

MATEMATIČKI FAKULTET
UNIVERZITET U BEOGRADU

MASTER RAD

*PRIMENA PROGRAMSKOG JEZIKA SCALA U
RAZVOJU VEB APLIKACIJA*

mentor: prof. dr Vladimir Filipović

Student: Aleksandar Četković

Sadržaj

1. Uvod	3
1.1 Istorijski razvoj.....	3
1.2 Upoređivanje veb i klasičnih aplikacija.....	3
2. Programiranje veb aplikacija	5
2.1 Razvojna okruženja	5
2.2 Arhitektura veb aplikacija.....	7
2.3 Veb aplikacije i pametni telefoni.....	8
2.4 Problemi pri razvoju i održavanju veb aplikacija.....	8
3. Opis funkcionalnosti razvijene veb aplikacije	10
3.1 Administratorski režim	10
3.2 Režim zaposlenog.....	14
3.2 Potrošački režim	16
4. Programski jezik Scala	19
4.1 Osnovni koncepti Scala programskog jezika	19
4.2 Upoznavanje sa Play framework-om.....	24
4.3 Kontroleri i forme.....	26
4.4 Rad sa stranicama	29
4.5 Rad sa bazom podataka	38
4.6 Čitači i pisaci.....	43
4.7 REST servisi	44
4.8 Asinhronne akcije.....	46
4.9 Osnovno o Akka-i	47
4.10 Veb soketi	50
4.11 Testovi jedinica	52
4.12 Scala i Java.....	54
4.13 Prednosti i mane programskog jezika Scala	56
5. Implementacija aplikacije OAF	58
5.1 Arhitektura aplikacije OAF.....	58
5.2 Projekat OAF	58
6. Zaključak	61
Literatura	62
Spisak slika	63

1. Uvod

U svetu računara, veb aplikacija se može opisati kao klijent-server aplikacija u kojoj se klijent izvršava u okviru veb pregledača (odnosno, korisnički interfejs se nalazi u okviru veb pregledača). Primeri složenijih veb aplikacija su: kupovina preko mreže, aukcije, instant poruke i mnogo drugih. Nije potpuno čista razlika između veb sajta i veb aplikacije. Moglo bi se reći da su veb sajtovi više statički i koriste se za neku jednostavniju prezentaciju. Međutim, u današnje vreme razvijeno je dosta alata koji veb sajtovima omogućavaju posedovanje naprednih funkcionalnosti. Bez obzira na to, veb aplikacije pružaju mnogo više fleksibilnosti i mnogo bolje rešavaju konkretne zahteve. U ovom radu je opisano i na primeru razvoja softvera otvorenog koda pokazano, kako se programski jezik Scala može koristiti u razvoju veb aplikacija.

Comment [VF1]: Странице би требало да имају бројеве страна

1.1 Istorijски razvoj

Comment [VF2]: 1.1

Preteča veb aplikacije je bila klijent-server aplikacija koja je imala sledeće karakteristike:

1. Klijentski kod morao je biti kompajliran i instaliran na svakom terminalu.
2. Svaka klijentska aplikacija morala je biti implementirana na operativnom sistemu koji je instaliran na klijentu.

Ovakav pristup je bio jako skup i težak za održavanje. Tim Berners-Lee je 1990. godine napravio infrastrukturu za veb, koja uključuje protokol HTTP, jezik HTML i prvi veb pregledač. Pojavom pregledača, veb aplikacije su doživele svoju ekspanziju. U početku su veb aplikacije pružale klijentima samo statičke HTML stranice. Svaki put kada bi klijent napravio neku izmenu na stranici, poziv je mora ići do servera.

Godina 1995. je bila krucijalna za internet. Kompanija Netscape je predstavila JavaScript, skript jezik koji se koristi na klijentskoj strani. Pojavom JavaScript-a, programerima je omogućeno da unaprede klijentski interfejs ubacivanjem dinamičkih elemenata. JavaScript je učinio da korisnicima rad na internetu bude brži i mnogo produktivniji, zato što nije više bilo potrebno da se pri svakoj promeni ide do serverske strane.

Godine 1996. kreiran je Flash od strane kompanije Macromedia. Flash je omogućio korišćenje vektorskih animacija na veb stranicama. Ovaj proizvod učinio je internet "svetlijim" i interaktivnijim.

Koncept veb aplikacije je 1999. godine uveden u Java jezik u obliku Servleta.

Godine 2005. dolazi do pojave Ajax-a, koji predstavlja asinhroni poziv ka serveru, napisan u JavaScript jeziku. Ajax je učinio veb aplikacije osetno bržim i moglo bi se reći da je ovo dovelo do dodatne ekspanzije veb aplikacija.

1.2 Upoređivanje veb i klasičnih aplikacija

Jedna od glavnih prednosti veb aplikacija u odnosu na klasične je činjenica da se veb aplikacija izvršava u pregledaču, što znači da je nezavisna od operativnog sistema.

Ubrzavanjem interneta i unapređenjem veb pregledača dobila se mogućnost da se skoro svaka klasična aplikacija realizuje kao veb aplikacija. Naravno, brzina izvršavanja je i dalje na strani klasičnih aplikacija, ali vremenom se i ta razlika smanjuje.

Prednosti veb aplikacija u odnosu na desktop:

1. Nema instalacije – Svi stariji korisnici PC računara pamte koliko je mukotrpno bilo instalirati sve aplikacije na novom računaru (video plejer, muzički plejer, programi za

obradu teksta). Ovo takođe oduzima i dosta vremena. Veb aplikacija ne zahteva instalaciju, ona zahteva samo posedovanje veb pregledača i eventualnu registraciju.

2. Lakše održavanje – Kada neka kompanija izdaje novu verziju svoje veb aplikacije, dovoljno je samo da promeni kod na serveru i nova verzija je dostupna svim korisnicima. Kod desktop aplikacija je potrebno da se svakom korisniku pošalje nova instalacija, pa se još mogu javiti problemi sa usklađivanjem podataka.

3. Sistem u oblaku – Neke kompanije pružaju korišćenje memorijskog resursa na internetu. To znači da su sve lične datoteke dostupne korisniku u svakom trenutku i sa bilo kog uređaja (taj uređaj ne mora biti u vlasništvu korisnika).

4. Nezavisnost od operativnog sistema – Pri implementaciji neke aplikacije, kompanija nema potrebe da vodi računa o razlikama operativnih sistema.

Mane veb aplikacija u odnosu na desktop:

1. Bezbednost i sigurnost – Ovo je najveći problem veb aplikacija. Sama činjenica da su podaci na internetu i da su dostupni sa bilo kog uređaja otvara mogućnost za razne vidove kriminala i zloupotrebe.

2. Zloupotreba naloga – Najčešće je slučaj sa aplikacijama za naručivanje robe ili plaćanje. Mada se može desiti i na društvenim mrežama, treće lice može obavljati aktivnosti u ime korisnika.

3. Zavisnost od internet mreže – Kada korisnik nema pristup internetu – ne može pristupiti ni jednoj aplikaciji, što nije slučaj sa desktop aplikacijama.

2. Programiranje veb aplikacija

Razvoj veb aplikacija je specifičan. Tu se jako često koristi metoda agilnog razvoja softvera. Agilna metoda razvoja softvera podrazumeva deljenje vremena predviđenog za razvoj u jednake intervale (iteracije) i u svakoj iteraciji potrebno je implementirati funkcionalnosti koje se mogu koristiti. Dakle, na kraju svake iteracije potrebno je da se aplikacija uredno prevodi, izvršava bez grešaka i da su uspešno realizovane funkcionalnosti planirane za tu iteraciju. Prva iteracija je obično duža i u njoj se postavlja osnova softvera koja se u daljim iteracijama proširuje novim funkcionalnostima. Često se uvodi tzv. nulta iteracija, ona se dešava pre početka implementacije. U nultoj iteraciji tim programera i analitičara odlučuje o sledećim stvarima:

- Odabir tehnologija za razvoj
- Arhitektura aplikacije
- Modeliranje baze podataka
- Određivanje redosleda implementacije funkcionalnosti

Naravno, ovo je samo primer radnji koje se obavljaju u nultoj iteraciji. Sama agilna metodologija je jako fleksibilna i u praksi je timovi prilagođavaju sebi. Inače, u praksi se pokazalo da su nulta i prva iteracija najbitnije za dalji razvoj aplikacije. Na primer, ako je loše postavljena arhitektura ili je baza loše modelirana, tada svaka sledeća iteracija snosi posledice u vidu sporije implementacije ili, još gore, nemogućnosti razvoja neke funkcionalnosti. U daljem tekstu su obrazloženi kriterijumi za odabir razvojnog okruženja, kao i opis arhitekture veb aplikacije koja može biti korišćena od strane pametnih telefona.

2.1 Razvojna okruženja

Veb aplikacije zbog svoje slojevitosti koriste više kombinovanih tehnologija. Naime, svaka veb aplikacija ima najmanje tri sloja. Najčešće se koristi MVC obrazac (Model-View-Controller). Za razvoj prezentacionog sloja (eng. View) uglavnom se koriste HTML i JavaScript. Za razvoj kontroler sloja uglavnom se koriste neki od kompleksnijih programskih jezika (C#, Java, Scala, Python, itd.). Za sloj modela koriste se razne relacije baze ili NO-SQL baze. Neki primeri su: *MySQL*, *MS SQL*, *Postgres SQL*, *MongoDB*, itd.

Kriterijumi za odabir razvojnog okruženja:

1. Namena veb aplikacije – Aplikacija koju je potrebno razvijati postavlja određeni skup kriterijuma. Naime, poznavanjem namene veb aplikacije može se preciznije odrediti:

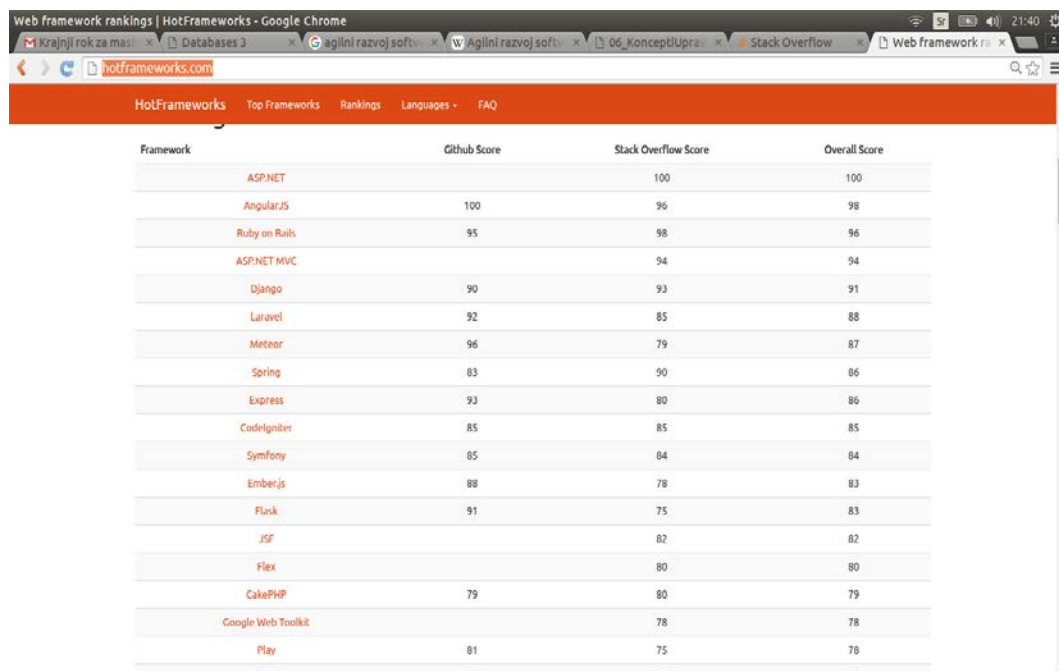
- o Veličina baze - Neke aplikacije koje čuvaju slike i video zahtevaju korišćenje neklasičnih baza podataka (eng. Big Data)
- o Efikasnost – Brzina je bitna kod aplikacija za grafičku obradu ili praćenje brzih promena na berzi.
- o Bezbednost – Ovo se uzima u obzir kod aplikacija koje koriste sisteme za plaćanje preko interneta ili kod onih koje sadrže poverljive informacije.
- o Proširivost – Ako aplikacija ima tendenciju za dodavanje novih funkcionalnosti, onda treba uzeti i ovaj parametar u obzir.
- o Kompatibilnost sa pametnim telefonima – Ako se aplikaciji pristupa i sa pametnih telefona, onda najčešće treba pametno odabrati tehnologije za razvoj prezentacionog sloja. Problemi se javljaju kod renderovanja stranica na različitim

veličinama ekrana (eng. Responsive Design).

2. Obuka programera – Treba uzeti u obzir koliko je tim programera koji razvija aplikaciju upoznat sa tehnologijom, ali takođe potrebno je uzeti u obzir koliko je vremena potrebno programeru koji nije upoznat sa tehnologijom da je savlada. Naime, može doći do potrebe za proširivanjem tima ili zamenom nekog člana, pa je onda potrebno da novi član savlada obuku za tehnologije koje se koriste u razvoju aplikacije. Ovde se može reći da je sama kompleksnost razvojnog okruženja jedan od faktora.

3. Cena – Postoji dosta besplatnih tehnologija za razvoj veb aplikacija, ali se i dalje smatra da su one manje pouzdane od onih koje se plaćaju (mada poslednjih godina ima velikih pomaka i promena i po ovom pitanju).

4. Popularnost – Ovo je jako bitan faktor. Od popularnosti tehnologije zavisi, između ostalog, obim i kvalitet dokumentacije koja je dostupna na internetu. Postoji veliki broj sajtova na kojima programeri postavljaju pitanja drugim programerima i dobijaju odgovore. Ovo je jako korisno, zato što se često dolazi do specifičnog problema koji je vezan za razvojno okruženje i odgovor na to pitanje se teško može naći u zvaničnoj dokumentaciji. Jedan od takvih sajtova je <http://stackoverflow.com/>. Postoje i sajtovi koji govore o popularnosti razvojnih okruženja, jedan od takvih sajtova je <http://hotframeworks.com/>. Github Score se odnosi na okruženja sa otvorenim kodom (eng. Open Source) i predstavlja koliko ima onih koji razvijaju dodatne funkcionalnosti. Stack Overflow Score predstavlja koliko su aktivni programeri na sajtu koji je opisan iznad. (Slika 1)

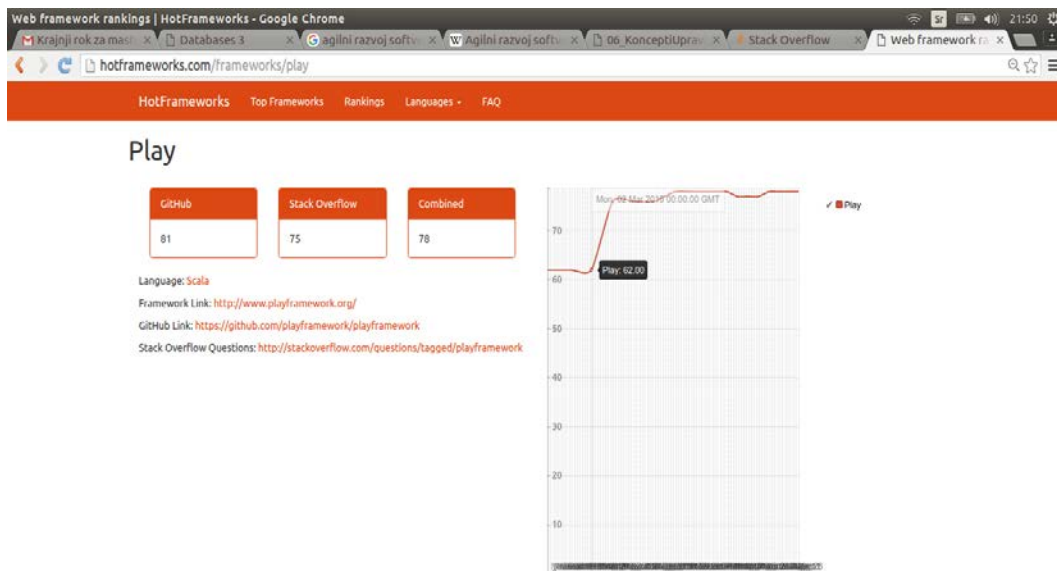


Framework	Github Score	Stack Overflow Score	Overall Score
ASP.NET		100	100
AngularJS	100	96	98
Ruby on Rails	95	98	96
ASP.NET MVC		94	94
Django	90	93	91
Laravel	92	85	88
Meteor	96	79	87
Spring	83	90	86
Express	93	80	86
CodeIgniter	85	85	85
Symfony	85	84	84
Ember.js	88	78	83
Flask	91	75	83
JSF		82	82
Flex		80	80
CakePHP	79	80	79
Google Web Toolkit		78	78
Play	81	75	78

Slika 1. Trenutnu popularnost okruženja za razvoj veb aplikacija.

Ovde se može primetiti da je *Play* okruženje, koje je obrađeno u ovom radu, među prvih 20 okruženja, što i nije loše pošto je konkurencija velika. Ovaj sajt pruža i dodatnu

moгуćnost, a to je grafik popularnosti kroz vreme. Sledi primer za *Play* okruženje (Slika 2).



Slika 2. Popularnost *Play* okruženja

2.2 Arhitektura veb aplikacija

Comment [VF3]: veb

Arhitektura veb aplikacije je jako bitna. Uglavnom se koriste različite varijacije MVC obrasca. Ono čemu se teži pri razvoju veb aplikacija, a često i ostalih aplikacija je slojevitost. Naime, ideja je da se aplikacija podeli u više nezavisnih slojeva, a cilj je da ti slojevi budu lako proširivi. Kada programer razvija aplikaciju mora biti svestan da u budućnosti ona ima tendenciju ka proširivanju i zato je jako bitno dobro postaviti arhitekturu projekta.

Postoje razne arhitekture, ali u ovom poglavlju treba izdvojiti arhitekturu mikroservisa. Globalna ideja ove arhitekture je razdvajanje cele aplikacije na više aplikacija koje obavljaju neke funkcionalnosti i komuniciraju međusobno.

U tekstu koji sledi opisan je jedan primera sa mikroservisima:

Neka postoji glavna veb aplikacija koja nema pristup bazi, nego taj pristup ostvaruje preko servisa koji je zadužen za rad sa bazom. Dakle, sva logika vezana za dohvaćanje, upisivanje, promenu entiteta je u servisu, a glavna aplikacija samo šalje pozive ka tom servisu (može se koristiti REST za komunikaciju, ili neki drugi protokol). Ovakav pristup je jako dobar iz više razloga, na primer, ako kompanija reši da razvija novu veb aplikaciju koja će koristiti istu bazu, potrebno je samo razviti glavnu aplikaciju i iskoristiti funkcionalnosti postojećeg servisa. Mikroservisi poboljšavaju skalabilnost cele ovako definisane aplikacije. Na primer, ako iz aplikacije izdvojimo mikroservis za registraciju korisnika, imajući u vidu da se ova funkcionalnost ne koristi toliko često koliko neke ostale u sistemu.

Pod skalabilnošću aplikacije smatramo njenu sposobnost da na povećan broj zahteva klijenata odgovara istom brzinom. To se postiže dodavanjem novih servera na kojima se

izvršava veb aplikacija. Pošto je aplikacija podeljena u mikroservise, svaki mikroservis možemo proširiti različitim brojem servera. Konkretno u ovom našem primeru registracija korisnika je na dva servera, a ostatak aplikacije na deset. Na ovaj način je napravljena bolja skalabilnost, pošto je funkcionalnost koja se manje koristi skalirana na dva servera, a ostatak aplikacije na deset. Na ovaj način se uštedi dosta novca za zakup ili kupovinu servera, a i sama skalabilnost se može primeniti na uska grla aplikacije.

2.3 Veb aplikacije i pametni telefoni

Postoje dva pristupa veb aplikaciji preko pametnih telefona. Jedan način je preko pregledača na telefonu, a drugi način je preko prirodne aplikacije (eng. Native Application). Prirodna aplikacija predstavlja program napisan na odgovarajućem jeziku koji se u kompiliranoj verziji pokreće na platformi telefona i predstavlja klijentsku verziju aplikacije koja pristupa veb serveru. Ovakav pristup podseća na prve korake u razvoju veb aplikacija koje su navedene u delu 1. Veb aplikacije koje predstavljaju serversku stranu u komunikaciji sa pametnim telefonima nazivaju se *API-serveri*. *API-server* je veb aplikacija koja prima zahtev, obrađuje ga i vraća odgovor klijentskoj aplikaciji. U novije vreme najčešće se koristi JSON format, zbog svoje kompatibilnosti sa telefonima. Naravno, ovakve veb aplikacije nemaju svoj prezentacioni sloj, ali se može implementirati korišćenjem Javaskripta i HTML-a. Taj prezentacioni sloj bi predstavljao još jednu klijentsku verziju aplikacije. Ovakav pristup je dobar, pošto se sva logika aplikacije nalazi na veb serveru, a ostale klijentske aplikacije, na pametnim telefonima ili vebu, samo koriste usluge.

2.4 Problemi pri razvoju i održavanju veb aplikacija

Sa programerskog aspekta ovo je jedna od najbitnijih stvari i njoj se posvećuje dosta vremena u novije vreme. Programeri troše dosta vremena istražujući razne tehnike samog kodiranja u cilju lakšeg održavanja aplikacije. Da bi aplikacija bila održiva i podložna promenama i razvoju novih funkcionalnosti potrebno je ispoštovati sledeće stvari:

- **Postaviti odgovarajuću arhitekturu aplikacije**
- **Pisati "čitljiv" kod**
- **Konzistentost aplikacije pri dodavanju novih funkcionalnosti**
- **Dobro modelovati bazu**
- **Koristiti dizajn paterne za rešavanje poznatih problema**

-O arhitekturi je bilo reči u delu 2.2.

- ***Pisati "čitljiv" kod***

Zbog ovih stvari programeri uvode razne konvencije i propise. *"Svako može kodirati, ali retko ko može dobro kodirati."* Ovo je jedna rečenica koja se provlači kroz radove koji su bazirani na pisanju dobrog koda. I baš ova stvar pravi razliku između dobrog i lošeg programera. Na primer, ako programer ne napiše čitljivo neki deo aplikacije, sledećem programeru je potrebno dosta vremena da shvati kod i uradi izmenu, u nekim situacijama je to i nemoguće. Ovde treba izdvojiti knjige (udzbenike) koje se bave problemom čitljivog koda:

- *"Clean Code: Agile Software Craftsmanship", autor: Robert C. Martin*
- *"The Pragmatic Programmer: From Journeyman to Master", autori: Andrew Hunt, David Thomas*

Na sam kvalitet koda utiče i editor koje se koristi. Na primer, *JetBrains* kompanija je razvila dodatak koji vrši proveru dobro napisanog koda u *C#*. Ovaj dodatak se naziva *ReSharper* i on radi kao predkompajler u *Visual Studio*-u. Naime njegova namena je da prepozna loše napisan kod i upozori programera tako što podvlači kod, a može se i podesiti da ne dozvoli kompajliranje. Na kvalitet koda utiče i sam programski jezik. *Scala* kao funkcionalan jezik bazira se na rekurziji umesto iteracije. Primeri koji ilustruju manji obim kodiranja u *Skali* su izloženi u delu 4.12.

- Konzistentost aplikacije pri dodavanju novih funkcionalnosti

Kada se radi na velikim projektima, sam programer nije upoznat sa celokupnim kodom aplikacije. Iz tog razloga, pri razvijanju nove funkcionalnosti, može doći do pojave greške u postojećoj funkcionalnosti. Najčešće se ovo dešava zbog bočnog efekta (eng. Side Effect) neke implementirane funkcije. Ovo je jako opasna greška, a razlog tome je što je nju teško uočiti i najčešće se ona uoči u produkcionom režimu rada. Pod produkcionim režimom rada smatra se gotov proizvod koji koriste klijenti. Razlog zašto se greška pojavljuje tek u produkciji leži u sledećem:

Programer razvija funkcionalnost A, a izazvao je grešku na postojećoj funkcionalnosti B. Nakon završetka implementacije, testeri testiraju aplikaciju i obično testiraju samo novu funkcionalnost u ovom slučaju funkcionalnost A. Testovi su prošli i aplikacija je spremna za produkciju (prodaju). Ostala je greška u funkcionalnosti B i ona se ispoljava kod klijenta, što u nekim slučajevima može naneti ogroman trošak klijentu. U tekstu koji sledi dati su neki od načina rešavanja ovog problema:

- **Regresivno testiranje (eng. Regression testing)** – Izdvoji se skup funkcionalnosti aplikacije koji mora raditi konzistentno u svakom trenutku. Formiraju se planovi testiranja ovih funkcionalnosti koje izvršavaju testeri. Regresivno testiranje se obavlja nakon testiranja novih funkcionalnosti. Na ovaj način se može prepoznati greška u B funkcionalnosti navedenog primera. Međutim, ovaj metod ima dosta mana:

- Ručno se testira, što opet može dovesti do greške (ljudski faktor)
- Previše vremena se troši
- Test planovi se menjaju tokom vremena

- **Testiranje jedinica (eng. Unit testing)** – Za svaku funkcionalnost mora postojati test. Testovi se kodiraju i predstavljaju deo koda. U samom testu se navode ulazni podaci i oni mogu biti specifični u cilju boljeg testiranja graničnih slučajeva. Princip se sastoji u tome da se pre startovanja aplikacije startuju testovi i ako su svi testovi zadovoljeni, aplikacija se može pokrenuti. Na ovaj način se proveravaju sve postojeće funkcionalnosti i time sprečava pojava greške u njima. *Scala* poseduje mehanizam automatskog testiranja i on je detaljno opisan u delu 4.11. Testiranje jedinica je dovelo do pojave nove metode razvoja softvera, to je *Razvoj Vođen Testovima (eng. Test Driven Development)*. U ovoj metodi se prvo razvijaja test, a nakon toga se razvija funkcionalnost koja mora zadovoljiti taj test.

3. Opis funkcionalnosti razvijene veb aplikacije

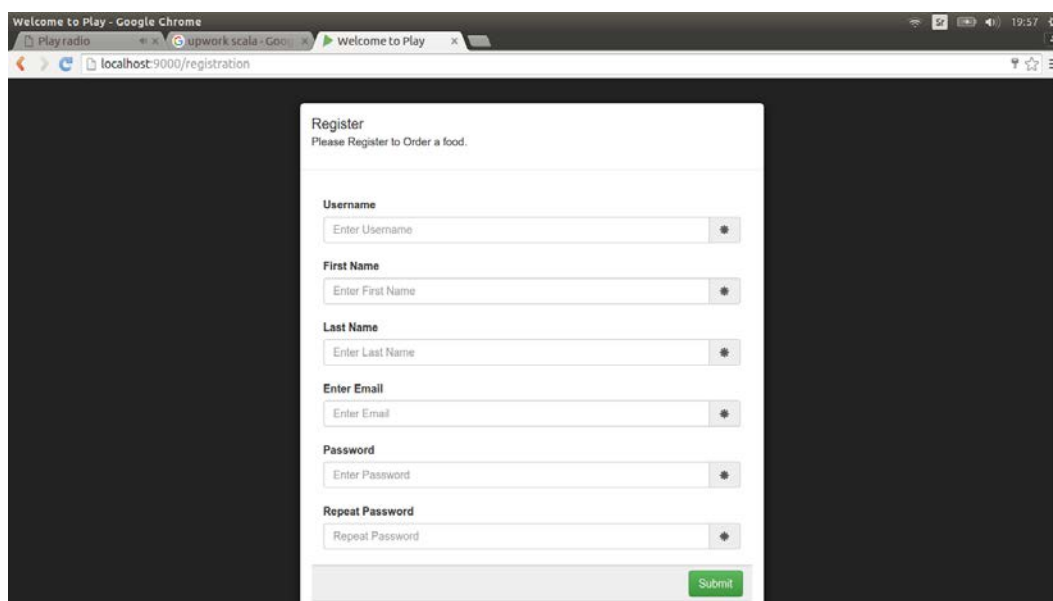
Osnovni konceptio ovog rada su opisani kroz aplikaciju za naručivanje hrane, čije ime je OAF (eng. Order a Food). Aplikacija OAF je implementirana u Play & Scala okruženju. Aplikacija ima tri režima rada:

- Administratorski režim
- Režim zaposlenog
- Potrošački režim

Postoje tri vrste korisnika, koji odgovaraju gore navedenim režimima rada, to su administrator, zaposleni i potrošač. Aplikacija zahteva od administratora i zaposlenog prijavljivanje. Sledi detaljan opis svakog od režima rada. Celokupan kod aplikacije je dostupan na Git repozitorijumu <https://github.com/CetkoS/OAF-UI>.

3.1 Administratorski režim

Pre nego što korisnik uđe u administratorski režim rada, potrebno je da napravi nalog. Stranica za otvaranje naloga je na OAF-APP-URL/registration. OAF-APP-URL je adresa servera na kome je aplikacija. Pri registraciji, korisnik unosi standardne podatke i šifru. (Slika 3) Nakon validacije korisnik postaje administrator i može da unese podatke o svojoj radnji.

A screenshot of a web browser showing a registration form. The browser's address bar displays 'localhost:9000/registration'. The form is titled 'Register' with the subtitle 'Please Register to Order a food.' It contains several input fields: 'Username' (placeholder 'Enter Username'), 'First Name' (placeholder 'Enter First Name'), 'Last Name' (placeholder 'Enter Last Name'), 'Enter Email' (placeholder 'Enter Email'), 'Password' (placeholder 'Enter Password'), and 'Repeat Password' (placeholder 'Repeat Password'). Each input field has a small eye icon to the right. At the bottom right of the form is a green 'Submit' button.

Slika 3. Registracija korisnika

3.1.1 Podaci o radnji

Unos podataka o radnji podeljen je u tri sekcije. U okviru prve sekcije unose se osnovne informacije o radnji, a to su ime, opis i broj telefona. Nakon ove sekcije ide sekcija za unos adrese u kojoj je moguće uneti sledeće informacije: Ulica i broj, poštanski broj, grad, država, longitude i latitude. Longitude i latitude su koordinate koje se koriste za prikazivanje lokacije na Gugl mapama i to se koristi u potrošačkom režimu rada. Treća sekcija je sa desne strane i koristi se za učitavanje logo-a radnje. Nakon inicijalnog kreiranja radnje, administrator može u svakom trenutku promeniti neki od podataka. (Slika 4)

Order a food

Engleski Srpski Admin Test

Radnja

Zaposleni

Artikli

Dodaci

Radnja

Logo

Choose File No file chosen

Osnovne Informacije

Ime

Opis

Broj telefona

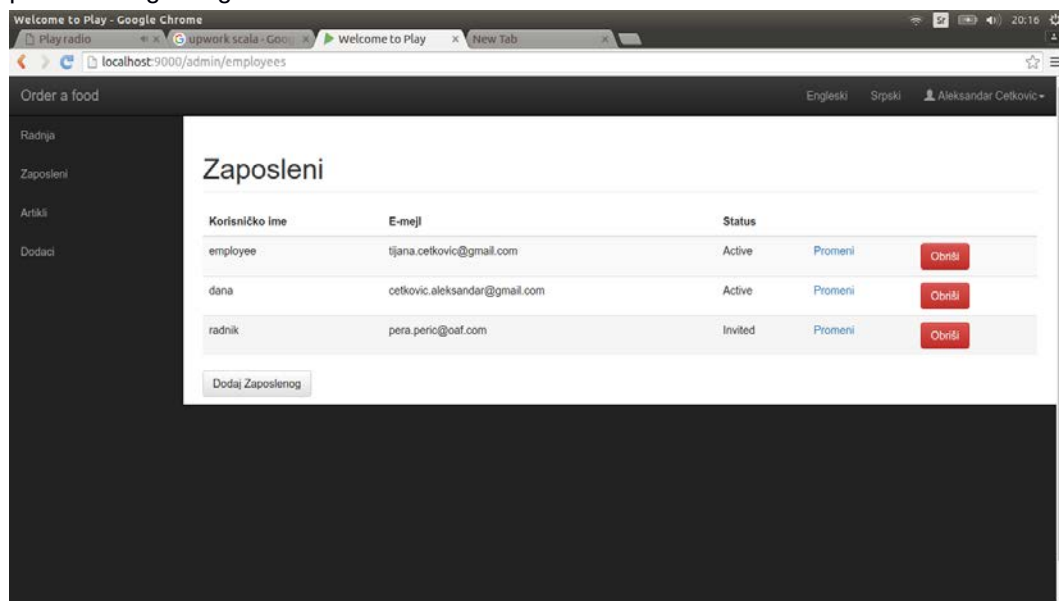
Adresa

Ulica i broj

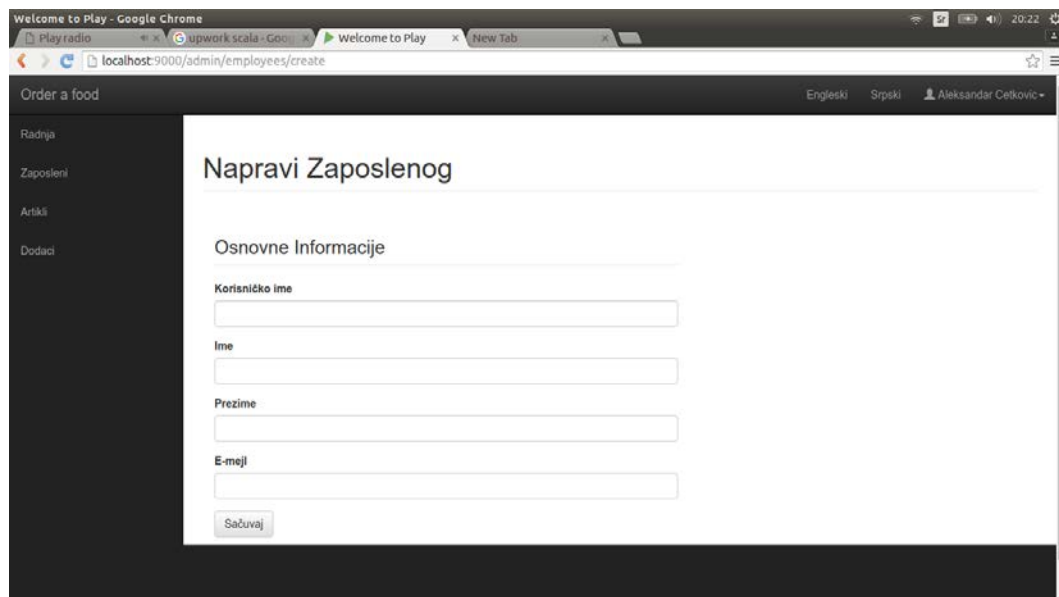
Slika 4. Unos podataka o radnji

3.1.2 Upravljanje zaposlenima

Adminstratoru je omogućeno da vidi sve zaposlene svoje kompanije u listi. Korisnik može da obriše ili promeni svakog zaposlenog (Slika 5). Stranica za dodavanje novog zaposlenog se otvara klikom na dugme "Dodaj Zaposlenog". Unose se osnovne informacije o radniku i email (Slika 6). Nakon uspešnog pravljenja zaposlenom stiže email sa linkom za registraciju, gde zaposleni unosi svoju šifru dva puta i na taj način vrši potvrdu svog naloga.



Slika 5. Lista zaposlenih



Slika 6. Stranica za dodavanje novog zaposlenog

3.1.3 Rad sa artiklima i dodacima

Linkovi “Artikli” i “Dodaci” sa leve strane omogućavaju pristup listi svih artikala (Slika 7) i dodataka (Slika 8) firme korisnika koji je trenutno aktivan. U okviru tih listi moguće je brisanje i promena, slično kao lista zaposlenih koja je opisana u prethodnom odeljku.

Ime	Cena	Težina	Promeni	Obriši
Pijesakavica	8.31	140.0	Promeni	Obriši
Pijesakavica Velika	10.2	200.0	Promeni	Obriši
Vesalica	9.0	150.0	Promeni	Obriši
Kobasica	7.0	140.0	Promeni	Obriši
Palacinka	5.0	100.0	Promeni	Obriši
Virale	7.0	100.0	Promeni	Obriši
Gumanaska	12.0	200.0	Promeni	Obriši

Slika 7. Lista artikala

Ime	Grupa Artikla	Promeni	Obriši
Kecap	Meal	Promeni	Obriši
Majonez	Meal	Promeni	Obriši
Krastavac	Meal	Promeni	Obriši
Paradajz	Meal	Promeni	Obriši
Eurokrem	Dessert	Promeni	Obriši
Plazma	Dessert	Promeni	Obriši
Dzem	Dessert	Promeni	Obriši

Slika 8. Lista dodataka

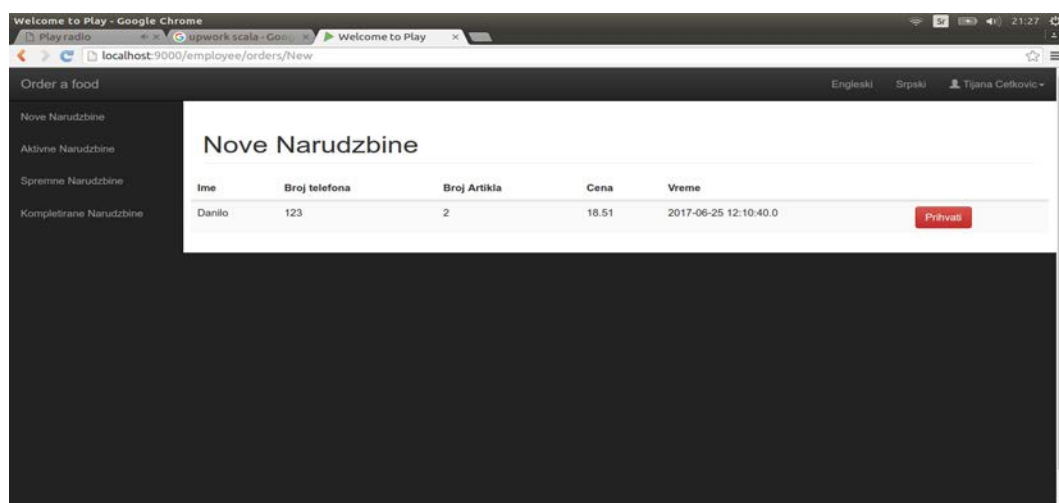
Pri pravljenju novog artikla unose se sledeći podaci: ime, opis, cena, težina, grupa artikla i slika. Postoje tri grupe artikla: obrok, dezert i napitak. Kod kreiranja novog dodatka popunjavaju se tri polja: ime, opis i grupa artikla. Ovde je bitno napomenuti da se grupa

artikla koristi za prikazivanje odgovarajućih dodataka. Ovo se vidi u potrošačkom režimu rada, kada korisnik formira svoju narudžbinu. Kada artikal pripada grupi Obrok, prikazivaće se samo dodaci koji se odnose na Obrok artikla, ista stvar je sa ostalim grupama.

3.2 Režim zaposlenog

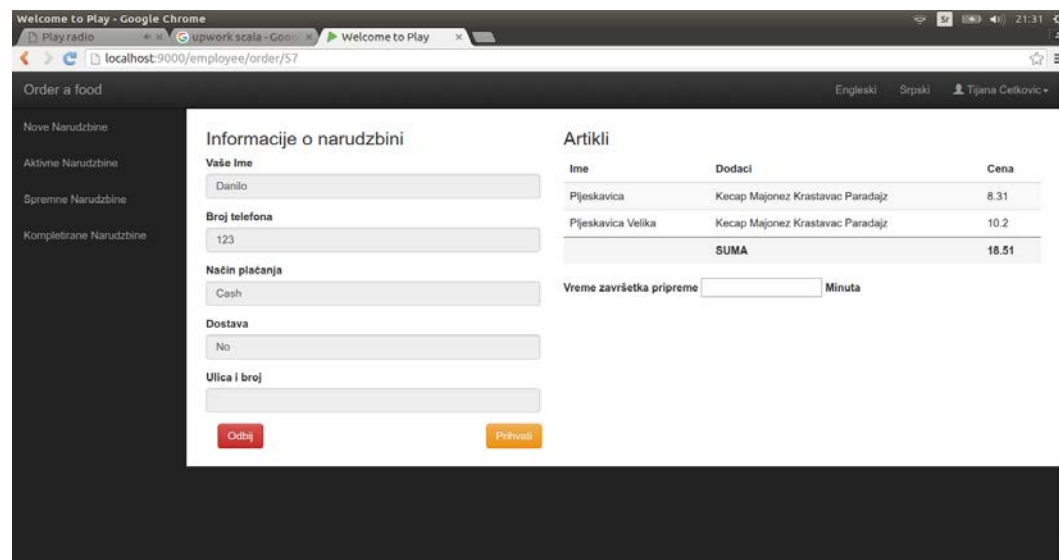
Nakon uspešnog prijavljivanja u aplikaciju, zaposleni ima pregled u četiri različite liste narudzbina:

- Nove narudzbine
- Aktivne narudzbine
- Spremne narudzbine
- Kompletirane narudzbine



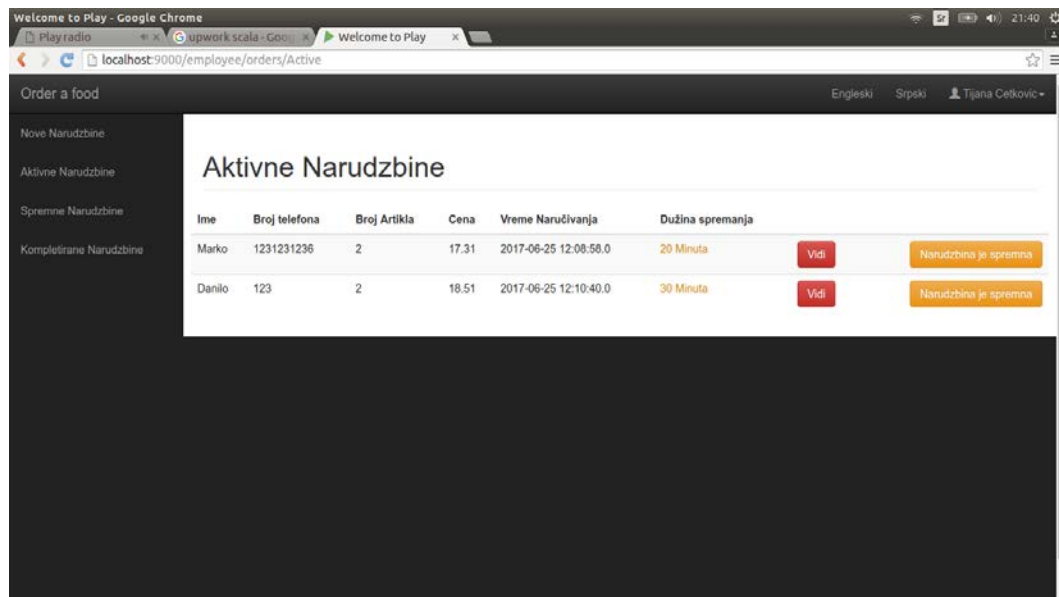
Slika 9. Lista novih narudžbina

Ova lista ujedno predstavlja životni vek jedne narudzbine. Naime, kada potrošač pošalje svoju narudžbinu, ona se snimi u bazi i ima status **Nova**. Zaposleni, vidi osnovne informacije o narudžbini u listi novih narudžbina (Slika 9), ako klikne na dugme "Prihvati" otvara se stranica sa detaljnim opisom svakog naručenog artikla. (Slika 10)



Slika 10. Detaljne informacije o narudžbini

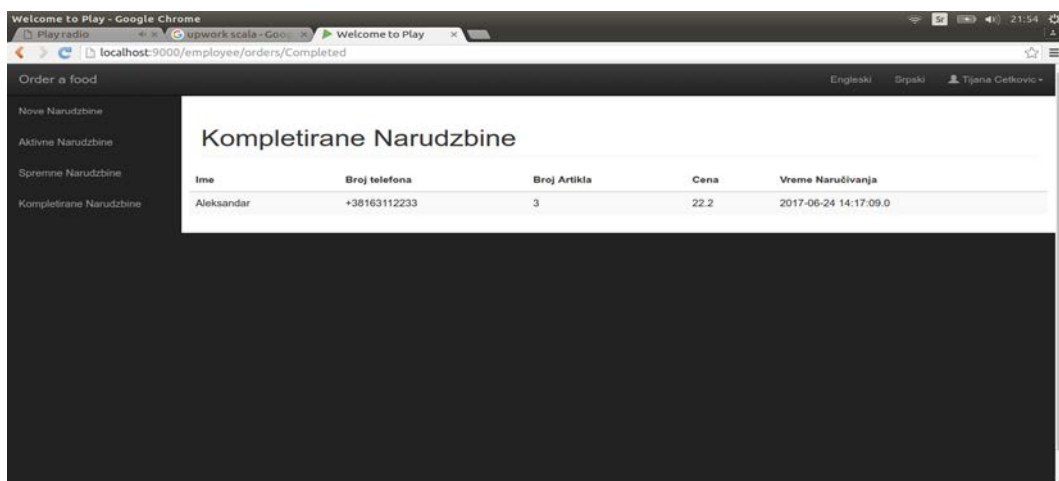
U slučaju sumnjivih okolnosti, zaposleni može da pozove naručioca i eventualno da odbije narudžbinu. Ako je narudžbina ispravna, zaposleni je dužan da unese vreme pripreme u minutima i klikom na dugme “Prihvati” narudžbina je prihvaćena i njen status prelazi u **Aktivna**. Spisak aktivnih narudžbina (Slika 11) sadrži osnovne informacije o narudžbini sa dodatnim podatkom o dužini spremanja, koji je unet pri prihvatanju narudžbine.



Ime	Broj telefona	Broj Artikla	Cena	Vreme Naručivanja	Dužina spremanja
Marko	1231231236	2	17.31	2017-06-25 12:08:58.0	20 Minuta
Danilo	123	2	18.51	2017-06-25 12:10:40.0	30 Minuta

Slika 11. Lista aktivnih narudžbina

Zaposleni ima na raspolaganju dve akcije koje može da izvrši nad aktivnim narudžbinama: Može da pregleda narudžbinu klikom na dugme “Vidi”, tada se otvara stranica sa detaljnim prikazom svih artikala – koja izgleda isto kao stranica za prihvatanje narudžbine. Druga akcija je konstatacija da je narudžbina spremna, izvršava se klikom na dugme “Narudžbina je spremna” i tada status narudžbine prelazi u **Spremna**. Sada se narudžbina nalazi u listi spremnih narudžbina i čeka na uručenje u zavisnosti od toga šta je naručilac izabrao u polju dostava. Kada je narudžbina uručena naručilocu, zaposleni je dužan da klikne na dugme “Narudžbina je kompletirana”, na taj način status narudžbine postaje **Kompletirana**. U listi kompletiranih narudžbina se nalaze osnovne informacije o narudžbinama i nema dodatnih akcija.

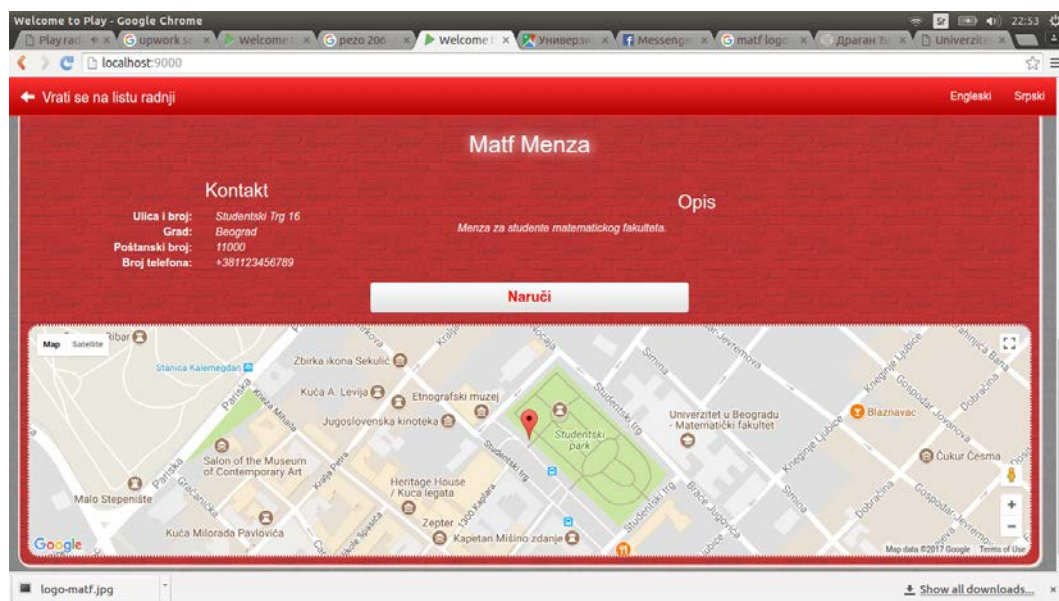


Ime	Broj telefona	Broj Artikla	Cena	Vreme Naručivanja
Aleksandar	+38163112233	3	22.2	2017-06-24 14:17:09.0

15
Slika 12. Lista kompletiranih narudžbina

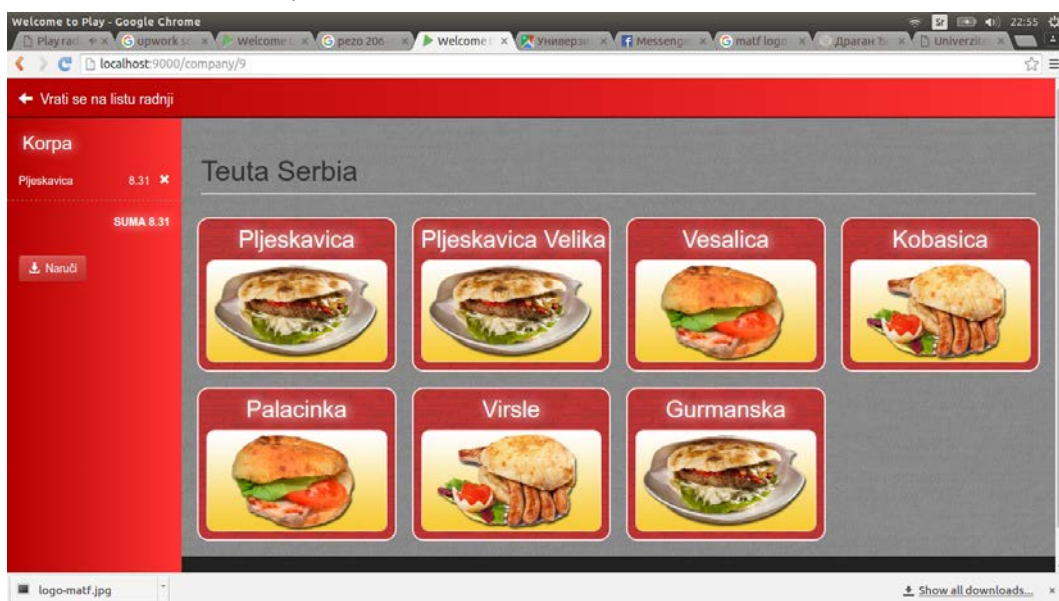
3.2 Potrošački režim

Potrošački režim ne zahteva prijavljivanje u aplikaciju i dostupan je na URL-u aplikacije bez dodatnog sufiksa. Na početnoj strani otvara se lista radnji i korisnik bira radnju u kojoj želi da naruči hranu. Svaka radnja, sadrži kontakt informacije, opis i mapu sa lokacijom, koja je određena na osnovu unesenih longitude i latitude koordinata u procesu kreiranja radnje. (Slika 13)



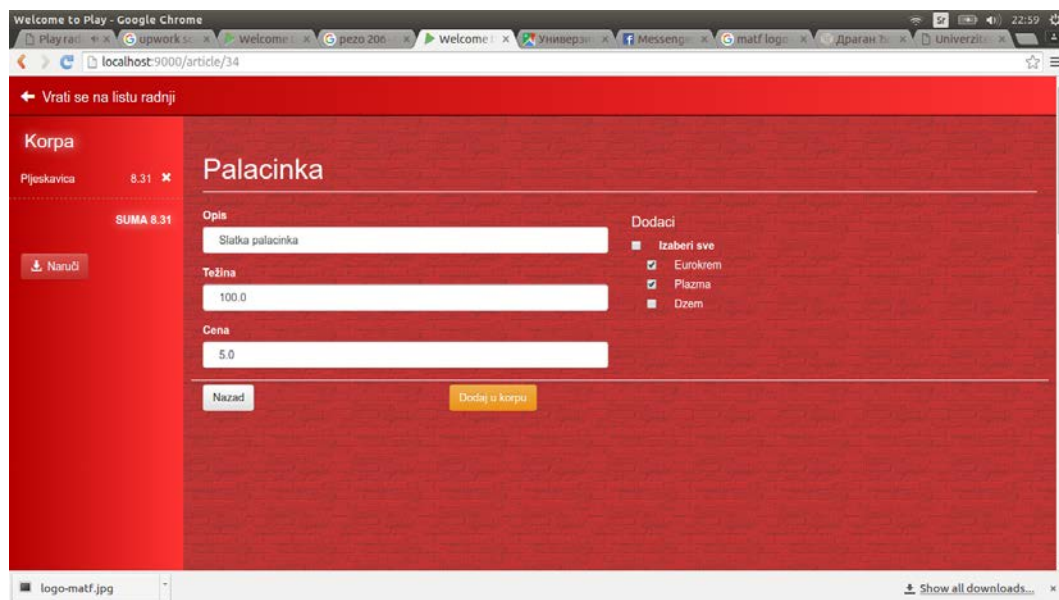
Slika 13. Lista radnji

Klikom na dugme "Naruči" otvara se stranica sa listom artikla izabrane radnje (Slika 14). Pored dostupnih artikala, stranica sadrži i aktivnu korpu sa leve strane. U korpi se nalaze izabrani artikli, kao i ukupna cena svih artikala.



Slika 14. Spisak artikla jedne radnje

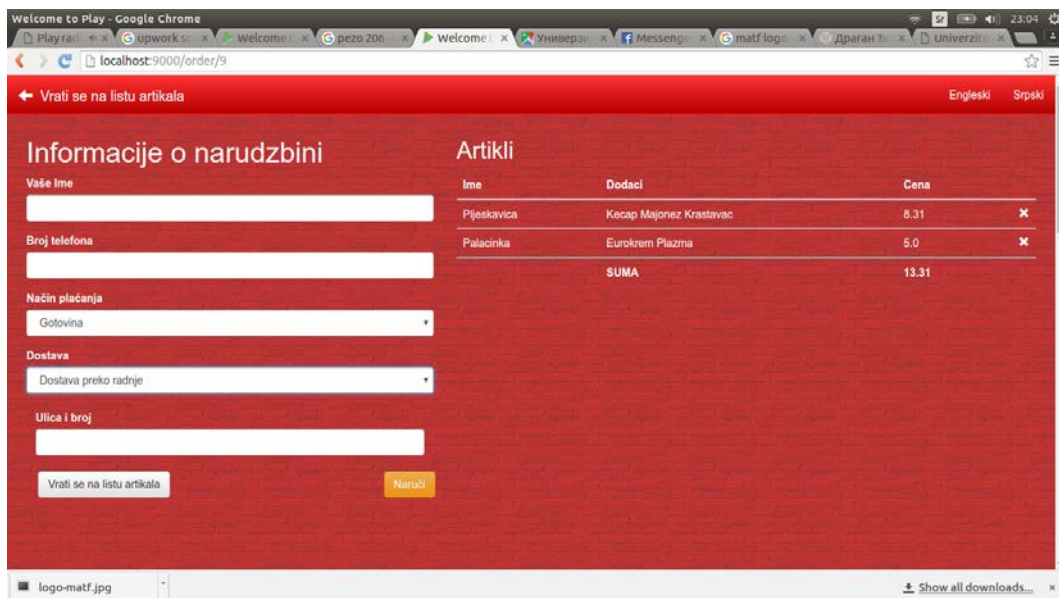
Klikom na artikal, otvara se stranica sa detaljnim informacijama o artiklu, kao i panel za odabir željenih dodataka. Klikom na dugme “Dodaj u korpu”, artikal postaje deo narudžbine. (Slika 15)



Slika 15. Dodavanje artikla u korpu

Nakon odabira svih željenih artikala korisnik klikom na dugme “Naruči” otvara stranicu u kojoj šalje narudžbinu. Stranica za slanje narudžbine zahteva unos sledećih polja:

- Ime
- Broj telefona
- Način plaćanja (gotovina, kartica)
- Dostava (Dostava preko radnje, taxi ili lično preuzimanje)
- Ulica i broj



Slika 16. Popunjavanje informacija o narudžbini

Na desnoj strani ove stranice je prikazan pregled svih artikala zajedno sa dodacima i cenom. Između ostalog, korisniku je omogućeno da izbaci neki od artikala iz narudžbine klikom na "x" dugme (Slika 16). Klikom na dugme "Naruči" narudžbina se kreira sa statusom **Nova** I kao takva prikazuje se zaposlenom u listi novih narudžbina.

4. Programski jezik Scala

Scala je moderni programski jezik sa snažnom podrškom za objekto-orijentisano funkcionalno i paralelno programiranja. Scala podržava statičko tipiziranje, izvršava se u okviru Java virtualne mašine (JVM) i kompatibilna je sa Javom i sa ostalim JVM jezicima.

U ovom odeljku rada su izloženi osnovni koncepti programskog jezika Scala. Nakon toga detaljno je objašnjeno Play okruženje koje u kombinaciji sa Skalom daje kompletan alat za implementaciju i pokretanje veb aplikacije na serveru.

4.1 Osnovni koncepti Scala programskog jezika

U ovom odeljku su opisane osnovne karakteristike programskog jezika Scala. Akcenat je na pojmovima koji su specifični za Scalu, dok su pojmovi koji su opšti za objektno orijentisano programiranje samo navedeni.

4.1.1 Tipovi podataka, klase, traitovi, nasleđivanje

Tipovi podataka

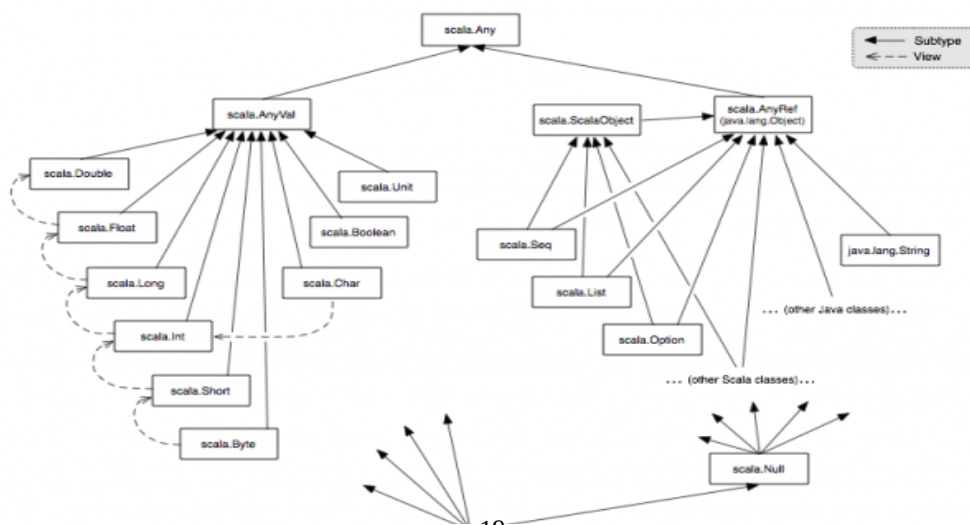
U programskom jeziku Scala postoje promenljive (eng. Variables) i vrednosti (eng. Values). Tako da neki objekat možemo deklarirati na dva načina:

```
var name1 = "mark1"
val name2 = "mark2"
```

U slučaju "var" vrednost objekta *name1* se može menjati u daljem izvršavanju programa, dok u slučaju "val" vrednost objekta *name2* se ne može menjati. Na ova dva načina se može definisati bilo koji objekat, a kompajler sam prepoznaje tip objekata. Ovde treba primetiti da Scala ne zahteva ";" na kraju naredbe, ali iako se stavi, kompajler to neće smatrati kao grešku. Tip podataka se može i eksplicitno navesti na sledeći način:

```
var name1: String = "mark1"
```

Za razliku od Java, u Skali su sve vrednosti objekti, uključujući primitivne numeričke tipove i funkcije. Shodno tome, Scala je klasno zasnovana, sve vrednosti su primeri neke klase. Sledeći dijagram ilustruje hijerarhiju klasa.



19
Slika 17. Hijerarhija klasa. Slika preuzeta iz [2]

Nadklasa svih klasa je **scala.Any**, ona ima dve direktno izvedene klase **scala.AnyVal** i **scala.AnyRef** koje predstavljaju dve različite grupe klasa: vrednosne klase i referentne klase. Sve vrednosne klase su predefinisane i one odgovaraju primitivnim tipovima iz Jave. Sve ostale klase predstavljaju referentne tipove. Kada korisnik definiše novu klasu, ta klasa je podrazumevano referentna klasa i automatski postaje izvedena klasa od klase **scala.AnyRef**. Ako se Scala koristi u kontekstu Java virtualne mašine (JVM) onda **scala.AnyRef** predstavlja **java.lang.Object**. (Slika 17)

Klase

Definicija klase je slična kao i u Javi sa malim sintaksnim izmenama, sto se može videti u sledećem primeru klase *Point*:

```
class Point(var x: Int, var y: Int) {  
  def move(dx: Int, dy: Int): Unit = {  
    x = x + dx  
    y = y + dy  
  }  
  override def toString: String = "(" + x + ", " + y + ")"  
}
```

Iz potpisa klase se može videti da je kreiran konstruktor koji prima dva argumenta i radi klasično dodeljivanje vrednosti klasnim promenljivama.

Trait-ovi (prev. Osobina)

Trait u Skali je isto sto i interfejs u Javi. On se ne može instancirati i predstavlja spisak osobina koje klasa koja ga implementira treba da zadovolji, tačnije ta klasa treba da definiše te osobine.

```
trait Similarity {  
  def isSimilar(x: Any): Boolean  
  def isNotSimilar(x: Any): Boolean = !isSimilar(x)  
}
```

Primer implementiranja traita:

```
class SomeClass extends Similarity {  
  ...  
}
```

U Skali važe svi osnovni koncepti objektno orijentisanog programiranja, tako da se oni neće navoditi u daljem tekstu. Pod ovim se smatra sva pravila pri nasleđivanju, predefinisane funkcije, prevazilaženje funkcija, apstraktne funkcije... U daljem tekstu je izdvojeno samo ono sto je različito u odnosu na Javu, a samim tim i celu grupu objekto orijentisanih jezika.

Objekti

U Skali postoji još jedan tip podataka i to je *Object*.

```
object Math {  
  def add(x: Int, y: Int): Int = {  
    x+y  
  }  
}
```

Objekat predstavlja unikat (eng. singleton). To je klasa koja se automatski instancira i u svakom trenutku izvršavanja programa postoji samo jedna njena instanca. Funkcijama i promenljivim se pristupa preko imena objekta:

Math.add(3,4)

Ovo je jako korisna stvar, pogotovo kod veb aplikacija, pošto je sam unikat jedan od najčešće korišćenih obrazaca dizajna pri razvoju veb aplikacija. Najčešći razlozi za korišćenje unikata su:

- * **Ušteda memorije** – kada je klasa prevelika, svaka njena instanca zauzima dosta radne memorije, pa je poželjno koristiti samo jednu instancu
- * **Konzistentnost podataka** – kada se nekom resursu ne sme pristupiti ako je on već zauzet, primer može biti konekcija ka bazi.

4.1.2 Funkcije, Karijeve funkcije, implicitne funkcije

Funkcija je jedna od glavnih prednosti Scale. Sama koncipiranost funkcija i njihovih mogućnosti se dosta bazira na matematici, što programeru sa dobrim matematičkim znanjem daje veliku prednost u odnosu na ostale programere. Takav programer može iskoristiti svu "moć" Scale, što mu omogućava da neki kompleksan problem svede na matematički jednostavan model.

Definicija funkcije:

```
def <ime>(<lista-argumenata>): <povratni-tip> = {<telo-funkcije>}
```

Pravila sa običnim funkcijama su ista kao u Javi, tako da je dalji tekst bazirati na nove stvari kao što su Karijeve funkcije i implicitne funkcije.

Karijeve funkcije

Karijeva funkcija je funkcija koja ima dva ili više lista argumenata. Primer:

```
def mod(n: Int)(x: Int): Int = x % n
```

Pri pozivu funkcije sa popunjenom jednom listom argumenata kao rezultat se dobija funkcija koja očekuje drugu listu argumenata. Tako je, dalje:

```
def mod2 = mod(2)
```

Sada **mod2** funkcija predstavlja:

```
def mod2(x: Int): Int = x % 2
```

mod2 funkcija predstavlja funkciju koja vraća ostatak pri deljenju sa brojem 2.

Ovo je jako dobra stvar, jer se na ovaj način mogu generalizovati neki delovi koda i samim tim smanjiti obim koda, a sa druge strane olakšati održavanje i dalju implementaciju samog projekta. Jedan realniji primer: Postoji 10 različitih entiteta i svaki od njih ima svoju tabelu u bazi. Treba napraviti metode koje brišu element iz tabele po ključu. Umesto da se piše 10 različitih metoda, može se definisati Karijeva funkcija, koja prima ključ i ime tabele.

```
def delete(table: String)(key: Int) = ...
```

Nakon toga, se za svaki entitet kreira funkcija samo prosleđivanjem imena tabele.

```
def deleteUser = delete("USER")
def deleteJob = delete("JOB")
```

...

Kao što se može primetiti sva logika je ostala u **delete** funkciji, tako da u slučaju neke promene programer menja samo nju, a ta promena će imati uticaj na ostalih 10 funkcija.

Implicitne funkcije

Implicitne funkcije dosta olakšavaju posao u razvoju veb aplikacija. Naime, savremene arhitekture u razvoju veb aplikacija zasnivaju se na više nezavisnih slojeva. I svaki sloj ima svoju reprezentaciju nekog objekta. Na primer, uzećemo da naša aplikacija ima 3 sloja prikaz (eng. View), servis i repozitorijum (pristup bazi). Imamo entitet **User**, koji se u sva tri sloja pojavljuje kao različit objekat (razlog ovome leži u tome što ne treba svim nivoima ceo skup podataka koje sadrži jedan **user**). Pošto je **User** različit na sva tri nivoa, moramo vršiti eksplicitnu konverziju pri svakom pozivu koji ide od jednog ka drugom nivou. Ovo se mnogo lakše rešava primenom implicitnih funkcija, naime potrebno je samo definisati konverter funkciju kao implicitnu i navesti joj kontekst u kome ona važi. Nakon toga, kompajler će sam prepoznati implicitnu konverziju i sam programer ne mora brinuti o tipu objekta koji prosleđuje funkciji iz drugog sloja.

Primer: u servisu imamo funkciju **add(user: UserServiceModel)** za dodavanje **user**-a, ona u sebi zove funkciju **addRepo(userRepo: UserRepoModel)**, kao što se može primetiti ove dve funkcije primaju različite tipove modela kao argument. Definiše se implicitna funkcija koja konvertuje jedan model u drugi:

```
implicit def convertFromServiceToRepoModel(usr: UserServiceModel): UserRepoModel
```

U servisu se uključi implicitni kontekst, kao import i kompajler će automatski prevesti tip modela koji je dobio u zahtevani tip modela. Čak je i programeru olakšano tako što će u razvojnom okruženju implicitna konverzija biti istaknuta, tako što je podvučena i iskošena.

```
def add(user: UserServiceModel): Unit = {
  UserRepo.addRepo(user)
}
```

Naravno da se implicitne funkcije mogu koristiti i u druge svrhe, a ne samo u svrhe konverzije.

4.1.3 Upoređivanje obrasca (eng. Pattern Matching)

U Scali postoji mehanizam za višestruko grananje. Ovaj mehanizam je analogan **switch** operatoru u Javi sa razlikom što argument poređenja može biti instanca bilo koje klase. Sledi primer kako izgleda sintaksa upoređivanja obrasca:

```
x match {  
    case 1 => "one"  
    case "two" => 2  
    case y: Int => "scala.Int"  
}
```

U ovom odeljku treba izdvojiti i opcioni tip podataka to je **Option[A]** gde je A bilo koja klasa. Opcioni tip se koristi u situacijama kada instanca neke klase može imati NULL vrednost. Inače, najčešća greška koja se javlja pri izvršavanju bilo koje aplikacije je *NullPointerException*, tako da je Scala ovim tipom dosta olakšala posao programerima. Postoji više načina za dohvaćanje vrednosti iz opcionog podatka, jedan od načina je korišćenje *pattern matching-a*:

```
val name: Option[A] = Some("Mark")  
  
name match {  
    case Some(value) => {  
        println(value)  
    }  
    case None => {  
        println("No value")  
    }  
}
```

Iz primera se jasno vidi da je **None** ekvivalentno *Null-u* iz Jave. Ključna reč **Some** se koristi u situaciji kada se opcionalnoj promenljivoj dodeljuje vrednost. Kada se koristi opcionalna promenljiva kompajler zahteva da se na gore opisan način dohvati njena vrednost, na taj način se izbegava *NullPointerException*.

4.1.4 Liste, funkcije za rad sa listama

Liste u programskom jeziku scala su manje više iste kao i u Javi. Ono što daje posebnu dimenziju listama su funkcije **map**, **filter** i **fold**.

Map i filter funkcije

Ove funkcije kao argument primaju funkciju. Map funkcija vrsi mapiranje cele liste u listu elemenata drugog tipa. Na primer, ako imamo listu List[A] i želimo da dobijemo listu elemenata tipa B (List[B]). Potrebno je napraviti funkciju F koja prima argument tipa A i kao rezultat vraća element tipa B. U tom slučaju poziv map funkcije bi izgledao:

```
list.map(F)
```

rezultat ovog poziva bi bila lista elementata tipa B.

Filter funkcija filtrira postojeću listu po nekom kriterijumu koji se zadaje kao argument. Taj kriterijum predstavlja funkciju koja vraća tačno ili netačno. Dakle, ako imamo List[A] i funkciju F koja kao argument prima objekat tipa A, a vraća rezultat koji je tipa Boolean. Poziv filter funkcije bi izgledao:

```
list.filter(F)
```

rezultat ovog poziva je lista elemenata tipa A, sa tim što će rezultat sadržati manje ili jednako elementa kao početna lista, a to zavisi od broja elemenata koji zadovoljavaju uslov F.

Scala podržava lambda funkcije koje su zasnovane na lambda računu [1]. Lambda funkcija je anonimna funkcija koja se definiše u jednoj liniji (eng. inline function).

Filter i map kao argument mogu primiti i lambda funkciju, što znači da nije potrebno pisati pravu definiciju funkcije F, nego se funkcija F može navesti kao lambda funkcija ako nije previše kompleksna.

Primer:

```
list.filter(x => x != None)
list.map(user => user.name)
```

Prvi slučaj izdvaja sve elemente iz liste koji su različiti od None, a u drugom slučaju listu korisnika prevodimo u listu njihovih imena.

Više reči o ovim funkcijama biće u delu *Rad sa Bazom*, pošto se sam rad sa bazom zasniva na ovim funkcijama.

Svi primeri u poglavlju 4.1 su delo autora rada i bazirani su na primerima iz [2].

4.2 Upoznavanje sa Play framework-om

Play okruženje predstavlja kompletan skup stvari koje su potrebne za razvoj jedne veb aplikacije. Ovo okruženje je razvijeno od strane **Typesafe** organizacije, koja inače razvija i sam programski jezik Scala.

Šta sve sadrži Play okruženje:

- Server za podizanje aplikacije
- Šablone za lakše kreiranje HTML strana
- Podršku za **JavaScript**
- Dosta dodataka (eng. plug-in)

Play okruženje je besplatno i dostupno je na stranici:

<https://www.playframework.com/documentation/2.5.x/Installing>

Instalacija je jako jednostavna, potrebno je samo skinuti **activator**. Aktivator će pri startovanju dovući sa interneta sve potrebne stvari, kao što su Scala biblioteke, sbt, play biblioteke... U roku od 15 minuta okruženje je spremno za rad.

4.2.1 MVC obrazac, organizacija klasa

Kao i većina okruženja za rad sa veb aplikacijama i Play okruženje prati poznati MVC obrazac. MVC obrazac razdvaja aplikaciju na 3 sloja, a to su *Model*, *View* i *Controller*.

Name	Größe
play-bookstore	--
app	--
controllers	--
views	--
conf	--
application.conf	1.9 KB
routes	323 Bytes
project	--
Build.scala	420 Bytes
plugins.sbt	283 Bytes
build.properties	19 Bytes
public	--
images	--
javascripts	--
stylesheets	--
test	--
README	151 Bytes

Slika 18. Struktura fajlova u Play projektu

Nakon instalacije, Play okruženje kreira inicijalnu strukturu koda razdeljenu po folderima. Ta struktura je u saglasnosti sa MVC obrascem. Kontroleri se smestaju u **controllers** folder i u njima se definišu serverske akcije. HTML stranice se smestaju u okviru **views** foldera. U okviru **conf** foldera se nalazi **application.conf** koji je zadužen za sve konfiguracione podatke, pored njega nalazi se i **routes** fajl u kome se definišu pravila rutiranja kroz veb aplikaciju. U okviru **project** foldera se nalaze fajlovi u kojima se konfigurise proces kompajliranja. **Public** folder sadrži slike, javascript fajlove kao i css fajlove. U okviru **test** foldera se smestaju Unit testovi. Ovo bi bio kratak opis strukture fajlova, koja nije fiksna i može biti promenjena. (Slika 18)

4.2.2 Kompajliranje

Za kompajliranje *Play* koristi *sbt* (eng. *Scala Build Tool*). Sbt je jako moćan alat, pomoću njega se dovlače i svi dodaci (eng. *Dependencies*). Pri pravljenju projekta u *Play*-u formira se i *activator.exe*. Pokretanjem aktivatora preko komandne linije ulazi se u fazu u kojoj može da se kompajlira aplikacija i pokrene server, tj aplikacija. (Slika 19)

```

cetko@cetko-X550LB: ~/project/OAF
cetko@cetko-X550LB:~$ cd project
cetko@cetko-X550LB:~/project$ ll
total 28
drwxr-xr-x 5 cetko cetko 4096 jun  4 17:50 ./
drwxr-xr-x 37 cetko cetko 4096 apr 26 22:50 ../
drwxr-xr-x 8 cetko cetko 4096 jun 12 22:39 .git/
-rw-rw-r-- 1 cetko cetko 185 maj 14 16:46 .gitignore
drwxr-xr-x 11 cetko cetko 4096 jun  6 22:31 OAF/
drwxr-xr-x 6 cetko cetko 4096 jun 12 20:59 OAF-DAL/
-rw-rw-r-- 1 cetko cetko  8 maj 14 16:46 README.md
cetko@cetko-X550LB:~/project$ cd OAF
cetko@cetko-X550LB:~/project/OAF$ ll
total 1272
drwxr-xr-x 11 cetko cetko 4096 jun  6 22:31 ./
drwxr-xr-x 5 cetko cetko 4096 jun  4 17:50 ../
-rwxr-xr-x 1 cetko cetko 9507 feb 28 21:51 activator*
-rwxr-xr-x 1 cetko cetko 7342 feb 28 21:51 activator.bat*
-rw-rw-r-- 1 cetko cetko 1213545 feb 28 21:51 activator-launch-1.3.7.jar
drwxr-xr-x 8 cetko cetko 4096 maj 17 23:04 app/
drwxr-xr-x 2 cetko cetko 4096 maj  8 23:38 app-2.11/
-rw-rw-r-- 1 cetko cetko  687 jun  4 13:24 build.sbt
drwxr-xr-x 3 cetko cetko 4096 jun 20 21:09 conf/
-rw-rw-r-- 1 cetko cetko  80 feb 28 21:51 .gitignore
drwxr-xr-x 5 cetko cetko 4096 apr 11 19:21 .idea/
-rw-rw-r-- 1 cetko cetko  591 feb 28 21:51 LICENSE
drwxr-xr-x 2 cetko cetko 4096 apr 11 21:55 logs/
drwxr-xr-x 4 cetko cetko 4096 maj 14 14:22 project/
-rw-rw-r-- 5 cetko cetko 4096 feb 28 21:51 public/
-rw-rw-r-- 1 cetko cetko 148 feb 28 21:51 README
-rw-rw-r-- 1 cetko cetko  41 feb 28 23:24 README.md
drwxr-xr-x 6 cetko cetko 4096 jun 12 22:48 target/
drwxr-xr-x 2 cetko cetko 4096 maj 31 21:05 test/
cetko@cetko-X550LB:~/project/OAF$ ./activator
[info] Loading project definition from /home/cetko/project/OAF/project
[info] Loading project definition from /home/cetko/project/OAF-DAL/project
[info] Set current project to OAF (in build file:/home/cetko/project/OAF/)
[OAF] $

```

Slika 19. Startovanje aktivatora

Aktivator sadrži više komandi, ovde treba izdvojiti par bitnih:

- o `compile` – kompajliranje projekta
- o `clean` – čišćenje projekata i brisanje privremenih fajlova koji se prave pri kompajliranju
- o `run` – startovanje aplikacije, automatski se aplikacije podiže na portu 9000, ako se ne navede eksplicitno drugi port
- o `reload` – ponovno učitavanje konfiguracionih fajlova
- o `stage` – pripremanje aplikacije za produkcion server
- o `start` – startovanje aplikacije na produkcionom serveru

Aktivator u pozadini zove komande sbt-a i na taj način je programerima olakšano pošto je sbt sintaksa dosta teža.

Ovde se treba dodati jedne važne činjenice. Naime, programski jezik Scala je u osnovi funkcionalan i kao takav u njemu se dosta koristi rekurzija. Opšte je poznato da je iteracija ekvivalentna rekurziji. Shodno tome, kompajler svaku rekurziju prevodi u iteraciju, zato što u praksi rekurzija nije efikasna - problem punjenja radne memorije. Razlog zbog kojeg je Scala izabrala rekurziju leži u boljoj čitljivosti koda, koja je ranije opisana u delu 2.4.

4.3 Kontroleri i forme

Kontroleri i forme predstavljaju prvi sloj u obradi zahteva koje poslat sa klijenta. Forme se koriste za validaciju i prevodjenje zahteva u primerak klase koji je poznat daljem kodu aplikacije (pravljenje objekta od JSON formata u kome obično stiže zahtev). U kontrolerima se definišu akcije, gde svaka akcija predstavljaju jednu pristupnu tačku veb aplikacije (eng. entry point).

4.3.1. Akcije i vrste akcija

Kontroler je specijalna klasa koja sadrži metode. Metode predstavljaju akcije. Svaka akcija se vezuje za jednu url adresu. Zavisno od prirode akcije, akcije mogu biti POST, PUT, GET i DELETE. Ovo je u saglasnosti sa HTTP protokolom. Sam kontroler predstavlja unikat.

Primer akcije:

```
object Application extends Controller {  
  def index = Action {  
    Ok("It works!")  
  }  
}
```

Svaki kontroler nasleđuje klasu **Controller** koja u sebi ima ugrađen mehanizam za rukovanje sa akcijama, a samim tim i HTTP zahtevima. Svaka akcija je specifična metoda čija definicija počinje **Action** blokom. Kao rezultat ona treba da vrati objekat tipa **Result**. U *Play* okruženju postoji dosta predefinisanih **Result** objekata, a to su:

- o **Ok("Hello world!")** - Vraća http status 200
- o **NotFound** – Vraća http status 404

- o **BadRequest**(views.html.form(formWithErrors)) – vraća status 400 i renderuje stranicu navedenu kao argument
- o **InternalServerError**("Oops") – vraća status 500
- o **Status**(488)("Strange response type") – vraća status koji je naveden kao argument
- o **Redirect**(views.html.Redirect) – vraća status 303 i renderuje stranicu navedenu kao argument

Bitno je napomenuti da svaka od ovih funkcija može kao argument primiti HTML stranicu. Ovo su najčešće korišćeni **Result** objekti. Primeri su preuzeti sa [3]

4.3.2. Validacija podataka pomocu forme

Forme su specifične klase čija namena je validacija podataka na serverskoj strani. Svaka forma se sastoji od **case** classe, koja se koristi za uzimanje podataka iz HTTP zahteva, i funkcije koja je zadužena za validaciju istih podataka. Primer.

```
case class LoginData(username: String, password: String)
```

```
object LoginForm {
  def getLoginData(implicit session: Session) = Form(
    mapping(
      "username" -> nonEmptyText,
      "password" -> nonEmptyText
    )(LoginData.apply)(LoginData.unapply)
    verifying ("Username or password are incorrect", result => result match {
      case loginData => checkCredentials(loginData.username, loginData.password)
    })
  )
}
```

LoginData je case klasa koja sadrži podatke **username** i **password**. **GetLoginData** je funkcija koja proverava validnost svakog podatka. U **mapping** klauzuli se sa leve strane navodi identifikator podatka u HTTP zahtevu, a sa desne strane se navodi uslov koji treba biti ispunjen. Posle **mapping** klauzule, može se navesti **verifying** klauzula u kojoj se navode specifični uslovi koji se odnose na dva ili više podataka iz zahteva. U ovom slučaju **verifying** proverava da li se **username** i **password** slažu i da li su validni.

NonEmptyString je ugrađen uslov, postoji još dosta najčešće korišćenih uslova koji su već definisani u *Play* okruženju. Naravno, programer može definisati i svoj uslov. Sledeći primer ilustruje kako se forme koriste u okviru akcije.

```
def authenticate = Action.async { implicit request =>
  play.api.db.slick.DB.withTransaction { implicit session =>
    LoginForm.getLoginData.bindFromRequest.fold(
      errors => Future.successful(BadRequest(views.html.login(errors))),
      loginData => {
        UserService.findByUsername(loginData.username) map { user =>
          gotoLoginSucceeded(user.username)
        } getOrElse (Future.successful(BadRequest("Wrong login")))
      }
    )
  }
}
```

}

Funkcija **BindFromRequest** proverava ispravnost podataka i vezuje podatke za instancu case klase **LoginData**, ako je validacija prošla uspešno. Ako validacija ne prođe, funkcija vraća **LoginForm**-u sa greškama, što se kasnije na HTML stranici koristi za prikazivanje grešaka. Primer je deo aplikacije OAF.

4.4 Rad sa stranicama

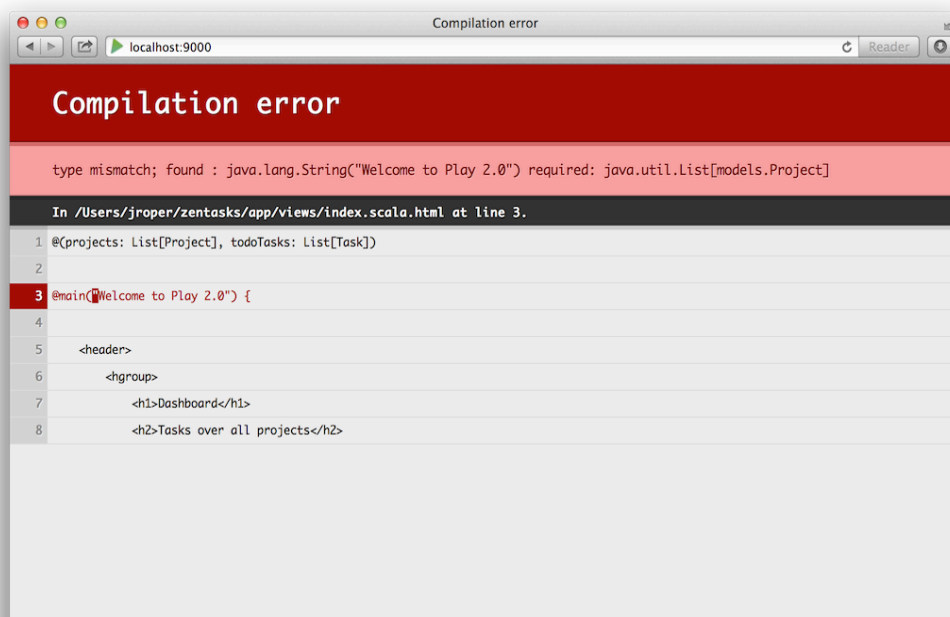
U ovom odeljku je opisano kako se prave HTML stranice i šabloni u Play okruženju. Takođe, prikazano je kako su stranice i šabloni organizovani u aplikaciji OAF.

4.4.1 Korišćenje Scala koda na html stranim

Play okruženje sadrži alat za lakše pravljenje HTML strana i taj alat se naziva *Twirl*. *Twirl* se zasniva na korišćenju Scala koda na HTML stranicama. Kasnije, kompajler prevodi taj Scala kod u ekvivalentan HTML kod. Osnovne karakteristike *Twirl*-a su:

- Kompaktan i izražajan
- lak za učenje
- nije novi jezik - oslanja se na Scalu

Ono što je bitno jeste da se HTML stranice kompajliraju što omogućava programeru da uoči grešku na stranici (Slika 20). Inače, pri pravljenju veb aplikacija nije moguće uočiti grešku (na stranici) u fazi kompajliranja nego tek pri izvršavanju programa.



Slika 20. Prikaz greske na HTML strani

Ove stranice se nazivaju *Play* Scala šabloni. *Play* Scala šabloni predstavljaju običan tekstualni fajl koji sadrži male blokove Scala koda. Bitno je navesti da ovi šabloni mogu generisati bilo koji tekstualni format, kao što su HTML, XML ili CSV.

Šabloni se kompajliraju kao standardne Scala funkcije, prateći jednostavnu konvenciju imenovanja. Ako napravimo **views/Application/index.scala.html** šablon fajl, on će generisati **views.html.Application.index** klasu koja ima **apply()** metod. Na primer:

```
@(customer: Customer, orders: List[Order])
```

```
<h1>Welcome @customer.name!</h1>
```

```
<ul>
@for(order <- orders) {
  <li>@order.title</li>
}
</ul>
```

Programer može u bilo kom delu koda pozvati ovu funkciju. Na primer:

```
val content = views.html.Application.index(c, o)
```

Sintaksa: Magični "@" karakter

Scala šabloni koriste "@" kao specijalni karakter. Svaki put kada se ovaj karakter pojavi u kodu, on označava početak dinamičkog bloka, odnosno bloka u kome je Scala kod. Ostali deo fajla predstavlja običan HTML.

Primeri:

1.

```
Hello @customer.name!
      ^^^^^^^^^^^^^^^^^
Dinamički kod
```

2.

```
Hello @(customer.firstName + customer.lastName)!
      ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
Dinamički kod
```

```
3. Hello @{val name = customer.firstName + customer.lastName; name}!
      ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
Dinamički kod
```

Primer 3 predstavlja složeniji dinamički kod i zato se navode vitičaste zagrade.

4.4.2 Prikazivanje podataka iz kontrolera

Pošto su šabloni kao funkcije, oni mogu primiti argumente. Argumenti se navode na vrhu šablon fajla.

```
@(customer: Customer, orders: List[Order])
```

tako na primer iz kontrolera možemo pozvati neku stranicu sa podacima koje je kontroler izračunao.

code of controller action

```
...
Ok(views.html.Application.index(customer, orders))
```

...

Šabloni omogućavaju korišćenje sledećih operatora na HTML stranice:

- If blokovi

```
@if(items.isEmpty) {  
  <h1>Nothing to display</h1>  
} else {  
  <h1>@items.size items!</h1>  
}
```

- For petlja

```
<ul>  
  @for(p <- products) {  
    <li>@p.name ($@p.price)</li>  
  }  
</ul>
```

- Uključivanje drugih Scala biblioteka

```
@import utils._
```

Na ovaj način se na stranici mogu koristiti Scala metode koje su ranije napravljene.

Scala šabloni omogućavaju i ugnježdavanje. Ugnježdavanje se koristi u situacijama kada se neki deo HTML koda ponavlja na više različitih stranica. Na primer: navigaciona tabla (tabla koja sadrži dugmad za navigaciju kroz aplikaciju), ulogovani korisnik...

Sledi primer iz OAF aplikacije:

Osnovni Šablon je **main.scala.html**:

```
@(title: String)(content: Html)  
@import play.api.i18n.Messages.Message  
<!DOCTYPE html>  
<html lang="en">  
  <head>  
    <title>@title</title>  
    <link rel="stylesheet" media="screen" href="@routes.Assets.versioned("stylesheets/main.css")">  
    <link rel="stylesheet" media="screen" href="@routes.Assets.versioned("stylesheets/bootstrap/bootstrap.css")">  
    <link rel="stylesheet" media="screen" href="@routes.Assets.versioned("stylesheets/bootstrap/bootstrap-theme.css")">  
    <link rel="stylesheet" media="screen" href="@routes.Assets.versioned("stylesheets/admin.css")">  
    <link rel="stylesheet" media="screen" href="@routes.Assets.versioned("stylesheets/customer.css")">  
    <link rel="shortcut icon" type="image/png" href="@routes.Assets.versioned("images/favicon.png")">  
    <script src="@routes.Assets.versioned("javascripts/hello.js")" type="text/javascript"></script>  
    <script src="@routes.Assets.versioned("javascripts/jquery/jquery-2.2.2.min.js")" type="text/javascript"></script>  
    <script src="@routes.Assets.versioned("javascripts/bootstrap/bootstrap.js")" type="text/javascript"></script>  
    <script src="@routes.Assets.versioned("javascripts/admin.js")" type="text/javascript"></script>  
    <script src="@routes.Assets.versioned("javascripts/customer.js")" type="text/javascript"></script>  
    <script src="@routes.Assets.versioned("javascripts/jquery/js.cookie.js")" type="text/javascript"></script>  
    <script async defer  
src="https://maps.googleapis.com/maps/api/js?key=AlzaSyAcX2vO7XHJBpoc9QabL_zZG1PXgca4GjU&callback=initMap"  
type="text/javascript"></script>  
  </head>  
  <body>  
    @content  
  </body>  
</html>
```

Ovaj šablon svaka stranica u OAF aplikaciji nasleđuje bilo direktno ili indirektno (obično se koristi još jedan među-šablon). Osnovni šablon se koristi za učitavanje svih potrebnih biblioteka. Kao što se može videti, tu su razne JavaScript i CSS biblioteke. HTML sadržaj se prosleđuje kao parametar na početku šablona – content: Html. Iz primera se vidi da se HTML kod stranice koja koristi ovaj šablon umeće u okviru sekcije body. Aplikacija OAF sadrži tri šablona koji nasleđuju osnovni šablon:

- adminMain.scala.html – koristi se za sve stranice u administratorskom režimu rada aplikacije.
- EmployeeMain.scala.html – koristi se za sve stranice u režimu rada zaposlenog.
- CustomerMain.scala.html – koristi se za sve stranice u korisničkom režimu rada.

Sledi detaljan opis **adminMain.scala.html** šablon:

```
@(title: String, user: User, currentForm: Option[Form[AnyRef]] = None)(content: Html)(implicit flash: Flash, lang: Lang)
```

```
@import play.api.i18n.Messages.Message
@main("Welcome to Play") {
  <div id="wrapper">
    <nav class="navbar navbar-inverse navbar-fixed-top" role="navigation">
      <div class="navbar-header">
        <button type="button" class="navbar-toggle" data-toggle="collapse" data-target=".navbar-ex1-collapse">
          <span class="sr-only">Toggle navigation</span>
          <span class="icon-bar"></span>
          <span class="icon-bar"></span>
          <span class="icon-bar"></span>
        </button>
        <a class="navbar-brand" href="@routes.CompanyController.get()">Order a food</a>
      </div>
      <ul class="nav navbar-right top-nav">
        <li>
          <a href="@routes.CompanyController.changeLanguage("en")">@Messages("language.english")</a>
        </li>
        <li>
          <a href="@routes.CompanyController.changeLanguage("sr")">@Messages("language.serbian")</a>
        </li>
        <li class="dropdown">
          <a href="#" class="dropdown-toggle" data-toggle="dropdown"> <span class="glyphicon glyphicon-user" aria-
            hidden="true"></span> @user.firstName @user.lastName<b class="caret"></b></a>
          <ul class="dropdown-menu">
            <li>
              <a href="#">@Messages("profile")</a>
            </li>
            <li class="divider"></li>
            <li>
              <a href="@routes.RegistrationController.logout()">@Messages("logout")</a>
            </li>
          </ul>
        </li>
      </ul>
      <div class="collapse navbar-collapse navbar-ex1-collapse">
        <ul class="nav navbar-nav side-nav">
          <li>
            <a href="@routes.CompanyController.get()">@Messages("company")</a>
          </li>
          <li>
            <a href="@routes.EmployeeController.getAll()">@Messages("employees")</a>
          </li>
          <li>
            <a href="@routes.ArticleController.getAll()">@Messages("articles")</a>
          </li>
          <li>
            <a href="@routes.AdditiveController.getAll()">@Messages("additives")</a>
          </li>
        </ul>
      </div>
    </nav>
    <div id="page-wrapper">
      <div class="container-fluid">
        @flash.get("success") match {
          case Some(message) => {
            <div class="alert alert-info">
              <a class="close" data-dismiss="alert">x</a>
              @Html(message)
            </div>
          }
          case _ => { }
        }
      </div>
    </div>
  </div>
}
```

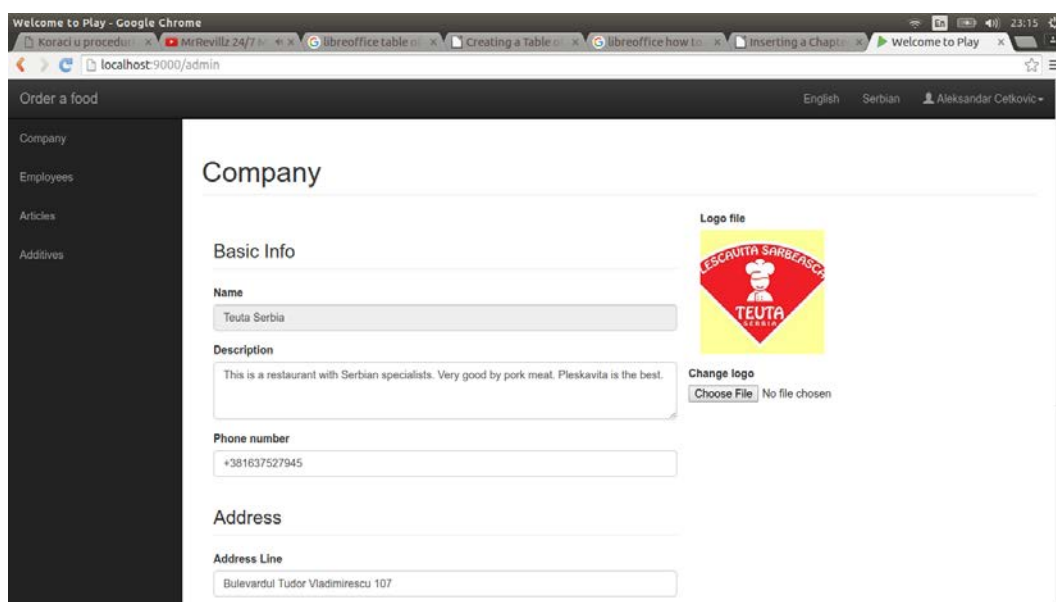
```

    @if(currentForm.isDefined){
      @if(currentForm.get.hasErrors) {
        <div class="alert alert-danger">
          <a class="close" data-dismiss="alert">x</a>
          @if(currentForm.get.errors.size > 0) {
            @for(error <- currentForm.get.errors) {
              @error.message
              <br>
            }
          } else {No error returned.}
        </div>
      }
    }
  }
  @content
</div>
</div>
}

```

Na početku stranice nalaze se ulazni parametri, a to su **title**, **user** i **currentForm**, kao i dva implicitna parametra **flash** i **lang**. Opis ulaznih podataka:

- o **User** – predstavlja trenutno prijavljenog korisnika i njegovi podaci se koriste za prikazivanje delova strane iz adminMain šablona koji se odnose na korisnika i njegov profil.
- o **Title** – predstavlja naslov strane koji će se videti na pregledaču
- o **CurrentForm** - predstavlja formu za validaciju, čija implementacija će biti prikazana u nastavku teksta. Na ovoj konkretnoj stranici njena primena je za prikazivanje greške, koje se takođe odvija u dnu adminMain šablonu.
- o **Flash** – je kolčić koji sadrži podatke koji su potrebni između dva zahteva. Najčešća primena je prikazivanje poruke o uspešno izvršenoj akciji. Bitno je napomenuti da se flash kolačić prazni pri svakom sledećem zahtevu ka serveru, dakle njegov životni vek je samo između dva zahteva.
- o **Lang** – ova promenljiva sadrži podatke o lokalizaciji. U sprezi sa @Messages(kod-poruke) određuje se na kom jeziku će poruka biti prikazana. Inače, Messages metoda pored koda poruke kao implicitni argument prima i lang i na osnovu toga određuje iz kog fajla će čitati poruku. Konkretno u ovoj aplikaciji postoje dva fajla, a samim tim i dva podržana jezika, a to su **messages** i **messages.rs**. Ovi fajlovi se nalaze na lokaciji OAF/conf i njihova struktura je slična rečniku: Kod-poruke=Poruka sa tim da oba fajla sadrže iste kodove.



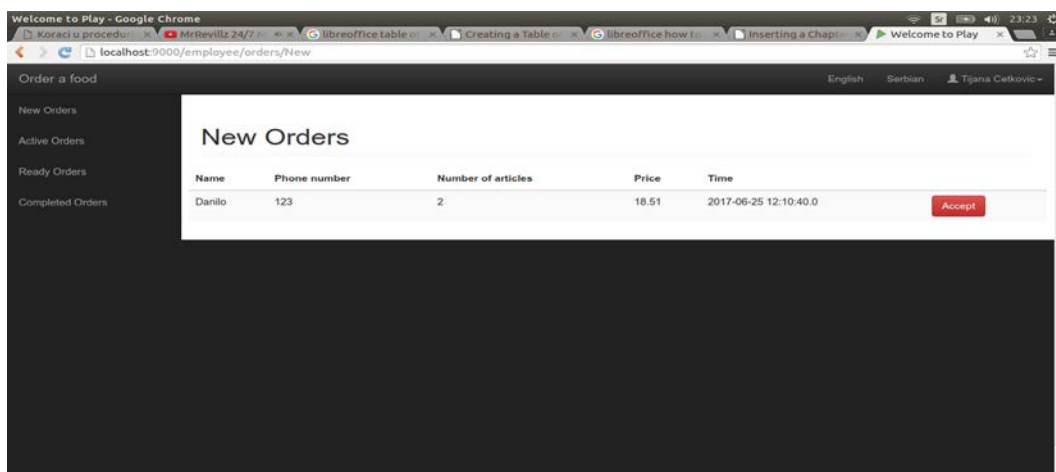
Slika 21. Izgled adminMain šablona

Izgled adminMain šablon je prikazan na slici (Slika 21), ono što čini šablon je:

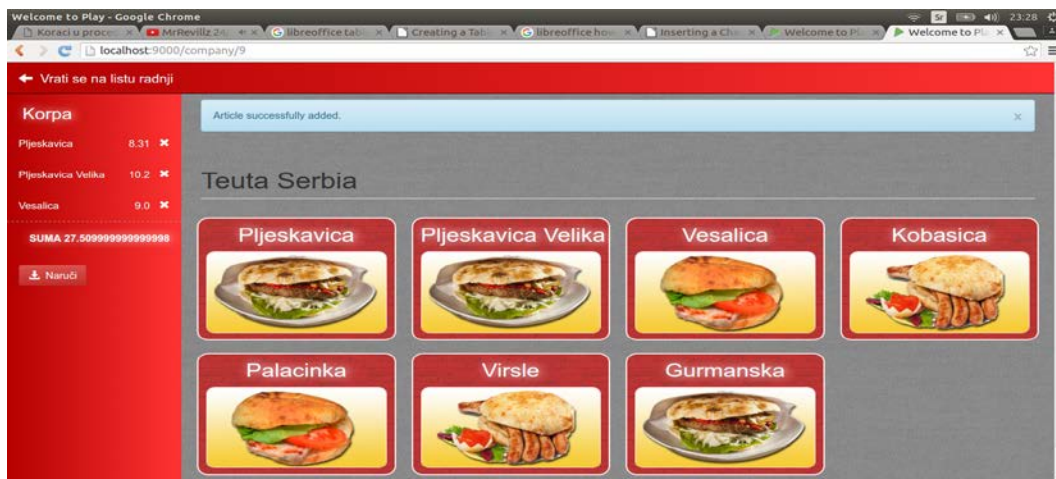
- Navigaciona linija sa leve strane
- Podaci o korisniku u gornjem desnom uglu
- Promena jezika levo od podataka o korisniku

Navigaciona linija se kod prikazivanja na manjem uređaju transformiše u dugme sa linkovima zbog bolje čitljivosti. Sve stranice u administratorskom režimu imaju isti okvir, samo se središnji (bela pozadina) deo menja. Ovo je konkretno stranica za promenu podataka o radnji (Company Edit).

Ostala dva šablona su slična samo se razlikuju linkovi u navigacionim linijama su prikazana na slikama (Slika 22 i Slika 23).



Slika 22. EmployeeMain šablon



Slika 23. CustomerMain šablon

4.4.3 Ubacivanje CSS i Javaskripta na HTML stranice

Ubacivanje CSS i Javaskript fajlova na HTML stranicu se odvija na uobičajen način.

```
<script src="@routes.Assets.at("javascripts/jquery.js")"></script>
<link rel="stylesheet" href="@routes.Assets.versioned("assets/css/app.css")">
```

bitno je samo navesti u **routes** fajlu gde se nalaze **asseti**, za ove konkretne primere to izgleda:

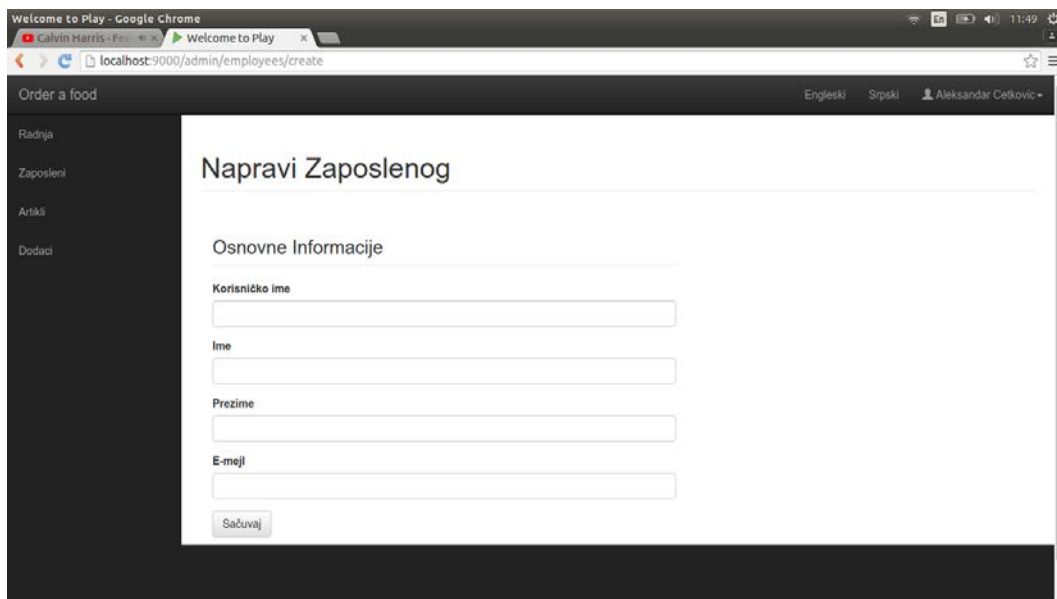
```
GET /assets/*file    controllers.Assets.at(path="/public", file)
GET /assets/*file    controllers.Assets.versioned(path="/public", file: Asset)
```

Na ovaj način kompajler pronalazi odgovarajuće fajlove i uključuje ih u HTML stranicu. Primeri su preuzeti sa [3].

4.4.4 Kompletan primer korišćenja kontrolera, forme i stranice

Ovaj primer je deo OAF aplikacije. Posmatraćemo primer pravljenja novih zaposlenih.
HTML strana (lokacija: OAF/app/views/admin/employeeCreate.scala.html)

```
@(user: User, employeeForm: Form[AnyRef])(implicit flash: Flash, lang: Lang)
@import helper._
@admin.adminMain("Welcome to Play", user, Some(employeeForm)) {
  <h1 class="page-header">
    @Messages("employee.create")
  </h1>
  <div class="col-lg-7">
    <h3 class="page-header">
      @Messages("basic.info")
    </h3>
    @helper.form(action = routes.EmployeeController.create()){
      <input name="companyId" value=@user.companyId.get class="form-control" type="hidden">
      <div class="form-group">
        <label>@Messages("username")</label>
        <input name="username" class="form-control" required>
      </div>
      <div class="form-group">
        <label>@Messages("firstName")</label>
        <input name="firstName" class="form-control">
      </div>
      <div class="form-group">
        <label>@Messages("lastName")</label>
        <input name="lastName" class="form-control" required>
      </div>
      <div class="form-group">
        <label>@Messages("email")</label>
        <input class="form-control" name="email" type="email" required>
      </div>
      <input type="submit" class="btn btn-default" value="@Messages("button.save")">
    }
  </div>
}
```



Slika 24. Stranica za pravljenje novog zaposlenog

Stranica `createEmployee` koristi šablon `adminMain`. `AdminMain` šablon obezbeđuje prikazivanje navigacionih menija (bočni i horizontalni), što je detaljnije objašnjeno u prethodnom tekstu. Sama stranica `createEmployee` predstavlja formu za unos novog zaposlenog (bela pozadina).

Stranica sadrži formu sa poljima za unos podataka o zaposlenom. Za formu se vezuje akcija kontrolera koja će se izvršiti kada korisnik klikne na dugme "Sačuvaj".

```
@helper.form(action = routes.EmployeeController.create())
```

Na ovaj način se definiše akcija koja će se izvršiti pri submitovanju forme. U ovom konkretnom slučaju izvršiće se akcija `create` iz `EmployeeController`-a. Promenljiva `routes` je globalna promenljiva Play okriženja, koja vrši mapiranje akcije kontrolera u URL akcije i obrnuto, oslanjajući se na `conf/routes` fajl, koji je bio opisan u prethodnom tekstu. Konkretno u ovom slučaju biće pozvana akcija:

POST `/admin/employees/create`

Sa podacima koji su uneti u formi.

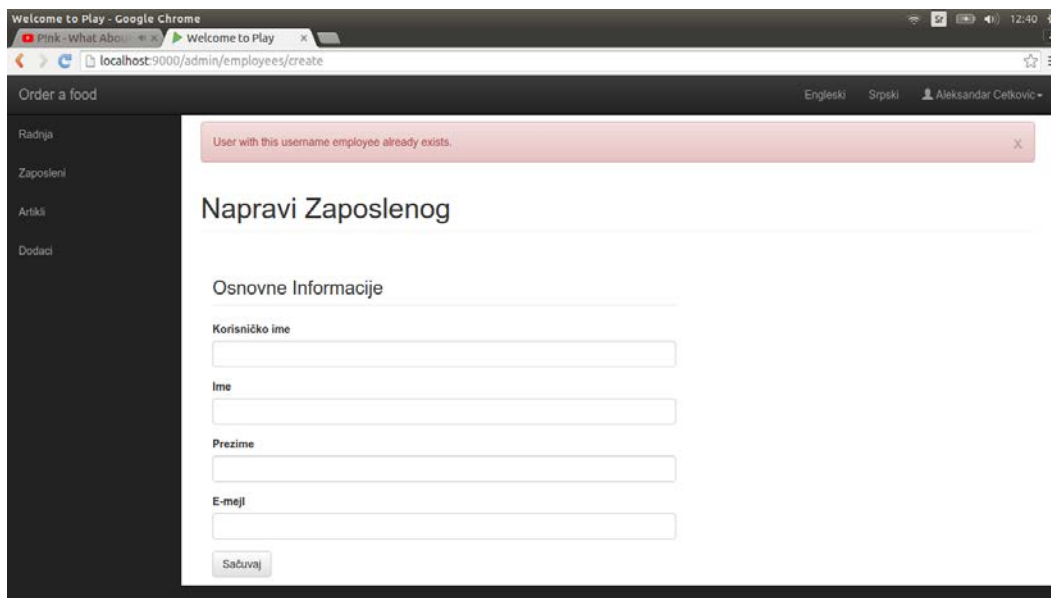
Kod pozvane akcije:

```
def create() = StackAction(AuthorityKey -> Administrator) { implicit request =>
  CreateEmployeeForm.getCreateEmployeeData.bindFromRequest.fold(
    formWithErrors => {
      BadRequest(views.html.admin.employeeCreate(loggedIn, formWithErrors.asInstanceOf[Form[AnyRef]]))
    },
    createEmployeeData => {
      UserService.createEmployee(createEmployeeData)
      Redirect(routes.EmployeeController.getAll()).flashing("success" -> "Employee successfully created.")
    }
  )
}
```

Akcija ima vrlo jednostavan izgled, prvo se podaci iz zahteva vezuju za formu, u ovom slučaju to je CreateEmployeeForm, forma izvrši validaciju i u zavisnosti od toga da li su podaci validni ulazi u odgovarajuću granu. Izgled forme za validaciju:

```
case class CreateEmployeeData
(
  firstName: String,
  lastName: String,
  username: String,
  email: String,
  companyId: Long
)
object CreateEmployeeForm {
  def getCreateEmployeeData = Form(
    mapping(
      "firstName" -> nonEmptyText,
      "lastName" -> nonEmptyText,
      "username" -> nonEmptyText.verifying(checkUsernameUnique),
      "email" -> email,
      "companyId" -> longNumber
    )(CreateEmployeeData.apply)(CreateEmployeeData.unapply)
  )
}
```

Validacija zahteva da polja firstName, lastName i username budu popunjena korišćenjem nonEmptyText ugrađenog validatora. Username takođe mora biti jedinstven, to proverava definisana checkUsernameUnique metoda. Ova metoda nije ugrađena i definisana je u Constraints.scala klasi. Polje email mora da zadovolji ugrađenu email validaciju. Ako validacija prođe bez grešaka getCreateEmployeeData metoda vraća kao rezultat instancu case klase CreateEmployeeData sa popunjenim podacima koju kontroler kasnije koristi u daljem izračunavalju. U slučaju greške pri validaciji, metoda vraća formu sa listom grešaka koje se kasnije prikazuju na stranici (Slika 25).



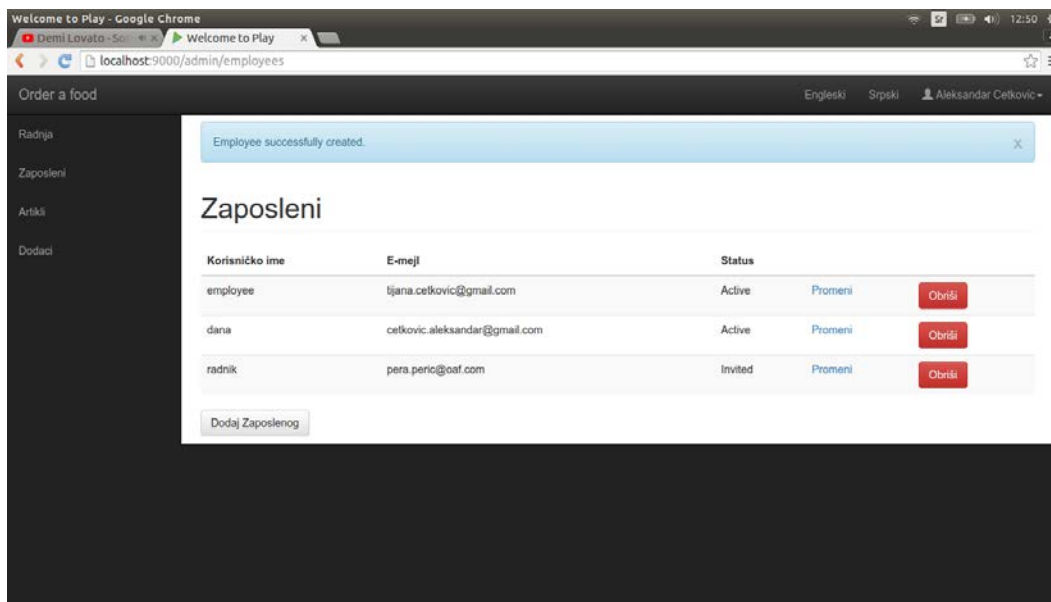
Slika 25. Prikazivanje greške pri neuspejoj validaciji podataka

U slučaju da validacija uspešno prođe, dolazi do poziva servisa:
UserService.createEmployee(createEmployeeData)

UserService pravi novog zaposlenog u bazi na osnovu podataka iz instance case klase createEmployeeData. O ovome će biti više reči u delu 4.5 rad sa bazom. Nakon što podaci budu sačuvani izvršava se redirekcija sa odgovarajućom porukom.

```
Redirect(routes.EmployeeController.getAll()).flashing("success" -> "Employee successfully created.")
```

Redirect je ugrađena metoda, koja u ovom konkretnom slučaju vrši redirekciju na stranicu gde su izlistani svi zaposleni sa porukom u flash kolačiću o uspešno napravljenom zaposlenom.



Slika 26. Uspešno napravljen radnik i redirekcija na drugu stranu

4.5 Rad sa bazom podataka

Comment [VF4]: bazom podataka

Baza podataka je sastavni deo svake veb aplikacije. Postoje razni alati koji olakšavaju rad sa bazom u Skali.

U ovom odeljku akcenat je na relacionim bazama. **Slick** je najčešće korišćeni alat za rad sa bazom. **Play** okruženje je kompatibilno sa **Slick** alatom i pruža dosta dodatnih mogućnosti.

4.5.1 Upoznavanje sa slick-om

Instalacija **Slick**-a je veoma jednostavna:

U okviru Build.sbt fajla potrebno je dodati zavisnost:

```
libraryDependencies += "com.typesafe.play" %% "play-slick" % "2.0.0"
```

Play-slick nema u sebi ugrađen *jdbc* drajver, tako da je u ovom slučaju potrebno dodati

odgovarajući drajver:

```
libraryDependencies += "mysql" % "mysql-connector-java" % "5.1.34"
```

Ovo je konkretan primer *jdbc* drajvera za **mysql** bazu podataka. Ako korisnik koristi neku drugu relacionu bazu, odgovarajući drajver može potražiti na zvaničnom *Slick* sajtu.

Da bi konfiguracija bila kompletna, potrebno je još u okviru *Application.conf* navesti sledeće parametre:

```
slick.dbs.default.driver = "slick.driver.MySQLDriver$"
slick.dbs.default.db.driver = "com.mysql.jdbc.Driver"
slick.dbs.default.db.url = "jdbc:mysql://localhost/playScalaSlickExample"
slick.dbs.default.db.user = "root"
slick.dbs.default.db.password = ""
```

Play-slick ima dodatnu mogućnost, a to je pravljenje evaluacionih skripti. Naime, veb aplikacija se tokom vremena širi, a samim tim i njena baza. Ono što predstavlja dodatni problem, jeste što u zavisnosti na kojoj verziji veb aplikacije se razvija dodatna funkcionalnost potrebna je odgovarajuća verzija baze podataka. Ovo je jako teško ispratiti i postoje razni dogovori u okviru tima, kako pratiti razvoj baze, ali svaki od dogovora je podložan greškama. Upustvo za konfiguraciju je preuzeto sa [4].

Evaluacione skripte, rešavaju taj problem. Pri svakoj promeni modela baze podataka, *Play* napravi evaluacionu skriptu, imenuje je sa rednim brojem (počinje se od 1.sql).

Ova skripta sadrži dva dela:

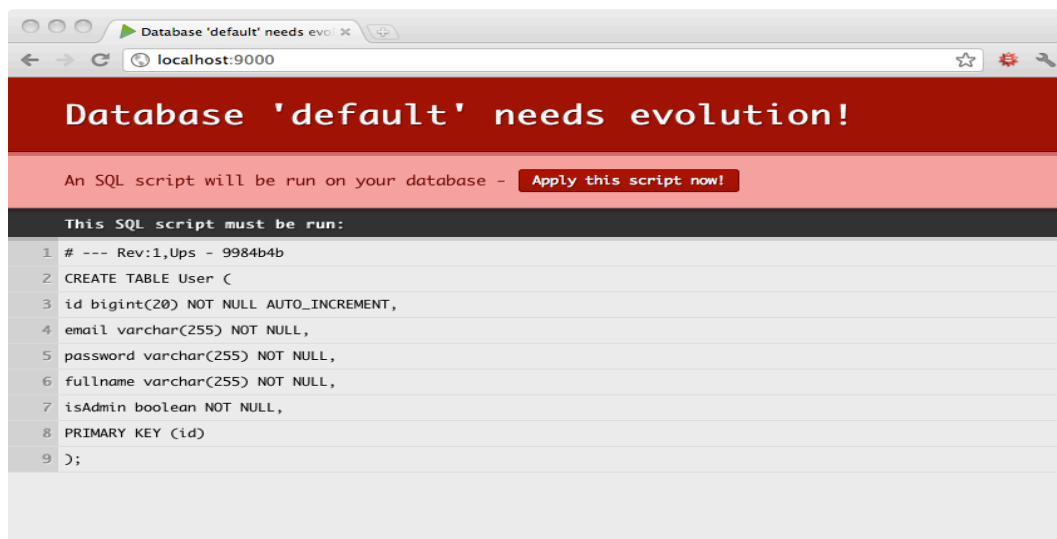
1. UPS

Ovo su promene koje treba izvršiti na bazi (npr. dodati novu kolonu ili celu tabelu)

2. DOWNS

Ovo su promene koje je potrebno izvršiti da bi se poništile promene iz UPS - ovo se koristi u slučajevima kada programer želi da se vrati na prethodnu verziju baze podataka.

Kada programer pokreće *Play* server, pre startovanja aplikacije u pretraživaču play okruženje ponudi izvršavanje evaluacionih skripti. Jednim klikom na dugme, sve skripte će biti izvršene i programer će imati saglasnu verziju baze sa verzijom aplikacije. (Slika 27)



Slika 27. Izvršavanje evaluacionih skripti

4.5.2 Kreiranje modela

Definisanje modela baze podataka biće ispraćeno kroz primer Article modela OAF aplikacije.

```
case class ArticleDBModel(id: Option[Long],
                          name: String,
                          description: Option[String],
                          price: Double,
                          pictureLocation: String,
                          weight: Option[Double],
                          companyId: Long,
                          status: ArticleStatus.Value,
                          group: ArticleGroup.Value)

class ArticleTable(tag: Tag) extends Table[ArticleDBModel](tag, "ARTICLE") {

  def id = column[Long]("id", O.PrimaryKey, O.AutoInc)
  def name = column[String]("name", O.NotNull)
  def description = column[Option[String]]("description", O.NotNull)
  def price = column[Double]("price", O.Nullable)
  def pictureLocation = column[String]("picture_location", O.Nullable)
  def weight = column[Option[Double]]("weight", O.Nullable)
  def companyId = column[Long]("company_id", O.NotNull)
  def status = column[ArticleStatus.Value]("status", O.NotNull)
  def group = column[ArticleGroup.Value]("group", O.NotNull)

  def * = (id?, name, description, price, pictureLocation, weight, companyId, status, group) <> (ArticleDBModel.tupled,
  ArticleDBModel.unapply _)
}
```

Ovo je jedan primer pravljenja modela. Nasleđuje se klasa **Table**, koja kao argument prima ime table. Polja se definišu korišćenjem operatora **column[Tip-podatka]**, u nastavku idu argumenti. Prvi argument predstavlja ime kolone u tabeli, a drugi neko posebno značenje, kao što je primarni ključ, podrazumevana vrednost, ne dozvoliti *null*... Na kraju ide definicija projekcije, koja se obeležava sa **"*"**. Da bi se validno koristili podaci u daljem kodu, potrebno je napraviti case klasu i na nju mapirati projekciju. U daljem kodu se koristi case klasa.

4.5.3 Pisanje upita

Prvo je potrebno definisati listu podataka za jedan entitet.

```
val articles = TableQuery[ArticleTable]
```

U daljem kodu **articles** predstavlja validnu Scala listu i na nju se mogu primeniti sve funkcije za rad sa listama.

Ekvivalent **where** klauzuli je **filter** funkcija:

```
val q = articles.filter( article => article.price > 8.0)
```

Ekvivalent elementima iz **select** linije je funkcija **map**

```
val q = articles.map( article => article.name)
```

Ono što je bitno, da se funkcije za rad sa listama mogu komponovati.

```
val q = articles.filter(_.price > 8.0).map(_.name)
```

Postoje još i metode za sortiranje, uzimanje n prvih elemenata, uzimanje n poslednjih elemenata. Detaljan opis se nalazi na [4].

Spajanje tabela ili poznatije JOIN operator iz SQL-a. Ovo se radi uz pomoć operatora *for* i *yield*:

Za ovaj primer je korišćen i entitet Company iz aplikacije OAF. Definicija Company entiteta:

```
case class CompanyDBModel(id: Option[Long],
                           name: String,
                           description: String,
                           addressId: Long,
                           logoUrl: String,
                           phoneNumber: String,
                           ownerId: Long)

class CompanyTable(tag: Tag) extends Table[CompanyDBModel](tag, "COMPANY") {
  def id = column[Long]("id", O.PrimaryKey, O.AutoInc)
  def name = column[String]("name", O.NotNull)
  def description = column[String]("description", O.NotNull)
  def addressId = column[Long]("address_id", O.NotNull)
  def logoUrl = column[String]("logo_url", O.NotNull)
  def phoneNumber = column[String]("phone_number", O.NotNull)
  def ownerId = column[Long]("owner_id", O.NotNull)
  def * = (id.?, name, description, addressId, logoUrl, phoneNumber, ownerId) <> (CompanyDBModel.tupled,
CompanyDBModel.unapply _)
}
```

Sledi primeri sa ekvivalentnim SQL kodom.

```
val crossJoin = for {
  (a, c) <- articels join companies
} yield (a.name, c.name)
// ekvivalentan SQL upit:
// select x2."name", x3."name" from "ARTICLE" x2
// inner join "COMPANY" x3

val innerJoin = for {
  (a, c) <- articels join companies on (_.companyId === _.id)
} yield (a.name, c.name)
// ekvivalentan SQL upit:
// select x2."name", x3."name" from "ARTICLE" x2
// inner join "COMPANY" x3
// on x2."companyId" = x3."id"

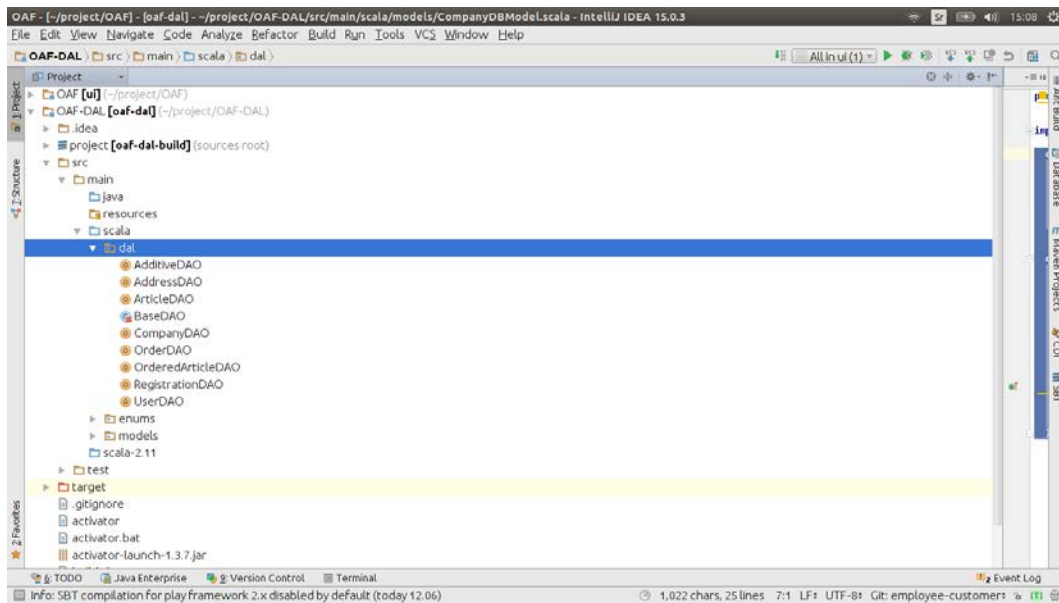
val leftOuterJoin = for {
  (a, c) <- articels joinLeft companies on (_.companyId === _.id)
} yield (a.name, c.map(_.name))
// ekvivalentan SQL upit:
// select x2."name", x3."name" from "ARTICLE" x2
// inner join "COMPANY" x3
// on x2."companyId" = x3."id"
```

Primeri su delo autora rada inspirisani primerima iz [4] .

4.5.4 Rad sa bazom u OAF aplikaciji

Aplikacija OAF sadrži poseban projekat OAF-DAL koji je predstavlja sloj podataka. (eng. Data Access Layer). Implementiran je kao običan scala projekat (nije Play projekat) i

koristi se isključivo za rad sa bazom. Struktura projekta je sledeća:



Slika 28. Struktura OAF-DAO projekta

Sadrži tri foldera:

- **dal**
- **enums** – Potrebni numerički tipovi
- **models** – Definicija svih entiteta u aplikaciji

Za svaki entitet postoji odgovarajuća DAO klasa u okviru dal foldera. Opis DAO klase biće urađen kroz primer ArticleDAO.scala:

```
object ArticleDAO extends BaseDAO[ArticleDBModel, Long, Session]{
  def findAll(implicit session: Session): List[ArticleDBModel] = {
    articles.filter(a => a.status === ArticleStatus.Active).list
  }
  def update(id: Long, model: ArticleDBModel)(implicit session: Session): Unit = {
    articles.filter(_id === id).update(model)
  }
  def findById(id: Long)(implicit session: Session): Option[ArticleDBModel] = {
    articles.filter(_id === id).firstOption
  }
  def findByNameAndCompany(name: String, companyId: Long)(implicit session: Session): Option[ArticleDBModel] = {
    articles.filter(article => article.companyId === companyId && article.name === name).firstOption
  }
  def delete(id: Long)(implicit session: Session): Unit = {
    val q = for { u <- articles if u.id === id } yield u.status
    q.update(ArticleStatus.Inactive).run
  }
  def create(model: ArticleDBModel)(implicit session: Session): Long = {
    (articles returning articles.map(_id)) += model
  }
  def findAllByCompanyId(companyId: Long)(implicit session: Session): List[ArticleDBModel] = {
    articles.filter(a => a.status === ArticleStatus.Active && a.companyId === companyId).list
  }
}
```

Sve DAO klase nasleđuju dinamički tipiziranu BaseDAO abstraktnu klasu. BaseDAO klasa zahteva tip-modela, tip-id-a i tip sesije za rad sa bazom. U ovom konkretnom slučaju to su **ArticleDBModel**, **Long** i **Session**. BaseDAO abstraktna klasa zahteva I implementaciju

sledećih metoda:

```
def findAll(implicit session: Session): List[Model]
def findById(id: ID)(implicit session: Session): Option[Model]
def create(model: Model)(implicit session: Session): ID
def update(id: ID, model: Model)(implicit session: Session): Unit
def delete(id: ID)(implicit session: Session): Unit
```

Na ovaj način svaka nova DAO klasa koja nasleđuje BaseDAO mora da obezbedi osnovne operacije za entitet kojem odgovara. U slučaju ArticleDAO klase to je entitet Article. Funkcije za dohvaćanje podataka su findAll, findById, findAllByCompanyId i findByNameAndCompany. Implementacije ovih metoda su slične, sve se zasnivaju na izvršavanju filter funkcije nad articles kolekcijom koja predstavlja tabelu ARTICLE iz baze. Povratni tip funkcija za dohvaćanja podataka je različit u zavisnosti od toga da li funkcija vraća jedan ili više objekata.

```
def findAllByCompanyId(companyId: Long)(implicit session: Session): List[ArticleDBModel] = {
  articles.filter(a => a.status === ArticleStatus.Active && a.companyId === companyId).list
}
```

Ova funkcija vraća listu svih aktivnih artikala kompanije čiji id je prosleđen.

```
def findById(id: Long)(implicit session: Session): Option[ArticleDBModel] = {
  articles.filter(_id === id).firstOption
}
```

FindById funkcija vraća artikal sa odgovarajućim id-em. Treba primetiti da se u ovom slučaju koristi Option klasa, koja u slučaju da artikal sa datim id-em ne postoji vraća None. Funkcija za brisanje artikla po id-u je implementirana kao funkcija za promenu vrednosti, gde se artiklu sa prosleđenim id-em menja status na neaktivan.

```
def update(id: Long, model: ArticleDBModel)(implicit session: Session): Unit = {
  articles.filter(_id === id).update(model)
}
```

Metoda za menjanje entiteta prima id i instancu modela. Pronalazi entitet u bazi po id-u i menja mu sva polja sa poljima iz prosleđenog modela. Funkcija za kreiranje novog modela dodaje instancu modela na articles listu – što kasnije Slick dodaje u SQL tabelu ARTICLE.

4.5.5 Rad sa ostalim bazama podataka

Comment [VF5]: bazama podataka

Scala podržava čitav skup različitih baza podataka. Od relacionih treba izdvojiti PostgreSQL. Slick podržava ovu bazu, treba samo uključiti odgovarajući drajver.

Scala podržava i NO-SQL baze podataka. Treba izdvojiti MongoDB kao jednu od najpopularnijih NO-SQL baza. Slick je koncipiran samo za rad sa relacionim bazama, tako da ne podržava MongoDB.

Međutim postoji dosta drugih alata koji podržavaju MongoDB, na primer Reactive Mongo [5].

4.6 Čitači i pisači

JSON format je jedan od najzastupljenijih formata za razmenu podataka između veb aplikacija, ali i u samoj aplikaciji. Razlog tome je što REST servisi koriste ovaj format. U starijim veb aplikacijama korišćen je XML format, koji je podržan od strane SOAP veb servisa.

U ovom odeljku su izložene osnovne stvari o elementima jezika Scala koji omogućavaju rad sa podacima u JSON format.

4.6.1 Čitanje JSON formata (čitač)

Čitač (eng. Reader) je objekat koji omogućava parsiranje JSON objekta u Scala objekat. Ovo drastično olakšava posao, jer programer nije dužan da za svaki JSON objekat pravi parser, dovoljno je samo napraviti Čitač.

Primer:

Resident je case klasa koja će u daljem kodu sadržati vrednost JSON-a

```
implicit val residentReads: Reads[Resident] = (  
  (JsPath \ "name").read[String](minLength[String](2)) and  
  (JsPath \ "age").read[Int](min(0) keepAnd max(150)) and  
  (JsPath \ "role").readNullable[String]  
)(Resident.apply _)
```

Sintaksa je jednostavna, sa leve strane u okviru zagrada navodi se JSON putanja do određene vrednosti.

Nakon toga zove se **read** metoda sa tipom podatka koji se očekuje. Nakon toga se opciono može navesti validacija samog podatka.

4.6.2 Pisanje objekata u JSON (pisač)

Pisač (eng. Writer) je objekat suprotan Čitaču. On od scala objekta pravi JSON objekat.

Primer:

```
implicit val residentWrites: Writes[Resident] = (  
  (JsPath \ "name").write[String] and  
  (JsPath \ "age").write[Int] and  
  (JsPath \ "role").writeNullable[String]  
)(unlift(Resident.unapply))
```

Sintaksa je analogna sintaksi Čitača.

Ono što je ovde zanimljivo jeste činjenica da su u ovim primerima čitač i pisač definisani kao implicitne vrednosti. Što znači da kompajler sam prepoznaje kada treba konvertovati objekat **Resident** u JSON i obrnuto. Na ovaj način se dosta smanjuje količina koda koji je potrebno napisati, što smanjuje verovatnoću greške, a i kod postaje više čitljiv. Svi primeri su preuzeti sa [3].

4.7 REST servisi

U svetu veb aplikacija često se koristi neki eksterni servis za neko izračunavanje ili uslugu. Ponekad je i sama veb aplikacija podeljena u mikroservise koji međusobno komuniciraju. Korišćenje eksternih servisa je jako korisno.

Neka bude razmotren primer slanja email poruke. Postoje mnogi provajderi koji nude slanje *email* poruke, a neka ovde bude izdvojen **Mandril**. Potrebno je registrovati se na

Mandril sajtu, tamo napraviti jedan ili više šablona za *email*. Onda programmer u okviru svoje veb aplikacije ispogramira slanje POST zahteva ka *Mandril*-u sa odgovarajućim kredencijalima i potrebnim podacima. *Mandril* na osnovu podataka, koje je dobio od veb aplikacije, bira šablon, popuni adrese i pošalje *email*.

Na ovaj način, programeru nije potrebno da konfiguriše svoj email server, niti dodatna konfiguracija, ni dodatno održavanje. Potrebno je samo platiti korišćenje *Mandril*-a, što je u redu u velikom broju poslovnih scenarija, onda kada programer u startu ne zna kakvu ekspanziju može doživeti veb aplikacija. U nekom trenutku je potrebno više email servera, u nekom manje, pa je onda rizično kupovati neki server više. U slučaju *Mandril*-a korisnik plaća onoliko koliko je potrošio, slično kao što stanovnici plaćaju struju.

Ovo je bio samo primer, ali se isti princip može koristiti i za neke druge eksterne servise.

4.7.1 Kreiranje zahteva

Scala omogućava pravljenje HTTP zahteva prema eksternim servisima preko klase **WS**. Sledi primer za GET zahtev:

```
val request: WSRequest = ws.url(url)

val complexRequest: WSRequest =
  request.withHeaders("Accept" -> "application/json")
    .withRequestTimeout(10000.millis)
    .withQueryString("search" -> "play")

val futureResponse: Future[WSResponse] = complexRequest.get()
```

Rezultat je objekat tipa *Future*, o kome će biti više reči u sledećem odeljku.

Primer za POST:

```
import play.api.libs.json._
val data = Json.obj(
  "key1" -> "value1",
  "key2" -> "value2"
)
val futureResponse: Future[WSResponse] = ws.url(url).post(data)
```

Obično eksterni servisi zahtevaju autentikaciju, pa je potrebno poslati kredencijale. To se radi na sledeći način:

```
ws.url(url).withAuth(user, password, WSAuthScheme.BASIC).get()
```

4.7.2 Obrada odgovora

Obrada dobijenog rezultata od eksternog servisa je malo složenija stvar. Najveći problem predstavlja to što naša aplikacija ne zna kada će odgovor stići. Iz tog razloga se koristi **Future[Tip-podatka]**. Naime *Future* ne blokira dalje izvršavanje koda, ali i nema vrednost, pa je sam *Future* objekat potrebno mapirati.

```
val futureResult: Future[String] = ws.url(url).get().map {
  response =>
    (response.json \ "person" \ "name").as[String]
}
```

Samim mapiranjem se ulazi u blok koji se izvršava kada *Future* dobije vrednost, odnosno kada eksterni servis vrati rezultat.

Više reči o *Future* objektima biće u narednom poglavlju.

Svi primeri su preuzeti sa [3] .

4.8 Asinhrona akcije

Asinhrona akcije omogućavaju neblokirajuću obradu zahteva. Na ovaj način se postiže bolja iskorišćenost procesorskog vremena na server. U ovom odeljku je opisan rad sa ovakvim tipom akcija.

4.8.1 Future objekat

Future daju mogućnost paralelnog izvršavanja više operacija- i to na efikasan neblokirajući način.

Future je objekat koji očekuje vrednost (eng. placeholder) koja još uvek ne postoji. Generalno, vrednost *Future* objekta se dobija konkurentno i naknadno može se koristiti.

Komponovanjem više konkurentnih zadataka na ovakav način doprinosi dobijanju rezultata na brži, asinhroni, neblokirajući način.

```
val inverseFuture: Future[Matrix] = Future {
  fatMatrix.inverse() // non-blocking long lasting computation
}(ExecutionContext)
```

Vrednost *Future* objekta je obično rezultat nekog drugog izračunavanja:

1. Ako izračunavanje nije završeno, kažemo da *Future* nije kompletan.
2. Ako je izračunavanje završeno uspešno ili sa greškom, kažemo da je *Future* kompletan.

Kompletnost *Future* objekta može imati dve forme:

1. Kada je *Future* kompletiran i dobio je vrednost, kažemo da je uspešno kompletiran.
2. Kada je *Future* kompletiran, ali nije dobio vrednost, zato što se u izračunavanju pojavio izuzetak, kažemo da je *Future* omašio. (eng. Failed)

Postoje razni načini kako možemo dohvatiti vrednost *Future* objekta. Evo jednog primera:

```
import scala.util.{Success, Failure}

val f: Future[List[String]] = Future {
  session.getRecentPosts
}
f onComplete {
  case Success(posts) => for (post <- posts) println(post)
  case Failure(t) => println("An error has occurred: " + t.getMessage)
}
```

4.8.2 Blokirajuće, neblokirajuće

Play okruženje nudi dva tipa akcija, sinhrona (blokirajuće) i asinhrona (neblokirajuće). Sinhrona su bile obrađene u delu 4.3.1. U ovom odeljku su obrađene asinhrona akcije.

Osnovna razlika je u rezultatu koji vraća akcija. Kod asinhronih akcija povratni tip je *Future[Result]*, dok je kod sinhronih bio *Result*. Da bi se koristile asinhrona akcije potrebno je da sva izračunavanja u okviru akcije budu asinhrona. Na ovaj način se postiže maksimalna efikasnost paralelnog izvršavanja.

```
import play.api.libs.concurrent.Execution.Implicits.defaultContext

val futurePIValue: Future[Double] = computePIAsynchronously()
val futureResult: Future[Result] = futurePIValue.map { pi =>
  Ok("PI value computed: " + pi)
}
```

Neki dodaci za *Play* okruženje su implementirani tako da izračunavanje bude asinhrono, tj. da je povratni tip tipa *Future*.

Na primer, *Reactive MongoDB* dodatak je tako implementiran [5]. Ovo programera navodi da u svakom sloju aplikacije koristi *Future* kao povratni tip. Ovakav način rada je malo teži u početku, ali se kasnije lako koristi. Na ovaj način se postiže velika efikasnost, što je jako bitno u razvoju veb aplikacija.

4.8.3 Sinhronizacija više asinhronih rezultata

U situaciji kada imamo više asinhronih poziva, a njihove rezultate treba kombinovati koristi se *FOR*.

```
val futureResponse: Future[WSResponse] = for {
  responseOne <- ws.url(urlOne).get()
  responseTwo <- ws.url(responseOne.body).get()
  responseThree <- ws.url(responseTwo.body).get()
} yield responseThree

futureResponse.recover {
  case e: Exception =>
    val exceptionData = Map("error" -> Seq(e.getMessage))
    ws.url(exceptionUrl).post(exceptionData)
}
```

Kao što se vidi iz primera, ako u *for* navedemo više asinhronih akcija, onda možemo smatrati kao da se one izvršavaju sekvencijalno.

Na primer, za dobijanje drugog rezultata koristimo rezultat prvog asinhronog poziva.

Na kraju *for* naredbe ide *yield* operator, koji govori šta treba izdvojiti kao rezultat cele *for* naredbe.

Primeri su preuzeti sa [6].

4.9 Osnovno o Akka-i

Akka predstavlja platformu za razvoj konkurentnih procesa u čijoj osnovi je Glumac (eng. *Actor*) model. Model Glumac predstavlja visok nivo abstrakcije za pisanje

konkurentnih i distribuiranih sistema. On ublažava razvoj konkurentnih sistema time što olakšava najveće probleme konkurentnog programiranja – deljenje resursa i sinhronizacija niti. Model Glumac je definisan još 1973. godine od strane Carl Hewitt, ali je svoju zapaženiju primenu dostigao u skorije vreme.

4.9.1 Koncept Glumac

Jedan Glumac predstavlja jednu nit. Glumac se implementira tako što se nasledi **Actor** osnovni *trait* i implementira metoda **receive**. *Receive* metoda treba da sadrži listu slučajeva i kako reagovati na svaki od njih. Jedan slučaj predstavlja jednu poruku. Poruka može biti bilo koji objekat, ali najčešće je to case klasa. *Receive* metoda sadrži standardan Scala upoređivač šablona, o kome je bilo reči u odeljku 4.1.3.

```
import akka.actor.Actor
import akka.actor.Props
import akka.event.Logging
class MyActor extends Actor {
    val log = Logging(context.system, this)
    def receive = {
        case "test" => log.info("received test")
        case _ => log.info("received unknown message")
    }
}
```

Iz primera se može videti da *MyActor* očekuje "test" poruku. Karakter "_" je univerzalni karakter i zamenjuje bilo koji drugi objekat.

Za instanciranje Glumaca koristi se klasa *Props*- ona je konfiguraciona klasa koja sadrži opcije koje su vezane za ceo sistem Glumaca.

```
import akka.actor.Props
val props1 = Props[MyActor]
val props2 = Props(new ActorWithArgs("arg"))
val props3 = Props(classOf[ActorWithArgs], "arg")
```

Treba imati u vidu da je klasa *Props* imutabilna. Instanciranje Glumaca korišćenjem *Props* klase se izvršava na sledeći način:

```
import akka.actor.ActorSystem
val system = ActorSystem("mySystem")
val myActor = system.actorOf(Props[MyActor], "myactor2")
```

Vrednost *myActor* se instancira korišćenjem *system* objekta, tačnije njegove metode

actorOf. Prvi argument je *Props* klasa, dok je drugi argument ime Glumca. Ovde treba napomenuti da je poželjno da aplikacija ima samo jedan *ActorSystem* pošto je ovo jako zahtevan objekat sa stanovišta memorije i procesorskog vremena.

Drugi način za pravljenje Glumaca je pravljenje deteta postojećeg Glumca.

```
class FirstActor extends Actor {  
  val child = context.actorOf(Props[MyActor], name = "myChild")  
}
```

4.9.2 Slanje poruka i obrada

Kao što je ranije rečeno poruka koja se prosleđuje Glumcu može biti bilo koji objekat. Glumac poruku obrađuje u okviru *receive* metode. Za slanje poruke koristi se operator *!*

referenca-glumca ! poruka

Poruka se može poslati iz bilo kog dela aplikacije, to može biti neki drugi Glumac, a može biti i neki nezavisni deo koda. Kada Glumac primi poruku, on može prepoznati pošaljioca, pozivom metode *sender()* tako da može poslati novu poruku pošaljiocu.

sender() ! "Hvala!"

U okviru Glumca postoji i promenljiva *self*, koja predstavlja referencu na sebe, tako da Glumac može poslati poruku sam sebi.

Poruka se može proslediti dalje (eng. Forward). Kada *actor1* primi poruku **message** on je na sledeći način može proslediti *actor2*:

actor2 ! message

Pri korišćenju *!* operatora pošaljioc poruke ne očekuje odgovor, on je samo poslao zahtev da se nešto izvrši i neće dobiti povratnu informaciju o tome. U tu svrhu postoji operator *?* (eng. Ask). Koristi se isto kao i *!* samo što pošaljioc poruke dobija odgovor od primaoca.

Ovaj odgovor je zapakovan u *Future* iz jasne namere da se konkurentni procesi, odnosno Glumci ne blokiraju.

```
import scala.concurrent.duration._
```

```
import akka.util.Timeout
```

```
import akka.pattern.ask
```

```
implicit val timeout = Timeout(5 seconds)
```

```
val future = myActor ? "hello"
```

Ovde se jasno vidi da je potrebno definisati i vreme čekanja (eng. Timeout). Ako se odgovor od *myActor*-a ne dobije u roku od 5 sekundi, doćiće do ispaljivanja izuzetka.

Na ovaj način se sprečava zaglavljivanje niti (eng. DeathLock). Nakon što se dobije

odgovor od Glumca, potrebno je izvući vrednost iz *Future* objekta, što je opisano ranije.

4.9.3 Životni vek Glumca

Postoje razni načini da se Glumac stopira. U kojim situacijama se to dešava:

1. Ako dođe do greške u radu Glumca
2. Ako je otac Glumca stopiran
3. Ako Glumac dobije poruku *PosionPill* – Ovo je statička poruka dostupna u celom sistemu

Bitno je spomenuti da AKKA-a omogućava i pravljenje distribuiranog sistema Glumaca. Ovo omogućava pravljenje sistema u kojima su niti distribuirane na različite servere, a sve one komuniciraju međusobno. Svaka nit (Glumac) ima utisak da su sve niti na istoj mašini (istom serveru) i ceo sistem funkcioniše kao običan sistem Glumaca u Skali. Ovo je jedna od najvećih prednosti koju Scala daje u konkurentnom programiranju. Naime, zbog ovog mehanizma se dosta podiže skalabilnost aplikacije [7]. Primeri su preuzeti iz [8] .

4.10 Veb soketi

Comment [VF6]: soketi
na svim mestima!!!

Do skoro, veb je podržavao samo komunikaciju u jednom smeru: pretraživač pošalje HTTP zahtev serveru i server vrati odgovor pretraživaču. Server nije imao mogućnost da pošalje bilo kakve podatke klijentu, ako pre toga nije dobio zahtev od klijenta. Za mnoge aplikacije ovo predstavlja problem. Jedan od standardnih primera, potrebe da server pošalje podatke klijentu bez prethodnog zahteva klijenta, je aplikacija za razmenu instant poruka (eng. Chat ili Instant Messenger).

Pojavili su se razni trikovi da se ovaj problem prevaziđe. Jedan od njih je poliranje (eng. Polling). Ovaj princip se zasniva na sledećem:

- Klijent periodično šalje HTTP zahtev ka serveru, interval može biti jedna sekunda
 - Server vraća praznu poruku u slučaju da nema podataka, a u suprotnom šalje podatke
- Ova metoda zahteva mnogo resursa u slučaju da treba postići odgovarajući odziv u aplikaciji za razmenu poruka.

Novi princip koji se javlja je **Comet**. Koncept ove metode je sledeći:

- Klijent šalje zahtev serveru
 - Server ne vraća odgovor, sve dok ne dobije potrebne podatke za klijenta.
- U odnosu na prethodnu metodu, koja šalje u 10 sekundi 10 zahteva, u Comet metodi klijent pošalje jedan zahtev, ali server čeka 10 sekundi (ako je toliko potrebno za dobijanje podataka) i tek nakon toga server pošalje odgovor.

U skorije vreme, pretraživači podržavaju novi standardizovani protokol za dvosmernu komunikaciju između klijenta i servera. Njen naziv je veb soket (eng. Web Socket). Veb soket zahtev se inicira regularnim HTTP zahtevom, sa tim što konekcija ostaje otvorena. Dok je konekcija otvorena, klijent i server mogu slati podatke jedan drugom. Ovim je omogućena dvosmerna komunikacija. Naravno, za uspostavljanje veb soket konekcije nije neophodan pretraživač, to se može ostvariti i kroz klijentsku aplikaciju na pametnom telefonu ili tabletu. U daljem tekstu je prikazano kako se u okviru Play okruženja koriste veb soketi.

4.10.1 Implementacija

Najčešća praksa implementacije veb soketa u okviru *Play* aplikacije je korišćenjem

Glumaca. Naime, osnovna ideja je da se pri otvaranju konekcije napravi jedan Glumac koji prati životni vek veb soketa. Samu implementaciju je izložena kroz sledeći primer:

```
def naiveActorWebSocket = WebSocket.using[String] {  
  val actor = Akka.system.actorOf(Props[NaiveEchoActor])  
  val out = Enumerator.imperative[String]()  
  actor ! NaiveStart(out)  
  val in = Iteratee.foreach[String] {  
    msg =>  
      actor ! Message(msg)  
  }  
  (in, out)  
}  
...  
case class NaiveStart(out: PushEnumerator[String])  
case class Message(msg: String)  
  
class NaiveEchoActor extends Actor {  
  var out: PushEnumerator[String] = _  
  
  override def receive = {  
    case NaiveStart(out) => this.out = out  
    case Message(msg) => this.out.push(msg)  
  }  
}
```

naiveActorWebSocket – predstavlja akciju kontrolera koja uspostavlja konekciju ka veb soku. Treba primetiti da ovo nije standardna akcija, nego **WebSocket.using[String]** za kojim sledi telo akcije. **WebSocket.using[String]** ovo znači da veb soket razmenjuje poruke koje su tipa string. Kao rezultat ove akcije vraća se kombinacija (**Iteratee**, **Enumerator**). **Iteratee** je kompleksan Scala objekat, koji se koristi uglavnom za tokove podataka (eng. Stream). Njegova uloga u implementaciji veb soketa je primanje i obrada poruke. **Enumerator** je takođe složen Scala objekat, koji se u ovom slučaju koristi kao kanal na koji se šalje poruka. Dakle, kombinacija (Iteratee, Enumerator) predstavlja (ulaz, izlaz) i kao takva predstavlja implementacionu reprezentaciju veb soketa.

NaiveEchoActor – predstavlja Glumca koji prati rad veb soketa. Glumac poseduje promenljivu **out** tipa **Enumerator**. Ova promenljiva se inicijalizuje slanjem poruke **NaiveStart(out)** što se događa u trenutku uspostave konekcije.

```
val in = Iteratee.foreach[String] {  
  msg =>  
    actor ! Message(msg)  
}
```

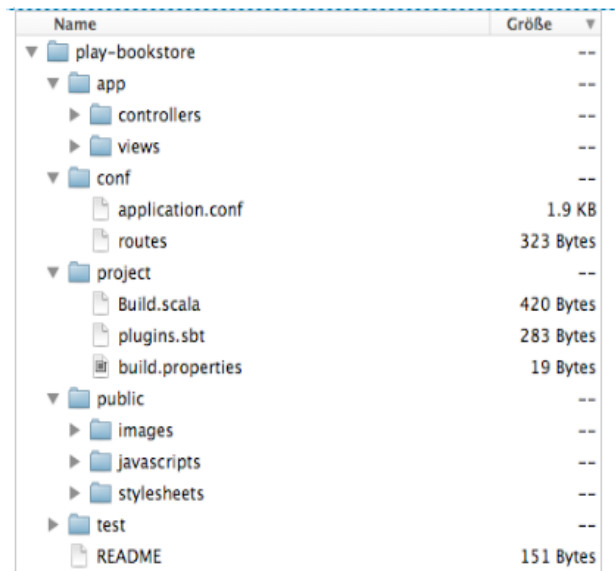
Ovaj deo primera, predstavlja inicijalizaciju **Iteratee** objekta. Pošto je primer jednostavni eho veb soket, iz implementacije zaključuje se da je svaka poruka koja stigne na veb soket prosleđena Glumcu. Glumac na ovu poruku reaguje tako što prosleđuje identičnu poruku na **Enumerator**. Ovo je jednostavan primer, ali jako lako je proširiv, baš zbog jednostavne sintakse Glumaca. Naime, dovoljno je dodati još neki tip poruke i na taj način se može postići kompleksnija funkcionalnost. Opisani primer je preuzet sa [9] .

4.11 Testovi jedinica

Comment [VF7]: obrisati!

Testovi jedinica predstavljaju jednu od bitnijih promenu u novijoj istoriji veb aplikacija, to je razmatrano u delu 2.4. Scala u paketu sa *Play* okruženjem pruža mogućnost pravljenja automatskih testova. U strikturi inicijalnog programa, može se videti folder **test**.

Comment [VF8]: , [to je razmatrano



Name	Größe
play-bookstore	--
app	--
controllers	--
views	--
conf	--
application.conf	1.9 KB
routes	323 Bytes
project	--
Build.scala	420 Bytes
plugins.sbt	283 Bytes
build.properties	19 Bytes
public	--
images	--
javascripts	--
stylesheets	--
test	--
README	151 Bytes

Slika 29. Test folder u okviru Play projekta

U okviru *test* foldera smeštaju se testovi jedinica. Svaki test predstavlja klasu koja nasleđuje *trait Specification*. Inače, zahvaljujući samom interfejsu *Specification*, testovi u Skali su jako čitljivi i lako se pišu. Naime, pišu se jako slično kao rečenice u prirodnom jeziku, sve to zahvaljujući prefiksnoj notaciji programskog jezika Scala. Scala ne zahteva korišćenje operatora “.” kada se nad nekim objektom zove operator ili funkcija.

```
Article match { ... }
```

Iz tog razloga, kod pisan u Skali postaje sličan nekom prirodnom jeziku. Ova sličnost se najviše ispoljava u pisanju testova, što je izloženo u sledećim primerima [3]:

```
import org.specs2.mutable._
```

```
class HelloWorldSpec extends Specification {
```

```
  "The 'Hello world' string" should {
    "contain 11 characters" in {
      "Hello world" must have size(11)
    }
    "start with 'Hello'" in {
      "Hello world" must startWith("Hello")
    }
    "end with 'world'" in {
      "Hello world" must endWith("world")
    }
  }
```

```
}
}
}
```

Ovo je primer test klase koja u sebi sadrži jedan test, a taj test proverava tri slučaja. Ime-uslova prati operator “in” posle čega se u okviru vitičastih zagrada navodi uslov koji mora biti ispunjen. Jako je bitno ime uslova, u slučaju da test nije zadovoljen, korisniku se u konzoli ispisuje ime uslova koji nije zadovoljen. Analiza jednog od uslova:

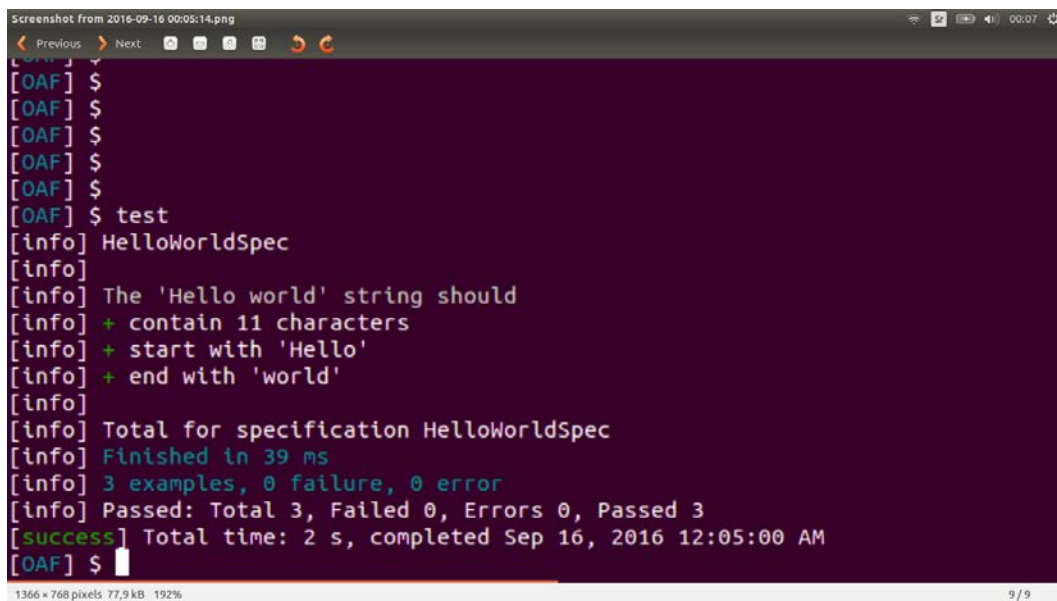
"Hello world" must have size(11)

Prethodni tekst izgleda kao obična rečenica u Engleskom jeziku, a i značenje je jasno. Dakle, string **"Hello world"** mora biti dužine 11. Naravno, ovo je samo primer, a u složenijim testovima koriste se i imena promenljivih.

Da bi se proverili testovi, potrebno je startovati aktivator (opisano u delu 4.2.2) i u njemu se može koristiti sledeće komande:

- test

Proverava sve testove iz test foldera i ispisuje rezultate uspešnosti.



```

[OAF] $
[OAF] $
[OAF] $
[OAF] $
[OAF] $
[OAF] $
[OAF] $ test
[info] HelloWorldSpec
[info]
[info] The 'Hello world' string should
[info] + contain 11 characters
[info] + start with 'Hello'
[info] + end with 'world'
[info]
[info] Total for specification HelloWorldSpec
[info] Finished in 39 ms
[info] 3 examples, 0 failure, 0 error
[info] Passed: Total 3, Failed 0, Errors 0, Passed 3
[success] Total time: 2 s, completed Sep 16, 2016 12:05:00 AM
[OAF] $
  
```

Slika 30. Primer pokretanja testova i ispisivanje rezultata

- test-only ime-testa

Proverava samo jednu test klasu

- test-quick

Proverava samo testove koji su u prethodnom pozivu pali.

U *Play* okruženju postoji dodatak *Mockito*. On se koristi u situacijama kada je potrebno “lažirati” podatke (eng. Mock Data). Ovo se koristi u situacijama, kada je aplikacija podeljena u više nezavisnih slojeva, pa je potrebno testirati metodu u svakom sloju

izolovano. Primer:

Ako se testira metoda kontrolera, koji u sebi poziva metodu servisa, tada servis treba biti "lažiran" i na taj način se postiže izolacija i testira se samo funkcionalnost metode kontrolera.

4.12 Scala i Java

Ovaj odeljak je posvećen čitljivosti koda. Ranije je u radu navedeno (deo 2.4.) koliko sama čitljivost koda predstavlja problem i koliko je ona bitna za održavanje aplikacije. Scala je jako dobro odgovorila na ovaj problem i u velikoj meri smanjila broj linija koda koji je potrebno napisati za neku funkcionalnost. Manje koda znači manja verovatnoća greške, a i sam kod je lakše protumačiti. Sledi niz primera Scala i Java koda, koji predstavljaju ekvivalentan kod.

Primer 1 [10]

Java:

```
List<String> list = new ArrayList<String>();  
list.add("1");  
list.add("2");  
list.add("3");
```

Scala:

```
val list = List("1", "2", "3")
```

Razlika u količini koda je očigledna, uvek je najbolje imati jednu liniju koda, nego više linija, zato što sledeći programer može ubaciti svoj kod između i na taj način narušiti semantičku celinu prethodnog programera. Ovim se čitljivost koda drastično smanjuje. Bitno je napomenuti, da Scala ne zahteva deklarisanje tipa promenljive, već sam kompajler to određuje u odnosu na desni operand operatora dodele, što dodatno smanjuje verovatnoću greške.

Primer 2 [10]

Java:

```
List<Integer> ints = new ArrayList<Integer>();  
for (String s : list) {  
    ints.add(Integer.parseInt(s));  
}
```

Scala:

```
val ints = list.map(s => s.toInt)
```

U ovom primeru dolazi do izražaja dobra strana korišćenje *map* funkcije nad listom i lambda funkcije kao argumenta. Kod je manji i čitljiviji. Treba приметiti i da je konverzija stringa u ceo broj jednostavnija u Skali.

Primer 3 [10]

Java:

```
public class User {
    private String name;
    private List<Order> orders;

    public User() {
        orders = new ArrayList<Order>();
    }

    public String getName() {
        return name;
    }

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

    public List<Order> getOrders() {
        return orders;
    }

    public void setOrders(List<Order> orders) {
        this.orders = orders;
    }
}
```

Scala:

```
class User {
    var name: String = _
    var orders: List[Order] = Nil
}
```

Skoro svaki Java programer je u svom radu naišao na “veliku” klasu. Ta klasa je velika najviše zbog getera i setera. Kako Scala rešava taj problem vidi se u ovom primeru. Međutim, članice klase su ipak javne, tako da nema provere kod dodeljivanja vrednosti. Na primer, ako je potrebno da se izbaci izuzetak pri postavljanju imena na *NULL*.

```
class User {
  private var _name: String = _
  var orders: List[Order] = Nil
  def name = _name
  def name_=(name: String) = {
    if (name == null) {
      throw new NullPointerException("User.name cannot be null!")
    }
    _name = name
  }
}
```

U tom slučaju je kod slične dužine kao kod Jave, ali je sama dodela vrednosti članu klase jednostavnija.

```
user.name = "John Doe"
```

Ovo su bili neki od primera kraćeg koda. Bitno je napomenuti, da u okviru Scala programa postoji mogućnost korišćenja Java biblioteka.

4.13 Prednosti i mane programskog jezika Scala

Prednosti:

Sve prednosti su navedene i opisan kroz rad, tako da su u ovom delu samo pobrojane.

1. Visok nivo čitljivosti koda
2. Dobra skalabilnost
3. Dobra podrška za razvoj višenitnih sistema – Sistem glumaca
4. Parcijalne funkcije (Karijeve funkcije)
5. Jednostavan rad sa bazom podataka – korišćenje funkcija za rad sa listama
6. Neblokirajući pozivi ka eksternim servisima – Future klasa
7. Brza instalacija – korišćenje aktivatora
8. Podrška za veb sokete

Mane:

1. Sporo kompajliranje – Glavni uzrok ovog problema je činjenica da kompajler prevodi svaku rekurziju u iteraciju. Naime, Scala je zasnovana na rekurziji, a opšte je poznato da je u praksi rekurzija manje efikasna od iteracije zbog vremena koje se gubi pri formiranju memorijskog okvira (eng. stack frame) pri svakom rekurzivnom pozivu. Pošto je iteracije ekvivalentna rekurziji, kompajler prevodi svaku rekurziju u iteraciju. Na ovaj način se kompajler značajno usporava i u poređenju sa kompajliranjem Jave, razlika u brzini je negde oko 1:3 u korist Jave.

2. Greške u alatima za razvoj (Eclipse, IntelliJ IDEA) – Dešava se da neke greške budu prijavljene od strane alata, a da kompilacija kroz komandnu liniju prođe bez greške. Na ovaj način se gubi dosta vremena, pošto sam alat za razvoj ne ukazuje programeru validne greške.

3. Mnogo dodataka je u ranoj fazi – Neki dodaci (eng. Plug-in) su u ranoj fazi i nisu

pouzdati, a alternativa ne postoji. Inače, uglavnom se dodaci razvijaju na **GitHub**-u i posle se kao zavisnosti (eng. Dependencies) uključuju u projekat.

4. Neoptimalnost Play okruženja – Samo okruženje je u ranoj fazi razvoja, pa neke od stvari nisu dovoljno optimizovane. Na primer, forme za validaciju podataka. Primer iz prakse:

Aplikacija ima akciju koja prihvata listu telefonskih kontakata i svaki kontakt čuva u bazi. Na veb aplikaciji javio se problem da HTTP POST zahtev sa oko 2000 kontakata traje između 40-50 sekundi, što je poprilično mnogo. Posle nedelju dana testiranja i merenja vremena u svakom segmentu aplikacije ustanovljeno je da je problem u konverziji iz JSON-a, koji stiže iz HTTP zahteva u **case** klasu koja predstavlja objekat forme. Ovu konverziju automatski vrši *Play* okruženje. Nakon kontaktiranja **Typesafe** oraganizacije (organizacija koja razvija *Play* okruženje) dobijen je odgovor da same forme nisu optimizovane u trenutnoj verziji.

5. Implementacija aplikacije OAF

U ovom odeljku je prikazana implementacija aplikacije OAF. U prethodnom tekstu, kroz primere su opisani neki kritični delovi koda u ovom odeljku su izloženi još neki delovi kritičnog koda u cilju da se stvori kompletna slika funkcionisanja aplikacije OAF.

5.1 Arhitektura aplikacije OAF

Aplikacija OAF se sastoji od dva projekta:

- OAF-DAL
- OAF

OAF-DAL je klasičan Scala projekat, koji u sebi sadrži Scala klase i objekte koji su zaduženi za pristup i rad sa bazom. Detaljan opis projekta OAF-DAL je bio izložen u delu 4.5.4. OAF je Play-Scala projekat, koji ima strukturu MVC projekta. Projekat OAF ima referencu na projekat OAF-DAL i zbog toga može pristupati njegovim klasama, na taj način je ostvarena komunikacija sa bazom podataka iz projekta OAF. Sa druge strane, OAF-DAL nema reference ka projektu OAF (nije mu ni potrebna) i ponaša se kao potpuno nezavisan projekat čija je namena da doprema i čuva podatke u bazi. Prednost ovakve arhitekture je višestruka:

- Nezavisnost sloja podataka
- Pogodno za testiranje jedinica (Nezavisno se može testirati samo sloj baze)
- Korišćenje sloja baze u drugim aplikacijama

Poslednja stavka je objašnjena kroz primer. Naime, ako je potrebno razviti novu aplikaciju čija je namena pravljenja izveštaja o prodaji hrane dovoljno je samo u toj aplikaciji referencirati na projekat OAF-DAL i sve metode za rad sa podacima su dostupne i već implementirane.

5.2 Projekat OAF

Projekat OAF je Play-Scala projekat koji predstavlja kompletnu veb aplikaciju. U ovom projektu se nalaze sledeće stvari:

1. **HTML stranice i šabloni** – opisano u delu 4.4
2. **Kontroleri i forme** – opisani u delu 4.3
3. **Servisi**
4. **Modeli**

Obrada jednog zahteva:

Na HTML stranici se unose podaci i klikom na dugme ili link formira se zahtev koji se šalje na server. Na serveru se uz pomoć routes fajla vrši preslikavanje URL-a iz zahteva u konkretnu akciju iz odgovarajućeg kontrolera. U okviru akcije, prvo se vrši validacija podataka pomoću forme, ako je validacija uspešna podaci se šalju odgovarajućoj metodi servisa. U okviru servisa se odvija glavni deo obrade svakog poziva. Naime, servis poziva metode iz OAF-DAL izvršava dodatna izračunavanja i vraća rezultat akciji kontrolera koja ga je pozvala. Akcija dobijene rezultate zapakuje u Response objekat i taj objekat šalje nazad klijentu koji je inicirao zahtev. O ovome je bilo više reči u delu 4.4.4. U ovom odeljku kroz primer je prikazan rad servisa i uloga modela.

5.2.1 Modeli

Modeli se koriste za prezentaciju. Naime, postoje modeli i u okviru projekta OAF-DAL i oni predstavljaju definiciju entiteta koji se čuvaju u bazi podataka, dok modeli iz aplikacije OAF predstavljaju njihovu kopiju sa manjim skupom podataka. Razlog tome leži u činjenici da nisu baš svi podaci koji se čuvaju u bazi potrebni na prezentacionom sloju.

Primer:

```
case class Article
(
  id: Option[Long],
  name: String,
  description: Option[String],
  price: Double,
  pictureLocation: Option[String],
  weight: Option[Double],
  companyId: Long,
  status: ArticleStatus.Value
)

object Article {

  implicit def convertToModel(articleDbModel: ArticleDBModel) : Article = {
    Article(articleDbModel.id, articleDbModel.name, articleDbModel.description, articleDbModel.price,
      articleDbModel.pictureLocation, articleDbModel.weight, articleDbModel.companyId,
      articleDbModel.status)
  }

  implicit def convertToDBModel(article: Article) : ArticleDBModel = {
    ArticleDBModel(article.id, article.name, article.description, article.price,
      article.pictureLocation, article.weight, article.companyId, article.status)
  }
}
```

Definicija modela sadrži dva dela, prvo je case klasa u kojoj se definišu polja, a drugi deo je objekat u kome se definišu metode za implicitnu konverziju iz DBModel-a (model iz projekta OAF-DAL) u Model (model iz projekta OAF) i obratno. Implicitne metode omogućavaju programeru da ne vodi računa u kom trenutku koristi koji model, jer sam kompajler prepoznaje potrebu za konverzijom i izvršava konverziju modela u odgovarajući tip. O implicitnim funkcijama je bilo više reči u delu 4.1.2.

5.2.2 Servisi

Ulogu servisa je izložena kroz primer jedne metode iz `CustomerService`. Namena metode je dodavanje artikla u korpu, što u programerskom smislu znači- pravljenje novog entiteta `OrderedArticle` i povezivanje istog sa trenutno aktivnom narudžbinom.

```
def createOrderedArticle(orderId: Long, addToCartData: AddToCartData): Long = {  
  play.api.db.slick.DB.withTransaction { implicit session =>  
    val article = ArticleService.findById(addToCartData.articleId)  
    val additives = addToCartData.additiveIds.map(AdditiveService.findById(_).get)  
    val orderedArticle = OrderedArticle(None, orderId, article.get, additives)  
    OrderedArticleDAO.create(orderedArticle)  
  }  
}
```

Ulazni parametri metode su `orderId` i `addToCartData`. Parametar `orderId` predstavlja vrednost primarnog ključa narudžbine za koju se vezuje artikal, dok parameter `addToCartData` predstavlja podatke koji su došli u zahtevu i koje je forma prethodno konvertovala u odgovarajuću case klasu. Na početku se formira transakcija, jer ova metoda izvršava više različitih operacija nad podacima u bazi i potrebno je da se one izvrše atomično.

```
val article = ArticleService.findById(addToCartData.articleId)
```

Prvo se nalazi odgovarajući artikal pomoću metode iz `ArticleService`-a, koja prima id artikla koji se dodaje u korpu kao argument.

```
val additives = addToCartData.additiveIds.map(AdditiveService.findById(_).get)
```

Nakon toga se pronalaze svi dodaci koje je korisnik izabrao za taj artikal. Pošto je u zahtevu stigla lista samo id-ijeva odabranih dodataka, potrebno je izvršiti mapiranje u kompletne modele za dodatke (`Additive model`). To se postiže pozivom funkcije `map`, koja kao argument ima funkciju `AdditiveService` koja nalazi željeni entitet po id-u.

```
val orderedArticle = OrderedArticle(None, orderId, article.get, additives)
```

U ovoj liniji se pravi primerak klase `OrderedArticle`, prosleđuje se redom: id, id-narudžbine, artikal i lista dodataka. U ovom slučaju je kao id prosledjeno `None`, zato što sam SQL server generiše id za ovaj entitet.

```
OrderedArticleDAO.create(orderedArticle)
```

Poslednji poziv predstavlja poziv metode iz DAO klase.

Na ovaj način je implementirana obrada svakog zahteva u aplikaciji OAF, time je postavljena konvencija u izradi ove aplikacije, što omogućava novom programeru da se lako uključi u razvoj aplikacije.

6. Zaključak

Comment [VF10]: 6.

U ovom radu izložena je primena programskog jezika Scala u razvoju veb aplikacija. Istaknuti su glavni problemi koji se javljaju pri razvoju veb aplikaciji i kako su ti problemi rešeni od strane programskog jezika Scala. Prikazano je i *Play* okruženje koje predstavlja kompletan skup potrebnih stvari za razvoj i pokretanje jedne veb aplikacije. U sklopu ovog rada razvijen je i softver otvorenog koda na programskom jeziku Scala koji koristi *Play* okruženje.

Razvijeni softver je dostupan na adresi <https://github.com/CetkoS/OAF>.

Dakle, može se zaključiti da se Scala jako dobro pokazala u razvoju veb aplikacija, ali se više preporučuje za razvoj manjih sistema, zbog same nestabilnosti *Play* okruženja i dodatka – što je detaljnije izloženo u delu 4.12. Međutim, i pored ovih problema, treba istaći da su neke od velikih kompanija (kao što su LinkedIn i Twitter) odlučile da koriste Skalu kao glavni programski jezik.

Comment [VF11]: Dakle, može se zaključiti

Literatura

- [1] H. Barendregt, "The Lambda Calculus".
- [2] J. D. Suereth, "Scala in Depth", Manning, 2012.
- [3] "Playframework," Typesafe, [Online]. Available: <https://www.playframework.com/documentation/2.3.x/ScalaHome>.
- [4] "Slick," Typesafe, [Online]. Available: <http://slick.lightbend.com/docs/>.
- [5] "Reactive Mongo," [Online]. Available: <http://reactivemongo.org/>.
- [6] "Scala-Lang," École Polytechnique Fédérale, [Online]. Available: <http://docs.scala-lang.org/overviews/core/futures.html>.
- [7] "Cluster Sharding," Typesafe, [Online]. Available: <http://doc.akka.io/docs/akka/snapshot/scala/cluster-sharding.html>.
- [8] D. Wyatt, "Akka Concurrency", Artima, 2013.
- [9] "tkszf," [Online]. Available: <https://blog.tkszf.org/2012/10/12/websockets-echo-using-play-scala-and-actors-part-ii/>.
- [10] "Toptal," [Online]. Available: <https://www.toptal.com/scala/why-should-i-learn-scala>.

Spisak slika

Slika 1. Trenutnu popularnost okruženja za razvoj veb aplikacija	6
Slika 2. Popularnost Play okruženja	7
Slika 3. Registracija korisnika	10
Slika 4. Unos podataka o radnji	11
Slika 5. Lista zaposlenih.....	12
Slika 6. Stranica za dodavanje novog zaposlenog	12
Slika 7. Lista artikala	13
Slika 8. Lista dodataka	13
Slika 9. Lista novih narudžbina	14
Slika 10. Detaljne informacije o narudžbini.....	14
Slika 11. Lista aktivnih narudžbina	15
Slika 12. Lista kompletiranih narudžbina	15
Slika 13. Lista radnji	16
Slika 14. Spisak artikla jedne radnje	16
Slika 15. Dodavanje artikla u korpu	17
Slika 16. Popunjavanje informacija o narudžbini.....	17
Slika 17. Hijerarhija klasa. Slika preuzeta iz [2]	19
Slika 18. Struktura fajlova u Play projektu	25
Slika 19. Startovanje aktivatora	26
Slika 20. Prikaz greske na HTML strani	29
Slika 21. Izgled adminMain šablona.....	33
Slika 22. EmployeeMain šablon	34
Slika 23. CustomerMain šablon.....	34
Slika 24. Stranica za pravljenje novog zaposlenog.....	36
Slika 25. Prikazivanje greške pri neuspeloj validaciji podataka	37
Slika 26. Uspešno napravljen radnik i redirekcija na drugu stranu	38
Slika 27. Izvršavanje evaluacionih skripti.....	39
Slika 28. Struktura OAF-DAO projekta.....	42
Slika 29. Test folder u okviru Play projekta.....	52
Slika 30. Primer pokretanja testova i ispisivanje rezultata.....	53