Univerzitet Union

Računarski fakultet

MASTER RAD

IMPLEMENTACIJA WEB APLIKACIJE ZA PREGLED STANJA NA BERZI KORIŠĆENJEM REACTIVE SPRING TEHNOLOGIJE

Mentor:

Dr Bojana Dimić Surla

Kandidat:

Nemanja Zirojević

Beograd, decembar 2018.

APSTRAKT: Razvojem IT industrije i porastom konkurencije među softverskim kompanijama raste i broj izazova i novih zahteva sa kojima se susreću moderne veb aplikacije, radi pružanja što boljih performansi i korisničkog iskustva klijentima. U cilju ostvarivanja izazova sa kojima se aplikacije susreću javila se potreba za što optimalnijim korišćenjem resursa koji joj stoje na raspolaganju, što slabije povezanim komponentama koji čine, većom optornošću na kvarove i na kraju što bržim izaskom u produkciju. Reaktivno programiranje iako nije nov fenomen pruža rješenja koja mogu poslužiti prilikom realizacije nekih od gore navedenih izazova. U ovom master radu, u cilju sticanja sveobuhvatne slike o uzajamnim sličnostima, razlikama, prednostima i manama tradicionalnog i reaktivnog pristupa, demonstriran je razvoj veb aplikacije za pregled stanja na berzi korišćenjem reaktivnog modula radnog okvira Spring. Aplikacija je po svojoj strukturi troslojna, monolitna i sastoji se od sledećih slojeva:

1. prezentacijski (klijentski) sloj,

2. sloj poslovne logike,

3. sloj podataka.

Za implementaciju gore navedenih slojeva, pored radnog okvira Spring, korišćene su različite tehnologije, koje uključuju: programski jezik Java, MySql i Mongo baze podataka, Jquery radni okvir i Hibernate ORM alat.

Rad je podeljen na nekoliko delova. U prvom delu rada dat je kratak opis gore navedenih tehnologija, njihova specifična uloga i upotreba u projektu. U drugom delu razmatraju se sličnosti i razlike između reaktivnog i tradicionalnog pristupa razvoju aplikacije, sa posebnim osvrtom na implementaciju reaktivnog toka podataka korišćenjem objekata iz klasa reaktivnog modula radnog okvira Spring, kao i na mehanizme osluškivanja i reagovanja na događaje u trenutku kada se oni dese.

Ključne reči: reaktivno programiranje, Spring framework

ABSTRACT: With the development of IT industry and the competition increase among software companies, the number of challenges that modern web applications face is growing as well, in order to provide the best performance and customer experience. In order to meet the challenges encountered by the applications, several needs emerged: the need for optimal use of the resources at their disposal, the weaker connected components, the greater resistance to failures and, finally, the faster production process. Reactive programming, although not a new phenomenon, provides solutions that can be used to rise to some of the above mentioned challenges. In order to obtain a comprehensive picture of the mutual similarities, differences, advantages and disadvantages of the traditional and reactive approach, this thesis demonstrates the development of a web application for stock market overview using the Spring Module reactive module. The application is monolithic and structured in three layers:

1. Presentation (customer) layer,
2. A business logic layer,
3. Data layer.

To implement the above mentioned layers, in addition to the Spring workspace, various technologies have been used, including: the Java, MySql and Mongo database programming language, the Jquery workbench and the Hibernate ORM tool.

This thesis contains several parts. The first part gives a brief description

Of the above mentioned technologies, their specific role and use in the project. The similarities and differences between the reactive and traditional approach to application development are considered in the second part, with a particular reference to the implementation of the reactive data stream using objects from the classes of the reactive module of the Spring framework, as well as to the mechanisms for listening and responding to events at the time of their occurrence.

Keywords: reactive programming, Spring framework

Sadržaj

1. 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 i manja količina podataka. Pojavom novih uređaja (mobilni uređaji, tableti i tako dalje) i najnovijeg načina pristupa (zasnovanog na cloud-u), pojavili su se i novi izazovi i zahtevi za veb aplikacije. Najvažniji izazovi sa kojima se susreću moderne veb aplikacije su [1] .

1. Vreme odziva u sekundi,
2. Stopostotna dostupnost,
3. Eksponencijalno povećanje obima podataka.

Različiti pristupi su se pojavili poslednjih godina da bi bili rešeni ovi novi izazovi, a jedan od njih je reaktivno programiranje. Iako reaktivno programiranje nije nov fenomen, to je jedan od pristupa koji su bili uspešni u rešavanju gore navedenih izazova.

Neki od važnih karakteristika reaktivnih sistema su sledeće [1] :

1. Brzina – sistemi odgovaraju brzo svojim korisnicima. Podešeni su jasni zahtevi za vreme vraćanja odgovora i sistem ih ispunjava u svim situacijama.

2. Prilagodljivost – Distribuirani sistemi su izgrađeni pomoću više komponenata. Greške mogu da se dese u bilo kojoj od njih. Reaktivni sistemi treba da budu dizajnirani tako da sadrže greške unutar lokalizovanog prostora, na primer unutar svake komponente, čime se sprečava da ceo sistem padne u slučajevima lokalne greške.

3. Elastičnost – Reaktivni sistemi ostaju brzi pod različitim opterećenjima. Kada su pod teškim opterećenjem, mogu da dodaju resurse, dok ih, kada se smanji opterećenje otpuštaju. Elastičnost je postignuta upotrebom hardvera i softvera.

4. Vođenje događajima – Reaktivni sistemi su vođeni događajima. To obezbeđuje slabo povezivanje među komponentama, što garantuje da različite komponente sistema mogu da budu skalirane nezavisno. Upotreba neblokirajuće komunikacije obezbeđuje da su programske niti žive u kraćem vremenskom roku.

Rekativni sistemi su prilagodljivi na različite vrste stimulacija. Sledi nekoliko primera:

1. Reakcija na događaje – Izgrađena je na osnovu prosleđene poruke i reaktivni sistemi brzo odgovaraju na događaje.

2. Reakcija na opterećenje – Reaktivni sistemi ostaju brzi pod različitim opterećenjima. Koriste više resursa pod velikim, a optuštaju ih pod manjim opterećenjem.

3.Reakcija na greške – Reaktivni sistemi mogu elegantno da obrade greške. Komponente reaktivnih sistema su izgrađene za lokalizovanje grešaka.

Eksterne komponente se koriste za praćenje dostupnosti komponenata i imaju sposobnost da repliciraju komponente kada je to potrebno.

4. Reakcija na korisnike – Reaktivni sistemi brzo odgovaraju korisnicima. Ne troše vreme na izvršavanje dodatne obrade kada korisnici nisu prijavljeni za specifične događaje.

1.1 Motivacija i opis zadatka

Imajući u vidu gore navedene izazove sa kojima se susreće savremena softverska industrija, kao i rešenja koja u tim slučajevima nudi reaktivni pristup u izradi aplikacije, u ovom radu biće demonstriran razvoj veb aplikacije za pregled cena akcija kompanija na berzi korišćenjem *Reactive Spring* tehnologije, s ciljem demonstracije pomenutih osobina reaktivnih sistema i benefita koje pruža takav razvoj aplikacije u odnosu na tradicionalni.

1.2 Izbor tehnologija

Jedna od ključnih stavki od koje u mnogome zavisi uspeh projekta je proces izbora tehnologija u kojima će biti razvijana aplikacija. Izbor tehnologije je naročito izazovan proces kada su u pitanju manje kompanije i startit-ovi, pre svega zbog obično ograničenog budžeta, i stoga je potrebno izabrati tehnologije koje će moći da odgovore potrebama korisnika i izvuku maksimum performansi za aplikaciju, a sve u sklopu predviđenog budžeta. Veće kompanije, uglavnom biraju tehnologije na osnovu procene svojih inženjerskih timova, njihovog poznavanja specifične tehnologije i potreba korisnika. Kada govorimo o izboru tehnologija pod time se podrazumeva

izbor tehnologija na klijentskoj i na serverskoj strani aplikacije. Neki od kriterijuma za odabir trebalo bi da budu [1] :

1. Mogućnosti tehnologije (prednost je u koliko je softver otvorenog koda) ,
2. Kompatibilnost sa drugim tehnologijama koje će se koristiti,
3. Dokumentovanost,
4. Aktuelnost, obim primenjenosti u industriji, što potencijalno može uticati na njenu sudbinu opstanka na tržištu.

U narednim odeljcima, predstaviće se neke od tehnologija koje su korišćene i u prototipskoj aplikaciji koja je tema ovog master rada.

1.2.1 Java

Java je objektno-orjentisani programski jezik koji je razvija kompanija Sun Microsystems početkom 1990tih godina . Trenutno je jedan od najčešće korišćenih programskih jezika. Na rang listi kompanije TIOBE , objavljenoj za 2017-tu godinu Java zauzima prvo mesto [2] . Poslednja dostupna verzija Jave, u trenutku pisanja ovog teksta, je Java 9, nastala 2017. godine. Za razvoj prototipske aplikacije korišćena je verzija Java 8, pošto počevši od verzije 5.0 radnog okvira Spring, ta verzija minimalna podržana verzija Java-e.

1.2.2 Radni okvir Spring

Spring je okvir (*eng.”framework”*) za Java platformu, koji obezbeđuje infrastrukturu za razvoj aplikacija različite namene [3] . Zahvaljujući tome, softverski inženjeri mogu da se fokusiraju na razvoj biznis zahteva aplikacije.

Spring je po dizajnu modularan. Nudi više od dvadeset modula, koji obezbeđuju različite infrastrukture [3] :

1. Osnovni Spring okvir (*eng. “Spring Core”*),
2. Okvir koji postavlja i podiže aplikaciju (*eng. “Spring Boot”),*
3. Okvir za rad sa bazama podataka (*eng.” Spring Data”*),

4. Okvir koji obezbeđuje alate za rad u oblaku (*eng. "Spring Cloud"*),
5. Okvir za sigurnost aplikacije (*eng. "Spring Security", "Spring LDAP"*),
6. Okvir za upravljanje porukama (*eng. "Spring Integration"," Spring AMQP"*),
7. Okvir za rad sa reaktivnim tokovima podataka (*eng. "Reactive Spring"*)itd..

Jedna od ključnih prednosti Spring-a je minimizaciji svojih zavisnosti ka drugim bibliotekama i okvirima. Jedina zavisnost osnovnog Spring okvira je ka biblioteci za logovanje aktivnosti u aplikaciji. Integracija Spring-a u aplikaciju je jednostavna uz korišćenje alata, kao što je, na primer, Maven (što je korišćeno i u prototipskoj aplikaciji).
Spring Boot je okvir koji omogućava kreiranje infrastrukture za samostalne aplikacije bez xml konfiguracije, spremne za produkciju [3] .Ima ugrađen veb server (Tomcat, Jetty ili Undertow), pa je aplikaciju dovoljno pokrenuti kao običnu Java aplikaciju. Zbog svoje lakoće u korišćenju, jednostavnosti i fleksibilnosti korišćen je u razvoju prototipske aplikacije. Spring Boot, takođe, postavlja inicijalni *pom.xml* fajl, za konfiguraciju sa Maven alatom. Spring Security je modul koji nudi rešenje za lako konfigurisanje autentifikacije i autorizacije korisnika. Spring Boot, kao podrazumevanu autentifikaciju, implementira osnovni tip Spring Security okvira. Takođe podržava OAuth2.
Za implementaciju REST kontrolera, u prototipskoj aplikaciji, korišćen je Spring Boot MVC okvir. REST kontroleri komuniciraju sa klijentskom stranom aplikacije preko HTTP/HTTPS protokola, razmenjujući podatke u JSON formatu. Za konfiguraciju se koriste anotacije. Na primer, anotacija *@RestController* označava da je klasa REST kontroler; anotacija *@RequestMapping* mapira URL zahtev koji dolazi od veb pretraživača na odgovarajući metod u kontroler klasi koji vraća odgovor. Opciono, ovoj anotaciji se kao argumenti može proslediti i tip zahteva (DOHVATI (*eng." GET"*), POSTAVI (*eng. "POST"*), OBRIŠI (*eng. "DELETE"*)) koji dolazi od veb pretraživača, kao i vrsta odgovora koji metod označen ovom anotacijom treba da vraća.
Za rad sa reaktivnim tokom podataka, u prototipskoj aplikaciji, korišćene su klase i interfejsi iz Reactive Spring modula, koji je sastavni dio Spring radnog okvira počevši od verzije Spring 5, koja je puštena u produkciju u julu 2017-te godine.

1.2.3 MongoDB

MongoDB je objekto-orjentisana NoSQL baza podataka , nezavisna od platforme na kojoj se izvršava. Dizajnirana pre svega da pruži visoke performanse, visoku dostupnost i jednostavnu skalabilnost, što je čini dobrim izborom za aplikacije koje koriste podatke u realnom vremenu (eng.*"real time applications"*), kao i za e-commerce rešenja, mobilne aplikacije, arhiviranje i slično. Poznati slučajevi korišćenja MongoDB obuhvataju "*big data*" podatke, upravljanje sadržajem, mobilnu i društvenu infrastrukturu i mnoge druge [4] . Takođe, pojavljuju se izazovi za korišćenje MongoDB za *Business Intelligence modele*. Za razliku od tradicionalne relacione stukture zasnovane na tabelama i njhovim međusobnim odnosima (relacijama), MongoDB je orjentisana ka dokumentima.
Popularnost nerelacionih baza porasla je sa ekspanzijom web-a i potrebe da se uskladište različiti multimedijalni sadržaji, koje je teško smestiti u relacione baze.

Osnovne karakteristike MogoDB su [4]:

- Smeštanje usmereno na dokumenta,
- Automatsko skaliranje,
- Moćni upiti, bazirani na dokumentima,
- Brzi update,
- Mapiranje/Redukovanje - fleksibilna agregacija i obrada podataka,
- Online shell pruža mogućnost isprobavanja MongoDB bez instalacije,
- Ad hoc upiti - MongoDB podržava pretragu po polju, upite po opsegu i pretrage po regularnim izrazima. Upiti mogu da vrate određena polja dokumenta, kao i da obuhvate korisnički definisane JavaScript funkcije itd..

Glavni argumenti za upotrebu MongoDB baze su:

- MongoDB može da uskladišti sadržaj bilo kog tipa,
- Ugrađena skalabilnost,
- Sigurnost,
- Otvorenog je koda,
- Dostupnost drajvera za više od deset programskih jezika,

- Procenjuju se manji troškovi u produkciji,
- ReactiveMongo podrška za reaktivne tokove podataka.

Za izradu aplikacije koja je tema ovog rada, poslednja stavka je od ključne važnosti.

Većina baza podataka su blokirajuće, odnosno programska nit čeka dok odgovor ne bude primljen iz baze podataka. Da bismo maksimalno iskoristili prednosti reaktivnog programiranja, komunikacija sa kraja na kraj mora da bude reaktivna, odnosno zasnovana na tokovima događaja. **ReactiveMongo** je projektovan da bude reaktivan i izbegne operacije blokiranja. Sve operacije uključujući selekcije, ažuriranje ili brisanje, vraćaju razultat odmah. Podaci mogu da budu uneseni u bazu podataka i vađeni iz nje pomoću tokova događaja.

1.2.4 Hibernate JPA implementacija

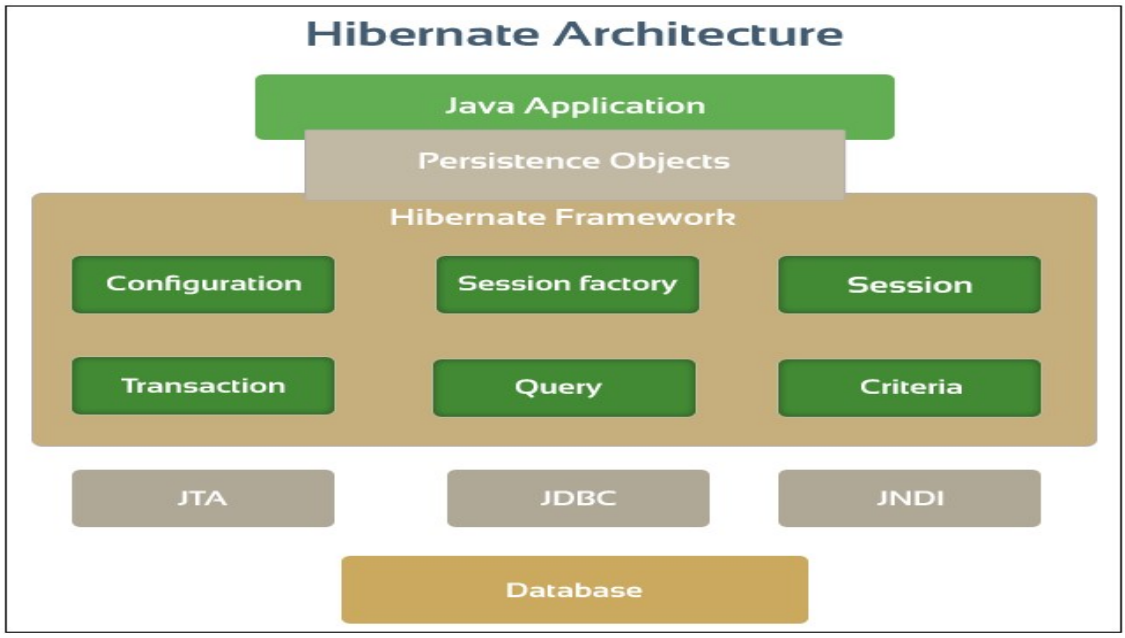
Razumevanje značaja perizistencije u razvoju savremenih aplikacija je važno za odabir skladišta podataka koji će se najbolje uklapati sa potrebama aplikacije. Perzistencija podataka predstavlja jedan od fundamentalnih koncepata u razvoju softverskih sistema. Uopšteno, pod perzistencijom podrazumevamo nastavak postojanja objekta i nakon završetka procesa koji ga je kreirao.U kontekstu Java aplikacija perzistentost se odnosi na čuvanje Java objekata u najčešće relacionoj bazi podataka. Transformacije kojima se vrši pretvaranje Java objekata i njihovih odnosa u tabele i relacije u okviru baze podataka i obrnuto, re-kreiranje Java objekata na osnovu podataka iz baze podataka, nazivaju se objektno-relaciono mapiranje (ORM). Hibernate je ORM alat koji potpuno implementira JPA7 specifikaciju [5] . Hibernate je po svojoj strukturi modularan. Sastoji se od tri glavna modula koji se mogu koristiti po potrebi, pojedinačno ili kombinovano [5] :

1. Hibernate Core - osnovni servis za perzistenciju, koristi xml datoteke sa metapodacima za mapiranje,
2. Hibernate Annotations – koristi anotacije za označavanje aplikacionih metapodataka i njihovo mapiranje u tabele baze podataka,
3. Hibernate EntityManager – opcioni modul koji se koristi uglavnom za upravljanje životnim ciklusom perzistentnih objekata i rad sa upitima.

Korišćenje ORM alata u razvoju aplikacije ima nekoliko prednosti :

1. Eliminisanje “ručnog” pretvaranja SQL ResultSet-a u Java objekte,
2. Izbegavanje pisanja JDBC i SQL koda niskog nivoa,
3. Većina ORM alata su besplatni i otvorenog koda,
4. Obezbeđuje nezavisnost baze podataka i šema,
5. Pruža visoko performansne operacije kao što su keširanje i optimizacija upita.

Arhitektura Hibernate radnog okvira prikazana je na slici 1.1.



Slika 1.1 Arhitektura Hibernate okvira
(preuzeto sa adrese https://www.oodlestechnologies.com/blogs/Introduction-and-Architecture-of-Hibernate-4)

U razvoju web aplikacije, hibernate okvir će biti korišćen za čuvanje, pretragu i ažuriranje korisnika i njihovih prava u MySQL bazi podataka. Trenutna verzija Hibernate-a u toku pisanja ovog rada koja je korištena i prototipskoj aplikaciji je 5.3 .
Podešavanja vezana za lokaciju i kredencijale za pristup bazi, čuvamo u *application.properties* fajlu.

1.2.5 Bootstrap css radni okvir

Bootstrap je css radni okvir koji se koristi za izradu fleksibilnih (*eng."responsive"*) veb stranica a sastoji od unapred definisanih HTML i CSS elemenata [6] . Razvijen kao projekat kompanije *Twitter* 2010. godine, Bootstrap je prošao kroz dvadesetak izmena među kojima su najvažnije *mobile-first* i *responsive* pristup koji su kao opcione komponente dodate u verziji Bootsrap 2. Počevši od verzije Bootstrap 3, *responsive mobile first* pristup postao je glavna funkcija Bootstrap radnog okvira, koja više nije opciona, već podrazumevana komponenta svih bootstrap klasa. Jedna od glavnih prednosti korišćenja pomenutog radnog okvira je upravo u dobijanju gotovog rešenja za prilagođavanje aplikacije različitim veličinama ekrana, bez mukotrpnog pisanja medijskih upita (*eng."media queria")* ,kao što je to bio slučaj ranije. Zbog toga, standardna praksa je koristiti jedan od postojećih radnih okvira koji u sebi sadrži rešene medijske upite (Bootstrap, Bulma, Materialize, KickUp…). Bitna prednost Bootsrap-a je njegova kompaktibilnost sa svim web pretraživačima kao što su Google Chrome, Safari, FireFox, IE, Opera. Trenutna verzija u toku pisanja ovog rada, koja je ujedno krošićena u prototipskoj aplikaciji je Bootstap 3.

1.2.6 Jquery javascript radni okvir

Jquery je popularna JavaScript biblioteka koncipirana tako da pojednostavi pisanje i izvršavanje JavaScript skripti [7]. Jquery je projekat otvorenog koda, besplatan za preuzimanje i korišćenje, a pored osnovne biblioteke dostupan je i veliki broj besplatnih plugin-ova (jquery-mask-plugin, cropper, js-cookie, jquery-ui, itd..). Osnovna prednost jquery-a je u pojednostavljivanju operacija kao što su manipulacija DOM (*eng. "document object model"*) elementima, rukovanje događajima, izrada animacije, kreiranje ajax poziva. Kompatibilan je sa svim verzijama popularnih veb pregledača. U pogledu kompleksnosti, jquery je mnogo efikasniji u odnosu na standardni vanilla javascript. Kod napisan u regularnom JavaScript jeziku se, upotrebom jquery biblioteke, može napisati u znatno redukovanom obliku. Postoje dve opcije korišćenja jquery-a. Prva opcija podrazumeva preuzimanje biblioteke sa oficijelnog sajta, dok je druga uključivanje biblioteke u HTML stranicu preko CDN *(eng." Content Delivery Network")* linka. Poslednja

opcija je korištena u prototipskoj aplikaciji zato što se tim putem preuzima biblioteka sa najbližeg servera u zavisnosti od korisničke lokacije, što povoljno utiče na performanse i korisničko iskustvo. Neke od najvećih svetskih IT kompanija koriste jquery na veb-u, kao što su: Google, Microsoft, IBM, Netflix,itd...
U razvoju prototipske aplikacije Jquery je korišćen za tabelarni prikaz podataka na klijentskoj strani i izradu grafikona za prikaz cena akcija u realnom vremenu.

1.2.7. MySQL baza podataka

MySQL je najpopularniji i najkorišćeniji *RDBMS* (eng."*relational database managment system"*) sistem otvorenog koda za upravljanje relacionim bazama podataka [8]. Programeri, administrator baza i DevOps inženjeri koriste MySQL da bi napravili savremene *cloud-based* veb aplikacije. MySQL zauzima centralni deo popularne LAMP platforme (Linux – Apache – MySQL – Perl/PHP/Python). Kao većina *RDBMS* rešenja MySQL je dostupan u nekoliko različitih verzija, koje se pokreću na Windows, OS X, Solaris FreeBSD i drugim varijantama Linux operativnog sistema. Mnogi od popularnih kompanija kao što su : Google, Facebook, Adobe, Zappos, YouTube koriste MySQL. Koriste ga i mnogi poznati open source projekti kao što su: WordPress, Joomla, Drupal. Jedna od prednosti korišćenja MySQL-a je bogata API podrška (konektori) za različite programske jezike. Neki od najpoznatijih konektora su Connector/J za Java platform, Connector/Net za .NET platformu,itd.. MySQL podržava korišćenje stored procedura, trigera, kursora, funkcija, ugnježdenih upita, keširanje upita. Počevši od verizije 8.0 MySQL podržava skladištenje dokumenta u vidu JSON fajlova. Osobine koje zajedno najbolje opisuju MySQL su: pouzdanost, visoka dostupnost, dobre performanse, jednostavnost korišćenja i sigurnost. Imajući u vidu gore navedene osobine MySQL baza podataka je izabrana kao tehnologija za čuvanje korisnika, njihovih kredencijala i privilegija u razvoju prototipske veb aplikacije.

1.2.8 Thymeleaf template engine

Thymeleaf je moderan serverski orjentisan Java generator forme, otvorenog tipa (*eng. "open source"*) koji se koristi kako za web okruženja, tako i za ostala okruženja [9]. Glavna prednost korišćenja Thymeleaf-a je u tome što je Thymeleaf "prirodni" generator forme, što znači da se uklapa u HTML dokument ne menjajući njegovu spoljašnu formu. Ključna prednost koja je uticala na izbor Thymeleaf-a, kao tehnologije koja se koristi u razvoju projekta je njegova laka integracija sa modulima Spring framework-a.

1.2.9. Maven

Maven je alat koji se koristi pri razvoju aplikacije u cilju lakše integracije sa postojećim bibliotekama klasa [10]. Maven se zasniva na plug-in arhitekturi, koja omogućava primenu plug-inova za različite zadatke (compile, test, build, deploy, checkstyle, pmd, scp-transfer) u projektu, bez potrebe njihove direktne instalacije. Sve biblioteke, potrebne datom projektu,u daljem tekstu zavisnosti (*eng.dependencies*) deklarišu se u *pom.xml* fajlu. Maven kreira lokalni repozitorijum na disku sa potrebnim jar-ovima tako da prilikom pokretanja projekta, sve bibilioteke navedene u *pom.xml* faju u početku traži lokalno, a zatim ako ih ne nađe preuzima ih sa centralnog maven repozitorijuma na internetu.

2. Arhitektura web aplikacije

U ovom poglavlju, u svrhu sticanja jasnije slike o radu prototipske aplikacije biće prikazani tipični slučajevi korišćenja, dijagrami klasa, dijagrami sekvenci, kao i model korisničkih podataka u relacionoj bazi podataka bazi. Bitno je naglasiti da se MySQL relaciona baza koristi za čuvanje podataka o korisnicima, dok se Mongo baza koristi za čuvanje podataka o kompanijama zbog svoje podrške radu sa reaktivnim tokovima podataka, kao što je naglašeno u poglavlju 1.4.1. Aplikacija je po svojoj strukturi trosloja, sastoji se od sledećih slojeva:

1. Prezentacioni sloj – implementiran korišćenjem *HTML5* tehnologije *bootstrap css* framework-a, *jQuery i Datatables.js* biblioteka,
2. Sloj poslovne logike,
3. Sloj podataka – implementiran korišćenjem funkcionalnosti *JpaRepository, ReactiveMongoRepository* interfejsa iz paketa *org.springframework.data.jpa.repository, org.springframework.data.mongodb.repository* respektivno.

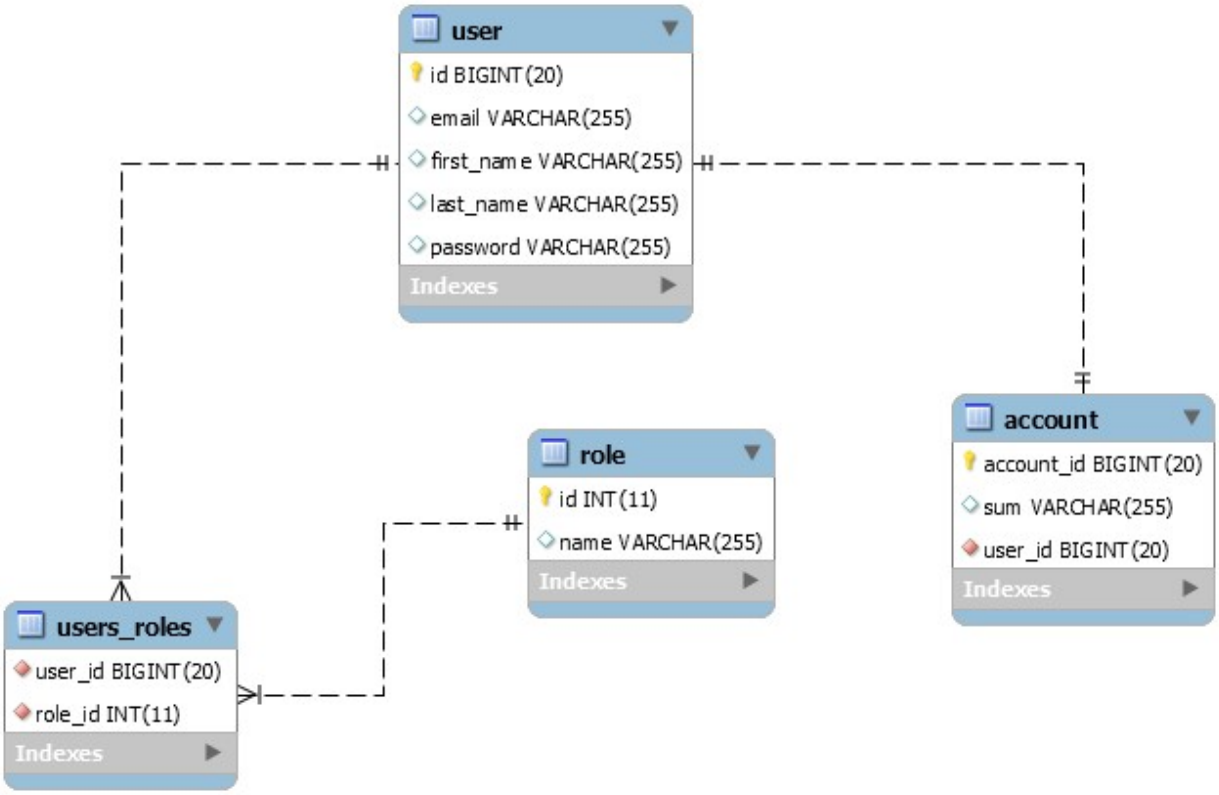
U narednim poglavljima biće detaljno opisana implementacija svakog sloja pojedinačno.

2.1 Model podataka

Za potrebe aplikacije za upravljanje akcijama na berzi, neophona je baza podataka u kojoj će se čuvati podaci o korisnicima, njima dodeljenim ulogama (*eng.“roles“*), računima, kao i podaci o kompanijama čije se akcije prikazuju na berzi. Pošto će se podaci o cenama akcija kompanija dobijati u vidu promenjivog toka podataka, kao i zbog lake integracije i pružanja interfejsa za podršku u radu sa reaktivnom Mongo bazom, podaci o akcijama i kompanijama čuvaće se Mongo bazi, dok će se podaci o korisnicima čuvati u MySQL bazi.
Za modelovanje baze podataka koriste se tehnike poput ERA (*eng.“entity-relationship”)* dijagrama i relacionog modela baze podataka koji je korišten u ovom radu.

2.1.1 Relacioni model baze podataka

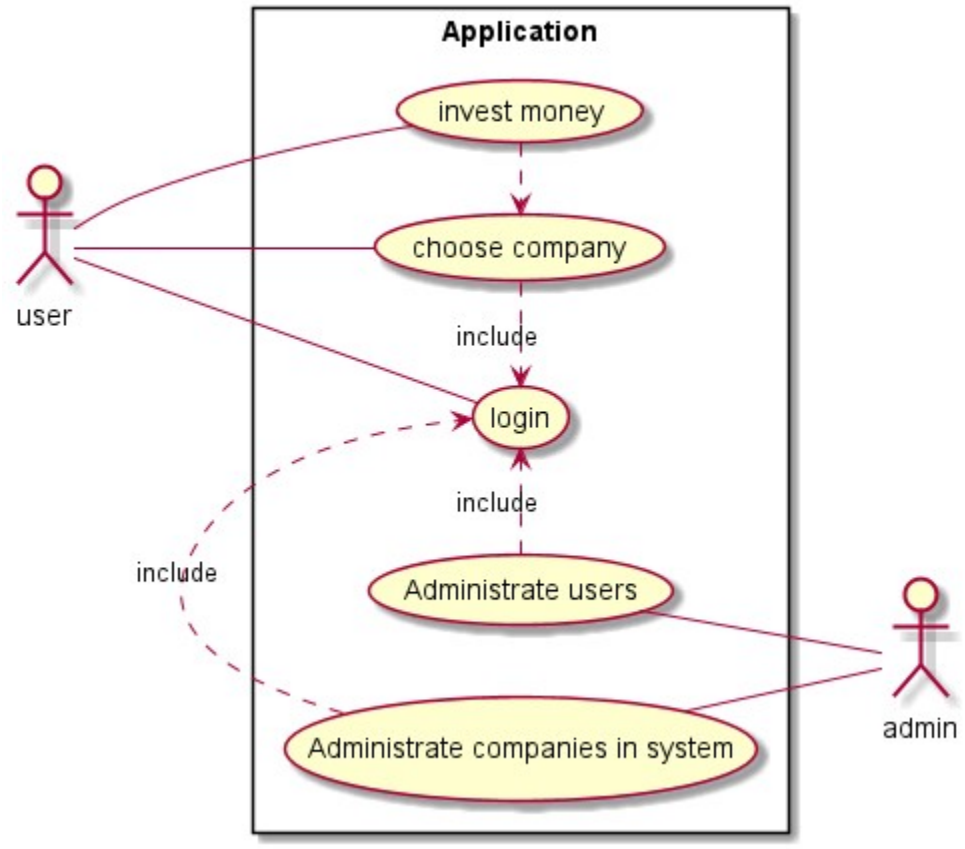
Relacioni model podataka je najčešće korišćeni model za čuvanje i obradu podataka. Koncept relacionog modela podataka zasniva se na organizaciji podataka u vidu tabela, između kojih su uspostavljene veze (relacije) na osnovu primarnih i sekundarnih ključeva. Dizajn baze podataka u alatu MySQL workbench prikazan je na dijagramu 2.1.



Dijagram 2.1 Model podataka

2.2. Dijagram slučajeva korišćenja

Na slici 2.2 prikazan je dijagram slučajeva korišćenja koji opisuje funkcionalnost aplikacije za upravljanje akcijama na berzi.

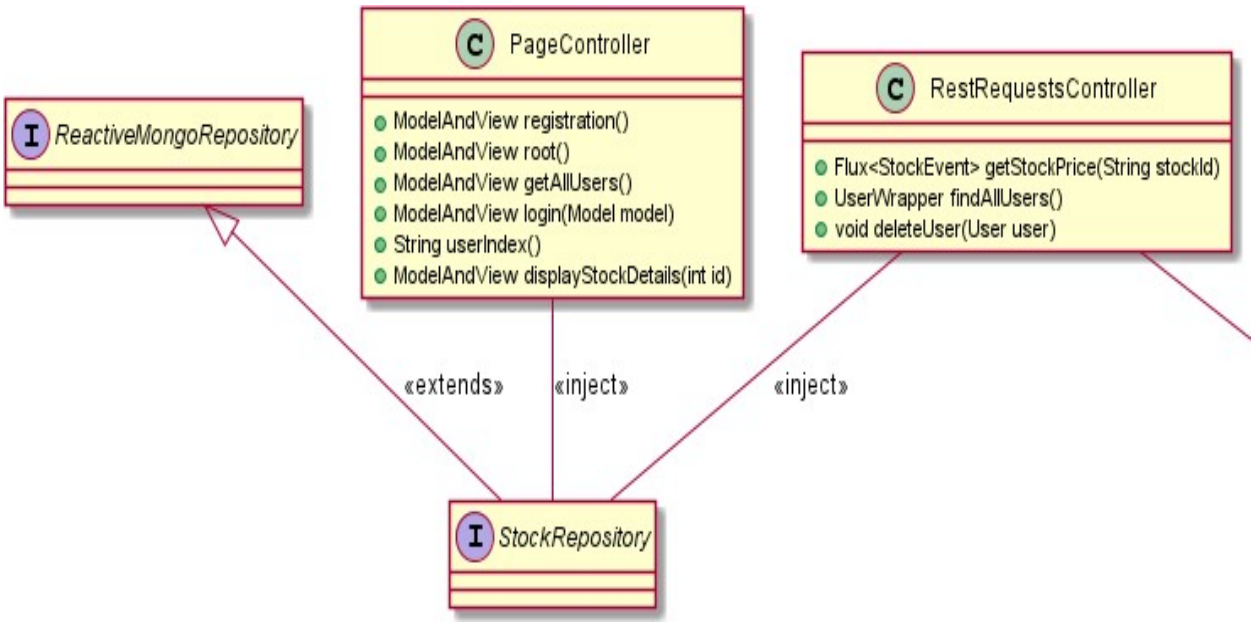


Slika 2.2 Dijagram slučajeva korišćenja

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 sa novca kojom korisnik raspolaže (ti podaci se čuvaju u posebnoj tabeli Account kao što je prikazano na slici *2.1*). Korisnici sa administratorskim privilegijama su u mogućnosti da dodaju nove i brišu postojeće kompanije čije se akcije prikazuju na berzi, kao i da upravljaju korisničkim nalozima (dodavanje, izmene, deaktiviranje i aktiviranje). Pored toga u mogućnosti su da vide statističke podatke vezane za rad aplikacije (procenat slobodne memorije, status baze podataka, broj aktivnih procesa, logove,itd..). Detaljan opis rada prototipske aplikacije biće opisan u posebnom poglavlju.

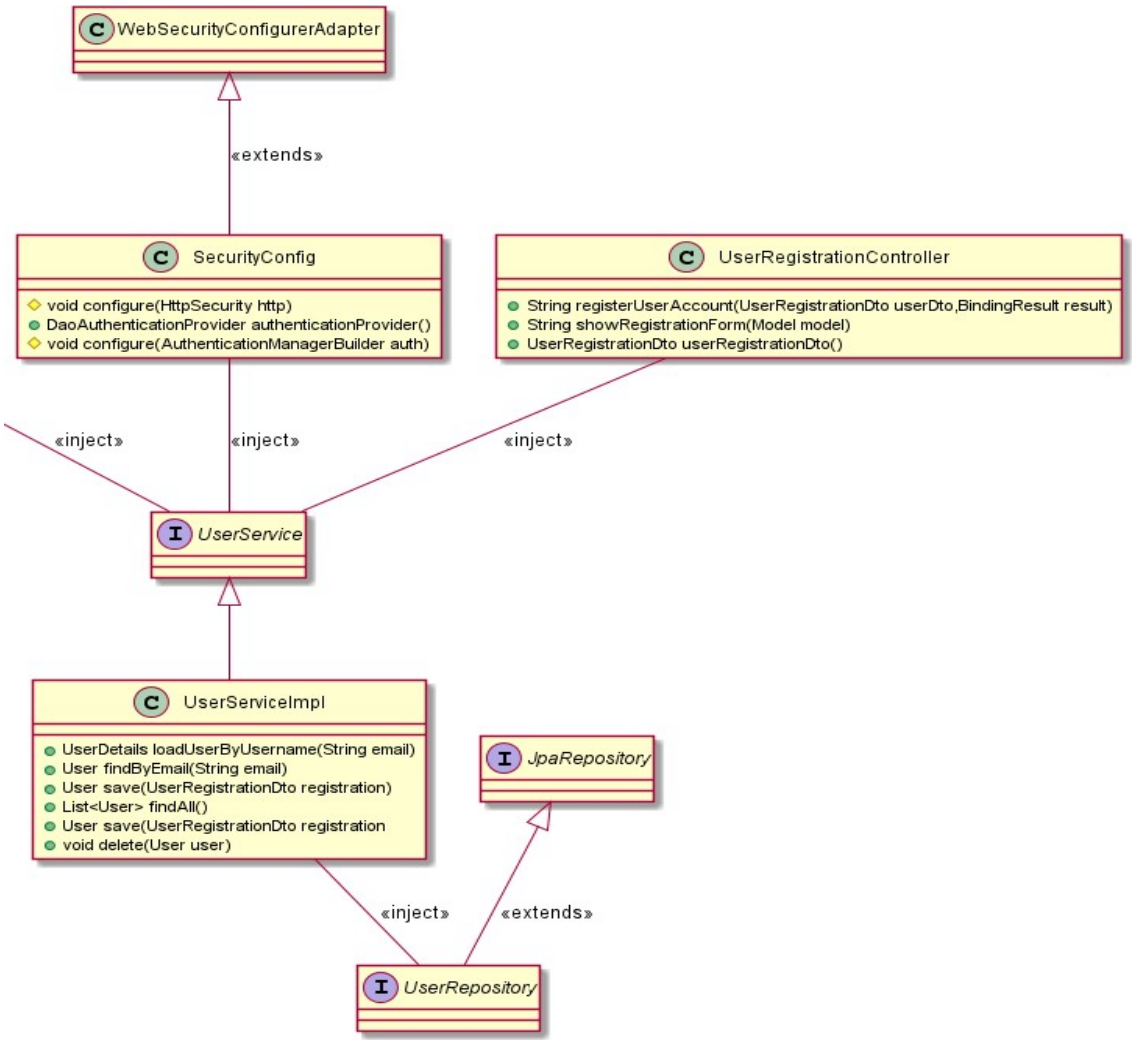
2.3. Dijagram klasa

U nastavku je prikazan kratak opis dijagrama klasa koje ulaze u sastav aplikacije. Kao što se vidi na slici *2.3 Dijagram klasa – kontroleri*, klase *PageController* i *RestRequestController* u sebi sadrže interfejs tipa *StockRepository* (koji se ubacuje mehanizmom dependency injection-a, koristeći Spring-ovu *@Autowired* anotaciju). Ovaj interfejs nasleđuje generički interfejs *ReactiveMongoRepository* iz paketa *org.springframework.data.mongo.repository* koji pruža metode za rukovanje reaktivnim tokovima podataka korišćenjem Mongo baze.



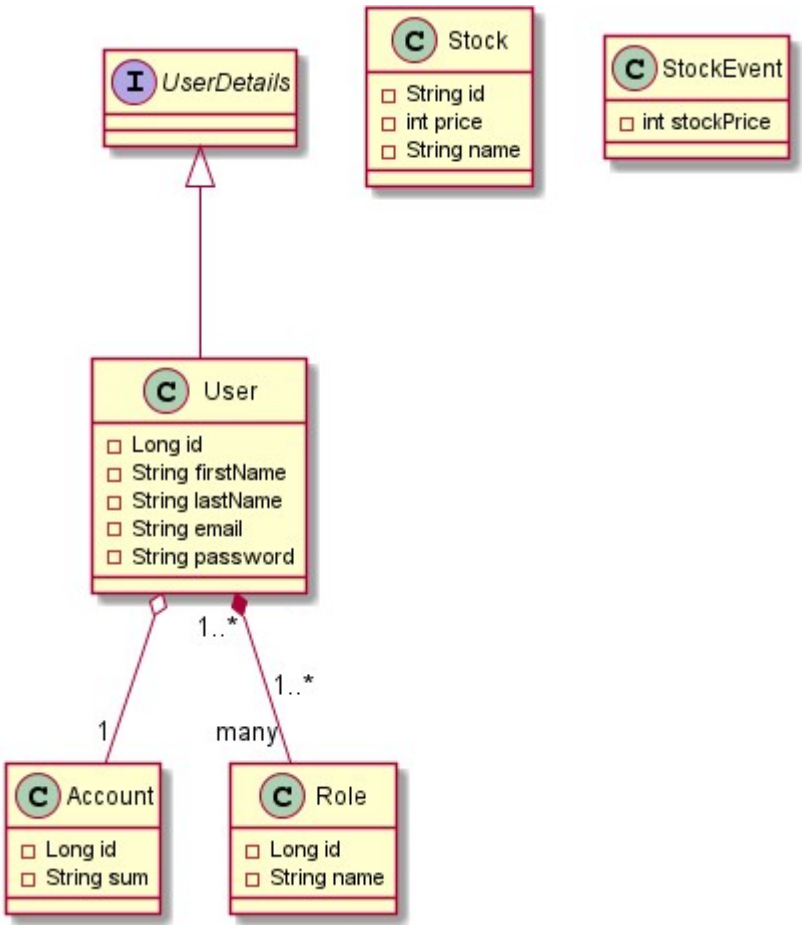
Slika 2.3.Dijagram klasa – kontroleri

Klasa *WebSecurityConfigurerAdapter* iz paketa *org.springframework.security.config.annotations.web.configuration* pruža metode za konfigurisanje sigurnosti u aplikaciji. Generički interfejs *JpaRepository* iz paketa org.springframework.data.jpa.repository obezbeđuje metode za rukovanje CRUD operacijama vezanim za dodavanje, brisanje, izmenu i pregledanje korisničkih naloga (*slika 2.4 Dijagrami klasa – repozitorijumi, servisi, klasa za kontrolu sigurnosti i registraciju korisnika*).



Slika 2.4 Dijagram klasa – repozitorijumi, servisi, klasa za kontrolu sigurnosti i registraciju korisnika

Interfejs *UserDetails* iz paketa *org.springframework.security.core.userdetails* pruža metode neophodne za autentikaciju i autorizaciju korisnika, upravljanje korisničkim nalozima, upravljanje korisničkim pravima.



Slika 2.5 Dijagram klasa – korisnički definisani modeli

3. Implementacija serverske strane aplikacije

U ovom poglavlju opisana je implementacija serverskog dela aplikacije koji se odnosi na upravljanje korisničkim nalozima (registraciju, brisanje, dodavanje i pregled korisnika), sigurnost veb aplikacije, višejezičnost, administraciju kompanija i prikaz statističkih podataka vezanih za stanje aplikacije (metrika). U narednom poglavlju razmatra se implementacija klasa i fukncionalnosti vezanih za upravljanje reaktivnim tokovima podataka.

3.1. Izrada perzistentnog sloja podataka

Skoro sve aplikacije imaju potrebu za perzistentim podatacima. Izrada posebnog srednjeg (posredničkog) sloja između baze podataka i biznis logike vodi ka fleksibilnijem dizajnu aplikacije i omogućava nekoliko pogodnosti, kao što je lakša migracija sa jednog na drugi tip skladišta podataka, bolju enkapsulaciju logike iz baze u jednistven sloj, što pruža olakšanja prilikom tesiranja, otklanjanja eventualnih grešaka i razvoja novih funkcionalnosti. Spring Boot projekat, kao projekat čija je krajnja namena brzo i jednostavno kreiranje i pokretanje Spring aplikacija, pruža mogućnost konfiguracije JPA Hibernate implementacije veoma jednostavno. Da bi se omogućio JPA u Spring aplikaciji, potrebno je najpre dodati *spring-boot-starter* i *spring-boot-starter-data-jpa* maven zavisnosti:

```
<dependency>
   <groupId>org.springframework.boot</groupId>
   <artifactId>spring-boot-starter</artifactId>
   <version>1.5.3.RELEASE</version>
</dependency>
<dependency>
   <groupId>org.springframework.boot</groupId>
   <artifactId>spring-boot-starter-data-jpa</artifactId>
   <version>1.5.3.RELEASE</version>
</dependency>
```

spring-boot-starter sadrži sve neophodne auto-konfiguracije za Spring JPA, dok *spring-boot-starter-data-jpa* sadrži sve neophodne zavisnosti (*eng.”dependencies”*), kao što je *hibernate-entitymanager.* Spring Boot konfiguriše Hibernate kao podrazumevani JPA provajder, tako da više nije neophodno da se definiše *entityManagerFactory* zrno (*eng.”bean”*) osim, ukoliko mi

želimo samostalno da ga konfigurišemo. Takođe, Spring Boot samostalno konfiguriše *dataSource* zrno koje se koristi za pristup bazi podataka. U zavisnosti od parametara koji se navedu u *application.properties* fajlu ili u posebnoj klasi označenoj sa *@Configuration* anotacijom. Ukoliko želimo da koristimo JPA sa MySql bazom, što je slučaj u prototipskoj aplikaciji, potrebno je najpre da dodamo *mysql-connector-java*maven zavisnost, a zatim da konfigurišemo parametre za kreiranja *dataSource* zrna u *application.properties* fajlu, na sledeći način:

```
spring.jpa.hibernate.ddl-auto=update
spring.datasource.url=jdbc:mysql://localhost:3306/users
spring.datasource.username=root
spring.datasource.password=root
spring.jpa.properties.hibernate.dialect = org.hibernate.dialect.MySQL5InnoDBDialect
```

Listing 3.1. application.properties fajl

Da bi *dataSource* zrno bilo kreirano, neophodno je da postoji kreirana MySQL baza, u našem slučaju sa imenom "users", sa kredencijalima za pristup koji se navode kao vrednosti parametara *spring.datasource.username* i *spring.datasource.password* u *application.properties* fajlu. Primetimo, da su u gore navedenom fajlu *spring.jpa.hibernate.ddl-auto* i*spring.jpa.properties.hibernate.dialect* parametri vezani za podešavanja hibernate-a okvira. Parametar *spring.jpa.hibernate.ddl-auto*se koristi za inicijalizaciju baze podataka. Vrednost *update* ovoga parametra omogućava dve stvari:

1. Kada se defineše model podataka (Java klasa), tabela zasnovana na tom modelu će biti automatski kreirana u bazi, pri čemu će polja u modelu (atributi java klase) odgovarati kolonama u tabeli.
2. Bilo koja izmena u modelu će takođe izazvati promene u tabeli. Na primer, ukoliko se promeni ime polja u modelu,ili ako se doda novo polje, promeniće se ime kolone u odgovarajućoj tabeli, odnosno dodaće se nova kolona.

Pored vrednosti *update*, gore navedeni parametar može da ima vrednosti *create*, *create-drop* i *validate*. *Create* i *Create-drop* se koriste za kreiranje tabela na osnovu modela, pri čemu je za svako novo pokretanje aplikacije sa tim vrednostima prouzrokuje ponovno kreiranje šeme sa svim tabelama i brisanje postojećih tabela sa podacima.Korišćenje vrednosti *update* ovog parametra je pogodno za razvojni proces. Međutim, za produkciju ovaj parameter bi trebao da

ima vrednost *validate* uz korišćenje alata za migraciju baza, kao što je *Flyway* za upravljanje promenama u šemi baze podataka.

Za potrebe prototipske aplikacije u *MySQL* bazi, koriste se tri modela podataka:

1. User model, za čuvanje informacija o korisnicima sistema;
2. Role model, za čuvanje i upravljanje korisničkim pravima;
3. Account model, za čuvanje korisničkih računa.

Podaci o kompanijama i njihovim akcijama čuvaju se u *MongoDB* bazi.

U listingu 3.2 prikazan je pregled *User* klase:

```
import javax.persistence.Id;
import org.springframework.security.core.GrantedAuthority;
import org.springframework.security.core.authority.SimpleGrantedAuthority;
import org.springframework.security.core.userdetails.UserDetails;
import java.util.Collection;
import java.util.stream.Collectors;
import javax.persistence.*;

@Entity
@Table(uniqueConstraints = @UniqueConstraint(columnNames = "email"))
@JsonPropertyOrder({ "id", "name", "lastname", "email" })
public class User{

@Id
 @GeneratedValue(strategy = GenerationType.AUTO)
private Long id;
@JsonProperty("name")
	private String firstName;
@JsonProperty("lastname")
	private String lastName;
@JsonProperty("email")
	private String email;
	@JsonIgnore
	private String password;

@ManyToMany(fetch = FetchType.EAGER, cascade = {CascadeType.MERGE,
CascadeType.PERSIST})
@JoinTable(
name = "users_roles",
joinColumns = @JoinColumn(
name = "user_id", referencedColumnName = "id"),
inverseJoinColumns = @JoinColumn(
name = "role_id", referencedColumnName = "id"))
```

```
private Collection<Role>roles;


@OneToOne(fetch = FetchType.LAZY,
cascade =  CascadeType.ALL,
mappedBy = "user")
private Account account;


	public Account getAccount() {
return account;
}

public void setAccount(Account account) {
this.account = account;
}

public User() {
	}

public User(String firstName, String lastName, String email, String password,
Collection<Role>roles) {
this.firstName = firstName;
		this.lastName = lastName;
		this.email = email;
		this.password = password;
		this.roles = roles;
}
```

Listing 3.2 User klasa

Bitno je napomenuti sledeće:

1. Svi modeli, kao što je gore navedena klasa, moraju biti označeni sa *@Entity* anotacijom, koja služi da markira klasu, kao *persistent* Java klasu, odnosno kao klasu koja se mapira u tabelu u bazi.
2. *@Table* anotacija se koristi da konfigurisanje detalja tabele u koju će biti mapirana označena java klasa.
3. *@Id* anotacija se koristi da definiše primarni ključ.
4. *@GeneratedValue* anotacija se koristi za konfiguraciju načina generisanja primarnog ključa. U gore navedenoj klasi način generisanja primarnog ključa je automatski.

Odnosno prilikom kreiranja novog reda u tabeli, kolona *Id* će biti automatski popunjena različitim vrednostima za svaki novi red u tabeli.

5. *@OneToOne* anotacija služi da označi 1-1 biderekcionalnu relaciju između tabela *User* i *Account*. Argument *fetch = FetchType.LAZY*, služi za kofigurisanje načina "izvlačenja" podataka iz povezane tabele *Account*. Argument *cascade = CascadeType.ALL* služi za konfigurisanje automatske primene izmena na *Account* tabelu ukoliko dođe do izmena *User* tabele. Na primer, ukoliko dođe do brisanja tabele *User*, tabela *Account* će, takođe, automatski biti izbrisana. *Mapped by= "user"* argument u klasi *User* daje instrukcije hibernate-u da entitet *User* nije vlasink (*eng."owner"*) ove veze i da treba potražiti polje sa imenom *user* u Account entitetu da bi našao konfiguraciju za *JoinColumn/Foregin key* kolonu. Nosilac veze, u ovom slučaju *Account* entitet, sadrži *@JoinColumn* anotaciju koja služi da bi se odredila kolona koja sadrži sekundarni ključ.
6. *@ManyToMany* anotacija služi da se označi više-više veza između entita *User* i *Role*, pri čemu se korišćenjem anotacije *@JoinTable* anotacije pravi posebna tabela, čije se ime navodi kao vrednost parametra *name* ove anotacije, koja kao kolone sadrži id korisnika i id role koja mu odgovara.

Na sličan način definišemo *Role* i *Account* klase :

```
package com.reactive.project.domain;

import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;

@Entity
public class Role {

@Id
   @GeneratedValue(strategy = GenerationType.AUTO)
private int id;
   private String name;

   public Role() {
   }

public Role(String name) {
```

```
this.name = name;
}

public int getId() {
return id;
}

public void setId(int id) {
this.id = id;
}

public String getName() {
return name;
}

public void setName(String name) {
this.name = name;
}

}
```

Listing 3.3 Role klasa

```
@Entity
public class Account {

@Id
   @GeneratedValue(strategy = GenerationType.AUTO)
public long accountId;

@Column(name = "sum")
public String sum;


@OneToOne(fetch = FetchType.LAZY)
@JoinColumn(name = "user_id", nullable = false)
   User user;

   private Account(){};

   public User getUser() {
return user;
}
```

```
public void setUser(User user) {
this.user = user;
}

public long getAccountId() {
return accountId;
}

public String getSum() {
return sum;
}

public void setAccountId(long accountId) {
this.accountId = accountId;
}

public void setSum(String sum) {
this.sum = sum;
}
}
```

Listing 3.4 Account klasa

3.1.1. JPA repository

Kreiranje objekta preslikane klase ne znači i njegovo automatsko čuvanje u bazi podataka. Dokle god se objekat ekplicitno ne sačuva Hibernate Session-jom, taj objekat je transijentan kao i ostali Java objekti. *Spring Data* obezbeđuje konzistentan model za pristup podacima iz različitih vrsta skladišta podataka. Neke od važnih funkcija projekta *Spring Data* su sledeće [11]:

1. Laka integracija sa više skladišta podataka kroz različita spremišta;
2. Mogućnost raščlanjivanja i formiranja upita na osnovu naziva metoda;
3. Obezbeđena standardna CRUD funkcionalnost;
4. Jaka integracijasa Spring MVC-om za otkrivanje REST kontrolera kroz Spring Data Rest.

Projekat *Spring Data* je krovni projekat, sastavljen od brojnih modula. Neki od važnih *Spring Data* modula su sledeći:

1. ***Spring Data Commons*** – Definiše uobičajne koncepte za sve Spring Data module (skladište I metode upita) ,

2. ***Spring Data JPA*** - Obezbeđuje laku integraciju sa relacionim bazama podataka,
3. ***Spring Data MongoDB*** – Obezbeđuje laku integraciju sa MongoDB-om skladištem podataka zasnovanom na dokumentu.
4. **Spring Data REST** – Obezbeđuje funkcionalnost za otkrivanje Spring Data skladišta kao REST servisa,
5. **Spring Data for Apache Cassandra** – Obezbeđuje laku integraciju sa bazom podataka Cassandra,
6. **Spring for Apche Handoop** – Obezbeđuje laku integraciju sa Handoop-om.

Kao što je napomenuto u poglavlju 1.2 , za čuvanje, manipulisanje i pregled korisničkih naloga u prototipskom projektu, kreiran je interfejs *UserRepository* koji nasleđuje *JpaRepository<User,Long>* generički interfejs,koji kao argument uzima *User* klasu i tip primarnog ključa, u našem slučaju *Long*. Zahvaljujući *SimpleJpaRepository* Spring klasi koju Spring Boot automatski konfiguriše za vreme pokretanja aplikacije, možemo da obavljamo sve CRUD (*eng. create, read, update, delete*) operacije nad korisničkim objektima bez implementacije metoda.Mehanizam kreiranja upita, koji je ugrađen u *Spring Data* infrastrukturu omogućuje kreiranje upita nad entitetima klase koja se navodi kao prvi parametar interfejsa *JpaRepository*. Mehanizam otklanja prefikse `find…By`, `read…By`, `query…By`, `count…By`, i `get…By` i nastavlja da parsira ostatak metoda, kreirajući automatski SQL upit za vreme poziva metoda, koji vraća rezultate izvučene prema nazivu atributa klase u odnosu na koji želimo da kreiramo upit,čija se referentna vrednost prosleđuje kao parametar metoda. Pored metoda za pronalaženje, možemo da upotrebimo metode za čitanje, kreiranje upita ili preuzimanje uz naziv metoda. Za potrebe autentikacije korisnika kreiran je metod *public User findUserByEmail (String email)* koji na osnovu prosleđene mail adrese nalazi korisnika u bazi, odnosno izbacuje *UserNotFoundException* ukoliko korisnik sa proseđenim emailom nije pronađen. Za čuvanje korisnika koristi se *public void save(User user)*. Bitno je napomenuti da se *UserRepository* interfejs mora označiti sa *@Repository* anotacijom, koja govori Spring Boot-u da prilikom pokretanja aplikacije kreira zrno tipa ove klase, koje služi kao medijator komunikacije sa bazom. Kreiranje klase za manipulisanje reaktivnim tokovima podataka biće detaljno opisano u narednim poglavljima ovog rada.

3.2 Poslovni sloj

U troslojnom generičkom modelu jasno se odvaja upravljanje podacima, aplikaciona logika i korisnički interfejs. Troslojna arhitektura softvera omogućava transparentno povezivanje korisničkih aplikacija sa različitim izvorima podataka na raznim platformama, a ne samo sa jednim serverom baze podataka. Suštinu ove arhitekture čini poslovni sloj. Poslovni sloj obično sadrži svu poslovnu logiku u aplikaciji. Radni okvir Spring se koristi u ovom sloju za spajanje zrna (*eng."autowiring"*). Ovo je, takođe, sloj u kome započinju granice upravljanja transakcijama. Upravljanje transakcijama može da bude implementirano pomoću *Spring AOP-a*ili *AspectJ-a* . Pre jedne decenije *Enterprise Java Beans (EJB)* je bio najpopularniji pristup za implementiranje poslovnog sloja. Zbog svoje jednostavne prirode, Spring je izabrani radni okvir za poslovni sloj. Za izradu poslovnog sloja u prototipskoj veb aplikaciji korišćena je *@Service* anotacija koja klasu *UserServiceImp.java*, u kojoj je smeštena poslovna logika za čuvanje, prikazivanje i pretragu korisnika na osnovu mail adrese i kreiranje prava za nove korisnike, označava kao servis, što govori Spring-u da prilikom pokretanja automatski kreira zrno (*eng."bean"*) tipa ove klase, koje će se koristiti za manipulaciju poslovnom logikom. Pošto je u protopiskoj aplikaciji, prilikom podešavanja sigurnosti aplikacije, konfigurisana autentikacija korisnika sa podacima iz baze podataka, ova klasa mora da implementira *UserDetailsService* interfejs iz paketa *org.springframework.security.core* i predefineše metod *public UserDetails loadUserByUsername(String email)* koji na osnovu email adrese, koju korisnik unosi prilikom logovanja vrać akorisnika, odnosno izbacuje *UserNotFoundException* ukoliko korisnik sa prosleđenom email adresom nije pronađen. Radi preglednosti i lakšeg održavanja kooda, kreiran je interfejs *UserService*, koji nasleđuje pomenuti interfejs i sadrži popis metoda za pretragu, čuvanje i prikazivanje svih korisnika. Klasa *UserServiceImpl,* kao što se vidi na slici ispod, implementira ovaj intefejs, a samim tim i *UserDetailsService* interfejs.

```
@Service
public class UserServiceImpl implements UserService {

@Autowired
private UserRepository userRepository;

@Autowired
```

```
private BCryptPasswordEncoder passwordEncoder;

@Override
public UserDetails loadUserByUsername(String email) throws UsernameNotFoundException {
        User user = userRepository.findByEmail(email);
        if (user == null){
throw new UsernameNotFoundException("Invalid username or password.");
}
return new org.springframework.security.core.userdetails.User(user.getEmail(),
user.getPassword(),
mapRolesToAuthorities(user.getRoles()));
}

@Override
public User findByEmail(String email){
return userRepository.findByEmail(email);
}


@Override
public User save(UserRegistrationDto registration){
        User user = new User();
user.setFirstName(registration.getFirstName());
user.setLastName(registration.getLastName());
user.setEmail(registration.getEmail());
user.setPassword(passwordEncoder.encode(registration.getPassword()));
user.setRoles(Arrays.asList(new Role("ROLE_USER")));
        return userRepository.save(user);
}

@Override
public List<User>findAll() {
return userRepository.findAll();
}

private Collection<? extends GrantedAuthority>mapRolesToAuthorities(Collection<Role>
roles){
return roles.stream()
                .map(role ->new SimpleGrantedAuthority(role.getName()))
                .collect(Collectors.toList());
}
}
```

Listing 3.5 UserServiceImpl klasa

Primetimo da se u poslovnom sloju koriste zrna tipa *UserRepository*i*BCryptPasswordEncoder* iz paketa *org.springframework.security.crypto.bcrypt*, koje služi za čuvanje korisničkih lozinki u kriptovanoj formi u bazi. Spring IOC container ubacuje (*eng."injection"*) pomenuta zrna u gore navedenu klasu prilikom pokretanja aplikacije, korišenjem *@Autowired* anotacije.

3.3 Sigurnost web aplikacije

Savremeni razvoj veb aplikacija ima mnogo izazova, a jedan od tih izazova je svakako sigurnost veb aplikacije, koja se često u korisničkim i razvojnim krugovima naglašava kao komponenta od esencijalnog značaja. Dobro dizajniran sigurnosni sistem trebao bi da obezbedi:

1. Poverljivost – dostupnost resursa samo ovlašćenim korisnicima,
2. Integritet – podrazumeva koezistentost podataka, nemogućnost manipulacijom podataka od strane neovlašćenih lica,
3. Dostupnost – dostupnost resursa ovlašćenim korisnicima u svakom trenutku,
4. Upotreba sistema isključivo od strane ovlašćenih korisnika.

Sigurnost veb aplikacije podrazumeva definisanje posebnih sigurnosnih mehanizama kako same aplikacije, tako i njenog okruženja, da bi se obezbedile navedene stavke. Dva centralna sigurnosna koncepta, koje implementira sama veb aplikacija su autentikacija i autorizacija. Autentikacija je proces utvrđivanja identiteta korisnika, dok je autorizacija provera da li korisnik ima pristup za izvršavanje specifične akcije.

Spring security obezbeđuje sveobuhvatno rešenje bezbednosti za Java EE poslovne aplikacije. Iako obezbeđuje odličnu podršku za aplikacije zasnovane na Spring-u, Spring Security može da se integriše i sa drugim radnim okvirima [11].

U sledećoj listi istaknuti su neki od mehanizama provere identiteta koje podržava Spring Security [11]:

1. **Provera identiteta zasnovana na obrascu**: jednostavna integracija za osnovne aplikacije,
2. **LDAP** – obično se koristi u većini poslovnih aplikacija,
3. **Java Authetnication and Authorization Service (JAAS)** – standard za proveru identiteta i autorizaciju, deo specifikacije Java EE standarda,
4. **Provera identiteta upravljanja kontejnerom**,

5. **Prilagođeni sistemi provere identiteta.**

Proces konfiguracije bezbednosti u prototpiskoj veb aplikaciji, prolazi kroz sledeće korake:

1. Dodavanje spring security maven zavisnosti,
2. Kreiranje klase koja će da sadrži sva neophodna podešavanja vezana za sigurnost veb aplikacije,
3. Konfigurisanje presretanja svih zahteva – predefinisanje ***protected void configure (HttpSecurity http*)** metoda,
4. Konfigurisanje enkripcije za korisničke lozinke,
5. Predefinisanje ***protected void configure (AuthenticationManagerBuilder auth)*** metoda klase WebSecurityConfigurerAdapter.

U nastaku sledi detaljan prikaz *SecurityConfig* klase zadužene za sigurnost veb aplikacije:

```
@Configuration
public class SecurityConfig extends WebSecurityConfigurerAdapter {


    @Autowired
private UserService userService;

    @Override
protected void configure(HttpSecurity http) throws Exception {
http
            .authorizeRequests()
            .antMatchers(
                  "/registration",
                  "/js/**",
                  "/css/**",
                  "/img/**",
                  "/webjars/**").permitAll()
            .anyRequest().authenticated()
            .and()
            .formLogin()
            .loginPage("/login")
            .permitAll()
            .and()
            .logout()
            .invalidateHttpSession(true)
            .clearAuthentication(true)
            .logoutRequestMatcher(new AntPathRequestMatcher("/logout"))
```

```
            .logoutSuccessUrl("/login?logout")
            .permitAll();
    }



    @Bean
public BCryptPasswordEncoder passwordEncoder(){
return new BCryptPasswordEncoder();
    }

    @Bean
public DaoAuthenticationProvider authenticationProvider(){
       DaoAuthenticationProvider auth = new DaoAuthenticationProvider();
auth.setUserDetailsService(userService);
auth.setPasswordEncoder(passwordEncoder());
return auth;
    }

    @Override
protected void configure(AuthenticationManagerBuilder auth) throws Exception {
auth.authenticationProvider(authenticationProvider());
    }



}
```

Bitno je da istaknemo sledeće stavke:

- *WebSecurityConfigurerAdapte*r – ova klasa obezbeđuje osnovne metode za kreiranje bezbednosne konfiguracije,
- *protected void configure(HttpSecurity http)* – ovaj metod obezbeđuje potrebnu bezbednost za različite URL-ove,
- *authorizeRequests().antMatchers("**/url/").permitAll()* – metod kojim se dopušta pristup url-u navedenom kao parametar antMatchers metoda, bez predhodne autentikacije,
- *formLogin().loginPage("/login")*– metod kojim se konfiguriše autentikacija korišćenjem forme, na adresi navedenoj kao ulazni parametar metoda loginPage,

- *logoutRequestMatcher* - metod kojim se konfiguriše odjavljivanje i završetak korisničke sesije. Kao ulazni parametar metod prima url adresu koja korisnika vodi na odjavljivanje,
- *BCryptPasswordEncoder* klasa iz paketa org.springframework.security.crypto.bcrypt nam koristi za kriptovanje korisničkih lozinki prilikom upisa u bazu i dekriptovanje istih, prilikom provere identiteta korisnika,
- *DaoAuthenticationProvider* klasa iz paketa *org.springframework.security.authentication.dao* nam služi za konfigurisanje autentikacije korišćenjem korisničkih naloga koji se čuvaju u bazi,
- Konačno, ***protected void configure (AuthenticationManagerBuilder auth)*** metod nam služi za konfigurisanje načina autentikacije korisnika.

3.4 Višejezičnost u veb aplikaciji

Jedan od fundamentalnih koncepata pri razvoju savremenih veb aplikacija je svakako internacionalizacija. Pod ovim pojmom podrazumeva se prilagođavanje sadržaja, izgleda, dokumentacije i uputsva za korišćenje veb aplikacije korisnicima koji pripadaju različitim govornim područijima, kulturama i geografskim regionima. To je naročito važno ukoliko je aplikacija namenjena za globalnu upotrebu. Umesto punog naziva "internacionalizacija", često se koristi skraćenica *"i18n"*, koja označava da postoji 18 znakova između znaka 'i' i znaka 'n' u nazivu na engleskom jeziku. Prototipska aplikacija je dizajnirana tako, da nudi opcije korisniku za izbor odgovarajućeg jezika.

Implementacija višejezičnosti u prototipskoj aplikaciji prolazi kroz sledeće etape:

1. Kreiranje fajlova sa porukama (*eng."message files")* u kojima će se prevod čuvati u formi par-ključ. Podrazumevano, Spring Boot aplikacija fajlove sa porukama traži na putanji *src/main/resources*. Fajlovi se čuvaju u formatu messages_XX.properties pri čemu je XX kod za odgovarajući jezik (kodovi za jezike dostupni u dokumentaciji Spring boot-a).

2. Kreiranje klase za konfiguraciju koja će biti označena sa *@Configuration* anotacijom i nasleđivati klasu *WebMvcConfigurerAdapter.java*, da bi Spring IOC container prilikom pokretanja znao da se ta klasa koristi za konfiguraciju aplikacije.Dalje,da bi Spring Boot aplikacija bila u stanju da odredi koji je lokalni jezik koji se koristi i na osnovu toga učita odgovarajući fajl za prevod,potrebno je da definišemo *LocalResolver* zrno na sledeći način :

```
@Bean
public LocaleResolver localeResolver() {
   SessionLocaleResolver slr = newSessionLocaleResolver();
   slr.setDefaultLocale(Locale.SR);
   returnslr;
}
```

3. KreiranjeLocalChangeInterceptor zrna koji služi za promenu lokalnog jezika u zavisnosti od vrednosti prosleđenog parametra *lang*, kroz http zahtev.

```
@Bean
Public LocaleChangeInterceptor localeChangeInterceptor() {
   LocaleChangeInterceptor lci = newLocaleChangeInterceptor();
   lci.setParamName("lang");
   returnlci;
}
```

4. Konačno, potrebno je da gore kreirano zrno bude dodato u registar presretanja aplikacije (*eng. "application interception registry").* Da bismo ovo postigli, u klasi za konfiguraciju trebamo da predefinišemo *addInterceptors()* metod na sledeći način:

```
@Override
public void addInterceptors(InterceptorRegistry registry) {
   registry.addInterceptor(localeChangeInterceptor());
}
```

3.5 Administriranje aplikacije

U nastavku ćemo opisati mehanizme korišćene u izradi administrativnog dela prototipske aplikacije, sa osvrtom na *Jackson* biblioteku, korištenu za serilizaciju i deserilizaciju Java klasa u JSON format i obrnuto [12]. U tu svrhu potrebno je u *pom.xml* fajl uključiti sledeću maven zavisnost:

<dependency>

<groupId>com.fasterxml.jackson.core</groupId>

<artifactId>Jackson-core</artifactId>

<version>2.9.5</version>

</dependency>

U trenutku pisanja ovog rada trenutna poslednja verzija biblioteke je 2.9.5

3.5.1 Prikazivanje liste svih registrovanih korisnika sistema

Dodavanje/pregled/izmena/brisanje korisnika je funkcionalnost kojoj mogu pristupiti samo administratori aplikacije. Kojim funkcionalnostima mogu pristupiti korisnici određuje se na osnovu uloga koje su im dodeljene. Definisane su tri uloge (*eng."roles"*):

ROLE_USER

ROLE_ADMIN

ROLE_SUPER_ADMIN

Ukoliko korisnik ima ulogu "korisnik" *(ROLE_USER)*, koja se automatski dodeljuje prilikom registracije, on nema pristup administrativnom delu aplikacije. Nove administratore aplikacije (korisnike sa ulogom *ROLE_ADMIN*) može da dodeljuje samo super administrator (korisnik sa ulogom *ROLE_SUPER_ADMIN*). Super administrator se dodaje direktno u bazu podataka. Kontrola pristupa metodu, na osnovu dodeljene korisničke uloge, implementirana je korišćenjem *@Secured* anotacije iz paketa *org.springframework.security.access.annotation*, koja na osnovu prosleđenog niza uloga kojima je dozvoljeno pozivanje metode, poredi ulogu koju ima trenutni korisnik sa prosleđenim nizom uloga i na osnovu toga izvršava telo metoda, odnosno ukoliko

korisnik nema prava da poziva taj metod, vraća odgovarajući izuzetak (*eng.“exception“*).Na serverskoj strani, REST kontroler klasa *RestRequestController* obezbeđuje listu korisnika sistema, pozivajući metod *public List<User> findAll()* skladišta *UserRepository*, definisanog u interfejsu *JpaRepository* koji dati interfejs nasleđuje,kao što je opisano u predhodnom poglavlju. Da bi se vršila automatska konverzija java objekata u JSON format prilikom poziva metoda za pregled korisnika, potrebno je da *RestRequestController* klasa bude markirana anotacijom *@RestRequestController. @RestRequestController* anotacija je specijalizovana verzija kontrolera. U sebi sadrži *@Controller i @ResponseBody* anotacije i kao rezultat pojednostavljuje implementaciju kontroler klase za rukovanje REST zahtevima.

```
@Secured("ROLE_ADMIN,ROLE_SUPER_ADMIN")
@RequestMapping(value = "/findAllUsers",method = RequestMethod.GET)
List<User>findAllUsers()
{
return userService.findAll();
}
```

Listing 3.6 metod za vraćanje svih registrovanih korisnika

Sledi primer JSON poruke koja se prosleđuje na klijentsku stranu aplikacije:


```
[
  {
    "id": 1,
    "name": "Nemanja",
    "lastname": "Zirojevic",
    "email": "nemanjax@gmail.com",


  },
    {
    "id": 2,
    "name": "Petar",
"lastname": "Petrovic",
    "email": "petar@gmail.com",
  }
]
```

Korišćenjem *Jackson* biblioteke postignuto je formatiranje polja JSON poruke, upravljanje redosledom prikazivanja polja poruke, kao i eliminacija polja password klase *User* u telu poruke. *Jackson* je vrlo popularna biblioteka za serilizaciju Java objekata u JSON format, i obrnuto, re-kreiranje Java objekata na osnovu JSON poruke [12]. Glavne prednosti *Jackson* biblioteke su sledeće:

1. **Jednostavna je za korištenje** – Jackson API obezbeđuje jednostavne metode kako bi se pojednostavili tipični slučajevi korišćenja;
2. **Pored jdk-a (*eng."java development kit"*), ne zahteva nikakve druge biblioteke**;
3. **Dobre peformanse**;
4. **Biblioteka je otvorenog koda**;

Navodimo neke od najčešće korišćenih *Jackson* anotacija:

@JsonPropertyOrder anotacija se koristi da bi se definisao redosled prikazivanja polja prilikom serilizacije objekta u JSON format ,

@JsonIgnoreProperty anotacija se koristi da bi se označilo da se određeno polje klase ne serilizuje u JSON format ,

@JsonProperty anotacija se koristi da bi se promenilo ime polja prilikom serilizacije u JSON format ,

@JsonSerialize se koristi da označi klasu koja će se koristiti za serilizaciju određenog polja ,

@JsonDeserilize se koristi da se označi klasa koja će se koristiti za deserilizaciju,itd...

Kreiranje tabele korisnika na klijentskoj strani korišćenjem podataka iz prosleđene JSON poruke, implementirano je korišćenjem *DataTables.js* biblioteke na klijentskoj strani.

3.5.2 Upravljanje korisničkim nalozima

Za dodavanje/izmenu i brisanje korisnika koriste se, respektivno, sledeće metode *RestRequestsController* klase:

```
@Secured("ROLE_ADMIN,ROLE_SUPER_ADMIN")
@RequestMapping(value="/deleteUser",method = RequestMethod.DELETE)
public void deleteUser(@RequestBody User user)
{
userService.delete(user);
}
```

```
@Secured("ROLE_ADMIN,ROLE_SUPER_ADMIN")
@RequestMapping(value = "/updateAndSaveUser",method = RequestMethod.POST)
User updateUser(@RequestBody User user)
{
return userService.save(user);
}
```

Listing 3.7 metodi za brisanje, izmenu korisnika

Primetimo da su metode *public User save (User user)*, *public void delete (User user)* implementirane u poslovnom sloju aplikacije,odnosno definisane u *UserRepository* interfejsu. Nakon što se na klijentskoj strani aplikacije u formi tabele prikažu korisnici sistema, administrator je u mogućnosti da izabere korisnika koga želi da obriše/izmeni detalje, i zatim se novi detalji za datog korisnika (koji uključuju njegov id, koji je skriven u tabeli, ime, prezime i email adresu) šalju korištenjem *ajax javascript fukcije* u JSON formatudo gore navedenih metoda, koji korišćenjem *@RequestBody* Spring anotacije re-kreiranju Java objekat, nakon čega se on šalje u bazu. Isti postupak važi i za brisanje korisnika, s tim da će *AJAX javascript fukcije* kao url parametar imati adresu *'/deleteUser'*, koja mapira zahtev do metoda za brisanje, opisanog iznad. Postupak dodavanja korisnika se svodi na unos imena, prezimena, email adrese u odgovarajuću formu, zatim se objekat klase *UserRegistrationDto* u JSON formatu prosleđuje do metoda na url putanji *'/addUser'* čija je implementacija prikazana u listingu 3.8:

```
@Secured("ROLE_ADMIN,ROLE_SUPER_ADMIN")
@RequestMapping(value = "/addUser",method = RequestMethod.POST)
public void addNewUser(@ModelAttribute("user") @Valid UserRegistrationDto userDto)
{
userService.save(userDto);
}
```

Listing 3.8 metod za dodavanje korisnika

Primetimo da ovaj metod, najpre proverava validnost polja prosleđene forme na osnovu primljenih vrednosti za polja iz JSON poruke, i tek pošto sva polja zadovoljavaju kriterijume validnosti, nastavlja se sa izvršavanjem tela metoda. Sledi prikaz *UserRegistrationDto* klase:

```
@FieldMatch.List({
@FieldMatch(first = "password", second = "confirmPassword", message = "The password fields must match"),
@FieldMatch(first = "email", second = "confirmEmail", message = "The email fields must match")
```

```
})
public class UserRegistrationDto {

@NotEmpty
private String firstName;

@NotEmpty
private String lastName;

@NotEmpty
private String password;

@NotEmpty
private String confirmPassword;

@Email
   @NotEmpty
private String email;

@Email
   @NotEmpty
private String confirmEmail;

@AssertTrue
private Boolean terms;



  //getters and setters…

}
```

Listing 3.9 UserRegistrationDto klasa

U opisanoj klasi anotacije *@NotEmpty, @Email, @AssertTrue* iz paketa *org.hibernate.validator.contrains ,* služe, respektivno , da bi se obezbedilo da polja markirana ovim anotacijama ne sadrže *null* vrednosti, budu u formi email-a (*xxx@domain.com*) i podrazumevana vrednost za polja tipa *Boolean* bude *true.*

3.5.3 Ažuriranje kompanija

Pored upravljanja i pregleda korisničkih naloga, administratoru aplikacije omogućen je pregled registrovanih kompanija, dodavanje novih i brisanje postojećih. S obzirom da se kompanije, odnosno cene njihovih akcija prikazuju kao reaktivni tok podataka sa izmenjenim vrednostima , detalja na implementacija perzistentnog sloja za CRUD operacije nad objektima kompanija u MongoDB biće posebno obrađena u narednom poglavlju. Mehanizam pregleda kompanija je identičan predhodno opisanom mehanizmu za pregled korisnika. Na klijentskoj strani prikazuje se tabela (korišćenjem pomenute *DataTables.js* biblioteke) sa svim kompanijama, formirana od sledećeg JSON niza:

```
data : [
  {
   "id": 1,
   "price": 134.45,
   "companyname": "Google"
  },
  {
   "id": 2,
   "price": 287,
   "companyname": "Amazon"
  }
]
```

Odabirom kompanije korisnik *ajax javascript funkcijom* prosleđuje JSON string u gore navedenom formatu do odgovarajućih metoda koje, korišćenjem opisane *@RequestBody* Spring anotacije, re-kreiraju java objekat korisnički definisane klase *Stock*, koji se zatim smešta, odnosno po potrebi briše iz Mongo baze.

3.5.4. Metrika

Za prikupljanje statistike aktivnosti JVM, trenutnog stanja aplikacije, HTTP poziva, kao i poziva REST kontrolera, serverska strana koristi *spring-boot-starter-actuator* biblioteku. Ovu biblioteku ubacujemo kao maven zavisnost u *pom.xml* fajl:

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-actuator</artifactId>

</dependency>

Uključivanjem gore opisane maven zavisnosti, Spring Boot automatski kreira krajnje REST tačke, za prikaz statistike vezane za aplikaciju.

URL putanje kojima možemo da pristupamo da bismo dobili gore navedene informacije su:

URL na putanji */health* vraća JSON string sa podacima o trenutnom stanju aplikacije. To se obično koristi za monitoring softvera da bi se pravovremeno reagovalo i lakše odklonile greške ukoliko dodje do problema kao što su gubljenje konekcije/nemogućnost povezivanja sa bazom, nedostatak prostora na disku, itd.. ,

URL na putanji */info* vraća JSON string sa podacima o aplikaciji, kao što su : trenutna verzija aplikacije, ime I kratak opis aplikacije. JSON string koji vraća navedene podatke formira se na osnovu vrednosti parametara *info.app.name, info.app.description, info.app.version* koje se postavljaju u *application.properties* fajlu;

URL na putanji */metrics* vraća JSON string sa podacima o operativnom sistemu na kome se izvršava aplikacija, statusu JVM, slobodnoj memoriji, broju aktivnih niti,itd.. Sledi primer jednog takvog JSON stringa :


```
{
   "mem" : 193024,
   "mem.free" : 87693,
   "processors" : 4,
   "instance.uptime" : 305027,
   "uptime" : 307077,
   "systemload.average" : 0.11,
   "heap.committed" : 193024,
   "heap.init" : 124928,
   "heap.used" : 105330,
   "heap" : 1764352,
   "threads.peak" : 22,
   "threads.daemon" : 19,
   "threads" : 22,
   "classes" : 5819,
   "classes.loaded" : 5819,
   "classes.unloaded" : 0,
```

```
    "gc.ps_scavenge.count" : 7,
    "gc.ps_scavenge.time" : 54,
    "gc.ps_marksweep.count" : 1,
    "gc.ps_marksweep.time" : 44,
    "httpsessions.max" : -1,
    "httpsessions.active" : 0,
    "counter.status.200.root" : 1,
    "gauge.response.root" : 37.0
}
```


URL na putanji */loggers* nam pruža uvid u nivo logovanja u aplikaciji i daje mogućnost njegove izmene;

URL na putanji */logfile* vraća logove aplikacije;

URL na putanji */sessions* vraća listu HTTP sesija dodeljenih od strane Spring Session-a, itd…

Prikazivanje statistike je podesivo postavljanjem parametara, u *application.properties* fajlu, na odgovarajuću vrednost. Na primer, ukoliko je parameter ***endpoints.loggers.enabled=false*** prisutan u gore pomenutom fajlu, prikazivanje statistike logovanja, koje je podrazumevano omogućeno, neće biti prikazivano korisnicima. Prikupljanje statistike je implementirano u klasi *RestRequestsController*, koja u sebi sadrži java zrna (*eng.“beans“*) tipa *MetricsEndpoint*, *HealthEndpoint, LoggersEndpoint,* sve iz paketa *org.springframework.actuate.endpoint,* kreirana za vreme pokretanja aplikacije, korištenjem *@Autowired* anotacije. REST kontroler *RestRequestsController* u sebi sadrži metode koje na osnovu *url* zahteva, pozivaju *invoke()* metod definisan nad pomenutim JAVA zrnima, koji vraća objekte serijalizovane u JSON string koji se šalje na klijentsku stranu.

3.5.5 Izrada Controller klase

Radi bolje preglednosti i lakšeg održavanja koda, logika za mapiranje URL zahteva za prikaz HTML stranica smeštena je u korisnički definisanu *PageController* klasu. Ova klasa označena je Spring-ovom *@Controller* anotacijom o kojoj je bilo reči u predhodnim poglavljima. Podrazumevano, Spring Boot statičke fajlove, kao što su html stranice, javascript, css fajlovi traži u okviru foldera *resources/static*. Ovo podešavanje je konfigurabilno, međutim za potrebe

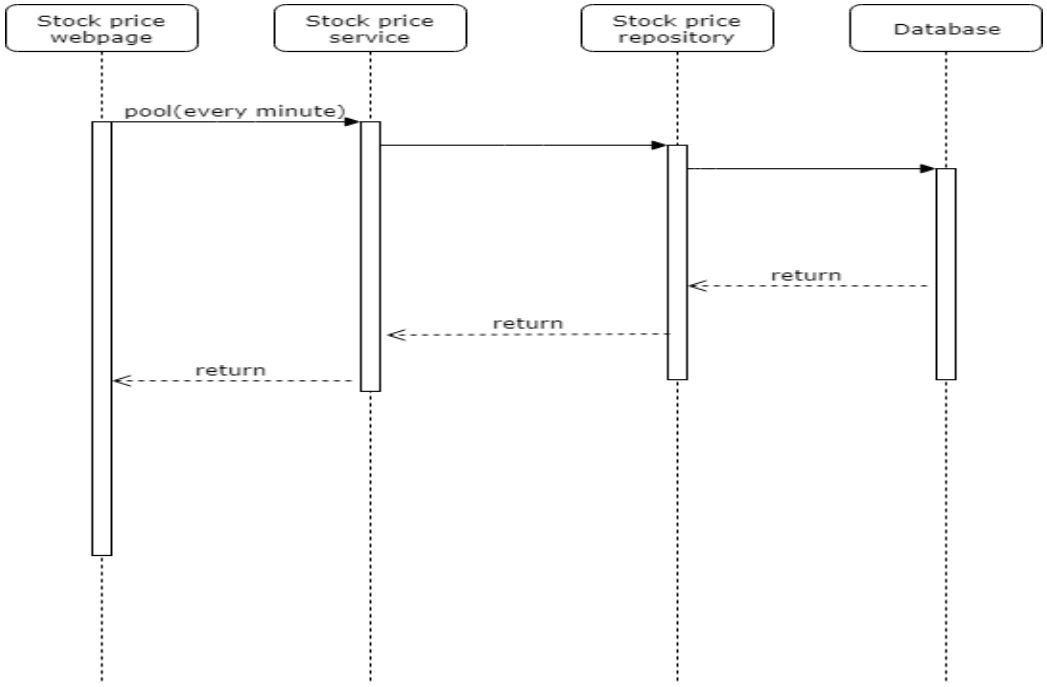
prototipske aplikacije, nije menjano. Za prikaz stranica korišten je interfejs *ModelAndView*, koji nam dozovoljava da čuvamo kako model (*eng.“model“*) , tako i prikaz (*eng.“view“*), koji zatim *DistpacherServlet* , spring-ova klasa, na osnovu unete url adrese vraća korišćenjem mehanizma pregledača (*eng.“view resolver“*).

4. Implementacija reaktivnog toka podataka za pregled cena akcija kompanija korišćenjem reactive Spring Flux tehnologije, reactive MongoDB

Na početku ovog poglavlja daćemo kratak pregled tradicionalnog i reaktivnog načina razvoja pomenute aplikacije, njihove uzajamne sličnosti, razlike i mane. Zatim, sledi prikaz implementacije reaktivnog perzistenog sloja podataka i detaljan opis reaktivnog toka podataka implementiranog korišćenjem *Reactive Spring* tehnologije.

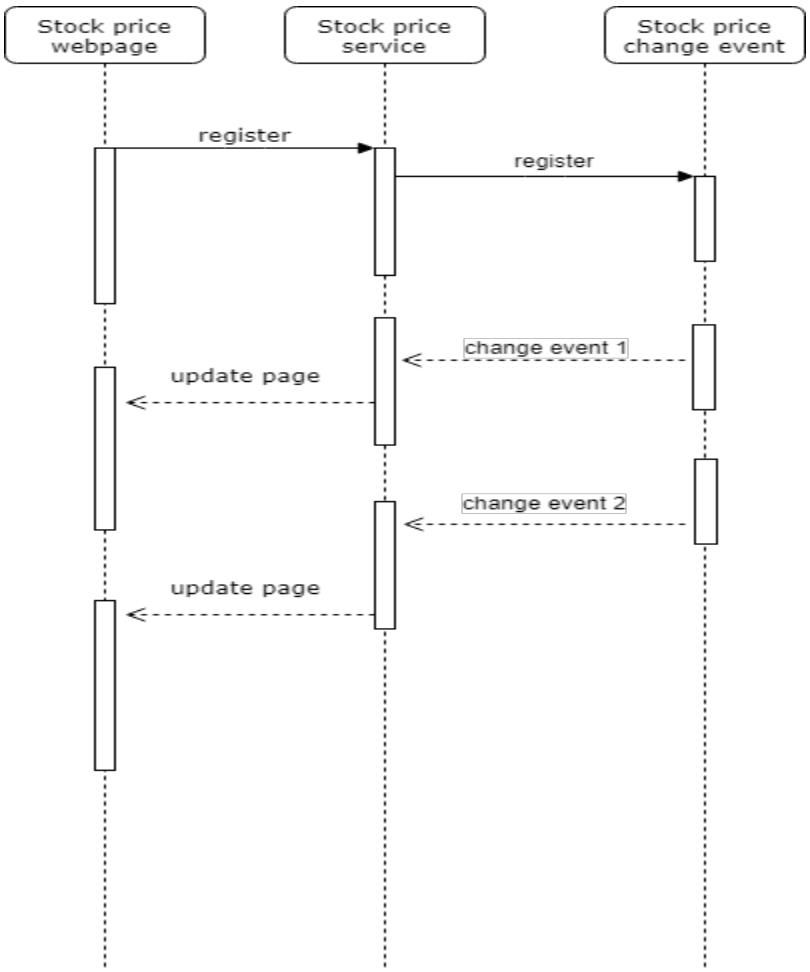
4.1 Razlika između tradicionalnog i reaktivnog pristupa

Tradicionalni pristup se zasniva na proveri da li je cena akcije promenjena. Na slici 4.1 prikazan je dijagram sekvenci koji ilustruje pomenuti pristup razvoju aplikacije.



Slika 4.1 Tradicionalni pristup razvoju aplikacije
(preuzeto iz ref [1])

Nakon što je učitana, stranica šalje AJAX zahtev ka metodu kontrolera za pregled cena akcija, za dobijanje najnovije cene u određenim vremenskim intervalima. 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. 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*. Na slici 4.2 prikazan je dijagram sekvence koji ilustruje reaktivni pristup razvoju aplikacije.



Slika 4.2 Reaktivni pristup razvoju aplikacije (preuzeto iz ref [1])

Reaktivni pristup obično uključuje tri koraka [13]:

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 [13]. Ako lanac događaja uključuje posrednika za poruke, postaje još

složeniji. 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 konkurencija za programske niti i njihove resurse. U rekativnom pristupu programske niti traju kratko, pa, prema tome, manja je konkurencija za resurse. Ovakav pristup obezbeđuje mogućnost obrade većeg broja prispelih korisničkih zahteva.Takođe, koristeći reaktivni 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.

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. 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*.

4.2 Skladištenje podataka u reactive MongoDB

Kao što je bilo reči u predhodnim poglavljima podaci o kompanijama čuvaju se u formi dokumenta u mongo bazi. Za kreiranje dokumenata koristićemo *@Documented* anotaciju koja korisnički definisanu klasu *Stock* označava kao dokument koji će biti ubačen u mongo bazu. Pomenuta klasa sadrži atribute koji opisuju kompaniju : id – što označava id kompanije, name- ime kompanije, price – početna cena akcija, kao što je prikazano u listingu 4.0

```
@Document
public class Stock {

    @Id
    private String id;

    private int price;
```

```
    private String name;


    public Stock()
    {

    }

    public Stock(String id, int price,String name) {
        this.id = id;
        this.price = price;
        this.name = name;
    }


    public int getPrice() {
        return price;
    }

    public void setPrice(int price) {
        this.price = price;
    }

    public String getId() {
        return id;
    }

    public void setId(String id) {
        this.id = id;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }
}
```

Listing 4.0 Stock klasa

Dokument koji se ubacuje u mongo bazu, ima sledeću JSON formu:

{
"id":"1",
"price" : "30",
"name":"Google"
}

Sve *reactive mongo* zavisnosti (*eng."dependencies"*), potrebne za rad sa asihronim tokovima podataka sadržane su u sledećem maven dependency-u koji uključujemo u *pom.xml* fajl:

<dependency>
<groupId>org.springframework.boot</groupId>
<artifactId>spring-boot-starter-data-mongodb-reactive</artifactId>
</dependency>

Uključivanjem pomenutog maven dependency-a dobijamo Spring Data MongoDB dependency, koji sadrži Data Commons 2.0 biblioteku sa reaktivnom podrškom, kao i asihronu i *reactive-streams* verziju MongoDB drajvera. Da bismo obezbedili podršku za *reactive Spring Data*, potrebno je da kreiramo konfiguracionu klasu koju označavamo sa *@EnableReactiveMongoRepositories* anotacijom, kao što je prikazano u listingu 4.1

```
package com.reactive.project.config;


import com.reactive.project.repository.UserRepository;
import org.springframework.context.annotation.Configuration;
import org.springframework.data.mongodb.repository.config.EnableMongoRepositories;

@EnableMongoRepositories(basePackageClasses = UserRepository.class)
@Configuration
public class MongoDBConfig  {

}
```

Listing 4.1 MongoDBConfig klasa

Za pristup mongo bazi, upis i čitanje reaktivnih objekata, potreban je reaktivni perzistenti sloj u aplikaciji. S tim ciljem, interfejs *StockRepository* nasleđuje interfejs *ReactiveMongoRepository<Stock,String>* iz paketa *org.springframework.data.mongodb.repository,* koji obezbeđuje metode za manipulaciju nad dokumentima. Zahvaljujući automatskoj konfiguraciji SpringBoot-a, implementacija ovog interefejsa *SimpleReactiveMongoRepository* se automatski kreira za vreme pokretanja aplikacije. Na taj način dobijaju se sve CRUD operacije već dostupne, bez potrebe pisanja koda. Sledi pregled nekih metoda iz *SimpleReactiveMongoRepository* klase:

reactor.core.publisher.Flux<T> findAll ();
reactor.core.publisher.Mono<T> findById (ID id);
<S extends T> reactor.core.publisher.Mono<S> save (S entity);
reactor.core.publisher.Mono<Void> delete (T entity);

Primetimo da su svi metodi asihroni i da vraćaju implementaciju *Publisher* interfejsa u formi *Flux* ili *Mono*. U listingu 4.2 dat je prikaz *StockRepository* klase:

```
package com.reactive.project.repository;

import com.reactive.project.model.Stock;
import org.springframework.data.mongodb.repository.ReactiveMongoRepository;
import org.springframework.stereotype.Repository;


@Repository
public interface StockRepository extends ReactiveMongoRepository<Stock,String> {
```

```
}
```
Listing 4.2 StockRepository klasa

Kreiranje reaktivnog poslovnog sloja je potpuno analogno kreiranju opisanog poslovnog sloja za manipulisanje korisnicima. Klasa *StockService* se označava *@Service* anotacijom i mehanizmom *dependency injectiona* za vreme pokretanja aplikacije koristeći *@Autowired* anotaciju ubacuje se instanca *StockRepository* klase, kao što je prikazano u listingu 4.3

```
import com.reactive.project.model.Stock;
import com.reactive.project.repository.StockRepository;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;
import reactor.core.publisher.Flux;
import reactor.core.publisher.Mono;

@Service
public class StockService {

    @Autowired
    StockRepository stockRepository;

    public Mono<Stock> findById(String stockId){
        return stockRepository.findById(stockId);
    }
    public void save(Stock stock)
    {
        stockRepository.save(stock);
    }
    public void delete(Stock stock)
    {
        stockRepository.delete(stock);
    }
    public Flux<Stock> saveAll(Flux<Stock> stocks)
    {
        return stockRepository.saveAll(stocks);
    }

}
```
Listing 4.3 StockService klasa

Instanca kreiranog servisa se *dependecy injection* mehanizmom za vreme pokretanja aplikacije ubacuje u kontroler klasu i koristi se u metodima za obradu URL zahteva.

4.3 Implementacija reaktivnog toka podataka

Spring *WebFlux* koristi biblioteku *Reactor* kao svoju reaktivnu podršku. *Reactor* je implementacija *Reactive Streams* 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. *Flux* predstavlja reaktivni tok podataka emituje 0 do n elemenata. Za početak je nephodno da dodamo sledeći *maven dependency* u pom.xml fajl:

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

Kreiraćemo klasu dokumenta *Company* sa atributima *id* i *name*, tipa *integer* i *String*, respektivno. Datu klasu označićemo anotacijom *@Documented* radi upisa u Mongo bazu. Takođe, kreiraćemo klasu *StockEvent*, koja za atribute ima objekat tipa *Company* i *price* atribut tipa *int* koji predstavlja cenu akcija za izabranu kompaniju, koji se menja u određenim vremenskim intervalima.

Za interakciju sa bazom, odnosno upis i pregled dostupnih kompanija potrebno nam je da kreiramo reaktivni perzistentni sloj.

S tim ciljem, kreiraćemo interfejs *StockRepository* koji nasleđuje Spring-ovu klasu *ReactiveMongoRepository<Company, String>*. Kreiraćemo instancu *CommandlineRunner interfejsa* za umetanje početnih podataka o akcijama u *ReactiveMongo* bazu prilikom pokretanja aplikacije, kao što je prikazano u listingu 4.4. Metod *stockReposity.save (Stock stock)* se koristi za čuvanje dokumenta *Stock* u Mongo bazu. Metod *subscribe* omogućava nam da nakon što je upis podataka u bazu uspešno izvršen, u konzoli vidimo unešene podatke.

```
@Bean
CommandLineRunner stocks(StockRepository stockRepository)
{
   return args ->
         stockRepository
               .deleteAll()
               .subscribe(null,null,() ->{

                  Stream.of(new Stock("1",200,"Google"),new Stock("2",300,"Yahoo"),new
Stock("3",200,"Amazon"),new Stock("4",400,"Microsoft"))
                              .forEach(stock ->
                             stockRepository
```

```
                                    .save(stock)
                                    .subscribe(System.out::println));

                });
}
```

Listing 4.4 CommandLineRunner zrno

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 koristi instancu reaktivnog servisa *StockService* za vraćanje instanci tipa *Flux<StockEvent>* prilikom izvršavanja metoda. U sledećem isečku koda prikazan je reaktivni kontroler.

```
@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);
                    });
   };
```

}

Metoda *Flux<StockEvent> getStockPrice (@PathVariable (“stockId”) String stockId)* generiše tok tipa *StockEvent*. Prikaz *StockEvent* klase dat je u listingu 4.5.

```
public class StockEvent {
    Stock stock;
    int stockPrice;

    public StockEvent(Stock stock, int stockPrice) {
        this.stock = stock;
        this.stockPrice = stockPrice;
    }

    public Stock getStock() {
        return stock;
    }

    public void setStock(Stock stock) {
        this.stock = stock;
    }

    public int getStockPrice() {
        return stockPrice;
    }

    public void setStockPrice(int stockPrice) {
        this.stockPrice = stockPrice;
    }
}
```

Listing 4.5 StockEvent klasa

Pomenuti metod na osnovu prosleđenog id-a kompanije, vraća naziv kompanije i 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. 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. Dakle, ideja je da se stvori lanac događaja pri čemu svaka karika u tom lancu reaguje događaj iz predhodne karike i na kraju se podaci prikazuju korisniku. U slučaju prototipske aplikacije, kada korisnik izabere neku od ponuđenih kompanija, mehanizam izvršavanja je sledeći:

Na klijentskoj strani kreira se instanca *EventSource* interfejsa, kojoj se prosleđuje gore navedena URL putanja */stock/events/{stockId}*, čime se vrši "prijava" za događaje sa pomenute URL putanje.

Pozivom metoda *stockService.findById(stockId)* pronalazi se kompanija sa prosleđenim ID-jem u Mongo bazi,

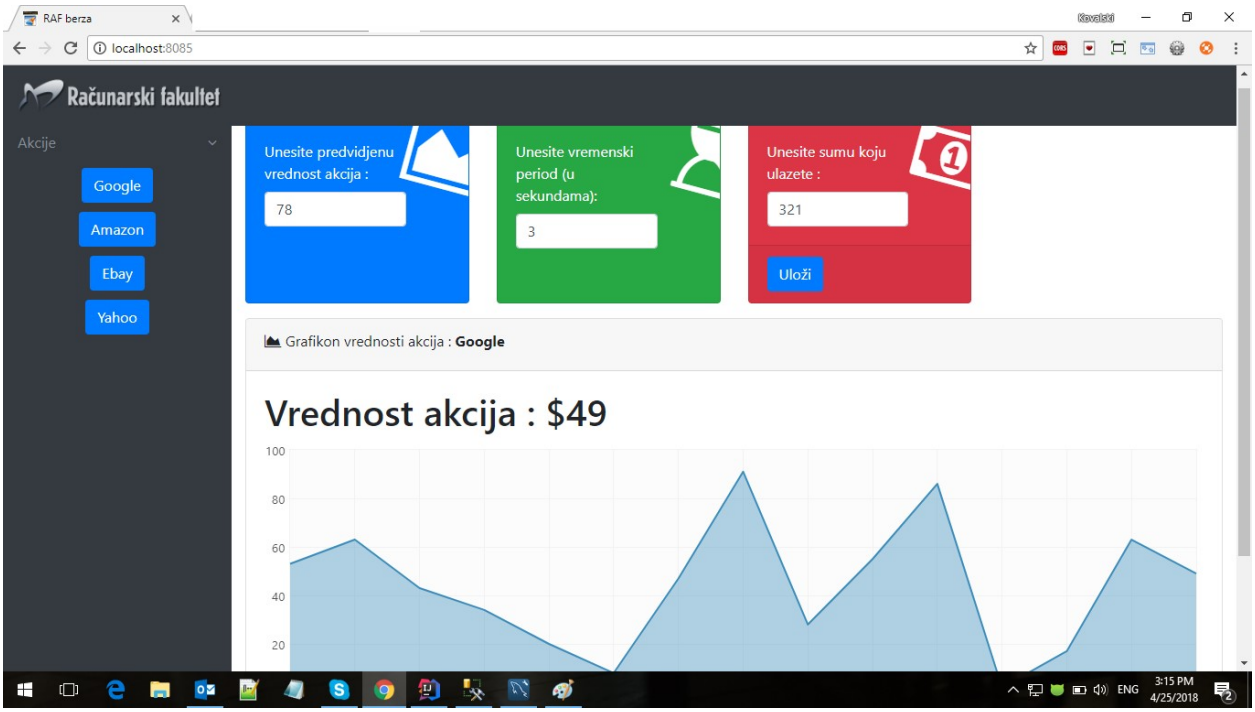
Pravi se vremenski interval od dve sekunde, *Flux<Long> interval = Flux.interval(Duration.ofSeconds(2))*, čime se obezbeđuje da se novi tok cena akcije *Flux<StockEvent> stockEventFlux = Flux.fromStream(Stream.generate(()->new StockEvent(stock,rand.nextInt(98) +1)))* formira svake dve sekunde, i vraća korisniku return *Flux.zip(interval,stockEventFlux),*

Kada se desi promena cene akcija, što je svake dve sekunde, tada se JSON niz vraća do klijentske strane i što dovodi do promene izgleda grafikona, odnosno reakcije na događaj.

Način na koji klijentska strana "prihvata" poslati JSON niz, kao i detalji kreiranja instance *EventSource* interfejsa biće prikazani u narednom poglavlju.

5. Implementacija klijentske strane aplikacije

Za prikaz podataka na klijentskoj strani aplikacije, koristi se *EventSource* javascript interfejs. Instanca pomenutog interfejsa otvara permanentnu konekciju ka serveru koji, kao što je napomenuto u predhodnom poglavlju, šalje događaje (*eng.”events”*) u *text/event-stream* formatu. Na listingu 5.1 dat je prikaz funkcije na klijentskoj strani aplikacije, koja se poziva izborom neke od kompanija iz panela *“Akcije”*, kao što je prikazano na slici 5.1.



Slika 5.1 Glavna stranica aplikacije

```
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, // Drawing is faster without shadows
            color: "#3c8dbc"
        },
        lines: {
            fill: true, //Converts the line chart to area chart
            color: "#3c8dbc"
        },
        yaxis: {
            min: 0,
            max: 100,
            show: true
        },
        xaxis: {
            show: true
        }
    };


    if (!!window.EventSource)
    {

        var source = new EventSource('/rest/stock/events/' + id);
        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);
```

```
    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;
    }

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



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


    }

    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) {

                var moneyInput = $("#input-money").val().trim();
                var newAmount = moneyInput - ( moneyInput * 90)/100;
                $("#input-money").val(parseInt(newAmount));
                optionsChanged = 2;
                count = 0;
            }
            if (stockPrice >= inputValue)
            {

                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;
            }

        }

    }
};
```

Listing 5.1 Javascript funkcije na glavnoj stranici

Primetimo da za instancu EventSource interfejsa vežemo tri osuškivača događaja, kao što je prikazano u listingu 5.2.

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

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

source.addEventListener('error', function (e) {

             // code

    }, false);
```

Listing 5.2 EventSouce registrovanje 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, što je prikazano u listingu 5.3.

```
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));

}
```

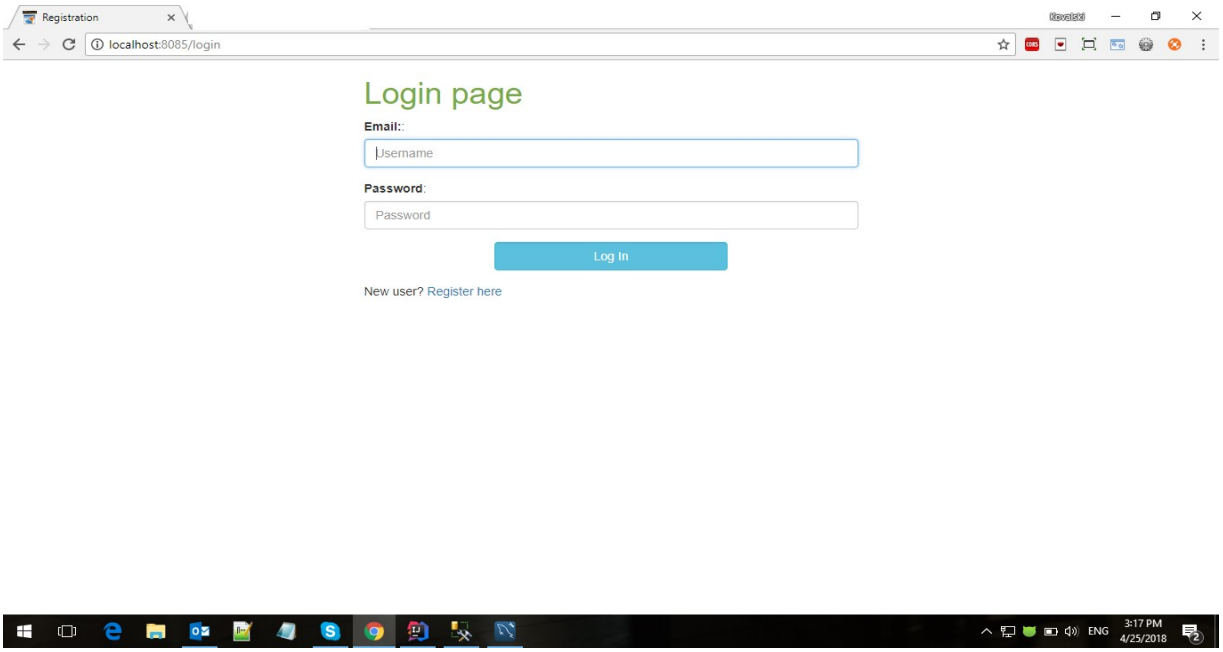
Listing 5.3 Telo metoda koji se izvršava kada dođe nova poruka na server

Incijalno javascript varijabla *optionsChanged* se postavlja na vrednost nula, što znači da se prilikom izbora kompanije za kreiranje grafikona njenih akcija koristi blok koda u 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. Korisniku je omogućeno da unosom vrednosti u tri input polja, respektivno, 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. U slučaju da korisnik izabere ulaganje novca, odnosno klikom na

dugme *"Uloži"*, 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"*.
Prilikom izrade klijentske strane aplikacije, za potrebe stilizacije i dobijanja responsive dizajna korišćen je *bootstrap* css framework. Za kreiranje grafikona korišćena je *jQuery plot* biblioteka.

6. Prikaz rada prototipske aplikacije

Nakon pokretanja aplikacije, korisnik može da se prijavi na sistem koristeći neki od predhodno kreiranih naloga, ili da kreira novi nalog. Na slici 6.1 prikazana je stranica za logovanje.



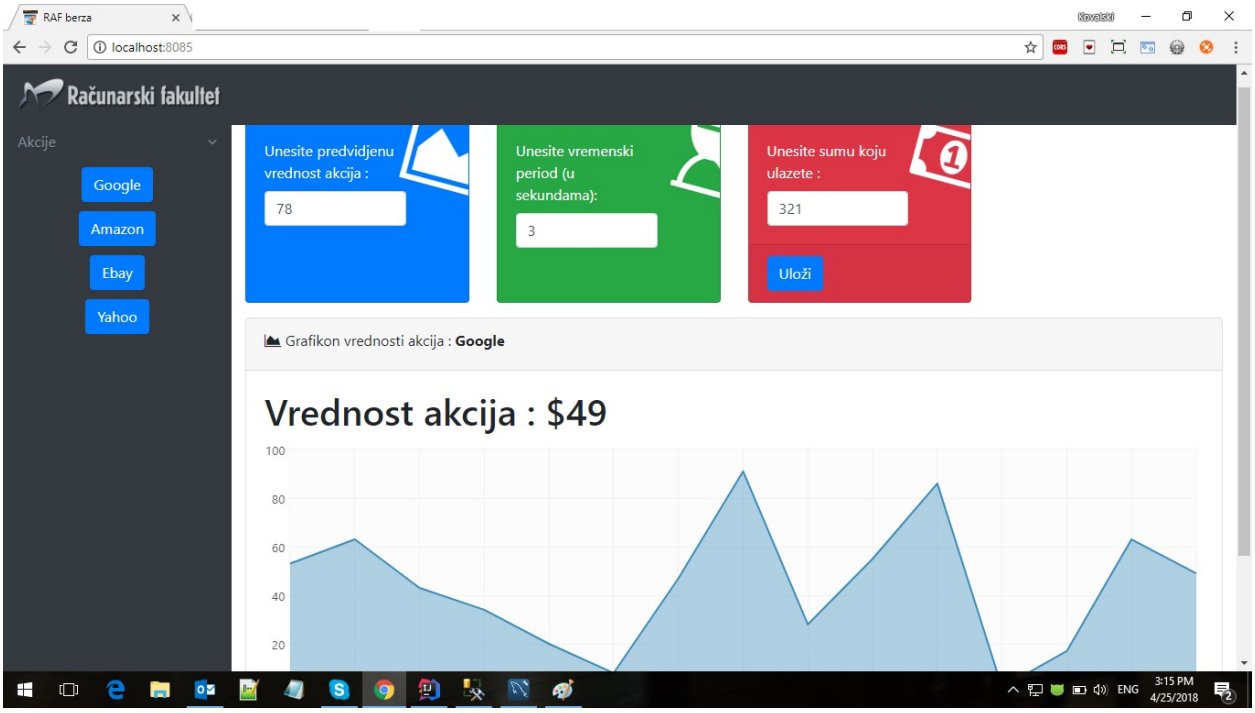
Slika 6.1 Ekran za logovanje u aplikaciju

Proces kreiranja novog naloga podrazumeva popunjavanje forme koja sadrži :

- Korisničko ime,
- Korisničko prezime,
- E-mail korisnika,
- Potvrdu unosa e-mail-a,
- Sumu novca sa kojom korisnik raspolaže,
- Lozinku i
- Potvrdu unosa lozinke.

Validacija unosa za gore navedena polja vrši se na serverskoj strani, u okviru klase *UserRegistrationDto* kao što je opisano u poglavlju 2.4.2.

Pošto je korisnik uspešno logovan na sistem, prikazuje se glavni ekran sa izborom kompanija čije su akcije dostupne na berzi, kao što je prikazano na slici 6.2.



Slika 6.2 Glavni ekran aplikacije

Izborom neke od kompanija sa liste ponuđenih, kreira se real-time grafikon sa cenama akcija. Korisnik je u mogućnosti da unese predviđeni iznos akcija i vreme za koje predviđa da će te akcije dostići predviđeni iznos, kao i sumu novca koji ulaže. Ukoliko je nakon isteka unesenog vremena cena akcija manja ili jednaka unesenoj ceni, uložena suma se povećava za 90% a u suprotnom se smanjuje za isti taj procenat. U realnim uslovima, kreirani korisnički nalog bi bio povezan sa korisnikovom platnom karticom, tako da bi se transakcije sredstava odvijale preko nekog od platnih sistema (kao što je PayPal), međutim za potrebe prototipske aplikacije nije se ulazilo u detaljniju implementaciju takve funkcionalnosti.

Nakon završenog rada, korisnik je u mogućnosti da se odjavi sa sistema i vrati na login stranicu.

7. Zaključak

U ovom radu dat je kratak uvod u reaktivnu programsku paradigmu, korišćenjem *Spring WebFlux* tehnologije za razvoj veb aplikacije za pregled stanja na berzi. Kao i sve ostale tehnologije, *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, 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, proširiva i skalabilna.

Literatura

1. Karanam, R. R. (June 2017). *Mastering Spring 5.0.*

2. *Tiobe ranking official website*, https://www.tiobe.com/tiobe-index/,

3. Johnson, R. *Spring Framework Reference Documentation*, https://docs.spring.io/spring-framework/docs/current/spring-frameworkreference/html/index.html

4. *Mongo database reference documentation,* https://docs.mongodb.com/,

5. Hibernate reference documentation, http://hibernate.org/,

6. *Bootstrap reference documentation,* http://getbootstrap.com/docs/4.1/getting-started/introduction/,

7. *Jquery reference documentation*, https://api.jquery.com/,

8. *MySql reference documentation*, https://dev.mysql.com/doc/,

9. *Thymeleaf reference documentation*, https://www.thymeleaf.org/documentation.html,

10. *Maven wikipadia page*, https://en.wikipedia.org/wiki/Apache_Maven,

11. Walls, C. (November 2014). *Spring in Action, 4th Edition*.

12. *Jackson tutorial,* https://www.tutorialspoint.com/jackson,

13. Nurkiewicz, T. and B. Christensen. (October 2016). *Reactive Programming with RxJava: Creating Asynchronous, Event-Based Applications 1st Edition.*