Uvod u reaktivnu programsku paradigmu korišćenjem radnog okvira SPRING

Nemanja Zirojević[1]

Sadržaj – U cilju sticanja sveobuhvatne slike o uzajamnim sličnostima, razlikama, prednostima i manama tradicionalnog i reaktivnog pristupa rešavanja nekih od problema sa kojima se susrećemo moderne veb aplikacije, kao i mogućnostima koje nam pruža reaktivni modul popularnog radnog okvira Spring (eng. *"framework"),* u ovom radu biće demonstriran razvoj veb aplikacije za upravljanje akcijama kompanija na berzi, koristeći oba od pomenutih pristupa.

Ključne reči – Reaktivno programiranje, Spring framework, Spring WebFlux

I UVOD

U istoriji razvoja softverske industrije postojao je period kada je većinu softverskih aplikacija karakterisalo vreme odziva od više sekundi, više sati održavanja van mreže, manji broj klijenata pa samim tim i manji protok podataka[1].Pojavom novih uređaja (mobilni uređaji, tableti i tako dalje) i najnovijeg načina pristupa (zasnovanog na cloud-u), kao i sve veće konkurencije među softverskim kompanijama, pojavili su se i novi izazovi i zahtevi za veb aplikacije. Neki od najvažnijih izazova sa kojima se susrećemo moderne veb aplikacije su:

1. Vreme odziva u sekudni,
2. Stopostotna dostupnost,
3. Eksponencijalno povećanje obima podataka,
4. Visoka otpornost na kvarove,
5. Pružanje što boljeg korisničkog iskustva.

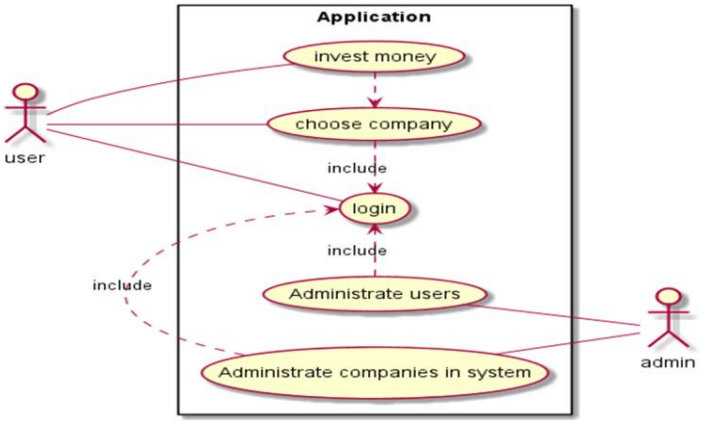
[1] Nemanja Zirojević, Računarski fakultet, Srbija (e mail: nzirojevic@raf.rs).

Postoje različiti pristupi rešavanja ovih izazova, a jedan od njih je reaktivno programiranje. Reaktivno programiranje je usmereno na tok podataka, odnosno na prenošenje izmena u trenutku promene podataka. U drugom delu ovog rada daje se kratak opis funkcionalnosti aplikacije i pregled razlika između reaktivnog i tradicionalnog načina razvoja. U delu III akcenat je na opisu implementacije reaktivnog toka podataka korišćenjem *Spring WebFlux* tehnologije, dok se u delu IV razmatra implementacija klijentske strane aplikacije.

II RAZLIKE IZMEĐU TRADICIONALNOG I REAKTIVNOG PRISTUPA

A. Opis aplikacije

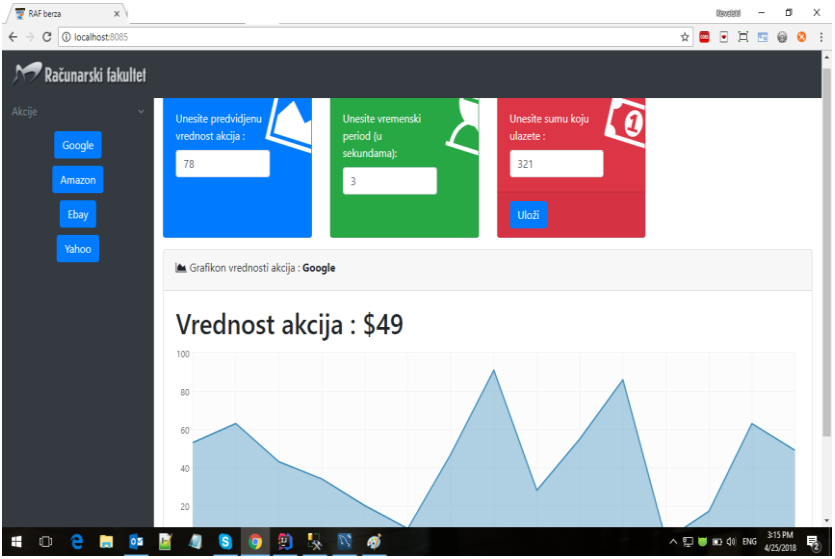
Na slici 1 prikazan je dijagram slučajeva korišćenja koji opisuje funkcionalnost aplikacije za upravljanje akcijama na berzi.



Sl.1. Dijagram slučajeva korišćenja

Kao što se vidi sa dijagrama, registrovani korisnici aplikacije su u mogućnosti da inverstiraju novac u akcije izabrane kompanije. Prilikom registracije svaki korisnik kreira svoj jedinstveni nalog koji pored podataka o korisniku (ime,prezime,email,lozinka), sadrži i podatke o sumi novca kojom korisnik raspolaže. Korisniku je omogućeno da prognozira cenu akcija koja će biti nakon isteka unetog vremena i da unese sumu novca koju ulaže i koja će se na kraju smanjiti odnosno uvećati za 90% u zavisnosti od toga da li je korisnik pogodio cenu akcija ili ne.Cene akcija za odabranu kompaniju

prikazuju se korisniku u vidu grafikona, pri čemu se izgled grafikona menja u zavisnosti od promene cena akcija, kao što je prikazano na slici 2.



Sl.2.Izgled početne strane aplikacije

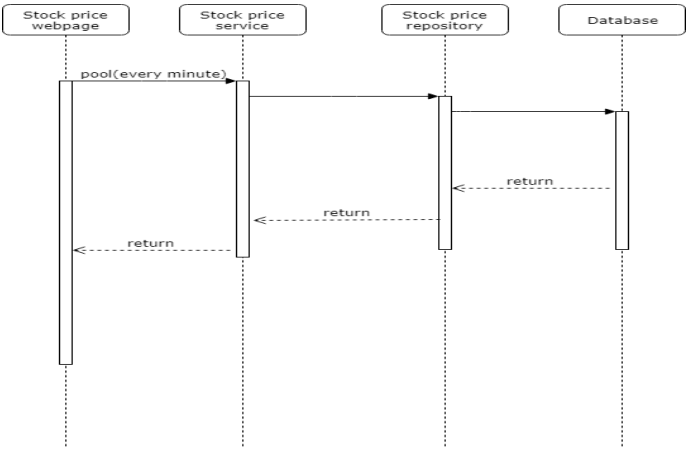
Pretpostavićemo da je aplikacija monolitna, troslojna po strukturi i sastoji se od sledećih slojeva:

1. Prezentacioni (klijentski) sloj,
2. Sloj poslovne logike,
3. Sloj podataka.

S obzirom da je fokus ovogpoglavlja u pregledu mehanizama tradicionalnog i reaktivnog načina razvoja opisane aplikacije, detaljivezani za implementacijugore navedenih slojeva nece bitiprikazivani. Takođe, pretpostavka je da je serverski deo aplikacije razvijan korišćenjem radnog okvira *Spring*.

B. Tradicionalni pristup razvoju aplikacije

Tradicionalni pristup se zasniva na proveri da li je cena akcije promenjena. Na slici 3 preuzetoj iz članka [1] prikazan je dijagram sekvenci koji ilustruje pomenuti pristup razvoju aplikacije.



Sl.3. Tradicionalni pristup razvoju aplikacije

Da bi se proverila cena akcija, na klijentskoj strani se u određenim vremenskim intervalima (na slici 3, taj interval iznosi jedan minut) izvršava AJAX poziv kome se kao parametar prosledi URL adresa metoda na serverskoj strani, koji vraća trenutnu cenu akcija. Ovakav način razvoja aplikacije ima svojih manjkavosti, pre svega u pogledu performansi i pružanja dobrog korisničkog iskustva. U pomenutom načinu razvoja aplikacije, na klijentskoj strani ne postoji nikakva kontrola nad promenama cena akcija koje se dešavaju, što znači da se mogu desiti situacije da se AJAX pozivi izvršavaju i u slučaju kada cene akcija nisu promenjene.

C. Reaktivni pristup razvoju aplikacije

Za razliku od opisanog načina razvoja aplikacije, reaktivni pristup podrazumeva reagovanje na događaje čim se dese. U ovom slučaju, kontrolu prikazivanja novih cena akcija, odnosno promene svoga stanja, preuzima komponenta na klijentskoj strani, koja osluškuje promene na serveru i na osnovu toga menja sopstveno stanje. To znači da kada se desi događaj promene cene akcije, tada se automatski reaguje na izmenu i najnovija cena akcije je ažurirana na veb stranici. Ovakav tip aplikacionog programskog interfejsa (eng.“*API*“) naziva se *callback-based API*. Reaktivni pristup obično uključuje tri koraka [1]:

1. Prijavljivanje za događaje,
2. Odigravanje događaja,

3. Odjavljivanje.

U reaktivnom pristupu razvoju aplikacije, kada je učitana, veb stranica za cenu akcija će se prijaviti za događaj promene cene akcija. Kada se desi događaj promene cene za specifične akcije, novi događaj je pokrenut za sve prijavljene članove događaja. Osluškivači će obezbediti da se veb stranica ažurira i da prikaže najnoviju cenu akcije. Kada je veb stranica zatvorena (ili osvežena), zahtev za odjavu je poslat do prijavljenog člana.

Tradicionalni pristup je veoma jednostavan, dok reaktivni zahteva implementiranje reaktivne prijave i lanca događaja. Ako lanac događaja uključuje posrednika za poruke, postaje još složeniji. Prava snaga reaktivnog načina razvoja aplikacija leži u neblokirajućem načinu obrade korisničkih zahteva. U poređenju sa tradicionalnim načinom obrade, gde se zahtevi obrađuju sihrono,jedan za drugim,kod reaktivnog programiranja zahtevi se obrađuju asihrono. U praktičnom primeru to bi značilo, da ukoliko aplikacija poziva neki eksterni API, ili je to poziv ka bazi, u reaktivnom načinu pristupa to može da se odradi tako da trenutna programska nit koja se izvršava ne blokira resurse,niti druge niti, što znači da je moguće obrađivati više zahteva kroz više programskih niti.U blokirajućem pristupu, bila bi moguća samo jedna programska nit koja bi obrađivala svaki zahtev posebno, odnosno kada krene izvršavanje jednog zahteva trenutna programska nit koja se izvršava blokira izvršavanje svih ostalih, tako da se obrađivanju idućeg zahteva može pristupiti tek nakon što se obradi predhodni zahtev. Reaktivni način razvoja aplikacije pruža mogućnost optimalnijeg korišćenja resursa. Kako je životni vek programskih niti u tradicionalnom pristupu duži, samim tim resursi koje koristi programska nit traju duže. Ako razmotrimo server koji obrađuje više zahteva u isto vreme, vidimo da je veća konkurentnost za programske niti i njihove resurse. U rekativnom pristupu programske niti traju kratko, pa, prema tome, manja je konkurentnost za resurse. Takođe, koristeći opisani pristup izradi aplikacije komponente aplikacije postaju slabije povezane, što vodi većoj otpornosti celokupne aplikacije na eventualne probleme koji mogu nastati u nekom od njenih servisa. . Dakle, ideja kod izgradnje reaktivnog sistema sastoji u kreiranju lanca događaja, pri čemu svaka karika u tom lancu reaguje događaj iz predhodne karike i na kraju se podaci prikazuju korisniku.

Skaliranje u tradicionalnom pristupu podrazumeva potrebu za većim hardverskim resursima, ili za dodavanjem više paralenih veb servera koji se zatim skaliraju po potrebi. Iako reaktivni pristup ima sve opcije skaliranja kao i tradicionalni, obezbeđuje više distribuiranih ocija. Na primer, pokretanje događaja promene cene akcija može da se prijavi pomoću

posrednika za poruke. To znači da veb aplikacija i aplikacija pokrenuta promenom cene akcije mogu da budu skalirane nezavisno jedna od druge, što daje više opcija za brzo skaliranje, a samim tim i bržeg postavljanja aplikacije u produkciono okruženje. Ova prednost posebno dolazi do izražaja kada su u pitanju aplikacije dizajnirane u mikroservisnoj arhitekturi. Postoji veliki broj radnih okvira koji obezbeđuju reaktivne funkcije, jedna od takvih i *Spring WebFlux*.

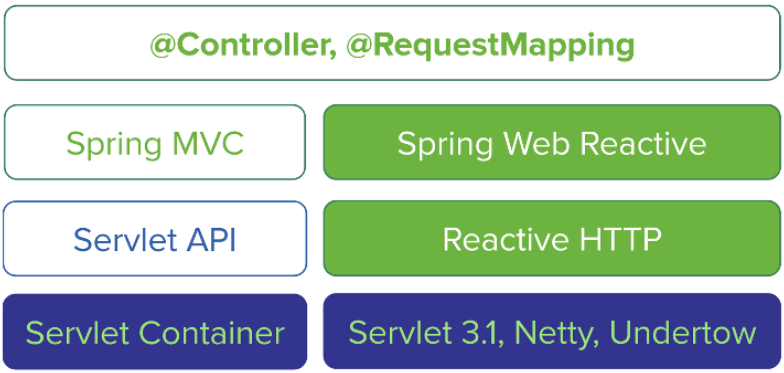
III IMPLEMENTACIJA REKATIVNOG TOKA PODATAKA

Spring *WebFlux*[2] koristi biblioteku *Reactor* kao svoju reaktivnu podršku.*Reactor* je implementacija *Reactive Streams*[3] specifikacije koja obezbeđuje dva glavna tipa za rad sa reaktivnim tokovima podataka: *Mono* i *Flux*. Oba navedena tipa implementairaju *Publisher* interfejs koji se nalazi u sklopu *Reactive Streams* specifikacije. Spring *WebFlux*pruža podršku za kreiranje rekativnih, serverskih veb aplikacija, ali takođe ima klijentske biblioteke za poziv različitih REST servisa. *Flux* predstavlja reaktivni tok podataka koji emituje 0 do n elemenata,dok s druge strane *Mono* predstavlja reaktivni tok koji emituje 0 ili 1 element. U većini slučajeva, kada se rade neka izračunavanja ili se poziva neki eksterni servis koji vraća tačno jedan rezultat, a ne kolekciju koja može da sadrži više rezultata, tada se po konvenciji koristi *Mono*.

Bitno je napomenuti da svrha korišćenja reaktivnih tipova nije omogućavanje bržeg obrađivanja korisničkih zahteva ili podataka, naprotiv, upotreba reaktivnih tipova može da dovede do sporije obrade u odnosu na tradicionalno, blokirajuće procesiranje. Snaga reaktivnih tokova leži u njihovom kapacitetu da služe većem broju zahteva istovremeno, i da operacije obavljaju latentno iefikasnije. Reaktivni API potražuje samo onu količinu podataka koju može da procesira i donosi nove mogućnosti, pošto radi sa tokom podataka a ne sa svakim elementom ponaosob. Za reaktivnu podršku i korišćenje navedenih klasa, potrebno je da se u pom.xml fajl (ukoliko se koristi Apache Maven[4] kao *dependency management* alat) dodati sledeći *dependency* :

```
<dependency>
<groupId>org.springframework.boot</groupId>
<artifactId>spring-boot-starter-webflux</artifactId>
<version>2.0.0.RELEASE</version>
</dependency>
```

Odnos između *Spring MVC* i *Spring Web Reactive*-aprikazan je na slici ispod.



Sl. 4.Odnos između Spring Web MVC i Spring Web Reactive-a

Spring *Web Reactive*[5], podržava konfiguraciju zasnovanu na anotacijama, isto kao i *Spring Web MVC*. *Spring Web Reactive* koristi neblokirajuću verziju 3.1 java servleta, a takođe kompaltibilan je sa asihronim radnim okvirima kao što je Netty i može da se izvršava na veb serverima kao što je Undertown.*Spring Web Reactive*redefiniše mnoge intefejse *Spring MVC-a* kao što su *HandlerMapping* i *HandlerAdapter* da budu asihroni, neblokirajući i da operišu nad reaktivnim HTTP zahtevima i odgovorima. Za početak potrebno je kreirati klasu koja će obrađivati korisničke zahteve i vraćati odgovarajući odgovor na klijentsku stranu. Ta klasa se naziva kontroler klasom i označava se sa *@Controller* ili *@RestController*anotacijom koja *Spring-u* "govori" da se radi o klasi sa opisanim zaduženjima. Razlika između pomenutih anotacija je u tome što se prilikom upotrebe *@RestController*anotacije vrši automatska konverzija objekata u JSON string, dok to nije slučaj prilikom upotrebe *@Controller* anotacije. Kreiranje reaktivnog kontrolera je veoma slično kreiranju Spring MVC kontrolera. Osnovna struktura je ista: klasa označena sa *@RestController* i različite anotacije *@RequestMapping* koje služe za mapiranje URL zahteva na odgovarajući metod koji se izvršava. Razlika je u tome što reaktivni kontroler prilikom izvršavanja svojih metoda vraća instance tipa *Flux<?> ili Mono<?>*. Primer implementacije reaktivnog kontrolera prikazan je na slici 5.

```
@RestController
@RequestMapping("/rest")
public class ReactiveRequestsController {

@Autowired
StockService stockService;

@RequestMapping(value = "/stock/events/{stockId}",method =
RequestMethod.GET, produces =
MediaType.TEXT_EVENT_STREAM_VALUE)
   Flux<StockEvent>getStockPrice(@PathVariable("stockId") String
stockId)
   {
      Random rand = new Random();
      return stockService.findById(stockId)
            .flatMapMany(stock -> {
               Flux<Long> interval = Flux.interval(Duration.ofSeconds(2));
Flux<StockEvent> stockEventFlux = Flux.fromStream(Stream.generate(()-
>new StockEvent(stock,rand.nextInt(98) + 1)));
               return Flux.zip(interval,stockEventFlux).map(Tuple2::getT2);
});
};

}
```

Sl.5. Reaktivni kontroler

Primetimo da metod *getStockPrice*na osnovu prosleđenog id-a kompanije, pronalazi kompaniju sa tim id-jem pozivom metoda *findById(String stockId)*klase *StockService*i vraćatok tipa *Flux<StockEvent>*, pri čemu je *StockEvent* korisnički definisana klasa, koja služi za smeštanje jedinstvenog id-a, nazivai trenutne cene akcijakomapanije u objekat, koji se šalje na klijentsku stranu aplikacije. U primeru iznad pretpostavka je da je *StockService* klasa koja pripada poslovnom sloju aplikacije i koji sadrži biznis logiku za vraćanje naziva kompanije na osnovu prosleđenog id-a. Prinicip sa slučajnim brojevima korišćen je u cilju simulacije izmene stanja na berzi, dok bi u realnom scenariju, cene akcija za pojedinačne kompanije pozivanjem eksternog servisa bile unošene u bazu u trenutku njihove promene, odakle bi se dalje povlačile i prikazivale krajnjem korisniku. U tu srhu, gore opisani metod generiše slučajne brojeve koji predstavljaju simulaciju promene stanja na berzi, odnosno promene cena akcija izabrane kompanije, u intervalu od dve sekunde. Primetimo da ovaj metod vraća

sadržaj u formatu tipa *text/event-stream*. Ovo znači da se objekti tipa *StockEven*t šalju na klijentsku stranu u sledećem obliku:

data : {"id":"1","price":"89","name" : "Google"}

data : {"id":"2","price":"29","name" : "Amazon"}

Podrazumevano, *Flux<StockEvent>*predstavlja tok podataka, dakle vraća tip *Stream,* pa je za očekivati da se na klijentsku stranu prosleđuje tok individualnih JSON objekata, međutim kao što se vidi to nije slučaj, pošto se na klijentsku stranu prosleđuje JSON niz. Razlog tome je što bi u slučaju da se tok individualnih JSON objekata prosleđuje, to ne bi bio validan JSON dokument posmatran kao celina. U slučaju prototipske aplikacije, kada korisnik izabere neku od ponuđenih kompanija, mehanizam izvršavanja je sledeći :

1. Na klijentskoj strani vrši se "prijava" za događaje na URL putanji */stock/events/{stockId},* pri čemu se varijabla *stockId* zamenjuje id-em izabrane kompanije.
2. Pozivom metoda *stockService.findById(stockId)* pronalazi se kompanija sa prosleđenim ID-jem i pravi se vremenski interval od dve sekunde, *Flux<Long> interval = Flux.interval(Duration.ofSeconds(2))*, čime se obezbeđuje da se novi tok cena akcije *Flux<StockEvent> stockEventFlux = Flux.fromStream(Stream.generate(()->new StockEvent(stock,rand.nextInt(98) +1)))* formira svake dve sekunde, i vraća korisniku return *Flux.zip(interval,stockEventFlux),*
3. Kada se desi promena cene akcija, što je svake dve sekunde, tada se JSON niz vraća do klijentske strane što dovodi do promene izgleda grafikona, odnosno reakcije na događaj.

Jedan od načina "prijave" na događaje, kao što je opisano u prvom koraku, je korišćenjem *EventSource* javascript interfejsa. Način na koji klijentska strana "prihvata" poslati JSON niz, kao i detalje kreiranja instance *EventSource* interfejsa razmotrićemo u narednom poglavlju.

IV IMPLEMENTACIJA KLIJENTSKE STRANE APLIKACIJE

Instanca*EventSource* javascript interfejsa otvara permanentnu konekciju ka serveru koji, kao što je napomenuto u predhodnom poglavlju, šalje događaje (*eng."events"*) u *text/event-stream* formatu.Na slikama 6a- 6dprikazna je

javascript funkcija na klijentskoj strani aplikacije, koja se poziva izborom neke od kompanija iz panela *"Akcije"*(slika 2).

```
function showStockPrice(id)
{
var optionsChanged = 0;
var plot;
var count = 0;
var options = { grid: {
borderColor: "#f3f3f3",
borderWidth: 1,
hoverable: true,
backgroundColor: '#fcfcfc',
tickColor: "#f3f3f3",},
series:{shadowSize:0,color:"#3c8dbc"}, lines: {fill:true,
color: "#3c8dbc"}, yaxis: {
min: 0,
max: 100,
show: true},
xaxis: {
show: true
}
    };
```

Sl.6a Funkcija za kreiranje i prikaz grafikona

```
if (!!window.EventSource)
    {
        source.addEventListener('message', function (e)
         {
if(optionsChanged==0)
    {
var object = JSON.parse(e.data);
update(parseInt(object.stockPrice, 10));

}
else if(optionsChanged==1)
    {

var moneySum = $("#input-value").val();
var time = $("#input-time").val();
var options = plot.getOptions();
options.grid.markings=[{ yaxis: { from: moneySum, to:
moneySum }, color: "#000000" }];
plot.options = options;
plot.setupGrid();
plot.draw();
var object = JSON.parse(e.data);
```

```
update(parseInt(object.stockPrice, 10));
var selectedPrice = parseInt(object.stockPrice, 10);
reset(selectedPrice,time);

}
if(optionsChanged==2)
    {
var options = plot.getOptions();
options.grid.markings=[];
plot.options = options;
plot.setupGrid();
plot.draw();
var object = JSON.parse(e.data);
update(parseInt(object.stockPrice, 10));

}

}, false);
```

Sl.6b Funkcija za kreiranje i prikaz grafikona

```
    source.addEventListener('open', function (e) {
    }, false);

source.addEventListener('error', function (e) {
if (e.readyState == EventSource.CLOSED) {
        }
    }, false);

} else {

}
var ypt = [], totalPoints = 15;

function initData() {
for (var i = 0; i < totalPoints; ++i)
            ypt.push(0);
return getPoints();
}
function getData(data) {
if (ypt.length >0)
            ypt = ypt.slice(1);
ypt.push(data);
return getPoints();
}
function getPoints() {
var ret = [];
for (var i = 0; i < ypt.length; ++i)
```

```
            ret.push([i, ypt[i]]);
return ret;
}

plot = $.plot($("#placeholder"), [initData()], options);

function update(data)
    {
        count++;
$('#priceHolder').text('$' + data);
plot.setData([getData(data)]);
plot.draw();

}
```

Sl.6c Funkcija za kreiranje i prikaz grafikona

```
var button = document.querySelector('#addLine');
button.addEventListener('click',function () {
        optionsChanged = 1;
count = 0;

});

function reset(stockPrice,time)
    {
var inputValue = $("#input-value").val().trim();
if(count == time)
        {
if (stockPrice < inputValue) {
console.log("You lose");
var moneyInput = $("#input-money").val().trim();
var newAmount = moneyInput - ( moneyInput * 90)/100;
$("#input-money").val(parseInt(newAmount));
optionsChanged = 2;
count = 0;
}
if (stockPrice >= inputValue)
            {
console.log("You won");
var moneyInput = $("#input-money").val().trim();
var newAmount = Number(moneyInput) + Number(( moneyInput
* 90)/100);
console.log("New amount :"+newAmount);
$("#input-money").val(parseInt(newAmount));
optionsChanged = 2;
count = 0;
}
```

```
        }
    }
};
```

Sl.6d Funkcija za kreiranje i prikaz grafikona

Primetimo da za instancu EventSource interfejsa vežemo tri osuškivača događaja kao što je prikazano na slici 7.

```
source.addEventListener('message', function (e){
     // code }, false);
source.addEventListener('open', function (e) {
     // code}, false);
source.addEventListener('error', function (e) {
     // code }, false);
```

Sl.7. Osluškivači događaja

U trenutku kada dođe do promene akcije na serveru, aktivira se osluškivač događaja koji reaguje na poruke *("message")* i izvršava se metod koji se navodi kao njegov drugi argument, dok se u slučaju otvaranja konekcije aktivira metod koji se navodi kao drugi argument za osluškivač koji se aktivira na događaj otvaranja konekcije *("open").* Takođe, kao što se vidi, definisan je i osluškivač koji se aktivira u slučaju pojave greške. U okviru metoda koji se navodi kao drugi argument osluškivača na događaje tipa *"message"* navodi se logika za kreiranje i izmenu grafikona na osnovu prispelih podataka. Incijalno javascript varijabla *optionsChanged* se postavlja na vrednost nula, što znači da se prilikom izbora kompanije za kreiranje grafikona njenih akcija koristi blok kodau okviru *if* naredbe koja se izvršava u slučaju kada ta varijabla ima vrednost nula. Prispeli JSON niz sa servera se parsira i čuva u objektu, koji se prosleđuje *update* funkciji koja postavlja nove podatke u grafikon i menja teksta u HTML paragraf elementu sa id-jem *placeHolder* imenom izabrane kompanije. U slučaju da korisnik izabere ulaganje novca, odnosno klikom na dugme *"Uloži"*(slika 2),vrednost *optionsChanged*varijable se postavlja na jedinicu i aktivira se blok koda u okviru if narebe koja se izvršava u tom slučaju. U okviru toga bloka naredbi vrši se prihvatanje unete sume novca i vremena i kreira se novi niz sa podešavanjima za prikaz grafikona kojima se kreira horizontalna linija paralelna sa x osom grafikona na visini prognozirane cene akcija. Konačno, varijabla *optionsChanged* se postavlja na vrednost dva nakon isteka vremena koje je korisnik prognozirao. U tom slučaju, dolazi do restartovanja

podešavanja za prikaz grafikona na ona koja su bila pre pritiska na dugme *"Uloži"*.Za kreiranje grafikona korišćena je *jQuery flot*[6] biblioteka.

V ZAKLJUČAK

U ovom radu dat je kratak uvod u reaktivnu programsku paradigmu, korišćenjem *SpringWebFlux* tehnologije za razvoj veb aplikacije za pregled stanja na berzi. Kao i sve ostale tehnologije, Spring *WebFlux* (i generalno reaktivna programska paradigma) ima svoje prednosti i nedostatke. *WebFlux* donosi benefite u pogledu performansi aplikacije, kao i pružanja boljeg korisničkog iskustva. S druge strane, reaktivni način programiranja dolazi sa novim načinom pisanja, testiranja i debagovanja koda, tako da tranzicija sa tradicionalnog načina pisanja koda na reaktivni nije uvek brza i jednostavna. Činjenica da opisani model koristi ne-blokirajući tok izvršavanja (*eng."non-blocking execution flow")*, razlikuje ga od tradicionalnog načina razvoja gde je svaki poziv blokiran dok ne dodje odgovor, čini ga pogodnim za razvoj brojih aplikacija koje se izvršavaju u realnom vremenu (eng."real time applications"), kao što je aplikacija opisana u ovome radu. Generalno, ne postoji pravilo prilikom izbora i korišćenja tehnologija u projektu, osim da te tehnologije daju dobra biznis rešenja, lako održiva, elastična i skalabilna.

LITERATURA

[1] Ranga Rao Karanam, "Mastering Spring 5.0"

[2] Craig Walls, "Spring in Action, 4th Edition"

[3] http://www.reactive-streams.org/

[4] https://maven.apache.org

[5] Spring Framework reference Documentation,https://docs.spring.io/spring-framework/docs/current/spring- frameworkreference/html/index.html

[6] http://www.flotcharts.org/

ABSTRACT

In order to acquire a comprehensive understanding of the mutual similarities, differences, advantages and disadvantages of the traditional and reactive approach to solving some of the problems encountered by modern web applications, as well as the possibilities offered by the reactive module of the

popular Spring framework, this article will demonstrate the development of the web applications that manage companies' shares on the stock market, using both of the aforementioned approaches.

INTRODUCTION TO REACTIVE PROGRAMMING PARADIGM BY USING SPRING FRAMEWORK

Nemanja Zirojević