UNIVERZITET U BANJOJ LUCI

ELEKTROTEHNIČKI FAKULTET

Tatjana Kovačević

IZRADA JAVASCRIPT APLIKACIJA

DIPLOMSKI RAD

Banja Luka, april 2019.

**Tema**: ИЗМЈEНА DOM МОДЕЛА ПУТЕМ ЕКСТЕНЗИЈЕ WEB ЧИТAЧА

Ključne riječi:

*Web* читач

*Web* страница

*DOM* модел

Екстензија *web* читача

**Komisija**: Проф. др Славко Марић, предсједник

Проф. др Зоран Ђурић, ментор

др Огњен Јолџић, члан

**Kandidat**:

Tatjana Kovačević

UNIVERZITET U BANJOJ LUCI

ELEKTROTEHNIČKI FAKULTET

KATEDRA ZA RAČUNARSTVO I INFORMATIKU

**Predmet**: ЕКСТЕНЗИЈЕ WEB ЧИТАЧА

**Tema**: ИЗМЈЕНА DOM МОДЕЛА ПУТЕМ ЕКСТЕНЗИЈЕ WEB ЧИТАЧА

**Zadatak**: Објаснити потребу за динамичком измјеном садржаја *web* странице. Описати *HTML* *DOM* (*Document Object Model*). *DOM* програмски интерфејс. *JavaScript* и *DOM*. Анализирати могућности измјене *DOМ‑*a путем екстензије *web* читача. Реализовати екстензију за одабрани *web* читач која мијења садржај *web* странице.

**Mentor**: Проф. др Зоран Ђурић

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

Bana Luka, april 2019.

Sadržaj

[1. Uvod 1](#_Toc5042009)

[2. HTML 2](#_Toc5042010)

[3. CSS3 8](#_Toc5042011)

[4. JavaScript 9](#_Toc5042012)

[4.1 Uvod 9](#_Toc5042013)

[4.2 Osobine *JavaScript*-a 9](#_Toc5042014)

[4.3 Client Side i Server Side JavaScript 10](#_Toc5042015)

[4.4 Događaji 11](#_Toc5042016)

[5. Node.js 13](#_Toc5042017)

[5.1 Uvod 13](#_Toc5042018)

[5.2 Event-Driven model programiranja 13](#_Toc5042019)

[5.3 Moduli 15](#_Toc5042020)

[5.4 Express 16](#_Toc5042021)

[5.5 Middleware 17](#_Toc5042022)

[6. TypeScript 19](#_Toc5042023)

[6.1 Uvod 19](#_Toc5042024)

[6.2 TypeScript sistem tipova 20](#_Toc5042025)

[6.3 TypeScript funkcionalnosti 20](#_Toc5042026)

[7. Angular.io 23](#_Toc5042027)

[7.1 Uvod 23](#_Toc5042028)

[7.2 Arhitektura 23](#_Toc5042029)

# Uvod

# 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*) kojim *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*

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 tekste koji će biti ispisan ukoliko slika iz nekog razloga nije učitana.

Osnovnu strukturu HTML dokumenta čine sjedeči 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. Sljede 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*, *justify*.
* *<font size=”10”>Tekst veličine 10.</font>* - Omogućuje 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*. Na slici je primjer korištenja liste.
* *<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, izbora elementa iz padajuće liste i slično). Atributom *method* se definiše način prenosa informacija koje su unešene (GET ili POST), dok se atributom *action* definiše adresa stranice (URL) 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, s tim što dodaje i jednu horizontalnu liniju ispred i iza koje se nalazi prazan red. Pomoću atributa *size* može se definisati visina, a širina pomoću *width*.
* *<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 efikasnije način nego što je to bilo ranije. [[3](#KFö)] Preporučen je i odobren od strane *W3C[[1]](#footnote-1)* organizacije (eng. *Wordl Wide Web Consortium)*.

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 neki od najznačajnih karakteristika.

Struktura i semantika dokumenta:

* <header> - Ovaj tag odnosi se na kontejner za naslov i dodadni uvodni sadržaj. Ne odnosi se samo na zaglavlje na vrhu stranice, već se mogu koristi 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> - Ukolilo je potrebna 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 zaglavnja ili *footer*-a.
* <figure> - Koristi se za prikazivanje nezavisnih slika, grafika, dijagrama i unutar njega se može nalaziti elemet <figcatption> za navođenje naslova.
* <meter> - Element za grafičko predstavljanje vrijednosti u zadatom opsegu.
* <progress> - Grafički predstavlja napredak 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>* u okviru pretraživača se može pogledati video sadržaj. Kako bi bila omogućena podrška za različite čitače i uređjaje, 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*

Kao što se vidi na slici, unutar taga *<video>*, navode se taogovi *<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>* predstavalja kontejner. Na slici je primjer.

*Canvas* element se pronalazi u *DOM*-u 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* 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 perzistento 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 prozara 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.

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.

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.

Globalni atributi:

Globalni atribut su atributi zajednički za sve *HTML* elemente, mogu se koristiti za sve elemente, mada na neke elemente nemaju uticaj. Neki od zajedničkih atributa: *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.

# 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. Korištenjem CSS 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.

Selektori se koriste za izbor elementa kome se definiše stil, neki od novih selektora u CSS3 su: *element1~elemnet2* – za selekciju svakog elemnet2 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 jedini element u roditeljskom elementu, *:ntn-child(n)* – selektuje element koji je n-ti u roditeljskom elementu, itd.

Tekstualni efekti u CSS3 imju 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 je prikazano korištenje slike kao ivice:

*Slika 2.3*

<!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>

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. JavaScript može funkcionisati kao proceduralni jezik ili kao jezik baziran na protipovima objekata. Objekti se kreiraju programski zajedno sa pripadajućim metodama i atributa. 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 *NodeJS* koje je opisano u poglavlju 5.

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

## Osobine *JavaScript*-a

U JavaScript-u nije dozvoljeno deklarisanje tipova podataka promjenjivih. JavaScript *interpreter* prepoznaje tip podatka i dinamički ga dodjeljuje promjenjivoj. JavaScript tipovi mogu biti podijeljeni u dvije kategorije: primitivni i objekti. Primitivni tipovi mogu bit: *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:

*Slika 4.1*

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;

Na slici je primjer funkcije koja kreira objekat.

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. 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:

*Slika 4.2*

function myFunction(param1, param2){

statements

}

## Client Side i Server Side JavaScript

Izvršanje *JavaScript* koda na klijentskoj strani u okviru čitača podrazumijeva da taj *web* čitač ima ugrađen *JavaScript* prevodilac (eng. *interpreter*). 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* (eng. *Document Object Model*) kojim se omogućava pristup objektima stranice.

*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 je neuobičajan i nepraktičan danas. Danas se pišu programi tako da se za svaki događaj poziva *event handler* ili *event listener* funkcija. Ove funkcije se pozivaju asinhrono kada se desi događaj za koju je funkcija napisana. 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*. 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:

*Slika 4.3*

buttonOk = document.getElementById(‘idButton’);

buttonOk.onclick = function(){

alert(‘button ok’);

}

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. 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*

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*-e. *Promise* je objekat koji predstavlja rezultat izvršavanja asinhrone operacije. Rezultat može bit da je operacija uspješno ili neuspješno izvršena. – OVJE REFERENCU-

*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: *peding* – 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. MDN*

*Slika 4.5*

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

mysql.getConnection(function(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);

})

# Node.js

## Uvod

Node.js[[3]](#footnote-3) je platforma za razvoj brzih i skalabilnih serverskih aplikacija napisanje u JavaScript-u. Dizajniran je tako da bude dobar za intezivne I/O aplikacije koje koriste *event-driven* arhitekturu. *Event-driven* model programiranja čini Node.js dobrim i efikasnim za izvršavanje aplikacija u realnom vremenu koje se pokreću na udaljenim uređajima.

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, događaj se može definisati kao asinhrone ulazno/izlazne (*asynchronous I/O*) poruke.

Postoji više razloga zašto je *Node.js* sve više pogodniji za razvoj serverske strane. Jedan od njih je *JavaScript* kao najčešće korišten programski jezik web aplikacija. Drugi razlog je jednostavnost *Node.js*-a. Osnovne funkcionalnosti su svedene na minimum. Još jedan od razloga je jednostavno preuzimanje i instaliranje, a zatim 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. Ovaj stil programiranja, gdje se umjesto povratne vrijednosti, definišu *callback* funkcije koje će biti pozvane kada se događaj desi i koje će obraditi zahtjeve i vratiti odgovor, još naziva asinhrono programiranje. Ovo je jedna od karakteristika koje definišu *Node.js*. Na Slici 5.1 je prikazan poziv *callback* funkcije koja dobavlja podatke iz baze:

*Slika 5.1*

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);

})

}

})

}

Unutar *getRestaurants()* poziva se funkcija *getRestaurantsDB()* i kao argument joj 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 poziva rukovaoca događaja). *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 i osnovne funkcionalnosti su svedene na minimum. 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. Module 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.

*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* moduo uključuje klase, metode i događaje za kreiranje *Node.js* *http* servera.
* *url* – *url* moduo uključuje metode za URL rezoluciju i parsiranje.
* *fs* – *fs* moduo uključuje klase, metode i događaje za rad sa fajlova

*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 prikazana je na Slici 5.2:

var http = require('http');

*Slika 5.2*

Funkcija *require* vraća *http* objekat koji implementira API (eng. *Application programming interface*) 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:

*Slika 5.3*

$ npm install <naziv modula>

Ovako kreirani moduli smještaju se u lokalni direktorijum *node\_modules* i mogu se učitati pomoću funkcije *require* i prosljeđivanjem imena modula, kao na Slici 4.3.2.

*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 NodeJs 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*

## Express

Express.js[[4]](#footnote-4) je najpopularnije razvojno okruženje za Node.js i osnovna biblioteka za brojna druga razvojna okruženja. MDN

Nudi sljedeće mehanizme:

* pisanje rukovaoca 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 instališe pomoću NPM, naredbom koja je prikazana na Slici 5.5:

$ npm install express --save

Slika 5.5

Na ovaj način će Express.js biti instaliran i sačuvan u listi zavisnosti (eng. *dependencies*). Instaliranje 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 koju 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*

## 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-respons* ciklusu aplikacije. Sljedeća funkcija *middleware* je označena varijablom *next*. *Middleware* funkcije mogu obavljati sljedeće zadatke: izvršavanje nekog koda, pormjenu *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.

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 smatrati 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 (*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:

*Slika 5.7*

// *controllers/user.controller.js*

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

function getAllUsers(req, res){

db.getAllUsers(function(data){

res.send(data);

})

}

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:

*Slika 5.8*

// 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);

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

# TypeScript

## Uvod

TypeScript[[5]](#footnote-5) je programski jezik koji je razvijen od strane Microsoft korporacije. TypeScript je proširenje JavaScript-a koje treba da omogući lakši razvoj velikih JavaScript aplikacija. 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:

* Obezbjeđuje opcioni sistem tipova za JavaScript
* Obezbjeđuje planirane funkcije iz narednih JavaScript verzija

Potreba za ovim ciljevima je opisana u nastavku.

## TypeScript sistem tipova

Tipovi imaju dokazanu 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*.

*Slika 6.1*

var foo = 456;

foo = ‘456’;

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

var foo: number = 123;

*Slika 6.2*

## TypeScript funkcionalnosti

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

Klase – TypeScript podržava objekto-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*

Nasljeđivanje – Klase u TypeScriptu podržavaju jednostruko nasljeđivanje korštenjem 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:

*Slika 6.4*

let fullname = (firstName, lastName) => {

firstName + ‘ ’ +lastName;

}

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*

Možemo primjetiti da objekat koji se prosljeđuje funkciji *printName* ne implementira interfejs Student. Objekat ima atribut *name* čime je zadovoljen zahtjev interfejsa. Takođe, redosljed 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 koristit mora biti uvezen pomoću odgovarajuće *import* forme. Na Slici 6.6 je primjer korštenja modula:

*Slika 6.6*

//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”);

# Angular.io

## Uvod

Angular.io[[6]](#footnote-6) 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.

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 donijla 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 instliran Node.js. Nakon instaliranja Node.js-a, instalira se Angular CLI (*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:

*Slika 7.1*

npm install –g @angular/cli

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

*Slika 7.2*

ng new Project1

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

---- ovdje možda mogu reći nešto o pokretanju aplikacije ng serve ili o konfiguracionim datotekama ---

## Arhitektura

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



*Slika 7.3*

Komponente i *tempate*-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 komponentama, kao što je ruter servis koji omoćava navigaciju. U nastavku su elementi opisani detaljnije.

Moduli – sistem modula u Angular-u se naziva *NgModules*. Aplikacija uvijek ima najmanje jedan *root* (korijenski) modul, ali uobičajno ima više funkcionalnih modula. *Root* modul se naziva *AppModule* koji obezbijeđuje mehanizam za pokretanje aplikacije. Moduli su veoma dobar način organizacije aplikacije i proširenje njenih mogućnosti pomoću drugih modula, kao što su *FormsModule*, *HttpClientModule* i *RouterModule*. Organizovanje koda u različite funkcionalne module pomaže u upravljanju razvoja složenih aplikacija i dizajniranju za ponovnu upotrebu. Pored toga, omogućava da se iskorisit prednost *lazy* učitavanja (eng. *lazy-loading*). *Lazy-loading* omoćuva da se prilikom pokretanja aplikacije sadržaj učitava dinamiči 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 korijeska 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:

*Slika 7.4*

// 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 {}

*Root* modul-*AppModule* uvijek ima korijensku komponentu, u primjeru sa Slike 6.4 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* - reltivna putanja na kojoj se nalazi HTML datoteka (*template*) komponente.
* *providers* - lista provajdera (eng. *providers*) koje komponenta zahtjeva.

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

*Slika 7.5*

// 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 {

/\* … \*/

}

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 podžava dvosmijerno 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* transformaciju u HTML

dokumentu. 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 DOM prema uputstvima koje daju direktive. Direktive su klase sa *@Directive* dekoratorom. Komponente su tehnički direktive. Komponente su karakteristične i centralne za Angular aplikacije, Angular ih definiše dekoratorom *@Component* koji proširuje dekorator *@Directive* sa karakteristikama orjentisanim na *template*. Pored komponenti, postoje još dvije vrste direktiva: strukturale i atributne. Strukturalne direktive mijenjaju izgled dodavanjem, uklanjanjem i zamijenom elemenata u DOM-u. Na Slici 7.6 je primjer dve ugrađene strukturalne direktive kaje dodaju logiku aplikaciji:

*Slika 7.6*

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

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

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

\*ngFor – iterator kroz listu *students*.

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

Atributne direktive mijenjaju izgled i ponašanje postojećeg elementa. Izgledaju kao regularni atribut, pa se zbog toga i nazivaju atributni. Direktiva *ngModel* implementira dvosmijerno 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:

*Slika 7.7*

<input [(ngModel)]=“student.name“>

Servisi – Servis (eng. *Service*) je klasa sa dobro definisanom svrhom i treba da uradi nešto specifično. Cilj je da komponenta bude što jednostavnija i treba da predstavi svojstva i metode za vezivanje podataka kako 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* - u Angularu se 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 obezbjeđuju Angularu da servis ubaci u komponentu kao zavisnost. Mehanizam za kreiranje zavisnosti je *injector*. *Injector* održava konetjner 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 :

*Slika 7.8*

// student.service.ts

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

@Injectable({

providedIn: ‘root’

})

export class StudentService {}

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. Pogledati Sliku 7.5.

*HttpClient* - Veliki broj klijetskih aplikacija koristi HTTP protokol za komunikaciju sa serverom. Moderni pretraživači podržavaju dva različita API-ja za kreiranje HTTP zahjteva: *XMLHttpRequest* interfejs i *fetch()* API. *HttpClient* iz *@angular/common/http* bibliotke

nudi pojednostavljen klijentski HTTP API za Angular aplikacije koji je zasnovan na *XMLHttpRequest* interfejsu podržan od strane pretraživača. Prednosti korištenja *HttpClient-*a su: testabilnost, tipizirani objekti zahtjeva i odgovora i pojednostavljeno rukovanje greškama. Prije korištenja *HttpClient*, aplikacija mora uvesti modul *HttpClientModule*, najčešće se 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* i pretraživač otvara odgovarajuću stranicu, klikom na *link* u stranici, pretraživač prelazi na sljedeču odgovarajuću stranicu, klikom na nazad ili naprijed, pretraživač se kreće kroz istoriju stranica unazadi ili naprijed. Angular ruter je sličan ovom modelu. Može interpretirati URL adresu *web* pretraživača kao instrukciju za navigaciju do određenog pogleda aplikacije, komponenti ciljanog pogleda može proslijediti opcione parametre na osnovu kojih će komponenta odrediti koji sadržaj treba prikazati u pogledu. Može preusmijeriti na određeni pogled kao reakciju na korisničku aktivnost (klik na dugme, izbor stavke iz padajuće liste). Navigacija izmedu pogleda se čuva u istoriji *web* pretraživača, tako da automatski rade tasteri pretraživača za naprijed i nazad.

# OPIS IMPLEMENTACIJE APLIKACIJE ZA NARUČIVANJE HRANE

U praktičnom dijelu ovog diplomskog rada implementirana je aplikacija za naručivanje hrane. Aplikacija je realizovana korišćenjem tehnologija koje su opisane u prethodnim poglavljima. Arhitektura aplikacije je klijent-server bazirana. Klijentski dio aplikacije je realizovan pomoću Angular-a, a serverski korištenjem Node.js platforme.

Korisnici aplikacije 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 „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.

--- Možda poglavlje aritektura i baza(ili da o bazi pričam u serverski dio) a arhitekturu da spomenem u uvodu... ---

## Serverski dio aplikacije

Serverska strana je realizovana pomoću Node.js-a i Express.js razvojnog okruženja za Node.js. Server je baziran na *RESTful* (eng. *REST* – *Representational State Transfer*) arhitekturi kojom se obezbjeđuje jednostavan pristup resursima i njihovoj modifikaciji od strane *REST* klijenata. Sve se odvija putem *HTTPS* (eng. *Hypertext Transfer Protocol Secure*) protokola. Na osnovu zahtjeva koji mogu biti GET, POST, PUT i slično, i na osnovu URI-ja 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[[7]](#footnote-7) (eng. *JSON Web Token*) tokenom za autorizaciju, jedinstvenim identifikatorom (skr. id) korisnika i ulogom (eng. *role*). Ukoliko korisnik nije registrovan, server šalje obavještenje „*Invalid login*“. 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 8.1 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, iduser: user.iduser, role: user.role};

res.send(toSend);

}

else{

res.send({error: ‘Invalid login’});

}

})

}

Nakon što se u funkciji *login* izvrši provjera da li je korisnik prijavljen na sistem, pozivom funkcije *loginUser* – koja komunicaira sa bazom podataka, o čemu će detaljnije pisati u nastavku, kreira se token. Token se kreira tako što se prvo učita *jsonwebtoken* modul što je prikazano na Slici . Modul je prethodno instaliran pomoću NPM repozirojuma. 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 zahjtev sa putanjom koja zahtjeva 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 koji zahtijeva. U slučaju da ova funkcija vrati poruku o grešci, server će tu poruku proslijediti korisniku. Na Slici je prikazana funkcija *createOrder* u kojoj se vrši verifikacija 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 izgledati na sljedeći način: Authorization: Bearer <token>.

Baza podataka je kreirana na *MySQL* serveru. 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 održavaju veze ka bazi aktivnim, tako da kada se traži veza, koristi jedna od aktivnih, umjesto da se kreiraju druge. Funkcija *getConnection* uzima jednu slobodnu vezu ka bazi. Modul *mysqlConnection* je prikazan na Slici 8.3.

// 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 8.4 je prikazana funkcija *loginUser* u kojoj se aplikacija povezuje sa bazom radi provjere da li je korisnik prijavljen na sistem.

//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’)); }

}

});

}

});

}

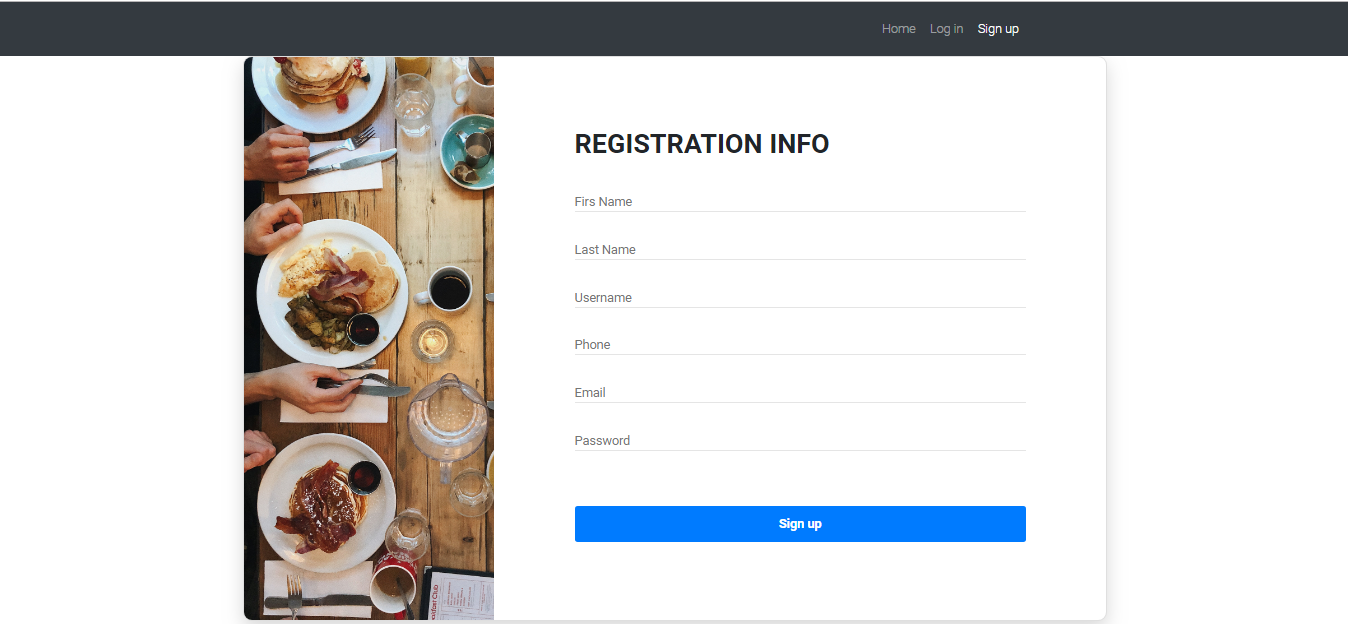
(treba li pisati više o samoj bazi(tabele i kolone i ... )

(opisati fju login i opisati jwt.sign i expireIn).. opisati kako se provjerava i verifikuje JWT kada e primi u zahtjevu—isto SLIKA. Možda malo ubaciti još o HTTPS-u. Opisati moduo sa Bazom tj kako se pozvaju fje koje komuniciraju sa bazom...

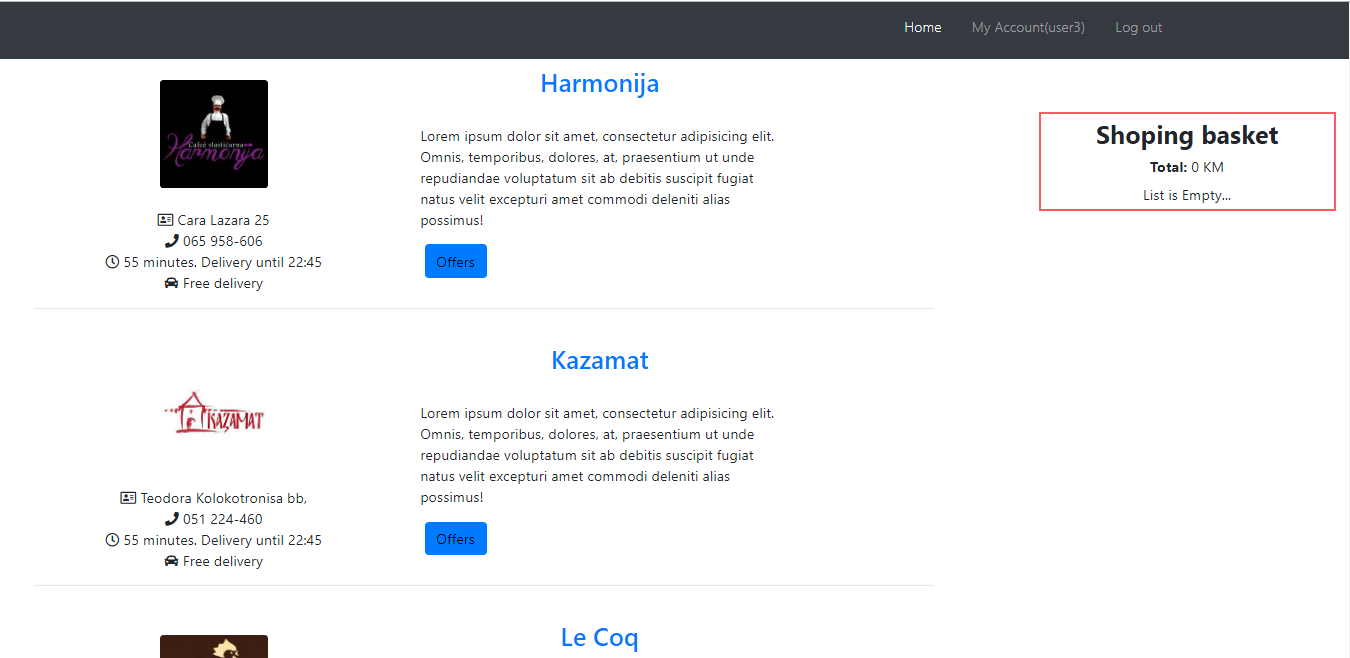
## Klijentski dio aplikacije

Za realizaciju klijentskog dijela aplikacije korišten je Angular verzija 6. Svi potrebni resursi na klinetskoj 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 mora biti prijavaljen na sistem. Korisnik koji nije prijavljen, a želi pogledati ponude nekog od restorna, prikazuje mu se poruka o tome da je potrebano da se prvo prijavi na sistem. Da bi se korisnici mogli prijaviti na sistenm 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.

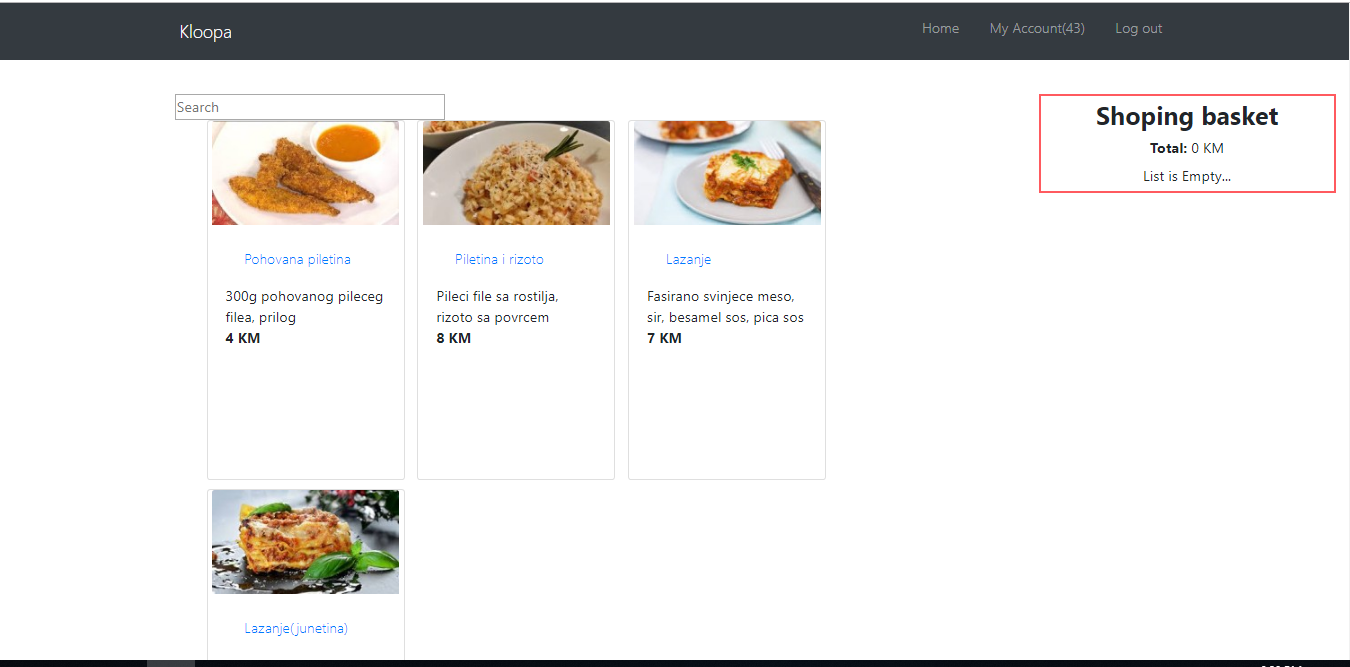


Kako bi se registrovao potrebno je da unese sva polja koja se nalaze u formi za registraciju. Ukoliko se korisnik pokuša registrovati sa *email*-om ili brojem telefona koji nije u skladu sa definisanim šablonom, dobiće poruku o grešci. Slično, ukoliko se pokuša registrovati sa korisničkim imenom koje već koristi neki drugi korisni, 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 prikazana je na Slici



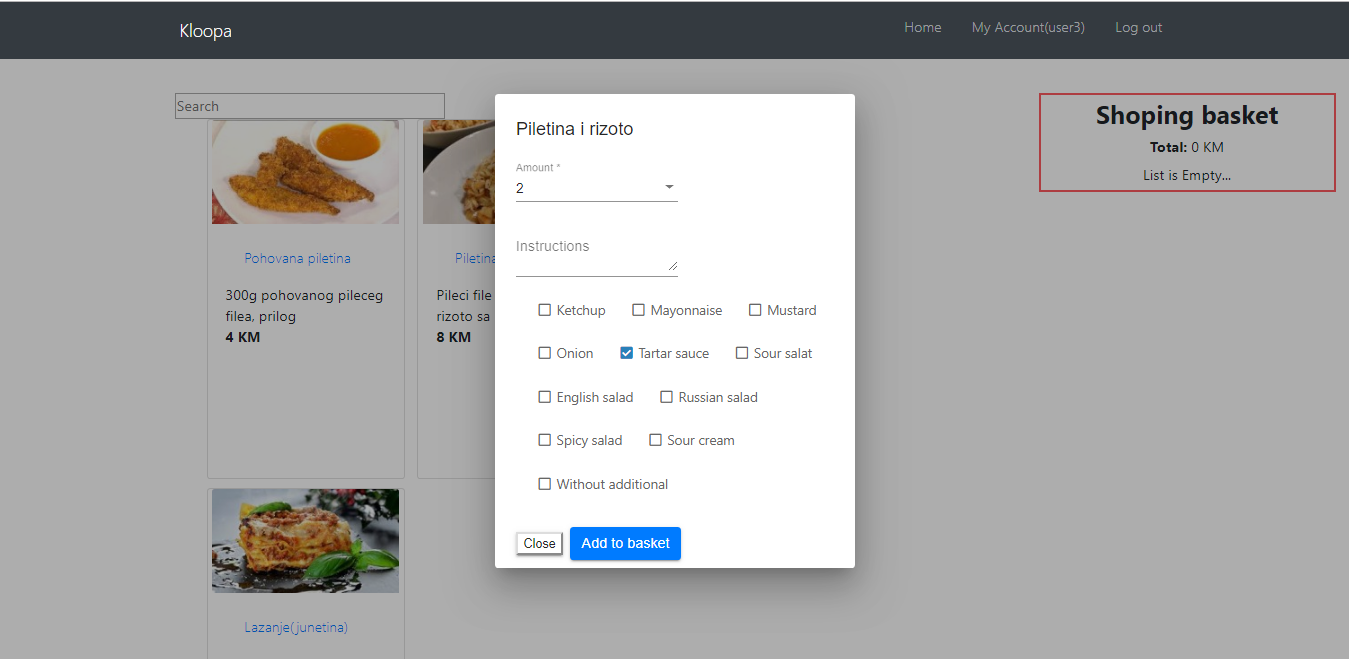
Figure

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 zahtijeva 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 prikazana je na Slici



Figure

Korisni može vršiti pretragu ponuda unošenjem ključne/ključnih riječi u polje *Search*. Po izboru neke od stavke 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



Figure

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 *createOrdere*, kreira se novo *http* zaglavlje sa autorizacionim tokenom i poziva metoda *post httpClient* objekta kojoj se proslijeđuje URL zahtjeva za kreiranje narudžbe. Na Slici je prikazan kod kojim se realizuje ovaj zahtjev.

Figure

//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’);

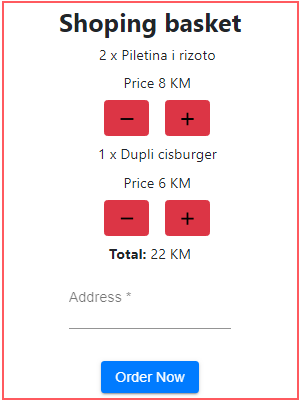
this.headers = new HttpHeaders({‘authorization’: ‘Bearer’ + token});

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

}

}

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 – kako bi umanjio broj porcija. Na Slici je prikazan izgled korpe sa izabranim stavkama koje korisnik želi naručiti.

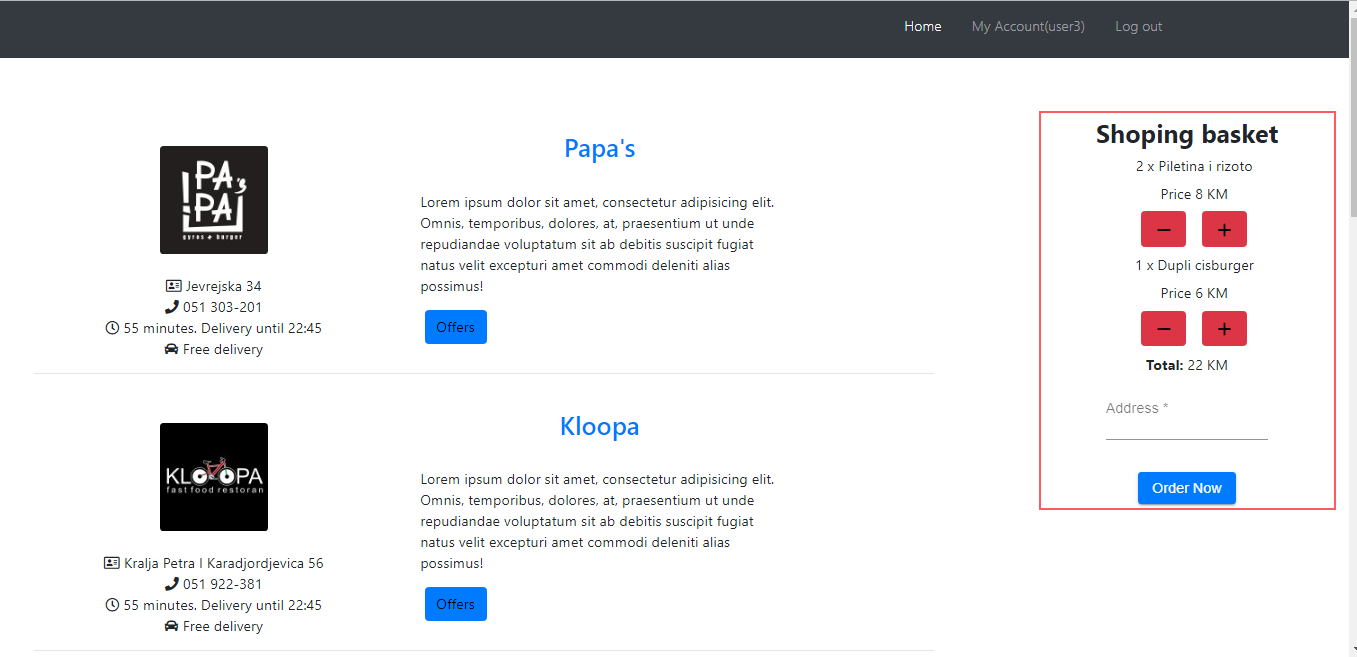


Figure

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

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

## Analiza performansi i skalabilnosti



Analiza performansi i skalabilnosti

x

|  |  |
| --- | --- |
| [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 |

x

--

1. <https://www.w3.org/> [↑](#footnote-ref-1)
2. <https://www.ecma-international.org/> [↑](#footnote-ref-2)
3. <https://nodejs.org/en/> [↑](#footnote-ref-3)
4. <https://expressjs.com/> [↑](#footnote-ref-4)
5. <https://www.typescriptlang.org/> [↑](#footnote-ref-5)
6. <https://angular.io/> [↑](#footnote-ref-6)
7. [https://jwt.io](https://jwt.io/) [↑](#footnote-ref-7)