

**UNIVERZITET SINGIDUNUM**  
FAKULTET ZA INFORMATIKU I RAČUNARSTVO

**Java Script Web Prodavnica**  
**- diplomski rad -**

**Mentor :**

Prof. Dr. Marko Šarac

**Kandidat :**

Marko Krstin

2018/270289

**Novi Sad 2022.**

## Sadržaj

|       |                                                         |    |
|-------|---------------------------------------------------------|----|
| 1     | Uvod.....                                               | 2  |
| 1.1   | Web programiranje.....                                  | 2  |
| 1.2   | Web Aplikacije.....                                     | 2  |
| 1.3   | Arhitektura web aplikacija .....                        | 3  |
| 1.4   | Java Script programski jezik .....                      | 4  |
| 2     | <i>Single-page</i> i <i>Multi-page</i> aplikacije ..... | 5  |
| 2.1   | Single-Page.....                                        | 5  |
| 2.2   | Multi-page .....                                        | 7  |
| 2.3   | Server-side rendering .....                             | 7  |
| 3     | NodeJS .....                                            | 8  |
| 3.1   | Express .....                                           | 9  |
| 3.2   | NestJS.....                                             | 10 |
| 3.2.1 | Pisanje NestJS aplikacije .....                         | 11 |
| 4     | Typescript .....                                        | 12 |
| 5     | Baze podataka .....                                     | 13 |
| 5.1   | MongoDB.....                                            | 14 |
| 5.1.1 | MongoDB Atlas.....                                      | 15 |
| 6     | WebStore.....                                           | 16 |
| 6.1   | Backend.....                                            | 17 |
| 6.1.1 | Backend kod.....                                        | 20 |
| 6.2   | Frontend .....                                          | 26 |
| 6.2.1 | Frontend kod .....                                      | 27 |
| 7     | Zaključak.....                                          | 32 |
| 8     | Literatura.....                                         | 33 |

# 1 Uvod

Imajući u vidu koliko je sam internet promenio život svih nas, od pregleda vremenske prognoze do kupovine automobila, ne možemo osporiti da tehnologija uvek ima tendenciju da olakša svakodnevne poslove ali i da ubrza upijanje informacija u realnom vremenu. Proteklih nekoliko godina, svedoci smo neverovatne ekspanzije web aplikacija ali i interesovanja za programiranje. Veliki broj kompanija se utrkuje sa konkurencijom i smišlja razne načine da se dopadne potrošaču. Razvijanjem ovakvog tržišta javila se i potreba za proširivanjem lepeze alata koji programeri imaju na raspolaganju, i o kojima će mo pričati nešto više u narednom delu.

Pre nego što počnemo detaljnije da govorimo o web aplikacija, potrebno je da ih dobro razumemo kao i sam pojam web programiranja.

## 1.1 Web programiranje

U svakoj karijeri programera dolazi trenutak kad on mora da se opredeli za jednu granu programiranja kojom će se baviti profesionalno. U ovom radu mi će mo se fokusirati na web programiranje koje je ujedno i najpopularnija grana programiranja na ovim prostorima. Ono se zasniva na pisanju programskog koda koji će služiti nekoj svrsi na pretraživaču korisnika, i doživelo je procvat razvijanjem samog interneta i njegovih tehnologija. Web programiranje kao i Web Aplikacije nastale su kad i sam *World Wide Web* i trenutno imaju najveći uticaj na život svih nas, što potvrđuje i istraživanje koje je sprovedeno na svim kontinentima. Statistika pokazuje da 66.2% ukupne populacije ima pristup internetu na kome postoje 370 miliona registrovanih domena [2].

## 1.2 Web Aplikacije

Web aplikacije možemo nazvatim svim aplikacijama koje se nalaze na internetu. Možemo ih definisati kao program koji se čuva na udaljenom serveru, prenosi internetom i prikazuje pomoću pretraživača [1]. Ove aplikacije su obično neki servis kao što je servis zakazivanja lekarskog pregleda ili servis naručivanja odeće.

Prednosti web aplikacija:

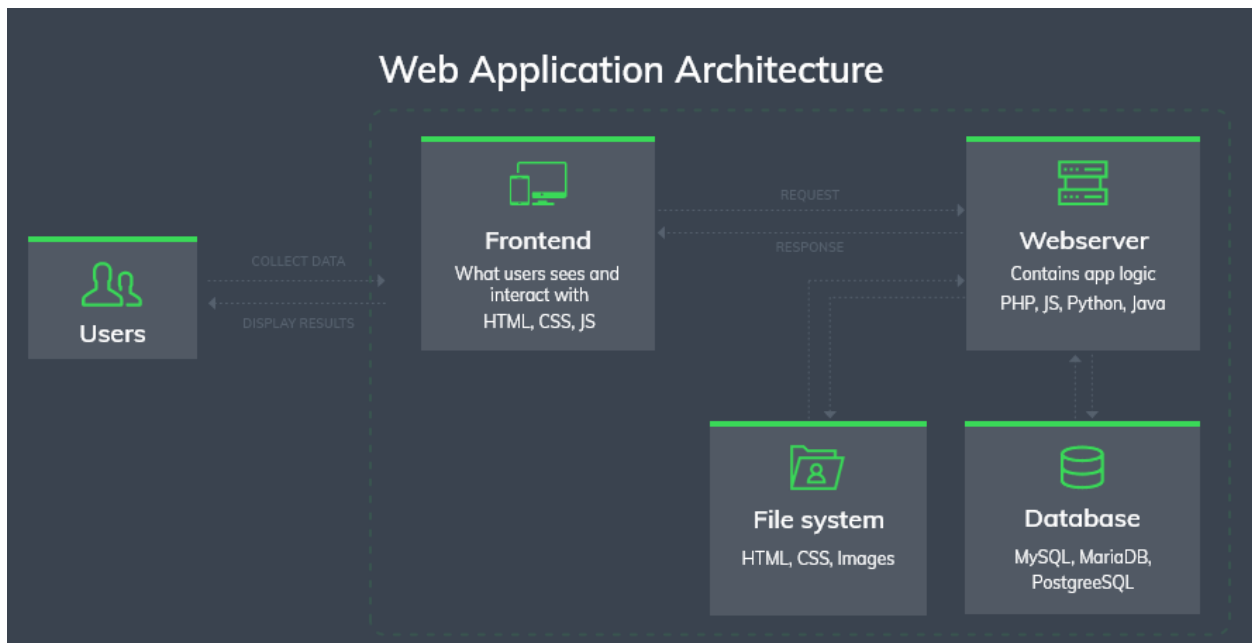
- Nije potrebno instalirati program ili aplikaciju na hard disk kompjutera

- Korisnik ne učestvuje u održavanju koda aplikacije
- Svi korisnici pristupaju istoj verziji aplikacije i ne moraju da vode računa o azuriranju koda
- Niži troškovi održavanja

### 1.3 Arhitektura web aplikacija

Svaka web aplikacija radi na sličan način u sledećim fazama;

- Korisnik pravi zahtev ka web serveru preko korisničkog interfejsa koji se prenosi putem interneta
- Web server prosledjuje ovaj zahtev ka odgovarajućem serveru za obradu podataka
- Ovaj server generise ( razrešava ) zahtev i pravi željeni rezultat
- Zatim, taj rezultat se šalje nazad ka web server
- Na kraju, web server prosleđuje željeni rezultat ka pretraživaču korisnika koji ga prikazuje



*Slika 1: Generična arhitektura web aplikacije*

Komunikacija između servera i pretraživača se izvršava putem *Hyper Text Transfer Protocol-a* ( HTTP ). Ovaj protokol se definiše kao mrežni protokol koji pripada sloju aplikacije OSI referentnog modela, predstavlja glavni i najčešći metod prenosa informacija na webu. Osnovna namena ovog protokola je isporučivanje *HTML* dokumenta [3]. Serveri slušaju, čekaju, na poruku ili zahtev klijenta , zatim ga razrešavaju i šalju odgovor putem spomenutog HTTP protokola.

Arhitektura gore navedenih web aplikacija takodje može biti podeljena i na

- *Frontend*
- *Backend*

Ova dva dela imaju svoje odvojene tehnologije koje se implementiraju na različite načine. *Frontend* deo najjednostavnije rečeno je ono što korisnik vidi pri korišćenju aplikacije tj. ono sa čim ima interakciju ( *User Interface* ), dok *Backend* deo aplikacije je logika samog koda koji korisnik nikad neće videti i koji je najčešće veoma dobro sakriven zbog bezbednosti baze podataka kao i bezbednosti cele aplikacije

Kako bismo nastavili dalju diskusiju o Frontend i Backend delu same aplikacije, i kako bismo pogledali projekat moramo se upoznati sa programskim jezikom koji je korišćen u izradi ove Web Prodavnice kao i sa bazom podataka koja je korišćena za čuvanje bitnih informacija.

#### 1.4 Java Script programski jezik

Programski jezik koji je nastao 1995. godine, i kojeg je pronasao Berndan Eich, danas predstavlja jedan od stubova modernog web programiranja.

U početku je dobio naziv Mocha, ali je ubrzo promenio ime u JavaScript kako bi se svideo široj javnosti, zbog tada popularnog programskog jezika **Java**.

Javascript se definiše kao “programski jezik za skriptovanje koji omogućava implementaciju kompleksnih rešenja na *webu* kao što su dopremanje i slanje podataka serverima, dinamički prikaz informacija i potpuna kontrola nad web stranicom” [4].

Danas, ovaj jezik je standardizovan pomoću **ECMA** organizacije, koja ima primarni zadatak da stvori uneverzalna pravila za sve Javascript **pokretače** o kojima ćemo pričati više u daljem tekstu.

Kako bi ovaj programski jezik bio razumljiv pretraživačima, oni koriste svoje *pokretače* koji su implementirani u njima. Najpoznatiji Javascript *pokretači* su:

- **V8 engine** (pokretač) koji je razvila kompanije Google i koji se koristi u njihovim pretraživačima ali i u Node JS-u. Ovo je i najpopularniji Javascript *engine*. Takodje, V8 tim se trudi da implementira nove funkcije u pretraživačima i predstavlja predstavnike Javascript zajednice koji su najveći inovatori.

- **SpiderMonkey engine** je razvila Mozilla koristi se u Firefox pretraživačima
- **JavaScriptCore** pokretač dolazi iz Apple kompanije I implementiran je u Safari pretraživaču
- **Chakra** je pokretač na ovoj listi koji ima najmanju pokrivenost nekih Javascript funkcija I trenutno se koristi u Internet Explorer-u I Edge pretraživaču

Razvijanjem *World Wide Web-a* programerima je bila potrebna veća podrška za kompleksnije zadatke I aplikacije, gde se Javascript najbolje uklopio I pružio sve potrebne alate za tako nešto.

Moramo imati na umu da u početku , ali I sredinom 2010. godine , ovi alati su tek počeli da se razvijaju I da sada imamo priliku da vidimo mnoštvo Javascript Biblioteka koje su svima nama olakšale pravljenje web aplikacija.

Javascript Biblioteka se može definisati kao paket već u napred napisanog Javascript koda koji je moguće skinuti sa interneta, modifikovati I implementirati u svoju aplikaciju

Web prodavnica, o kojoj ćemo detaljnije pričati u nastavku, je u potpunosti pisana u Javascript programskom jeziku

Kako bismo bolje razumeli arhitekturu ove web aplikacije, moramo se bolje upoznati sa trenutno dostupnom tehnologijom kao i sa načinom njihovog funkcionisanja.

## 2 *Single-page* i *Multi-page* aplikacije

Na samom početku pravljenja aplikacije , tim koji je zadužen za osmišljavanje arhitekture i odabir tehnologije ispred sebe ima veliki izazov. Naravno, svaki projekat je posebna tema za sebe u zavisnosti od zahteva i raspoloživih sredstava, ali kako bi smo uprostiti našu diskusiju, tehnologije za izradu **frontend** dela aplikacije možemo podeliti na:

- *Single-page aplikacije*
- *Multi-page aplikacije*

### 2.1 Single-Page

**Single-page aplikacije (SPA)** je najlakše razumeti kao aplikacije koje se izvršavaju u pretraživaču korisnika i koje **ne zahtevaju reload** (ponovno učitavanje) stranice svaki put kad se nešto dinamički promeni na njoj. Ovakve aplikacije koristimo svakodnevno, a neke od najpopularnijih su : Gmail, Google Maps, Facebook, Netflix, GitHub i mnoge druge.

Ova vrsta aplikacije se trudi da pruži sto približniji osećaj *native aplikacije* (aplikacija koja je izrađena za određeni operativni sistem i koja radi van pretraživača korisnika)

Pri korišćenju ovakve aplikacije korisnik stvarno ima ovakav osećaj, jer se sve promene dešavaju u relanom vremenu.

Single-page aplikacije emituju ceo Java Script kod pri inicijalnom učitavanju stranice, koji je zatim potrebno da se izvrši na korisničkom pretraživaču odjednom. Ovaj proces izvršavaju Java Script **pokretači** o kojima smo diskutovali u ranijem delu teksta. Nakon inicijalnog učitavanja, web aplikacije se više neće opet učitati, već će se samo menjati dinamički u realnom vremenu.

Ovakav način pravljenja modernih aplikacija je voma zastupljen u trenutku pisanja ovog diplomskog rada i ima najavljenju svetlu budućnost u domenu web programiranja.

Trenutno najpopularnije tehnologije za pisanje SPA su:

- ReactJS (razvijen u Facebook-u)
- Angular (razvijen od strane Google tima)
- VueJS
- Svelte

Prednosti SPA:

- Ovakve aplikacije su veoma brze jer se većina resursa (HTML+CSS+Skripte) izvršavaju samo jednom, dok se razgovaranje frontenda sa backend izvršava neometano
- *Development* (pravljenje, kodiranje) ovakvih aplikacije je prilično standardizovano, što znači da su „najbolje prakse“ dobro poznate i dostupne

Mane SPA:

- Preuzimanje cele skripte odjednom i njeno izvršavanje može biti naporno za pretraživač. Problem može nastati ako korisnik ima loš protok interneta ili loše performanse računara, pa zato ovaj proces može potrajati duže.
- Problem takođe može nastati ako korisnik isključi Java Script u pretraživaču. Ako se to desi aplikacije neće moći da se prikaže. Rešenje ovakvog problema može biti ***Server-side Rendering arhitektura*** o kojoj ćemo pričati više u nastavku teksta.
- Za prikaz aplikacije potrebno je da se svi procesi učitavanja završe

## 2.2 Multi-page

**Multi-Page aplikacije (MPA)** se mogu smatrati „tradicionalnim“ načinom prikazivanja i učitavanja aplikacije. Naime, svaka promena, npr. prikazivanje podataka ili slanje podataka iz forme, zahteva ponovno učitavanje stranice. Ovakve aplikacije su obično veće od SPA. Pri izradi ovakvih aplikacija koriste se HTML CSS i JavaScript koji nije upakovan u neki *framerowk* (okvir)

Prednosti MPA:

- Ovakva arhitektura može biti savršeno rešenje gde nisu potrebne jake performanse SPA. Jedan od primera gde bi MPA bilo dobro rešenje su blogovi. Kod blogova nemamo potrebu za preterano dinamicnim web stranicama
- Lako upravljanje sa SEO optimizacijom

Mane MPA :

- Razvijanje ovakvih aplikacija može da bude veoma kompleksno ako govorimo o aplikacijama koje imaju veliki broj web stranica ili veliki broj promena na web stranici.

## 2.3 Server-side rendering

Postoji još jedna arhitektura kod SPA koja je trenutno dobila veliki značaj u fonrtend svetu. Naime, reč je o **server-side** renderovanju, koja se sve više koristi, kako zbog performansi , tako i zbog lakoće razvijanja ovakvih aplikacija.

**Server-side rendering aplikacije (SSR)** možemo definisati kao aplikacije koje vrše učitavanje i renderovanje aplikacije na strani servera. Ovako učitane i renderovane HTML stranice se šalju korisniku na pretraživač koji od tog trenutka nema nikakvog posla, osim da prikaže već učitanu stranicu.

Ovakva arhitektura je i SEO friendly, jer algoritmi za pretraživanje mogu da prodju kroz index pretrage pre nego sto HTML stranica stigne do korisnika

Najpopularniji *framework* (okviri) koji podržavaju SSR su:

- Angular Universal
- NextJS
- NuxtJS

Trenutno većina popularnih SPA ima i implementacije u SSR aplikacija. Ovo omogućava developerima da se lakše odluče za arhitekturu koja više odgovara trenutnim projektima

Neki od primera popularnih SPA i SSR implementacija je React koji podržava i NextJS. NextJS predstavlja SRA aplikaciju sa react tehnologijom i njenom sintaksom koja ima mogućnost da renderuje (učita) svoj html,css i javascript kod na strani servera.

Webstore aplikacija o kojoj ćemo detaljnije pričati u nastavku je pisana u React tehnologiji.

### 3 NodeJS

**NodeJs** možemo definisati kao višeplatformsko Java Script orkuženje otvorenog koda za izvršavanje Java Script koda na serverskoj strani (izvršavanje *backend* koda). Bitno je napomenuti da se NodeJS izvršava pomoću V8 pokretača o kome smo pričali u tekstu gore i koji je jedan od najzastupljenijih pokretača na tržištu.

Backend programiranje je usmereno ka pisanju koda koji će se nalaziti na serveru i koji neće biti vidljiv korisniku aplikacije, kao što je frontend kod vidljiv. Ovaj kod ima primarni cilj da izvršava kompleksne algoritme ali i da ima primarnu konekciju ka bazi podataka. Pomoću backend koda developeri dobavljaju i modifikuju podatke iz baze, koje zatim šalju frontend delu aplikacije preko API interfesa o kojima ćemo pričati vižše u nastavku.

NodeJs je jedan od popularnijih okruženja za pisanje gore navedenog *backend koda*. Kao i što sam naziv kaže, pisanje i okrišćenje NodeJS-a se ostvaruje pomoću JavaScript programskog jezika. Od toga i potiče naziv JS.

Takodje NodeJS je otvorenog koda, što bi značilo da je kod koji se izvršava „iza kulisa“ samog okruženja dostupan široj javnosti i moguće ga je pronaći na internetu.

Samo neke od kompanije koje koriste NodeJS u svojim aplikacijama su GoDaddy, Groupon, IBM, LinkedIn, Microsoft, Netflix, PayPal, SAP, Walmart, AWS ...

```

var sys = require("sys"),
    http = require("http"),
    url = require("url"),
    path = require("path"),
    fs = require("fs");

http.createServer(function(request, response) {
    var uri = url.parse(request.url).pathname;
    var filename = path.join(process.cwd(), uri);
    path.exists(filename, function(exists) {
        if(!exists) {
            response.writeHead(404, {"Content-Type": "text/plain"});
            response.end("404 Not Foundn");
            return;
        }

        fs.readFile(filename, "binary", function(err, file) {
            if(err) {
                response.writeHead(500, {"Content-Type": "text/plain"});
                response.end(err + "n");
                return;
            }

            response.writeHead(200);
            response.end(file, "binary");
        });
    });
}).listen(8080);

console.log("Server running at http://localhost:8080/");

```

*Slika 2: Primer NodeJS koda*

### 3.1 Express

ExpressJS možemo definisati kao unapredjeni i bolji NodeJS koje je u suštini samo okvir (framework) NodeJS-a. Osnivač Express-a je prvobitno dobio inspiraciju iz Sinatra softverskim serverom, kojeg karakteriše mala veličina sa velikim mogućnostima za nadogradnju [5].

Kako bismo bolje razumeli tehnologiju koja se koristila u izradi Web Prodavnice, moramo imati i kratak pregled tehnologije koja je omogućila lako pisanje koda, jer tako nešto nije bilo moguće oduvek.

Neke od prednosti koje je ExpressJS doneo u odnosu na obican NodeJS:

- Robusno rutiranje
- Express je koncentrisan isključivo na što bolje performanse
- Dolazi sa HTTP pomoćnicima (koji omogućavaju redirekciju, keširanje...)

```
// Import the Express library.
const express = require('express');

// Initializing the app.
const app = express();

// Getting the path request and sending the response with text
app.get('/', (req,res) => {
  res.send('Hi, your request has been received');
});

// Listen on port 2000
app.listen(2000, () => {
  console.log('listening at http://localhost:2000');
});
```

**Slika 3: Primer ExpressJS koda**

## 3.2 NestJS

Tehnologija koja se koristila za pisanje *backend* dela ove web aplikacije je NestJS

Kao nastavak ExpressJS , NestJS je takodje NodeJS okriv (framework) koji developerima daje nove alate i olakšava pisanje backend koda.

Naime, pisanje velikih aplikacija u NodeJS-u bey dobro osmišljenog plana ili strategije pisanja koda se može završiti veoma loše. NestJS je taj problem rešio pisanje modularnog koda (koda koji se sastoji iz više modula ili delova) koji je obez

bedio mnogo veću čitljivost. Takodje ovakav kod je mnogo lakše održavati ili nadograđivati.

Još jednu bitnu stvar koju je omogućio NestJS, je ta da on automatski dolazi sa preduslovom da se kod u njemu pise u TypeScriptu, što je takođe omogućilo pisanje „čistijeg“ koda, koda sa manje *bagova*.

### 3.2.1 Pisanje NestJS aplikacije

Kao što smo pomenuli u ranije tekstu NestJS je pisan u modulima (delovima koda). Obično se glavni modul u aplikaciji naziva **app.module.ts** koji „spaja“ sve ostale module u jedan, i koji se može smatrati glavom cele aplikacije. Kako bi Nest znao da se radi o modulu, potrebno je napisati na samom početku koda *dekorator* **@Module**. Ovaj dekorator govori samom Nest-u tačno koji kontroler, servis ili neki drugi resurs će se koristiti u aplikaciji.

Aplikacija, kada se instancira, dolazi sa samo jednim modulom, app.module.ts o kom smo pričali u tekstu gore. Kako se aplikacija širi tako će se javljati novi moduli koji će se spajati u jedan root module.

**Kontroleri** sa druge strane imaju ulogu da regulišu HTTP zahteve, kao što su GET, POST, PATCH, DELETE. Kako bi Nest znao da se radi o kontroleru, potrebno je na početku koda naglasiti sa **@Controller**.

**Provajderi** ili **Provajderi Servisa** se koriste za pisanje logike aplikacije. Dobavljanje podataka iz baze, njeno modifikovanje ili čak kompleksni algoritmi samo su od neke logike koja može biti implementirana u ovim provajderima. Provajderi se označavaju sa **@Injectable** dekoratorom.

Sad kada imamo pregled svih komponenti jedne Nest aplikacije možemo videti i njenu najčešću arhitekturu. Poziv sa frontend dela aplikacije se šalje ka kontroleru koji prima taj poziv i poziva odgovarajuću logiku iz provajdera koja odrađuje ceo težak posao.

```
import { Body, Controller, Get, Post } from "@nestjs/common";
import { Order } from "../order.schema";
import { OrdersService } from "../orders.service";

@Controller('orders')
export class OrdersController {
  constructor(private readonly ordersService: OrdersService) {}

  @Post()
  async createOrder(@Body() createOrder: Order) {
    return this.ordersService.create(createOrder);
  }

  @Get()
  async getAllOrders(): Promise<Order[]> {
    return this.ordersService.getAllOrders({});
  }
}
```

**Slika 4: Primer NestJS kontrolera**

## 4 Typescript

Typescript je dodatna sintaksa Javascripta koja dopotpunjava sam programski jezik i omogućava lakše uočavanje grešaka u kodu. Trenutno predstavlja budućnost pisanja Javascripta i ima sve veću primenu kako u većim kompanijama sa ogromnim aplikacijama, tako i u malim kompanijama koje nemaju tako velike aplikacije.

Što je aplikacija veća, korišćenje ovakve sitakse ima više smisla.

U suštini, Typescript omogućava *tipiziranje* varijabli i metoda u kodu, što nije normalno ponašanje za Javascript, jer Javascript nije jezik koji je originalno koristio ovakavu sintaksu.

Moderni Pretraživači ne mogu da pokrenu Typescript, pa je ovakav kod potrebno konvertovati u obican Javascript. Naime Typescript kod se izvršava samo jednom, i to pri inicijalizaciji koda, i u tom trenutku ćemo dobiti informaciju od code editora da li je sve u redu sa napisanom logikom. Nakon toga, Typescript se pretvara u obican Javascript koji je kasnije moguće izvršiti u pretraživaču.

Neke od prednosti koje pruža korišćenje Typescript-a:

- Čitljivost koda
- Lakše uočavanje grešaka
- Lakše održavanje već napisanih aplikacija

```
class Validators {  
  static min(min: number): ValidatorFn  
  static max(max: number): ValidatorFn  
  static required(control: AbstractControl<any, any>): ValidationErrors | null  
  static requiredTrue(control: AbstractControl<any, any>): ValidationErrors | null  
  static email(control: AbstractControl<any, any>): ValidationErrors | null  
  static minLength(minLength: number): ValidatorFn  
  static maxLength(maxLength: number): ValidatorFn  
  static pattern(pattern: string | RegExp): ValidatorFn  
  static nullValidator(control: AbstractControl<any, any>): ValidationErrors | null  
  static compose(validators: ValidatorFn[]): ValidatorFn | null  
  static composeAsync(validators: AsyncValidatorFn[]): AsyncValidatorFn | null  
}
```

*Slika 5: Primer Typescript koda, iz Angular dokumentacije*

Bitno je napomenuti da je, web aplikacija o kojoj ćemo pričati u nastavku, pisana u potpunosti u Typescript-u. Kako na frontend delu koda tako i na backend delu.

Za frontend korišćena je TSX sintaksa, dok je za backend korišćena obična Typescript sintaksa u NestJS *framework-u*.

## 5 Baze podataka

Slobodno možemo reći da skoro svaka aplikacija mora da ima bazu podataka koja će čuvati sve bitne informacije koje su potrebne za rad. Te informaciju se mogu odnositi na prikaz nekih informacija klijentu (npr. prikaz svih raspoloživih proizvoda koje neka prodavnica nudi) ili interne informacije (informacije kojima klijent nema pristup i odnose se na npr. podatke o korisnicima ili podatke o fizičkim radnjama)

Ovi podaci se čuvaju na udaljenom serveru koji ima jako obezbeđenje i koji će servirati te iste podatke klijentu po potrebi. Klijent ni u jednom trenutku nema pristup svim informacijama, već samo onim informacijama koje će mu programeri, koji su napravili aplikaciju, dozvoliti.

Strukture ovih baza podataka su prava umetnost i postoji ogroman broj načina kako se one mogu dizajnirati. U zavistnosti od potreba aplikacije ali i od količine podataka koju je potrebno skladištiti, programeri na raspolaganju imaju :

- **Key-value database** (Redis, Memecached) - ovakvu arhitekturu baze podataka je najlakše zamisliti kao Javascript objekat. Što bi značilo da ona sadrži kjuč, i taj ključ sadrži neki podatak. Beoma je brza, ali ima ograničeni prostor za skladištenje
- **Wide-column database** (Cassandra, HBASE) – arhitektura koja je slična predhodnoj, sa tim da dodaje novu dimenziju čuvanja podataka. Isto kao i u key-value arhitekturi, ova baza čuva kjuč koji u ovom slučaju ne čuva samo jedan podatak već čitavu kolonu podataka
- **Document database** (MongoDB, FirebaseFirestore) – u ovoj arhitekturi, čivanje podataka se vrši u dokumentima. Jedan podatak je jedan dokument. Takođe, bitno je napomenuti da se u jednom dokumentu može naći mnoštvo drugih dokumenata. Ovakva arhitektura se još popularno naziva **NoSQL** baze podataka, jer se dokumenti ne nalaze u nekoj relaciji. Zbog tog razloga ne moguće je praviti upite i tako izvlačiti podatke
- **Relational database** (MySQL, PostgreSQL) – ovo je najpoznatija arhitektura koja je sa nama skoro 50 godina. Najveća prednost ovakve baze podataka je pisanje upita, koji omogućavaju pisanje ali i detaljno čitanje podataka iz baze koji su povezani vezama. Više tabela povezanih relacijama iz kojih je moguće pozivati upite. Iz toga i potiče naziv relacione baze.

- **Graph database** (Neo4J) – Slična relacionoj bazi, ova arhitektura u sebi sadrži čvorove, koji predstavljaju podatke, koji se međusobno povezuju sa ivicama, relacijama preko kojih je moguće pozivati upite. Ova vrsta baze se obično koristi u finansijskom sektoru ali je poznata i po tome što je koristi AirnBNB pokretač.

## 5.1 MongoDB

U projektu o kome ćemo pričati detaljnije u nastavku, odlučeno je da se koristi MongoDB kao arhitektura baze podataka. Tačnije, korišćen je **MongoDB Atlas**.

MongoDB spada u grupu *document* baza, što bi značilo da se ona sastoji od više dokumenata između kojih ne postoje relacije, i zbog toga se još naziva i NoSQL baza podataka.

Kao što je rečeno u predhodnom tekstu, ovi dokumenti mogu da sadrže druge dokumente u sebi.

Izabrana je ova baza podataka jer je bila najjednostavnija za implementaciju u ovoj vrsti aplikacije i nije bilo potrebe za drugim arhitekturama baza, koja bi omogućila lakše snalaženje u mnogo većim količinama podataka.

Ovakva arhitektura je i više nego dovoljna za neke manje ili srednje aplikacije koje nemaju ogromne zahteve u pogledu pretraživanja i upisa velike količine podataka.



**Slika 6: Primer slike jednog dokumenta iz MongoDB-a**

### 5.1.1 MongoDB Atlas

MongoDB bazu je moguće pokrenuti na nekoliko načina:

- U lokalu (gde lokalni računar obaljšva posao servera koji šalje podatke)
- Na udaljenom serveru
- U oblaku (*cloud*)

U našem slučaju, odlučili smo se za rešenje u oblaku pod nazivom **MongoDB Atlas**. Naime, moguće je iznajmiti deo računarskog hardvera koji će se koristiti za skladištenje i odpremanje informacija.

Ovo Cloud rešenje kao i svako drugo, podrazumeva da mi kao korisnici ne vodimo računa o hardveru i njegovom neometanom izvršavanju. Mi kao korisnici smo tu samo da koristimo te resurse. Izabrani server koji koristimo je na teritoriji Frankfurta, i pripada AWS udruženju cloud provajdera.

#### Dedicated Development Clusters ⓘ

|            |          |               |           |
|------------|----------|---------------|-----------|
| <b>M10</b> | 2 GB RAM | 10 GB Storage | 0.2 vCPUs |
| <b>M20</b> | 4 GB RAM | 20 GB Storage | 0.4 vCPUs |

#### Dedicated Production Clusters ⓘ

|                        |            |                 |          |
|------------------------|------------|-----------------|----------|
| ✓ <b>M30</b>           | 8 GB RAM   | 40 GB Storage   | 2 vCPUs  |
| > <b>M40</b>           | 16 GB RAM  | 80 GB Storage   | 4 vCPUs  |
| > <b>M50</b>           | 32 GB RAM  | 160 GB Storage  | 8 vCPUs  |
| > <b>M60</b>           | 64 GB RAM  | 320 GB Storage  | 16 vCPUs |
| <b>M80</b> LOW CPU     | 122 GB RAM | 750 GB Storage  | 16 vCPUs |
| <b>M140</b>            | 192 GB RAM | 1000 GB Storage | 48 vCPUs |
| ▽ <b>M200</b> STANDARD | 256 GB RAM | 1500 GB Storage | 64 vCPUs |
| LOW CPU                | 244 GB RAM | 1500 GB Storage | 32 vCPUs |
| <b>M300</b>            | 384 GB RAM | 2000 GB Storage | 96 vCPUs |
| <b>M400</b> LOW CPU    | 488 GB RAM | 3000 GB Storage | 64 vCPUs |

*Slika 7: Neki od ponuđenih hardverskih paketa*

## 6 WebStore

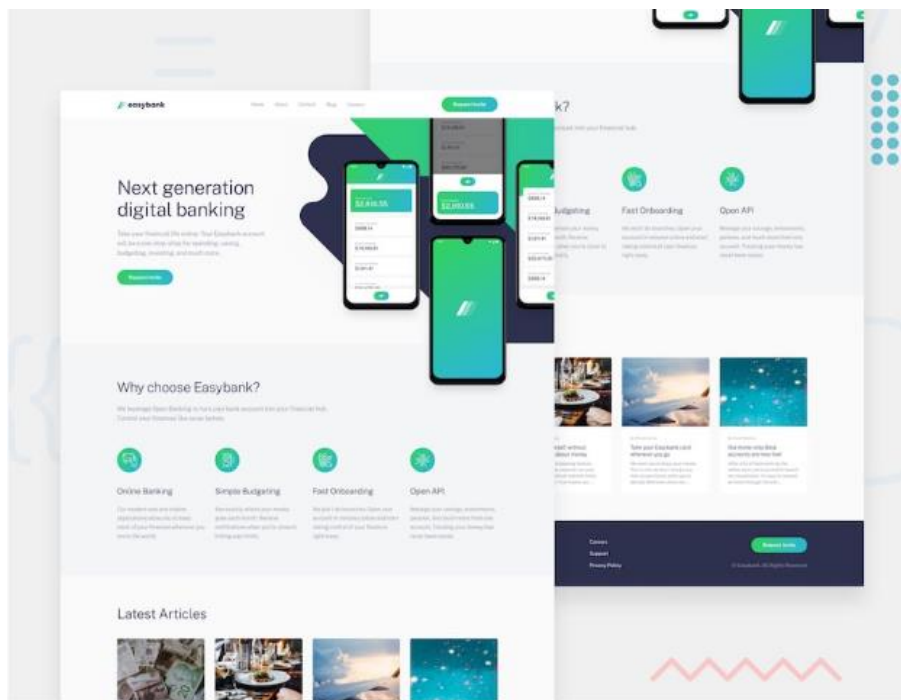
Sa osnovnim znanjem o tehnologijama koje smo dobili u prethodnom delu tesкта, možemo preći na pregled same aplikacije. WebStore kako je prvobitno nazvan, ovaj projekat predstavlja prodavnicu na internetu, sa svim svojim manama i prednostima moderne aplikacije. Ova prodavnica je univerzalna, što bi značilo da je dizajn odgovarajući za veliki dijapazon proizvoda, od mobilnih telefona pa sve do odeće.

Zbog dobro poznatih problema pri pisanju frontend koda, a koji je usmeren ka pisanju koda bez ikakvog dizajna, za ovaj projekat odlučeno je da koristimo *open source* dizajn sa interneta.

Naime pisanje, pozicioniranje blokova, biranje boja i celokupan izgled aplikacije obično nije posao samih programera. Ovaj deo posla obavljaju web dizajneri, koji na zahtev klijenta osmišljavaju dizajn aplikacije. Sa ovim dizajnom, koji se najčešće pravi u Figma ili drugim alatima, programeri prate upustvo dizajnera i sastavljaju program.

U našem slučaju, ovakav proces bi nam oduzeo dosta vremena, ali i novca, ako bismo se odlučili da platimo nekoga da sačini dizajn aplikacije. Imajući ovo u vidu, došli smo do rešenja javnog *open source* dizajna sa interneta. Celokupan dizajn folder je uzet sa websajta FrontendMentor, koji u sbi sadrži ogroman broj besplatnih dizajna koji su dostupni svima. Ovi dizajni sadrže samo skicu i izgled stranice, i **ne sadrže napisan kod**.

Za naš projekat odlučeno je da se koristi *Easybank* dizajn.



**Slika 8: Inicijalni dizajn stranice**

Bitno je napomenuti da se u folderu dobija:

- Jpeg slika sa dizajnom za desktop (1500px) , i dizajn za mobilne telefone (400px)
- Vodič za fontove (linkovi ka fontovima koji bi trebali da se koriste)
- Sve slike koje je potrebno postaviti
- README fajl koji sadrži dodatna uputstva, kao što su kodovi boja i linkovi ka Figma projektu

Takođe, bitno je napomenuti da je ovaj dizajn bio samo globalna ideja za aplikaciju, i da je dosta promenjena tokom razvijanja frontend dela koda.

## 6.1 Backend

Kako smo već upoznati u tekstu iznad, o NestJS okviru za pisanje backend koda, nastavak rasprave nam neće biti težak za razumevanje.

Za razgovaranje i modifikovanje MongoDB baze podataka korišćen je **Mongoose**. On je obezbedio neke u napred definisane metode za:

- Upis u bazu
- Brisanje iz baze
- Modifikovanje podataka

Bitno je napomenuti da je Mongoose napravljen za korišćenje samo u MongoDB bazama.

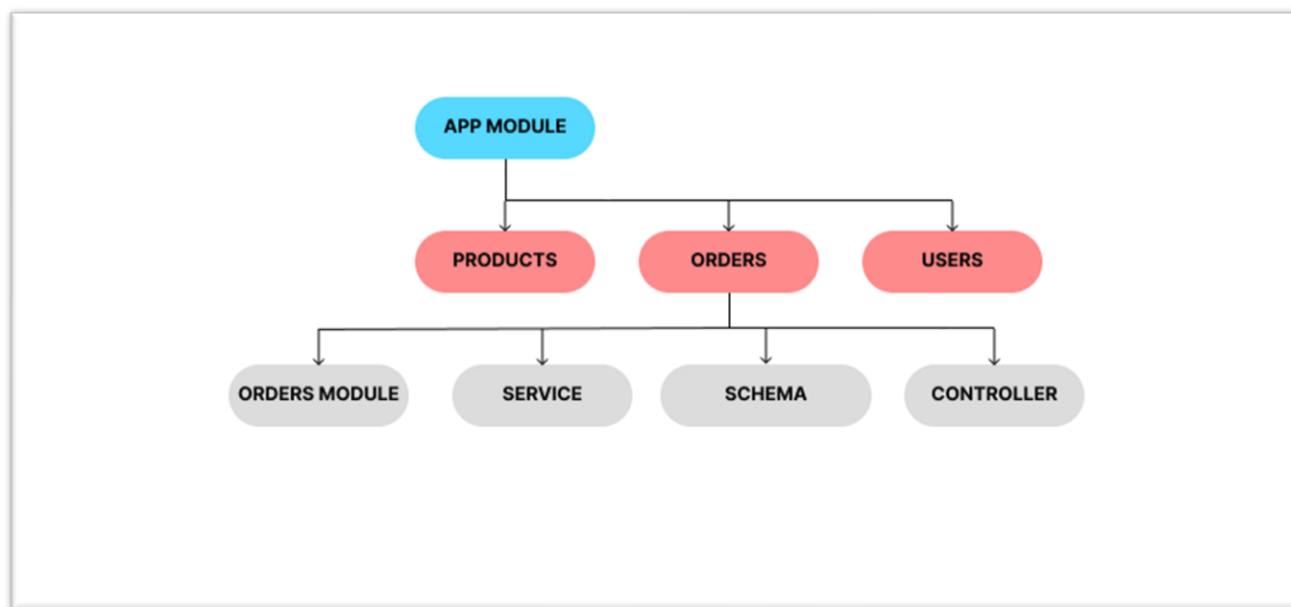
Neke od metoda koje su se koristile, a koje su došle direktno iz Mongoose biblioteke su:

- .save()
- .find()
- .deleteOne()
- .findOneAndUpdate()

Ove metode su dosta pomogle prilikom razvijanja aplikacije, jer nismo morali da pravimo naša rešenja.

Takođe, kao što smo ranije pomenuli, MongoDB baza ne podržava kompleksne upite, već samo jednostavne, što bi značilo da je moguće dopremiti jedan dokument u jednom upitu. Dokumenti međusobno nisu u nikakvoj relaciji. Naravno, ako taj dokument poseduje druge dokumente u sebi, i oni će biti dopremljeni.

Struktura NestJS fajlova je sledeća:



**Slika 9: Struktura NestJS fajlova**

Na slici iznad možemo videti da smo funkcionalnosti podelili na 3 foldera, koji u sebi sadrže sve vezano za taj deo aplikacije.

Takođe možemo videti i da se u svakom folderu nalazi ista struktura tj. da se nalaze:

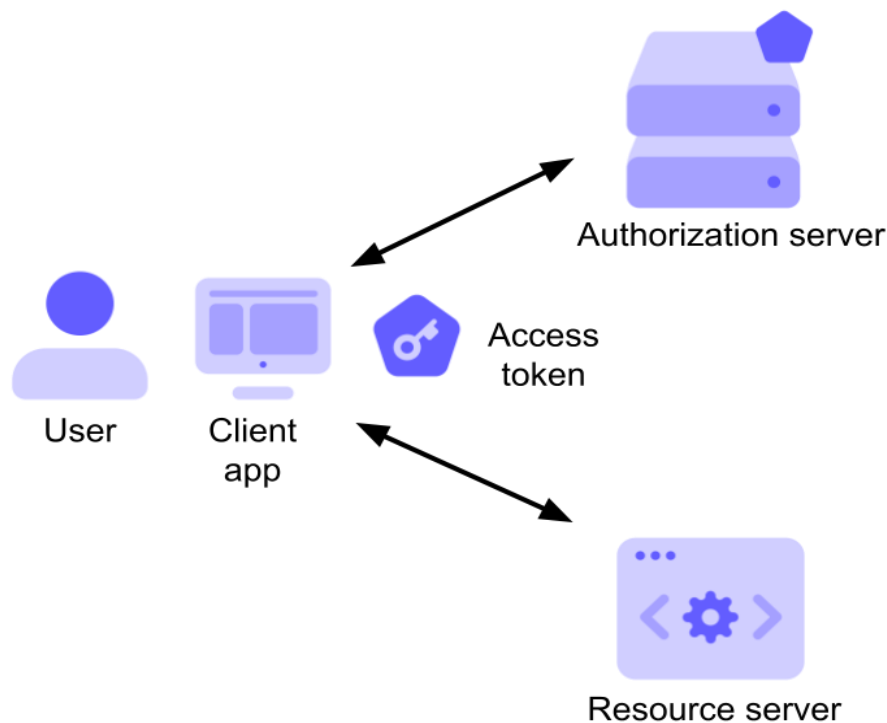
- *Controller* koji sadrži RestAPI metode, kao što su GET , POST , PATCH ...
- *Schema* koja u sebi sadrži šemu, izgled podataka koji će se prenositi
- *Service* koji u stvari radi sav posao pri pozivanju jedne od metoda iz kontrolera. On sadrži svu biznis logiku koja modifikuje informacije, čuva ih ili servis u sebi može imati kompleksne algoritme.
- *Module* je kao glava jednog od delova backend koda. U njemu se importuju svi kontroleri, servisi jednog dela aplikacije

U nastavku teksta ćemo detaljnije pregledati jedan od ovih fajlova, i tačno videti kako se neki od zahteva sa frontenda razrešavaju na backendu, ali pre toga potrebno je i spomenuti korišćenje *tokena*.

Naime, tokeni se koriste kako bi se backend podaci zaštitili od napada, tako što se prilikom svakog zahteva sa frontenda backendu u njemu šalje i access token. Ovaj token potvrđuje identitet usera koji je taj zahtev poslao

Svaki user prilikom Login-a ili prilikom Sign Up procesa na kraju dobija svoj lični token koji je nasumično generisan od strane backenda. Taj token ističe nakon određenog vremena, i nakon toga od korisnika se opet traži proces Login-ovanja. Dok token traje , prilikom svakog slanja zahteva ka backendu šalje se i ovaj token. Backend server prima token sa zahtevom i proverava da li ovaj token stvarno postoji. Ako postoji, korisnik je potvrdio svoj identitet i tek onda se nastavlja proces odgovaranja na zahtev korisniku.

Možemo reći da je ovo moderan način potvrđivanja identiteta korisnika i trenutno se koristi u velikom obimu. Naš projekat **ne koristi** ovaj način provere, jer njena implementacija nije jednostavna i za nju je potrebno veće znanje backend razvijanja. U našem prjektu **svako** može da pošalje zahtev ka serveru i da dobije odgovor, ne postoji nikakva identifikacija na strani backenda. Ako pričamo o implementaciji ovakve neke aplikacije u realnom svetu, smatramo da je ovakva bezbednosna ograda **neophodna**, kako aplikacija ne bi satlno bila u riziku od napada.



*Slika 10: Proces korišćenja tokena*

### 6.1.1 Backend kod

Pregledom našeg backend koda dobićemo veću sliku kako ova aplikacija radi na strani servera. Podsetićemo se da se naš kod odrađuje lokalno, jer naš kompjuter glumi server. Ovo ne sprečava ovu aplikaciju da se u budućnosti postavi na pravi server, na kome će se nalaziti backend logika i baza podataka.

U narednom delu teksta pogledaćemo *Orders* modul koji u sebi sadrži sve metode vezane za ovaj deo aplikacije.

#### 6.1.1.1 Order Module

```
2 import { Module } from "@nestjs/common";
3 import { MongooseModule } from "@nestjs/mongoose";
4 import { OrdersController } from "../order.controller";
5 import { Order, OrderSchema } from "../order.schema";
6 import { OrdersService } from "../orders.service";
7
8
9 @Module({
10   imports: [MongooseModule.forFeature([{ name: Order.name, schema: OrderSchema }])],
11   controllers: [OrdersController],
12   providers: [OrdersService]
13 })
14
15 export class OrderModule { }
```

**Slika 11: Orders module**

U slici iznad, možemo videti Orders modul u kome se nalaze svi importi koje koristi naš kod. On predstavlja „glavu“, tj. okuplja sve kontrolere i servise u jedan fajl.

@Module označava da se radi o Modulu, dok u zagradama možemo videti više detalja šta on sadrži. Kod importa nalazimo .forFeature() metodu koja povezuje šemu i daje joj ime. U našem slučaju koristimo OrderSchema, koja u sebi sadrži tipiziranu klasu koja bolje opisuje podatke o sa kojima ćemo raditi.

Bitno je napomenuti da prilikom izvršavanja metoda iz kontrolera i servisa, Nest proverava da li su poslani ili podaci koji su stigli podudaraju sa ovom šemom. U nastavku teksta ćemo detaljnije pogledati i sam kod ove šeme.

Ispod importa imamo polje *controllers* u kome definišemo kontrolere koje će ovaj modul koristiti, ali imamo i *providers* polje u kome definišemo servise koje koristimo. Broj ovih importa nije ograničen, ali zbog veličine aplikacije, nama nije bio potreban veći broj fajlova.

#### 6.1.1.2 Order Schema

U ovom fajlu možemo videti tipizirane objekte koji će se koristiti u komunikacijama u ovom delu aplikacije.

```
import { Prop, Schema, SchemaFactory } from "@nestjsjs/mongoose";
import { Product } from "src/products/product.model";
import { Document } from 'mongoose'

export interface OneOrder {
  email: string,
  userId: string,
  fName: string,
  lName: string,
  city: string,
  adress: string,
  phoneNumber: string
}

export type OrderDocument = Order & Document

@Schema()
export class Order {
  @Prop()
  orderDetails: [OneOrder]
  @Prop()
  orderedItems: [Product]
  @Prop()
  shipped: boolean
}

export const OrderSchema = SchemaFactory.createForClass(Order);
```

**Slika 12: Order Schema**

Na vrhu koda imamo jedan *interface* koji je deo Typescript sintakse i koji predstavlja objekat jedne narudžbine. U toj narudžbini imamo nekoliko informacija koje su tipizirane. U našem slučaju sve su tipa *string*.

Ispod toga imamo dekorator *@Schema* koji u sebi takođe ima tipizirana svojstva. *@Prop* u našem slučaju predstavlja *property* (svojstvo) našeg objekta.

Glavni cilj ovog fajla je da se poziva u drugim fajlovima i tako referencira ove tipove na pravim mestima.

### 6.1.1.3 Order Controller

U kontroler fajlu nalaze se svi pozivi, tj. metode koje je moguće poslati na backend. U našem projektu za svaki kontroler je napravljen CRUD (create, remove, update, delete).

```
@Post()
async createOrder(@Body() createOrder: Order) {
  return this.ordersService.create(createOrder)
}

@Get()
async getAllOrders(): Promise<Order[]> {
  return this.ordersService.getAllOrders({});
}

@Delete('/:id')
async deleteOrder(@Param("_id") orderId: string) {
  return await this.ordersService.deleteOrder(orderId)
}

@Patch('/:id')
async updateOrder(@Param('_id') _id: string, @Body() updatedOrder: { orderDetails: [OneOrder], orderedItems: [Product], shipped: boolean }): Promise<any> {
  return this.ordersService.findOneAndUpdate({ _id }, updatedOrder)
}
```

**Slika 13: Order Controller**

U slici iznad, možemo videti da imamo nekoliko dekoratora:

- `@Post` koji poziva metodu `createOrder()`
- `@Get` koji poziva metodu `getAllOrders()`
- `@Delete` koji poziva metodu `deleteOrder()`
- `@Patch` koji poziva metodu `updateOrder()`

Svaka od ovih metoda prima neke atribute u sebi. To može biti parameter ili to može biti `@Body` dekorator koji prilikom slanja zahteva na backend dobija i neki *payload*. Tako se recimo prilikom POST metode šalje i neki paket (payload) koji će se kasnije obraditi na backendu.

Bitno je napomenuti da ove metode (POST, GET, DELETE...) u kontroleru samo primaju ovakve zahteve i da zatim te zahteve delegiraju ka servisu, koji u sebi sadrži obiman kod za obradu tih zahteva

Na slici iznad, možemo videti metodu PATCH, koja prima :id parametar, tj. id narudžbine koju bi smo zeleli da promenimo. Takođe možemo videti i da se tu pojavljuju interfejs iz jedne šeme o kojima smo pričali u tekstu iznad

#### 6.1.1.4 Order Service

I na kraju imamo naš servis, u kome imamo obiman kod za skladištenje ili obradu podataka. U nastavku će mo prikazati Order Service, koji je jedan od jednostavnijih servisa, kako bi slika i kod bio što čitljiviji

```
export class OrdersService {
  constructor(@InjectModel(Order.name) private orderModel: Model<OrderDocument>) { }

  async create(order: Order): Promise<Order> {
    const newOrder = new this.orderModel(order);
    return (await newOrder.save())._id
  }

  async getAllOrders(usersFilterQuery: FilterQuery<Order>): Promise<Order[]> {
    return this.orderModel.find(usersFilterQuery)
  }

  async deleteOrder(_id: string) {
    const result = await this.orderModel.deleteOne({ _id: _id }).exec()
    return result
  }

  async findOneAndUpdate(orderFilterQuery: FilterQuery<Order>, updatedOrder: Partial<Order>): Promise<Order> {
    return this.orderModel.findOneAndUpdate(orderFilterQuery,
      updatedOrder)
  }
}
```

**Slika 14: Order Service**

U njemu možemo videti razne metode koje prepoznajemo iz kontrolera. Ako se osvrnemo na PATCH metodu koja poziva findOneAndUpdate(), možemo videti da ona koristi mongoose metodu koja se naziva isto tako, i koja pronalazi jedan dokument u bazi koji se podudara sa id-om koji mu prosledimo. Takođe, on prima i drugi argument a to je sa čim bi smo želeli da taj već postojeći objekat zamenimo. Ako sve prođe kako treba, u odgovoru dobijamo novo-postavljeni objekat koji je sačuvan i zamenjen u bazi sa starim.

Ovo je samo jedan od primera koji se nalaze u ovom fajlu, ali je dovoljan kako bismo razumeli strukturu rada našeg backend koda.

#### 6.1.1.5 SendGrid

Prilikom slanja nove narudžbine, korisnika je nekako bilo potrebno obavestiti da je sve prošlo kao treba i da je narudžbina spremna za otprem. U našem sličaju, koristili smo SendGrid kao rešenje ovog problema

Naime, SendGrid je servis koji ima mogućnost da šalje veliku količinu elektronske pošte na mnogo adresa. Sav ovaj proces se kontroliše preko koda, koji je potrebno napisati na backend strani aplikacije.

Bitno je napomenuti da SendGrid ima **veoma** jaku bezbednost. Prilikom registracije za ovaj servis, nama je postavljeno više od 20 pitanja, koje je morao da verifikuje pravi čovek a ne računar. Neka od pitanja su bili:

- Da li će korisnik imati mogućnost da blokira vaše poruke?
- Na kom fakultetu studirate?
- Za šta će se koristiti aplikacija?
- Kolika količina poruka će se slati na dnevnoj bazi?
- Poslati zvaničan sajt fakulteta
- Poslati zvanični registracioni kod kursa za koji je potreban ovaj projekat
- API ključ moramo zaštititi, kako ćemo to uraditi?
- Da li planirate da se ova aplikacije koristi u komercijalne svrhe u budućnosti?
- Odakle nabavljate email adrese korisnika?
- Ako je u pitanju testiranje aplikacije, koliko dugo će vam biti potrebne naše usluge

Posle svih ovih pitanja, bilo je potrebno oko dve nedelje da se potvrde ovi odgovori od strane SendGrid-a, i nakon toga dobili smo API ključ koji je potrebno pozvati u kodu.

Ovaj API ključ je tu kako bi nas identifikovao prilikom slanja nekog zahteva ka SendGrid servisu, i mora biti skriven od korisnika.

Kako bismo koristili ovaj servis, napravili smo jednu POST metodu u *order.controller* fajlu. U ovoj metodi kao parametre *payload-a*, sa frontenda smo slali ka backendu email korisnika i ID narudžbine.

Izgled same poruke, koja će stići korisniku na email inbox, je moguće menjati pomoću HTML koda. Moguće je stilizovati poruku u potpunosti sa HTML-om i *inline* CSS-om. U našem slučaju nismo mnogo vremena trošili na ovo, te smo našu poruku napisalo samo preko HTML-a. U njoj se nalazi potvrda da je narudžbina primljena i da će biti poslata što je pre moguće.

Takođe, bitno je napomenuti da ako nekoliko korisnika prijavi vašu poruku za *spam*, automatski će vam opasti SendGrid poeni i samim tim će vam opasti pouzdanost vašeg profila.

[illegible]

### Slika 15: SendGrid kod

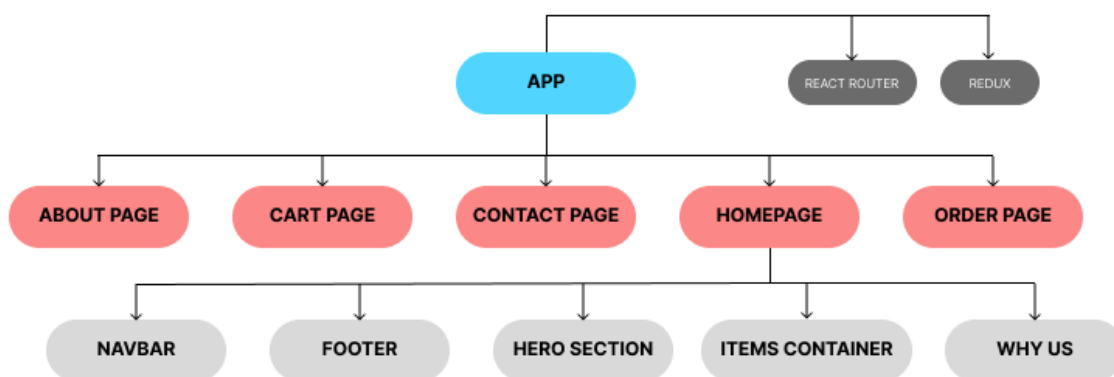
Sav kod koji smo pogledali do sad je moguće postaviti na neki od cloud rešenja kao što su AWS, Azure ili GCP.

U našem slučaju AWS bi bio najbolje rešenje. Troškovi ne bi bili veliki ali takođe u njegovim servisima bi pronašli rešenja na naše trenutne probleme. Naime, SendGrid bi zamenili za AWS servis koji radi istu stvar kao i on. Ovaj servis se naziva *Amazon Simple Email Service* i veoma je lak za upravljanje. Takođe MongoDB Atlas se već nalazi na AWS serverima tako da je veoma kompatibilan sa AWS-om

## 6.2 Frontend

Sad kada dobro poznajemo *backend* deo aplikacije, biće nam mnogo lakše da razumemo i frontend stranu. Naime, za pisanje frontend koda korišćen je React okvir.

Na slici ispod možemo videti da se React struktura sastoji od komponenti. Ove komponente se slažu jedna sa drugom i tako stvaraju korisnički interfejs.



**Slika 16: Struktura React aplikacije**

Bitna karakteristika komponenti je ta da se one mogu koristiti na više mesta, i samim tim ne dolazi do dupliranja koda. Npr. moguće je napraviti jednu komponentu koja će u sebi imati dugme. To dugme je moguće koristiti na svim mestima gde je ono potrebno.

Ovaj proces može pomoći programerima i umanjiti im potrošeno vreme na pisanje istog koda više puta.

Takodje, bitno je napomenuti da je React okvir *Single-Page-Application*. O ovoj arhitekturi smo pričali u tekstu iznad. React koristi prednosti i mane ovakve arhitekture da stvori virtuelni DOM i tako manipuliše stvarima koje će se prikazati ili ukloniti sa korisničkoj interfejsa.

O samom funkcionisanju React komponenti i njihovom *renderovanju* pričaćemo više u nastavku kada budemo videli i neke primere koda, jer njihovo razumevanje može biti zbunjuće.

### 6.2.1 Frontend kod

Kao što je već ranije spomenuto, radi se o *open source* dizajnu koji je promenjen za naše potrebe.

Aplikacija je pisana u *Typescript-u*, iako on nije neophodan za rad u React-u. React koristi JSX sintaksu, ali kako smo mi koristili Typescript verziju, mi smo koristili TSX sintaksu. Ovakav način pisanja nam je omogućio lakše hvatanje bagova ali i lakše održavanje koda.

Frontend deo aplikacije nije pisan odjednom, već se nadograđivao tokom perioda od nekoliko meseci. U ovom procesu nadograđivanja, typescript je mnogo pomogao. Pravljenje ali i dodavanje novih komponenti je moglo da stvori nepredviđene probleme, ali to je sprečeno korišćenjem typescripta.

Kao što možemo videti na slici iznad, glavna komponenta se naziva *APP*. Ona u sebi poziva sve ostale komponente. Prilikom pisanja TSX koda, dlučeno je da se u ovoj APP komponenti postave sve rute ove prodavnice.

```
<BrowserRouter>
  <Routes>
    { /* HOMEPAGE ROUTE */ }
    <Route path="/" element={
      isLoading ? <FullPageLoader /> : <React.Suspense fallback={<FullPageLoader />}>
        <Homepage />
      </React.Suspense> } />
    </Route>

    { /* FALLBACK ROUTE IF NO PATH WAS FOUND */ }
    <Route path="*" element={<Navigate to="/" /> } />

    { /* ABOUTPAGE ROUTE */ }
    <Route path="/about" element={
      isLoading ? <FullPageLoader /> : <React.Suspense fallback={<FullPageLoader />}>
        <AboutPage />
      </React.Suspense> } />
  />
```

**Slika 17: Nekoliko ruta aplikacije**

Naime, SP aplikacije nikada ne učitavaju novu stranicu, već se ceo JS kod povlači sa servera samo jednom. Tu uskaču ruteri, koji rešavaju ovaj problem tako što dozvoljavaju da korisnik ima utisak kao da aplikacija ima više stranica.

U našoj aplikaciji korišćen je ReactRouterV6, koji se dosta razlikuje od ranijih verzija.

Na slici iznad možemo videti da je za svaku rutu označena komponenta koja treba da se prikaže prilikom njenog učitavanja.

U suštini APP komponenta sadrži samo rutere i logiku za prvo učitavanje stranice.

```
useEffect(() => {
  if (isAuthenticated) {
    const getAllUsersFunc = (allUsers: User[]) => {
      let loggedInUser

      if (user?.email !== undefined) {
        loggedInUser = allUsers.find((userFromAllUsers: User) => userFromAllUsers.email === user?.email);
        dispatch(loggedInUserActions.setLoggedInUserInfo(loggedInUser));
      }
      if (loggedInUser === undefined) {
        const makeNewUserFunc = (response: User) => {
          loggedInUser = response;
          dispatch(loggedInUserActions.setLoggedInUserInfo(loggedInUser));
        }
        sendRequest({ url: "http://localhost:3333/users", method: "POST", headers: { 'Content-Type': 'application/json' }, body: {
          email: user?.email, cart: [] } }, makeNewUserFunc);
      }
    }
    sendRequest({ url: 'http://localhost:3333/users' }, getAllUsersFunc);
  }
}, [isAuthenticated]);
```

*Slika 18: Logika koja se izvršava pri prvom učitavanju stranice*

Postoje dva načina za pisanje React komponenti:

- Komponente kao klase (class based)
- Komponente kao funkcije (function based)

Pisanje komponenti u klasama se smatra starim načinom. Ako se programer odluči za ovakav način pisanja, biće primoran da kontroliše renderovanje stranice pomoću metoda koje se izvršavaju u određenom trenutku *životnog ciklusa* komponente. Neke od tih metoda su :

- componentWillMount()
- render()
- componentDidMount()
- shouldComponentUpdate()
- componentDidUpdate()

Sa druge strane, moderan način pisanja je pisanje komponenti u funkcijama. U ovom načinu pisanja cela komponenta je jedna funkcija. Na kraju te funkcije imamo *return* koji vraća JSX HTML kod. Iznad tog *return*-a, nalazi se logika same komponente. Kako bi programer imao kontrolu nad renderovanjem komponente, tj. kada će se neki deo koda odraditi, koriste se React *hook* metode. Metode koje se najviše koriste su:

- **useState()**, koristi se za čuvanje i ažuriranje stanja neke promenljive, koja je bitna za promenu na korisničkom interfejsu
- **useReducer()**, kao zamena za *useState()*, ovaj hook se koristi kada komponenta ima potrebu za više *useState()* čuvanja stanja, i ona omogućava pregledniji kod
- **useEffect()**, ovaj hook se koristi kada želimo da se neki kod odradi prilikom stvaranja komponente, ili kada želimo da se on odradi prilikom promene neke promenljive.
- **useMemo()**, koji vraća *memoisanu* vrednost neke metode, koja bi bila suviše “skupa” za izvršavanje prilikom svakog rendera komponente
- **useRef()**, ovaj hook se obično koristi za pristup ka DOM elementima, tj. pristup njihovoj reference

Za naš projekat odlučili smo se za pisanje komponenti na moderan način, jer se pokazalo kao lakši za razumevanje i brži za razvijanje.

Na slici 18. možemo videti upotrebu *useEffect()* hooka, koji nam je u tom slučaju omogućio da prilikom prvog učitavanja aplikacije, mi proverimo da li je user prijavljen ili nije. O samoj implementaciji profila i logovanja pričaćemo u nastavku teksta.

Ispod ove logike nalazi se *return*, koji vraća JSX kod. U našem slučaju, kao što smo ranije pomenuli, ovde se nalaze rute aplikacije.

Što se tiče samih proizvoda, ova prodavnica koristi javni API, koji je dostupan svima, i koji u sebi sadrži listu od 200+ proizvoda.

```
await fetch('https://dummyjson.com/products?limit=40')
```

**Slika 19: Povlačenje proizvoda sa javnog API-ja**

U slici iznad možemo videti da u ovom primeru povlačimo 40 proizvoda, koje zatim čuvamo u aplikaciji i prikazujemo korisniku.

U odgovoru koji dobijemo sa ovog javnog API-ja su razne informacije, npr: ime proizvoda, detaljniji opis, cena, ocena, glavna slika proizvoda, sporedne slike, količina koja je dostupna, identifikacioni broj proizvoda, težina artikla, proizvođač, model proizvoda...

### 6.2.1.1 Redux Toolkit

Kako bi komponente mogle da razmenjuju informacije i podatke između sebe, React nudi nekoliko rešenja:

- *Prop* provlačenje informacija. Ovo rešenje se zasniva na tome da svaka komponenta može da pošalje informaciju ili podatak svom roditelju ili detetu. Tu nailazimo na problem velikog broja komponenti i komplikovanog *provlačenja* informacija sa jedne komponente na drugu.
- *Context Hook* je hook koji je uskočio u pomoć kako bi rešio gore navedeni problem. Naime, Context se koristi kao mesto u aplikaciji koje može da prima i odprema podatke. Glavna prednost ovakvog sistema je što svaka komponenta, gde god se ona nalazila, može da komunicira sa Contextom, bez komplikovanog provlačenja informacija.
- Neki od alata za Globalno čuvanje stanja u aplikaciji. Kao i Context postoje i drugi alati koje je moguće koristiti. Najpopularniji od svih je Redux i Redux Toolkit. Redux je dodao nekoliko slojeva kompleksnosti na Context sistem. Redux je generalno teže koristiti u početku jer je komplikovaniji za pisanje, ali pruža mnogo bolji raspored koda.

Naime u Redux-u, mesto u kom ćemo čuvati stanje (podatke) se obično naziva **Store** (prodavnica). Ova prodavnica sadrži logiku za primanje, obradu i odprem podataka iz nje. Sastoji se od nekoliko delova

- State (stanje)
- Reducer
- Dispatch

Svaka prodavnica u sebi ima nekoliko početnih stanja, koja se tokom vremena menjaju i na koje se možemo prikačiti pomoću *Selector* metoda, kako bismo koristili ove podatke. Takođe, imamo i Reducer metode, pomoću kojih obrađujemo podatke koji dolaze u State. Ove metode mogu da samo popunjavaju State ili da ih nekako modifikuju pre popunjavanja.

```
const loggedInUserInitialState = { loggedInUser: {} };
const loggedInUserSlice = createSlice({
  name: 'loggedInUser',
  initialState: loggedInUserInitialState,
  reducers: {
    setLoggedInUserInfo(state, action) {
      state.loggedInUser = action.payload
    }
  }
})
```

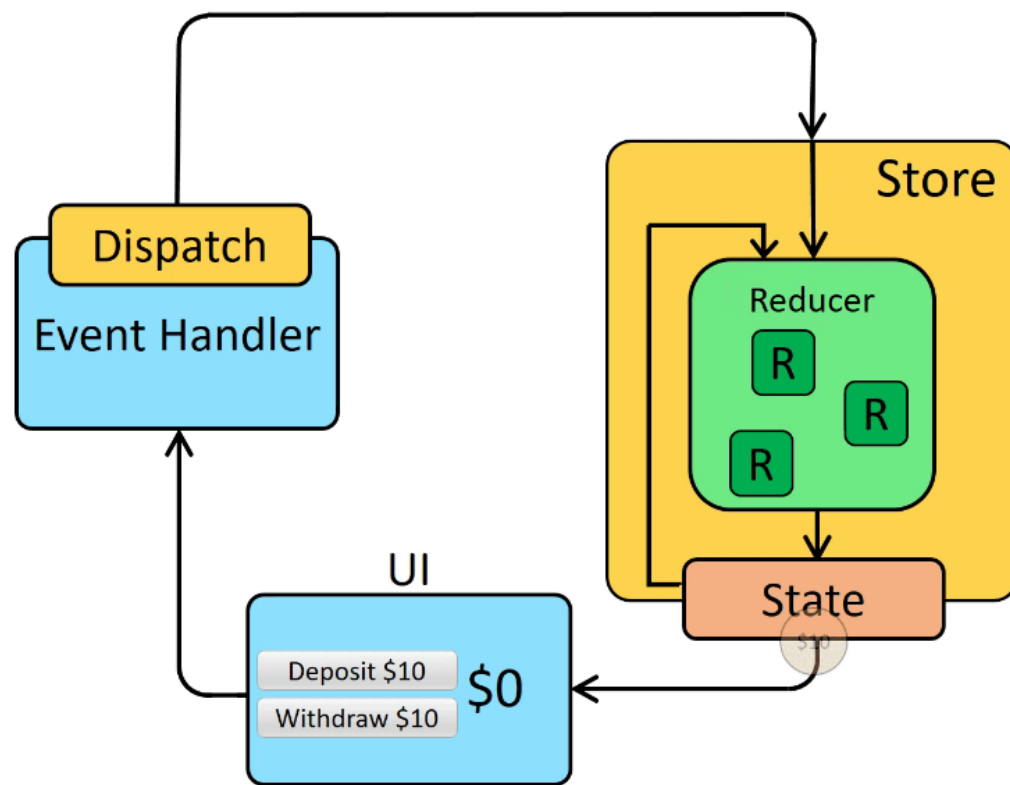
**Slika 20: Primer dela Redux prodavnice**

I na kraju imamo Dispatch metode pomoću kojih se iz komponente šalju podaci u prodanicu. Komponenta zna da one postoje jer na kraju naše prodavnice imamo konfiguraciju koja ima spisak svih reducera koji mogu da se pozovu iz komponente.

```
const store = configureStore({ reducer: { product: productsSlice.reducer, cart: cartSlice.reducer, loggedInUser: loggedInUserSlice.reducer } })
```

**Slika 21: Konfiguracija prodavnice**

Samo razumevanje ovog celog sistema može biti teško i komplikovano. Iz tog razloga u nastavku imamo sliku koja opisuje ceo sistem rada Redux alata.



**Slika 22: Redux sistem**

Potrebno je naglasiti da je Redux veoma pomogao u razvijanju ove aplikacije. Stvaranjem sve više komponenti nastao je dobro poznat problem razgovaranja između njih. Problem je rešen uvođenjem Redux sistema.

Redux pakete **ne** održava React tim, već skroz odvojena grupa ljudi.

### 6.2.1.2 Auth0 autentifikacija korisnika

Problem profila i razlaganje korisnika je jedna od mogućnosti modernih web sajtova koja je postala neizostavna stvar.

Ovakav sistem profila, prijavljivanja i čuvanja informacija o korisnicima, nije lako napraviti i oduzima mnogo vremena. Mi smo se odlučili za Auth0 servis koji omogućava baš to.

Auth0 nam je omogućio da se prilikom prijavljivanja korisnika njegove informacije čuvaju u njihovoj bazi podataka. Takođe, on nam je omogućio veoma bezbedan način proveravanja korisnika pomoću tokena.

Admin Auth0 ima pristup spisku korisnika koji su se prijavili na naš webstore. Admin ima mogućnost da menja njihove informacije ali **nikad** nema mogućnost da vidi korisnikovu sifru. Sifra je sačuvana nakon što je prošla kroz algoritam koji je skriva. Tako sakrivena se čuva u bazi.

Proces izgleda ovako:

1. Korisnik klikne dugme za *login*
2. Webstore aplikacija ga putem obezbeđene veze vodi ka Auth0 stranici za login
3. Korisnik unosi svoje informacija
4. Auth0 servis proverava da li korisnik postoji, i ako postoji vraća ga na našu aplikaciju sa podacima da je sve prošlo kako treba

## 7 Zaključak

Na prvi pogled ova aplikacija, i njeni delovi se čine komplikovani, ali svaki njen deo je dobro promišljen i svaki njen deo je izabran sa namerom da aplikacija bude što lakša za održavanje u budućnosti. Sa druge strane, koristili smo popularne tehnologije, React i Nest, tako da snalaženje sa njima ne bi trebao da bude problem za programere koji će naslediti kod web prodavnice. Ove navedene tehnologije su **trenutno** popularne, i veoma korišćene. To ne znači da će ova aplikacija ostati na njima, već su moguće promene. Možda se u budućnosti pojavi nova arhitektura baze podataka ili novi okvir za pisanje frontend koda, koji bi bio primenljiviji za našu aplikaciju.

Sama pomisao da će se ovakve inovacije sigurno desiti u budućnosti je interesantna i uzbudljiva, a na nama ostaje da ih iskoristimo kako bismo učinili ovaj svet boljim mestom.

Hvala na pažnji,

*Marko Krstin*

## 8 Literatura

[1]

*M. Šarac, D. Radovanović, A. Jevremović and M. Mravik, Internet marketing, Univerzitet Singidunum, Beograd, 2020.*

[2]

*Broadband Search: Globalna statistika korisnika internet mreže*

<https://www.broadbandsearch.net/blog/internet-statistics>

[3]

*Anonymous contributor: Definition of Web Applications*

<https://www.techtarget.com/searchsoftwarequality/definition/Web-application-Web-app>

[4]

*MDN documentation: What Is javascript*

[https://developer.mozilla.org/en-US/docs/Learn/JavaScript/First\\_steps/What\\_is\\_JavaScript](https://developer.mozilla.org/en-US/docs/Learn/JavaScript/First_steps/What_is_JavaScript)

[5]

*ExpressJS definicija, Wikipedia*

<https://sr.wikipedia.org/wiki/Express.js>