kreiranje objekata biti obavljeno unutar konstruktora, objekti klasa: *Monitor*, *CPU* i *KeyBoard* ni na koji način nisu razdvojeni od klase *Computer*.

Sledi predlog inicijalne strukture klase glavne komponente i njenog šablonu.

Vaš zadatak: Modifikovati predlog i primeniti umetanje zavisnosti, a to znači da je cilj da se kreiranje prikazanih instanci obavi na drugom mestu, a ne u konstruktoru klase Computer,

Predlaže se sledeći sadržaj datoteke **app.component.ts**:

```
import { Component } from '@angular/core';
import { Computer } from './Computer';

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent {
  public computer: Computer;

  constructor(){
    this.computer = new Computer();
  }
  makeComputer(){
    return this.computer.complete();
  }
}
```

```
}
```

Predlaže se sledeći sadržaj datoteke **app.component.html**:

```
<div style="text-align:center">
  <h1>
    {{makeComputer()}}!!
  </h1>
</div>
```

✓ Poglavlje 9

Domaći zadatak 9

DOMAĆI ZADATAK (PREDVIĐENO VREME 180 MIN)

Samostalna izrada domaćeg zadatka.

Na projekat nastavljen prethodnim domaćim zadatkom dodati sledeće funkcionalnosti koje uključuju DI:

1. dodati `RoomService` unutar kog ćete definisati dodatnu metodu `getPrice(numberOfNights: number)` i
2. izračunati cenu na osnovu broja noći.
3. koristiti koncept umetanja zavisnosti prilikom izrade DZ.

Domaći zadatak dodati na [Github](#) pod "commitom" `IT255-DZ09` i poslati obaveštenje predmetnom asistentu o postavljenom domaćem zadatku.

▼ Poglavlje 10

Zaključak

ZAKLJUČAK

Lekcija se bavila konceptom umetanja zavisnosti u savremenim veb aplikacijama.

Cilj lekcije je bio fokusiranje na veoma korišćeni koncept umetanja zavisnosti u savremenim aplikacijama. O ovom konceptu je detaljno diskutovano u objektima učenja ove lekcije.

Posebno je istaknuto, a kasnije i u lekciji pokazano, kako napreduje gradivo ovog predmeta i kako se aktuelnim lekcijama uvode novi koncepti, programi, koji prate izlaganje, postaju sve veći i kompleksniji. U tom svetu se i javila potreba da komponente komuniciraju sa ostalim modulima programa. **U situaciji kada je za izvršavanje modula A neophodan modul B, kaže se da je B zavisnost za A.**

Jedan od opštih načina za pristupanje zavisnostima predstavlja importovanje odgovarajuće datoteke, što je u brojnim slučajevima i dovoljno. Međutim, postoje i situacije kada je neophodno obezbediti zavisnosti na sofisticiraniji način. U lekciji je istaknuto da to može biti neki od sledećih scenarija:

- zamena klase B klasom *MockB* tokom izvođenja testiranja;
- deljenje jedinstvene instance klase B preko cele aplikacije (*Singleton šablon - pattern*);
- kreiranje nove instance klase B svaki put kada se koristi (*Producioni šablon - Factory pattern*).

Lekcija je identifikovala rešenje. Navedene probleme je moguće rešiti primenom koncepta umetanja zavisnosti. DI može biti definisano na sledeći način - **Umetanje zavisnosti (dependency injection - DI)** prestavlja sistem koji omogućava da delovi nekog programa budu dostupni ostalim delovima tog programa, a programeri podešavaju načine kako se to izvodi.

Posebno je pokazano da umetanje zavisnosti može da ima neki od sledeća dva oblika:

- opisivanje dizajn šablona (kojeg koriste brojni radni okviri) i
- specifična implementacija za *DI* ugrađena u *Angular* okvir.

Glavna korist primene umetanja zavisnosti ogleda se u činjenici da klijent komponenta ne mora da bude upućena u način kreiranja same zavisnosti. Sve što klijent komponenta mora da zna jeste kako da komunicira sa zavisnošću.

Za demonstraciju gradiva ove lekcije je pažljivo izabran i uveden pokazni primer za lakše razumevanje diskusije koja je, u početku, mogla studentima da bude apstraktna i nerazumljiva.

DODATNA ZAKLJUČNA RAZMATRANJA

Kratak rezime primene servisa i koncepta umetanja zavisnosti.

Kratak rezime primene servisa i koncepta umetanja zavisnosti:

- Za podatke ili logiku koja nije povezana sa određenim prikazom, a koju želimo da podelimo sa više različitih komponenti, kreiramo servisnu klasu (*servis*);
- Umetanje zavisnosti (*Dependency injection*) je proces u kome jedna komponenta definiše zavisnost od drugih komponenata;
- Definiciji servisne klase uvek prethodi dekorator `@Injectable()`;
- Dekorator pruža metapodatke koji omogućavaju servisu da ubaci u neku komponentu klijenta zavisnost (npr. komponenti koja pristupa veb serveru pomoću HTTP request, ubaciti HTTP servise u komponentu).
- Injektiranje zavisnosti definiše i dinamički injektira objekat zavisnosti u drugi objekat, pa sve funkcije koje obezbeđuje objekat zavisnosti postaju dostupne.
- Angular obezbeđuje injektiranje zavisnosti pomoću *provajdera* i *servisa* injektora.
- Da bi se koristilo injektiranje zavisnosti na drugoj direktivi ili komponenti, potrebno je da se u okviru modula za aplikaciju doda naziv klase direktive ili komponente u metapodatke *declarations* u dekoratoru `@NgModule`, čime se niz direktiva uvozi u aplikaciju.
- Da bi se u *Angularu* uneli podaci u drugu direktivu ili komponentu, potrebno je da se uveze dekorator `Input` iz paketa `@angular/core`:

```
import {Component, Input} from '@angular/core';
```

- Kada se uveze ovaj dekorator, možemo da počnemo da definišemo podatke koje ćemo da unesemo u direktivu. Definisanje dekoratora se radi korišćenjem `@input()`, koji koristi string podatak kao parametar.

```
@Input('name') personName: string;
```

- U *Angular* okviru, za registrovanje zavisnosti neophodno je povezati zavisnost sa nekim objektom koji će je identifikovati. Ova identifikacija je poznata kao token zavisnosti (*dependency token*).
- U *Angularu* - u, "umetač" (eng. *injector*) održava skladište servisa i podataka od kojih zavisi funkcionisanje svakog pojedinačnog servisa.
- Provajder (*provider*) - povezuje token (koji može biti string ili klasa) sa listom zavisnosti. Navodi Angular kako da kreira objekat na osnovu tokena;
- *Umotač ili injektor (injector)* - sadrži skup povezivanja (binding) i odgovoran je za rešavanje zavisnosti i njihovo umetanje prilikom kreiranja objekata;
- Konačno, zavisnost (*dependency*) je vrednost koja je umetnuta.

LITERATURA

Za pripremu lekcije korišćena je aktuelna pisana i elektronska literatura.

Pisana literatura:

1. Nate Murray, Felipe Coury, Ari Lerner, Carlos Taborda, ng-Book – The Complete Book on Angular 6, Fullstack.io, 2018
2. Cody Lindley, Frontend Handbook – 2017, Frontend masters, 2017
3. Sandeep Panda, AngularJS – From Novice to Ninja, SitePoint Pty. Ltd, 2014

Elektronska literatura:

4. <https://angular.io/>
5. <https://angular.io/tutorial>
6. <https://www.w3schools.com/angular/>
7. <https://www.tutorialspoint.com/angular4/>
8. <https://nodejs.org/en/>
9. <https://code.visualstudio.com/>

10. https://www.w3schools.com/whatis/whatis_htmldom.asp
11. <https://angular.io/guide/dependency-injection>
12. <http://www.ucim-programiranje.com/2013/02/dependency-injection-inversion-of-control/>
13. <https://www.codingame.com/playgrounds/8003/angular-dependency-injection-tutorial>



IT255 - VEB SISTEMI 1

HTTP i Angular

Lekcija 10

PRIRUČNIK ZA STUDENTE

IT255 - VEB SISTEMI 1

Lekcija 10

HTTP I ANGULAR

- ✓ HTTP i Angular
- ✓ Poglavlje 1: Primena angular/common/http
- ✓ Poglavlje 2: Kreiranje YouTubeSearchComponent komponente
- ✓ Poglavlje 3: Modifikacija primera za Angular13+
- ✓ Poglavlje 4: angular/common/http API - ostali HTTP zahtevi
- ✓ Poglavlje 5: Kreiranje i testiranje zahteva
- ✓ Poglavlje 6: Angular i HTTP- video matrijali
- ✓ Poglavlje 7: Dodatni nastavni materijali
- ✓ Poglavlje 8: Pokazna vežba 10
- ✓ Poglavlje 9: Individualna vežba 9
- ✓ Poglavlje 10: Domaći zadatak 10
- ✓ Zaključak

Copyright © 2017 – UNIVERZITET METROPOLITAN, Beograd. Sva prava zadržana. Bez prethodne pismene dozvole od strane Univerziteta METROPOLITAN zabranjena je reprodukcija, transfer, distribucija ili memorisanje nekog dela ili čitavih sadržaja ovog dokumenta., kopiranjem, snimanjem, elektronskim putem, skeniranjem ili na bilo koji drugi način.

Copyright © 2017 BELGRADE METROPOLITAN UNIVERSITY. All rights reserved. No part of this publication may be reproduced, stored in a retrieval system or transmitted in any form or by any means, electronic, mechanical, photocopying, recording, scanning or otherwise, without the prior written permission of Belgrade Metropolitan University.

✓ Uvod

UVOD

Angular poseduje HTTP biblioteku koja može biti pogodna za obavljanje poziva ka eksternim API

Radni okvir *Angular* distribuira vlastitu *HTTP biblioteku* koja može biti veoma pogodna za obavljanje poziva ka eksternim *API* - jima.

Kada se obavljaju pozivi ka eksternim serverima, neophodno je obezbediti da korisniku kontinuiranu interakciju sa stranicom veb aplikacije. To praktično znači, da je neophodno spričiti da se stranica "zamrzne" dok ne stigne HTTP odgovor sa servera. Da bi navedeno bilo moguće realizovati, *HTTP zahtevi moraju biti asinhroni*.

Rad sa asinhronim kodom, a to ima i istorijsko uporište u programiranju, može biti složeniji nego rukovanje sinhronim kodom.

U JavaScript jeziku, postoje tri pristupa za pomoć u rukovanju asinhronim kodom:

1. povratni pozivi (*callback*);
2. obećanja (*promises*);
3. osmatranja (*observable*).

U Angular okviru preferirani metod manipulisanja asinhronim kodom predstavlja treći slučaj - osmatranje, na čemu će biti i bazirano izlaganje u ovoj lekciji.

Lekcija će pokriti tri ključne stvaki:

1. analiza i diskusija osnovnog *HttpClient* primera;
2. kreiranje *YouTube* komponente za pretragu;
3. diskusija o API detaljima vezanim za *HttpClient* biblioteku.

Savladavanjem ove lekcije student će razumeti i biti u mogućnosti da rukuje asinhronim HTTP zahtevima u Angular aplikacijama.

Ova lekcija sadrži video materijal. Ukoliko želite da pogledate ovaj video morate da otvorite LAMS lekciju.

▼ Poglavlje 1

Primena angular/common/http

HTTP U ANGULARU

Podrška radu sa HTTP protokolom je u Angularu izdeljena u unutar posebnog modula.

Podrška radu sa [HTTP](#) protokolom je u Angularu izdeljena u unutar posebnog modula. To, najjednostavnije rečeno, znači ukoliko je neophodna navedena podrška za kreiranje koda, tekućeg [Angular](#) projekta, neophodno je obaviti uvoz iz paketa [@angular/common/http](#), na sledeći način:

```
import {  
  // NgModule za upotrebu @angular/common/http  
  HttpClientModule,  
  
  // konstante klase  
  HttpClient  
} from '@angular/common/http';
```

Ova lekcija sadrži video materijal. Ukoliko želite da pogledate ovaj video morate da otvorite LAMS lekciju.

UVOZ IZ ANGULAR/COMMON/HTTP

HttpClientModule predstavlja kolekciju pogodnih modula.

Ukoliko je potrebno da u veb aplikaciji koja se kreira primenu nađe i podrška za rad sa HTTP protokolom, prvi korak koji je neophodno obaviti jeste uvoz biblioteke [HttpClientModule](#) koja predstavlja kolekciju pogodnih modula.

Kao i u prethodnim lekcijama biće uveden adekvatan primer za lakše razumevanje i praćenje gradiva. Primer se jednostavno naziva [http](#) i moguće ga je preuzeti na kraju lekcije u sekciji [Shared Resources](#).

Dakle, prvi korak jeste importovanje navedenih modula na način urađen u sledećem listingu koji predstavlja deo sadržaja datoteke [app.module.ts](#).

```
import { BrowserModule } from '@angular/platform-browser';  
import { NgModule } from '@angular/core';  
import { FormsModule } from '@angular/forms';  
import { HttpClientModule } from '@angular/common/http';
```

U narednom koraku, u dekoratoru `@NgModule`, neophodno je dodati `HttpClientModule` u listu importovanih modula. Ovim se postiže efekat mogućnosti umetanja `Http` (i nekoliko drugih modula) unutar komponenata Angular aplikacije.

```
@NgModule({
  declarations: [
    AppComponent,
    SimpleHttpComponent,
    MoreHttpRequestsComponent,
    YouTubeSearchComponent,
    SearchResultComponent,
    SearchBoxComponent
  ],
  imports: [
    BrowserModule,
    FormsModule,
    HttpClientModule // <-- upravo ovde
  ],
  providers: [youTubeSearchInjectables],
  bootstrap: [AppComponent]
})
export class AppModule {}
```

Sada je moguće obaviti umetanje `HttpClient` servisa u komponente ili bilo gde drugo u programu gde se koristi umetanje zavisnosti, na primer kao na sledećoj slici:

```
class MojaKomponenta {
  constructor(public http: HttpClient) {
  }

  makeRequest(): void {
    // uradi nešto sa this.http ...
  }
}
```

Slika 1.1 Umetanje HttpClient putem konstruktora klase komponente [izvor: autor]

RUKOVANJE OSNOVNIM ZAHTEVOM

Pokazivanje rukovanja osnovnim GET HTTP zahtevom

U nastavku je cilj pokazivanje rukovanja osnovnim `GET` HTTP zahtevom. Zbog svega navedenog iskristalisala jedna ideja - prosleđivanje jednostavnog `GET` zahteva ka `jsonplaceholder API` (<https://jsonplaceholder.typicode.com/>). Neophodno je obaviti sledeće:

1. obezbediti dugme za poziv konkretnе metode `makeRequest()`;
2. metoda `makeRequest()` će pozvati `http` biblioteku za izvođenje `GET` zahteva nad izabranim `API`;
3. kada se vrati odgovor na prosleđeni zahtev, biće ažurirana vrednost `this.data` rezultatima u formi podataka koji će se koristiti za kreiranje pogleda veb aplikacije.

Sledećom slikom je prikazan izlaz u formi kreiranog pogleda na osnovu vraćenog odgovora na prosleđeni *GET* zahtev ka *API*.

Angular HTTP primer

Osnovni zahtev

```
Uputi zahtev
{
  "userId": 1,
  "id": 1,
  "title": "sunt aut facere repellat provident occaecati excepturi optio reprehenderit",
  "body": "quia et suscipit\nrecusandae consequuntur expedita et cum\nreprehenderit molestiae ut ut quas totam\nnostrum rerum est autem sunt rem even"
}
```

Slika 1.2 Izvršavanje osnovnog GET zahteva [izvor: autor]

KREIRANJE DEFINICIJE KOMPONENTE SIMPLEHTTPCOMPONENT

Importovanje odgovarajućih modula i specifikacija selektora unutar dekoratora komponente.

Prethodnom slikom je demonstrirana funkcionalnost komponente aplikacije koja bi u narednom izlaganju trebalo da bude definisana i analizirana. Prvo što je potrebno učiniti jeste importovanje odgovarajućih modula i specifikacija selektora unutar dekoratora komponente *@Component* koja se u projektu čuva na lokaciji *src/app/simple-http/simple-http.component.ts*.

```
import { Component, OnInit } from '@angular/core';
import { HttpClient } from '@angular/common/http';

@Component({
  selector: 'app-simple-http',
  templateUrl: './simple-http.component.html'
})
export class SimpleHttpComponent implements OnInit {
  data: Object;
  loading: boolean;

  constructor(private http: HttpClient) {}
```

Priloženim listingom nije završena definicija klase komponente. O ostatku listinga, između ostalog i pomenutoj metodi *makeRequest()*, biće detaljno govora u nastavku.

KREIRANJE ŠABLONA KOMPONENTE SIMPLEHTTPCOMPONENT

Za nastavak analize od presudnog značaja je kreiranje šablona komponente.

U prethodnom izlaganju data je delimična definicija klase kreirane komponente *SimpleHttpComponent*. Za nastavak analize od presudnog značaja je kreiranje šablonu komponente *SimpleHttpComponent*.

Sledećim listingom dat je *HTML* kod šablonu koji se čuva u datoteci *src/app/simple-http/simple-http.component.html*:

```
<h2>Osnovni zahtev</h2>
<button type="button" (click)="makeRequest()">Uputi zahtev</button>
<div *ngIf="loading">loading...</div>
<pre>{{data | json}}</pre>
```

Kada se pogleda priloženi listing, moguće je primetiti da šablon sadrži tri zanimljiva detalja:

- dugme "Uputi zahtev";
- indikator učitavanja zahetva (loading);
- podatke.

Kontrola dugme je povezana akcijom (click) sa pozivom metode *makeRequest()* kontrolera (klase) komponente. Ova metoda će biti definisana uskoro,

Indikator pokazuje korisniku da se prosleđeni zahtev učitava, prezentacijom stringa "loading...", ukoliko objektna promenljiva *loading* poseduje vrednost *true* u direktivi grananja *ngIf*.

Promenljiva *data* predstavlja objekat. Odličan način rukovanja objektima jeste njihovo prikazivanje u *JSON (JavaScript Object Notation)* formatu. Na ovaj način dobija se veoma čitljiv i lako upotrebljiv format prikaza objekata.

KREIRANJE SIMPLEHTTPCOMPONENT KONTROLERA

Ono što klasi nedostaje su konstruktor i metoda makeRequest() koja šalje GET zahtev ka serveru.

Ono što trenutno postoji, a odnosi se na primer koji služi kao podrška trenutnom izlaganju, jeste osnovna definicija klase *src/app/simple-http/simple-http.component.ts*:

```
export class SimpleHttpComponent implements OnInit {
  data: Object;
  loading: boolean;
```

Klasa poseduje dve objektne promenljive: *data* i *loading* koje će biti upotrebljene da sačuvaju API povratnu vrednost i indikator učitavanja zahteva, respektivno.

Ono što klasi nedostaje, da bi njena definicija bila zaokružena, su:

- konstruktor;
- metoda *makeRequest()* koja šalje *GET* zahtev ka serveru;

Dakle, u nastavku je prvo neophodno dodati u kod konstruktor, kao na sledeći način:

```
constructor(private http: HttpClient) {}
```

Telo konstruktora je prazno ali ipak, konstruktor ovde obavlja jednu važnu ulogu - vrši umetanje jednog od ključnih modula *HttpClient*.

Sada je sve spremno za kodiranje metode kojom će biti omogućeno kreiranje prvog HTTP zahteva. Ova metoda kontrolera se naziva *makeRequest()* i njen kod je priložen u sledećem listingu:

```
makeRequest(): void {
    this.loading = true;
    this.http
        .get('https://jsonplaceholder.typicode.com/posts/1')
        .subscribe(data => {
            this.data = data;
            this.loading = false;
        });
}
```

ANALIZA KODA KONTROLERA

this.http.get() prihvata URL prema kojem se upućuje zahtev.

Kada se pozove funkcija *makeRequest()*, prva stvar koja se dešava je podešavanje vrednosti objektne promenljive: *this.loading = true*. Ovom akcijom će biti uključen, i prikazan u pogledu, indikator učitavanja zahteva (slika 3).

Prosleđivanje HTTP zahteva je veoma jednostavno: poziva se *get()* metoda na sledeći način: *this.http.get()* kojoj se prosleđuje, kao argument, *URL* prema kojem se upućuje zahtev. Metoda kao rezultat vraća *Observable* objekat koji osmatra promene u podacima nakon prosleđenog zahteva.

Dalje, *Observable* objekat, da bi obavio svoju dužnost, prijavljuje se da osmatra promene pozivom funkcije *subscribe()* - slično kao *Promise* u nativnom *JavaScript*-u.

Kada se obrađeni zahtev *http.request* vrati od servera, ulazni tok će emitovati odgovor (*response*) kao instancu tipa *Object*. Primenom *JSON*-a vrši se raspakivanje tela objekta čija se vrednost pridružuje objektnoj promenljivoj *this.data*. Budući da je odgovor dobijen, indikator slanja zahteva više nije potreban i podešava se *this.loading=false*. Dobijen odgovor se u pogledu prikazuje izvršavanjem izraza *{ {data | json} }* (slika 4).

Angular HTTP primer

Osnovni zahtev



Slika 1.3 Indikator slanja zahteva [izvor: autor]

Angular HTTP primer

Osnovni zahtev

```
Upisi zahtev
{
  "userId": 1,
  "id": 1,
  "title": "sunt aut facere repellat provident occaecati excepturi optio reprehenderit",
  "body": "quia et suscipit\\nsuscipit recusandae consequuntur expedita et cum\\nreprehenderit molestiae ut ut quas totam\\nnostrum rerum est autem sunt rem even
}
```

Slika 1.4 Dobijeni odgovor od servera [izvor: autor]

ZAOKRUŽENA DEFINICIJA KOMPONENTE SIMPLEHTTPCOMPONENT

Komponenta je prikazivana iz delova, a sada je objedinjena u celinu.

U prethodnom izlaganju je kreirana komponenta SimpleHttpComponent, a pogotovo njena kontrolerska klasa, prikazivana iz delova koji su posebno obrađivani i diskutovani.

Za sticanje šire slike je, ipak, potrebno delove sklopiti i prikazati u formi funkcionalne celine.

Sledi listing objedinjene klase komponente (datoteka: [src/app/simple-http/simple-http.component.ts](#)):

```
import { Component, OnInit } from '@angular/core';
import { HttpClient } from '@angular/common/http';

@Component({
  selector: 'app-simple-http',
  templateUrl: './simple-http.component.html'
})
export class SimpleHttpComponent implements OnInit {
  data: Object;
  loading: boolean;

  constructor(private http: HttpClient) {}

  ngOnInit() {}

  makeRequest(): void {
    this.loading = true;
    this.http
      .get('https://jsonplaceholder.typicode.com/posts/1')
      .subscribe(data => {
        this.data = data;
        this.loading = false;
      });
  }
}
```

▼ Poglavlje 2

Kreiranje YouTubeSearchComponent komponente

POKUŠAJ SLOŽENIJEG API POZIVA

Biće kreirana komponenta koja sadrži funkcionalnost pretrage YouTube- a kao tipa podataka.

U prethodnom primeru je pokazano kako Angular web aplikacija upućuje jednostavan GET zahtev ka serveru i kako se prikazuje dobijeni odgovor u šablonu komponente.

U nastavku lekcije, cilj je da se uključe naprednije funkcionalnosti u aplikaciju, a jedna od njih može da bude mehanizam pretrage - biće kreirana komponenta koja sadrži funkcionalnost pretrage YouTube - a kao tipa podataka. Kada je pretraga završena, kao rezultat će biti prikazana lista sa odgovarajućim video klipovima (slika 1) zajedno sa opisom i odgovarajućim linkom za svaki od pronađenih klipova,

Sledećom slikom je prikazan rezultat pretrage, u aplikaciji koja predstavlja primer za podršku izlaganja lekcije, za upit pretrage "državni posao epizode".



Slika 2.1 Rezultat pretrage komponente aplikacije [izvor: autor]

U svetu realizovanja postavljenih ciljeva, nove komponente aplikacije, biće kreirano nekoliko stvari:

1. Objekat SearchResult će čuvati podatke koji se dobijaju iz svakog rezultata pretrage;
2. Servis YouTubeSearchService će upravljati API zahtevima ka serveru YouTube i obavljaće konverziju rezultata u niz SearchResult[];
3. Komponenta SearchBoxComponent će imati zadatak poziva YouTube servisa;
4. Komponenta SearchResultComponent će kreirati specifičan SearchResult;

5. Komponenta *YouTubeSearchComponent* će enkapsulirati celokupnu aplikaciju za pretragu *YouTube*-a i kreirati i prikazati listu rezultata pretrage.

KLASA SEARCHRESULT

Klasa je model za komponentu i biće pogodan način čuvanja informacija - rezultata pretrage.

Razvoj komponente, dobiće zvaničan naziv *you-tube-search*, započinje kreiranjem osnovne klase *SearchResult*. Ova klasa će praktično predstavljati model za komponentu i biće pogodan način čuvanja specifičnih polja koja odgovaraju rezultatima pretrage. Kod klase se čuva u datoteci */src/app/you-tube-search/search-result.model.ts* i priložen je sledećim listingom:

```
/**  
 * SearchResult je struktura podataka koja čuva pojedinačne  
 * rezultate YouTube video pretrage  
 */  
export class SearchResult {  
    id: string;  
    title: string;  
    description: string;  
    thumbnailUrl: string;  
    videoUrl: string;  
  
    constructor(obj?: any) {  
        this.id          = obj && obj.id          || null;  
        this.title      = obj && obj.title      || null;  
        this.description = obj && obj.description || null;  
        this.thumbnailUrl = obj && obj.thumbnailUrl || null;  
        this.videoUrl   = obj && obj.videoUrl   ||  
                         `https://www.youtube.com/watch?v=${this.id}`;  
    }  
}
```

Posebnu pažnju bi trebalo obratiti na deo koda koji je predstavljen konstruktorom klase. Uočava se prvi put šablon u formi *obj?: any*, tako da je ovde prioritet da se razotkrije njegova uloga. Navedeni šablon, zapravo, dozvoljava simuliranje argumenata ključnih reči pretrage. Ideja je da je moguće kreirati nov *SearchResult* objekat i obaviti prosleđivanje objekta koji sadrži specificirane ključne reči.

Jedino što ovde treba istaći je da se konstruiše *videoUrl* koristeći teško kodirani (*hard coded*) URL format. Možda bi ovo bi bilo dobro promeniti u funkciju koja uzima više argumenata ili bi trebalo upotrebiti *id* videa, direktno u HTML šablonu, za kreiranje traženog URL, ako se za to ukaže potreba.

KREIRANJE SERVISA YOUTUBESEARCHSERVICE - API

Za kreiranje navedenog servisa biće neophodno koristiti YouTube v3 search API.

U nastavku sledi kreiranje servisa koji će komponenti omogućiti izvođenje pretrage nad *YouTube* platformom. Za kreiranje navedenog servisa biće neophodno koristiti *YouTube v3 search API* (<https://developers.google.com/youtube/v3/docs/search/list>). Da bi navedeni API mogao da bude upotrebljen neophodno je nabaviti API ključ. U primeru, koji je priložen uz ovu lekciju, ugrađen je API ključ kojeg je moguće koristiti. Međutim, kada student dođe u fazu čitanja i izučavanja ove lekcije, upotreba navedenog ključa je možda premašila dozvoljeni broj korišćenja. Ukoliko se to desi, neophodno je da studenti nabave vlastiti API ključ.

Da bi student nabavio vlastiti API ključ neophodno je da konsultuje sledeću dokumentaciju: https://developers.google.com/youtube/registering_an_application#Create_API_Keys. Zbog jednostavnosti rada na ovom projektu, registrovan je serverski ključ ali bi verovatno trebalo koristiti i ključ za veb pregledač ukoliko je cilj publikovanje JavaScript koda na Internetu (*online*).

Za početak je neophodno podesiti dve konstante preko kojih se *YouTubeSearchService* mapira sa API ključem i API URL:

```
let YOUTUBE_API_KEY: string = "XXX_YOUR_KEY_HERE_XXX";
let YOUTUBE_API_URL: string = "https://www.googleapis.com/youtube/v3/search";
```

KREIRANJE SERVISA YOUTUBESEARCHSERVICE I UMETANJE ZAVISNOSTI

Konstante API ključa moraju da imaju sposobnost primene u umetanju zavisnosti.

U izvesnim situacijama javiće se potreba za testiranjem aktuelne aplikacije. Jedna od stvari koja bi mogla da se javi prilikom testiranja da nije nužno da se aplikacija testira samo u fazi produkcije, već se testiranje obavlja kroz sve faze životnog ciklusa razvoja aplikacije i za korišćeni razvojni API.

Da bi podešavanje ovakvog okruženja bilo uspešno, navedene konstante moraju da imaju sposobnost primene u umetanju zavisnosti (*injectable*).

Postavlja se pitanje: **zašto se koriste navedene konstante preko umetanja zavisnosti, a ne na uobičajen način?** Odgovor leži u činjenici da ako su one dostupne za umetanje u aplikaciji, tada:

1. moguće je imati kod koji injektuje pravu konstantu za dato okruženje i vreme izvršavanja;
2. moguće je jednostavno menjanje umetnute vrednosti tokom vremena testiranja.

Umetanjem navedenih vrednosti, dobija se mnogo veća fleksibilnost u radu u pogledu njihovih vrednosti. Da bi bilo omogućeno korišćenje navedenih vrednosti za umetanje u ostalim delovima aplikacije, koristi se sintaksa `{ provide: ... , useValue: ... }` na sledeći način (datoteka:[/src/app/you-tube-search/you-tube-search.injectables.ts](#)):

```

import {
  YouTubeSearchService,
  YOUTUBE_API_KEY,
  YOUTUBE_API_URL
} from './you-tube-search.service';

export const youTubeSearchInjectables: Array<any> = [
  {provide: YouTubeSearchService, useClass: YouTubeSearchService},
  {provide: YOUTUBE_API_KEY, useValue: YOUTUBE_API_KEY},
  {provide: YOUTUBE_API_URL, useValue: YOUTUBE_API_URL}
];

```

Primenom prikazanog koda specificirano je da se povezivanje `YOUTUBE_API_KEY` sa vrednošću nabavljenog API ključa (`YOUTUBE_API_KEY`) obavlja preko umetanja zavisnosti. Isto važi i za `YOUTUBE_API_URL`.

KOREKCIJE KODA GLAVNE KOMPONENTE APLIKACIJE

Za injekovanje servis mora da

U ovom trenutku je važno prisetiti se da, ukoliko je nešto dostupno za umetanje u različitim modulima aplikacije, neophodno je da bude registrovano pod ključem `providers` u klasi obeleženoj dekoratorom `@NgModule`. Budući da se vrši eksportovanje `youTubeServiceInjectables`, u datoteci glavne komponente `app.module.ts` neophodno je dodati linije koda koje su istaknute na sledećoj slici:

```

// http/app.ts
import { HttpClientModule } from '@angular/common/http';
import { youTubeServiceInjectables } from "components/YouTubeSearchComponent";

// ...
// još koda dole
// ...

@NgModule({
  declarations: [
    HttpApp,
    // ostalo ....
  ],
  imports: [ BrowserModule, HttpClientModule ],
  bootstrap: [ HttpApp ],
  providers: [
    youTubeServiceInjectables // <--- upravo ovde
  ]
})
class AppModule {}

```

Slika 2.2 Registrovanje YouTubeServiceInjectables u AppModule [izvor: autor]

Sada je u potpunosti omogućeno umetanje **YOUTUBE_API_KEY** (iz *YouTubeServiceInjectables*) umesto direktnе upotrebe promenljive.

KREIRANJE KONSTRUKTORA ZA YOUTUBESEARCHSERVICE

Kao što je već dobro poznato, iz prethodnog izlaganja, servis se kreira pravljjenjem servisne klase.

Kao što je već dobro poznato, iz prethodnog izlaganja, servis **YouTubeSearchService** se kreira pravljjenjem servisne klase. Sledećim listingom je dat kod vezan za inicijalni razvoj navedene klase:

```
/**  
 * YouTubeService se povezuje na YouTube API  
 * Pogledajte: * https://developers.google.com/youtube/v3/docs/search/list  
 */  
@Injectable()  
export class YouTubeSearchService {  
    constructor(  
        private http: HttpClient,  
        @Inject(YOUTUBE_API_KEY) private apiKey: string,  
        @Inject(YOUTUBE_API_URL) private apiUrl: string  
    ) {}
```

Primena anotacije **@Injectable** dozvoljava da se konkretnе vrednosti umetnu u konstruktor klase.

U konstruktor su, putem umetanja zavisnosti, dodate tri stvari:

1. **HttpClient**;
2. **YOUTUBE_API_KEY**;
3. **YOUTUBE_API_URL**.

Važno je primetiti da se iz sva tri argumenta kreiraju objektne promenljive, a to znači da im je moguće pristupiti preko: **this.http**, **this.apiKey**, i **this.apiUrl**, respektivno.

Takođe, umetanje zavisnosti je obavljeno na eksplicitan način preko sintakse **@Inject(YOUTUBE_API_KEY)**.

DODAVANJE FUNKCIJE PRETRAGE U SERVISNU KLASU

Da bi obavljanje pretrage bilo moguće neophodno je u servis dodati odgovarajuću funkciju.

Aktuelni primer još ne poseduje traženu funkcionalnost pretrage video klipova preko YouTube platforme. Da bi navedeno bilo moguće, neophodno je u servisnu klasu, nakon konstruktora, dodati odgovarajuću funkciju pod nazivom `search()`. Funkcija će kao argument da uzme string koji odgovara upitu pretrage, a vratiće objekat `Observable` koji će emitovati tok `SearchResult[]`. To znači, u praksi, da svaka emitovana stavka predstavlja niz tipa `SearchResults`.

Sledi prvi deo listinga navedene funkcije:

```
search(query: string): Observable<SearchResult[]> {
    const params: string = [
        `q=${query}`,
        `key=${this.apiKey}`,
        `part=snippet`,
        `type=video`,
        `maxResults=10`
    ].join('&');
    const queryUrl = `${this.apiUrl}?${params}`;
```

Iz priloženog listinga se vidi da je kreiranje `queryUrl` urađeno na manuelan način. Jednostavno, procedura je započela dodavanjem parametra `query` funkcije `search()` u njenu konstantu `params`. Značenje svake od navedenih vrednosti možete potražiti u API dokumentaciji na linku <https://developers.google.com/youtube/v3/docs/search/list>.

U nastavku, nastavlja se izgradnja `queryUrl` spajanjem vrednosti za `apiUrl` i `params`.

Sada kada je `queryUrl` dostupan, moguće je kreirati zahtev. U konkretnom slučaju, zahtev će biti realizovan pozivom `http.get`, iako `HttpClient` može da obavi bilo koji tip `HTTP` zahteva (`POST`, `DELETE`, `GET` i tako dalje). Navedeno je implementirano u kompletiranom listingu metode `search()`:

```
search(query: string): Observable<SearchResult[]> {
    const params: string = [
        `q=${query}`,
        `key=${this.apiKey}`,
        `part=snippet`,
        `type=video`,
        `maxResults=10`
    ].join('&');
    const queryUrl = `${this.apiUrl}?${params}`;
    return this.http.get(queryUrl).map(response => {
        return <any>response['items'].map(item => {
            // console.log("raw item", item); // uklonite komentar za debug
            return new SearchResult({
                id: item.id.videoId,
                title: item.snippet.title,
                description: item.snippet.description,
                thumbnailUrl: item.snippet.thumbnails.high.url
            });
        });
    });
}
```

Gledano od linije koda 13, koriste se povratna vrednost `http.get` i funkcija (`map`) za dobijanje odgovora (`response`) iz prosleđenog zahteva. Iz odgovora (`response`) se izvlači telo kao objekat primenom `.json()`, obavlja se iteracija preko svake stavke (`item`) koja se konvertuje u `SearchResult`.

KOMPLETAN LISTING ZA YOUTUBESEARCHSERVICE

Za širu sliku kreirani listinzi klase se sklapaju u celinu.

Za širu sliku kreirani listinzi klase se sklapaju u celinu.

```
/**  
 * YouTubeService se povezuje na YouTube API  
 * Pogledajte: * https://developers.google.com/youtube/v3/docs/search/list  
 */  
@Injectable()  
export class YouTubeSearchService {  
  constructor(  
    private http: HttpClient,  
    @Inject(YOUTUBE_API_KEY) private apiKey: string,  
    @Inject(YOUTUBE_API_URL) private apiUrl: string  
  ) {}  
  
  search(query: string): Observable<SearchResult[]> {  
    const params: string = [  
      `q=${query}`,  
      `key=${this.apiKey}`,  
      `part=snippet`,  
      `type=video`,  
      `maxResults=10`  
    ].join('&');  
    const queryUrl = `${this.apiUrl}?${params}`;  
    return this.http.get(queryUrl).map(response => {  
      return <any>response['items'].map(item => {  
        // console.log("raw item", item); // uklonite komentar za debug  
        return new SearchResult({  
          id: item.id.videoId,  
          title: item.snippet.title,  
          description: item.snippet.description,  
          thumbnailUrl: item.snippet.thumbnails.high.url  
        });  
      });  
    });  
  }  
}
```

Da bi primer bio u potpunosti funkcionalan, u daljem radu se pristupa kreiranju preostalih delova komponente.

KLASA SEARCHBOXCOMPONENT

SearchBoxComponent je posrednik između korisničkog interfejsa i YouTubeSearchService servisa.

Klasa *SearchBoxComponent* ima ključnu ulogu u aplikaciji. Ona je posrednik između korisničkog interfejsa i *YouTubeSearchService* servisa.

Klasa *SearchBoxComponent* će obavljati sledeće zadatke:

1. vodi računa o unosu sa tastature i slanju unetog stringa ka servisu *YouTubeSearchService*;
2. emituje događaj učitavanja;
3. emituje događaj rezultata kada postoje novi rezultati.

Neka je data definicija klase *SearchBoxComponent* na sledeći način (datoteka:<http://src/app/you-tube-search/search-box.component.ts>):

```
@Component({
  selector: 'app-search-box',
  template: `
    <input type="text" class="form-control" placeholder="Search" autofocus>
  `
})
export class SearchBoxComponent implements OnInit {
  @Output() loading: EventEmitter<boolean> = new EventEmitter<boolean>();
  @Output() results: EventEmitter<SearchResult[]> = new
EventEmitter<SearchResult[]>();

  constructor(private youtube: YouTubeSearchService,
              private el: ElementRef) {
}
```

Selektor je analiziran veliki broj puta, ovde omogućava kreiranje taga *<app-search-box>*. Dve osobine obeležene dekoratorom *@Outputs* ukazuju da će događaji biti emitovani iz ove klase. To znači da je u pogledu moguće koristiti sintaksu *(output)="callback()"* za osluškivanje događaja ove komponente. Na primer, na sledeći način je moguće upotrebiti tag *<app-search-box>* u pogledu:

```
<app-search-box
  (loading)="loading = $event"
  (results)="updateResults($event)">
</app-search-box>
```

U konkretnom primeru, kada *SearchBoxComponent* klasa emituje događaj učitavanja, podešava se promenljiva *loading* u kontekstu roditeljske komponente. Na isti način, kada *SearchBoxComponent* emituje događaje prikupljanja rezultata, poziva se funkcija *updateResults()*, sa vrednostima, u kontekstu roditeljske komponente (*YouTubeSearchComponent*).

U klasi, obeleženoj dekoratorom `@Component` (`YouTubeSearchComponent`) specificiraju se osobine događaja pod nazivima: `loading` i `results`. U konkretnom primeru, svaki događaj će imati odgovarajući `EventEmitter` kao objektne promenljive kontrolerske klase. Navedeno će biti implementirano u najkraćem roku.

Za sada, dovoljno je rezonovati o `@Component` kao javnom API za komponentu pa se ovde samo specificiraju nazivi događaja.

DDEFINICIJA ŠABLONA ZA SEARCHBOXCOMPONENT

Jednostavan šablon, sa jednim input tagom.

Šablon `SearchBoxComponent` je poprilično jednostavan i sadrži jedan `input` tag integriran pod ključem `template` dekoratora `@Component` kase `SearchBoxComponent` (datoteka: `src/app/you-tube-search/search-box.component.ts`).

```
@Component({
  selector: 'app-search-box',
  template: `
    <input type="text" class="form-control" placeholder="Search" autofocus>
  `
})
```

DEFINICIJA KONTROLERA ZA SEARCHBOXCOMPONENT

Klasa implementira metodu `OnInit` iz razloga što je neophodno koristiti povratni poziv (callback) `ngOnInit` životnog ciklusa

Klase kontroler `SearchBoxComponent` je nova klase koja se u projektu čuva u datoteci `/src/app/you-tube-search/search-box.component.ts`. Početna definicija klase je data sledećim listingom:

```
export class SearchBoxComponent implements OnInit {
  @Output() loading: EventEmitter<boolean> = new EventEmitter<boolean>();
  @Output() results: EventEmitter<SearchResult[]> = new
  EventEmitter<SearchResult[]>();
```

Veoma je bitno da ova klasa implementira metodu `OnInit` iz razloga što je neophodno koristiti povratni poziv (callback) `ngOnInit` životnog ciklusa. Ukoliko klasa implementira metodu `OnInit`, metoda `ngOnInit` će biti pozvana čim se prva promena detektuje. Ova metoda je odlično mesto za zadatke inicijalizacije (za razliku od konstruktora) iz razloga što skup ulaza u komponentu nije dostupan u konstruktoru.

Za obe osobine klase: `loading` i `results`, kreira se `EventEmitters` koji će vratiti podatak tipa `boolean` kada se pretraga učitava i niz tipa `SearchResults` kada je pretraga završena.

KONSTRUKTOR KONTROLERA ZA SEARCHBOXCOMPONENT

Definicija klase `SearchBoxComponent` se proširuje kreiranjem konstruktora.

Definicija klase `SearchBoxComponent` se proširuje kreiranjem konstruktora.

```
constructor(private youtube: YouTubeSearchService,  
           private el: ElementRef) {  
}
```

U konstruktor je umetnuto sledeće:

1. servis `YouTubeSearchService`;
2. element `el` na koji ova komponenta povezana. Ovaj element je objekat tipa `ElementRef` koji predstavlja Angular omotač oko nativnih elemenata.

Oba parametra konstruktora su preko umetanja zavisnosti podešeni kao objektne promenljive.

FUNKCIJA ONINIT KONTROLERA ZA SEARCHBOXCOMPONENT

Odlično rešenje predstavlja uključivanje događaja unosa sa tastature u osmatrani (`Observable`) tok.

Ako se obrati pažnja na šablon komponente `SearchBoxComponent` moguće je primetiti da on sadrži polje za unos teksta preko kojeg se unose ključne reči za pretragu `YouTube` video klipova.

U ovom svetu je moguće uraditi tri stvari sa ciljem unapređenja korisničkog iskustva:

1. filtriraju se (i odbacuju) svi prazni i prekratki upiti;
2. automatski se pokreće pretraga u zavisnosti od vremena proteklog od unosa poslednjeg karaktera (na primer najkraće od 250 ms);
3. odbacuju se sve stare pretrage, ukoliko je korisnik inicirao novu pretragu.

U kodu je moguće manuelno povezati (bind) unos sa tastature i poziv funkcije za svaki događaj unosa i, potom, primeniti filtriranje i vremensko pokretanje pretrage, po automatizmu. Međutim bolje rešenje predstavlja uključivanje događaja unosa sa tastature u osmatrani (`Observable`) tok.

Reaktivna biblioteka `RxJS` (*Reactive Extensions Library for JavaScript* - <https://rxjs-dev.firebaseapp.com/>) obezbeđuje mehanizme osluškivanje događaja konkretnog elementa na osnovu poziva funkcije `Rx.Observable.fromEvent()`. Navedeno je moguće obaviti na sledeći način:

```
ngOnInit(): void {
    // konvertuje unos sa tastature u observable tok
    Observable.fromEvent(this.el.nativeElement, 'keyup')
```

Iz priloženog koda metode `fromEvent()` moguće je primetiti sledeće:

- prvi element je `this.el.nativeElement` (nativni DOM element na koji je komponenta povezana);
- drugi argument predstavlja string '`keyup`', koji predstavlja naziv događaja kojeg je neophodno konvertovati u tok.

RXJS FUNKCIONALNOSTI U ONINIT METODI

Demonstracija benefita koji se ostvaruju primenom RxJS funkcionalnosti.

Definicija metode `onInit()` klase komponente još uvek nije gotova. U ovom delu izlaganja je cilj demonstracija benefita koji se ostvaruju primenom RxJS funkcionalnosti. Izlaganje će ići korak po korak sa ciljem prezentovanja i implementacije navedenih funkcionalnosti.

Navedeni tok događaja tastature (`keyup`) moguće je povezati u lanac sa većim brojem metoda. Upravo, u sledećem izlaganju, nekoliko funkcija će biti povezano u lanac zajedno sa tokom pri čemu će biti obavljena transformacija toka u niz `SearchResult[]`. Na samu kraju svi listinzi koji su opisivali detalje primera biće povezani u jednu celinu zbog jasnijeg sagledavanja rešenja.

U prvom koraku je neophodno izolovati vrednost dobijenu preko `<input>` taga (uneto sa tastature):

```
.map((e: any) => e.target.value) // izvlači vrednost inputa
```

Priloženi listing kaže da je neophodno obaviti mapiranje svakog događaja tastature (`keyup`), a zatim pronaći ciljni element događaja (`e.target`), a to je u ovom slučaju input element. Kada je to gotovo neophodno je izvući vrednost tog elementa. To praktično znači da je posmatrani tok sada konvertovan u tok stringova.

U sledećem koraku u lanac se dodaje poziv sledeće funkcije:

```
.filter((text: string) => text.length > 1) // odbacuje ulaz ako je prazan
```

Ovaj filter obezbeđuje da za string dužine manje od 1 neće biti vršena pretraga. Programer, ukoliko misli da je zgodno, ovaj broj može povećati za drugačiju definiciju ignorisanja pretrage po kratkim stringovima.

U nastavku, u lanac metoda je moguće dodati i sledeći poziv:

```
.debounceTime(250) // na unos stringa pokreće pretragu za vreme nakon 250ms
```

Metoda debounceTime() obezbeđuje da zahtevi koji stignu u intervalu manjem od 250ms neće biti uzimani u obzir. To znači, neće biti obavljana pretraga za svaki pritisak na taster tastature, već kada je napravljena mala pauza nakon poslednjeg pritiska na taster.

Sledeći poziv u lancu metoda može biti:

```
.do(() => this.loading.emit(true)) // omogućava učitavanje
```

Primenom metode do na tok je način izvođenja funkcije među-toka za svaki događaj, ali još uvek ne menja ništa u toku. Ideja je sledeća: postoji pretraga, postoji dovoljan broj karaktera ključne reči pretrage, our search, it has enough characters, prošao je dovoljan vremenski interval i pretraga može biti započeta, odnosno uključuje se učitavanje. Vrednost `this.loading` je tipa `EventEmitter`. Učitavanje se uključuje emitovanjem vrednosti `true` kao sledećeg događaja. Za `EventEmitter` vrši se emitovanje događaja pozivom `next`: Sintaksa `this.loading.emit (true)` govori da se emituje događaj "true" na učitavanju objekta `EventEmitter` - vrednost \$event je true.

DODATNE FUNCKIJE U LANCU TOKA DOGAĐAJA

Prethodni lanac je moguće nastaviti pozivima novih korisnih metoda.

Prethodni lanac je moguće nastaviti pozivima novih korisnih metoda. Na primer, moguće je dodati sledeće:

```
// pretraga, odbacuju se stari događaji ukoliko se javi nov input
    .map((query: string) => this.youtube.search(query))
    .switch()
```

Poziv `.map()` se koristi za pozivanje izvođenja pretrage po svakom postavljenom upitu koji je emitovan. Pozivom `switch()` ignorišu se svi rezultati pretrage osim poslednjeg. To praktično znači, ukoliko je dostupan rezultat najnovije pretrage on se prikazuje, a stariji se odbacuju.

Konačno, za svaki emitovani upit, vrši se pretraga preko kreiranog servisa `YouTubeSearchService`. Objedinjavanjem lanca poziva metoda u celinu, dobija se sledeći listing:

```
ngOnInit(): void {
    // konvertuje unos sa tastature u observable tok
    Observable.fromEvent(this.el.nativeElement, 'keyup')
        .map((e: any) => e.target.value) // izvlači vrednost inputa
        .filter((text: string) => text.length > 1) // odbacuje ulaz ako je prazan
        .debounceTime(250) // na unos stringa pokreće
    pretragu za vreme do 250ms
        .do(() => this.loading.emit(true)) // omogućava učitavanje
    // pretraga, odbacuju se stari događaji ukoliko se javi nov input
        .map((query: string) => this.youtube.search(query))
        .switch()
    // postupa po vraćanju rezultata
        .subscribe(
            (results: SearchResult[]) => { // uspešno (onSuccess)
```

```

        this.loading.emit(false);
        this.results.emit(results);
    },
    (err: any) => { // greška (onError)
        console.log(err);
        this.loading.emit(false);
    },
    () => { // kompletirano (onComplete)
        this.loading.emit(false);
    }
);
}
}

```

U konačnu definiciju lanca metoda dodat je i poziv `subscribe()`. Budući da se vrši obraćanje servisu `YouTubeSearchService`, posmatrani tok je sada tok izgrađen od objekata tipa `SearchResult[]`. Pozivom metode `subscribe()` vrši se prijavljivanje na ovaj tok i omogućeno je izvođenje odgovarajućih operacija. Metoda uzima tri argumenta:

- `onSuccess`,
- `onError`,
- `onCompletion`.

Prvi argument ukazuje šta je neophodno uraditi ukoliko tok emituje regularan događaj. Događaj se emituje za oba `EventEmitter` - a:

1. pozivom `this.loading.emit(false)` se ukazuje da je učitavanje zaustavljeno;
2. pozivom `this.results.emit(results)` se ukazuje na emitovanje događaja koji sadrži listu sa rezultatima.

Drugi argument ukazuje šta se dešava ukoliko tok poseduje događaj sa greškom. Zaustavlja se učitavanje i u logu pregledača se dobija informacija o grešci.

Treći argument ukazuje šta se dešava kada se stigne do kraja toka. Takođe, vrši se emitovanje da je učitavanje završeno.

SEARCHBOXCOMPONENT - KOMPLETAN LISTING

Za sagledavanje šire slike, prethodni segmenti koda se sklapaju u funkcionalnu celinu.

Za sagledavanje šire slike, prethodni segmenti koda se sklapaju u funkcionalnu celinu.

Klasa `SearchBoxComponent` se čuva u datoteci sa sledeće lokacije:[/src/app/you-tube-search/search-box.component.ts](#) i priložena je sledećim listingom:

```

import {
    Component,
    OnInit,
    Output,
    EventEmitter,
}

```

```
 ElementRef
} from '@angular/core';

import { Observable } from 'rxjs/Rx';
import 'rxjs/add/observable/fromEvent';
import 'rxjs/add/operator/map';
import 'rxjs/add/operator/filter';
import 'rxjs/add/operator/debounceTime';
import 'rxjs/add/operator/do';
import 'rxjs/add/operator/switch';

import { YouTubeSearchService } from './you-tube-search.service';
import { SearchResult } from './search-result.model';

@Component({
  selector: 'app-search-box',
  template: `
    <input type="text" class="form-control" placeholder="Search" autofocus>
  `
})
export class SearchBoxComponent implements OnInit {
  @Output() loading: EventEmitter<boolean> = new EventEmitter<boolean>();
  @Output() results: EventEmitter<SearchResult[]> = new
EventEmitter<SearchResult[]>();

  constructor(private youtube: YouTubeSearchService,
              private el: ElementRef) {
  }

  ngOnInit(): void {
    // konvertuje unos sa tastature u observable tok
    Observable.fromEvent(this.el.nativeElement, 'keyup')
      .map((e: any) => e.target.value) // izvlači vrednost inputa
      .filter((text: string) => text.length > 1) // odbacuje ulaz ako je prazan
      .debounceTime(250) // na unos stringa pokreće
    pretragu za vreme do 250ms
      .do(() => this.loading.emit(true)) // omogućava učitavanje
    // pretraga, odbacuju se stari događaji ukoliko se javi nov input
      .map((query: string) => this.youtube.search(query))
      .switch()
    // postupa po vraćanju rezultata
      .subscribe(
        (results: SearchResult[]) => { // uspešno (onSuccess)
          this.loading.emit(false);
          this.results.emit(results);
        },
        (err: any) => { // greška (onError)
          console.log(err);
          this.loading.emit(false);
        },
        () => { // kompletirano (onComplete)
          this.loading.emit(false);
        }
      )
  }
}
```

```
        }  
    );  
}  
}
```

KREIRANJE KOMPONENTE SEARCHRESULTCOMPONENT

Zadatak ove komponente jeste kreiranje i prikazivanje pojedinačnih SearchResult objekata.

U prethodnom izlaganju je bilo diskusije i rada na veoma složenoj komponenti, za ova nivo znanja, pod nazivom *SearchBoxComponent*. Sada je moguće posvetiti pažnju jednostavnijoj komponenti *SearchResultComponent*. Zadatak ove komponente jeste kreiranje i prikazivanje pojedinačnih rezultata pretrage, tj. jednog *SearchResult* objekta.

Kod klase ove komponente predstavlja "već viđeno" i priložen je sledećim listingom (datoteka: *src/app/you-tube-search/search-result.component.ts*):

```
import {  
  Component,  
  OnInit,  
  Input  
} from '@angular/core';  
import { SearchResult } from './search-result.model';  
  
@Component({  
  selector: 'app-search-result',  
  templateUrl: './search-result.component.html'  
)  
export class SearchResultComponent implements OnInit {  
  @Input() result: SearchResult;  
  
  constructor() {}  
  
  ngOnInit() {}  
}
```

Komponenta poseduje jednu (ulaznu) objektnu promenljivu *result*, koja predstavlja instancu tipa *SearchResult*.

Šablon komponente će sadržati: naziv, opis i sliku vezanu za pronađeni klip, a zatim i link ka videu kojem se pristupa putem dugmeta (slika 3). Sledi listing šablonu (datoteka: *src/app/you-tube-search/search-result.component.html*):

```
<div class="col-sm-6 col-md-3">
  <div class="thumbnail">
    
      <h3>{{result.title}}</h3>
      <p>{{result.description}}</p>
      <p><a href="{{result.videoUrl}}" class="btn btn-default" role="button">Gledaj</a></p>
    </div>
  </div>
</div>
```



Slika 2.3 Jeden SearchResult objekat [izvor: autor]

KREIRANJE GLAVNE KOMPONENTE ZA YOUTUBESEARCHCOMPONENT

Kreiranje komponente koja objedinjuje sve ostale komponente u celinu.

Ovaj deo projekta se okončava kreiranjem glavne komponente - `YouTubeSearchComponent` čiji je zadatak da poveže sve prikazane komponente i njihove funkcionalnosti u jednu celinu.

Definicija komponente započinje kreiranjem dekoratora `@Component` i osobina klase na sledeći način (datoteka: `/src/app/you-tube-search/you-tube-search.component.ts`):

```
@Component({
  selector: 'app-you-tube-search',
  templateUrl: './you-tube-search.component.html'
})
export class YouTubeSearchComponent implements OnInit {
  results: SearchResult[];
  loading: boolean;
```

Klasa definiše identifikovanje komponente putem taga `<app-you-tube-search>` i primenjuje dve objektne promenljive: `results` i `loading`.

Da bi kreirana klasa mogla da se smatra kontrolerom, ona mora da implementira neke pogodne metode. Na prethodni listing dodaje se kod koji odgovara pomenutim metodama.

```
import { Component, OnInit } from '@angular/core';
import { SearchResult } from './search-result.model';

@Component({
  selector: 'app-you-tube-search',
  templateUrl: './you-tube-search.component.html'
})
export class YouTubeSearchComponent implements OnInit {
  results: SearchResult[];
  loading: boolean;

  constructor() { }
  ngOnInit() { }

  updateResults(results: SearchResult[]): void {
    this.results = results;
    // console.log("results:", this.results); // uklonite komentar da vidite
rezultat
  }
}
```

Dodate metode su konstruktor i `updateResults()`. Funkcija `updateResults()` se ponaša veoma jednostavno - uzima bilo koji novi rezultat pretrage (`SearchResult[]`) i podešava `this.results` na novu vrednost.

KREIRANJE ŠABLONA ZA YOUTUBESEARCHCOMPONENT

Kreiranjem šablona glavne komponente završena je definicija primera.

Kreiranjem šablona glavne komponente završena je definicija primera. Ovaj pogled bi trebalo da obavi sledeće tri stvari:

1. prikazuje indikator učitavanja, ako je zahtev u toku;
2. osluškuje događaje za kontrolu unosa ključnih reči pretrage;
3. prikazuje rezultate pretrage.

Za početak, data je osnovna struktura šablona sa GIF animacijom koja ukazuje da je učitavanje zahteva u toku (slika 4).

```
<div class='container'>
  <div class="page-header">
    <h1>YouTube pretraga
      <img
        style="float: right;"*
        *ngIf="loading"
        src='assets/images/loading.gif' />
    </h1>
  </div>
```

YouTube pretraga



Slika 2.4 Indikator učitavanja zahteva [izvor: autor]

Navedena slika se prikazuje samo ako je vrednost promenljive `loading true`. Za ovo je upotrebljena `ngIf` direktiva.

Sada je neophodno uključiti u pogled i kontrolu za pretragu:

```
<div class="row">
  <div class="input-group input-group-lg col-md-12">
    <app-search-box
      (loading)="loading = $event"
      (results)="updateResults($event)"
    ></app-search-box>
  </div>
</div>
```

Interesantan detalj predstavlja povezivanje: `loading` i `results` izlaza. Primećuje se korišćenje sintakse, opšteg oblika, `(output)="action()"`.

Za učitavanje izlaza, pokreće se izraz `loading = $event`. `$event` biće zamenjena vrednošću koja je emitovana od strane `EventEmitter`. To znači, kada se u komponenti `SearchBoxComponent` pozove `this.loading.emit(true)`, `$event` se podešava na vrednost `true`.

Slično, za izlaz `results` poziva se funkcija `updateResults()` uvek prilikom emitovanja novog skupa rezultata pretrage. Ovo za efekat ima ažuriranje objektne promenljive `results` komponente.

Konačno, neophodno je u pogledu prikazati rezultate pretrage.

```
<div class="row">
  <app-search-result
    *ngFor="let result of results"
    [result]="result">
    </app-search-result>
  </div>
</div>
```

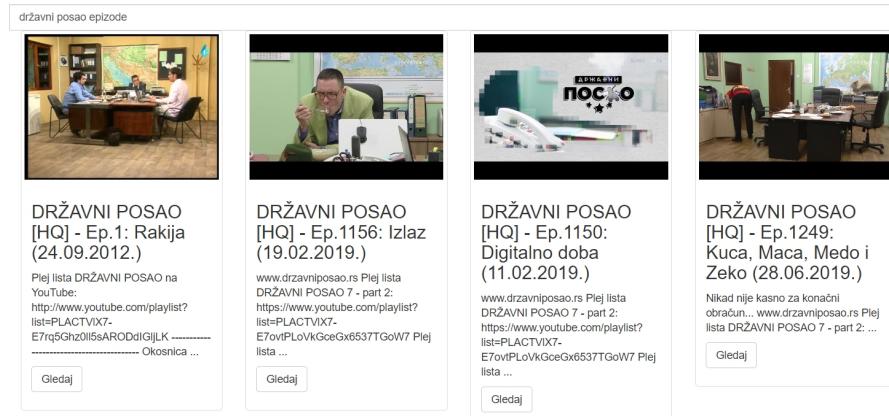
DEMO PRIMERA

Sledi demonstracija urađenog primera.

Sledi demonstracija urađenog primera. Prvo se unose ključne reči pretrage i ako zadovoljavaju uslov, pokreće se pretraga (ovo je prikazano slikom 4).

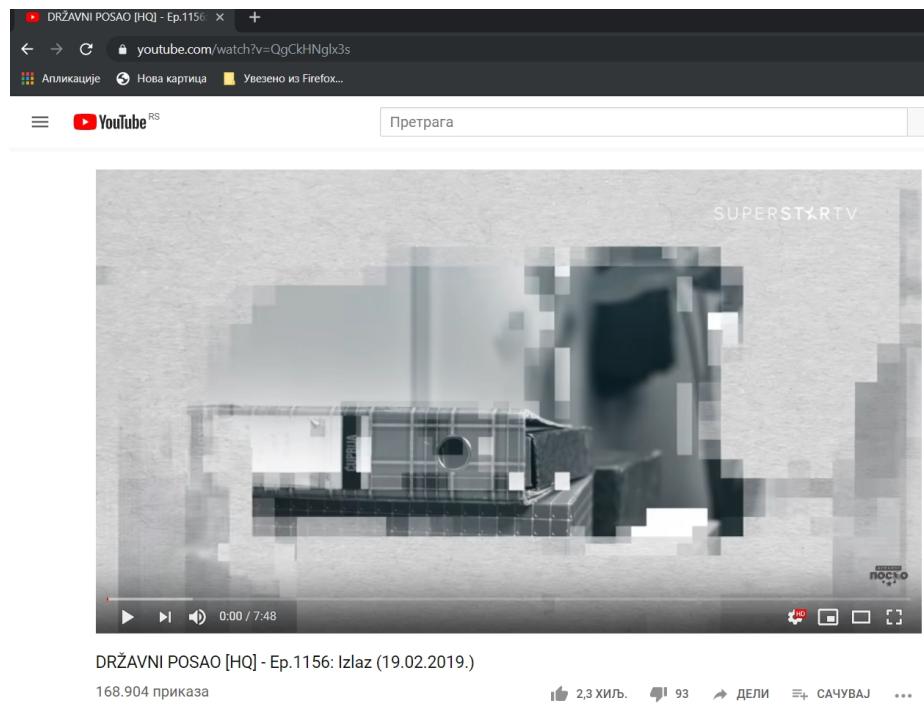
Uskoro prikazuju se rezultati pretrage - ograničeno na 10 rezultata (slika 5).

YouTube pretraga



Slika 2.5 Rezultati pretrage [izvor: autor]

Iz rezultata pretrage bira se jedan i klikom na dugme "Gledaj" pokreće se u YouTube aplikaciji izabrani video klip (slika 6).



Slika 2.6 Izbor iz rezultata pretrage [izvor: autor]

▼ Poglavlje 3

Modifikacija primera za Angular13+

PROMENA U RXJS BIBLIOTECI

Promena u RxJS biblioteci zahteva korekcije aplikacije.

Promena u RxJS biblioteci zahteva korekcije aplikacije ukoliko se radi na njenom razvoju primenom poslednjih ažuriranja radnog okvira Angular.

U aktuelnom primeru, za klasu komponente `search-box-component` na sledeći način su uvezene RxJS zavisnosti:

```
import { Observable } from 'rxjs/Rx';
import 'rxjs/add/observable/fromEvent';
import 'rxjs/add/operator/map';
import 'rxjs/add/operator/filter';
import 'rxjs/add/operator/debounceTime';
import 'rxjs/add/operator/do';
import 'rxjs/add/operator/switch';
```

Međuti, u poslednjim verzijama okvira, kompjajler bi stalno prijavljivao grešku jer je sama lokacija klase `Observable`, a i svih primenjenih operatora promenjena. Neki od navedenih operatora, na primer `do`, u poslednjim verzijama okvira Angular nisu više u primeni. Prethodni listing bi trebalo izmeniti na sledeći način:

```
import {
  Observable,
  fromEvent
} from 'rxjs'

import {
  map,
  filter,
  debounceTime,
  tap,
  switchAll
} from 'rxjs/operators';
```

MODIFIKACIJA KODA KOMPONTE ZA PRETRAGU

Modifikacija koda kompone za pretragu na osnovu izmena u RxJS.

Modifikacija koda komponente za pretragu na osnovu izmena u [RxJS](#) predstavlja sledeći korak kojeg je potrebno izvesti sa ciljem daljeg održavanja i razvoja kreiranog [Angular](#) projekta.

Neka je dat sledeći segment koda *klase search-box-component*:

```

ngOnInit(): void {
    // konvertuje unos sa tastature u observable tok
    Observable.fromEvent(this.el.nativeElement, 'keyup')
        .map((e: any) => e.target.value) // izvlači vrednost inputa
        .filter((text: string) => text.length > 1) // odbacuje ulaz ako je prazan
        .debounceTime(250) // na unos stringa pokreće
    pretragu za vreme do 250ms
        .do(() => this.loading.emit(true)) // omogućava učitavanje
    // pretraga, odbacuju se stari događaji ukoliko se javi nov input
        .map((query: string) => this.youtube.search(query))
        .switch()
    // postupa po vraćanju rezultata
        .subscribe(
            (results: SearchResult[]) => { // uspešno (onSuccess)
                this.loading.emit(false);
                this.results.emit(results);
            },
            (err: any) => { // greška (onError)
                console.log(err);
                this.loading.emit(false);
            },
            () => { // kompletirano (onComplete)
                this.loading.emit(false);
            }
        );
    }
}

```

Ovakav kod bi proizveo niz grešaka na poslednjoj verziji kompajlera, upravo iz prethodno navedenog razloga - došlo je do izmene u [RxJS](#) biblioteci. U novom kodu, nije potrebno da [Observable](#) pozove metodu [fromEvent\(\)](#). U novom listingu samostalni poziv ove metode vraća Observable objekat koji se prijavljuje na izmene u podacima. Dalje, ne vrši se ulančavanje operatora, kao u prethodnom slučaju, neko se taj lanac pravi tako što se operatori navode kao argumenti metode - operatora [pipe\(\)](#). Još jedna razlika, umesto operatora [do](#), operator [tap](#) će obaviti zadatke emitovanja događaja unosa nakon 250ms. Sledi izmenjeni listing:

```

ngOnInit(): void {

    // konvertuje unos sa tastature u observable tok
    fromEvent(this.el.nativeElement as any, 'keyup').pipe(
        map((e: any) => e.target.value), // izvlači vrednost inputa

        // filter((text:string) => text.length > 1), //da li je unos prazan

        debounceTime(250), //pokreće pretragu nakon 250 ms

        tap(() => this.loading.emit(true)), // omogućava učitavanje
    )
}

```

```
// poziv servisa pretrage

map((query: string) => this.youtube.search(query)),
// odbacuje stare unose pretrage

switchAll()
    // postupa po vraćanju rezultata
.subscribe(
    (results: SearchResult[]) => { // uspešno (onSuccess)
        this.loading.emit(false);
        this.results.emit(results);

    },
    (err: any) => { // greška (onError)
        console.log(err);
        this.loading.emit(false);
    },
    () => { // kompletirano (onComplete)
        this.loading.emit(false);
    }
);

}
```

PROMENA U KONFIGURACIJI

Napravljeno je još malo modifikacija, pre svega u konfiguracionoj datoteci `tsconfig.json`.

Za potrebe inovirane aplikacije, napravljeno je još malo modifikacija, pre svega u konfiguracionoj datoteci `tsconfig.json`. Dodata su sledeća podešavanja:

```
*****
"strictPropertyInitialization":false,
"noImplicitAny": false,
*****
```

U verzijama `TypeScript` jezika 2,7 i novijim, kompajler zahteva da sve osobine klase budu inicijalizovane, a to predstavlja problem kod modelskih klasa čiji objekti treba naknadno da dobiju vrednosti navedenih osobina. Linija koda 2 rešava navedeni problem.

U nekim slučajevima gde nema napomena tipa, `TypeScript` će se vratiti na bilo koji (`any`) tip za promenljivu kada ne može da zaključi tip. Linija koda 3, upravo, sprečava pojavljivanje grešaka na strani kompajlera

U poslednjem delu pokaznog primera upotrebljen je `CSS` okvir `Bootstrap` za stilizovanje unosa i prikaza rezultata pretrage `YouTube` klipova. On je dostupan globalno u aplikaciji tako što je prvo instaliran na komandnoj liniji na sledeći način:

```
npm install bootstrap --save
// -- save znači da će biti automatski dodat u node_modules
//opcionalno može biti dodat i jQuery
npm install jquery --save
```

Nakon ovoga je potrebno uključiti *Bootstrap*, na početku koda globalne *CSS* datoteke, *styles.css* na sledeći način:

```
@charset "UTF-8";
@import "~bootstrap/dist/css/bootstrap.css";
//ostatak CSS listinga
```

Ukoliko primer bude pravio problem a fontovima tipa *Glyphicon Halflings*, na potpuno isti način, kao prethodnom slučaju, sa komandne linije je neophodno instalirati odgovarajuću biblioteku. Ovo može da bude razlog ukoliko je ona uklonjena iz poslednje verzije okvira *Bootstrap*.

```
npm installglyphicon-halflings --save
```

Sve što je potrebno, dalje, jeste ponovno prevođenje aplikacije i ona će normalno da funkcioniše, baš kao u prethodno demonstriranom slučaju.

Više o RxJS sledi u narednim lekcijama, ovde je cilj bio dobijanje bogatije aplikacije koja je što bliža realnoj.

Preuzmite urađeni primer na, ažuriran po novim zahtevima, odmah nakon ovog objekta učenja.

▼ Poglavlje 4

angular/common/http API - ostali HTTP zahtevi

IZVOĐENJE POST ZAHTEVA

API jsonplaceholder obezbeđuje pogodan URL za testiranje POST zahteva.

U prethodnom izlaganju akcenat je bio na primeni osnovnog HTTP zahteva GET. Naravno, važno je da se shvati da je na veoma jednostavan način u Angular aplikacijama moguće napraviti i ostale HTTP zahteve.

Prvi zahtev koji dolazi na red za diskusiju je POST zahtev. Kreiranje POST zahteva primenom paketa @angular/common/http veoma podseća na kreiranje GET zahteva uključujući jedan dodatni parametar: body.

API jsonplaceholder (<http://jsonplaceholder.typicode.com/>), takođe, obezbeđuje pogodan URL za testiranje POST zahteva. U narednom izlaganju će upravo navedeno biti i pokazano.

Za početak je kreirana nova komponenta more-http-requests.component. Sledećom listingom je prikazana metoda makePost() kontrolerske klase komponente čiji se kod čuva u datoteci src/app/more-http-requests/more-http-requests.component.ts.

```
makePost(): void {
  this.loading = true;
  this.http
    .post(
      'https://jsonplaceholder.typicode.com/posts',
      JSON.stringify({
        body: 'bar',
        title: 'foo',
        userId: 1
      })
    )
    .subscribe(data => {
      this.data = data;
      this.loading = false;
    });
}
```

U drugom argumentu, kao što je moguće primetiti iz listinga, uzima se Object koji se prevodi u JSON string pozivom JSON.stringify().

PUT / PATCH / DELETE / HEAD

Postoje i ostali HTTP zahtevi koji se pozivaju na veoma sličan način kao prikazani GET i POST.

Postoje i ostali, prilično opšti HTTP zahtevi koji se pozivaju na veoma sličan način kao prikazani **GET** i **POST** zahtev:

- **http.put** i **http.patch** mapiraju se u **PUT** i **PATCH**, respektivno, i oba koriste **URL** i **body**;
- **http.delete** i **http.head** mapiraju se u **DELETE** i **HEAD**, respektivno i koriste samo **URL**.

Sledi primer izvođenja **DELETE** zahteva (datoteka: *src/app/more-http-requests/more-http-requests.component.ts*):

```
makeDelete(): void {
  this.loading = true;
  this.http
    .delete('https://jsonplaceholder.typicode.com/posts/1')
    .subscribe(data => {
      this.data = data;
      this.loading = false;
    });
}
```

PRILAGOĐENA HTTP ZAGLAVLJA

Kreiranje GET zahteva koji koristi specijalno X-API-TOKEN zaglavlje.

Konačno, recimo da je cilj kreiranje GET zahteva koji koristi specijalno **X-API-TOKEN** zaglavlje. Moguće je kreirati zahtev sa navedenim zaglavljem na sledeći način:

```
makeHeaders(): void {
  const headers: HttpHeaders = new HttpHeaders({
    'X-API-TOKEN': 'IT-255'
  });

  const req = new HttpRequest(
    'GET',
    'https://jsonplaceholder.typicode.com/posts/1',
    {
      headers: headers
    }
  );

  this.http.request(req).subscribe(data => {
    this.data = data['body'];
  });
}
```

OBJEDINJENI LISTINZI KONTROLERA I ŠABLONA KOMPONENTE

Za širu sliku kreirani listinzi kontrolera i šablonu komponente se sklapaju u celinu.

Sledi celokupan listing klase komponente:

```
import { Component, OnInit } from '@angular/core';
import {
  HttpClient,
  HttpRequest,
  HttpHeaders
} from '@angular/common/http';

@Component({
  selector: 'app-more-http-requests',
  templateUrl: './more-http-requests.component.html'
})
export class MoreHttpRequestsComponent implements OnInit {
  data: Object;
  loading: boolean;

  constructor(private http: HttpClient) {}

  ngOnInit() {}

  makePost(): void {
    this.loading = true;
    this.http
      .post(
        'https://jsonplaceholder.typicode.com/posts',
        JSON.stringify({
          body: 'bar',
          title: 'foo',
          userId: 1
        })
      )
      .subscribe(data => {
        this.data = data;
        this.loading = false;
      });
  }

  makeDelete(): void {
    this.loading = true;
    this.http
      .delete('https://jsonplaceholder.typicode.com/posts/1')
      .subscribe(data => {
        this.data = data;
        this.loading = false;
      });
  }
}
```

```
    });

makeHeaders(): void {
  const headers: HttpHeaders = new HttpHeaders({
    'X-API-TOKEN': 'IT-255'
  });

  const req = new HttpRequest(
    'GET',
    'https://jsonplaceholder.typicode.com/posts/1',
    {
      headers: headers
    }
  );

  this.http.request(req).subscribe(data => {
    this.data = data['body'];
  });
}

}
```

Sledi celokupan listing šablona komponente:

```
<h2>Još zahteva</h2>
<button type="button" (click)="makePost()">Kreiraj Post</button>
<button type="button" (click)="makeDelete()">Kreiraj Delete</button>
<button type="button" (click)="makeHeaders()">Kreiraj Headers</button>
<div *ngIf="loading">loading...</div>
<pre>{{data | json}}</pre>
```

▼ Poglavlje 5

Kreiranje i testiranje zahteva

PRIMENA KOMPONENTE MORE-HTTP-REQUESTS.COMPONENT - DEMO

Izlaganje završava demonstracijom izvršavanja poslednjeg primera.

Izlaganje završava demonstracijom izvršavanja poslednjeg primera. Sledećom slikom prikazan je rezultat dobijen klikom na dugme "Kreiraj POST".

Još zahteva

Kreiraj Post	Kreiraj Delete	Kreiraj Headers
{ "id": 101 }		

Slika 5.1 Kreiranje i izvršavanje POST zahteva [izvor: autor]

Sledećom slikom prikazan je rezultat dobijen klikom na dugme "Kreiraj DELETE".

Još zahteva

Kreiraj Post	Kreiraj Delete	Kreiraj Headers
{}		

Slika 5.2 Kreiranje i izvršavanje DELETE zahteva [izvor: autor]

Sledećom slikom prikazan je rezultat dobijen klikom na dugme "Kreiraj HEADERS".

Još zahteva

Kreiraj Post	Kreiraj Delete	Kreiraj Headers
{ "userId": 1, "id": 1, "title": "sunt aut facere repellat provident occaecati excepturi optio reprehenderit", "body": "quia et suscipit\\nsuscipit recusandae consequuntur expedita et cum\\nreprehenderit molestiae ut beatae vitae dictum sint", "author": "Quia et suscipit\\nsuscipit recusandae consequuntur expedita et cum\\nreprehenderit molestiae ut beatae vitae dictum sint" }		

Slika 5.3 Kreiranje i izvršavanje HEADER zahteva [izvor: autor]

Odmah, nakon ovog objekta učenja, možete preuzeti, analizirati i testirati kompletno urađen pokazni primer koji je koristio kao podrška u izlaganju ove lekcije.

ANGULAR I HTTP - DODATNI POGLED

Kada se pravi Angular aplikacija tako da koristi REST API, najbitniji servis je svakako HttpClient

Kada se pravi Angular aplikacija tako da koristi REST API, najbitniji servis je svakako [HttpClient](#), jer se pomoću njega mogu dalje implementirati specifični servisi koji komuniciraju sa spoljnom aplikacijom i koji prikupljaju ili šalju podatke. U tom slučaju, ovakvi servisi služe kao veza između klijentskog i serverskog dela aplikacije. Servis [HttpClient](#) ima obezbeđene metode za svaki tip zahteva ([GET](#), [POST](#), [PUT](#), [DELETE](#) ...) i svaka od ovih metoda vraća vrednost na koju se korisnik servisa kasnije može preplatiti.

▼ Poglavlje 6

Angular i HTTP- video matrijali

VIDEO 1

Angular HTTP Client Quick Start Tutorial - Trajanje 9:55

Ova lekcija sadrži video materijal. Ukoliko želite da pogledate ovaj video morate da otvorite LAMS lekciju.

VIDEO 2

Angular 8 Tutorial - 20 - HTTP and Observables - Trajanje 7:40

Ova lekcija sadrži video materijal. Ukoliko želite da pogledate ovaj video morate da otvorite LAMS lekciju.

VIDEO 3

Angular 6 Tutorial 12: HTTP Requests - Trajanje 13:33.

Ova lekcija sadrži video materijal. Ukoliko želite da pogledate ovaj video morate da otvorite LAMS lekciju.

✓ Poglavlje 7

Dodatni nastavni materijali

DODATNI VEB MATERIJALI

Proširivanje znanja stečenog na predavanjima.

1. <https://angular.io/>
2. <https://angular.io/tutorial>
3. <https://www.w3schools.com/angular/>
4. <https://www.tutorialspoint.com/angular4/>
5. <https://nodejs.org/en/>
6. <https://code.visualstudio.com/>
7. https://www.w3schools.com/whatis/whatis_htmldom.asp

▼ Poglavlje 8

Pokazna vežba 10

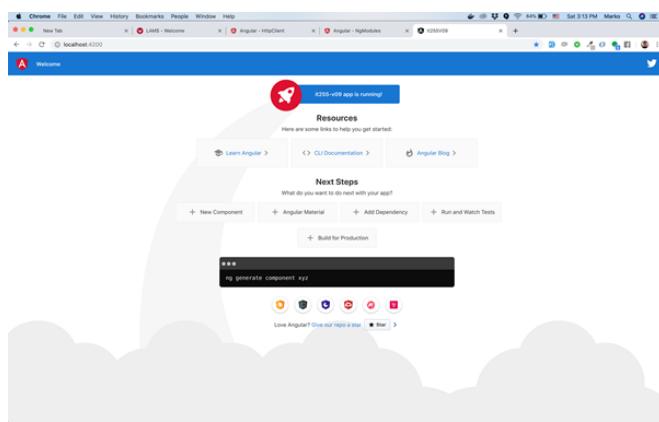
HTTP I ANGULAR - POKAZNA VEŽBA (TRAJANJE 45 MINUTA)

Prednost Angular - a, u odnosu na neke druge frameworke je što dolazi sa već gotovim bibliotekama za kreiranje HTTP zahteva

Prednost [Angular](#) - a, u odnosu na neke druge radne okvire (framework) je što dolazi sa već gotovim bibliotekama za kreiranje i obradu [HTTP](#) zahteva. Konkretno, u našem slučaju koristićemo [HttpClient](#). Prethodno navedena biblioteka dolazi kao sastavni deo modula [HttpClientModule](#), koji je neophodno da uključimo unutar [imports](#) sekcijs kroz datoteku [app.module.ts](#).

Za potrebe ove vežbe, biće kreiran novi projekat kao što je prikazano u prethodnim vežbama koristeći komandu: [ng new IT255-v09](#).

Po kreiranju projekta pokrećemo isti i imamo rezultat kao na slici:



Slika 8.1 Inicijalni izgled kreiranog projekta [izvor: autor]

Kako bismo malo stilizovali našu aplikaciju, učitaćemo [bootstrap](#) - ov [CSS](#) fajl unutar naše aplikacije, tako što ćemo dodati sledeće parče koda u [head](#) sekciiju [index.html](#) fajla:

```
<link rel="stylesheet" href="https://stackpath.bootstrapcdn.com/bootstrap/4.3.1/css/bootstrap.min.css" crossorigin="anonymous">
```

Nakon toga, kako bismo mogli da obavljamo [HTTP](#) pozive, neophodno je da uključimo [HttpClientModule](#), unutar [imports](#) sekcijs na nivou čitave aplikacije kroz [app.module.ts](#). Potrebno da je prvo uvezemo biblioteku iz [Angular](#) okvira na sledeći način:

```
import { HttpClientModule } from '@angular/common/http';
```

A potom da istu biblioteku uključimo unutar *Imports* dela na sledeći način:

```
import { BrowserModule } from '@angular/platform-browser';
import { NgModule } from '@angular/core';

import { AppRoutingModule } from './app-routing.module';
import { AppComponent } from './app.component';
import { HttpClientModule } from '@angular/common/http';

@NgModule({
  declarations: [
    AppComponent
  ],
  imports: [
    BrowserModule,
    HttpClientModule,
    AppRoutingModule
  ],
  providers: [],
  bootstrap: [AppComponent]
})
export class AppModule { }
```

Više informacija o tome se može naći na sledećem linku: <https://angular.io/guide/http>

PRIMENA GOTOVIH SERVISA I KREIRANJE MODELA

JSONPlaceholder omogućava da obavimo testiranje HTTP poziva kroz Angular okvir.

Za potrebe ove vežbe, koristićemo već dostupni REST servis pod nazivom JSONPlaceholder, koji će nam omogućiti da obavimo testiranje HTTP poziva kroz Angular okvir. Primenjujući znanje stečeno na časovima predavanja, vežbi i radeći servise, odmah pristupamo kreiranju servisa pod nazivom PostService unutar services foldera. Za to ćemo iskoristiti komandu: ng g service services/post.

Unutar tog servisa, kroz konstruktor neophodno je da inicijalizujemo HttpClient - a na sledeći način:

```
constructor(private _httpClient: HttpClient) { }
```

A pre toga je potrebno da uvezemo biblioteku na sledeći način:

```
import { HttpClient } from '@angular/common/http';
```

Imajući u vidu sve navedeno, moguće je prikazati kod servisa PostService u celini, sledećim listingom:

```
import { Injectable } from '@angular/core';
import { HttpClient } from '@angular/common/http';

@Injectable({
  providedIn: 'root'
})
export class PostService {

  constructor(private _httpClient: HttpClient) { }
}
```

Unutar servisa ćemo definisati metode pod imenima: `getPost`, `deletePost`, `createPost`. Postojaće i par pomoćnih metoda, ali o njima će biti reči malo kasnije.

Kako bismo ispoštovali standarde prethodno definisane u vežbama, kreiraćemo model pod naziv `Post` sledećom komandom:

- `ng g class models/post`

Sam `model Post` sadržaće atribute koji su prikazani na slici ispod:

```
- {
  userId: 1,
  id: 1,
  title: "sunt aut face",
  body: "quia et suscip
},
r
```

Slika 8.2 Atributi modela [izvor: autor]

KOD KLASE MODELA I DOPUNA DEFINICIJE SERVISA

Na osnovu istaknutih osobina modela, kodira se i odgovarajuća klasa.

Na osnovu istaknutih osobina modela, kodira se i odgovarajuća klasa. Klasa se naziva `Post`, čuva se u `TypeScript` datoteci `post.ts` i priložena je sledećim listingom:

```
import { Optional } from '@angular/core';

export class Post {
  id: number;
  userId: number;
  title: string;
  body: string;
```

```
constructor(userId: number, title: string, body: string, @Optional() id: number) {
    this.id = id;
    this.userId = userId;
    this.title = title;
    this.body = body;
}

}
```

Kao što je istaknuto u izlaganju sa prethodne sekcije, unutar servisa `PostService` ćemo definisati metode pod imenima: `getPost`, `deletePost`, `createPost`, kao i par pomoćnih metoda. Ove metode su date primerom koda ispod:

```
import { Injectable } from '@angular/core';
import { HttpClient } from '@angular/common/http';
import { Observable } from 'rxjs';
import { Post } from '../models/post';
import { map } from 'rxjs/operators';

@Injectable({
  providedIn: 'root'
})
export class PostService {
  private baseUrl = 'https://jsonplaceholder.typicode.com/posts';

  constructor(private _httpClient: HttpClient) { }

  public getPosts() : Observable<Post[]> {
    return this._httpClient.get(this.baseUrl).pipe(
      map((data: any[]) => data.map((item: any) =>
this._createPostFromObject(item))),
    );
  }

  public getPost(id: number) : Observable<Post> {
    return this._httpClient.get(this.baseUrl + '/' + id).pipe(
      map((data: any) => this._createPostFromObject(data)),
    );
  }

  public deletePost(id: number) : Observable<Post> {
    return this._httpClient.delete(this.baseUrl + '/' + id).pipe(
      map((data: any) => this._createPostFromObject(data)),
    );
  }

  public createPost(post: Post) : Observable<Post> {
    return this._httpClient.post(this.baseUrl, post).pipe(
      map((data: any) => this._createPostFromObject(data)),
    );
  }
}
```

```
}
```

```
private _createPostFromObject(item:any) {
    return new Post(item.userId, item.title, item.body, item.id);
}

}
```

Dodatne metode koje možemo videti unutar koda su:

- `pipe()` koja služi za vezivanje više metoda u cilju manipulacije izlaznog toka podataka.
- `map()` - Izvršava više funkcija nad izlaznim tokovima podataka.
- `_createPostFromObject()` - Pomoćna metoda koju koristimo za kreiranje modela podatka na osnovu JSON reprezentacija istog.

GLAVNA KOMPONENTA APLIKACIJE

Neophodno je da kreiramo i formu, koja će sadržati polja za kreiranje novog Post - a.

Unutar `app.component.ts` datoteke, u kojoj se čuva glavna klasa glavne komponente aplikacije `AppComponent`, koju ćemo koristiti za reprezentativni primer manipulacije podataka, imaćemo više podataka koje ćemo pozivati u raznim životnim ciklusima aplikacije, kao i prilikom određenih akcija, poput `getPost` i `deletePost`. Takođe, neophodno je da kreiramo i `formu`, koja će sadržati polja za kreiranje novog `Post` - a. Kod navedene forme je priložen sledećim listinzzima:

Inicijalizacija forme:

```
public initForm() {
    this.postForm = new FormGroup({
        title: new FormControl('', [
            Validators.required
        ]),
        userId: new FormControl(1, [
            Validators.required
        ]),
        body: new FormControl('', [
            Validators.required
        ])
    });
}
```

HTML kod forme (implementirano u šablonu komponente):

```
<form [formGroup]="postForm" class="container mt-5 pt-5" (ngSubmit)="submitForm()">
    <div class="form-group">
        <label for="title">Title</label>
        <input class="form-control" formControlName="title" type="text" name="title">
```

```
</div>

<div class="form-group">
    <label for="body">Body</label>
    <input class="form-control" formControlName="body" type="text" name="body">
</div>

<div class="form-group">
    <label for="userId">userId</label>
    <input class="form-control" formControlName="userId" type="number"
name="userId">
</div>
<div class="col button-holder d-flex justify-content-center align-items-center
mt-4">
    <button type="submit" name="submit" class="btn btn-success">Dodaj</button>
</div>
</form>
```

Prilikom inicijalizacije same komponente, vrši se učitavanje svih postova koristeći metodu koja je navedena ispod:

```
constructor(private _postService: PostService) {
    this._postService.getPosts().subscribe((data) => {
        this.posts = data;
    })
    this.initForm();
}
```

CELOVITI PRIKAZ GLAVNE KOMPONENTE APLIKACIJE

Neophodno je sve prikazane segmente komponente povezati u funkcionalnu celinu.

Finalni zadatak razvoja glavne komponente aktuelne Angular aplikacije ukazuje da je neophodno je sve prikazane segmente komponente povezati u funkcionalnu celinu.

Sledi listing klase komponente *AppComponent*:

```
import { Component } from '@angular/core';
import { PostService } from './services/post.service';
import { Post } from './models/post';
import { FormGroup, FormControl, Validators } from '@angular/forms';

@Component({
    selector: 'app-root',
    templateUrl: './app.component.html',
    styleUrls: ['./app.component.scss']
})
export class AppComponent {
    public postForm: FormGroup;
```

```
public posts: Post[] = [];

constructor(private _postService: PostService) {
  this._postService.getPosts().subscribe((data) => {
    this.posts = data;
  })
  this.initForm();
}

public initForm() {
  this.postForm = new FormGroup({
    title: new FormControl('', [
      Validators.required
    ]),
    userId: new FormControl(1, [
      Validators.required
    ]),
    body: new FormControl('', [
      Validators.required
    ])
  });
}

public submitForm() {
  let title = this.postForm.get('title').value;
  let userId = this.postForm.get('userId').value;
  let body = this.postForm.get('body').value;
  let post = new Post(userId, title, body, 0);
  this.createPost(post)
}

public getPost(id: number) {
  this._postService.getPost(id).subscribe((data) => {
    alert(JSON.stringify(data));
  })
}

public createPost(post: Post) {
  this._postService.createPost(post).subscribe((data) => {
    this.posts.unshift(data);
  })
}

public deletePost(id: number) {
  this._postService.deletePost(id).subscribe((data) => {
    this._removePostFromList(id);
    alert("Post je obrisan sa servera");
  })
}

private _removePostFromList(id: number) {
  let ind = this.posts.findIndex(post => post.id == id);
  this.posts.splice(ind, 1);
}
```

```
}
```

```
}
```

Sledi listing šablona komponente iz datoteke *app.component.html*:

```
<form [formGroup]="postForm" class="container mt-5 pt-5" (ngSubmit)="submitForm()">
  <div class="form-group">
    <label for="title">Title</label>
    <input class="form-control" formControlName="title" type="text" name="title">
  </div>

  <div class="form-group">
    <label for="body">Body</label>
    <input class="form-control" formControlName="body" type="text" name="body">
  </div>

  <div class="form-group">
    <label for="userId">userId</label>
    <input class="form-control" formControlName="userId" type="number" name="userId">
  </div>
  <div class="col button-holder d-flex justify-content-center align-items-center mt-4">
    <button type="submit" name="submit" class="btn btn-success">Dodaj</button>
  </div>
</form>

<div class="container mt-5 pt-5">
  <table class="table">
    <thead>
      <tr>
        <th scope="col">#</th>
        <th scope="col">User id</th>
        <th scope="col">Title</th>
        <th scope="col">Description</th>
        <th scope="col">Delete</th>
      </tr>
    </thead>
    <tbody>
      <tr *ngFor="let post of posts; index as i;">
        <th scope="row">{{post.id}}</th>
        <th>{{post.userId}}</th>
        <td>{{post.title}}</td>
        <td>{{post.body}}</td>
        <td>
          <button (click)="getPost(post.id)" type="button" class="btn btn-primary">Get post</button>
        </td>
        <td>
          <button (click)="deletePost(post.id)" type="button" class="btn btn-danger">Delete post</button>
        </td>
      </tr>
    </tbody>
  </table>
</div>
```

```
</tr>
</tbody>
</table>
</div>
```

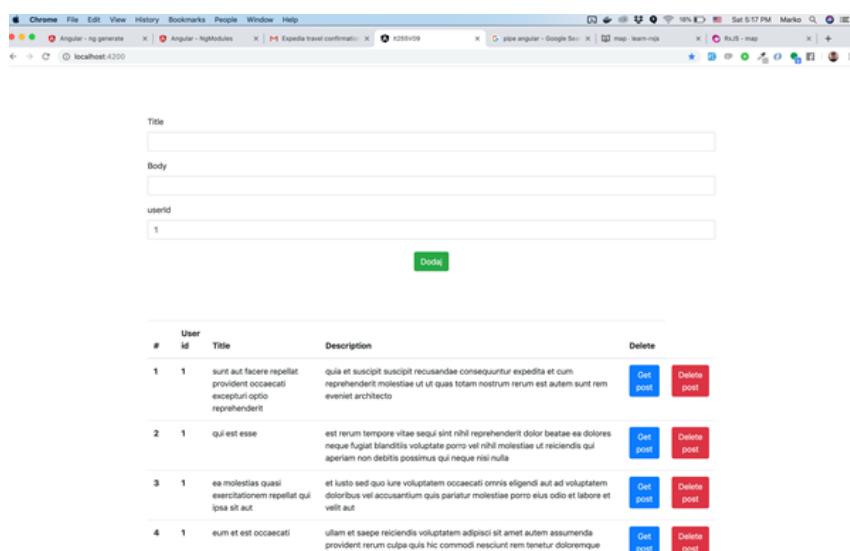
DEMO APLIKACIJE

Sledi demonstracija urađenog pokaznog primera.

Sledi demonstracija urađenog pokaznog primera. Pre same demonstracije sledi bitna napomena.

Kako je u pitanju mockup REST server, postovi neće biti kreirani niti obrisani na serveru, već samo lokalno, dodali smo da se prikazuju samo unutar aplikacije rezultati tih akcija.

Aplikacija se snima, prevodi i angažuje pozivom: [ng serve](#). Nakon toga se pokreće u veb pregledaču na [portu 4200](#). Izgled finalne aplikacije dat je na slici ispod:



Slika 8.3 Demo aplikacije [izvor: autor]

Kod vežbe je dostupan na sledećem linku:

<https://github.com/markorajevic/IT255-v09>

✓ Poglavlje 9

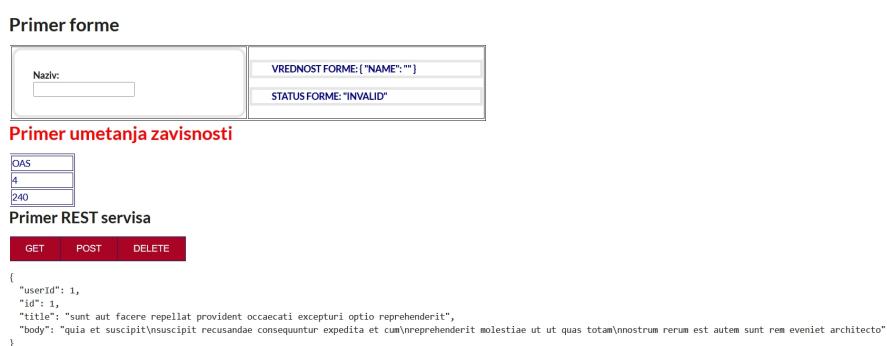
Individualna vežba 9

INDIVIDUALNA VEŽBA (TRAJANJE 90 MINUTA)

Samostalno vežbanje implementacije naučenon na predavanjima i pokaznoj vežbi.

Primenite iskustvo sa predavanja i pokaznih vežbi.

Preuzmite primer koji je rađen na dosadašnjim pokaznim vežbama. Koristeći JSON placeholder, demonstiran na predavanjima ili neki drugi, napravite 3 REST API za HTTP metode GET, POST, DELETE, tako da aplikacija na kojoj radimo dobije sledeći izgled:



Slika 9.1 Dodavanje REST funkcionalnosti

1. Neophodno je da obavite dodatavnje nove komponente `http`;
2. Pod čvorom imports, u `AppModule`, da registrujete `HttpClientModule`
3. Koristeći koncept umetanja zavisnosti u konstruktoru klase komponente `http.component.ts` umetnite instancu `HttpClient`.

✓ Poglavlje 10

Domaći zadatak 10

DOMAĆI ZADATAK (PREDVIĐENO VREME 120 MIN)

Samostalna izrada domaćeg zadatka.

Uradite domaći zadatka prema sledećim zahtevima:

1. Nastavite rad na projektu MetHotels,
2. Pogledajte lekciju iz predmeta CS230, Rad sa veb servisima
3. Koristeći MySql kreirajte bazu podataka MetHotels, sa jednom tabelom rooms
4. Kreirajte REST servise na backendu kao što smo to radili na predavanjima, vežbama, DZ i ispitu iz CS230
5. Krajnje tačke REST servisa (URI linkove) možete pronaći u generisanom JS REST Klijentu na serverskoj strani aplikacije
6. Alternativno, dozvoljeno je da umesto ovog pristupa koristite i neku "lažnu" bazu podataka sa room objektima u formi JSON (uputsvo: <https://blogs.sap.com/2021/01/14/how-to-create-a-fake-rest-api-with-json-server/>)
7. Iz vaše Angular aplikacije obezbedite mogućnost dodavanja, ažuriranja, brisanja i prikazivanja podataka o dostupnim sobama.

Domaći zadatak dodati na [Github](#) pod "commit - om" **IT255-DZ10** i poslati obaveštenje predmetnom asistentu o postavljenom domaćem zadatku.

✓ Poglavlje 11

Zaključak

ZAKLJUČAK

Lekcija se bavila izučavanjem mehanizama sinhronog rukovanja HTTP zahtevima.

Na samom početku lekcije je istaknuto kako radni okvir *Angular* distribuirala vlastitu *HTTP biblioteku* koja može biti veoma pogodna za obavljanje poziva ka eksternim API - jima.

Posebna tema bavljenja lekcije je bila problematika poziva ka eksternim serverima. U tom slučaju, neophodno je obezbediti da korisnik poseduje kontinuiranu interakciju sa stranicom veb aplikacije. To praktično znači, da je neophodno sprečiti da se stranica "zamrzne" dok ne stigne HTTP odgovor sa servera. Da bi navedeno bilo moguće realizovati, *HTTP zahtevi moraju biti asinhroni* i to je bio ključni zaključak uvodnog razmatranja lekcije.

U tom svetlu, lekcija ističe da rad sa asinhronim kodom, a to ima i istorijsko uporište u programiranju, može biti složeniji nego rukovanje sinhronim kodom.

U JavaScript jeziku, kao osnovi Angular TypeScript jezika, postoje tri pristupa za pomoć u rukovanju asinhronim kodom:

1. povratni pozivi (*callback*);
2. obećanja (*promises*);
3. osmatranja (*observable*).

U Angular okviru preferirani metod manipulisanja asinhronim kodom predstavlja treći slučaj - osmatranje, na čemu će biti i bazirano izlaganje u ovoj lekciji.

Imajući u vidi sve navedeno, opredeljeno je da lekcija pokrije tri ključne stvake:

1. analiza i diskusija osnovnog *HttpClient* primera;
2. kreiranje *YouTube* komponente za pretragu;
3. diskusija o API detaljima vezanim za *HttpClient* biblioteku.

Savladavanjem ove lekcije student je razumeo i sposobljen je da rukuje asinhronim *HTTP zahtevima* u *Angular* aplikacijama.

LITERATURA

Za pripremu lekcije korišćena je aktuelna pisana i elektronska literatura.

Pisana literatura:

1. Nate Murray, Felipe Coury, Ari Lerner, Carlos Taborda, ng-Book – The Complete Book on Angular 6, Fullstack.io, 2018
2. Cody Lindley, Frontend Handbook – 2017, Frontend masters, 2017
3. Sandeep Panda, AngularJS – From Novice to Ninja, SitePoint Pty. Ltd, 2014

Elektronska literatura:

4. <https://angular.io/>
5. <https://angular.io/tutorial>
6. <https://www.w3schools.com/angular/>
7. <https://www.tutorialspoint.com/angular4/>
8. <https://nodejs.org/en/>
9. <https://code.visualstudio.com/>
10. https://www.w3schools.com/whatis/whatis_htmldom.asp



IT255 - VEB SISTEMI 1

Rutiranje

Lekcija 11

PRIRUČNIK ZA STUDENTE

IT255 - VEB SISTEMI 1

Lekcija 11

RUTIRANJE

- ✓ Rutiranje
- ✓ Poglavlje 1: Značaj rutiranja u veb aplikacijama
- ✓ Poglavlje 2: Rutiranje na klijent strani
- ✓ Poglavlje 3: Pisanje prve rute
- ✓ Poglavlje 4: Komponente rutiranja u Angularu
- ✓ Poglavlje 5: Komponente u rutiranju
- ✓ Poglavlje 6: Podešavanje ruta
- ✓ Poglavlje 7: Parametri ruta
- ✓ Poglavlje 8: Hvatači ruta
- ✓ Poglavlje 9: Ugnježdene rute
- ✓ Poglavlje 10: Dodatni nastavni materijali
- ✓ Poglavlje 11: Pokazna vežba 10 (Trajanje 45 minuta)
- ✓ Poglavlje 12: Individualna vežba 10
- ✓ Poglavlje 13: Domaći zadatak 10
- ✓ Zaključak

Copyright © 2017 – UNIVERZITET METROPOLITAN, Beograd. Sva prava zadržana. Bez prethodne pismene dozvole od strane Univerziteta METROPOLITAN zabranjena je reprodukcija, transfer, distribucija ili memorisanje nekog dela ili čitavih sadržaja ovog dokumenta., kopiranjem, snimanjem, elektronskim putem, skeniranjem ili na bilo koji drugi način.

Copyright © 2017 BELGRADE METROPOLITAN UNIVERSITY. All rights reserved. No part of this publication may be reproduced, stored in a retrieval system or transmitted in any form or by any means, electronic, mechanical, photocopying, recording, scanning or otherwise, without the prior written permission of Belgrade Metropolitan University.

✓ Uvod

UVOD

Lekcija će se baviti konceptom rutiranja, veoma važnim problemom veb razvoja.

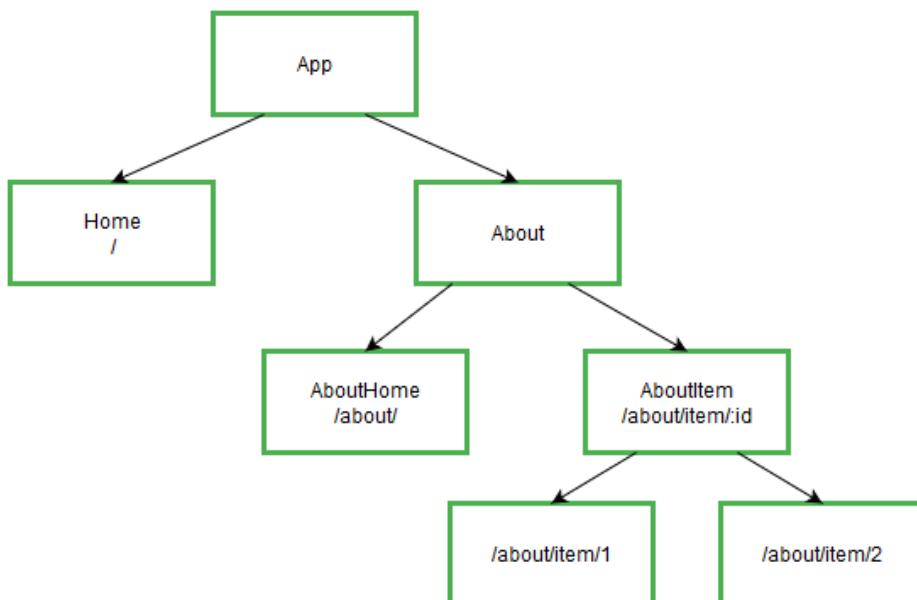
Kako predmet odmiče, dolaze na red sve napredniji *Angular* koncepti i principi. U nastavku, lekcija će se baviti konceptom rutiranja (*routing*), veoma važnim problemom veb razvoja.

U savremenom veb razvoju, pod pojmom rutiranja se podrazumeva proces razdvajanja veb aplikacije na različite oblasti, uglavnom, na osnovu pravila izvedenih iz konkretnih *URL* navedenih u veb pregledaču.

Na primer, ukoliko korisnik poseti putanju "/" veb sajta, on će verovatno posetiti početnu (*home*) rutu navedenog veb sajta. Sa druge strane, ukoliko korisnik poseti putanju "/about" veb sajta, zahtevaće kreiranje i prikazivanje veb stranice, na primer pod nazivom "*about*", i tako dalje (slika 1).

Budući na značaj, složenost i učestalost primene same problematike, u izlaganje, koje prati lekciju, neophodno je uključiti i odgovarajući pokazni primer sa ciljem lakšeg razumevanja i savladavanja lekcije.

Savladavanjem lekcije, studenti će biti sposobljeni ka koriste koncept rutiranja prilikom izrade vlastitih veb aplikacija u *Angular* radnom okviru.



Slika-1 Ilustracija primera rutiranja u veb aplikaciji [izvor: autor]

UVODNI VIDEO

Trajanje video snimka: 5min 53sek

Ova lekcija sadrži video materijal. Ukoliko želite da pogledate ovaj video morate da otvorite LAMS lekciju.

✓ Poglavlje 1

Značaj rutiranja u veb aplikacijama

RUTIRANJE U VEB APLIKACIJI

Rutiranjem u veb aplikaciji razdvajaju se različite oblasti veb aplikacije.

Definisanje ruta u Angular veb aplikacijama može biti značajno iz više razloga:

- razdvajaju se različite oblasti veb aplikacije;
- olakšano je održavanje sam aplikacije;
- omogućena je zaštita delova aplikacije bazirana na primeni izvesnih pravila.

Na primer, moguće je zamisliti da se kreira aplikacija za vođenje zaliha, po analogiji sa primerima demonstriranim u prethodnim lekcijama.

Ova lekcija sadrži video materijal. Ukoliko želite da pogledate ovaj video morate da otvorite LAMS lekciju.

Kada se prvi put poseti aplikacija moguće je primetiti polje za pretragu u koje je neophodno uneti ključnu reč za pretragu i dobijanje liste proizvoda koji odgovaraju unetoj ključnoj reči.

Nakon obavljene pretrage, klikom je moguće izabrati željeni proizvod nakon čega se otvara posebna stranica koja prikazuje detalje izabranog proizvoda. Budući da aplikacija predstavlja klijent stranu (*client-side*), nije tehnički potrebno menjati URL kada se menja veb stranica. Međutim, bilo bi dobro zastati u izlaganju za trenutak i postaviti pitanje "koje su posledice primene identičnog URL za sve stranice veb aplikacije"?

PROBLEM PRIMENE JEDINSTVENOG URL

Moguće je definisati niz različitih ruta za svaku od različitih aktivnosti aplikacije.

U prethodnom izlaganju je jasno postavljeno sledeće pitanje "koje su posledice primene identičnog URL za sve stranice veb aplikacije"? Odgovor na postavljeno pitanje moguće je izložiti u nekoliko teza:

- onemogućeno je osvežavanje (*refresh*) stranica;
- onemogućeno je čuvanje lokacije korisnika u aplikaciji;
- nije moguće pamćenje (*bookmark*) stranice sa ciljem kasnijeg vraćanja na tu stranicu;
- nije moguće deljenje URL - a stranice sa drugim korisnicima.

Sve je moguće posmatrati sa pozitivnije strane, primenom koncepta rutiranja dozvoljeno je definisanje URL stringa koji specificira gde bi u aplikaciji korisnik trebalo da se nalazi.

U konkretnom primeru, aplikacije za vođenje zaliha, moguće je definisati niz različitih ruta za svaku od različitih aktivnosti aplikacije, na primer:

- Inicijalna ruta može biti reprezentovana URL stringom `http://our-app/`.
- Kada korisnik poseti ovu stranicu, može biti preusmeren na rutu “`home`” određenu URL linkom `http://our-app/home`.
- Kada se pristupa sekciji aplikacije pod nazivom ‘`About Us`’ , URL string dobija oblik `http://our-app/about`.
- Ukoliko se drugom korisniku prosledi URL `http://our-app/about`, njegovim unosom, putem veb pregledača, drugi korisnik će sa naći na identičnoj stranici aktuelne veb aplikacije.

✓ Poglavlje 2

Rutiranje na klijent strani

FUNKCIONISANJE RUTIRANJA NA KLIJENT STRANI

Rutiranje na klijent i server strani je jako slični koncepti ali se razlikuju u implementaciji.

Za trenutno izlaganje je neophodno zamisliti da je na serverskoj strani (*server - side, backend*) kreiran kod koji omogućava rutiranje (iako za kompletiranje izlaganja ove lekcije to neće biti potrebno). U osnovi, kada se koristi serverska strana za rutiranje, kada stigne HTTP zahtev, server će angažovati odgovarajuću kontrolersku klasu u zavisnosti od dolaznog URL stringa.

Na primer, ako se koristi *Express.js* (<http://expressjs.com/en/guide/routing.html>), moguće je napisati kod priložen sledećim listingom:

```
var express = require('express');
var router = express.Router();

// definisanje rute "about"
router.get('/about', function(req, res) {
  res.send('About us');
});
```

Ako se koristi Ruby on Rails (<https://rubyonrails.org/>), moguće je željenu funkcionalnost postići sledećim kodom:

```
# routes.rb
get '/about', to: 'pages#about'

# PagesController.rb
class PagesController < ApplicationController::Base
  def about
    render
  end
end
```

Prikazani šablon varira u zavisnosti od radnog okvira (framework) koji ga implementira. Ali u svim navedenim slučajevima, postoji server koji prihvata zahtev i prosleđuje ga (rutira) ka kontroleru koji u okviru svoje odgovarajuće metode izvodi specifičnu akciju, u zavisnosti od putanje i parametara,

Rutiranje na klijent strani je jako sličan koncept ali se razlikuje u implementaciji. Kod rutiranja na strani klijenta nije potrebno kreiranje zahteva ka serveru za svaku URL promenu. U

kontekstu **Angular** aplikacija, govor se o aplikacijama sa jednom stranicom (*Single Page Apps- SPA*) iz razloga što server daje samo jednu stranicu ali je **JavaScript** zadužen za prikazivanje različitih stranica.

Ovde je ključno pitanje: "Kako je moguće obezbediti različite rute primenom JavaScript koda"?

PRIMENA SIDRO TAGA

U aplikacijama sa jednom stranicom koristi se sidro tag (anchor tag) kao URL klijentske strane.

Rutiranje na strani klijenta, za razliku od standardnog rutiranja na strani servera, započinje jednim zanimljivim trikom. Umesto korišćenja URL - a, kao što je praksa prilikom standardnog rutiranja na strani servera, u aplikacijama sa jednom stranicom (*Single Page Apps- SPA*) koristi se sidro tag (*anchor tag*) `<a>` kao URL klijentske strane.

Kao što je već poznato, navedeni tag ima tradicionalnu primenu u situacijama kada je neophodno obezbediti direktni link ka konkretnom sadržaju neke veb stranice, pri čemu ukazuje veb pregledaču da pomeri prikaz do mesta definisanog navedenim tagom. Na primer, ako se tag sidro definiše na sledeći način:

```
<!-- ... prethodni sadržaj stranice ... -->
<a name="about"><h1>About</h1></a>
```

i ako se nakon toga u veb pregledaču poseti URL `http://something/#about`, veb pregledač će automatski preusmeriti prikazivanje sadržaja ka `<H1>` tagu identifikovanog sidrom "`about`".

Još mudriji način, koji primenjuju radni okviri za razvoj klijentskih aplikacija (poput Angular-a), predstavlja upotreba sidro tagova formatiranih kao putanje. Na primer, ruta "`about`", u aplikacijama sa jednom stranicom, mogla bi da bude formatirana na sledeći način: `http://something/#/about`. U praksi je navedeno poznato kao "heš" bazirano rutiranje (*hash-based routing*). Izvođenjem ovog trika sve deluje kao primena "normalnog" URL iz razloga što sidro započinje kosom crtom (`/about`).

EVOLUCIJA ILI HTML5 RUTIRANJE NA KLIJENT STRANI

U Angular okviru, HTML5 predstavlja podrazumevani režim rada.

Sa uvođenjem **HTML5**, veb pregledači su stekli mogućnost za programsko kreiranje novih unosa u istoriju pregledača koji menjaju prikazani **URL** bez potrebe za novim zahtevom. Navedeno se postiže primenom metode `history.pushState()` koja otkriva istoriju navigacije pregledača **JavaScript**-u. Otuda, umesto oslanjanja na trik sa sidro tagom za navigaciju kroz rute, savremeni radni okviri se oslanjanju na primenu metode `pushState()` za izvođenje manipulacije istorijom navigacije bez ponovnih učitavanja.

U Angular okviru, HTML5 predstavlja podrazumevani režim rada. Kasnije, u lekciji će posebno biti pokazano kako se prelazi sa HTML5 režima u stariji režim koji podrazumeva primenu usidrenja.

Takođe, od velike je važnosti, u ovom delu izlaganja, skrenuti pažnju na sledeće dve stvari:

1. Ne podržavaju svi pregledači **HTML5** način rutiranja, tako da ako bi trebalo da se obezbedi podrška za starije pregledače, programeri mogu biti "zaglavljeni" neko vreme sa "heš" baziranim rutiranjem (*hash-based routing*);
2. Server mora da podržava **HTML5** bazirano rutiranje.

Ako u ovom trenutku postoje neke nejasnoće, o svemu navedenom će biti detaljno govora u ovoj lekciji.

▼ Poglavlje 3

Pisanje prve rute

PISANJE PRVE RUTE PRIMENOM PUTANJA

U Angular-u rute se podešavaju mapiranjem putanja ka komponentama koje će njima rukovati.

Angular dokumentacija preporučuje primenu [HTML5](#) režima rutiranja (<https://angular.io/guide/router#!%23browser-url-styles>). Međutim, imajući u vidu prethodno navedena ograničenja, zbog jednostavnosti biće primenjen režim sa "heš" baziranim rutiranjem ([hash-based routing](#)) u primeru koji služi kao podrška izlaganju ove lekcije.

U Angular radnom okviru rute se podešavaju mapiranjem putanja ka komponentama koje će njima rukovati. Za primer će biti kreirana aplikacija koja će posedovati više ruta:

- ruta glavne stranice ([home](#)) će koristiti putanju: `/#/home`;
- ruta stranice "O nama" ([about](#)) će koristiti putanju: `/#/about`;
- ruta stranice "Kontakt" ([contact](#)) će koristiti putanju: `/#/contact`;

NAPOMENA: Kada korisnik navede korensku putanju (`/#/`) biće obavljeno preusmeravanje na putanju [home](#).

▼ Poglavlje 4

Komponente rutiranja u Angularu

OSNOVNE KOMPONENTE RUTIRANJA

Postoje tri osnovne komponente koje programeri koriti prilikom podešavanja ruta u Angular okviru.

Postoje tri osnovne komponente koje programeri koriti prilikom podešavanja ruta u Angular okviru:

- *Routes* - opisuje rute podržane aplikacijom;
- *RouterOutlet* - komponenta koja ukazuje Angular-u gde da sačuva sadržaj svake od ruta;
- *RouterLink* - direktiva koja se koristi za povezivanje sa rutama.

Navedene komponente je neophodno detaljno sagledati.

IMPORT PAKETA ANGULAR/ROUTER

Sa ciljem primene rutiranja u Angularu, neophodno je obaviti uvoz paketa angular/router

Sa ciljem primene mehanizma rutiranja u Angularu (*rutera*), neophodno je obaviti uvoz paketa `@angular/router` u `app.module.ts` datoteci. To je moguće obaviti na sledeći način:

```
import {  
  RouterModule,  
  Routes  
 } from '@angular/router';
```

Sada su ispunjeni svi uslovi za podešavanje rutera.

DEFINISANJE RUTA

Za definisanje ruta u aktuelnoj aplikaciji, neophodno je kreirati Routes konfiguraciju.

Za definisanje ruta u aktuelnoj aplikaciji, neophodno je kreirati *Routes* konfiguraciju, a potom iskoristiti poziv `RouterModule.forRoot(routes)` za obezbeđivanje svih neophodnih zavisnosti u aplikaciji, neophodnih za upotrebu *rutera*.

Proširenjem datoteke `app.module.ts` dodaje se kod kojim se definišu rute:

```
const routes: Routes = [
  // osnovne rute
  { path: '', redirectTo: 'home', pathMatch: 'full' },
  { path: 'home', component: HomeComponent },
  { path: 'about', component: AboutComponent },
  { path: 'contact', component: ContactComponent },
  { path: 'contactus', redirectTo: 'contact' },

  // provera korisnika
  { path: 'login', component: LoginComponent },
  {
    path: 'protected',
    component: ProtectedComponent,
    canActivate: [ LoggedInGuard ]
  },

  // ugnježdeno
  {
    path: 'products',
    component: ProductsComponent,
    children: childRoutes
  }
];
```

Neophodno je primetiti par stvari u vezi sa rutama:

- `path` ukazuje na URL kojim ruta rukuje;
- `component` povezuje rutu sa komponentom koje rukuje rutom;
- opcionalno, `redirectTo` ukazuje na preusmeravanje određene putanje na postojeću rutu.

U ovoj lekciji će biti skenirana svaka od navedenih ruta. Cilj rute je da definiše koja će komponenta rukovati određenom putanjom.

PREUSMERAVANJE

Primenom `redirectTo` na definiciju rute, veb pregledač obavlja preusmeravanje na drugu rutu.

Iz prethodnog listinga je bilo moguće primetiti da su se prilikom podešavanja ruta, u formi (ključ, vrednost) kao ključevi pojavili: `path`, `component` i `redirectTo`. Kada se primenjuje ključ `redirectTo` na definiciju rute, ukazuje se ruteru da je, kada se poseti putanja konkretnе rute, neophodno je da veb pregledač obavi preusmeravanje na drugu rutu.

Poseban deo prethodnog listinga, istaknut crvenom bojom na sledećoj slici, govori da ukoliko se poseti putanja `http://localhost:4200/#/`, u veb pregledaču će biti obavljeno preusmeravanje na rutu "`/home`".

```
const routes: Routes = [
  // basic routes
  { path: '', redirectTo: 'home', pathMatch: 'full' },
  { path: 'home', component: HomeComponent },
  { path: 'about', component: AboutComponent },
  { path: 'contact', component: ContactComponent },
  { path: 'contactus', redirectTo: 'contact' },
]
```

Slika 4.1 Definicije ruta sa preusmeravanjem [izvor: autor]

Sldeći primer predstavlja ruta "*contactus*", a to je istaknuto plavom bojom na slici. U ovom slučaju ako se poseti URL <http://localhost:4200/#/contactus>, veb pregledač će obaviti preusmeravanje na rutu "*/contact*".

INSTALIRANJE RUTA

Kada postoji kolekcija podešenih ruta, neophodno je obaviti njihovo instaliranje

Sada kada postoji kolekcija podešenih ruta *Routes*, neophodno je obaviti njihovo instaliranje. Da bi rute mogle da budu upotrebljene u konkretnoj veb aplikaciji, neophodno je obaviti sledeće u NgModul - u:

1. importovanje *RouterModule* -a;
2. instaliranje ruta pozivom *RouterModule.forRoot(routes)* pod ključem *imports* dekoratora *@NgModule*.

Sada je neophodno kodom demonstrirati navedeno, podešavanje ruta u dekoratoru *@NgModul* za konkretnu aplikaciju (datoteka: *src/app/app.module.ts*)

```
const routes: Routes = [
  // osnovne rute
  { path: '', redirectTo: 'home', pathMatch: 'full' },
  { path: 'home', component: HomeComponent },
  { path: 'about', component: AboutComponent },
  { path: 'contact', component: ContactComponent },
  { path: 'contactus', redirectTo: 'contact' },
]

// ovde će ići još podešavanja u nastavku

@NgModule({
  declarations: [
    AppComponent,
    // ovde će ići deklaracije komponenata
    // koje će biti kreirane
  ],
  imports: [
    BrowserModule,
]
```

```
FormsModule,  
HttpClientModule,  
RouterModule.forRoot(routes), // <-- intaliranje ruta  
  
// dodati ovo za postojeći modul potomka  
ProductsModule  
],
```

PRIMENA DIREKTIVE ROUTEROUTLET

Da bi Angular-u kreirao i prikazao inovirani sadržaj, vezan za poziv rute, koristi se RouterOutlet.

U *Angular* aplikacijama, kada se ruta promeni neophodno je zadržati spoljašnji izgled *HTML* šablona i omogućiti izmenu samo unutrašnje sekcije koja se kreira na osnovu komponente rute.

Da bi Angular-u bilo naglašeno gde na stranici da kreira i prikaže inovirani sadržaj, vezan za poziv rute, koristi se direktiva *RouterOutlet*.

Dekorator *@Component* poseduje ključ *template* kojim se specificira izvesna struktura izdeljena na sekcije primenom *<div>* taga (*division = sekcija*). Sekcija za navigaciju, putem ruta, u šablonu komponente određena je primenom direktive pod nazivom *router-outlet*. Elementom *<router-outlet>* ukazuje se gde će sadržaj svake rute, u šablonu, biti kreiran i prikazan.

Da bi navedeno bilo jasno, neophodno je prikazati kod klase (prvi listing) i šablonu, koji ima ulogu omotača navigacije, glavne komponente aplikacije.

```
@Component({  
  selector: 'app-root',  
  templateUrl: './app.component.html',  
  styleUrls: ['./app.component.css']  
})  
export class AppComponent {  
  constructor(private router: Router) {}  
}
```

Kod šablonu za klasu *AppComponent* (obe datoteke se nalaze u projektu na lokaciji: *src/app/*) priložen je sledećim listingom.

```
// Šablon glavne komponente AppComponent - listing  
  
<div class="page-header">  
  <div class="container">  
    <h1>Rutiranje - primer</h1>  
    <div class="navLinks">  
      <a [routerLink]=["'/home']>Početna</a>
```

```

<a [routerLink]=["'/about']> O nama</a>
<a [routerLink]=["'/contact']> Kontakt</a>
|
<a [routerLink]=["'/products']>Proizvodi</a>
<a [routerLink]=["'/login']> Prijavljanje</a>
<a [routerLink]=["'/protected']> Zaštićeno</a>
</div>
</div>
</div>

<div id="content">
  <div class="container">
    <router-outlet></router-outlet>
  </div>
</div>

```

Linijama koda 18 - 22, definisana je sekcija koja sadrži **router-outlet** element, ispod menija za navigaciju. U ovoj sekciji, izborom linka **/home** biće učitan i prikazan sadržaj **HomeComponent** šablonu. Potpuno je isto za ostale opcije iz menija.

PRIMENA DIREKTIVE ROUTERLINK

RouterLink je direktiva za povezivanje sa rutama bez ponovnog učitavanja stranica.

Budući da je sada jasno gde će šabloni ruta biti prikazivani, neophodno je da se pronađe mehanizam koji će zahtevati od Angular - a da obavi navigaciju ka specifičnoj ruti. Najjednostavnija ideja, koja bi programeru mogla da padne na pamet, jeste upotreba direktnog rutiranja putem **HTML**, na sledeći način:

```
<a href="/#/home">Početna</a>
```

Međutim, ukoliko se uradi na ovaj način, veoma brzo će postati jasno da ponovno učitavanje stranica, na osnovu pokretanja linkova, definitivno nije poželjno prilikom programiranja aplikacija sa jednom stranicom (kao što su Angular aplikacije).

Za rešavanje navedenog problema, Angular obezbeđuje rešenju koje može biti upotrebljeno za povezivanje sa rutama bez ponovnog učitavanja stranica. Rešenje je dato u formi primene direktive **RouterLink**. Ovom direktivom je omogućeno programerima da pišu veoma korisne linije koda koristeći specijalnu sintaksu



```

<div class="page-header">
  <div class="container">
    <h1>Rutiranje - primer</h1>
    <div class="nav-links">
      <a [routerLink]=["'/home"]>Početna</a>
      <a [routerLink]=["'/about']> O nama</a>
      <a [routerLink]=["'/contact']> Kontakt</a>
      |
      <a [routerLink]=["'/products']>Proizvodi</a>
      <a [routerLink]=["'/login']> Prijavljanje</a>
      <a [routerLink]=["'/protected']> Zaštićeno</a>
    </div>
  </div>
</div>

<div id="content">
  <div class="container">
    <router-outlet></router-outlet>
  </div>
</div>

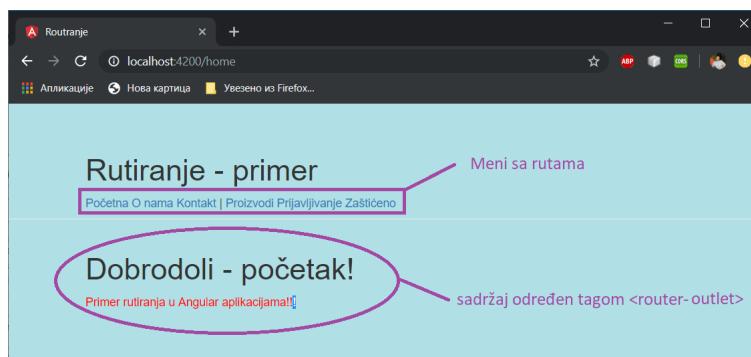
```

Slika 4.2 Primena direktive RouterLink [izvor: autor]

Sa priložene slike je moguće primetiti, leva strana izraza, da se `[routerLink]` primenjuje kao direktiva na određeni element (u ovom slučaju tag).

Na desnoj strani izraza nalazi se niz sa putanjom rute kao prvim elementom, poput "`['/home']`" ili "`['/about']`" koji ukazuje prema kojoj ruti se vrši navigacija kada se klikne na određeni element.

Možda može da deluje čudno da je vrednost za `routerLink` string u formatu niza koji sadrži string ("`['/home']`", na primer). Ovo je pogodno iz razloga što je neophodno obezbediti dodatne informacije prilikom povezivanja sa rutama. O ovome će biti više govora kada se bude diskutovalo o rutama potomcima (*child routes*) i parametrima ruta. Za sada, u glavnoj komponenti, koriste se samo nazivi ruta.



Slika 4.3 Sekcije za izbor ruta i prikazivanje sadržaja odgovarajuće komponente [izvor: autor]

POČETNI LINK APLIKACIJE I NGMODULE

Angular Router takođe koristi href tag za određivanje kako da konstruiše informacije rutiranja.

Neophodno je dati objašnjenje šta se dešava kada se pokrene ovakva aplikacija. Sledećim listingom je dat HTML kod stranice `index.html` koji učitava centralnu komponentu aplikacije.

```
<!doctype html>
<html>
<head>
  <meta charset="utf-8">
  <title>Routranje</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 style="background-color:powderblue;">
  <app-root>Loading...</app-root>
</body>
</html>
```

Ovaj kod je veoma poznat, sa izuzetkom jedne linije:

```
<base href="/">
```

Linija ukazuje na osnovni HTML tag, koji se tradicionalno koristi da ukaže pregledaču gde da potraži slike i druge resurse deklarisane primenom relativnih putanja.

Angular Router takođe koristi navedeni tag za određivanje kako da konstruiše informacije rutiranja.

Na primer, ukoliko postoji ruta sa putanjom `/hello` i osnovni element deklařiše `href="/app"`, aplikacija će koristiti `app/#` kao konkretnu putanju.

Ponekad, programeri *Angular* aplikacija nemaju pristup glavnoj HTML (`head`) sekciji. Ovo je moguće da se desi u situaciji ponovnog korišćenja zaglavljia i podnožja (header, footer) velikih, postojećih aplikacija.

Na sreću i za ovo postoji rešenje. Moguće je programski definisati osnovnu putanju aplikacije podešavanjem `NgModul` - a primenom provajdera `APP_BASE_HREF`, kao na sledećoj slici:

```
@NgModule({
  declarations: [ RoutesDemoApp ],
  imports: [
    BrowserModule,
    RouterModule.forRoot(routes) // <-- routes
  ],
  bootstrap: [ RoutesDemoApp ],
  providers: [
    { provide: LocationStrategy, useClass: HashLocationStrategy },
    { provide: APP_BASE_HREF, useValue: '/' } //<-- podešavanje osnovne putanje
  ]
})
```

Slika 4.4 Programsko definisanje osnovne putanje [izvor: autor]

Dodavanje `{ provide: APP_BASE_HREF, useValue: '/' }`, pod ključem providers, ekvivalentno je primeni `<base href="/">` u HTML zaglavlju.

▼ Poglavlje 5

Komponente u rutiranju

KOMPONENTA HOMECOMPONENT

Za svaku od kreiranih ruta, neophodno je obaviti kreiranje odgovarajuće komponente.

Za svaku od kreiranih **ruta** u **Angular** programu, neophodno je obaviti kreiranje odgovarajuće komponente koja će da upravlja kreiranjem i prikazivanjem sadržaja na osnovu izabrane rute. Pre nego što bude kreirana glavna komponenta aplikacije, neophodno je obaviti kreiranje 3 konkretnih komponenti:

- **HomeComponent** - prikazuje podatke da korisnik nalazi na početnoj stranici;
- **AboutComponent** - prikazuje stranicu sa opisom programerskog tima koji je kreirao aplikaciju;
- **ContactComponent** - prikazuje stranicu sa kontakt podacima programerskog tima koji je kreirao aplikaciju;

Prva komponenta koja će biti kreirana jeste komponenta **HomeComponent** i ona će biti zadužena za prikazivanje jednostavnog pogleda prilikom posete linka **/home** i aktiviranja odgovarajuće rute.

Sledećim listingom je definisan šablon komponente kojim će biti prikazan navedeni sadržaj (datoteka: **src/app/home/home.component.html**):

```
<h1>Dobrodoli na početak!</h1>
<font color="red">
<p>Primer rutiranja u Angular aplikacijama!!!</p>
</font>
```

Da bi definicija komponente bila uspešno završena, neophodno je dodati u nju i definiciju klase komponente (datoteka: **src/app/home/home.component.ts**):

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

@Component({
  selector: 'app-home',
  templateUrl: './home.component.html',
  styleUrls: ['./home.component.css']
})
export class HomeComponent implements OnInit {

  constructor() { }
```

```
ngOnInit() {  
}  
}
```

Selektorom je definisan tag `app-home`, a sa `templateUrl` je definisan prethodno prikazani šablon (`home.component.html`) koji će biti učitan nakon poziva prikazanog taga u šablonu roditeljske komponente.

KOMPONENTA ABOUTCOMPONENT

AboutComponent - prikazuje stranicu sa opisom programerskog tima koji je kreirao aplikaciju.

Sledeća komponenta koju je neophodno kreirati, pre definisanja roditeljske komponente, jeste komponenta `AboutComponent`. Ova komponenta omogućava prikazivanje stranice sa opisom programerskog tima koji je kreirao aplikaciju, na način definisan kodom šablonu ove komponente (datoteka: `src/app/about/about.component.html`):

```
<h1>0 nama : </h1>  
<font color="red">  
<p>IT255 - Veb sistemi 1 (2019/2020)</p>  
</font>
```

Da bi definicija komponente bila uspešno završena, neophodno je dodati u nju i definiciju odgovarajućeklase komponente (datoteka: `src/app/home/about.component.ts`):

```
import { Component, OnInit } from '@angular/core';  
  
@Component({  
  selector: 'app-about',  
  templateUrl: './about.component.html',  
  styleUrls: ['./about.component.css']  
})  
export class AboutComponent implements OnInit {  
  
  constructor() {}  
  
  ngOnInit() {}  
}
```

Selektorom je definisan tag `app-about`, a sa `templateUrl` je definisan prethodno prikazani šablon (`about.component.html`) koji će biti učitan nakon poziva prikazanog taga u šablonu roditeljske komponente.

KOMPONENTA CONTACTCOMPONENT

ContactComponent - prikazuje stranicu sa kontakt podacima tima koji je kreirao aplikaciju.

Poslednja komponenta koju je neophodno kreirati, pre definisanja roditeljske komponente, jeste komponenta [ContactComponent](#). Ova komponenta omogućava prikazivanje stranice sa kontakt podacima programerskog tima koji je kreirao aplikaciju, na način definisan kodom šablonu ove komponente (datoteka: [src/app/about/contact.component.html](#)):

```
<h1>Kontakt : </h1>
<font color="red">
<p>Univerzitet Metropolitan
<br/> Tadeuša Košćuška 63
<br/> Beograd
</p>
</font>
```

Da bi definicija komponente bila uspešno završena, neophodno je dodati u nju i definiciju odgovarajućeklase komponente (datoteka: [src/app/home/contact.component.ts](#)):

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

@Component({
  selector: 'app-contact',
  templateUrl: './contact.component.html',
  styleUrls: ['./contact.component.css']
})
export class ContactComponent implements OnInit {

  constructor() { }

  ngOnInit() {
  }
}
```

Selektrom je definisan tag [app-contact](#), a sa [templateUrl](#) je definisan prethodno prikazani šablon ([contact.component.html](#)) koji će biti učitan nakon poziva prikazanog taga u šablonu roditeljske komponente.

GLAVNA KOMPONENTA APLIKACIJE APPMODULE

Neophodno je obaviti povezivanje komponeneta u celinu kreiranjem centralne komponente.

Pošto su, u prethodnom izlaganju, kreirane tri komponente, koje odgovaraju konkretnim rutama, neophodno je obaviti njihovo povezivanje u celinu kreiranjem glavne komponente aplikacije (*root-level* komponenta).

U prvom koraku je neophodno obaviti početna podešavanja i učitavanja u glavnem modulu (klasa *AppModule*) aplikacije (datoteka: */src/app/app.module.ts*):

```
import { BrowserModule } from '@angular/platform-browser';
import { NgModule } from '@angular/core';
import { FormsModule } from '@angular/forms';
import { HttpClientModule } from '@angular/common/http';
import {
  RouterModule,
  Routes
} from '@angular/router';
```

U nastavku kreiranja koda glavnog modula, neophodno je obaviti učitavanje kreiranih komponenata.

```
import { AppComponent } from './app.component';
import { HomeComponent } from './home/home.component';
import { ContactComponent } from './contact/contact.component';
import { AboutComponent } from './about/about.component';
```

GLAVNA KOMPONENTA APLIKACIJE - ŠABLON KOMPONENTE

Za glavnu komponentu biće neophodno obezbediti primenu dveju direktiva za rutiranje.

Za glavnu komponentu biće neophodno obezbediti primenu dveju direktiva za rutiranje:

- *RouterOutlet* i
- *RouterLink*.

Navedene direktive, zajedno sa ostalim direktivama za rutiranje, učitane su dodavanjem modula *RouterModule* u *import* sekciju dekoratora *@NgModule*.

Još jednom, direktivom *RouterOutlet* je određeno gde će u šablonu sadržaj rute biti prikazan, a ovo je reprezentovano odgovarajućim kodom u šablonu *AppComponent* komponente.

Direktiva *RouterLink* kreira navigacione linkove ka rutama.

NAPOMENA: U šablonu je moguće primetiti postojanje još tri rute: products, login i protected. One će biti od značaja u nastavku, a sada je fokus na tri već kreirane rute.

Sledi listing šablonu glavne komponente aplikacije sa implementiranim direktivama *RouterOutlet* i *RouterLink* (datoteka: */src/app/app.component.html*) :

```
<div class="page-header">
  <div class="container">
    <h1>Rutiranje - primer</h1>
    <div class="navLinks">
      <a [routerLink]="/home">Početna</a>
      <a [routerLink]="/about"> O nama</a>
      <a [routerLink]="/contact"> Kontakt</a>
      |
      <a [routerLink]="/products">Proizvodi</a>
      <a [routerLink]="/login"> Prijavljanje</a>
      <a [routerLink]="/protected"> Zaštićeno</a>
    </div>
  </div>
</div>

<div id="content">
  <div class="container">
    <router-outlet></router-outlet>
  </div>
</div>
```

Koristeći `[routerLink]` Angular-u se daje instrukcija da preuzme kontrolu nad "click" događajem i, nakon toga, da usmeri rutiranje ka pravom mestu, na osnovu definicije rute.

✓ Poglavlje 6

Podešavanje ruta

DEKLARISANJE I INSTALIRANJE RUTA

Deklarisanje ruta se vrši u formi niza objekata koji odgovaraju tipu Routes.

Nastavlja se usavršavanje definicije glavnog modula aplikacije deklarisanjem ruta u formi niza objekata koji odgovaraju tipu podataka [Routes](#):

```
const routes: Routes = [
  // osnovne rute
  { path: '', redirectTo: 'home', pathMatch: 'full' },
  { path: 'home', component: HomeComponent },
  { path: 'about', component: AboutComponent },
  { path: 'contact', component: ContactComponent },
  { path: 'contactus', redirectTo: 'contact' },
```

Zatim je neophodno dodati još podešavanja u dekorator [@NgModule](#):

```
@NgModule({
  declarations: [
    AppComponent,
    HomeComponent,
    ContactComponent,
    AboutComponent,
    // donje dve komponente su od značaja za izlaganje koje tek sledi
    LoginComponent,
    ProtectedComponent,
  ],
  imports: [
    BrowserModule,
    FormsModule,
    HttpClientModule,
    RouterModule.forRoot(routes), // <-- instaliranje ruta

    // dodati ovo za postojeći modul potomka
    ProductsModule
  ],
  providers: [
    // sklonite ovo za "hash-bang" rutiranje
    // { provide: LocationStrategy, useClass: HashLocationStrategy }
    AUTH_PROVIDERS,
    LoggedInGuard
```

```
],
bootstrap: [AppComponent]
})
export class AppModule { }
```

Ako se dobro pogleda priloženi listing moguće je, u *import* sekciji, primetiti poziv funkcije *RouterModule.forRoot(routes)* koja na ulazu preuzima kolekciju ruta, podešava ruter i vraća listu zavisnosti poput: *RouteRegistry*, *Location* i još nekih neophodnih za uspešno obavljanje zadataka rutiranja.

Pod ključem *providers* postoji jedan zanimljiv objekat, doduše trenutno je obeležen kao komentar, istaknut sledećim kodom:

```
{ provide: LocationStrategy, useClass: HashLocationStrategy }
```

Upravo u narednom izlaganju će detaljno biti govora o strategijama rutiranja.

STRATEGIJE RUTIRANJA

Strategija lokacije je način na koji Angular obrađuje i kreira putanje koje izlaze/ulaze u rute.

Strategija lokacije je način na koji Angular obrađuje (parsira) i kreira putanje koje izlaze ili ulaze u rute, respektivno.

Podrazumevana strategija je označena pod nazivom *PathLocationStrategy* i poznata je još pod nazivom HTML5 rutiranje. Prilikom korišćenja navedene strategije, rute su reprezentovane regularnim putanjama, na primer */home*, */contact* i tako dalje.

Moguće je promeniti strategiju lokacije, za konkretnu aplikaciju, povezivanjem klase *LocationStrategy* sa novom, konkretnom klasom strategije. Umesto klase *PathLocationStrategy* moguće je koristiti klasu *HashLocationStrategy*.

Razlog zašto se u konkretnom primeru koristi *HashLocationStrategy*, kao podrazumevana strategija, krije se u činjenici da ako bi bilo korišćeno HTML5 rutiranje, konkretni URL rutiranja bi se završavali u formi regularnih putanja (a to znači da ne bi koristili oznaku *#/sidro*). Na ovaj način, ruta bi radila kada bi bio obavljen klik na odgovarajući link i obavila bi navigaciju na klijent strani, na primer iz */about* ka */contact*. Ukoliko bi stranica bila osvežena, umesto da se traži od servera odgovor za osnovni (*root*) URL (koji se obrađuje), biće prosleđen zahtev */about* ili */contact*. Budući da mu nije poznata stranica koja se učitava pozivom */about*, server će vratiti kod grešku sa kodom *404 - "Not Found"*.

Navedena strategija, ako se izabere kao podrazumevana, funkcioniše sa heš putanjama, poput *#/home* ili *#/contact* koje server razume kao */putanja* (ovo je podrazumevana strategija u Angular 1 okviru).

Ako je cilj da aplikacija koristi navedenu novu strategiju, neophodno je obaviti uvoz (*import*) klasu *LocationStrategy* i *HashLocationStrategy*, a zatim dodati izabranoj strategiji lokacije u *NgModule*.

POKRETANJE APLIAKCIJE

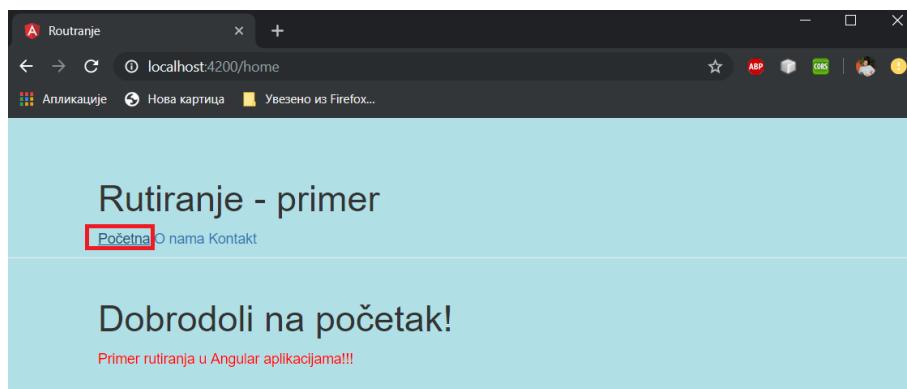
Testira se funkcionisanje rutiranja.

Testira se funkcionisanje rutiranja sledećim pozivom u terminalu razvojnog okruženja:

```
ng serve
```

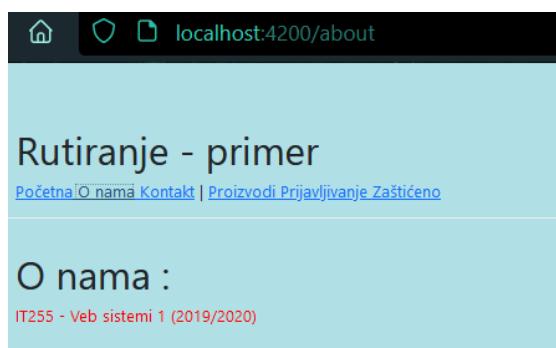
Ubrzo aplikacija se prevodi i moguće je unosom `localhost:4200` proveriti šta je do sada urađeno.

Učitava se početna ruta, a to je prikazano sledećom slikom.



Slika 6.1 Početna ruta aplikacije [izvor: autor]

Klikom na link "[O nama](#)" učitava se ruta "[About](#)", a to je prikazano sledećom slikom.



Slika 6.2 Ruta About [izvor: autor]

Klikom na link "[Kontakt](#)" učitava se ruta "[Contact](#)", a to je prikazano sledećom slikom.



Slika 6.3 Ruta Contact [izvor: autor]

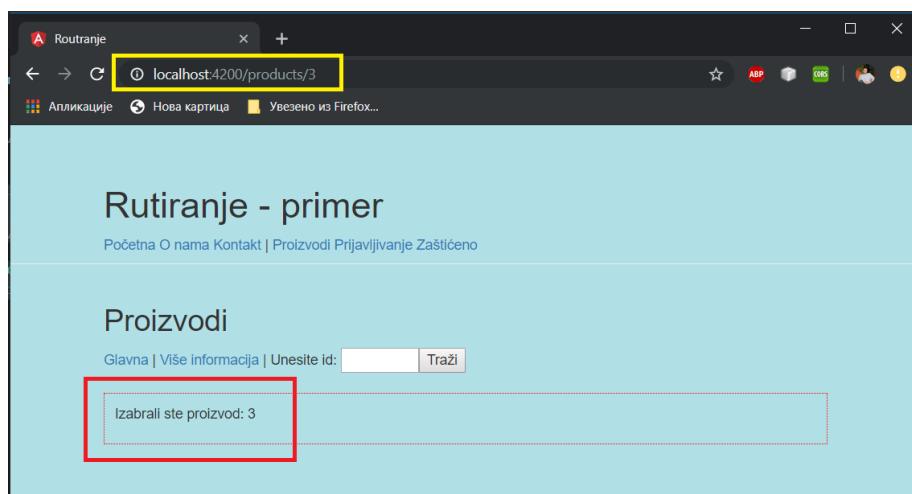
✓ Poglavlje 7

Parametri ruta

NAVIGACIJA KA SPECIFIČNOM RESURSU

U Angular aplikacijama često se dešava navigacija prema specifičnom resursu

U Angular aplikacijama često se dešava navigacija prema specifičnom resursu. Na primer, postoji veb sajt sa novostima vezanim za veliki broj proizvoda. Svaki od proizvoda poseduje vlastiti ID. Ukoliko je potrebno dobiti podatke o proizvodu čiji je ID 3, neophodno je obaviti navigaciju ka konkretnom proizvodu koristeći URL: /products/3 (sledeća slika). Na sličan način, ukoliko je potrebno dobiti podatke o proizvodu čiji je ID 4, neophodno je obaviti navigaciju ka konkretnom proizvodu koristeći URL: /products/4, i tako dalje.



Slika 7.1 Dobijanje informacije preko parametra rute [izvor: autor]

Očigledno da nije rešenje pisanje specifičnih ruta za svaki od dostupnih proizvoda. Umesto toga moguće je koristiti odgovarajuću promenljivu ili parametar rute.

Da bi ruta koristila parametar primenjuje se specifična sintaksa:

```
/ ruta/:parametar
```

U konkretnom primeru, a to se vidi i sa slike, specifikacija rute sa parametrom može biti urađena na sledeći način:

```
/product/:id
```

Dodavanje parametra u konfiguraciju rutera moguće je obaviti na sledeći način:

```
const routes: Routes = [  
  { path: 'product/:id', component: ProductComponent },  
];
```

Ukoliko se poseti ruta `/product/123`, vrednost 123 će biti prosleđena u rutu kao njen id parametar.

Sada se otvara novi problem. Kako je moguće vratiti parametar konkretne rute?

PRIMENA INTERFEJSA ACTIVATEDROUTE

Sa ciljem upotrebe parametara ruta neophodno je importovanje interfejsa `ActivatedRoute`.

Sa ciljem upotrebe parametara ruta, prvi korak kojeg je neophodno izvesti jeste importovanje interfejsa `ActivatedRoute`. U klasi komponente, zadužene za upravljanje rutom, dodaje se sledeći kod:

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

U sledećem koraku, neophodno je obaviti umetanje zavisnosti putem konstruktora navedene klase. U konstruktoru klase komponente se obavlja umetanje objekta tipa `ActivatedRoute`. Na primer, neka u nizu specificiranih ruta (tip `Routes`), u modulu komponente (npr. klasa `ProductModule` iz fajla `product.module.ts`) postoji i sledeći objekat:

```
const routes: Routes = [  
  { path: 'product/:id', component: ProductComponent }  
];
```

Sada je moguće u klasi komponente `ProductComponent` dodati `ActivatedRoute` objekat kao jedan od parametara konstruktora:

```
export class ProductComponent {  
  id: string;  
  
  constructor(private route: ActivatedRoute) {  
    route.params.subscribe(params => { this.id = params['id']; });  
  }  
}
```

Primećuje se da je `route.params` je osmatran (*Observable*). Moguće je izvući vrednost parametra u primenjivi oblik primenom metode `.subscribe()`. U ovom slučaju, pridružuje se vrednost `params['id']` objektnoj osobini `id` klase komponente.

Sada, ukoliko se u pregledaču poseti `/product/123`, atribut `id` komponente će dobiti vrednost 123.

▼ Poglavlje 8

Hvatači ruta

PRIMENA ZAŠTIĆENIH RUTA

Postoje situacije kada je neophodno izvesti određenu akciju za promenu ruta, npr. login.

Postoje situacije kada je neophodno izvesti određenu akciju za promenu ruta. Klasičan primer je primena provere korisnika (*authentication*) pre promene rute. Za demonstraciju je neophodno prepostaviti postojanje dve nove rute `/login` i `/protected`.

Pristup zaštićenoj (`/protected`) ruti je dozvoljen samo ukoliko je korisnik uneo ispravne podatke za korisničko ime (`username`) i lozinku (`password`) na stranici za prijavljivanje (`/login`).

Da bi navedeno bilo moguće, neophodno je dodati hvatač (*hook*) u životni ciklus rutera i zatražiti obaveštenje o aktiviranju zaštićene rute. Tada je moguće pozvati servis za proveru korisnika (*authentication service*) i proslediti upit da li je, ili nije, korisnik uneo ispravne podatke.

Sa ciljem provere da li komponenta može da bude aktivirana dodaje se klasa čuvar (guard class) pod ključem `canActivate` u konfiguraciji rutera.

Neophodno je vratiti se na početnu aplikaciju, dodati polja za unos korisničkog imena i lozinke korisnika i novu zaštićenu rutu koja funkcioniše samo ako se obezbede ispravni podaci za korisničko ime i lozinku.

SERVIS ZA PROVERU KORISNIKA AUTHSERVICE

Implementacija jednostavne provere korisnika u Angular aplikaciji.

Da bi izlaganje dalje teklo u željenom smeru, neophodno je obaviti kreiranje jednostavne i minimalne implementacije servisa, odgovornog za proveru i odobravanje pristupa resursima aplikacije. Navedeno će biti implementirano u klasi `AuthService` (datoteka primera: `/src/app/auth.service.ts`):

```
import { Injectable } from '@angular/core';

@Injectable()
export class AuthService {
  login(user: string, password: string): boolean {
    if (user === 'korisnik' && password === '1234') {
      localStorage.setItem('username', user);
    }
  }
}
```

```
        return true;
    }

    return false;
}
```

Kreirana funkcija `login()` će vratiti vrednost `true` ukoliko korisničko ime i lozinka odgovaraju stringovima "korisnik" i "1234", respektivno. Takođe, ukoliko je uspešno obavljeno podudaranje korisničkog imena i lozinke sa unetim podacima, biće omogućeno njihovo čuvanje na disku (linija koda 7). Ovo će takođe poslužiti kao indikator (`flag`) da li, ili ne, postoji aktivni prijavljeni korisnik.

Metoda za odjavljivanje korisnika (`logout()`) jednostavno uklanja sačuvanu vrednost za korisničko ime. To je moguće ostvariti dodavanjem sledećeg koda na postojeći:

```
logout(): any {
    localStorage.removeItem('username');
}
```

Konačno, moguće je dodati još dve korisne funkcije:

- `getUser()` - vraća korisničko ime (`username`) ili `null`;
- `isLoggedIn()` - koristi `getUser()` i vraća `true` ukoliko postoji korisnik prijavljen na aplikaciju.

```
getUser(): any {
    return localStorage.getItem('username');
}

isLoggedIn(): boolean {
    return this.getUser() !== null;
}
```

Poslednje što je potrebno dodati je konstanta `AUTH_PROVIDERS` da bi servis mogao da bude umetnut u komponentu za prijavljivanje korisnika. malo kasnije, servis će biti zadužen i za zaštitu komponente `ProtectedComponent`.

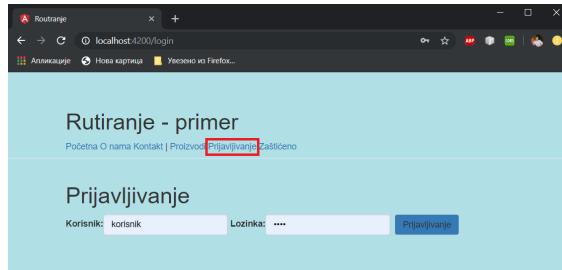
```
export const AUTH_PROVIDERS: Array<any> = [
    { provide: AuthService, useClass: AuthService }
];
```

KOMPONENTA ZA PRIJAVLJIVANJE KORISNIKA LOGINCOMPONENT

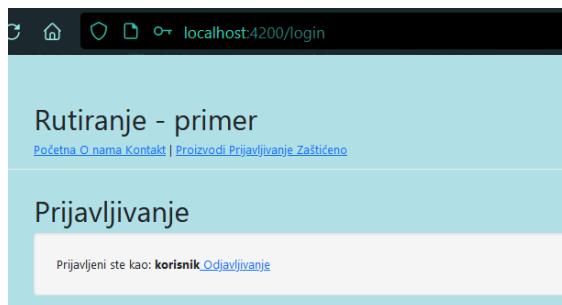
Zadatak komponente je da omogući prikazivanje forme za prijavljivanje i banera sa podacima i linkom za odjavljivanje.

Zadatak komponente `LoginComponent` je da omogući prikazivanje forme (slika 1), kada korisnik nije prijavljen, za učitavanje korisničkih podataka za prijavljivanje na aplikaciju ili

određeni njen deo ili baner sa informacijom o prijavljenom korisniku i linkom za odjavljivanje korisnika (slika 2).



Slika 8.1 Forma za prijavljivanje podataka [izvor: autor]



Slika 8.2 Baner sa informacijom o prijavljenom korisniku i linkom za odjavljivanje [izvor: autor]

Relevantan kod klase *LoginComponent* (datoteka: *src/app/login/login.component.ts*) nalazi se u metodama *login()* i *logout()*:

```
export class LoginComponent {
  message: string;

  constructor(public authService: AuthService) {
    this.message = '';
  }

  login(username: string, password: string): boolean {
    this.message = '';
    if (!this.authService.login(username, password)) {
      this.message = 'Vaši podaci nisu tačni ili ne postoje!!!.';
      const self = this;
      setTimeout(function() {
        self.message = '';
      }.bind(this), 2500);
    }
    return false;
  }

  logout(): boolean {
    this.authService.logout();
    return false;
  }
}
```

```
}
```

Kada servis uspešno proveri podatke korisnika, on može da se prijavi na aplikaciju.

ŠABLON KOMPONENTE LOGINCOMPONENT

Šablon komponente ima dve sekcije koje se prikazuju u zavisnosti da li je korisnik prijavljen, ili ne.

Šablon komponente (datoteka: *src/app/login/login.component.html*) ima dve sekcije koje se prikazuju u zavisnosti da li je korisnik prijavljen, ili ne:

- *forma za prijavljivanje* - prikazuje se kada korisnik nije prijavljen i zaštićena je direktivom: **ngIf="!authService.getUser()"*;
- *baner sa prijavljenim korisnikom i linkom za odjavljivanje* - prikazuje se kada je korisnik prijavljen i zaštićena je direktivom: **ngIf="authService.getUser()"*.

Forma za prijavljivanje:

```
<h1>Prijavljanje</h1>

<div class="alert alert-danger" role="alert" *ngIf="message">
  {{ message }}
</div>

<form class="form-inline" *ngIf="!userService.getUser()">
  <div class="form-group">
    <label for="username">Korisnik:</label>
    <input class="form-control" name="username" #username>
  </div>

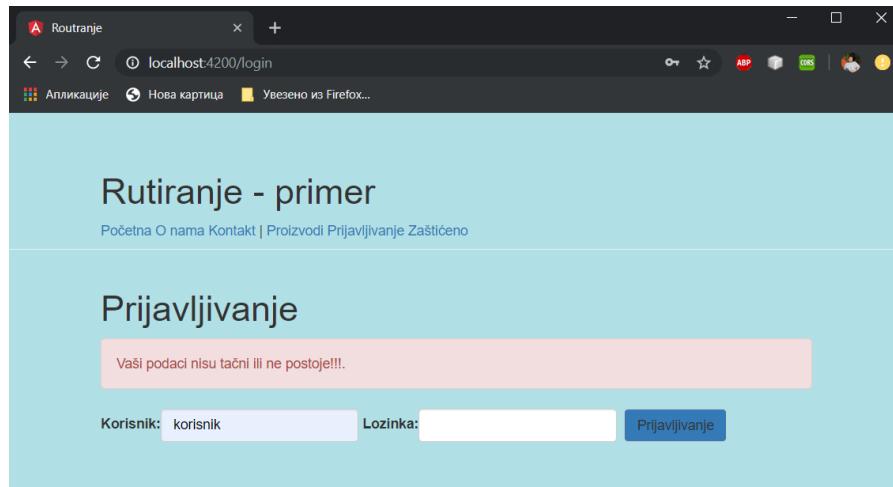
  <div class="form-group">
    <label for="password">&nbsp;Lozinka:</label>
    <input class="form-control" type="password" name="password" #password>
    &nbsp;
  </div>

  <a class="btn btn-default" (click)="login(username.value, password.value)">
    Prijavljanje
  </a>
</form>
```

Baner sa podatkom o prijavljenom korisniku i linkom za odjavljivanje:

```
<div class="well" *ngIf="userService.getUser()">
  Prijavljeni ste kao: <b>{{ userService.getUser() }}</b>
  <a href (click)="logout()"> Odjavljivanje</a>
</div>
```

Ako se obrati pažnja na listing klase `LoginComponent` iz prethodne sekcije (linije koda 10 - 15) i listing šablonu za formu za prijavljivanje (linije koda 3 - 5) moguće je primetiti obezbeđen prostor za prikazivanje greške u slučaju neuspešnog unosa korisničkih podataka (slika 3). Greška se prikazuje u formi poruke na stranici i traje 2,5 sekunde (2500 ms - linija koda 14 u listingu klase).



Slika 8.3 Prikazivanje greške - neuspešno prijavljivanje korisnika [izvor: autor]

KLASA ČUVAR RUTE

Klasa čuvare mora da implementira CanActivate interfejs.

Kreira se nova datoteka `logged-in.guard.ts` na korenskoj lokaciji `/src/app/logged-in.guard.ts` koja čuva klasu `LoggedInGuard` definisanu sledećim listingom:

```
import { Injectable } from '@angular/core';
import {
  CanActivate,
  ActivatedRouteSnapshot,
  RouterStateSnapshot
} from '@angular/router';
import { Observable } from 'rxjs/Observable';
import { AuthService } from './auth.service';

@Injectable()
export class LoggedInGuard implements CanActivate {
  constructor(private authService: AuthService) {}

  canActivate(
    next: ActivatedRouteSnapshot,
    state: RouterStateSnapshot): Observable<boolean> | Promise<boolean> | boolean {
    const isLoggedIn = this.authService.isLoggedIn();
    console.log('canActivate', isLoggedIn);
    return isLoggedIn;
  }
}
```

Klasa `LoggedInGuard` implementira `CanActivate` interfejs i kao takva naziva se klasom čuvarem (*Guard Class*). Implementacija navedenog interfejsa realizuje se konkretizacijom definicije metode `canActivate()` (linija koda 14).

Putem konstruktora klase čuvara vrši se umetanje servisa za proveru korisnika `AuthService` koji se čuva kao privatna promenljiva `authService`.

U `canActivate()` funkciji proverava se `this.authService.isLoggedIn()` sa ciljem provere da li je korisnik prijavljen na aplikaciju ili nije..

PODEŠAVANJE RUTERA

Da bi u aplikaciji bilo moguće koristiti klasu čuvara, neophodno je dodatno podesiti ruter.

Da bi u aplikaciji bilo moguće koristiti klasu čuvara, neophodno je obaviti podešavanje rutera na sledeći način:

1. izvršiti importovanje klase čuvara (u ovom slučaju `LoggedInGuard`);
2. iskoristiti klasu `LoggedInGuard` u konfiguraciji ruta;
3. uključiti klasu `LoggedInGuard` u listu provajdera (da bi mogla da bude umetnuta).

Navedeni koraci obavljaju se proširivanjem listinga centralnog modula aplikacije u datoteci `app.module.ts`.

Importovanje klase `LoggedInGuard` obavljeno je na sledeći način:

```
import { AUTH_PROVIDERS } from './auth.service';
import { LoggedInGuard } from './logged-in.guard';
```

Upotreba klase `LoggedInGuard` u konfiguraciji ruta određena je dodavanjem `canActivate` ključa sa vrednošću koja odgovara klasu `LoggedInGuard` za komponentu `ProtectedComponent` (sledeći listing, linije koda 12 - 14).

```
const routes: Routes = [
  // osnovne rute
  { path: '', redirectTo: 'home', pathMatch: 'full' },
  { path: 'home', component: HomeComponent },
  { path: 'about', component: AboutComponent },
  { path: 'contact', component: ContactComponent },
  { path: 'contactus', redirectTo: 'contact' },

  // provera korisnika
  { path: 'login', component: LoginComponent },
  {
    path: 'protected',
    component: ProtectedComponent,
    canActivate: [ LoggedInGuard ]
  },
]
```

```
// ugnježdeno
{
  path: 'products',
  component: ProductsComponent,
  children: childRoutes
}
];
```

Konačno, uključivanje klase *LoggedInGuard* u listu provajdera je obavljeno na sledeći način:

```
providers: [
  // sklonite ovo za "hash-bang" rutiranje
  // { provide: LocationStrategy, useClass: HashLocationStrategy }
  AUTH_PROVIDERS,
  LoggedInGuard
],
```

DOPUNA I DEMO ZA POVEZIVANJE KORISNIKA

U datoteci `app.module.ts` neophodno je obaviti importovanje komponente `LoginComponent`.

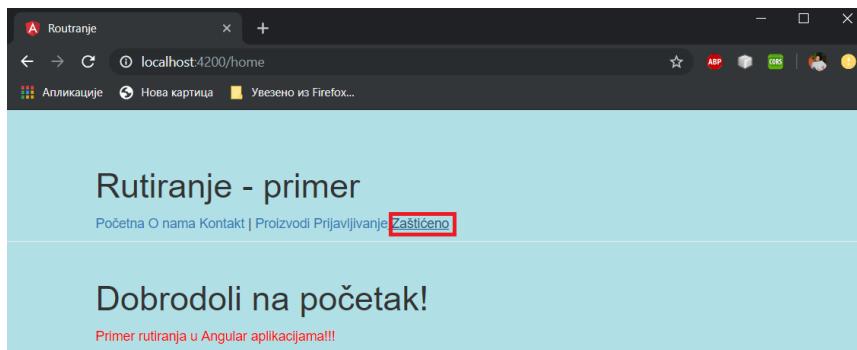
U datoteci `/src/app/app.module.ts` neophodno je, konačno, obaviti importovanje komponente `LoginComponent`:

```
import { LoginComponent } from './login/login.component';
import { ProtectedComponent } from './protected/protected.component';
```

Da bi bilo moguće pristupiti funkcionalnostima komponente u aplikaciji, neophodno je:

- imati rutu povezану са `LoginComponent`;
- imati nov link за заштићену rutu.

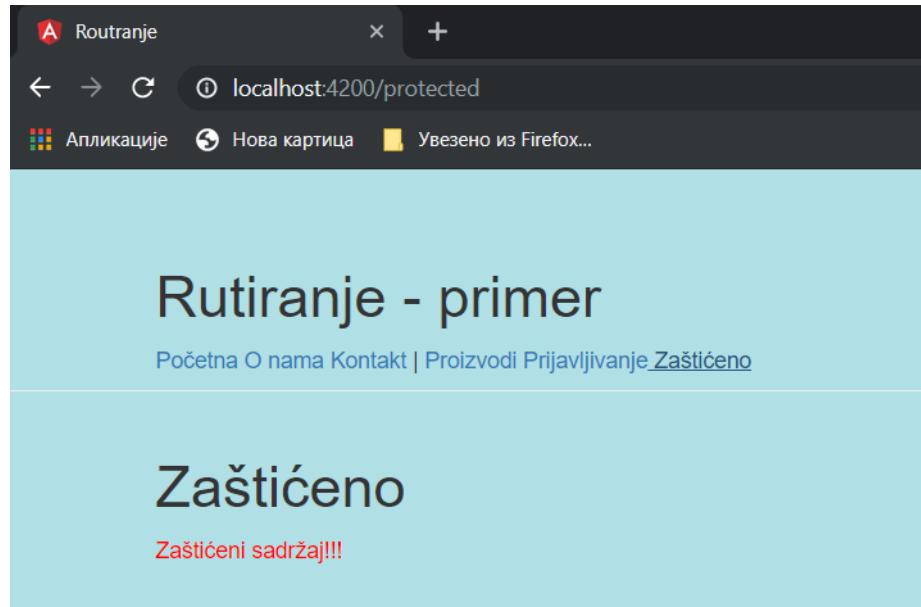
Sada je neophodno ponovo prevesti aplikaciju u pokrenuti je na portu 4200. Prijavljivanje korisnika je već demonstrirano i ovde neće biti ponovo prikazano. Sada je neophodno pokušati pokretanje zaštićene rute klikom na odgovarajući link (sledeća slika).



Slika 8.4 Poziv zaštićene rute pre prijavljivanja korisnika [izvor: autor]

Ono što je moguće primetiti jeste da aplikacija ne reaguje. Razlog je dobro poznat - nije moguće pristupiti zaštićenoj komponenti ukoliko korisnik nije prijavljen na aplikaciju.

Na dobro poznat način, korisnik se prijavljuje na aplikaciju, i ponavlja se pokušaj pokretanja zaštićene rute klikom na link "[Zaštićeno](#)". Rezultat se vidi na sledećoj slici.



Slika 8.5 Aktiviranje zaštićene rute [izvor: autor]

ZAŠTIĆENA KOMPONENTA

Neophodno je da se kreira komponenta koja je kandidat za zaštićenu ulogu.

Pre nego što se obavi obezbeđivanje komponente, neophodno je da se kreira komponenta koja je kandidat za takvu ulogu. U tekućem projektu će biti kreirana nova komponenta pod nazivom `ProtectedComponent` čija je klasa smeštena u datoteku `src/app/protected/protected.component.ts` i čiji je kod priložen sledećim listingom:

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

@Component({
  selector: 'app-protected',
  templateUrl: './protected.component.html',
  styleUrls: ['./protected.component.css']
})
export class ProtectedComponent implements OnInit {

  constructor() { }

  ngOnInit() {
  }
}
```

U listingu klase nema ništa novo i moguće je odmah preći na kreiranje šablona ove komponente (datoteka: [src/app/protected/protected.component.html](#)):

```
<h1>Zaštićeno</h1>
<p>
  <font color="red">
    Zaštićeni sadržaj!!!
  </font>
</p>
```

ANGULAR RUTIRANJE KROZ VIDEO MATERIJAL

Izlaganje je neophodno zaokružiti pogodnim video materijalima.

Routing and Navigation in Angular | Mosh - Trajanje: 24:31

Ova lekcija sadrži video materijal. Ukoliko želite da pogledate ovaj video morate da otvorite LAMS lekciju.

▼ Poglavlje 9

Ugnježdene rute

SCENARIO UGNJEŽDENIH RUTA

Ugnježdena ruta (nested route) je koncept kojim je definisana mogućnost da neka ruta sadrži drugu.

Angular kao radni okvir podržava primenu koncepta "ugnježdenih ruta". Ugnježdena ruta (nested route) je koncept kojim je definisana mogućnost da neka ruta sadrži drugu. Ugnježdavanjem ruta data je nova mogućnost enkapsuliranja funkcionalnosti roditeljskih ruta i njihove primene u rutama potomcima.

Na primer, postoji veb sajt koji sadrži oblast koja dozvoljava korisnicima da sagledaju detaljnije informacije u vezi sa dostupnim proizvodima. Kako aplikacija bude rasla biće potrebno obezbediti dodatne mehanizme za prikazivanje pojedinačnih informacija u vezi svakog dostupnog proizvoda. Za podršku ovakvom scenariju, Angular ruter dozvoljava definisanje ugnježdenih ruta. Rešenje, za ovaj problem, je moguće naći u postojanju većeg broja ugnježdenih router-outlet elemenata. To znači da svaka komponenta aplikacije može da poseduje potomak komponentu koje, takođe, poseduju vlastite router-outlet elemente.

Da bi stvari bile jasnije, biće vraćen akcenat na aktuelni primer koji ide u dodatnu doradu. Postojaće sekcija za proizvode, koja će služiti za demonstraciju primene ugnježdenih ruta.

PODEŠAVANJE RUTA

Definicija rute za proizvode počinje ponovnim vraćanjem na centralni modul aplikacije.

Opis i definicija rute za proizvode počine počinje ponovnim vraćanjem na centralni modul aplikacije app.module.ts:

```
const routes: Routes = [
  // osnovne rute
  { path: '', redirectTo: 'home', pathMatch: 'full' },
  { path: 'home', component: HomeComponent },
  { path: 'about', component: AboutComponent },
  { path: 'contact', component: ContactComponent },
  { path: 'contactus', redirectTo: 'contact' },

  // provera korisnika
  { path: 'login', component: LoginComponent },
```

```
{  
  path: 'protected',  
  component: ProtectedComponent,  
  canActivate: [ LoggedInGuard ]  
},  
  
// ugnježdeno  
{  
  path: 'products',  

```

Ugnježdavanje ruta je definisano linijama koda 19 - 22.

Neophodno je primetiti da ruta "*products*" poseduje parametar *children*. Odakle se on javlja? Vrednost ovog parametra *pochildRoutes* je određena u novom modulu: *ProductsModule*. Upravo će sledeće izlaganje da de bavi novim modulom.

MODUL PRODUCTSMODULE

Modul ProductsModule poseduje vlastitu konfiguraciju putanja.

Modul *ProductsModule* poseduje vlastitu konfiguraciju putanja. To je prikazano sledećim listingom, preuzetim iz datoteke: *src/app/products/products.module.ts*.

```
export const routes: Routes = [  
  { path: '', redirectTo: 'main', pathMatch: 'full' },  
  { path: 'main', component: MainComponent },  
  { path: 'more-info', component: MoreInfoComponent },  
  { path: ':id', component: ProductComponent }  
];
```

Iz priloženog listinga je moguće primetiti postojanje prazne putanje za prvi objekat. Ovo je urađeno iz razloga da bi se obezbedilo da sa posećivanjem putanje */products*, vrši se preusmeravanje na glavnu rutu, označenu sa *main*.

Sledeće ruta, kojoj bi trebalo posvetiti više pažnje, je *:id*. U ovom slučaju, ukoliko korisnik poseti link koji ne odgovara nijednoj postojećoj ruti, biće usmeren na ovu rutu. Sve što je prosleđeno nakon znaka "/" biće dodeljeno parametru rute, označenim sa *id*.

ŠABLON KOMPONENTE

Neophodno je obezrediti linkove za svaku statičku potomak rutu, pojedinačno.

Kreiranjem komponente za proizvode, neophodno je obezbediti linkove za svaku statičku potomak rutu, pojedinačno. Navedeno je moguće opisati sledećim listingom (datoteka: `src/app/products/products.component.html`):

```
<h2>Proizvodi</h2>

<div class="navLinks">
  <a [routerLink]="'./main'">Glavna</a> |
  <a [routerLink]="'./more-info'"> Više informacija</a> |
```

Iz listinga se primećuje da su svi linkovi ruta dati u formatu `'./main'`, sa navođenjem `./` ispred naziva rute. Ovo ukazuje da se vrši navigacija glavnom rutom u odnosu na trenutni kontekst rute.

Ruta je mogla da bude deklarisana primenom notacije `['products', 'main']`. Loša strana je ta što, radeći na ovaj način, ruta potomak ima fiksne informacije roditeljske rute i ako bi ova komponenta bila premeštena i ponovo upotrebljena, bilo bi neophodno ponovo napisati sve linkove rute.

Nakon dodavanja linkova, neophodno je obezbediti ulaz koji će omogućiti korisniku unos `id` proizvoda, zajedno sa dugmetom za navigaciju.

Definicija šablona se završava dodavanjem elementa `router-outlet`.

```
Unesite id: <input #id size="6">
  <button (click)="goToProduct(id.value)">Traži</button>
</div>

<div class="products-area">
  <router-outlet></router-outlet>
</div>
```

DEFINICIJA KOMPONENTE PRODUCTSCOMPONENT

Definicija komponente `ProductsComponent` je realizovana kreiranjem klase komponente.

Definicija komponente `ProductsComponent` je realizovana kreiranjem klase komponente (datoteka: `src/app/products/products.component.ts`):

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

@Component({
  selector: 'app-products',
  templateUrl: './products.component.html',
  styleUrls: ['./products.component.css']
```

```
})
export class ProductsComponent {
  constructor(private router: Router, private route: ActivatedRoute) {}

  goToProduct(id: string): void {
    this.router.navigate(['./', id], {relativeTo: this.route});
  }
}
```

Na prvom mestu je u konstruktoru deklasirana objektna promenljiva tipa Router, koja će biti upotrebljena za navigaciju ka konkretnom proizvodu preko id.

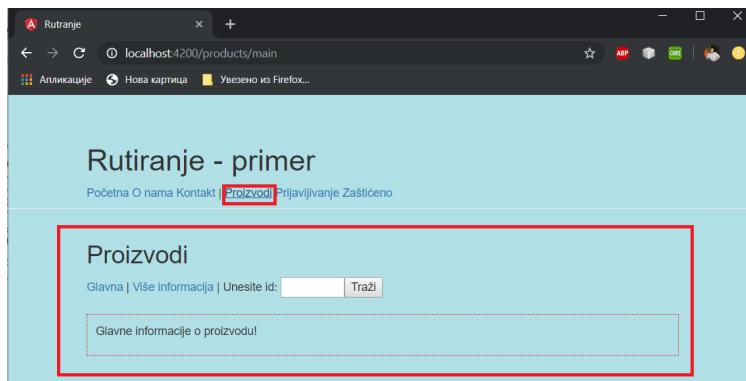
Kada je neophodno pristupiti konkretnom proizvodu, biće upotrebljena metoda `goToProduct()`. U ovoj metodi se poziva metoda `navigate()` rutera i obezbeđuje se naziv rute i objekat sa parametrima rute. U konkretnom slučaju, jednostavno se prosleđuje id.

Neophodno je primetiti da se koristi relativna `./` putanja u navigacionoj funkciji. Da bi ovo moglo da bude upotrebljeno koristi se `relativeTo` objekat koji ukazuje ruteru na šta se ruta konkretno odnosi..

DEMONSTRACIJA UGNJEŽDENIH RUTA

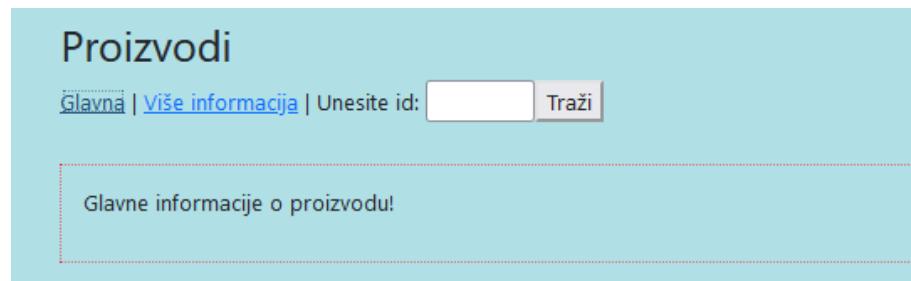
Prevodi se aplikacija, ponovo pokreće i testira se izvršavanje ugnježdenih ruta.

Prevodi se aplikacija, ponovo pokreće i testira se izvršavanje ugnježdenih ruta. Izborom opcije "Proizvodi" pokreće se glavna ruta i dobija se rezultat prikazan sledećom slikom:



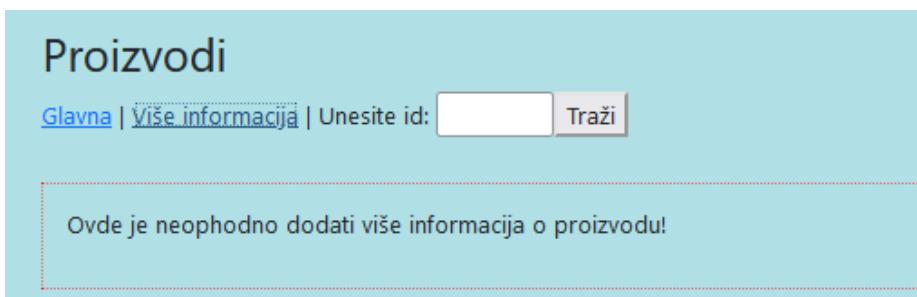
Slika 9.1 Main ruta za proizvode [izvor: autor]

Sada je moguće koristiti ugnježdene rute. Sledećom slikom je istaknut detalj sa prethodne slike koji pokazuje detalje rute `Main`.



Slika 9.2 Detalji rute Main [izvor: autor]

Posetom linku [products/more-info](#) ili klikom na opciju "Više informacija" pokreće se ugnježdena ruta koja obezbeđuje dodatne informacije o proizvodima.



Slika 9.3 Više informacija o pojedinačnim proizvodima [izvor: autor]

Konačno, ugnježdena ruta koja rukuje id vrednostima proizvoda omogućava prikazivanje informacija o pojedinačnim proizvodima.



Slika 9.4 Prikazivanje informacija o pojedinačnim proizvodima [izvor: autor]

UGNJEŽDENE RUTE KROZ VIDEO MATERIJAL

Angular 8 Tutorial - 29 - Child Routes - Trajanje 6:57.

Ova lekcija sadrži video materijal. Ukoliko želite da pogledate ovaj video morate da otvorite LAMS lekciju.

Potpuno urađen pokazni primer ove lekcije možete preuzeti odmah nakon ovog objekta učenja.

✓ Poglavlje 10

Dodatni nastavni materijali

DODATNI MATERIJALI

Proširite znanje sa predavanja

1. <https://angular.io/>
2. <https://angular.io/tutorial>
3. <https://www.w3schools.com/angular/>
4. <https://www.tutorialspoint.com/angular4/>
5. <https://nodejs.org/en/>
6. <https://code.visualstudio.com/>
7. <http://expressjs.com/en/guide/routing.html>
8. <https://rubyonrails.org/>
9. <https://angular.io/guide/router#!%23browser-url-styles>

✓ Poglavlje 11

Pokazna vežba 10 (Trajanje 45 minuta)

UVOĐENJE RUTA

Vežbanje na času rada sa rutama u Angular aplikacijama.

Definicija rutiranja unutar naše aplikacije bitna je prvenstveno zbog razdvajanja delova aplikacije, održavanja stanja aplikacije kao i zaštite određenih delova na osnovu već definisanog seta pravila.

Rutiranje je u toku godina evoluiralo, tako da danas možemo promeniti rutu web aplikacije, bez osvežavanja veb pregledača. Sve ovo nam je omogućilo HTML5 klijentsko rutiranje.

Tri glavne komponente će definisati način na koji konfigurišemo ruter na nivou aplikacije su:

- *Routes* – Predstavlja listu svih ruta naše aplikacije.
- *RouterOutlet* – predstavlja mesto gde će Angular prikazivati sadržaj svake rute na promenu.
- *RouterLink* – Predstavlja direktivu koju koristimo da pokazuje na nove rute.

Ideja je da naš dosadašnji pokazni primer dovedemo u formu da koristi rute za prikazivanje sadržaja pojedinačnih komponenata.

Sledećom slikom je prikazan početni izgled primera:



Slika 11.1 Početni izgled primera [izvor: autor]

Ono što je moguće primetiti jeste da je data lista sa linkovima koja predstavlja izvor rutiranja na ovoj stranici. Početna ruta odgovara komponenti *StudentComponent* injen sadržaj je, upravo, prikazan kao tabela unutar taga *<router-outlet>*.

PODEŠAVANJE LISTE RUTA

Slede konfiguracioni zadaci vezani za podešavanje ruta unutar aktuelnog projekta.

Slede konfiguracioni zadaci vezani za podešavanje ruta unutar aktuelnog projekta. U našem slučaju, imaćemo sledećih 5 ruta:

- `/student`
- `/buttongroup`
- `/forma`
- `/di`
- `/http`

Otvorićemo fajl pod nazivom `app.module`, u kom ćemo izvršiti definiciju ruta na sledeći način:

```
const routes: Routes = [
  // osnovne rute
  { path: '', redirectTo: 'student', pathMatch: 'full' },
  { path: 'student', component: StudentComponent },
  { path: 'buttongroup', component: ButtongroupcomponentComponent },
  { path: 'forma', component: FormaComponent },
  { path: 'di', component: DIComponent },
  { path: 'http', component: HttpComponent },
];
```

U nastavku, neophodno je obaviti dodavanje zavisnosti i instaliranje kreiranog rutera. U posmatranoj datoteci vrši se dodavanje sledeće import instrukcije:

```
import {
  RouterModule,
  Routes
} from '@angular/router';
```

Konačno, obavlja se instalacija rutera:

```
imports: [
  BrowserModule, ReactiveFormsModule, HttpClientModule,
  RouterModule.forRoot(routes)
]
```

Konačna definicija `app.module` datoteke data je sledećim listingom

```
import { NgModule } from '@angular/core';
import { BrowserModule } from '@angular/platform-browser';
import { ReactiveFormsModule } from '@angular/forms';
import { HttpClientModule } from '@angular/common/http';

import { AppComponent } from './app.component';
```

```
import { StudentComponent } from './studentcomponent/studentcomponent.component';
import { NavbarcomponentnewComponent } from './navbarcomponentnew/
navbarcomponentnew.component';
import { ButtongroupcomponentComponent } from './buttongroupcomponent/
buttongroupcomponent.component';
import { FormaComponent } from './forma/forma.component';
import { Studije } from './model/studije';
import { MetService } from './services/metService';
import { DIComponent } from './di/di.component';
import { HttpComponent } from './http/http.component';

import {
  RouterModule,
  Routes
} from '@angular/router';

const routes: Routes = [
  // osnovne rute
  { path: '', redirectTo: 'student', pathMatch: 'full' },
  { path: 'student', component: StudentComponent },
  { path: 'buttongroup', component: ButtongroupcomponentComponent },
  { path: 'forma', component: FormaComponent },
  { path: 'di', component: DIComponent },
  { path: 'http', component: HttpComponent },
];

@NgModule({
  declarations: [
    AppComponent,
    StudentComponent,
    NavbarcomponentnewComponent,
    ButtongroupcomponentComponent,
    FormaComponent,
    DIComponent,
    HttpComponent
  ],
  imports: [
    BrowserModule, ReactiveFormsModule, HttpClientModule,
    RouterModule.forRoot(routes)
  ],
  providers: [Studije, MetService],
  bootstrap: [AppComponent]
})
export class AppModule { }
```

IZMENA LOGIKE GLAVNE KOMPONENTE KLASE

Podešava se glavna komponenta da primeni rutiranje.

Definisan je ruter, a nakon toga je i instaliran - kao što je prikazano u prethodnom izlaganju.

Neophodno je omogućiti da grafički korisnički interfejs, a i njegova upravljačka klasa, primenjuju podešeno rutiranje. Budući da smo gradili primer na način da je šablon ugrađen direktno u klasu, pod ključem *template*, sledećim listingom pokazujemo implementaciju rutiranja u šablonu komponente kao niz *<a>* tagova (linije koda 12 - 20)

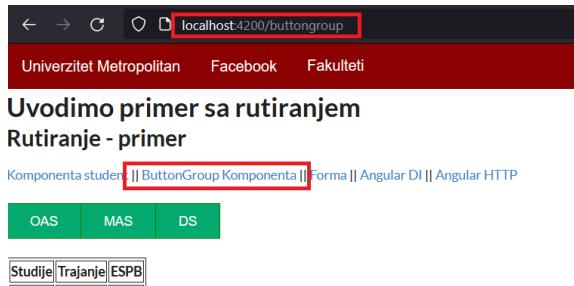
Listing klase i šablonu:

```
@Component({  
  
    selector: 'my-app',  
    template: `  
        <app-navbarcomponentnew></app-navbarcomponentnew>  
        <div>  
            <h1>{{pageheader}}</h1>  
        </div>  
        <div class="page-header">  
            <div class="container">  
                <h2>Rutiranje - primer</h2>  
                <div class="navLinks">  
                    <a [routerLink]=["['/student']">Komponenta student </a>  
                    ||  
                    <a [routerLink]=["['/buttongroup']">ButtonGroup  
Komponenta </a>  
                    ||  
                    <a [routerLink]=["['/forma']">Forma </a>  
                    ||  
                    <a [routerLink]=["['/di']">Angular DI </a>  
                    ||  
                    <a [routerLink]=["['/http']"> Angular HTTP</a>  
                </div>  
            </div>  
        </div>  
        <div id="content">  
            <div class="container">  
                <router-outlet></router-outlet>  
            </div>  
        </div>  
        `  
    })  
  
    export class AppComponent {  
        pageheader: string = "Uvodimo primer sa rutiranjem"  
    }`
```

DEMONSTRACIJA

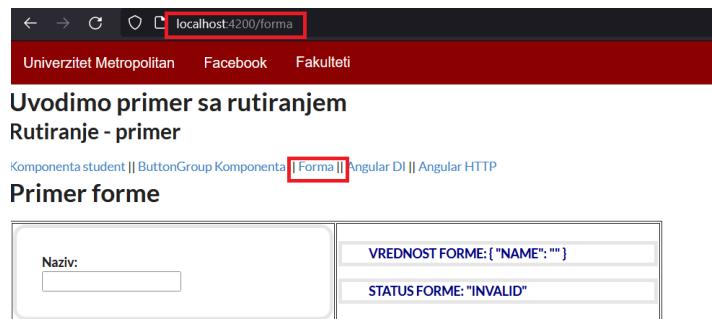
Demonstracija primene kreiranog ruteru u aktuelnom primeru

Prethodno prikazana slika 1 ukazuje na početnu rutu ili scenario naknadnog izbora linka *Komponenta student*. Izaberimo sada sledeći link, *ButtonGroup* komponenta i izgled aplikacije će biti promenjen, baš kao na sledećoj slici:



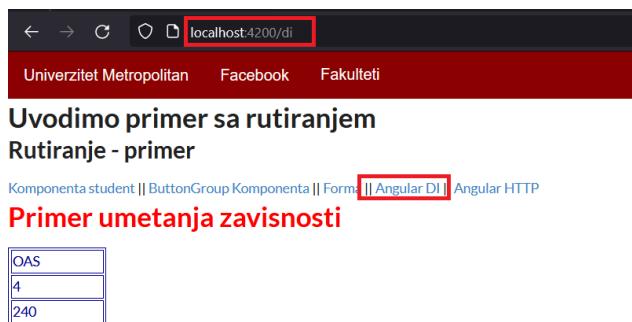
Slika 11.2 ButtonGroup komponenta kao izabrana ruta [izvor: autor]

Idemo dalje u navigaciju i biramo sledeći link koji nas navodi na rutu *Forma*. Rezultat je prikazan sledećom slikom:



Slika 11.3 FormaComponent kao izabrana ruta [izvor: autor]

Primer koji smo radili u lekciji "*Umetanje zavisnosti*" ugrađen je u aplikaciju na način da mu se pristupa rutom */di*. Navedeno je ilustrovano sledećom slikom:



Slika 11.4 DIComponent kao izabrana ruta [izvor: autor]

Konačno, poslednja ruta određuje primer iz prethodne lekcije:



Slika 11.5 HttpComponent kao izabrana ruta [izvor: autor]

PREUZMITE URAĐENI PRIMER IZ DODATNIH MATERIJALA NA KRAJU OVOG OU.

✓ Poglavlje 12

Individualna vežba 10

INDIVIDUALNA VEŽBA (TRAJANJE 90 MINUTA)

Samostalno vežbanje rutiranja nakon obrađenih materijala predavanja i pokazne vežbe.

Nastavlja se rad na primeru sa pokaznih vežbi.

1. Kreirajte novu komponentu koja prikazuje listu sa vašim položenim ispitima u tabelarnoj formi;
2. Dodajte stilove za šablon komponente;
3. Kreirajte rutu za ovu komponentu - link u roditeljskoj komponenti;
4. Dodajte je u ruter i instalirajte;
5. Pokrenite primer i pratite valjanost njegovog izvršavanja.
6. U slučaju problema obratite se asistentu ili profesoru.

ZADATAK ZA 2 POENA ZA ZALAGANJE - Implementirajte ugnježdenu rutu: Bira se godina studija, a onda i položeni ispiti iz te godine.

✓ Poglavlje 13

Domaći zadatak 10

DOMAĆI ZADATAK (PREDVIĐENO VREME 120 MIN)

Samostalna izrada domaćeg zadatka sa implementacijom Angular rutiranja.

Uradite domaći zadatka prema sledećim zahtevima:

1. Nastavite rad na projektu MetHotels,
2. Izaberite najmanje tri funkcionalnosti na kojima ste radili (različite komponente)
3. Kreirajte i instalirajte ruter koji sadrži rute za navigaciju ka sadržaju izabranih komponenata;
4. Definišite konačni izgled aplikacije prema novim zahtevima - predvideti router-outlet za prikazivanje sadržaja određenog rutama.

Domaći zadatak dodati na [Github](#) pod "commit - om" **IT255-DZ11** i poslati obaveštenje predmetnom asistentu o postavljenom domaćem zadatku.

▼ Poglavlje 14

Zaključak

ZAKLJUČAK

Lekcija se bavila izučavanjem koncepta rutiranja, kao veoma važnog problema veb razvoja.

Tokom izučavanja problematike ovog predmeta uključuju se napredniji [Angular](#) koncepti i principi. Tako da je aktuelna lekcija poseban akcenat stavila na koncept rutiranja ([routing](#)), koji se smatra veoma važnim problemom savremenog veb razvoja.

Posebno je bilo važno istaći šta, zapravo, rutiranje predstavlja? U savremenom veb razvoju, pod pojmom rutiranja se podrazumeva proces razdvajanja veb aplikacije na različite oblasti, uglavnom, na osnovu pravila izvedenih iz konkretnih [URL](#) navedenih u veb pregledaču.

Šta navedena definicija rutiranja znači u praksi? Na primer, ukoliko korisnik poseti putanju ["/"](#) veb sajta, on će verovatno posetiti početnu ([home](#)) rutu navedenog veb sajta. Sa druge strane, ukoliko korisnik poseti putanju ["/about"](#) veb sajta, zahtevaće kreiranje i prikazivanje veb stranice, na primer pod nazivom ["about"](#), i tako dalje.

Budući na značaj, složenost i učestalost primene same problematike, u izlaganje, koje prati lekciju, uključen je odgovarajući i pažljivo biran i kreiran pokazni primer sa ciljem lakšeg razumevanja i savladavanja lekcije.

Savladavanjem lekcije, studenti su, sada, osposobljeni ka koriste koncept rutiranja prilikom izrade vlastitih veb aplikacija u [Angular](#) radnom okviru.

LITERATURA

Za pripremu lekcije korišćena je aktuelna pisana i elektronska literatura.

Pisana literatura:

1. Nate Murray, Felipe Coury, Ari Lerner, Carlos Taborda, ng-Book – The Complete Book on Angular 6, Fullstack.io, 2018
2. Cody Lindley, Frontend Handbook – 2017, Frontend masters, 2017
3. Sandeep Panda, AngularJS – From Novice to Ninja, SitePoint Pty. Ltd, 2014

Elektronska literatura:

4. <https://angular.io/>
5. <https://angular.io/tutorial>

6. <https://www.w3schools.com/angular/>
7. <https://www.tutorialspoint.com/angular4/>
8. <https://nodejs.org/en/>
9. <https://code.visualstudio.com/>

10. <http://expressjs.com/en/guide/routing.html>
11. <https://rubyonrails.org/>

12. <https://angular.io/guide/router#!%23browser-url-styles>



IT255 - VEB SISTEMI 1

Angular i arhitektura podataka – Servisi i pogledi

Lekcija 12

PRIRUČNIK ZA STUDENTE

IT255 - VEB SISTEMI 1

Lekcija 12

ANGULAR I ARHITEKTURA PODATAKA – SERVISI I POGLEDI

- ✓ Angular i arhitektura podataka – Servisi i pogledi
- ✓ Poglavlje 1: Arhitektura podataka sa Observables - servisi
- ✓ Poglavlje 2: Obesrvables – aplikacija za čakanje
- ✓ Poglavlje 3: Implementiranje modela
- ✓ Poglavlje 4: Implementiranje servisa
- ✓ Poglavlje 5: Arhitektura podataka sa Observables - pogledi
- ✓ Poglavlje 6: Dodatni materijali za rad
- ✓ Poglavlje 7: Pokazna Vežba 11
- ✓ Poglavlje 8: Individualna vežba 12
- ✓ Poglavlje 9: Domaći zadatak 12
- ✓ Zaključak

Copyright © 2017 – UNIVERZITET METROPOLITAN, Beograd. Sva prava zadržana. Bez prethodne pismene dozvole od strane Univerziteta METROPOLITAN zabranjena je reprodukcija, transfer, distribucija ili memorisanje nekog dela ili čitavih sadržaja ovog dokumenta., kopiranjem, snimanjem, elektronskim putem, skeniranjem ili na bilo koji drugi način.

Copyright © 2017 BELGRADE METROPOLITAN UNIVERSITY. All rights reserved. No part of this publication may be reproduced, stored in a retrieval system or transmitted in any form or by any means, electronic, mechanical, photocopying, recording, scanning or otherwise, without the prior written permission of Belgrade Metropolitan University.

✓ Uvod

UVOD

Upravljanje podacima može biti jedan od najzahtevnijih aspekata pisanja održivih aplikacija.

Upravljanje podacima može biti jedan od najzahtevnijih aspekata pisanja održivih aplikacija. Postoje brojni načini pribavljanja podataka u veb aplikacijama, poput:

- *AJAX HTTP zahtevi*
- *WebSocket;*
- *Indexdb;*
- *LocalStorage*
- *Service Workers*, i tako dalje.

Problem koji se odnosi na *arhitekturu podataka* moguće je razdvojiti u više pitanja:

- Kako je moguće povezati sve navedene različite izvore u koherentan sistem?
- Kako je moguće izbeći greške izazvane neočekivanim efektima sa strane?
- Kako je moguće strukturirati kod na način da je lakši za održavanje i uključivanje novih članova softverskog razvojnog tima;
- Kako je moguće obezbediti da se aplikacija izvršava na najbrži mogući način kada dođe do izmene u podacima.

Ova lekcija sadrži video materijal. Ukoliko želite da pogledate ovaj video morate da otvorite LAMS lekciju.

ARHITEKTURA PODATAKA U ANGULAR-U

Angular pokazuje visok stepen fleksibilnosti arhitekture podataka i ne favorizuje nijedan pristup.

Brojne godine unazad *MVC* je bio dominantan šablon organizovanja podataka u aplikacijama:

- *Model* - čuva izvesnu domensku logiku;
- *View* - prikazuje podatke;
- *Controller* - povezuje prve dve arhitekturne komponente.

Međutim, primećuje se da, iz perspektive klijent usmerenih radnih okvira, a *Angular* je jedan od njih, *MVC* arhitektura se ne prevodi dovoljno dobro u veb aplikacije klijentske strane.

Da bi navedeni nedostatak bio prevaziđen, u skorije vreme je nastala prava renesansa u ovoj oblasti:

- *MVW (Model-View-Whatever)* - Dvosmerno povezivanje i *Angular* prva podrazumevana arhitektura. \$scope omogućava dvosmerno povezivanje podataka - cela aplikacija deli iste strukture podataka, a promena u jednoj oblasti se širi na ostatak aplikacije. Više i *MVW* šablonu moguće je pogledati na sledećem linku: <https://dzone.com/articles/angularjs-tutorial-lesson-1>.
- *Flux* (<https://facebook.github.io/flux/>) - koristi jednosmerni protok podataka. Koristi sledeće koncepte: *Stores* - čuva podatke, *View* - prikazuje šta se nalazi u *Stores* i *Actions* - menja podatke koji se čuvaju u *Stores*. Iako postoji malo složenija procedura za podešavanje Flux - a, jednosmerni tok podataka je jednostavniji za razumevanje;
- *Observables* - obezbeđuje tokove podataka. U aplikaciji se vrši prijavljivanje na tokove (*subscribe*), a zatim se izvode operacije kao reakcija na promene u podacima. *RxJS* (<https://github.com/Reactive-Extensions/RxJS>) je najpopularnija biblioteka reaktivnih tokova za *JavaScript* i koja daje moćne operatore za povezivanje operacija na tokovima podataka.

Angular okvir pokazuje veoma visok stepen fleksibilnosti arhitekture podataka i ne favorizuje nijedan pristup. Prednost navedenog je u tome što postoji fleksibilnost uklapanja *Angular* okvira u gotovo svaku situaciju. Loša strana je da programeri moraju sami da donose odluke o tome koji arhitekturni pristup je neophodno primeniti za konkretan projekat.

▼ Poglavlje 1

Arhitektura podataka sa Observables - servisi

VIDEO PREDAVANJE ZA OBJEKAT "ARHITEKTURA PODATAKA SA OBSERVABLES - SERVISI"

Trajanje video snimka: 13min 33sek

Ova lekcija sadrži video materijal. Ukoliko želite da pogledate ovaj video morate da otvorite LAMS lekciju.

OBSERVABLES I RXJS

Primena Observables označena je u savremenoj literaturi i praksi kao reaktivno programiranje.

U Angular okviru, moguće je strukturirati aplikaciju primenom koncepta osmatranja ili Observables kao oslonca strukture podataka. Primena ovog koncepta za organizovanje podataka u aplikaciji predstavlja nešto što je u savremenoj literaturi i praksi označeno kao reaktivno programiranje.

Šta, zapravo, u praksi predstavljaju Observables i reaktivno programiranje. **Reaktivno programiranje je način rada sa asinhronim tokovima podataka. Osmatranje ili Observables je osnova struktura podataka za implementaciju reaktivnog programiranja.** U brojnoj literaturi čete naći podudarne stavove autora koji govore da su navedena dva termina nespretno izabrani i da nisu baš najjasniji. Po dobroj praksi, da bi sve proteklo što jasnije i jednostavnije, biće uveden konkretni primer koji će pratiti izlaganje ove lekcije.

Još jednu bitnu stvar je imati na umu prilikom izučavanja gradiva ove lekcije. Ovde se studenti prvi put sreću sa konceptom reaktivnog programiranja i neophodno je dati osnovnu sliku u vezi sa primenom reaktivnog programiranja u Angular okviru. Nema prostora za iscrpljenje isključivo RxJS i reaktivnim programiranjem. Ovde će biti detaljno objašnjen RxJS koncept i API-ji neophodni za podršku reaktivnom programiranju

Za širu sliku i stepen znanja koji prevazilazi gradivo ove lekcije, studenti mogu da posete sledeće linkove:

- **The introduction to Reactive Programming you've been missing** - autor Andre Staltz (<https://gist.github.com/staltz/868e7e9bc2a7b8c1f754>);

- <https://underscorejs.org/>.
- **Which static operators to use to create streams?** (<https://github.com/Reactive-Extensions/RxJS/blob/master/doc/gettingstarted/which-static.md>);
- **Which instance operators to use on streams?** (<https://github.com/Reactive-Extensions/RxJS/blob/master/doc/gettingstarted/which-instance.md>);
- **RxMarbles** - interaktivni dijagrami za različite operacije nad tokovima (<https://rxmarbles.com/>).

Pojam toka podataka je osnovni koncept reaktivnog programiranja. Direktna implementacija toka podataka predstavlja Observable. Observable se može shvatiti kao entitet koji se posmatra tokom vremena i koji emituje vrednosti toka podataka. Sam Observable temelji se na dva obrasca oblikovanja: Observer obrazac i Iterator obrazac.

DODATNE NAPOMENE

Početak učenja RxJS često praćen sa brojnim nejasnoćama.

Brojni autori se slažu da je početak učenja *RxJS* često praćen sa brojnim nejasnoćama. Međutim, kada ga jednom savladaju programeri imaju moćan alat u rukama koji se vremenom veoma isplati. Sledi nekoliko glavnih ideja u vezi sa tokovima podataka, a koji mogu biti od pomoći u razumevanju daljeg izlaganja:

- 1. *Promises* (obećanja - još jedan nezgodno izabrani termin) emituju jedinstvenu vrednost, dok u slučaju *tokova* vrši se istovremeno emitovanje većeg broja vrednosti. Tokovi vrše identičnu ulogu kao i obećanja u aplikaciji. Istorijски gledano, koncept obećanja predstavlja iskorak unapred u odnosu na povratne pozive (*callback*) u smislu čitljivosti i održivosti podataka. Na isti način, tokovi predstavljaju unapređenje u odnosu na *Promises* šablone na način da je omogućeno kontinuirano odgovaranje na promene u podacima unutar toka (suprotno jednovremenom odgovaranju na promene u podacima u slučaju primene koncepta obećanja).
- 2. *Imperativni kod* povlači podatke sa bilo koje lokacije na kojoj je reaktivni tok ostavio podatke. U reaktivnom programiranju kod se prijavljuje ili pretplaćuje (*subscribe*) da bude obavešten o promenama u podacima, a tokovi "guraju" podatke ka pretplatnicima, odnosno odgovarajućim segmentima koda.
- 3. *RxJS je funkcionalan* - ukoliko je programer privržen primeni funkcionalnih operatora, poput mapa, reduktora ili filtera, naći će se na pravom mestu iz razloga što se kroz primenu *RxJS* tokovi mogu smatrati listama, a na taj način svi važni funkcionalni operatori mogu biti primjenjeni;
- 4. *Tokovi su povezivi* - tokovi mogu da se shvate kao nizovi ili cevi (*pipeline*) operacija nad podacima. Moguće je obaviti prijavljivanje za bilo koji tok, čak je moguće i njihovo kombinovanje sa ciljem kreiranja novih tokova.

Video materijal: **Angular Tutorial - What is an Observable ?**

Ova lekcija sadrži video materijal. Ukoliko želite da pogledate ovaj video morate da otvorite LAMS lekciju.

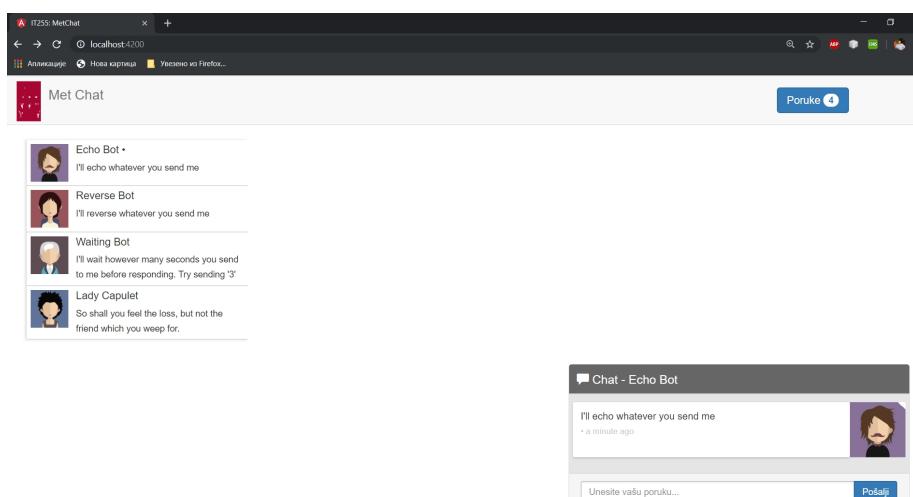
▼ Poglavlje 2

Obesrvables – aplikacija za časkanje

PREGLED APLIKACIJE

Opis postupaka i koncepata koji se koriste prilikom primene RxJS za kreiranje čet aplikacije.

U suštini, cilj lekcije jeste opis postupaka i koncepata koji se koriste prilikom primene **RxJS** za kreiranje aplikacije za časkanje (*chat app*). Sledećom slikom je prikazan idejni izgled ekrana aplikacije koja će biti kreirana kroz razmatranje gradiva ove lekcije.



Slika 2.1 Konačni izgled željene aplikacije [izvor: autor]

Kompletno urađenu aplikaciju studenti mogu da preuzmu na kraju lekcije, iz aktivnosti *Shared Resources*. Potrebno je da je raspakuju, na primer, u folderu `code/rxjs` i sledećim naredbama da je pripreme za izvršavanje:

```
cd code/rxjs/rxjs-chat  
npm install  
npm start
```

Sada je moguće otvoriti aplikaciju posećivanjem sledeće adrese u veb pregledaču:

```
http://localhost:4200
```

Ako se pogleda priložena slika, moguće je primetiti sledeće stvari u vezi sa aplikacijom:

- Moguće je kliknuti na niti ([Threads](#)) za izbor osobe za čakanje;
- Botovi će odgovoriti na poslatu poruku;
- Broj nepročitanih poruka u gornjem desnom uglu ostaje sinhronizovan sa aktuelnim brojem nepročitanih poruka.

Moguće je ukratko izneti ideju kako će ova aplikacija biti kreirana. Postojaće:

- 3 komponente najvišeg nivoa ([top-level](#));
- 3 modela;
- 3 servisa.

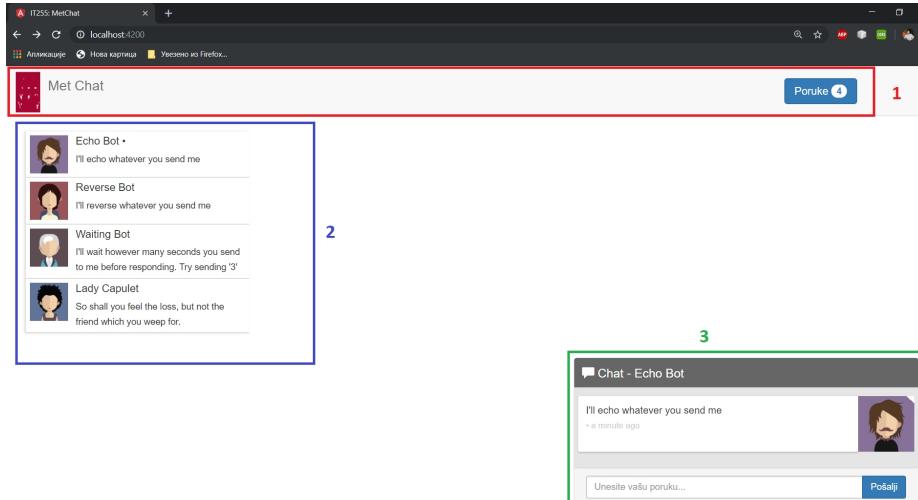
KOMPONENTE

Stranica aplikacije je podeljena u sekcije koje kreiraju tri glavne komponente.

Stranica aplikacije je podeljena u sekcije koje kreiraju tri glavne [komponente](#):

- [ChatNavBarComponent](#) - sadrži broj nepročitanih poruka;
- [ChatThreadsComponent](#) - prikazuje listu niti na koje se može kliknuti, kao i najnoviju poruku i avatar razgovora;
- [ChatWindowComponent](#) - prikazuje poruke u tekućoj niti i polje za unos teksta za slanje nove poruke.

Sledećom slikom su jasno izdvojene navedene komponente u stranici aplikacije.

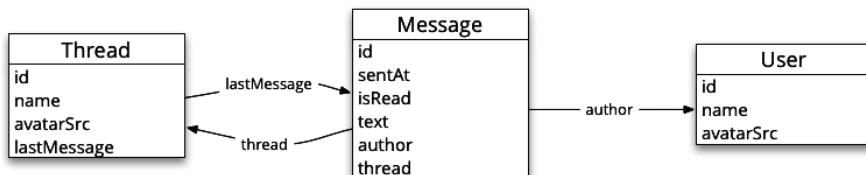


Slika 2.2 Tri izdvojene komponente u stranici aplikacije [izvor: autor]

MODELI

Aplikacija poseduje tri modelske klase.

Aplikacija, kao što je već istaknuto, poseduje tri modelske klase. [Modeli](#), kao i njihove veze, ilustrovani su sledećom slikom.



Slika 2.3 Modeli čet aplikacije [izvor: autor]

Sa slike je jasno moguće izdvojiti tri klase:

- *User* - čuva informacije o učesnicima u časkanju.
- *Message* - čuva individualne poruke;
- *Thread* - čuva kolekciju *Message* objekata kao i izvesne podatke u vezi sa konverzacijom.

SERVISI

Neophodno je obezbeđivanje servisa za svaki kreirani model.

U aplikaciji, za svaki od kreiranih modela biće potrebno obezbediti odgovarajući servis. Servisi će biti objekti jedinci (singleton) kojima će biti dodeljene dve uloge:

- Obezbeđivanje toka podataka na kojeg aplikacija može da se prijavi;
- Obezbeđivanje operacija koje modifikuju podatke.

Na primer, posmatra se servis pod nazivom *UserService*:

- objavljuje tok koji emituje tekućeg korisnika;
- nudi funkciju *setCurrentUser()* koja će podesiti aktuelnog korisnika (to znači, emitovanje tekućeg korisnika iz toka *currentUser*):

REZIME OU

Servisi održavaju tokove koji emituju modele, komponente se prijavljuju na tokove i kreiraju izlaze.

Na najvišem nivou, arhitektura podataka aplikacije je jednostavna:

- servisi održavaju tokove koji emituju modele (na primer, *Messages*);
- komponente se prijavljuju na tokove i kreiraju izlaze u skladu sa najnovijim vrednostima.

Na primer, komponenta *ChatThreads* osluškuje najnoviju listu niti iz servisa *ThreadService*, a *ChatWindow* komponenta se prijavljuje za korišćenje najnovije liste poruka.

U nastavku lekcije, cilj je da se dublje sagleda implementacija navedenog primenom Angular-a i RxJS. Započinje se kreiranjem modela, a nakon toga će biti kreirani i servisi za rukovanje tokovima i na samom kraju i komponente.

✓ Poglavlje 3

Implementiranje modela

USER I THREAD MODELI APLIKACIJE

Aplikacija poseduje tri modelske klase za koje će biti razvijani servisi.

Aplikacija, kao što je već istaknuto, poseduje tri modelske klase:

- *User* - čuva informacije o učesnicima u časkanju.
- *Message* - čuva individualne poruke;
- *Thread* - čuva kolekciju *Message* objekata kao i izvesne podatke u vezi sa konverzacijom.

U narednom izlaganju će biti neophodno priložiti listinge modelskih klasa kao osnova za dalju diskusiju.

Modelska klasa *User* je veoma jednostavna, sadrži polja: *id*, *name* i *avatar*. Klasa se nalazi u datoteci */src/app/user/user.model.ts* aktuelnog projekta i priložena je sledećim listingom:

```
import { uuid } from '../util/uuid';

/**
 * A reprezentuje agenta koji šalje poruke
 */
export class User {
    id: string;

    constructor(public name: string,
                public avatarSrc: string) {
        this.id = uuid();
    }
}
```

Modelska klasa *Thread* je, takođe, veoma jednostavna *TypeScript* klasa, koja sadrži polja: *id*, *lastMessage*, *name* i *avatarSrc*. Klasa se nalazi u datoteci *src/app/thread/thread.model.ts* aktuelnog projekta i priložena je sledećim listingom:

```
import { Message } from '../message/message.model';
import { uuid } from '../util/uuid';

/**
 * Thread reprezentuje grupu objekata tipa Users koji razmenjuju poruke
 */
export class Thread {
    id: string;
```

```
lastMessage: Message;  
name: string;  
avatarSrc: string;  
  
constructor(id?: string,  
           name?: string,  
           avatarSrc?: string) {  
  this.id = id || uuid();  
  this.name = name;  
  this.avatarSrc = avatarSrc;  
}  
}
```

Iz priloženog listinga je moguće je primetiti da se u klasi čuva referenca na poslednju poruku. Ovim je omogućeno prikazivanje pregleda najnovije poruke iz listi niti.

MESSAGE MODEL APIAKCIJE

Message je, takođe, jednostavna TypeScript klasa ali sa drugačijom formom konstruktora.

Message je, takođe, jednostavna *TypeScript* klasa ali sa drugačijom formom konstruktora nego što je bio slučaj sa prethodne dve klase. Klasa se čuva u datoteci: */src/app/message/message.model.ts* i njen kod je priložen u formi sledećeg listinga:

```
import { User } from '../user/user.model';  
import { Thread } from '../thread/thread.model';  
import { uuid } from '../../util/uuid';  
  
/**  
 * Message representuje jednu poruku poslatu u Thread  
 */  
export class Message {  
  id: string;  
  sentAt: Date;  
  isRead: boolean;  
  author: User;  
  text: string;  
  thread: Thread;  
  
  constructor(obj?: any) {  
    this.id          = obj && obj.id          || uuid();  
    this.isRead     = obj && obj.isRead     || false;  
    this.sentAt     = obj && obj.sentAt     || new Date();  
    this.author     = obj && obj.author     || null;  
    this.text       = obj && obj.text       || null;  
    this.thread     = obj && obj.thread     || null;  
  }  
}
```

Šablon koji je predstavljen unutra konstruktora dozvoljava simuliranje primene argumenata ključnih reči unutar konstruktora. Koristeći navedeni šablon, moguće je kreirati nov Message objekat koristeći bilo koji dostupan podatak, pri čemu nije potrebno voditi računa o redosledu argumenata. Na primer, moguće je uraditi sledeće:

```
let msg1 = new Message();  
  
# ili sledeće  
  
let msg2 = new Message({  
text: "Pozdrav Vlado!!!"  
})
```

Sada je prvi zadatak uspešno obavljen - kreirani su modeli. Sledi bavljenje sve složenijim zadacima, a prvi od njih je obezbeđivanje adekvatnih servisa za svaku od kreiranih modelskih klasa.

▼ Poglavlje 4

Implementiranje servisa

IMPLEMENTACIJA SERVISA USERSERVICE

Poenta primene servisa UserService jeste upravljanje tekućim korisnikom.

Poenta primene servisa `UserService` jeste da obezbedi mesto na kojem će aplikacija moći da pribavi saznanja u vezi sa aktuelnim korisnikom i da obavesti ostale komponente aplikacije ukoliko je došlo do promena u vezi sa aktuelnim korisnikom.

Prvi korak koji je potrebno učiniti jeste kreiranje `TypeScript` klase i dodavanje dekoratora `@Injectable`.

Kreirana klasa `UserService` se nalazi na sledećoj lokaciji `/src/app/user/users.service.ts` i određena je sledećim listingom:

```
import { Injectable } from '@angular/core';
import { Subject, BehaviorSubject } from 'rxjs';
import { User } from './user.model';

/**
 * UserService upravlja tekućim korisnikom
 */
@Injectable()
export class UserService {
    // `currentUser` sadrži tekućeg korisnika
    currentUser: Subject<User> = new BehaviorSubject<User>(null);

    public setCurrentUser(newUser: User): void {
        this.currentUser.next(newUser);
    }
}

export const userServiceInjectables: Array<any> = [
    UserService
];
```

TOK PODATAKA CURRENTUSER

Podešavanje toka podataka koji će rukovati tekućim korisnikom.

U prethodnom listingu postoji jedan detalj koji je od presudnog značaja ja trenutno izlaganje - podešavanje toka podataka koji će rukovati tekućim korisnikom. Taj detalj je izolovan sledećim listingom:

```
// `currentUser` sadrži tekućeg korisnika
currentUser: Subject<User> = new BehaviorSubject<User>(null);
```

Ovde se mnogo toga dešava i neophodno je odmah preći na suštinu:

- definisana je objektna promenljiva `currentUser` koja predstavlja tok podataka `Subject`;
- konkretno, `currentUser` je objekat tipa `BehaviorSubject` (`RxJS` klasa) koji sadrži objekat `User`;
- početna vrednost ovog toka podataka je `null` (argument konstruktora);

Ukoliko se student nije do sada susretao sa `RxJS`, onda verovatno ima poteškoća da shvati šta predstavlja: `Subject` ili `BehaviorSubject`. `Subject` se može jednostavno shvatiti kao tok podataka u koji je moguće upisivati i iz kojeg je moguće čitati ("read/write" stream).

Tehnički `Subject` implementira i `Observable` (<https://github.com/Reactive-Extensions/RxJS/blob/master/doc/api/core/observable.md>) i `Observer` (<https://github.com/Reactive-Extensions/RxJS/blob/master/doc/api/core/observer.md>) interfejse.

Jedna od posledica primene tokova može biti da novi pretplatnik rizikuje da propusti najnoviju vrednost iz toka iz razloga što se poruke objavljuju trenutno. Klasom `BehaviourSubject` kompenzuje se navedeni nedostatak. Klasa `BehaviourSubject` (<https://github.com/Reactive-Extensions/RxJS/blob/master/doc/api/subjects/behaviorsubject.md>) poseduje specijalnu osobinu u kojoj se čuva poslednja vrednost. To znači da će svaki pretplatnik toka dobiti poslednju vrednost. Ovo je odlično iz razloga što se svaki deo aplikacije može prijaviti na tok `UserService.currentUser` i da dobije informaciju ko je tekući korisnik.

U nastavku je neophodno definisati mehanizme za podešavanje novog korisnika.

DEFINISANJE NOVOG KORISNIKA

Neophodno je definisati mehanizme za podešavanje novog korisnika.

U nastavku je neophodno definisati način objavljuvanja novog korisnika u okviru toka podataka kad god se trenutni korisnik promeni (na primer, u slučaju prijavljivanja na aplikaciju - `login`).

Postoje dva načina za predstavljanje API-ja za realizovanje navedenog:

- 1. **Dodavanje novog korisnika direktno u tok podataka**

Najjednostavniji način ažuriranja trenutnog korisnika jeste posedovanje klijentata servisa `UserService` koji na jednostavan način dodaju novog korisnika direktno u tok, na primer na sledeći način:

```
UserService.currentUser.subscribe(newUser) => {
  console.log('Novi korisnik je :', newUser.name);
```

```
}

// => Novi korisnik je : originalUserName

let u = new User('Vlada', 'anImgSrc');
UsersService.currentUser.next(u);

// => Novi korisnik je : Vlada
```

Razlog za korišćenje ovog pristupa je mogućnost ponovnog korišćenja postojećeg API - ja iz toka, bez potrebe za uvođenjem novog koda ili API-ja.

- 2. Kreiranje `setCurrentUser(newUser: User)` metode

Drugi način na koji je moguće obaviti ažuriranje trenutnog korisnika je kreiranje pomoćne metode servisa `UsersService`, na sledeći način (datoteka:`/src/app/user/users.service.ts`):

```
public setCurrentUser(newUser: User): void {
    this.currentUser.next(newUser);
}
```

Moguće je primetiti da se i dalje primenjuje metoda `next()` na tok `currentUser`, pa se postavlja pitanje: "Zašto bi trebalo o ovome voditi računa"?

Obuhvatanjem poziva metode `next()` unutar metode `setCurrentUser()` dobija se dodatni prostor za izmenu implementacije servisa `UsersService` bez potrebe za invazivnim operacijama nad klijentima.

U ovom slučaju, teško je izvršiti favorizovanje jednog od navedenih pristupa, ali razlika u slučaju održavanja velikih projekata može biti značajna.

SERVIS MESSAGESSERVICE

Servis MessagesService je stub aplikacije.

Servis `MessagesService` je stub aplikacije. U konkretnom primeru, sve poruke "teku" kroz servis `MessagesService`.

Ovaj servis poseduje sofisticiranije tokove podataka u odnosu na prethodno demonstrirani `UsersService`. Postoji 5 tokova koji čine servis `MessagesService`, od toga tri toka za upravljanje podacima (*data management streams*) i dva toka akcija (*action streams*).

Tri toka za upravljanje podacima su:

- `newMessages` - emituje svaku novu poruku jedanput;;
- `messages` - emituje niz `current Messages`;
- `updates` - izvodi operacije nad porukama.

TOK NEWMESSAGES

*newMessages je tok tipa **Subject** za objavljivanje svakog novog objekta tipa **Message** tačno jednom.*

Tok podataka *newMessages* je tok tipa **Subject** čiji je zadatak objavljivanje svakog novog objekta tipa **Message** (nove poruke) tačno jednom.

Servis **MessageServis** definiše navedeni tok unutar koda koji će biti čuvan unutar datoteke */src/app/message/messages.service.ts* i njegova definicija započinje sledećim listingom:

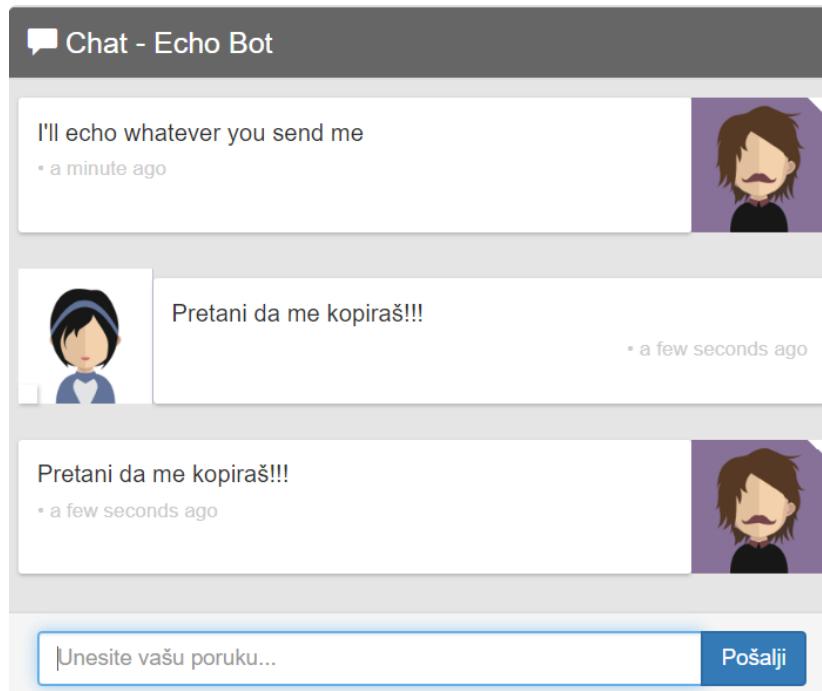
```
@Injectable()  
export class MessagesService {  
    // tok koji samo jednom objavljuje poruke  
    newMessages: Subject<Message> = new Subject<Message>();
```

Pored deklaracije toka *newMessages* moguće je primetiti i da je servisna klasa obeležena dekoratorom **@Injectable()** što znači da je primenljiva tokom izvođenja koncepta umetanja zavisnosti u ostalim delovima ove aplikacije.

Dalje, od posebnog značaja za nastavak izlaganja jeste kreiranje pomoćne metode **addMessages()** za kreirani tok koju je neophodno dodati u klasu servisa **MessageServis**. Definicija metode je prikazana sledećim listingom:

```
// imperativna funkcija za tok akcije  
addMessage(message: Message): void {  
    this.newMessages.next(message);  
}
```

Takođe bi bilo korisno ukoliko postoji tok koji će dobiti sve poruke iz niti koja ne potiče od određenog korisnika. Na primer, uzmite u obzir čet korisnika pod nazivom **Echo Bot** koji ponavlja svaku unetu poruku korisnika koji sa njim časka (sledećom slikom je demonstrirano časkanje sa ovim korisnikom).



Slika 4.1 Časkanje sa Echo Bot korisnikom [izvor: autor]

DODATNA RAZMATRANJA TA TOK NEMESSAGES

Implementacijom Echo Bot - a neophodno sprečiti da program upadne u beskonačnu petlju.

Neophodno je vratiti se na sliku priloženu u prethodnoj sekciji i dati dodatna pojašnjenja u vezi sa implementacijom simuliranog čet korisnika pod nazivom *Echo Bot*.

Implementacijom *Echo Bot* - a neophodno je sve obezbediti da program ne dođe u situaciju upadanja u beskonačnu petlju što bi dovelo do toga bi *Echo Bot* mogao da ponavlja vlastite poruke.

Da bi implementacija protekla korektno neophodno je obaviti prijavljivanje na tok *newMessages* i obaviti filtriranje svih poruka koje su:

1. deo odgovarajuće niti;
2. nisu kreirane od stane simuliranog čet korisnika pod nazivom *Echo Bot*.

Ovo je moguće zamisliti kao da se kaže da se za određenu temu želi tok poruka koje su „za“ konkretnog korisnika.

Zbog svega navedenog, a za obezbeđivanje korektno obavljene implementacije, u aktuelnu servisnu klasu *MessageService*, neophodno je dodati metodu pod nazivom *messagesForThreadUser()* koja je priložena sledećim listingom:

```
messagesForThreadUser(thread: Thread, user: User): Observable<Message> {
    return this.newMessages
        .filter((message: Message) => {
```

```
// pripada ovoj niti
return (message.thread.id === thread.id) &&
// nije odobreno od ovog korisnika
(message.author.id !== user.id);
});
}
```

Kreirana metoda `messagesForThreadUser()` kao argumente uzima nit (`Thread` objekat) i korisnika (`User` objekat) i vraća nov tok poruka (`Messages`) koje su filtrirane i nisu kreirane od strane korisnika. To znači, ovo je tok poruka „svih ostalih“ u ovoj niti.

TOK MESSAGES

Tok `newMessages` emituje pojedinačne poruke, dok tok `Messages` emituje niz poslednjih poruka.

Neophodno je napraviti paralelu između ovog i prethodnog toka. Tok `newMessages` emituje pojedinačne poruke, dok tok `Messages` emituje niz poslednje kreiranih poruka.

Tok `messages` se deklariše unutar klase servisa `MessageService` na sledeći način:

```
// `messages` tok koji emituje niz najsvežijih poruka
messages: Observable<Message[]>;
```

NAPOMENA: Tip `Message[]` je potpuno isto što i `Array<Message>`. Drugi način prisanja, ekvivalentan prethodnom, bi bio: `Observable<Array<Message>>`. Kada se definiše da je tip poruka `Observable<Message[]>` podrazumeva se da ovaj tok emituje niz (objekata `Message`), a ne pojedinačne poruke.

Postavlja se novo pitanje: "Kako se poruke popunjavaju?" Za to je neophodno razviti novu diskusiju koja uključuje nov tok pod nazivom `updates` i šablon pod nazivom tok operacija (*operation stream*).

ŠABLON TOKA OPERACIJA (OPERATION STREAM PATTERN)

Uvek se čuva niz najaktuelnijih poruka i koristi tok funkcija koje se primenjuju na poruke.

Pre bilo kakvog upuštanja u detaljnu analizu i diskusije, neophodno je izložiti ključne ideje:

1. stanje poruka će biti održavano tako da se uvek čuva niz najaktuelnijih poruka (poslednjih);
2. koristi se tok `updates` koji predstavlja tok funkcija koje se primenjuju na poruke.

Dalje, moguće je rezonovati na sledeći način: bilo koja funkcija koja je dodata na tok *updates* izmeniće trenutnu listu poruka. Funkcija koja je dodata na tok *updates* trebalo bi da prihvati listu objekata *Message*, a kao rezultat da vrati izmenjenu listu objekata *Message* (listu poruka). Sledećim interfejsom je formalizovana navedena ideja (interfejs je dodat, takođe, u datoteku *src/app/message/messages.service.ts*):

```
interface IMessagesOperation extends Function {  
  (messages: Message[]): Message[];  
}
```

Sada je moguće, u okviru iste datoteke i unutar servisne klase *MessageService*, pristupiti kreiranju pomenutog toka operacija *updates*. To je obavljeno dodavanjem koda iz sledećeg listinga:

```
// `updates` prima operacije koje se primenjuju na `messages`  
// način na koji se izvode promene nad svim porukama  
// sačuvanim u `messages`  
updates: Subject<any> = new Subject<any>();
```

Poznato je da tok *updates* prihvata operacije koje će biti primenjene na listu poruka. Međutim, otvara se novo pitanje. Kako se obavlja konekcija? Ovde je potrebno pozabaviti se konstruktorom servisne klase *MessagesService* na sledeći način:

```
constructor() {  
  this.messages = this.updates  
    // osmatra ažuriranja i akumulira opeeracije poruka  
    .scan((messages: Message[],  
      operation: IMessagesOperation) => {  
      return operation(messages);  
    },  
    initialMessages)  
  // koristi se najskorija lista poruka za sve
```

Priloženim kodom je uvedena nova funkcija toka *scan()* (detaljnije na linku: <https://github.com/Reactive-Extensions/RxJS/blob/master/doc/api/core/operators/scan.md>). Ukoliko je nekome poznato funkcionalno programiranje, funkcija *scan()* se slično ponaša funkciji *reduce()* čiji je zadatak izvršavanje funkcije za svaki element dolaznog toka i akumuliranje vrednosti. Specifičnosti metode *scan()* su što ona omogućava emitovanje vrednosti za svaki međurezultat. To znači, da ona ne čeka da se tok kompletira da bi objavila trenutni rezultat, a to je upravo ono što se želi postići.

Pozivom *this.updates.scan* kreira se nov tok koji je pretplaćen na tok *updates* i u svakom prolasku se dešava:

1. poruke se akumuliraju;
2. nova operacija se primenjuje;
3. vraća se nov *Message[]*.

DELJENJE TOKA

Veoma bitne stvar u vezi sa tokovima jeste da oni nisu podrazumevano deljivi.

Veoma bitne stvar u vezi sa tokovima jeste da oni nisu podrazumevano deljivi. To znači, ukoliko jedan od pretplatnika pročita vrednosti iz toka ona može nestati zauvek. U slučaju poruka, koje su u centru pažnje aktuelnog primer, neophodno je da se :

1. obezbedi deljenje toka između većeg broja pretplatnika;
2. obezbedi ponavljanje poslednje vrednosti za svakog od pretplatnika koji kasni sa zahtevom za tom vrednošću.

Da bi navedena funkcionalnost bila dostupna, koriste se dva operatora:

- `publishReplay` - dozvoljava deljenje pretplate između većeg broja pretplatnika i ponavlja veći broj vrednosti za buduće pretplatnike;
- `refCount` - olakšava upotrebu povratne vrednosti objavljivanja, upravljanjem kada će *Observable* objekti emitovati vrednosti.

Da bi navedeno bilo implementirano unutar postojećeg servisa `MessageService` (datoteka:[/src/app/message/messages.service.ts](#)) neophodno je kod navedene klase proširiti dodavanjem koda koji odgovara sledećem pozivu pomenute metode `scan()`:

```
// osmatra ažuriranja i akumulira operacije poruka
.scan((messages: Message[],
       operation: IMessagesOperation) => {
    return operation(messages);
},
      initialMessages)
// koristi se najskorija lista poruka za sve
// zainteresovane za prijavljivanje i kešira se poslednja poznata lista poruka

.publishReplay(1)
.refCount();
```

DODAVANJE MESSAGE OBJEKATA U TOK MESSAGES

Reaktivni način kreiranja nove poruke podrazumeva posedovanje toka za dodavanje poruka u listu.

U nastavku je cilj manipulisanje objektima poruka, odnosno obezbeđivanje mehanizma za njihovo dodavanje u tok messages. To je moguće uraditi na sledeći način:

```
var myMessage = new Message(/* ovde idu parametri... */);
```

```
updates.next( messages: Message[] ): Message[] => {
    return messages.concat(myMessage);
}
```

U priloženom kodu egzistira deo koji se odnosi na dodavanje operacije u tok `updates`. Ovo ima za efekat da je tok `messages` preplaćen na funkcije iz toka `updates` koji će primenjivati operacije pomoću kojih će biti obavljeno spajanje (`concat`) novih poruka sa akumuliranim listom poruka.

Jedan od problema navedenog pristupa ogleda se u činjenici da je složen za upotrebu. Bilo bi poželjno da se ne mora svaki put pisati navedena unutrašnja funkcija. Na primer, alternativno je moguće uraditi sledeće:

```
addMessage(newMessage: Message) {
    updates.next( messages: Message[] ): Message[] => {
        return messages.concat(newMessage);
    }
}

var myMessage = new Message(/* Ovde idu parametri... */);
MessagesService.addMessage(myMessage);
```

Ovo je malo bolje, ali to nije "reaktivan način". Delom jer se ova radnja stvaranja poruke ne može povezivati sa drugim tokovima.

Reaktivan način kreiranja nove poruke podrazumeva posedovanje toka koji prihvata objekte `Messages` koje bi trebalo dodati u listu. Ponovo, ovo može biti novo za studente koji nisu još vešti u ovakvom načinu softverskog razmišljanja. Slede smernice kako je ovo moguće implementirati.

U prvom koraku je neophodno kreirati toka akcija pod nazivom `create`. Ovde je važno napomenuti da se izraz tok akcija (`action stream`) koristi isključivo da bi opisao njegovu ulogu u servisu.

```
// tokovi akcija
create: Subject<Message> = new Subject<Message>();
```

Sledeći korak je podešavanja ovog toka unutar konstruktora servisne klase.

```
this.create
    .map( message: Message ): IMessagesOperation {
        return (messages: Message[] ) => {
            return messages.concat(message);
        };
    }
}
```

Operator `map` se ponaša kao ugrađena `JavaScript` funkcija `Array.map` sa razlikom što radi sa tokovima.

PRETPLATA I OSLUŠKIVANJE TOKA

Za svaku ulaznu poruku, neophodno je vratiti `IMessagesOperation` koji dodaje poruke na listu.

Vraća se akcenat na prethodnu sekciju. Sada postoji sledeći slučaj: "za svaku poruku (`Message` objekat) koja se prihvata kao ulaz, neophodno je vratiti objekat `IMessagesOperation` koji dodaje poruke na listu". Drugim rečima, tok će emitovati funkciju koja će prihvati listu `Message` objekata i dodati `Message` objekat na postojeću listu.

Sada kada je kreiran tok, ostala je još jedna stvar da se uradi: neophodno ga je zakačiti za tok `updates`. Ovo se čini pozivom metode `subscribe()` kao što je učinjeno u sledećem listingu:

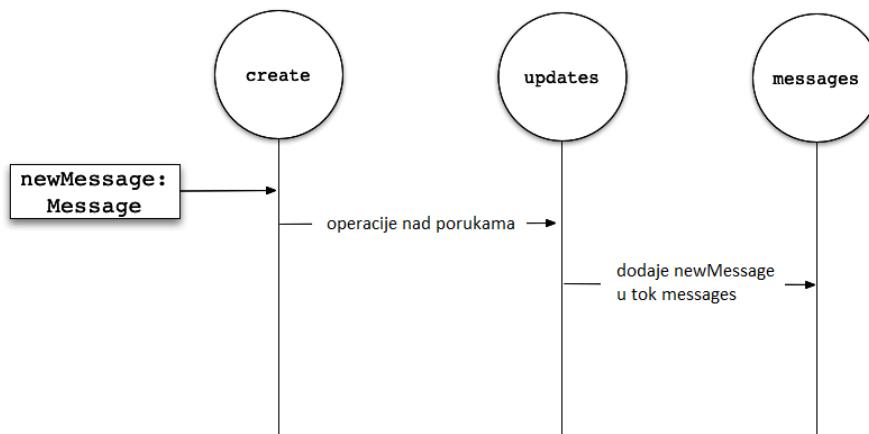
```

this.create
    .map( function(message: Message): IMessagesOperation {
        return (messages: Message[]) => {
            return messages.concat(message);
        };
    })
    .subscribe(this.updates);

```

Ono što je upravo urađeno jeste prijavljivanje (preplata) toka `updates` za osluškivanje toka `create`. To znači da ukoliko tok `create` primi poruku (`Message` objekat) on će emitovati objekat `IMessagesOperation` kojeg će primiti tok `updates` i posmatrana poruka će biti dodata u listu poruka (`Message` objekata).

Sledećim dijagramom tokova je prikazana trenutna situacija:



Slika 4.2 Kreiranje nove poruke - početak iz toka "create" [izvor: autor]

Demonstrirano je odlično jer sada postoji:

1. tekuća lista poruka iz toka `messages`;
2. način obrade operacija nad tekućom listom poruka (preko toka `updates`);
3. jednostavan za upotrebu tok za čuvanje operacija koje se izvode nad članicama toka `updates` (preko toka `create`).

U bilo kojem delu koda, ukoliko je potrebno dobiti informacije iz najaktuelnije liste poruka, neophodno je samo posetiti tok *messages*.

POVEZIVANJE TOKA NEWMESSAGE U KOMPOZICIJU TOKOVA (FLOW)

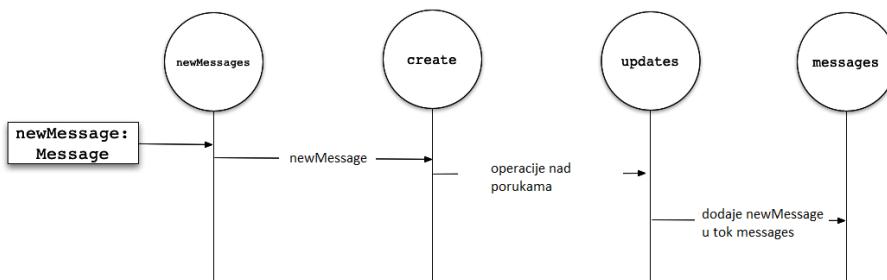
Na prikazanu kompoziciju tokova neophodno je povezati i tok newMessages.

Kao što je upravo istaknuto, u bilo kojem delu koda, ukoliko je potrebno dobiti informacije iz najaktuelnije liste poruka, neophodno je samo posetiti tok *messages*. Međutim, otvara se nov problem. Na prikazanu kompoziciju tokova još uvek nije povezan tok *newMessages*.

Zaista bi bilo poželjno ukoliko bi postojao mehanizam za jednostavno povezivanje na nov složeni tok bilo kojeg *Message* objekta koji potiče iz toka *newMessages*. Pokazuje se da je moguće ukoliko se u aktuelnoj servisnoj klasi (datoteka: */src/app/message/messages.service.ts*) doda još veoma malo koda:

```
this.newMessages
    .subscribe(this.create);
```

Dijagram tokova sada izgleda kao na sledećoj slici:



Slika 4.3 Kreiranje nove poruke - početak iz toka "newMessages" [izvor: autor]

Sada je tok podataka za posmatranu aplikaciju kompletiran. Velika korist iz obavljenih zadataka ogleda se u činjenici da je moguće obaviti prijavljivanje na tok individualnih poruka (preko *newMessages*) ali ukoliko je potrebna najažurnija lista poruka, dostupno je prijavljivanje na i tok *messages*.

Za sticanje zaokružene slike, konačno, potrebno je sve izolovane i prikazane listinge delova servisne klase *MessageService* sklopliti u celinu. Sledećim listingom dat je celokupan kod servisa *MessageService*.

```
import { Injectable } from '@angular/core';
import { Subject, Observable } from 'rxjs';
import { User } from '../user/user.model';
import { Thread } from '../thread/thread.model';
import { Message } from '../message/message.model';
```

```
const initialMessages: Message[] = [];

interface IMessagesOperation extends Function {
  (messages: Message[]): Message[];
}

@Injectable()
export class MessagesService {
  // tok koji samo jednom objavljuje poruke
  newMessages: Subject<Message> = new Subject<Message>();

  // `messages` tok koji emituje niz najsvežijih poruka
  messages: Observable<Message[]>;

  // `updates` prima operacije koje se primenjuju na `messages`
  // način na koji se izvode promene nad svim porukama
  // sačuvanim u `messages`
  updates: Subject<any> = new Subject<any>();

  // tokovi akcija
  create: Subject<Message> = new Subject<Message>();
  markThreadAsRead: Subject<any> = new Subject<any>();

  constructor() {
    this.messages = this.updates
      // osmatra ažuriranja i akumulira operacije poruka
      .scan((messages: Message[],
        operation: IMessagesOperation) => {
        return operation(messages);
      },
      initialMessages)
      // koristi se najskorija lista poruka za sve
      // zainteresovane za prijavljivanje i kešira se poslednja poznata lista poruka

      .publishReplay(1)
      .RefCount();
  }

  // `create` uzima Message i dodaje operaciju (unutrašnju funkciju)

  this.create
    .map( function(message: Message): IMessagesOperation {
      return (messages: Message[]) => {
        return messages.concat(message);
      };
    })
    .subscribe(this.updates);

  this.newMessages
    .subscribe(this.create);

  // `markThreadAsRead` uzima Thread i dodaje operaciju
```

```
this.markThreadAsRead
    .map( (thread: Thread) => {
        return (messages: Message[]) => {
            return messages.map( (message: Message) => {
                // ovde je direktno manipolisanje `message`
                if (message.thread.id === thread.id) {
                    message.isRead = true;
                }
                return message;
            });
        };
    })
    .subscribe(this.updates);

}

// imperativna funckija za tok akcije
addMessage(message: Message): void {
    this.newMessages.next(message);
}

messagesForThreadUser(thread: Thread, user: User): Observable<Message> {
    return this.newMessages
        .filter((message: Message) => {
            // pripada ovoj niti
            return (message.thread.id === thread.id) &&
                // nije odobreno od ovog korisnika
                (message.author.id !== user.id);
        });
}

export const messagesServiceInjectables: Array<any> = [
    MessagesService
];
```

TESTIRANJE KREIRANOG SERVISA MESSAGESSERVICE

Testiranje kreiranog servisa `MessagesService` i simulacija njegovog funkcionisanja u aplikaciji.

Moguće je, u nastavku, obaviti testiranje kreiranog servisa `MessagesService` sa ciljem simulacije njegovog funkcionisanja u aplikaciji. Test primer je kreiran u datoteci: `src/app/message/messages.service.spec.ts` i sledi njegov početni listing u kojem je kreirano nekoliko instanci modela:

```
import { MessagesService } from './messages.service';

import { Message } from './message.model';
import { Thread } from '../../thread/thread.model';
import { User } from '../../user/user.model';

describe('MessagesService', () => {
  it('should test', () => {

    const user: User = new User('Vlada', '');
    const thread: Thread = new Thread('t1', 'Vlada', '');
    const m1: Message = new Message({
      author: user,
      text: 'Ćao!',
      thread: thread
    });

    const m2: Message = new Message({
      author: user,
      text: 'Pozdrav!',
      thread: thread
    });
  });
});
```

Dalje se razvija test primer, dodaju se pretplate na neke od kreiranih tokova:

```
const messagesService: MessagesService = new MessagesService();

// osluškuje svaku individualno pristiglu poruku
messagesService.newMessages
  .subscribe( message: Message ) => {
    console.log('=> newMessages: ' + message.text);
  };

// osluškuje tok aktuelnih poruka
messagesService.messages
  .subscribe( messages: Message[] ) => {
    console.log('=> messages: ' + messages.length);
  };

messagesService.addMessage(m1);
messagesService.addMessage(m2);

// => messages: 1
// => newMessages: Ćao!
// => messages: 2
// => newMessages: Pozdrav!
});
```

Neophodno je primetiti da iako je prvo obavljena pretplata na `newMessages`, a `newMessages` je pozvan direktno od `addMessage`, `messages` pretplata je prva učitana. Razlog za to je zato što je tok `messages` pretplaćen na `newMessages` ranije od pretplate obavljene na ovom testu

POKRETANJE TESTA

U terminalu razvojnog okruženja se u par koraka pokreće test i prati njegovo izvršavanje.

U terminalu razvojnog okruženja se u par koraka pokreće test i prati njegovo izvršavanje:

1. `cd /rxjs-chat` // <-- vaša putanja može da se razlikuje;
2. `npm install`
3. `npm run test`

Sledećom slikom je prikazan rezultat izvršavanja kreiranog test slučaja:

```
> angular-redux-chat@0.0.0 test C:\Users\Vlada\Documents\Angular\rxjs-chat
> ng test

10% building modules 1/1 modules 0 active(node:7008) DeprecationWarning: Tapable.plugin is deprecated. Use new API on `hooks` instead
01.09.2019 19:43:06.732:INFO [Karma]: Karma v1.4.1 server started at http://0.0.0.0:9876/
01.09.2019 19:43:06.735:INFO [Launcher]: Launching browser Chrome with unlimited concurrency
01.09.2019 19:43:06.740:INFO [Launcher]: Starting browser chrome
LOG: '>> messages: 1'
Chrome 76.0.3809 (Windows 10.0.0): Executed 0 of 12 (skipped 9) SUCCESS (0 secs / 0 secs)
LOG: '>> messages: 1'
LOG: '>> newMessages: Čao!'
Chrome 76.0.3809 (Windows 10.0.0): Executed 0 of 12 (skipped 9) SUCCESS (0 secs / 0 secs)
LOG: '>> newMessages: Čao!'
LOG: '>> messages: 2'
Chrome 76.0.3809 (Windows 10.0.0): Executed 0 of 12 (skipped 9) SUCCESS (0 secs / 0 secs)
LOG: '>> messages: 2'
LOG: '>> newMessages: Pozdrav!'
```

Slika 4.4 Pokretanje testa za MessageService [izvor: autor]

Pošto je sve obavljeno na traženi način, pristupa se kreiranju novog servisa aplikacije pod nazivom `ThreadsService`.

RAZVOJ SERVISA THREADSSERVICE

Razvoj servisa ThreadsService za dodatno sticanje znanja u vezi primene RxJS .

Prilikom razvoja klase `ThreadsService` biće razmatrano definisane četiri toka koji vrše emitovanje sledećim redosledom:

1. mapa tekućeg skupa niti (`Threads`) - tok `threads`;
2. hronološka lista niti, najnovija ide na prvo mesto - tok `orderedthreads`;
3. trenutno izabrana nit - tok `currentThread`;
4. lista poruka (`Message` objekata) za trenutno izabranu nit - tok `orderedthreads`.

U narednom izlaganju biće obrađeni navedeni tokovi, a to će predstavljati odličnu podlogu za dodatno sticanje znanja u vezi primene `RxJS` u `Angular` aplikacijama.

MAPA TEKUĆEG SKUPA NITI (THREADS) I TOK THREADS

Početak je rezervisan za kreiranje objektne promenljive thread koja emituje objekte tipa Thread.

Sam početak definisanja klase `ThreadService` je rezervisan za kreiranje objektne promenljive `thread` koja će imati zadatak da emituje objekte tipa `Thread`:

```
import { Injectable } from '@angular/core';
import { Subject, BehaviorSubject, Observable } from 'rxjs/Rx';
import { Thread } from './thread.model';
import { Message } from '../message/message.model';
import { MessagesService } from '../message/messages.service';
import * as _ from 'lodash';

@Injectable()
export class ThreadsService {

    // `threads` je Observable varijabla koja sadrži listu najsvežijih niti
    threads: Observable<{ [key: string]: Thread }>;
```

Primećuje se da definisani tok emituje mapu koja kao ključ poseduje `id Thread` objekta, dok sam `Thread` objekat predstavlja vrednost dostupnu preko navedenog ključa.

Za kreiranje toka koji održava tekuću listu niti, neophodno je dodati sledeće na tok `messagesService.messages`:

```
threads: Observable<{ [key: string]: Thread }>;
```

Neophodno je i podsetiti se da svaki put kada se nova poruka (`Message`) doda u tok, tok `messages` će emitovati niz tekućih poruka. Biće pregledana svaka poruka i neophodno je vratiti jedinstvenu listu niti.

```
constructor(public messagesService: MessagesService) {

    this.threads = messagesService.messages
        .map( (messages: Message[]) => {
            const threads: {[key: string]: Thread} = {};
            // čuva nit poruke u akumuliranom toku `threads`
            messages.map((message: Message) => {
                threads[message.thread.id] = threads[message.thread.id] ||
                    message.thread;
            });
            return threads;
        });
}
```

Moguće je primetiti da će svaki put biti kreirana nova lista niti. Razlog za ovo je mogućnost neželjenog brisanja neke poruke (na primer, napuštanjem konverzacije). Iz razloga što se svaki put preračunava lista niti, prirodno se briše svaka prazna nit.

U listi niti, cilj je prikazivanje pregleda časkanja preko sadržaja (teksta) najskorije korišćenih poruka u dатoj niti.



Slika 4.5 Lista niti sa pregledom časkanja [izvor: autor]

ČUVANJE NOVIH PORUKA ZA SVAKU NIT

Vodenje računa o tome koja je poruka najsvežija prepušteno je poređenju vrednosti sentAt.

Sa ciljem dobijanja prethodne liste biće neophodno omogućiti čuvanje najnovijih *Message* objekata za svaku od niti. Vodenje računa o tome koja je poruka najsvežija prepušteno je poređenju vrednosti *sentAt*.

Da bi navedeno bilo realizovano, neophodno je započeti konstruktor klase *ThreadService* dopuniti na sledeći način:

```
// kešira najnovije poruke za svaku nit
const messagesThread: Thread = threads[message.thread.id];
if (!messagesThread.lastMessage ||
    messagesThread.lastMessage.sentAt < message.sentAt) {
    messagesThread.lastMessage = message;
}
);
return threads;
});
```

Objedinjeno, *threads* izgleda kao u sledećem listingu:

```
this.threads = messagesService.messages
.map( (messages: Message[]) => {
    const threads: {[key: string]: Thread} = {};
    // čuva nit poruke u akumuliranom toku `threads`
    messages.map((message: Message) => {
        threads[message.thread.id] = threads[message.thread.id] ||
            message.thread;

        // kešira najnovije poruke za svaku nit
        const messagesThread: Thread = threads[message.thread.id];
        if (!messagesThread.lastMessage ||
```

```
        messagesThread.lastMessage.sentAt < message.sentAt) {  
            messagesThread.lastMessage = message;  
        }  
    });  
    return threads;  
});
```

TESTIRANJE SERVISA THREADSSERVICE

Sada je moguće obaviti testiranje servisa ThreadsService.

Sada je moguće obaviti testiranje servisa *ThreadsService*. U prvom koraku će biti implementirano nekoliko modela neophodnih za rad (test slučajevi - datoteka: */src/app/thread/thread.service.spec.ts*):

```
import { Message } from './message/message.model';  
import { Thread } from './thread.model';  
import { User } from './user/user.model';  
  
import { ThreadsService } from './threads.service';  
import { MessagesService } from './message/messages.service';  
import * as _ from 'lodash';  
  
describe('ThreadsService', () => {  
  it('should collect the Threads from Messages', () => {  
  
    const nate: User = new User('Vladimir Milićević', '');  
    const felipe: User = new User('Marko Rajević', '');  
  
    const t1: Thread = new Thread('t1', 'Thread 1', '');  
    const t2: Thread = new Thread('t2', 'Thread 2', '');  
  
    const m1: Message = new Message({  
      author: nate,  
      text: 'Ćao!',  
      thread: t1  
    });  
  
    const m2: Message = new Message({  
      author: felipe,  
      text: 'Ovo je prva poruka!!!',  
      thread: t1  
    });  
  
    const m3: Message = new Message({  
      author: nate,  
      text: 'Ovo je druga poruka!!!',  
      thread: t2  
    });
```

```
});
```

Kreiranje test slučaja se nastavlja instanciranjem poznatih servisa i prijavljivanjem na tok *threads*:

```
const messagesService: MessagesService = new MessagesService();
const threadsService: ThreadsService = new ThreadsService(messagesService);

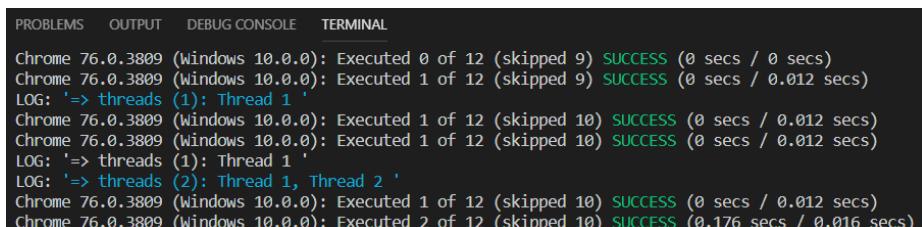
threadsService.threads
  .subscribe( (threadIdx: { [key: string]: Thread }) => {
    const threads: Thread[] = _.values(threadIdx);
    const threadNames: string = _.map(threads, (t: Thread) => t.name)
      .join(', ');
    console.log(`=> threads (${threads.length}): ${threadNames}`);
  });

messagesService.addMessage(m1);
messagesService.addMessage(m2);
messagesService.addMessage(m3);

// => threads (1): Thread 1
// => threads (1): Thread 1
// => threads (2): Thread 1, Thread 2

});
});
```

Pozivom: *ng test* dobija se sledeći izlaz:



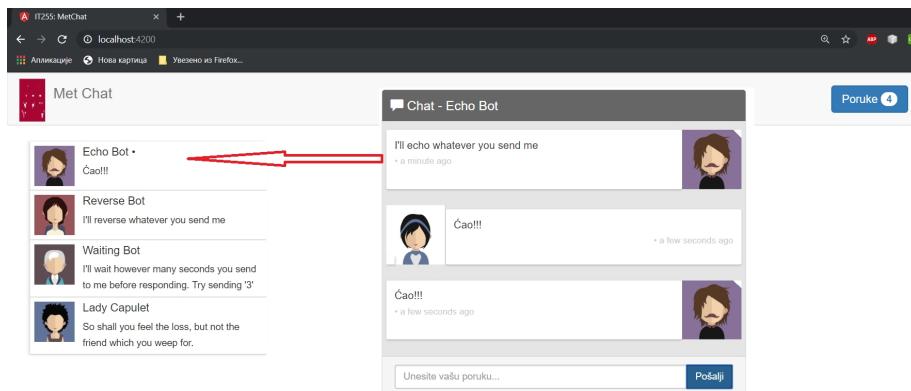
```
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL
Chrome 76.0.3809 (Windows 10.0.0): Executed 0 of 12 (skipped 9) SUCCESS (0 secs / 0 secs)
Chrome 76.0.3809 (Windows 10.0.0): Executed 1 of 12 (skipped 9) SUCCESS (0 secs / 0.012 secs)
LOG: '=> threads (1): Thread 1 '
Chrome 76.0.3809 (Windows 10.0.0): Executed 1 of 12 (skipped 10) SUCCESS (0 secs / 0.012 secs)
Chrome 76.0.3809 (Windows 10.0.0): Executed 1 of 12 (skipped 10) SUCCESS (0 secs / 0.012 secs)
LOG: '=> threads (1): Thread 1 '
LOG: '=> threads (2): Thread 1, Thread 2 '
Chrome 76.0.3809 (Windows 10.0.0): Executed 1 of 12 (skipped 10) SUCCESS (0 secs / 0.012 secs)
Chrome 76.0.3809 (Windows 10.0.0): Executed 2 of 12 (skipped 10) SUCCESS (0.176 secs / 0.016 secs)
```

Slika 4.6 Test izlaz za kreirani test slučaj [izvor: autor]

HRONOLOŠKA LISTA NITI

Cilj je da korisničke niti u pogledima budu poređane prema poslednjim pristiglim porukama

Tok *threads* obezbeđuje mapu koja sadrži listu niti poređanih po indeksima. Međutim, kao što je više puta istaknuto, cilj je da *korisničke niti* u pogledima budu poređane prema poslednjim pristiglim porukama.



Slika 4.7 Hronološka lista niti [izvor: autor]

Da bi navedeno uspešno bilo obavljeno, neophodno je obaviti kreiranje niza objekata klase [Thread](#) poredanih prema vremenu pristizanja poslednjih poruka.

U sledećem koraku započinje definisanje objektne promenljive [orderedThreads](#) unutar klase [ThreadService](#):

```
// `orderedThreads` sarži hronološku listu niti - najnoviji je prvi
orderedThreads: Observable<Thread[]>;
```

Sledeći korak podrazumeva ponovno vraćanje na konstruktor klase [ThreadService](#). Unutar konstruktora biće definisan tok [orderedThreads](#), preko prijavljivanja na tok [threads](#), organizovan prema poslednjim pristiglim porukama. Navedeno je realizovano kreiranjem koda prikazanog u sledećem listingu:

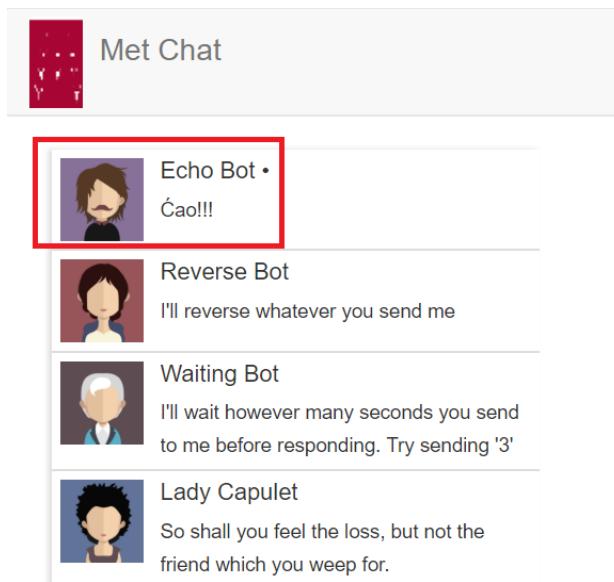
```
this.orderedThreads = this.threads
    .map((threadGroups: { [key: string]: Thread }) => {
      const threads: Thread[] = _.values(threadGroups);
      return _.sortBy(threads, (t: Thread) => t.lastMessage.sentAt).reverse();
   });
```

TRENUTNO IZABRANE NITI

Kreiranje mehanizma koji aplikaciji daje do znanja koja korisnička nit je trenutno izabrana.

Sledeća važna funkcionalnost aplikacije, koja mora da bude obezbeđena, jeste kreiranje mehanizma koji aplikaciji daje do znanja koja korisnička nit je trenutno izabrana. Ovde je posebno potrebno obratiti pažnju, prilikom implementacije:

- koju korisničku nit bi trebalo prikazati u prozoru za čakanje;
- koju nit bi trebalo na adekvatan način obeležiti kao izabranu u listi korisničkih niti (sledeća slika).



Slika 4.8 Obeležavanje izabrane niti simbolom "tačka" [izvor: autor]

Potrebno je da se kreira objekat *BehaviorSubject* (<https://www.learnrxjs.io/subjects/behaviorsubject.html>) koji će čuvati *currentThread*:

```
// `currentThread` sadrži trenutno izabranu nit
currentThread: Subject<Thread> =
  new BehaviorSubject<Thread>(new Thread());
```

Moguće je primetiti da se kao početna vrednost javlja prazan *Thread* objekat. Nikakva dalja podešavanja za *currentThread* nisu potrebna ovde.

NAPOMENA: BehaviorSubject je specijalizovani tipSubject koji zahteva inicijalnu vrednosti i emituje tekuću vrednost novim preplatnicima.

IZBOR TRENUITNE NITI

Izbor trenutne niti kreiranjem nove pomoćne metode koja će obaviti traženi zadatak.

Izbor trenutne niti može biti obavljen na dva načina:

1. potvrdom nove niti direktno preko metode *next()*;
2. kreiranjem nove pomoćne metode koja će obaviti traženi zadatak.

Akcentat će biti na tački 2 i kreiranju adekvatne pomoćne metode (*helper method*) pod nazivom *setCurrentThread()* koja može biti upotrebljena za izbor sledeće niti (nastavak rada u datoteci */src/app/thread/threads.service.ts*):

```
setCurrentThread(newThread: Thread): void {  
    this.currentThread.next(newThread);  
}
```

OZNAČAVANJE TEKUĆE NITI KAO PROČITANE

Kada se izabere nova korisnička nit sve njene poruke biće označene kao pročitane.

Još jedna važna i korisna funkcionalnost čet aplikacije mogla bi da bude mogućnost praćenja broja nepročitanih poruka. Kada se izabere nova korisnička nit, otvorice se i prozor sa listom primljenih poruka i tada sve te poruke moguće je klasifikovati kao pročitane. Neophodno je obaviti sledeće:

1. metoda `messagesService.markThreadAsRead()` će prihvati izabranu nit i sve njene metode će označiti kao pročitane;
2. tok `currentThread` će emitovati jednu nit (`Thread`) koja će predstavljati tekuću izabranu nit.

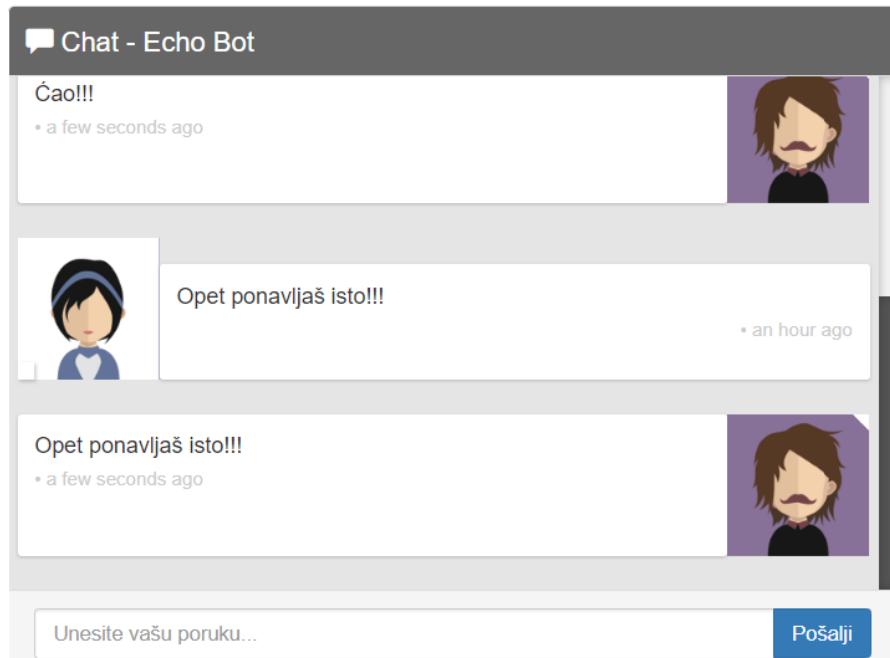
Sledećim listingom je moguće povezati sve gore navedeno:

```
this.currentThread.subscribe(this.messagesService.markThreadAsRead);
```

LISTA PORUKA IZABRANE KORISNIČKE NITI I MOGUĆI PROBLEM

Za tekuću nit je potrebno obaviti prikazivanje liste njenih aktuelnih poruka (Message objekata).

U prethodnom izlaganju je jasno opisano kako je moguće obaviti zbor tekuće niti. Za tekuću nit je potrebno obaviti prikazivanje liste njenih aktuelnih poruka (`Message` objekata).



Slika 4.9 Lista aktuelnih poruka za korisničku nit Echo Bot [izvor: autor]

Implementacija željene funkcionalnosti je malo zahtevnija nego što to izgleda na prvi pogled. Prepostavka je da je navedeno implementirano na sledeći način:

```
var theCurrentThread: Thread;

this.currentThread.subscribe((thread: Thread) => {
    theCurrentThread = thread;
})

this.currentThreadMessages.map(
    (messages: Message[]) => {
        return _.filter(messages,
            (message: Message) => {
                return message.thread.id == theCurrentThread.id;
            }
        )
    }
)
```

Postavlja se pitanje: "Šta ne valja u navedenom pristupu?" Ukoliko dođe do promene u *currentThread*, tok *currentThreadMessages* neće biti obavešten o nastaloj promeni i u njemu će se naći lista zastarelih poruka.

Sledeće pitanje: "Ako se postupi obrnuto, tekuća lista sačuva u varijabli i pretplati na promene toka *currentThread*, šta će se desiti? Javiće se sličan problem. Ovog puta će biti dostupna informacija o promeni niti ali neće biti dostupna o tome kada je poruka stigla.

REŠENJE PROBLEMA

RxJS daje skup operatora koje je moguće upotrebiti za kombinovanje većeg broja tokova.

Konačno, postavlja se ključno pitanje: "Kako je moguće naći rešenje za probleme identifikovane prethodnim pitanjima?"

Rešenje je *RxJS* koji daje skup operatora koje je moguće upotrebiti za kombinovanje većeg broja tokova. U konkretnom slučaju želi se istaći sledeće: "ukoliko se desi promena *currentThread* ili *messagesService.messages*, neophodno je emitovati odgovarajuću vrednost". U ovu svrhu se koristi *RxJS* operator *combineLatest* (<https://github.com/Reactive-Extensions/RxJS/blob/master/doc/api/core/operators/combinelatestproto.md>).

Za početak diskusije neophodno je dodati još malo novog koda u klasu *ThreadService*:

```
this.currentThreadMessages = this.currentThread
    .combineLatest(messagesService.messages,
        (currentThread: Thread, messages: Message[]) => {
```

Kada se obavi kombinovanje dva različita toka, njihovo pojavljivanje se ne dešava istovremeno i ne postoji garancija o dobijanju željene vrednosti iz oba toka. Iz navedenog razloga je neophodno obaviti proveru da li je dobijeno upravo ono što je potrebno; u suprotnom će se kao rezultat najverovatnije javiti prazna lista.

Sada, kada postoje aktuelna nit i poruke, moguće je obaviti filtriranje poruka koje su od značaja (nastavak započetog listinga):

```
this.currentThreadMessages = this.currentThread
    .combineLatest(messagesService.messages,
        (currentThread: Thread, messages: Message[]) => {
            if (currentThread && messages.length > 0) {
                return _.chain(messages)
                    .filter((message: Message) =>
                        (message.thread.id === currentThread.id))
```

Još jedan detalj, pošto su već sagledane poruke za tekuću korisničku nit, dolazi se do prave oblasti u kojoj poruke mogu biti obeležene kao pročitane:

```
return _.chain(messages)
    .filter((message: Message) =>
        (message.thread.id === currentThread.id))
    .map((message: Message) => {
        message.isRead = true;
        return message; })
    .value();
```

Konačno, objedinjavanjem prikazanih kodnih segmenata, moguće je dati konačnu definiciju toka `currentThreadMessages`:

```

this.currentThreadMessages = this.currentThread
    .combineLatest(messagesService.messages,
        (currentThread: Thread, messages: Message[]) => {
    if (currentThread && messages.length > 0) {
        return _.chain(messages)
            .filter((message: Message) =>
                (message.thread.id === currentThread.id))
            .map((message: Message) => {
                message.isRead = true;
                return message; })
            .value();
    } else {
        return [];
    }
});

```

OBJEDINJEN LISTING SERVISNE KLASE THREADSSERVICE

Potrebno je sve izolovane i prikazane listinge delova servisne klase ThreadsService sklopiti u celinu

Za sticanje zaokružene slike, konačno, potrebno je sve izolovane i prikazane listinge delova servisne klase `ThreadsService` sklopiti u celinu. Sledećim listingom dat je celokupan kod servisa `ThreadsService`.

```

import { Injectable } from '@angular/core';
import { Subject, BehaviorSubject, Observable } from 'rxjs/Rx';
import { Thread } from './thread.model';
import { Message } from '../message/message.model';
import { MessagesService } from '../message/messages.service';
import * as _ from 'lodash';

@Injectable()
export class ThreadsService {

    // `threads` je Observable varijabla koja sadrži listu najsvežijih niti
    threads: Observable<{ [key: string]: Thread }>;

    // `orderedThreads` sarži hronološku listu niti - najnoviji je prvi
    orderedThreads: Observable<Thread[]>;

    // `currentThread` sadrži trenutno izabranu nit
    currentThread: Subject<Thread> =
        new BehaviorSubject<Thread>(new Thread());
}

```

```
// `currentThreadMessages` sadrži set poruka za
// trenutno izabrano nit
currentThreadMessages: Observable<Message[]>;

constructor(public messagesService: MessagesService) {

    this.threads = messagesService.messages
        .map( (messages: Message[]) => {
            const threads: {[key: string]: Thread} = {};
            // čuva nit poruke u akumuliranom toku `threads`
            messages.map((message: Message) => {
                threads[message.thread.id] = threads[message.thread.id] ||
                    message.thread;

                // kešira najnovije poruke za svaku nit
                const messagesThread: Thread = threads[message.thread.id];
                if (!messagesThread.lastMessage ||
                    messagesThread.lastMessage.sentAt < message.sentAt) {
                    messagesThread.lastMessage = message;
                }
            });
            return threads;
        });

    this.orderedThreads = this.threads
        .map((threadGroups: { [key: string]: Thread }) => {
            const threads: Thread[] = _.values(threadGroups);
            return _.sortBy(threads, (t: Thread) => t.lastMessage.sentAt).reverse();
        });

    this.currentThreadMessages = this.currentThread
        .combineLatest(messagesService.messages,
            (currentThread: Thread, messages: Message[]) => {
                if (currentThread && messages.length > 0) {
                    return _.chain(messages)
                        .filter((message: Message) =>
                            (message.thread.id === currentThread.id))
                        .map((message: Message) => {
                            message.isRead = true;
                            return message; })
                        .value();
                } else {
                    return [];
                }
            });
}

this.currentThread.subscribe(this.messagesService.markThreadAsRead);
}

setCurrentThread(newThread: Thread): void {
    this.currentThread.next(newThread);
}
```

```
}
```

```
export const threadsServiceInjectables: Array<any> = [
  ThreadsService
];
```

Model podataka i servisi su sada kompletirani! Sve što je potrebno učiniti u nastavku jeste njihovo povezivanje sa pogledima odgovarajućih komponenata.

✓ Poglavlje 5

Arhitektura podataka sa Observables - pogledi

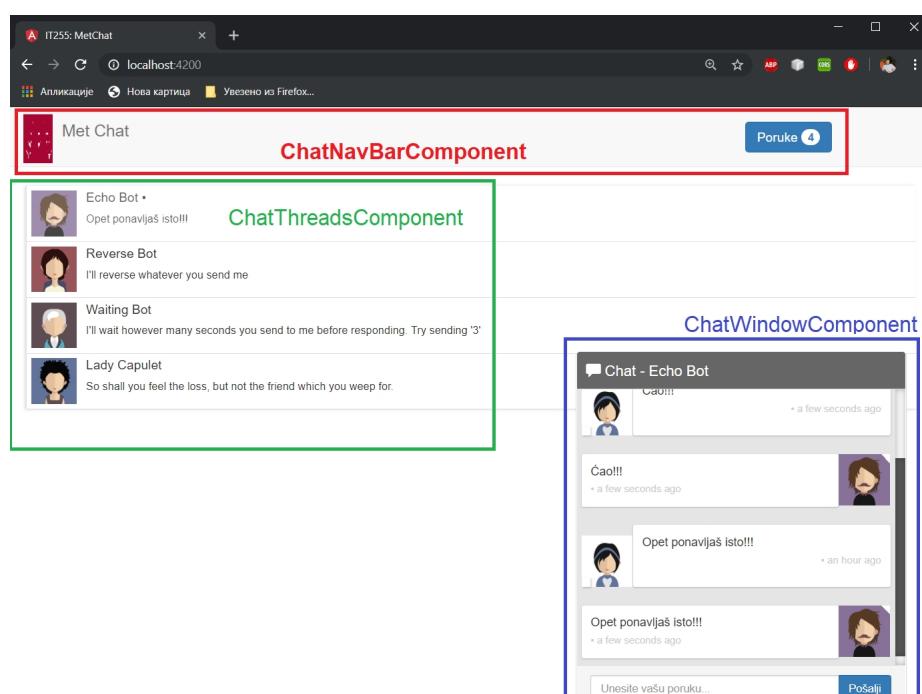
ARHITEKTURA GLAVNE KOMPONENTE APLIKACIJE APPCOMPONENT

Komponentu najvišeg nivoa AppComponent integriše tri komponente visokog nivoa.

Nakon što su kreirani servisi za čakanje i učinjena detaljna analiza i diskusija novog gradiva, neophodno je obaviti aktivnosti kompletiranja pokaznog primera kroz zadatke koji su, u ovom trenutku, studentima poznatiji - **implementacija komponenata i njihovih pogleda**.

U prvom koraku, pažnja će biti usmerena na komponentu najvišeg nivoa **AppComponent** koja integriše tri komponente visokog nivoa (slika 1):

- **ChatNavBarComponent** - prikazuje broj nepročitanih poruka;
- **ChatThreadsComponent** - sadrži listu korisničkih niti iz koje je moguće klikom obaviti izbor, najnoviju poruku i avatar konverzacije;
- **ChatWindowComponent** - sadrži poruke tekuće niti zajedno sa poljem za unos nove poruke i dugmetom za njeno slanje.



Slika 5.1.1 AppComponent koja integriše tri komponente visokog nivoa [izvor: autor]

KODIRANJE GLAVNE KOMPONENTE APLIKACIJE

Komponenta najvišeg nivoa u Angular aplikacijama je definisana klasom AppComponent.

Komponenta najvišeg nivoa u Angular aplikacijama je definisana klasom AppComponent. U ovom delu lekcije je neophodno dati njenu potpunu definiciju, a to je učinjeno sledećim listingom:

```
import { Component, Inject } from '@angular/core';
import { ChatExampleData } from './data/chat-example-data';

import { UsersService } from './user/users.service';
import { ThreadsService } from './thread/thread.service';
import { MessagesService } from './message/messages.service';

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent {
  constructor(public messagesService: MessagesService,
              public threadsService: ThreadsService,
              public usersService: UsersService) {
    ChatExampleData.init(messagesService, threadsService, usersService);
  }
}
```

Šta je učinjeno:

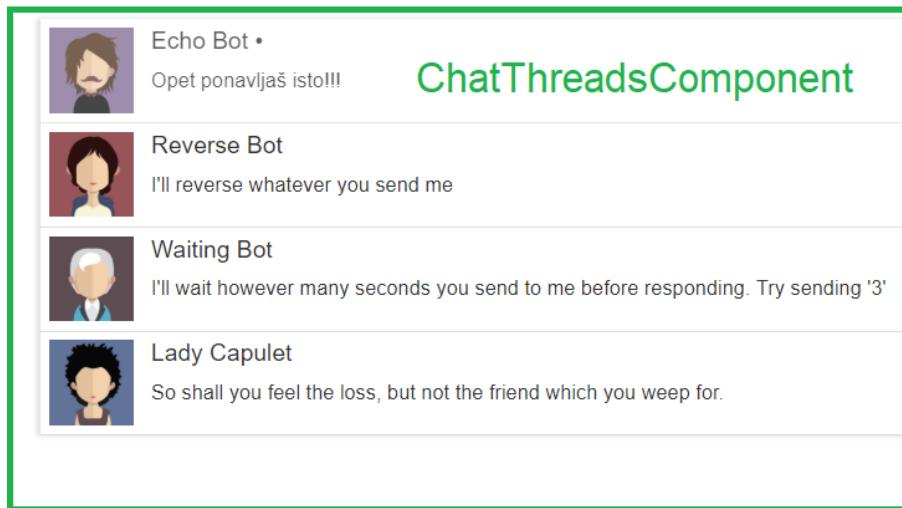
- selektorom je određen tag `<app-root>` za učitavanje šablona glavne komponente u stranicu `index.html`;
- sa `app.component.html` je definisana datoteka u kojoj će biti kodiran `HTML` šablon glavne komponente;
- unutar konstruktora je obavljeno *umetanje* (*injektovanje*) kreiranih servisa: `MessagesService`, `ThreadsService` i `UsersService`.

▼ 5.1 Kreiranje komponenata pogleda

KREIRANJE KOMPONENTE CHATTHREADSCOMPONENT

ChatThreadsComponent koja sadrži listu korisničkih niti, najnoviju poruku i avatar konverzacije.

Kao što je istaknuto u prethodnom uvodnom izlaganju, glavna [Angular komponenta AppComponent](#) je razbijena na tri celine koje je neophodno prodiskutovati i objasniti. Prva od njih je komponenta [ChatThreadsComponent](#) koja sadrži listu korisničkih niti iz koje je moguće klikom obaviti izbor, najnoviju poruku i avatar konverzacije. Izvršavanjem navedene komponente dobija se izlaz u formi pogleda sa sledeće slike:



Slika 5.2.1 Uređena lista korisničkih niti [izvor: autor]

Sledi listing klase komponente [ChatThreadsComponent](#):

```
import {  
  Component,  
  OnInit,  
  Inject  
} from '@angular/core';  
import { Observable } from 'rxjs';  
import { Thread } from '../thread/thread.model';  
import { ThreadsService } from '../../../thread/thread.service';  
  
@Component({  
  selector: 'chat-threads',  
  templateUrl: './chat-threads.component.html',  
  styleUrls: ['./chat-threads.component.css']  
})  
export class ChatThreadsComponent {  
  threads: Observable<any>;  
  
  constructor(public threadsService: ThreadsService) {  
    this.threads = threadsService.orderedThreads;  
  }  
}
```

Najveću pažnju je neophodno obratiti na konstruktor klase u kojem se vrši umetanje instance [ThreadService](#), a nakon toga čuvanje reference za [orderedThreads](#).

POGLED KOMPONENTE CHATTHREADSCOMPONENT

U pogledu komponente je implementiran kanal (pipe) za asinhronu obradu.

Sledećim listingom je dat veoma zanimljiv šablon komponente `ChatThreadsComponent` sa svojom konfiguracijom (datoteka: `/src/app/chat-threads/chat-threads.component.html`):

```
<!-- konverzacija -->
<div class="row">
  <div class="conversation-wrap">

    <chat-thread
      *ngFor="let thread of threads | async"
      [thread]="thread">
    </chat-thread>

  </div>
</div>
```

Ovde je potrebno обратити pažnju na tri stvari:

- Direktiva petlje `ngFor` sa asinhronim kanalom (`pipe`);
- `ChangeDetectionStrategy`;
- Primenu komponente `ChatThreadComponent`.

Petlja `ngFor` iterira preko liste korisničkih niti i prosleđuje ulaz `[thread]` komponenti `ChatThreadComponent`. Međutim, ovde je moguće primetiti nešto novo *kanal za asinhronu obradu (asinhrona cev, eng. async pipe)*. Asinhrona obrada je ovde implementirana preko `AsyncPipe` i omogućava upotrebu `RxJS Observable` na konkretnom mestu u pogledu. Ovako je moguće je koristiti `Observable` instance kao sinhronizovanu kolekciju.

Asinhrona cev se prijavljuje naObservable ili Promise i vraća poslednju emitovanu vrednost. Kada je nova vrednost emitovana, asinhrona cev obeležava komponentu koja se proverava za promene. Kada je komponenta uništена, asinhrona cev se automatski odjavljuje i na taj način se sprečava "curenje" memorije.

Za ovu komponentu je specificirana prilagođena (custom) primena sistema za detekciju (<https://blog.angular-university.io/how-does-angular-2-change-detection-really-work/>). `Angular` je fleksibilan i efikasan sistem za detekciju promena. Jedan od benefita bi bio ukoliko postoji komponenta sa nepromenljivim (`immutable`) ili osmatranim (`observable`) povezivanjem, moguće je sistemu za detekciju promena dati nagovještaj (`hint`) koji će učiniti da se aplikacija izvršava jako efikasno.

U konkretnom slučaju, umesto osmatranja promena na nizu objekata tipa `Thread`, Angular će se prijaviti za promene u osmatranom (`observable`) toku `threads` i pokrenuti ažuriranje kada se emituje nov događaj.

KOMPONENTA CHATTHREADCOMPONENT (JEDINSTVENA NIT)

Komponenta ChatThreadComponent će biti upotrebljena za prikazivanje pojedinačnih niti.

Razvoj i diskusija se nastavlja razvojem komponente *ChatThreadComponent*. Ova komponenta će biti upotrebljena za prikazivanje pojedinačnih korisničkih niti. Klasa komponente će biti kreirana u datoteci */src/app/chat-thread/chat-thread.component.ts* i biće definisana sledećim kodom:

```
import {  
  Component,  
  OnInit,  
  Input,  
  Output,  
  EventEmitter  
} from '@angular/core';  
import { Observable } from 'rxjs';  
import { ThreadsService } from './thread/thread.service';  
import { Thread } from '../thread/thread.model';  
  
@Component({  
  selector: 'chat-thread',  
  templateUrl: './chat-thread.component.html',  
  styleUrls: ['./chat-thread.component.css']  
})  
export class ChatThreadComponent implements OnInit {  
  @Input() thread: Thread;  
  selected = false;  
  
  constructor(public threadsService: ThreadsService) {  
  }  
  
  ngOnInit(): void {  
    this.threadsService.currentThread  
      .subscribe( (currentThread: Thread) => {  
        this.selected = currentThread &&  
          this.thread &&  
          (currentThread.id === this.thread.id);  
      });  
  }  
  
  clicked(event: any): void {  
    this.threadsService.setCurrentThread(this.thread);  
    event.preventDefault();  
  }  
}
```

Pre nego što se pažnja posveti šablonu komponente, neophodno je obraditi priloženi kod kontrolera. Kontroler implementira jedan nov interfejs pod nazivom *OnInit*. Angular

komponente mogu da istaknu da imaju sposobnost osluškivanja izvesnih događaja životnog ciklusa.

U ovom slučaju, klasa implementira interfejs *OnInit*, a to povlači da će metoda *ngOnInit()* biti pozvana kada ova komponenta prvi put bude proverena na promene. Ključni razlog primene ove metode leži u činjenici da *thread* osobina neće biti dostupna unutar konstruktora.

Iz listinga se, takođe, primećuje da je u metodi *ngOnInit()* obavljeno prijavljivanje na tok *threadsService.currentThread* i ukoliko *currentThread* odgovara osobini *thread* ove komponente, osobina *selected* dobija vrednost *true*, u suprotnom, dobija vrednost *false*.

Takođe, podešen je i obrađivač događaja *clicked*. Ovo je način rukovanja izborom tekuće korisničke niti. U šablonu (biće uskoro prikazan), povezuje se *clicked()* sa klikom na pogled korisničke niti. Ukoliko se primi *clicked()*

tada se javlja *threadsService* - u da se tekuća nit postavlja kao *Thread* osobina ove komponente.

KOMPONENTA CHATTHREADCOMPONENT - ŠABLON KOMPONENTE

*Upotrebљена veoma jednostavna povezivanja i direktiva *ngIf* koja prikazuje tačku za izabrano nit.*

Nakon opisanog kontrolera, sledi i diskusija u vezi sa šablonom komponente *ChatThreadComponent*. Sledi listing koji se čuva u datoteci *chat-thread.component.html*:

```
<div class="media conversation">
  <div class="pull-left">
    
    <h5 class="media-heading contact-name">{{thread.name}}</h5>
    <span *ngIf="selected">&bull;</span>
    <small class="message-preview">{{thread.lastMessage.text}}</small>
  </div>
  <a (click)="clicked($event)" class="div-link">Select</a>
</div>
```

Prvo što je moguće primetiti jeste da su upotrebljena veoma jednostavna povezivanja:

- *{{thread.avatarSrc}},*
- *{{thread.name}},*
- *{{thread.lastMessage.text}}.*

Zatim je upotrebljena direktiva **ngIf* koja omogućava prikazivanje simbola "•" ukoliko je data nit izabrana.

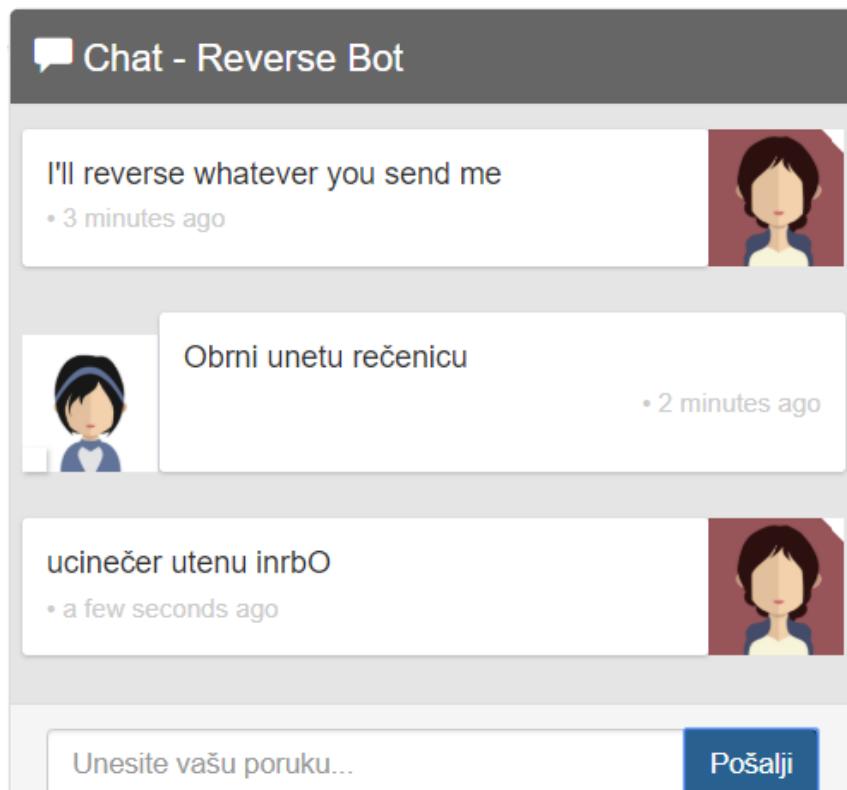
Na kraju, povezuje se događaj (*click*) sa pozivom obradivača događaja *clicked()*. Primećuje se da se prilikom ovog poziva prosleđuje argument *\$event*. Ovo je specijalna *Angular* varijabla kojom se opisuje događaj. U obradivaču događaja ova varijabla se koristi na sledeći način:

`event.preventDefault();`. Na ovaj način je obezbeđeno da se ne vrši navigacija na različitu stranicu.

KOMPONENTA CHATWINDOWCOMPONENT

Komponenta ChatWindowComponent je najsloženija komponenta aplikacije.

Komponenta `ChatWindowComponent` je najsloženija komponenta aplikacije i zato će biti obrađivana "deo po deo". Sledećom slikom je prikazan sadržaj pogleda generisanog primenom ove komponente.



Slika 5.2.2 Čet prozor [izvor: autor]

Za početak moguće je definisati dekorator `@Component` ove komponente:

```
@Component({
  selector: 'chat-window',
  templateUrl: './chat-window.component.html',
  styleUrls: ['./chat-window.component.css'],
  changeDetection: ChangeDetectionStrategy.OnPush
})
```

OSOBINE KLASE KOMPONENTE CHATWINDOWCOMPONENT

Klasa ChatWindowComponent poseduje 4 osobine.

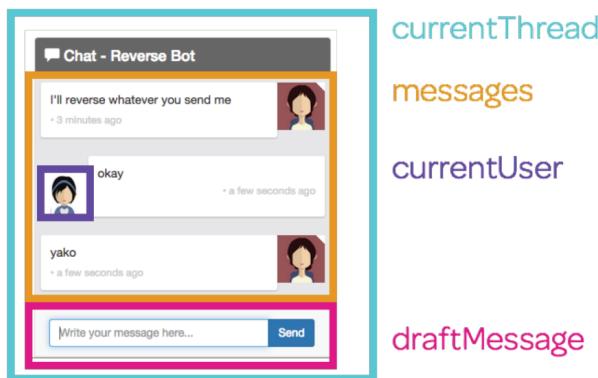
Klasa *ChatWindowComponent* poseduje 4 osobine:

- *messages*;
- *currentThread*;
- *draftMessage*;
- *currentUser*.

Deklaracija navedenih osobina je prikazana sledećim listingom:

```
export class ChatWindowComponent implements OnInit {  
  messages: Observable<any>;  
  currentThread: Thread;  
  draftMessage: Message;  
  currentUser: User;
```

Sledećom slikom je prikazana svaka od navedenih osobina:



Slika 5.2.3 Osobine klase ChatWindowComponent [izvor: autor]

Unutar konstruktora vrši se umetanje četiri vrednosti:

```
constructor(public messagesService: MessagesService,  
           public threadsService: ThreadsService,  
           public UsersService: UsersService,  
           public el: ElementRef) {  
}
```

Prve tri vrednosti predstavljaju servise. Četvrta vrednost, *el* je *ElementRef* instanca koja omogućava dobijanje pristupa konkretnom *host DOM elementu*. Ovo će biti korišćeno kada se obavi skrolovanje na dno čet prozora kada se kreira ili prima nova poruka.

CHATWINDOWCOMPONENT I METODA NGONINIT

*Inicijalizacija tekuće komponente će biti obavljena unutar metode *ngOnInit()*.*

Inicijalizacija tekuće komponente će biti obavljena unutar metode *ngOnInit()*. Glavna stvar koja će ovde biti obavljena je podešavanje prijave na *Observable* instance koje će menjati osobine komponente.

```
ngOnInit(): void {
    this.messages = this.threadsService.currentThreadMessages;

    this.draftMessage = new Message();
```

U prvom koraku, vrši se snimanje `currentThreadMessages` u osobinu `messages`. Zatim se kreira prazna poruka za podrazumevanu `draftMessage` vrednost.

Kada se pošalje nova poruka, neophodno je obezbediti da će taj Message objekat sačuvati referencu na nit za slanje, a to je uvek tekuća nit. Navedeno je obavljeno sledećom dopunom prethodnog listinga:

```
this.threadsService.currentThread.subscribe(
    (thread: Thread) => {
        this.currentThread = thread;
});
```

Takođe, neophodno je da novu poruku pošalje aktuelni korisnik pa će navedeno biti implementirano osobinom `currentUser`:

```
this.UserService.currentUser
    .subscribe(
        (user: User) => {
            this.currentUser = user;
});
```

CHATWINDOWCOMPONENT I FUNKCIJE SENDMESSAGE I ONENTER

Implementacijom funkcije `sendMessage()` omogućeno je slanje nove poruke.

Implementacijom funkcije `sendMessage()` omogućeno je slanje nove poruke:

```
sendMessage(): void {
    const m: Message = this.draftMessage;
    m.author = this.currentUser;
    m.thread = this.currentThread;
    m.isRead = true;
    this.messagesService.addMessage(m);
    this.draftMessage = new Message();
}
```

Funkcija `sendMessage()` uzima osobnu `draftMessage`, podešava autora i nit koristeći osobine klase `ChatWindowComponent`. Svaka poslata poruka odmah dobija status pročitane (tako se i obeležava).

U pogledu, poruka može biti poslata primenom nekog od sledeća dva scenarija:

- klikom na dugme "Pošalji";
- pritiskom na taster `Enter`.

Drugi scenario je omogućen implementacijom metode `onEnter()`:

```
onEnter(event: any): void {
    this.sendMessage();
    event.preventDefault();
}
```

SCROLLTOBOTTOM FUNKCIJA

Za slanje ili prijem nove poruke neophodno je obaviti skrolovanje na dno čet prozora

Kada je potrebno poslati, ili primiti novu poruku, neophodno je obaviti skrolovanje na dno čet prozora. Da bi ovo bilo moguće neophodno je obaviti podešavanje osobine `scrollTop` host elementa.

```
scrollToBottom(): void {
    const scrollPane: any = this.el
        .nativeElement.querySelector('.msg-container-base');
    scrollPane.scrollTop = scrollPane.scrollHeight;
}
```

Sada je neophodno se vratiti u metodu `ngOnInit()` i obaviti prijavljivanje na listu `currentThreadMessages` i obaviti skrolovanje na dno stranice svaki put kada stigne nova poruka.

```
this.messages
    .subscribe(
        (messages: Array<Message>) => {
            setTimeout(() => {
                this.scrollToBottom();
            });
        });
    }
```

ŠABLON KOMPONENTE CHATWINDOWCOMPONENT

Šablon komponente ChatWindowComponent bi trebalo da zaokruži razvoj primera.

Početak definicije šablona je poznat:

```
<div class="chat-window-container">
    <div class="chat-window">
        <div class="panel-container">
            <div class="panel panel-default">

                <div class="panel-heading top-bar">
```

```

<div class="panel-title-container">
  <h3 class="panel-title">
    <span class="glyphicon glyphicon-comment"></span>
    Chat - {{currentThread.name}}
  </h3>
</div>
<div class="panel-buttons-container">
  <!-- moguće je ovde dodati dugmiće za minimiziranje i zatvaranje -->
</div>
</div>

```

Zatim sledi lista sa porukama. Koristi se direktiva `ngFor` zajedno sa asinhronom cevi za iteraciju preko liste poruka. Komponenta određena tagom `chat-message` će biti uskoro objašnjena.

```

<div class="panel-body msg-container-base">
  <chat-message
    *ngFor="let message of messages | async"
    [message]="message">
  </chat-message>
</div>

```

Konačno, dodaju se polje za unos nove poruke, dugme za slanje i tagovi za kraj koda.

```

<div class="panel-footer">
  <div class="input-group">
    <input type="text"
      class="chat-input"
      placeholder="Unesite vašu poruku..."
      (keydown.enter)="onEnter($event)"
      [(ngModel)]="draftMessage.text" />
    <span class="input-group-btn">
      <button class="btn-chat"
        (click)="onEnter($event)">Pošalji</button>
    </span>
  </div>
</div>
</div>

```

Od posebnog značaja je poziv: `[(ngModel)]="draftMessage.text"`. Angular, u osnovi ne podržava dvosmerno povezivanje. Međutim, moglo bi biti veoma korisno obezbediti ovakav vid povezivanja između komponente i njenog pogleda, sve dok su sporedni efekti lokalni za komponentu, za sinhronizovanje osobina komponente sa pogledom.

U ovom slučaju, uspostavlja se dvosmerna veza između vrednosti `input` taga i vrednosti `draftMessage.text`. To znači, ukoliko se obavi unos u `input` tag, `draftMessage.text` će automatski biti podešen na tu vrednost. Važi i obrnuto, ukoliko se u kodu ažurira `draftMessage.text`, promeniće se i vrednost `input` taga u pogledu.

CHATWINDOWCOMPONENT KAO CELINA

Za objedinjenu sliku ove komponente, neophodno je prikazivanje njenih listinga u celosti.

Za objedinjenu sliku ove komponente, neophodno je prikazivanje njenih listinga u celosti.

Klasa komponente *ChatWindowComponent*:

```
import {  
  Component,  
  Inject,  
  ElementRef,  
  OnInit,  
  ChangeDetectionStrategy  
} from '@angular/core';  
import { Observable } from 'rxjs';  
  
import { User } from '../user/user.model';  
import { UsersService } from '../user/users.service';  
import { Thread } from '../thread/thread.model';  
import { ThreadsService } from '../thread/threads.service';  
import { Message } from '../message/message.model';  
import { MessagesService } from '../message/messages.service';  
  
@Component({  
  selector: 'chat-window',  
  templateUrl: './chat-window.component.html',  
  styleUrls: ['./chat-window.component.css'],  
  changeDetection: ChangeDetectionStrategy.OnPush  
})  
export class ChatWindowComponent implements OnInit {  
  messages: Observable<any>;  
  currentThread: Thread;  
  draftMessage: Message;  
  currentUser: User;  
  
  constructor(public messagesService: MessagesService,  
              public threadsService: ThreadsService,  
              public UsersService: UsersService,  
              public el: ElementRef) {  
  }  
  
  ngOnInit(): void {  
    this.messages = this.threadsService.currentThreadMessages;  
  
    this.draftMessage = new Message();  
  
    this.threadsService.currentThread.subscribe(  
      (thread: Thread) => {  
        this.currentThread = thread;  
      }  
    );  
  }  
}
```

```

    });

    this.UserService.currentUser
        .subscribe(
            (user: User) => {
                this.currentUser = user;
            });
}

this.messages
    .subscribe(
        (messages: Array<Message>) => {
            setTimeout(() => {
                this.scrollToBottom();
            });
        });
}

onEnter(event: any): void {
    this.sendMessage();
    event.preventDefault();
}

sendMessage(): void {
    const m: Message = this.draftMessage;
    m.author = this.currentUser;
    m.thread = this.currentThread;
    m.isRead = true;
    this.messagesService.addMessage(m);
    this.draftMessage = new Message();
}

scrollToBottom(): void {
    const scrollPane: any = this.el
        .nativeElement.querySelector('.msg-container-base');
    scrollPane.scrollTop = scrollPane.scrollHeight;
}
}
}

```

Šablon komponente *ChatWindowComponent* :

```

<div class="chat-window-container">
    <div class="chat-window">
        <div class="panel-container">
            <div class="panel panel-default">

                <div class="panel-heading top-bar">
                    <div class="panel-title-container">
                        <h3 class="panel-title">
                            <span class="glyphicon glyphicon-comment"></span>
                            Chat - {{currentThread.name}}
                        </h3>
                    </div>
                <div class="panel-buttons-container">

```

```
<!-- moguće je ovde dodati dugmiće za minimiziranje i zatvaranje -->
</div>
</div>

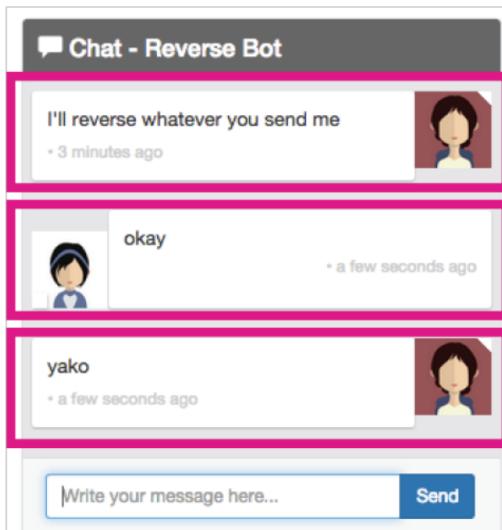
<div class="panel-body msg-container-base">
  <chat-message
    *ngFor="let message of messages | async"
    [message]="message">
  </chat-message>
</div>

<div class="panel-footer">
  <div class="input-group">
    <input type="text"
      class="chat-input"
      placeholder="Unesite vašu poruku..."
      (keydown.enter)="onEnter($event)"
      [(ngModel)]="draftMessage.text" />
    <span class="input-group-btn">
      <button class="btn-chat"
        (click)="onEnter($event)">Pošalji</button>
    </span>
  </div>
</div>
</div>
```

KONTROLER KOMPONENTE CHATMESSAGECOMPONENT

Svaka poruka je prikazana primenom komponente ChatMessageComponent.

Svaka poruka je prikazana primenom komponente *ChatMessageComponent*.



ChatMessage

ChatMessage

ChatMessage

Slika 5.2.4 Komponenta ChatMessageComponent [izvor: autor]

Ova komponenta je relativno jednostavna. Osnovna logika kreira blago drugačiji pogled u zavisnosti da li je poruka kreirana od strane tekućeg korisnika. Ukoliko poruku nije kreirao aktuelni korisnik, poruka se smatra dolaznom.

Takođe, trebalo bi imati na umu da svaki objekat *ChatMessageComponent* pripada jednom *Message* objektu. Zbog navedenog u metodi *ngOnInit()* biće obavljeno prijavljivanje na tok *currentUser* i podešavanje da dolaznu poruku predstavlja *Message* objekat kojeg nije kreirao tekući korisnik.

Još jedna stvar, pre prilaganja koda kontrolera ove komponente, u konstruktoru je obavljeno umetanje servisa *UserService*.

```
import {
  Component,
  OnInit,
  Input
} from '@angular/core';
import { Observable } from 'rxjs';

import { UserService } from './user/users.service';
import { ThreadsService } from './thread/threads.service';
import { MessagesService } from './message/messages.service';

import { Message } from './message/message.model';
import { Thread } from './thread/thread.model';
import { User } from './user/user.model';

@Component({
  selector: 'chat-message',
  templateUrl: './chat-message.component.html',
  styleUrls: ['./chat-message.component.css']
})
export class ChatMessageComponent implements OnInit {
  @Input() message: Message;
```

```
currentUser: User;
incoming: boolean;

constructor(public UsersService: UsersService) {
}

ngOnInit(): void {
    this.UsersService.currentUser
        .subscribe(
            (user: User) => {
                this.currentUser = user;
                if (this.message.author && user) {
                    this.incoming = this.message.author.id !== user.id;
                }
            });
}
}
```

ŠABLON KOMPONENTE A CHATMESSAGECOMPONENT

Šablon implementira dve interesantne ideje: FromNowPipe i [ngClass].

Šablon implementira dve interesantne ideje:

- *FromNowPipe* i
- *[ngClass]*.

Navedene ideje su implementirane sledećim kodom:

```
<div class="msg-container"
  [ngClass]="{'base-sent': !incoming, 'base-receive': incoming}>

  <div class="avatar"
    *ngIf="!incoming">
      
    <p>{{message.text}}</p>
    <p class="time">{{message.sender}} • {{message.sentAt | fromNow}}</p>
  </div>

  <div class="avatar"
    *ngIf="incoming">
      ).

Takođe u pogledu postoji značajna primena **ngClass**. Ideja je kada se pozove izraz:

```
[ngClass]="{'msg-sent': !incoming, 'msg-receive': incoming}"
```

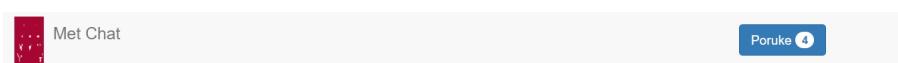
da se ukaže Angularu - u da primeni **msg-receive** klasu ukoliko parametar **incoming** ima vrednost **true**, a u suprotnom da primeni klasu **msg-sent**.

Primenom osobine **incoming** moguće je prikazivanje odlaznih i dolaznih poruka na različit način.

## KONTROLER KOMPONENTE CHATNAVBARCOMPONENT

*Poslednja komponenta aplikacije prikazuje broj nepročitanih poruka.*

Poslednja komponenta aplikacije prikazuje broj nepročitanih poruka. Izvršavanje koda komponente **ChatNavBarComponent** daje izlaz kao na sledećoj slici.



Slika 5.2.6 ChatNavBarComponent komponenta

Jedina stvar o kojoj ova komponenta mora da vodi računa jeste brojanje nepročitanih poruka i čuvanje preko **unreadMessagesCount**. Ovo je malo komplikovanije nego što izgleda na prvi pogled.

Najjednostavniji način bi bio jednostavno osluškivanje toka **messagesService.messages** i sabiranje broja **Message** objekata čija osobina **isRead** ima vrednost **false**. Ovo funkcioniše odlično za sve poruke izvan tekuće niti. Međutim, ne postoji garancije da će nove poruke u tekućoj niti biti označene kao pročitane u vreme emitovanja nove vrednosti.

Najbezbedniji način za rukovanje navedenim problemom jeste kombinovanje tokova *messages* i *currentThread* sa vođenjem računa da se ne broje poruke koje su deo tekuće niti. Ovo se postiže primenom operatora *combineLatest* koji je objašnjen u nekom od prethodnih izlaganja.

Nakon pojašnjenja sada je moguće priložiti listing kontrolera komponente.

```
import {
 Component,
 Inject,
 OnInit
} from '@angular/core';
import * as _ from 'lodash';

import { ThreadsService } from './thread/thread.service';
import { MessagesService } from './message/message.service';

import { Thread } from './thread/thread.model';
import { Message } from './message/message.model';

@Component({
 selector: 'chat-nav-bar',
 templateUrl: './chat-nav-bar.component.html',
 styleUrls: ['./chat-nav-bar.component.css']
})
export class ChatNavBarComponent implements OnInit {
 unreadMessagesCount: number;

 constructor(public messagesService: MessagesService,
 public threadsService: ThreadsService) {
 }

 ngOnInit(): void {
 this.messagesService.messages
 .combineLatest(
 this.threadsService.currentThread,
 (messages: Message[], currentThread: Thread) =>
 [currentThread, messages])

 .subscribe(([currentThread, messages]: [Thread, Message[]]) => {
 this.unreadMessagesCount =
 _.reduce(
 messages,
 (sum: number, m: Message) => {
 const messageIsInCurrentThread: boolean = m.thread &&
 currentThread &&
 (currentThread.id === m.thread.id);
 // u "realnoj" aplikaciji bi takođe trebalo isključiti
 // poruke odobrene od aktuelnog korisnika
 // one su već pročitane
 if (m && !m.isRead && !messageIsInCurrentThread) {
 sum = sum + 1;
 }
 })
 })
 }
}
```

```
 return sum;
 },
 0);
});
}
```

Pozivom povratne funkcije `combineLatest()` vraća se niz sa dva elementa `currentThread` i `messages`. Zatim se obavlja prijavljivanje na ove tokove i destrukcija ovih objekata u pozivu funkcije. Zatim se vrši redukcija (linija koda 35) broja poruka kandidata prebrojavanjem broja nepročitanih poruka koje ne pripadaju tekućoj niti.

## ŠABLON KOMPONENTE CHATNAVBARCOMPONENT

*Jedina stvar koju pogled prikazuje je broj nepročitanih poruka.*

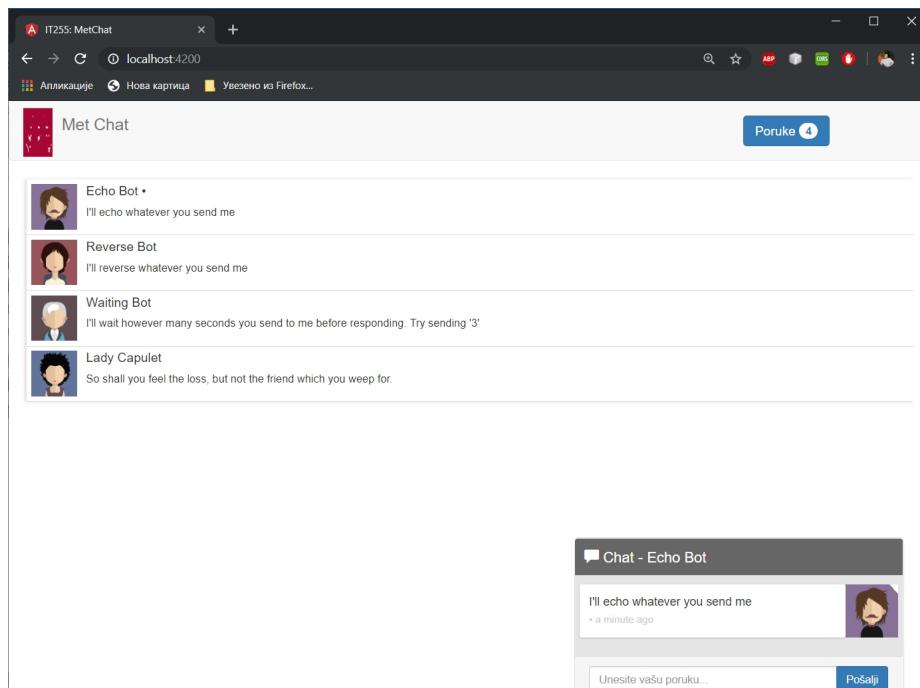
Jedina stvar koju pogled prikazuje je broj nepročitanih poruka i to je prikazano sledećim listingom.

```
<nav class="navbar navbar-default">
 <div class="container-fluid">
 <div class="navbar-header">

 Met Chat

 </div>
 <p class="navbar-text navbar-right">
 <button class="btn btn-primary" type="button">
 Poruke {{ unreadMessagesCount }}
 </button>
 </p>
 </div>
</nav>
```

Konačan izgled aplikacije, učitane primenom veb pregledača, prikazan je sledećom slikom.



Slika 5.2.7 Konačan izgled aplikacije [izvor: autor]

**Potpuno urađen primer možete preuzeti odmah nakon ovog objekta učenja.**

## VIDEO MATERIJALI

*Izlaganje lekcije biće zaokruženo prilaganjem adekvatnih video materijala.*

**OBSERVABLES, OBSERVERS & SUBSCRIPTIONS | RxJS TUTORIAL** - trajanje 17:39.

**Ova lekcija sadrži video materijal. Ukoliko želite da pogledate ovaj video morate da otvorite LAMS lekciju.**

**Learn RxJS in 60 Minutes for Beginners - Free Crash Course** - trajanje 59:49.

**Ova lekcija sadrži video materijal. Ukoliko želite da pogledate ovaj video morate da otvorite LAMS lekciju.**

## ✓ Poglavlje 6

### Dodatni materijali za rad

#### DODATNI MATERIJALI

*Proširite vaše znanje*

1. <https://facebook.github.io/flux/>
2. <https://github.com/Reactive-Extensions/RxJS>
3. <https://github.com/Reactive-Extensions/RxJS/blob/master/doc/api/core/observable.md>
4. <https://github.com/Reactive-Extensions/RxJS/blob/master/doc/api/core/observer.md>
5. <https://blog.angular-university.io/how-does-angular-2-change-detection-really-work/>
6. <https://momentjs.com/>

## ✓ Poglavlje 7

### Pokazna Vežba 11

#### POKAZNA VEŽBA - KREIRANJE PROJEKTA IT255-V12 (VREME 45 MINUTA)

*Kroz ovu vežbu obradićemo mapiranje podataka: servis - niz modela.*

Kroz ovu vežbu obradićemo **mapiranje** podataka, koji će biti preuzet iz određenog veb servisa, sa prethodno kreiranim nizom modela. Za potrebe ove vežbe, nastavljamo sa projektnom iz prethodne lekcije.

Kreiraćemo novu komponentu pod nazivom **rxjs**, navođenjem sledeće komande u okviru postojećeg projekta:

```
ng generate component rxjs
```

Budući da ćemo u ovoj komponenti koristiti i modele, kao i servise, kreirajmo još par datoteka za aktuelni projekat na sledeći način:

```
ng g service services/cat
```

```
ng g class models/cat
```

Kako bismo malo stilizovali našu aplikaciju, učitaćemo **bootstrap** - ov **CSS** fajl unutar naše aplikacije, tako što ćemo dodati sledeće parče koda u globalnu datoteku stilova **styles.css**:

```
@import '~bootstrap/dist/css/bootstrap.min.css';
```

Ukoliko već to nije urađeno, za aktuelni projekat će biti potrebno instalirati **Bootstrap** na sledeći način:

```
npm install bootstrap --save
```

Još malo početnih podešavanja. Kako bismo mogli da obavljamo **HTTP** pozive, neophodno je da uključimo **HttpClientModule**, unutar **imports** sekciјe na nivou čitave aplikacije kroz **app.module.ts**. Potrebno da je prvo uvezemo biblioteku iz **Angular** - a na sledeći način, ukoliko ona nije importovana tokom prethodnog rada:

```
import { HttpClientModule } from '@angular/common/http';
```

A potom da istu uključimo unutar **Imports** dela na sledeći način:

```
import { BrowserModule } from '@angular/platform-browser';
import { NgModule } from '@angular/core';

import { AppRoutingModule } from './app-routing.module';
import { AppComponent } from './app.component';
import { HttpClientModule } from '@angular/common/http';

@NgModule({
 declarations: [
 AppComponent
],
 imports: [
 BrowserModule,
 HttpClientModule,
 AppRoutingModule
],
 providers: [],
 bootstrap: [AppComponent]
})
export class AppModule { }
```

Više informacija o tome se može naći na sledećem linku: <https://angular.io/guide/http>.

## PRIMENA SERVISA

*Za potrebe ove vežbe, koristićemo već dostupni REST servis pod nazivom JSONPlaceholder.*

Za potrebe ove vežbe, koristićemo već dostupni *REST* servis koji će nam omogućiti da istestiramo *HTTP* pozive kroz *Angular*.

```
ng g service services/cat
```

Unutar tog servisa, kroz konstruktor neophodno je da inicijalizujemo *HttpClient*-a na sledeći način:

```
constructor(private _httpClient: HttpClient) { }
```

A pre toga je potrebno da uvezemo biblioteku na sledeći način:

```
import { HttpClient } from '@angular/common/http';
```

Servisna klasa bi trebalo da omogući preuzimanje objekata (koji se odnose na domaću životinju mačku) grupno ili pojedinačno primenom dveju *GET* metoda. Finalni kod servisne klase izgleda ovako:

```
import { Injectable } from '@angular/core';
import { Observable } from 'rxjs';
import { of } from 'rxjs';
```

```
import {Cat} from '../model/cat'
import { HttpClient, HttpHeaders } from '@angular/common/http';

@Injectable()
export class CatService {

 constructor(private http: HttpClient) { }
 //ovde možete da koristite bilo koji URL
 private lambdaCats = 'https://ast9z666ll.execute-api.us-east-2.amazonaws.com/
dev/cats'

 getCats(): Observable<Cat[]> {
 return this.http.get<Cat[]>(this.lambdaCats)
 }

 getCat(id: number): Observable<Cat> {
 const url = `${this.lambdaCats}/${id}`
 return this.http.get<Cat>(url)
 }
}
```

Ovde možemo da primetimo kreiranje dva *Observable* objekta za tokove koji vraćaju rezultate poziva GET metoda.

## PRIMENA MODELA

*U sledećem koraku, kreiraćemo model pod nazivom Cat.*

Kako bismo ispoštovali standarde prethodno definisane u vežbama, kreiraćemo model pod nazivom *Cat* sledećom komandom:

```
ng g class models/cat
```

Izgled *JSON* - a je dat na slici ispod:

```
{
 "id": "id",
 "name": "ime mačke",
 "pictureURL": "veb link ka slici"
}
```

Slede konačni listing modelske klase:

### Cat.ts

```
export class Cat {
 id: string;
 name: string;
```

```
pictureuRL: string;

constructor(id: string, name: string, pictureuRL: string) {
 this.id = id;
 this.name = name;
 this.pictureuRL = pictureuRL;
}

}
```

## KOMPONENTE APLIKACIJE

*Na nivou app.component.html fajla vršićemo učitavanje svih cat objekata.*

Kreirana komponenta biće dostupna putem nove rute u aktuelnom projektu. Pre nego što definišemo novu rutu, uvešćemo šablon i klasu nove komponente u projekat.

Sledećim listingom je prikazan šablon ove komponente:

```
<div class="col-md-4" *ngFor="let cat of cats; index as i">
 <div class="card w-100">
 <div class="card-body">
 <h5 class="card-title">{{cat?.id}}</h5>
 {{cat?.name}}</p>
 </div>
 </div>
</div>
```

Sledećim listingom je prikazana je klasa ove komponente:

```
import { Component, OnInit, Input } from '@angular/core';
import { Cat } from 'src/app/model/cat';
import { CatService } from '../services/cat.service';

@Component({
 selector: 'app-single-cat',
 templateUrl: './rxjs.component.html',
 styleUrls: ['./rxjs.component.css']
})
export class RxjsComponent implements OnInit {

 cats: Cat[];

 constructor(private catService: CatService) { }
```

```
ngOnInit() {
 this.getCats();
}

getCats(): void {
 this.catService.getCats()
 .subscribe(cats => this.cats = cats);
}
}
```

Ovde možemo da primetimo prijavljivanje dva *Observable* objekta na tokove koji vraćaju rezultate poziva *GET* metoda.

## KREIRANJE RUTE

### *Modifikacija roditeljske komponente*

Sledi modifikacija roditeljske komponente sa ciljem primene reaktivnog programiranja, odnosno prethodno urađenog posla.

Za početak u šablon *app.component* se dodaje nova ruta (linija koda 24):

```
@Component({
 selector: 'my-app',
 template: `
 <app-navbarcomponentnew></app-navbarcomponentnew>

 <div class="page-header">
 <div class="container">
 <h1>{{pageheader}}</h1>
 <h2>Rutiranje - primer</h2>
 <div class="navLinks">
 <a [routerLink]=["/student"]>Komponenta student
 ||
 <a [routerLink]=["/buttongroup"]>ButtonGroup
Komponenta
 ||
 <a [routerLink]=["/forma"]>Forma
 ||
 <a [routerLink]=["/di"]>Angular DI
 ||
 <a [routerLink]=["/http"]> Angular HTTP
 ||
 <a [routerLink]=["/rxjs"]> Observable i
operatori
 </div>
 </div>
 </div>
 `
```

```
</div>
<div id="content">
 <div class="container">
 <router-outlet></router-outlet>
 </div>
</div>

`
```

Za istu komponentu modifikujemo konstruktor sa ciljem umetanja i upotrebe kreiranog servisa:

```
export class AppComponent {
 pageheader: string = "Uvodimo primer sa rutiranjem"

 public cats: Cat[];
 constructor(private _catService: CatService) {
 this._catService.getCats().subscribe((data) => {
 this.cats = data;
 })
 }
}
```

## FINALNO PODEŠAVANJE APLIKACIJE

*Finalno podešavanje aplikacije sa ciljem modifikacije rutera i dodavanja nove DI komponente*

Finalno podešavanje aplikacije sa ciljem modifikacije rutera i dodavanja nove DI komponente započinje proširenjem postojećeg rutera , kao u 9. liniji koda ([app.module.ts](#)):

```
const routes: Routes = [
 // osnovne rute
 { path: '', redirectTo: 'student', pathMatch: 'full' },
 { path: 'student', component: StudentComponent },
 { path: 'buttongroup', component: ButtongroupcomponentComponent },
 { path: 'forma', component: FormaComponent },
 { path: 'di', component: DIComponent },
 { path: 'http', component: HttpComponent },
 { path: 'rxjs', component: RxjsComponent }

];
```

Dalje, kreiranjem komponente putem [Angular CLI](#), komponenta [rxjs.component](#) je automatski registrovana u listi komponenata aplikacije (linija koda broj 10):

```
@NgModule({
 declarations: [
```

```
AppComponent,
StudentComponent,
NavbarcomponentnewComponent,
ButtongroupcomponentComponent,
FormaComponent,
DIComponent,
HttpComponent,
RxjsComponent,

],
```

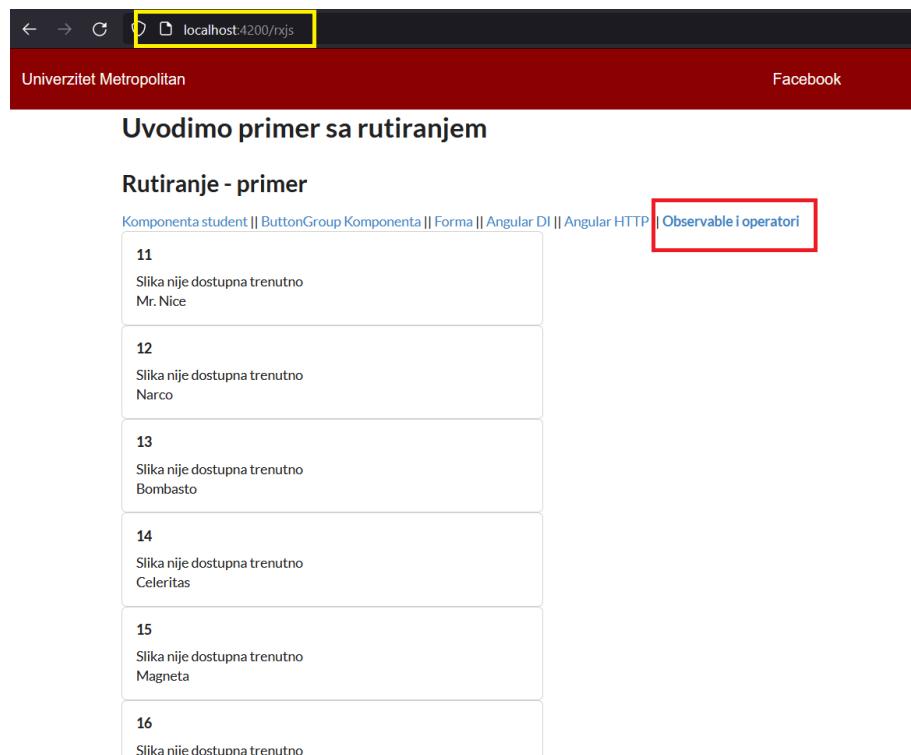
Konačno, kreirani servis je uključen u [Angular DI](#) na sledeći način (linija koda broj 4):

```
imports: [
 BrowserModule, ReactiveFormsModule, HttpClientModule,
 RouterModule.forRoot(routes)
,
 providers: [Studije, MetService, CatService],
 bootstrap: [AppComponent]
}
export class AppModule { }
```

## IT255 – V12 (DEMONSTRACIJA)

*Demonstacija konačnog izgleda aplikacije.*

Finalni izgled aplikacije prikazan je na slici ispod:



Slika 7.1 Demonstacija konačnog izgleda aplikacije [izvor: autor]

**Potpuno urađen primer možete preuzeti odmah nakon ovog objekta učenja.**

## ✓ Poglavlje 8

### Individualna vežba 12

#### INDIVIDUALNA VEŽBA (TRAJANJE 90 MINUTA)

*Samostalno vežbanje rada sa servisima i pogledima na osnovu obrađenih materijala predavanja i pokazne vežbe.*

Koristeći znanje stečeno na predavanjima i vežbama pokušajte samostalno da uredite Angular veb aplikaciju na osnovu sledećih zahteva:

- pronađite neki otvoren servis;
- integrirajte `rxjs` biblioteku za dovlačenje entiteta;
- kao ilustraciju koristite sledeći primer: <https://codecraft.tv/courses/angular/http/http-with-observables/>

Napišite izveštaj u kojem opisujete sve preuzete korake u ovom primeru!!!

Nakon urađene vežbe pozovite predmetnog asistenta i demonstrirajte funkcionisanje uvedenih funkcionalnosti u aplikaciju.

## ✓ Poglavlje 9

### Domaći zadatak 12

#### DOMAĆI ZADATAK (PREDVIĐENO VREME 120 MIN)

*Samostalna izrada domaćeg zadatka sa servisima i pogledima na osnovu obrađenih materijala predavanja i vežbi.*

Uradite domaći zadatka prema sledećim zahtevima:

- Pronaći domaći zadatak *MetHotels* iz prethodnih vežbi i domaćih zadataka;
- Integrисати pregled soba kroz poglede (servis za pregled soba).
- Takođe, neophodno je pratiti sve standarde tako da **svi entiteti moraju biti predstavljeni kroz modele.**

Domaći zadatak dodati na *Github* pod "commit - om" *IT255-DZ12* i poslati obaveštenje predmetnom asistentu o postavljenom domaćem zadatku.

## ▼ Poglavlje 10

### Zaključak

## ZAKLJUČAK

*Lekcija je identificovala nekoliko ključnih putanja arhitekture podataka u veb aplikacijama.*

Lekcija je diskutovala da upravljanje podacima može biti jedan od najzahtevnijih aspekata pisanja održivih aplikacija. Postoje brojni načini pribavljanja podataka u veb aplikacijama što je jasno istaknuto u uvodnom razmatranju lekcije.

Međutim, problem koji se odnosi na *arhitekturu podataka* moguće je razdvojiti u više pitanja, na koje je lekcija morala da da odgovore:

- Kako je moguće povezati sve navedene različite izvore u koherentan sistem?
- Kako je moguće izbeći greške izazvane neočekivanim efektima sa strane?
- Kako je moguće strukturirati kod na način da je lakši za održavanje i uključivanje novih članova softverskog razvojnog tima;
- Kako je moguće obezbediti da se aplikacija izvršava na najbrži mogući način kada dođe do izmene u podacima.

Primećeno je da bojne godine unazad *MVC* je bio dominantan šablon organizovanja podataka u aplikacijama.

Međutim, primećuje se da, iz perspektive klijent usmerenih radnih okvira, a Angular je jedan od njih, *MVC* arhitektura se ne prevodi dovoljno dobro u veb aplikacije klijentske strane.

Da bi navedeni nedostatak bio prevaziđen, u skorije vreme je nastala prava renesansa u ovoj oblasti, u formi arhitekturnih pristupa:

- *MVW (Model-View-Whatever);*
- *Flux;*
- *Observables.*

Poslednji pristup je bio u fokusu lekcije sa ciljem davanja odgovora na postavljena pitanja.

Student je sada sposobljen da razume, kreira i koristi servise u *Angular* veb aplikacijama, kao i da kreira i koristi *poglede* u Angular veb aplikacijama na način opisan u lekciji primenom konkretnog šablona arhitekture podataka. U tom svetlu, lekcija je i podeljena na dve celine.

## LITERATURA

*Za pripremu lekcije korišćena je aktuelna pisana i elektronska literatura.*

### **Pisana literatura:**

1. Nate Murray, Felipe Coury, Ari Lerner, Carlos Taborda, ng-Book – The Complete Book on Angular 6, Fullstack.io, 2018
2. Cody Lindley, Frontend Handbook – 2017, Frontend masters, 2017
3. Sandeep Panda, AngularJS – From Novice to Ninja, SitePoint Pty. Ltd, 2014

### **Elektronska literatura:**

4. <https://angular.io/>
5. <https://angular.io/tutorial>
6. <https://www.w3schools.com/angular/>
7. <https://www.tutorialspoint.com/angular4/>
8. <https://nodejs.org/en/>
9. <https://code.visualstudio.com/>
10. <https://facebook.github.io/flux/>
12. <https://github.com/Reactive-Extensions/RxJS>
13. <https://github.com/Reactive-Extensions/RxJS/blob/master/doc/api/core/observable.md>
14. <https://github.com/Reactive-Extensions/RxJS/blob/master/doc/api/core/observer.md>
15. <https://blog.angular-university.io/how-does-angular-2-change-detection-really-work/>
16. <https://momentjs.com/>



IT255 - VEB SISTEMI 1

## TypeScript, Angular i redukcija aplikacije

Lekcija 13

PRIRUČNIK ZA STUDENTE

# IT255 - VEB SISTEMI 1

## Lekcija 13

### ***TYPESCRIPT, ANGULAR I REDUKCIJA APLIKACIJE***

- ✓ TypeScript, Angular i redukcija aplikacije
- ✓ Poglavlje 1: Redux - ključne ideje
- ✓ Poglavlje 2: Skladište stanja
- ✓ Poglavlje 3: Implementacija reduktora u aplikaciji za časovanje
- ✓ Poglavlje 4: Redux u Angular okviru
- ✓ Poglavlje 5: Dodatni materijali za rad
- ✓ Poglavlje 6: Vežba 13
- ✓ Poglavlje 7: Individualna vežba 13
- ✓ Poglavlje 8: Domaći zadatak 13
- ✓ Zaključak

Copyright © 2017 – UNIVERZITET METROPOLITAN, Beograd. Sva prava zadržana. Bez prethodne pismene dozvole od strane Univerziteta METROPOLITAN zabranjena je reprodukcija, transfer, distribucija ili memorisanje nekog dela ili čitavih sadržaja ovog dokumenta., kopiranjem, snimanjem, elektronskim putem, skeniranjem ili na bilo koji drugi način.

Copyright © 2017 BELGRADE METROPOLITAN UNIVERSITY. All rights reserved. No part of this publication may be reproduced, stored in a retrieval system or transmitted in any form or by any means, electronic, mechanical, photocopying, recording, scanning or otherwise, without the prior written permission of Belgrade Metropolitan University.

## ✓ Uvod

### UVOD

*U ovoj lekciji akcenat će biti na izučavanju šablonu arhitekture podataka pod nazivom Redux.*

U ovoj lekciji akcenat će biti na izučavanju šablonu arhitekture podataka pod nazivom *Redux*. Prva razmatranja će biti vezana za diskusiju o idejama koje stoje iza primene *Redux*-a, pisanje mini verzije aplikacije, a zatim njeno povezivanje sa *Angular* okvirom.

U većini dosadašnjih projekata, upravljanje stanjima je obavljano na prilično direktni način:

- uzimaju se podaci iz servisa;
- podaci se obrađuju u komponentama i prosleđuju kroz stablo komponenata.

Ovakav vid upravljanja aplikacijom može biti odličan za male aplikacije, ali kako aplikacije rastu, dobijaju sve veći broj komponenata, upravljanje različitim stanjima aplikacije može biti veoma složen i težak posao.

Na primer, prosleđivanje podataka niz stablo komponenata može biti praćeno sledećim problemima:

- *posredno prosleđivanje osobina* - Sa ciljem dobijanja stanja bilo koje komponente neophodno je obaviti prosleđivanje vrednosti niz stablo komponenata preko ulaza (input). To znači da se na tom putu može naći više posredničkih komponenata koje prosleđuju stanje koje se ne koristi direktno ili nije bitno;
- *nefleksibilni refactoring* - Iz razloga što se prosleđuju ulazi niz stablo komponenata, postoji povezanost između roditeljskih i komponenata potomaka koja je često nepotrebna. Na ovaj način je otežano premeštanje potomak komponente na neko drugo mesto u hijerarhiji jer bi bilo potrebno učiniti izmene nad svim roditeljskim komponentama za prosleđivanje stanja.
- *stablo stanja (state tree) i DOM stablo (DOM tree) se ne podudaraju* - oblik stanja često ne odgovara obliku pogled / komponenta hijerarhije. Prosleđivanjem svih podataka kroz stablo komponenata, preko osobina, ulazi se u nov problem - referenciranje podataka u udaljenim granama stabla.
- *stanje aplikacije u celini* - ukoliko se upravlja stanjem preko komponenata, teško je sagledati stanje aplikacije u celini. Ovo može otežati saznanje koja komponenta „poseduje“ određeni deo podataka i na koje komponente promene utiču.

Manipulisanje podacima primenom servisa i komponenta pomaže puno u razvoju veb aplikacija. Ako se servisi sagledaju kao vlasnici podataka, postavlja se novo pitanje: "koja je najbolja praksa za podatke u vlasništvu servisa? Da li postoji pogodan šablon za implementaciju rešenja?" Odgovor je: "Da, postoji!" Upravo u ovoj lekciji će biti akcenat

na šablonu arhitekture podataka poznatog pod nazivom *Redux* koji je od velike pomoći za rešavanje navedenog problema.

## UVODNI VIDEO

*Trajanje video snimka: 5min 18sek*

**Ova lekcija sadrži video materijal. Ukoliko želite da pogledate ovaj video morate da otvorite LAMS lekciju.**

## ✓ Poglavlje 1

# Redux - ključne ideje

## UVODNO RAZMATRANJE

*Tradicionalni načini strukturiranja podataka nisu u potpunosti adekvatni za velike veb aplikacije.*

Ukoliko student ne poseduje nikakvo saznanje u vezi sa šablonom arhitekture podataka **Redux**, može informativno da poseti zvaničnu prezentaciju na putem linka: <https://redux.js.org/>.

**Arhitektura podataka** veb aplikacija je u stalnoj evoluciji pa se pokazuje da tradicionalni načini strukturiranja podataka nisu u potpunosti adekvatni za primenu u velikim veb aplikacijama. **Redux** je postao veoma popularan iz razloga što je jednostavan za primenu i razumevanje, a sa druge strane predstavlja izuzetno moćan alat.

Arhitektura podataka može da predstavlja veoma složenu temu, a **Redux** je verovatno najbolje rešenje koje taj problem može da pojednostavi. Samo jezgro **Redux** - a je veoma jednostavno i poseduje manje od 100 linija koda.

U praksi se pokazalo da je veoma jednostavno moguće kreirati "bogatu" aplikaciju, jednostavnu za razumevanje, sa **Redux** šablonom arhitekture podataka kao osloncem. Za početak je neophodno sagledati kako je moguće kreirati mali **Redux** da bi, u nastavku, bili razrađeni šabloni koji će implementirati navedene ideje u većim aplikacijama.

**Ova lekcija sadrži video materijal. Ukoliko želite da pogledate ovaj video morate da otvorite LAMS lekciju.**

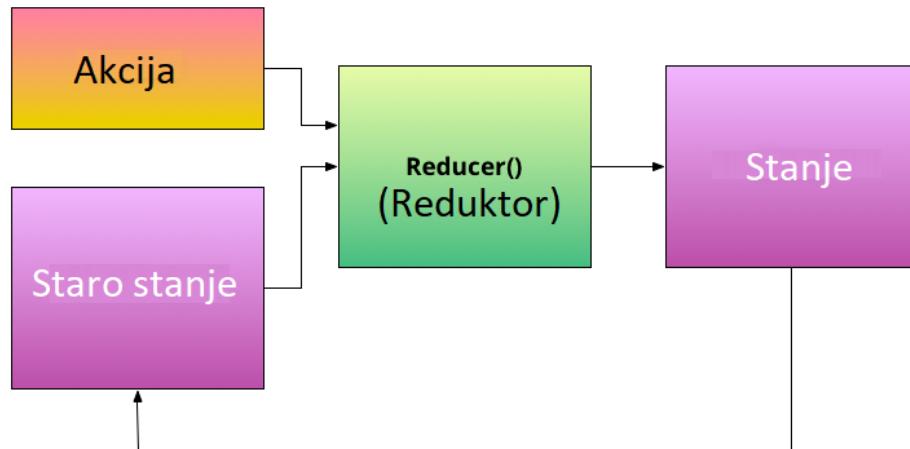
## KLJUČNE IDEJE PRIMENE ŠABLONA REDUX

*Postoje brojne ideje za primenu šablona Redux u Angular aplikacijama.*

Postoje brojne ideje za primenu šablona **Redux** u **Angular** aplikacijama. Slede najznačajnije:

- svi podaci aplikacije se nalaze u jedinstvenoj strukturi podataka koja se naziva *stanje (state)* i čuva se u *skladištu (store)*;
- aplikacija čita stanje iz skladišta;
- Interakcija korisnika (kao i ostali kod) pokreće neku akciju koja opisuje šta se upravo desilo u aplikaciji;
- novo stanje se kreira kombinovanjem starog stanja i navedene akcije primenom funkcije koja se naziva *reduktor (reducer)*

Sledećom slikom je ilustrovano jezgro šablona *Redux*.



Slika 1.1 Ilustracija jezgra šablona Redux [izvor: autor]

## REDUKTOR

*Reduktor uzima staro stanje i akciju i vraća novo stanje.*

Kao što je istaknuto u prethodnom izlaganju, novo stanje aplikacije se kreira kombinovanjem starog stanja i navedene akcije primenom funkcije koja se naziva *reduktor (reducer)*.

Upravo to govori i definicija funkcije *reducer()* : reduktor uzima staro stanje i akciju i vraća novo stanje. Reduktor mora da bude "čista" funkcija (pure function) - [https://en.wikipedia.org/wiki/Pure\\_function](https://en.wikipedia.org/wiki/Pure_function). To znači sledeće:

- reduktor ne sme direktno da mutira tekuće stanje;
- ne sme da koristi druge podatke osim vlastitih argumenata.

To može da se sagleda i drugačije, čista funkcija će uvek vratiti istu vrednost, za dati skup argumenata. Čista funkcija nikada neće pozvati bilo koju funkciju koja ima uticaj na okruženje, na primer na: poziv baze podataka, HTTP poziv, mutiranje strukture podataka okruženja i tako dalje.

Reduktor uvek tretira trenutno stanje kao "samo za čitanje" (*read - only*). Reduktor ne menja stanje već vraća novo.

Sada je pravi trenutak za definisanje prvog reduktora. Treba imati na umu sledeće tri stvari koje će biti uključene:

1. *Action* - definiše šta bi trebalo uraditi (sa opcionalnim argumentima)
2. *state* - čuva sve podatke aplikacije;
3. *Reducer* - uzima tekuće stanje i akciju i vraća novo stanje.

## INTERFEJSI REDUCER I ACTION

*Podrška za primenu tipova podataka - podešeni su interfejsi Reducer i Action.*

Budući da je izabran **TypeScript** kao jezik za kodiranje u pokaznim primerima ovog predmeta, i ovde je neophodno obezbediti sve preduslove da ceo proces izrade primera bude snabdeven podrškom za primenu tipova podataka. U tom svetlu biće podešeni interfejsi **Reducer** i **Action**.

I u ovom slučaju biće kreiran pokazni primer (**redux-chat**) kojeg studenti mogu da preuzmu nakon prelaska objekata učenja pradavanja.

Interfejs **Action** izgleda kao u sledećem listingu (datoteka primera: /redux-chat/tutorial/01-identity-reducer.ts):

```
interface Action {
 type: string;
 payload?: any;
}
```

Ovde je moguće primetiti da interfejs **Action** poseduje dva polja:

1. **type**;
2. **payload**.

Poljem **type** je određen string koji identificuje akciju, na primer **INCREMENT** ili **ADD\_USER**, i tako dalje. Znak pitanja, uz polje **payload**, znači da je posmatrano polje optionalno.

Interfejs **Reducer** izgleda kao u sledećem listingu (datoteka primera: /redux-chat/tutorial/01-identity-reducer.ts):

```
interface Reducer<T> {
 (state: T, action: Action): T;
}
```

Interfejs **Reducer** koristi **TypeScript** podršku za rad sa generičkim tipovima podataka. U konkretnom slučaju, tip **T** označava tip stanja. Ovim interfejsom je definisana funkcija **state** (povratnog tipa **T**) koja uzima stanje tipa **T** i akciju, a zatim vraća novo stanje navedenog tipa podataka **T**.

## KREIRANJE I POKRETANJE PRVOG REDUKTORA

*Najjednostavniji reduktor ima sposobnost da vrati vlastito stanje.*

Najjednostavniji reduktor ima sposobnost da vrati vlastito stanje. Ovakav reduktor može da se nazove i identičnim reduktorom (identity reducer) jer primenjuje identičnu funkciju (

[https://en.wikipedia.org/wiki/Identity\\_function](https://en.wikipedia.org/wiki/Identity_function)) na stanje. Ovo je podrazumevani slučaj za sve reduktore. Ponovo je pažnja na datoteci [/redux-chat/tutorial/01-identity-reducer.ts](#) u kojoj je neophodno, nakon interfejsa, dodati još malo koda:

```
let reducer: Reducer<number> = (state: number, action: Action) => {
 return state;
};
```

Iz priloženog listinga je moguće primetiti da *Reducer* konkretizuje generički tip podataka u tip *number* primenom sintakse *Reducer<number>*. Argument tipa *Action* se još uvek ne koristi ali je ipak moguće isprobati kreirani reduktor. Pre toga je neophodno priložiti objedinjeni listing:

```
interface Action {
 type: string;
 payload?: any;
}

interface Reducer<T> {
 (state: T, action: Action): T;
}

let reducer: Reducer<number> = (state: number, action: Action) => {
 return state;
};

console.log(reducer(0, null)); // -> 0
```

Ovaj primer se izvršava izvan pregledača i pokreće ga **node.js**.

Sada je neophodno obaviti navigaciju do direktorijuma u kojem se čuva datoteka sa priloženim kodom (u priloženom primeru to je [/redux-chat/tutorial](#), pa bi za pokretanje ovog primera u terminalu trebalo kucati sledeće:

```
cd Angular/redux-chat/tutorial //ovo koda vas može biti drugačije
npm install
./node_modules/.bin/ts-node 01-identity-reducer.ts
```

Iako je ovo primer veoma jednostavnog koda, on ukazuje na jednu veoma bitnu stvar: po osnovnim podešavanjima, reduktor vraća originalno stanje.

U konkretnom slučaju, metodi *reducer()* je prosleđen broj 0 i akcija *null*. Rezultat dobijen reduktorom je stanje 0 (vidi sliku).

U nastavku, akcenat će biti na zanimljivijim detaljima koji utiču na promenu stanja.



```
PROBLEMS 1 OUTPUT DEBUG CONSOLE TERMINAL
PS C:\Users\Vlada\Documents\Angular\redux-chat\tutorial> ./node_modules/.bin/ts-node 01-identity-reducer.ts
0
PS C:\Users\Vlada\Documents\Angular\redux-chat\tutorial>
```

Slika 1.2 Rezultat dobijen reduktoriom je stanje 0 [izvor: autor]

## PODEŠAVANJE BROJAČA SA AKCIJAMA

*Kreira akcija koja će dati reduktoru instrukciju kako da generiše novo stanje - promenu brojača.*

Budući da će vrlo uskoro stanje predstavljati sofisticiranija vrednost nego što je to jedan broj, neophodno je obezbediti bolju strukturu podataka za stanje. Za sada, ovakva vrednost (jedan broj) omogućava lakše fokusiranje na druge probleme. Pa je moguće nastaviti dalje sa idejom da zapravo ta numerička vrednost predstavlja brojač.

Širi se razmišljanje i želja je da se kreira akcija koja će dati reduktoru instrukciju kako da generiše novo stanje. Akcija će imati konkretan zadatka da **promeni stanje brojača**. Pre bilo kakvog kodiranja trebalo bi imati na umu da jedina obavezna osobina akcije je **type**. Imajući u vidu navedeno, moguće je definisati prvu akciju na sledeći način:

```
let incrementAction: Action = { type: 'INCREMENT' }
```

Moguće je kreirati i drugu akciju kojom se smanjuje vrednost brojača:

```
let decrementAction: Action = { type: 'DECREMENT' }
```

Sada je moguće isprobati kreirane akcije kroz primenu u reduktoru (datoteka: */redux-chat/tutorial/02-adjusting-reducer.ts*):

```
let reducer: Reducer<number> = (state: number, action: Action) => {
 if (action.type === 'INCREMENT') {
 return state + 1;
 }
 if (action.type === 'DECREMENT') {
 return state - 1;
 }
 return state;
};
```

Neophodno je još malo koda za kompletiranje reduktora:

```
let incrementAction: Action = { type: 'INCREMENT' };

console.log(reducer(0, incrementAction)); // -> 1
console.log(reducer(1, incrementAction)); // -> 2

let decrementAction: Action = { type: 'DECREMENT' };

console.log(reducer(100, decrementAction)); // -> 99
```

Sada, konačno, **postoji nova vrednost stanja koja je vraćena u zavisnosti od akcije prosleđene reduktoru**. Navođenjem naredbe:

```
./node_modules/.bin/ts-node 02-adjusting-reducer.ts
```

iz terminala se pokreće izvršavanje reduktora i dobija se izlaz kao na sledećoj slici:

```
PS C:\Users\Vlada\Documents\Angular\redux-chat\tutorial> ./node_modules/.bin/ts-node 02-adjusting-reducer.ts
1
2
99
PS C:\Users\Vlada\Documents\Angular\redux-chat\tutorial>
```

Slika 1.3 Nova vrednost stanja vraćena u zavisnosti od akcije prosleđene reduktoru [izvor: autor]

## REDUKTOR SWITCH

*Za smanjenje broja if iskaza u telu reduktora je moguće primeniti switch.*

Umesto primene velikog broja if naredni, dobra praksa je konvertovanje tela reduktora u formu switch iskaza (nova datoteka: `/redux-chat/tutorial/03-adjusting-reducer-switch.ts`):

```
let reducer: Reducer<number> = (state: number, action: Action) => {
 switch (action.type) {
 case 'INCREMENT':
 return state + 1;
 case 'DECREMENT':
 return state - 1;
 default:
 return state; // <-- ne sme da se zaboravi!!!
 }
};
```

Neophodno je još malo koda za kompletiranje reduktora:

```
let incrementAction: Action = { type: 'INCREMENT' };
console.log(reducer(0, incrementAction)); // -> 1
console.log(reducer(1, incrementAction)); // -> 2

let decrementAction: Action = { type: 'DECREMENT' };
console.log(reducer(100, decrementAction)); // -> 99

// bilo koja druga akcija vraća ulazno stanje
let unknownAction: Action = { type: 'UNKNOWN' };
console.log(reducer(100, unknownAction)); // -> 100
```

Neophodno je primetiti da `default` slučaj switch naredbe vraća originalno stanje. Na ovaj način je obezbeđeno da, ukoliko se prosledi nepoznata akcija, neće doći do greške i dobiće se originalno stanje nepromenjeno.

Navođenjem naredbe:

```
./node_modules/.bin/ts-node 03-adjusting-reducer-switch.ts
```

iz terminala se pokreće izvršavanje reduktora i dobija se izlaz kao na sledećoj slici:

```
PS C:\Users\Vlada\Documents\Angular\redux-chat\tutorial> ./node_modules/.bin/ts-node 03-adjusting-reducer-switch.ts
1
2
99
100
PS C:\Users\Vlada\Documents\Angular\redux-chat\tutorial>
```

Slika 1.4 Primena switch bloka u reduktoru [izvor: autor]

## ARGUMENTI AKCIJA

*Često promene u aplikaciji ne mogu biti opisane pomoću jedinstvene vrednosti.*

U poslednjem primeru akcija je sadržala samo argument type koji je ukazivao reduktoru da uveća ili smanji stanje za 1.

Međutim, često promene u aplikaciji ne mogu biti opisane pomoću jedinstvene vrednosti. Umesto toga, koriste se parametri da opišu promenu. Upravo iz navedenog razloga postoji polje **payload**, koje je označeno kao opcionalno, a koje može biti od velike koristi prilikom definisanja akcija. Na primer, potrebno je dodati 9 na brojač. Jedan od načina je slanje 9 **INCREMENT** akcija ka reduktoru ali, očigledno da to nije efikasan način (zamislite još veći broj, na primer 9000).

Umesto navedenog neefikasnog pristupa, biće uključena drugačija logika - biće kreirana akcija pod nazivom **PLUS** koja će koristiti parametar **payload** za slanje broja kojim se ukazuje koliko bi trebalo uvećati vrednost brojača. Definisanje ove akcije je veoma jednostavno:

```
let plusSevenAction = { type: 'PLUS', payload: 7 };
```

Za podršku izvođenju ove akcije neophodno je kreirati novi **case** slučaj reduktora za rukovanje akcijom **PLUS**. Svi prethodni primeri su bili razdvojeni o posebne datoteke, tako će i ovde biti slučaj. Sledećim listingom je priložen kod reduktora koji se čuva u datoteci **/redux-chat/tutorial/04-plus-action.ts**:

```
let reducer: Reducer<number> = (state: number, action: Action) => {
 switch (action.type) {
 case 'INCREMENT':
 return state + 1;
 case 'DECREMENT':
 return state - 1;
 case 'PLUS':
 return state + action.payload;
 default:
 return state;
 }
};
```

Neophodno je još malo koda za kompletiranje reduktora:

```
console.log(reducer(3, { type: 'PLUS', payload: 7 })); // -> 10
console.log(reducer(3, { type: 'PLUS', payload: 9000 })); // -> 9003
console.log(reducer(3, { type: 'PLUS', payload: -2 })); // -> 1
```

Navođenjem naredbe:

```
./node_modules/.bin/ts-node 04-plus-action.ts
```

iz terminala se pokreće izvršavanje reduktora i dobija se izlaz kao na sledećoj slici:

```
PS C:\Users\Vlada\Documents\Angular\redux-chat\tutorial> ./node_modules/.bin/ts-node 04-plus-action.ts
10
9003
1
PS C:\Users\Vlada\Documents\Angular\redux-chat\tutorial>
```

Slika 1.5 Primena argumenata akcija [izvor: autor]

U prvoj liniji na stanje 3 pozvana je akcija **PLUS** sa **payload** vrednošću 7 i dobijen je rezultat 10. Ono što je moguće primetiti da se nijednom nije desila promena originalnog stanja, to je zbog toga što ne postoji mogućnost čuvanja reduktorovih promena i njihove ponovne upotrebe.

## VIDEO MATERIJAL

*Angular 6 Tutorial 31: Redux - Introduction - trajanje: 6:35.*

**Ova lekcija sadrži video materijal. Ukoliko želite da pogledate ovaj video morate da otvorite LAMS lekciju.**

## ▼ Poglavlje 2

### Skladište stanja

#### ČUVANJE STANJA

*Skladište poseduje odgovornost za pokretanje reduktora i čuvanje novih stanja.*

Reduktori su čiste funkcije i ne mogu da menjaju podatke iz okruženja. Problem u aplikaciji je što postoji mnogo promenljivih stvari. Posebno se to odnosi na stanja čije promene moraju negde da se čuvaju. **Redux** insistira na čuvanju stanja unutar **skladišta stanja** (store). **Skladište poseduje odgovornost za pokretanje reduktora i čuvanje novih stanja.**

Sledećim listingom je opisano minimalno skladište stanja (datoteka: [redux-chat/tutorial/05-minimal-store.ts](#)):

```
class Store<T> {
 private _state: T;

 constructor(
 private reducer: Reducer<T>,
 initialState: T
) {
 this._state = initialState;
 }

 getState(): T {
 return this._state;
 }

 dispatch(action: Action): void {
 this._state = this.reducer(this._state, action);
 }
}
```

Prvo što je moguće primetiti jeste da je klasa **Store** generička - specificiran je tip stanja opštim tipom **T**. Stanje se čuva u privatnoj promenljivoj pod nazivom **\_state**. Klasi **Store** je pridružen i reduktor, umetanjem putem konstruktora, koji je, takođe, generičkog tipa **T**. Ovo je učinjeno iz razloga što svakom skladištu odgovara specifičan reduktor. Reduktor (objekat tipa **Reducer**) čuva se u privatnoj promenljivoj **reducer**.

**U Redux - u, uopšteno postoji jedno skladište stanja i jedan reduktor najvišeg nivoa (top-level reducer) po aplikaciji.**

Sada bi bilo poželjno detaljnije obratiti pažnju na sve priložene metode klase `Store` - skladišta stanja:

- u konstruktoru, osobina `_state` se podešava na inicijalno stanje `initialState`;
- metoda `getState()` jednostavno vraća tekuće stanje;
- metoda `dispatch()` uzima akciju, prosleđuje je reduktoru i ažurira vrednost osobine `_state` preko povratne vrednosti metode `reducer()`.

Važno je primetiti da metoda `dispatch()` ne vraća nikakvu vrednost, ona samo obavlja ažuriranje stanja iz skladišta. Ovo je važan princip primene `Redux` - a koji kaže da je raspoređivanje akcija manevar tipa "*ispali i zaboravi (fire and forget)*". Raspoređivanje akcija ne predstavlja direktnu manipulaciju stanjem i ne vraća novu vrednost stanja. Kada se šalje akcija, šalje se obaveštenje šta se desilo, a stanje se proverava iz skladišta.

## PRIMENA SKLADIŠTA

*Neophodno je dodati još malu količinu koda sa ciljem demonstracije čuvanja stanja unutar skladišta*

Nakon inicijalne definicije skladišta, neophodno je dodati još malu količinu, ali značajnog, koda sa ciljem demonstracije čuvanja stanja unutar skladišta.

Ponovo je akcenat na datoteci `/redux-chat/tutorial/05-minimal-store.ts` u kojoj će kod biti proširen sledećim listingom:

```
// isti reduktor kao pre
let reducer: Reducer<number> = (state: number, action: Action) => {
 switch (action.type) {
 case 'INCREMENT':
 return state + 1;
 case 'DECREMENT':
 return state - 1;
 case 'PLUS':
 return state + action.payload;
 default:
 return state;
 }
};

// kreira novo skladište
let store = new Store<number>(reducer, 0);
console.log(store.getState()); // -> 0

store.dispatch({ type: 'INCREMENT' });
console.log(store.getState()); // -> 1

store.dispatch({ type: 'INCREMENT' });
console.log(store.getState()); // -> 2
```

```
store.dispatch({ type: 'DECREMENT' });
console.log(store.getState()); // -> 1
```

Iz listinga se vidi (linija koda 16) kreiranje novog objekta klase `Store` koji sada može biti upotrebljen da prikaže tekuće stanje i prosledi akciju.

U prvom koraku je stanje inicijalno podešeno na vrednost 0, a zatim je dva puta angažovana akcija `INCREMENT`, a potom jedanput i akcija `DECREMENT`. Konačno stanje odgovara vrednosti 1 (videti sledeću sliku).

Navođenjem naredbe:

```
./node_modules/.bin/ts-node 05-minimal-store.ts
```

iz terminala se pokreće izvršavanje reduktora, čija stanja se sada čuvaju u skladištu `Store`, i dobija se izlaz kao na sledećoj slici:

```
PS C:\Users\Vlada\Documents\Angular\redux-chat\tutorial> ./node_modules/.bin/ts-node 05-minimal-store.ts
0
1
2
1
PS C:\Users\Vlada\Documents\Angular\redux-chat\tutorial>
```

Slika 2.1 Korišćenje reduktora sa skladištem stanja [izvor: autor]

## DOBIJANJE OBAVEŠTENJA SA SUBSCRIBE()

*Mogućnost dobijanja trenutne informacije kada je nova akcija prosleđena.*

Odlična mogućnost je to što skladište stanja čuva informacije o promenama. Međutim, u prethodnom primeru je u kodu je zatražena izmena stanja direktno preko poziva `store.getState()`. Bilo bi odlično ukoliko bi postajala mogućnost dobijanja trenutne informacije kada je nova akcija prosleđena da bi pravovremen odgovor mogao da bude pripremljen. Za realizovanje navedenih ciljeva neophodno je obaviti implementaciju šablonu `Observer`. To znači da će biti registrovana povratna funkcija koja će se prijaviti (`subscribe`) na sve promene.

Sledećim koracima je moguće postići navedeno:

1. registrovanje osluškivačke (`listener`) funkcije preko `subscribe()`;
2. kada se pozove `dispatch()`, biće obavljena iteracija preko svih osluškivača i njihovo pozivanje, a to upravo predstavlja obaveštenje (notifikaciju) da je došlo do promene.

## REGISTROVANJE OSLUŠKIVAČA

*Osluškivačke metode će biti funkcije bez argumenata i kao takve će biti i kreirane.*

Osluškivačke metode će biti funkcije bez argumenata i kao takve će biti i kreirane. Sledеćim listingom je prikazan interfejs preko kojeg će sve lakše biti prikazano i opisano:

```
interface ListenerCallback {
 (): void;
}
```

Nakon registrovanja (prijavljivanja) osluškivača, neophodno je obezbediti i mehanizam za njegovo odjavljivanje. U tu svrhu se kreira nov interfejs (u istoj datoteci kao i prethodni [/redux-chat/tutorial/06-store-w-subscribe.ts](#)):

```
interface UnsubscribeCallback {
 (): void;
}
```

Ni ovde se puno toga ne dešava, radi se o još jednoj funkciji bez argumenata koja ne uzima argumente i ne vraća vrednost.

Međutim, definisanjem navedenih tipova postiže se jasniji i čitljiviji kod.

Sada je moguće dopuniti kod skladišta na način da će klasa `Store` imati mogućnost čuvanja liste objekata tipa `ListenerCallback`.

```
class Store<T> {
 private _state: T;
 private _listeners: ListenerCallback[] = [];
```

U nastavku je neophodno omogućiti dodavanje na listu `_listeners` primenom `subscribe()` funkcije:

```
subscribe(listener: ListenerCallback): UnsubscribeCallback {
 this._listeners.push(listener);
 return () => { // vraća "unsubscribe" funkciju
 this._listeners = this._listeners.filter(l => l !== listener);
 };
}
```

Definisana funkcija `subscribe()` prihvata `ListenerCallback` (na primer, funkciju bez argumenata i povratne vrednosti), a vraća `UnsubscribeCallback` (sa istim potpisom). Dodavanje novog osluškivača je veoma jednostavno - samo ga je potrebno dodati na `_listeners` niz.

Povratna vrednost za metodu `subscribe()` predstavlja funkcija koja vrši ažuriranje liste `_listeners` u formi liste osluškivača bez poslednje dodatog objekta `listener`. To znači da vraća `UnsubscribeCallback` kojeg koristi za uklanjanje navedenog osluškivača iz liste.

## OBAVEŠTAVANJE OSLUŠKIVAČA I OBJEDINJAVANJE KODA KLASE STORE

*Svaki put kada se stanje promeni, neophodno je obaviti pozivanje navedenih osluškivačkih funkcija.*

Svaki put kada se stanje promeni, neophodno je obaviti pozivanje navedenih osluškivačkih funkcija. To znači, svaki put kada se pošalje nova [akcija](#) i svaki put kada se promeni stanje, neophodno je pozvati sve osluškivače (dopuna datoteke [/redux-chat/tutorial/06-store-with-subscribe.ts](#)):

```
dispatch(action: Action): void {
 this._state = this.reducer(this._state, action);
 this._listeners.forEach((listener: ListenerCallback) => listener());
}
```

Pre nego što se pristupi testiranju dopunjenoj koda neophodno je klasu Store sagledati kao celinu, odnosno neophodno je priložiti celokupan njen kod sa izmenama koje su elaborirane u prethodnom izlaganju:

```
class Store<T> {
 private _state: T;
 private _listeners: ListenerCallback[] = [];

 constructor(
 private reducer: Reducer<T>,
 initialState: T
) {
 this._state = initialState;
 }

 getState(): T {
 return this._state;
 }

 dispatch(action: Action): void {
 this._state = this.reducer(this._state, action);
 this._listeners.forEach((listener: ListenerCallback) => listener());
 }

 subscribe(listener: ListenerCallback): UnsubscribeCallback {
 this._listeners.push(listener);
 return () => { // vraća "unsubscribe" funkciju
 this._listeners = this._listeners.filter(l => l !== listener);
 };
 }
}
```

Da bi metoda [subscribe\(\)](#) mogla da bude primenjena za prijavljivanje i osluškivanje promena skladišta stanja, na prethodni listing neophodno je dodati još malo koda:

```
// novo skladište
let store = new Store<number>(reducer, 0);
console.log(store.getState()); // -> 0

// subscribe
let unsubscribe = store.subscribe(() => {
 console.log('subscribed: ', store.getState());
```

```
});

store.dispatch({ type: 'INCREMENT' }); // -> subscribed: 1
store.dispatch({ type: 'INCREMENT' }); // -> subscribed: 2

unsubscribe();
store.dispatch({ type: 'DECREMENT' }); // (nothing logged)

// dekrement se desio, iako nije bio osluškivan
console.log(store.getState()); // -> 1
```

U priloženom kodu je obavljeno prijavljivanje na skladište stanja *Store*, a zatim je u povratnoj (*callback*) funkciji u log - u prikazan string : " *subscribed : tekućeStanje*". Zatim je pozvana metoda *unsubscribe()* ali bez log prikaza. I dalje je moguće slati akcije ali se ne prikazuju rezultati sve dok se to ne zatraži od skladišta.

Kao u prethodnim slučajevima, iz terminala se pokreće izvršavanje reduktora, čija stanja se sada čuvaju u skladištu Store, i dobija se izlaz kao na sledećoj slici:

```
PS C:\Users\Vlada\Documents\Angular\redux-chat\tutorial> ./node_modules/.bin/ts-node 06-store-w-subscribe.ts
0
subscribed: 1
subscribed: 2
1
PS C:\Users\Vlada\Documents\Angular\redux-chat\tutorial> []
```

Slika 2.2 Korišćenje reduktora sa skladištem stanja i obaveštavanjem osluškivača [izvor: autor]

## REDUX JEZGRO

*Klasa Store je prikaz osnovnog jezgra Redux arhitekture podataka.*

Prikazano skladište stanja (klasa *Store*) je prikaz osnovnog jezgra *Redux* arhitekture podataka. *Reduktor uzima tekuće stanje i akciju i njihovim kombinovanjem omogućava vraćanje novog stanja*. Ono što je veoma bitno, a sledi iz prethodnog izlaganja, *vraćeno novo stanje se čuva u skladištu stanja*.

Sasvim je jasno, čak i sa ovim nivoom znanja o *Redux* arhitekturi podataka, da je potrebno obaviti još mnogo zadataka sa ciljem izgradnje velike *Angular* aplikacije, spremne za produkciju.

Međutim, sve nove ideje, koje će u nastavku biti izložene i pokrivenе, predstavljaju šablonе koji polaze od ove jednostavne ideje o nepromenljivom, centralnom skladištu stanja. Ukoliko su studenti uspešno savladali sve ideje, koje su prethodnom izlaganju iznete i obrađene, biće u mogućnosti da primene brojne šablonе i biblioteke koje je moguće sresti u naprednim *Redux* aplikacijama.

Međutim, da bi sve navedeno postalo realnost neophodno je pokriti još dosta toga neophodnog za svakodnevno korišćenje *Redux* arhitekture podataka u savremenim moćnim *Angular* veb aplikacijama. Na primer, neophodno je savladati sledeće:

- Kako je moguće na pažljiv način rukovati kompleksnim strukturama podataka u stanju;
- Kako je moguće poslati obaveštenje kada se stanje promeni bez potrebe za obraćanjem stanju (preko prijavljivanja - *subscription*);

- Kako je moguće presresti metodu `dispatch()` sa ciljem provere grešaka (`debugging`);
- Kako je moguće obraditi izvedene vrednosti (primenom selektora);
- Kako je moguće podeliti reduktore u manje, lakše upravljive i obaviti njihovo ponovno kombinovanje;
- Kako je moguće rukovati asinhronim podacima.

Upravo, u nastavku izlaganja ove lekcije, postavlja se nov cilj koji predstavlja analizu i diskusiju svake od navedenih naprednih `Redux` tema. U prvom koraku je neophodno potražiti odgovor na pitanje: "Kako je moguće na pažljiv način rukovati kompleksnim strukturama podataka u stanju?" U tu svrhu je neophodno pozabaviti se zanimljivijim i nešto složenijim primerom nego što je to bio slučaj sa brojačem. Ponovo će akcenat biti na aplikaciji za dopisivanje u kojoj korisnici mogu da šalju poruke jedni drugima.

## ▼ Poglavlje 3

# Implementacija reduktora u aplikaciji za časkanje

## APLIKACIJA ZA ĆASKANJE I REDUX PRIMENA

*U Redux aplikacijama, postoje tri glavna dela modela podataka: stanje, akcije i reduktori.*

Kao što je istaknuto u prethodnom izlaganju, u nastavku je potrebno dati odgovor na napredna pitanja koja se javljaju prilikom kreiranja složenih **Redux** aplikacija. U prvom koraku je neophodno potražiti odgovor na pitanje: "Kako je moguće na pažljiv način rukovati kompleksnim strukturama podataka u stanju?" U tu svrhu je neophodno pozabaviti se zanimljivijim i nešto složenijim primerom nego što je to bio slučaj sa brojačem. Ponovo će akcenat biti na aplikaciji za dopisivanje u kojoj korisnici mogu da šalju poruke jedni drugima. U konkretnom slučaju, u aplikaciji za časkanje, kao i u ostalim **Redux** aplikacijama, postoje tri glavna dela modela podataka:

- *stanje;*
- *akcije;*
- *reduktori.*

## STANJE

*U slučaju aplikacije za časkanje, stanje će predstavljati objekat.*

Ako se pažnja vrati na primer sa brojačem, tamo je **stanje** bilo prikazivano jednostavnom vrednošću - brojem. **U slučaju aplikacije za časkanje, stanje će predstavljati objekat.**

Navedeno objektno stanje će posedovati jednu osobinu pod nazivom **messages**. Osobina **messages** će predstavljati niz stringova pri čemu će svaki od navedenih stringova predstavljati jednu poruku u aplikaciji. Na primer, moguće je kreirati kod priložen sledećim listingom:

```
// primer vrednosti stanja
{
 messages: [
 'ovo je prva poruka',
 'ovo je druga poruka'
]
}
```

Tip stanja aplikacije može biti definisan na veoma jednostavan način uvođenjem novog interfejsa. Navedeno može biti ilustrovano sledećim listingom (datoteka aktuelnog primera:[/redux-chat/tutorial/07-messages-reducer.ts](#)) :

```
interface AppState {
 messages: string[];
}
```

## AKCIJE

*Predviđeno je da aplikacija obrađuje dve različite akcije:  
**ADD\_MESSAGE**i **DELETE\_MESSAGE**.*

Predviđeno je da aplikacija, koja odgovara pokaznom primeru, obrađuje dve različite akcije:

- **ADD\_MESSAGE** i
- **DELETE\_MESSAGE**.

Objekat prve akcije, pod nazivom **ADD\_MESSAGE**, će uvek posedovati osobinu *message* koja odgovara poruci koju je neophodno dodati u stanje. Objekat akcije **ADD\_MESSAGE** ima sledeći oblik:

```
{
 type: 'ADD_MESSAGE',
 message: 'Da li je IT255 težak predmet?' //ili neka druga poruka
}
```

Objekat akcije **DELETE\_MESSAGE** ima zadatka da obezbedi brisanje određene poruke iz stanja. Ovde je sada poseban izazov mogućnost specificiranja poruke koja će biti obrisana. Ukoliko su poruke objekti, svakoj poruci može biti dodeljena osobina *id* prilikom njenog kreiranja.

Međutim, zbog lakšeg razumevanja, problem može biti pojednostavljen. Poruke su samo obični stringovi i njihovo rukovanje će biti rešeno na drugi način. Najlakši način predstavlja upotreba indeksa poruke iz odgovarajućeg niza poruka (analogija za ID).

Imajući na umu navedeno, objekat akcije **DELETE\_MESSAGE** ima sledeći oblik:

```
{
 type: 'DELETE_MESSAGE',
 index: 1 //ili neki drugi odgovarajući indeks
}
```

Tipovi navedenih akcija mogu veoma jednostavno da se definišu uvođenjem adekvatnih interfejsa (povratak u datoteku:[/redux-chat/tutorial/07-messages-reducer.ts](#)):

```
interface AddMessageAction extends Action {
 message: string;
}
```

```
interface DeleteMessageAction extends Action {
 index: number;
}
```

Na ovaj način interfejs `AddMessageAction` poseduje sposbnost specificiranja poruke, a interfejs `DeleteMessageAction` poseduje sposbnost specificiranja indeksa.

## REDUKTOR

*Reduktor ima sposobnost rukovanja dvema akcijama: ADD\_MESSAGE and DELETE\_MESSAGE.*

Reduktor bi trebalo da ima sposobnost rukovanja dvema akcijama: `ADD_MESSAGE` and `DELETE_MESSAGE`. U tom svetlu teći će dve odvojene diskusije, za svaku od navedenih akcija pojedinačno.

### Redukcija akcije ADD\_MESSAGE

Za implementaciju koristi se reduktor koji integriše naredbu `switch`. Navedeno je priloženo sledećim listingom:

```
et reducer: Reducer<AppState> =
(state: AppState, action: Action): AppState => {
 switch (action.type) {
 case 'ADD_MESSAGE':
 return {
 messages: state.messages.concat(
 (<AddMessageAction>action).message
),
 };
 };
```

Kada se rukuje akcijom `ADD_MESSAGE`, neophodno je dodati odgovarajuću poruku u stanje. Kao što je poznato, ako rezultat će biti vraćeno novo stanje. Ono što bi sve vreme trebalo imati na umu da su reduktori "čiste" funkcije koje ne vrše mutiranje stanja. Imajući navedeno na umu, postavlja se pitanje: "U čemu je problem sa kodom iz sledećeg listinga?"

```
case 'ADD_MESSAGE':
 state.messages.push(action.message);
 return { messages: messages };
// ...
```

Odgovor leži u činjenici da ovakav kod vrši mutiranje niza `state.messages` kojim se menja staro stanje.

Umesto navedenog neophodno je kreirati kopiju niza `state.messages` i dodati novu poruku na kopiju, kao u slučaju prikazanim sledećim listingom:

```
case 'ADD_MESSAGE':
 return {
```

```
messages: state.messages.concat(
 (<AddMessageAction>action).message
)
};
```

Sve vreme bi trebalo imati na umu da bi reduktor trebalo da vrati *AppState*. Kada se vrati objekat iz reduktora on mora da odgovara formatu *AppState* koji je bio na ulazu. U konkretnom slučaju je neophodno sačuvati ključ *messages*, ali u slučaju komplikovanih stanja postoji veći broj polj (ključeva) o kojima bi trebalo voditi računa.

## BRISANJE STAVKI BEZ MUTACIJE

*Rukovanjem akcijom `DELETE_MESSAGE` vrši se prosleđivanje indeksa elementa u nizu poruka.*

Na razmatranje dolazi sledeća akcija kojom bi trebalo da rukuje reduktor nosi naziv *DELETE\_MESSAGE* pri čemu se vrši prosleđivanje indeksa koji odgovara položaju elementa u nizu poruka. U slučaju niza sa kompleksnijim objektima, u analognom slučaju, vršilo bi se prosleđivanje *ID* osobine objekta. Ovde bi, ponovo, bilo potrebno voditi računa da je nedopušteno mutiranje niza poruka. Prema tome, rukovanje navedenom akcijom bi trebalo obaviti na pažljiv način, kao u kodu koji je priložen sledećim listingom:

```
case 'DELETE_MESSAGE':
 let idx = (<DeleteMessageAction>action).index;
 return {
 messages: [
 ...state.messages.slice(0, idx),
 ...state.messages.slice(idx + 1, state.messages.length)
]
 };
```

Iz priloženog listinga je moguće primetiti da je dva puta upotrebljen operator pod nazivom *slice*. U prvom koraku se uzimaju svi elementi niza do elementa koji se briše, a zatim se vrši spajanje elemenata koji dolaze posle.

**Postoje 4 opšte nemutirajuće operacije:**

- 1. Dodavanje elementa u niz;**
- 2. Uklanjanje elementa iz niza;**
- 3. Dodavanje / menjanje ključau objektu;**
- 4. Uklanjanje ključa iz objekta.**

## DEMONSTRIRANJE AKCIJA

*Neophodno je dodati još malo koda koji je, upravo, zadužen za pokretanje akcija.*

Za demonstraciju rukovanja kreiranim akcijama, neophodno je dodati još malo koda koji je, upravo, zadužen za pokretanje akcija. Ponovo se pažnja usmerava na datoteku aktuelnog primera/[redux-chat/tutorial/07-messages-reducer.ts](#) koja se proširuje kodom iz sledećeg listinga:

```
// kreira novo skladište
let store = new Store<AppState>(reducer, { messages: [] });
console.log(store.getState()); // -> { messages: [] }

store.dispatch({
 type: 'ADD_MESSAGE',
 message: 'Da li je IT255 težak predmet?',
} as AddMessageAction);

store.dispatch({
 type: 'ADD_MESSAGE',
 message: 'Nije težak, ali mora da se uči!!!!'
} as AddMessageAction);

store.dispatch({
 type: 'ADD_MESSAGE',
 message: 'Položićemo ga u februaru!!!!'
} as AddMessageAction);

console.log(store.getState());
// ->
// { messages:
// ['Da li je IT255 težak predmet?',
// 'Nije težak, ali mora da se uči!!!!',
// 'Položićemo ga u februaru!!!!'] }

store.dispatch({
 type: 'DELETE_MESSAGE',
 index: 1
} as DeleteMessageAction);

console.log(store.getState());
// ->
// { messages:
// ['Da li je IT255 težak predmet?',
// 'Položićemo ga u februaru!!!!'] }

store.dispatch({
 type: 'DELETE_MESSAGE',
 index: 0
} as DeleteMessageAction);

console.log(store.getState());
// ->
// { messages: ['Položićemo ga u februaru!!!!'] }
```

Sam početak je vezan za kreiranje novog skladišta, poziv `store.getState()` i dobijanje praznog niza poruka. U nastavku, dodata su tri poruke u skladište. Za svaku od poruka je specificiran

je objekat sa tipom akcije `ADD_MESSAGE`, pri čemu se svaki objekat prevodi (`cast`) u `AddMessageAction`. Konačno, u log - u se prikazuje novo stanje i moguće je videti niz messages koji sadrži sve tri poruke.

Većina veb programera se slaže da je sintaksa `dispatch` iskaza "ružna" (baš tim rečima). Postoje dva razloga za to:

1. svaki put je neophodno manuelno specificirati tip string. Moguće je koristiti konstantu, ali bi bilo dobro ako bi ovo moglo da se izbegne;
2. neophodno je manuelno obaviti prevođenje u `AddMessageAction`.

Umesto direktnog kreiranja navedenih objekata, moguće je kreirati funkciju koja će kreirati posmatrane objekte. Ideja je da se funkcijom kreiraju akcije koje su opšte u Redux - u. Ovaj šablon je poznat pod nazivom **Kreator akcija (Action Creator)**.

Kao u prethodnim slučajevima, iz terminala se pokreće izvršavanje reduktora, čija stanja se sada čuvaju u skladištu `Store`, i dobija se izlaz kao na sledećoj slici:

```
PS C:\Users\Vlada\Documents\Angular\redux-chat\tutorial> ./node_modules/.bin/ts-node 07-messages-reducer.ts
{ messages: [] }
{ messages:
 ['Da li je IT255 težak predmet?',
 'Nije težak, ali mora da se uči!!!',
 'Položićemo ga u februaru!!!']
}
{ messages:
 ['Da li je IT255 težak predmet?',
 'Položićemo ga u februaru!!!']
}
{ messages: ['Položićemo ga u februaru!!!'] }
```

Slika 3.1 Aplikacija za čakanje - akcije [izvor: autor]

## KREATORI AKCIJA (ACTION CREATORS)

*Ideja je da se funkcijom kreiraju akcije koje su opšte u Redux - u.*

Nastavlja se istraživanje u smeru traganja za što efikasnijim razvojem veb aplikacija primenom šablona arhitekture podataka `Redux`. Pa je upravo iz tog razloga, u prethodnom izlaganju jasno istaknuto da umesto direktnog kreiranja navedenih objekata, moguće je kreirati funkciju koja će kreirati posmatrane objekte. Ideja je da se funkcijom kreiraju akcije koje su opšte u `Redux` - u. Ovaj šablon je poznat pod nazivom **Kreator akcija (Action Creator)**.

Hajde da fokus bude na konkretnoj akciji - umesto direktnog kreiranja akcije `ADD_MESSAGE`, biće kreirana funkcija koja će obaviti ovaj zadatak. Ako ste preuzezeli urađeni primer sa kraja lekcije, pažnja se usmerava na datoteku `/redux-chat/tutorial/08-action-creators.ts`.

```
class MessageActions {
 static addMessage(message: string): AddMessageAction {
 return {
 type: 'ADD_MESSAGE',
 message
 };
 }
 static deleteMessage(index: number): DeleteMessageAction {
 return {
 type: 'DELETE_MESSAGE',
 };
 }
}
```

```
 index: index
);
}
}
```

Iz priloženog listinga se primećuje da je kreirana klasa sa dve statičke metode: `addMessage()` i `deleteMessage()`. Pozivom kreiranih metoda vraćaju se vrednosti `AddMessageAction` i `DeleteMessageAction`, tim redosledom.

Pošto su kreirani, sada kreatori akcija mogu da budu i primenjeni:

```
let reducer: Reducer<AppState> =
 (state: AppState, action: Action) => {
 switch (action.type) {
 case 'ADD_MESSAGE':
 return {
 messages: state.messages.concat((<AddMessageAction>action).message),
 };
 case 'DELETE_MESSAGE':
 let idx = (<DeleteMessageAction>action).index;
 return {
 messages: [
 ...state.messages.slice(0, idx),
 ...state.messages.slice(idx + 1, state.messages.length)
]
 };
 default:
 return state;
 }
};

// kreiranje novog sladišta
let store = new Store<AppState>(reducer, { messages: [] });
console.log(store.getState()); // -> { messages: [] }

store.dispatch(
 MessageActions.addMessage('Da li je IT255 težak predmet?'));
store.dispatch(
 MessageActions.addMessage('Nije težak, ali mora da se uči!!!'));
store.dispatch(
 MessageActions.addMessage('Položićemo ga u februaru!!!'));

console.log(store.getState());
// ->
// { messages:
// ['Da li je IT255 težak predmet?',
// 'Nije težak, ali mora da se uči!!!',
// 'Položićemo ga u februaru!!!'] }

store.dispatch(MessageActions.deleteMessage(1));
```

```

console.log(store.getState());
// ->
// { messages:
// ['Da li je IT255 težak predmet?',
// 'Položićemo ga u februaru!!!!'] }

store.dispatch(MessageActions.deleteMessage(0));

console.log(store.getState());
// ->
// { messages: ['Položićemo ga u februaru!!!!'] }

```

## KREATORI AKCIJA KROZ DISKUSIJU I DEMONSTRACIJU

*Ukoliko je potrebno promeniti format poruke, to je moguće učiniti bez ažuriranja dispatch iskaza.*

Sada je sve lepše i funkcionalnije, a kao benefit je moguće navesti da ukoliko je potrebno promeniti format poruke, to je moguće učiniti bez potrebe za ažuriranjem dispatch iskaza. Na primer, potrebno je dodati i vreme kada je svaka od poruka iz niza kreirana. U tu svrhu je moguće dodati polje `created_at` u metodu `addMessage()` i sada će sve `AddMessageActions` dobiti polje `created_at`.

```

class MessageActions {
 static addMessage(message: string): AddMessageAction {
 return {
 type: 'ADD_MESSAGE',
 message,
 // nešto poput ovog
 created_at: new Date()
 };
 }
 //
}

```

Kao u prethodnim slučajevima, iz terminala se pokreće izvršavanje reduktora, čija stanja se sada čuvaju u skladištu `Store`, i dobija se izlaz kao na sledećoj slici:

```

PS C:\Users\Vlada\Documents\Angular\redux-chat\tutorial> ./node_modules/.bin/ts-node 08-action-creators.ts
{ messages: [] }
{ messages:
 ['Da li je IT255 težak predmet?',
 'Nije težak, ali mora da se uči!!!',
 'Položićemo ga u februaru!!!!'] }
{ messages:
 ['Da li je IT255 težak predmet?',
 'Položićemo ga u februaru!!!!'] }
{ messages: ['Položićemo ga u februaru!!!!'] }
PS C:\Users\Vlada\Documents\Angular\redux-chat\tutorial>

```

Slika 3.2 Primena kreatora akcija u Redux - u [izvor: autor]

## PRIMENA REALNE REDUX ARHITEKTURE

*Pošto je izgrađen vlastiti mini-redux veoma jednostavno se kreira realna Redux arhitektura.*

Pošto je izgrađen vlastiti mini-redux, postavlja se novo pitanje: "Šta je neophodno dodatno uraditi za implementaciju realne *Redux* arhitekture?" Odgovor je veoma jednostavan - ne mnogo! Prvo što je potrebno jeste ažuriranje prethodnog koga sa ciljem primene realne Redux arhitekture - vrši se importovanje *Action*, *Reducer* i *Store* iz paketa *redux* (datoteka: */redux-chat/tutorial/09-real-redux.ts*).

```
import {
 Action,
 Reducer,
 Store,
 createStore,
 AnyAction
} from 'redux';
```

U sledećem koraku, umesto specificiranja inicijalnog stanja prilikom kreiranja skladišta biće dozvoljeno reduktoru da sam kreira inicijalno stanje. Ovo će biti urađeno u formi podrazumevanog argumenta reduktora. Na ovaj način, ukoliko ne postoji stanje koje bi trebalo poslediti, koristiće se inicijalno stanje.

```
let initialState: AppState = { messages: [] };

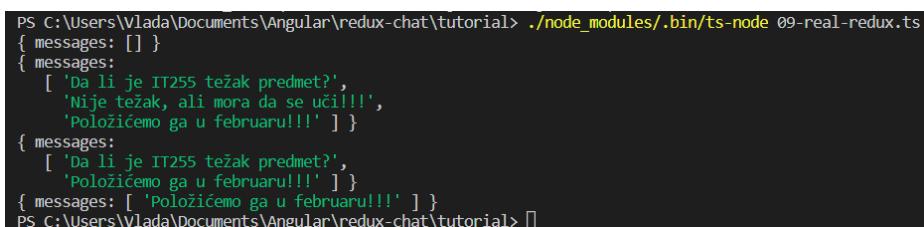
let reducer: Reducer<AppState> =
 (state: AppState = initialState, action: Action) => {
```

Ostatak reduktora je isti kao u prethodnom slučaju i neće biti razmatran.

Poslednja stvar koju bi trebalo kreirati jeste skladište primenom pomoćne metode *createStore()* iz *Redux*-a (<https://redux.js.org/api/createstore>).

```
// kreiranje novog skladišta
let store: Store<AppState> = createStore<AppState, AnyAction, AnyAction,
AnyAction>(reducer);
```

Sve ostalo je identično i funkcioniše na način prikazan sledećom slikom.



```
PS C:\Users\Vlada\Documents\Angular\redux-chat\tutorial> ./node_modules/.bin/ts-node 09-real-redux.ts
{ messages: [] }
{ messages:
 ['Da li je IT255 težak predmet?',
 'Nije težak, ali mora da se uči!!!',
 'Položićemo ga u februarul!!!'] }
{ messages:
 ['Da li je IT255 težak predmet?',
 'Položićemo ga u februarul!!!'] }
{ messages: ['Položićemo ga u februarul!!!'] }
PS C:\Users\Vlada\Documents\Angular\redux-chat\tutorial> []
```

Slika 3.3 Primena realne Redux arhitekture - demo [izvor: autor]

Nakon primene *Redux* arhitekture u izolaciji sve je spremno za njenu primenu u *Angular* veb aplikaciji i to je upravo cilj sledećeg izlaganja.

**Potpuno urađen primer primene Redux arhitekture u izolaciji preuzmite odmah posle ovog objekta učenja u aktivnosti Shared Resources.**

## ✓ Poglavlje 4

### Redux u Angular okviru

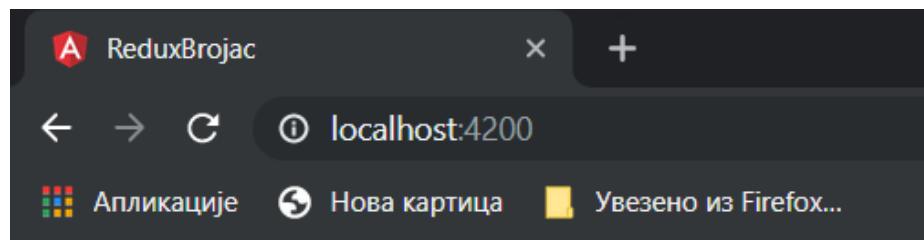
## UVOD U INTEGRACIJU REDUX - ANGULAR

*Integracija šabloni arhitekture podataka Redux sa radnim okvirom Angular.*

U prethodnim objektima učenja dat je presek i detaljna analiza [Redux](#) jezgra i pokazano je kako se kreiraju reduktori i koriste skladišta stanja za upravljanje podacima u izolaciji. Sada je trenutak kada je moguće preći na viši nivo koji podrazumeva integraciju šabloni arhitekture podataka [Redux](#) sa radnim okvirom za razvoj [frontend](#) veb aplikacija [Angular](#).

U ovom delu lekcije biće kreirana mala [Angular veb aplikacija](#) koja sadrži brojač čija vrednost može da se povećava ili smanjuje interakcijom korisnika i aplikacije preko odgovarajućih dugmadi "[Uvećaj](#)" i "[Smanji](#)".

Sledećom slikom je prikazan izgled aplikacije koja će predstavljati integraciju [Redux - Angular](#).



### Brojač

Tekuće skladište

Vrednost brojača je: **2**

[Uvećaj](#) [Smanji](#)

Slika 4.1.1 Izgled aplikacije za integraciju Redux - Angular [izvor: autor]

Primenom ovako male aplikacije moguće je usmeriti visok stepen pažnje na integraciju između [Redux](#) arhitekture podataka i [Angular](#) okvira. Sledi analiza i diskusija u vezi sa kreiranjem pokaznog primera veb aplikacije brojača.

## PLANIRANJE APLIKACIJE

*Postoje tri koraka planiranja kada je u pitanju primena Redux koncepata u veb aplikaciji.*

Kao što je bio slučaj primene Redux koncepata u izolaciji, **postoje tri koraka planiranja kada je u pitanju i primena Redux koncepata u veb aplikaciji**. Radi se o sledećim koracima:

1. Definisanje strukture stanja centralne aplikacije;
2. Definisanje akcija za upravljanje posmatranim stanjem;
3. Definisanje reduktora koji preuzimaju staro stanje i akciju i vraćaju novo stanje.

Za konkretnu veb aplikaciju, akcijama će biti izvođeno inkrementiranje i dekrementiranje brojača. Ovo je rađeno u izolaciji u prethodnom izlaganju, poznato je i trebalo bi ga podići na viši nivo i implementirati u *Angular* veb aplikaciji.

Sa druge strane kada se kreiraju *Angular* aplikacije na programerima je odluka gde i kako će kreirati komponente. U konkretno slučaju, postojeće komponenta najvišeg nivoa *AppComponent* čiji deo predstavlja pogled kojim se generiše stranica prikazana na slici 1.

Na najvišem nivou ove Angular aplikacije biće urađeno sledeće:

1. Kreiranje skladišta stanja *Store* koje će biti dostupno u celoj aplikaciji putem koncepta umetanja zavisnosti;
2. Prijavljivanje na promene u skladištu stanja *Store* i njihovo prikazivanje u komponentama;
3. Kada se javi odgovarajući događaj (klik na neko od dva dugmeta) prosleđuje se (*dispatch*) odgovarajuća akcija u skladište *Store*.

Ovim izlaganjem se završava planiranje i prelazi se na praktičan rad.

### ▼ 4.1 Podešavanje Redux - a

#### DEFINISANJE STANJA VEB APLIKACIJE

*U prvom koraku je neophodno kreirati adekvatan interfejs za definisanje stanja posmatrane veb aplikacije.*

U prvom koraku je neophodno kreirati adekvatan interfejs za definisanje stanja posmatrane veb aplikacije. Navedeni interfejs će dobiti naziv *AppState* i njegova definicija je priložena sledećim listingom (nov primer, datoteka: */redux-counter/src/app/app.state.ts*):

```
export interface AppState {
 counter: number;
};
```

Kreiranim interfejsom `AppState` je definisana osnovna struktura stanja - to je objekat sa jednim ključem (osobinom) `counter`. Za sada ovo je dovoljno i omogućava suštinsko sagledavanje primene `Redux` - a u veb aplikacijama.

## DEFINISANJE REDUKTORA

*Nakon što je definisano centralno stanje, moguće je pristupiti procesu definisanja reduktora.*

Nakon što je definisano centralno stanje za posmatranu veb aplikaciju, moguće je pristupiti procesu definisanja reduktora. Zadatak reduktora, koji se kreiraju za posmatranu veb aplikaciju, biće rukovanje akcijama inkrementiranja i dekrementiranja brojača. Javljanje ovih akcija će biti direktna posledica korisnikove interakcije da dugmadima koji su kreirani u šablonu glavne komponente aplikacije.

U datoteci `/redux-counter/src/app/counter.reducer.ts`, primera koji se dalje razvija, biće ugrađen kod priložen sledećim listingom:

```
import {
 INCREMENT,
 DECREMENT
} from './counter.actions';

const initialState: AppState = { counter: 0 };

// kreiranje vlastitog reduktora za upravljanje promenama stanja
export const counterReducer: Reducer<AppState> =
 (state: AppState = initialState, action: Action): AppState => {
 switch (action.type) {
 case INCREMENT:
 return Object.assign({}, state, { counter: state.counter + 1 });
 case DECREMENT:
 return Object.assign({}, state, { counter: state.counter - 1 });
 default:
 return state;
 }
 };
```

Kreiranje koda reduktora je započelo importovanjem konstanti `INCREMENT` i `DECREMENT` iz kreiranih kreatora akcija (`actioncreators`).

Početno stanje je `AppState` koje podešava brojač na vrednost 0.

Reduktor `counterReducer` rukuje dvema akcijama:

- `INCREMENT`, koja dodaje vrednost 1 na trenutnu vrednost brojača;
- `DECREMENT`, koja oduzima vrednost 1 od trenutne vrednosti brojača.

Obe akcije koriste `Object.assign` za obezbeđivanje da ne dođe do mutacije starog stanja, umesto čega će biti kreiran nov objekat koji će biti vraćen kao novo stanje.

Pošto je sve uspešno obavljeno, sada je moguće preći na definisanje kreatora akcija.

## DEFINISANJE KREATORA AKCIJA

*Nakon definisanja stanja i reduktora sada je moguće preći na definisanje kreatora akcija.*

Ako se vratimo na urađeni posao, urađeno je sledeće:

- kreiran je adekvatan interfejs za definisanje stanja posmatrane veb aplikacije;
- definisan je reduktor za rukovanje akcijama inkrementiranja i dekrementiranja brojača.

Pošto su svi nabrojani zadaci uspešno obavljeni, sada je moguće preći na definisanje kreatora akcija.

U konkretnom slučaju, kreatori akcija će biti funkcije koje vraćaju objekte koji definišu akcije koje je neophodno izvesti. Akcije *increment* i *decrement*, priložene listingom koji sledi, vraćaju objekat koji definiše odgovarajuću osobinu *type*.

Kreatori akcija su definisani u datoteci */redux-counter/src/app/counter.actions.ts*, aktuelnog primera, i dati su sledećim listingom:

```
import {
 Action,
 ActionCreator
} from 'redux';

export const INCREMENT: string = 'INCREMENT';
export const increment: ActionCreator<Action> = () => ({
 type: INCREMENT
});

export const DECREMENT: string = 'DECREMENT';
export const decrement: ActionCreator<Action> = () => ({
 type: DECREMENT
});
```

Potrebno je primetiti da funkcija kreatora akcije vraća tip *ActionCreator<Action>*. *ActionCreator* je generička klasa definisana od strane *Redux* - a koja se koristi za definisanje funkcija koje kreiraju akcije. U konkretnom slučaju, koristi se klasa *Action*, ali u izvesnim situacijama moguće je koristiti i specifične *Action* klase, kao što je *AddMessageAction* koja je prikazana u prethodnom izlaganju..

## KREIRANJE SKLADIŠTA I PRIMENA REDUX DEVTOOLS

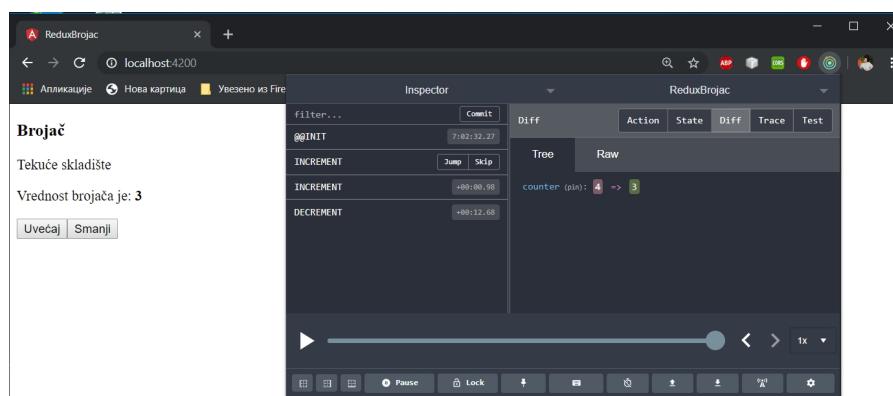
*Postojanje reduktora i stanja je ispunjen uslov za kreiranje skladišta stanja.*

Postojanje reduktora i stanja je ispunjen uslov za kreiranje skladišta stanja. To je do sada rađeno na sledeći način:

```
let store: Store<AppState> = createStore<AppState>(counterReducer);
```

Međutim, *Redux* poseduje jako bogat skup razvojnih alata što ga čini još kvalitetnijim za upotrebu prilikom razvoja veb aplikacija primenom *Angular* okvira.

Posebno za veb pregledač *Google Chrome* je razvijen dodatak (ekstenzija - eng. extension) koji omogućava praćenje stanja pokrenute aplikacije i angažovanje akcija. Instalacija dodatka *Redux DevTools* dostupna je putem linka: <https://chrome.google.com/webstore/detail/redux-devtools/lmhkpmbekcpmknklioeibfkpmffibljd?hl=en>.



Slika 4.2.1 Praćenje izvravanja Redux aplikacije preko Redux DevTools [izvor: autor]

Ono što je veoma pogodno prilikom korišćenja alata *Redux DevTools* jeste što pruža jasnu sliku u vezi sa svakom akcijom koja teče kroz sistem i utiče na promenu stanja. Na sledećem linku je moguće dobiti više informacija u vezi sa korišćenjem alata *Redux DevTools*: <https://chrome.google.com/webstore/detail/redux-devtools/lmhkpmbekcpmknklioeibfkpmffibljd?hl=en>.

Sa ciljem korišćenja navedenog razvojnog alata, neophodno je uraditi jednu stvar - obaviti njegovo dodavanje unutar koda datoteke skladišta stanja (u konkretnom primeru: [/redux-counter/src/app/app.store.ts](#)):

```
const devtools: StoreEnhancer<AppState> =
 window['devToolsExtension'] ?
 window['devToolsExtension']() : f => f;
```

Važno je napomenuti da za funkcionisanje ove aplikacije nije potrebno obaviti instaliranje alata *Redux DevTools*, on je tu da pomogne i olaška razvoj i praćenje aplikacije. Sledеćim pozivom: `window.devToolsExtension`, proverava da li u pregledaču *Google Chrome* postoji podrška za *Redux DevTools* koja će, ukoliko postoji, biti upotrebljena. Ukoliko ne postoji, prostо će biti vraćena identična funkcija (`f => f`) koja će vratiti identičan rezultat koji joj je na ulazu prosleđen.

## REDUX DEVTOOLS I SREDNJI SLOJ

*Alat Redux Devtools je jedna od brojnih dostupnih biblioteka srednjeg sloja za Redux.*

*Srednji sloj (Middleware)* je izraz za funkciju koja proširuje funkcionalnosti neke druge biblioteke. Alat *Redux Devtools* je jedna od brojnih dostupnih biblioteka srednjeg sloja za *Redux* (više informacija studenti mogu pronaći na sledećem linku: <https://redux.js.org/advanced/middleware>). *Redux* podržava veliki broj ovakvih biblioteka, a veoma je lako kreirati i vlastitu.

Sa ciljem korišćenja prikazanog razvojnog alata, neophodno ga je dodati kao srednji sloj u kod datoteke skladišta stanja (*/redux-counter/src/app/app.store.ts*):

```
export function createAppStore(): Store<AppState> {
 return createStore<AppState>(
 reducer,
 compose(devtools)
);
}
```

Sada je, nakon implementacije prikazanog listinga, za svaku prosleđenu akciju i svako promenjeno stanje, u veb pregledaču moguće pratiti šta se dešava (kao na slici 1).

## DODAVANJE SKLADIŠTA STANJA

*Skladište stanja preko DI mora da bude dostupna bilo gde u aplikaciji gde se za njom ukaže potreba.*

U prethodnim koracima je obavljeno podešavanje jezgra *Redux* arhitekture podataka, pa je sada moguće usmeriti pažnju na *Angular* komponente. Od posebnog značaja za ovu aplikaciju je definisanje i kreiranje glavne komponente aplikacije *AppComponent* koja će biti upotrebljena da pokrene celokupnu aplikaciju i poveže je sa *Angular* - om.

Budući da aplikacija, koja služi ovde kao pokazni primer, predstavlja *Redux* aplikaciju, **biće potrebno obezbediti da instanca skladišta stanja bude dostupna bilo gde u aplikaciji gde se za njom ukaže potreba**. Navedeno se ostvaruje primenom poznatog koncepta, čiji naziv glasi: **umetanje zavisnosti (dependency injection - DI)**.

Ako se obavi prisećanje na lekciju koja se bavila umetanjem zavisnosti, neophodno je obezbediti neku vrednost u aplikaciji primenom ovog koncepta pa je, u tu svrhu, potrebno upotrebiti podešavanje provajdera dodavanjem u listu provajdera u *@NgModule* dekoratoru.

Kada se nešto čini dostupnim za primenu u DI sistemu, specificiraju se sledeće stvari:

- *token* koji se koristi da ukaže na konkretnu zavisnost koju je potrebno umetnuti;
- način umetanja zavisnosti.

Veoma često, ukoliko je cilj obezbeđivanje *singleton* servisa koristi se opcija *useClass* na sledeći način:

```
{ provide: SpotifyService, useClass: SpotifyService }
```

U prikazanom slučaju, koristi se klasa *SpotifyService* kao token u DI sistemu. Opcija *useClass* ukazuje *Angular* - u da kreira instancu klase *SpotifyService* i de je ponovo upotrebi svaki put kada je zahtevano umetanje servisa *SpotifyService*.

Problem sa ovakvim pristupom, u konkretnom slučaju, je što se ne želi da *Angular* kreira skladište stanja - to je urađeno u jednom od prethodnih listinga sa *createStore()*. Ovde je cilj korišćenje postojećeg skladišta. To se postiže primenom opcije *useValue*. Ova opcija je i ranije korišćena sa podesivim vrednostima, na primer na sledeći način:

```
{ provide: API_URL, useValue: 'http://localhost/api' }
```

## PRIMENA TOKENA SKLADIŠTA STANJA

*Nije moguće koristiti interfejs kao ključ za umetanje zavisnosti.*

Nastavlja se diskusija započeta u prethodnoj sekciji. Sledeći problem koji je potrebno rešiti jeste određivanje tokena koji se koristi prilikom umetanja zavisnosti. Tip konkretnog skladišta stanja je *Store<AppState>* (dopuna koda datoteke: */redux-counter/src/app/app.store.ts*):

```
export function createAppStore(): Store<AppState> {
 return createStore<AppState>(
 reducer,
 compose(devtools)
);
}

export const appStoreProviders = [
 { provide: AppStore, useFactory: createAppStore }
];
```

*Store* je interfejs, nije klasa u pravom smislu reči, i nije moguće koristiti interfejs kao ključ za umetanje zavisnosti. To znači da je neophodno kreirati vlastiti token koji će biti upotrebljen za umetanje skladišta. I ovaj problem je lako rešiv u *Angular* - u. Neophodno je kreirati token u posebnoj datoteci koju je moguće importovati na bilo kojem mestu u aplikaciji (dopuna koda datoteke: */redux-counter/src/app/app.store.ts*):

```
export const AppStore = new InjectionToken('App.store');
```

Dakle, kreirana je konstanta *AppStore* koja koristi *Angular* klasu *InjectionToken*. Sada je moguće upotrebiti navedeni token pod ključem *provide* (listing sa početka sekcije - linija koda 9).

## NGMODULE KLASA APLIKACIJE

*Definisanje dekoratora @NgModule koji vrši ključnu ulogu u pokretanju Angular aplikacije*

U nastavku će akcenat biti isključivo na Angular konceptima. Prelazi se na datoteku `app.module.ts`, i definisanje dekoratora `@NgModule` koji vrši ključnu ulogu u pokretanju naše `Angular / Redux` aplikacije:

```
import { BrowserModule } from '@angular/platform-browser';
import { NgModule } from '@angular/core';
import { FormsModule } from '@angular/forms';
import { HttpClientModule } from '@angular/http';

import { appStoreProviders } from './app.store';

import { AppComponent } from './app.component';

@NgModule({
 declarations: [
 AppComponent
],
 imports: [
 BrowserModule,
 FormsModule,
 HttpClientModule
],
 providers: [appStoreProviders],
 bootstrap: [AppComponent]
})
export class AppModule { }
```

Sada kada je omogućeno dobijanje reference za `Redux` skladište bilo gde u aplikaciji, putem umetanja zavisnosti, neophodno je uputiti se ka mestu gde je ona najpotrebnija - ka komponenti `AppComponent`.

## ▼ 4.2 Angular komponenta AppComponent

### IMPORT INSTRUKCIJE U KOMPONENTI APPCOMPONENT

*Reference za Redux skladište je ona najpotrebnija u komponenti AppComponent.*

Veoma bitna stvar je istaknuta na kraju prethodne sekcije - sada kada je omogućeno dobijanje reference za `Redux` skladište bilo gde u aplikaciji, putem umetanja zavisnosti, neophodno je uputiti se ka mestu gde je ona najpotrebnija - ka `Angular` komponenti `AppComponent`.

Definisanje sadržaja klase *AppComponent* (datoteka primera: /redux-counter/src/app/app.component.ts) započinje *import* instrukcijama kao u sledećem listingu:

```
import { Component, Inject } from '@angular/core';
import { Store } from 'redux';
import { AppStore } from './app.store';
import { AppState } from './app.state';
import * as CounterActions from './counter.actions';
```

U glavnu komponentu su na ovaj način dodate sledeće stvari:

- interfejs *Store* iz *Redux* - a;
- token *AppStore* koji omogućava dobijanje reference na singleton instancu skladišta;
- *AppState* tip koji omogućava sagledavanje strukture centralnog stanja;
- kreatore akcija preko izraza "*\* as CounterActions*". Ova sintaksa, na primer, omogućava poziv *CounterActions.increment()* za kreiranje *INCREMENT* akcije.

## KONSTRUKTOR U KOMPONENTI APPCOMPONENT

*Konstruktor je mesto u klasi komponente gde se dešava umetanje zavisnosti Redux skladišta.*

Konstruktor je mesto u klasi komponente gde se, po pravilu, dešava umetanje zavisnosti. Klasa *AppComponent* je zavisna od skladišta *Store*, pa je unutar konstruktora neophodno obaviti umetanje zavisnosti. Na sledeći način je kreirani token *AppStore* dodat u komponentu *AppComponent* preko umetanja zavisnosti:

```
import { Component, Inject } from '@angular/core';
import { Store } from 'redux';
import { AppStore } from './app.store';
import { AppState } from './app.state';
import * as CounterActions from './counter.actions';

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

 constructor(@Inject(AppStore) private store: Store<AppState>) {
 store.subscribe(() => this.readState());
 this.readState();
 }

 readState() {
 const state: AppState = this.store.getState() as AppState;
 this.counter = state.counter;
 }
}
```

```
increment() {
 this.store.dispatch(CounterActions.increment());
}

decrement() {
 this.store.dispatch(CounterActions.decrement());
}
}
```

Iz listinga je moguće primetiti upotrebu `@Inject` dekoratora za umetanje tokena `AppStore` - takođe definisan je i tip promenljive skladišta sa `Store<AppState>`. Privatna objektna promenljiva `store` je podešena kao skladište i sada je, u nastavku, moguće osluškivati promene. U tom svetlu se poziva `store.subscribe` i `this.readState()`.

Skladište (objekat `store`) će pozvati `subscribe()` samo u slučaju kada je pokrenuta nova akcija, pa u ovom slučaju mora da bude obezbeđen manuelni poziv metode `readState()`, najmanje jedanput, da bi se obezbedilo da komponenta dobije inicijalne podatke.

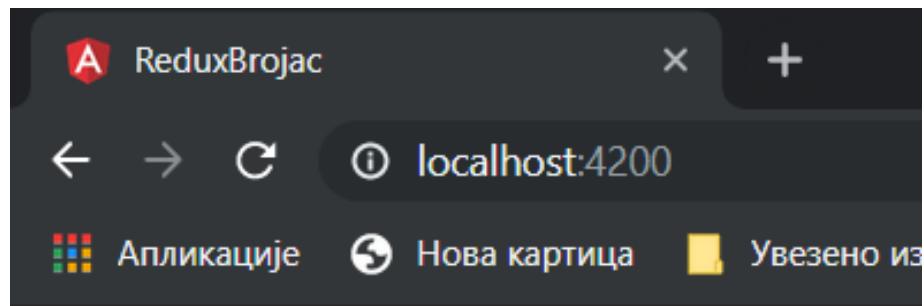
Metoda `readState()` čita iz skladišta i ažurira `this.counter` na tekuću vrednost. Pošto je `this.counter` osobina klase komponente koja je povezana u pogledu (obrađeno u sledećoj sekciji), `Angular` će detektovati promene i izvršiti ažuriranje pogleda.

Konačno, definisane su i dve pomoćne metode: `increment()` i `decrement()`, čiji je zadatak slanje odgovarajućih akcija ka skladištu stanja.

## ŠABLON KOMPONENTE APPCOMPONENT

*Za kraj primera je prikazan pogled generisan primenom šablona komponente AppComponent.*

Sledećom slikom je prikazan pogled generisan primenom šablona komponente `AppComponent`.



## Brojač

Tekuće skladište

Vrednost brojača je: 3

**Uvećaj** **Smanji**

Slika 4.3.1 Pogled generisan primenom šablona komponente [izvor: autor]

Sledi listing datoteke */redux-counter/src/app/app.component.html*:

```
<div class="row">
 <div class="col-sm-6 col-md-4">
 <div class="thumbnail">
 <div class="caption">
 <h3>Brojač</h3>
 <p>Текуће складиšте</p>

 <p>
 Вредност бројача је:
 {{ counter }}
 </p>

 <p>
 <button (click)="increment()" class="btn btn-primary">
 Увећай
 </button>
 <button (click)="decrement()" class="btn btn-default">
 Смањи
 </button>
 </p>
 </div>
 </div>
 </div>
</div>
```

```
</div>
</div>
```

Ovde je važno обратити pažnju на tri stvari:

- prikazivanje vrednosti бројача преко израза `{{ counter }}`;
- pozivanje функције `increment()` преко дугмента;
- pozivanje функције `decrement()` преко дугмента.

**Preuzmite урађени пример на kraju ovog objekta учења из активности Shared Resources.**

## VIDEO MATERIJAL

*Izlaganje lekcije ће бити заокруžено изабраним video materijalima*

**Ngrx Store Tutorial for Angular - Learn State Management for Angular** - trajanje 27:46.

**Ova lekcija садржи video materijal. Уколико ћете да pogledate ovaj video morate da otvorite LAMS lekciju.**

**Angular ngrx Redux Quick Start Tutorial** - trajanje: 9:00

**Ova lekcija садржи video materijal. Уколико ћете да pogledate ovaj video morate da otvorite LAMS lekciju.**

## ✓ Poglavlje 5

### Dodatni materijali za rad

#### DODATNI MATERIJALI

*Proširivanje znanja stečenog na predavanjima.*

Dopunite znanje kroz izučavanje sledećih materijala:

1. <https://redux.js.org/>
2. <https://redux.js.org/api/createstore>
3. <https://redux.js.org/advanced/middleware>

## ✓ Poglavlje 6

### Vežba 13

#### POKAZNA VEŽBA IT255-V13 (TRAJANJE 45 MINUTA)

*U ovoj vežbi biće obrađen ngrx, odnosno menadžer stanja aplikacije.*

**NGRX je grupa biblioteka “inspirisanih” Redux šablonom.**

U ovoj vežbi biće obrađen [ngrx](https://ngrx.io/) (<https://ngrx.io/>), odnosno menadžer stanja aplikacije. Kako se radi o specifičnom paketu, i stoga je neophodno da instaliramo u okviru naše aplikacije sledećom komandom:

```
npm install @ngrx/store --save
```

Po uspešnoj instalaciji unutar projekta kreiraćemo folder store, unutar app foldera, gde ćemo smestiti naše akcije, efekte, reduktore, selektore, stanja i slično.

Takođe, neophodno je i da kreiramo 2 interfejsa, sa nazivima:

- [Post](#) i
- [PostHttp](#).

[Post](#) predstavlja jedan resurs postova, dok [PostHttp](#) predstavlja entitet dovučen sa servera, odnosno listu postova.

```
ng g i interfaces/post
ng g i interfaces/post-http
```

Sadržaj kreiranih fajlova će biti sledeći: post.ts

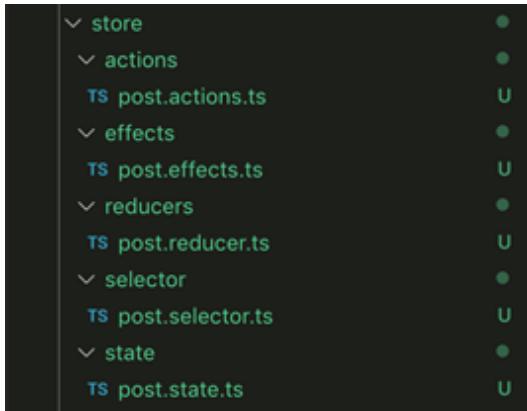
#### **post.ts**

```
export interface Post {
 id: number;
 title:string;
 body:string;
}
```

#### **post-http.ts**

```
export interface PostHttp {
 posts: []
}
```

Finalni izgled našeg store foldera dat je na slici ispod:



Slika 6.1 Finalni izgled našeg store foldera [izvor: autor]

## STANJA I REDUKTORI

*Krećemo od stanja aplikacije na čijem nivou ćemo čuvati sve učitane postove*

Krećemo od stanja aplikacije na čijem nivou ćemo čuvati sve učitane postove i gde ćemo čuvati stanje istih.

Kod stanja dat je ispod:

```
import { Post } from '../interfaces/post';

export interface PostState {
 posts: Array<Post>;
}
export const initialPostsState: PostState = {
 posts: []
};
```

Potom dolazimo do selektora koji bi trebao da nam vrati stanje u vidu niza i to je navedeno u delu koda ispod:

### Post.selector.ts

```
import { createSelector } from '@ngrx/store';

const selectPosts = (state: any) => state.post;

export const selectedPosts = createSelector(
 selectPosts,
 (state: any) => {
```

```
 return state.posts.posts;
 }
);
```

Reduktor koji nam vraća stanje aplikacije koje ne može biti promenjeno, naveden je u delu koda ispod:

```
import { PostActions, EnumPostAction } from '../actions/post.actions';
import { initialPostsState } from '../state/post.state';

export function postReducer (
 state = initialPostsState,
 action: PostActions
){
 switch (action.type) {
 case EnumPostAction.GetPostsSuccess: {
 return {
 ...state,
 posts: action.payload
 };
 }

 default:
 return state;
 }
};
```

## POZIV SERVISA ZA DOVLAČENJE POSTOVA

*PostEffect* klasa vrši poziv servisa za dovlačenje postova putem *Http*.

*PostEffect* klasa vrši poziv servisa za dovlačenje postova putem *Http* - a. Listing klase je naveden je u delu koji sledi:

```
import { Injectable } from "@angular/core";
import { Actions, Effect, ofType } from "@ngrx/effects";
import { of } from "rxjs";
import { GetPosts, EnumPostAction, GetPostsSuccess} from '../actions/post.actions';
import { switchMap } from 'rxjs/operators';
import { Post } from '../interfaces/post';
import { PostService } from '../services/post.service';

@Injectable()
export class PostEffect {
 constructor(private _actions$: Actions,
 private _postService : PostService) {

 }

 @Effect()
 getPosts$ = this._actions$.pipe(
```

```
ofType<GetPosts>(EnumPostAction.GetPosts),
switchMap(() => this._postService.fetchPosts()),
switchMap((postHttp: Post[]) => {
 return of(new GetPostsSuccess(postHttp))
})
};

}
```

Akcija koja se poziva po uspešnom dovlačenju posta navedena je ispod:

```
import { Action } from '@ngrx/store';
import { Post } from '../interfaces/post';

export enum EnumPostAction {
 GetPosts = '[Post] Get Posts',
 GetPostsSuccess = '[Post] Get Posts Success'
}

export class GetPosts implements Action {
 public readonly type = EnumPostAction.GetPosts;
}

export class GetPostsSuccess implements Action {
 public readonly type = EnumPostAction.GetPostsSuccess;
 constructor(public payload: Post[]) {}
}

export type PostActions = GetPosts | GetPostsSuccess;
```

## SERVISNA KLASA POSTSERVICE

*Servis aplikacije za dovlačenje postova kroz Http i smeštanje istih unutar skladišta (store).*

Sam servis aplikacije za dovlačenje postova kroz [Http](#) i smeštanje istih unutar skladišta ([store](#)), vrši se kroz [post.service](#), čiji je finalni izgled koda naveden ispod:

```
import { HttpClient } from '@angular/common/http';
import { Observable } from 'rxjs';
import { Injectable } from '@angular/core';
import { Post } from '../interfaces/post';
import { select, Store } from '@ngrx/store';
import { PostState } from '../store/state/post.state';
import { selectedPosts } from '../store/selector/post.selector';

@Injectable()
export class PostService {
 public posts$: Observable<Post[]>;
```

```
public constructor(private _http: HttpClient, private _store: Store<PostState>)
{
 this.posts$ = this._store.pipe(select(selectedPosts));

}
public fetchPosts(): Observable<[Post]> {
 return this._http.get<[Post]>('https://jsonplaceholder.typicode.com/posts');
}

}
```

## GLAVNA KOMPONENTA APPCOMPONENT

*Pozvaćemo selector za postove, kao i slanje (dispatch) akcije za dovlačenje postova.*

Unutar `app.component.ts` fajla, pozvaćemo selektor za postove, kao i slanje (`dispatch`) akcije za dovlačenje postova. Prilikom inicijalizacije aplikacije, neophodno je da pozovemo akciju `GetPosts`, na sledeći način:

```
ngOnInit() {
 this._store.dispatch(new GetPosts());
}
```

Dalje, unutar konstruktora ćemo izvršiti dovlačenje postova iz skladišta na sledeći način:

```
public posts$: Observable<Post>;
title = 'IT255-v12';

constructor(private _store: Store<PostState>, private _router: Router) {
 this.posts$ = this._store.pipe(select(selectedPosts));
}
```

Krajnji izgled `app.component.ts` fajla je:

```
import { Component } from '@angular/core';
import { Observable } from 'rxjs';
import { Post } from './interfaces/post';
import { Store, select } from '@ngrx/store';
import { PostState } from './store/state/post.state';
import { Router } from '@angular/router';
import { GetPosts } from './store/actions/post.actions';
import { selectedPosts } from './store(selector/post.selector';

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

```
export class AppComponent {
 public posts$: Observable<Post>;
 title = 'IT255-v12';

 constructor(private _store: Store<PostState>, private _router: Router) {
 this.posts$ = this._store.pipe(select(selectedPosts));
 }

 ngOnInit() {
 this._store.dispatch(new GetPosts());
 }
}
```

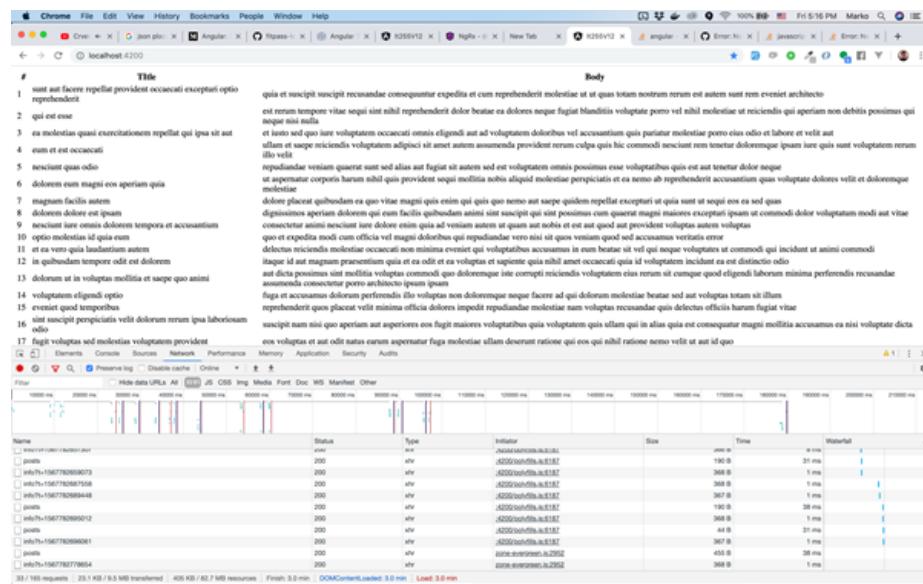
## GLAVNA KOMPONENTA APPCOMPONENT, ŠABLON I DEMO

*Kreiranjem pogleda završava se rad na pokaznom primeru.*

Kreiranjem pogleda završava se rad na pokaznom primeru. Krajnji izgled [app.component.html](#) fajla je naveden ispod:

```
<table class="table table-hover">
 <thead class="thead-dark">
 <tr>
 <th scope="col">#</th>
 <th scope="col">Title</th>
 <th scope="col">Body</th>
 </tr>
 </thead>
 <tbody>
 <tr *ngFor="let post of posts$ | async">
 <td>{{ post.id }}</td>
 <td>{{ post.title }}</td>
 <td>{{ post.body }}</td>
 </tr>
 </tbody>
</table>
```

Finalni izgled aplikacije dat je na slici ispod:



Slika 6.2 Finalni izgled aplikacije [izvor: autor]

## ✓ Poglavlje 7

### Individualna vežba 13

#### INDIVIDUALNA VEŽBA (TRAJANJE 90 MINUTA)

*Modifikovati projekat tekući tako da koristi ngrx (Redux) skladište.*

Koristeći znanje stečeno na predavanjima i vežbama pokušajte samostalno da uradite Angular veb aplikaciju na osnovu sledećih zahteva:

1. Kreirati projekat tako da koristi *ngrx (Redux)* skladište ,
2. Omogućiti čitanje iz skladišta (store), kao i upisivanje u isto.

Za primenu koncepta *ngrx* prilaže se sledeći korisni linkovi:

- <https://medium.com/codingthesmartway-com-blog/angular-and-redux-ecd22ea53492> (primer 1)
- <https://www.javatpoint.com/angular-redux> (primer 2) <https://www.npmjs.com/package/ng-redux> (Angular instalacija)
- <https://ngrx.io/>;
- <https://ngrx.io/docs>

Nakon urađene vežbe pozovite predmetnog asistenta i demonstrirajte funkcionisanje uvedenih funkcionalnosti u aplikaciju.

## ✓ Poglavlje 8

### Domaći zadatak 13

#### DOMAĆI ZADATAK (PREDVIĐENO VREME 120 MIN)

*Samostalna izrada domaćeg zadatka sa servisima i pogledima na osnovu obrađenih materijala predavanja i vežbi.*

Uradite domaći zadatka prema sledećim zahtevima:

1. Pronaći domaći zadatak *MetHotels* iz prethodnih vežbi i domaćih zadataka;
2. Domaći zadatak modifikovati tako da koristi *ngrx* skladište (*store*)
3. Omogućiti upisivanje stanja modifikovane aplikacije unutar skladišta.
4. Kao u primeru sa individualnih vežbi omogućite da se napravi lista zadataka, npr: očistiti sobu, rezervisati sobu, dostaviti hranu u sobu i slično.

Domaći zadatak dodati na *Github* pod "commit - om" *IT255-DZ13* i poslati obaveštenje predmetnom asistentu o postavljenom domaćem zadatku.

## ▼ Poglavlje 9

### Zaključak

## ZAKLJUČAK

*Predmet izučavanja lekcije je bio šablon arhitekture podataka pod nazivom Redux.*

U ovoj lekciji akcenat je bio na izučavanju šablona arhitekture podataka pod nazivom *Redux*. U tom svetlu, prva razmatranja su bila vezana za diskusiju o idejama koje stoje iza primene *Redux*-a, pisanje mini verzije aplikacije, a zatim njeno povezivanje sa *Angular* okvirom.

Lekcija je stavila do znanja da je u većini dosadašnjih projekata, upravljanje stanjima je obavljano na prilično direktn način:

- uzimaju se podaci iz servisa;
- podaci se obrađuju u komponentama i prosleđuju kroz stablo komponenata.

Zbog navedenog je jasno izvučen zaključak da ovakav vid upravljanja aplikacijom može biti odličan za male aplikacije, ali kako aplikacije rastu, dobijaju sve veći broj komponenata, upravljanje različitim stanjima aplikacije može biti veoma složen i težak posao.

Uočeno je da prosleđivanje podataka niz stablo komponenata može biti praćeno sledećim problemima:

- *posredno prosleđivanje osobina;*
- *nefleksibilni refaktoring;*
- *stablo stanja (state tree) i DOM stablo (DOM tree) se ne podudaraju; posredno prosleđivanje osobina;*
- *praćenje stanja aplikacije u celini .*

Jasno je zaključeno da manipulisanje podacima primenom servisa i komponenta pomaže puno u razvoju veb aplikacija. Međutim, otvoren je i nov problem - ako se servisi sagledaju kao vlasnici podataka, postavlja se novo pitanje: "koja je najbolja praksa za podatke u vlasništvu servisa? Da li postoji pogodan šablon za implementaciju rešenja?" Odgovor je: "Da, postoji!"

Upravo zbog svega navedenog lekcija je predložila novo rešenje sa akcentom na šablonu arhitekture podataka poznatog pod nazivom *Redux* koji je od velike pomoći za rešavanje navedenog problema.

Savladavanjem navedene lekcije studenti su osposobljeni ka koriste redukciju u Angular aplikacijama i na taj način značajno smanje količinu pisanog koda.

## LITERATURA

*Za pripremu lekcije korišćena je aktuelna pisana i elektronska literatura.*

### **Pisana literatura:**

1. Nate Murray, Felipe Coury, Ari Lerner, Carlos Taborda, ng-Book – The Complete Book on Angular 6, Fullstack.io, 2018
2. Cody Lindley, Frontend Handbook – 2017, Frontend masters, 2017
3. Sandeep Panda, AngularJS – From Novice to Ninja, SitePoint Pty. Ltd, 2014

### **Elektronska literatura:**

4. <https://angular.io/>
5. <https://angular.io/tutorial>
6. <https://www.w3schools.com/angular/>
7. <https://www.tutorialspoint.com/angular4/>
8. <https://nodejs.org/en/>
9. <https://code.visualstudio.com/>
10. <https://redux.js.org/>
11. <https://redux.js.org/api/createstore>
12. <https://redux.js.org/advanced/middleware>



IT255 - VEB SISTEMI 1

## Testiranje veb aplikacije

Lekcija 14

PRIRUČNIK ZA STUDENTE

# IT255 - VEB SISTEMI 1

## Lekcija 14

### ***TESTIRANJE VEB APLIKACIJE***

- ✓ Testiranje veb aplikacije
- ✓ Poglavlje 1: Izvođenje testova – End-to-end ili Unit testiranje
- ✓ Poglavlje 2: Podešavanje testiranja
- ✓ Poglavlje 3: Testiranje servisa i HTTP zahteva
- ✓ Poglavlje 4: Testiranje rutiranja ka komponentama
- ✓ Poglavlje 5: Testiranje formi
- ✓ Poglavlje 6: Dodatni materijali za rad
- ✓ Poglavlje 7: Pokazna vežba 14, testiranje HTTP zahteva
- ✓ Poglavlje 8: Individualna vežba 14
- ✓ Poglavlje 9: Domaći zadatak 14
- ✓ Zaključak

Copyright © 2017 – UNIVERZITET METROPOLITAN, Beograd. Sva prava zadržana. Bez prethodne pismene dozvole od strane Univerziteta METROPOLITAN zabranjena je reprodukcija, transfer, distribucija ili memorisanje nekog dela ili čitavih sadržaja ovog dokumenta., kopiranjem, snimanjem, elektronskim putem, skeniranjem ili na bilo koji drugi način.

Copyright © 2017 BELGRADE METROPOLITAN UNIVERSITY. All rights reserved. No part of this publication may be reproduced, stored in a retrieval system or transmitted in any form or by any means, electronic, mechanical, photocopying, recording, scanning or otherwise, without the prior written permission of Belgrade Metropolitan University.

## ▼ Uvod

### UVOD

*Lekcija 14 se bavi načinima i mehanizmina za testiranje kreiranih veb aplikacija pre produkcije.*

Kada se utroše sati, dani i meseci za razvoj veb aplikacije, pojavi se, konačno, i trenutak kada je potrebno objaviti aplikaciju korisnicima i pustiti je u produkciju. Mnoštvo napornog rada i utrošenog vremena bi, konačno, trebalo da se isplati. Međutim, u praksi baš i ne ide sve glatko. Na primer, korisnik pokrene aplikaciju i, desi se nešto što niko iz razvojnog tima nije predviđao, blokirajuća greška u programu koja onemogućava korisniku da se prijavi na aplikaciju .

*Testiranje* aplikacije pomaže proces otkrivanja grešaka pre nego što se one i pojave, povećava stepen poverenja u kreiranu veb aplikaciju i olakšava uvođenje novih programera u dalji razvoj aplikacije.

Istina, među profesionalcima postoji podeljeno mišljenje o moći i značaju procesa testiranja tokom projekta razvoja nekog softverskog rešenja. Dobra stvar je da u vezi sa navedenom problematikom teče neprekidna debata koja, sasvim sigurno, dovodi do sve kvalitetnijih test alata i tehnika, a posledično i do kvalitetnijih softverskih rešenja.

Kroz debatovanje, najčešće se otvaraju pitanja poput sledećih: "Da li je bolje prvo pisati testove, a zatim kreirati implementaciju koja omogućava da kreirani testovi prođu ili je bolje obaviti prvo kodiranje, a tek nakon toga validaciju koda?"

Možda je najbolja praksa, a možda će neko od kolega misliti drugačije, za veb aplikacije je baziranje na kreiranju koda pogodnog za testiranje, pogotovo ako je u pozadini bogato iskustvo prototipskog pristupa razvoju aplikacija.

Iako iskustvo može da varira, pokazuje se dok se koristi pristup razvoju aplikacije baziran na prototipu, testiranje pojedinačnih delova koda, koji se verovatno menjaju, može da udvostruči ili utrostruči količinu posla potrebnog da se njihov razvoj nastavi. Suprotno ovome, fokus je na razvoju aplikacije sa malim komponentama i držanjem velikog broja funkcionalnosti podeljenih u nekoliko metoda koje omogućavaju testiranje funkcionalnosti kao dela šire slike. Upravo se na ovo mislilo kada je pomenut termin koda pogodnog za testiranje (testable code).

Za lekciju će od posebnog značaja biti sledeće teme:

- *End-to-end ili Unit Testing?*
- Test alati: *Jasmine, Karma*
- Pisanje jediničnih testova
- *Angular* okvir za jedinično testiranje
- Podešavanje testiranja
- Testiranje servisa i *HTTP* zahteva

Cilj je da student bude osposobljen sa samostalno piše i izvodi testove *frontend Angular* veb aplikacija.

## UVODNI VIDEO

*Trajanje video snimka: 4min 32sek*

**Ova lekcija sadrži video materijal. Ukoliko želite da pogledate ovaj video morate da otvorite LAMS lekciju.**

## ▼ Poglavlje 1

# Izvođenje testova – End-to-end ili Unit testiranje

## END-TO-END ILI JEDINIČNO TESTIRANJE

*Postoje dva glavna načina testiranja aplikacija: End-to-end i jedinično (unit) testiranje.*

Postoje dva glavna načina testiranja aplikacija: "s kraja na kraj" **End-to-End** i jedinično (**unit**) testiranje.

Ako se primeni pristup testiranju "odozgo na dole" ( **top-down**), pišu se testovi koji aplikaciju posmatraju kao crnu kutiju ( **black box** ). Sa aplikacijom se vrši interakcija kao što bi to radio korisnik i procenjuje se

"s polja" da li aplikacija radi na pravi način. Ovakav pristup testiranju aplikacije naziva se "s kraja na kraj" ( **End-to-End** ) testiranje.

Drugim pristupom izdvoji se jedan deo aplikacije i testira u izolaciji. Ovakva forma testiranja je poznata kao jedinično (**Unit**) testiranje. U jediničnom testiranju se pišu testovi koji obezbeđuju koji obezbeđuju ulaz datom aspektu posmatrane jedinice i procenjuje se izlaz sa ciljem potvrđivanja stepena njegovog podudaranja sa očekivanim rezultatima.

U narednom izlaganju će akcenat biti na kreiranju i izvoženju jediničnih testova u **Angular** okviru.

## JASMINE I KARMA TEST ALATI

*Sa ciljem testiranja kreiranih Angular aplikacija, biće korišćena dva alata: Jasmin i Karma.*

Sa ciljem testiranja kreiranih Angular aplikacija, biće korišćena dva alata:

- **Jasmine**
- **Karma**.

**Jasmin** (više preko linka: <https://jasmine.github.io/2.4/introduction.html>) je razvojni okvir baziran na ponašanju (behavior-driven) za testiranje JavaScript koda. Primenom okvira **Jasmin**, moguće je podesiti očekivanja (izlaz) koja bi kod trebalo da zadovolji kada se pokrene.

Na primer, posmatra se funkcija za sabiranje `sum()` objekta `Calculator`. Cilj testiranja je provera korektnosti funkcije za trivijalan slučaj:  $1 + 1 = 2$ . Na sledeći način je moguće izraziti odgovarajući test slučaj (u `Angular` - u se označava sa `_spec`):

```
describe('Calculator', () => {
 it('sums 1 and 1 to 2', () => {
 var calc = new Calculator();
 expect(calc.sum(1, 1)).toEqual(2);
 });
});
```

Jako dobra stvar u vezi Jasmine okvira jeste visok stepen čitljivosti testova. Ovde se jasno vidi da se očekuje (`expect`) da operacija `calc.sum()` da vrednost jednaku 2.

Na veoma jednostavan način, testovi se organizuju unutar `describe` i `it` blokova.

U normalnim situacijama, blok `describe` se koristi za svaku logičku jedinicu, koja je predmet testiranja, i unutar koje se koristi jedan `it` blok za svako očekivanje koje je neophodno prepostaviti. Međutim, često je moguće sresti i `it` blok koji sadrži nekoliko očekivanja.

U prikazanom primeru za `Calculator`, koristi se veoma jednostavan objekat. Iz tog razloga, koristi se jedan `describe` blok za celu klasu i po jedan `it` blok za svaku od njenih metoda.

Međutim ovo nije uvek slučaj. Na primer, metode koje produkuju različite izlaze u zavisnosti od ulaza, verovatno će posedovati više od jednog pridruženih `it` blokova. U ovakvim slučajevima, odlično se primenjuju ugnježdeni `describe` blokovi: jedan za objekat, a po jedan za svaku od metoda, a potom i odgovarajući `it` blokovi sa različitim prepostavkama unutar njih.

Primenom okvira `Jasmine` lako se iskazuju testovi i njihova očekivanja. Veoma brezo se dolazi do scenarija u kojem je neophodno pokrenuti kreirani test, a ovde je od posebnog značaja izvršavanje testova u okruženju obezbeđenim adekvatnim veb pregledačem. Tu na scenu stupa `Karma`. Alat `Karma` omogućava pokretanje JavaScript koda u pregledačima kao što su `Chrome` ili `Firefox`.

## PISANJE JEDINIČNIH TESTOVA

*Glavni fokus lekcije je razumevanje pisanja jediničnih testova nad različitim delovima aplikacije.*

Glavni fokus lekcije je razumevanje pisanja jediničnih testova nad različitim delovima Angular aplikacije.

Akcenat će biti na učenju načina kako se testiraju:

- *Servisi*;
- *Komponente*;
- *HTTP zahtevi*, i tako dalje.

Uz put će biti govora o tehnikama koje je moguće primeniti kako bi kreirani kod bio pogodniji za obavljanje testova.

## ANGULAR OKVIR ZA JEDINIČNO TESTIRANJE

*Angular obezbeđuje vlastiti set klasa izgrađenih nad okvirom Jasmine.*

*Angular* obezbeđuje vlastiti set klasa izgrađenih nad okvirom *Jasmine* koje pomažu prilikom kreiranja jediničnih test slučajeva primenom okvira.

Glavni radni okvir za *Angular* jedinično testiranje se nalazi u paketu: [@angular/core/testing package](#). Međutim, za testiranje komponenata moguće je iskoristiti pakete [@angular/compiler/testing](#) ili [@angular/platformbrowser/testing](#) za dodatne pomoći i olakšanja.

Ukoliko se student prvi put susreće sa testiranjem u *Angular* - u neophodno ga je uputiti u jednu veoma bitnu stvar:**pisanju test slučajeva u Angular - u prethodi malo podešavanja**. Na primer, ukoliko je neophodno obaviti umetanje zavisnosti, obično se konfigurisanje obavi manuelno. Kada se pristupa testiranju komponente, neophodno je koristiti pomoćne funkcije (testing - helpers: <https://www.npmjs.com/package/angular-unit-testing-helpers>) za njihovo inicijalizovanje. Kada se pristupa testiranju rutiranja, postoji veliki broj zavisnosti koje je potrebno strukturirati. Na prvi pogled deluje kao mnogo podešavanja, međutim nema razloga za brigu - veoma brzo će biti jasno da se podešavanje ne menja puno od projekta do projekta. Pored toga, diskusija će biti široka, a po već prihvaćenoj praksi detaljno pokrivena pokaznim primerima-

Uvod u Angular okvir za jedinično testiranje moguće je ilustrovati sledećim kratkim video materijalom:

**Ova lekcija sadrži video materijal. Ukoliko želite da pogledate ovaj video morate da otvorite LAMS lekciju.**

## ▼ Poglavlje 2

### Podešavanje testiranja

## VIDEO PREDAVANJE ZA OBJEKAT "PODEŠAVANJE TESTIRANJA"

*Trajanje video snimka: 14min 30sek*

**Ova lekcija sadrži video materijal. Ukoliko želite da pogledate ovaj video morate da otvorite LAMS lekciju.**

## PODEŠAVANJE ALATA KARMA

*Alat Karma zahteva da se obavi podešavanje da bi mogao da radi*

Odličan primer za pisanje testova može biti primer koji implementira rutiranje. U tom svetlu, kao prateći primer ovoj lekciji, dodat je primer za pretragu muzike. Cilj je pisanje testova za ovu aplikaciju.

Alat Karma zahteva da se obavi podešavanje da bi mogao da radi. Prvi korak u podešavanju alata Karma jeste kreiranje konfiguracione datoteke pod nazivom karma.conf.js. U glavnom folderu projekta (koren putanje - slika 1) je navedeni konfiguracioni fajl čiji je sadržaj priložen u sledećem listingu:

```
module.exports = function(config) {
 let configuration = {
 basePath: '',
 frameworks: ['jasmine', '@angular-devkit/build-angular'],
 plugins: [
 require('karma-jasmine'),
 require('karma-chrome-launcher'),
 require('karma-jasmine-html-reporter'),
 require('karma-coverage-istanbul-reporter'),
 require('@angular-devkit/build-angular/plugins/karma')
],
 client: {
 clearContext: false // ostavlja Jasmine Spec Runner output vidljiv u
 pregledaču
 },
 coverageIstanbulReporter: {
 dir: require('path').join(__dirname, '../coverage'),
 reports: ['html', 'lcovonly'],
 fixWebpackSourcePaths: true
 }
}
```

```
},
reporters: ['progress', 'kjhtml'],
port: 9876,
colors: true,
logLevel: config.LOG_INFO,
autoWatch: true,
browsers: ['Chrome'],
singleRun: false
};

if (process.env.TRAVIS) {
 configuration.customLaunchers = {
 Chrome_travis_ci: {
 base: 'Chrome',
 flags: ['--no-sandbox']
 }
 };
 configuration.browsers = ['Chrome_travis_ci'];
}

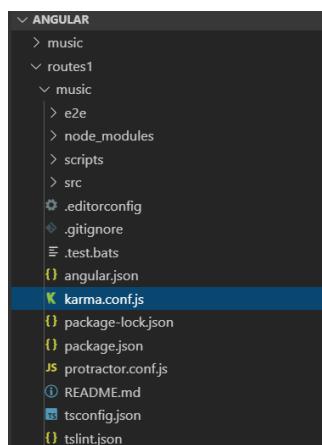
config.set(configuration);
};
```

Sadržaj priložene datoteke može biti trenutno zbumujući, međutim, u ovom trenutku bi trebalo imati na umu samo par stvari:

- podešen je [Google Chrome](#) kao ciljni test pregledač;
- koristi se [Jasmine](#) / [Karma](#) okvir za testiranje;
- koristi se [WebPack](#) modul, pod nazivom *test.bundle.js*, koji u osnovi omotava celokupan test i aplikativni kod.

U sledećem koraku je veoma bitno kreiranje novog *test* foldera u kojem će biti čuvani kreirani test fajlovi:

```
mkdir test
```



Slika 2.1 Lokacija konfiguracione datoteke karma.conf.js u Angular projektu [izvor: autor]

## VIDEO MATERIJAL

*JavaScript Ep. 2: Quick Karma - trajanje: 7:03*

**Ova lekcija sadrži video materijal. Ukoliko želite da pogledate ovaj video morate da otvorite LAMS lekciju.**

## ▼ Poglavlje 3

# Testiranje servisa i HTTP zahteva

## SPOTIFYSERVICE KLASA

*Servisi u okviru Angular započinju vlastitu egzistenciju kao obične klase - olakšano testiranje.*

Servisi u okviru Angular započinju vlastitu egzistenciju kao obične (plain) klase. U neku ruku, ovo čini servise lakšim za primenu jer je ponekad moguće obaviti njihovo testiranje direktno, bez primene okvira Angular. Sledi listing servisne klase SpotifyService za koju će u nastavku biti pisani odgovarajući testovi (datoteka: /src/app/spotify.service.ts).

```
import { Injectable } from '@angular/core';
import {
 Http,
 Headers,
 RequestOptions
} from '@angular/http';
import { Observable } from 'rxjs/Observable';
import 'rxjs/Rx';
import { environment } from '../environments/environment';

/**
 * SpotifyService šalje upit na Spotify Web API
 * https://developer.spotify.com/web-api/
 */

@Injectable()
export class SpotifyService {
 static BASE_URL = 'https://api.spotify.com/v1';

 constructor(private http: Http) {}

 query(
 URL: string,
 params?: Array<string>
): Observable<any[]> {
 let queryURL = `${SpotifyService.BASE_URL}${URL}`;
 if (params) {
 queryURL = `${queryURL}?${params.join('&')}`;
 }
 const apiKey = environment.spotifyApiKey;
 const headers = new Headers({
 Authorization: `Bearer ${apiKey}`
 });
 return this.http.get(queryURL, { headers });
 }
}
```

```
});

const options = new RequestOptions({
 headers: headers
});

return this.http
 .request(queryURL, options)
 .map((res: any) => res.json());
}

search(query: string, type: string): Observable<any[]> {
 return this.query(`/search`, [
 `q=${query}`,
 `type=${type}`
]);
}

searchTrack(query: string): Observable<any[]> {
 return this.search(query, 'track');
}

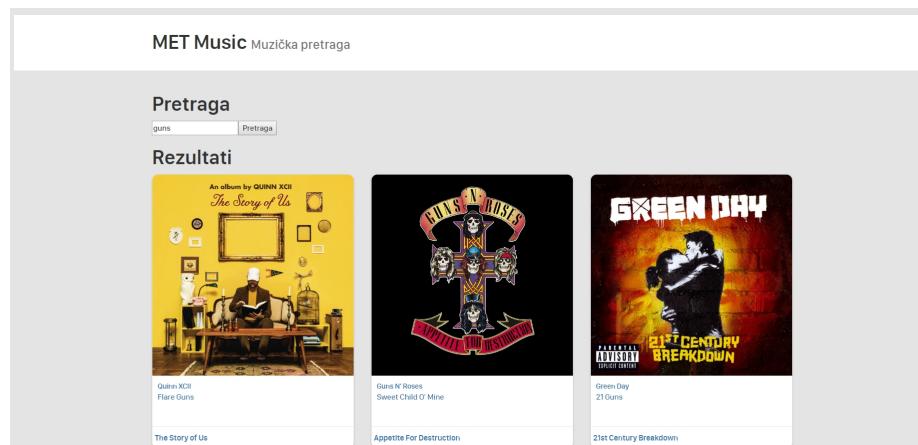
getTrack(id: string): Observable<any[]> {
 return this.query(`/tracks/${id}`);
}

getArtist(id: string): Observable<any[]> {
 return this.query(`/artists/${id}`);
}

getAlbum(id: string): Observable<any[]> {
 return this.query(`/albums/${id}`);
}

export const SPOTIFY_PROVIDERS: Array<any> = [
 { provide: SpotifyService, useClass: SpotifyService }
];
```

Funkcionisanje ovog servisa se ogleda kroz interakciju sa *SpotifyAPI* - jem za dobijanje informacija u vezi sa albumom, pesmom i izvođačem. Navedeno je prikazano sledećom slikom:



Slika 3.1 Pretaga nakon interakcije servisa i SpotifyAPI [izvor: autor]

## PRIPREMA ZA TESTIRANJE SERVISA I IMPORT INSTRUKCIJE

*Prvo što je potrebno uraditi jeste importovanje test pomoćnih funkcija i klase i klase koja se testira.*

Pošto je alat [Karma](#) podešen, moguće je pristupiti testiranju kreirane servisne klase [SpotifyService](#). U prethodnom izlaganju je opisano kako ovaj servis funkcioniše (kroz interakciju sa [SpotifyAPI](#)) i koje informacije pribavlja aplikaciji.

Postoje dva pristupa kreiranju test datoteka:

- kreira se podfolder foldera test (npr. [services](#)) i u njemu se čuvaju svi test fajlovi;
- test fajlovi se čuvaju u istom folderu kao i klase koje testiraju.

Ovde će biti prihvaćen drugi pristup i pored datoteke servisne klase biće kreirana prva test datoteka [spotify.service.spec.ts](#).

Sada je moguće kreirati sadržaj ove datoteke. Prvo što je potrebno uraditi jeste importovanje test pomoćnih funkcija ([test helpers](#)) iz paketa:

[@angular/core/testing](#). Navedeno je ilustrovano sledećim listingom (datoteka primera: [/src/app/spotify.service.spec.ts](#)):

```
import {
 inject,
 fakeAsync,
 tick,
 TestBed
} from '@angular/core/testing';
```

U nastavku je neophodno obaviti dodavanje još nekoliko korisnih klasa u test datoteku:

```
import {MockBackend} from '@angular/http/testing';
import {
 Http,
}
```

```
ConnectionBackend,
BaseRequestOptions,
Response,
ResponseOptions
} from '@angular/http';
```

Budući da servis koristi [HTTP](#) zahteve, neophodno je importovati klasu [MockBackend](#) iz paketa [@angular/http/testing](#). Ova klasa će pomoći prilikom podešavanja očekivanja i provere HTTP zahteva.

Poslednja klasa koju je neophodno učitati u test predstavlja klasa koja se testira, a to je servisna klase [SpotifyService](#).

```
import { SpotifyService } from './spotify.service';
```

## HTTP RAZMATRANJA

*HTTP zahtevi su relativno spori i vremenom je potrebno sve više vremena za izvršavanje testova.*

Već je sada moguće započeti pisanje testova ali jedno bi trebalo imati na umu. Prilikom izvršavanja svakog od testova biće pozivan i angažovan [Spotify](#) server. Ovo je daleko od idealnog iz više razloga:

1. [HTTP zahtevi](#) su relativno spori i, dok skup testova bude rastao, veoma brzo će biti jasno kako je potrebno sve više i više vremena za izvršavanje svih potrebnih testova;
2. [Spotify API](#) ima kvotu poziva i, ako ceo tim radi testove, vrlo brzo može doći do bespotrebne upotrebe resursa za API pozive.
3. Ukoliko tester nije na mreži ili je [Spotify](#) server oboren ili nedostupan, testovi će početi da "pucaju", čak iako je njegov, kao i kod kojeg testiraju, tehnički ispravan.

Ovde se nameće odlična ideja za kreiranje jediničnih testova - izolovati [sve što nije moguće kontrolisati pre testiranja](#). U konkretnom slučaju jedinicom se smatra klasa servisa i rešenje bi bilo zamena [HTTP](#) zahteva nečim što ima identično ponašanje ali ne dotiče realni [Spotify](#) server.

Obavljanje ovakvog test zadatka u svetu testiranja se naziva simuliranje ([mocking](#)) zavisnosti, a nekad i ublažavanje ([stubbing](#)) zavisnosti.

O [stub](#) i [mock](#) konceptima biće puno govora u sledećem predmetu [IT355 - Veb sistemi 2](#). Ovde će biti ukratko predstavljeni sa ciljem demonstriranja njihovog značaja za testiranje frontend veb aplikacija. Iako ih ljudi često mešaju, ovo nisu isti koncepti (više u članku "Mocks aren't Stubs : <https://martinfowler.com/articles/mocksarentstubs.html>").

Da bi sve bilo mnogi jasnije, za trenutak će biti ostavljen po strani aktuelni primer, a uveden pomoćni - jednostavniji. Prepostavka je da se kreira kod koji zavisi od klase automobila, pod nazivom [Car](#). Ova klasa može da ima veliki broj metoda, poput: [start\(\)](#), [stop\(\)](#), [park\(\)](#), [speedUp\(\)](#) i tako dalje. Upravo je ova klasa odlična polazna tačka za demonstraciju primene [stub](#) i [mock](#) koncepcata za pisanje testova zavisnih od prikazane klase.

# STUB I MOCK KONCEPTI TESTIRANJA

*Iako ih ljudi često mešaju, Stub i Mock nisu isti koncepti*

## Stub objekti:

*Stub objekti* su objekti koji se kreiraju u hodu, sa podskupom ponašanja koje posmatrana zavisnost poseduje.

Za demonstraciju, piše se test koji interaguje sa metodom start() klase Car. Moguće je kreirati stub objekat klase *Car* i obaviti njegovo umetanje u klasu koja se testira:

```
describe('Speedtrap', function() {
 it('kazna za brzinu preko 60km/h', function() {
 var stubCar = { getSpeed: function() { return 61; } };
 var speedTrap = new SpeedTrap(stubCar);
 speedTrap.ticketCount = 0;
 speedTrap.checkSpeed();
 expect(speedTrap.ticketCount).toEqual(1);
 });
})
```

Ovo bi bio klasičan slučaj primene *stub* objekta koji bi bio, verovatno, upotrebljen lokalno u testu

## Mock objekti:

*Mock* objekat, u ovom slučaju, predstavlja kompletniju reprezentaciju objekata koja redefiniše delove ili celokupno ponašanje zavisnosti. *Mock* objekat može, a u najvećem broju slučajeva je tako, biti upotrebljen više puta u nekom skupu testova.

*Mock* objekti mogu biti upotrebljeni za proveru da li su date metode pozvane na način na koji bi trebalo da budu pozvane.

Jedan primer simulirane (*mock*) verzije klase *Car* može da bude:

```
class MockCar {
 startCallCount: number = 0;

 start() {
 this.startCallCount++;
 }
}
```

Kreirani *MockCar* objekat može biti upotrebljen u testu na sledeći način:

```
describe('CarRemote', function() {
 it('pokreće se okretanjem ključa', function() {
 var car = new MockCar();
 var remote = new CarRemote();
 remote.holdButton('start');
 expect(car.startCallCount).toEqual(1);
 });
})
```

```
});
});
```

## STUB I MOCK DODATNA POJAŠNJENJA

*Neophodno objasniti suštinsku razliku između dva navedena test koncepta - Stub i Mock.*

U nastavku je neophodno objasniti suštinsku razliku između dva navedena test koncepta - *Stub i Mock*:

- *stub* obezbeđuje podskup funkcionalnosti sa "manuelno" redefinisanim ponašanjem;
- *mock*, generalno, postavlja očekivanja i potvrđuje da su određene metode bile pozvane.

Tests pisani primenom *mock* objekata, obično slede sledeći šablon testiranja:

```
inicijalizacija -> podešavanje očekivanja -> izvođenje -> provera
```

Dok, unapred napisani stub objekti, obično forsiraju sledeći šablon testiranja:

```
inicijalizacija -> izvođenje -> provera
```

## HTTP MOCKBACKEND

*Angular omogućava simuliranje HTTP poziva primenom klase MockBackend*

Pošto je napravljen kraći uvod u primenu Mock i Stub objekata za pisanje testova provere funkcionalnosti veb aplikacije, moguće je vratiti se na konkretan primer i započeti pisanje koda testa servisne klase.

Interakcija sa realnim *Spotify* serverom i njegovim "živim" servisima je loša ideja, već je opisano iz kojeg razloga, ali **Angular omogućava simuliranje HTTP poziva primenom klase MockBackend**.

Navedena klasa može da bude umetnuta u *Http* instancu i daje kontrolu nad ponašanjem *HTTP* interakcije. Podešavanja i prepostavke mogu biti definisani na brojne načine:

- odgovor (*response*) može biti manuelno podešen;
- moguće je simulirati *HTTP* grešku (*HTTP error*);
- moguće je definisati očekivane vrednosti, kao što je potvrda da se traženi *URL* podudara s onim što je traženo, ako su navedeni parametri zahteva tačni i još mnogo toga.

Ideja je da se u kod doda simulirana ("lažna") *Http* biblioteka. Simulirana biblioteka će se u kodu ponašati kao realna *Http* biblioteka: sve metode će se podudarati, vraćaće odgovore i tako dalje. Međutim, ne dolazi do kreiranja realnih zahteva.

Zapravo, umesto kreiranja realnih zahteva, klasa `MockBackend` će dozvoliti da testeri podeše očekivanja i posmatraju ponašanje koje se očekuje.

## ANGULAR TEST BED I PROVAJDERI

*Angular Test Bed (ATB) je okvir visokog nivoa za testiranje ponašanja koja zavise od Angular-a.*

Angular TestBed (ATB) je okvir visokog nivoa za testiranje, primenljiv samo u kombinaciji sa Angular okvirom, koji dozvoljava testiranje ponašanja koja zavise od Angular okvira na veoma lak način.

Kada se testira Angular aplikacija neophodno je proveriti da li je konfigurisan NgModule najvišeg nivoa koji će biti korišćen za ovaj test. Kada se ovo radi, moguće je: podešiti provajdere, deklarisati komponente i importovati ostale module, baš kao kada se u aplikaciji koristi NgModule, generalno u aplikaciji.

Ponekad, kada se vrši testiranje Angular koda, neophodno je ručno podešiti umetanja. Ovo je dobro jer daje veću kontrolu nad onim što se zapravo testira.

U slučaju testiranja HTTP zahteva, ne vrši se umetanje realne Http klase, već nečega što liči na nju ali stvarno presreće zahteve i vraća odgovore koji će biti konfigurisani.

Da bi navedeno bilo moguće, kreira se verzija Http klase koja interno koristi MockBackend. U tu svrhu se koristi poziv TestBed.configureTestingModuleTestingModule unutar hvatača beforeEach(). Ovaj hvatač uzima funkciju koja će biti pozivana pre pokretanja svakog testa, dajući na taj način odličnu mogućnost podešavanja implementacije alternativnih klasa.

## ANGULAR TEST BED I PROVAJDERI I KREIRANJE TESTOVA

*TestBed.configureTestingModuleTestingModule() prihvata niz pod ključem providers koji će koristiti test umetač.*

Sada je ponovo pažnja na datoteci za testiranje posmatranog veb servisa: /src/app/spotify.service.spec.ts unutar koje se dodaje sledeći kod:

```
describe('SpotifyService', () => {
 beforeEach(() => {
 TestBed.configureTestingModule({
 providers: [
 BaseRequestOptions,
 MockBackend,
 SpotifyService,
 { provide: Http,
 useFactory: (backend: ConnectionBackend,
 defaultOptions: BaseRequestOptions) => {
```

```
 return new Http(backend, defaultOptions);
 }, deps: [MockBackend, BaseRequestOptions] ,
]
});
}
});
```

Primeću se da poziv metode  `TestBed.configureTestingModuleTestingModule()` prihvata niz pod ključem `providers` koji će koristiti test umetač (injector).

`BaseRequestOptions` i `SpotifyService` su podrazumevane implementacije navedenih klasa . Međutim, poslednji provajder u nizu je komplikovaniji:

```
{ provide: Http,
 useFactory: (backend: ConnectionBackend,
 defaultOptions: BaseRequestOptions) => {
 return new Http(backend, defaultOptions);
 }, deps: [MockBackend, BaseRequestOptions] ,
}
```

Kod koristi ključ `provide` sa `useFactory` za kreiranje verzije `Http` klase, primenom fabrikacije (produkcione metode).

Produkciona metoda poseduje potpis kojim očekuje `ConnectionBackend` i `BaseRequestOption` instance.

Drugi ključ, prikazanog objekta je, `deps: [MockBackend, BaseRequestOptions]`. Ovo ukazuje da će `MockBackend` biti upotrebljen kao prvi parametar fabrikacije, a `BaseRequestOptions` (podrazumevana implementacija) kao drugi.

Konačno, vraća se prilagođena (custom) `Http` klasa sa `MockBackend` kao rezultat navedene funkcije. Koje su koristi od svega navedenog? Svaki put (u okviru testa) kada kod zahteva umetanje `Http` instance, dobiće prilagođenu (custom) instancu.

Ovo je odlična ideja sa velikom primenom u testiranju: koristi umetanje zavisnosti za prilagođavanje zavisnosti i izoluje funkcionalnosti koje se testiraju.

## TESTIRANJE METODE GETTRACK()

*Testiranje poziva ispravnog URL u test slučaju servisne klase.*

Nakon detaljne analize i diskusije, konačno je započeto pisanje testova za konkretni servis. Neka je, u nastavku, cilj testiranje poziva ispravnog URL - a.

Osnov za kreiranje testova će biti metoda `getTrack()` posmatrane servisne klase. Sledećim listingom je dat kod ove metode:

```
getTrack(id: string): Observable<any[]> {
 return this.query(`/tracks/${id}`);
}
```

Napomena: Ova metoda koristi metodu query() koja kreira URL na osnovu primljenih parametara:

```
query(
 URL: string,
 params?: Array<string>
) : Observable<any[]> {
 let queryURL = `${SpotifyService.BASE_URL}${URL}`;
 if (params) {
 queryURL = `${queryURL}?${params.join('&')}`;
 }
 const apiKey = environment.spotifyApiKey;
 const headers = new Headers({
 Authorization: `Bearer ${apiKey}`
 });
 const options = new RequestOptions({
 headers: headers
 });

 return this.http
 .request(queryURL, options)
 .map((res: any) => res.json());
}
```

Budući da se prosleđuje `/tracks/${id}`, prepostavlja se da će prilikom poziva metode `getTrack('TRACK_ID')` očekivani `URL` biti: `https://api.spotify.com/v1/tracks/TRACK_ID`.

Za navedeno se kreira sledeći test:

```
describe('getTrack', () => {
 it('retrieves using the track ID',
 inject([SpotifyService, MockBackend], fakeAsync((svc, backend) => {
 let res;
 expectURL(backend, 'https://api.spotify.com/v1/tracks/TRACK_ID');
 svc.getTrack('TRACK_ID').subscribe(_res => {
 res = _res;
 });
 tick();
 expect(res.name).toBe('vlada');
 }))
);
});
```

U test primeru postoji mnogi novih detalja, pa će diskusija biti razbijena na sitnije detalje izložene u narednim sekcijama.

## TESTIRANJE METODE GETTRACK() - UMETANJE ZAVISNOSTI

*Kada se piše test sa zavisnostima, neophodno je da Angular obezbedi instance ovih klasa.*

Svaki put kada se piše test sa zavisnostima, neophodno je zatražiti od *Angular* okvira da obezbedi instance potrebnih klasa. Da bi navedeno bilo učinjeno, koristi se sledeće:

```
inject([Class1, /* ..., */ ClassN],
 (instance1, /* ..., */ instanceN) => {
 // ... testirajući kod ...
})
```

Kada testirajući kod vrati obećanje (*Promise*) ili *RxJS Observable* tip, moguće je upotrebiti pomoćnu metodu *fakeAsync()* za testiranje asinhronog koda na sinhron način (<https://alligator.io/angular/testing-async-fakeasync/>). Na ovaj način obećanja (*Promises* objekti) su zadovoljena, a *Observable* objekti su dobili notifikaciju odmah nakon poziva metode *tick()*.

Dakle, u sledećem kodu:

```
inject([SpotifyService, MockBackend],
 fakeAsync((svc, backend) =>
 //testirajući kod
));
```

uzimaju se dve promenljive: *spotifyService* i *mockBackend*. Prva predstavlja konkretnu instancu servisa *SpotifyService*, a druga je instanca klase *MockBackend*. Takođe je moguće primetiti da su argumenti unutrašnje funkcije (*spotifyService*, *mockBackend*) umetanja klasa specificiranih nizom iz prvog argumenta funkcije *inject()*.

Ovde se takođe izvršava i metoda *fakeAsync()* što znači da će asinhroni kod biti izvršavan kao sinhron kada se pozove metoda *tick()*.

## TESTIRANJE METODE GETTRACK() I PISANJE TEST KODA

*Nakon podešavanja umetanja zavisnosti i konteksta testiranja, moguće je započeti pisanje test koda.*

Nakon podešavanja umetanja zavisnosti i konteksta testiranja, moguće je započeti pisanje test koda. Započinje se deklarisanjem *res* promenljive koja će na kraju dobiti odgovor *HTTP* poziva. Nakon toga se vrši prijavljivanje na *mockBackend.connections*:

```
var res;
mockBackend.connections.subscribe(c => { ... });
```

Ovde je rečeno da svaki put kada se ostvari nova konekcija sa `mockBackend` neophodno je dobiti notifikaciju (na primer, poziv ove funkcije).

Takođe, cilj je da se proveri da li servis `SpotifyService` poziva ispravan `URL` na osnovu `id` pesme `TRACK_ID`. Moguće je dobiti URL iz konekcije `c` preko poziva `c.request.url`. Zbog navedenog se podešava očekivanje da bi `c.request.url` trebalo da bude string '`https://api.spotify.com/v1/tracks/TRACK_ID`':

```
expect(c.request.url).toBe('https://api.spotify.com/v1/tracks/TRACK_ID');
```

Sada kada bi ovakav test bio pokrenut, ukoliko traženi URL ne bi bio odgovarajući, test bi pao (`fail`). Nakon prijema zahteva i provere njegove ispravnosti, neophodno je pripremiti odgovor koji će test vratiti. Ovo se obavlja kreiranjem nove instance klase `ResponseOptions`. Ovde će biti specificirano da će `JSON` string `{"name": "vlada"}` biti vraćen kao telo odgovora.

```
let response = new ResponseOptions({body: '{"name": "vlada"}'});
```

Konačno, ukazuje se objektu `connection` da zameni odgovor `Response` objektom koji omotava kreiranu instancu `ResponseOptions`:

```
c.mockRespond(new Response(response));
```

Sada je sve podešeno za pozivanje metode `getTrack()` sa parametrom `TRACK_ID` i praćenje odgovora unutar promenljive `res`:

```
spotifyService.getTrack('TRACK_ID').subscribe(_res) => {
 res = _res;
};
```

Ukoliko bi test ovde bio završen, moralo bi da se sačeka da se obavi `HTTP` poziv i kreira odgovor pre poziva unutrašnje (`callback`) funkcije. To bi se, takođe, desilo na drugačijoj putanji izvršenja te bi bilo potrebno dodatno upravljati kodom za sinhronizaciju navedenih stavki. Srećom, metoda `fakeAsync()` uklanja navedeni problem. Sve što je potrebno jeste pozivanje metode `tick()` i, poput magije, asinhroni kod će biti izvršen:

```
tick();
```

Sada je moguće izvršiti konačnu proveru, a to je da je podešeni odgovor, upravo onaj koji se prima:

```
expect(res.name).toBe('vlada');
```

## PISANJE TESTOVA ZA OSTALE METODE SERVISA SPOTIFYSERVICE

*Zajednička očekivanja za metode mogu biti izolovana u posebnu funkciju.*

Budući da je kod svih metoda posmatranog servisa veoma sličan, moguće je uvesti dodatno olakšanje u kreiranje koda testa. Sva zajednička očekivanja za metode (u konkretnom slučaju *URL*) mogu biti izolovana u posebnu funkciju, na primer sa nazivom *expectURL()* (datoteka projekta: */src/app/spotify.service.spec.ts*):

```
// podešava očekivanja da će ispravan URL biti zahtevan
function expectURL(backend: MockBackend, url: string) {
 backend.connections.subscribe(c => {
 expect(c.request.url).toBe(url);
 const response = new ResponseOptions({body: '{"name": "vlada"}'});
 c.mockRespond(new Response(response));
 });
}
```

Prateći instrukcije iz prethodne sekcije i koristeći kreiranu pomoćnu metodu za proveru ispravnosti URL - a, lako se kreira test za proveru metode *getArtist()* i *getAlbum()*. Sledi listing testa za metodu *getArtist()*:

```
describe('getArtist', () => {
 it('retrieves using the artist ID',
 inject([SpotifyService, MockBackend], fakeAsync((svc, backend) => {
 let res;
 expectURL(backend, 'https://api.spotify.com/v1/artists/ARTIST_ID');
 svc.getArtist('ARTIST_ID').subscribe(_res => {
 res = _res;
 });
 tick();
 expect(res.name).toBe('vlada');
 }))
);
});
```

Sledi listing testa za metodu *getAlbum()*:

```
describe('getAlbum', () => {
 it('retrieves using the album ID',
 inject([SpotifyService, MockBackend], fakeAsync((svc, backend) => {
 let res;
 expectURL(backend, 'https://api.spotify.com/v1/albums/ALBUM_ID');
 svc.getAlbum('ALBUM_ID').subscribe(_res => {
 res = _res;
 });
 tick();
 expect(res.name).toBe('vlada');
 }))
);
});
```

## TESTOVI ZA OSTALE METODE SERVISA SPOTIFYSERVICE I SEARCHTRACK()

*Umesto poziva metode query(), searchTrack(), unutar svog tela, koristi metodu search():*

Metoda `searchTrack()` se malo razlikuje od ostalih metoda servisa. Umesto poziva metode `query()`, metoda `searchTrack()`, unutra svog tela, koristi metodu `search()`:

```
searchTrack(query: string): Observable<any[]> {
 return this.search(query, 'track');
}
```

A potom metoda `search()` poziva metodu `query()` sa `/search`, kao prvim argumentom i nizom koji sadrži elemente `q=<query>` i `type=track`, kao drugim argumentom:

```
search(query: string, type: string): Observable<any[]> {
 return this.query(`/search`, [
 `q=${query}`,
 `type=${type}`
]);
}
```

Konačno, `query()` će transformisati parametre u URL putanju sa `QueryString` (više o modulu na linku: <https://nodejs.org/api/querystring.html>). Pa će sada, URL očekivani URL se završavati sa:

```
/search?q=<query>&type=track.
```

Sada je moguće napisati test za ovu metodu koji će uzeti u razmatranje sve što je upravo navedeno (datoteka projekta: `/src/app/spotify.service.spec.ts`).

```
describe('searchTrack', () => {
 it('searches type and term',
 inject([SpotifyService, MockBackend], fakeAsync((svc, backend) => {
 let res;
 expectURL(backend, 'https://api.spotify.com/v1/search?q=TERM&type=track');
 svc.searchTrack('TERM').subscribe(_res => {
 res = _res;
 });
 tick();
 expect(res.name).toBe('vlada');
 }));
);
});
```

Poslednji test slučaj je sličan prethodnim i sledećim tezama je izzaknuto šta on radi:

- povezuje se na HTTP životni ciklus, dodavanjem unutrašnje metode (callback) kada je nova HTTP konekcija inicijalizovana;

- podešava očekivanja za URL koja traže da `connection` objekat uključi `q=TERM` i `type=track`;
- poziva metodu koja se testira, `searchTrack()`;
- poziva `Angular` da kompletira sve započete asinhronne pozive;
- na kraju, daje pretpostavku (tvrđnju) da postoji očekivani odgovor.

## POKRETANJE TESTOVA U KARMA ALATU

*Simuliranje prolaska i padova kreiranih testova.*

U nastavku, primenom podrazumevanog pregledača `Google Chrome`, učitava se test alat `Karma` sa ciljem provere kreiranih testova. U terminalu razvojnog okruženja kuca se naredba:

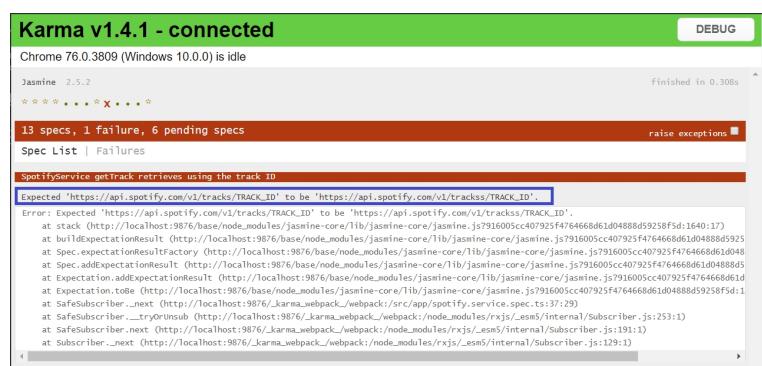
```
ng test --watch
```

Ubrzo se u okviru veb pregledača učitava test alat `Karma` koji prikazuje da su svi testovi uspešno proverili metode za koje su pisani, a to je prikazano sledećom slikom.



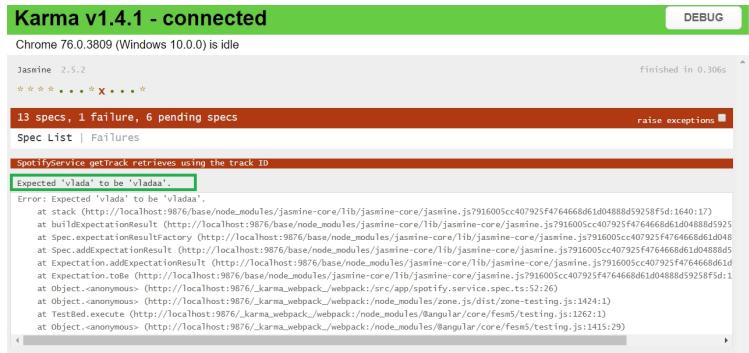
Slika 3.2 Karma prikazuje da su svi testovi uspešno proverili metode [izvor: autor]

Naravno, test može da se modifikuje da padne. U prvom slučaju (slika 3) namešten je nekorektni URL i test je pao.



Slika 3.3 Pad testa usled nekorektnog URL [izvor: autor]

U drugo slučaju (slika 3) namešten je nekorektna vrednost za `res.name` i test je pao.



Slika 3.4 Pad testa usled nekorektnog res.name [izvor: autor]

## TESTIRANJE SERVISA KROZ ZAKLJUČNA RAZMATRANJA

*Kada se testiraju servisi, postoje jasni ciljevi koji moraju da budu ispunjeni.*

U osnovi, kada se testiraju servisi ciljevi bi trebalo da budu sledeći:

- izolovanje svih zavisnosti primenom stub ili mock koncepata;
- u slučaju asinhronih poziva, koriste se metode `fakeAsync()` i `tick()` za obezbeđivanje da oni budu ispunjeni;
- pozivanje servisne metode koja se testira;
- provera da povratna vrednost metode odgovara očekivanoj vrednosti.

Sada je sve spremno za korak dalje - testiranje klase koje konzumiraju servise, a to su komponente.

Video materijal: **Angular 5.x - Unit Testing and HttpClient** - trajanje 10:53.

**Ova lekcija sadrži video materijal. Ukoliko želite da pogledate ovaj video morate da otvorite LAMS lekciju.**

## ▼ Poglavlje 4

# Testiranje rutiranja ka komponentama

## UVOD U TESTIRANJE KOMPONENTA

*Testiranje komponenta je moguće obaviti primenom black box i white box tehnika testiranja,*

Kada se testiraju komponente, moguće je uraditi sledeće:

1. kreirati testove koji će interagovati sa komponentom iz okruženja, prosleđujući atributе i proveravajući kako će to uticati na [HTML](#) (šablone);
2. kreirati testove za proveru individualnih metoda komponente i njihovih izlaza.

Navedene strategije testiranja su poznate kao tehnike crne ([black box](#)) i bele ([white box](#)) kutije. U nastavku izlaganja, akcenat će biti na kombinovanju navedenih tehnika.

Vraća se pažnja na pokazni primer, koji je bio predmet izučavanja i u prethodnom objektu učenja. Uskoro će biti ispunjeni svi uslovi za početak kreiranja testova za klasu [ArtistComponent](#) koja, kao jedna od jednostavnijih klasa, predstavlja odličnu podlogu za savladavanje problematike vezane za testiranje [Angular](#) komponenta. Početni skup testova će biti usmeren ka provjeri unutrašnjih detalja navedene klase pa potпадa pod kategoriju "white box" tehnika testiranja.

Pre prelaska na sam proces testiranja i izrade testova, neophodno je sagledati šta posmatrana klasa [ArtistComponent](#) radi. Prva stvar koja mora da bude uočena jeste da konstruktor klase uzima [id](#) parametar iz kolekcije [routeParams](#) (datoteka priloženog primera: [/src/app/artist/artist.component.ts](#)):

```
constructor(private route: ActivatedRoute, private spotify: SpotifyService,
 private location: Location) {
 route.params.subscribe(params => { this.id = params['id']; });
}
```

Ovde se sada javlja jedan problem. Kako je moguće pronaći ID rute bez dostupnog pokretačkog rutera?

## KREIRANJE RUTERA ZA TESTIRANJE

*Rutiranje (i testiranje komponenti) ima veliki broj zavisnosti kojih je potrebno umetnuti*

Kada se pišu testovi u *Angular* okviru, neophodno je manuelno obaviti podešavanje brojnih klasa koje se umeću (injektuju). *Rutiranje* (i *testiranje komponenti*) ima veliki broj zavisnosti kojih je potrebno umetnuti. Ali je takođe i istaknuto da, nakon što je jednom konfigurisano, to nije nešto što se mnogo menja i veoma je jednostavno za korišćenje.

Kada se pišu testovi, često je pogodno koristiti hvatač *beforeEach* sa  *TestBed.configureTestingModuleTestingModule*  (bilo je govora u prethodnom objektu učenja) za podešavanje zavisnosti koje je neophodno ubrizgati (umetnuti). U konkretnom slučaju, testiranja komponente *ArtistComponent*, biće kreirana posebna funkcija čiji će zadatak biti kreiranje i konfigurisanje ruteru za testiranje. Navedeno se realizuje u test datoteci za klasu komponente (datoteka: */src/app/artist/artist.component.spec.ts*):

```
describe('ArtistComponent', () => {
 beforeEach(async(() => {
 configureMusicTests();
 }));
});
```

U nastavku, neophodno je definisati metodu *configureMusicTests()*, čiji je poziv upravo obavljen, u pomoćnoj datoteci *MusicTestHelpers.ts*. Pa hajde da se to i uradi. Sledi implementacija metode *configureMusicTests()* u okviru datoteke primera: */src/app/test/test.module.ts*. Sve će biti analizirano i objašnjeno "korak po korak".

```
export function configureMusicTests() {
 const mockSpotifyService: MockSpotifyService = new MockSpotifyService();

 TestBed.configureTestingModule({
 imports: [
 {
 ngModule: RouterTestingModule,
 providers: [provideRoutes(routerConfig)]
 },
 TestModule
],
 providers: [
 mockSpotifyService.getProviders(),
 {
 provide: ActivatedRoute,
 useFactory: (r: Router) => r.routerState.root, deps: [Router]
 }
]
 });
}
```

Započinje se kreiranjem instance servisa *MockSpotifyService* koja će koristiti simulaciju (*mock*) realnog servisa *SpotifyService*. U nastavku se koristi klasa *TestBed* koja poziva statičku metodu *configureTestingModule()*. *TestBed* je pomoćna biblioteka, koja se isporučuje sa okvirom *Angular* sa ciljem olakšavanja zadataka testiranja.

## KRAJ DEFINICIJE RUTERA ZA TESTIRANJE

*TestBed.configureTestingModule() je upotrebljen da se podesi NgModule neophodan za testiranje.*

Nastavlja se diskusija započeta u prethodnoj sekciji. U ovom slučaju, poziv `TestBed.configureTestingModule()` je upotrebljen da se podesi `NgModule` neophodan za testiranje.

I priloženog listinga metode `configureMusicTests()` moguće je primetiti da je `NgModule` konfiguracija data u formi argumenta koji poseduje ključeve:

1. `imports`;
2. `providers`.

Pod ključem `imports` vrši se dodavanje:

- `RouterTestingModule` i njegovo podešavanje preko `routerConfig` - ovo podešava rute za testiranje;
- `TestModule` - to je `NgModule` koji deklariše sve komponente koje će biti testirane.

Pod ključem `providers` je obezbeđeno:

- `MockSpotifyService` (preko `mockSpotifyService.getProviders()`), i
- `ActivatedRoute`.

Sada je moguće ozbiljnije pozabaviti se podešavanjem rutera.

## RUTER (ROUTER)

*Bitan segment testiranja je izbor ruta koje će biti korišćene u testovima.*

Bitan segment testiranja je izbor ruta koje će biti korišćene u testovima. Postoje brojni načini da se to uradi, ovo je jedan od njih (datoteka: `/src/app/test/test.module.ts`):

```
@Component({
 selector: 'blank-cmp',
 template: ``
})
export class BlankCmp { }

@Component({
 selector: 'root-cmp',
 template: `<router-outlet></router-outlet>`
})
```

```
export class RootCmp {
}

export const routerConfig: Routes = [
 { path: '', component: BlankCmp },
 { path: 'search', component: SearchComponent },
 { path: 'artists/:id', component: ArtistComponent },
 { path: 'tracks/:id', component: TrackComponent },
 { path: 'albums/:id', component: AlbumComponent }
];
```

Ovde umesto preusmeravanja (koje bi bilo primenjeno u realnom ruteru) za prazan URL, koristi se jednostavno *BlankCmp*.

Naravno, ukoliko je neophodno korišćenje identičnih podešavanja *RouterConfig* kao u komponenti najvišeg nivoa, neophodno ih je negde eksportovati, pa importovati ovde.

Ukoliko postoji još složeniji scenario u kojem postoji potreba za većim brojem različitih konfiguracija ruta, moguće je čak prihvati parametar u funkciji, npr. *musicTestProviders()*, gde se svaki put koristi nova konfiguracija rutera.

Ovde su dostupne brojne mogućnosti, a bira se ono što najbolje odgovara trenutnoj situaciji i razvojnog (test) timu. Prikazana konfiguracija funkcioniše u najvećem broju slučajeva gde su rute relativno statične i jedna konfiguracija funkcioniše za sve testove.

Sada, kada su sve zavisnosti dostupne, kreira se nov ruter (objekat Router) i za njega se poziva *r.initialNavigation()*.

Servis *ActivatedRoute* čuva informaciju o "tekućoj ruti". Zahteva ruter kao zavisnost koja se dodaje u *deps* i potom se ubrizgava (umeće).

*MockSpotifyService* se odnosio na testiranje servisa *SpotifyServis* preko simuliranja (*mocking*) HTTP biblioteke. Ovde će biti simuliran celokupan servis i na ovome će se bazirati naredno izlaganje.

## SIMULIRANE (MOCKING) ZAVISNOSTI

*Objektne promenljive simuliranog (mock) servisa imaće ulogu "špijunskih objekata" (spies).*

U nastavku je potrebno zaviriti u kreirani folder test i pronaći datoteku koja je od posebnog značaja za dalju diskusiju: *spotify.service.mock.ts*. U okviru ove datoteke čuva se simulirani (*mock*) servis pod nazivom *MockSpotifyService*, čija definicija započinje sledećim listingom (datoteka: */src/app/test/spotify.service.mock.ts*):

```
import {SpyObject} from './test.helpers';
import {SpotifyService} from '../spotify.service';

export class MockSpotifyService extends SpyObject {
 getAlbumSpy;
 getArtistSpy;
```

```
getTrackSpy;
searchTrackSpy;
mockObservable;
fakeResponse;
```

Prikazanim listingom je deklarisana klasa `MockSpotifyService`, koja predstavlja simuliranu verziju realnog servisa `SpotifyService`. Objektne promenljive ove klase imaće ulogu "špijuna" (`spies`) - specifičnih tipova simuliranih (mock) objekata.

## ŠPIJUNSKI OBJEKTI (SPIES)

*Špijunski objekti (Spies) predstavljaju specifične tipove simuliranih (mock) objekata.*

Špijunski objekti (`Spies`) predstavljaju specifične tipove simuliranih (`mock`) objekata čijom primenom je moguće imati sledeće koristi:

1. moguće je simulirati povratne vrednosti;
2. moguće je prebrojati koliko je puta metoda pozvana i sa kojim parametrima.

Sa ciljem primene špijunkih objekata u Angular okviru, koristi se unutrašnja klasa `SpyObject` (koristi je `Angular` za samotestiranje). Moguće je deklarisati klasu kreiranjem novog `SpyObject` objekta ili je moguće kreirati vlastitu simuliranu (`mock`) klasu koja nasleđuje klasu `SpyObject`, kao što će to ovde biti urađeno.

Odlična stvar, u vezi sa nasleđivanjem ili primenom navedene klase, jeste mogućnost primene metode `spy()`. Metodu je dozvoljava redefinisanje i diktiranje povratne vrednosti (takođe i posmatranje i obezbeđivanje da je metoda pozvana).

Metoda `spy()` se koristi unutar konstruktora klase (nastavlja se sa datotekom: `/src/app/test/spotify.service.mock.ts`):

```
constructor() {
 super(SpotifyService);

 this.fakeResponse = null;
 this.getAlbumSpy = this.spy('getAlbum').and.returnValue(this);
 this.getArtistSpy = this.spy('getArtist').and.returnValue(this);
 this.getTrackSpy = this.spy('getTrack').and.returnValue(this);
 this.searchTrackSpy = this.spy('searchTrack').and.returnValue(this);
}
```

Prva linija koda konstruktora poziva `SpyObject` konstruktor, prosleđujući konkretnu klasu koja se simulira (žarg. mokuje). Poziv `super(...)` je opcionalan, ali prilikom simuliranja klase naslediće sve metode posmatrane klase, pa, na taj način, moguće je redefinisati samo deliće koji se testiraju.

Nakon poziva konstruktora super klase, obavlja se inicijalizacija polja `fakeResponse`, koje će kasnije biti korišćeno, na `null`.

Sada se deklarišu "špijuni" koji de zameniti metode konkretnе klase. Posedovanjem ovih referenci biće omogućeno podešavanje očekivanja i simuliranje odgovora tokom pisanja testova.

## ŠPIJUNSKI OBJEKTI (SPIES) KROZ DODATNO RAZMATRANJE

*Uvođenje dodatnih metoda simuliranog servisa.*

Kada se koristi servis `SpotifyService` unutar klase `ArtistComponent`, realna metoda `getArtist()` vraća tip `Observable`, a metoda koja se iz komponenata poziva je `subscribe()` (datoteka: `/src/app/artist/artist.component.ts`):

```
ngOnInit(): void {
 this.spotify
 .getArtist(this.id)
 .subscribe((res: any) => this.renderArtist(res));
}
```

Međutim, u simuliranom servisu moguće je postupiti malo drugačije. Umesto vraćanja `Observable` vrednosti metodom `getArtist()`, biće vraćen `this` - konkretan primerak servisa `MockSpotifyService`. To znači da će povratna vrednost za `this.spotify.getArtist(this.id)` biti `MockSpotifyService`.

Međutim, postoji jedan problem sa ovim: komponenta `ArtistComponent` očekuje da se pretplati na `Observable`. Da bi ovo bilo uzeto u obzir, neophodno je definisati metodu `subscribe()` u simuliranom servisu `MockSpotifyService` (datoteka: `/src/app/test/spotify.service.mock.ts`):

```
subscribe(callback) {
 callback(this.fakeResponse);
}
```

Sada je moguće pozvati ovu metodu za instancu simuliranog servisa, istovremeno pozivajući argument `callback`, čineći da se asinhroni poziv izvršava sinhrono.

Takođe primećuje se poziv funkcije `callback()` sa argumentom `this.fakeResponse`. Ovo upućuje na sledeću metodu (datoteka: `/src/app/test/spotify.service.mock.ts`):

```
setResponse(json: any): void {
 this.fakeResponse = json;
}
```

Ova metoda ne menja ništa u konkretnom servisu, ali kao pomoćna metoda dozvoljava test kodu da podesi dati odgovor (koji će doći iz servisa u konkretnu klasu) i na taj način simulira različite odgovore.

```
getProviders(): Array<any> {
 return [{ provide: SpotifyService, useValue: this }];
}
```

Poslednja metoda je takođe pomoćna metoda koja se koristi za provajdere  [TestBed.configureTestingModuleTestingModule](#)  - detaljnije kada bude demonstriran test kod komponente.

## OBJEDINJEN KOD SIMULIRANOG SERVISA

*Objedinjen kod simuliranog servisa se prilaže za sticanje šire slike.*

Objedinjen kod simuliranog servisa se prilaže za sticanje šire slike u vezi sa njegovom definicijom.

```
import {SpyObject} from './test.helpers';
import {SpotifyService} from '../spotify.service';

export class MockSpotifyService extends SpyObject {
 getAlbumSpy;
 getArtistSpy;
 getTrackSpy;
 searchTrackSpy;
 mockObservable;
 fakeResponse;

 constructor() {
 super(SpotifyService);

 this.fakeResponse = null;
 this.getAlbumSpy = this.spy('getAlbum').and.returnValue(this);
 this.getArtistSpy = this.spy('getArtist').and.returnValue(this);
 this.getTrackSpy = this.spy('getTrack').and.returnValue(this);
 this.searchTrackSpy = this.spy('searchTrack').and.returnValue(this);
 }

 subscribe(callback) {
 callback(this.fakeResponse);
 }

 setResponse(json: any): void {
 this.fakeResponse = json;
 }

 getProviders(): Array<any> {
 return [{ provide: SpotifyService, useValue: this }];
 }
}
```

Video materijal: **EP 13.4 - Angular / Unit Testing / Mocks & Spies** - trajanje 12:40.

**Ova lekcija sadrži video materijal. Ukoliko želite da pogledate ovaj video morate da otvorite LAMS lekciju.**

## KREIRANJE TEST KODA

*Stavljanjem svih potrebnih zavisnosti pod kontrolu, pristupa se kreiranju test slučajeva.*

Stavljanjem svih potrebnih zavisnosti pod kontrolu, pristupa se kreiranju test slučajeva. Započinje se pisanje testa za komponentu [ArtistComponent](#). Po ustaljenom šablonu, početak je vezan za import instrukcije. Takođe, ovde je izabran pristup da se test datoteka nalazi u istom folderu kao i klasa koju testira (`/src/app/artist/artist.component.spec.ts`).

```
import {
 async,
 ComponentFixture,
 TestBed,
 inject,
 fakeAsync,
} from '@angular/core/testing';
import { Router } from '@angular/router';
import { Location } from '@angular/common';
import {
 advance,
 createRoot,
 RootCmp,
 configureMusicTests
} from '../test/test.module';

import { MockSpotifyService } from '../test/spotify.service.mock';
import { SpotifyService } from '../spotify.service';
import { ArtistComponent } from './artist.component';
```

Sada, pre nego što testovi počnu da se izvršavaju, poziva se `configureMusicTests()` da obezbedi pristup provajderima u svakom testu.

```
describe('ArtistComponent', () => {
 beforeEach(async(() => {
 configureMusicTests();
 }));
```

U nastavku, prelazi se na testiranje koda kojim je opisana inicijalizacija komponente.

## TESTIRANJE INICIJALIZACIJE KOMPONENTE

*Prelazi se na testiranje koda kojim je opisana inicijalizacija komponente.*

Kao što je istaknuto, prelazi se na testiranje koda kojim je opisana inicijalizacija komponente. Prvo je neophodno sagledati šta se dešava prilikom inicijalizacije komponente *ArtistComponent*:

```
export class ArtistComponent implements OnInit {
 id: string;
 artist: Object;

 constructor(private route: ActivatedRoute, private spotify: SpotifyService,
 private location: Location) {
 route.params.subscribe(params => { this.id = params['id']; });
 }

 ngOnInit(): void {
 this.spotify
 .getArtist(this.id)
 .subscribe((res: any) => this.renderArtist(res));
 }

 //ostatak koda klase
}
```

Tokom kreiranja komponente korišćeno je *route.params* za dobijanje parametra *id* tekuće rute i njegovo čuvanje u atributu *id* posmatrane klase. Kada je komponenta inicijalizovana, *Angular* pokreće metodu *ngOnInit()*, a to je obavezno jer komponenta implementira interfejs *OnInit*. Dalje, koristi se servis *SpotifyService* za dobijanje objekta *artist* za primljenu *id* vrednost, a zatim se vrši poziv *subscribe()* - prijavljivanje na vraćeni *Observable*. Kada je artist konačno dostupan, poziva se *renderArtist()* metoda uz prosleđivanje podataka o objektu *artist*.

Važna ideja je upotreba umetanja zavisnosti za dobijanje servisa *SpotifyService*, međutim, kreiran je simulirani *MockSpotifyService*!

Sa ciljem testiranja navedenog ponašanja, neophodno je uraditi sledeće:

1. koristiti ruter za navigaciju ka komponenti *ArtistComponent* što će inicijalizovati komponentu;
2. proveriti servis *MockSpotifyService* i uveriti se da je komponenta *ArtistComponent*, zaista, pokušala da dobije *artist* objekat sa odgovarajućim *id*.

Sledi kod testa:

```
describe('initialization', () => {
 it('retrieves the artist', fakeAsync(
 inject([Router, SpotifyService],
 (router: Router,
 mockSpotifyService: MockSpotifyService) => {
 const fixture = createRoot(router, RootCmp);

 router.navigateByUrl('/artists/2');
 advance(fixture);
```

```
 expect(mockSpotifyService.getArtistSpy).toHaveBeenCalledWith('2');
 }));
});
```

## TESTIRANJE INICIJALIZACIJE KOMPONENTE - FAKEASYNC I ADVANCE

*Prvi korak prilikom kreiranja testa bio je umotavanje testa u fakeAsync() metodu.*

Ako se usmeri pažnja na upravo priloženi listing, može se primetiti da je prvi korak prilikom kreiranja testa bio umotavanje testa u `fakeAsync()` metodu. Na ovaj način uspostavljena je veća kontrola nad detekcijom promena i pojavljivanjem asinhronih operacija. Kao posledica urađenog javlja se potreba da se eksplicitno ukaže komponenti da ona mora da detektuje promene nakon što se promene načine u testu. Tokom izvođenja testova, manipulisanje procesom detekcije promena mora da se obavlja sa posebnom pažnjom.

Ako se preskoči par linija koda, uočava se primena funkcije pod nazivom `advance()` koja potiče iz datoteke `/src/app/test/test.module.ts`. Sledećim listingom je data definicija navedene metode:

```
export function advance(fixture: ComponentFixture<any>): void {
 tick();
 fixture.detectChanges();
}
```

Prikazana metoda radi dve stvari:

1. ukazuje komponenti da detektuje promene i
2. poziva metodu `tick()`.

Kada se koristi `fakeAsync()`, brojači (timers) su zapravo sinhroni i pozivom metode `tick()` simulira se asinhroni prolazak vremena.

Praktično rečeno, metoda `advance()` se poziva svaki put kada *Angular* radi nešto "magično": navigacija ka novoj ruti, ažuriranje elementa forme, kreiranje HTTP zahteva i tako dalje.

## UMETANJE ZAVISNOSTI I FUNKCIJA INJECT()

*U testu koji se gradi neophodno je obezrediti određene zavisnosti putem umetanja zavisnosti.*

U testu koji se gradi neophodno je obezrediti određene zavisnosti. Do njih se dolazi putem umetanja zavisnosti. Funkcija `inject()` uzima dva argumenta:

1. niz tokena za umetanje;
2. funkciju unutar koje je obavljeno umetanje zavisnosti.

Koje klase će koristiti `inject()`? Provajderi definisani u `TestBed.configureTestingModuleTestingModule()` pod ključem `providers`. Primećuje se da se ubrizgava:

1. `Router`:
2. `SpotifyService`.

Ruter (Router) koji će biti umetnut je ruter koji je konfigurisan u `configureMusicTests()` pod ključem `providers`.

```
providers: [
 mockSpotifyService.getProviders(),
 {
 provide: ActivatedRoute,
 useFactory: (r: Router) => r.routerState.root, deps: [Router]
 }
]
```

Za servis `SpotifyService`, primećuje se da se traži umetanje tokena `SpotifyService`, ali se dobija `MockSpotifyService`. Malo škakljivo, ali ima smisla s obzirom na sve ono o čemu je do sada diskutovano.

## TESTIRANJE INICIJALIZACIJE KOMPONENTE KROZ DODATNO RAZMATRANJE

*Sa pozivom metode `createRoot()` vraća se nova korenska (root) komponenta.*

Sledećim listingom je izolovan sadržaj našeg stvarnog testa:

```
const fixture = createRoot(router, RootCmp);

 router.navigateByUrl('/artists/2');
 advance(fixture);

 expect(mockSpotifyService.getArtistSpy).toHaveBeenCalledWith('2');
```

Prvo što je moguće primetiti jeste instanca `RootCmp` koju koristi metoda `createRoot()`. Sledećim listingom je data definicija pomoćne metode `createRoot()` koja je definisana unutar datoteke: [/src/app/test/test.module.ts](#).

```
export function createRoot(router: Router,
 componentType: any): ComponentFixture<any> {
 const f = TestBed.createComponent(componentType);
 advance(f);
 (<any>router).initialNavigation();
 advance(f);
 return f;
}
```

Sa pozivom metode `createRoot()` dešava se sledeće:

1. kreira se instanca korenske (`root`) komponente;
2. poziva se `advance()`;
3. podešava se inicijalna navigacija preko rutera;
4. poziva se `advance()`;
5. vraća se nova korenska (`root`) komponenta.

Ovo bi trebalo dobro zapamtiti jer se radi uvek kada se testira komponenta koja zavisi od rutiranja, pa je jako zgodno imati ovakvu pomoćnu funkciju.

Ponovo se koristi biblioteka  `TestBed`  za poziv  `TestBed.createComponent()` . Ova funkcija kreira komponentu odgovarajućeg tipa.

U nastavku se koristi ruter za navigaciju ka URL `/artists/2` i poziva se `advance()`. Kada se vrši navigacija ka pomenutom linku, komponenta `ArtistComponent` bi trebalo da je inicijalizovana, pa se tvrdi da je `getArtist()` metoda, servisa `SpotifyService`, pozvana sa odgovarajućom vrednošću.

## TESTIRANJE METODA KOMPONENTE (ARTISTCOMPONENT)

*Provera poziva metode back() - da li ruter preusmerava korisnika na prethodnu lokaciju.*

Još jednu funkcionalnost komponente `ArtistComponent` bi trebalo istaći - postojanje linka (`href`) koji poziva funkciju `back()` (datoteka: `/src/app/artist/artist.component.ts`):

```
back(): void {
 this.location.back();
}
```

U nastavku je upravo cilj provera poziva metode `back()` odgovarajućim testom - provera da li ruter preusmerava korisnika na prethodnu lokaciju.

Tekuća lokacija poseduje stanje kontrolisano preko servisa `Location`. Kada je neophodno preusmeriti korisnika na prethodnu lokaciju, koristi se metoda `back()` servisa `Location`.

Sledećim listingom je demonstrirano kako se testira metoda `back()` (datoteka: `/src/app/artist/artist.component.spec.ts`):

```
describe('back', () => {
 it('returns to the previous location', fakeAsync(
 inject([Router, Location],
 (router: Router, location: Location) => {
 const fixture = createRoot(router, RootCmp);
 expect(location.path()).toEqual('/');

 router.navigateByUrl('/artists/2');
 advance(fixture);
```

```
expect(location.path()).toEqual('/artists/2');

const artist = fixture.debugElement.children[1].componentInstance;
artist.back();
advance(fixture);

expect(location.path()).toEqual('/');
})));
});
```

Inicijalna struktura je slična: vrši se umetanje zavisnosti i kreira se nova komponenta. Postoji novo očekivanje - tvrdi se da je `location.path()` jednako očekivanoj vrednosti ('/').

Javlja se i nova ideja: pristup metodama same komponente `ArtistComponent`. Referenca na instancu tipa `ArtistComponent` dobija se preko sledeće linije koda:

```
fixture.debugElement.children[1].componentInstance
```

Sada kada postoji instanca komponente, ona može da poziva direktno metode komponente, na primer: `artist.back()`.

Nakon poziva metode `back()`, ponovo se poziva `advance()` i, konačno, proverava se da li `location.path()` ima očekivanu vrednost.

## TESTIRANJE ŠABLONA KOMPONENTE ARTISTCOMPONENT

*Testira se da li šablon komponente prikazuje korisnika.*

Poslednja stvar koju bi trebalo proveriti za komponentu `ArtistComponent` jeste da li šablon komponente prikazuje korisnika. Sledi listing šablona komponente `ArtistComponent`:

```
<div *ngIf="artist">
 <h1>{{ artist.name }}</h1>

 <p>

 </p>

 <p><a href (click)="back()">Nazad</p>
</div>
```

Neophodno je podsetiti se da je objektna promenljiva `artist` podešena pozivom metode `getArtist()` servisa `SpotifyService`. Budući da se simulira `SpotifyService` sa `MockSpotifyService`, podaci koji bi trebalo da se pojave u šablonu trebalo bi da odgovaraju povratnim vrednostima iz `mockSpotifyService`. To se radi na sledeći način (datoteka: `/src/app/artist/artist.component.spec.ts`):

```
describe('renderArtist', () => {
 it('renders album info', fakeAsync(
 inject([Router, SpotifyService],
 (router: Router,
 mockSpotifyService: MockSpotifyService) => {
 const fixture = createRoot(router, RootCmp);

 const artist = {name: 'ARTIST NAME', images: [{url: 'IMAGE_1'}]};
 mockSpotifyService.setResponse(artist);

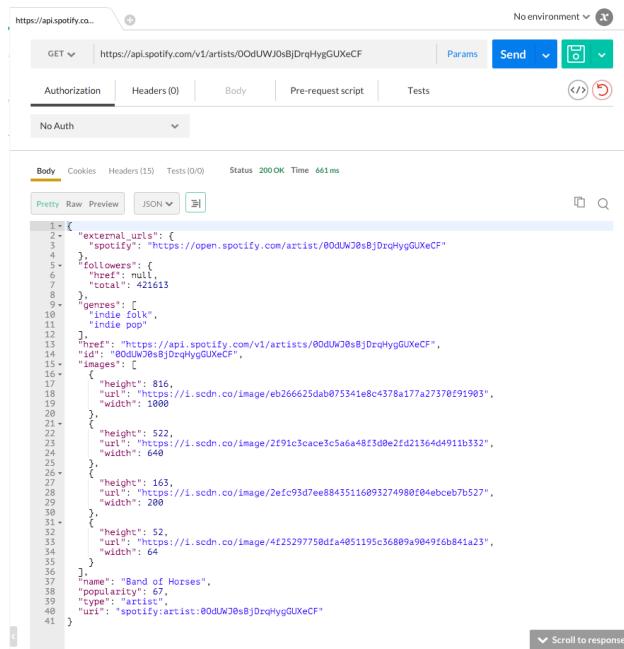
 router.navigateByUrl('/artists/2');
 advance(fixture);

 const compiled = fixture.debugElement.nativeElement;

 expect(compiled.querySelector('h1').innerHTML).toContain('ARTIST NAME');
 expect(compiled.querySelector('img').src).toContain('IMAGE_1');
 })));
});
```

Prva stvar, koja je ovde nova, jeste manuelno podešavanje odgovora instance servisa `mockSpotifyService` sa `setResponse()`.

Promenljiva `artist` je reprezentacija onoga što se dobija iz `Spotify API` - ja, pozivom `GET` zahteva <https://api.spotify.com/v1/artists/{id}>. Sledećom slikom je prikazan stvaran `JSON`:



Slika 4.1 Postman - Spotify Get Artist krajnja tačka (Endpoint) [izvor: autor]

## TESTIRANJE ŠABLONA KOMPONENTE KROZ DOPUNSKA RAZMATRANJA

*Za tekući test dovoljne su osobine name i image.*

Za tekući test dovoljne su osobine *name* i *image*.

Kada se poziva metoda *setResponse()*, taj odgovor objekat će biti upotrebljen za budući poziv bilo koje metode servisa. U ovom slučaju, cilj je da metoda *getArtist()* vrati ovaj odgovor.

U nastavku, preko ruteru se vrši navigacija i izvršava se *advance()* metoda. Sada kada je pogled kreiran, moguće je upotrebiti **DOM** reprezentaciju pogleda komponente za proveru da li je *artist* ispravno prikazan.

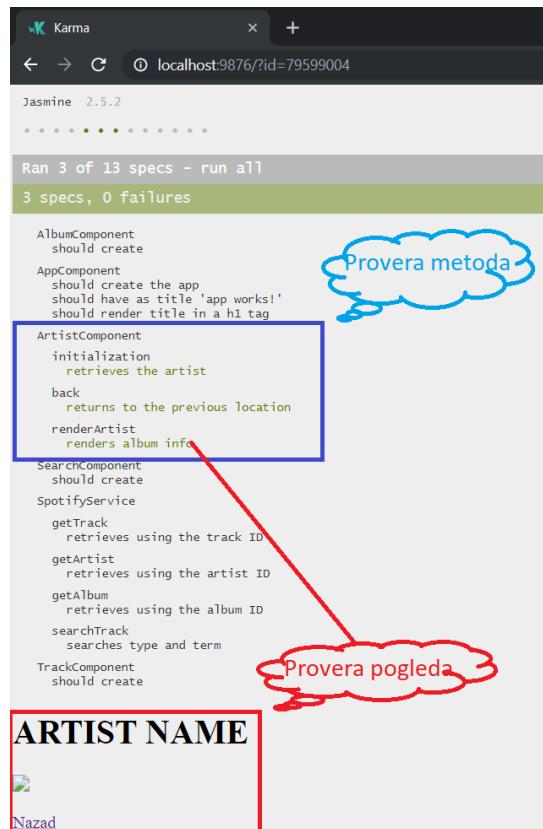
Ovo se postiže uzimanjem osobine *nativeElement* instance klase *DebugElement*, kao u sledećoj liniji koda:

```
debugElement.nativeElement
```

U očekivanjima na kraju testa, jasno je istaknuto da se očekuje pojavljivanje *H1* taga koji sadrži naziv umetnika - konkretno *ARTIST NAME* jer je tako eksplicitno traženo u testu. Za proveru ovakvih uslova, koristi se metoda *querySelector()* objekta tipa *NativeElement*. Ova metoda će vratiti prvi element koji se podudara sa obezbeđenim *CSS* selektorom. Za element *H1* proverava se da li je tekst zaista "*ARTIST NAME*" i za sliku, proverava se da li je vrednost osobine *src* "*IMAGE1*".

Sa ovim je završeno testiranje klase komponente *ArtistComponent*.

U test alatu Karma, rezultati testiranja komponente *ArtistComponent* su prikazani sledećom slikom:



Slika 4.2 Rezultati testiranja komponente *ArtistComponent* [izvor: autor]

## VIDEO MATERIJAL

*EP 13.9 - Angular / Unit Testing / Components - Trajanje 10:21.*

**Ova lekcija sadrži video materijal. Ukoliko želite da pogledate ovaj video morate da otvorite LAMS lekciju.**

**Kompletno urađen prateći primer možete preuzeti i isprobati odmah nakon ovog objekta učenja.**

## ✓ Poglavlje 5

### Testiranje formi

#### PRIMER SA FORMOM

*Neophodan je primer koji obuhvata primenu: FormBuilder-a, validatora i rukovanja događajima.*

Za testiranje ispravnosti da li forma radi ispravno, u kreiranoj Angular aplikaciji, neophodno je vratiti se na projekat sa formama, iz lekcije "L07 - Ugrađene directive i rad sa formama", i fokusirati se na primer [demo-form-with-events](#). Primer sa test slučajevima, biće dostupan za preuzimanje odmah iza ovog objekta učenja.

Navedeni primer je odličan kandidat za demonstraciju kreiranja testova za proveru formi, iz razloga što obuhvata primenu:

- [FormBuilder-a](#),
- validatora i
- rukovanja događajima.

Za podsećanje, sledi kod klase odgovarajuće komponente:

```
import { Component, OnInit } from '@angular/core';
import {
 FormBuilder,
 FormGroup,
 Validators,
 AbstractControl
} from '@angular/forms';

@Component({
 selector: 'app-demo-form-with-events',
 templateUrl: './demo-form-with-events.component.html',
 styles: []
})
export class DemoFormWithEventsComponent implements OnInit {
 myForm: FormGroup;
 sku: AbstractControl;

 ngOnInit() {}

 constructor(fb: FormBuilder) {
 this.myForm = fb.group({
 'sku': ['', Validators.required]
 })
}
```

```
});

this.sku = this.myForm.controls['sku'];

this.sku.valueChanges.subscribe(
 (value: string) => {
 console.log('sku je promenjen u:', value);
 }
);

this.myForm.valueChanges.subscribe(
 (form: any) => {
 console.log('forma je promenjena u:', form);
 }
);

}

onSubmit(form: any): void {
 console.log('Prosleđena vrednost je:', form.sku);
}

}
```

Sledećim listingom je priložen i šablon ove komponente:

```
<div class="ui raised segment">
 <h2 class="ui header">Demo forme: sa događajima</h2>
 <form [formGroup]="myForm"
 (ngSubmit)="onSubmit(myForm.value)"
 class="ui form">

 <div class="field"
 [class.error]="!sku.valid && sku.touched">
 <label for="skuInput">SKU</label>
 <input type="text"
 class="form-control"
 id="skuInput"
 placeholder="SKU"
 [FormControl]="sku">
 <div *ngIf="!sku.valid"
 class="ui error message">SKU nije validan</div>
 <div *ngIf="sku.hasError('required')"
 class="ui error message">SKU je obavezan</div>
 </div>

 <div *ngIf="!myForm.valid"
 class="ui error message">Forma nije validna</div>

 <button type="submit" class="ui button">Pošalji</button>
 </form>
</div>
```

## PRIMER SA FORMOM KROZ DODATNE NAPOMENE

*Da testovi ne bi bili zavisni od eksternih zavisnosti, neophodno je simulirati spoljašnje zavisnosti.*

Veoma je bitno, pre prelaska na pisanje testova, sagledavanje ponašanja priloženog koda:

1. kada nije uneta vrednost za polje *SKU*, biće prikazane dve greške validacije: "*SKU nije validan*" i "*SKU je obavezan*";
2. kada se promeni vrednost za polje *SKU*, prikazuje se log poruka u konzoli veb pregledača;
3. kada se forma promeni, takođe se prikazuje se log poruka u konzoli veb pregledača;
4. kada se forma potvrđena, klikom na dugme "*Pošalji*", prikazuje se konačna log poruka u konzoli veb pregledača.

Očigledno je da postoji jedna očigledna eksterna zavisnost, a to je konzola. Da testovi ne bi bili zavisni od eksternih zavisnosti, neophodno je simulirati (žarg. mokovati) spoljašnje zavisnosti.

Sledećom slikom je prikazana forma koja je rezultat izvršavanja koda priloženog prethodnim listinzima.

The screenshot shows a simple web form with a light gray background. At the top, the title "Demo forme: sa događajima" is centered. Below the title is a horizontal input field with the label "SKU" to its left. Inside the input field, the value "123sađ" is typed. At the bottom of the form is a single button with the text "Pošalji" in a dark gray font. The entire form is enclosed in a thin gray border.

Slika 5.1 Forma kreirana komponentom primera [izvor: autor]

## KREIRANJE CONSOLESPLY OBJEKTA

*Uместо upotrebe SpyObject objekata za kreiranje mock simulacije biće korišćen ConsoleSpy.*

U ovom slučaju, umesto upotrebe obrađenih *SpyObject* objekata za kreiranje *mock* simulacije, biće upotrebljeno nešto jednostavnije, jer sve što se koristi iz konzole je *log* metoda.

Biće zamenjena originalna instanca *console*, koja se čuva u objektu *window.console* objektom koji je kontrolisan i pripada tipu *ConsoleSpy*.

Za studente je od interesa sledeća datoteka, priloženog primera */src/app/utils.ts*:

```
export class ConsoleSpy {
 public logs: string[] = [];
 log(...args) {
 this.logs.push(args.join(' '));
 }
 warn(...args) {
 this.log(...args);
 }
}
```

Pa umesto pisanja u konzoli, `log` objekti će biti čuvani u nizu. Ako kod test poziva `console.log` tri puta:

```
console.log('Prva poruka ', 'je', 123);
console.log('Druga poruka');
console.log('Treća poruka');
```

očekivano je da `_logs` polje čuva sledeći niz:

`ConsoleSpy` je objekat koji preuzima sve što je prikazano u `log` - u, jednostavno prevodi u string i čuva kao internu listu svega što je prikazano u konzoli veb pregledača preko `log` metode.

```
['Prva poruka je 123', 'Druga poruka', 'Treća poruka'].
```

## CONSOLESPI INSTALACIJA

*Potrebno je obaviti deklarisanje dveju promenljivih: `originalConsole` i `fakeConsole`.*

Za korišćenje ovakvog tipa špijunskega objekata u testovima, potrebno je obaviti deklarisanje dveju promenljivih:

- `originalConsole` - čuva referencu na originalnu `console` instancu;
- `fakeConsole` - čuva simuliranu (`mock`) verziju konzole.

takođe, obavlja se i deklarisanje nekolicine promenljivih koje su od pomoći u testiranju ulaza i elemenata forme. Ovde je za studente interesantna test datoteka:[`/src/app/demo-form-with-events/demo-form-with-events.component.spec.ts`](#).

```
describe('DemoFormWithEventsComponent', () => {
 let component: DemoFormWithEventsComponent;
 let fixture: ComponentFixture<DemoFormWithEventsComponent>;
 let originalConsole, fakeConsole;
 let el, input, form;
```

Odmah nakon navedenog moguće je instalirati simuliranu konzolu i specificirati provajdere:

```
beforeEach(async(() => {
 // menja realan window.console špijunom
 fakeConsole = new ConsoleSpy();
 originalConsole = window.console;
 (<any>window).console = fakeConsole;

 TestBed.configureTestingModule({
 imports: [FormsModule, ReactiveFormsModule],
 declarations: [DemoFormWithEventsComponent]
 })
 .compileComponents();
}));
```

Ako se pogleda priloženi test kod, svaki put kada se menja originalna konzola simuliranim originalna instanca će biti sačuvana (linija koda 4).

Konačno, pozivom metode `afterAll()`, vraća se originalna instanca konzole kako bismo bili sigurni da ne prodire u druge testove.

```
// obnavlja realnu konzolu
afterAll(() => (<any>window).console = originalConsole);
```

## PODEŠAVANJE TEST MODULA

*U hvataču `beforeEach` se obavlja poziv `TestBed.configureTestingModuleTestingModule` za `NgModule` podešavanje.*

Primećuje se, takođe, i primena hvatača `beforeEach` unutar kojeg se obavlja poziv `TestBed.configureTestingModuleTestingModule` - za podešavanje korenskog (`root`) `NgModule` za ovaj test.

U ovom slučaju vrši se importovanje dva modula formi i deklarisanje komponente `DemoFormWithEvents`.

Sada kada postoji potpuna kontrola nad formom, moguće je pisati testove za proveru forme.

## TEST KOD ZA PROVERU FORME

*Proveravaju se greške validacije i događaji forme.*

Proveravaju se greške validacije i događaji forme. Prva stvar koju je neophodno obaviti jeste uzimanje referenci za SKU polje za unos i elemente forme (datoteka: `/src/app/demo-form-with-events/demo-form-with-events.component.1.spec.ts`).

```
it('validates and triggers events', fakeAsync(() => {
 fixture = TestBed.createComponent(DemoFormWithEventsComponent);
 component = fixture.componentInstance;
 el = fixture.debugElement.nativeElement;
 input = fixture.debugElement.query(By.css('input')).nativeElement;
```

```
form = fixture.debugElement.query(By.css('form')).nativeElement;
fixture.detectChanges();
```

Poslednja linija govori *Angularu* da dovrši sve promene koje su u toku, na sličan način kao u prethodnom slučaju sa rutiranjem. Sledeće, podešava se SKU ulazna vrednost na prazan string:

```
input.value = '';
dispatchEvent(input, 'input');
fixture.detectChanges();
tick();
```

Ovde se primećuje upotreba metode *dispatchEvent()* kojom se obaveštava Angular da je ulazni element promenjen i po drugi put se pokreće detekcija promena. Konačno, poziva se metoda *tick()* koja obezbeđuje da će sav pokrenuti asinhroni kod biti izvršen.

Razlog zašto se ovde koriste *fakeAsync()* i *tick()* se krije u činjenici da je neophodno obezbediti pokretanje događaja forme. Ako bi se, suprotno, koristili *async()* i *inject()* moglo bi doći do okončanja koda pre pokretanja događaja.

Kada je promenjena ulazna vrednost, moguće je pristupiti proveri da li validacija funkcioniše. Biće prosleđen upit elementu komponente (preko promenljive *el*) za sve elemente potomke (poruke sa greškama), a potom će uslediti provera da li su obe poruke sa greškama prikazane.

```
// test poruka sa greškama
let msgs = el.querySelectorAll('.ui.error.message');
expect(msgs[0].innerHTML).toContain('SKU nije validan');
expect(msgs[1].innerHTML).toContain('SKU je obavezan');
```

Sada je moguće uraditi nešto slično već rađenom, ali umesto praznog stringa ulaz dobija vrednost u formi stringa "XYZ".

```
input.value = 'XYZ';
dispatchEvent(input, 'input');
fixture.detectChanges();
tick();
```

Provera da li su sve poruke sa greškama nestale:

```
msgs = el.querySelectorAll('.ui.error.message');
expect(msgs.length).toEqual(0);
```

Konačno, pokreće se događaj "submit" koji odgovara potvrdi unosa u polje forme.

```
fixture.detectChanges();
dispatchEvent(form, 'submit');
tick();
```

Konačno, proverava se da li je događaj "ispaljen" preko poruke koja bi se pojavila u konzoli veb pregledača kada se u aplikaciji popuni i potvrdi forma.

```
expect(fakeConsole.logs).toContain('you submitted value: XYZ');
```

```
expect(fakeConsole.logs).toContain('you submitted value: XYZ');
```

```
expect(fakeConsole.logs).toContain('you submitted value: XYZ');
```

## POKRETANJE I PROVERA TESTA

Ovako definisan test se pokreće i prolazi.

Ovako definisan test se pokreće i prolazi. A to je moguće primetiti na sledećoj slici:

DemoFormWithEventsComponent (Long)  
validates and triggers events

Slika 5.2 Uspešna test provera [izvor: autor]

Moglo se ići dalje i dodati nove verifikacije, za ostala dva događaja koje forma pokreće: SKU promena i promena forme. Međutim, test bi poprilično narastao.

Generalno, test radi ono što se od njega očekuje, ali ipak ima ozbiljne nedostatke;

- dugačak *it* uslov (više od 5 -10 linija);
- više od jednog ili dva očekivanja po *it* uslovu;
- reč "*and*" u opisu testa.

## REFAKTORING KREIRANOG TESTA

Započinje se optimizacija koda testa. Test je mnogo čistiji i fokusira se na jednu stvar.

Uočeni problemi mogu biti rešeni izvlačenjem koda koji kreira komponentu i uzima element komponente, a takođe i elemente za ulaz i formu. Za kreiranje optimizovanog testa pažnja će biti na drugoj test datoteci: [forms/src/app/demo-form-with-events/demo-form-with-events.component.spec.ts](#):

```
fixture = TestBed.createComponent(DemoFormWithEventsComponent);
```

Kod metode *createComponent()* je prilično jednostavan: kreira komponentu preko poziva *TestBed.createComponent* donoseći sve potrebne elemente i pozive *detectChanges()*.

Prva stvar koju je neophodno proveriti je prazno *SKU* polje i trebalo bi da rezultuje pojavom dveju poruka sa greškama:

```
it('displays errors with no sku', fakeAsync(() => {
 input.value = '';
 dispatchEvent(input, 'input');
```

```
fixture.detectChanges();

// test poruka sa greškama
const msgs = el.querySelectorAll('.ui.error.message');
expect(msgs[0].innerHTML).toContain('SKU nije validan');
expect(msgs[1].innerHTML).toContain('SKU je obavezan');
});
```

Test je mnogo čistiji i fokusira se na jednu stvar. Odlično urađen posao!

Koristeći novu strukturu, dodavanje sledećeg testa je poprilično olakšano. Ovaj put, cilj je provera da li su poruke sa greškama nestale dodavanjem vrednosti u **SKU** polje.

```
it('displays no errors when sku has a value', fakeAsync(() => {
 input.value = 'XYZ';
 dispatchEvent(input, 'input');
 fixture.detectChanges();

 const msgs = el.querySelectorAll('.ui.error.message');
 expect(msgs.length).toEqual(0);
}));
```

## REFAKTORING KREIRANOG TESTA, METODE FAKEASYNC I TICK()

*Kombinacija fakeAsync i tick() se koristi samo kada se proverava da li je nešto dodato u konzolu.*

Sledeći bonus: koristi se kombinacija fakeAsync i tick() samo kada se proverava da li je nešto dodato u konzolu, jer je to sve što rukovalac događajima forme radi.

Sledeći test to upravo radi - kada se **SKU** vrednost promeni, trebalo bi u konzoli da se javi odgovarajuća poruka.

```
it('handles sku value changes', fakeAsync(() => {
 input.value = 'XYZ';
 dispatchEvent(input, 'input');
 tick();

 expect(fakeConsole.logs).toContain('sku changed to: XYZ');
}));
```

Jako sličan kod se tiče testova za pokrivanje obeju promena forme.

```
it('handles form changes', fakeAsync(() => {
 input.value = 'XYZ';
 dispatchEvent(input, 'input');
 tick();
```

```
 expect(fakeConsole.logs).toContain('form changed to: [object Object]');
});
```

Test za proveru potvrđivanje forme:

```
it('handles form submission', fakeAsync((tcb) => {
 input.value = 'ABC';
 dispatchEvent(input, 'input');
 tick();

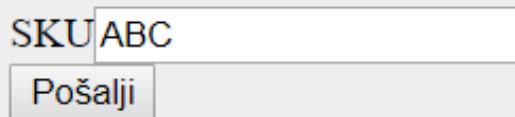
 fixture.detectChanges();
 dispatchEvent(form, 'submit');
 tick();

 expect(fakeConsole.logs).toContain('you submitted value: ABC');
}));
```

Kada se pokrene test, dobija se mnogo lepši izlaz:

```
DemoFormWithEventsComponent
 displays errors with no sku
 displays no errors when sku has a value
 handles sku value changes
 handles form changes
 handles form submission
```

## Demo form: sa događajima



Slika 5.3 Izlaz iz testa posle refactoringa [izvor: autor]

## DODATNO RAZMATRANJE

*Posebno bitna stvar iz obrađenog refakoringa test koda odmah se primeti ako nešto krene pogrešno.*

Posebno bitna stvar iz obrađenog refakoringa test koda odmah se primeti ako nešto krene pogrešno. Ako se pažnja ponovo usmeri na kod komponente, u kojem će biti promenjena poruka koja se prikazuje u konzoli kada je forma potvrđena (dodato jedno slovo a na kraj poruke), sa ciljem forsiranja pada testa:

```
onSubmit(form: any): void {
 console.log('you submitted valuea:', form.sku);
}
```

u alatu Karma će se dobiti sledeći test izlaz:

```
DemoFormWithEventsComponent handles form submission
Expected ['sku changed to: ABC', 'form changed to: [object Object]'], ['you submitted value: ABC'] to contain 'you submitted value: ABC'.
Error: Expected ['sku changed to: ABC', 'form changed to: [object Object]'], ['you submitted value: ABC'] to contain 'you submitted value: ABC'.
at stack (http://localhost:9876/base/node_modules/jasmine-core/jasmine.js?7916005cc407925f4764668d1d0488bd59258f5d:1610:14)
at toContain (http://localhost:9876/base/node_modules/jasmine-core/jasmine.js?7916005cc407925f4764668d1d0488bd59258f5d:1640:27)
at Spec.expectationResultFactory (http://localhost:9876/base/node_modules/jasmine-core/jasmine.js?7916005cc407925f4764668d1d0488bd59258f5d:155:18)
at Spec.addExpectationResult (http://localhost:9876/base/node_modules/jasmine-core/jasmine.js?7916005cc407925f4764668d1d0488bd59258f5d:342:34)
at Expectation.addExpectationResult (http://localhost:9876/base/node_modules/jasmine-core/jasmine.js?7916005cc407925f4764668d1d0488bd59258f5d:159:21)
at Expectation.toContain (http://localhost:9876/base/node_modules/jasmine-core/jasmine.js?7916005cc407925f4764668d1d0488bd59258f5d:1564:12)
at Object.<anonymous> (http://localhost:9876/_karma_webpack_/src/app/demo-form-with-events/demo-form-with-events.component.spec.ts:97:30)
at Object.<anonymous> (http://localhost:9876/_karma_webpack_/src/app/demo-form-with-events/demo-form-with-events.component.spec.ts:97:30)
at ZoneDelegate..node_modules/zone.js/dist/zone.js.ZoneDelegate.invoke (http://localhost:9876/_karma_webpack_/src/app/demo-form-with-events/demo-form-with-events.component.spec.ts:97:30)
at ProxyZoneSpec.push../node_modules/zone.js/dist/zone-testing.js.ProxyZoneSpec.onInvoke (http://localhost:9876/_karma_webpack_/src/app/demo-form-with-events/demo-form-with-events.component.spec.ts:97:30)
```

Slika 5.4 Pad testa zbog nepodudaranja poruke sa očekivanom [izvor: autor]

Nije odmah sve jasno šta je "palo". Međutim, kad se pročita kod greške brzo se uvidi da je pala provera poruke sa greškom vezane za potvrdu unosa u formu.

Kod prvog pristupa, postoji još jedan problem. Nije moguće sa sigurnošću tvrditi da li je to jedina stvar koja je "pukla" prilikom provere koda komponente, budući da ostali uslovi, koji slede nakon pada, neće dobiti šansu da budu izvršeni.

Slika 5 pokazuje pad testa (crvenim slovima) kada je primenjen prvi pristup, a slika 6 prikazuje pad testa nakon refakoringa. Na slici 6 se lepo vidi da su i ostali uslovi provereni (zelenim slovima)

**DemoFormWithEventsComponent (Long)  
validates and triggers events**

Slika 5.5 Pad testa pre refaktoringa [izvor: autor]

**DemoFormWithEventsComponent  
displays errors with no sku  
displays no errors when sku has a value  
handles sku value changes  
handles form changes  
handles form submission**

Slika 5.6 Pad testa posle refaktoringa [izvor: autor]

## VIDEO MATERIJAL

*Testing Angular 4 Reactive Forms - trajanje 22:02.*

**Ova lekcija sadrži video materijal. Ukoliko želite da pogledate ovaj video morate da otvorite LAMS lekciju.**

## ✓ Poglavlje 6

### Dodatni materijali za rad

#### DODATNI MATERIJALI

*Proširivanje znanja sa predavanja.*

1. <https://jasmine.github.io/2.4/introduction.html>
2. <https://alligator.io/angular/testing-async-fakeasync/>
3. <https://nodejs.org/api/querystring.html>

## ▼ Poglavlje 7

# Pokazna vežba 14, testiranje HTTP zahteva

## POKAZNA VEŽBA (TRAJANJE 45 MINUTA)

*Demonstacija testiranja HTTP zahteva u Angular aplikaciji.*

HTTP interakciju je moguće testirati, u aktuelnoj Angular aplikaciji, primenom identične strategije: kreira se simulirana (mock) verzija za klase Http ili HttpClient, budući da su to eksterne zavisnosti.

Ali pošto velika većina aplikacija napisana pomoću okvira kao što je Angular, koristi HTTP interakciju za API interakciju, Angular-ova biblioteka za testiranje već nudi alternativu: HttpTestingController.

Odlična osnova za rad je pokazni primer priložen uz lekciju "L09 - Http i Angular". Studenti mogu da ga preuzmu i da se na praktičan način pozabave testiranjem HTTP zahteva.

Cilj ovih vežbi je demonstracija kako se obavlja testiranje različitih HTTP metoda, poput POST ili DELETE, a takođe i provera ispravnosti HTTP zaglavlja (header) koji se šalju.

### Video materijal:

**Testing Angular HTTP Services - Test Setup with HttpClientTestingModule** - trajanje 5:26

**Ova lekcija sadrži video materijal. Ukoliko želite da pogledate ovaj video morate da otvorite LAMS lekciju.**

## PODEŠAVANJE TESTA ZA POST ZAHTEV

*U prvom koraku, biće kreiran test koji proverava ispravnost POST zahteva za metodu makePost().*

U prvom koraku, biće kreiran test koji proverava ispravnost POST zahteva za metodu makePost(). Ako preuzmete ovaj primer iz L09, od značaja je folder /http/src/app/more-http-requests/more-http-requests.component.ts:

```
makePost(): void {
 this.loading = true;
 this.http
 .post(
 'https://jsonplaceholder.typicode.com/posts',
```

```
 JSON.stringify({
 body: 'bar',
 title: 'foo',
 userId: 1
 })
)
.subscribe(data => {
 this.data = data;
 this.loading = false;
});
}
}
```

Upravo je priložena metoda `makePost()` koja predstavlja osnov za kreiranje sledećeg testa. Cilj je testiranje dveju stvari:

- korektnost zahteva (*POST*) metode;
  - korektnost navedene *URL* adrese.

Sada je navedeno potrebno pretvoriti u test. Prvo je neophodno obaviti podešavanje da test koristi:[\*HttpClientTestingModule\*](#) i [\*HttpTestingController\*](#) (datoteka: `/src/app/more-http-requests/more-http-requests.component.spec.ts`):

```
import {
 async,
 inject,
 ComponentFixture,
 TestBed
} from '@angular/core/testing';

import { HttpClient, HttpRequest, HttpHeaders } from '@angular/common/http';
import {
 HttpTestingController,
 HttpClientTestingModule
} from '@angular/common/http/testing';

import { MoreHttpRequestsComponent } from './more-http-requests.component';

describe('MoreHttpRequestsComponent', () => {
 let component: MoreHttpRequestsComponent;
 let fixture: ComponentFixture<MoreHttpRequestsComponent>;
 let httpMock: HttpTestingController;

 beforeEach(
 async(() => {
 TestBed.configureTestingModule({
 declarations: [MoreHttpRequestsComponent],
 imports: [HttpClientTestingModule]
 });
 })
);

 beforeEach(
 async(
 () => {
 fixture = TestBed.createComponent(MoreHttpRequestsComponent);
 component = fixture.componentInstance;
 fixture.detectChanges();
 }
)
);

 it('should render <h1> element', () => {
 expect(fixture.nativeElement.querySelector('h1')).toBeTruthy();
 });
});
```

```
async(
 inject([HttpTestingController], _httpMock => {
 fixture = TestBed.createComponent(MoreHttpRequestsComponent);
 component = fixture.componentInstance;
 fixture.detectChanges();
 httpMock = _httpMock;
 })
);

afterEach(
 inject([HttpTestingController], (httpMock: HttpTestingController) => {
 httpMock.verify();
 })
);
```

## KREIRANJE TESTA ZA POST ZAHTEV

*Nakon podešavanja zavisnosti, sve je spremno za pisanje i izvođenje testa POST zahteva.*

U priloženom kodu, obavljeno je podešavanje test modula da importuje [HttpClientTestingModule](#). Nakon toga je obavljeno umetanje [HttpTestingController](#) i njegovo čuvanje u varijabli [httpMock](#).

Sve je spremno za pisanje testa:

```
it(
 'performs a POST',
 async(() => {
 component.makePost();

 const req = httpMock.expectOne(
 'https://jsonplaceholder.typicode.com/posts'
);
 expect(req.request.method).toEqual('POST');
 req.flush({ response: 'OK' });
 expect(component.data).toEqual({ response: 'OK' });

 httpMock.verify();
 })
);
```

Započinje se pozivanjem funkcije [makePost\(\)](#) direktno na komponenti. Ovo može da bude čudno jer se ovakvi pozivi obično ne izvode direktno na komponentama. Međutim, ono što se ovde pokušava je izazivanje da [HTTP](#) zahtev bude obavljen, pa je na taj način moguće testirati očekivanja.

Dalje se koristi objektna promenljiva [httpMock](#) i očekuje se da je napravljen jedan zahtev ka [jsonplaceholder](#), primenom funkcije [expectOne\(\)](#) i očekuje se da je zahtev tipa [POST](#).

U liniji `req.flush` šalje se simulirani “*mock*” odgovor za taj HTTP zahtev, a potom se očekuje da component.data odgovara tom odgovoru.

Na kraju, poziva se `httpMock.verify()` da finalizuje preostala očekivanja.

Sada kada je sve jasno kako ovo funkcioniše, dodavanje drugog testa za metodu `DELETE` je veoma jednostavno.

## TESTIRANJE DELETE ZAHTEVA, KREIRANJE TESTA

*Nakon prvog testa, sve je spremno za pisanje i izvođenje testa `DELETE` zahteva.*

Nakon kreiranog prvog testa, sve je spremno za pisanje i izvođenje testa `DELETE` zahteva. Osnova za kreiranje i izvođenje testa je metoda `makeDelete()` čijim pozivom aplikacija izvodi `DELETE` zahtev:

```
makeDelete(): void {
 this.loading = true;
 this.http
 .delete('https://jsonplaceholder.typicode.com/posts/1')
 .subscribe(data => {
 this.data = data;
 this.loading = false;
 });
}
```

Kod za testiranje ove metode ima sledeći oblik:

```
it(
 'performs a DELETE',
 async() => {
 component.makeDelete();

 const req = httpMock.expectOne(
 'https://jsonplaceholder.typicode.com/posts/1'
);

 expect(req.request.method).toEqual('DELETE');
 req.flush({ response: 'OK' });
 expect(component.data).toEqual({ response: 'OK' });

 httpMock.verify();
 }
);
```

Ovde je sve isto, izuzev URL koji je malo promenjen i HTTP metode, koja je sada `RequestMethod.Delete`.

# TESTIRANJE HTTP ZAGLAVLJA, KREIRANJE TESTA

*Poslednja metoda koja se testira, za posmatranu klasu, jeste makeHeaders.*

Poslednja metoda koja se testira, za posmatranu klasu, jeste `makeHeaders()`.

```
makeHeaders(): void {
 const headers: HttpHeaders = new HttpHeaders({
 'X-API-TOKEN': 'IT-255'
 });

 const req = new HttpRequest(
 'GET',
 'https://jsonplaceholder.typicode.com/posts/1',
 {
 headers: headers
 }
);

 this.http.request(req).subscribe(data => {
 this.data = data['body'];
 });
}
```

U ovom slučaju, test bi trebalo da se fokusira da li je token zaglavlja `X-API-TOKEN` podešen ispravno na vrednost IT255:

```
it(
 'sends correct headers',
 async() => {
 component.makeHeaders();

 const req = httpMock.expectOne(
 req =>
 req.headers.has('X-API-TOKEN') &&
 req.headers.get('X-API-TOKEN') == 'IT-255'
);

 req.flush({ response: 'OK' });
 expect(component.data).toEqual({ response: 'OK' });

 httpMock.verify();
)
});
```

Atribut `req.headers` vraća zaglavlj i koriste se dve metode za izvođenje dva različita tvrđenja:

- metoda `has()` proverava da li dato zaglavlj podešeno, ignorišući njegovu vrednost;
- metoda `get()`, koja vraća podešenu vrednost.

Ukoliko je dovoljno da postoji podešeno zaglavlje, biće primenjena metoda `has()`. U suprotnom, ukoliko je potrebno proveriti podešenu vrednost, biće primenjena metoda `get()`.

## POKRETANJE TESTA I PRAĆENJE REZULTATA U ALATU KARMA

*Konačno, u alatu Karma se prati izvšavanje kreiranih testova.*

Konačno, u alatu Karma se prati izvšavanje kreiranih testova. Na komandnoj liniji terminala razvojnog okruženja, kuca se sledeća naredba:

```
ng test --watch
```

U veb pregledaču Google Chrome, pokreće se test alat Karma i prikazuje rezultate izvršavanja kreiranih testova, a to je prikazano sledećom slikom:



Slika 7.1 Pokretanje testa i praćenje rezultata u alatu Karma [izvor: autor]

## ✓ Poglavlje 8

### Individualna vežba 14

#### INDIVIDUALNA VEŽBA (TRAJANJE 90 MINUTA)

*Vežbanje na računaru pisanja testova za Angular aplikacije.*

Proučite i pokušajte da uradite primer sa linka:

<https://medium.com/geekculture/setting-up-a-mock-backend-with-angular-13-applications-26a21788f7da>.

Vežba ima za cilj savladavanje izrade simuliranog serverskog okruženja za testiranje u radnom okviru Angular 13.

## ✓ Poglavlje 9

### Domaći zadatak 14

#### DOMAĆI ZADATAK (PREDVIĐENO VREME 120 MIN)

*Samostalna izrada domaćeg zadatka iz oblasti testiranja Angular aplikacija.*

Osnova za izradu domaćeg zadatka su domaći zadaci iz lekcija, kao i primer sa individualne i pokazne vežbe:

- "L08-Ugrađene directive i rad sa formama"
- "L10 - Http i Angular"

##### **Kreirajte Domaći zadatak 14 na osnovu sledećih zahteva:**

1. Za Domaći zadatak 8 kreirati test slučajeve na sledeći način:

- implementirajte simulirano (mock) okruženje za izvođenje testova;
- kreirajte jedan test slučaj za proveru forme.

2. Za Domaći zadatak 10 kreirati test slučajeve na sledeći način:

- implementirajte simulirano (mock) okruženje za izvođenje testova;
- definišite test slučajeve za servisne metode i HTTP pozive;

**NAPOMENA:** Domaći zadatak dodati kao *GitHub* "commit" na prethodni zadatak, a ne kreirati novi repozitorijum.

## ▼ Poglavlje 10

### Zaključak

#### ZAKLJUČAK

*Lekcija 14 je izučavala alate i mehanizme za testiranje kreiranih veb aplikacija pre produkcije.*

Lekcija je prvo dala osvrt na pojам i značaj procesa testiranja veb aplikacije. Istaknuto je da **testiranje** aplikacije pomaže proces otkrivanja grešaka pre nego što se one i pojave, povećava stepen poverenja u kreiranu veb aplikaciju i olakšava uvođenje novih programera u dalji razvoj aplikacije.

Posebno je diskutovano kako među profesionalcima postoji podeljeno mišljenje o moći i značaju procesa testiranja tokom projekta razvoja nekog softverskog rešenja ali je takođe istaknuto da to nosi i neke prednosti u procesu razvoja savremenih softverskih rešenja - dobra stvar , u vezi sa navedenom problematikom, je da teče neprekidna debata koja, sasvim sigurno, dovodi do sve kvalitetnijih test alata i tehnika, a posledično i do kvalitetnijih softverskih rešenja.

U uvodnom delu lekcije je posebno istaknuto da kroz debatovanje, najčešće se otvaraju pitanja poput sledećih: "Da li je bolje prvo pisati testove, a zatim kreirati implementaciju koja omogućava da kreirani testovi prođu ili je bolje obaviti prvo kodiranje, a tek nakon toga validaciju koda?"

Posebno je diskutovano o potencijalno najboljoj praksi testiranja. Možda će neko od kolega misliti drugačije, za veb aplikacije je baziranje na kreiranju koda pogodnog za testiranje, pogotovo ako je u pozadini bogato iskustvo prototipskog pristupa razvoju aplikacija.

Iako iskustvo može da varira, pokazuje se dok se koristi pristup razvoju aplikacije baziran na prototipu, testiranje pojedinačnih delova koda, koji se verovatno menjaju, može da udvostruči ili utrostruči količinu posla potrebnog da se njihov razvoj nastavi. Suprotno ovome, **fokus je na razvoju aplikacije sa malim komponentama i držanjem velikog broja funkcionalnosti podeljenih u nekoliko metoda koje omogućavaju testiranje funkcionalnosti kao dela šire slike.**

Lekcija je analizu i diskusiju bazirala na sledećim temama:

- *End-to-end ili Unit Testing - izbor?*
- Test alati: *Jasmine, Karma;*
- Pisanje jediničnih testova
- *Angular okvir za jedinično testiranje*
- Podešavanje testiranja
- Testiranje servisa i *HTTP* zahteva

Savladavanjem ove lekcije student je osposobljen sa samostalno piše i izvodi testove *frontend Angular* veb aplikacija.

## LITERATURA

*Za pripremu lekcije korišćena je aktuelna pisana i elektronska literatura.*

**Pisana literatura:**

1. Nate Murray, Felipe Coury, Ari Lerner, Carlos Taborda, ng-Book – The Complete Book on Angular 6, Fullstack.io, 2018
2. Cody Lindley, Frontend Handbook – 2017, Frontend masters, 2017
3. Sandeep Panda, AngularJS – From Novice to Ninja, SitePoint Pty. Ltd, 2014

**Elektronska literatura:**

4. <https://angular.io/>
5. <https://angular.io/tutorial>
6. <https://www.w3schools.com/angular/>
7. <https://www.tutorialspoint.com/angular4/>
8. <https://nodejs.org/en/>
9. <https://code.visualstudio.com/>
10. <https://jasmine.github.io/2.4/introduction.html>
11. <https://alligator.io/angular/testing-async-fakeeasy/>
12. <https://nodejs.org/api/querystring.html>



IT255 - VEB SISTEMI 1

## Razvoj kompletne Angular aplikacije

Lekcija 15

PRIRUČNIK ZA STUDENTE

# IT255 - VEB SISTEMI 1

## Lekcija 15

### ***RAZVOJ KOMPLETNE ANGULAR APLIKACIJE***

- ✓ Razvoj kompletne Angular aplikacije
- ✓ Poglavlje 1: Slučajevi korišćenja
- ✓ Poglavlje 2: Zahtevi sistema
- ✓ Poglavlje 3: Projektovanje i arhitektura sistema
- ✓ Poglavlje 4: Implementacija
- ✓ Poglavlje 5: Demo aplikacije
- ✓ Poglavlje 6: Vežbe 15 - Prezentacije i ocene projekata-135 min
- ✓ Poglavlje 7: Domaći zadatak 15 -
- ✓ Zaključak

Copyright © 2017 – UNIVERZITET METROPOLITAN, Beograd. Sva prava zadržana. Bez prethodne pismene dozvole od strane Univerziteta METROPOLITAN zabranjena je reprodukcija, transfer, distribucija ili memorisanje nekog dela ili čitavih sadržaja ovog dokumenta., kopiranjem, snimanjem, elektronskim putem, skeniranjem ili na bilo koji drugi način.

Copyright © 2017 BELGRADE METROPOLITAN UNIVERSITY. All rights reserved. No part of this publication may be reproduced, stored in a retrieval system or transmitted in any form or by any means, electronic, mechanical, photocopying, recording, scanning or otherwise, without the prior written permission of Belgrade Metropolitan University.

## ❖ Uvod

### UVOD

*Cilj je da se zaokruži izlaganje predmeta primerom celokupnog razvoja veb Angular aplikacije.*

Savladavanjem prethodnog gradiva ove lekcije student je osposobljen sa samostalno razvija frontend aplikacije u *Angular* okviru do nivoa potreba junior Angular programera. Cilj je da se zaokruži izlaganje predmeta primerom celokupnog razvoja veb aplikacije bazirane na razvoju klijentskog dela primenom *Angular* okvira

Lekcija će biti razvijana kao projektna dokumentacija konkretnog veb projekta.

U uvodnom delu je neophodno opisati sistem kao i njegove funkcionalnosti.

**U ovoj lekciji vršimo rezime naučenog iz prethodnih 14 lekcija. Imaćete dva testa za proveru znanja kao uvod u predstojeći ispit. Obnovite prvih 7 lekcija za test 1 i dodatno lekcije 8 - 14 za test 2.**

**Ova lekcija sadrži video materijal. Ukoliko želite da pogledate ovaj video morate da otvorite LAMS lekciju.**

### OPIS I NAMENA SISTEMA

*„Met hotel“ predstavlja informacioni sistem jednog hotela.*

#### **OPIS SISTEMA:**

„Met hotel“ predstavlja informacioni sistem jednog hotela. Ovaj sistem omogućuje korisnicima pregled ponuda hotela kao i dodatnih ponuda.

Sistem pored prijavljivanja i registracije na sistem pruža korisniku i uvid u ponude hotela kao što su: jednokrevetna i dvokrevetna soba, delux i senior apartmani. Takođe sistem korisnicima pruža mogućnost rezervacije smeštaja. Pored toga sistem korisniku nudi i informacije o dodatnim uslugama u vidu parking mesta, bazena, sauna, salon za negu lepote, sale za konferenciju. Takođe sistem pruža korisniku i informacije o samom hotelu kao i stranu vezanu za kontakt i rezervaciju, lokacije.

Što se tiče front-end - a projekat će biti kreiran kroz Angularokvir kao i uz pomoć *Sass(scss)* tehnologije.

Back-end (**koji za studente ovog predmeta nije obavezan i optionalan je**) će biti realizovan putem PHP jezika dok će se konekcija na bazu vršiti putem MYSQL-a.

Kod će biti kodiran u Visual Studio Code razvojnom okruženju.

### NAMENA SISTEMA:

Namena onog sistema je pružanje korisnicima uvid u ponude hotela i mogućnost rezervacije smeštaja bez odlaska direktno u hotel.

## ✓ Poglavlje 1

### Slučajevi korišćenja

#### UČESNICI (AKTERI) SISTEMA

*U ovom delu dokumenta vršiće se analiza učesnika sistema kroz slučajeve korišćenja.*

U ovom delu dokumenta (budući da lekcija opisuje projekat, može da se smatra projektnom dokumentacijom) vršiće se analiza učesnika sistema kroz slučajeve korišćenja.

U ovom sistemu nalaze se dva učesnika:

1. Korisnik
2. Administrator

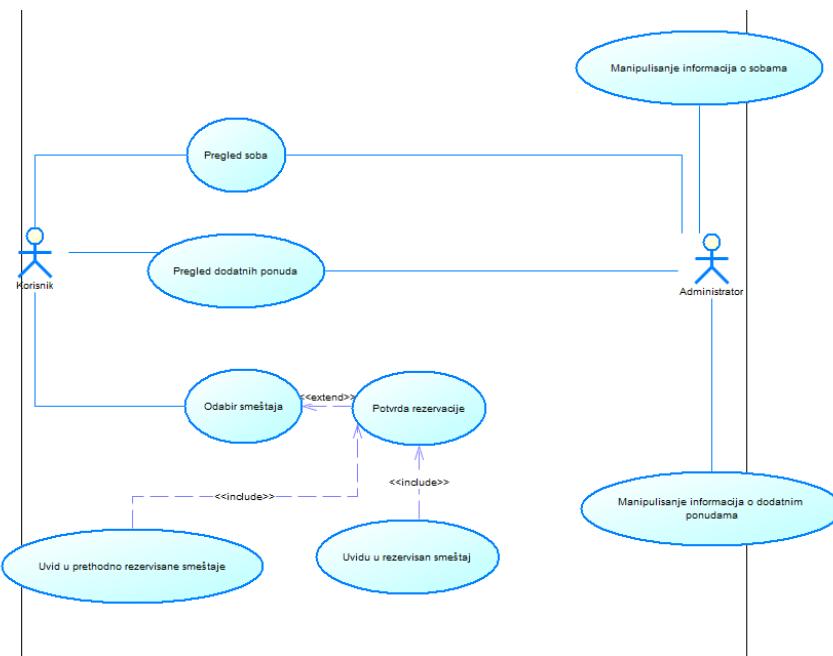
**Korisnik** vrši registraciju na sistem kao i pregled ponuda hotela. Korisnik takođe vrši i odabir smeštaja ukoliko to želi i nakon toga potvrđuje rezervaciju. Pored toga korisnik je u mogućnosti i da pregleda rezervisan smeštaj kao i njegove prethodno rezervisane smeštaje. **Administrator** ima absolutnu kontrolu nad podacima koji se pojavljuju na sistemu, on može da manipuliše podacima kao što su ponude hotela, dodatne ponude u vidu bazena, sauna

**Ova lekcija sadrži video materijal. Ukoliko želite da pogledate ovaj video morate da otvorite LAMS lekciju.**

#### SLUČAJEVI KORIŠĆENJA 1 I 2

*Primenom Use-Case dijagrama možemo videti sve slučajeve korišćenja za oba aktera sistema.*

Na sledećoj slici kroz dijagram (Use-Case) možemo videti sve slučajeve korišćenja za oba aktera sistema.



Slika 1.1 Prikaz dijagrama slučajeva korišćenja [izvor: autor]

Za detaljno opisivanje slučajeva korišćenja neophodno je kreirati i na pregledan način tabelama opisati scenarije upotrebe sistema.

U prvom koraku biće opisan slučaj korišćenja: *Pregled soba* koji je prikazan sledećom tabelom.

Slučaj korišćenja: Pregled soba	
Opis:	Pregled soba iz ponude hotela
Aktor:	Korisnik, Administrator
Preduslov:	1. Pristup sistemu
Tok događaja:	1. Korisnik selektuje željeni tip sobe 2. Korisnik unosi datum dolaska i odlaska
Uslovni tok:	1. Korisnik nema nalog 2. Korisnik ne nalazi sobu
Postuslov:	1. Korisnik uspešno pronađe sobu

Slika 1.2 Prvi slučaj korišćenja [izvor: autor]

U nastavku biće opisan slučaj korišćenja: *Pregled dodatnih ponuda* koji je prikazan sledećom tabelom.

Slučaj korišćenja: Pregled dodatnih ponuda	
Opis:	Pregled dodatnih ponuda hotela u vidu bazena, spa centra...
Aktor:	Korisnik, administrator
Preduslov:	1. Pristup sistemu
Tok događaja:	1. Korisnik selektuje tip dodatne ponude 2. Korisnik vrši uvid u odabrane ponude
Uslovni tok:	1. Korisnik nema nalog 2. Korisnik ne nalazi željenu ponudu
Postuslov:	1. Korisnik uspešno pronađe željenu ponudu

Slika 1.3 Drugi slučaj korišćenja [izvor: autor]

## SLUČAJEVI KORIŠĆENJA 3, 4 I 5

### *Nastavlja se definisanje i opisivanje različitih scenarija korišćenja*

Sistem, pre kodiranja, bi trebalo da bude detaljno opisan i u tom svetlu se nastavlja definisanje i opisivanje različitih scenarija korišćenja.

Kao treći scenario biće opisan slučaj korišćenja: *Odabir smeštaja* koji je prikazan sledećom tabelom.

Slučaj korišćenja: Odabir smeštaja	
Opis:	Odabir smeštaja na osnovu prethodne pretrage
Aktor:	Korisnik
Preduslov:	1. Pristup sistemu 2. Korisnik je ulogovan na sistem 3. Korisnik je pronašao smeštaj
Tok događaja:	1. Korisnik klikne na željeni smeštaj 2. Korisnik klikne na dugme rezerviši
Uslovni tok:	1. Korisnik nema nalog 2. Korisnik ne nalazi željenu sobu
Postuslov:	1. Korisnik uspešno vrši odabir sobe

Slika 1.4 Slučaj korišćenja: Odabir smeštaja [izvor: autor]

U daljem radu kao poseban scenario upotrebe sistema scenario biće opisan slučaj korišćenja: *Potvrda rezervacije* koji je prikazan sledećom tabelom.

Slučaj korišćenja: Potvrda rezervacije	
Opis:	Potvrda rezervacije sobe na osnovu prethodnog odabira
Aktor:	Korisnik
Preduslov:	1. Korinik pristupio sistemu 2. Korisnik ima nalog 3. Korisnik odabrao sobu
Tok događaja:	1. Korisnik unosi lične podatke 2. Korisnik potvrđuje rezervaciju
Uslovni tok:	1. Korisnik nema nalog 2. Korisnik unosi pogrešne podatke
Postuslov:	1. Uspešno rezervisana soba

Slika 1.5 Slučaj korišćenja: Potvrdra rezervacije [izvor: autor]

Neophodno je da sistem omogući uvid u prethodne rezervacije, a to će omogućiti komponenta razvijena iz slučaja korišćenja: *Uvid u prethodnu rezervaciju smeštaja* koji je prikazan sledećom tabelom.

Slučaj korišćenja: Uvid u prethodnu rezervaciju smeštaja	
Opis:	Korisnik vrši uvid u prethodne rezervacije smeštaja
Aktor:	Korisnik
Preduslov:	1. Korisnik pristupio sistemu 2. Korisnik ima nalog 3. Korisnik je imao prethodnih rezervacija
Tok događaja:	1. Korisnik selektuje uvid 2. Korisnik vrši pregled
Uslovni tok:	1. Korisnik nema nalog 2. Korisnik nije imao prethodnih rezervacija
Postuslov:	1. Uspešan prikaz prethodnih rezervacija

Slika 1.6 Slučaj korišćenja: Uvid u prethodnu rezervaciju smeštaja [izvor: autor]

## SLUČAJEVI KORIŠĆENJA 6, 7 I 8

*Neophodno je detaljno i precizno opisati sve slučajeve upotrebe sistema.*

Za zaokruživanje izlaganja o mogućim scenarijima upotrebe aplikacije, neophodno je razraditi još sledeća tri scenarija:

- *Uvid u trenutnu rezervaciju smeštaja (slika 7);*
- *Manipulisanje informacija o sobama od strane administratora (slika 8);*
- *Manipulisanje informacija o dodatnim uslugama (slika 9); .*

Slučaj korišćenja: Uvid u trenutnu rezervaciju	
Opis:	Korisnik vrši uvid u trenutnu rezervaciju smeštaja
Aktor:	Korisnik
Preduslov:	1. Korisnik pristupio sistemu 2. Korisnik ima nalog 3. Korisnik ima trenutnu rezervaciju
Tok događaja:	1. Korisnik selektuje uvid 2. Korisnik vrši pregled
Uslovni tok:	1. Korisnik nema nalog 2. Korisnik nema trenutnu rezervaciju
Postuslov:	1. Uspešan prikaz trenutne rezervacije

Slika 1.7 Uvid u trenutnu rezervaciju smeštaja [izvor: autor]

Slučaj korišćenja: Manipulisanje informacija o sobama	
Opis:	Manipulisanje informacija o sobama od strane administratora
Aktor:	Administrator
Preduslov:	<ol style="list-style-type: none"><li>1. Administrator je pristupio sistemu</li><li>2. Administrator je ulogovan na sistem</li></ol>
Tok događaja:	<ol style="list-style-type: none"><li>1. Administrator dodaje nove sobe</li><li>2. Administrator menja podatke o postojećim sobama</li><li>3. Administrator briše neke od postojećih soba</li></ol>
Uslovni tok:	<ol style="list-style-type: none"><li>1. Ne postoji soba za manipulisanje</li></ol>
Postuslov:	<ol style="list-style-type: none"><li>1. Uspešno dodata soba</li><li>2. Uspešno promenjen podatak o sobi</li><li>3. Uspešno obrisana soba</li></ol>

Slika 1.8 Manipulisanje informacija o sobama [izvor: autor]

Slučaj korišćenja: Manipulisanje informacija o dodatnim uslugama	
Opis:	Administrator manipuliše informacija o dodatnim uslugama
Aktor:	Administrator
Preduslov:	<ol style="list-style-type: none"><li>1. Administrator je na sistemu</li><li>2. Administrator je ulogovan na sistem</li></ol>
Tok događaja:	<ol style="list-style-type: none"><li>1. Administrator dodaje novu dodatnu uslugu</li><li>2. Administrator menja informacije o već postojećoj dodatnoj usluzi</li><li>3. Administrator briše neku od postojećih dodatnih usluga</li></ol>
Uslovni tok:	<ol style="list-style-type: none"><li>1. Ne postoji dodatna usluga za manipulaciju</li></ol>
Postuslov:	<ol style="list-style-type: none"><li>1. Uspešno dodana nova dodatna usluga</li><li>2. Uspešno promenja informacija o već postojećoj dodatnoj usluzi</li><li>3. Uspešno obrisana postojeća dodatna usluga</li></ol>

Slika 1.9 Manipulisanje informacija o dodatnim uslugama [izvor: autor]

## ▼ Poglavlje 2

# Zahtevi sistema

## FUNKCIONALNI ZAHTEVI SISTEMA

*Funkcionalni zahtevi opisuju kako sistem reaguje na neke podsticaje ili kako se ponaša u pojedinim situacijama.*

Softverski zahtevi se u osnovi dele na:

- Funkcionalne zahteve (*Functional requirements*) - zahtevi opisuju kako sistem reaguje na neke podsticaje ili kako se ponaša u pojedinim situacijama.
- Nefunkcionalne zahteve (*Nonfunctional requirements*) - zahtevi koji se najčešće odnose na neka ograničenja funkcionalnosti i servisa koje pruža sistem.

Funkcionalni zahtevi sistema koji se razvija u okviru ovog projekta mogu biti prikazani sledećom slikom, a identifikovani su kao:

- *Pretraga soba hotela;*
- *Pretraga dodatnih usluga hotela;*
- *Rezervacija smeštaja;*
- *Uvid u informacije u hotelu;*
- *Uvid u trenutnu rezervaciju;*
- *Uvid u prethodne rezervacije;*
- **Administrator** - *Manipulisanje informacijama o sobama;*
- **Administrator** - *Manipulisanje informacijama o dodatnoj ponudi.*

	Title ID	Full Description	Code	Priority	Workload	Risk	Status
1	1.	<b>Pretraga soba hotela</b> <ul style="list-style-type: none"><li>• Sistem omogućuje korisniku da vrši pretragu soba hotela, bilo po tipu sobe ili po datumu dolaska.</li></ul>	REQ_0001	Undefined		Undefined	Draft
2	2.	<b>Pretraga dodatnih usluga hotela</b> <ul style="list-style-type: none"><li>• Sistem omogućuje korisniku da vrši pretragu dodatnih usluga hotela, u vidu bazena, spa, restorana, parking garaža.</li></ul>	REQ_0002	Undefined		Undefined	Draft
3	3.	<b>Rezervacija smeštaja</b> <ul style="list-style-type: none"><li>• Sistem omogućuje korisniku da vrši rezervaciju smeštaja na osnovu prethodno izvršene pretrage</li></ul>	REQ_0003	Undefined		Undefined	Draft
4	4.	<b>Uvid u informacije u hotelu</b> <ul style="list-style-type: none"><li>• Sistem omogućuje korisniku uvid u informacije o hotelu, u vidu informacija o znamenih hotelima, sobama, usluzama, osobama hotelata.</li></ul>	REQ_0005	Undefined		Undefined	Draft
5	5.	<b>Uvid u trenutnu rezervaciju</b> <ul style="list-style-type: none"><li>• Sistem omogućuje korisniku uvid u rezervaciju smestaja koji je u toku</li></ul>	REQ_0006	Undefined		Undefined	Draft
6	6.	<b>Uvid u prethodne rezervacije</b> <ul style="list-style-type: none"><li>• Sistem omogućuje korisniku uvid u prethodne rezervacije</li></ul>	REQ_0007	Undefined		Undefined	Draft
7	7.	<b>Administrator - Manipulisanje informacijama o sobama</b> <ul style="list-style-type: none"><li>• Sistem omogućuje administratoru da vrši manipulaciju na podacima o sobama</li><li>• Sistem omogućuje administratoru da dodaje nove sobe</li><li>• Sistem omogućuje administratoru da menja podatke o sobama</li><li>• Sistem omogućuje administratoru da briše sobu</li></ul>	REQ_0008	Undefined		Undefined	Draft
8	8.	<b>Administrator - Manipulisanje informacijama o dodatnoj ponudi</b> <ul style="list-style-type: none"><li>• Sistem omogućuje administratoru da vrši manipulaciju na podacima o dodatnim uslugama</li><li>• Sistem omogućuje administratoru da dodaje nove dodatne usluge</li><li>• Sistem omogućuje administratoru da menja podatke o dodatnim uslugama</li><li>• Sistem omogućuje administratoru da briše neke od dodatnih usluga</li></ul>	REQ_0009	Undefined		Undefined	Draft

Slika 2.1 Prikaz funkcionalnih zahtevi [izvor: autor]

## NEFUNKCIONALNI ZAHTEVI SISTEMA

*Funkcionalni zahtevi se odnose na neka ograničenja funkcionalnosti i servisa koje pruža sistem.*

Kao što je istaknuto u prethodnom izlaganju, **nefunkcionalni zahtevi** se najčešće odnose na neka ograničenja funkcionalnosti i servisa koje pruža sistem

Ova ograničenja su najčešće nametnuta standardima koji važe u domenu problema, ali mogu biti i vremenska ograničenja ili ograničenja pojedinih procesa. Nefunkcionalni zahtevi se obično odnose na sistem kao celinu.

Nefunkcionalni zahtevi sistema koji se razvija u okviru ovog projekta mogu biti prikazani sledećom slikom, , a identifikovani su kao:

- *performanse;*
- *bezbednost;*
- *sigurnost;*
- *pouzdanost;*
- *korisnički interfejs.*

	Title ID	Full Description	Code	Priority	Workload	Risk	Status
1	1.	<b>Performanse</b> <ul style="list-style-type: none"> <li>• Sistem treba da omogući brzo i precizno pretraživanje podataka.</li> <li>• Sistem treba da omogući predrad i preprezervativ, precizije rečeno da omogući brzo rešenje polikliničkih klijenata poslovnih pronosa.</li> </ul>	REQ_0001	Undefined		Undefined	Draft
2	2.	<b>Bezbednost</b> <ul style="list-style-type: none"> <li>• U okviru sistema biće implementirana ograničenja i procedure za validaciju uneseni podatki.</li> </ul>	REQ_0002	Undefined		Undefined	Draft
3	3.	<b>Sigurnost</b> <ul style="list-style-type: none"> <li>• Sistem ne dozvoljava pristup podacima bilo kojim korisnicima, već samo ovlašćenim.</li> </ul>	REQ_0003	Undefined		Undefined	Draft
4	4.	<b>Pouzdanost</b> <ul style="list-style-type: none"> <li>• Sistem treba da radim konstantno bez prekida, jer dno u slučaju pošledica greške u radu operativnog sistema komunikacija sa internet konsekvenčno.</li> </ul>	REQ_0004	Undefined		Undefined	Draft
-	5.	<b>Korisnički interfejs</b> <ul style="list-style-type: none"> <li>• Korisnički interfejs je tako dizajniran da korisnik bude što jednostavniji za upotrebu.</li> </ul>	REQ_0005	Undefined		Undefined	Draft

Slika 2.2 Nefunkcionalni zahtevi sistema [izvor: autor]

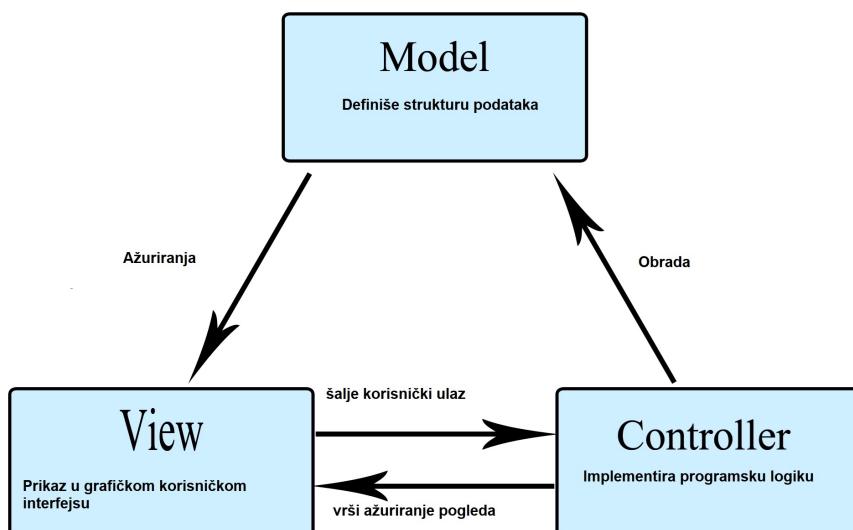
## ✓ Poglavlje 3

# Projektovanje i arhitektura sistema

## MVC ARHITEKTURA

*Ova faza predstavlja definisanje potrebne arhitekture sistema.*

Ova faza predstavlja definisanje potrebne arhitekture sistema kao i kreiranje baze podataka (opcionalno za ovaj predmet). Sledećom slikom je prikazan Model-View-Controller (MVC) šablon na kojem će se bazirati izrada projekta.



Slika 3.1 MVC arhitektura [izvor: autor]

MVC arhitektura predstavlja softversku arhitekturu, koja je najčešće korišćenja uz implementaciju korisničkog interfejsa, pa samim tim i najpogodnija je za arhitekturu web aplikacija. U glavnom ova arhitektura razdvaja logiku aplikacije u 3 odvojena dela, što dovodi do modularnosti i bolje komunikacije među modulima, samim tim i aplikacija je fleksibilnija.

- **Model** – definiše šta bi aplikacija trebala da sadrži. Ukoliko se stanje nekog od podataka promeni, onda je na modelu da obavesti pogled (**View**) i ponekad kontroler (**Controller**) o nastaloj promeni.
- **View** – definiše kako bi aplikacija trebalo da bude prikazana;
- **Controller** – sadrži logiku koju šalje **Model**-u ili **View**-u kao odziv na korisnikovu akciju.

**Model-View-Controller (MVC)** je arhitekturni šablon koji se koristi u razvoju softvera. U složenim aplikacijama koje prikazuju korisniku ogromne količine podataka programeri često žele da razdvoje kod koji se bavi podacima od onog koji se bavi interfejsom, tako da razvoj oba postane lakši i jednostavniji.

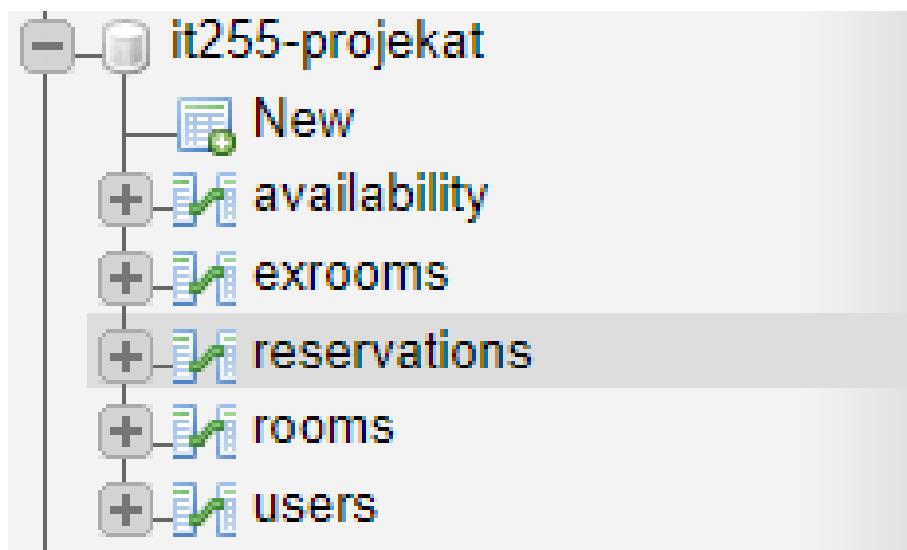
MVC rešava ovaj problem razdvajanjem podataka i biznis logike od njihovog prikaza i interakcije sa korisnikom, uz to uvodeći i komponentu zaduženu za koordinisanje prve dve.

## BAZA PODATAKA (OPCIONALNO ZA OVAJ PREDMET) - LISTA TABELA

*Nije obavezno korišćenje baze podataka na ovom nivou veb programiranja*

Budući da je veb izučavanje podeljeno jasno po predmetima: **frontend (IT255 - Veb sistemi 1)** i **backend (IT355 - Veb sistemi 2)**, nije obavezno korišćenje baze podataka na ovom nivou veb programiranja. Ali, sa bazom podataka sve dobija drugačiju i ozbiljniju dimenziju.

Sledećom slikom je prikazana lista tabela koje se nalaze u bazi podataka projekta.



Slika 3.2 Šema baze podataka projekta [izvor: autor]

Na osnovu slike moguće je utvrditi da baza podataka za ovaj sistem sadrži sledeće tabele:

- *Availability* – ona definiše koliko ima slobodnih mesta za svaku sobu;
- *Exrooms* – ona predstavlja tabelu koja sadrži informacije o korisnikovim prethodnim rezervacijama;
- *Reservations* – sadrži podatke o trenutnim rezervacijama korisnika;

- *Rooms* – sadrži podatke o sobama iz ponude hotela;
- *Users* – sadrži informacije o korisniku sistema.

## PRIKAZ TABELA BAZE PODATAKA

*Baza podataka projekta sadrži veći broj tabela koje je potrebno vizuelno reprezentovati.*

Kao što je prikazano prethodnom slikom baza podataka projekta sadrži veći broj tabela. Sledi njihova grafička reprezentacija.

U prvom koraku sledi prikaz tabele *Availability*:

#	Name	Type	Collation	Attributes	Null	Default	Comments	Extra	Action
1	availability_id	int(11)			No	None		AUTO_INCREMENT	Primary  Index  Fulltext  Add to central columns
2	room_id	int(11)			No	None		Primary  Index  Fulltext  Add to central columns	
3	available	int(11)			No	None		Primary  Index  Fulltext  Add to central columns	

Slika 3.3 Prikaz tabele Availability [izvor: autor]

Kao što je moguće primeniti tabela *Availability* ima 3 atributa. Prvi atributi je *ID* on ujedno i predstavlja primarni ključ ove tabele. Pored njega to je i *ID* sobe na koji se odnosi kao i atribut *available* koji označava broj slobodnih soba.

U nastavku sledi prikaz tabele *Exrooms*:

#	Name	Type	Collation	Attributes	Null	Default	Comments	Extra	Action
1	ex_id	int(11)			No	None		AUTO_INCREMENT	Primary  Index  Fulltext  Add to central columns
2	room_id	int(11)			No	None		Primary  Index  Fulltext  Add to central columns	
3	user_id	int(11)			No	None		Primary  Index  Fulltext  Add to central columns	
4	date_from	date			No	None		Primary  Index  Fulltext  Add to central columns	
5	date_to	date			No	None		Primary  Index  Fulltext  Add to central columns	
6	then_cost	double			No	None		Primary  Index  Fulltext  Add to central columns	

Slika 3.4 Prikaz tabele Exrooms [izvor: autor]

Na osnovu slike 4 moguće je videti da tabela sadrži 6 atributa. *ID* koji predstavlja primarni ključ tabele, *room* i *user ID* koji predstavljaju sobu i korisnika, datume dolaska i odlaska kao i cenu koja je tada bila aktuelna.

Baza se dalje izlaže prikazom tabele *Reservations*:

#	Name	Type	Collation	Attributes	Null	Default	Comments	Extra	Action
1	res_id	int(11)			No	None		AUTO_INCREMENT	Primary  Index  Fulltext  Add to central columns
2	room_id	int(11)			No	None		Primary  Index  Fulltext  Add to central columns	
3	user_id	int(11)			No	None		Primary  Index  Fulltext  Add to central columns	
4	date_from	date			No	None		Primary  Index  Fulltext  Add to central columns	
5	date_to	date			No	None		Primary  Index  Fulltext  Add to central columns	

Slika 3.5 Prikaz tabele Reservations [izvor: autor]

Sa slike je moguće primetiti sa ova tabela ima 5 atributa. *ID* koji predstavlja primarni ključ tabele, *room id* i *user id* koji predstavljaju korisnika i sobu koja je rezervisana i datume od kad do kad je soba rezervisana.

U nastavku sledi prikaz tabele *Rooms*:

#	Name	Type	Collation	Attributes	Null	Default	Comments	Extra	Action
1	room_id	int(11)			No	None		AUTO_INCREMENT	Primary  Index  Fulltext  Add to central columns
2	room_name	varchar(30)	latin1_swedish_ci		No	None		Primary  Index  Fulltext  Add to central columns	
3	room_desc	text	latin1_swedish_ci		No	None		Primary  Index  Fulltext  Add to central columns	
4	room_desc	varchar(255)	latin1_swedish_ci		No	None		Primary  Index  Fulltext  Add to central columns	
5	price	double			No	None		Primary  Index  Fulltext  Add to central columns	
6	night	varchar(20)	latin1_swedish_ci		No	None		Primary  Index  Fulltext  Add to central columns	

Slika 3.6 Prikaz tabele Rooms [izvor: autor]

Ova tabela sadrži 6 atributa. *ID* koji predstavlja primarni ključ, ime sobe, sliku sobe, deskripciju sobe, cenu sobe, i dodatni opis sobe.

#	Name	Type	Collation	Attributes	Null	Default	Comments	Extra	Action
1	<i>id</i>	int(11)		No	None	AUTO_INCREMENT			Primary  Index  Fulltext  Add to central columns
2	<i>name</i>	varchar(20)	latin1_swedish_ci	No	None				Primary  Index  Fulltext  Add to central columns
3	<i>last_name</i>	varchar(20)	latin1_swedish_ci	No	None				Primary  Index  Fulltext  Add to central columns
4	<i>username</i>	varchar(10)	latin1_swedish_ci	No	None				Primary  Index  Fulltext  Add to central columns
5	<i>password</i>	varchar(15)	latin1_swedish_ci	No	None				Primary  Index  Fulltext  Add to central columns
6	<i>email</i>	varchar(30)	latin1_swedish_ci	No	None				Primary  Index  Fulltext  Add to central columns
7	<i>privileges</i>	int(11)		No	None				Primary  Index  Fulltext  Add to central columns
8	<i>token</i>	text	latin1_swedish_ci	No	None				Primary  Index  Fulltext  Add to central columns

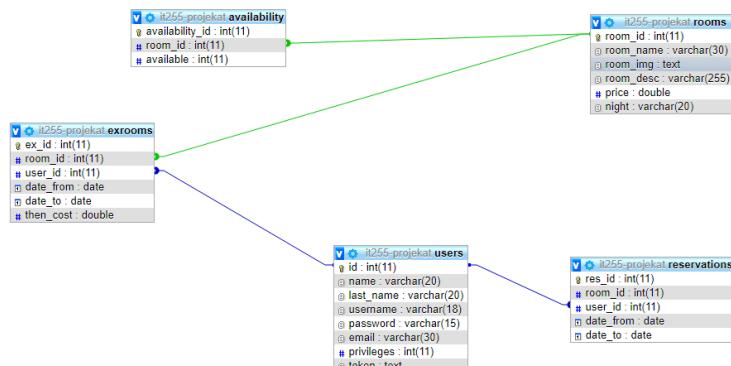
Slika 3.7 Prikaz tabele Users [izvor: autor]

Konačno, poslednja tabela *Users* sadrži 8 atributa i služi za čuvanje informacija o korisnicima. *ID* predstavlja primarni ključ tabele, ime i prezime korisnika, *username*, *password*, *email*, privilegije i token.

## PRIKAZ VEZA IZMEĐU TABELA BAZE PODATAKA

*Za konačan opis baze podataka, neophodno je priložiti prikaz veza između tabela.*

Za konačan opis baze podataka, neophodno je priložiti prikaz veza između tabela. Na osnovu ovoga moguće je videti koji je primarni ključ neke tabele spušten kao strani ključ u neku drugu tabelu.



Slika 3.8 Prikaz veza između tabela [izvor: autor]

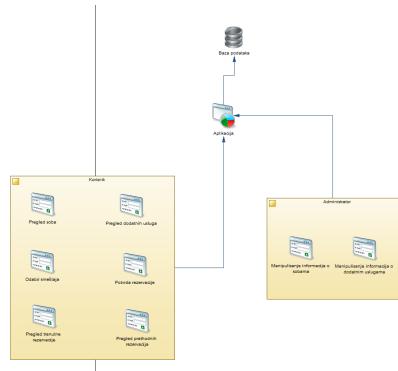
Nakon detaljnog opisa baze sistema projekta, moguć je preseliti pažnju na njegovu arhitekturu.

## ARHITEKTURA SISTEMA

*Nakon detaljnog opisa baze sistema projekta, moguć je preseliti pažnju na njegovu arhitekturu.*

Kao što je istaknuto u prethodnom izlaganju, nakon detaljnog opisa baze sistema projekta, moguće je preseliti pažnju na njegovu arhitekturu.

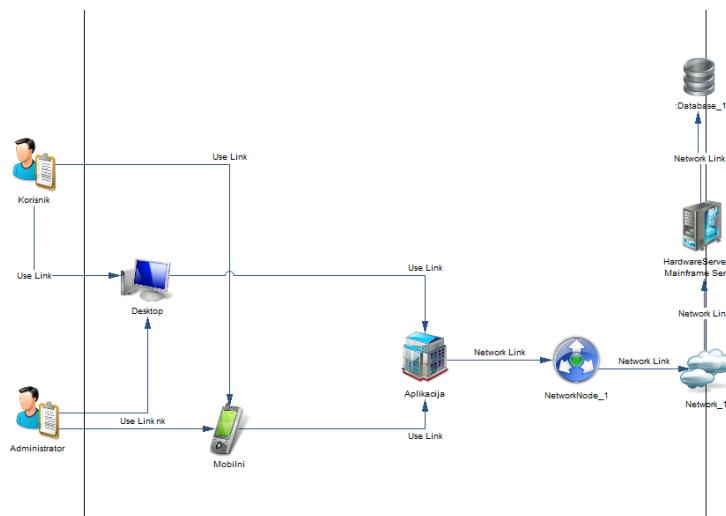
U prvom koraku biće ilustrovan **prikaz aplikacionog dijagrama arhitekture**. To je učinjeno sledećom slikom.



Slika 3.9 Prikaz aplikacionog dijagrama arhitekture [izvor: autor]

Na slici 9 je moguće primetiti da postojeći dva učesnika u sistemu koji imaju mogućnost pristupa ka navedenim funkcionalnostima kao i to da aplikacija komunicira sa bazom podataka (**MySQL DBMS**). Što se tiče funkcionalnosti korisnika moguće je videti da on može vršiti: pregled soba, pregled dodatnih usluga, odabir smeštaja, potvrdu rezervacije, pregled trenutnih rezervacija, pregled prethodnih rezervacija. Dok administrator ima sledeće funkcionalnosti: pored pregleda soba i dodatnih usluga on još ima i manipulisanje informacijama o sobama i manipulisanje informacijama o dodatnim uslugama..

U nastavku biće ilustrovan **prikaz infrastrukturnog dijagrama sistema**. To je učinjeno sledećom slikom.



Slika 3.10 Prikaz infrastrukturnog dijagrama sistema [izvor: autor]

Sa dijagrama na slici 10 moguće je primetiti kako korisnici pristupaju aplikaciji putem mreže. Dalje, moguće je uočiti da postoje dva učesnika: korisnik i administrator. Oni putem radnog okruženja (mobilni uređaj ili desktop uređaj) se konektuju na aplikaciju, zatim se aplikacija povezuje na mrežni čvor nakon čega se on konektuje na internet. Zatim putem interneta se aplikacija konektuje na server gde se on na kraju povezuje sa bazom podataka.

## ▼ Poglavlje 4

# Implementacija

## KOMPONENTE I DATOTEKE PROJEKTA

*Sledi implementacija softverkog rešenja iz predloženog modela*

Nakon opisa zahteva i demonstracije predložene arhitekture sistema, sledi njegova implementacija iz njegovog predloženog modela. Projekat je realizovan kroz:

- Microsoft Visual Code razvojno okruženje.
- Deo front-end - a je urađen u Angular okviru uz pomoć Bootstrap tehnologije.
- Što se tiče serverske strane korišćen je PHP pri realizaciji servisa vezanih za bazu podataka.

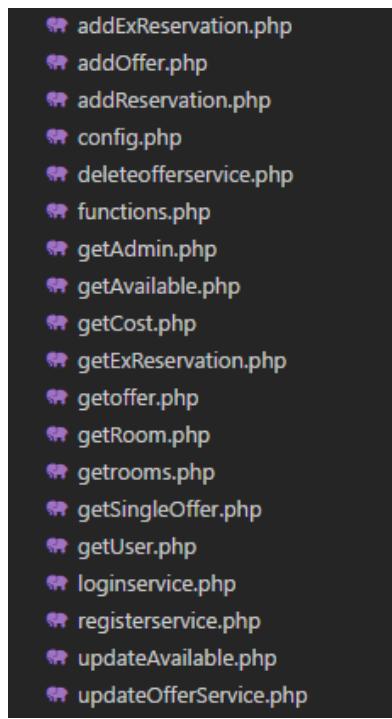
Serverski deo je opcionalan za ovaj predmet i ovde je prikazan zbog sagledavanja kompletno urađenog veb projekta.

## DATOTEKE SERVISA

*Deo koji studenti mogu da izostave tokom izrade njihovih projekata, jer ih čeka u predmetu IT355.*

Upravo će taj deo, koji studenti mogu da izostave tokom izrade njihovih projekata, jer ih čeka u predmetu **IT355 Veb sistemi 1**, biti demonstriran na prvom mestu.

Sledećom slikom su prikazani servisi aplikacije.



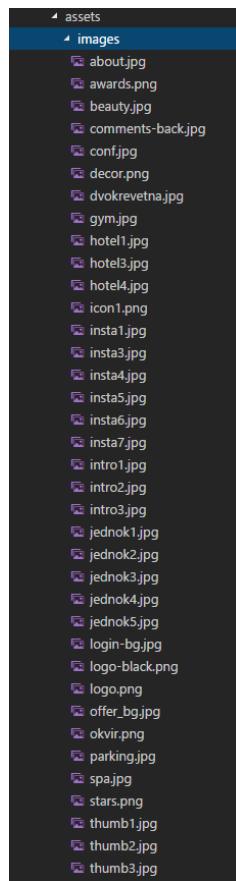
Slika 4.1.1 Prikaz servisa [izvor: autor]

Na ovoj slici moguće je videti predstavljene sve servise korišćene na serverskoj strani koje komuniciraju sa bazom podataka i na osnovu **POST** i **GET** zahteva vraćaju aplikaciji (u front-end) određene podatke. Sve validacije su obavljene kroz ove servise.

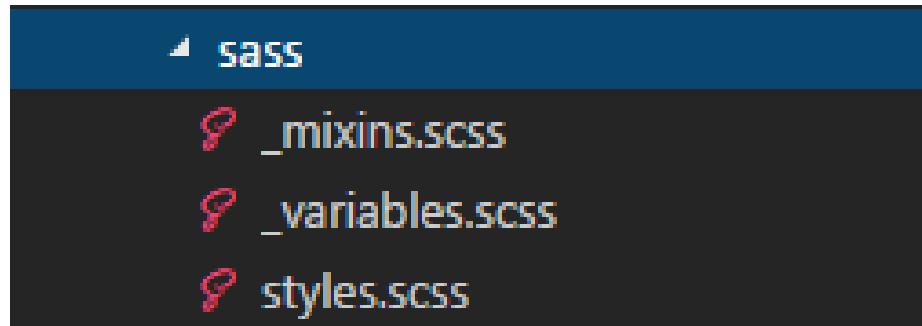
## DATOTEKE SLIKA I STILOVA

*Sledi prikaz datoteka slika i SCSS datoteka korišćenih u projektu.*

Prikaz „*Assets*“ foldera sa slikama je prikazan sledećom slikom.



Slika 4.1.2 Slike korišćene u projektu [izvor: autor]



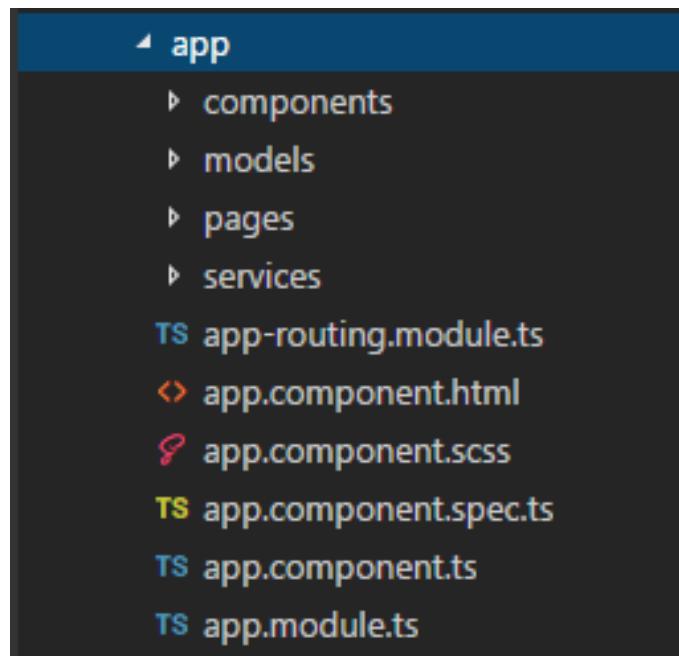
Slika 4.1.3 Datoteke stilova korišćene u projektu [izvor: autor]

Tehnologija koja je korišćena za stilizaciju stranica je **SASS**, pa samim tim u ovom folderu se nalazi glavni **SASS** fajl uz „**\_mixins**“ i „**\_variables**“ fajlove koji sadrže pomenute elemente.

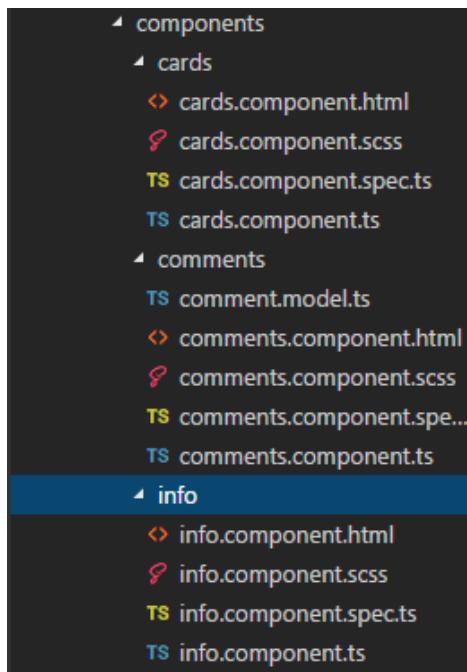
## APP FOLDER I KOMPONENTE

*Sledi pregled datoteka Angular klijentske strane aplikacije.*

Sledi pregled datoteka Angular klijentske strane aplikacije. Sledećom slikom je prikazan sadržaj glavne komponente sistema.



Slika 4.1.4 Sadržaj App foldera [izvor: autor]



Slika 4.1.5 Komponente Angular aplikacije [izvor: autor]

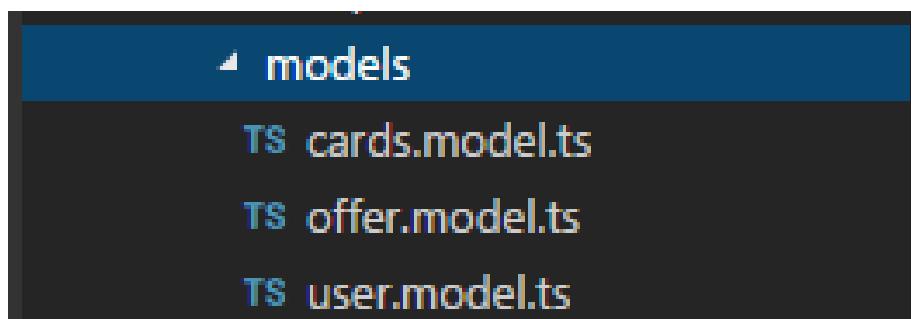
Što se tiče ove slike, ovde su prikazane komponente koje se pojavljuje na više mesta u stranici, te da se ne bi svaka komponente pri potrebnoj promeni menjala, ponaosob, kreirani su kao odvojene komponente i na kraju pozvani u aplikaciji. Pa ukoliko dođe do promena ona će se vršiti samo na jedom mestu.

## MODELI, FRONTEND SERVISI I STRANICE

*Modeli, frontend servisi i stranice su datoteke kojima se završava definicija aplikacije.*

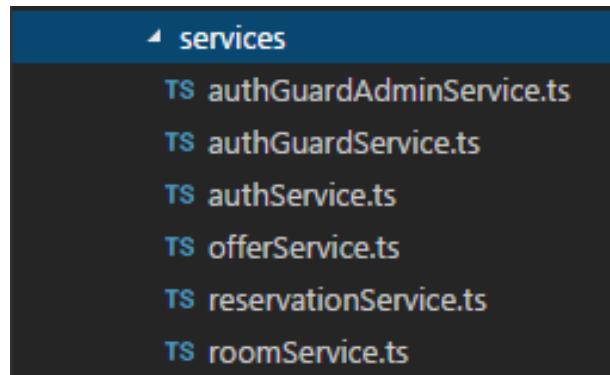
Modeli, frontend servisi i stranice su datoteke kojima se završava definicija aplikacije.

Sledećom slikom je prikazana lokacija kreiranih modelskih klasa projekta.



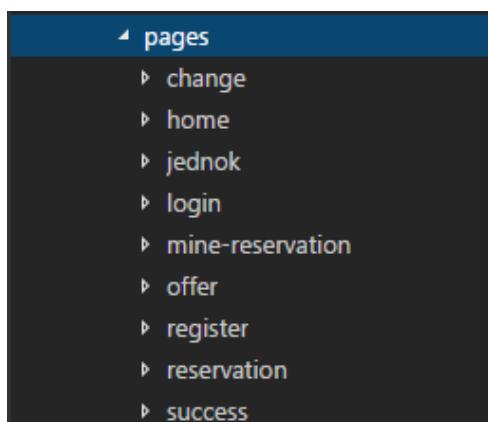
Slika 4.1.6 Modelske klase Angular projekta [izvor: autor]

Sledećom slikom je dat prikaz lokacije frontend servisa koji su korišćeni kroz ovaj projekat.



Slika 4.1.7 Lokacija Angular frontend servisa [izvor: autor]

U folderu predstavljenom na slici 8 nalaze se sve stranice kreirane za ovu aplikaciju.





Slika 4.1.8 Prikaz „Pages“ foldera [izvor: autor]

## 4.1 Implementacija - listinzi

### PHP SERVISI

*Sledi listing beckend servisa korišćenih u aplikaciji.*

Sledi listing **beckend** servisa korišćenih u aplikaciji. Još jednom je potrebno istaći da je ovaj deo projekta priložen zbog prikazivanja kompletne veb aplikacije sa klijent i serverskom stranom. **U ovom trenutku je ovo opcionalan deo projekta, nije obavezan u ovom predmetu**, jer se razvoj serverske strane veb aplikacije izučava na predmetima *IT355 - Veb sistemi 2* i *CS230 - Distribuirani sistemi*.

Za početak je obezbeđen prikaz „*config.php*“ fajla. Celokupan kod se nalazi unutar taga *<?php ?>*:

```
$servername = "localhost";
$username = "root";
$password = "";
$db = "IT255-projekat";

$conn = new mysqli($servername, $username, $password, $db);
```

Iz listinga je moguće primetiti da je predstavljeni fajl namenjen za povezivanje sa bazom podataka.

Sledećim listingom je predstavljen je jedan servis korišćen u ovom projektu. Ovaj servis služi za dodavanje kreirane rezervacije u bazu podataka. Ovaj servis poziva funkcije koje se nalazi u „*functions.php*“ fajlu. Celokupan kod se nalazi unutar taga *<?php ?>*:

```
include("config.php");
header("Access-Control-Allow-Origin: *");
header('Access-Control-Allow-Methods: POST, GET');
header("Access-Control-Allow-Headers: *");
header('Content-Type: application/json');

if ($_SERVER['REQUEST_METHOD'] === 'OPTIONS') {
 die();
}

function decodeJSON(){
 return json_decode(file_get_contents('php://input'), true);
}

function checkIfLoggedIn(){
 global $conn;
 if(isset($_SERVER['HTTP_TOKEN'])){
 $token = $_SERVER['HTTP_TOKEN'];
```

```
$result = $conn->prepare("SELECT * FROM users WHERE token=?");
$result->bind_param("s",$token);
$result->execute();
$result->store_result();
$num_rows = $result->num_rows;
if($num_rows > 0)
{
 return true;
}
else{
 return false;
}
else{
 return false;
}

function login($username, $password){
 global $conn;
 $rarray = array();
 $errors = "";
 if($username=="null"){
 $errors .= "Username is empty\r\n";
 }
 if($password=="null"){
 $errors .= "Password is empty\r\n";
 }
 if(checkLogin($username,$password)){
 $id = sha1(uniqid());
 $result2 = $conn->prepare("UPDATE users SET token=? WHERE username=?");
 $result2->bind_param("ss",$id,$username);
 $result2->execute();
 $rarray['token'] = $id;
 $rarray['admin'] = checkAdmin1($id);
 } else{
 header('HTTP/1.1 401 Unauthorized');
 $errors .= "Invalid username or password\r\n";
 $rarray['error'] = json_encode($errors);
 }
 return json_encode($rarray);
}

function checkLogin($username, $password){
 global $conn;
 $password = md5($password);
 $result = $conn->prepare("SELECT * FROM users WHERE username=? AND password=?");
 $result->bind_param("ss",$username,$password);
 $result->execute();
 $result->store_result();
 $num_rows = $result->num_rows;
 if($num_rows > 0)
 {
```

```
 return true;
 }
 else{
 return false;
 }
}

function checkAdmin1($id){
 global $conn;
 $result = $conn->prepare("SELECT * FROM users WHERE token=?");
 return $conn->query("SELECT privileges FROM users WHERE token =
'$id'")->fetch_object()->privileges;
}

function checkAdmin($id){
 global $conn;
 $rarray = array();
 $result = $conn->query("SELECT * from users where token='$id'");
 $num_rows = $result->num_rows;
 $user = array();
 if($num_rows > 0)
 {
 $result2 = $conn->query("SELECT * from users where token='$id'");
 while($row = $result2->fetch_assoc()) {
 $one_user = array();
 $one_user['privileges'] = $row['privileges'];

 array_push($user,$one_user);
 }
 }
 $rarray['user'] = $user;
 return json_encode($rarray);
}

function register($username, $password, $firstname, $lastname,$privileges,$email){
 global $conn;
 $rarray = array();
 $errors = "";
 if(checkIfUserExists($username)){
 $errors .= "Username već postoji\r\n";
 }
 if(checkIfEmailExists($email)){
 $errors .= "Email već postoji\r\n";
 }
 if(strlen($username) < 5){
 $errors .= "Username mora imati najmanje 5 karaktera\r\n";
 }
 if(strlen($password) < 5){
 $errors .= "Šifra mora imati najmanje 5 karaktera\r\n";
 }
 if(strlen($firstname) < 3){
 $errors .= "Ime mora imati najmanje 3 karaktera\r\n";
 }
}
```

```
if(strlen($lastname) < 3){
 $errors .= "Prezime mora imati najmanje 3 karaktera\r\n";
}
if($errors == ""){
 $stmt = $conn->prepare("INSERT INTO users (name, last_name, username,
password,privileges,email) VALUES (?, ?, ?, ?, ?,?,?)");
 $pass =md5($password);
 $stmt->bind_param("sssis", $firstname, $lastname, $username,
$pass,$privileges,$email);
 if($stmt->execute()){
 $id = sha1(uniqid());
 $result2 = $conn->prepare("UPDATE users SET token=? WHERE username=?");
 $result2->bind_param("ss",$id,$username);
 $result2->execute();
 $rarray['token'] = $id;
 }else{
 header('HTTP/1.1 400 Bad request');
 $rarray['error'] = "Database connection error";
 }
} else{
 header('HTTP/1.1 400 Bad request');
 $rarray['error'] = json_encode($errors);
}

return json_encode($rarray);
}

function checkIfUserExists($username){
 global $conn;
 $result = $conn->prepare("SELECT * FROM users WHERE username=?");
 $result->bind_param("s",$username);
 $result->execute();
 $result->store_result();
 $num_rows = $result->num_rows;
 if($num_rows > 0)
 {
 return true;
 }
 else{
 return false;
 }
}

function checkIfEmailExists($email){
 global $conn;
 $result = $conn->prepare("SELECT * FROM users WHERE email=?");
 $result->bind_param("s",$email);
 $result->execute();
 $result->store_result();
 $num_rows = $result->num_rows;
 if($num_rows > 0)
 {
 return true;
 }
}
```

```

 }
 else{
 return false;
 }
}

function getRooms(){
 global $conn;
 $rarray = array();
 $result = $conn->query("SELECT room_name, room_img, room_desc, price, night
FROM rooms");
 $num_rows = $result->num_rows;
 $rooms = array();
 if($num_rows > 0)
 {
 $result2 = $conn->query("SELECT room_name, room_img, room_desc, price,
night,(SELECT available FROM availability WHERE room_id=rooms.room_id) as ava FROM
rooms");
 while($row = $result2->fetch_assoc()) {
 $one_room = array();
 $one_room['room_name'] = $row['room_name'];
 $one_room['room_img'] = $row['room_img'];
 $one_room['room_desc'] = $row['room_desc'];
 $one_room['price'] = $row['price'];
 $one_room['night'] = $row['night'];
 $one_room['ava'] = $row['ava'];
 array_push($rooms,$one_room);
 }
 }
 $rarray['rooms'] = $rooms;
 return json_encode($rarray);
}

function getOffers(){
 global $conn;
 $rarray = array();
 $result = $conn->query("SELECT * FROM offers");
 $num_rows = $result->num_rows;
 $offers = array();
 if($num_rows > 0)
 {
 $result2 = $conn->query("SELECT name, img, available, code FROM
offers");
 while($row = $result2->fetch_assoc()) {
 $one_offer = array();
 $one_offer['name'] = $row['name'];
 $one_offer['img'] = $row['img'];
 $one_offer['available'] = $row['available'];
 $one_offer['code'] = $row['code'];
 array_push($offers,$one_offer);
 }
 }
 $rarray['offers'] = $offers;
}

```

```
 return json_encode($rarray);
}

function deleteOffer($name){
 global $conn;
 if(checkOffer($name)){
 $rarray = array();
 $result = $conn->prepare("DELETE FROM offers WHERE name=?");
 $result->bind_param("s",$name);
 $result->execute();
 $rarray['success'] = "Deleted successfully";
 }else{
 header('HTTP/1.1 400 Bad request');
 $rarray['error'] = $name.", ne postoji.";
 }
 return json_encode($rarray);
}

function checkOffer($name){
 global $conn;
 $result = $conn->query("SELECT * FROM offers where name='$name'");
 $num_rows = $result->num_rows;
 if($num_rows > 0)
 {
 return true;
 }else{
 return false;
 }
}

function getOffer($name){
 global $conn;
 $rarray = array();
 $result = $conn->query("SELECT * FROM offers where name='$name'");
 $num_rows = $result->num_rows;
 $offers = array();
 if($num_rows > 0)
 {
 while($row = $result->fetch_assoc()) {
 $one_offer = array();
 $one_offer['name'] = $row['name'];
 $one_offer['img'] = $row['img'];
 $one_offer['available'] = $row['available'];
 $one_offer['code'] = $row['code'];
 array_push($offers,$one_offer);
 }
 }
 $rarray['offers'] = $offers;
 return json_encode($rarray);
}

function updateOffer($name,$img,$available,$code){
 global $conn;
```

```

$rray = array();
$stmt = $conn->prepare("UPDATE offers SET name=?, img=?, available=? WHERE
code=?");
$stmt->bind_param("ssis", $name, $img, $available,$code);
if($stmt->execute()){
 $rray['success'] = "updated";
} else{
 $rray['error'] = "Database connection error";
}
return json_encode($rray);
}

function addOffer($name, $img, $available, $code){
global $conn;
$errors = "";
if(checkifCodeExists($code)){
 $errors .= "Username već postoji\r\n";
}
if(strlen($name) < 5){
 $errors .= "Ime mora imati minimum 5 karaktera\r\n";
}
if(strlen($img) < 4){
 $errors .= "Put do slike mora imati najmanje 4 karaktera\r\n";
}
if(strlen($available) < 1){
 $errors .= "Polje je ostalo prazno\r\n";
}
if(strlen($code) < 3){
 $errors .= "Sifra mora imati najmanje 3 karaktera\r\n";
}
$rray = array();
if($errors=="")
{
$stmt = $conn->prepare("INSERT INTO offers (name, img, available, code)
VALUES (?, ?, ?, ?)");
$stmt->bind_param("ssis", $name, $img, $available, $code);
if($stmt->execute()){
 $rray['success'] = "ok";
}
} else{
 header('HTTP/1.1 400 Bad request');
 $rray['error'] = json_encode($errors);
}
return json_encode($rray);
}

function checkifCodeExists($code){
global $conn;
$result = $conn->prepare("SELECT * FROM offers WHERE code=?");
$result->bind_param("s",$code);
$result->execute();
$result->store_result();
$num_rows = $result->num_rows;
}

```

```

if($num_rows > 0)
{
 return true;
}
else{
 return false;
}

function getExReservation($token){
 global $conn;
 $rarray = array();
 $result = $conn->query("SELECT * from exrooms where user_id = (SELECT
users.id from users where users.token='$token')");
 $num_rows = $result->num_rows;
 $rooms = array();
 if($num_rows > 0)
 {
 $result2 = $conn->query("SELECT (SELECT rooms.room_name FROM rooms
WHERE exrooms.room_id=rooms.room_id) as room, date_from,date_to,then_cost from
exrooms where user_id = (SELECT users.id from users where users.token='$token')");
 while($row = $result2->fetch_assoc()) {
 $one_room = array();
 $one_room['room'] = $row['room'];
 $one_room['date_from'] = $row['date_from'];
 $one_room['date_to'] = $row['date_to'];
 $one_room['then_cost'] = $row['then_cost'];
 array_push($rooms,$one_room);
 }
 }
 $rarray['rooms'] = $rooms;
 return json_encode($rarray);
}

function addReservation($token, $room, $date_from, $date_to, $number){
 global $conn;
 $available = intval(getAvailable($room));
 $errors = "";
 $today = date("Y-m-d");
 if($today>$date_from){
 $errors .= "Datum dolaska mora biti isti ili veci od danasnog!\r\n";
 }
 if($today>=$date_to){
 $errors .= "Datum odjave mora biti veci od danasnog!\r\n";
 }
 if($date_from>$date_to){
 $errors .= "Datum dolaska mora biti manji od datuma odjave!\r\n";
 }
 if($available<=0){
 $errors .= "Trenutno nema slobonih soba\r\n";
 }
}

```

```

 if($errors==""){
 $id = intval(getUser($token));
 $room_id = intval(getRoom($room));
 $cost = getCost($room)*intval($number);
 $stmt = $conn->prepare("INSERT INTO reservations (room_id, user_id,
date_from, date_to, cost) VALUES (?, ?, ?, ?, ?, ?)");
 $stmt->bind_param("iissd", $room_id, $id, $date_from, $date_to,$cost);
 if($stmt->execute()){
 $rarray['success'] = "ok";
 addExReservation($token,$room,$date_from,$date_to,$cost);
 updateAvailable($available-1,$room_id);
 }
 }else{
 header('HTTP/1.1 400 Bad request');
 $rarray['error'] = json_encode($errors);
 }
 return json_encode($rarray);
 }

function addExReservation($token, $room, $date_from, $date_to,$cost){
 global $conn;
 $id = intval(getUser($token));
 $room_id = intval(getRoom($room));
 $stmt = $conn->prepare("INSERT INTO exrooms (room_id, user_id, date_from,
date_to, then_cost) VALUES (?, ?, ?, ?, ?, ?)");
 $stmt->bind_param("iissd", $room_id, $id, $date_from, $date_to,$cost);
 if($stmt->execute()){
 $rarray['success'] = "ok";
 }
 return json_encode($rarray);
}

function getUser($token){
 global $conn;
 return $conn->query("SELECT id FROM users WHERE token =
'$token'")->fetch_object()->id;
}

function getRoom($name){
 global $conn;
 return $conn->query("SELECT room_id FROM rooms WHERE room_name =
'$name'")->fetch_object()->room_id;
}

function getCost($name){
 global $conn;
 return $conn->query("SELECT price FROM rooms WHERE room_name =
'$name'")->fetch_object()->price;
}

function getAvailable($name){
 global $conn;
 $id = intval(getRoom($name));
}

```

```
 return $conn->query("SELECT available FROM availability WHERE room_id = '$id'")->fetch_object()->available;
 }

function updateAvailable($available,$id){
 global $conn;
 $rarray = array();
 $stmt = $conn->prepare("UPDATE availability SET available=? WHERE room_id=?");
 $stmt->bind_param("ii",$available,$id);
 if($stmt->execute()){
 $rarray['success'] = "updated";
 }else{
 $rarray['error'] = "Database connection error";
 }
 return json_encode($rarray);
}
```

## PREDSTAVLJANJE ROUTER MODULA

*U ovom delu projekta predstavljene su rute na osnovu kojih se vrši navigacija po aplikaciji.*

U ovom delu projekta predstavljene su rute na osnovu kojih se vrši navigacija po aplikaciji. Sve rute se nalaze u datoteci [src/app/app-routing.module.ts](#):

```
const routes: Routes = [
 {
 path: '',
 redirectTo: 'home',
 pathMatch: 'full'
 },
 {
 path: "home",
 component: HomeComponent
 },
 {
 path: "jednok",
 component: JednokComponent
 },
 {
 path: "login",
 component: LoginComponent
 },
 {
 path: "register",
 component: RegisterComponent
 },
 {
 path: "mineReservation",
```

```
 component: MineReservationComponent,
 canActivate: [AuthGuardService]
 },
 {
 path: "offer",
 component: OfferComponent,
 canActivate: [AuthGuardService]
 },
 {
 path: "change",
 component: ChangeComponent,
 canActivate: [AuthGuardAdminService]
 },
 {
 path: "reservation",
 component: ReservationComponent,
 canActivate: [AuthGuardService]
 }
 ,{
 path: "success",
 component: SuccessComponent,
 canActivate: [AuthGuardService]
 }
];

@NgModule({
 imports: [RouterModule.forRoot(routes)],
 exports: [RouterModule]
})
export class AppRoutingModule { }
```

U ovoj fajlu kao prethodno navede nalaze se rute. Neke od tih ruta kao što su *mineReservation*, *offer*, *reservation*, *succes* su zaštićene od strane ne autorizovanih korisnika kroz *AuthGuard* servis, dok je ruta *change* zaštićena od običnih korisnika odnosno korisnika koji nisu tipa *admin*.

## FRONTEND SERVISI U APLIKACIJI - ROOMSERVICE I RESERVATIONSERVICE

*Frontend servisi u aplikaciji su sledeće datoteke koje je neophodno izložiti u dokumentaciji.*

Frontend servisi u aplikaciji su sledeće datoteke koje je neophodno izložiti u dokumentaciji.

Servis iz sledećeg listinga predstavlja pristup serverskoj strani, tačnije za pristup selekciji soba iz baze podataka:

```
import { Injectable, OnInit } from '@angular/core';
import { ApiService } from './api.service';
```

```
@Injectable()
export class RoomService implements OnInit{

 constructor(private apiService: ApiService) {}

 ngOnInit(){

 }

 public getRooms(){
 return this.apiService.get('getRooms.php').toPromise();
 }

}
```

Sledeći servis predstavlja manipulaciju podacima vezanim za rezervaciju i tu se čuvaju rezervacije kao i trenutno izvršena rezervacija, pored toga služi i za komunikaciju sa serverskom stranom.

```
import { Injectable, OnInit } from '@angular/core';
import { ApiService } from './api.service';

@Injectable()
export class ReservationService implements OnInit {

 private data: any[];
 private reservation: any[] = [];

 constructor(private apiService: ApiService) {}

 ngOnInit() {

 }

 public setData(data) {
 this.data = data;
 }

 public getData() {
 return this.data;
 }

 public getReservation() {
 return this.reservation;
 }

 public getReservations() {
 let data = {
 "id": 1,
 "name": "Room 1"
 };
 this.reservation = data;
 }
}
```

```
 token: localStorage.getItem('token')
 }
 return this.apiService.post('getExReservation.php', data).toPromise();
}

public addReservation(name, date_from, date_to, number) {
 let data = {
 name,
 token: localStorage.getItem('token'),
 date_from,
 date_to,
 number
 }
 this.reservation.push(name, date_from, date_to, number * 19);
 return this.apiService.post('addReservation.php', data).toPromise();
}
}
```

## FRONTEND SERVISI U APLIKACIJI: OFFERSERVICE I AUTHSERVICE

*Servis za proveru korisnika je od velikog značaja za svaku veb aplikaciju.*

Sledeći servis predstavlja manipulaciju podacima vezanim za ponude hotela. Kroz ovaj servis moguće je vaditi, dodavati, brisati i menjati ponude kao i čuvati postojeće ponude.

```
import { Injectable, OnInit } from '@angular/core';
import { Offer } from '../models/offer.model';
import { ApiService } from './api.service';

@Injectable()
export class OfferService implements OnInit {

 private data: any[];

 constructor(private apiService: ApiService) {}

 ngOnInit() {}

 public setData(data) {
 this.data = data;
 }

 public getData() {
 return this.data;
 }
}
```

```
public getOffers() {
 return this.apiService.get('getoffer.php').toPromise();
}

public deleteOffer(offer: Offer) {
 return this.apiService.post('deleteofferservice.php', {
 name: offer.name
 }).toPromise();
}

public updateOffer(offer: Offer) {
 return this.apiService.post('updateOfferService.php', offer).toPromise();
}

public getSingleOffer(name) {
 return this.apiService.post('getSingleOffer.php', {name: name}).toPromise();
}

public addOffer(offer: Offer) {
 return this.apiService.post('addOffer.php', offer).toPromise();
}
}
```

Servis za proveru korisnika je od velikog značaja za svaku veb aplikaciju. Ovaj servis predstavlja manipulaciju podataka koji se tiču korisnika i preko ovog servisa vrši se logovanje i registrovanje na aplikaciju kao i validacija da li je korisnik *admin* ili običan korisnik. Pored toga ovaj servis služi i za proveru da li je korisnik trenutno ulogovan na nalog koji je primenjen na sledećem servisu.

```
import { Injectable, OnInit } from '@angular/core';
import { ApiService } from './api.service';

@Injectable()
export class AuthService implements OnInit {

 private isAuthenticated: boolean;
 private isAdmin: boolean;

 constructor(private apiService: ApiService) {
 this.isAuthenticated = this.isAuthenticated();
 this.isPrivileged();
 }

 ngOnInit() {

 }

 public isAuthenticated(): boolean {

 const token = localStorage.getItem('token');
 if (token != null) {
 this.isAuthenticated = true;
 }
 }
}
```

```
 return true;
 }
 else {
 this.isAuthenticated = false;
 return false;
 }
}

private isPrivilegedHelp(): Promise<any> {
 return this.apiService.post('getAdmin.php', {token:
localStorage.getItem('token')}).toPromise();
}

public isPrivileged() {
 this.isPrivilegedHelp().then((data) => {
 if (data.user[0].privileges == 1) {
 this.isAdmin=true;
 }
 else {
 this.isAdmin=false;
 }
 }).catch((err) => {
 });
}

public login(user): Promise<any> {
 return this.apiService.post('loginservice.php', user).toPromise();
}

public register(user): Promise<any> {
 return this.apiService.post('registerservice.php', user).toPromise();
}

public setAuth(auth: boolean) {
 this.isAuthenticated = auth;
}

public getAuth(): boolean {
 return this.isAuthenticated;
}

public setAdmin(auth: boolean) {
 this.isAdmin = auth;
}

public getAdmin(): boolean {
 return this.isAdmin;
}
}
```

## AUTHGUARDSERVICE I AUTHGUARDADMINSERVICE

*Dodatne funkcionalnosti provere korisnika kroz dva dopunska servisa.*

Sledeći servis služi za proveru da li je korisnik ulogovan na nalog ili ne i na osnovu toga se vrši dozvola "šetanja" kroz navigaciju (spomenuto u prethodnom delu dokumenta).

```
import { Injectable } from '@angular/core';
import { Router, CanActivate } from '@angular/router';
import { AuthService } from './authService';
@Injectable()
export class AuthGuardService implements CanActivate {
 constructor(public auth: AuthService, public router: Router) {}
 public canActivate(): boolean {
 if (!this.auth.isAuthenticated()) {
 this.router.navigate(['login']);
 return false;
 }
 return true;
 }
}
```

Poslednji servis služi za proveru da li je korisnik ulogovan na sistem ali i proveru da li je korisnik **admin**, time se obični korisnik isključuje za korišćenje određenih stranica.

```
import { Injectable } from '@angular/core';
import { Router, CanActivate } from '@angular/router';
import { AuthService } from './authService';
@Injectable()
export class AuthGuardAdminService implements CanActivate {
 constructor(public auth: AuthService, public router: Router) {}
 public canActivate(): boolean {
 if (!this.auth.getAdmin()) {
 this.router.navigate(['home']);
 return false;
 }
 return true;
 }
}
```

## GLAVNA (RODITELJSKA) KOMPONENTA ANGULAR APIAKACIJE

*Glavna (roditeljska) komponenta Angular aplikacije i njen sadržaj su obavezni u dokumentaciji.*

Glavna (roditeljska) komponenta Angular aplikacije i njen sadržaj su obavezni u dokumentaciji.

Prva datoteka je **NgModule** klasa aplikacije **AppModule**:

```
import { BrowserModule } from '@angular/platform-browser';
import { NgModule } from '@angular/core';

import { AppRoutingModule } from './app-routing.module';
import { AppComponent } from './app.component';
import { HomeComponent } from './pages/home/home.component';
import { JednokComponent } from './pages/jednok/jednok.component';
import { InfoComponent } from './components/info/info.component';
import { FormsModule, ReactiveFormsModule } from '@angular/forms'
import { AuthGuardService } from './services/authGuardService';
import { AuthService } from './services/authService';
import { RoomService } from './services/roomService';
import { MineReservationComponent } from './pages/mine-reservation/mine-reservation.component';
import { OfferComponent } from './pages/offer/offer.component';
import { OfferService } from './services/offerService';
import { AuthGuardAdminService } from './services/authGuardAdminService';
import { ChangeComponent } from './pages/change/change.component';
import { ReservationService } from './services/reservationService';
import { RegisterComponent } from './pages/register/register.component';
import { LoginComponent } from './pages/login/login.component';
import { CardsComponent } from './components/cards/cards.component';
import { CommentsComponent } from './components/comments/comments.component';
import { ReservationComponent } from './pages/reservation/reservation.component';
import { SuccessComponent } from './pages/success/success.component';
import { HttpClientModule } from '@angular/common/http';

@NgModule({
 declarations: [
 AppComponent,
 HomeComponent,
 JednokComponent,
 InfoComponent,
 CommentsComponent,
 CardsComponent,
 LoginComponent,
 RegisterComponent,
 MineReservationComponent,
 OfferComponent,
 ChangeComponent,
 ReservationComponent,
 SuccessComponent
],
 imports: [
 BrowserModule,
 AppRoutingModule,
 FormsModule,
 ReactiveFormsModule,
 HttpClientModule
],
 providers: [
 AuthGuardService, AuthService, RoomService, OfferService, AuthGuardAdminService, ReservationService],
})
```

```
bootstrap: [AppComponent]
})
export class AppModule { }
```

U ovom fajlu može se videti importovanje modula kao i deklarisanje kreiranih komponenata. Pored toga potrebno je i *providers* polje dodati i sve servise koji su napravljeni u projektu.

Sledećim listingom je prikazan sadržaj klase glavne komponente aplikacije *AppComponent*. U ovom fajlu nalaze se *HostListener*-i koji služe za osluškivanje dešavanja na prozoru. Pa samim tim sve funkcije koje su potrebne za promenu veličine prozora kao i skrolovanje dodele su ovde.

```
import { Component, HostListener } from '@angular/core';
import { AuthService } from './services/authService';
import { Router } from '@angular/router';

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

 title = 'app';
 images: String[] = [
 'instal.jpg', 'insta7.jpg', 'insta3.jpg', 'insta4.jpg', 'insta5.jpg',
 'insta6.jpg'
];

 constructor(private _auth: AuthService) { }

 ngOnInit(): void {
 }

 public onLogOut() {
 localStorage.removeItem('token');
 this._auth.setAuth(false);
 this._auth.setAdmin(false);
 }

 @HostListener('window:scroll', ['$event'])
 onWindowScroll($event) {
 let header = document.getElementById("header");
 let image = document.getElementById("logo") as HTMLImageElement;
 if (window.pageYOffset > 100) {
 header.classList.add("sticky");
 image.src = "../../assets/images/logo-black.png"
 } else {
 header.classList.remove("sticky");
 image.src = "../../assets/images/logo.png"
 }
 }
}
```

```
@HostListener('window:resize', ['$event'])
onWindowResize($event) {
 let header = document.getElementById("header");
 if (document.body.clientWidth <= 1000) {
 document.getElementById("upper-header").classList.add("d-none");
 document.getElementById("upper-header").classList.add("d-md-block");
 header.classList.add("sticky-mini");
 }
 else {
 document.getElementById("upper-header").classList.remove("d-none");
 document.getElementById("upper-header").classList.remove("d-md-block");
 header.classList.remove("sticky-mini");
 }
}
```

## MODELSKE KLASE

*Modelske klase su neizostavan deo MVC šablonu razvoja veb aplikacija.*

*Modelske klase* su neizostavan deo MVC šablonu razvoja veb aplikacija. Upravo, u ovom delu dokumenta predstavljeni su modeli koja su bili potrebni pri realizaciji ovog projekta.

Sledi listing klase *Card*:

```
export class Card{
 constructor(
 public heading: string,
 public img: string,
 public desc: string,
 public route: string,
 public price: number,
 public night: string
){}
}
```

Sledi listing klase *Offer*:

```
export class Offer{
 constructor(
 public name: string,
 public shape: boolean,
 public img: string,
 public available: number,
 public code
){}
}
```

Sledi listing klase *User*:

```
export class User{
 constructor(
 public name: string,
 public lastName: string,
 public username: string,
 public email: string,
 public privileges: number,
 public password: string
){}
}
```

Sledi listing klase *Comment*:

```
export class Comment {
 constructor(
 public img: string,
 public comment: string,
 public name: string,
 public workplace: string
) {}
}
```

## PRIMER HTML I SCSS FAJLOVA JEDNE STRANE

*Neophodno je dokumentovati i HTML i SCSS fajlove bar jedne strane.*

Neophodno je dokumentovati i HTML SCSS i TypeScript fajlove bar jedne strane. To će upravo biti dokumentovano u ovom delu projektne dokumentacije.

Ovaj primer vezan je za komponentu koja reguliše logovanje u aplikaciju (*login.component.html*).

```
<section id="login">
 <div class="container">
 <div id="login-frame">
 <div id="login-header">

 <h3>Ulogujte se</h3>
 <div id="errors">
 <ul *ngFor="let error of errors">

 <p>{{error}}</p>

 </div>
 </div>
 <div id="login-body">
 <form id="login-form" role="form" [FormGroup]="loginForm">
 <div id="username">
 <div id="left-log">
 <i class="fa fa-user fa-2x" aria-hidden="true"></i>

```

```
</div>
<div id="right-log">
 <input type="username" name="username" class="form-control"
id="usernameLog" placeholder="Unesite Vaš username" required
 formControlName='username'>
 </div>
</div>
<div class="clr"></div>
<div class="form-group" id="password">
 <div id="left-log">
 <i class="fa fa-lock fa-2x" aria-hidden="true"></i>
 </div>
 <div id="right-log">
 <input type="password" name="password" class="form-control"
id="passwordLog" placeholder="Unesite Vaš password" required
 formControlName='password'>
 </div>
 </div>
 <div class="clr"></div>
 <button type="submit" name="submit" class="btn btn-outline-secondary"
(click)="onLogin()">Login
 <i class="fa fa-sign-in" aria-hidden="true"></i>
 </button>
</form>
</div>
<div id="login-footer">
 <p>Nemate nalog?
 Registrujte se
 </p>
</div>
</div>
</div>
</section>
```

U ovom fajlu nalazi se HTML odnosno ono što korisnik vidi. U ovom listingu nalazi se HTML prikaz login komponente.

[SCSS](#) kod koji stilizuje prikaz prethodno viđenog HTML koda dat je sledećim listingom:

```
@import '../../../../../sass/variables';
.clr {
 clear: both;
}

#login {
 background: url("../../assets/images/login-bg.jpg") no-repeat;
 background-size: cover;
 padding-top: 150px;
 padding-bottom: 25px;
 #errors {
 color: red;
 text-align: center;
 font-weight: bold;
 }
}
```

```
}

#login-frame {
 background: rgba($color: #000000, $alpha: 0.3);
 text-align: center;
}

#login-header {
 img {
 width: 400px;
 }
}

h3 {
 padding: 20px;
 color: $gold;
}

#login-body {
 padding: 20px;
 #username, #password {
 display: inline-block;
 #left-log {
 float: left;
 padding: 0 10px;
 border-right: 1px solid white;
 i{
 color: white;
 }
 }
 #right-log {
 float: left;
 padding: 0 10px;
 }
 }
 .btn{
 color: $gold;
 border-color: $gold;
 border-width: 2px;
 &:hover {
 color: black;
 background-color: $gold;
 }
 }
}

#login-footer {
 color: $gold;
 padding-bottom: 25px;
 a{
 color: white;
 }
}

}

@media screen and (max-width: 576px) {
 #login-header{
 img{
```

```
 width:200px !important;
}
}
}
```

## PRIMER TYPESCRIPT DATOTEKE JEDNE STRANE

*Neophodno je dokumentovati i TypeScript fajl bar jedne strane.*

Neophodno je dokumentovati i *TypeScript* fajl bar jedne strane. Sledećim listingom je dat prikaz *TypeScript* fajla login komponente. U njemu se čuva forma koja je povezana sa formom koja se nalazi u HTML fajlu. Pored toga tu se nalazi i promenljiva za *error* koja se puni istim ukoliko do njih dođe. Pored toga ona ima i *onLogin()* funkciju koja rešava korisnikom klik na dugme *Login*.

Listing klase komponente *LoginComponent* priložen je sledećim listingom:

```
import { Component, OnInit } from '@angular/core';
import { User } from '../../models/user.model';
import { Router } from '@angular/router';
import { AuthService } from '../../services/authService';
import { FormControl, FormGroup } from '@angular/forms';

@Component({
 selector: 'app-login',
 templateUrl: './login.component.html',
 styleUrls: ['./login.component.scss']
})
export class LoginComponent implements OnInit {

 public errors: string[] = [];

 public loginForm = new FormGroup({
 username: new FormControl(),
 password: new FormControl()
 });

 private user: User;

 constructor(private _router: Router, private _auth: AuthService) { }

 ngOnInit() {
 }

 public onLogin() {
 this.errors = [];
 this.user = new User("", "", this.loginForm.value.username, "", 0,
this.loginForm.value.password);
 this._auth.login(this.user)
 .then((user) => {
 this._router.navigateByUrl("home");
 })
 .catch(error => {
 this.errors.push(error.message);
 });
 }
}
```

```
localStorage.setItem('token', user.token);
if(user.admin==1){
 this._auth.setAdmin(true);
} else{
 this._auth.setAdmin(false);
}
this._auth.setAuth(true);
})
.catch((err) => {
 let res = err.error.error.split("\\r\\n");
 res.pop();
 res[0] = res[0].substr(1);
 res.forEach(element => {
 this.errors.push(element);
 });
});
}
}
```

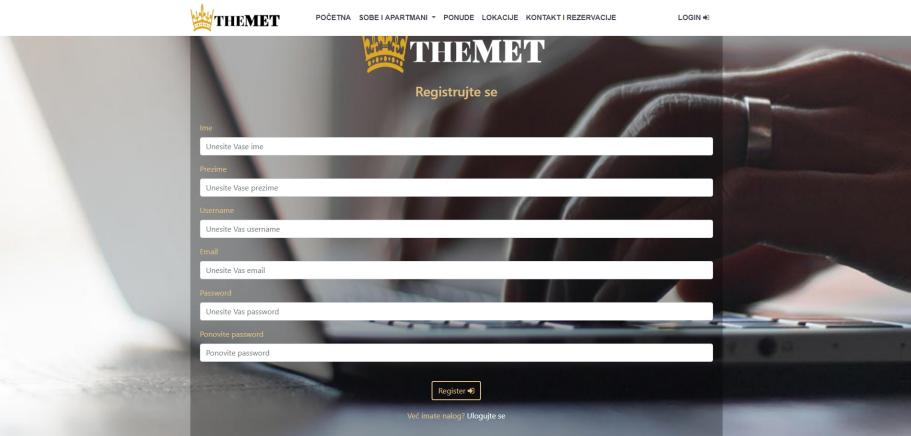
## ✓ Poglavlje 5

### Demo aplikacije

#### PRIKAZ APLIKACIJE (LOGOVANJE I REGISTRACIJA)

*Startovanjem aplikacije korisnik se prvo susreće sa stranicom za prijavljivanje.*

Startovanjem aplikacije korisnik se prvo susreće sa stranicom za prijavljivanje, a to je **prvo što je potrebno** - korisnik registruje na sistem. To može učiniti na ekranu koji je prikazan na slici.



The screenshot shows the Themet hotel registration form. At the top, there is a navigation bar with links for 'POČETNA', 'SOBE I APARTMANI', 'PONUDE', 'LOKACIJE', 'KONTAKT I REZERVACIJE', and 'LOGIN'. The main title 'THEMET' is displayed above the registration form. The registration form itself has several input fields: 'Ime' (Name), 'Prezime' (Last Name), 'Username', 'Email', 'Password', 'Ponovite password' (Repeat Password), and a 'Register' button. Below the form, there is a link 'Vec imate nalog? Ulogujte se' (Already have an account? Log in).

Slika 5.1 Prikaz registracije [izvor: autor]

Nakon uspešne registracije potrebno je da se korisnik uloguje na svoj nalog. Ecran logovanja prikazan je na slici.



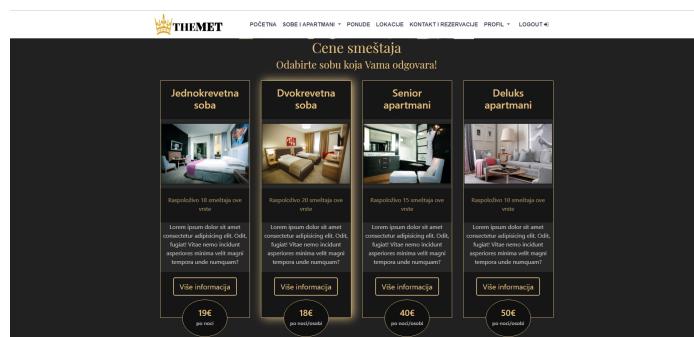
The screenshot shows the Themet hotel login page. At the top, there is a navigation bar with links for 'POČETNA', 'SOBE I APARTMANI', 'PONUDE', 'LOKACIJE', 'KONTAKT I REZERVACIJE', and 'LOGIN'. The main title 'THEMET' is displayed above the login form. The login form has two input fields: 'Unesite Vaš username' (Enter your username) and 'Unesite Vaš password' (Enter your password). Below the form is a 'Login' button. At the bottom of the page, there is a link 'Nemate nalog? Registrirajte se' (Don't have an account? Register) and a footer section containing information about the hotel, including its address, contact number, and social media links for Instagram and Facebook.

Slika 5.2 Prikaz logovanja [izvor: autor]

# STRANE ZA KORISNIKA ZA PRISTUP POČETNOM EKRANU

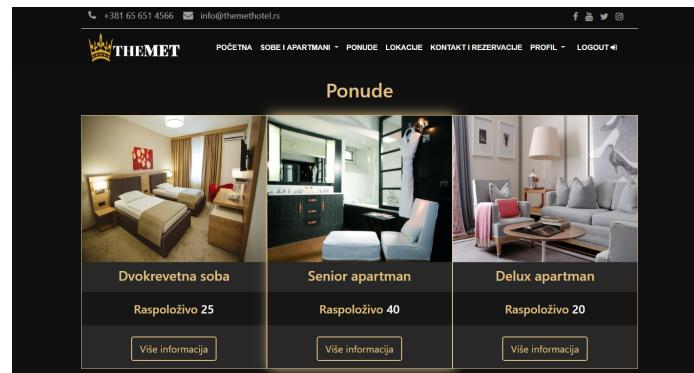
*Nakon uspešnog logovanja na sistem korisnik pristupa početnom ekranu.*

Nakon uspešnog logovanja na sistem korisnik pristupa početnom ekranu. Deo tog *početnog ekranu* prikazan je na slici.



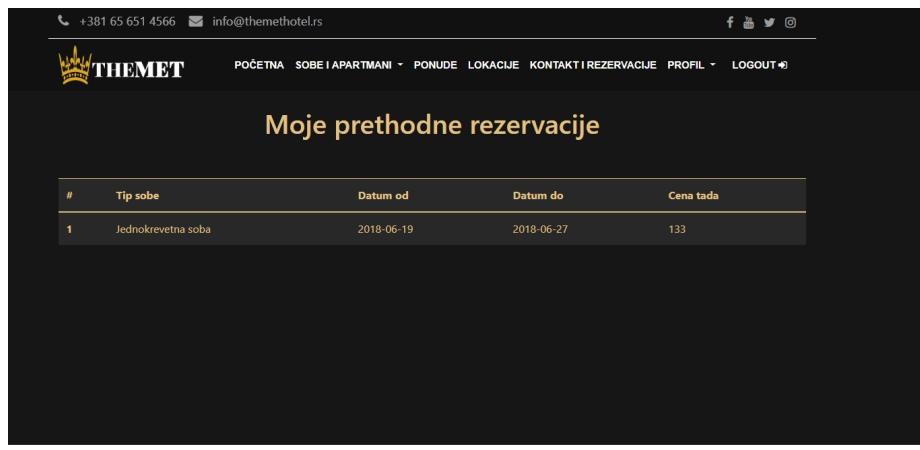
Slika 5.3 Prikaz jednog dela početne strane [izvor: autor]

Jedna od strana koju korisnik može da poseti jeste *Ponude*. Tu korisnik može da vidi sve dodatne ponude hotela.



Slika 5.4 Prikaz strane Ponude [izvor: autor]

Sledeća strana koju korisnik može da poseti jeste *Moje rezervacije* koja se nalazi u *Profil* navigaciji. Tu korisnik može da vidi prethodno realizovane rezervacije.



#### Pratite nas

Lorem ipsum dolor sit amet, consectetur adipisicing elit. Qui cumque nostrum veniam  
 tempora eius facilis apertam commodi non placeat architecto.



#### Newsletter

Lorem, ipsum dolor sit amet consectetur adipisicing elit. Laudantium animi corrupti  
 quidem corporis aut atque!

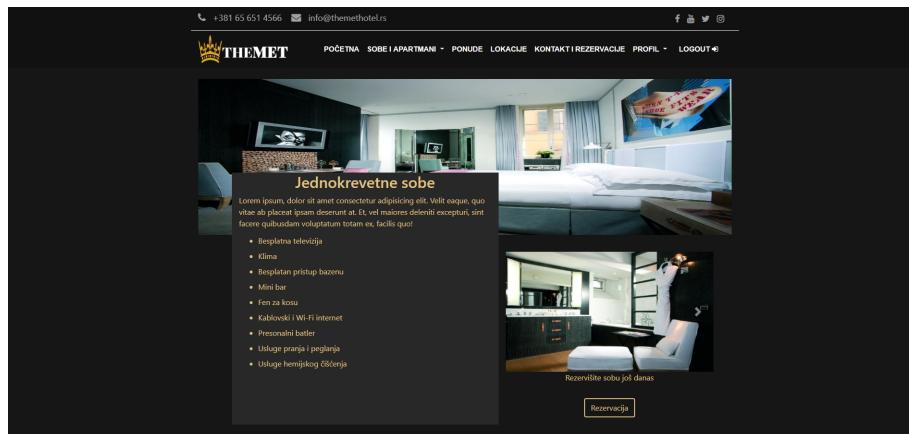


Slika 5.5 Prikaz strane Moje rezervacije [izvor: autor]

## STRANE ZA KORISNIKA ZA REZERVACIJU

*Korisnik može da vidi informacije o sobi kao i da izvrši rezervaciju.*

Korisnik takođe može da pristupi strani *Jednokrevetne sobe*. Gde može da vidi informacije o ovoj sobi kao i da izvrši rezervaciju.



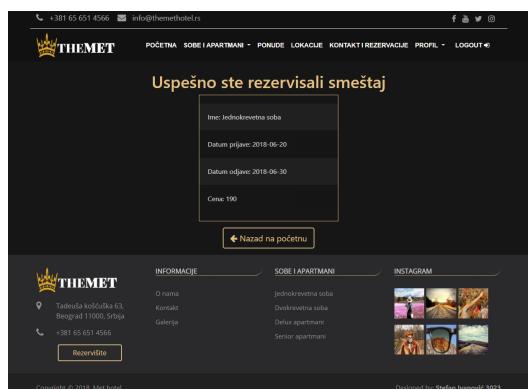
Slika 5.6 Prikaz strane Jednokrevetne sobe [izvor: autor]

Nakon što korisnik klikne na dugme *Rezervacija* sa prethodne slike on se dovodi na ekran predstavljen na sledećoj slici. Tu je potrebno da korisnik odabere datum dolaska i datum odjave kao i broj noćenja. Pritom se vrši validacija da li je datum dolaska stariji od današnjeg, kao i da li je datum odjave stariji od datum dolaska. Ukoliko dođe do takve greške korisnik se obaveštava o tome.



Slika 5.7 Prikaz rezervacije [izvor: autor]

Nakon uspešne registracije korisnik se dovodi na sledeći ekran. Tu se obaveštava o datumu prijave i odjave, imenu sobe kao i ukupnoj ceni za sve noći.

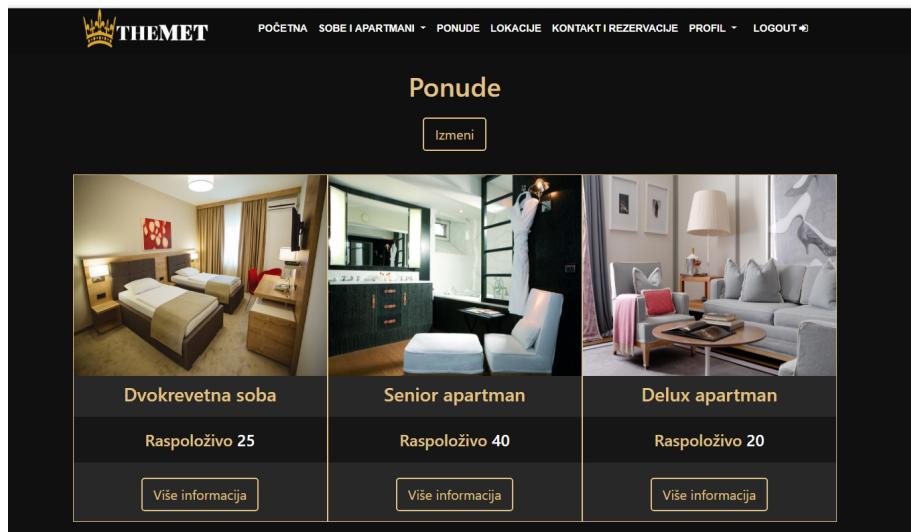


Slika 5.8 Prikaz uspešne rezervacije [izvor: autor]

## ADMIN STRANICE

*Ukoliko se korisnik uloguje kao admin on će imati opciju za izmene na strani vezanoj za ponude.*

Ukoliko se korisnik uloguje kao administrator (admin) on će imati opciju za izmene na strani vezanoj za ponude.



Slika 5.9 Prikaz strane Ponude za admina [izvor: autor]

Klikom na dugme "izmeni" sa prethodne slike korisnik se dovodi na ekran prikazan na narednoj slici. Tu korisnik može da bira da li želi da doda novi element, da obriše postojeći element ili da izmeni već postojeći element.



Slika 5.10 Prikaz opcije za izmenu ponude [izvor: autor]

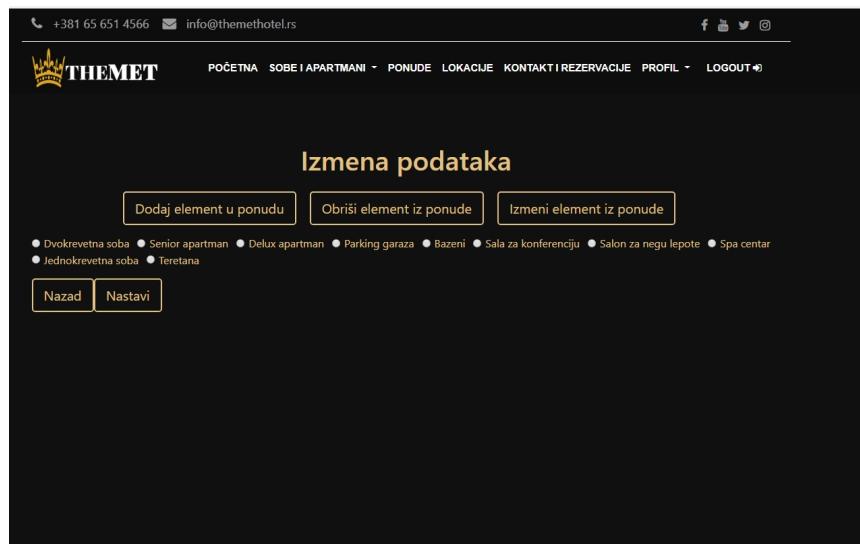
Ukoliko korisnik odabere da doda element njemu se pojavljuje forma prikazana na slici tu je potrebno da unese podatke i doda novi element.

Slika 5.11 Prikaz opcije za izmenu ponude (Dodaj element u ponudu) [izvor: autor]

# ADMIN STRANICE ZA IZMENU PODATAKA

*Korisnik može da odabere da izmeni podatak o elementu iz ponude.*

Ukoliko korisnik odabere da izmeni podatak o elementu iz ponude njemu će se prikazati sve raspoložive ponude u vidu radio dugmadi i onda je potrebno da selektuje ponudu i klikne na "nastavi".

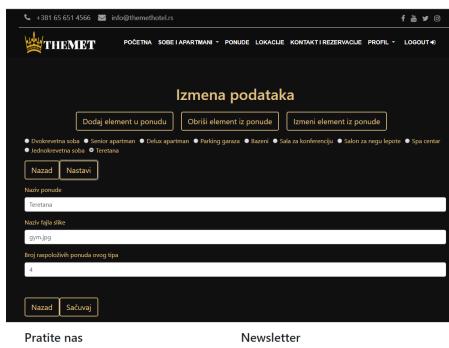


Slika 5.12 Prikaz opcije za izmenu elementa [izvor: autor]

Nakon selektovan ponude korisniku se pojavljuje forma prikazana na sledećoj slici sa ispunjenim trenutnim podacima o ponudi. Ukoliko korisnik izvrši izmenu potrebno je klikne na dugme Sačuvaj i završi sa izmenom ponude.

Slika 5.13 Prikaz selektovane ponude za izmenu [izvor: autor]

Ukoliko korisnik odabere da obriše element iz ponude njemu se javljaju sve raspoložive ponude u vidu **check-box** kontrola gde može da selektuje više njih. Nakon što je korisnik selektovao željene ponude za brisanje potrebno je da klikne dugme "Obriši" i izvrši brisanje.



Slika 5.14 Prikaz brisanja elemenata iz ponude [izvor: autor]

## VIDEO MATERIJAL

*Pogledajte video materijale, mogu biti od velike pomoći za izradu domaćeg zadatka.*

**Angular - Common Questions (and Answers!)** - trajanje 34:06

**Ova lekcija sadrži video materijal. Ukoliko želite da pogledate ovaj video morate da otvorite LAMS lekciju.**

**Angular Interview Questions and Answers | Angular 8 Interview Preparation | Edureka** - trajanje 37:20.

**Ova lekcija sadrži video materijal. Ukoliko želite da pogledate ovaj video morate da otvorite LAMS lekciju.**

## ✓ Poglavlje 6

### Vežbe 15 - Prezentacije i ocene projekata-135 min

## PRAVILA IZRADE I ODBRANE PROJEKTA, PROJEKAT I RAD NA PROJEKTU

*Projekat je najvažnija i najznačajnija predispitna obaveza studenta.*

#### **Projekat:**

Projekat je najvažnija i najznačajnija predispitna obaveza studenta. Na početku semestra objavljuje se spisak tema projektnih zadataka. Student može da izabere jednu od ponuđenih tema i da u dogovoru sa asistentom, dobije sve neophodne dodatne informacije i instrukcije za rešavanje izabranog projektnog zadatka. Svaki student dobija jedinstven i specifični projektni zadatak najkasnije do 3. nedelje nastave. Uz poseban zahtev i obrazloženje, asistent može studentu odobriti i projektni zadatak na temu koju predloži student, ali asistent je dužan da definiše ostale neophodne podatke i informacije za rad, tj. određuje kontekst u slučaju u kome se rešava prihvaćen projektni zadatak. Projektni zadatak definiše sve aktivnosti koje student treba da uradi da bi došao da krajnjeg rezultata koji se traži. Student je dužan da u Izveštaju u urađenom projektnom zadatku, prvo navede (citira) projektni zadatak, a onda da u svakom poglavlju koje se odnosi na jednu aktivnost projekta, svoj rad i rezultata rada na toj aktivnosti. Pri tome, poštuje redosled aktivnosti definisane projektnom zadatkom. Na kraju projekta, izlaže konačan rezultat i daje zaključak izveštaja. Tekst Izveštaja se piše u trećem licu jednine i mora da bude tehnički, jezički i profesionalno dobro urađen, usklađen za uzorkom dokumenta koji definiše asistent.

#### **Rad na projektu:**

Student, po pravilu, kontinualno radi svoj projekat tokom semestra, u skladu sa stečenim novim i potrebnim znanjima. Preporučuje se studentu da u toku rada, konsultuje asistenta, tj. da mu da na uvid urađeni deo projekta, kako bi bio siguran da je dobro shvatio svoj zadatak i primenio pravilan način rešavanja projektnog zadatka. Projekat se ne radi na kraju semestra ili kasnije. On se radi paralelno sa savladanjem gradiva, kako bi olakšao studentu savladavanje novog gradiva i time pomogao u pripremi za savladavanje narednog gradiva na predmetu.

## PREDAJA I ODBRANA PROJEKTA

*Pravila vezana za predaju i ocenjivanje projekta.*

#### **Predaja projekta:**

Zahtevan rok za predaju Izveštaja o urađenom projektnom zadatku je do kraja 14. nedelje nastave za studente tradicionalne studente nastave (Beograd) ili hibridne nastave (Centar u Nišu), a za studente onlajn nastave – 10 dana pre ispita. Studentima tradicionalne i hibridne nastave koji predaju Izveštaj o urađenom projektnom zadatku 15 i više dana po završenoj nastavi na predmetu, umanjuje se broj ostvarenih poena za 30%. **Krajnji rok za predaju projekta, za sve studente, je 10 dana pre ispitnog roka u kome žele da polažu ispit.** Studentima onlajn nastave (koji ne prate nastavu u Centru u Nišu) se ne umanjuje broj poena, jer su oni, po pravilu zaposleni, ili sprečeni da redovno studiraju (primenom tradicionalnog i hibridnog oblika nastave).

### ***Obrana i ocenjivanje projekta:***

Ako student prilikom ocenjivanja projekta ne dobije najmanje 50% predviđenih poena (15 poena), on mora da ga doradi. U suprotnom, dobija 0 poena. Student koji ne dobije više od 50% predviđenih poena ne može izaći na ispit. Student je dužan da projekat odbrani kod asistenta. Odbrana projekata studenata tradicionalne nastave i hibridne nastave (u Centru u Nišu) vrši se u 15. nedelji jesenjeg semestra za vreme vežbi, a ako je potrebno, i van vežbi – na dodatnim časovima. Van ovog termina, student može da brani projekat u posebnom terminu koji odredi asistent pred svaki ispitni rok. Cilj odbrane projekta je da asistent ustanozi da li je student samostalno radio projekat i u tom cilju odbrana projekta podrazumeva da student demonstrira znanje koje je iskoristio za izradu projekta. Odbrana projekta za studente internet nastave obavlja se preko Skype-a, a ako je za studenta prihvatljivo, u prostorijama univerziteta.

Na LAMS sistemu se nalazi uputstvo za izradu projekta.

## ✓ Poglavlje 7

### Domaći zadatak 15 -

#### ZADATAK

*Priprema za posao junior frontend programera.*

Nakon pređenih nastavnih materijala, prostudirajte i sledeće linkove. Cilj je da pokušate da simulirate razgovor za posao junior frontend programera.

1. <https://hackr.io/blog/angular-interview-questions>
2. <https://github.com/sudheerj/angular-interview-questions>
3. <https://www.greycampus.com/blog/programming/top-30-interview-questions-and-answers-on-angular-5>

Uradite sledeći test za simulaciju razgovora sa Angular poslodavcem:

- <https://www.testdome.com/d/angular-interview-questions/77>

Na dnu stranice (prethodni link) imate više korisnih testova za simulaciju razgovora sa Angular poslodavcem, uradite jedna po vašem izboru.

**Dokumentujte rezultate urađenog testa.**

## ▼ Poglavlje 8

### Zaključak

## ZAKLJUČAK

*Lekcija je razvijana kao projektna dokumentacija konkretnog veb projekta.*

Savladavanjem gradiva predmeta IT255 - Veb sistemi 1 lekcije student je osposobljen sa samostalno razvija frontend aplikacije u *Angular* okviru do nivoa potreba junior Angular programera. Cilj ove lekcije je bio da se zaokruži izlaganje predmeta primerom celokupnog razvoja veb aplikacije bazirane na razvoju klijentskog dela primenom *Angular* okvira

Lekcija je razvijana kao projektna dokumentacija konkretnog veb projekta.

Studenti će nastaviti da nadograđuju vlastito znanje iz veb razvoja u narednim predmetima, koji se fokusiraju na serversku stranu veb aplikacija, *CS330 - Distribuirani sistemi* i *IT355 - Veb sistemi 2*.

## LITERATURA

*Za pripremu lekcije korišćena je aktuelna pisana i elektronska literatura.*

### **Pisana literatura:**

1. Nate Murray, Felipe Coury, Ari Lerner, Carlos Taborda, ng-Book – The Complete Book on Angular 6, Fullstack.io, 2018
2. Cody Lindley, Frontend Handbook – 2017, Frontend masters, 2017
3. Sandeep Panda, AngularJS – From Novice to Ninja, SitePoint Pty. Ltd, 2014

### **Elektronska literatura:**

4. <https://angular.io/>
5. <https://angular.io/tutorial>
6. <https://www.w3schools.com/angular/>
7. <https://www.tutorialspoint.com/angular4/>
8. <https://nodejs.org/en/>
9. <https://code.visualstudio.com/>
10. [https://www.w3schools.com/whatis/whatis\\_htmldom.asp](https://www.w3schools.com/whatis/whatis_htmldom.asp)
11. <https://angular.io/guide/dependency-injection>

12. <https://hackr.io/blog/angular-interview-questions>
13. <https://github.com/sudheerj/angular-interview-questions>
14. <https://www.greycampus.com/blog/programming/top-30-interview-questions-and-answers-on-angular-5>
15. <https://www.testdome.com/d/angular-interview-questions/77>