Funkcionalno reaktivno programiranje i primena u razvoju veb aplikacija

Aleksandar Stojmenović

***Sadržaj* — U ovom radu prikazani su osnovni koncepti funkcionalnog i reaktivnog programiranja. Na kraju rada je predstavljena aplikacija za studentsku anketu koja primenjuje koncepte funkcionalnog i reaktivnog programiranja. Aplikacija omogućava studentima da unesu ocene za predmete, profesore i studentsku službu, studenti unose ocenu od 1 do 5 za svako pitanje, nakon što se ocene unesu ažurira se tabela u realnom vremenu koja pokazuje koliko je studenata dalo određenu ocenu, koliko ima ukupno ocena i koja je prosečna ocena za odgovarajuće pitanje.**

Tehnologije koje su korišćene za izradu ove aplikacije su Node.js i Express.js na backend-u, MySQL baza i na frontend-u je korišćen Angular sa RxJS bibliotekom.

***Ključne reči* — Funkcionalno programiranje, reaktivno programiranje, funkcionalno reaktivno programiranje, RxJS.**

I. Uvod

FUNKCIONALNO programiranje je deklarativna programska paradigma gde se programi kreiraju sastavljanjem čistih funkcija.

Svaka funkcija uzima ulaznu vrednost i vraća odgovarajući izlaz, bez promene ili uticaja stanja programa.

Ove funkcije izvršavaju jednu operaciju i mogu da se sastavljaju u sekvencama da bi izvršile neki kompleksniji zadatak. Funkcionalna paradigma čini naš kod visoko modularnim, jer se funkcije mogu ponovo koristiti u programu i mogu se pozvati, proslediti kao parametri ili vratiti iz drugih funkcija. Funkcionalno programiranje ima nekoliko važnih koncepata koje ćemo prikazati u radu.

Aleksandar Stojmenovic, Beograd, Srbija (e-mail: aleksandar.stojmenovic9@gmail.com).

Reaktivno programiranje opisuje paradigmu dizajna koja se oslanja na logiku asinhronog programiranja za rukovanje ažuriranjem podataka u realnom vremenu za statični sadržaj. Omogućava korišćenje automatizovanih tokova podataka da ažurira sadržaj kad god korisnik kreira upit.

Tokovi podataka se šalju iz izvora kao što su: senzor pokreta, merač temperature ili baze podataka kao rekacija na okidač. Taj okidač može biti događaj (engl. Event), poziv (engl. Call) ili poruka koju sistem šalje korisniku.

U ovom radu prvo će biti opisani glavni elementi funkcionalnog i reaktivnog programiranja, a zatim će biti prikazana aplikacija za studentsku anketu kroz koju će biti ilustrovana primena koncepata funkcionalnog i reaktivnog programiranja. Alikacija će sadržati tabelu u kojoj se podaci ažuriraju u realnom vremenu. Za izradu aplikacije korišćen je Node.js i Express.js na backend-u, MySQL baza, i Angular sa RxJS bibliotekom na frontend-u.

II. Funkcionalno programiranje

Funkcionalno programiranje je proces kreiranja softvera komponovanjem čistih funkcija, izbegavanjem deljenog stanja, promenjivih podataka i bočnih efekata [1, 3, 4]. Funkcionalno programiranje je radije deklarativno nego imperativno programiranje i stanje aplikacije teče kroz čiste funkcije. Za razliku od objektno-orijentisanog programiranja, gde se stanje aplikacije obično deli i nalazi zajedno sa metodama u objektima.

Funkcionalno programiranje ima drugačiji pristup razvoja softvera od objektno-orijentisanog programiranja. Objektno orijentisano programiranje je dobar alat za rešavanje raznih problema, ali ima svoje nedostatke. Paralelno programiranje u objektno-orijentisanom kontekstu je teško jer se stanje može promeniti različitim nitima sa nepoznatim nuspojavama.

Funkcionalno programiranje ne dozvoljava promenu stanja. Funkcije deluju kao gradivni blokovi u funkcionalnom programiranju. Javascript nije funkcionalni programski jezik, ali ipak koristi neke principe funkcionalnog programiranja.

Osnovni koncepti u funkcionalnom programiranju su:

- Čiste funkcije
- Kompozicija funkcije
- Izbegavanje deljenog stanja
- Izbegavanje promenljivog stanja

- Izbegavanje bočnih efekata

A. *Čiste funkcije*

Čista funkcija je funkcija koja:

- za istu ulaznu vrednost, uvek vraća istu izlaznu vrednost
- ne proizvodi bočne efekte

Čiste funkcije su potpuno nezavisne od spoljašnjeg stanja, i kao takve, imune su na čitave klase grešaka koje imaju veze sa zajedničkim promenljivim stanjem. Na slici 1 prikazan je primer čistih (pure) i nečistih (impure) funkcija. U primeru vidimo da sve nečiste funkcije koriste promenljive koje se nalaze izvan funkcije, dok čista funkcija koristi samo parametar koji je prosleđen.

```
function impure(arg){
    return obj.total * 5
}

function impure(arg){
    let obj = obj.total * arg
}

function impure(arg){
    obj.total = 100
    return arg * obj.total
}

function pure(arg){
    return arg * 5
}
```

Sl. 1. Primer nečistih i čistih funkcija

B. *Mutacija*

Objekti kao argumenti u Javascript-u su reference, što znači da ako bi funkcija mutirala svojstvo objekta ili niza, to bi mutiralo i stanje koje je dostupno van funkcije. Čiste funckije ne smeju da mutiraju spoljno stanje.

U primeru sa slike 2 ako promenimo objekat newCar, promeniće se i objekat car.

```
const car = {
  model: 'Toyota',
  year: 2020
  };

const newCar = car;
  newCar.model = 'BMW';
```

Sl. 2. Primer mutacije

C. Imperativno programiranje i deklarativno programiranje

Veb programeri najčešće koriste i imperativno i deklarativno programiranje čak i kada nisu upoznati sa terminima. Programski jezici imaju tendenciju da koriste više paradigmi mešajući imperativnu i deklarativnu sintaksu.

Proceduralno i objektno-orijentisano programiranje pripada pod imperativnom paradigmom koja se koristi u jezicima kao što su C, C++, C#, PHP, Java i Asembler.

U imperativnoj paradigmi kod se fokusira na kreiranje izraza koji menjaju stanje programa stvarajući algoritme koji računaru govore šta da radi. To je usko povezano sa načinom rada hardvera. Obično kod će da koristi uslovne izraze, petlje, i nasleđivanje klasa.

Na slici 3 prikazan je primer imperativnog programiranja u JavaScript-u za množenje dva broja:

```
class Number {

    constructor(number = 0) {
        this.number = number;
    }

    multiply (x) {
        this.number = this.number * x
    }
}

const myNumber = new Number(5);

myNumber.multiply(4);

console.log(myNumber.number); //20
```

Sl. 3. Imperativno programiranje u JavaScript-u

Logički, funkcionalni i domensko-specifični jezici pripadaju pod deklarativnom paradigmom i oni nisu uvek Tjuring kompletni, kao na primer HTML, XML, CSS, SQL, ali ima i onih koji jesu kao na primer Prolog, Haskell, F# i Lisp.

Deklarativni kod se fokusira na izgradnju logike softvera, a da zapravo ne opisuje njegov tok.

Funkcionalni programski jezici zasnovano na lambda računu su Tjuring kompletno, izbegavaju postojanje stanja, bočne efekte i mutaciju podataka. Kreiraju se izrazi umesto naredbi i osnovna gradivna jedinica programa je funkcija. Umesto petlji u čistom funkcionalnom programiranju koristi se rekurzija i za isti argument funkcija će uvek da vrati istu vrednost.

Na slici 4 prikazan je primer deklarativnog koda u JavaScript-u za množenje dva broja:

```
const multiply = a => b => a * b;
console.log(multiply(5)(4)); //20
```

Sl. 4. Deklarativno programiranje

D. *Bočni efekti*

Bočni efekat je svaka promena stanja programa koju funkcija vrši izvan svog konteksta, osim njegove povratne vrednosti. Bočni efekti sadrže:

- Izmene bilo koje spoljne promenljive ili svojstva objekta (npr. Globalna promenljiva, ili promenljiva u lancu opsega roditeljske funkcije)
- Logovanje na konzoli
- Pisanje na ekranu
- Pisanje u fajl
- Pisanje na mreži
- Pokretanje bilo kojeg spoljnog procesa
- Pozivanje bilo koje druge funkcije sa bočnim efektima

E. *Funkcije višeg reda*

U Javascript-u ili bilo kojem drugom funkcionalnom programskom jeziku funkcije se posmatraju kao objekti.

Funkcije višeg reda su funkcije koje rade na drugim funkcijama, bilo da ih uzimaju kao argumente ili vraćaju ih. Jednostavnim rečima, funkcija višeg reda je funkcija koja prima funkciju kao argument ili vraća funkciju kao izlaz. Funkcije višeg reda nisu uobičajene u tradicionalnom programiranju.

Dok u imperativnom kodu programer može koristiti petlju da iterira kroz niz, u funkcionalnom programiranju bi se koristio skroz drugačiji pristup. Korišćenjem funkcije višeg reda, na nizu se može raditi primenom te funkcije na svaku stavku u nizu da bi se kreirao novi niz. Ovo je glavna ideja funkcionalnog programiranja. Ono što funkcije višeg reda dozvoljavaju je mogućnost prosleđivanja logike drugim funkcijama, baš kao i objektima.

Na primer, Array.prototype.map(), Array.prototype.filter() i Array.prototype.reduce() su neke od funkcija višeg reda u Javascript-u.

Pogledajmo primer funkcije višeg reda map() i uporedimo rešenja gde ne koristimo funkciju višeg reda (slike 5 i 6).

Metoda map() kreira novi niz pozivanjem callback funkcije koja je navedena kao argument za svaki element u ulaznom nizu. Metoda map() će uzeti svaku povratnu vrednost iz callback funkcije i kreiraće novi niz koristeći te vrednosti.

Recimo da imamo niz brojeva i želimo da kreiramo novi niz koji sadrži pet puta veću vrednost za svaki element. Na slikama 5 i 6 možemo da vidimo kako da rešimo ovaj problem bez i sa funkcijom višeg reda.

```
const arr1 = [1, 2, 3];
const arr2 = [];

for(let i = 0; i < arr1.length; i++) {
    arr2.push(arr1[i] * 5);
}

// prints [5, 10, 15]
console.log(arr2);
```

Sl. 5. Primer implementacije bez funkcije višeg reda

```
const arr1 = [1, 2, 3];

const arr2 = arr1.map(item => item * 5);

console.log(arr2);
```

Sl. 6. Primer implementacije sa funkcijom višeg reda map()

F. Deljeno stanje

Deljeno stanje je bilo koja promenljiva, objekat, ili memorijski prostor koji postoji u deljenom opsegu ili kao svojstvo objekta koji se prosleđuje između opsega. Deljeni opseg može biti globalni opseg ili closure opseg. Često, u objektno-orijentisanom programiranju, objekti su deljeni između opsega dodavanjem svojstva drugim objektima.

Funkcionalno programiranje izbegava deljeno stanje - umesto toga se oslanja na nepromenjive strukture podataka i čiste kalkulacije kako bi se dobili novi podaci od postojećih podataka.

III. Uvod u RxJS

Reaktivna ekstenzija za Javascript (RxJS) je elegantna zamena za biblioteke koje su zasnovane na callback funkcijama i Promise-ima, koristeći model programiranja koji tretira sve izvore događaja na potpuno isti način, bilo da se radi o čitanju fajla, kreiranju HTTP poziva, kliku na dugme ili pomeranju miša [2, 5, 10, 11].

RxJS je biblioteka za kreiranje asinhronih programa i programa koji su zasnovani na događajima koristeći Observable sekvence. Omogućava jedan osnovni tip Observable, satelitske tipove (Observer, Schedulers, Subjects) i operatore inspirisane dodacima za nizove kao što su (map, filter, reduce, every…) kako bi se omogućilo rukovanje asinhronim događajima kao kolekcijama.

ReactiveX kombinuje Observer patern sa Iterator paternom i funkcionalno programiranje sa kolekcijama da ispuni potrebu za idealnim načinom upravljanja sekvencama događaja.

Osnovni koncepti u RxJS-u koji rešavaju upravljanje asinhronih događaja su:

- Observable: predstavlja ideju kolekcija budućih vrednosti ili događaja koji se mogu pozvati.
- Observer: predstavlja kolekciju callback-ova koje znaju kako da osluškuju vrednosti koje je poslao Observable.
- Subscription: predstavlja izvršenje Observable-a, korisna za otkazivanje izvršenja.
- Operatori: su čiste funkcije koje omogućavaju stil funkcionalnog programiranja koje se bave kolekcijama preko operacija kao što su map, filter, concat, reduce, itd.
- Subject: je ekvivalent EventEmitter-u, jedini način za multikastovanje vrednosti ili događaja na više Observera.
- Schedulers: predstavlja centralizovani dispečer za kontrolu konkurentnosti, dozvoljavajući nam da koordiniramo kada se računanje dogodi npr. setTimeout ili requestAnimationFrame.

A. Tokovi podataka

RxJS tretira sve izvore podataka na isti način uključujući pritisak tastera, događaje miša, HTTP pozive ili pojedinačne brojeve. Ove izvore podataka u reaktivnom programiranju nazivamo tokovima.

Termin tok se koristio u programski jezicima kao apstraktni objekat koji se odnosi na I/O operacije, kao što je čitanje fajla, čitanje soketa, ili traženje podataka sa HTTP servera. U reaktivnom programiranju termin tok proširujemo tako da on podrazumeva bilo koji izvor podataka koji se može koristiti.

B. Proizvođači podataka

Proizvođači podataka su izvori naših podataka. Tok uvek mora da ima proizvođača podataka, što će biti polazna tačka za svaku logiku koju ćemo izvoditi u RxJS. U praksi, proizvođač se kreira od nečega što generiše događaje nezavisno (To može biti jedna vrednost, niz, klik mišem, tok bajtova pročitanih iz fajla). Proizvođače podataka u RxJS-u zovemo Observable.

C. Potrošači

Potrošač da prihvata događaje od proizvođača i obrađuje ih na neki specifičan način. Kada potrošač počne da osluškuje proizvođača za događaje koje će konzumirati, tada imamo tok, i u ovom trenutku tok počinje da šalje događaje. U RxJS-u potrošača zovemo Observer. Tokovi putuju samo od proizvođača do potrošača, a ne obrnuto. Drugim rečima, korisnik koji kuca na tastaturi proizvodi događaje koji teku naniže da bi ih potrošio neki drugi proces. U RxJS-u, tok će uvek teći od uzvodnog Observable-a do nizvodnog observer-a, a obe komponente su labavo povezane, što povećava modularnost naše aplikacije.

D. Observable

Observable-i su Push kolekcije više vrednosti. Observable vraća tok podataka observer-u tokom vremena [6].

Observable konstruktor uzima jedan argument: subscribe() funkciju. U primeru na slici 7 kreiran je Observable koji emituje string 'Hello, World!' svake sekunde pretplatniku.

```
import { Observable } from 'rxjs';

const observable = new Observable(function subscribe(subscriber) {
    const id = setInterval(() => {
        subscriber.next('Hello, World!')
    }, 1000);
});
```

Sl. 7. Kreiranje Observable-a

E. Observer

Observer je potrošač vrednosti koje su isporučene od Observable-a [7]. Observeri su jednostavno setovi callback funkcija, jedan za svaki tip notifikacija koje su isporučene od Observable-a: next, error i complete. Na slici 8 prikazan je primer koji predstavlja uobičajeni Observer objekat:

next (val) : void – Dobija sledeću vrednost od observable-a. Ovo je ekvivalentno ažuriranju u observer paternu. Kada se funkcija prosledi subscribe() funkciji umesto observer objekta, funkcija se mapira na funkciju observera next().

error (exception) : void – Dobija notifikaciju greške od observable-a. Ovo označava da je naišao na izuzetak i da neće više da emituje poruke observer-u. Prosleđuje se objekat greške.

complete () : void – Dobija obaveštenje o završetku od observable-a. Naredni pozivi next() se ignorišu.

Da bismo koristili Observer, moramo da mu obezbedimo pretplatu na Observable.

```
const observer = {
    next: x => console.log('Next value: ' + x),
    error: err => console.error('Error: ' + err),
    complete: () => console.log('Complete notification')
};

observable.subscribe(observer);
```

Sl. 8. Kreiranje Observer-a

F. Operatori

Jedna od prednosti RxJS-a je ta što možemo manipulisati ili uređivati podatke dok prolaze od proizvođača do potrošača [8]. Ovde se koriste Observable operatori.

Operatori nam omogućavaju da radimo razne stvari sa podacima koji su proizvedeni od Observable-a.

To uključuje manipulisanje i oblikovanje podataka kako nam odgovara, kombinovanje podataka od više Observable-a, skupljanje podataka i još mnogo toga.

Operator je funkcija koja se koristi za transformaciju i manipulaciju elemenata iz Observable toka.

Operatori mogu da se koriste i u sekvencama koristeći pipe() metodu Observable-a. Na slici 9 prikazan je primer operatora koji koriste metodu pipe().

Kreiran je Observable koristeći metodu of() koja ima elemente 3, 5, 7 i nad tim Observable-om pozvana je metoda pipe() koja u sebi ima skup operatora. Operatori se redom izvršavaju nad svakim elementom iz Observable-a. Dakle, operator map() ima ulazni Observable 3, 5, 7, uzima prvi element i množi ga sa 3, zatim taj element prosleđuje sledećem operatoru tap() koji

ispisuje u konzolu taj element i na kraju dodamo take(2) koji ubacuje samo dva elementa u novi izlazni Observable.

```
of(3, 5, 7)
    .pipe(
        map(item => item * 3),
        tap(item => console.log(item)),
        take(2)
    ).subscribe(console.log);
```

Sl. 9. Primer operatora

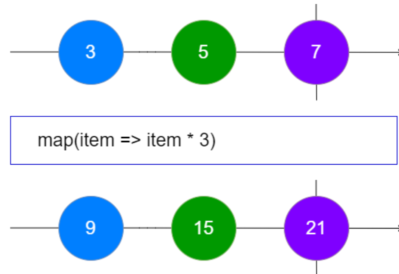
G. Marble dijagrami

Postoji mnogo operatora i ponekad je teško razumeti šta koji operator radi. Zbog toga je kreirana specijalna notacija sa ciljem da olakša razumevanje rada određenog operatora. Zovu se marble dijagrami.

Gornji deo dijagrama ima horizontalnu strelicu koja ide sa leva na desno, ovo predstavlja izvorni observable tokom vremena. Observable-i proizvode vrednosti tokom vremena i te vrednosti će biti prikazane kao krugovi na toj liniji. Na sl. 10 izvorni Observable će da proizvede vrednosti 3, 5, 7. Vertikalna linija na kraju strelice predstavlja uspešan zavrsetak Observable-a. Neki dijagrami će da modeluju greške i oni će biti prikazani na ovoj liniji kao X. A ukoliko Observable proizvodi vrednosti bez završetka, na kraju linije neće biti prikazan nikakav simbol.

Ispod linije, koja predstavlja izvorni Observable, će biti naziv operatora koji se opisuje, često zajedno sa primerom konfiguracije. Na slici 10 je prikazan marble dijagram za metodu map(). Konfiguraciona funkcija koje se prosleđuje operatoru će da pomnoži sa 3 svaku vrednost koja je proizvedena od strane izvornog observable-a.

Ispod toga se nalazi još jedna linija sa strelicom koja predstavlja Observable koji je vraćen od operatora. Krugovi na liniji predstavljaju vrednosti koje su proizvedene nakon što je operator primenjen na izvorni Observable, različite boje nam pomažu da povežemo vrednosti izvorne vremenske linije sa izlaznom vremenskom linijom.



Sl. 10. Marble dijagram za metodu map()

H. Subject i BehaviourSubject

Subject je poseban tip Observable-a koji je u isto vreme i Observable i Observer [9]. Subject ćemo da koristimo za tok akcija. Razlika između Observable-a i Subject-a je ta što svaki subscriber dobija samo jedan Observable, dok Subject može da ima više subscriber-a.

BehaviourSubject je isto što i Subject, samo što ima početnu vrednost (npr. kada imamo padajuću listu pa želimo da već bude selektovana neka opcija pre nego što mi menjamo). Na slici 11 je prikazan primer kreiranja Subject-a.

```
private categorySelectedSubject = new Subject<number>();
categorySelectedAction$ = this.categorySelectedSubject.asObservable();

onSelected(categoryId): void {
    this.categorySelectedSubject.next(+categoryId);
}

cars$ = combineLatest([
    this.carService.cars$,
    this.categorySelectedAction$
])
.pipe(
    map(([cars, categoryId]) =>
        cars.filter(car =>
            categoryId ? car.categoryId === categoryId : true
        )
    )
);
```

Sl. 11. Kreiranje Subject-a

I. Reagovanje na Add operaciju

U ovom delu je prikazano kako se reaguje na dodavanje elementa u Observable. Koristićemo 2 operatora.

Operator merge(a$, b$, c$) kombinuje više tokova podataka tako što spaja njihovu emisiju.

Operator scan((acc, curr) => acc + curr) se koristi za izračunavanje elemenata u toku podataka.

On se sastoji od acc koji predstavlja ukupan broj i curr koji predstavlja trenutni element u toku podataka. Takođe možemo da stavimo početnu vrednost za acc, kao drugi parametar za scan() metodu.

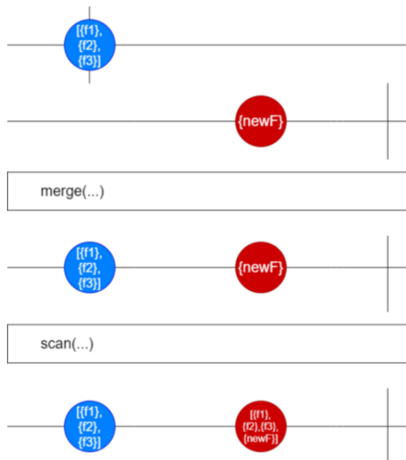
Kombinacija merge() i scan() radi odlično za dodavanje novih elemenata u tok podataka. Na slici 12 je prikazan primer dodavanja novih elemenata u tok podataka, a na slici 13 je prikazan marble dijagram za add operaciju.

U ovom primeru možemo da vidimo da imamo tok podataka koji se sastoji od 3 objekta, zatim dodajemo novi objekat u insertAction$, nakon toga spajamo naše objekte sa tokom akcija, odnosno novim objektom u novi tok podataka i na kraju odradimo scan() da ih stavimo zajedno u isti niz objekata. I na kraju će se ovaj novi element pokazati na svakoj komponenti koja je povezana sa ovim Observable-om.

```
merge(
    this.fruits$,
    this.insertAction$
)

.pipe(
    scan((acc: Fruit[],
          value: Fruit) =>
          [...acc, value])
);
```

Sl. 12. Primer add operacije



Sl. 13. Add operacija u marble dijagramu

IV. PRIMENA RXJS U PRIMERU APLIKACIJE STUDENTSKE ANKETE

U ovoj sekciji predstavljena je aplikacija za studentsku anketu koja koristi principe funkcionalnog i reaktivnog programiranja. Aplikacija omogućava studentima da unesu ocene za predmete, profesore i studentsku službu, studenti unose ocenu od 1 do 5 za svako pitanje, nakon što se ocene unesu ažurira se tabela u realnom vremenu koja pokazuje koliko je studenata dalo određenu ocenu, koliko ukupno ima ocena i koja je prosečna ocena za odgovarajuće pitanje.

Tehnologije koje su korišćene za izradu ove aplikacije su Node.js i Express.js na backend-u, MySQL baza i na frontend-u je korišćen Angular sa RxJS bibliotekom [12, 13, 14, 15].

Komponente u Angular-u ne bi trebalo da hvataju i čuvaju podatke direktno već bi trebalo da se fokusiraju na prezentovanje podataka i delegiranje pristupa podacima servisu. Zato ćemo kreirati result.service.ts servis koji će da komunicira sa backend-om, tako što će da poziva endpoint-e iz backend-a.

Napravićemo Observable koji će da get-uje odgovore ankete, odnosno id_pitanja i ocenu. Na slici 14 je prikazano kreiranja Observable-a odgovori$. Observable ćemo nazvati odgovori$, dolar se piše zbog konvencije i označava da je ta promenljiva Observable, zatim pozivamo metodu get() iz http servisa (klijenta) i prosleđujemo mu putanju endpoint-a. On vraća podatke koji će biti tipa Result[]. Na slici 15 je prikazan primer kreiranja modela Result.

```
result.service.ts

…
readonly baseURL = 'http://localhost:4000/odgovori/';

odgovori$ = this.http.get<Result[]>(this.baseURL)
  .pipe(
    catchError(this.handleError)
   );
…
```

Sl. 14. Kreiranje observable-a odgovori$

result.model.ts

```
export class Result {
    id_pitanja: number;
    ocena: number;
}
```

Sl. 15. Model Result

Nakon što smo kreirali Observable koji vraća odgovore iz baze, kreiraćemo tok akcija koji ćemo da spojimo sa tim Observable-om. Tok akcija će nam služiti da ubacimo nove ocene iz ankete u Observable koji je vratio podatke iz baze. Tako ćemo imati sve podatke zajedno preko kojih ćemo moći da izračunamo novi prosek ocena i ukupan broj ocena, nakon toga ti podaci će biti prikazani u korisničkom interfejsu u realnom vremenu.

Kreiraćemo subjekat koji će biti tipa Result, i označićemo taj subjekat sa asObservable(). Na slici 16 je prikazan primer kreiranja subjekta.

result.service.ts

```
...
private odgInsertedSubject = new Subject<Result>();
odgInsertedAction$ = this.odgInsertedSubject.asObservable();
...
```

Sl. 16. Kreiranje subjekta

Nakon toga, kreiraćemo Observable koji će da spoji odgovori$ sa odgInsertedAction$ i koji će unutar pipe() metode da izvrši scan() operator koji će da doda novi podatak u odgovori$ Observable. Kada se doda novi podatak, odgovoriAdd$ Observable će imati sve podatke kao i odgovori$ Observable plus nove podatke. Na slici 17 je prikazan primer kreiranja Observable-a odgovoriAdd$.

result.service.ts

```
...
odgovoriAdd$ = merge(
    this.odgovori$,
    this.odgInsertedAction$
  )
    .pipe(
      scan((acc: Result[], value: Result) => [...acc, value]),
      catchError(err => {
        console.error(err);
        return throwError(err);
      })
    );
...
```

Sl. 17. Kreiranje Observable-a odgovoriAdd$

Sada ćemo taj novi Observable da pozivamo iz questions.component.ts komponente da bismo učitali te podatke u komponentu. To radimo na sledeći način.

Za početak dodaćemo changeDetection: ChangeDetectionStrategy.OnPush u okviru @Component da bismo automatski učitali nove promene koje se dese u Observable-u i prikazali ih u korisničkom interfejsu. Na slici 18 je prikazan dekorator @Component.

```
questions.component.ts
...
@Component({
  selector: 'app-questions',
  templateUrl: './questions.component.html',
  styleUrls: ['./questions.component.css'],
  changeDetection: ChangeDetectionStrategy.OnPush,
  providers: [ResultService]
})
...
```

Sl. 18. Dekorator @Component

Kada smo to uradili, kreiraćemo Observable koji će da poziva odgovoriAdd$ Observable iz servisa i koji će da filtrira podatke tako što će da uzme samo odgovore na prvo pitanje. Na slici 19 prikazan Observable odgovori1$.

```
questions.component.ts
...
odgovori1$ = this.resultService.odgovoriAdd$
    .pipe(
      map(odg =>
        odg.filter(odg =>
          odg.id_pitanja === 1
        )
      )
    );
...
```

Sl. 19. Observable odgovori1$

Nakon toga, kreiraćemo Observable sum1$ koji će da izračuna zbir svih ocena koje ćemo kasnije da koristimo za izračunavanje proseka. Na slici 20 je prikazan Observable sum1$.

```
questions.component.ts___________________________________________________
...
sum1$ = this.odgovori1$.pipe(
    map(odg =>
      odg.reduce((accumulator, current) => accumulator + current.ocena, 0),
    ));

...
```

Sl. 20. Observable sum1$

Posle toga, kreiraćemo Observable count1$ koji će da prebroji koliko ocena ima to pitanje. Na slici 21 je prikazan Observable count1$.

```
questions.component.ts___________________________________________________
...
count1$ = this.odgovori1$.pipe(
    map(odg =>
      odg.length
      )
    );

...
```

Sl. 21. Observable count1$

Da bismo izračunali prosek ocena, kreiraćemo average1$ Observable koji će da izracčna prosek ocena za prvo pitanje. Koristićemo metodu combineLatest() za kombinovanje sum1$ i count1$ Observable-a. Na slici 22 je prikazan Observable average$.

```
questions.component.ts___________________________________________________
...
average1$ = combineLatest([
        this.sum1$,
        this.count1$
      ]).pipe(
        map(([sum, count]) => sum / count)
    );

...
```

Sl. 22. Observable average$

Zatim, da prikazemo recimo broj petica za prvo pitanje, kreiraćemo Observable ocena5$ koji će da filtrira ocene za prvo pitanje, tako što će da uzme samo petice. Na slici 23 je prikazan Observable ocena5$.

```
questions.component.ts
...
ocena5$ = this.odgovori1$.pipe(
    map(odg =>
      odg.filter(odg =>
        odg.ocena === 5
    )));
...
```

Sl. 23. Observable ocena5$

Sada kada smo vratili sve petice, napravićemo Observable brojOcena5$ koji će da prebroji ocene. Postupak je isti za sve ocene. Na slici 24 je prikazan Observable brojOcena5$.

```
questions.component.ts
...
brojOcena5$ = this.ocena5$.pipe(
    map(odg =>
      odg.length
    )
  )
...
```

Sl. 24. Observable brojOcena5$

Kada smo to uradili, potrebno je da podatke prikažemo u template-u, to ćemo uraditi na sledeći način. Na primer za prosek ocena, dodaćemo *ngIf="average1$ | async as avg1" koji se povezuje na average1$ Observable ako postoji, koristićemo async da bi se podaci automatski ažurirali kada se desi neka promena u Observable-u. Postupak je isti i za ostale podatke. Na slici 25 je prikazano povezivanja Observable-a sa template-om.

```
questions.template.html_____________________________________________________
...
<table class="responsive-table highlight">
    <thead>
        <tr>
            <th>Pitanje</th>
            <th>1</th>
            <th>2</th>
            <th>3</th>
            <th>4</th>
            <th>5</th>
            <th>Ukupno</th>
            <th>Prosecna ocena</th>
        </tr>
    </thead>
        <tr>
            <td>Pitanje 1</td>
            <td *ngIf="brojOcena1$ | async as ocena1">{{ocena1}}</td>
            <td *ngIf="brojOcena2$ | async as ocena2">{{ocena2}}</td>
            <td *ngIf="brojOcena3$ | async as ocena3">{{ocena3}}</td>
            <td *ngIf="brojOcena4$ | async as ocena4">{{ocena4}}</td>
            <td *ngIf="brojOcena5$ | async as ocena5">{{ocena5}}</td>
            <td *ngIf="count1$ | async as count1">{{count1}}</td>
            <td *ngIf="average1$ | async as avg1">{{avg1 | number:'1.2-
            2'}}</td>
        </tr>

        ...
</table>
...
```

Sl. 25. Povezivanje Observable-a sa template-om

Pogledajmo kako sve to izgleda u korisničkom interfejsu. Podaci se ažuriraju u realnom vremenu, kad god se unese neka ocena. Unećemo ocene za prvo i drugo pitanje. Na slici 26 je prikazan interfejs pre unosa ocena.

Ocenite profesora iz Dizajna i analize algoritama?
5

Ocenite rad studentske sluzbe?
5

Ocenite predmet Masinsko ucenje?
Unesite ocenu od 1 do 5

Submit

Pitanje	1	2	3	4	5	Ukupno	Prosecna ocena
Pitanje 1	19	15	23	58	79	194	3.84
Pitanje 2	11	10	26	27	96	170	4.10
Pitanje 3	18	7	16	40	83	164	3.99

Sl. 26. Primer korisničkog interfejsa pre unosa ocena

Kada smo uneli ocene, možemo da vidimo da se tabela ažurirala, vidimo da se promenio broj petica za prvo i drugo pitanje, ukupan broj ocena i prosek ocena. Na slici 27 je prikazan primer korisničkog interfejsa nakon

unosa ocena.

Ocenite profesora iz Dizajna i analize algoritama?
5

Ocenite rad studentske sluzbe?
5

Ocenite predmet Masinsko ucenje?
Unesite ocenu od 1 do 5

Submit

Saved successfully

Pitanje	1	2	3	4	5	Ukupno	Prosecna ocena
Pitanje 1	19	15	23	58	80	195	3.85
Pitanje 2	11	10	26	27	97	171	4.11
Pitanje 3	18	7	16	40	83	164	3.99

Sl. 27. Primer korisničkog interfejsa nakon unosa ocena

V. Zaključak

Funkcionalno programiranje je deklarativna programska paradigma gde se programi kreiraju sastavljanjem čistih funkcija. Svaka funkcija uzima ulaznu vrednost i vraća odgovarajući izlaz, bez promene ili uticaja na stanje programa. Ove funkcije izvršavaju jednu operaciju i mogu da se sastavljaju u sekvencama da bi izvršile neki kompleksniji zadatak. Funkcionalna paradigma čini naš kod visoko modularnim, jer se funkcije mogu ponovo koristiti u programu i mogu se pozvati, proslediti kao parametri ili vratiti.

Čiste funkcije se lakše razumeju jer ne menjaju nijedno stanje i zavise samo od unosa koji im je dat. Koji god izlaz da proizvedu, to je povratna vrednost koju daju. Njihova oznaka funkcije daje sve informacije o njima, odnosno njihov povratni tip i njihove argumente.

Sposobnost funkcionalnih programskih jezika da tretiraju funkcije kao vrednosti i prosledjuju ih drugim funkcijama kao parametre, čine kod čitljivijim i lako razumljivim.

Testiranje i debagovanje u funkcionalnom programiranju je lakše, budući da čiste funkcije uzimaju samo argumente i proizvode izlaz, oni ne proizvode nikakve promene, ne uzimaju unos i ne stvaraju neki skriveni izlaz. Koriste nepromenjive vrednosti, tako da je lakše proveriti neke probleme u programima koji su napisani koristeći čiste funkcije.

Funkcionalno programiranje se najviše koristi kod konkurentnosti i paralelizma, jer čiste funkcije ne menjaju promenljive ili bilo koje druge podatke izvan nje.

Reaktivno programiranje opisuje paradigmu dizajna koja se oslanja na logiku asinhronog programiranja za rukovanje ažuriranja podataka u realnom vremenu za statični sadržaj. Omogućava korišćenje automatizovanih tokova podataka da ažurira sadržaj kad god korisnik kreira upit.

Tokovi podataka se šalju iz izvora kao što su: senzor pokreta, merač temperature ili baze podataka kao rekacija na okidač. Taj okidač može biti dogadjaj (engl. Event), poziv (engl. Call) ili poruka koju sistem šalje korisniku.

Reaktivno programiranje nam omogućava jednostavno sastavljanje tokova podataka, pomoću njega kreiramo, filtriramo i kombinujemo tokove podataka koji emituju vrednost, grešku i završen signal.

U radu je predstavljena aplikacija za studentsku anketu gde smo primenili koncepte funkcionalnog i reaktivnog programiranja, prikazana je tabela gde se podaci ažuriraju u realnom vremenu. Koristili smo Node.js i Express.js na backend-u, MySQL bazu, i Angular sa RxJS bibliotekom na frontend-u.

Reaktivno programiranje je izuzetno pogodno za korišćenje kod aplikacije u kojima je potrebna izmena prikaza u realnom vremenu.

Ukoliko bi se ova aplikacija nastavila dalje razvijati dodale bi se nove funkcionalnosti kao što su registracija i logovanje korisnika, profilne strane korisnika na kojoj se nalaze sve ankete koje je korisnik popunio, kreiranje novih anketa, naslovnu stranu na kojoj se nalazi lista anketa, pretraživanje anketa.

LITERATURA

[1] Ved Antani, Simon Timms, Dan Mantyla, “JavaScript: Functional Programming for JavaScript Developers “, ISBN 978-1-78712-466-0
[2] Paul P. Daniels, Luis Atencio, “RxJS in Action”, ISBN 978-1-61729-341-2
[3] Funkcionalno programiranje. Available: https://medium.com/javascript-scene/master-the-javascript-interview-what-is-functional-programming-7f218c68b3a0
[4] Funkcionalno programiranje. Available: https://www.telerik.com/blogs/functional-programming-javascript
[5] RxJS Overview. Available: https://rxjs.dev/guide/overview
[6] RxJS Observable. Available: https://rxjs.dev/guide/observable
[7] RxJS Observer. Available: https://rxjs.dev/guide/observer
[8] RxJS Operators. Available: https://rxjs.dev/guide/operators
[9] RxJS Subject. Available: https://rxjs.dev/guide/subject
[10] RxJS Course. Available: https://www.pluralsight.com/courses/rxjs-getting-started
[11] RxJS Course. Available: https://www.pluralsight.com/courses/rxjs-angular-reactive-development
[12] NodeJS. Available: https://nodejs.org/en/
[13] Angular Framework. Available: https://angular.io/
[14] Express.js Framework. Available: https://expressjs.com/

[15] MySQL Database. Available: https://www.mysql.com/

ABSTRACT

This paper presents the basic concepts of functional and reactive programming. At the end of the paper, an application for a student survey is presented, which applies the concepts of functional and reactive programming. The application allows students to enter grades for subjects, professors and student service, students enter a grade from 1 to 5 for each question, after the grades are entered the table is updated in real-time showing how many students gave a certain grade, how many grades are there and average grade for the corresponding question.

The technologies used to create this application are Node.js and Express.js on the backend, MySQL database, Angular with the RxJS library on the frontend.

Key words - Functional programming, reactive programming, functional reactive programming, RxJS.

FUNCTIONAL REACTIVE PROGRAMMING

Aleksandar Stojmenović