**UNIVERZITET U BANJOJ LUCI**

**ELEKTROTEHNIČKI FAKULTET**

**Tatjana Kovačević**

**RAZVOJ JAVASCRIPT BAZIRANIH APLIKACIJA**

**DIPLOMSKI RAD**

**Banja Luka, maj 2019.**

Tema: RAZVOJ JAVASCRIPT BAZIRANIH APLIKACIJA

Ključne riječi:

*Web* Aplikacije

JavaScript

Node.js

TypeScript

Angular

Komisija: Prof. dr Zoran Đurić, predsjednik

Prof. dr Slavko Marić, mentor

Igor Dujlović, ma, član

Uz rad je priložen CD.

Kandidat:

Tatjana Kovačević

UNIVERZITET U BANJOJ LUCI

ELEKTROTEHNIČKI FAKULTET

KATEDRA ZA RAČUNARSTVO I INFORMATIKU

**Predmet**: DIPLOMSKI RAD

**Tema**: RAZVOJ JAVASCRIPT BAZIRANIH APLIKACIJA

**Zadatak**: Dati pregled najznačajnijih osobina JavaScript-a i HTML-a. Napraviti sistem koji omogućava naručivanje hrane iz različitih restorana koristeći Angular platformu. Osim web aplikacije, razviti i jednostavnu mobilnu aplikaciju pomoću JavaScript razvojnih alata, kao i serversku aplikaciju pomoću NodeJS-a. Detaljno opisati korištene tehnologije i izvršiti analizu performansi i skalabilnosti.

**Mentor**: Prof. dr Slavko Marić

**Kandidat**: Tatjana Kovačević (1221/12)

Bana Luka, jun 2019.

Sadržaj

[1. Uvod 1](#_Toc9960897)

[2. HTML 3](#_Toc9960898)

[3. CSS3 9](#_Toc9960899)

[4. JavaScript 11](#_Toc9960900)

[4.1 Uvod 11](#_Toc9960901)

[4.2 Osobine *JavaScript*-a 11](#_Toc9960902)

[4.3 Client Side i Server Side *JavaScript* 12](#_Toc9960903)

[4.4 Događaji 12](#_Toc9960904)

[5. Node.js 15](#_Toc9960905)

[5.1 Uvod 15](#_Toc9960906)

[5.2 Event-Driven model programiranja 15](#_Toc9960907)

[5.3 Moduli 16](#_Toc9960908)

[5.4 Express.js 18](#_Toc9960909)

[5.5 Middleware 19](#_Toc9960910)

[6. TypeScript 21](#_Toc9960911)

[6.1 Uvod 21](#_Toc9960912)

[6.2 TypeScript sistem tipova 21](#_Toc9960913)

[6.3 TypeScript funkcionalnosti 22](#_Toc9960914)

[7. Angular.io 25](#_Toc9960915)

[7.1 Uvod 25](#_Toc9960916)

[7.2 Arhitektura 26](#_Toc9960917)

[8. Ionic 31](#_Toc9960918)

[8.1 Uvod 31](#_Toc9960919)

[8.2 Arhitektura 32](#_Toc9960920)

[8.3 *User Interface* komponente 32](#_Toc9960921)

[9. OPIS IMPLEMENTACIJE APLIKACIJE ZA NARUČIVANJE HRANE 34](#_Toc9960922)

[9.1 Serverski dio aplikacije 35](#_Toc9960923)

[9.2 Klijentski dio aplikacije 39](#_Toc9960924)

[9.3 Analiza performansi i skalabilnosti 47](#_Toc9960925)

[10. Zaključak 51](#_Toc9960926)

[Literatura 53](#_Toc9960927)

# Uvod

Sve masovnija upotreba Interneta, koji je trenutno najpopularniji komunikacioni medij, dovela je do toga da *web* bazirane aplikacije sve češće zamjenjuju *desktop* aplikacije. *Web* aplikacije su postale važan alat za poslovanje i najčešće se koriste za komunikaciju sa klijentima, saradnju sa zaposlenima, čuvanje i prezentaciju podataka. Za razliku od *desktop* aplikacija koje zahtijevaju instalaciju i nadograđivanje, *web* bazirane aplikacije zahtijevaju samo internet konekciju. *Web* aplikacije se izvršavaju u okviru *web* čitača i korisnicima su dostupne dok god imaju internet konekciju.

Tema ovog diplomskog rada je razvoj *JavaScript* baziranih aplikacija, te će u narednim poglavljima biti opisani *JavaScript* jezik i tehnologije bazirane na *JavaScript*-u. Opisane tehnologije su korištene za razvoj praktičnog dijela diplomskog rada. Praktični dio rada je razvoj aplikacije za naručivanje hrane iz različitih restorana.

*JavaScript* je programski jezik koji se može koristiti za pisanje *web* aplikacija, podržan u *web* čitačima. Zajedno sa HTML-om i CSS-om čini osnovu za razvoj dinamičkih i skalabilnih web aplikacija. *JavaScript* nudi veliki broj tehnologija i biblioteka za razvoj kompleksnih aplikacija. Tehnologije za razvoj *web* aplikacija koje su bazirane na *JavaScript*-u, vremenom su postale sve naprednije i jednostavnije za korištenje. Prednost *JavaScript*-a je ta što se može koristiti za razvoj i klijentske i serverske strane *web* aplikacija.

U drugom poglavlju je opisan HTML jezik, bez koga izrada *web* stranica i *web* aplikacija nije moguća. Koristi se za opis strukture *web* stranica. Opisani su najznačajniji tagovi i atributi, te dati jednostavni primjeri njihove primjene. Takođe, opisana je i posljednja verzija ovog standarda.

U trećem poglavlju dat je opis CSS3 kao posljednje verzije CSS tehnologije za definisanje stila i izgleda *web* stranica.

U četvrtom poglavlju opisan je *JavaScript* programski jezik za razvoj web aplikacija. Dat je kratak uvod u kome se opisuje na koji način se *JavaScript* može koristiti i kako je nastao. Zatim su date osobine ovog jezika, zajedno sa primjerima. Opisani su događaji u *JavaScript-*u koji se koriste za asinhrono izvršavanje programa.

U petom poglavlju opisana je tehnologija bazirana na *JavaScript*-u za razvoj serverske strane *web* aplikacije. Dat je pregled svih komponenti ove tehnologije koje su potrebne za razvoj aplikacije koja se zasniva na događajima.

Sadržaj šestog poglavlja je opis *TypeScript* jezika kao proširenje *JavaScript* jezika. Dati su ciljevi *TypeScript*-a, zašto se koristi i osnovne funkcionalnosti sa primjerima.

U sedmom poglavlju opisana je tehnologija za razvoj klijentskih *web* aplikacija. Ova tehnologija je bazirana na *TypeScript*-u i HTML-u. Dat je uvod o samoj tehnologiji, a zatim su opisani elemeti koji čine aplikaciju koja se razvija u ovoj tehnologiji.

U osmom poglavlju opisan je razvojni okvir pomoću kojeg se kreiraju mobilne aplikacije. Dat je uvid u arhitekturu i komponente ovog razvojnog okvira.

Sadržaj devetog poglavlja je opis implementacije praktičnog dijela rada. Opisane su funkcionalnosti aplikacije i način na koji su realizovane. U ovom poglavlju opisane su performanse i skalabilnost aplikacije, dok je u posljednjem, desetom poglavlju dat zaključak.

# HTML

HTML (eng. *Hypertext Markup Language*) je opisni jezik koji specifikuje strukturu *web* stranica. Može se reći da je HTML jezik za opis *hypertext*-a, gdje *hypertext* predstavlja tekst koji sadrži veze ka drugim stranicama ili ka segmentima stranice na kojoj se nalaze. HTML je prilično jednostavan jezik sastavljen od elemenata koji strukturiraju dokument u logičke cjeline, formatiraju tekst i ugrađuju sadržaj kao što su slike i video zapisi na stranicu. [[1](#Int19)]

HTML dokument ili stranica se sastoji od teksta i tagova (eng. *markups*) koji *web* čitaču daju instrukcije kako da prikaže HTML stranicu. Svaki tag ima svoje ime i odgovarajuće atribute i navodi se pomoću para tagova, otvarajućeg taga *<ime\_taga>* i zatvarajućeg taga *</ime\_taga>*. Takođe postoje i prazni tagovi koji se navode na sljedeći način *<ime\_taga/>*. Tag zajedno sa sadržajem koji se navodi između otvarajućeg i zatvarajućeg taga predstavlja element HTML dokumenta. Atributi taga se navode unutar otvarajućeg taga i njihove vrijednosti se uokviruju navodnicima. Atributi nam omogućavaju korištenje elemenata na različite načine zavisno od okolnosti. Na Slici 2.1 prikazan je primjer korištenja atributa.

<html>

<head>

<title>Primjer</title>

</head>

<body>

<img src=”img.jpg” alt=”slika”/>

</body>

</html>

*Slika 2.1 – Primjer korištenja atributa u HTML-u*

Tag *<img/>* u HTML datoteci definiše prikazivanje slike, gdje atribut *src* definiše ime slike ili putanju do slike koja se treba prikazati, a atribut *alt* definiše tekst koji će biti ispisan ukoliko slika iz nekog razloga nije učitana.

Osnovnu strukturu HTML dokumenta čine sjedeći tagovi: *<html>*, *<head>* i *<body>*. Svaki HTML dokument počinje *<html>* tagom u okviru kojeg se nalaze ostali tagovi i završava *</html>* tagom. Tag *<head>* sadrži metapodatke (podatke o HTML datoteci: ko je autor, ključne riječi, itd.), veze ka CSS stranicama, tag *<title>* koji definiše naslov stranice. Sve što se nalazi u tagu *<head>,* osim sadržaja taga *<title>* neće biti prikazano kada *web* čitač učtita stranicu. Sadržaj taga *<body>* je sve ono što će biti prikazano kada se stranica učita. Može imati atribute kojima će se specifikovati karakteristike dokumenta kao što su: boja teksta, boja pozadine ili slika pozadine, akcije koje će se izvršiti kada se dokument učita ili ne učita i slično. [[2](#ETi)]

*Web* čitač ne prikazuje HTML tagove, nego ih koristi da bi odlučio kako da prikaže sadržaj dokumenta i prepozna paragraf, listu, link, zaglavlje i ostale elemente HTML dokumenta.

Kao što je rečeno, postoje elementi koji se označavaju parom tagova i kaže se da takvi elementi definišu određen sadržaj u stranici. Takođe, postoje elementi sa praznim tagom i takvi elementi se koriste za ubacivanje nekog sadržaja u stranicu. Slijede primjeri ovakvih elemenata.

Parovi tagova:

* *<p>Ovo je pasus.</p>* - Pasusi se navode između *<p>* tagova. Unutar početnog taga može se navesti atribut *align* kojim se definiše horizontalno poravnanje pasusa i može imati jednu od sljedećih vrijednosti: *left*, *center*, *right* i *justify*.
* *<font size=”10”>Tekst veličine 10.</font>* - Omogućava promjenu veličine, boje i vrste fonta.
* *<ul>* - Neoznačena (eng. *Unordered*) lista kojom se definiše lista elemenata ispred kojih se nalazi *bullet*. Svaki element definiše se tagom *<li>*. Ukoliko se umjesto *<ul>* koristi tag *<ol>*, radi se o numerisanoj (eng. *Ordered*) listi. Postoje i definicione liste koje se definišu pomoću para tagova *<dl>* i sastoje se iz dva dijela: *termina* i *definicije*.
* *<a href=”https://etf.unibl.org/”>ETF</a>* - Linkovi se definišu pomoću para tagova *<a>*. Atributom href se definiše pozicija na koju se stiže ukoliko se klikne na dati link.
* *<iframe src=”https://etf.unibl.org/”></iframe>* - Ovim tagom se definiše *inline frame*, kojim se omogućava ubacivanje drugog dokumenta unutar trenutnog.
* *<form>* - Ovaj par tagova definiše formu koja omogućava korisniku da unese neke informacije (npr. unos korisničkog imena, lozinke, izbor elementa iz padajuće liste i slično). Atributom *method* se definiše način prenosa informacija (GET ili POST), dok se atributom *action* definiše adresa stranice na koju će informacije biti prenešene. Najčešće korišten tag unutar *<form>* taga je *<input>* kojim se definiše element forme. Tip unosa određuje se pomću atributa *type* i tipovi mogu biti *text*, *password*, *image*, *file*, *select*, *radio*, *checkbox*, *submit*, *button*.

Prazni tagovi:

* *<img/>* - Ovo je tag pomoću kojeg se ubacuje slika u stranicu.
* *<br/>* - Koristi se za dodavanje nove linije.
* *<hr/>* - Slično prethodnom, koristi za razdvajanje teksta. Predstavlja horizontalnu liniju sa praznim redom ispred i iza te linije. Pomoću atributa *size* može se definisati visina, a pomoću *width* širina.
* *<meta>* - Definišu dodatne informacije koje *web* čitač ne prikazuje.

HTML5 je najnovija verzija HTML standarda koja nudi nove karakteristike (eng. *features*), poboljšava podršku za kreiranje *web* aplikacija za komunikaciju sa klijentima, sa njihovim lokalnim podacima i serverima na jednostavnji i efikasniji način nego što je to bilo ranije. [[3](#KFö)] Preporučen je i odobren od strane *W3C[[1]](#footnote-1)* (eng. *World Wide Web Consortium)* organizacije.

Svaki HTML5 dokument počinje deklaracijom tipa dokumenta *<!DOCTYPE html>*, koja je obavezna i dosta kraća od onih u prethodnoj verziji. Takođe u ovoj verziji se primjenjuje kraći način za određivanja kodovanja *<meta charset=”UTF-8”>*. Pored ovoga HTML5 ima dosta novih tagova i atributa ali i onih koji se u ovoj verziji više ne koriste. Slijede neke od najznačajnih karakteristika.

Struktura i semantika dokumenta:

* *<header>* - Ovaj tag odnosi se na kontejner za naslov i dodatni uvodni sadržaj. Ne odnosi se samo na zaglavlje na vrhu stranice, već se može koristiti i u ostalim dijelovima stranice.
* *<article>* - Reprezentuje nezavisno područje web dokumenta, za npr. novosti, blog ili sličan sadržaj.
* *<footer>* - Ovaj element sadrži dodatne informacije o sekciji na koju se odnosi, kao što su: autor, povezane stranice, autorska prava i slično. Može da se nalazi unutar *<article>* elemeta.
* *<nav>* - Koristi se ukoliko je potrebno kreirati navigacione blokove preko kojih se prelazi na neku labelu unutar dokumenta ili na neku drugu povezanu stranicu.
* *<aside>* - Koristi se za sadržaj koji je sporedan u odnosu na najbitniji sadržaj stranice i treba da bude vezan za taj njega.
* *<section>* - Sadrži sekciju dokumenta kao što je poglavlje eseja, u sklopu zaglavlja ili *footer*-a.
* *<figure>* - Koristi se za prikazivanje nezavisnih slika, grafika, dijagrama i unutar njega se može nalaziti element *<figcatption>* za navođenje naslova.
* *<meter>* - Element za grafičko predstavljanje vrijednosti u zadatom opsegu.
* *<progress>* - Grafički predstavlja napredak procesa završetka zadatka.

Pametne forme:

HTML5 specifikacija poboljšava *<input>* element dozvoljavajući nove vrijednosti za *type* atribut, a to su: *tel* – ulazna vrijednost je telefonski broj, *search* – ulazno polje je *search*, *url* – ulazna vrijednost je *url*, *email* – ulazna vrijednost je *email*, *datetime*, *date*, *month*, *week*, *time*, *datetime-local*, *number*, *range*, *color* – u heksadecimalnom zapisu. Takođe, HTML5 nudi nekoliko novih atributa za elemente forme: *autofocus*, *placeholder*, *required*.

Novi elementi forme:

*<datalist>* - Padajuća lista opcija za ulaznu vrijednost. Input element koristi atribut *list* za povezivanje sa *datalist*.

*<keygen>* - Koristi se za generisanje para kriptografskih ključeva. Javni ključ se šalje zajedno sa svim podacima forme kako bi bio dostupan serveru, dok privatni ključ ostaje na strani klijenta.

*<output>* - Predstavlja rezultat izračunavanja, gdje izračunavanje npr. može biti rezultat množenja unešenog broja u input polje sa nekom vrijednošću.

Video i Audio:

Pomoću taga *<video>* se u okviru pretraživača može pogledati video sadržaj. Kako bi bila omogućena podrška za različite čitače i uređaje, video treba da bude enkodovan u više formata. Na Slici 2.2 je prikazan primjer upotrebe taga.

<!DOCTYPE HTML>

<html>

<head>

<title>Video</title>

</head>

<body>

<video width=”320” height=”240” controls>

<source src=”movie.mp4” type=”video/mp4”>

<source src=”movie.ogg” type=”video/ogg”>

<source src=”movie.webm” type=”video/webm”>

Your browser does not support the video tag.

</video>

</body>

</html>

*Slika 2.2 – Primjer upotrebe taga u HTML5*

Kao što se vidi na slici, unutar taga *<video>*, navode se tagovi *<source>* kojima se obezbijeđuje kompatibilnost za različite *web* čitače. Navođenjem atributa *controls* obezbijeđuje se prikazivanje kontrolnih elemenata za video.

Za uključivanje audio fajla u stranicu koristi se tag *<audio>*. Slično kao kod *<video>*, elementom *<source>* se obezbijeđuju različiti audio fajlovi, kako bi web čitač izabrao odgovarajući. [[3](#KFö)]

Canvas:

Jedan od najinteresantnijih elemenata u HTML5 je <canvas>. Njime je obezbijeđeno iscrtavanje grafičkih elemenata. Iscrtavanje se vrši pomoću JavaScript-a, a element <canvas> predstavlja kontejner.

Canvas element se pronalazi u DOM-u (eng. *Document Object Model*) pomoću id-a, a onda se nad tim elementom poziva metoda *getContext()* koja vraća drawing context. Sada kada je drawing context definisan, moguće je iscrtavanje elemenata.

Geolocation API:

*Geolocation API* (eng. *Application programming interface*) omogućava korisniku da *web* aplikaciji otkrije svoju lokaciju, ukoliko to želi. Kako bi se obezbijedila privatnost, od korisnika se traži dozvola za dobijanje informacija o lokaciji. *Geolocation API* je dostupan putem  *navigator.geolocation* objekta. Lokacija korisnika se može dobiti pozivom metode *getCurrentPosition()*. Lokacijske informacije se mogu dobiti pomoću različitih izvora: *IP* adrese, *WiFi*, *GPS*, korisničkog unosa. [[4](#19Fe)]

Web Storage i Offline Web Applications:

*Web Storage* predstavlja interfejs koji definiše atribute i metode za perzistentno čuvanje podataka na klijentskoj strani. Podaci se čuvaju kao parovi ključ-vrijednost i moguća su dva mehanizma za njihovo skladištenje. Prvi mehanizam je *session storage* - podaci smješteni u ovaj *storage* dostupni su samo od strane aktivnog prozora aplikacije koji ih je kreirao. Po zatvaranju prozora podaci se brišu iz *session storage*-a. *Local storage* je drugi mehanizam u kome se podaci čuvaju i po završetku sesije. Ukoliko korisnik u istom čitaču otvori novi prozor istog *web* sajta i izmjeni vrijednost određenog ključa u *local storage*-u, ova vrijednost biće izmjenjena i za prvi prozor istog sajta. Ovo je osnovni nedostatak čuvanja podataka u *local storage*-u u odnosu na *session storage*. [[3](#KFö)]

*Offline Web Applications* - kako bi klijent mogao pokrenuti aplikaciju bez pristupa mreži, potrebno je da se HTML, *JavaScript* i multimedijalni fajlovi čuvaju na klijentskoj mašini. *Web* čitač će potrebne podatke čuvati u lokalnoj *cache* memoriji. Kako bi aplikacija radila *offline*, potrebna je *manifest* datoteka kojom se čitaču govori šta je potrebno čuvati u *cache* memoriji. U situacijama kada klijent nema mrežnu konekciju ili pokušava pristupiti resursu koji se ne nalazi u lokalnom (*offline*) *cache*-u, HTML5 može da specifikuje *fallback* slučaj kojim se klijentu uslužuje neki drugi resurs. [[3](#KFö)]

Web Sockets:

*Web Socket* predstavlja najmoćnije svojstvo komunikacije u HTML5 specifikaciji. Definiše *full-duplex* komunikacijski kanal koji radi preko jednog *socket*-a preko internet mreže, što znači da klijent i server mogu slati podatke u bilo kojem trenutku. [[5](#PLu11)]

Web Workers:

*Web Workers* obezbjeđuju izvršavanje zadataka u pozadini tako da web stranica može nastavi sa radom. *Web Worker* i pozivajuća skripta komuniciraju na sljedeći način: pozivajuća skripta kreira objekat *Worker* i u konstruktoru prosljeđuje naziv skripte koju *worker* treba da izvši. Sada pozivajuća skripta može da pošalje poruku *worker*-u pozivajući *postMessage()* metodu. U *worker*-u je definisana *event handler* *onmessage* koji će obraditi podatke poslane kroz *postMessage()* metodu. Na isti način *worker* može poslati poruku kao odgovor. *Web Worker* nema pristup *DOM* elemetima i time se potencijalno sprečava da dvije skripte pokušaju pristupiti istom elemetu. [[3](#KFö)] [[6](#MDN192)]

Microdata:

*Microdata* predstavlja način da se *meta* podaci smještaju unutar sadržaja *web* stranice. *Search engines*, *web crawlers* i *browsers* mogu izdvojiti i pretražiti *microdata* i time korisnicima obezbjediti bogatiji i relevantniji pregled stranice. [[7](#MDN191)]

Globalni atributi:

Globalni atributi su atributi zajednički za sve HTML elemente, mogu se koristiti za sve elemente, ali na neke elemente nemaju uticaj. Neki od zajedničkih atributa su: *accesskey* - definiše prijedlog za generisanje prečice na tastaturi za trenutni element, *autocapitalize* - definiše da li i na koji način se unešeni tekst automatski piše velikim slovom, *class*-klase dozvoljavaju *CSS*-u i *JavaScript*-u da pristupe određenim elementima preko selektora klasa, *contenteditable*-atribut koji definiše da li se dati element može uređivati od strane korisnika, *id* - jedinstveni identifikator elementa koji je jedinstven u čitavom dokumentu, njegova svrha je da dozvoli *CSS*-u i *JavaScript*-u pristup datom elementu, *style* - sadrži *CSS* deklaracije stila koje treba da se primjene na element. [[8](#MDN19)]

# CSS3

CSS (*Cascading Style Sheets*) se koristi za stil i izgled *web* stranice. Stilovi se definišu za HTML elemente i mogu da promijene boju, font ili pozadinu elementa. [[9](#W3C19)] Korištenjem CSS-a kao eksternog dokumenta razdvaja se sadržaj i formatiranje stranice, pa se *web* stranice brže učitavaju. Takođe, ovim je omogućeno ažuriranje više različitih stranica.

CSS3 predstavlja posljednju verziju evolucije CSS-a i u potpunosti je kompatibilan sa prethodnim verzijama. Najveća promjena u CSS3 verziji su moduli kojima se postiže efikasniji i laksi rad. Neki od najvažnijih modula su: selektori, tekstualni efekti, pozadine i ivice, 2D i 3D transformacije, animacije. [[10](#MDN193)]

Selektori se koriste za izbor elementa kome se definiše stil, neki od novih selektora u CSS3 su: *element1~element2* – za selekciju svakog element2 ispred koga se nalazi element1, *[attribute]* – za selekciju elemenata koji imaju određeni atribut, *[attribute=value]* – za selekciju elemenata koji imaju istu vrijednost određenog atributa, *[attribute\*=value]* – za selekciju elemenata kod kojih vrijednost određenog atributa sadrži određenu riječ, *[attribute^=value]* – za selekciju elemenata kod kojih vrijednost određenog atributa počinje određenom riječi, *[attribute$=value]* – za selekciju elemenata kod kojih vrijednost određenog atributa završava određenom riječi, *:active* – za selekciju aktivnog elementa, npr. linka, *::after* – za ubacivanje sadržaja nakon određenog elementa, npr.: *p::after, :only-child* – selektuje svaki element koji je jedini element u roditeljskom elementu, *:nth-child(n)* – selektuje element koji je n-ti u roditeljskom elementu, itd.

Tekstualni efekti u CSS3 imaju nove osobine: *text-align-last, text-emphasis, text-overflow, text-justify, text-outline, text-shadow, word-breake, word-wrap*.

Pozadine i ivice takođe imaju nove osobine, pozadine: *background-color, background-image, background-repeat, background-attachment*; ivice: *border-color, border-style, border-width, border-radius, border-image-(source, slice, width, outset, repeat)*. Na Slici 3.1 je prikazano korištenje slike kao ivice.

<!DOCTYPE html>

<html>

<head>

<style>

#borderimg {

border-image: url(border.png) 30 30 round;

}

</style>

</head>

<body>

<p id=borderimg>Border-image</p>

</body>

</html>

*Slika 3.1 – Primjer korištenja slike kao ivice*

CSS3 podržava 2D i 3D transformacije. Transformacija je efekat koji omogućava da element promijeni oblik, veličinu i poziciju. Osobine transformacija u CSS3: *transform*, *translate*, *scale*, *rotate*, *skew*, *matrix*.

Animacija omogućava da element postepeno mijenja izgled. Da bi se koristile animacije, potrebno je odrediti neke ključne okvire *keyframes* kojima se definiše izgled elementa u određenom trenutku.

# JavaScript

## Uvod

*JavaScript* je platformski nezavisan, objektno orijentisan programski jezik koji se koristi u svrhu kreiranja dinamičkih *web* stranica. [[11](#MDN194)] *JavaScript* može funkcionisati kao proceduralni jezik ili kao jezik baziran na protipovima objekata. Objekti se kreiraju programski zajedno sa pripadajućim metodama i atributima. Jednom kreiran objekat može se koristiti kao prototip za kreiranje sličnih objekata.

*JavaScript* se sve češće koristi u razvojnim okruženjima koji su zasnovani na *JavaScript*-u za razvoj serverske strane. Jedno od takvih okruženja je *Node.js* koje je opisano u poglavlju 5.

JavaScript je nastao u kompaniji *Netscape*, koji je predstavio ovaj jezik *ECMA[[2]](#footnote-2) (*eng. *Europen Computer Manufacturers Association)* organizaciji za standarde. *ECMA* je razvila standard i preimenovala jezik u *ECMAScript*. U praksi se ipak češće koristi naziv *JavaScript*. [[12](#DFl11)]

## Osobine *JavaScript*-a

U *JavaScript*-u nije dozvoljeno deklarisanje tipova podataka promjenjivih. *JavaScript* tumač (eng. *interpreter*) prepoznaje tip podatka i dinamički ga dodjeljuje promjenjivoj. *JavaScript* tipovi mogu biti podijeljeni u dvije kategorije: primitivni i objekti. Primitivni tipovi mogu biti: *boolean*, *null*, *undefined*, *number*, *string*, *symbol*. Objekti se mogu posmatrati kao kolekcija atributa. Vrijednosti atributa mogu biti bilo kojeg tipa, i mogu se dinamički dodati ili ukloniti. Na Slici 4.1 je prikazan jedan od načina kreiranja i korištenja objekta u JavaScript-u.

function Person(first\_name, last\_name, age){

this.first\_name = first\_name;

this.last\_name = last\_name;

this.age = age;

}

var person1 = new Person(“John”, “Doe”, 25);

document.getElementById(“demo”).innerHTML =

“My name is ” + person1.first\_name;

*Slika 4.1 – Primjer kreiranja objekta u JavaScript-u*

Funkcije se posmatraju kao objekti, jer mogu imati svoje atribute i metode kao i svaki objekat. Razlika u odnosu na objekte je ta da funkcije mogu biti pozvane. [[13](#MDN195)] Funkcija se može proslijediti kao argument neke druge funkcije, može biti povratna vrijednost druge funkcije i može biti dodjeljena kao vrijednost određenoj varijabli. Definicija funkcije je prikazana na Slici 4.2.

function myFunction(param1, param2){

// statements

}

*Slika 4.2 – Definicija funkcije u JavaScript-u*

## Client Side i Server Side *JavaScript*

Izvršavanje *JavaScript* koda na klijentskoj strani u okviru čitača podrazumijeva da taj *web* čitač ima ugrađen *JavaScript* *interpreter*. [[12](#DFl11)] U HTML stranici koja će biti učitana, pomoću taga *<script>* *web* čitaču se daje naznaka da sve što se nalazi unutar ovog taga treba biti tumačeno kao *JavaScript* skripta. Tada skripta može dinamički da mijenja sadržaj učitane stranice, koristeći *DOM* kojim se omogućava pristup objektima stranice.

Postoji više *JavaScript* platformi za razvoj klijentskih *web* aplikacija, a neke od popularnih su *Angular*, *ReactJS[[3]](#footnote-3)*, *Ember.js[[4]](#footnote-4)*, *Vue.js[[5]](#footnote-5)*. Pomoću ovih platformi programeri mogu na brz i jednostavan način razvijati dinamičke web aplikacije. Takođe, postoje različite biblioteke koje se mogu koristiti prema zahtjevima. Za razvoj grafičkog korisničkog interfejsa često se koriste biblioteke *Bootstrap* i *jQuery*.

*JavaScript* se za razvoj serverske strane može koristi za komunikaciju sa bazom podataka, prihvatanje i obradu klijentskih zahtjeva, kreiranje i slanje odgovora klijentu ili manipulaciju fajlovima.

## Događaji

Ranije su se *JavaScript* programi mogli izvršavati isključivo sinhrono, što znači da se u jednom trenutku može izvšiti samo jedna operacija. Ovakav način izvršavanja danas je neuobičajen i nepraktičan. Danas se programi pišu tako da se za svaki događaj poziva *event handler* funkcija. Ove funkcije se pozivaju asinhrono kada se desi događaj za koju je funkcija napisana. Asinhrono programiran je stil programiranja, gdje se umjesto povratne vrijednosti, definišu funkcije koje će biti pozvane kada se desi događaj. Događaji mogu biti učitavanje stranice, klik na dugme, unos teksta u polje i slično. Svaki događaj ima svoj tip i ciljani objekat, gdje ciljani objekat predstavlja *DOM* element. Tipovi događaja mogu biti *click*, *change*, *load*, *mouseover*, *keypress* ili *readystatechange*. [[14](#W3C191)] Ukoliko želimo da naš program odgovori na događaj klik na dugme pomoću funkcije *event handler*, treba da registrujemo, odnosno napišemo ovu funkciju na način da tipu događaja dodamo prefiks *on.* Na Slici 4.3 je primjer.

buttonOk = document.getElementById(‘idButton’);

buttonOk.onclick = function(){

alert(‘button ok’);

}

*Slika 4.3 – Primjer funkcije koja se poziva na događaj*

U *JavaScript*-u se za asinhrono izvršavanje i vraćanje povratne vrijednosti kada je rezultat spreman koriste još i *Callback* funkcije. *Callback* funkcije se izvršavaju nakon što se neka druga funkcija izvrši. Kako su funkcije u *JavaScript*-u objekti prve klase, *Callback* funkcija se proslijeđuje kao argument nekoj drugoj funkciji iz koje će se *Callback* funkcija pozvati. [[15](#MDN196)] Na Slici 4.4 slijedi primjer.

{

function greeting(name){

alert(‘Hello ’ + name);

}

function processUserInput(callback){

var name = prompt(‘Please enter your name.’);

callback(name);

}

processUserInput(greeting);

}

*Slika 4.4 – Primjer kreiranja i pozivanja Callback funkcije*

Prvo se izvršava funkcija *processUserInput()* i kao argument joj se prosljeđuje *Callback* funkcija *greeting*. Kada se prosljeđuje *Callback* funkcija kao argument, prosljeđuje se samo njena definicija. Funkcija *processUserInput()* od korisnika zahtjeva unos imena. Kada korisnik unese ime, poziva se *Callback* funkcija i prosljeđuje joj se ime koje je upisano u varijablu *name*.

Pored *Callback* funkcija koriste se i *Promise*-i. *Promise* je objekat koji predstavlja rezultat izvršavanja asinhrone operacije. Rezultat može biti status da li je operacija uspješno ili neuspješno izvršena. [[16](#MDN197)]

*Promise* obezbjeđuje povezivanje rukovaoca sa uspješnim izvršavanjem ili neuspješnim izvršavanjem. Ovim se omogućava da asinhrona metoda vrati „obećanje“ da će vratiti vrijednost u nekom budućem trenutku. *Promise* može da ima jedno od sljedeća tri stanja: *pending* – inicijalno stanje, *fulfilled* – znači da je operacija uspješno izvršena, *rejected* – označava da je operacija neuspješna. Kada se desi jedna od navedenih opcija, pridruženi rukovaoci koji su stavljeni u red čekanja *then* metode *Promise*-e se pozivaju. Na Slici 4.5 je primjer kreiranja i korištenja *Promise.*

var promise = new Promise((resolve, reject)=> {

mysql.getConnection((err, connection)=> {

if(err) reject(err);

else{

connection.query('SELECT \* FROM restaurant', function(err, rows){

resolve(JSON.stringify(rows));

});

}

});

});

promise.then(data=>{

console.log(data);

}).catch(еrr=>{

console.log(err);

})

*Slika 4.5 – Primjer kreiraanja i pozivanja Promise*

# Node.js

## Uvod

*Node.js*[[6]](#footnote-6) je platforma za razvoj brzih i skalabilnih serverskih aplikacija baziran na *JavaScript*-u. [[17](#PTe13)] Dizajniran je tako da bude dobar za intezivne I/O aplikacije koje koriste *event-driven* arhitekturu. *Event-driven* programiranje čini Node.js dobrim i efikasnim za izvršavanje aplikacija koje se u realnom vremenu pokreću na udaljenim uređajima. [[18](#CGa13)]

*Node.js* pokreće jednu nit i sve zahtjeve izvršava asinhrono. To znači da kada se desi događaj, poziva se *callback* funkcija, registrovana za taj događaj. Dok se čeka izvršavanje ove funkcije, sljedeći događaj ili funkcija je u redu za izvršavanje. Na serverskoj strani, događaji se mogu definisati kao asinhrone ulazno/izlazne (*asynchronous I/O*) poruke. [[19](#CIh13)]

Postoji više razloga zašto je *Node.js* sve više pogodan za razvoj serverske strane. Jedan od njih je taj što je *JavaScript* najčešće korišten programski jezik web aplikacija, jednostavan je i osnovne funkcionalnosti su svedene na minimum. Još jedna od činjenica koja ide u prilog korištenju *Node.js*-a za razvoj serverske strane *web* aplikacija je jednostavno preuzimanje i instaliranje, a zatim i pokretanje u svega nekoliko minuta.

## Event-Driven model programiranja

*Event-Driven* programiranje predstavlja stil programiranja kod koga je tok izvršavanja programa određen događajima. Događajima se rukuje pomoću *callback* funkcija. Ovo je jedna od karakteristika koje definišu *Node.js*. Na Slici 5.1 je prikazan poziv *callback* funkcije koja dobavlja podatke iz baze.

function getRestaurants(){

database.getRestaurantsDB(function(data){

console.log(data);

});

}

//database

function getRestaurantsDB(callback){

mysql.getConnection(function(err, connection){

if(err) callback(err);

else{

connection.query('SELECT \* FROM restaurants',

function(err, rows){

callback(rows);

})

}

})

}

*Slika 5.1 – Primjer Callback funkcije u kojoj se kreira upit na bazu*

Unutar *getRestaurants()* poziva se funkcija *getRestaurantsDB()* i kao argument joj se prosljeđuje *Callback* funkcija *function(data)*. Funkcija *getRestaurantsDB()* izvršava upit nad bazom, ukoliko je došlo do greške pozvaće se *callback* i kao argument se prosljeđuje *err* (greška nastala pri izvršavanju funkcije *getConnection*). U slučaju da je upit uspješan, funkciji *callback* se kao argument prosljeđuje *rows* (rekordi koji se dobiju izvršavanjem upita). U oba slučaja, po pozivu, funkcija *callback* će ispisati prosljeđeni argument (*err* ili *rows*).

Kod *Event-Driven* modela programiranja trenutni proces ne blokira I/O operacije, pa se zbog toga može pojaviti više I/O operacija u paraleli za koje će biti pozvane odgovarajuće *callback* funkcije.

*Event-Driven* stil programiranja praćen je *event-loop*-om. *Event-loop* je konstrukcija koja obavlja dvije funkcije u neprekidnoj petlji - *event detection* (funkcija koja detektuje događaj) i *event handler triggering* (funkcija za poziv rukovaoca događajem). [[17](#PTe13)] *Event-loop* mora da detektuje događaj kada se desi i da odredi odgovarajuću *callback* funkciju i pozove je. *Event-loop* je samo jedna nit koja se izvodi unutar jednog procesa, što znači, da kada se desi događaj, *event handler* može da radi bez prekida. To znači da u svakom trenutku postoji samo jedan *event handler* i *event handler* koji je pokrenut biće izvršen bez prekidanja. Ovo je razlog zašto programeri ne moraju brinuti o izvršavanju istovremenih niti.

## Moduli

Kao što je ranije rečeno, *Node.js* je jednostavan za korištenje i za kreiranje jednostavnih aplikacija mogu se koristiti samo osnovne funkcionalnosti *Node.js*-a. Kada je potrebno kreirati kompleksniju aplikaciju, tada se na jednostavan način mogu uključiti *Node.js* moduli. Pomoću modula postiže se bolja organizacija koda i izbjegavaju se greške i sigurnosni problemi.

U *Node.js*-u moduli se referenciraju po putanji ili po imenu. Moduli su podijeljeni u tri grupe. Postoje *core* moduli koji se učitavaju pri pokretanju *Node.js* procesa. Drugi tip modula su *third-party* koji se instaliraju u aplikaciji. I treći tip modula su *local* moduli koje kreira programer. [[17](#PTe13)]

*Core* ili osnovni moduli su kompajlirani u binarne distribucije i iako se automatski učitavaju pri pokretanju *Node.js*-a, potrebno ih je prvo uvesti kako bi se mogli koristiti. Neki od osnovnih modula:

* *http* – *http* modul uključuje klase, metode i događaje za kreiranje *Node.js* *http* servera.
* *url* – *url* modul uključuje metode za URL (eng. *Uniform Resource Locator*) rezoluciju i parsiranje.
* *fs* – *fs* modul uključuje klase, metode i događaje za rad sa fajlovima

*Core* moduli se referenciraju samo pomoću imena, a ne putanje i učitavaju se iako postoji *third-party* moduo sa istim imenom. Učitavanje *http* *core* modula prikazano je na Slici 5.2.

var http = require('http');

*Slika 5.2 – Učitavanje http modula u Node.js-u*

Funkcija *require* vraća *http* objekat koji implementira API pomoću kojeg *Node.js* prenosi podatke preko HTTP (eng. *HyperText Transfer Protocol*) protokola.

*Third-party* ili eksterni moduli instaliraju se u aplikaciji pomoću NPM (eng. *Node Package Manager*). NPM predstavlja repozitorijum *Node.js* modula i dolazi zajedno sa instalacijom *Node.js*-a. Na Slici 5.3 je prikazano kako se instalira modul pomoću NPM-a.

$ npm install <naziv modula>

*Slika 5.3 – Instaliranje modula u Node.js-u*

Ovako kreirani moduli smještaju se u lokalni direktorijum *node\_modules* i mogu se učitati pomoću funkcije *require* i prosljeđivanjem imena modula, kao na Slici 5.2.

*Local* modul se kreira na način da se kreira *JavaScript* datoteka koja izvozi objekat, koji predstavlja API modula. Ovi moduli se mogu distribuirati putem NPM-a, tako da ih *Node.js* zajednica može koristiti. Referenciranje *local* modula vrši se pomoću apsolutne ili relativne putanje. Na Slici 5.4 je prikazano kreiranje *local* modula.

// message.js

function info(info){

console.log(‘Info: ’ + info);

}

function error(error){

console.log(‘Error: ’ + error);

}

module.exports = {

info,

error

}

*Slika 5.4 – Primjer kreiranja lokalnog modula*

## Express.js

*Express.js*[[7]](#footnote-7) je najpopularnije razvojni okvir (eng. *framework*) za *Node.js* i osnovna biblioteka za brojna druga razvojna okruženja. [[20](#MDN198)]

Nudi sljedeće mehanizme:

* pisanje rukovaoca (eng. *handlers*) za korisničke zahtjeve sa različitim HTTP metodama i URL putanjama
* integrisanje sa „*view*“ mehanizmima za prikazivanje, kako bi se generisali odgovori ubacivanjem podataka u šablone (eng. *templates*)
* postavljanje *web* postavki aplikacije kao što su: port koji se koristi za povezivanje, lokaciju šablona koji se koriste za prikazivanje odgovora
* dodavanje „middleware“ za dodatnu obradu korisničkih zahtjeva

Express.js se instalira pomoću NPM, naredbom koja je prikazana na Slici 5.5.

$ npm install express --save

*Slika 5.5 – Instaliranje Express framework-a*

Na ovaj način će *Express.js* biti instaliran i sačuvan u listi zavisnosti (eng. *dependencies*). Instaliranjem *Express.js*-a dobijaju se datoteke na osnovu kojih se bazira nova aplikacija. Prva je manifest datoteka *package.json*. Osnovna namjena ove datoteke je ta da sadrži listu zavisnosti modula koje NPM može koristiti da bi ih instalirao. Kada se instaliraju ove zavisnosti smještaju se u *node\_modules* folder. Sljedeći je *public* folder, koji sadrži foldere za statičke datoteke koji mogu biti potrebni aplikaciji. Neki od standardnih foldera su: *images*, *javascripts*, *stylesheets* sa svojim datotekama koje korisnici mogu mijenjati. Pored ovoga, *Express.js* kreira datoteku *app.js* koja sadrži inicijalizacioni kod servera. Na Slici 5.6 je prikazana jednostavna *Express.js* aplikacija koja pokreće server i sluša na portu 5000.

var express = require(‘express’);

var app = express();

app.get('/', function(req, res){

res.send('Hello');

})

var server = app.listen(5000, function(){

var host = server.address().address;

var port = server.address().port;

console.log('App listening at http://%s:%s', host, port);

})

*Slika 5.6 – Kreiranja servera u Express aplikaciji*

## Middleware

Inicijalizacioni kod *app.js* datoteke dodaje *middleware* funkcije. *Express.js* aplikacija je u suštini serija poziva *middleware* funkcija.

*Middleware* funkcije su funkcije koje imaju pristup *request* objektu, *response* objektu i sljedećoj *middleware* funkciji u *request-response* ciklusu aplikacije. Sljedeća funkcija *middleware* je označena varijablom *next*. *Middleware* funkcije mogu obavljati sljedeće zadatke: izvršavanje nekog koda, promjenu *request* i *response* objekta, završiti *request-response* ciklus, poziv sljedeće *middleware* funkcije sa *stack*-a. Ukoliko tekuća *middleware* funkcija ne završi *request-response* ciklus, mora pozvati sljedeću *middleware* funkciju kojoj će proslijediti kontrolu. [[21](#Exp19)]

Express aplikacije mogu da koriste sljedeće tipove *middleware*-a:

* *Application-level middleware* – povezuje se *application-level middleware* sa objektom aplikacije koristeći *app.use()* i *app.METHOD()* funkcije, gdje *METHOD* može biti *GET*, *PUT* ili *POST*,
* *Router-level middleware* – radi na isti način kao prethodna, s tim što se veže za instancu *express. Router()*,
* *Error-handling middleware* - funkcija se definiše isto kao i druge *middleware* funkcije, ali ima još jedan argument *err*,
* *Built-in middleware* – Express ima sljedeće *build-in middleware* funkcije: *express.static*, *express.json*, *express.urlencoded*,
* *Third-party middleware* – koristi se za dodavanje novih funkcionalnosti Express aplikaciji.

Rutiranje zahtjeva – različiti dijelovi aplikacije treba da aktiviraju različito ponašanje servera. Npr. *GET* i *POST* zahtjevi se trebaju posmatrati različito i različiti URL-ovi takođe aktiviraju različite odgovore. Ukoliko želimo da neka funkcija vraća korisničke profile koji se čuvaju npr. u bazi podataka, različitom kombinacijom HTTP metoda i URL-ova dobijaju se različiti odgovori:

* *GET/users* – za prikazivanje liste korisničkih profila,
* *GET/users/:username* – za prikazivanje profila korisnika sa proslijeđenim korisničkim imenom,
* *POST/users* – za kreiranje korisničkog profila,
* *PUT/users/:username* - za ažuriranje profila korisnika sa proslijeđenim korisničkim imenom.

Prethodna lista predstavlja listu rutiranja, gdje se kombinacije metoda i URL-ova mapiraju u odgovarajuće akcije, odnosno *callback* funkcije. *Callback* funkcije su najčešće definisane u odvojenim modulima koji se nazivaju kontroleri (eng. *controllers*). Ukoliko je potrebno kreirati funkciju koja šalje odgovor na zahtjev *GET/users* iz prethodne liste, kreira se modul *controllers/user.controler.js* koji je prikazan na Slici 5.7.

// *controllers/user.controller.js*

var db = require(‘./database/user.database’);

function getAllUsers(req, res){

db.getAllUsers(function(data){

res.send(data);

})

}

*Slika 5.7 – Primjer kreiranja kontrolera*

U kodu na prethodnoj slici, pozivom *require(‘./database/user.database’)* uvezen je moduo sa funkcijama koje dobavljaju podatke iz baze podataka. Na sličan način se kreiraju *callback* funkcije za druge zahtjeve iz liste.

Dobra praksa je da su rute za različite API-je (npr.: *Users API*, *Tasks API*) definisane u različitim modulima. Moduli se kreiraju u folderu *routes*. Na Slici 5.8 je primjer.

// routes/user.routes.js

module.exports = function(app){

var usersControler = require(‘./../controllers/user.controller.js’);

app.get(‘/users’, usersControler.getAllUsers);

app.post(‘/user’, usersControler.createUser);

app.put(‘/user’, usersControler.updateUser);

app.get(‘user/:id’, usersControler.getUser);

}

// app.js

var app = express();

require(‘./routes/user.routes’)(app);

*Slika 5.8 – Primjer kreiranja modula za rutiranje*

U prethodnim primjerima primjenjen je *application-level* tip *middleware*-a.

# TypeScript

## Uvod

*TypeScript*[[8]](#footnote-8) je programski jezik koji je razvijen od strane *Microsoft[[9]](#footnote-9)* korporacije. *TypeScript* je proširenje *JavaScript*-a koje treba da omogući lakši razvoj velikih *JavaScript* aplikacija. [[22](#NBr16)] Svaki *JavaScript* program je *TypeScript*, s tim da *TypeScript* po sintaksi predstavlja nadskup *JavaScript*-a. *TypeScript* kompajler prevodi *TypeScript* datoteku i emituje *JavaScript* ekvivalent u svrhu izvršavanja u okviru pretraživača ili nekog razvojnog okruženja.

Postoje dva osnovna cilja *TypeScript*-a:

* Obezbijeđivanje opcionog sistema tipova za *JavaScript*
* Obezbijeđivanje planiranih funkcija iz narednih *JavaScript* verzija [[23](#BAS17)]

Potreba za ovim ciljevima je opisana u nastavku.

## TypeScript sistem tipova

Tipovi podataka imaju sposobnost da poboljšavaju kvalitet i razumijevanje koda. Npr. bolje je da kompajler prijavi grešku u kodu, ukoliko ona postoji, nego da do greške dođe u toku izvršavanja programa.

Tip može biti implicitan ili eksplicitan. Kada je tip implicitan, *TypeScript* će pokušati da zaključi onoliko informacija o tipu koliko može da sa sigurnošću odredi tip uz minimalne troškove produktivnosti pri razvoju koda. Na Slici 6.1 u prvoj liniji koda *TypeScript* će znati da je *foo* tipa *number*, a u drugoj liniji će prijaviti grešku: *Type ‘456’ is not assignable to type number*.

let foo = 456;

foo = ‘456’;

*Slika 6.1 – Primjer implicitne deklaracije tipa podatka*

Kod eksplicitnog navođenje tipa, *TypeScript* koristi *postfix* anotaciju, što je prikazano na Slici 6.2.

let foo: number = 123;

*Slika 6.2 - Primjer ekspicitne deklaracije tipa podatka*

U prethodnim primjerima korištena je *let* deklaracija varijabli. Ovaj način deklarisanja uveden je u *TypeScript* kako bi se riješili nedostaci *var* deklaracije. Opseg *let* varijable je blok-opseg, što znači da je dostupna samo u bloku koji je sadrži, npr.: funkcija, *if else* blok ili *loop*. Varijabla *let* ne može biti ponovo deklarisana u bloku u kome se nalazi. Pored *let* deklaracije postoji i *const* koja ima isti blok-opseg. Za razliku od *let*, vrijednost *const* varijable ne može biti promijenjena. *Const* varijabla mora biti inicijalizovana u istoj liniji gdje je i deklarisana.

## TypeScript funkcionalnosti

*TypeScript* nudi funkcionalnosti koje su planirane u *EcmaScript6*. Slijedi objašnjenje ovih funkcionalnosti:

Klase – *TypeScript* podržava objektno-orjentisano programiranje zasnovano na klasama. Kreiranje klase Student je prikazano na Slici 6.3.

class Student {

firstName: string;

lastName: string;

constructor(firstName: string, lastName: lastName){

this.firstName = firstName;

this.lastName = lastName;

}

fullname(){

return this.firstName + ‘ ’ + this.lastName;

}

}

*Slika 6.3 – Primjer kreiranja klase u TypeScript-u*

Nasljeđivanje – Klase u *TypeScriptu* podržavaju jednostruko nasljeđivanje korištenjem ključne riječi *extends*.

*Arrow* funkcije - *Arrow* funkcija je sažeti način pisanja funkcije:

* Nije potrebno pisati *function*
* Leksički obuhvata značenje *this* referenci
* Leksički obuhvata značenje argumenata

Na Slici 6.4 je primjer *arrow* funkcije:

let fullname = (firstName, lastName):string => {

return firstName + ‘ ’ +lastName;

}

*Slika 6.4 – Primjer kreiranja Arrow funkcije*

Vrijednost reference *this* unutar funkcije u *JavaScript*-u zavisi od toga kako je funkcija pozvana. U *Arrow* funkcijama vrijednost reference *this* je sačuvana. Kada se pokrene kod sa Slike 6.5, vrijednost reference *this*, čitač će prepoznati kao objekat *window*, tada će funkcija *nextYear* pristupiti *undefined* atributu *year*, pa će rezultat ispisa biti 2.

*Slika 6.5*

function Student(year){

this.year = year;

this.nextYear = function() {

this.year++;

}

}

var student new Student(2);

setTimeout(student.next,1000);

setTimeout(function() { console.log(student.year); },2000);

Ovaj nedostatak se može riješiti upotrebom *arrow* funkcije, pa će rezultat ispisa biti 3, primjer je prikazan na Slici 6.6

*Slika 6.6*

function Student(year){

this.year = year;

this.nextYear = () => {

this.year++;

}

}

var student new Student(2);

setTimeout(student.next,1000);

setTimeout(function() { console.log(student.year); },2000);

Interfejsi – Jedno od osnovnih *TypeScript* principa je da se provjera tipova fokusira na oblik koji vrijednosti imaju. U *TypeScript*-u, interfejsi ispunjavaju ovaj princip. Na Slici 6.5 je prikazano kreiranje interfejsa:

interface Student {

name: string

}

function printName(obj: Student){

console.log(obj.name);

}

let student = {year: 4, name: “Mark”};

printName(student);

*Slika 6.5 – Primjer kreiranja interfejsa u TypeScript-u*

Možemo primjetiti da objekat koji se prosljeđuje funkciji *printName* ne implementira interfejs Student. Objekat ima atribut *name* čime je zadovoljen zahtjev interfejsa. Takođe, redoslijed kojim svojstva dolaze nije bitan, bitno je da su svojstva koja interfejs zahtjeva prisutna i da imaju zahtjevani tip.

Moduli – Slično kao što je opisano u Node.js-u i *TypeScript* koristi module. Moduli se izvršavaju u okviru svog opsega, osim ukoliko su eksplicitno izvezeni pomoću *export* forme. Pored toga, kako bi se moduo mogao negdje koristiti mora biti uvezen pomoću odgovarajuće *import* forme. Na Slici 6.6 je primjer korištenja modula.

//student.ts

export class Student{

firstName: string;

lastName: string;

constructor(firstName: string, lastName: lastName){

this.firstName = firstName;

this.lastName = lastName;

}

}

// home.ts

import {Student} from ‘./../student’;

let student = new Student(“Marko”, “Marković”);

*Slika 6.6 – Primjer kreiranja modula u TypeScript-u*

# Angular.io

## Uvod

Angular.io[[10]](#footnote-10) je platforma za razvoj klijentskih aplikacija pomoću HTML-a i *TypeScript*-a. Angular omogućava razvoj *web*, mobilnih i desktop aplikacija. Implementira osnovne i opcione funkcionalnosti kao skup *TypeScript* biblioteka koje se mogu uvesti (eng. *import*) u aplikaciju. [[24](#Ang19)]

Prva verzija koja je objavljena je Angular 1 i poznata je pod nazivom AngularJS. Ova verzija je praćena Angular 2 verzijom, a posljednja je Angular 7. Angular 2 verzija je donijela mnoge promjene u odnosu na prvu verziju. Prije svega arhitektura prve je *model-view-controller* bazirana, dok je od verzije 2 struktura bazirana na *components/services* arhitekturi.

Mnogi alati koji se koriste za razvoj Angular aplikacija se oslanjaju na Node.js. Zbog toga je za instalaciju Angular-a potrebno da bude instaliran Node.js. Nakon instaliranja Node.js-a, instalira se Angular CLI (eng. *Command Line Interface*) paket. Angular CLI paket je standardni način za kreiranje i upravljanje Angular projektima. Komanda za instaliranje Angular CLI paketa je prikazana na Slici 7.1.

npm install –g @angular/cli

*Slika 7.1 – Intaliranje Angular CLI pakta*

Za kreiranje novog Angular projekta pod nazivom *Project1* koristi se komanda prikazana na Slici 7.2.

ng new Project1

*Slika 7.2 - Primjer kreiranja Angular projekta*

Komandu *ng* obezbjeđuje Angular CLI. Instalacioni proces kreira folder *Project1* koji sadrži sve konfiguracione datoteke koje su potrebne za razvoj aplikacije i NPM pakete potrebne za razvoj, pokretanje i korištenje Angular aplikacije.

## Arhitektura

U ovom poglavlju definisani su osnovni elementi koji čine Angular aplikacije i njihove mogućnosti. Na Slici 7.3 je dijagram koji prikazuje kako su ti elementi povezani:



*Slika 7.3 – Dijagram Angular elemenata*

Komponente i *template*-i zajedno definišu ono što će biti prikazano u Angular aplikaciji. Dekoratori u klasi komponente dodaju metapodatke. Direktive i *binding* elementi modifikuju sadržaj pogleda na osnovu programskih podataka i logike. *Dependency injector* pruža usluge kao što je ruter servis koji omoćava navigaciju komponentama. U nastavku su elementi opisani detaljnije.

Moduli – sistem modula u Angular-u se naziva *NgModules*. Aplikacija uvijek ima najmanje jedan *root* (korjenski) modul, ali uobičajeno ima više funkcionalnih modula. *Root* modul se naziva *AppModule* i on obezbijeđuje mehanizam za pokretanje aplikacije. Moduli su veoma dobar način za organizaciju aplikacije i proširenje njenih mogućnosti pomoću drugih modula, kao što su *FormsModule*, *HttpClientModule* i *RouterModule*. Organizovanje koda u različite funkcionalne module pomaže u upravljanju razvojem složenih aplikacija i dizajniranja za ponovnu upotrebu. Pored toga, omogućava da se iskorisiti prednost *lazy* učitavanja (eng. *lazy-loading*). *Lazy-loading* omogućava da se prilikom pokretanja aplikacije sadržaj učitava dinamički na zahtjev korisnika.

Najvažnija svojstva kojima se opisuju moduli:

* *declarations* - klase koje pripadaju *NgModule*-u. Ove klase su komponente (eng. *components*), direktive (eng. *directives*) i *pipes*.
* *exports* - podskup deklaracija koje se trebaju vidjeti i koristiti u komponentama drugih *NgModules*-a.
* *imports* - moduli čije su izvezene klase (eng. *exported classes*) potrebne komponentama deklarisanim u ovom modulu.
* *providers* - kreatori servisa (eng. *services*) koje ovaj modul pruža u globalnoj kolekciji servisa.
* *bootstrap* - glavni pogled aplikacije, koja se naziva korjeska komponenta (eng. *root component*) i sadrži sve ostale poglede aplikacije. Samo *AppModue* smije imati ovo svojstvo.

Na Slici 7.4 je jednostavna definicija *root* *NgModule*-a:

// src/app/app.module.ts

import { NgModule } from ‘@angular/core’;

import { BrowserModule } from ‘@angular/platform-browser’;

@NgModule({

imports: [ BrowserModule ],

providers: [ Login ],

declarations: [ AppComponent ],

exports: [ AppComponent ],

bootstrap: [ AppComponent ]

})

export class AppModule {}

*Slika 7.4 – Definicija root modula*

*Root* modul - *AppModule* uvijek ima korjensku komponentu, u primjeru sa Slike 7.4 to je AppComponent koja se kreira podizanjem (eng. *bootstrapping*) modula. Svaki *NgModule* može uključiti više komponenti, koje se mogu učitati kroz ruter.

Angular biblioteke – kolekcija *JavaScript* modula koja čini Angular se posmatra kao biblioteka u drugim programskim jezicima. Ime svake Angular biblioteke ima prefiks *@angular*. Ukoliko potrebne biblioteke nema, može se instalirati pomoću NPM repozitorijuma.

Komponente – svaka Angular aplikacija ima najmanje jednu komponentu (*root component*) koja povezuje hijerarhiju komponenti sa DOM-om. Svaka komponenta definiše klasu koja sadrži logiku i podatke o aplikaciji, a povezana je sa HMTL *template*-om (datotekom) koji definiše ono što će biti prikazano. Dekorator *@Component* identifikuje klasu kao komponentu i definiše metapodatke specifične za komponentu. Klasa komunicira sa HMTL *template*-om pomoću svojstava i metoda. Najvažniji podaci koji opisuju komponentu su:

* *selector* - CSS selektor koji kaže Angular-u da kreira i ubaci odgovarajući tag u HTML stranicu.
* *templateUrl* - relativna putanja na kojoj se nalazi HTML datoteka (*template*) komponente.
* *providers* - lista provajdera (eng. *providers*) koje komponenta zahtijeva.

Na Slici 7.5 je prikazan jednostavan primjer klase komponente sa metapodacima.

// student-list.component.ts

import { Component } from ‘@angular/core’;

@Component({

selector: ‘app-student-list’,

templateUrl: ‘./student-list.component.html’,

providers: [ StudentService ]

})

Export class StudentListComponent implements OnInt {

/\* … \*/

}

*Slika 7.5 – Primjer klase za komponentu student-list*

Pogled (eng. *view*) komponente se definiše pomoću *template*-a. *Template* je oblika HTML dokumenta sa proširenom Angular sintaksom pomoću koje se sadržaj mijenja na osnovu logike komponente i stanja aplikacije.

*Data-binding* (vezanje podataka) - Angular podržava dvosmjerno vezanje podataka, mehanizam za koordinaciju dijelova *template*-a i dijelova komponente. U suprotnom, programer bi bio zadužen za ubacivanje vrijednosti podataka u HTML i pretvaranje korisničkih unosa u akcije i ažuriranje vrijednosti. U HTML *template* -u se dodaje oznaka za vezanje podataka kako bi Angular znao kako da poveže obe strane. Slijede primjeri ovih oznaka:

* {{value}}
* [property] = “value”
* (event) = “handler”
* [(ng-model)] = “property” – *two-way data binding.*

*Pipes* – Angular *pipes* obezbijeđuju deklaraciju *display-value* transformacije u HTMLdokumentu. Klasa koja ima *@Pipe* dekorator definiše funkciju koja transformiše ulaznu vrijednost u izlaznu kako bi se prikazala. Kako bi se specifikovala vrijednost transformacije u HTML-u, koristi se *pipe* operator | .

Direktive – Kada Angular prikazuje sadržaj *template*-a, transformiše se DOM prema uputstvima koje daju direktive. Direktive su klase sa *@Directive* dekoratorom. Postoje tri vrste direktiva: komponenta u Angular-u je spoj direktive i pogleda komponente. Pored komponenti, tu su i strukturalne i atributne. Strukturalne direktive mijenjaju izgled dodavanjem, uklanjanjem i zamijenom elemenata u DOM-u. Primjer dve ugrađene strukturalne direktive koje dodaju logiku aplikaciji:

\*ngFor – iterator kroz listu *students,*

\*ngIf – uslovna direktiva, uključuje *StudentList* komponentu ukoliko *selectedStudent* postoji.

Na Slici 7.6 je prikazana primjena ove dvije direktive.

// src/app/students-list.component.html

<li \*ngFor=”let student of students”></li>

<app-student-list \*ngIf=”selectedStudent”></app-student-list>

*Slika 7.6 – Primjer strukturalnih direktiva*

Atributne direktive mijenjaju izgled i ponašanje postojećeg elementa. Izgledaju kao regularni atribut, pa se zbog toga i nazivaju atributni. Direktiva *ngModel* implementira dvosmjerno vezivanje podataka. Najčešće se koristi za *<input>* element, tako što prikazuje vrijednost i odgovara na događaje kao što je promjena vrijednosti unutar *<input>* elementa. Na Slici 7.7 je primjer korištenja *ngModel* direktive.

<input [(ngModel)]=“student.name“>

*Slika 7.7 – Primjer atributne direktive*

Servisi – Servis (eng. *Service*) je klasa sa jasno definisanom svrhom koja treba da uradi nešto specifično. Cilj je da komponenta bude što jednostavnija i treba da predstavi svojstva i metode za vezivanje podataka koje bi posredovale između pogleda i logike aplikacije. Servisima se ostavljaju složeniji zadaci kao što su dohvatanje podataka sa servera i validacija korisničkog unosa. Angular razlikuje komponente i servise kako bi povećao modularnost i ponovnu upotrbu.

*Dependency injection* – se u Angularu koristi kako bi komponente mogle koristiti servise, odnosno funkcionalnosti koje servisi nude. Servis se može ubaciti u komponentu, dajući komponenti pristup toj klasi servisa. Da bi se klasa u Angularu definisala kao servis, koristi se dekorator *@Injectable*. Metapodaci obezbijeđuju Angularu da servis ubaci u komponentu kao zavisnost. Mehanizam za kreiranje zavisnosti je *injector*. *Injector* održava kontejner instanci zavisnosti, odnosno servisa, koje su ranije kreirane. Ukoliko traženi servis nije u kontejneru, *injector* kreira novu instancu i dodaje je u kontejner. Kada Angular kreira komponentu, prvo od *injector*-a saznaje koje servise ta komponenta zahtjeva. Kada su svi servisi kreirani, Angular poziva konstruktore komponenti sa traženim servisima kao argumentima. Ovo je način na koji se u Angularu implementira *dependency injection*.

Kako bi *injector* znao koji servisi su mu dostupni i kako ih kreirati, trebaju se registrovati provajderi (eng. *providers*) za svaki servis. Provajder može biti dio metapodataka servisa, čineći servis dostupnim svuda. Provajder se može registrovati u metapodacima modula *ngModule* ili neke komponente. Podrazumijevano, Angular CLI komanda *ng generate service*, registuje provajder sa *root injector*-om za servis, uključujući metapodatke provajdera u *Injectable* dekorator, što je prikazano na Slici 7.8.

// student.service.ts

import { Injectable } from ‘@angular/core’;

@Injectable({

providedIn: ‘root’

})

export class StudentService {}

*Slika 7.8 – Primjer registracije provajdera*

U ovom slučaju, Angular kreira jednu, dijeljenu instancu servisa i ubacuje je u klasu koja zahtijeva ovaj servis.

Kada se provajder registruje u modulu *NgModule*, jedna instanca servisa je dostupna svim komponentama koje su ovom modulu. Ovaj način registracije može se vidjeti na Slici 7.4. U slučaju da je provajder registrovan na nivou komponente, sa svakom novom instancom komponente, kreira se nova instanca servisa, što je prikazano na Slici 7.5.

*HttpClient* - Veliki broj klijentskih aplikacija koristi HTTP protokol za komunikaciju sa serverom. Moderni pretraživači podržavaju dva različita API-ja za kreiranje HTTP zahtjeva: *XMLHttpRequest* interfejs i *fetch()* API. *HttpClient* iz *@angular/common/http* biblioteke nudi pojednostavljen klijentski HTTP API za Angular aplikacije koji je zasnovan na *XMLHttpRequest* interfejsu podržan od strane pretraživača. Prednosti korištenja *HttpClient-*a su: mogućnost testiranja API-ja krajnjih tačaka, tipizirani objekti zahtjeva i odgovora i pojednostavljeno rukovanje greškama. Prije korištenja *HttpClient*, aplikacija mora uvesti modul *HttpClientModule*, koji se najčešće uvozi u *AppModule*, nakon čega se može ubaciti u željenu klasu.

Rutiranje - Angular ruter omogućava navigaciju od jednog do drugog pogleda dok korisnik koristi aplikaciju. Model navigacije pretraživača je sljedeći: unosi se URL u *address bar* gdje pretraživač otvara odgovarajuću stranicu, zatim klikom na *link* u stranici pretraživač prelazi na sljedeću odgovarajuću stranicu, dok se klikom na nazad ili naprijed, pretraživač kreće kroz istoriju stranica unazad ili naprijed. Angular ruter je sličan ovom modelu. Može interpretirati URL adresu *web* pretraživača kao instrukciju za navigaciju do određenog pogleda aplikacije, gdje komponenti ciljanog pogleda može proslijediti opcione parametre na osnovu kojih će komponenta odrediti koji se sadržaj treba prikazati u pogledu. Takođe, određeni pogled može preusmjeriti kao reakciju na korisničku aktivnost (klik na dugme, izbor stavke iz padajuće liste, itd.). Navigacija između pogleda se čuva u istoriji *web* pretraživača, tako da automatski rade tasteri pretraživača za naprijed i nazad.

# Ionic

## Uvod

*Ionic* je razvojni okvir, otvorenog koda za kreiranje mobilnih i *desktop* aplikacija visokih performansi koje koriste *web* tehnologije HTML, CSS i *JavaScript*. Fokusira se na interakciju korisnika i korisničkog interfejsa aplikacije. Jednostavno se integriše sa drugim bibliotekama i okvirima, kao što je *Angular* ili se može koristiti samostano. [[25](#Ion19)]

Prva verzija *Ionic*-a razvijena je pomoću *AngularJS*-a i *Apache Cordova*-e 2013. godine. U januaru 2019. godine izašla je posljednja *Ionic* 4 verzija, realizovana kao skup *web* komponenti. U posljednoj verziji može se izabrati jedan od integracionih okvira (*Angular*, *Vue* ili *React*) ili upotreba *Ionic* komponenti bez ikakvog okvira.

Instalacija *Ionic*-a realizuje se pomoću NPM repozitorijuma komandom prikazanom na Slici 8.1.

*Slika 8.1 – Instaliranje Ionic-a pomoću NPM repozitorijum*

npm install –g ionic

Prilikom kreiranja *Ionic* aplikacije može se koristiti jedan od predefinisanih šablona. Tri najčešće korištena šablona su *blank*, *tabs* i *sidemenu*. Na Slici 8.2 je prikazana komanda kojom se kreira *Ionic* projekat i instaliraju potrebne zavisnosti projekta.

*Slika 8.2 – Kreiranje Ionic projekta*

ionic start <name> <template> [options]

Prvi argument označava ime aplikacije, drugi argument predstavlja šablon, a treći argument su opcije, npr. opcijom *--type* se navodi tip *Ionic* projekta. Tipovi projekta mogu biti *ionic1*, *ionic-angular*, *angular* ili *custom*.

Konfiguracione vrijednosti se čuvaju u JSON formatu. Svaki *Ionic* projekat ima konfiguracioni fajl *ionic.config.json* u *root* direktorijumu. Na Slici 8.3 je primjer konfiguracionog fajla.

*Slika 8.3 – Primjer konfiguracionog fajla ionic.config.json*

{

“name”: “My App”,

“integrations”: {

“cordova”: {

…

}

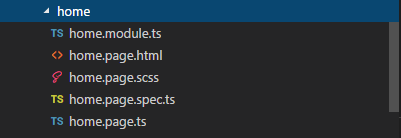
},

“type”: “angular”

}

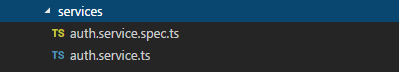
## Arhitektura

Osnovna struktura *Ionic* 4 aplikacije za *template* argument ima vrijednost *blank*, a za *type* *angular*. *Ionic* CLI omogućava kreiranje stranica, komponenti, direktiva i servisa. Svaku kreiranu stranicu čine sljedeći fajlovi: *page.html* – HTML *template* u kome se definiše šta će bit prikazano, *page.scss* – definiše izgled stranice, *page.ts* – klasa u kojoj se definiše logika, *module.ts* – predstavlja modul stranice i koristi se kada je potrebno učitati stranicu i *page.spec.ts* – *unit test* izvorne datoteke *page.ts*. *Ionic* 4 koristi *Angular Router* za navigaciju. Zbog toga, svaka kreirana stranica automatski se dodaje u app-routing.module.ts – modul za konfiguraciju ruta. Na Slici 8.4 je primjer strukture stranice *home*.



*Slika 8.4 – Primjer strukture stranice “home”*

Kao i u *Angular* aplikacijama, servisi se kreiraju kako bi se realizovala modularnost i ponovna upotreba. Na Slici 8.4 je primjer strukture servisa *auth*.



*Slika 8.5 – Primjer strukture servisa “auth”*

## *User Interface* komponente

Razvoj mobilnih aplikacija danas podrazumjeva da su one dostupne i na iOS i Android uređajima. Izazov koji se javlja je kako razviti korisnički interfejs za sve platforme tako da se zadrži isti izgled aplikacije. Iako postoje HTML komponente koje su na raspolaganju, kao što su dugmad, *input* polja, *sliders*, *radio* dugmad, oni nisu dovoljni za kreiranje korisničkog interfejsa modernih aplikacija. Zbog toga su razvijene moderne biblioteke UI (eng. *User Interface*) komponenti koje imaju brojne gradivne blokove za razvoj aplikacije.

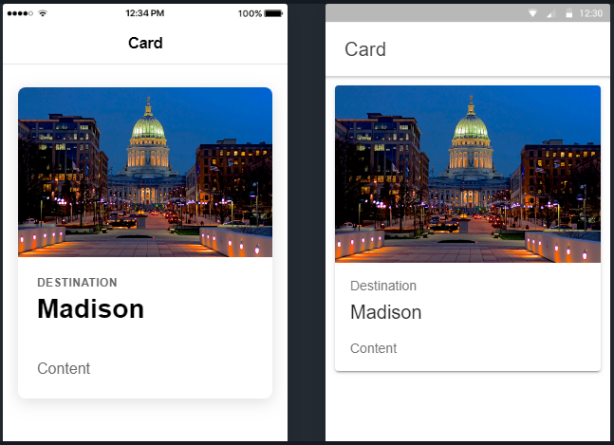
Komponente su konstruisane u dva dijela: kako izgledaju – UI i kako funkcionišu – UX (eng. *User Experience*). *Ionic* dolazi sa brojnim UI komponentama kao što su katrice (eng. *cards*), liste (eng. *lists*) i tabovi (eng. *tabs*).

Kartice su standardni dio korisničkog interfejsa i služe kao ulazna tačka za prikaz detalja. Kartica može biti jedina komponenta, ali se najčešće sastoji od zaglavlja, naslova, podnaslova i sadržaja, tako da tada *<ion-card>* čine druge podkomponene: *<ion-card-header>*, *<ion-card-title>*, *<ion-card-subtitle>* i *<ion-card-content>*.

Liste čine redovi stavki koje mogu da sadrže tekst, slike, dugmad i još mnogo toga. Liste obično sadrže stavke sa sličnim sadržajem podataka. *<ion-list>* komponenta može da ima zaglavlje *<ion-list-header>* i *virtual scroll* za prikazivanje virtuelne „beskonačne“ liste. Korištenjem *virtual scroll*-a za liste sa velikm brojem stavki povećavaju se performanse iz razloga što se u ovom slučaju ne učitavaju sve stavke odjednom, već samo skup stavki dovoljan za popunjavanje prikaza na ekranu.

Tabovi su *top-level* navigacione komponente za implementaciju navigacije zasnovane na tabovima. *<ion-tabs>* komponenta radi kao ruter za navigaciju. Ova komponenta ne pruža nikakvu povratnu spregu za prebacivanje između kartica, a da bi se to postiglo *<ion-tab-bar>* bi trebao biti obezbijeđen kao dio *<ion-tabs>*. *<ion-tab-button>* je komponenta koja određuje izgled ikone i labele taba i nalazi se u komponenti *<ion-tab-bar>*.

Na Slici 8.6 je primjer kartice sa slikom, naslovom i sadržajem na iOS i Android platformama.



*Slika 8.6 – Primjer izgleda komponente ion-card*

*Ionic* pored navedenih ima brojne komponente pomoću kojih se mogu kreirati moderne aplikacije koje će imati isti izgled i funkcionalnosti i na mobilnim i na *desktop* platformama.

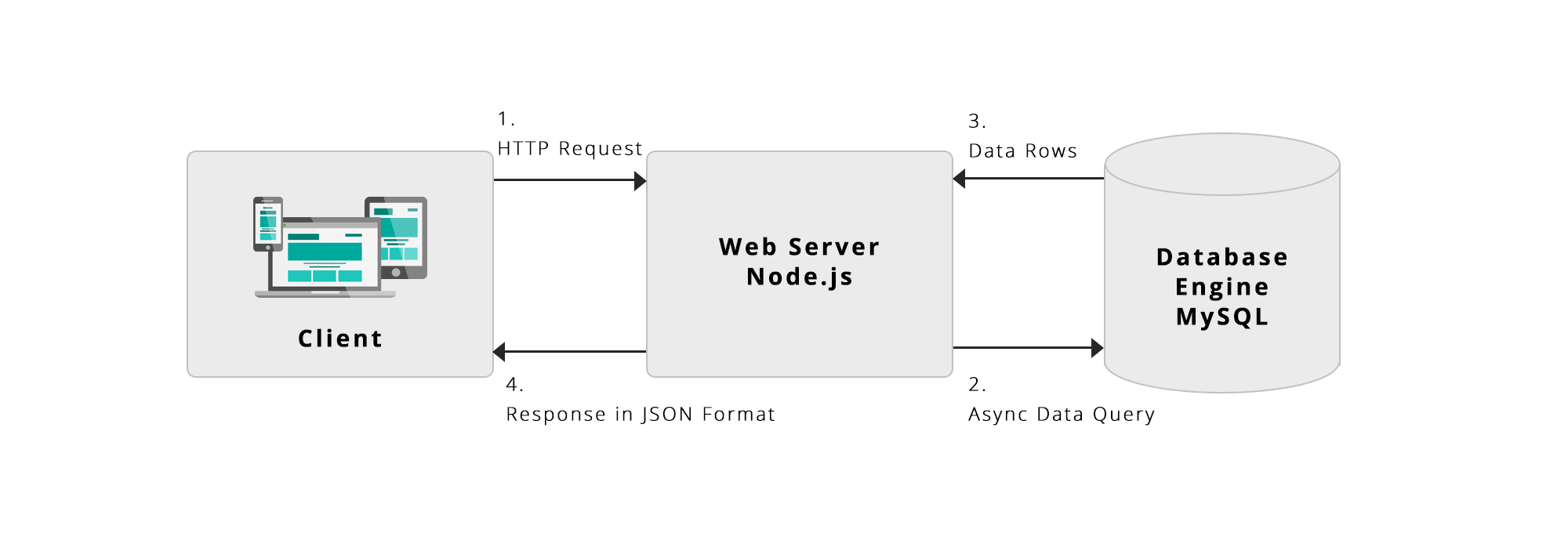
# OPIS IMPLEMENTACIJE APLIKACIJE ZA NARUČIVANJE HRANE

U praktičnom dijelu ovog diplomskog rada implementirane su *web* i mobilna aplikacije za naručivanje hrane. Aplikacije su realizovane korišćenjem tehnologija koje su opisane u prethodnim poglavljima. Klijentski dio *web* aplikacije je realizovan pomoću Angular-a, a mobilne pomoću Ionic-a. Serverski dio aplikacije realizovan je korištenjem Node.js platforme.

Korisnici aplikacija imaju mogućnost da pregledaju i naruče ponude iz različitih restorana. Nakon što se prijave na sistem, korisnici mogu pregledati njihove prethodne narudžbe ili kreirati novu, Izborom ponuda koje se „dodaju u korpu“ i poručuju unošenjem adrese na koju žele da im narudžba stigne. U ovom poglavlju su detaljno opisani i serverski i klijentski dio aplikacije.

Pored opisa implementacije aplikacije, izvršena je analiza performansi i skalabilnosti aplikacije čiji rezultati su u nastavku ovog dijela rada.

Arhitektura sistema je klijent-server bazirana. Serverski dio aplikacije baziran je na *RESTful* (eng. *REST* – *Representational State Transfer*) arhitekturi kojom se obezbjeđuje jednostavan pristup resursima i njihovoj modifikaciji od strane *REST* klijenata. Serverska aplikacija podatke dobavlja iz baze podataka koja je kreirana na MySQL serveru. Klijentski dio *web* aplikacije dostupan je putem *web* čitača, a mobilna aplikacija je dostupna na svim mobilnim platformama. Na Slici 9.1 je prikazana arhitektura sistema.



*Slika 9.1 – Arhitektura sistema*

## Serverski dio aplikacije

Serverska strana je realizovana pomoću Node.js-a i Express.js *framework*-a za Node.js. Kao što je rečeno, server je realizovan kao *RESTful* *web* servis i sva komunikacija se obavlja putem *HTTPS* (eng. *Hypertext Transfer Protocol Secure*) protokola. Na osnovu zahtjeva koji mogu biti GET, POST, PUT i DELETE, i na osnovu URI-ja (eng. *Uniform Resource Identifier*) kojim se definiše resurs iz zahtjeva, server vrši rutiranje. Rutiranjem se zapravo aktivira odgovarajuća *callback* funkcija iz odgovarajućeg modula, što je opisano u poglavlju 5.5. Kada se korisnik prijavi na sistem, server poziva funkciju *login* koja provjerava da li je korisnik registrovan, i ukoliko je registrovan, kao odgovor klijentu šalje JSON objekat sa kreiranim JWT[[11]](#footnote-11) (eng. *JSON Web Token*) tokenom za autorizaciju i svim podacima o korisniku. Ukoliko korisnik nije registrovan, server šalje obavještenje o grešci. Nakon uspješne prijave na sistem, kreirani JWT token se čuva na strani klijenta i kasnije šalje kroz zahtjeve za one URI-je resursa koji zahtjevaju zaštitu.

*JSON Web Token* je standard koji definiše kompaktan i samostalan način za sigurno prenošenje podataka. Token je *string* i može biti digitalno potpisan tajnim ključem. Na Slici 9.2 je prikazana funkcija *login* u kojoj se kreira *JWT* token.

//user.controller.js

var db = require(./../node\_modules/datebase/userDB);

const jwt = require(‘jsonwebtoken’);

function login(req, res){

let username = req.body.username.trim();

let password = req.body.password.trim();

db.loginUser(username, password, function(user){

if(user.iduser !== undefined){

var token = jwt.sign({userID: user.iduser}, ‘secretKey’, {expiresIn: ‘900000’});

var toSend = {token: token, user: user};

res.send(toSend);

}

else{

res.send({error: ‘Invalid login’});

}

})

}

*Slike 9.2 – Kreiranje JWT tokena*

Nakon što se u funkciji *login* izvrši provjera da li je korisnik prijavljen na sistem, pozivom funkcije *loginUser* – koja komunicaira sa bazom podataka, kreira se token. Token se kreira tako što se prvo učita *jsonwebtoken* modul što je prikazano na Slici 9.2. Modul je prethodno instaliran pomoću NPM repozitorijuma. Pozivom funkcije *jwt.sign* kojoj se prosljeđuju id korisnika, tajni ključ i vrijeme u milisekundima za koje token prestaje da važi, kreira se token. Token se sada može poslati na klijentsku stranu gdje se čuva u *sessionStorage*-u kako bi se kasnije mogao slati kroz zahtjeve (poželjno u zaglavlju) da bi se izvršila autorizacija.

Kada server dobije zahtjev sa putanjom koja zahtijeva autorizaciju, prvo će izvršiti provjeru da li je token iz zahtjeva definisan i ukoliko jeste izvršiće i verifikaciju tokena. Ukoliko token nije definisan u zaglavlju zahtjeva, server vraća poruku o grešci, u suprotnom token i tajni ključ se prosljeđuju funkciji *jwt.verify* koja će izvršiti verifikaciju. Nakon uspješne verifikcije, korisnik može pristupiti resursu koje zahtijeva. U slučaju da ova funkcija vrati poruku o grešci, server će tu poruku proslijediti korisniku. Na Slici 9.3 je prikazana funkcija *createOrder* u kojoj se vrši verifikacija tokena.

*Slika 9.3 – Verifikacija JWT tokena*

//

app.post(‘/orders/order’, checkToken, createOrder);

// checkToken = (req, res, next) => {

const header = req.headers['authorization'];

if(header !== undefined){

const bearer = header.split(' ');

const token = bearer[1];

req.token = token;

next();

}

else{

res.send(error: 'Token is undefined');

}

}

//

function creteOrder(req, res){

jwt.verify(req.token, 'secretKey', function(err, authData){

if(err){

res.send(error: 'Token error');

}

else{

//

}

})

}

JWT token se sa klijentske strane uobičajno šalje u *Authorization* zaglavlju pomoću *Bearer* šablona. Sadržaj zaglavlja treba da izgleda na sljedeći način: Authorization: Bearer <token>.

Za povezivanje sa bazom iz Node.js aplikacije koristi se *mysql* modul koji implementira metode za rad sa bazom. Za potrebe povezivanja sa bazom podataka, kreiran je modul *mysqlConnection* u kome se implementira *connection pool* i funkcija *getConnection* koja se *export*-uje. *Connection pool* mehanizmom se veze ka bazi održavaju aktivnim, tako da kada se traži veza, koristi se jedna od aktivnih, umjesto da se kreiraju druge. Funkcija *getConnection* uzima jednu slobodnu vezu ka bazi. Modul *mysqlConnection* je prikazan na Slici 9.4.

*Slika 9.4 – Kreiranje modula mysqlConnection*

// mysqlConnection.js

// učitavanje mysql modula

var mysql = requite(‘mysql’);

var pool = mysql.createPool({

connectionLimit: 10;

host: “localhost”;

port: “3300”;

user: “root”;

password: “”;

database: “project”

});

exports.getConnection = function(callback){

pool.getConnection(function(err, connection){

callback(err, connection);

});

};

Kako bi organizacija koda bila što bolja, za svaki *API* (*user*, *restaurant*, *offer*) sa metodama za pristup bazi, kreiran je zaseban modul. Kao primjer biće naveden modul *userDB* i jedna od funkcija *loginUser* koje se *export*-uju. Sve druge funkcije realizovane su na sličan način. Na Slici 9.5 je prikazana funkcija *loginUser* u kojoj se aplikacija povezuje sa bazom radi provjere da li je korisnik prijavljen na sistem.

*Slika 9.5 – Slanje upita na bazu u funkciji “loginUser”*

//userDB.js

//

var mysql = require(‘./mysqlConnection’);

const User = require(‘./../../model/user’);

exports.loginUser = function(username, password, cb){

mysql.getConnection(function(err, connection){

if(err) cb(err);

else{

let sql = ‘SELECT \* FROM user WHERE username = ? AND password = ?’;

let values = [username, password];

connection.query(sql, values, function(err, rows){

if(err) cb(err);

else{

connection.release();

if(rows[0] !== undefined){

user = new User(rows[0].iduser, rows[0].username, rows[0].first\_name, rows[0].last\_name, rows[0].phone, rows[0].email, rows[0].role);

cb(user);

}

else { cb(new Error(‘Username or Password is incorrect’)); }

}

});

}

});

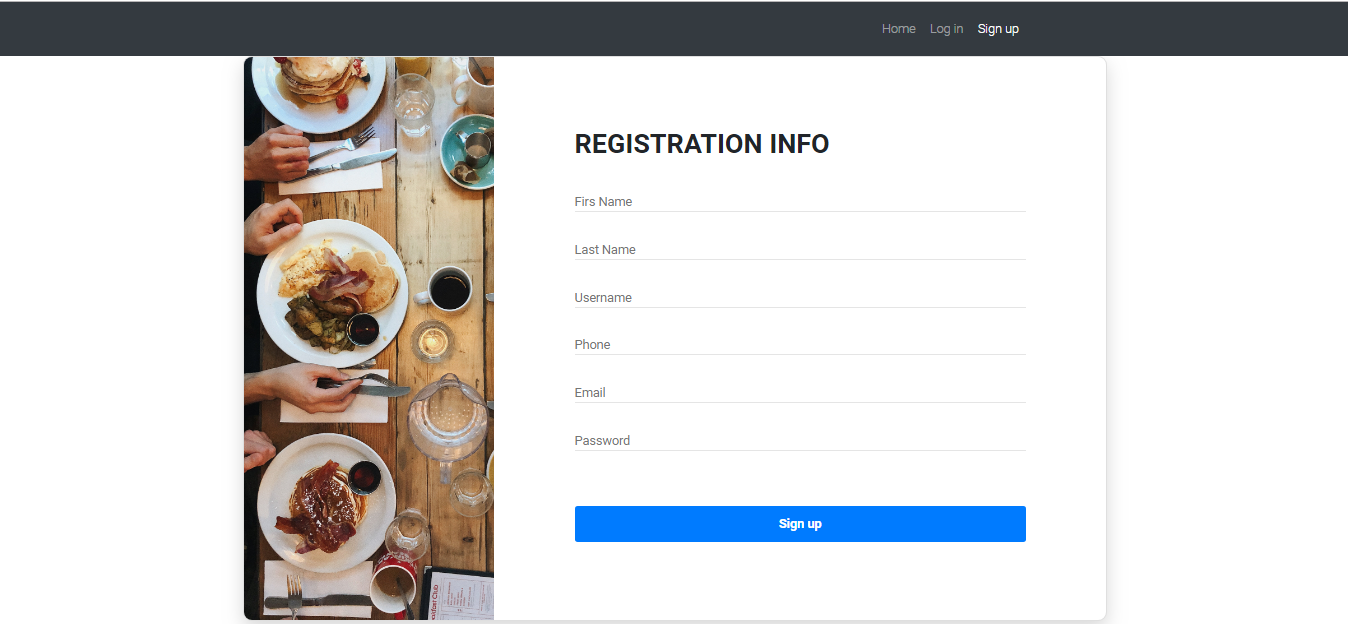
}

Kao parametri funkcije *loginUser* proslijeđeni su *username* i *password* korisnika, na osnovu kojih se kreira *sql* upit da li u bazi postoji korisnik sa ovim vrijednostima. Ukoliko korisnik postoji u bazi, kreira se objekat *user* sa svim atributima dobijenim iz baze podataka.

## Klijentski dio aplikacije

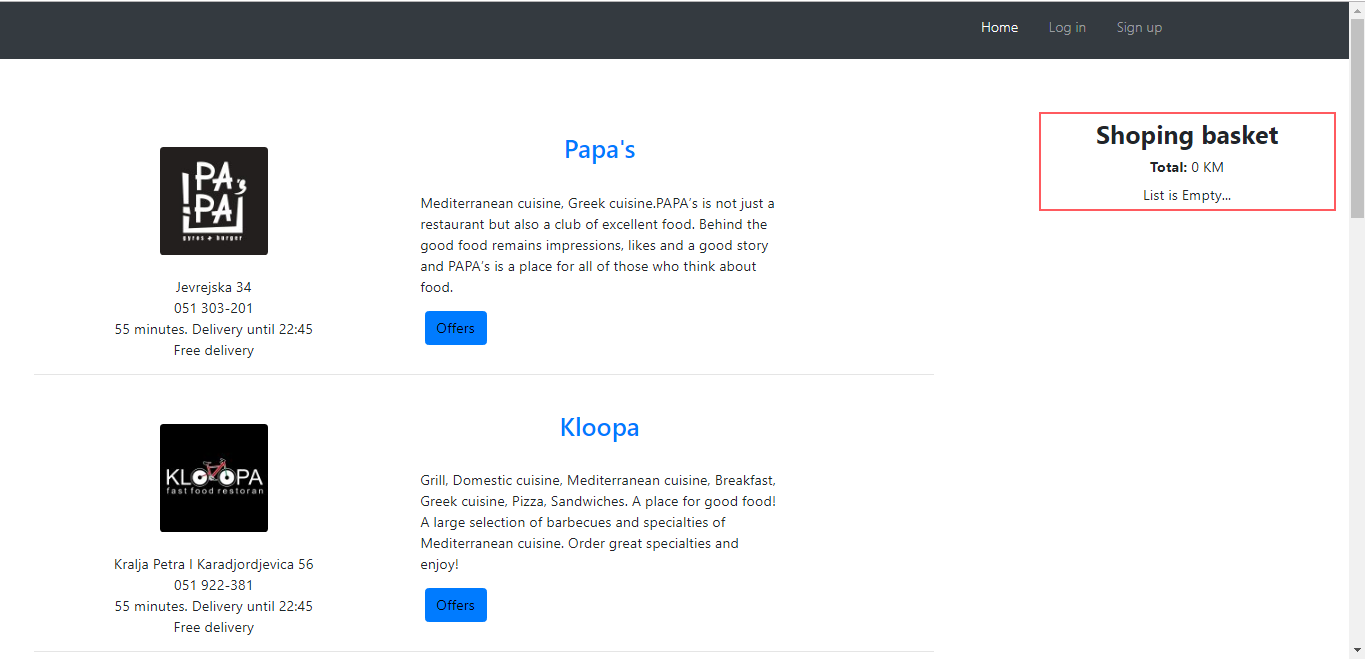
Za realizaciju klijentskog dijela *web* aplikacije korišten je Angular verzija 6. Svi potrebni resursi na klijentskoj strani se preuzimaju sa prethodno opisanog servera.

Kada se aplikacija pokrene, prvo se prikazuje *home* stranica kojoj imaju pristup svi korisnici. Na *home* stranici je prikazana lista restorana sa osnovnim informacijama. Ukoliko korisnik želi pogledati ponude i naručiti ih, potrebno je da bude prijavljen na sistem. Korisniku koji nije prijavljen, a želi pogledati ponude nekog od restorana, prikazuje se poruka o tome da je potrebno da se prvo prijavi na sistem. Da bi se korisnici mogli prijaviti na sistem treba da budu registrovani. Korisnici se mogu registrovati izborom navigacione veze *Sign up* iz linije za navigaciju, kojom se otvara stranica za registraciju koja je prikazana na Slici 9.6.



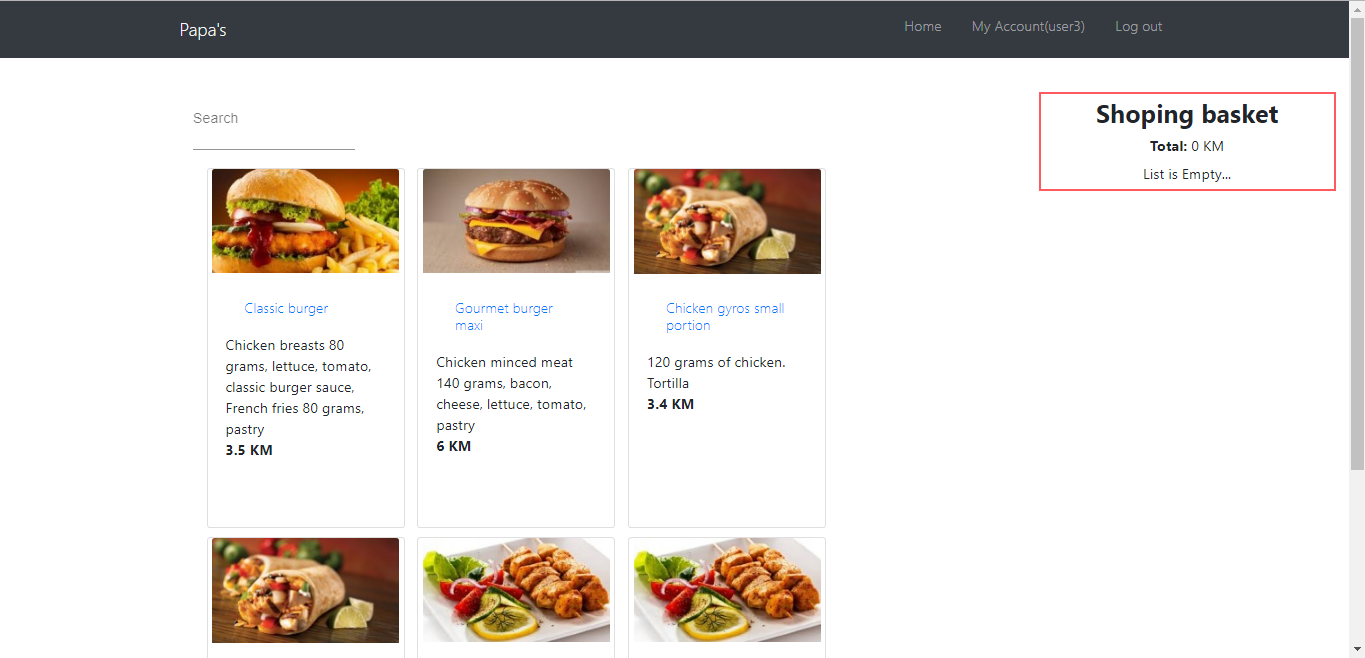
*Slika 9.6 – Izgled stranice za registraciju korisnika*

Kako bi se registrovao potrebno je da unese sva polja koja se nalaze u formi za registraciju. Ukoliko se korisnik pokuša registrovati sa *email*-om ili brojem telefona koji nije u skladu sa definisanim šablonom, dobiće poruku o grešci. Slično, ukoliko se pokuša registrovati sa korisničkim imenom koje već koristi neki drugi korisnik dobiće poruku da to korisničko ime već postoji u sistemu. Po uspješnoj registraciji, korisnik dobija poruku o tome i sada se može prijaviti na sistem. Izgled *home* stranice prikazan je na Slici 9.7.



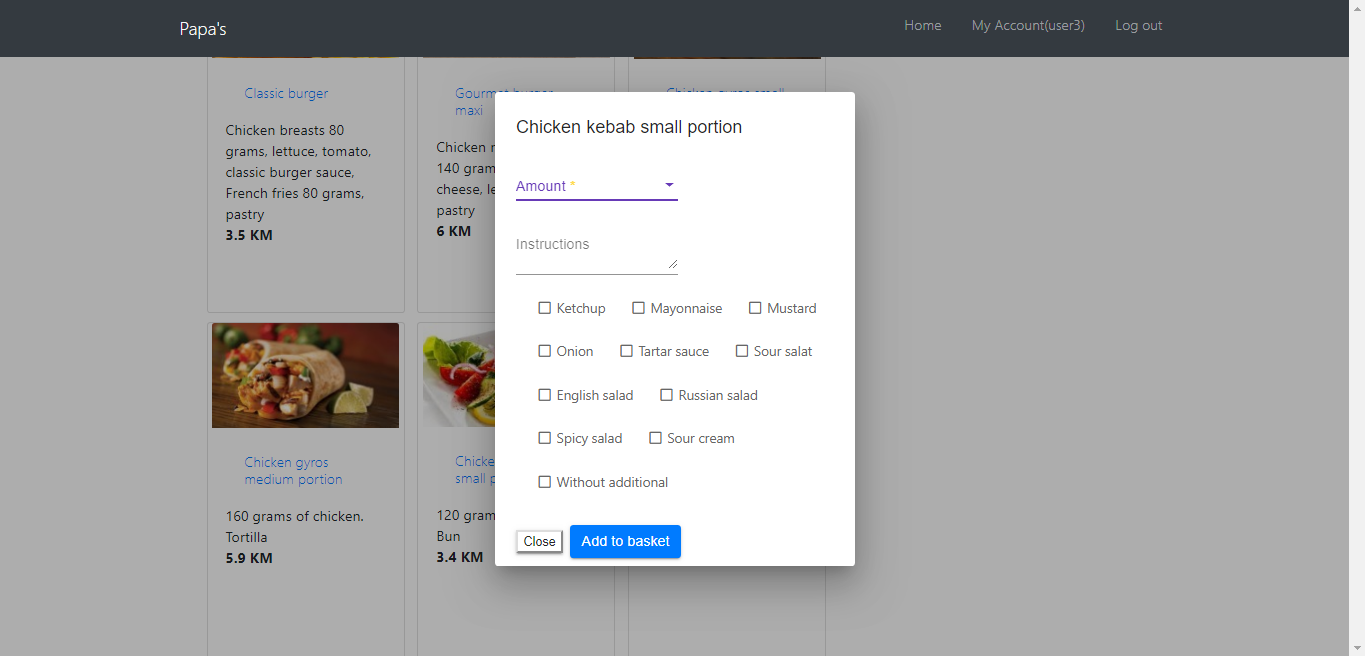
*Slika 9.7 – Izgled poćetne stranice*

Kada se korisnik prijavi na sistem, otvara mu se početna stranica *home*, sada sa svim mogućnostima i *sidebar* komponentom koja predstavlja „korpu“. Server prilikom prijave korisnika kreira JWT token, na način opisan ranije, a klijent ga čuva u *sessionStorage*-u i šalje kroz zaglavlje svih zahtjeva koji su zaštićeni. Korisnik na *home* stranici može pregledati sve ponude restorana i po želji izabrati neke i dodati ih u „korpu“ kako bi ih mogao naručiti. Liste restorana i ponuda se prikazuju pomoću Angular direktive *\*ngFor*. Izgled stranice sa listom ponuda prikazan je na Slici 9.8.



*Slika 9.8 – Izgled stranice sa ponudama restorana*

Korisnik može vršiti pretragu ponuda unošenjem ključne/ključnih riječi u polje *Search*. Po izboru neke od stavki otvara se *dialog box* u kome korisnik može izabrati koliko porcija izabrane stavke želi, da unese instrukcije i izabere priloge, nakon čega stavku može dodati u korpu. Izgled *dialog box*-a prikazan je na Slici 9.9:



*Slika 9.9 – Izgled dialog-box-a*

Sve izabrane stavke nalaze se u *sidebar* korpi tako da korisnik u svakom trenutku može da vidi šta je dodao, da poveća broj porcija stavke ili je ukloni iz korpe i na kraju unese adresu gdje želi dostavu i pošalje narudžbu. Prilikom kreiranja zahtjeva za narudžu, klijent treba da pošalje sačuvani token iz *sessionStorage*-a na serversku stranu kako bi se izvršila potrebna verifikacija. U servis klasi *RestaurantService* je prvo potrebno učitati *HttpClient* i *HttpHeaders* API-je iz *@angular/common/http* biblioteke. Prilikom kreiranja konstruktora ove klase, kreira se *private* varijabla *httpClient* tipa *HttpClient*. U metodi *createOrder*, kreira se novo *http* zaglavlje sa autorizacionim tokenom i poziva metoda *post httpClient* objekta kojoj se prosljeđuje URL zahtjeva za kreiranje narudžbe. Na Slici 9.10 je prikazan kod kojim se realizuje ovaj zahtjev.

//restaurant.service.ts

import { HttpClient, HttpHeaders } from ‘@angular/common/http’;

export class RestaurantService{

private headers = new HttpHeaders();

constructor(private httpClient: HttpClient){ }

public createOrder(order){

const token = sessionStorage.get(‘token’);

this.headers = new HttpHeaders({‘authorization’: ‘Bearer’ + token});

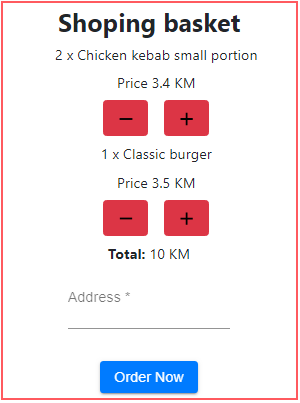
return this.httpClient.post(‘https://localhost:5000/restaurants/order’, order, {headers: this.headers});

}

}

*Slika 9.10 – Kreiranje “post” zahtjeva za narudžu*

Za svaku stavku koja se nalazi u korpi postoji mogućnost da korisnik klikom na dugme sa simbolom + doda još jednu porciju ili na dugme sa simbolom – umanji broj porcija. Na Slici 9.11 je prikazan izgled korpe sa izabranim stavkama koje korisnik želi naručiti.



*Slika 9.11 – Izgled “korpe” sa izabranim stavkama*

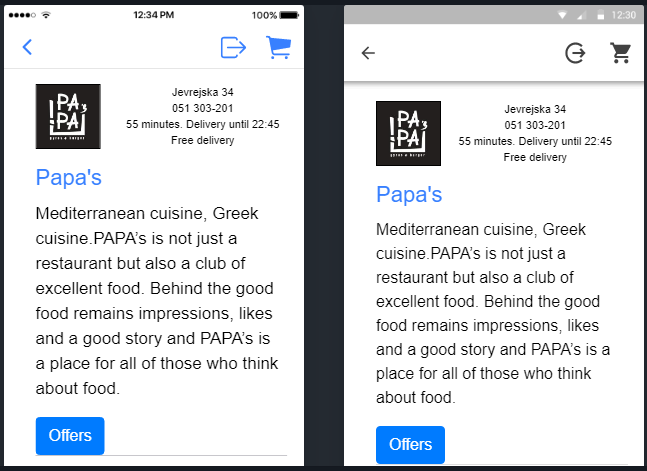
Po slanju narudže korisnik će dobiti poruku da li je bila uspješna ili ne. Korisnik sada može ponovo birati stavke i dodavati ih u korpu.

Pored navedenih, korisniku je omogućena funkcionalnost izmjene svog korisničkog profila i pregled istorije narudži koje je kreirao. Prilikom pregleda istorije narudžbi korisnik može poručiti već kreiranu narudžbu.

Za realizaciju klijentskog dijela mobilne aplikacije korišten je Ionic verzija 4. Prilikom kreiranja projekta izabran je *blank* šablon i tip projekta *angular.*

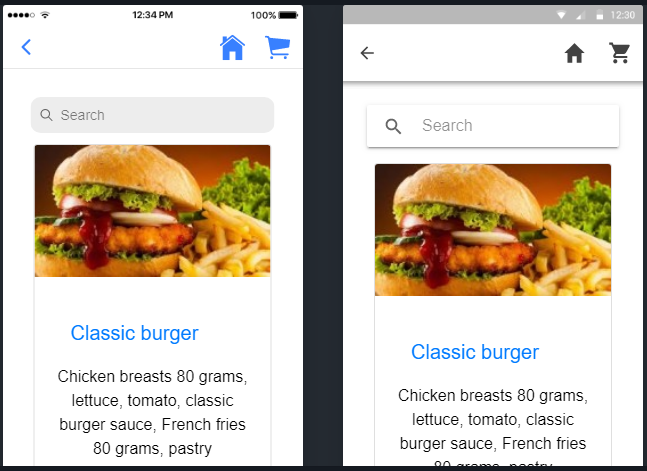
Kada klijent pokrene mobilnu aplikaciju prikazuje mu se *login* forma. Po uspješnoj prijavi otvara se stranica *restaurants* se listom restorana i njihovim osnovnim informacijama, slikom i opisom. Kao i kod *web* aplikacije, potrebni resursi se preuzimaju sa opisanog servera. Priliko prijavljivanja na sistem, server kreira i šalje JWT token klijentu gdje se token čuva u *sessionStorage*-u i kasnije šalje kroz zahtjeve.

Korisnik ima mogućnost pregleda i izbora ponuda restorana. U liniji za navigaciju na stranicama *restaurants* i *offers* nalaze se tabovi za odjavu sa sistema (eng. *logout*) i za „korpu“ sa izabranim stavkama. Na Slici 9.12 je prikazan izgled stranice *restaurants* za *iOS* i *Android* platforme.



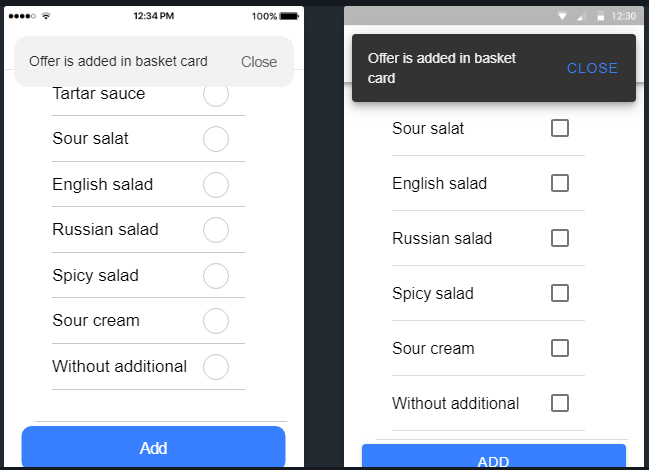
*Slika 9.12 – Izgled stranice “restaurants” na iOS i Android platformi*

Po izboru jednog od restorana, korisniku se otvara stranica *offers* na kojoj se nalaze stavke restorana. Kada korisnik izabere stavku otvara se modalni dijalog u kome korisnik može izabrati koliko porcija izabrane stavke želi, da unese instrukcije i izabere priloge, nakon čega stavku može dodati u „korpu“. Nakon dodavanja stavke, korisnik dobija povratnu informaciju u obliku *toast* komponente koja se pojavljuje na vrhu ekrana. Na Slici 9.13 je prikazana stranica *offers*.



*Slika 9.13 -* *Izgled stranice “offers” na iOS i android platformi*

Izgled modalnog dialoga kada se stavka doda u korpu prikazan je na Slici 9.14.



*Slika 9.14 -* *Izgled modalnog dijaloga na iOS i Android platformi*

Korisnik u svakom trenutku može pogledati šta je izabrao klikom na ikonu koja označava „korpu“ u zaglavlju aplikacije. Tada se otvara stranica sa izabranim stavkama i forma za slanje narudžbe. Kada se narudža pošalje, slično kao kod dodavanja stavke, prikazuje se poruka na vrhu ekrana.

## Analiza performansi i skalabilnosti

Koje vrijeme odziva *web* aplikacije je dovoljno brzo predstavlja pitanje kojim se programeri odavno bave. U posljednjih nekoliko godina vrijeme odziva je više nego bitno. Korisnici uobičajeno očekuju učitavanje stranice u istom trenutku kada i pošalju zahtjev za njom. Ne postoji neki univerzalni standard koji opisuje dovoljno dobro vrijeme odziva ali postoji dosta istraživanja i članaka na tu temu.

U ovom dijelu rada, opisane su performanse aplikacije za naručivanje hrane. Vrijeme otvaranja aplikacije je izmjereno u različitim *web* čitačima, izraženo je u sekundama i vrijednosti su upoređene. Mjerenja su prikazana na Slici 9.15. Takođe, izmjerena je veličina aplikacije i količina podataka koja se prenosi preko mreže.

*Slika 9.15 – Dijagram mjerenja vremena potrebnog za otvaranje aplikacije*

Kao što je prikazano na dijagramu, izvršena su po četiri mjerenja za svaki *web* čitač i može se vidjeti da je vrijeme otvaranja aplikacije najkraće u *Firefox* čitaču. Vrijeme otvaranja aplikacije zavisi od mnogih faktora, kao što su kompleksnost klijentske aplikacije, serverske aplikacije, baze podataka. Postoje različiti načini kojima se vrijeme učitavanja može smanjiti, jedan od njih je *Lazy loading*, spomenut u poglavlju 7.2. *Lazy loading* je mehanizam kojim se umjesto učitavanja čitave aplikacije, učitavaju samo moduli zahtijevani u nekom trenutku. Kako bi se ovaj mehanizam implementirao, potrebni su *routing.module.ts* i *module.ts* fajlovi za komponentu koja će se učitavati na zahtjev. Npr. za komponentu *offer*, fajl *offer.module.ts* je prikazan na Slici 9.16:

// offer.module.ts

import { NgModule } from ‘@angular/core’;

import { CommonModule } from ‘@angular/common;

import { OfferRoutingModule } from ‘./offer-routing.module’;

import { NgModule } from ‘@angular/core’;

import { OfferComponent } from ‘./offer.component’;

@NgModule({

imports: [

CommonModule,

OfferRoutingModule

],

declarations: [OfferComponent]

})

export class OfferModule{}

*Slika 9.16 – Primjer module.ts fajla*

Ovi fajlovi se kreiraju komandom *ng generate module offer–routing*, pri čemu se *OfferRoutingModule* učitava u *OfferModule*. Na Slici 9.17 je prikazan moduo *offer-routing*.

// offer-routing.module.ts

import { NgModule } from ‘@angular/core’;

import { Routes, RouterModule } from ‘@angular/route;

import { OfferComponent } from ‘./offer.component’;

const routes: Routes = [

{

path: ‘’,

component: OfferComponent

}

];

@NgModule({

imports: [RouterModule.forChild(routes)],

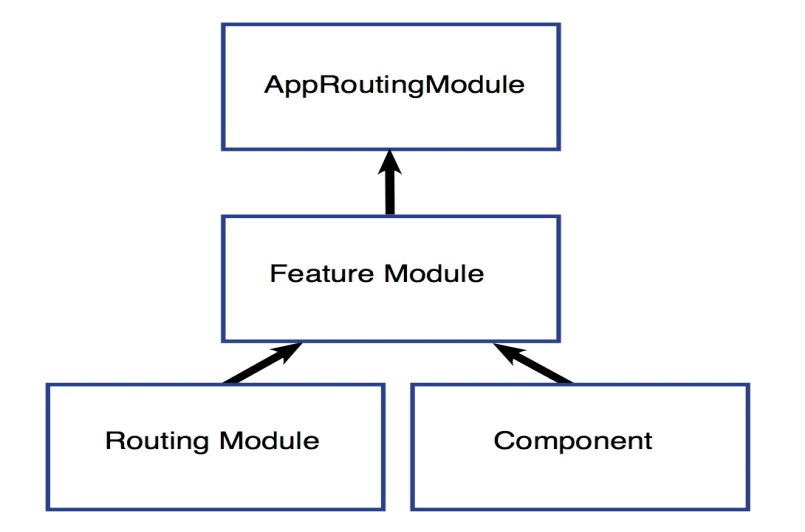
exports: [RouterModule]

})

export class OfferRoutingModule {}

*Slika 9.17 – Primjer routing-module.ts fajla*

Moduo *OfferRoutingModule* će obraditi svako usmjeravanje koje je u vezi sa komponentom *Offer*. Ovim se održava organizovana struktura aplikacije kako ona raste i omogućava ponovno korištenje modula, a da se njegovo usmjeravanje ne mijenja. Moduo *OfferModule* mora biti povezan sa *AppRoutingModule* modulom, tako da ruter zna za komponetu. Struktura je prikazana na Slici 9.18:



*Slika 9.18 – Struktura povezanosti modula kod “Lazy loading” mehanizma*

*Feature Module* u ovom slučaju predstavlja *OfferModule*. U *AppRoutingModule* se konfiguriše ruta do *OfferModule* modula koji ga sada povezuje sa *OfferRoutingModule* i na taj način ruter će znati gdje se treba učitati odgovarajuća komponenta. Na Slici 9.19 je prikazan modul *AppRoutingModule*.

//app-routing.module.ts

const routes: Routes = [

{

path: ‘offer’,

loadChildren: ./offer/offer.module#OfferModule

}

];

*Slika 9.19 – Primjer modula AppRouting*

*Lazy loading* mehanizam je korišten pri razvoju praktičnog dijela rada. Pored toga i slike se učitavaju na sličan način. Kako bi se smanjilo vrijeme odziva aplikacije, prilikom prikazivanja slika na stranici, učitavaju se samo one slike koje korisnik može odmah vidjeti, bez skrolovanja (eng. *screen roll*). Kako korisnik dalje prolazi kroz stranicu, učitavaju se slike koje trebaju biti u okviru za prikaz.

Izmjerena veličina klijentske aplikacije pri učitavanju u pretraživaču je 1.7MB, a količina prenešenih podataka preko mreže je 4.1KB.

Aplikacija je testirana u različitim web čitačima i u svim funkcioniše na isti način.

Aplikaciju je moguće proširiti dodatnim funkcionalnostima. Jedan od mogućih zahtjeva kojim se sistem može proširiti je da korisnici prije početka pretraživanja izaberu grad i lokaciju dijela grada. Tada će moći pretražiti i naručiti ponude restorana koji se nalaze u tom dijelu grada. Ovim bi se moglo smanjiti vrijeme čekanja narudžbe.

Korisnici ovog sistema narudžbu plaćaju pouezećem. Aplikacija se može nadograditi tako što se plaćanje može izvršiti *online*. Korisnici bi mogli birati na koji način žele izvršiti plaćanje.

Upoređujući ovu aplikaciju sa nekim postojećim, prednost koju ona ima je mogućnost kreiranja narudže sa stavkama iz različitih restorana, dok se kod drugih trebaju kreirati posebne narudžbe za različite restorane.

Aplikacija se može dalje nastaviti razvijati i nadograđivati na osnovu već kreiranih modula. Na ovaj način bi se unaprijedile funkcionalnosti aplikacije.

# Zaključak

Upotreba Interneta postala je neizbiježan dio našeg života, kako poslovnog tako i privatnog. Posljedica ovoga je da se sve više razvijaju *web* i mobilne aplikacije koje se pokreću u okviru *web* čitača ili na mobilnim uređajima. Na ovaj način se eliminiše potrebno vrijeme i složenost instaliranja aplikacije na korisničkom uređaju. Pored toga, za izvršavanje *web* aplikacija korisnicima nije potreban uređaj sa izuzetno dobrom hardverskom konfiguracijom, već je dovoljan uređaj sa *web* čitačem. Većina modernih čitača podržavaju tehnologije za izvršavanje *web* i mobilnih aplikacija, pa su one široko dostupne.

Najznačajnija tehnologija za razvoj *web* baziranih aplikacija je *JavaScript* koji je u osnovi bio tema ovog diplomskog rada. *JavaScript* omogućava da aplikacije budu interaktivne, a ne samo stranice za prikaz statičkih podataka. Ranije se sva logika aplikacije smiještala na serversku stranu, dok su moderne *web* aplikacije klijentski orijentisane, tako što se logika aplikacije implementira u okviru čitača koristeći upravo *JavaScript*. Na ovaj način se smanjuje opterećenje servera.

Cilj praktičnog dijela rada je da se demonstrira razvoj brzih i skalabilnih *web* aplikacija korištenjem opisanih tehnologija. Angular kao dio *JavaScript*-a, trenutno je jedna od najpopularnijih platformi za razvoj *web* aplikacija. Angular aplikacije se kreiraju pomoću *TypeScript*-a koji je proširenje *JavaScript*-a. Pored svih prednosti *JavaScript*-a, *TypeScript* ima nove funkcionalnosti koje su opisanje u poglavlju 6. Pored toga, *TypeScript* prevodilac prijavljuje greške u toku pisanaja programa. Na taj način se izbjegavaju potencijalne greške u toku izvršavanja programa i kreira se jasniji i čitljiviji kod, iako je nedostatak prevođenja *TypeScript* datoteteka sporije pokretanje programa. Angular aplikacije su organizovane u module što Angular čini modularnim. Još jedna prednost Angulara je ponovna upotreba (eng. *reusability*). U Angularu je svaka kreirana komponenta samostalna i može se koristiti u različitim dijelovima aplikacije. Komponente se mogu lako izmijeniti ili zamijeniti boljom implementacijom, što Angluar aplikacije čini održivim (eng. *maintainability*).

Ionic okvir omogućava kreiranje visoko interaktivnih mobilnih aplikacija koje se mogu koristiti na različitim pametnim telefonima i čitačima. Upotrebom UI komponenti skraćuje se vrijeme razvoja aplikacija koje se izvršavaju na različitim platformama. Ionic je *open-source framework* koji za razvoj aplikacija koristi *web* tehnologije (*JavaScript*, HTML5 i CSS), pa *web* programeri mogu lako i brzo da prave i mobilne aplikacije. Troškovi razvoja Ionic aplikacija su manji u poređenu sa razvojem nativnih aplikacija. Nedostatak je što Ionic aplikacije imaju slabije performase od nativnih, mada je u većini slučaja razlika u efikasnosti zanemariva.

Dobre strane *JavaScript*-a koriste se i za razvoj serverske strane aplikacije. Node.js koristi *event-driven* mehanizam koji ne blokira I/O operacije i čini server veoma skalabilnim. To je jedna od ključnih prednosti Node.js. Node.js je izgrađen na *Chrome*-ovoj V8 *JavaScript* mašini (eng. *Chrome’s V8 JavaScript engine[[12]](#footnote-12)*), pa je veoma brz u izvršavanju koda. Aplikacije napisane u Node.js-u se brzo izvršavaju i imaju visoku skalabilnost.

Sve dobre osobine opisanih tehnologija doprinijele su da realizovane aplikacije za naručivanje hrane budu funkcionalne, brze, skalabilne, jednostavne i intuitivne za korištenje. Postoje razne mogućnosti njene nadogradnje koristeći već realizovane module, što je velika prednost opisanih tehnologija.

Popularnost *web* aplikacija u stalnom je porastu i sve veći dio ljudske populacije se oslanja na nove tehnologije u svakodnevnom životu i radu. U početku su se *web* aplikacije razvijale za potrebe marketinga, a sada zbog svojih prednosti, sve više za poslovne svrhe. Samim tim i tehnologije koje su opisane u ovom radu za realizaciju *web* aplikacija će se sve više koristiti i unapređivati.

x

# Literatura

|  |  |
| --- | --- |
| [1] | Introduction to HTML. [Na mreži], <https://developer.mozilla.org/en-US/docs/Learn/HTML/Introduction_to_HTML>, Posjećeno: 2019, Februar |
| [2] | E. Tittel, J. Noble, *HTML, XHTML and CSS for Dummies*.: Wiley Publishing Inc., Indianapolis 2011. |
| [3] | B. Öggl K. Förster, *HTML5 Guidelines for Developers*.: Pearson Education, Inc. 2011. |
| [4] | Geolocation API. [Na mreži], <https://developer.mozilla.org/en-US/docs/Web/API/Geolocation_API>, Posjećeno: 2019, Februar |
| [5] | B. Albers, F. Salim P. Lubbers, *Pro HTML5 Programming*.: Paul Manning, New York, Apress, 2011. |
| [6] | MDN web docs. [Na mreži], <https://developer.mozilla.org/en-US/docs/Web/API/Web_Workers_API/Using_web_workers>, Posjećeno: 2019, Februar |
| [7] | MDN web docs. [Na mreži], <https://developer.mozilla.org/en-US/docs/Web/HTML/Global_attributes>, Posjećeno: 2019, Februar |
| [8] | MDN web docs. [Na mreži], <https://developer.mozilla.org/en-US/docs/Web/HTML/Microdata>, Posjećeno: 2019, Februar |
| [9] | W3C Schools. [Na mreži], <https://www.w3schools.com/css/>, Posjećeno: 2019, Februar |
| [10] | MDN web docs. [Na mreži], <https://developer.mozilla.org/en-US/docs/Web/CSS/CSS3>, Posjećeno: 2019, Februar |
| [11] | MDN web docs. [Na mreži], <https://developer.mozilla.org/en-US/docs/Web/JavaScript/About_JavaScript>, Posjećeno: 2019, Februar |
| [12] | D. Flanagan, *JavaScript: The Definitive Guide*.: O’Reilly Media, Sebastopol, 2011. |
| [13] | MDN web dosc. [Na mreži], <https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Functions>, Posjećeno: 2019, Februar |
| [14] | W3C Schools. [Na mreži], <https://www.w3schools.com/js/js_events.asp>, Posjećeno: 2019, Februar |
| [15] | MDN web docs. [Na mreži], <https://developer.mozilla.org/en-US/docs/Glossary/Callback_function>, Posjećeno: 2019, Februar |
| [16] | MDN web docs. [Na mreži], <https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise>, Posjećeno: 2019, Februar |
| [17] | P. Teikeira, *Professional Node.js*.: John Wiley, Sons, Inc., Indianapolis, 2013. |
| [18] | C. Gackenheimer, *Node.js Recipes*.: Apress, New York, 2013. |
| [19] | C. Ihrig, *Pro Node.js for Developers*.: Apress, New York, 2013. |
| [20] | MDN web docs. [Na mreži], <https://developer.mozilla.org/en-US/docs/Learn/Server-side/Express_Nodejs>, Posjećeno: 2019, Februar |
| [21] | Express. [Na mreži], <https://expressjs.com/en/guide/using-middleware.html#middleware.built-in>, Posjećeno: 2019, Februar |
| [22] | N. Brown, *TypeScript: JavaScript Development Guide*.: CreateSpace, California, 2016. |
| [23] | B. A. Syed, *TypeScript Deep Dive*.: Samurai Media, 2017. |
| [24] | Angular. [Na mreži], <https://angular.io/docs>, Posjećeno: 2019, Februar |
| [25] | Ionic Docs. [Na mreži], <https://ionicframework.com/docs/intro>, Posjećeno: 2019, Maj |

x

1. <https://www.w3.org/> [↑](#footnote-ref-1)
2. <https://www.ecma-international.org/> [↑](#footnote-ref-2)
3. <https://reactjs.org/> [↑](#footnote-ref-3)
4. <https://emberjs.com/> [↑](#footnote-ref-4)
5. <https://vuejs.org/> [↑](#footnote-ref-5)
6. <https://nodejs.org/en/> [↑](#footnote-ref-6)
7. <https://expressjs.com/> [↑](#footnote-ref-7)
8. <https://www.typescriptlang.org/> [↑](#footnote-ref-8)
9. <https://www.microsoft.com/bs-ba/> [↑](#footnote-ref-9)
10. <https://angular.io/> [↑](#footnote-ref-10)
11. [https://jwt.io](https://jwt.io/) [↑](#footnote-ref-11)
12. <https://v8.dev/> [↑](#footnote-ref-12)