Migracija web aplikacije na novije tehnologije s reaktivnim i asinkronim izvršavanjem

Hrnčić, Zoran

Master's thesis / Diplomski rad

2019

Degree Grantor / Ustanova koja je dodijelila akademski / stručni stupanj: **University of Zagreb, Faculty of Organization and Informatics / Sveučilište u Zagrebu, Fakultet organizacije i informatike**

Permanent link / Trajna poveznica: https://urn.nsk.hr/urn:nbn:hr:211:326720

SVEUČILIŠTE U ZAGREBU

FAKULTET ORGANIZACIJE I INFORMATIKE

V A R A Ž D I N

Zoran Hrnčić

MIGRACIJA WEB APLIKACIJE NA NOVIJE TEHNOLOGIJE S REAKTIVNIM I ASINKRONIM IZVRŠAVANJEM

DIPLOMSKI RAD

Varaždin, 2019.

SVEUČILIŠTE U ZAGREBU

FAKULTET ORGANIZACIJE I INFORMATIKE

V A R A Ž D I N

Zoran Hrnčić

Matični broj: 0016110897

Studij: Informacijsko i programsko inženjerstvo

MIGRACIJA WEB APLIKACIJE NA NOVIJE TEHNOLOGIJE S REAKTIVNIM I ASINKRONIM IZVRŠAVANJEM

DIPLOMSKI RAD

Mentor:

Prof. dr. sc. Dragutin Kermek

Varaždin, rujan 2019.

Zoran Hrnčić

Izjava o izvornosti

Izjavljujem da je moj završni/diplomski rad izvorni rezultat mojeg rada te da se u izradi istoga nisam koristio drugim izvorima osim onima koji su u njemu navedeni. Za izradu rada su korištene etički prikladne i prihvatljive metode i tehnike rada.

Autor potvrdio prihvaćanjem odredbi u sustavu FOI-radovi

_____

Sažetak

U diplomskom radu bit će obrađen kompletan postupak migracije web aplikacije sa starije tehnologije (JSF,Primefaces) na nove tehnologije (Angular 2+, Angular-material, Spring boot-Groovy). Najveći izazov rada je migracija programskog koda koji je napisan u jeziku Java (izvršavanje je sinkrono) u Angular (asinkrono/reaktivno izvršavanje). Za pisanje Angular aplikacije bit će korišten jezik Typescript. Kompletan Angular je baziran na reaktivnom programiranju čiji su glavni koncepti Observables, Observers, and Operators. Bit će objašnjeni principi emitiranja stream-ova (Observables), osluškivanja stream-ova i reagiranje na njih (Observers). Nadalje ovdje dolaze pojmovi kao što su: Subject, Behaviour Subject, Replay Subject, Promise, forkJoin, mergeMap.

Ključne riječi: web, migracija, programski okvir, sinkrono, asinkrono, reaktivno, uzorak dizajna, Java, Angular.

Sadržaj

1. Uvod

Biti web programer ili programer korisničkog sučelja (*eng. frontend developer*) je vrlo dinamičan posao, a pogotovo posljednjih godina. Koju tehnologiju najbolje odabrati za razvoj aplikacije, a da će biti što duže aktualna i podržana od svih web-preglednika? Svi programeri koji razvijaju web aplikacije, siguran sam da će reći da je ovo vrlo nezahvalan posao. Određena tehnologija se pokaže odličnim izborom i određeno poduzeće (veliki enterprise sustav) odluči da će se nadalje sve aplikacije razvijati u odabranoj tehnologiji jer je ona „kao najbolja“, a uz to je cilj da se sve postojeće aplikacije migriraju u najnoviju tehnologiju. Tako svi web programeri počinju učiti novu tehnologiju i u njoj razvijati aplikacije. Nakon nekoliko godina, kad programeri konačno usavrše tehnologiju, ova ista tehnologija je već postala zastarjela, a u velikim sustavima još nisu uspjeli migrirati sve postojeće aplikacije. Kako izlaze nove biblioteke i razvojni okviri, cilj svih velikih organizacija (npr. u ovom slučaju banke) je da koriste najnovije tehnologije za razvoj web dijela aplikacija jer će tako biti sigurni da će se aplikacija pravilno prikazivati u web pregledniku i da će se ispravno i najbrže izvršavati. Siguran sam da se svaki web-programer, koji radi u velikom sustavu više od 15 godina, tijekom svoje karijere više puta susreo s migracijom iste aplikacije kroz više web tehnologija.

Migracija web-aplikacija nije nimalo beznačajan, jednostavan i brz posao. To je posao koji oduzima puno vremena i tehnološkog znanja programera. Preduvjet svake uspješne migracije je da programer vrlo napredno poznaje obje tehnologije razvoja; staru i novu. Pod pojmom „migracija web-aplikacije“ podrazumijevamo prelazak sa starije tehnologije razvoja prezentacijskog sloja aplikacija na noviju tehnologiju. Sloj poslovne logike, odnosno sloj pristupa podatcima, se ne mijenja jer se ne radi nikakvo unapređenje ili promjena u aplikaciji. U sklopu ovog rada bit će obrađena migracija prezentacijskog djela postojeće aplikacije koja se već dugo koristi u određenom bankarskom sustavu.

U informacijskom sustavu velikih banaka računalni se programi koriste od najranijih dana. Prvi programi koji su se razvijali su bili oni koji su ključni za poslovanje i izvještavanje, a razvijali su se u tehnologijama koje su svojevremeno bile aktualne. Još uvijek se nerijetko glavne transakcije takvih sustava izvršavaju u prastarom Cobol-u. Poslovna logika tih transakcija je vrlo složena. Kako te transakcije rade vrlo točno, s minimalnim ili nikakvim greškama, a ujedno i vrlo brzo, određene stvari se nikada nisu migrirale, nego su se samo nadograđivali aplikacijskim slojevima (CICS, SOAP, REST) kako bi ih nove tehnologije mogle koristiti.

2. Sinkrono izvršavanje u usporedbi s asinkronim izvršavanjem

U sljedećem poglavlju bit će objašnjeno što je to sinkrono te asinkrono izvršavanje programskog koda. Bit će objašnjeno koje su osobine pojedinog načina izvršavanja te u kojim situacijama se pojedino izvršavanje koristi. Potom će biti prikazan primjer programske logike izveden na oba načina te kako se određeni način manifestira kod izvršavanja aplikacije.

2.1. Osobine web aplikacija sa sinkronim izvršavanjem

Synchronous ili Synchronized bi u prijevodu na neki način značilo „povezan“ ili „ovisan“. Drugim riječima, dva sinkrona zadatka moraju biti svjesni jedan drugoga. Jedan zadatak mora biti izvršen tako da je ovisan o drugom, kao naprimjer da on čeka da krene u izvršavanje tek kad se drugi zadatak dovrši.

Općenito kod sinkronog izvršavanja određenog posla, ključna je stvar u tome da kad se određeni posao počinje odrađivati, potrebno je pričekati da se započeti posao dovrši i tek onda se može početi odrađivati sljedeći posao. To bi mogli jednostavnije pojasniti kao izvršavanja niza poslova u seriji. Kao da primjerice postoji određeni proces koji se sastoji od mnogo poslova koje je potrebno odraditi i u jednom trenutku se odrađuje samo jedan posao. Kada trenutni posao završi, tek onda se počinje odrađivati sljedeći posao.

To se u kontekstu računala izražava kao izvršavanje procesa ili zadatka na drugoj dretvi (*eng. thread*). Dretva je niz naredbi (blok koda) koji postoji kao „jedinica posla“ (*eng. unit of work*). Dretve predstavljaju oblik paralelizacije na razini procesa. Operacijski sustav može upravljati s više dretvi tako da određenoj dretvi dodjeli dio procesorskog vremena, a nakon toga oduzme joj procesorsko vrijeme i dodjeli ga drugoj dretvi da odradi neki posao. Jezgra procesora može u jednom trenutku izvršavati samo jednu naredbu i nema koncepta da odrađuje dvije stvari odjednom. Operacijski sustav to simulira na način da dodjeljuje procesorsko vrijeme različitim dretvama. (Wu, 2016)

U današnje vrijeme kada se koriste višejezgreni procesori, poslovi se zapravo mogu odrađivati u isto vrijeme. Operacijski sustav može dodijeliti vrijeme jednoj dretvi na prvoj jezgri, a zatim isti blok vremena drugoj dretvi na drugoj jezgri.

Na sljedećoj slici (Slika 1) prikazano je kako izgleda sinkrono izvršavanje poslova na jednoj dretvi. U ovom primjeru može se zamisliti da postoje tri bloka programskog koda koje je potrebno izvršiti; blok A, blok B i blok C. Operacijski sustav će programski kod na jednoj jezgri izvršavati na takav način da će najprije izvršiti naredbe iz bloka A, a nakon što završi izvršavanje svih naredbi iz bloka A krenut će u izvršavanje naredbi iz bloka B i tako redom.

Slika 1: Sinkrono izvršavanje – jedna dretva (Wu, 2016)

Na slici 2 je slikovno prikazano sinkrono izvršavanje blokova naredbi na više dretvi. U ovom slučaju postoje tri bloka naredbe; blok A, blok B i blok C. Kako je ovo sinkrono izvršavanje, u jednom trenutku vremena se izvršava samo jedan blok naredbi na jednoj od tri dretve. Tek nakon što završi izvršavanje bloka A na dretvi A, započinje izvršavanja bloka B na drugoj dretvi. Blok B se izvršava na drugoj dretvi, ali još uvijek je izvršavanje sinkrono jer se čekalo da se prethodno završi izvršavanje naredbi iz bloka A.

Slika 2. Sinkrono izvršavanje na više dretvi (Wu, 2016)

2.2. Osobine web aplikacija s reaktivnim i asinkronim izvršavanjem

Razlika između sinkronog i asinkronog izvršavanja može se na prvu činiti pomalo zbunjujuća. Izvođenje programa na većini jezika više razine (*eng. high-level language*) je obično vrlo jednosmjerno (*eng. straightforward*). Izvršavanje počinje na prvoj liniji programskog koda i svaki redak koda izvršava se uzastopno.

Izvođenje sinkronog programa je nešto slično prethodno opisanom. Program se izvršava liniju po liniju, odnosno jednu liniju u jednom trenutku. Svaki put kad se funkcija pozove, izvršavanje koda zastane i čeka da funkcija vrati rezultat i tek onda nastavlja izvršavanje sljedeće linije.

Ova metoda izvršavanja može imati neželjene posljedice. Može se pretpostaviti da funkcija koja se poziva započinje proces koji troši puno vremena. Što u situaciji kada se želi zaustaviti dugotrajan proces? Uz sinkrono izvršavanje program je „zaglavljen“ (*eng. „stuck“*) čekajući da proces završi, bez načina za izlaz.

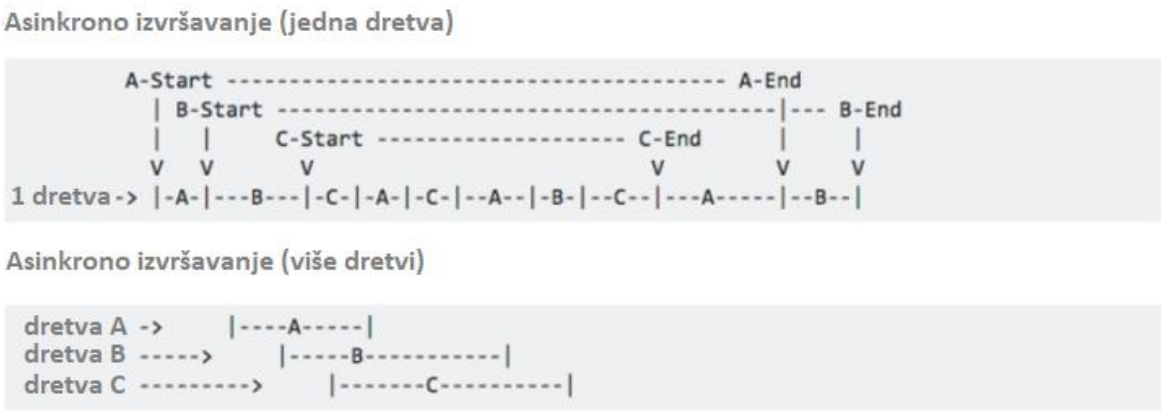
Asinkrono izvršavanje izbjegava ovo usko grlo. Programer je svjestan da će poziv ove funkcije dugo trajati, ali ne želi da program za to vrijeme stoji i čeka da se izvršavanje dovrši. (Software Bisque, bez dat.)

```
// vrijeme u sekundama
fotografiraj(long vrijeme){
    Pocetak

    Do
      Sve potrebne korake za fotografiranje...

    While (korisnikNijeOdustao)

    if (korisnikNijeOdustao == TRUE)
        return NemaGreske;
    else
        return Greska;
    Kraj

    Main
    Pocetak
      fotografiraj(120);
      Print("Funkcija fotografiraj() vraca podatke")
    Kraj
}
```

U gore navedenom pseudo-kodu, koristeći sinkrono izvršavanje mora se pričekati dvije minute da poziv funkcije *TakiePictures()* vrati odgovor i tek onda će prikazati poruku „TakePictures() function returns!“. Nema načina kako bi se fotografiranje otkazalo.

Koristeći asinkrono izvršavanje poziv funkcije *TakePictures()* odmah vraća odgovor i prikazuje se poruka. Iako proces od dvije minute nije dovršen, program može nastaviti s izvršavanjem. Tako program može postaviti varijablu *korisnikNijeOdustao* na FALSE kako bi poništio fotografiranje.

Svaki asinkroni model omogućuje da se više stvari događa istovremeno. Kada se pokrene izvršavanje određenog posla, ostatak programa se nastavlja dalje izvršavati. Kada se akcija završi, program se „obavještava“ i dobiva pristup rezultatu (npr. čitanje podataka s diska).



Slika 3: Asinkrono izvršavanje (Wu, 2016)

U oba gornja primjera (Slika 3) prikazani su primjeri asinkronog izvršavanja. Svaki zaseban zadatak se može izvoditi asinkrono. Zadatci ne moraju biti na posebnim dretvama. Čak i računalo s jednom jezgrom (CPU) i jednom dretvom za izvršavanje može biti programirano tako da pokrene izvršavanje drugog zadatka prije nego završi izvršavanje prethodnog zadatka. Jedini kriterij za takvo izvršavanje je da rezultati iz prvog zadatka nisu potrebni kao ulazi za rješavanje sljedećeg zadatka. Sve dok se početno i završno vrijeme zadatka preklapaju (moguće je samo ako izlaz nijednog zadatka nije potreban kao ulaz drugog), izvršavanje se odvija asinkrono, neovisno koliko je dretvi u upotrebi. (Clymer, 2013)

2.3. Usporedba izvršavanja sinkronog i asinkronog izvođenja koda

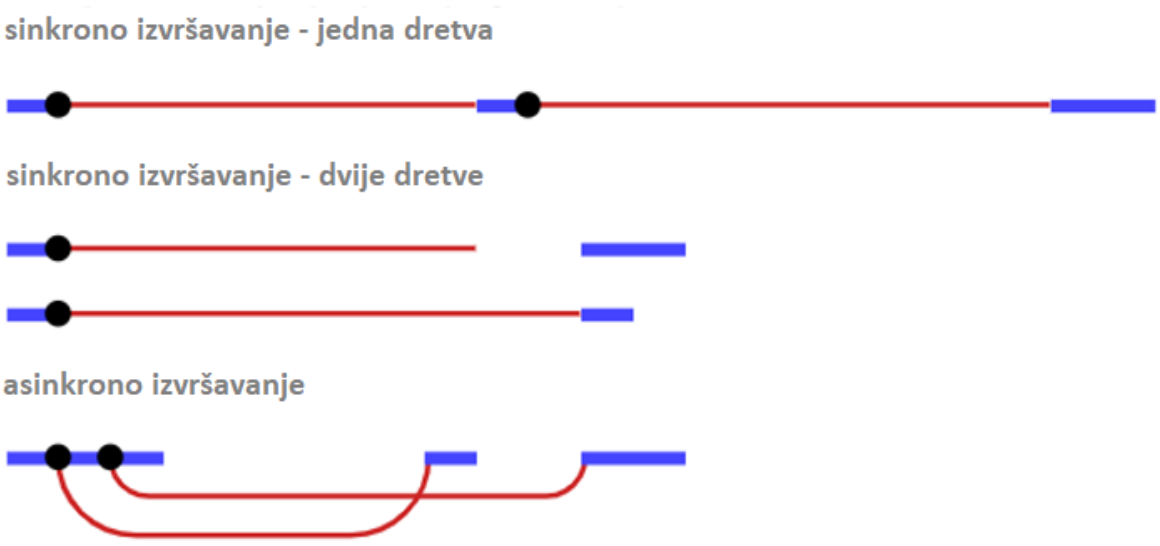
Usporedba sinkronog i asinkronog programiranja može se prikazati pomoću jednog malog programa:

- Program koji dohvaća dva resursa iz mreže i zatim kombinira rezultate

Kod sinkronog programiranja gdje funkcija (zahtjev za mrežnim resursom) vraća odgovor tek kad završi, najlakši način za izvršenje ovog zadatka je da se zahtjevi izvršavaju jedan za drugim. Nedostatak je da će drugi zahtjev biti pokrenut tek nakon završetka prvog. Ukupno vrijeme dohvata oba podatka je minimalno zbroj dvaju vremena odgovora (*eng. response time*).

Rješenje ovog problema kod sinkronog programiranja je pokretanje dodatne dretve. Dretva je još jedan pokrenut program. Budući da većina modernih računala sadrži više procesora, istovremeno se može izvršavati više dretvi na različitim procesorima. Druga dretva može pokrenuti drugi zahtjev, a zatim obje dretve čekaju da se vrati rezultat. Nakon dohvaćenih rezultata dretve se ponovno moraju sinkronizirati kako bi kombinirale svoje rezultate.

U sljedećem dijagramu deblje linije predstavljaju vrijeme koje program provodi u normalnom radu, a tanke linije predstavljaju vrijeme koje je čekao na odgovor mreže. U sinkronom modelu vrijeme koje zauzima mreža je dio vremenske linije određene dretve, a u asinkronom modelu pokretanje mrežne akcije koncepcijski uzrokuje podjelu vremenske linije. Program koji je pokrenuo akciju nastavlja se izvoditi, a akcija se događa „u pozadini“ i na kraju obavještava program o završetku.

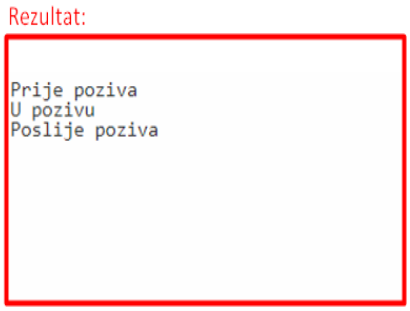


Slika 4. Sinkroni i asinkroni poziv – trajanje (Wu, 2016)

Ovdje će se na jednom malom isječku prikazati redoslijed izvođenja koda prilikom sinkronog izvršavanja, odnosno prilikom asinkronog.

Na sljedećoj slici može se vidjeti isječak koda koji se izvršava sinkrono te njegov rezultat. Prilikom poziva funkcije radi(), čeka se da funkcija završi i tek onda se nastavlja izvršavanje glavnog programa.

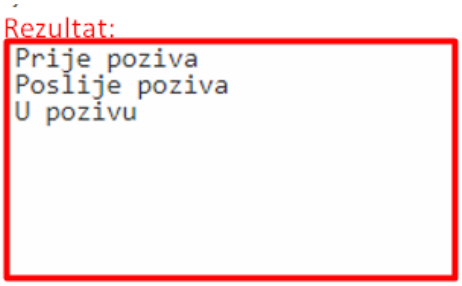
```
public static void Main(){
    Console.WriteLine("Prije
poziva");
    radi();
    Console.WriteLine("Poslije
poziva");
}

public static void radi(){
    Console.WriteLine("U
poziva");
}
```



Slika 5: Rezultat sinkronog izvršavanja

Ispod je prikazan kod koji poziva funkciju koja se izvršava asinkrono, i ovdje se može vidjeti da glavni program ne čeka završetak funkcije, već nastavlja s izvođenjem.

```
public static void Main(){
    Console.WriteLine("Prije
poziva");
    radi();
    Console.WriteLine("Poslije
poziva");
}

public static void radi(){
    Task.Delay(0)
      .ContinueWith(t =>
Console.WriteLine("U poziva"));
}
```



Slika 6: Rezultat asinkronog izvršavanja

2.4. Zašto koristiti (ne koristiti) asinkrono programiranje

Zašto koristiti (*eng. Implement*) asinkrono programiranje? Jedno od shvaćanja programera je da asinkrono programiranje navodno poboljšava performanse. Pod time se često smatra da to znači „moj kod će se brže izvoditi“. Međutim, ovo je u potpunosti netočno. Asinkrono programiranje neće ubrzati izvršavanje programskog koda.

Ono što asinkrono programiranje doista čini je povećanje količine zahtjeva koji se mogu obraditi u isto vrijeme s istim resursima. Ista količina dretvi može obraditi mnogo više istovremenih zahtjeva u asinkronom sustavu nego u sinkronom. Ukratko rečeno, asinkrono programiranje ne poboljšava performance, nego poboljšava propusnost (*eng. throughput*). (Jones, 2019)

Uzme li se za primjer kod gdje se mora napraviti deset tisuća zahtjeva da bi se dohvatio neki resurs, rezultat asinkronog programiranja doista izgleda kao da radi brže, ali u stvarnosti sustav samo radi više odjednom te omogućuje veću propusnost.

Najbolji primjer za dobro korištenje asinkronog programiranja je AJAX (*eng. asynchronous JavaScript and XML*). U JavaScriptu, kada se u petlji izršavaju zahtjevi koristeći AJAX, oni ne čekaju međusobno jedan drugog niti blokiraju proces. Preglednik ne želi utjecati na korisničko iskustvo sa zamrznutom web-lokacijom za vrijeme dok se ne izvrše svi zahtjevi. Svi zahtjevi se šalju gotovo istodobno bez čekanja da prethodni zahtjev završi (vrati odgovor). Kako dolaze odgovori, tako se podatci prikazuju na ekranu neovisno o redoslijedu pozivanja.

Postoji pravilo koje govori da se ne radi asinkrono ažuriranje (obrada) podataka koji su međusobno ovisni/povezani jer bi to bi bila vrlo loša ideja. Na taj način se vrlo brzo može dogoditi da podatci na ekranu više nisu međusobno sinkronizirani. (Stringfellow, 2017)

Programeri znaju često koristiti asinkrono programiranje kako bi izveli neke jednostavne i osnovne izračune. Ovdje jednostavno ne postoji stvarna korist korištenja asinkronog izvršavanja. Brzina izvršavanja je jednaka kao i kod korištenja sinkronog programiranja. (Stringfellow, 2017)

Nije dobro koristiti asinkrono programiranje ako servis poziva jednu bazu podataka koja ne pamti veze ili nije skalabilna. U ovom slučaju baza postaje usko grlo jer je odjednom zapunjena velikom količinom zahtjeva. (Sikder, 2018)

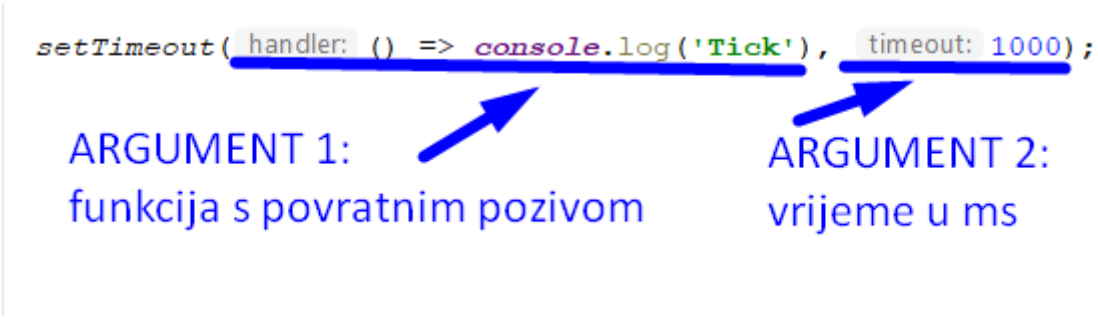
3. Koncepti reaktivnog programiranja i uzorci dizajna

U sljedećem poglavlju bit će obrađeno nekoliko ključnih koncepata koje je potrebno razumjeti kako bi se moglo lakše pristupiti asinkronom programiranju web-aplikacija.

Kreće se od upoznavanja funkcija s povratnim pozivom (*eng. callback function*), te će se prijeći na naprednije gotove uzorke: Promise, Observables.

Jedan od pristupa asinkronog programiranja je kreirati funkciju koja za obavljanje dugotrajnih/sporih aktivnosti prima jedan dodatan argument, funkciju s povratnim pozivom. Akcija se pokreće, a kada ona završi poziva se funkcija s povratnim pozivom i prosljeđuje joj se rezultat obrade.

Primjerice, funkcija *setTimeout* dostupna unutar Node.js-a i u preglednicima. Kao prvi argument prima funkciju s povratnim pozivom, a kao drugi argument prima vrijeme u milisekundama. Funkcija čeka određen broj milisekundi i nakon što istekne vrijeme poziva funkciju s povratnim pozivom. U ovom primjeru u konzolu ispiše „*Tick*“.



Slika 7: Funkcija s povratnim pozivom kao argument

Izvođenje višestrukih asinkronih akcija u slijedu pomoću funkcija s povratnim pozivom znači da se mora nastaviti prosljeđivati nove funkcije kako bi se moglo dalje upravljati nastavkom obrade podataka nakon obavljenih akcija.

```
prvaFunkcija(args, function () {
  drugaFunkcija(args, function () {
    trecaFunkcija(args, function () {
      // i tako dalje...
    });
  });
});
```

Ovakav stil programiranja je djelotvoran, ali se razina uvlačenja/ugnježđivanja (*eng. nested functions*) povećava sa svakom novom asinkronom radnjom jer uvijek izvođenje programa završava u drugoj/prosljeđenoj funkciji s povratnim pozivom. Izvođenje složenijih stvari kao što je istodobno pokretanje više radnji može biti pomalo zbunjujuće i teško za upravljanje.

Problem kod asinkronosti je taj što je ona na neki način „zarazna“. Svaka funkcija koja poziva funkciju koja radi asinkrono mora i sama biti asinkrona. Funkcija mora koristiti funkciju s povratnim pozivom ili slični mehanizam za isporuku rezultata. Također, mora se razmišljati i o nastanku mogućih grešaka za vrijeme izvođenja programa, pa je često potrebno koristiti dodatnu funkciju s povratnim pozivom za upravljanje greškama (*eng. error handling*). Bez nekog dobrog mehanizma (uzorka dizajna), ovakvo programiranje može biti abnormalno komplicirano i složeno. (Meyghani, 2018) (Haverbeke, 2018)

3.1. Promises

U JavaScriptu verzije ES6 uveden je jedan super koristan mehanizam koji olakšava asinkrono web programiranje i rad koristeći funkcije s povratnim pozivom. Mehanizam se zove **Promise**.

Promise je rezervirano mjesto (*eng. placeholder*) za buduću vrijednost. Služi istoj svrsi kao i funkcije s povratnim pozivom, ali ima mnogo ljepšu sintaksu i olakšava upravljanje greškama. To je asinkrona aktivnost koja u nekom trenutku može završiti i producirati/vratiti vrijednost. U mogućnosti je obavijestiti svakoga tko je zainteresiran kada je njegova vrijednost dostupna. (Parker, 2015)

3.1.1. Kreiranje Promise-a

Kreiranje instance promise se izvodi jednostavno izvršavanje operatora ***new*** na ***Promise*** klasi.

```
var promise = new Promise((resolve, reject) => {
});
```

Prilikom kreiranja prosljeđuje se unutarnja funkcija (moglo bi se reći funkcija s povratnim pozivom) koja prima dva argumenta *(resolve, reject)*. Budući da programer sam definira funkciju, ovi argumenti mogu biti nazvani proizvoljno, međutim konvencija je da se nazovu ***resolve*** i ***reject***. Argumenti *resolve* i *reject* su zapravo funkcije.

Unutar „tijela“ promise-a obavlja se asinkrona obrada i kada ista završi jednostavno se pozove funkcija *resolve()*. Ako dođe do bilo kakve greške prilikom obavljanja asinkronog posla tada se pozove *reject()*.

```
function radiAsinkroniPosao() {
  var promise = new Promise((resolve, reject) => {
    setTimeout(() => {
      console.log("Asinkroni posao je uspješno završen.");
      if (error) {
        // vraća string poruke greške
        reject('Došlo je do pogreške');
      } else {
        // vraća string da je posao uspješno završen
        resolve('Završeno');
      }
    }, 1000);
  });
  return promise;
}
```

3.1.2. Obavijesti iz Promise-a

Postoji mogućnost „čekanja“ obavijesti ili povratne vrijednosti kada *Promise* pozove funkciju *resolve()* tako da se na instanci klase tipa *Promise* pozove metoda *.then(arg1, arg2)* i kao prvi argument doda rukovatelj u slučaju uspješnog odgovora iz Promise-a (*eng. success handler*), odnosno funkcija koja će se izvršiti kada *Promise* uspješno obavi posao i vrati vrijednost. Kao drugi argument može se (ali ne mora) dodati rukovatelj pogreškama (*eng. error handler*), odnosno funkcija koja će se izvršiti u slučaju ako *Promise* pozove funkciju *reject()*, tj. ako dođe do kakve pogreške prilikom izvođenja asinkrone obrade. U nastavku primjer s korištenjem prethodno definirane metode:

```
radiAsinkroniPosao ().then(
  (val) => console.log(val), // 'Završeno'
  (err) => console.error(err) // 'Došlo je do pogreške'
);
```

3.1.3. Trenutno izvršavanje Promise-a (Resolution ili Rejection)

Može se kreirati objekt *Promise* koji odmah vrati rezultat koristeći metodu *Promise.resolve(),* odnosno može se odmah vratiti grešku na način *Promise.reject()*. Jedna od lijepih stvari prilikom korištenja *Promise-a* je to da ako se doda *.then()* nakon što promise pozove metodu *resolve()* ili *reject()*, određeni rukovatelj će svejedno biti pozvan i kod će se ispravno i očekivano izvršiti.

```
let promisePozoveResolve = Promise.resolve('Uspješno završeno');
let promisePozoveReject = Promise.reject('Dogodila se pogreška!!!');

promisePozoveResolve.then(
     (val) => console.log(val) // 'Uspješno završeno'
);

promisePozoveReject.then(
     (val) => console.log(val),
     (err) => console.error(err) // 'Dogodila se pogreška!!!'
);
```

U primjeru iznad, iako je *Promise* izvršio pozive metode *Resolve()* ili *Reject()* prije nego mu je pridružen rukovatelj za uspjeh ili grešku, *Promise* mehanizam svejedno poziva odgovarajuću funkciju, rukovatelja.

3.1.4. Ulančavanje (*eng. Chaining*)

Može se povezivati niz *.then()* rukovatelja zajedno u lanac. Ovdje je vrlo korisna stvar to što *Promise* prosljeđuju greške niz lanca tako dugo dok dođe do rukovatelja greškama ako postoji. Iz tog razloga nije potrebno kreirati rukovatelj greškama za svaku *.then()* funkciju već se može dodati samo jedan na kraju lanca.

```
Promise.reject('Dogodila se pogreška!!!')
  .then((val) => console.log(val)) // Ne ispisuje ništa
  .then(
    (val) => console.log(val),
    (err) => console.error(err) // 'Dogodila se pogreška!!'
  );
```

Ako se generira iznimka (*eng. throw an exception*) unutar *Promise* funkcije ili unutar jednog od rukovatelja uspjeha, rukovatelj greškama je odmah pozvan. Ovdje također postoji i funkcija ***.catch()*** koja radi na isti način kao i rukovatelj greškama unutar *.then()*. Ova funkcija samo jasnije i eksplicitnije opisuje namjeru da se obradi pogreška i to na jednom mjestu.

```
Promise.resolve('Uspješno završeno')
  .then((val) => {throw new Error("Dogodila se pogreška!!!")})
  .then((val) => console.log(val)) // Ne ispisuje ništa
  .catch((err) => console.error(err)); // 'Dogodila se pogreška!!'
```

Promise-i su, za razliku od funkcija s povratnim pozivom, mnogo čišće rješenje za pisanje asinkronog koda. Napisani kod je lakše čitljiv i često se piše redoslijedom kako će se aplikacija izvršavati. Tako je programeru lakše u glavi pratiti izvođenje koda. (Parker, 2015)

3.2. Observables, Observers

Observables je uzorak dizajna koji daje mogućnost za prijenos poruka između oglašivača i pretplatnika (eng. publisher and subscriber) unutar aplikacije. Observables nudi značajne prednosti u odnosu na ostale tehnike za upravljanje događajima, asinkrono programiranje i upravljanje višestrukim vrijednostima.

Observables su deklarativni, a to znači da se definira funkcija za objavljivanje (*eng. publishing*) vrijednosti, ali se ta funkcija ne izvrši tako dugo dok se korisnik na nju ne pretplati. Pretplaćeni korisnik zatim prima obavijesti sve dok funkcija ne završi ili se korisnik odjavi (eng. unsubscribe).

Observable može isporučivati višestruke vrijednosti bilo kojeg tipa – literale, poruke, događaje. (Fain, Moiseev, 2019)

3.2.1. Osnovna upotreba i uvjeti

Kao oglašivač potrebno je definirati instancu tipa *Observable* koja definira pretplatničku funkciju (*eng. subscriber function*). To je funkcija koja će biti izvršena kada potrošač na ovoj instanci pozove metodu *.subscribe().* Pretplatnička funkcija definira kako dohvatiti ili generirati vrijednosti koje će biti objavljene.

Da bi se izvršio observable koji je prethodno kreiran te bi se započelo primanje obavijesti /vrijednosti, potrebno je na njemu pozvati metodu *.subscribe()* u koju se prosljeđuje *observer*. To je JavaScript objekt koji definira rukovatelje za primljene obavijesti. Poziv metode *.subscribe()* će vratiti objekt tipa *Subscription* koji ima metodu *.unsubscribe()*. Pozivom metode *.unsubscribe()* potrošač prestaje primati obavijesti.

U primjeru se može vidjeti način kreiranja proizvođača (Observable) koji će emitirati poruku nakon jedne sekunde, te se nakon toga vidi gdje se potrošač (Observer) pretplati na proizvođača. Potrošač ispisuje svaku poruku koju emitira proizvođač.

```
const observable = new Observable(observer => {
  setTimeout(() => observer.next('pozdrav iz proizvođača!'), 1000);
});
observable.subscribe(v => {
        console.log(v); // 'pozdrav iz proizvođača!'
 });
```

U ovom primjeru se može primjetiti da je observer objekt dostupan samo unutar konstruktora proizvođača (Observable). Izvan implementacije proizvođača ne može se

pristupiti Observeru i pozvati metodu *.next().* Ovakav doseg (*eng. scope*) osigurava da samo proizvođač zna kako i kada je potrebno emitirati događaje za pretplatnike koji osluškuju.

Observables su po zadanom stanju (*eng. default*) „*Cold*“, to znači da su *lazy* i da neće izvršiti nikakav kod sve do trenutka kad će se netko na njih pretplatiti, *subscribe()*. Također po zadanom ponašanju ne dijele svoj rad između više pretplatnika. Na primjeru gornjeg koda, ako se tri pretplatnika pretplate na Observable, svaki put će se ponovno kreirati *setTimeout()* funkcija i svaki će zasebno (samo jednom svojem pretplatniku) emitirati vrijednost. (Koutnik, 2018)

3.3. Subject

Subject je jedna vrsta tipa Observable dostupna unutar biblioteke RxJS. Subject isto poput Observable može emitirati višestruke vrijednosti. Međutim, Subject omogućuje pretplatnicima određene instance tipa Subject da mogu sami emitirati događaje/vrijednosti. Svi pretplatnici primaju obavijest kada bilo tko emitira vrijednost na dotičnom Subjectu. (Koutnik, 2018)

```
const subject = new Subject();
subject.next('propuštena poruka iz Subject-a');
subject.subscribe(v => console.log("Prvi potrošač: "+ v));
subject.next('pozdrav iz Subject-a!');
subject.subscribe(v => console.error("Drugi potrošač: "+ v ));
```

```
Console was cleared
Prvi potrošač: pozdrav iz Subject-a!
```

Slika 8: Rezultat izvršavanja prethodnog koda

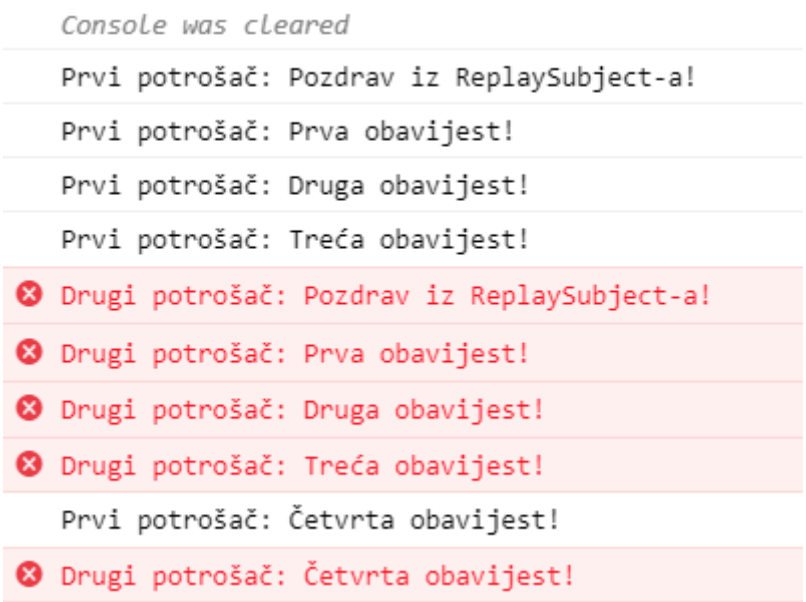
Onome tko se s ovime prvi put susreće, siguran sam da ovo izgleda vrlo zbunjujuće. Vidljivo je kako se najprije poziva *.next()* i emitira poruka 'propuštena poruka iz Subject-a' prije nego postoji bilo koji pretplatnik. Subject je za razliku od normalnog Observable „Hot“, a to znači da on može emitirati događaje prije nego se bilo tko na njih preplati. Stoga pretplatnik može propustiti sve događaje koji su bili emitirani prije nego se on pretplatio.

Subject, za razliku od Observable, dijeli svoj rad sa svim pretplatnicima. Ovdje se sada javlja jedno pitanje; što ako se prekasno korisnik pretplati i želi dobiti prethodno emitirane vrijednosti? Naravno da postoji rješenje, zove se ReplaySubject a to je još jedan tip Observable.

3.4. ReplaySubject

ReplaySubject kao i regularni Subject može pokrenuti emitiranje događaja izvan konstruktora, te je također „Hot“ Observable. Za razliku od regularnog Subjecta, ReplaySubject će ponoviti sve prethodno emitirane događaje ako se potrošač pretplatio nakon što su događaji bili emitirani.

```
const replaySubject = new ReplaySubject();
replaySubject.next('Pozdrav iz ReplaySubject-a!');

replaySubject.next('Prva obavijest!');
replaySubject.next('Druga obavijest!');
replaySubject.subscribe(v => console.log("Prvi potrošač: "+ v ));
replaySubject.next('Treća obavijest!');

replaySubject.subscribe(v => console.error("Drugi potrošač: "+ v ));
replaySubject.next('Četvrta obavijest!');
```

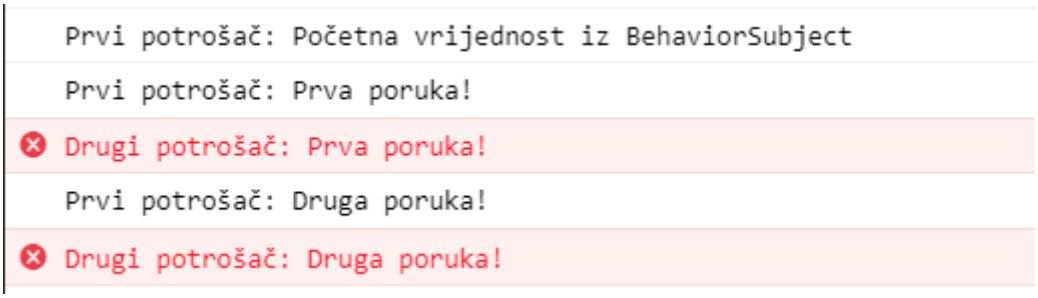


Slika 9: Rezultat izvršavanja prethodnog koda

3.5. BehaviorSubject

BehaviorSubject ima slično ponašanje kao i ReplaySubject. Također je „Hot“ Observable i vraća SAMO posljednju emitiranu vrijednost svim pretplaćenim potrošačima koji su se pretplatili nakon sto je vrijednost bila emitirana. Međutim, BehaviorSubject dodaje još jedan dio funkcionalnosti u kojoj se prilikom inicijalizacije subjecta mora postaviti početna vrijednost.

```
const behaviorSubject = new BehaviorSubject('Početna vrijednost iz
BehaviorSubject');

behaviorSubject.subscribe(v => console.log("Prvi potrošač: "+ v ));

behaviorSubject.next('Prva poruka!');

behaviorSubject.subscribe(v => console.error("Drugi potrošač: "+ v ));
behaviorSubject.next('Druga poruka!');
```



Slika 10: Rezultat izvršavanja prethodnog koda

BehaviorSubject je najčešće korišten subject za primjenu u aplikacijama s reaktivnim stilom programiranja gdje se želi imati jedno centralno stanje/informaciju koja se dijeli diljem programskog koda.

3.6. AsyncSubject

AsyncSubject je najrjeđe korišten subject. On ne emitira nikakve vrijednosti prije svog završetka, odnosno poziva .complete(). Nakon završetka „šalje“ posljednju emitiranu vrijednost svim svojim pretplatnicima.

3.7. RxJS

RxJS je danas jedna od najkorištenijih biblioteka u web razvoju. Nudeći snažan i funkcionalan pristup za rad s različitim događajima koji može biti korišten u različitim razvojnim okvirima i bibliotekama za razvoj korisničkog sučelja. Razloga za učenje Rx koncepata nikad nije bilo više. Jednom kada se nauči Rx, znanje se može iskoristiti kroz gotovo svaki jezik i uz to se ima vrlo napredno razumijevanje reaktivnog programiranja.

Učenje RxJS i reaktivno programiranje je vrlo teško. Postoji mnoštvo koncepata, veliko API sučelje i što je najteže, zahtijeva temeljit pomak u razmišljanju tijekom programiranja. Zahtijeva prelazak s imperativnog na deklarativni stil programiranja/razmišljanja. Svi koji krenu u učenje Rx moraju biti svjesni da će to biti jako teško, ali svakako je vrijedno truda. (Mansilla, 2018)

3.7.1. Operatori

Operatori su glavna pokretačka snaga (*eng. horse-power*) iza Observables-a, pružajući elegantno, deklarativno rješenje za kompleksne asinkrone zadatke.

Operatori se dijeli u nekoliko kategorija:

- Operatori kombinacija (forkJoin, mergeAll, concat, combineLatest)
- Operatori uvjetovanja (defaultEmpty, every, iif, sequenceFrom)
- Operatori stvaranja (from, fromEvent, interval, of, throw, timer)
- Operatori upravljanja pogreškama (catch/catch Error, retry, retryWhen)
- Višesmjerni operatori (*eng. multicasting*) (publish, milticast, share, shareReplay)
- Operatori filtriranja (debounceTime, distinctUntilChanged, filter, take)
- Operatori transformacije (concatMap,map,mergeMap,switchMap)
- Pomoćni korisni operatori (*eng. utility*) (tap/do, delay, finalize/finallly, timeInterval, toPromise)

3.7.1.1. merge

Ovaj operator služi da više observable-a spoji u jedan observable na koji se onda može pretplatiti. Kada bilo koji od spojenih observable-a emitira događaj, taj događaj će biti emitiran kroz jedinstveni observable.

```
// RxJS v6+
import { merge } from 'rxjs/operators';
import { interval } from 'rxjs';

// emitira događaj svakih 2.5 sekundi
const prvi = interval(2500);
// emitira događaj svaku 1 sekundu
const drugi = interval(1000);
const spojeno = prvi.pipe(merge(drugi));
// izlaz: 0,1,0,2....
const subscribe = spojeno.subscribe(val => console.log(val));
```

3.7.1.2. ajax

Kreira observable za Ajax poziv prema određenoj mrežnoj lokaciji. Observable emitira objekt koji je odgovor poziva mrežne lokacije/servisa.

```
// RxJS v6+
import { ajax } from 'rxjs/ajax';

const githubUsers = `https://api.github.com/users?per_page=2`;

const users = ajax(githubUsers);

const subscribe = users.subscribe(
  res => console.log(res),
  err => console.error(err)
);
```

3.7.1.3. from

Operator kreira observable iz liste objekata. Novokreirani observable emitira redom sve elemente liste jednog po jednog.

```
// RxJS v6+
import { from } from 'rxjs';

// emitira listu kao niz vrijednosti
const arraySource = from([1, 2, 3, 4, 5]);
// izlaz: 1,2,3,4,5
const subscribe = arraySource.subscribe(val => console.log(val));
```

3.7.1.4. of

Of radi slično kao i operator From, emitira redom sve vrijednosti koje su mu proslijeđene kroz konstruktor i nakon toga emitira događaj *.complete()*.

```
// RxJS v6+
import { of } from 'rxjs';
// emitira redom svaki prosljeđeni broj
const source = of(1, 2, 3, 4, 5);
// izlaz: 1,2,3,4,5
const subscribe = source.subscribe(val => console.log(val));
```

3.7.1.5. catch/ catchError

Operator koji savršeno upravlja greškama observable sekvencama. Vrlo je bitno da operator vrati Observable iz *catchError* funkcije kako bi pretplaćeni potrošač mogao primiti obavijest o pogrešci.

```
// RxJS v6+
import { throwError, of } from 'rxjs';
import { catchError } from 'rxjs/operators';
// operator koji emitira grešku
const source = throwError('Ovo je greška!');
// hvata grešku, vraća observable sa porukm greške
const example = source.pipe(catchError(val => of(`Uhvatio sam: ${val}`)));
// izlaz: 'Uhvatio sam: Ovo je greška!
const subscribe = example.subscribe(val => console.log(val));
```

3.7.1.6. debounceTime

Operator odbacuje emitirane vrijednosti koje su emitirane prije nego je prošlo definirano vrijeme od prethodne emitirane vrijednosti. Ovaj operator se često koristi u scenarijima gdje se treba kontrolirati brzina unosa podataka korisnika putem tipkovnice.

```
// RxJS v6+
import { fromEvent } from 'rxjs';
import { debounceTime, map } from 'rxjs/operators';

// referenca na html input element
const searchBox = document.getElementById('search');

// fromEvent je operator koji kreira stream koje emitira na „keyup“
const keyup$ = fromEvent(searchBox, 'keyup');

// čeka 0.5s između keyups događaja i onda emitira trenutno vrijednost
keyup$
  .pipe(
    map((i: any) => i.currentTarget.value),
    debounceTime(500)
  )
  .subscribe(console.log);
```

3.7.1.7. filter

Operator koji služi za filtriranje događaja/vrijednosti koje emitira određeni observable. On dalje emitira/propušta samo vrijednosti koje zadovoljavaju definirane uvjete.

```
// RxJS v6+
import { from } from 'rxjs';
import { filter } from 'rxjs/operators';

// emitira redom vrijednosti 1, 2, 3, 4, 5
const source = from([1, 2, 3, 4, 5]);
// filter ne propušta neparne brojeve
const example = source.pipe(filter(num => num % 2 === 0));
// izlaz: "Parni broj: 2", "Parni broj: 4"
const subscribe = example.subscribe(val => console.log(`Parni broj:
${val}`));
```

3.7.1.8. forkJoin

Operator koji kao ulaz prima listu observable-a i tek kada svi observable-i završe (.complete()), onda u obliku liste emitira posljednju emitiranu vrijednosti od svakoga. Najbolja upotreba je kada se želi napraviti više neovisnih paralelnih poziva servisa i tek kada svi završe, onda se može raditi neka akcija s odgovorima. Kod upotrebe treba biti oprezan s upravljanjem greškama. Ako bilo koji observable ne završi uspješno, forkJoin neće nikada emitirati vrijednosti ostalih.

```
// RxJS v6.5+
import { ajax } from 'rxjs/ajax';
import { forkJoin } from 'rxjs';

/*
    Kada svi observable-i završe, vraća
    zadnju emitiranu vrijednost iz svakoga u obliku riječnika (eng. dictionary)
*/
forkJoin(
  {
    google: ajax.getJSON('https://api.github.com/users/google'),
    microsoft: ajax.getJSON('https://api.github.com/users/microsoft'),
    users: ajax.getJSON('https://api.github.com/users')
  }
) .subscribe(console.log);
// { google: object, microsoft: object, users: array }
```

3.7.1.9. switchMap

Po mojem mišljenju jedan od najkorisnijih operatora. Koristi u slučaju kada se žele određene asinkrone stvari raditi u seriji/sekvenci. Kao primjer može se navesti situacija kada je potrebno više puta za redom pozivati web-servis, međutim odgovor iz jednog poziva su podatci za pozivanje drugog web servisa. Uglavnom služi za dohvaćanje iz baze podataka ili nekog drugog izvora. Preporučuje se koristiti za čitanje podataka, nikako nije pogodan za upisivanje. (Troncone, 2019)

```
// RxJS v6+
import { interval, fromEvent } from 'rxjs';
import { switchMap } from 'rxjs/operators';

fromEvent(document, 'click')
  .pipe(
    // resetira brojač na svaki klik mišem
    switchMap(() => interval(1000))
  )
  .subscribe(console.log);
```

4. Programski okviri i biblioteke za reaktivno i asinkrono izvršavanje web aplikacija

4.1. Web 2.0

Web 2.0 je naziv koji se koristi za opisivanje druge generacije World Wide Weba. Ovdje je napravljen pomak s korištenja statičkih web stranica koje su sada postale dinamične i tako omogućuju mnogo interaktivnije i dinamičnije web iskustvo. Web 2.0 je fokusiran na mogućnost suradnje i dijeljenje informacija putem društvenih medija, blogova i web-zajednica. (*eng. Web-based communities*). (Technopedia, bez dat.)

Web 2.0 je signalizirao promjenu u kojoj je svjetska mreža postala interaktivno iskustvo između korisnika i web izdavača, a ne jednosmjerni razgovor kako je to bilo ranije. Ona također predstavlja populističku verziju weba, gdje su novi alati omogućili gotovo svaki doprinos, bez obzira na tehničko znanje. (Technopedia, bez dat.)

Web 2.0 je interaktivni web. Sve ideje o uvođenju moći ljudi izravno u internet ne bi bilo moguće izvesti bez tehnologije koja bi to podržavala. Kako bi kolektivno znanje ljudi bilo iskorišteno, web stranice moraju biti dovoljno jednostavne za korištenje da ne stoje na putu ljudima koji koriste internet da podijele svoje znanje.

S jedne strane (socijalnog aspekta) web 2.0 je ideja o stvaranju društvenog weba, međutim gledajući s tehnološke strane, to je stvaranje više interaktivnijeg i prilagodljivog weba (*eng. responsive web design*). Tako tehnike kao što je AJAX postaju središnje mjesto ideje za Web 2.0. AJAX (*eng. Asynchronous JavaScript And XML*) dopušta web-stranici (web aplikaciji) da u pozadini komunicira sa serverom i to bez korisničke interakcije. To znači da nije potrebno nigdje na stranici kliknuti ili napraviti neku akciju da bi se dogodila nekakva promjena na stranici. Današnje web 2.0 aplikacije/stranice se jednom učitaju u preglednik i stranica se dalje koristi bez osvježavanja stranice. Web stranica u pozadini sa serverom izmjenjuje podatke te generira sadržaj na ekranu. Između web stranice i servera se izmjenjuju čisti sirovi podatci, a ne dijelovi stranice (ne prenosi se HTML kod).

To sve zvuči jednostavno, ali to nije nešto što je bilo moguće u ranijim razdobljima weba. To je značilo da bi web stranice morale biti više dinamičnije i prilagodljivije te sličnije desktop aplikacijama. Tako bi web stranice bile jednostavne za korištenje. Dakle, da bi se iskoristila kolektivna moć ljudi u cilju stvaranja socijalnog weba, stranice moraju biti vrlo jednostavne za korištenje. (Nations, 2019)

4.2. Vue.js

Vue.js je progresivni razvojni okvir (*eng. framework*) za izgradnju korisničkih sučelja. Za razliku od drugih monolitnih razvojnih okvira, Vue je dizajniran od temelja kako bi bio postupno prihvatljiv. Glavna biblioteka fokusirana je samo na sloj prikaza i lako se integrira s drugim bibliotekama ili postojećim projektima. S druge strane, Vue je također sposoban pokretati veće aplikacije na jednoj stranici (*eng. Single-Page application*) kada se koristi u kombinaciji s modernim alatima i podržavajućim bibliotekama.

Vue se naziva progresivni razvojni okvir, što to točno znači? To znači da se prilagođava potrebama programera. Ostali zahtijevaju od programera i cijelog tima da u potpunosti poznaju način rada razvojnog okvira kako bi mogli imalo raditi na razvoju aplikacije i često zahtijevaju da se kompletna aplikacija ponovno napiše. Vue se (na sreću) jednostavno koristi unutar postojeće aplikacije. Doslovno je potrebno samo napraviti jedan prazan tag i taj tag postaje nova Vue aplikacija koja može rasti zajedno sa potrebama projekta. Moguće je iz male aplikacije od 3 linije napraviti veliku aplikaciju za upravljanje cijelim prezentacijskim slojem (*eng. view layer*). Vue pruža puno veću prilagodljivost i to je jedan od glavnih razloga da ga je puno lakše i brže naučiti nego Angular ili React.

Vue aplikacija se može vrlo jednostavno kreirati unutar bilo koje web stranice. U primjeru se vidi kreiranje jednostavne Vue aplikacije koja interpolacijom na ekranu prikazuje poruku „Hello Vue!“. Kako se mijenjaju vrijednosti pojedine varijable, tako se automatski na ekranu mijenja prikaz bez osvježavanja stranice. (Hanchett, Listwon, 2018)

```
<div id="app">
  {{ message }}
</div>
var app = new Vue({
  el: '#app',
  data: {
    message: 'Hello Vue!'
  }
})
```

Rezultat:

```
Hello Vue!
```

4.3. React

React je JavaScript biblioteka za kreiranje korisničkih sučelja razvijena od strane Facebook-a. Prvobitno je objavljena 2013. godine. Facebook koristi React u svim svojim proizvodima (Facebook, Instagram, WhatsApp). Trenutna stabilna verzija je 16.x objavljena u studenom 2018. godine. (Daityary, 2019)

Za one koji kreću s učenjem React-a dovoljan je i jedan sat da se uz pomoć dokumentacije kreira (*eng. set up*) aplikacija. React nije kompletan programski okvir, on je samo biblioteka tako da za naprednije značajke zahtjeva korištenje drugih biblioteka (*eng. third-party libraries*). Iz tog razloga krivulja učenja jezgrenog okvira nije strma.

React omogućuje bezbolno stvaranje interaktivnih korisničkih sučelja. Potrebno je samo dizajnirati jednostavne ekrane/poglede (*eng. views*) za svako stanje u aplikaciji i React će učinkovito ažurirati i prikazivati prave komponente kada se podatci promjene. Deklarativni ekrani čine kod lakšim za održavanje i otklanjanje pogrešaka.

Omogućuje komponentni (*eng. component-based*) način rada. To znači da se prilikom razvoja aplikacije kreiraju enkapsulirane komponente koje upravljaju vlastitim stanjem. Sastavljanjem manjih komponenti može se kreirati veliko složeno korisničko sučelje.

Budući da je logika komponente pisana u JavaScriptu umjesto u predlošku, mogu se jednostavno prenositi složeni podatci kroz aplikaciju i zadržavati stanje izvan DOM-a. React se također može koristiti na serverskoj strani za kreiranje stranica (*eng. rendering*) koristeći NodeJS, te isti kod može pokretati mobilne aplikacije koristeći React Native. (Vaughn, 2019)

Komponente React-a implementiraju metodu *render()* koja prima ulazne podatke i vraća što će se prikazati na ekranu. Ovaj primjer koristi sintaksu nalik XML-u pod nazivom JSX. Ulaznim podatcima koji se prosljeđuju u komponentu može se pristupiti unutar metode *render()* preko *this.props*.

Na sljedećem primjeru prikazan je primjer kreiranja jedne vrlo jednostavne „Hello world“ komponente koristeći React. (Vaughn, 2019)

```
class HelloMessage extends React.Component {
  render() {
    return (
      <div>
        Pozdrav {this.props.ime}
      </div>
    );
  }
}

ReactDOM.render(
  <HelloMessage ime="Pero" />,
  document.getElementById('hello-example')
);
```

Rezultat:

```
Pozdrav Pero
```

4.4. Usporedba biblioteka/programskih okvira

Gledajući tri najpopularnije biblioteke za razvoj frontend aplikacija donosi se zaključak da je svaki od njih odličan izbor ovisno o vrsti i složenosti aplikacije. Vue.js i React su bolji izbor za manje složene aplikacije pošto je arhitektura samih biblioteka jednostavnija te je puno lakše i brže savladati rad s njima. Angular je loš izbor za malene aplikacije zato što je to vrlo kompleksan i robustan programski okvir s vrlo mnogo koncepata i potrebno je puno znanja da bi se mogao kvalitetno koristiti. Angular je namijenjen za velike složene (enterprise) sustave i tu se vidi njegova snaga i prednosti. Slijedi cijelo poglavlje o njemu.

Tablica 1: Usporedba Angular, React, Vue

	Angular	**React**	**Vue**
Prvo izdanje	2010	2013	2014
Službena stranica	angular.io	reactjs.org	vuejs.org
Veličina (KB)	500	100	80
Trenutna verzija	7	16.6.3	2.17
Korištenje	Google, Wix	Facebook, Uber	Alibaba, GitLab

(Izvor: https://www.codeinwp.com, 15.07.2019)

5. Osobine programskog okvira Angular i programskog jezika Typescript

Prije nego se krene s učenjem Angulara i razvijanja aplikacija koristeći ovaj izvanredan razvojni okvir, programeri bi trebali biti svjesni da Angular ima strmu krivulju učenja (*eng. the steep learning curve*). To znači da je učenje vrlo teško i potrebno je uložiti mnogo vremena i truda. Prije upuštanja u učenje ovog razvojnog okvira, Angular zahtijeva određeno predznanje. Podrazumijeva se da se od programera traži temeljno znanje tehnologija za razvoj korisničkog sučelja (*eng. frontend technologies*) kao što su JavaScript, HTML, CSS, te je poželjno razumijevanje MVC (*eng. Model, View and Controller*) uzorka i rada poslužiteljskih servisa (*eng. backend services*). (Pham, 2018)

Angular je moderan razvojni okvir u cijelosti izgrađen u TypeScript-u i kao rezultat korištenje TypeScript-a s Angularom pruža besprijekorno iskustvo. Iz prethodnog je jasno da je potrebno poznavanje jezika TypeScript. Međutim, Angular podržava i JavaScript, ali se preporuča korištenje TypeScript-a i programeri ga uglavnom koriste.

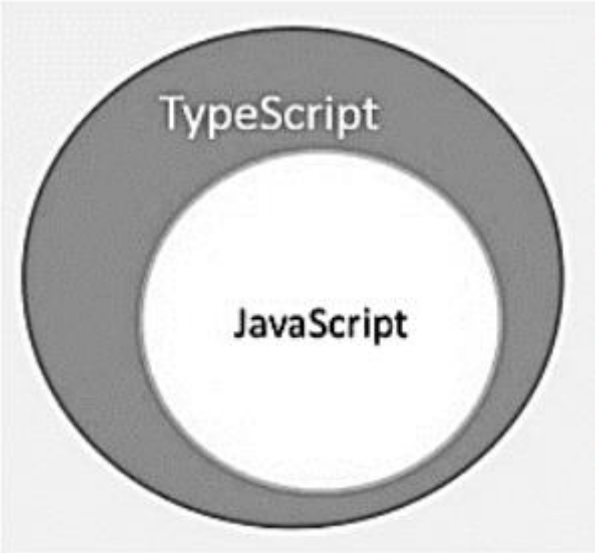
Sljedeća stvar koju je svakako potrebno razumijeti je biblioteka RxJS odnosno koncepti kao što su Observables, Promise, Subject. Kako bi se maksimalno iskoristilo ovu biblioteku potrebno je znanje asinkronog programiranja u JavaScriptu.

Vidljivo je kako Angular traži opsežno predznanje da bi se moglo kvalitetno programirati koristeći ovaj robustan razvojni okvir. U sljedećem poglavlju bit će detaljno objašnjena arhitektura te osnovni koncepti Angulara. (Pham, 2018)

5.1. Typescript

TypeScript programerima omogućuje pisanje JavaScript koda na način kako oni žele. TypeScript je tipiziran nadskup JavaScript-a koji se kompilira u čisti (*eng. plain*) JavaScript kod. TypeScript je također pravi objektno orijentirani jezik s klasama, sučeljima i tipovima podataka kao što je to u C# ili Java-i. Popularni JavaScript programski okvir Angular 2.0 je kompletno napisan u TypeScriptu. Učenje TypeScripta može pomoći programerima da pišu objektno-orijentirane programe i da ih kompiliraju u JavaScript. Može se koristiti na serverskoj i klijentskoj strani.

TypeScript je izrazito tipiziran, objektno orijentiran jezik koji se kompilira u JavaScript. Dizajnirao ga je Anders Hejlsberg (dizajner jezika C#) u Microsoftu. TypeScript je ujedno i jezik i skup alata. Moglo bi se reći da je TypeScript zapravo JavaScript s nekim dodatnim značajkama (*eng. Features*). (Microsoft, bez dat.)



Slika 11:TypeScript i JavaScript

5.1.1. Sintaksa

Sintaksa u svakom programskom jeziku definira skup pravila za pisanje programa. Svaki programski jezik definira vlastitu sintaksu.

Svaki TypeScript program se sastoji od:

- modula, funkcija, varijabli, izraza (*eng. Statements and Expressions*), komentara

TypeScript:

```
let message:string = "Hello World"
console.log(message)
```

Kompilirano u JavaScript:

```
let message = "Hello World";
console.log(message);
```

U sljedećem primjeru prikazano je kako izgleda klasa s jednom metodom napisana u jeziku TypeScript, te isti kod kompiliran u JavaScript.

TypeScript:

```
class HelloWorld {
  public a: number;
  public b: number;
  public c: number;
  constructor() {
    this.a = 2;
    this.b = 3;
  }
  public izracunaj(): void {
    this.c = this.a + this.b;
    console.log('Rezultat:' + this.a + this.b);
  }
}
const obj: HelloWorld = new HelloWorld(); // kreiranje instance
obj.izracunaj(); // pozivanje metode
```

Kompilirano u JavaScript:

```
var HelloWorld = /** @class */ (function () {
  function HelloWorld() {
    this.a = 2;
    this.b = 3;
  }

  HelloWorld.prototype.izracunaj = function () {
    this.c = this.a + this.b;
    console.log('Rezultat:' + this.a + this.b);
  };
  return HelloWorld;
}());
var obj = new HelloWorld(); // kreiranje instance
obj.izracunaj(); // pozivanje metode
```

5.1.2. Ugrađeni tipovi podataka

Tip podataka *any* je super tip svih tipova podataka u jeziku Typescript. Ovaj tip je nešto slično kao klasa *Object* u jeziku Java. Korištenje tipa *any* je jednako kao korištenje klasičnih varijabli u JavaScriptu gdje nema provjere tipa podataka. (Cherny, 2019)

Tablica 2: TypeScript ugrađeni tipovi podataka

Tip	Ključna riječ	Opis
Number	Number	Dvostruka preciznost 64-bitne vrijednosti s pomičnim zarezom. Može se koristiti za prikazivanje cijelih i decimalnih brojeva
String	String	Predstavlja niz Unicode znakova
Boolean	Boolean	Predstavlja logičku vrijednost, *true* ili *false*
Void	Void	Koristi se za tipove funkcija koje ne vraćaju vrijednosti
Null	Null	Predstavlja namjernu odsutnost vrijednosti objekta
Undefined	Undefined	Označava vrijednost danu svim neinicijaliziranim varijablama

(Izvor: Cherny, 2019)

5.2. Angular

Posjeti li se Angular-ova službena web stranica, prva stvar koja se uočava je njegov logo i službeni slogan; *„One framework. Mobile & desktop“.* Iz slogana je vidljivo da se koristeći njega moguće razvijati na „svim“ platformama. Kad se jednom nauči razvijati aplikacija koristeći Angular, taj isti kod može biti iskorišten za razvoj na bilo kojoj platformi. Može se koristiti za razvoj web, mobilni web, native mobile i native desktop aplikacija.

Angular aplikacije danas postižu najveću moguću brzinu izvršavanja unutar modernih web preglednika, te nastavljaju povećavati brzinu koristeći web-radnike (*eng. Web Workers*) i generiranje na strani poslužitelja (*eng. server-side rendering*).

Mnoga integrirana razvojna okruženja (*eng. Integrated development environment*) ili skraćeno IDE imaju savršenu podršku za razvoj Angular aplikacija tako da se programeri ne moraju mnogo mučiti s pisanjem „koda koji će proraditi“, nego se mogu fokusirati na izradu aplikacija. Također postoji puno gotovih UI Angular komponenti (tablice, liste, izbornici, inputi, padajući izbornici, kalendari) koje se mogu svugdje jednostavno iskoristiti i nije ih potrebno svaki put pisati ispočetka.

Angular donosi produktivnu i skalabilnu infrastrukturu koja podržava najveće Google-ove aplikacija. Pogleda li se rad bilo koje Google-ove aplikacije, može se zaključiti da je ovo stvarno moćan razvojni okvir. (Google, Angular Docs, bez dat.)

5.2.1. Angular CLI

Angular CLI je komandno sučelje koje mnogo olakšava rad sa Angualarom. Angular CLI se vrlo jednostavno instalira pomoću *npm* upravitelja paketima (*eng. npm package manager*). Potrebno ga je instalirati samo jednom i nakon toga ga koristiti za kreiranje i rad s Angular aplikacijama. Koristeći CLI moguće je pozivom samo četiri naredbe kreirati novu Angular aplikaciju i pokrenuti lokalni poslužitelj na kojem se može testirati novokreirana aplikacija. U ovom primjeru prva se naredba koristi za instalaciju CLI-a. (Google, Angular CLI, bez dat.)

```
npm install -g @angular/cli
ng new moja-nova-aplikacija
cd moja-nova-aplikacija
ng serve
```

- **ng new** - koristi najbolje prakse te kreira gotovu Angular aplikaciju koja je odmah spremna za pokretanje.

- **ng generate** – služi za jednostavno generiranje komponenti, ruta (*eng. routes*), servisa i cijevi (*eng pipes*). Automatski kreira jednostavne testove za sve kreirane stvari.
- **ng serve** – pokreće lokalni poslužitelj na kojem se jednostavno može testirati aplikacija i automatski su vidljive sve promjene koje radimo tijekom razvoja
- **Test, Lint** – lint se koristi da kod lijepo izgleda (jednaka poravnanja, imenovanja, struktura koda), dok se *Test* koristi za pokretanje jediničnih i integracijskih (*eng. end-to-end*) testova

5.2.2. Pregled arhitekture

Angular je platforma i razvojni okvir za izgradnju klijentskih aplikacija u HTML-u i TypeScriptu (i Angular je sam napisan u ovom jeziku). On implementira osnovnu i opcionalnu funkcionalnost kao skup TypeScript biblioteka koje se mogu uvesti (*eng. import*) u aplikaciju.

Osnovni gradivni blokovi Angular aplikacije su NgModul-i, koji pružaju kompilacijski kontekst za komponente. NgModuli objedinjuju (*eng. consolidate*) povezani kod u funkcionalne cjeline. Angular aplikacija definirana je skupom NgModula. Aplikacija uvijek ima najmanje jedan modul i to korijenski (*eng. a root module*) koji omogućuje pokretanje aplikacije (*eng. bootstrapping*). Uglavnom aplikacije imaju mnogo više modula (*eng. feature modules*).

- Komponente definiraju *poglede* (*eng. views*). Pogledi su skupovi elemenata zaslona koje Angular može prikazivati i skrivati ovisno o logici programa.
- Komponente koriste *servise* koji pružaju specifičnu funkcionalnost koja nije izravno povezana s pogledom. Servisi (*eng. service providers*) mogu biti ubačeni (*eng. inject*) u komponentu kao ovisnosti čineći tako kod modularnim, ponovljivim i učinkovitim.

Komponente i *servisi* su jednostavne klase s dekoraterima koji označavaju njihov tip i pružaju metapodatke koji govore Angularu kako ih koristiti.

- Metapodatci za **klasu komponente** povezuju komponentu s predloškom koji definira pogled i po potrebi s datotekom u kojoj se nalaze stilska obilježja (.css, .sass, .scss). Predložak kombinira običan HTML s Angular direktivama i oznakama za povezivanje (*eng. binding markup*) koje omogućuju Angularu izmjenu HTML-a prije prikaza.
- Metapodatci za **klasu servisa** pružaju informacije koje Angular treba kako bi učinio servis dostupnim komponentama kroz *dependency injection (DI)*.

Komponente aplikacije uglavnom definiraju više pogleda koji su raspoređeni hijerarhijski. Angular pruža navigacijski servis/usmjerivač (*eng. the Router service*) koji nam pomaže definirati navigaciju između pogleda. Usmjerivač pruža sofisticirane pregledničke (*eng. in-browser*) navigacijske mogućnosti.

5.2.3. Moduli

Angular NgModuli se razlikuju, odnosno nadopunjuju JavaScript (ES2015) module. NgModule deklarira kompilacijski kontekst za skup komponenti koje su usko povezane, tj. komponenti koje služe za realizaciju jedne funkcionalnosti. NgModuli mogu povezati komponente sa zajedničkim/povezanim kodom implementiranim unutar servisa i tako oblikovati funkcionalne jedinice.

Svaka Angular aplikacija ima korijenski modul koji se po konvenciji najčešće naziva *AppModule*. Ovaj modul sadrži mehanizam za pokretanje (*eng. bootstrap*) koji pokreće aplikaciju. Aplikacija obično sadrži više funkcionalnih modula.

Kao i JavaScript moduli, NgModuli mogu uključiti (*eng. import*) funkcionalnosti iz ostalih NgModula, te također mogu izvesti (*eng. export*) vlastitu funkcionalnost da bude dostupna drugim NgModulima. Primjerice da biste koristili navigacijski servis u aplikaciji, potrebno je uključiti *Router* NgModule.

Organiziranje koda u različite funkcionalne module pomaže u upravljanju razvoja velikih složenih aplikacija. Također se unutar jednog modula može „zapakirati“ kod koji će biti ponovno korišten na više različitih mjesta/aplikacija. Osim navedenog, ova tehnika omogućuje da se iskoristi prednost kasnog učitavanja (*eng. lazy-loading*) . Koristeći *kasno-učitavanja* se svaki pojedinačni modul učitava u preglednik tek na zahtjev kada je potreban. Tako se prilikom pokretanja aplikacije u preglednik učitaju samo moduli koji su neophodni za pokretanje, a ostali se učitavaju na zahtjev tek kada su potrebni za izvršavanje određene funkcionalnosti.

```
import { BrowserModule } from '@angular/platform-browser';
import { NgModule } from '@angular/core';
import { AppComponent } from './app.component';
import {AppRoutingModule} from './app-routing.module';
@NgModule({
  declarations: [AppComponent],
  imports: [BrowserModule, AppRoutingModule],
  providers: [],
  bootstrap: [AppComponent]
})
export class AppModule { }
```

5.3. Komponente

Svaka Angular aplikacija ima najmanje jednu komponentu; korijensku komponentu koja povezuje hijerarhiju komponenti s DOM-om (*eng. document object model*) stranice. Svaka komponenta definira klasu koja sadrži aplikacijske podatke i logiku te je povezana s HTML predloškom koji definira *pogled* koji će biti prikazan u ciljanom okruženju. Dekorator *@Component()* identificira klasu koja se nalazi neposredno ispod kao komponentu te definira predložak i srodne metapodatke specifične za tu komponentu. (Fain, Moiseev, 2019)

> *Dekoratori su funkcije koje modificiraju JavaScript klase. Angular ima definirane mnoge dekoratore koji dodjeljuju klasama specifične metapodatke. Te informacije daju sustavu informaciju što su zapravo ove klase (komponenta, servis, dekorator, pipe) i kako one moraju raditi. (Google, Angular Docs, bez dat.)*

```
import { Component } from '@angular/core';

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent {
  title = 'diplomski-rad';
}
```

5.3.1. Predlošci, direktive i povezivanje podataka

Predlošci kombiniraju HTML s Angular oznakama (*eng. markup*) koje mogu izmijeniti HTML elemente prije nego će oni biti prikazani na ekranu. ***Direktive*** predloška omogućuje programsku logiku i povezuju aplikacijske podatke s DOM-om.

Postoje dvije vrste povezivanja (*eng. data-binding*):

- **Vezanje-događaja (*eng. Event binding*)** - omogućuje aplikaciji da odgovori (*eng. respond*) na korisničke unose u ciljanom okruženju tako da ažurira aplikacijske podatke
- **Vezanje-svojstava (*eng. Property binding*)** – omogućuje prikazivanje aplikacijskih podataka unutar HTML (npr. prikazivanje vrijednosti varijabli, liste podataka)

Prije nego je pogled prikazan, Angular evaluira (*eng. evaluates*) direktive i rješava (*eng. resolve*) sintaksu za vezanje (*eng. binding syntax*) u predlošku te izmjenjuje DOM i HTML elemente u skladu s programskim podatcima i logikom. Angular podržava dvosmjerno vezanje

podataka (*eng. two-way data binding*). To znači da se izmjene podataka unutar DOM-a (korisnički unos, izbori) automatski reflektiraju na podatke unutar programske logike.

Predlošci mogu koristiti i **cijevi** *(eng. pipes)* za poboljšanje korisničkog iskustva transformirajući vrijednosti koje će se prikazivati na ekranu. Na primjer, *cijevi* se koriste za prikaz datuma i vrijednosti valuta koji su prikladni za lokalizaciju korisnika (*eng. user's locale*). Angular pruža mnogo predefiniranih pipe-ova koji se najčešće koriste te također dozvoljava kreiranje vlastitih.

```
<div style="text-align:center">
  <h1>
    Dobrodošli u aplikaciju: {{ title }}!
  </h1>
</div>
```

5.3.2. Servisi i dependency injection

Za podatke i programsku logiku koja nije povezana s određenim pogledom i dijeljena je između više komponenti kreira se klasa koja je ***servis***. Definiciji klase servisa neposredno prethodi dekorator *@Injectable()*. Dekorator pruža metapodatke koji omogućuju opskrbljivačima (*eng. providers*) da mogu servis ***ubaciti*** kao ovisnosti (*eng. injected as dependencies*) unutar klase koja ga treba koristiti.

Dependency injection (DI) omogućuje održavanje klasa komponenti manjim (manje programskog koda) i učinkovitijim. Komponente tako ne dohvaćaju podatke izravno sa servera te ne ispisuju direktno u konzolu. Takve zadatke delegiraju da se obavljaju unutar servisa.

```
import { Injectable } from '@angular/core';

@Injectable({providedIn: 'root'})
export class AppService {

  constructor() { }
}
```

5.3.3. Routing

Angular Router NgModule pruža servis koji omogućuje navigaciju između različitih pogleda unutar aplikacije. Radi po uzoru na poznate konvencije o navigaciji unutar preglednika:

- Unese se URL u adresnu traku i preglednik kreće na odgovarajuću stranicu
- Klikom na poveznicu preglednik otvara/navigira novu stranicu
- Klikom na gumbe „povratak“ i „naprijed“ preglednik se navigira kroz povijest stranica koju su prethodno posjećivane

Ruter mapira URL (odnosno putanje) na poglede umjesto na stranice. Kada korisnik izvrši radnju, kao što je klik na poveznicu koja bi učitala novu stranicu u pregledniku, usmjernik presreće (*eng. intercept*) ponašanje preglednika i prikazuje ili skriva određene poglede.

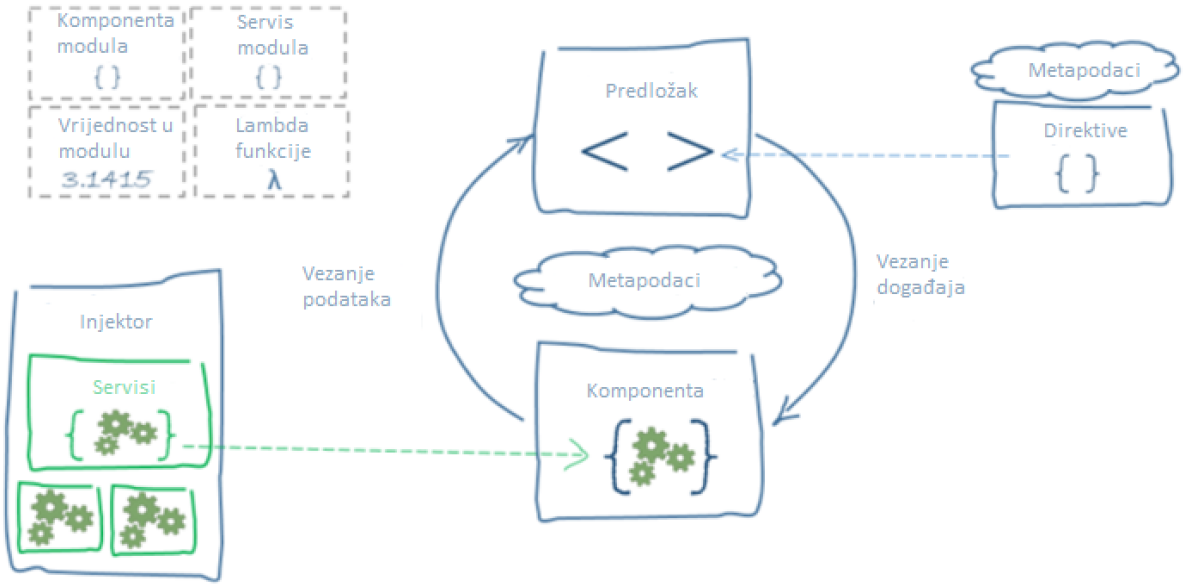
Ako usmjernik utvrdi da trenutno stanje aplikacije zahtjeva određenu funkcionalnost, a modul koji je sadrži se još nije učitao, ruter može koristeći kasno-učitavanje i na zahtjev dohvatiti potreban modul.

Usmjernik bilježi sve aktivnosti u povijesti preglednika tako da se mogu koristiti i navigacijske tipke preglednika.

```
import { NgModule } from '@angular/core';
import { Routes, RouterModule } from '@angular/router';
import {AppComponent} from './app.component';

const routes: Routes = [
  { path: '', component: AppComponent }
];

@NgModule({
  imports: [RouterModule.forChild(routes)],
  exports: [RouterModule]
})
export class AppRoutingModule { }
```

5.3.4. Dijagram arhitekture

Nakon što su u nekoliko prethodnih poglavlja objašnjene osnovne stvari o glavnim gradivnim elementima Angular aplikacije. Na sljedećem dijagramu je prikazano kako su svi ključni dijelovi aplikacije međusobno povezani.

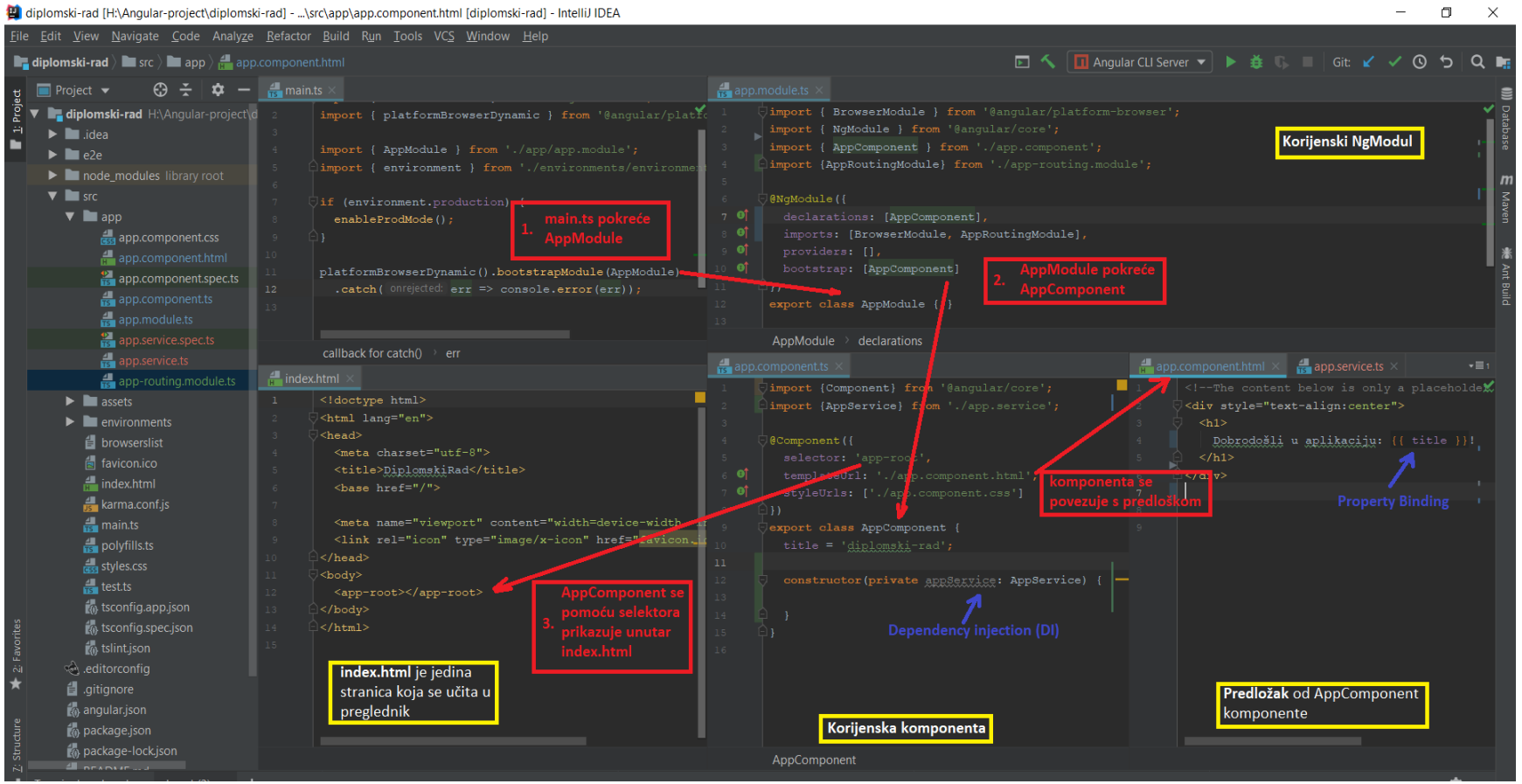


Slika 12: Dijagram arhitekture (Google, Angular Docs, bez dat.)

Komponenta i predložak zajedno definiraju jedan Angular pogled.

- Dekorator na klasi komponente dodaje metapodatke uključujući putanju do pripadajućeg predloška.
- Direktive i oznake za povezivanje (*eng. binding-markup*) u predlošku komponente mijenjaju *poglede* ovisno o programskoj logici i podatcima.

Dependency injector pruža (*eng. provide*) *servise* komponentama.



Slika 13: Prikaz sastavnih djelova Angular aplika

6. Postupak migracije web aplikacije sa starije tehnologije

6.1. Opis starije tehnologije (JSF, Primefaces)

Web dio stare aplikacije izveden je u Java tehnologiji JSF (*JavaServer Faces*) te koristi PrimeFaces komponente za izgradnju korisničkog sučelja. Ova aplikacija u svojem sloju poslovne logike poziva pohranjene procedure i web servise u kojima se izvršava prava poslovna logika, pristup bazi podataka i poziv CICS transakcija.

6.1.1. JavaServer Faces

JavaServer Faces (*JSF*) je standardna Java tehnologija za izgradnju korisničkih sučelja u obliku više komponenti (*eng. component-based*) i orijentiranih na događaje (eng. *event-oriented)*. JSF je zapravo programski okvir za razvoj web aplikacija, izgradnju komponenti korisničkog sučelja na poslužiteljskoj strani, te njihovo korištenje u web aplikaciji. JSF tehnologija se temelji na MVC (Model – View - Controller) arhitekturi za odvajanje logike od prezentacije.

MVC uzorak dizajna sastavlja/dizajnira aplikaciju koristeći tri odvojene cjeline.

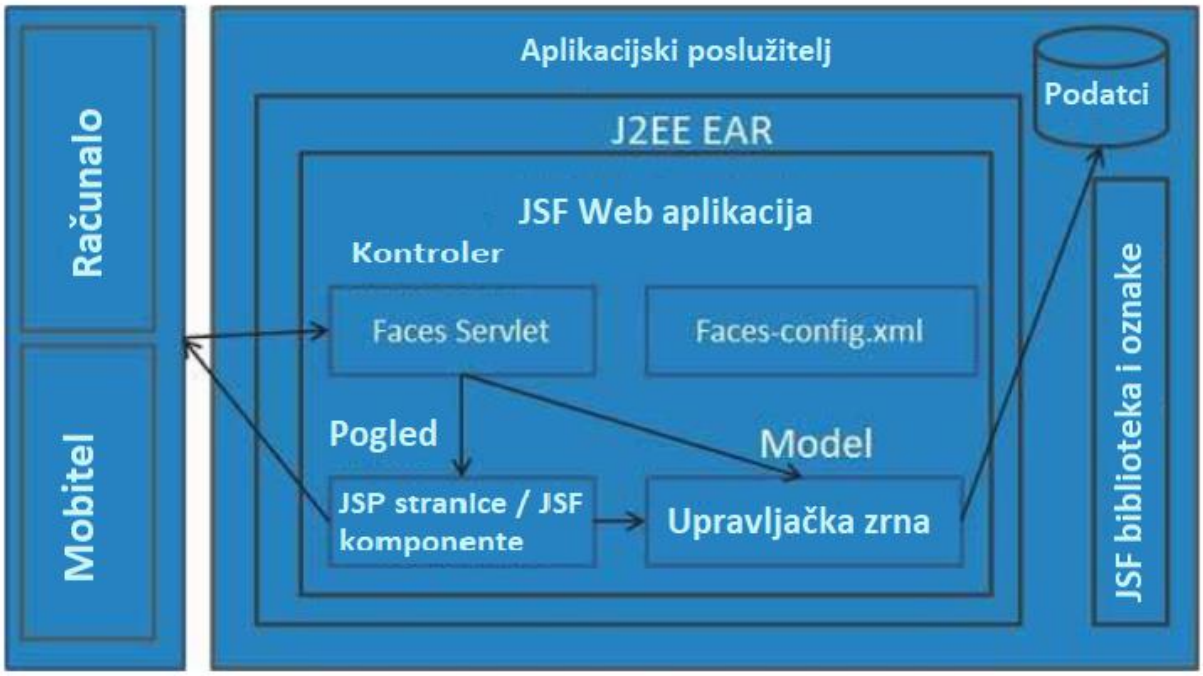
- ***Model*** – dio za rad s podatcima, pristup podatcima
- ***View*** – dio za izradu i prikazivanje korisničkog sučelja
- ***Controller*** – dio za upravljanje procesima aplikacije

6.1.2. JSF Arhitektura

JSF aplikacija je slična bilo kojoj drugoj Java web aplikaciji. Pokreće se unutar Java servlet kontejnera i sadrži: (Tyson, bez dat.)

- JavaBeans komponente kao modele koji sadrže funkcionalnost i podatke specifične za aplikaciju
- Prilagođena biblioteka oznaka (*eng. tag library*) za predstavljanje rukovatelja događaja (*eng. event-handlera*) i validatora
- Prilagođena biblioteka oznaka za prikazivanje UI komponenti
- UI komponente predstavljene su kao objekti sa stanjem (*eng. stateful*) na poslužitelju
- Poslužiteljske pomoćne klase (*eng. Server-side helper classes*)
- Validatori, rukovatelji-događaja, navigacijski-rukovatelji (*eng. navigation-handlers*)

- *Konfiguracijska datoteka (eng. Application configuration resource)* za konfiguraciju resursa aplikacije



Slika 14: JSF arhitektura, (Tutorials Point, bez dat.)

Postoje kontroleri koji se mogu koristiti za izvođenje korisničkih radnji. UI kreira autor web stranice, a poslovna logika može biti izvedena koristeći upravljačka zrna (*eng. managed beans*).

JSF pruža nekoliko mehanizama za prikazivanje/generiranje (*eng. rendering*) pojedinačne komponente. Ovaj mehanizam može odabrati web dizajner, a programer aplikacije uopće ne treba znati o kojem se mehanizmu radi. (Tyson, bez dat.)

6.1.3. PrimeFaces

PrimeFaces je jedan vrlo popularan open source razvojni okvir za JavaServer Faces koji sadrži više od 100 gotovih UI komponenti. Sve komponente su optimizirane za rad na dodir (mobiteli, tableti). Okvir također sadrži mehanizme za validaciju na strani korisnika i još mnogo toga. (PrimeFaces, bez dat.)

Aplikacija	Vrsta oznaka Posla	Kategorija	Šifra namjene	Način korištenja	Oznaka vrste plasmana	Poredak
AKR			L24	03	L/C with cash collateral	10
KRE	370			03	STL revolving	2
GAR			P46		Custom L/G	7
GAR			P13		Payment L/G	7

Slika 15: PrimeFaces dizajn tablice

6.2. Opis nove tehnologije

Kod izrade nove aplikacije koristit će se dvije ključne tehnologije. Jedna tehnologija je ***Angular 2+,*** korištena za izradu web 2.0 korisničke (*eng. frontend*) aplikacije koja je izgrađena koristeći ***Angular-material*** UI komponente. Druga tehnologija je Spring Boot aplikacija. Druga aplikacije je zapravo REST servisi kojima pristupa Angular aplikacija. REST servisi u svojoj izvedbi pozivaju postojeće pohranjene procedure i SOAP servise u kojima se nalazi poslovna logika cjelokupne aplikacije. Angular je detaljno objašnjen u prethodnim poglavljima, tako da će ovdje biti objašnjen samo Angular Material i Spring-boot.

6.2.1. Angular Material

Angular Material je skup gotovih komponenti korisničkog sučelja koje se besplatno mogu koristiti za kreiranje vlastite Angular aplikacije. To je skup sveobuhvatnih, modernih UI komponenti koje rade na webu, mobitelu i stolnom računalo. Kompletna tema svih komponenti se može prilagoditi korisniku. (Google, Angular Material, bez dat.)

Slika 16: Angular Material polja za unos (Google, Angular Material, bez dat.)

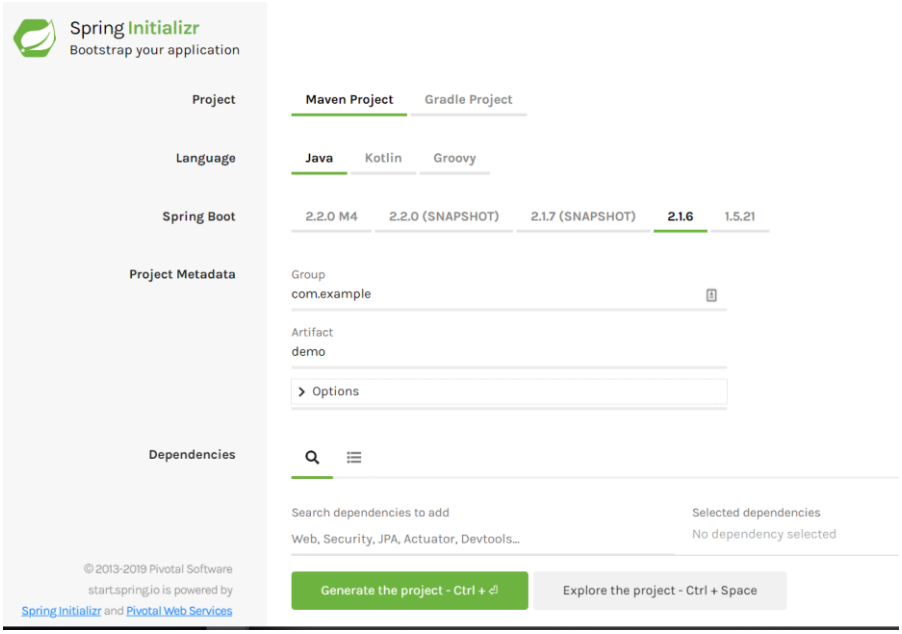
6.2.2. Spring Boot

Spring Boot je jedan izvanredan projekt/alat unutar Spring razvojnog okvira koji omogućuje jednostavno kreiranje samostalne (*eng. stand-alone*) Spring aplikacije koju je doslovno potrebno „samo pokrenuti“. Nema dodatnog posla s konfiguracijom servera, instalacijom *.jar ili *.war aplikacije na server i ostalih komplikacija koje s time dolaze. Spring Boot je osmišljen sa ciljem da se krene razvijati Spring aplikacija koristeći najmanji napor i s minimalnim potrebama za konfiguracijom. Nakon kreiranja, odmah se može krenuti na programiranje poslovne logike. Nakon programiranja, aplikacija se samo pokrene.

Svaka Spring Boot aplikacija ima ugrađeni (*eng. embedded*) Tomcat poslužitelj (može i Jetty ili Undertow) tako da nije potrebna „ručna“ instalacija WAR datoteka na određeni poslužitelj. Prilikom kreiranja nove aplikacije mogu se navesti „pokretačke“ ovisnosti (*eng. starter dependencies*) kako bi se pojednostavila konfiguracija izgradnje (*eng. build configuration*). (Walls, 2015)

6.2.2.1. Spring Initializr

Spring Initializr je u konačnici klasična web aplikacija koja može kreirati strukturu Spring Boot projekta. Neće generirati nikakav programski kod, ali će generirati osnovnu strukturu projekta (često ovisno o pokretačkim ovisnostima). Također će generirati Maven ili Gradle projekt i specifikaciju (*eng. build specification*) za izgradnju (*eng. build*) aplikacije. Sve što programer treba raditi je pisati programski kod aplikacije. (Walls, 2015)

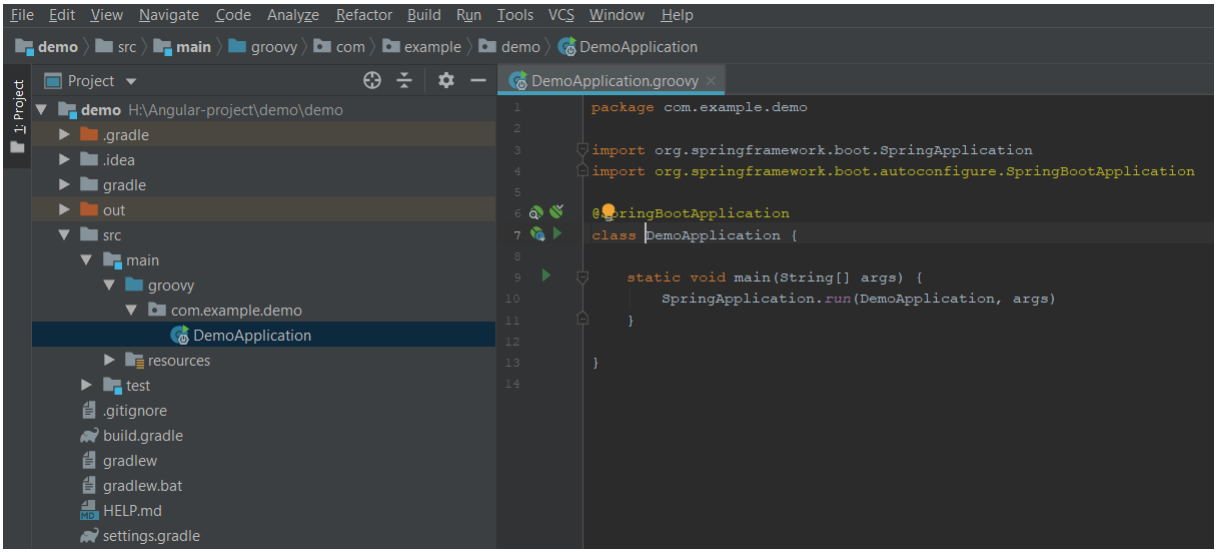


Slika 17: Spring Initializr web sučelje

6.2.2.2. Spring Boot RESTful Web Servis

Za kreiranje jednostavnog („Hello World“) RESTful web servisa potrebno je svega 15 minuta uloženog vremena. Ovo vrijeme je potrebno pod pretpostavkom da postoji instalirana Java JDK 1.8 +, Gradle 4+ i integrirano razvojno okruženje za pisanje programskog koda.

Koristeći Spring Boot initilizr jednostavno se odabere vrsta projekta (npr. Gradle), odabere programski jezik (npr. Groovy), definira naziv projekta i početnog paketa (*eng. package*). I što je najbitnije, pod dependencies se odabere „*Spring Web Starter*“. Initializr kreira aplikaciju koja je spremna za pokretanje.

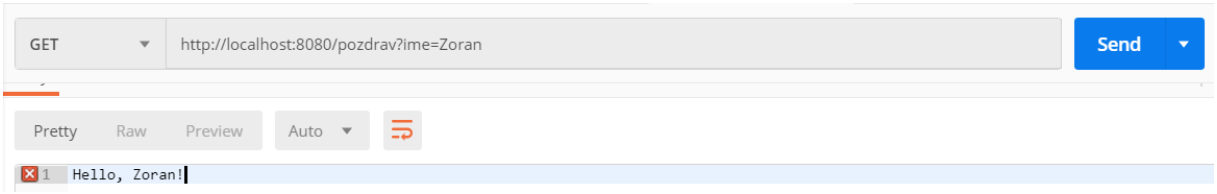


Slika 18: Spring Boot – Struktura projekta - Main metoda

Da bi se kreirao jedan jednostavan REST servis, sve što je potrebno napraviti je kreirati (čak i to nije potrebno) jednu novu klasu i anotirati je sa anotacijom *@RestController*. U toj klasi potrebno je jednu metodu anotirati s *@GetMapping*. Ova metoda će se izvršiti kad se na serveru pozove GET zahtjev na definiranoj putanji. U ovom slučaju *http://localhost:8080/pozdrav?ime=Zoran*.

```
package com.example.demo
import org.springframework.web.bind.annotation.GetMapping
import org.springframework.web.bind.annotation.RequestParam
import org.springframework.web.bind.annotation.RestController

@RestController
public class HelloController {
    private static final String template = "Hello, %s!"

    @GetMapping("/pozdrav")
    public String pozdrav(@RequestParam(value = "ime", defaultValue =
"World") String name) {
        return String.format(template, name)
    }}
```

Nakon pokretanja aplikacije može se pozvati servis na određenoj putanji i eto odgovora:



Slika 19: Poziv REST servisa

Izgled klase kontrolera je vrlo jednostavan, međutim ovdje ima mnogo toga što se događa „ispod haube“ zahvaljujući anotacijama i Spring razvojnom okviru. Razdvoji li se korak-po-korak može se primijetiti da postoji *@GetMapping* anotacija koja osigurava da su HTTP GET zahtjevi na krajnju točku (*eng. endpoint*) *„/pozdrav“* pridruženi metodi *pozdrav()*. Za definiranje određene vrste zahtjeva (GET, PUT, POST) može se koristiti sljedeća anotacija s definiranom metodom: *@RequestMapping(method=PUT).*

@RequestParam povezuje vrijednosti parametra iz poziva. U ovom slučaju povezuje parametar s nazivom *„ime“* na varijablu *String name*. Implementacija gornje metode će vratiti običan string sa definiranim sadržajem.

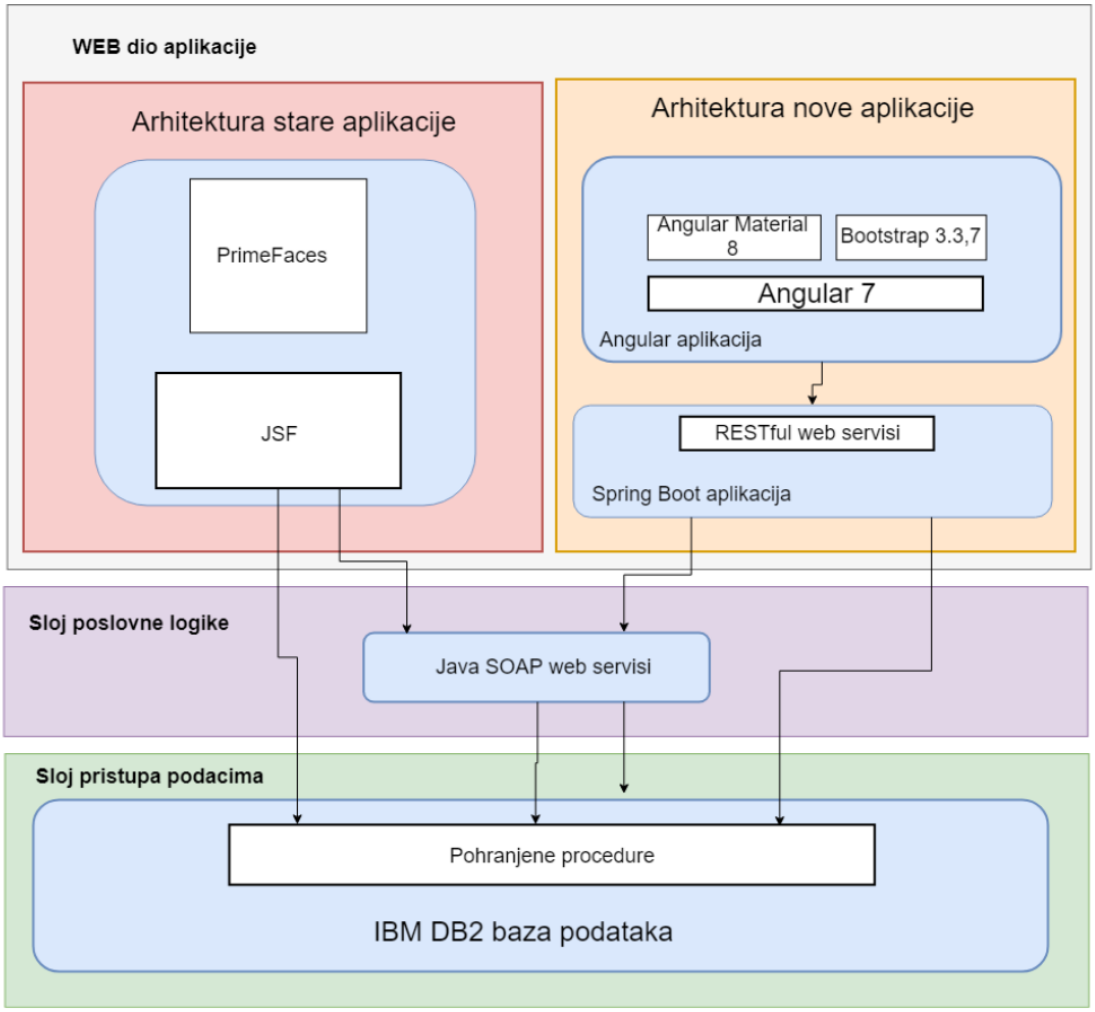
Ključna razlika između tradicionalnog MVC kontrolera i RESTful web servis kontrolera kao ovaj iznad je način na koji je tijelo HTTP odgovora (*eng. HTTP response body*) kreirano. Umjesto da se oslanja na tehnologiju prikaza koja na poslužitelju generira html stranicu, RESTfull web servis kontroler jednostavno vraća samo podatke. Ako kontroler vraća tip podataka koji je nekakav objekt, podatci o objektu će unutar HTTP odgovora biti zapisani u JSON formatu. (Walls, 2015)

6.3. Prikaz arhitekture starije aplikacije u usporedbi s novom

Ovom migracijom bit će obuhvaćeno migriranje web aplikacije. Gledajući sljedeću sliku, to je ovaj najgornji dio (WEB dio aplikacije). Iz jedne postojeće Java aplikacije, koja se temelji na JSF tehnologiji u kombinaciji PrimeFaces UI komponentama, kreirat će se dvije nove aplikacije.

Jedna aplikacija je prava korisnička web 2.0 aplikacija koja se temelji na razvojnom okviru Angular. Ova aplikacija putem REST servisa poziva drugu poslužiteljsku aplikaciju (*eng. backend*).

Ova druga aplikacija je zapravo REST servis izveden u tehnologiji Spring Boot i Groovy. REST servis u svojoj implementaciji metoda poziva ostale SOAP servise i pohranjene procedure.



Slika 20: Arhitektura stare i nove aplikacije

7. Primjer realizacije migracije postojeće Web aplikacije

7.1. Opis i arhitektura postojeće aplikacije

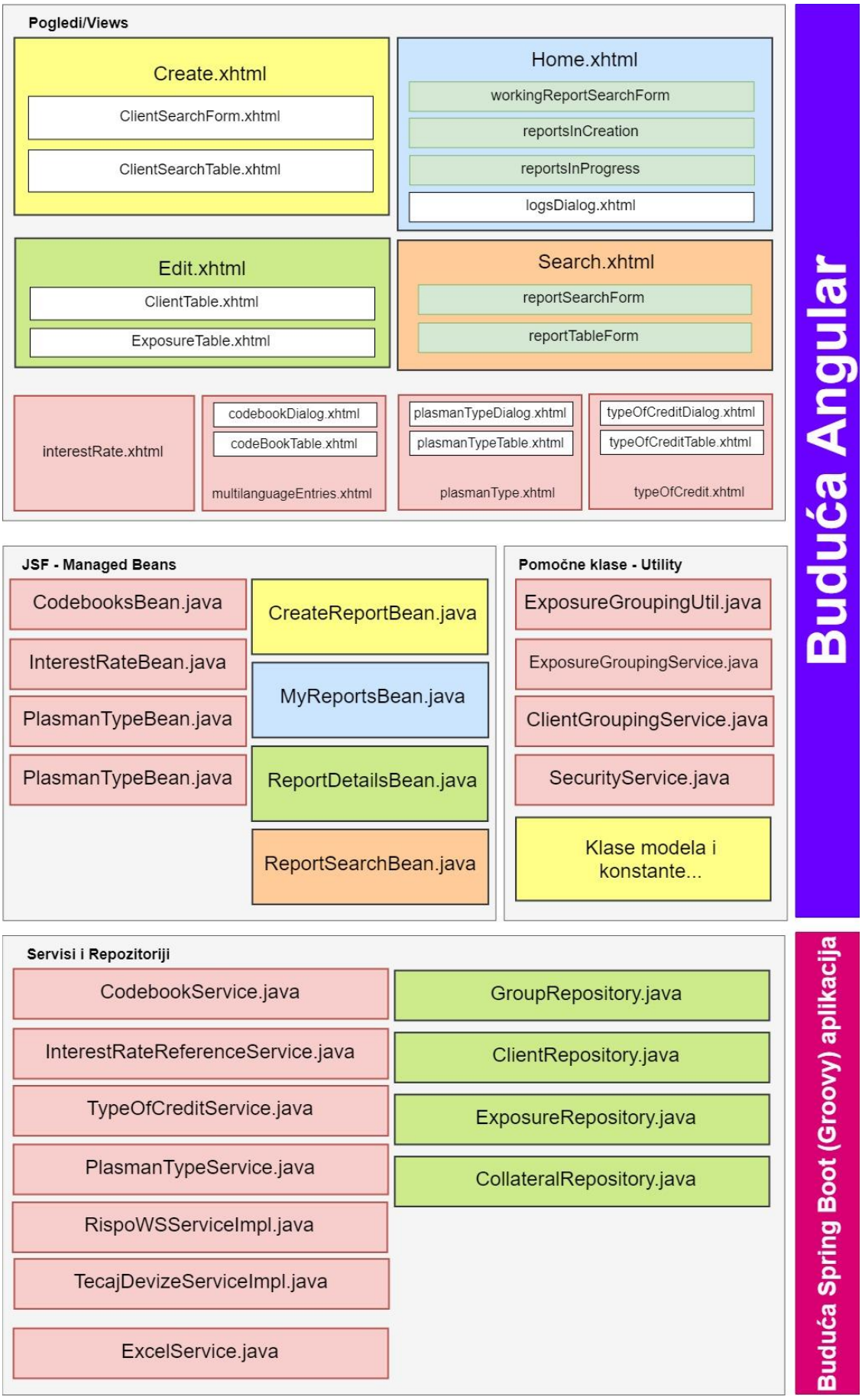
Na sljedećoj slici prikazano je od čega se sastoji postojeća Java Web aplikacija. Kako je aplikacija izvedena u tehnologiji JavaServer Faces (JSF), arhitektura je izvedena u tri logička sloja.

U prvi sloj mogu se svrstati pogledi (*eng. Views*) tj. predlošci. Predlošci su zapravo XML dokumenti s ekstenzijom *.xhtml. (*eng. EXtensible HyperText Markup Language*) Sintaksa predložaka jako podsjeća na klasičan HTML, međutim to nije to, već XML kod koji ima mnogo naprednih mogućnosti. Glavna ideja JSF-a je enkapsulacija klijentskih tehnologija (HTML, CSS, JavaScript) tako da programeri mogu razvijati web aplikacije bez poznavanje ijedne od ove tri tehnologije, dovoljno je samo znanje programskog jezika Java. Predlošci komuniciraju s upravljačkim zrnima (*eng. Managed Beans*) tako da mogu prikazivati podatke koji se u njima nalaze, odnosno na određene događaje pozivaju metode iz zrna.

U drugi sloj svrstavaju se upravljačka zrna. To su Java klase s točno definiranom strukturom (npr. moraju imati javno dostupne get-ere i set-ere). Zrna „komuniciraju“ s predlošcima i ona su na neki način veza između predloška i podatkovnog sloja, sloja u kojem se pozivaju web servisi i baza podataka (u ovom aplikaciji to su servisi i repozitoriji). Svako zrno ima određeni doseg (*eng. scope*). Doseg definira njegovo trajanje, tj, trenutak kreiranja i uništavanja (*eng. destroy*). Doseg može biti definiran kao: *RequestScoped*, *ViewScoped*, *SessionScoped*. (Tijms, Schlotz, 2018)

Trećem sloju pripadaju klase koje služe za rad s podatcima (dohvat, pohrana). Ove klase su anotirane sa @Service. Ova anotacija označava da postoji samo jedna instanca ove klase u aplikaciji i da je ta instanca u ostale klase po potrebi „injektirana“ koristeći *Dependency Injection*. U klasama sa sufiksom „*Repository*“ se pristupa bazi podataka tako da se u njima izvršavaju upiti nad bazom, odnosno pozivaju pohranjene procedure. Klase sa sufiksom „Service“ se koriste za pozive prema web servisima.

Na slici s desne strane je označeno koji dijelovi (klase) aplikacije će biti migrirane u korisničku (Angular) aplikaciju, odnosno što će postati buduća poslužiteljska aplikacija (REST servisi). Svi predlošci i upravljačka zrna će postati nova web 2.0 aplikacija, dok će klase zadužene za rad s podatcima postati REST servisi.



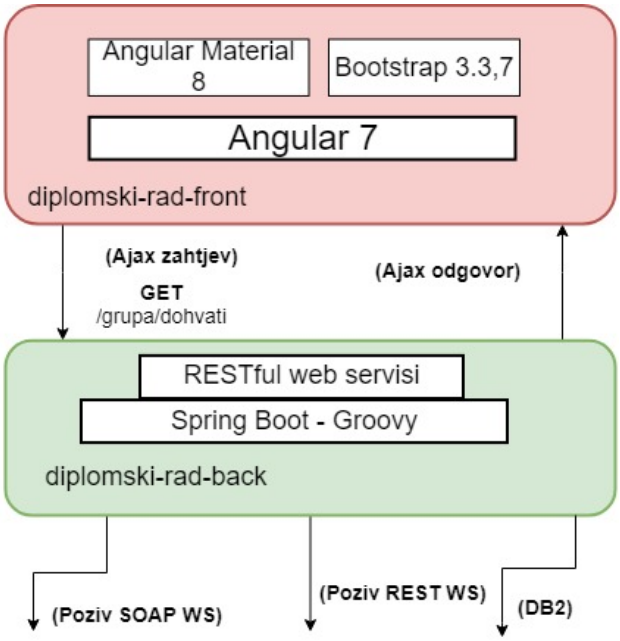
Slika 21: Arhitektura stare aplikacije

7.2. Arhitektura nove aplikacije

Kako u novoj „aplikaciji“ zapravo postoje dvije aplikacije, arhitektura se može proučavati na dvije razine. Na prvoj razini je vidljivo kako su aplikacije međusobno povezane te kako ostvaruju komunikaciju, dok se na drugoj razini može proučavati arhitektura unutar svake pojedine aplikacije.

Nova Angular (korisnička) aplikacija koristeći AJAX poziva RESTful web servise implementirane u novoj Spring Boot – Groovy (poslužiteljska) aplikaciji. Poslužiteljska aplikacija potom poziva sve potrebne SOAP i REST web servise, odnosno pohranjene procedure na određenoj bazi podataka.

Na sljedećoj slici može se vidjeti od čega se sastoji korisnička, odnosno poslužiteljska aplikacija. Korisnička aplikacija (diplomski-rad-front) je izvedena u tehnologiji Angular, te koristi Bootstrap 3.3.7 i Angular Material komponente grafičkog sučelja. Poslužiteljska aplikacija (diplomski-rad-back) napisna je u programskom jeziku Groovy koristeći Spring razvojni okvir. Korisnička aplikacija komunicira s poslužiteljskom koristeći REST web servis.



Slika 22: Povezivanje korisničke i poslužiteljske aplikacije

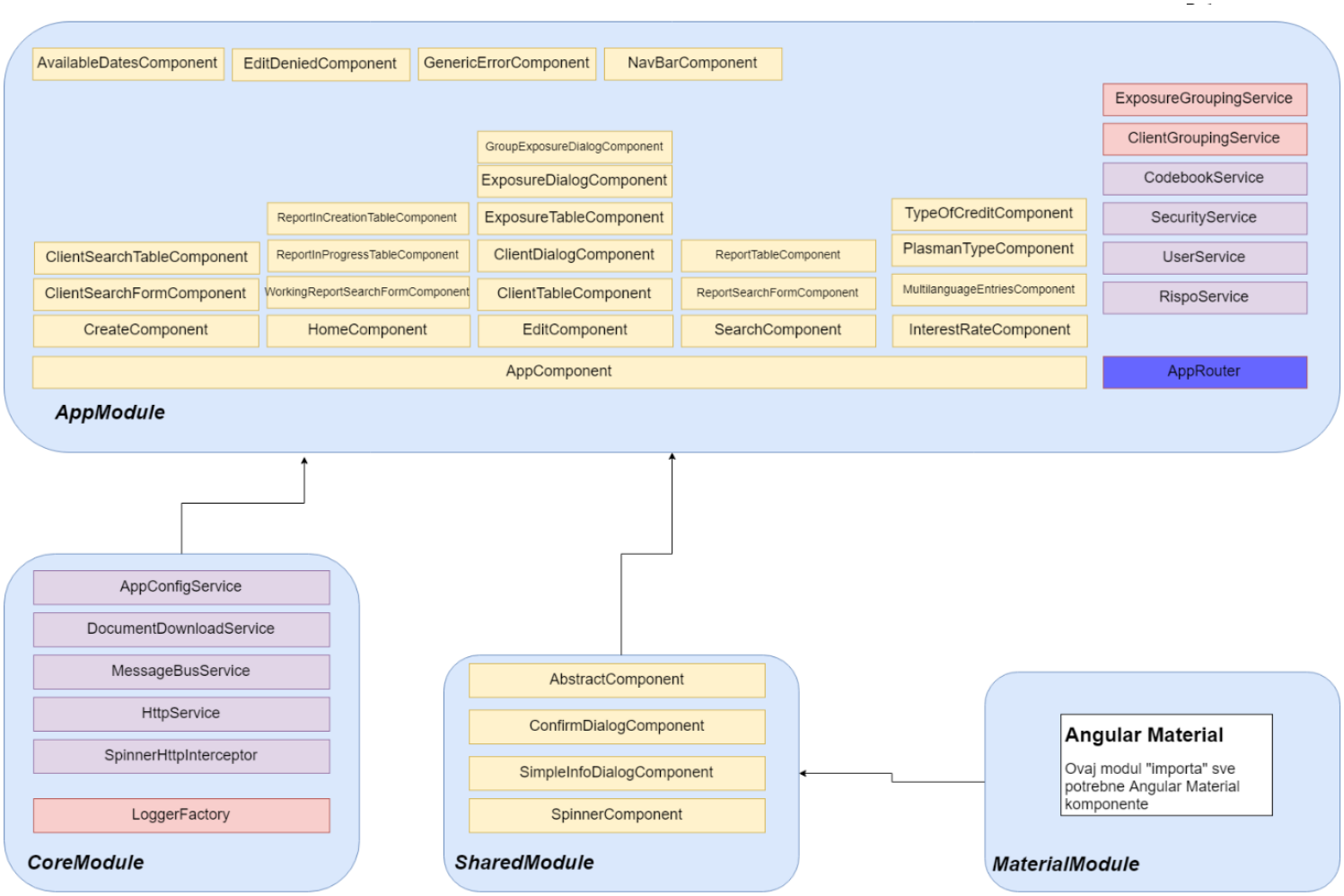
7.2.1. Arhitektura Angular aplikacije

Cijela Angular aplikacija građena je modularno. Svaki modul je jedna logička cjelina koja sadrži komponente i servise koji su zaduženi za izvršavanje istog posla. Ova aplikacija se sastoji od jednog glavnog modula, koji je ujedno i korijenski modul, te nekoliko pomočnih modula. Slika 23 prikazuje od čega se sastoji pokoji modul te kako su moduli međusobno povezani.

Glavni modul se zove *AppModule* i u njemu se nalaze sve komponente i servisi čitave aplikacije. Komponente služe za izgradnju ekrana i zadužene su za pravilan prikaz podataka, odnosno upravljanje korisničkim događajima. Servisi služe za obavljanje složenije poslovne logike, odnosno za pozivanje REST servisa koristeći AJAX.

CoreModule je pomoćni modul u kojem se nalaze svi servisi koji se koriste u cijeloj aplikaciji, odnosno u svim modulima (ako ih ima više). Glavna značajka ovog modula je to što on može biti samo jednom uvezen (*eng. import*) u aplikaciju, najbolje u korijenski modul. Na taj način se osigurava da u čitavoj aplikaciji postoji samo jedna instanca pojedinog servisa iz ovog modula.

Sljedeći pomoćni moduli su *SharedModule* i *MaterialModule*. U *SharedModule-u* su sadržane komponente koje se mogu (i potrebne su) koristiti na svim djelovima (u svim modulima) aplikacije. Modul može biti uvezen u bilo koji modul gdje je potrebno koristiti komponente koje on sadrži. *MaterialModule* sadrži sve Angular Material komponente koje se koriste u aplikaciji.



Slika 23: Arhitektura Angular aplikacije

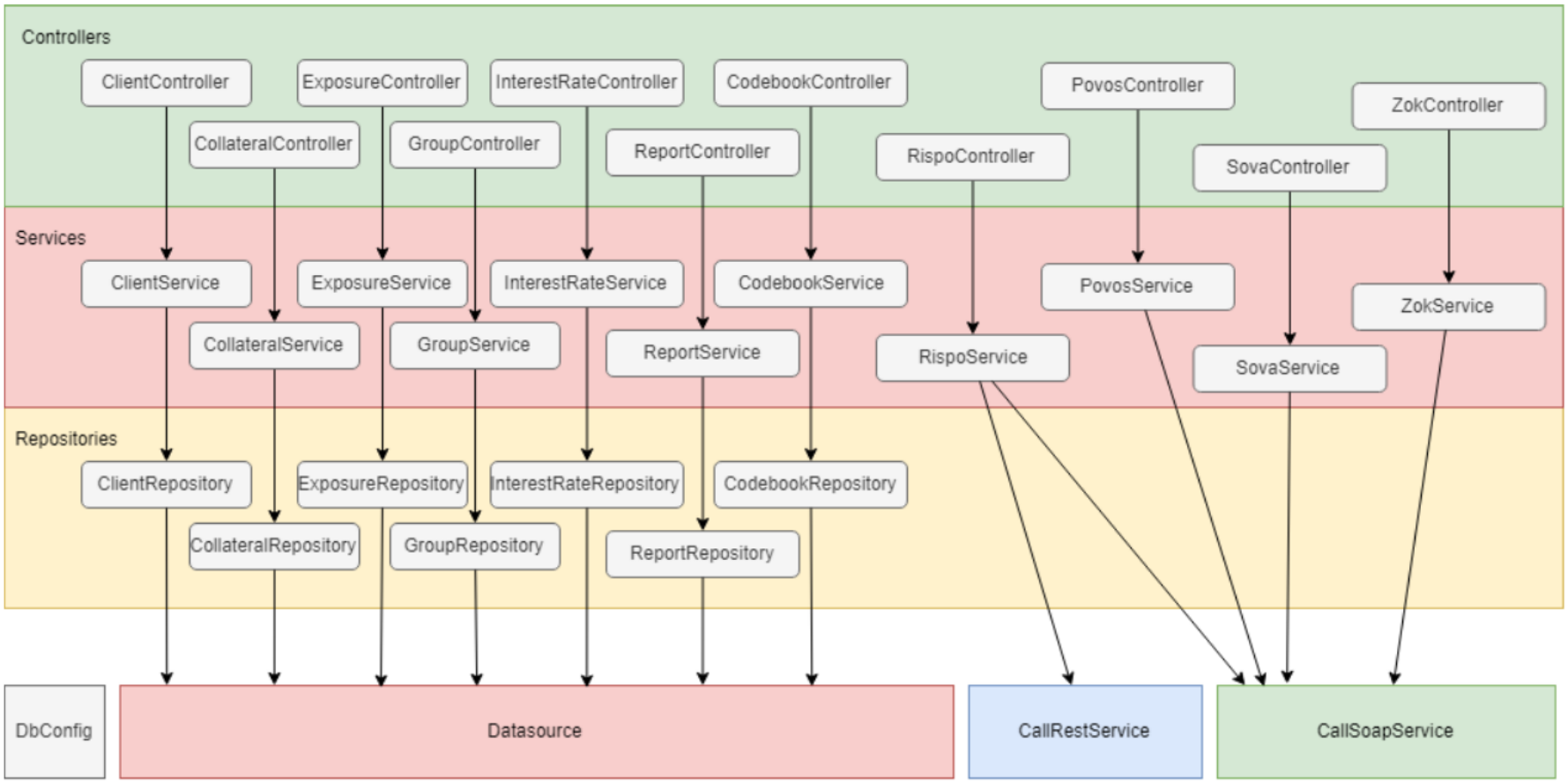
7.2.2. Arhitektura Spring Boot – Groovy aplikacije

Poslužiteljska aplikacija je izvedena koristeći Spring razvojni okvir i napisana je u programskom jeziku Groovy. Spring je korišten iz razloga što ima jako dobru podršku za jednostavno kreiranje REST servisa te svega potrebnog za konfiguraciju aplikacije. Kompletna aplikacija ima slojevitu arhitekturu. Na sljedećoj slici (Slika 24) je prikazano u kojem sloju se nalaze pokoje klase unutar aplikacije.

U prvom sloju se nalaze klase kontroleri (*eng. Controllers*) kojima je glavna odgovornost da definiraju na kojoj putanji će biti dostupna pojedina metoda REST servisa, te određuju vrstu HTTP metode. Kontroler također definira tip ulaznih i izlaznih podataka pojedine metode servisa. Svaka metoda unutar kontrolera poziva određenu metodu iz klase unutar sljedećeg sloja.

Drugom sloju pripadaju klase servisi (*eng. Services*). U ovim klasama nalazi se sva potrebna poslovna logika REST servisa. Za rad s podatcima koriste se klase koje se nalaze unutar trećeg sloja.

Unutar trećeg sloja smještene su klase kojima je glavna i jedina odgovornost rad s podatcima. Klase u ovom sloju se nazivaju repozitoriji i imaju sufiks *Repository*. Klase sadrže logiku potrebnu za rad s bazom podataka. Unutar ove aplikacije klase uglavnom pozivaju pohranjene procedure (*eng. Stored Procedures*) i na taj način dohvaćaju odnosno pohranjuju podatke u bazu.



Slika 24: Arhitektura Spring aplikacije

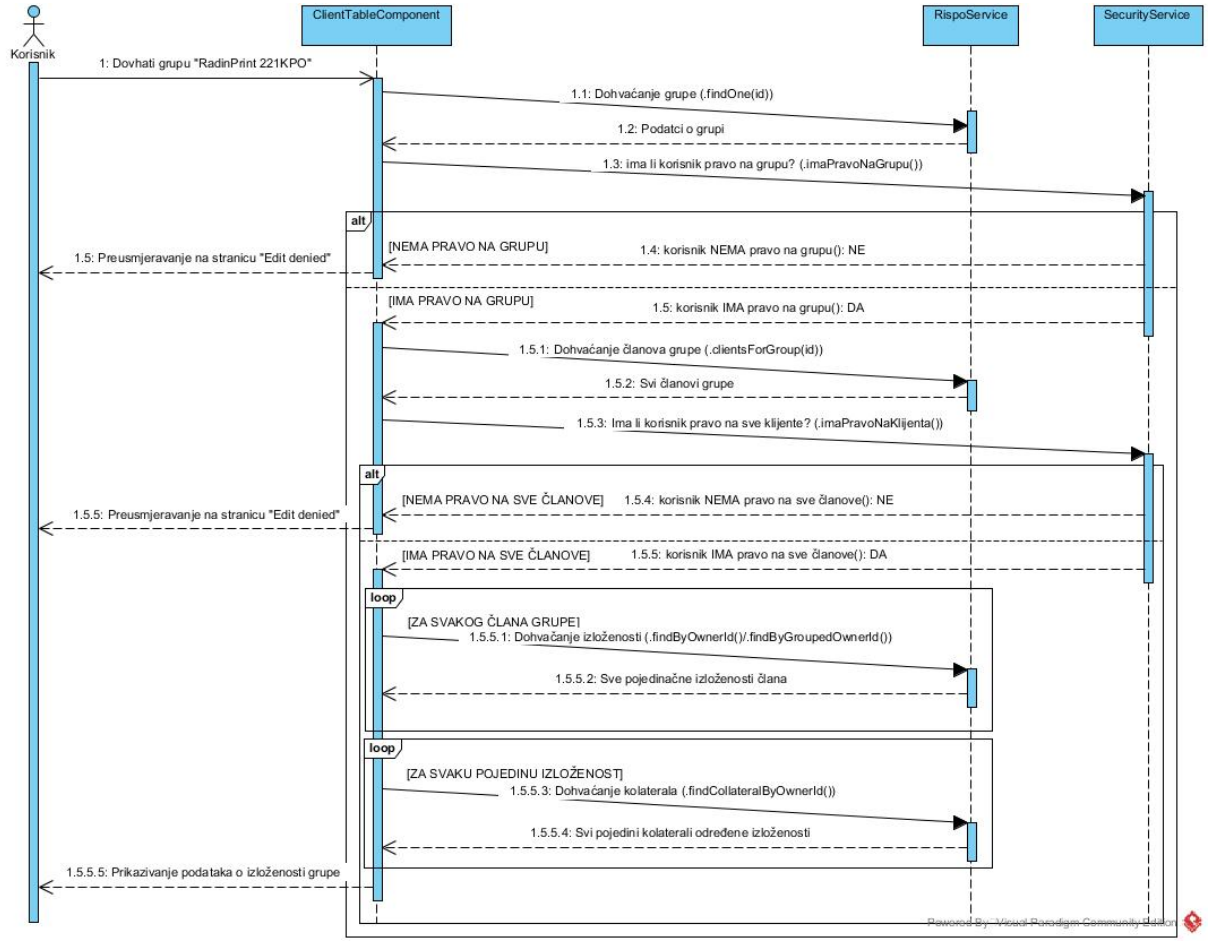
7.3. Usporedba programskog koda i koncepata stare i nove aplikacije

Kako bi se prikazalo koliko se u nekim segmentima razlikuje programski kod, odnosno izvršavanje programskog koda stare aplikacije u usporedbi s novom, naveden je jedan složeniji primjer iz aplikacije. Radi se o jednom zanimljivom primjeru u kojem se redom pozivaju web servisi tako da je odgovor prethodnog poziva parametar za poziv sljedećeg. Ovo je meni bio najizazovniji primjer za isprogramirati koristeći asinkrono programiranje.

Potrebno je izvršiti niz poziva servisa kako bi se dohvatili svi podatci o grupi, članovima grupe, izloženosti članova i kolateralima izloženosti.

Redoslijed pozivanja web-servisa:

1. Dohvatiti podatke o grupi s ID-em
2. Provjeriti ima li prijavljeni korisnik pravo vidjeti sve podatke o grupi
3. Ako ima pravo, dohvatiti sve članove grupe
4. Provjeriti ima li prijavljeni korisnik pravo vidjeti podatke svih članova grupe
5. Ako ima pravo, dohvatiti SVE izloženosti prema SVIM članovima grupe
6. Za svaku dohvaćenu izloženost dohvatiti SVE postojeće kolaterale



Slika 25: UML sekvencijalni dijagram – dohvaćanje podataka grupe

Na prethodnoj slici prikazan je UML sekvencijalni dijagram na kojem je vidljivo kako se redom pozivaju WEB servisi. Važno je napomenuti da je odgovor prethodnog servisa preduvjet za poziv sljedećeg servisa.

U staroj aplikaciji ovaj scenarij još i nije nešto previše zahtjevan iz razloga što se sve izvršava sinkrono i servisi se pozivaju sinkrono. Na liniji u kojoj se poziva web servis program „stoji“ tako dugo dok servis ne vrati odgovor i potom se izvršava sljedeća linija programskog koda. Sva logika se poprilično jednostavno izvede u dvije ugnježđene for petlje. Na sljedećoj stranici prikazan je kod koji služi za dohvaćanje svih podataka grupe u staroj aplikaciji.

```
private void loadGroupData(String id, boolean checkSecurity) {
try {
int index = 1, indexWithExposure = 1;
//  {1} dohvaćanje grupe
group = getRispoServiceManager().getGroupRepository().findOne(id);
if (checkSecurity) {
//  {2} dohvaćanje prava na grupe
    if
(getRispoServiceManager().getSecurityService().imaPravoNaGrupu(group,
userService.getUser().getOrgJeds())) {
        checkSecurity = false; //ima pravo na grupu, nemoramo gledati
klijente
    } else {
    if (group.getStatus() == ReportStatus.IN_PROGRESS) {
        // nema pravo na grupu i grupa ima status U_RADU
        //- nema prava jer za listu izvještaja u radu provjeravamo u
bazi grupu i klijente
        //jedini način da se ovo desi jest pristup direktno preko linka
        redirect();
        return;
    }
    } // END else
} // END if (checkSecurity)
if (group != null && group.getId() != null) {
    currency = group.getCurrency();
    //  {3} dohvaćanje članova grupe

group.setMembers(getRispoServiceManager().getClientRepository().getClien
tsForGroup(group.getId()));
    if (group.getMembers() != null) {
    for (Client member : group.getMembers()) {
        //  {4} provjera prava na sve članove
        if (checkSecurity &&
!getRispoServiceManager().getSecurityService().imaPravoNaKlijenta(member
, userService.getUser())) {
            redirect();
            return;
        }

        group.updateIntRate(member.getIntRateHRK(),
member.getIntRateEUR());
        group.updateFees(member.getFeesHRK(), member.getFeesEUR());
        member.setIndex(index++);

        //NT
        if (member.provjeriVrstuOsobe(Client.VRSTA_OSOBE_ZEMLJA)) {
            member.setIndexWithExposures(indexWithExposure++);
            continue;
        }

        //  {5} dohvaćanje SVIH izloženosti svakog člana
        if (member.isGrouped()) {
member.setExposures(getRispoServiceManager().getExposureRepository().fin
dByGroupedOwnerId(member.getId()));
        } else {
member.setExposures(getRispoServiceManager().getExposureRepository().fin
dByOwnerId(member.getId()));
        }
        if (member.getExposures() != null &&
!member.getExposures().isEmpty()) {
```

```
            member.setIndexWithExposures(indexWithExposure++);
            for (Exposure exposure : member.getExposures()) {
            member.getTotal().add(exposure);
            group.getTotal().add(exposure);

            //  {6} dohvaćanje SVIH kolaterala kod SVAKE izloženosti
exposure.setCollaterals(getRispoServiceManager().getCollateralRepository
().findByOwnerId(exposure.getId()));
            } // END for (Collaterals)
        } else if (member.isManualInput() || member.isError()) {
            member.setIndexWithExposures(indexWithExposure++);
        } // END else if
    } // END for (Exposures)
    } // END if (group.getMembers() != null)
} // END if (group != null && group.getId() != null)
} catch (Exception e) {
    redirectGeneric(); //NT
}
}
```

Kad se krene s programiranjem ovog scenarija u novoj aplikaciji vrlo brzo će se javiti mnogi problemi. U novoj aplikaciji se svi pozivi servisa izvršavaju asinkrono i izvršavanje ne „stoji“ dok se čeka odgovor servisa, nego se nastavlja izvršavanje. Kako je svaki odgovor servisa parametar za poziv sljedećeg servisa, najprimitivnije rješenje bi bilo u svaku funkciju s povratnim pozivom staviti poziv sljedećeg servisa i tako ugnježđavati u dubinu. Ovakav kod bi bio vrlo nepregledan, težak za održavanje i s vrlo mnogo potencijalnih pogrešaka. Sljedeći problem bi nastao kad treba za jedan odgovor servisa obaviti X poziva sljedećeg servisa i na kraju spojiti rezultate u jedan objekt ili jednu listu.

Programiranje ovakvih scenarija je gotovo nemoguće precizno izvesti bez naprednog znanja koncepata asinkronog programiranja odnosno dobrog poznavanja i korištenja RxJS operatora. Ovdje najviše pomažu operatori *switchMap* i *forkJoin*. Prvi operator omogućuje na jednostavan način izvesti da je rezultat jednog asniknonog poziva ulazni parametar za obavljanje drugog asinkonog poziva, a operator *forkJoin* omogućuje paralelno izvršavanje X asninkronih poziva i prikupljanje svih odgovora u listu nakon što posljednji završi.

```
private loadGroupDataPromise(id: string, checkSecurity: boolean):
Promise<Group> {

  return new Promise<Group>((resolve, reject) => {
    try {
      this.index = 1;
      this.indexWithExposure = 1;

      const context: LoadGroupModel = {id: id, checkSecurity:
checkSecurity, group: new Group()};

         // 1. dohvaćanje grupe
      fromPromise(this.findOneGroupPromise(context)).pipe(
        // 2. provjera prava na grupu
        switchMap(q => this.checkSecurityAndloadGroupMembersPromise(q)),
        // 3. dohvaćanje članova grupe
        switchMap(q => this.loadGroupMembersPromise(q)),
        // 4. provjera prava na sve članove grupe
        switchMap(q => this.checkSecurityAndloadExposurePromise(q)),
        // 5. dohvaćanje SVIH izloženosti svakog člana
        switchMap(q => this.loadExposureForEachMembersPromise(q)),
        // 6. dohvaćanje SVIH kolaterala svake izloženosti
        switchMap(q => this.laodAllCollateralsPromise(q))
      ).subscribe(response => {
        // završeno dohvaćanje svih podataka grupe
        // svi podatci su agregirani u jedan objekt klase "Group"
        this.group = response.group;
        this.rispoService.setReportsDetailsGroup(this.group);

        const group = this.group;

        for (const c of group.members) {
          if (c.shouldHaveExposure() && c.error) {
            this.addMessage(
              'GREŠKA',
              'Dogodila se greska kod dohvata izlozenosti za klijente
obojane crvenom bojom. ' +
              'Pokusajte rucno ponoviti dohvat samo za te klijente!');
            break;
          }
        }
        resolve(group);

      }, error1 => reject(error1));

    } catch (e) {
      this.log('ERROR loadGroupData' + e);
      this.loadGroupDataErrorHandling(e);
    }
  });
}
```

Na prethodnom djelu koda može se vidjeti korištenje operatora *switchMap* prilikom dohvaćanja svih podataka jedne grupe. Važno je napomenuti da je ovdje svaka metoda (*this.findOneGroupPromise(context), this.checkSecurityAndloadGroupMembersPromise(q)*) koja se poziva asinkrona i u sebi sadrži asinkorinkroni poziv web servisa i pripadajuću logiku.

```
private loadExposureForEachMembersPromise(loadGroupModel:
LoadGroupModel): Promise<LoadGroupModel> {

  return new Promise<LoadGroupModel>((resolve, reject) => {
    try {
  // 1. kreiranje liste tipa Promise
  // lista sadrži funkcije s povratnim pozivom
      const membersPromiseArray: Array<Promise<Client>> = new
Array<Promise<Client>>();

      loadGroupModel.group.members.forEach(member => {
        loadGroupModel.group.updateIntRate(member.intRateHRK,
member.intRateEUR);
        loadGroupModel.group.updateFees(member.feesHRK, member.feesEUR);
        member.index = this.index++;
        if (member.provjeriVrstuOsobe(Client.VRSTA_OSOBE_ZEMLJA)) {
          member.indexWithExposures = this.indexWithExposure;
          this.indexWithExposure++;
          return;
        }
 // 2. za svakog člana kreira se ASINKRONI poziv i dodaje se u listu
 // svaki poziv dohvaća SVE izloženosti JEDNOG člana
 membersPromiseArray.push(this.loadExposureForOneMemberPromise(member));
      });

// 3. forkJoin pokreće paralelno izvršavanje liste ASINKRONIH poziva
      forkJoin(
        membersPromiseArray
      ).subscribe(clients => {
// 4. svaki element liste "clients" je odgovor od jednog poziva WS
        clients.forEach(clientNew => {
// 5. ažuriranje podataka o svim članovima
          for (let i = 0; !!loadGroupModel.group.members && i <
loadGroupModel.group.members.length; i++) {
            if (loadGroupModel.group.members[i].id === clientNew.id) {
              loadGroupModel.group.members[i] = clientNew;
              continue;
            }
          }
        });

      }, error1 => {
        reject(error1);
      }, () => {
        resolve(loadGroupModel);
      });
    } catch (e) {
      this.log('ERROR: loadExposureForEachMembers -> ' + e);
      reject(e);
    }
  });
}
```

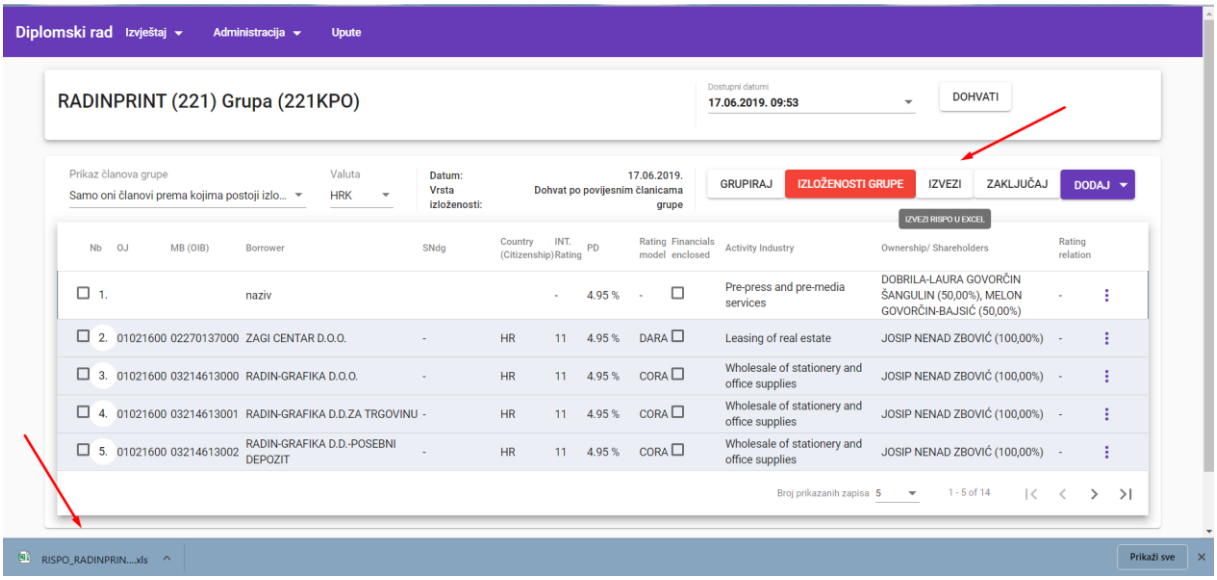
Iznad je prikazan isječak koda metode *loadExposureForEachMembersPromise()* koja obavlja dohvaćanje SVIH izloženosti SVIJU članova grupe. U prvom djelu koda može se vidjeti kako se za SVAKOG člana grupe kreira jedan asinkroni poziv i poziv se pohranjuje u listu

membersPromiseArray. U drugom djelu koda se nalazi operator *forkJoin* koji pokreće paralelno izvršavanje cijele liste asinkronih poziva i nakon toga se u *.subscribe()* nalazi funkcija s povratnim pozivom koja se izvršava nakon što svi pozivi vrate odgovor.

U ovom malom dijelu programskog koda može se primjetiti kako se asinkorno programiranje poprilično razlikuje u odnosu na klasično pisanje koda.

7.4. Usporedba izvoza izvještaja u Excel dokument u staroj i novoj aplikaciji

Na glavnom ekranu aplikacije, gdje se pregledavaju i uređuju kreirani izvještaji moguće je klikom na gumb „IZVEZI“ preuzeti izvještaj u obliku Excel dokumenta. U kreiranom dokumentu se prikazuju svi podatci vidljivi na ekranu, a također je dizajn dokumenta prilagođen za ispis na pisaču. Slika prikazuje ekran na kojem nalazi gumb za izvoz podataka u Excel dokument.



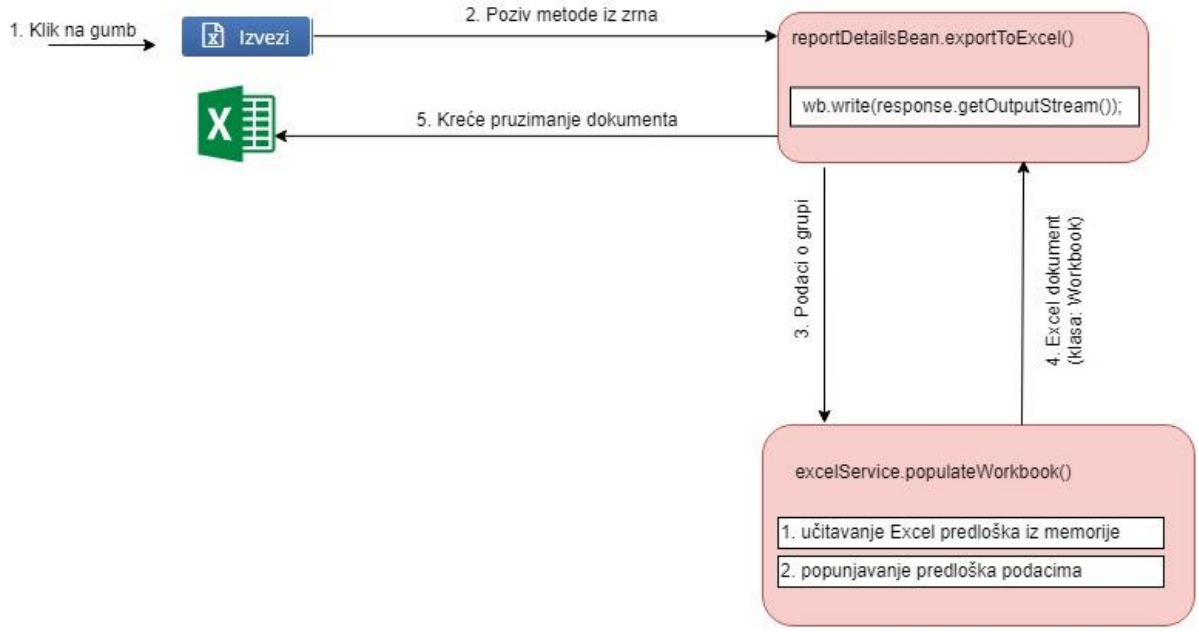
Slika 26: Ekran za pregled i izvoz izvještaja

Za kreiranje Excel izvještaja koristi se unaprijed izrađeni Excel predložak u koji se onda programski upisuju podatci. Kako bi se koristeći programski jezik Java moglo uređivati Excel dokument potrebno je dodatno preuzeti *Apache POI 3.6* biblioteku. Biblioteka omogućuje rad sa svim Microsoft dokumentima (Microsoft Word, Microsoft Excel). Kreiranje, odnosno prijenos dokumenta je u novoj aplikaciji izveden drugačije u odnosu na staru, a isto je prikazano u nastavku.

7.4.1. Izvoz podataka grupe u Excel dokument – STARA APLIKACIJA

U staroj aplikaciji je kompletan postupak kreiranja i preuzimanja izvještaja izveden jednostavnije u odnosu na novu aplikaciju iz razloga što je web aplikacija pisana u programskom jeziku Java. Angular ne podržava korištenje biblioteke Apache POI, te se u novoj aplikaciji rad s Excel dokumentom ne može izvoditi samo u korisničkoj aplikaciji, potrebno je koristiti poslužiteljsku aplikaciju. Više o tome kasnije.

Na sljedećoj slici prikazan je postupak kako je izveden izvoz podataka u staroj aplikaciji. Klikom na gumb „Izvezi“ izvršava se poziv metode *exportToExcel()* koja se nalazi u zrnu *ReportDetailsBean*. Važno je napomenuti da se ovdje izvršava klasično slanje čitave stranice prema poslužitelju, ne izvršava se AJAX poziv. Unutar zrna se poziva metoda *populateWorkbook()* iz klase *ExcelService* u kojoj se najprije iz memorije dohvaća Excel predložak i potom se pomoću *Apache POI* biblioteke u njega upisuju podatci. Kad je dokument spreman pokreće se preuzimanje.



Slika 27: Postupak izvoza podataka u Excel – STARA aplikacija

U nastavku je prikazan programski kod metode iz zrna. U kodu je vidljivo kako se najprije dohvaća HTTP odgovor i potom se poziva kreiranje Excel dokumenta. Nakon što je dokument uspješno kreiran unutar odgovora se postavljaju potrebni metapodatci i Excel dokument u obliku polja bajtova (*eng. bytes*).

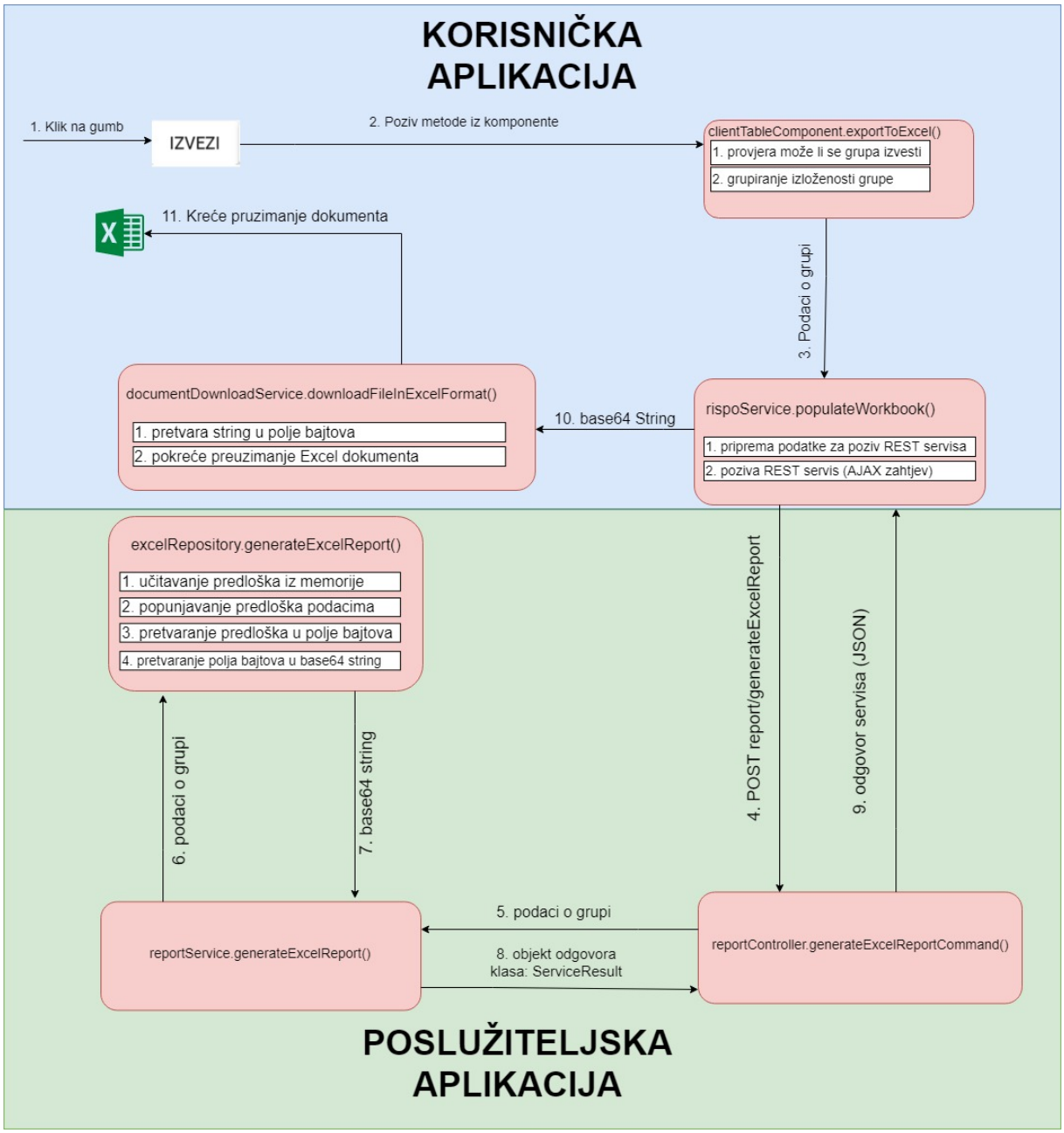
```
@StartEndLog
public void exportToExcel() throws IOException {
// dohvaćanje HTTP odgovora
    HttpServletResponse response = (HttpServletResponse)
FacesContext.getCurrentInstance()
            .getExternalContext()
            .getResponse();
// provjerava se može li grupa biti izvezena u Excel dokument
    if (!canExportGroup()) {
        FacesContext context = FacesContext.getCurrentInstance();
        context.getExternalContext()
                .getFlash()
                .setKeepMessages(true);
        context.getExternalContext()
                .redirect("Edit.xhtml?id=" + group.getId());
        return;
    }
    try {
// podatci se pripremaju za izvoz - grupiranje izloženosti
        Group tmpGroup = groupRetailExposures(group);
// poziva se metoda populateWorkbook() u kojoj se popunjava predložak
        Workbook wb = getRispoServiceManager()
                .getExcelService()
                .populateWorkbook(tmpGroup,
tmpGroup.getReportDate().toDate(), currency);
// unutar HTTP odgovora se postavljaju metapodatci
        response.setContentType("application/ms-excel; charset=UTF-8");
        response.setCharacterEncoding("UTF-8");
        response.setHeader(
                "Content-Disposition",
                "attachment; filename=" +
URLEncoder.encode(createFileName(),
                        "UTF-8").replace("+", "_"));
// unutar HTTP odgovora se "upisuje" Excel dokument
        wb.write(response.getOutputStream());
        wb.close();
        FacesContext.getCurrentInstance().responseComplete();
    } catch (Exception e) {
        log.error("Greska kod exporta grupe sa ID-jem " + group.getId(),
e);
        addMessage(FacesMessage.SEVERITY_ERROR,
                getTranslation(Const.EXPORT_EXCEL),
                getTranslation(Const.EXPORT_EXCEL_ERROR));
        FacesContext context = FacesContext.getCurrentInstance();
        context.getExternalContext()
                .getFlash()
                .setKeepMessages(true);
        context.getExternalContext()
                .redirect("Edit.xhtml?id=" + group.getId());
    }
}
```

U sljedećem isječku koda prikazana je metoda koja upisuje podatke u Excel predložak. Može se primijetiti kako je koristeći Apache POI biblioteku rad s Excel dokumentom vrlo jednostavan.

```
public Workbook populateWorkbook(Group group, Date date, String
exportCurrency)
        throws EncryptedDocumentException, InvalidFormatException,
IOException {
// učitavanje predloška iz memorije
    HSSFWorkbook wb = loadTemplate();
// dohvaćanje prve stranice dokumenta
    HSSFSheet sheet = wb.getSheetAt(0);
// dohvaćanje određene ćelije i upisivanje podataka
    getCellByReference(sheet, "H1")
            .setCellValue(group.getName());
    getCellByReference(sheet, "P4")
            .setCellValue(date);
    getCellByReference(sheet, "B53")
            .setCellValue("UniCredit Group (amounts in T" +
exportCurrency + ")");
    getCellByReference(sheet, "A567")
            .setCellValue("Credit Lines in T" + exportCurrency + "
(inclusive of indirect Risks) according UCI Rules");
    getCellByReference(sheet, "A583")
            .setCellValue("Credit Lines in T" + exportCurrency + "
according BACA Rules");
    getCellByReference(sheet, "A610")
            .setCellValue("Credit Lines in T" + exportCurrency + "
(inclusive of indirect Risks) according UCI Rules*)");
    getCellByReference(sheet, "A626")
            .setCellValue("Credit Lines in T" + exportCurrency + "
according BACA Rules");
// popunjavanje podataka članova grupe
    populateClients(group, sheet);
// popunjavanje podataka o izloženosti grupe
    populateExposures(group, sheet, exportCurrency, wb);
// pokretanje izračunavanje svih formula definiranih unutar predloška
    wb.setForceFormulaRecalculation(true);
    return wb;
}
```

7.4.2. Izvoz podataka grupe u Excel dokument – NOVA APLIKACIJA

Izvoz podataka u Excel dokument je u novoj aplikaciji nešto složeniji postupak u odnosu na staru. Glavni razlog je u tome što je aplikacija podijeljena na korisničku i poslužiteljsku. Kako bi se Excel dokument popunio podatcima najprije je potrebno iz korisničke aplikacije pozivom REST servisa poslati podatke o grupi u poslužiteljsku aplikaciju. Poslužiteljska aplikacija (Spring Boot - Groovy) tada iz memorije dohvaća Excel dokument/predložak i pomoću Apache POI biblioteke u dokument upisuje podatke o grupi. Dokument se potom pretvara u polje bajtova koje se tada koje se tada pretvara/enkodira (*eng. encode*) u base64 niz znakova (*eng. string*). Ovo pretvaranje (*eng. encode*) se radi iz razloga kako bi se dokument kroz mrežu (odgovor REST servisa) prenosio u obliku niza znakova. Korisnička aplikacija u metodi *populateWorkbook()* unutar klase *RispoService* kao odgovor REST web servisa prima niz znakova (base64 enkodirano polje bajtova). Niz znakova se prosljeđuje u metodu *downloadFileInExcelFormat()* unutar klase *DocumentDownloadService* u kojoj se tada dekodira (*eng. decode*) u polje bajtova te se pokreće preuzimanje datoteke. Na sljedećoj slici je prikazano kako korisnička i poslužiteljska aplikacija međusobno komuniciraju kao bi se kreirao i preuzeo Excel dokument. Također je prikazano koje klase, odnosno metode sudjeluju u implementaciji ove funkcionalnosti.



Slika 28: Postupak izvoza podataka u Excel – NOVA aplikacija

Na sljedećem isječku koda prikazana je metoda koja se poziva nakon klika na gumb „IZVEZI“. Metoda se nalazi unutar korisničke aplikacije u klasi *ClientTableComponent*. U metodi se najprije mora provjeriti može li grupa biti izvezena u Excel dokument i u slučaju da nije moguće uopće se ne poziva (AJAX zahtjev) REST servis za kreiranje dokumenta. Ako grupa može biti izvezena u dokument poziva se metoda *populateWorkbook()* unutar klase *RispoService*.

```
exportToExcel(): void {
  try {
 // 1. provjerava se može li grupa biti izvezena u Excel dokument
    if (!this.canExportGroup()) {
      return;
    }
 // 2. podatci se pripremaju za izvoz - grupiranje izloženosti
    const tmpGroup: Group =
this.groupRetailExposures(this.rispoService.getReportsDetailsGroup());
 // 3.poziva se WEB servis u kojem se kreira Excel dokument
    this.rispoService
      .populateWorkbook(tmpGroup, tmpGroup.reportDate, this.currency,
this.createFileName())
      .subscribe(value => {
        this.log('Uspješno kreiran Excell report');
      });
  } catch (e) {
    this.addMessage(
      Constants.EXPORT_EXCEL.toString(),
      Constants.EXPORT_EXCEL_ERROR.toString());
  }
}
```

U nastavku je prikazana metoda *populateWorkbook* koja se nalazi unutar klase *RispoService*. Metoda kreira DTO (*eng. data transfer object*) objekt za poziv REST servisa POST metodom, te potom izvršava pozivanje servisa. Nakon što servis vrati odgovor u obliku base64 niza znakova poziva se metoda *downloadFileInExcelFormat* unutar klase *DocumentDownloadService* koja će pokrenuti preuzimanje Excel dokumenta.

```
populateWorkbook(group: Group, date: Date, exportCurrency: string, name:
string): Observable<boolean> {

  // 1. pripremanje/kreiranje podataka za poziv REST servisa
  const command: GenerateExcelReportCommand = new
GenerateExcelReportCommand();
  command.group = new GroupCommand(group);
  command.date = date.getTime();
  command.exportCurrency = exportCurrency;

  try {

    // 2. pozivanje REST servisa POST metodom
    return this.httpService.submitRequestAndReturnData<any>({
      serviceUrl: `${RispoService.GENERATE_EXCEL}`,
      parseResponse: true,
      body: command,
      additionalHeaders: new Headers(
        {'x-call-tracking-token': RispoService.CALL_TRACKING_TOKEN}
        )
    }).pipe(
      map(data => {
        // odgovor (base64 string) REST servisa
        // se šalje u klasu koja će pokrenuti preuzimanje datoteke
        this.documentDownloadService
          .downloadFileInExcelFormat(data, name);
        return true;
      }), catchError(e => {
        return throwError(this.errorHandler(e));
      })
    );

  } catch (e) {
    this.logger.info('Error populateWorkbook: ERROR: ' + e);
    return throwError(
      this.errorHandler(e, 'Greska kod kreiranja Excell izvještaja!')
    );
  }
}
```

Sljedeći kod prikazuje još tri metode koje se također nalaze unutar korisničke aplikacije u klasi DocumentDownloadService. Naredne metode najprije pretvaraju base64 niz znakova u polje bajtova, a potom pokreću preuzimanje dokumenta s ekstenzijom .xls (Excel dokument).

```
  downloadFileInExcelFormat(base64str: any, fileName: string): void {
    this.downloadFile(
      base64str,
      fileName,
      'application/vnd.ms-excel',
      'xls'
    );
  }

  downloadFile(base64str: any, fileName: string, fileType: string,
extension: string): void {

    let view;
// pretvaranje (eng. decode) base64 stringa u polje bajtova
    const binary = atob(base64str.replace(/\s/g, ''));
    const len = binary.length;
    const buffer = new ArrayBuffer(len);
    view = new Uint8Array(buffer);
    for (let i = 0; i < len; i++) {
      view[i] = binary.charCodeAt(i);
    }
    this.downloadFileByte(view, fileName, fileType, extension);
  }

  downloadFileByte(binary: any, fileName: string, fileType: string,
extension: string): void {
    const blob = new Blob([binary], {type: fileType});
    if (window.navigator.msSaveOrOpenBlob) {
      window.navigator.msSaveOrOpenBlob(blob, fileName + '.' +
extension);
    } else {
      const a = window.document.createElement('a');
      a.href = window.URL.createObjectURL(blob);
      a.download = fileName + '.' + extension;
      document.body.appendChild(a);
      a.click();
      document.body.removeChild(a);
    }
  }
```

Dok se sve prethodne metode nalaze u korisničkoj aplikaciji, u nastavku će biti prikazane metode iz poslužiteljske aplikacije koje su zadužene za ostvarivanje funkcionalnosti izvoza podataka grupe u Excel dokument. Prva od njih je metoda *generateExcelReportCommand* koja se nalazi u klasi *ReportController*. Ova metoda definira na kojoj će putanji (*/generateExcelReport*) i s kojom HTTP metodom (*@PostMapping*) biti moguće pozvati određenu metodu (*reportService.generateExcelReport*()) REST web servisa. Također ova metoda kontrolera definira koji tip ulaznih parametara (*@RequestBody GenerateExcelReportCommand command*) prima ova metoda REST servisa. Važno je napomenuti da svaka metoda REST servisa u ovoj poslužiteljskoj aplikaciji vraća isti tip/klasu odgovora (*klasa: ServiceResult*). Klasa *ServiceResult* sadrži podatke o uspjehu izvršavanja metode web servisa, te sadrži podatke koji su rezultat izvršavanja određene metode. Klasa također može sadržavati listu poruka o greškama koje su se javile tjekom izvršavanja metode web servisa. Metoda *generateExcelReport* unutar klase *ReportService* poziva metodu (*excelRepository.generateExcelReport()*) za kreiranje Excel dokumenta i potom kreira objekt tipa *ServiceResult* koji će biti odgovor REST servisa.

```
@PostMapping("/generateExcelReport")
def generateExcelReportCommand(@RequestBody GenerateExcelReportCommand
command) {
    ServiceResult serviceResult =
reportService.generateExcelReport(command)
    return dataConversionService.parseServiceResult(serviceResult)
}
```

```
ServiceResult generateExcelReport(GenerateExcelReportCommand command) {
   log.info "ReportService.generateExcelReport $command"
   ServiceResult serviceResult = new ServiceResult()
   String result
   try  {
      result = this.excelRepository.generateExcelReport(command)
      // kreira objekt tipa ServiseResult
      // u "result" stavlja base64 string
      serviceResult = new ServiceResult(success: true, result: result)
   } catch (RispoException e) {
      serviceResult = new ServiceResult(
            success: false,
            errorMessageCodeList: ["rispo.error"],
            errorMessageTextList: [e.message])
   }
   log.info "ReportService.generateExcelReport"
   serviceResult
}
```

Metoda *generateExcelReport()* unutar klase *ExcelRepository* zadužena je za kreiranje Excel dokumenta. Unutar metode se najprije iz memorije učita Excel predložak i potom se poziva metoda *populateWorkbook()* koja koristeći Apache POI biblioteku upisuje podatke u predložak. Nakon upisa podataka dokument se najprije pretvara u polje bajtova, a potom se polje bajtova pretvara (*eng. encode*) u base64 niz znakova. Metoda vraća niz znakova (string).

```
String generateExcelReport(GenerateExcelReportCommand command) throws
RispoException {
    String retVal
    Workbook resultWorkbook
    HSSFWorkbook resultHSSFWorkbook
    byte[] resultByte
    try {
        // dohvaćanje Excel predloška iz memorij
        resultHSSFWorkbook = this.loadWorkbookReportTemplate()
        // pozivanje metode koja će upisati vrijednosti u predložak
        resultWorkbook = this.populateWorkbook(
                resultHSSFWorkbook,
                command.group,
                command.date,
                command.exportCurrency
        )
        // pretvaranje predloška u bolje bajtova
        resultByte = this.workbookToByte(resultWorkbook)
        // pretvaranje polja bajtova u base64 string
        retVal = this.byteToEncodedString(resultByte)
        resultWorkbook.close()
    } catch (Exception e) {
        throw new RispoException(
     "RispoException ExcelRepository.generateExcelReport $e.message"
        )
    }
    retVal
}
```

7.5. Prednosti i mane nove aplikacije

Nakon uspješno završene nove aplikacije u novoj tehnologiji (Angular), može se napraviti usporedba u odnosu na staru aplikaciju. Mogle bi se izdvojiti određene dobre i loše strane nove aplikacije, a po mojem mišljenju ima više prednosti nego nedostataka.

Kao prednosti aplikacije mogu se navesti da je čitava aplikacija izvedena u WEB 2.0 tehnologiji tako da nema osvježavanja stranice za vrijeme korištenja iste. Sljedeća, po meni, najveća prednost aplikacije je modularnost. Kompletna aplikacija može vrlo jednostavno biti dio druge veće Angular aplikacije kao jedan cjeloviti modul. Iduća bitna stvar koju će korisnici najprije primjetiti je brzina dohvaćanja podataka. Nova aplikacija višestruko brže dohvaća podatke i učitava čitav ekran u odnosu na staru.

Kao nedostatak nove aplikacije može se navesti vrlo zahtjevna tehnologija (Angular, RxJS) tako da je potrebno poprilično mnogo učenja da bi se uspjelo barem nešto malo napraviti. Također je potrebna dodatna poslužiteljska aplikacija (REST servisi) koja poziva postojeće SOAP servise i pohranjene procedure. Ova aplikacija je potrebna iz razloga što Angular ne može direktno pozivati SOAP servise.

7.5.1. Performanse nove aplikacije

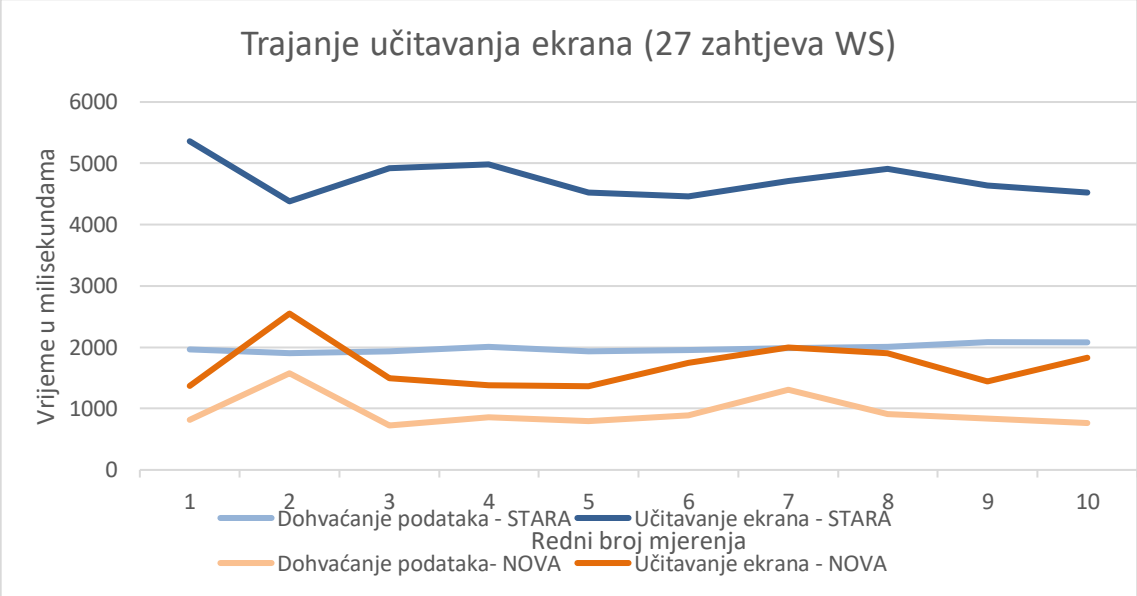
Obavljeno je mjerenje ponekih performansi nove aplikacije u odnosu na staru. Odabran je jedan najsloženiji/najzahtjevniji slučaj iz aplikacije i mjereno je vrijeme dohvata podataka, odnosno vrijeme potrebno za učitavanje kompletnog ekrana. Radi se o glavnom ekranu na kojem se dohvaćaju svi podatci odabrane grupe. U prethodnom poglavlju je detaljno opisano kako se redom pozivaju web servisi da bi se dohvatili svi podatci jedne grupe.

Izvedeno je mjerenje u dva slučaja. U prvom slučaju dohvaća se manja grupa od 10 članova, dok je u drugom slučaju odabrana jedna velika grupa koja se sastoji od 271 člana. U svakom mjerenju programski se mjeri vrijeme učitavanja svih podataka (***Doh. pod. grupe***), a štopericom se mjeri proteklo vrijeme od klika na željenu grupu do trenutka kad su se na ekranu pokazali svi podatci (***Učit. ekrana***). Za svaki slučaj obavljeno je više mjerenja i na kraju su uzeti prosječni rezultati.

Nakon obavljenog mjerenja na manjoj grupi već se mogu primjetiti pozitivni rezultati migracije. Na ovom primjeru nova aplikacija radi 2-3 puta brže. Umjesto prethodnih 4.7 sekundi potrebnih za učitavanje ekrana, nova aplikacija isti ekran prikaže za 1.7 sekundi. Rezultati mjerenja su sljedeći:

Tablica 3: Prikaz podataka o mjerenju brzine učitavanja manje grupe

	KPO		221KPO	
	Naziv		RADINPRINT (221) Grupa	
	Datum kreiranja		30.08.2019. 10:39	
	Broj članova grupe		9	
	Broj članova prema kojima postoji izloženost		7	
	Broj poziva (zahtjeva)WS		26	
	Mjerna jedinica		milisekunde (ms)	
	stara aplikacija		nova aplikacija	
Br. Testa	**Doh. pod. grupe**	**Učit. ekrana**	**Doh. pod. grupe2**	**Učit. ekrana2**
1	1969	5360	820	1370
2	1904	4380	1576	2550
3	1929	4920	725	1500
4	2012	4980	855	1380
5	1931	4520	795	1365
6	1956	4460	888	1750
7	1987	4710	1306	2000
8	2012	4910	906	1900
9	2084	4640	833	1444
10	2076	4520	766	1830
Prosjek:	1986	4740	947	1709
		Ubrzanje (staro/novo):	2	3

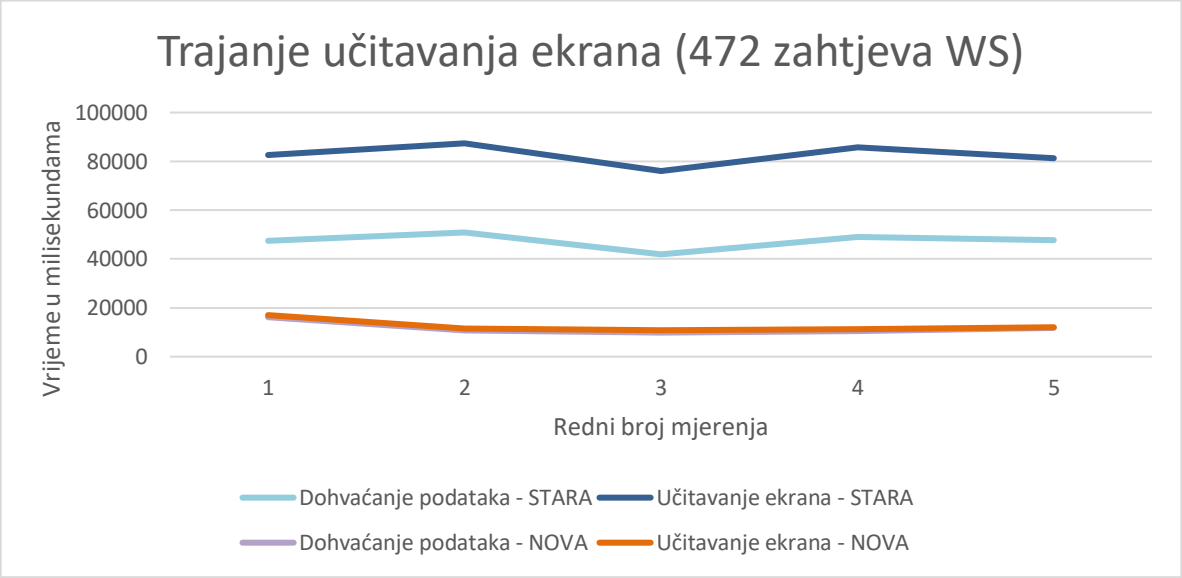


Slika 29: Rezultat mjerenja na maloj grupi

Nakon obavljenog mjerenja na većoj grupi primjećuju se izvanredni rezultati migracije. Na ovom primjeru nova aplikacija radi gotovo 7 puta brže u odnosu na staru aplikaciju. Umjesto prethodnih 1 minutu i 20 sekundi potrebnih za učitavanje ekrana, nova aplikacija isti ekran prikaže za svega 12 sekundi.

Tablica 4: Prikaz podataka o mjerenju brzine učitavanja velike grupe

	KPO		290KPO	
	Naziv		AGROKOR GRUPA	
	Datum kreiranja		30.08.2019. 11:07	
	Broj članova grupe		271	
	Broj članova prema kojima postoji izloženost		113	
	Broj poziva (zahtjeva)WS		472	
	Mjerna jedinica		milisekunde (ms)	
	stara aplikacija		nova aplikacija	
Br. Testa	**Doh. pod. Grupe**	**Učit. Ekrana**	**Doh. pod. grupe2**	**Učit. Ekrana2**
1	47432	82580	16163	17020
2	50870	87380	10645	11410
3	41916	76040	9855	10830
4	49156	85640	10493	11260
5	47633	81220	11839	11950
Prosjek:	47401	82572	11799	12494
		Ubrzanje (staro/novo):	4	7



Slika 30: Rezultat mjerenja na velikoj grupi

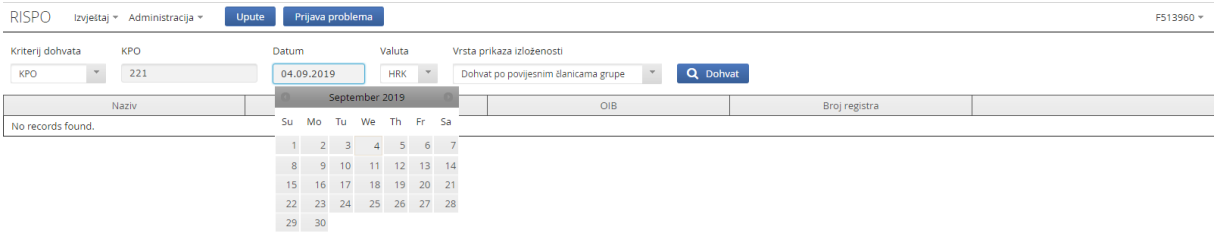
Što se tiče brzine rada aplikacije, rezultati su mnogo bolji nego je uopće bilo planirano. Obzirom da je korisnicima brzina izvođenja aplikacije jedna od ključnih osobina, može se zaključiti da je migracija uspješno završila. Također se može primjetiti da je u novoj aplikaciji vrijeme učitavanja ekrana gotovo jednako vremenu dohvaćanja svih podataka.

7.6. Izgled ekrana nove i stare aplikacije

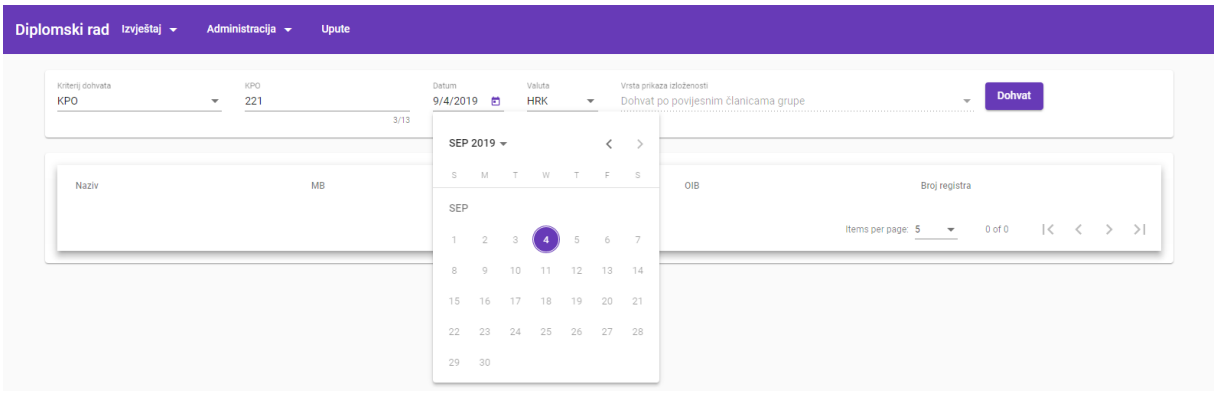
U ovom poglavlju bit će prikazani izgledi bitnijih ekrana stare i nove aplikacije, postoje još pokoji ekrani i dijalozi koji nisu toliko važni, te neće biti ovdje posebno navedeni.

7.6.1. Kreiranje izvještaja

Ekran na kojem se za određeni datum kreira izvještaj za određenu grupu. Grupa se može odabrati po više kriterija (KPO, Matični broj, Naziv, OIB).



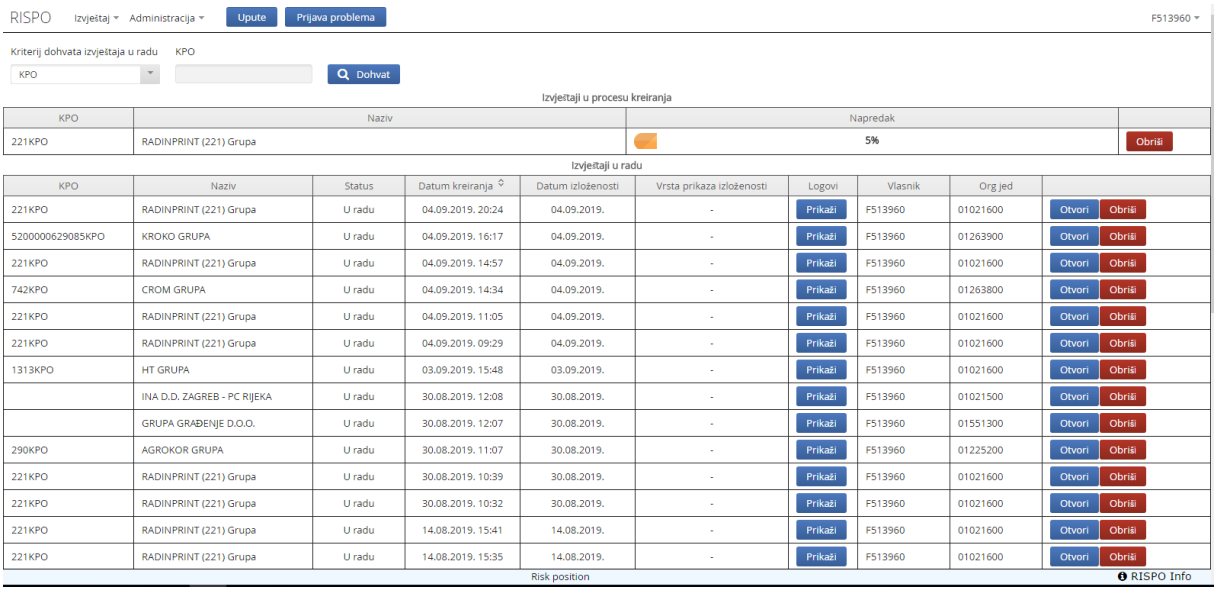
Slika 31: Kreiranje izvještaja – STARA APLIKACIJA



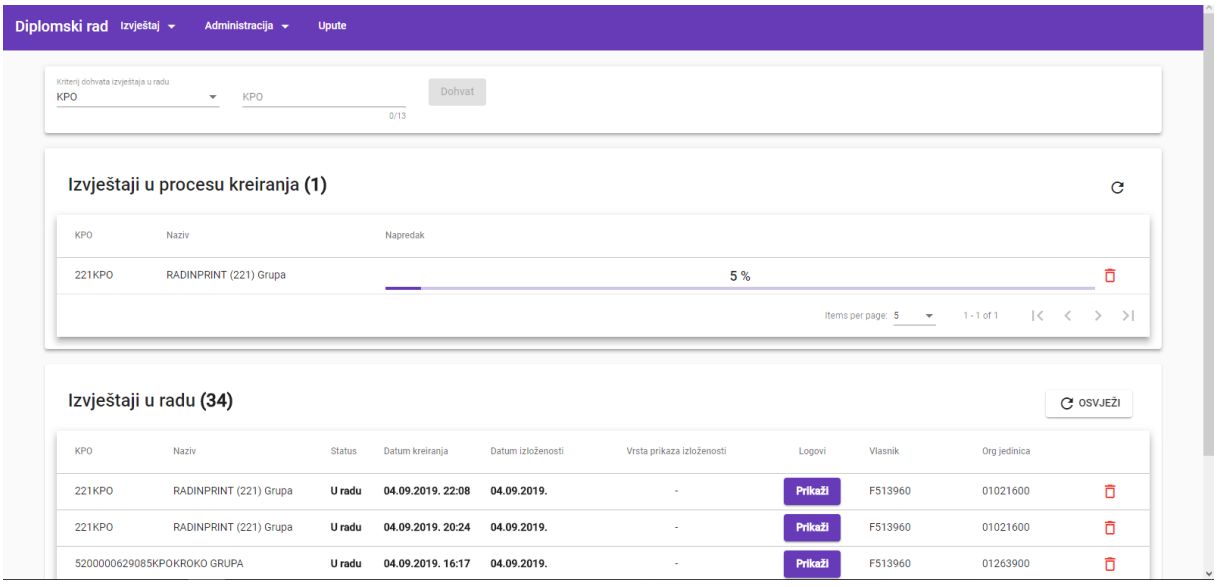
Slika 32: Kreiranje izvještaja – NOVA APLIKACIJA

7.6.2. Pregled izvještaja – početni ekran

Početni ekran aplikacije sastoji se od dva dijela. Na gornjem djelu se mogu vidjeti svi (na koje prijavljeni korisnik ima pravo) izvještaji koji su trenutno u procesu kreiranja, te koji je njihov napredak. U drugom djelu se mogu vidjeti svi najsvježije kreirani izvještaji. Izvještaj se može obrisati ili otvoriti u novom ekranu.



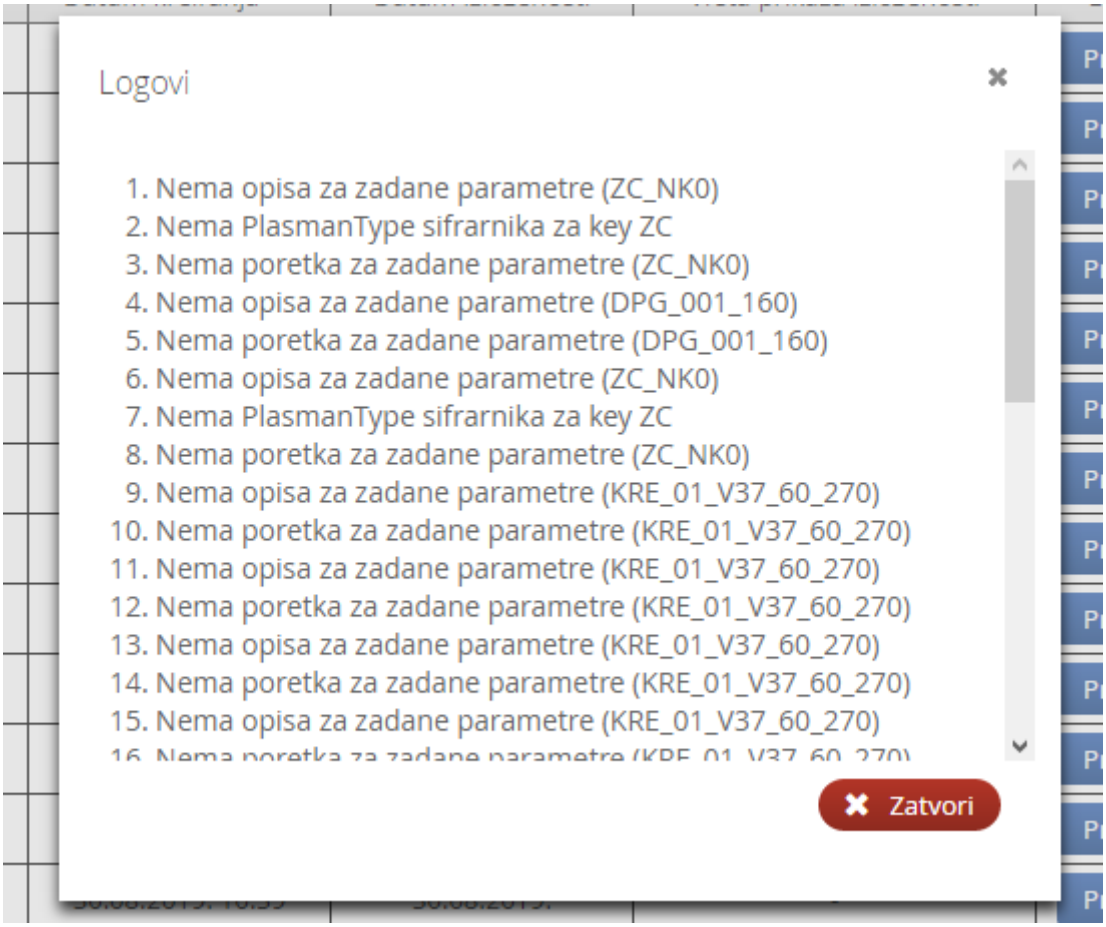
Slika 33: Izvještaji u radu – STARA APLIKACIJA



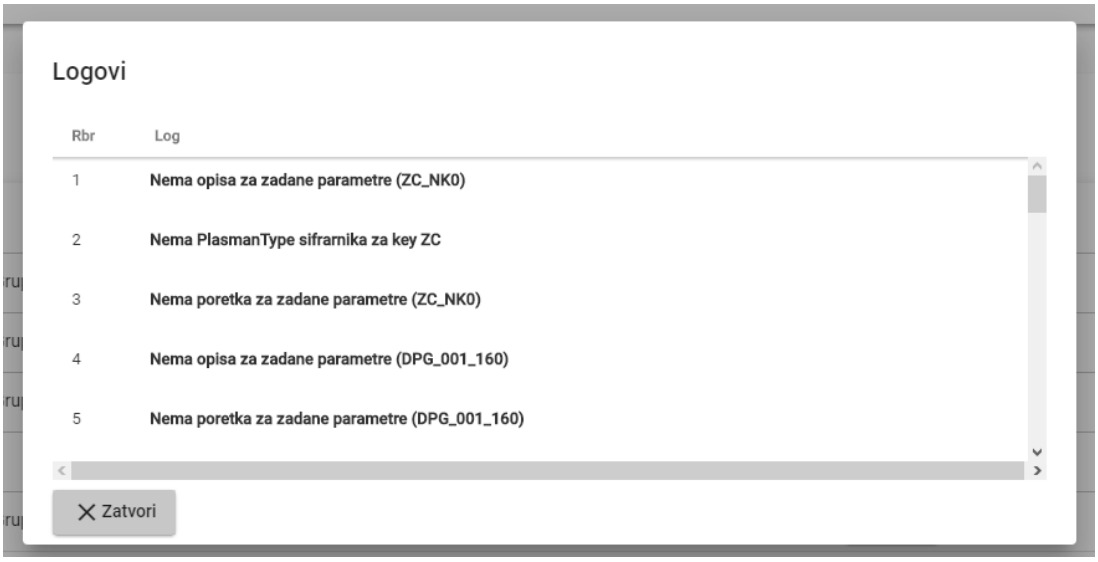
Slika 34: Izvještaji u radu – NOVA APLIKACIJA

7.6.3. Pregled dnevnika zapisa

Ovo je pomoćni dijalog u kojem se mogu vidjeti svi zabilježeni zapisi jednog izvještaja. Zapisi se generiraju tijekom kreiranja izvještaja, odnosno tijekom rada izmjena na istome.



Slika 35: Pregled dnevnika zapisa – STARA APLIKACIJA



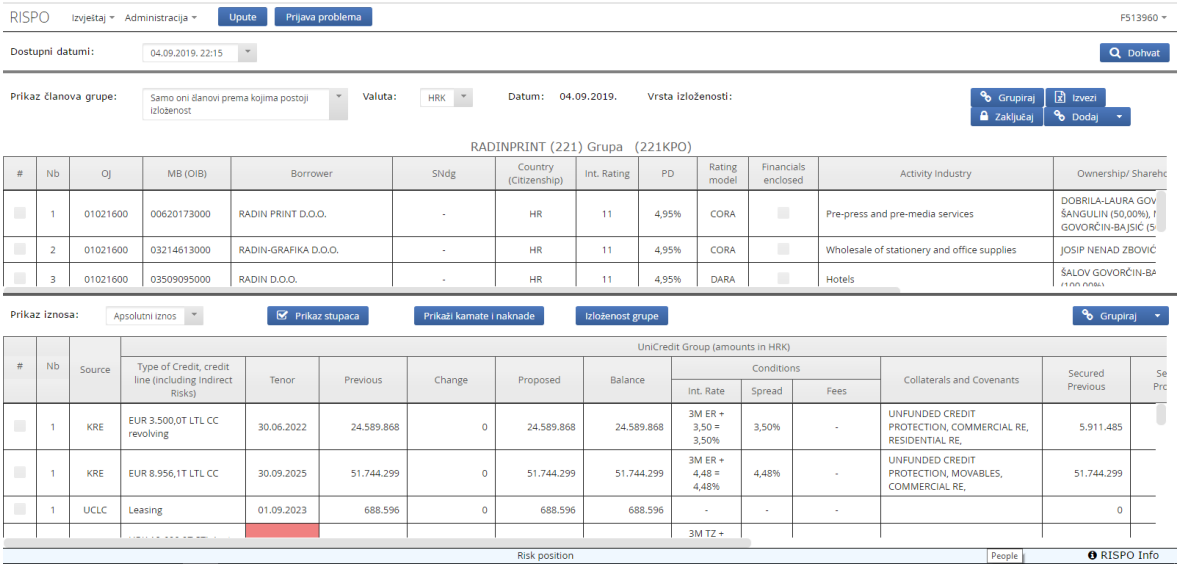
Slika 36: Pregled dnevnika zapisa – NOVA APLIKACIJA

7.6.4. Izvještaj jedne grupe

Ovdje je prikazan glavni, a ujedno i najsloženiji ekran čitave aplikacije. Ekran prikazuje jedan izvještaj jedne grupe. Izvještaj služi za pregled (i izvoz u Excel) ukupne izloženosti prema grupi povezanih klijenata (članovi grupe) na određeni datum. Na vrhu ekrana se nalazi padajući izbornik (Dostupni datumi) sa svim ostalim izvještajima za trenutnu grupu, sortirano prema datumu kreiranja.

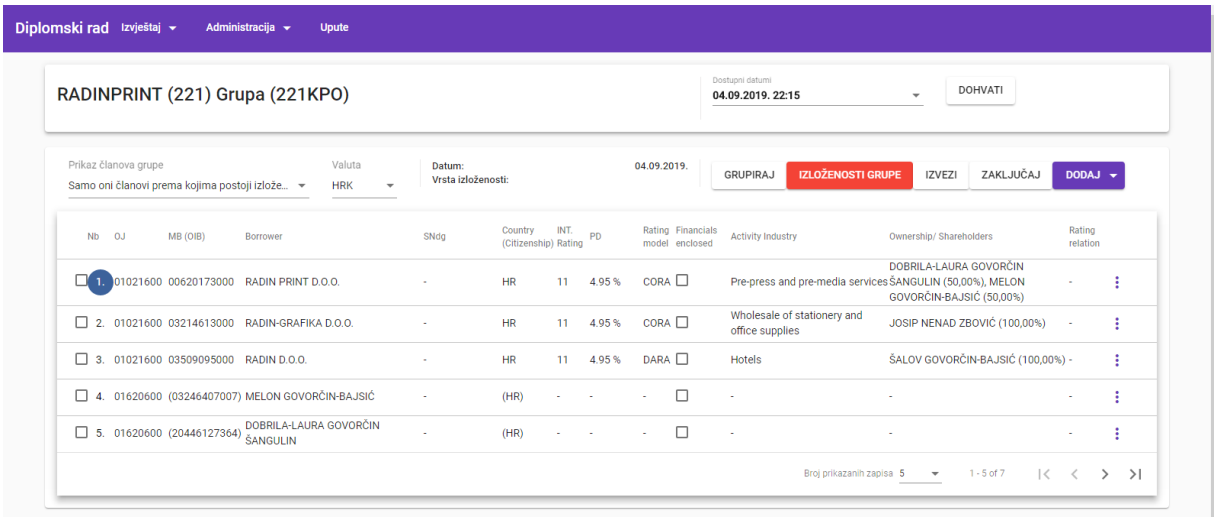
U prvom djelu ekrana nalazi se tablica s popisom članova grupe. Ovdje se mogu raditi razne akcije s članovima (brisanje, uređivanje i dodavanje novih članova, grupiranje članova, dodavanje izloženosti članu)

U drugom djelu ekrana nalazi se tablica s prikazom svih izloženosti svih članova grupe. Izloženosti se mogu pregledavati i grupirati po različitim uvjetima.

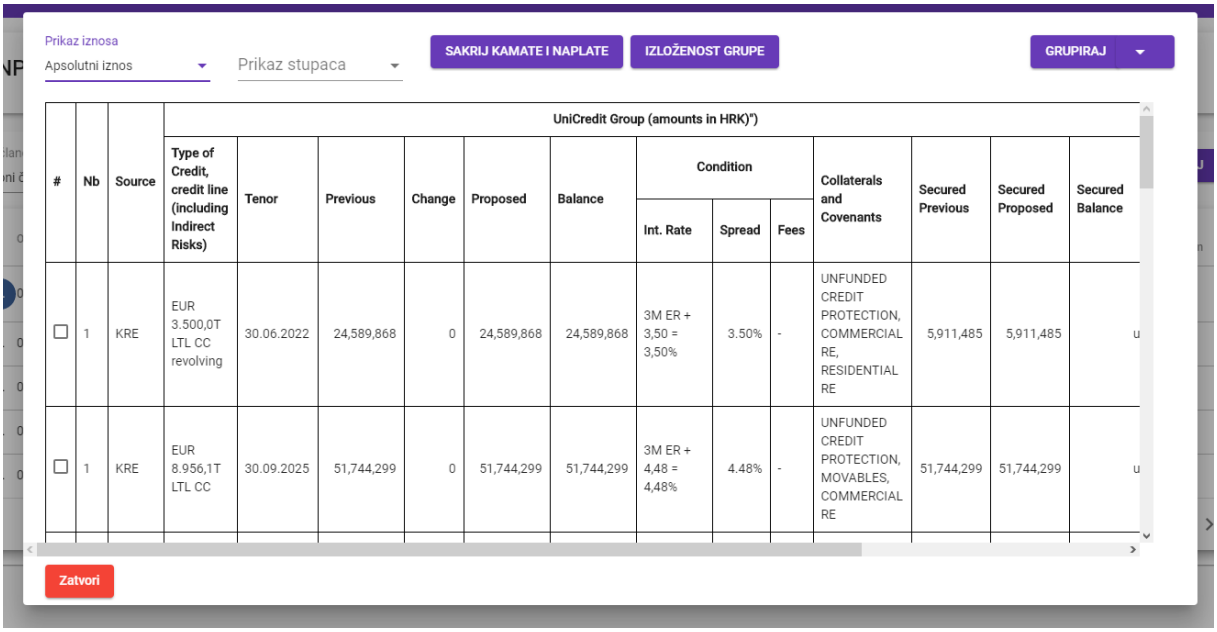


Slika 37: Izvještaj grupe – STARA APLIKACIJA

U novoj aplikaciji su članovi grupe prikazani odmah po dolasku na ekran, dok je za pregled izloženosti grupe je potrebno kliknuti na gumb „IZLOŽENOST GRUPE“. Pregled izloženosti se otvara unutar novog dijaloga.



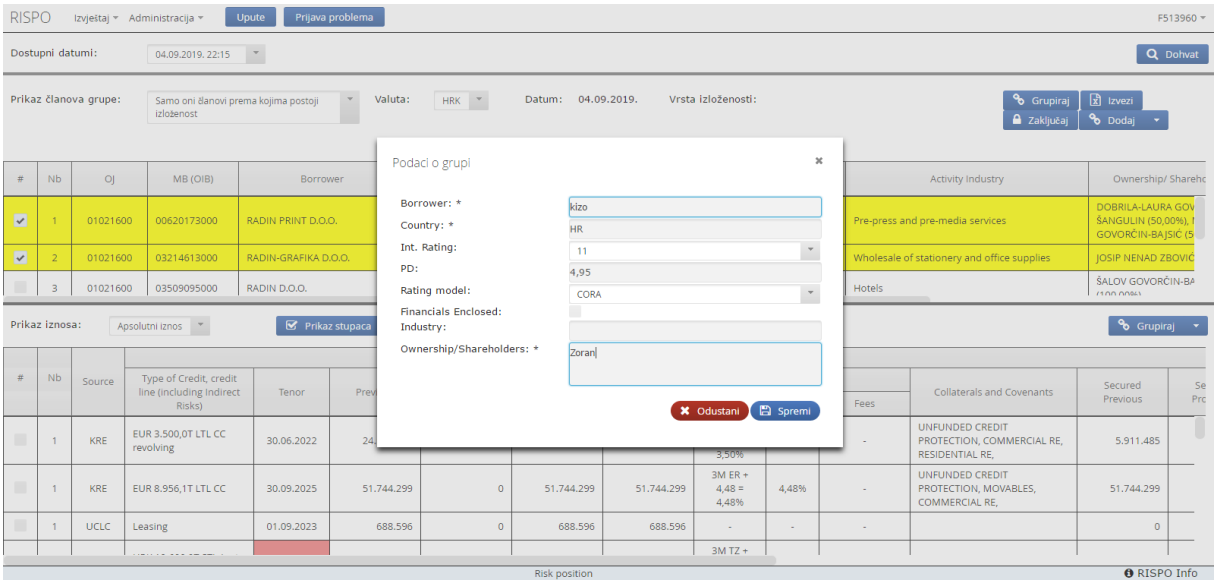
Slika 38: Izvještaj grupe (članovi) – NOVA APLIKACIJA



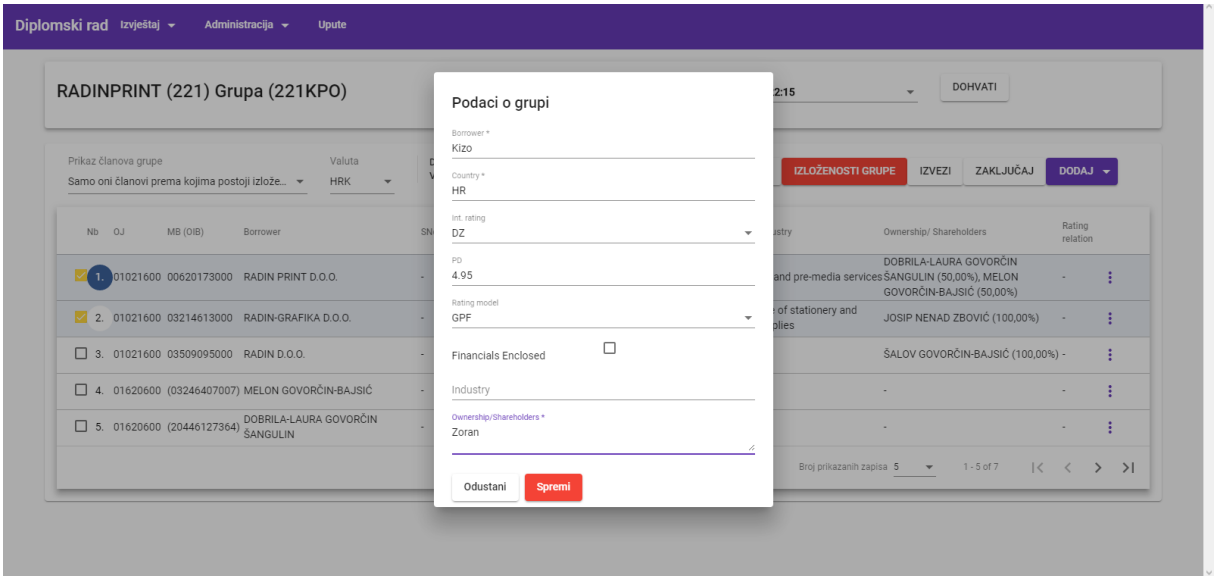
Slika 39: Izvještaj grupe (izloženosti) – NOVA APLIKACIJA

7.6.5. Grupiranje članova grupe

Na sljedećim slikama prikazan je obrazac (*eng. form*) za unos podataka prilikom grupiranja članova grupe.



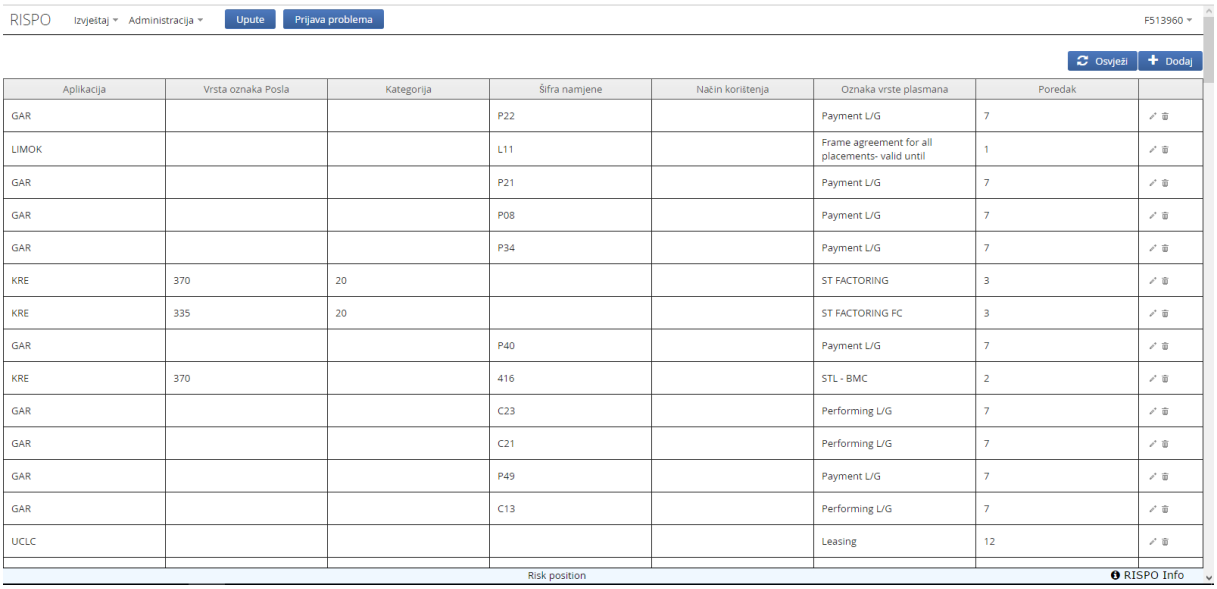
Slika 40: Grupiranje članova grupe – STARA APLIKACIJA



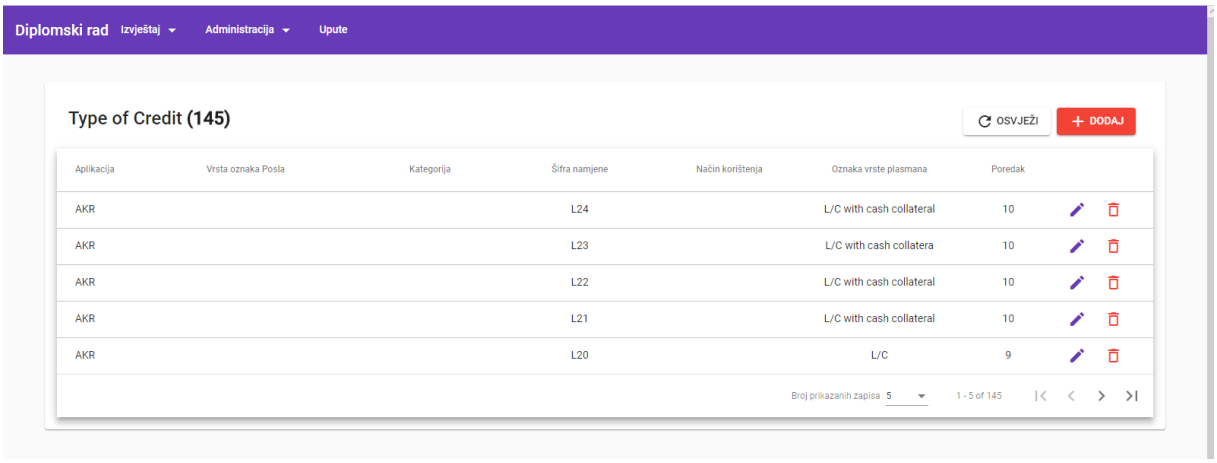
Slika 41: Grupiranje članova grupe – NOVA APLIKACIJA

7.6.6. Šifrarnici

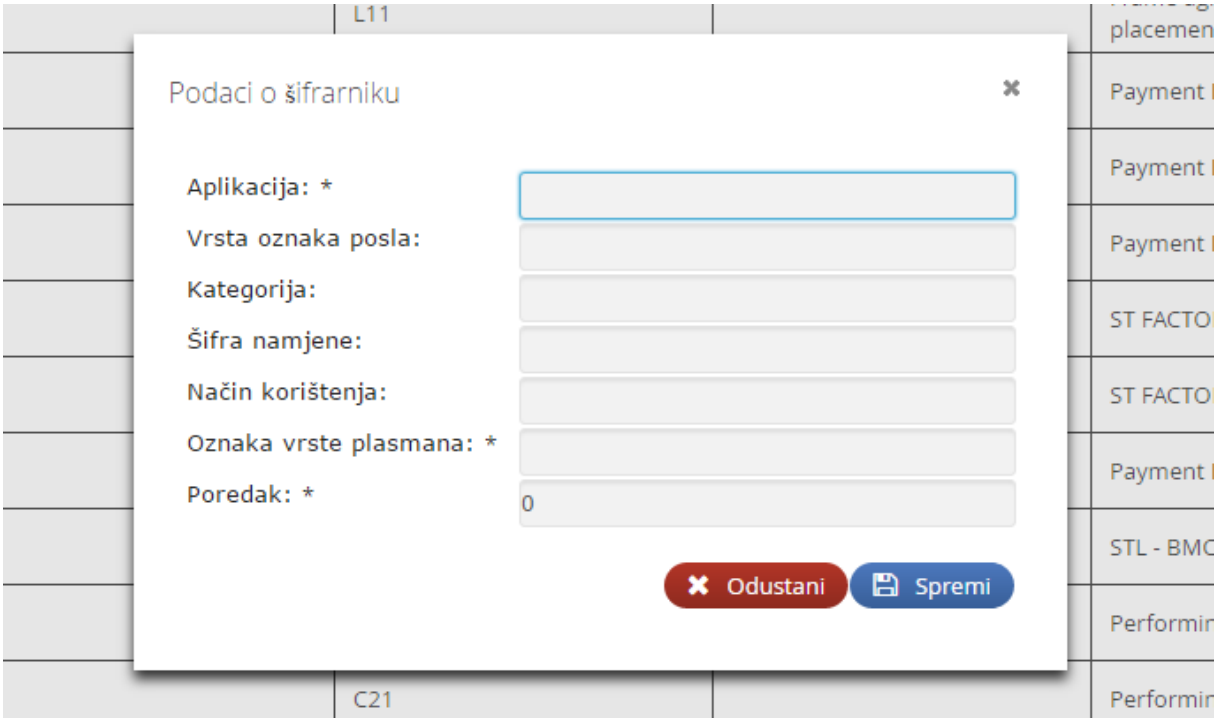
Unutar aplikacije koriste se četiri šifrarnika (Type of Credit, Višejezični šifrarnici, Kamatna stopa, Plasman type). Za tri šifrarnika kreirani su ekrani koji omogućuju kompletnu administraciju, odnosno izvršavanje CRUD (eng. Create, read, update and delete) funkcija, dok je za šifrarnik „Kamatna stopa“ omogućen samo pregled.



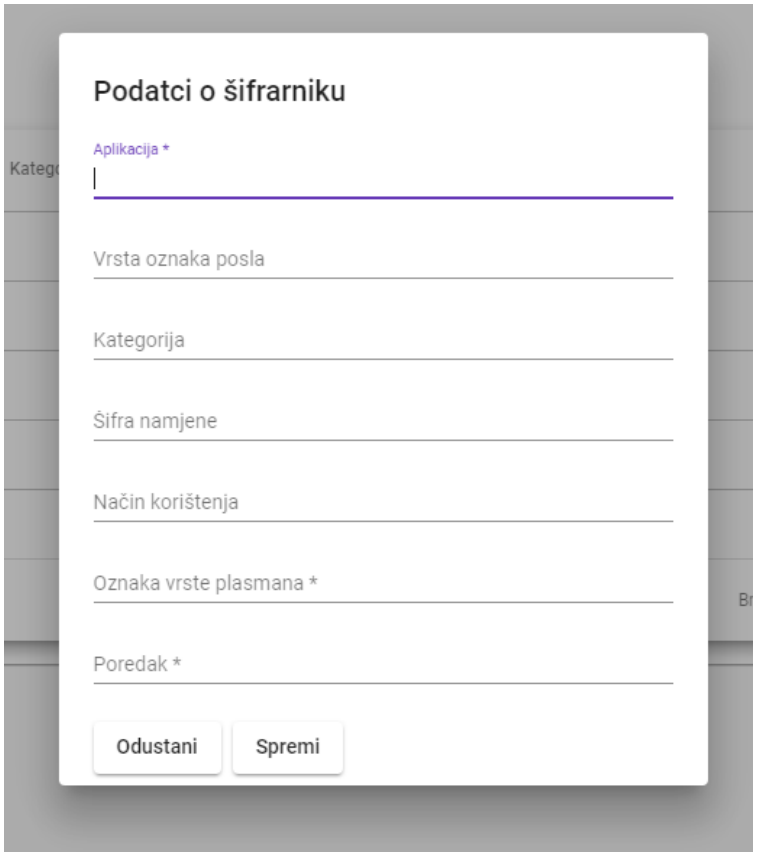
Slika 42: Pregled šifrarnika – STARA APLIKACIJA



Slika 43: Pregled šifrarnika – NOVA APLIKACIJA



Slika 44: Dodavanje/uređivanje šifrarnika – STARA APLIKACIJA



Slika 45: Dodavanje/uređivanje šifrarnika – NOVA APLIKACIJA

8. Zaključak

Na kraju rada na temu „Migracija web aplikacije na novije tehnologije s reaktivnim i asinkronim izvršavanjem“, može se zaključiti kako je cjelokupna migracija protekla vrlo uspješno. Najveći argument za uspjeh migracije je konačna brzina dohvaćanja podataka i brzina rada cjelokupne aplikacije. Brzina učitavanja najsloženijih ekrana je do sedam puta brža u odnosu na učitavanje istih ekrana u staroj aplikaciji. Ovim poboljšanjem će korisnici biti najsretniji. Također nova web aplikacija je sada programirana u najnovijim tehnologijama koristeći suvremeni dizajn. Nova aplikacija je prava WEB 2.0 aplikacija izvedena koristeći najnoviji Angular. Takva web aplikacija je maksimalno podržana od svih suvremenih web preglednika.

Ovom migracijom napravljena su još dva bitna poboljšanja, smanjeno je opterećenje mreže i smanjeno je opterećenje glavnog poslužitelja. Kako je nova aplikacija izvedena kao dvije odvojene aplikacije, korisnička i poslužiteljska, tako se kroz mrežu prenose samo podatci, nema konstantnih prijenosa čitavih HTML stranica. Na taj način je uveliko smanjeno veliko opterećenje mreže. Ovakvim načinom rada poslužiteljska aplikacija više ne mora obavljati nikakve poslove vezane uz kreiranje gotovih ekrana (HTML stranica), te je zadužena samo za obavljanje složene poslovne logike i pristup podatcima. Velik dio posla koji je prije obavljao jedan poslužitelj sada obavlja web preglednik svakog pojedinog korisnika. Time su učinjene i financijske uštede jer su za kvalitetan rad aplikacije dovoljni slabiji poslužitelji, te manja propusnost/brzina mreže.

Prilikom pristupa bilo kakvoj migraciji aplikacije vrlo je bitno razmotriti složenost i tehnologiju postojeće aplikacije. S obzirom na postojeće stanje trebaju se zadati ciljevi koji se žele postići migracijom. Nakon dogovorenih ciljeva odabire se tehnologija za razvoj nove aplikacije. Prilikom kretanja u projekt migracije web aplikacije treba unaprijed razmišljati da je za obaviti ovakav posao potrebno imati ljude koji moraju poznavati mnogo tehnologija. Potrebno je dobro poznavanje svih tehnologija u kojima je izvedena stara aplikacija, a također je potrebno napredno znanje tehnologija u kojima će biti napisana nova aplikacija.

Važno je napomenuti da je migracija između dvije toliko različite tehnologije vrlo složeni proces. Najveći izazov u tome je savladavanje programiranja aplikacije s reaktivnim/asinkronim izvršavanjem. Bez kvalitetnog znanja korištenja RxJS operatora programiranje kompleksnijih stvari je gotovo neizvedivo.

Ovakve, a i složenije, migracije su u velikim informacijskim sustavima gotovo neizbježne. Tehnologija napreduje ogromnom brzinom i da bi web aplikacije pravilno radile i maksimalno iskoristile moć suvremenih web preglednika, vrlo je bitno da su izvedene koristeći najnaprednije i najnovije tehnologije.

Popis literature

[1] Cherny, C. (2019). *Programming TypeScript: Making Your JavaScript Applications Scale*. USA: O'Reilly Media

[2] Clymer, A. (2013). *Pro Asynchronous Programming with .NET*. New York, USA: Apress Media LLC

[3] Daityary, S. (2019). *Angular vs React vs Vue: Which Framework to Choose in 2019* Preuzeto 21.08.2019. s https://www.codeinwp.com/blog/angular-vs-vue-vs-react/

[4] Fain, J., Moiseev, A. (2019) *Angular Development with TypeScript:second edition*. New York, USA: Manning Publications

[5] Google, Angular CLI (bez dat.) *Angular CLI: A command line interface for Angular* Preuzeto 21.08.2019. s https://cli.angular.io/

[6] Google, Angular Docs (bez dat.) *Angular Docs* Preuzeto 13.07.2019. s https://angular.io/docs

[7] Google, Angular Material (bez dat.) *Angular Material Material Design components for Angular* Preuzeto 21.08.2019. s https://material.angular.ioZell, L. (2019). *Dealing with nested callbacks* Preuzeto 13.07.2019. s https://zellwk.com/blog/nested-callbacks/

[8] Hanchett, E., Listwon, B. (2018). *Vue.js in Action*. New York, USA: Manning Publications

[9] Haverbeke, M. (2018). *Eloquent JavaScript-Third edition: A Modern Introduction to Programming*, USA, Creative Commons attribution-noncommercial licence: No Starch Press

[10] Hussain, A. (bez dat.) *Angular: ES6 JavaScript & TypeScript - Promises* Preuzeto 13.07.2019. s https://codecraft.tv/courses/angular/es6-typescript/promises/

[11] Jones, M. (2019). *The Ultimate Guide to Asynchronous Programming in C# and ASP.NET* Preuzeto 13.07.2019. s https://exceptionnotfound.net/asynchronous-programming-in-asp-net-csharp-ultimate-guide/

[12] Koutnik, R. (2018). *Build Reactive Websites with RxJS*. USA:Pragmatic Bookshelf

[13] Mansilla, S. (2018). *Reactive Programming with RyJS 5: Unstangle Your Asynchronous JavaScript Code*. USA:Pragmatic Bookshelf

[14] Meyghani, A. (2018). *Asynchronous JavaScript*, Seattle, Washington, USA: Kindle Direct Publishing

[15] Microsoft (bez dat.) *TypeScript:Documentation* Preuzeto 21.08.2019. s https://www.typescriptlang.org

[16] Nations, D. (2019). *What Does 'Web 2.0' Even Mean?* Preuzeto 21.08.2019. s https://www.lifewire.com/what-is-web-2-0-p2-3486624

[17] Parker, D. (2015). *JavaScript with Promises: Managing asynchronus code*. USA: O'Reilly Media

[18] Pham, W. (2018). *How To Absolutely Beat The Learning Curve Of Angular* Preuzeto 21. 08.2019. s https://medium.com/@vupham2909/beating-the-steep-learning-curve-for-angular-76fed11131e6

[19] PrimeFaces (bez dat.) *Prime User Interface* Preuzeto 21.08.2019. s https://www.primefaces.org

[20] Rylan, C. (2018). *RxJS Observables versus Subjects* Preuzeto 13.07.2019. s https://coryrylan.com/blog/rxjs-observables-versus-subjects

[21] Sikder, R. (2018). *Asynchronous Programming : Why, When and Why Not!* Preuzeto 13.07.2019. s https://medium.com/@rajatsikder/asynchronous-programming-use-cases-86727de31992

[22] Software Bisque (bez dat.) *Synchronous vs. Asynchronous Execution* Pruzeto 13.07.2019. s https://www.bisque.com/products/orchestrate/RASCOMHelp/RASCOM/Synchronous_vs_Asynchronous_Execution.htm

[23] Stringfellow, A. (2017). *When to Use (and Not to Use) Asynchronous Programming* Preuzeto 13.07.2019. s https://stackify.com/when-to-use-asynchronous-programming/

[24] Technopedia (bez dat.) *Web 2.0* Preuzeto 21.08.2019. s https://www.techopedia.com/definition/4922/web-20

[25] Tijms, A., Schlotz, B. (2018). *The Definitive Guide to JSF in Java EE 8: Building Web Applications with JavaServer Faces*, New York, USA: Apress Media LLC

[26] Troncone, B. (2019). *Learn RxJS* Preuzeto 13.07.2019. s https://www.learnrxjs.io/

[27] Tutorials Point (bez dat.) *Learn JSF* Preuzeto 21.08.2019. s https://www.tutorialspoint.com/jsf/jsf_architecture.htm

[28] Tutorials Point (bez dat.) *Learn TypeScript* Preuzeto 21.08.2019. s https://www.tutorialspoint.com/typescript/typescript_types.htm

[29] Tyson,M. (bez dat.) *What is JSF? Introducing JavaServer Faces*, Preuzeto 21.08.2019. s https://www.javaworld.com/article/3322533/what-is-jsf-introducing-javaserver-faces.html

[30] Vaughn, B. (2019). *React, A JavaScript library for building user interfaces* Preuzeto 21.08.2019. s https://reactjs.org/

[31] Walls, C. (2015) *Spring Boot in Action*. New York, USA: Manning Publications

[32] Wu, W. R. (2016). *William's Computer Science* Preuzeto 13.07.2019. s https://williamswu.wordpress.com

[33] You, E., Sajnog, M., Tepluhina, N., Yerburgh, E. (2019) *The Progressive JavaScript Framework* Preuzeto 21.08.2019. s https://vuejs.org/

[34] Zell, L. (2019). *Dealing with nested callbacks* Preuzeto 13.07.2019. s https://zellwk.com/blog/nested-callbacks/

Popis slika

Popis tablica