Web aplikacije u programskom jeziku Java i razvojni okvir Play

Karlović, Ana Marija

SVEUČILIŠTE U ZAGREBU

PRIRODOSLOVNO–MATEMATIČKI FAKULTET

MATEMATIČKI ODSJEK

Ana Marija Karlović

WEB APLIKACIJE U PROGRAMSKOM JEZIKU JAVA I RAZVOJNI OKVIR PLAY

Diplomski rad

Voditelj rada:
doc. dr. sc. Zvonimir Bujanović
Suvoditelj rada:
dr. sc. Goran Igaly

Zagreb, Veljača 2017.

Ovaj diplomski rad obranjen je dana _____ pred ispitnim povjerenstvom u sastavu:

1. _____, predsjednik
2. _____, član
3. _____, član

Povjerenstvo je rad ocijenilo ocjenom _____.

Potpisi članova povjerenstva:

1. _____
2. _____
3. _____

Mojoj obitelji koja je uvijek bila uz mene i mojoj drugoj polovici koja će zauvijek biti uz mene.

Sadržaj

Uvod

Jedna od najplodonosnijih grana programiranja danas je izrada web aplikacija, u različitim programskim okruženjima. Višeslojna arhitektura aplikacije predstavlja klijent-poslužitelj model aplikacije u kojem i klijentska i poslužiteljska strana mogu biti dodatno raslojene. Ti različiti slojevi međusobno komuniciraju postavljajući zahtjeve i odgovarajući na njih. Najrašireniji tip višeslojne arhitekture aplikacija je troslojna arhitektura, koju čine sljedeći slojevi:

- Prezentacijski sloj
- Sloj poslovne logike
- Podatkovni sloj

Razdvajanje aplikacije u slojeve povećava njenu robusnost, omogućava ponovno korištenje njenih dijelova, generalizira strukturu aplikacije i omogućava lakše ugrađivanje oblikovnih obrazaca. Također, olakšava njen razvoj, te se isti tako lakše može odvojiti u vremenski i resursno odvojene cjeline, odnosno razvoj se može rasporediti na više progamera, od kojih je svaki stručan u svom polju, s tim da njihov rad ne mora nužno biti paralelan.

Svaki od ovih slojeva može se dodatno podijeliti na nove slojeve, omogućujući tako daljnju generalizaciju i povećavajući mogućnosti aplikacije. Najrašireniji obrazac po kojem se gradi prezentacijski sloj, danas je MVC *(Model - View - Controller)* ili model - pogled - upravljač. U ovom radu bit će predstavljen razvojni okvir *Play* u programskom jeziku *Java*. *Play* podržava izradu web-aplikacija prateći MVC obrazac.

Rad je podijeljen u tri poglavlja. U prvom poglavlju govori se općenito o obrascima u programiranju, zatim konkretno o MVC obrascu, te o programskom jeziku *Java* u kojem je pisana aplikacija Redomat. Nakon toga daje se uvid u strukturu i prednosti razvojnog okvira *Play* u drugom poglavlju. Na kraju, u trećem poglavlju, opisana je aplikacija Redomat, njene funkcionalnosti, implementacija i dijelovi.

Poglavlje 1

Java i MVC oblikovni obrazac

1.1 Oblikovni obrasci

Pojam

U programskom inženjerstvu, oblikovni obrazac je opće, višestruko upotrebljivo rješenje za najčešće probleme koji se javljaju prilikom razvoja softvera. To nije gotovo rješenje koje se može izravno prebaciti u kod, već opis ili predložak za rješavanje problema, koji se može koristiti u različitim situacijama.

Prema [1], oblikovni obrasci su formalizirane, najbolje prakse rješavanja problema koji se najčešće javljaju pri izradi programa ili sustava. Definiraju ih i formaliziraju stručnjaci iz područja teorijskog i primjenjenog računarstva, a njihova uporaba je slobodna.

Oblikovni obrasci ubrzavaju razvojni proces pružajući dokazane razvojne paradigme. Učinkoviti razvoj softvera zahtjeva predviđanje problema koji možda nisu vidljivi prije same implementacije. Korištenje oblikovnih obrazaca pomaže u prevenciji problema koji se tek u kasnijim fazama razviju u značajne prepreke izvedbi zamišljene arhitekture rješenja, te poboljšava čitljivost koda za programere koji su upoznati s obrascima.

Vrlo često je već poznato rješenje sličnog problema s kojim se programeri susreću i tehnike koje se koriste prilikom izrade tog rješenja su jasne, ali problem nastaje kada treba prilagoditi te tehnike kako bi bile pogodne i za novi, konkretni problem. Oblikovni obrasci pružaju općenita rješenja, dokumentirana u formatu koji ne zahtijeva specifičnosti vezane za određeni problem.

Vrste

Osnovna klasifikacija obrazaca bi bila sljedeća: obrasci stvaranja, obrasci ponašanja, strukturni obrasci te arhitekturalni obrasci. Prva tri navedena tipa obrazaca definirana su konceptima raspodjele, grupiranja i savjetovanja i primjenjuju se u objektno orijentiranom programiranju [1]. Najnoviji tip, arhitekturalni obrazac, odnosi se na strukturu softvera. Najviše se primjenjuje pri dizajniranju web aplikacija.

Najpoznatiji obrasci stvaranja su *Factory method*, *Abstract factory* i *Prototype*. *Factory method* definira apstraktnu klasu ili sučelje [2] za stvaranje objekta, a stvaranje konkretnog objekta prepušta potklasama, odnosno konkretnim implementacijama sučelja [3]. Slično, *Abstract factory* služi za stvaranje više vezanih ili ovisnih objekata bez navođenja konkretnih klasa (npr. pomoću više metoda tvornica). *Prototype* služi za kreiranje objekta polazeći od danog prototipa, pozivanjem metode `clone()`. Bazna klasa *Prototype* deklarira virtualnu metodu `clone()`, a izvedene klase implementiraju tu metodu. Njen rezultat je objekt identičan objektu s kojim je metoda pozvana.

Predstavnici obrazaca ponašanja su *Observer* i *Visitor*. *Observer* se koristi kada više različitih objekata ovisi o stanju jednog objekta. Kreira se subjekt koji predstavlja objekt čije se stanje promatra. Klasa *Subject* sadrži metodu `notify()` koja obaviještava sve promatrače o promjeni stanja. Promatrači su konkretne implementacije apstraktne klase *Observer*. Proizvoljan broj njih može biti pretplaćen na jednog subjekta, a njegova metoda `notify()` poziva metodu `update()` na svim promatračima pretplaćenima na tog subjekta. *Visitor* nalazi primjenu kada je potrebno dodati operaciju nizu objekata, moguće i različitog tipa, bez mijenjanja klasa koje predstavljaju te objekte. U sve klase kojima je potrebno dodati operaciju, definira se metoda `accept()` koja prima tip *Visitor*. Time te klase implementiraju sučelje *Visitable*. *Visitor* je sučelje koje sadrži metode `visit()` za svaku klasu koja implementira sučelje *Visitable*, a koja prima tip odgovorajuće klase. Na taj način svaka klasa koja implementira sučelje *Visitor* mora definirati metode `visit()` za svaku klasu iz sučelja *Visitable*.

Među strukturnim obrascima ističu se *Composite* i *Adapter*. *Composite* se koristi kada je potrebno strukturirati složeni objekt kao stablo koje predstavlja hijerarhiju sastavnih dijelova. Cilj je da se može jednako tretirati i dio i cjelinu. *Component* predstavlja sučelje svih komponenti u hijerarhiji. Nudi implementaciju koja je zajednička jednostavnim i složenim komponentama. Jednostavne komponente predstavljene su klasom *Leaf*, a složene, koje

[1] Jedan od pristupa programiranju kojemu je težište na projektiranju aplikacije kao skupa objekata koji izmjenjuju informacije između sebe.

[2] Klasa određuje tip objekta. Apstraktna klasa (ili razred) nema definirana tijela svih metoda, apstraktne metode su one koje imaju definiran samo naziv i parametre. Sučelje je klasa koja ima definirane samo nazive i parametre metoda, ne može sadržavati niti jednu implementaciju.

[3] Klase mogu naslijeđivati ili implementirati apstraktne klase ili sučelja, što je jedini način da se navedeni koriste.

sadrže druge komponente, klasom *Composite*. *Adapter*, kako mu i sam naziv govori, prilagođava sučelje jedne klase drugoj klasi. Klasa *Adaptor* sadrži referencu na objekt klase koju je potrebno prilagoditi. Metode u klasi *Adaptor* pozivaju metode u na tom objektu i oblikuju sučelje onako kako je to potrebno.

Arhitekturalni obrasci su npr. MVC i ORM. ORM ili *Object-Relational Mapping* je tehnika pretvaranja podataka između međusobno nekompatilnih sustava, konkretno između baze i aplikacije. ORM služi za jednostavno punjenje modela iz baze, a programer ga može sam implementirati ili iskoristiti neki od već postojećih komercijalnih ili slobodnih paketa.

1.2 MVC Obrazac

Model-pogled-upravljač je arhitekturalni tip oblikovnog obrasca koji se koristi prilikom izrade korisničkog sučelja. Prilikom korištenja MVC oblikovnog obrazca, aplikacija se dijeli u tri cjeline koje su međusobno povezane kako bi se odvojile interne reprezentacije podataka od načina na koji će ti podaci biti prikazani korisniku odnosno načina na koji će korisnik unositi nove podatke. Ova arhitektura je tradicionalno korištena za izradu grafičkog korisničkog sučelja, no postala je popularna za izradu web aplikacija, odnosno njezinog prezentacijskog sloja. [8]

Cjeline MVC obrasca su sljedeće:

- Model: neposredno upravlja podacima
- Pogled (*View*): pogled može biti bilo koja reprezentacija podataka iz baze ili forma za unos podataka od strane korisnika
- Upravljač (*Controller*): upravljač prima ulazne podatke (iz baze ili od korisnika) i pretvara ih u naredbe koje izvršavaju model ili pogled

Povijest MVC obrasca

Sedamdesetih godina prošlog stoljeća, Trygve Reenskaug predstavio je MVC obrazac u sklopu programskog jezika Smalltalk[5]. Desetak godina kasnije, Jim Althoff s kolegama je implementirao verziju MVC obrasca unutar skupa klasa Smalltalk-80. Tek 1988. godine izašao je članak u *The Journal of Object Technology (JOT)* u kojem je MVC proglašen opće prihvaćenim konceptom.

Upotreba MVC obrasca u izradi web aplikacija postala je izrazito popularna 1996. godine uvođenjem *Apple-ova* okruženja *WebObjects*, da bi se kasnije ukomponirao u razvojna okruženja u *Javi*, kao npr. *Spring, Play*, te u drugim programskim jezicima koji su imali naglasak na ubrzanom razvoju softvera. U ovim okruženjima MVC obrazac se primjenjuje u arhitekturi prezentacijskog sloja.

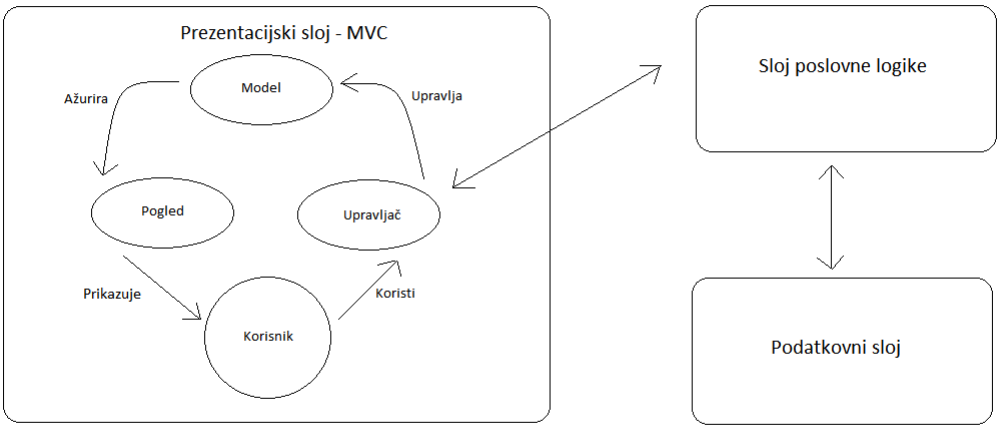
Interpretacije MVC oblikovnog obrazca unutar različitih programskih radnih okruženja variraju. Najviše varijacija se nalazi u raspodjeli odgovornosti i posla između klijenta i poslužitelja, kao što vidimo iz sljedećih primjera podtipova MVC obrasca:

- hijerarhijski MVC (HMVC) - upravljač odabire model, a zatim pogled, tako da ima mogućnost prilagođenog prikaza različitih pogleda ili istih podataka na više načina na istoj stranici
- model-pogled-adapter (MVA) - upravljač ima potpunu kontrolu nad podacima koji se izmjenjuju između modela i pogleda, te omogućava neometanu obradu velike količine podataka u modelu dok se pogled ne mijenja
- model-pogled-prezenter (MVP) - prezenter samo prilagođava podatke iz modela za prikaz u pogledu, a pogled je pasivno sučelje koje služi za prikaz podataka iz modela i prosljeđuje naredbe prezenteru
- model-pogled-pogledmodel (MVVM) - *pogledmodel* barata većinom ili cijelom logikom prikaza pogleda, te na taj način omogućava da podacima iz modela koji su predstavljeni objektima, bude lako upravljati, pretvaranjem u objekte prilagođene za pogled

MVC u web aplikacijama

Uz podjelu prezentacijskog sloja u tri cjeline, MVC oblikovni obrazac definira i interakcije između njih.

- Model pohranjuje podatke koji su dohvaćeni na temelju naredbi koje je izdao upravljač, a prikazuju se u pogledu
- Na temelju promjena u modelu generiraju se novi izlazni podaci koje će pogled prezentirati korisniku
- Upravljač može slati naredbe modelu kako bi promijenio stanje modela (npr. uređivanje dokumenta). Također, može slati naredbe pripadajućem pogledu kako bi promijenio pogledovu prezentaciju modela (npr. listanje kroz dokument). On implementira poslovnu logiku ili njen dio, ovisno o strukturi ostatka aplikacije



Slika 1.1: Aplikacijski slojevi

U ovom radu opisana je struktura web aplikacije napisane u programskom jeziku *Java*, koristeći razvojni okvir *Play* koji prati MVC oblikovni obrazac. U većim aplikacijama i ostali slojevi, osim prezentacijskog, dijele se na cjeline koje međusobno komuniciraju i svaka ima svoju točno određenu ulogu, npr. spremanje podataka u određenim strukturama, implementacija poslovnih pravila za rukovanje tim podacima, te pretvaranje podataka u oblik pogodan za prikaz na korisničkom sučelju.

Važno je naglasiti da, kakva god struktura ostalih slojeva bila, prezentacijski sloj razvijen prema MVC oblikovnom obrascu komunicira s ostalim slojevima preko svog (MVC) upravljača. Upravljač interpretira informacije dobivene od korisnika i prosljeđuje nove podatke ostalim dijelovima aplikacije. Oni će se tamo ili samo spremiti ili će se primijeniti neka poslovna logika i na temelju dobivenog rezultata napraviti određene promjene u samom (aplikacijskom) modelu.

Sam prezentacijski sloj također ima svoj (MVC) model koji služi za čuvanje podataka u strukturama koje su kreirane za konkretni (MVC) pogled i na taj način omogućuju njihov lakši prikaz i prikupljanje informacija s korisničkog sučelja.

1.3 Programski jezik Java

Nastanak Jave

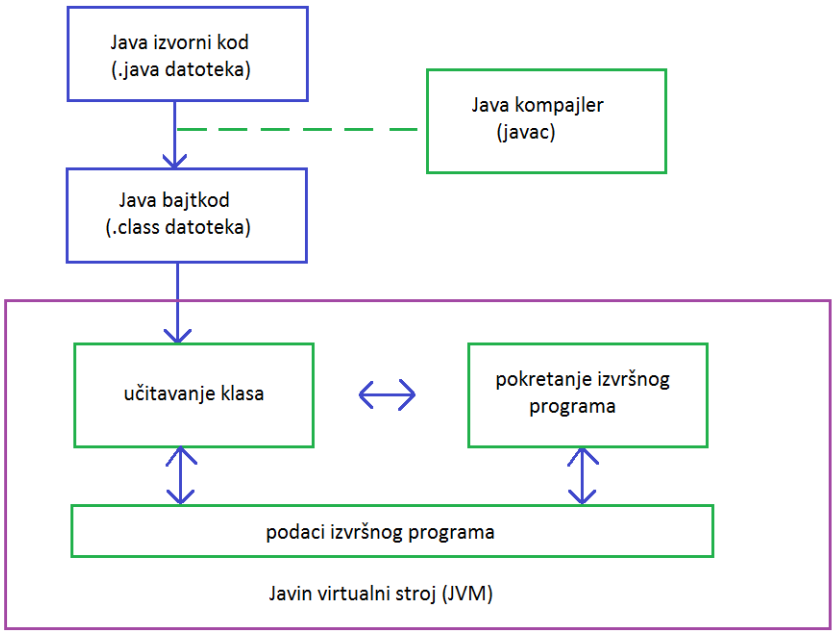
Programski jezik *Java* nastao je iz programskog jezika *Oak*. Od 1995. godine kada je u internetski preglednik *Netscape Navigator* ugrađena podrška za Javu pa sve do danas, jezik se razvija i širi te je danas praktički sveprisutan. Jedna od temeljnih vodilja u razvoju ovog jezika, koja je dovela do ovakve opće prihvaćenosti, jest ideja *napiši-jednom-pokreni-bilo-gdje*.

Centralna ideja u razvoju *Jave* je potpora za višeplatformnost, odnosno ideja da se programeru ponudi apstraktni pogled na računalo. Programer tako ne mora brinuti o platformski specifičnim detaljima poput širine podatkovne riječi centralnog procesora, direktnom pristupu upravljačkim programima operacijskog sustava ili podržanom sklopovlju. Umjesto toga, *Java* definira apstraktni računarski stroj kod kojeg je sve propisano i opisano specifikacijom. Slijedi opis osnovnih pojmova vezanih za *Javu* preuzet iz [10].

Javin virtualni stroj

Da bi *pokretanje-bilo-gdje* bilo moguće, prilikom programiranja u Javi programi se pišu za Javin virtualni stroj (*JVM - Java Virtual Machine*). To je stroj koji je definiran specifikacijom [9] i na kojem se izvode svi programi pisani u Javi. Zahvaljujući ovakvoj virtualizaciji, programer doista ne treba (i ne može) razmišljati o specifičnostima platforme na kojoj će se izvoditi program koji je napisao, a to upravo i jest ideja izrade aplikacija za više platformi odjednom. Dakako, na svakoj konkretnoj platformi morat će postojati implementacija Javinog virtualnog stroja koja će u konačnici biti svjesna na kojoj se platformi izvodi te kako se na toj platformi obavljaju pojedini zadaci (poput pristupa datotečnom sustavu, mrežnim resursima, višedretvenosti i slično).

Programeri programe pišu zadavanjem izvornog koda u programskom jeziku Java (tekstovne datoteke s ekstenzijom .java). Izvorni se kod potom prevodi u izvršni kod odnosno bajtkod (datoteke s ekstenzijom .class). Da bi se izvršni program mogao pokrenuti, potreban je Javin virtualni stroj te skup biblioteka (kompajliranih dijelova koda) čije se postojanje garantira svim Java programima. S obzirom na opisani proces, jasna je podjela platforme na dvije komponente.

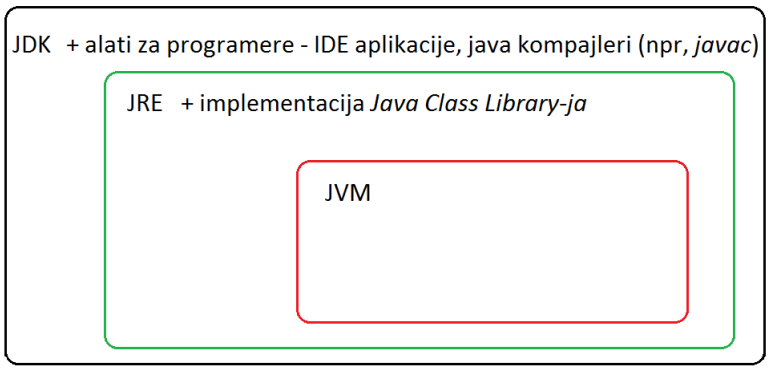


Slika 1.2: Java radno okruženje

JRE (Java Runtime Environment) predstavlja osnovni podskup Java platforme koji korisnicima nudi mogućnost pokretanja prevedenih programa. JRE se sastoji od implementacije Javinog virtualnog stroja te obaveznih biblioteka čije se postojanje garantira programima. Ovo je minimum koji će svima omogućiti da pokreću izvršne Java programe. Uz navedeno, JRE sadrži i implementaciju dodatka za web-preglednike koji i njima omogućava izvođenje Java programa koje korisnici preuzimaju direktno s Interneta.

Pojam JDK (Java Development Kit) predstavlja nadskup opisane platforme. JDK sadrži sve što je potrebno kako bi programer mogao prevoditi izvorni kod Java programa u bajtkod te kako bi mogao izvoditi Java programe. To znači da JDK u sebi uključuje JRE te donosi još i implementaciju prevodioca i drugih pomoćnih alata. JDK pri tome ne sadrži okolinu za razvoj Java programa, on programeru osigurava samo mogućnost prevođenja koda.

Na sljedećoj slici prikazan je odnos ovih platformi.



Slika 1.3: Java radno okruženje

Korištenje biblioteka

Programski jezik Java sam po sebi je vrlo jednostavan - pravila pisanja koda i ključne riječi su poprilično jednostavni, međutim, sam jezik je poprilično beskoristan. Da bi se mogli djelotvorno pisati programi, potreban je skup biblioteka čijom će uporabom ti programi moći obavljati željene funkcije (pristupiti datoteci, stvoriti prozor u grafičkom korisničkom sučelju, raditi s različitim vrstama podatkovnih kolekcija i slično). Osnovni skup biblioteka koje programeru stoje na raspolaganju dio su Java platforme i opisane su u okviru dokumenta Java API [3]. Neovisno o platformi na kojoj se Java program izvodi, programeru se garantira da će mu na raspolaganju biti sve biblioteke koje su opisane u okviru tog dokumenta.

Osim tih standardnih biblioteka koje su dostupne u okviru Java platforme, oko Java platforme stvoren je bogat i živ ekosustav koji je iznjedrio čitav niz drugih biblioteka, razvojnih okvira pa čak i novih programskih jezika koji su danas u uporabi. Primjerice, ako pogledamo izradu aplikacija za web, uz standardne tehnologije danas na raspolaganju imamo i niz razvojnih okvira, a trenutno najkorišteniji u komercijalne svrhe su JavaServer Faces, Vaadin, Google Web Toolkit i Grails.

Java SE i Java EE

Do sada opisano o programskom jeziku Java predstavlja samo jedan mali podskup Java platforme koji je poznat pod nazivom *Java Standard Edition*, odnosno Java SE. Puno šira specifikacija poznata pod nazivom *Java Enterprise Edition* donosi niz tehnologija koje pokrivaju izradu web aplikacija, rad s bazama podataka i raspodijeljeno transakcijsko pos-

lovanje, komuniciranje porukama i još niz drugih primjena. Postoji i specifikacija koja pokriva izradu Java aplikacija za mobilne uređaje koja međutim iz različitih razloga u ovom trenutku nije baš najbolje prihvaćena, što je pak dovelo do razvoja Googleove platforme Android koja je postala daleko raširenija, a također koristi programski jezik Java.

Poglavlje 2

Razvojni okvir Play

2.1 Osnovno o Play-u

Play je razvojni okvir za izradu web aplikacija. Napisan je u programskom jeziku Scala i otvorenog je kôda. Prilikom izrade aplikacija korištenjem razvojnog okvira *Play* moguće je koristiti programske jezike Scala ili Java. *Play* slijedi arhitekturalni obrazac MVC. Osnovne značajke *Play-a* koje programeru olakšavaju razvoj aplikacije su korištenje principa "konvencija prije konfiguracije" (što manji broj odluka koje programer mora donijeti prilikom korištenja nekog okvira, a da pritom nužno ne gubi fleksibilnost), ponovno učitavanje dijelova kôda bez ponovnog pokretanja te prikaz grešaka u pregledniku.

Jezgra razvojnog okvira prepisana je u verziji 2.0 u programskom jeziku Scala, a izrada i isporuka prebačena je na *SBT* [4]. *SBT* za Scala i Java projekte, slično kao Maven i Ant, je alat za automatizirano kompajliranje izvornog koda u bajtkod, spremanje bajtkoda u pakete (*.jar* datoteke), te pokretanje testova.

Povijest Play-a

Play je izradio Guillaume Bort dok je radio za Zengularity SA. Iako rana izdanja nisu dostupna, postoje dokazi da je Play postojao u svibnju 2007. Verzija 1.1 je izdana u studenom 2010. Trenutno aktualna verija 2.5.x izdana je 2016. godine. Najpoznatiji korisnici su LinkedIn i Coursera.

Motivacija

Razvojni okvir Play inspiriran je tehnologijama ASP.NET MVC, Ruby on Rails i Django, te je nalik toj porodici razvojnih okvira. Nudi okruženje koje je manje pod utjecajem Java

[4] Source Build Tool

EE, što omogućuje jednostavniji razvoj aplikacija u usporedbi s ostalim Java orijentiranim platformama.
Razlike u odnosu na ostala Java razvojna okruženja:

- koristi stateless protokol[5]: Play 2 je u potpunosti RESTful [6] - ne koriste se Java EE sesije [7] po konekciji
- integrirano testiranje: JUnit [8] je podržan u samoj jezgri
- najčešće korišteni elementi dolaze ugrađeni u sam API [9]
- statičke metode: sve ulazne točke (metode) upravljača (controller) su definirane statički
- asinkrone ulazno-izlazne operacije: zbog korištenja JBoss Netty [10] kao web poslužitelja, Play može posluživati dugotrajne zahtjeve asinkrono, umjesto da je vezan uz HTTP dretve koje izvršavaju poslovnu logiku kao što je slučaj kod Java EE razvojnih okruženja
- modularna arhitektura
- nativna podrška za programski jezik Scala: Play 2 interno koristi Scala programski jezik, ali izlaže i Play API i Java API; Java API je namjerno nešto drugačiji kako bi odgovarao Java konvencijama; a Play API je i dalje u potpunosti interoperabilan s Javom

Komponente

Play 2.0 koristi sljedeće Java biblioteke (dokumentacija dostupna na [6]):

- JBoss Netty kao poslužitelj

[5] Komunikacijski protokol koji svaki zahtjev tretira kao nezavisnu transakciju i ne zahtijeva od poslužitelja da sprema podatke o sesijama. Primjeri su Internet Protocol i Hypertext Transfer Protocol (HTTP).

[6] REST - Representational state transfer. RESTful aplikacija je ona koja s poslužiteljem komunicira koristeći ujednačene i predefinirane stateless operacije.

[7] Sesija je veza između klijenta i poslužitelja. U Javi EE poslužitelji spremaju podatke o sesijama u obliku objekata koji implementiraju isto sučelje.

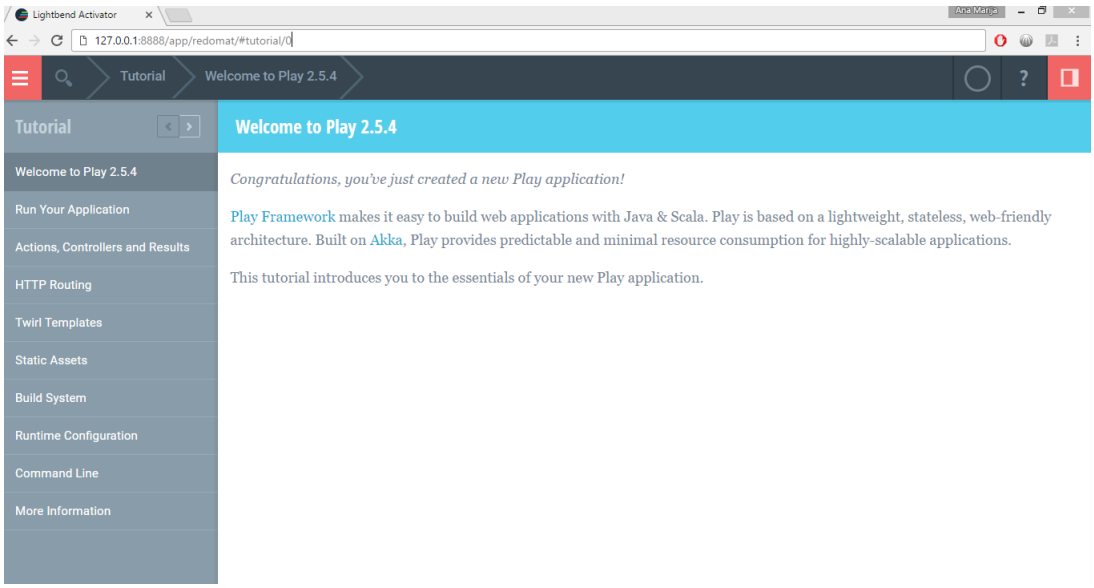
[8] Razvojno okruženje za testiranje Java programa.

[9] Application Programming Interface

[10] Razvojno okruženje za izradu web aplikacija kao što su klijent-poslužitelj protokoli

- ne zahtijeva ORM, ali za pristup bazi podataka omogućeno je korištenje Anorm-a [11] (Scala) i Ebean-a [11] (Java)
- Scala za predloške u *view-ovima*
- SBT za dohvat vanjskih biblioteka (upravljanje *dependency-ima*)

Play pruža ugrađeno okruženje za testiranje koje omogućuje testiranje cjelina (unit testing) i funkcijsko testiranje. Testovi se jednostavno pokreću u pregledniku putem URL-a [12]: <server-url>/@tests. Svi testovi će se izvršiti nad ugrađenom H2 bazom podataka (in-memory baza podataka).



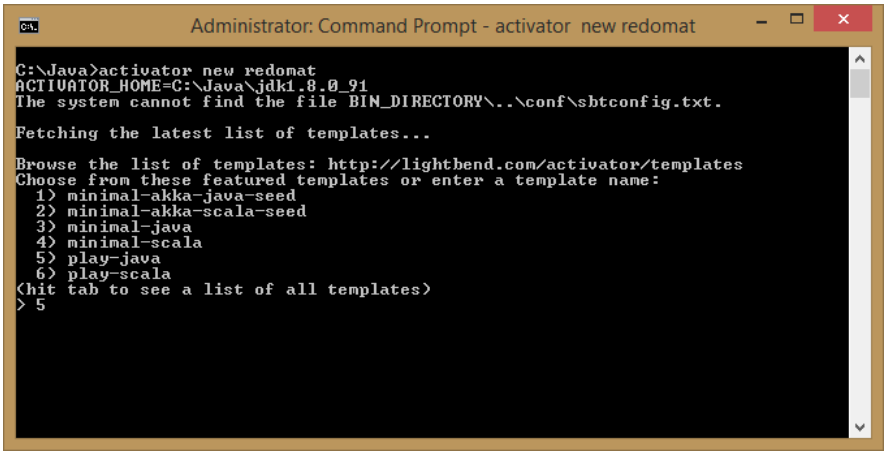
Slika 2.1: SBT sučelje s uputama za rad u Play-u

2.2 Alati za izradu aplikacije u razvojnom okviru Play

Kreiranje novog *Play* projekta, kompajliranje, pokretanje i testiranje omogućeno je preko aplikacije *Lightbend Activator*, dostupne na [4], koja obuhvaća *SBT* - alat za automatizirano kompajliranje izvornog koda, grafičko korisničko sučelje za osnovno upravljanje aplikacijom, katalog predložaka aplikacija, upute za rad s *Play-om* i slično.

[11] Sučelje za objektno-relacijsko mapiranje.
[12] Uniform Resource Locator

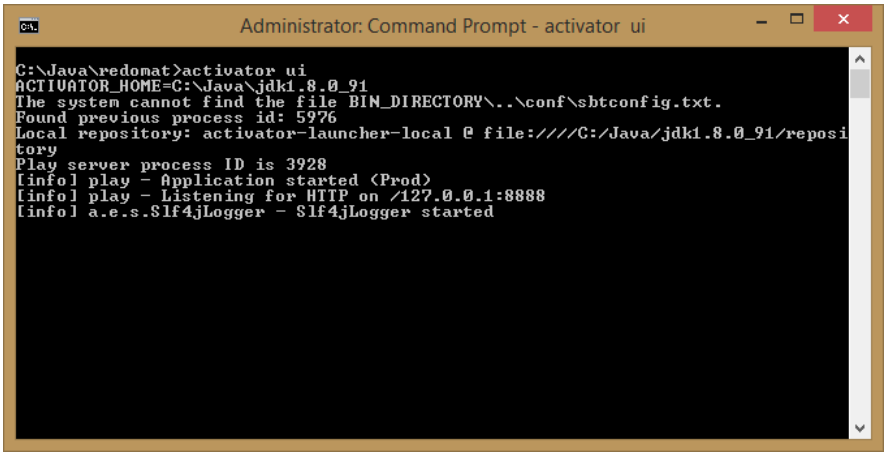
Aplikacija za potrebe ovog diplomskog rađena je na operacijskom sustavu *Windows 8.1*. Nakon instalacije *Lightbend Activator-a*, novi projekt se kreira upisivanjem naredbe *activator new* i odabirom predloška koji želimo koristiti za novu aplikaciju (*Java Play*). Time je kreirana struktura direktorija i datoteka koja je opisana u sljedećem podpoglavlju.



Slika 2.2: Kreiranje novog Play projekta

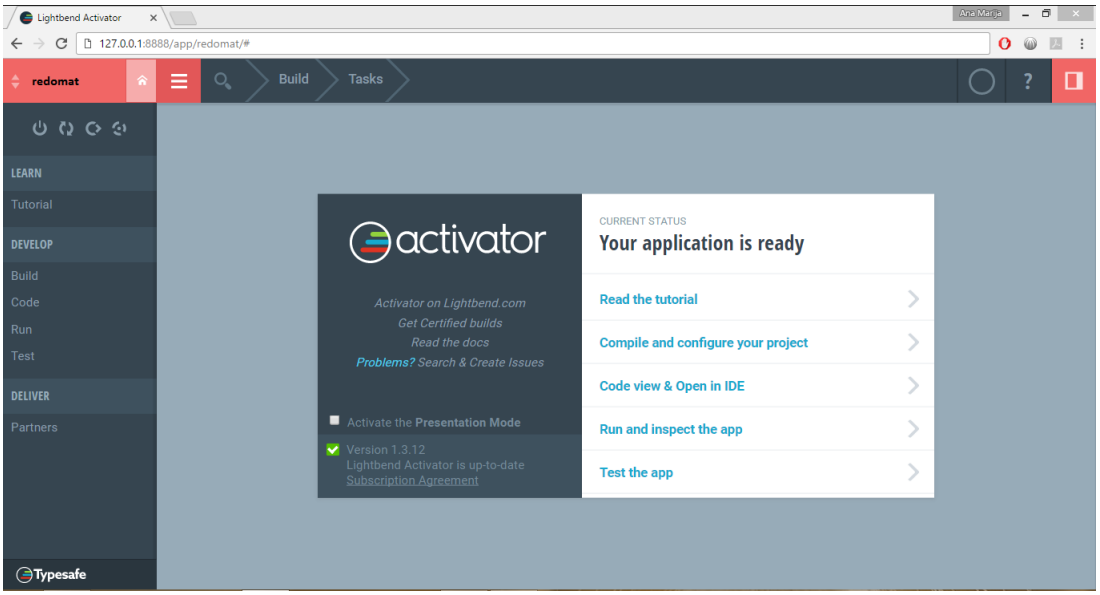
Pokretanje projekta koji smo već kreirali postiže se pozicioniranjem u direktorij u kojem se nalazi sam projekt i upisivanjem naredbe *activator ui* u konzolu.

U zadanom web pregledniku pokreće se grafičko sučelje u kojem su omogućene osnovne operacije upravljanja aplikacijom - pregled strukture aplikacije i samog koda, njegovo editiranje, te kompajliranje i pokretanje aplikacije. Što se tiče editiranja, uputno je



Slika 2.3: Pokretanje Play projekta

koristiti neki napredniji tekstualni editor ili IDE poput *IntelliJ IDEA*-e u koji ima podršku za automatsko završavanje naredbi, pomoć pri odabiru potrebnih biblioteka i slično. Aplikacija je, nakon kompajliranja i pokretanja, dostupna na URL-u `http://localhost:9000`.



Slika 2.4: Prikaz SBT sučelja nakon pokretanja aplikacije

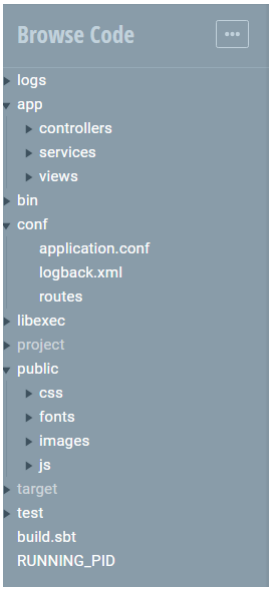
2.3 Organizacija koda Play aplikacije

Razvojni okvir, kako je dosad objašnjeno, daje smjernice programeru kako organizirati kod, odnose između dijelova koda te njegovu opću namjenu kako bi aplikacija bila što prilagodljivija i kako bi ju bilo lakše održavati.

Play organizira kod u sljedeću strukturu direktorija:

- app - izvorni kod aplikacije, **kontroleri**, **klase**, te **pogledi**, svaki u svom poddirektoriju
- public - *javascript* kod, *css* stil, fontovi te slike koje se koriste u aplikaciji, također svaki u svom poddirektoriju
- conf - konfiguracijske datoteke za aplikaciju, uključujući datoteke *application.conf* u kojoj se, između ostalog, definiraju podaci za spajanje na bazu, te *routes*, u kojoj se mapiraju rute na akcije u upravljačima
- project - konfiguracijske datoteke za SBT
- libexec - biblioteke, odnosno *.jar* datoteke koje nisu izravno uključene u generiranje bajtkoda aplikacije (npr. *.jar* datoteka za *Activator*)
- bin - izvršna datoteka za pokretanje *Activatora*

- logs - zapisi o izvođenju aplikacije
- target - kompajlirane datoteke: klase izvornog koda, razvojnog okvira (*Play-a*), datoteke s *css* ili *javascript* resursima (kompajlirani *LESS, CoffeeScript*)
- test - datoteke s testovima za aplikaciju



Slika 2.5: Organizacija koda Play aplikacije

2.4 Početna konfiguracija Play aplikacije

Predložak aplikacije u *Play-u* kreira tri **kontrolera**, od kojih je najvažniji onaj koji sadrži metodu `index`, a to je `HomeController`. Slijedi njegov kod koji je nadograđivan prilikom izrade aplikacije Redomat opisane u trećem poglavlju.

```
1  package controllers;
2
3  import play.mvc.*;
4
5  import views.html.*;
6
7  /**
8   * This controller contains an action to handle HTTP requests
9   * to the application's home page.
10  */
11 public class HomeController extends Controller {
12
13     /**
14      * An action that renders an HTML page with a welcome message.
15      * The configuration in the routes file means that
```

```
16        * this method will be called when the application receives a
17        * GET request with a path of / or /index.
18        */
19       public Result index() {
20           return ok(index.render("Your new application is ready."));
21       }
22
23   }
```

Definirana metoda `index()` je akcija koja rezultira prikazom određenog *view-a*, konkretno datoteke *index.html*. Povratni tip metode je klasa `Result`, a objekt te klase instanciramo koristeći statičku metodu `ok()` iz klase `Results`, koja je sadržana u paketu `play.mvc`. Metodi `ok()` trebamo proslijediti jedan parametar, ono što želimo prikazati na URL-u `localhost:9000/index`. To može biti *html* dokument ili obična *string* varijabla.

U ovom slučaju proslijedili smo dokument koji se u nalazi u direktoriju *views* s nazivom *index.scala.html*, iskoristivši metodu `render()` i proslijedivši joj jedan *string* parametar. Za početak je to obična poruka koju ćemo prikazati na početnoj stranici aplikacije kako bismo se uvjerili da aplikacija radi.

Sve datoteke u direktoriju *views* imaju nastavak *.scala.html* jer se moraju kompajlirati u *Scala-i* prije nego se prikažu u pregledniku.
Pogledajmo sada sadržaj datoteke *index.scala.html*.

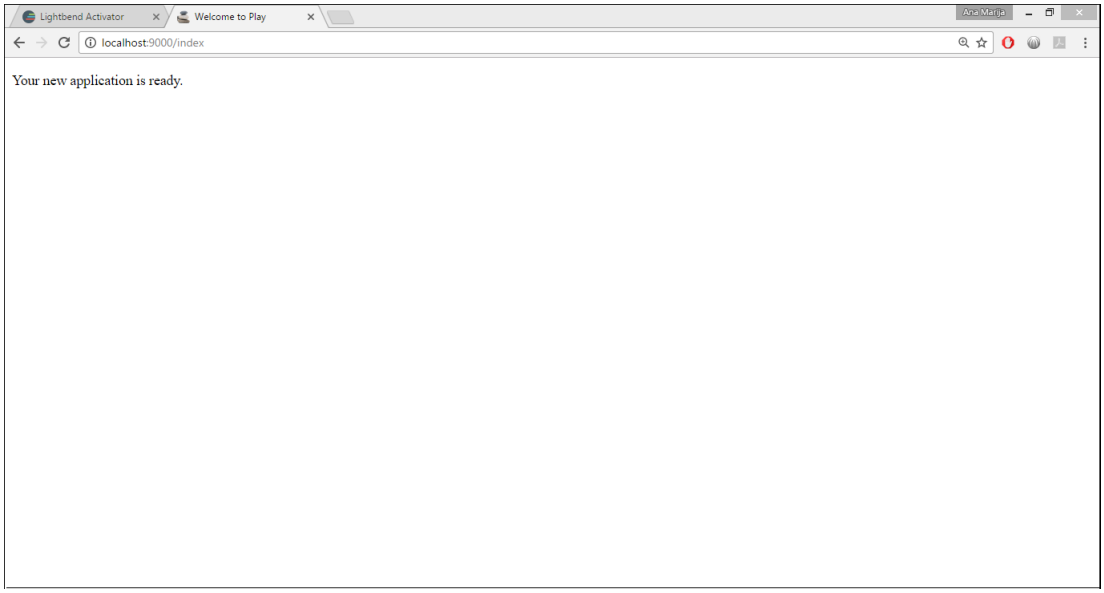
```
1  @*
2   * This template takes a single argument, a String containing a
3   * message to display.
4   *@
5  @(message: String)
6
7  @*
8   * Call the ‘main‘ template with two arguments. The first
9   * argument is a ‘String‘ with the title of the page, the second
10  * argument is an ‘Html‘ object containing the body of the page.
11  *@
12 @main("Welcome to Play") {
13
14     <p> @message </p>
15 }
```

Na samom početku definirani su parametri koje smo proslijedili metodi `render()`, u ovom slučaju je to jedan *string* parametar koji se sprema u varijablu `message`. Nakon toga slijedi poziv metode `main()` u *Scali* kojoj se također može proslijediti proizvoljan broj parametara.
Metoda `main()` će prikazati kompajlirani kod iz datoteke pod nazivom *main.scala.html*, te će proslijediti zadani parametar, još jedan *string*. Unutar vitičastih zagrada navode se metode koje će generirati sadržaj unutar dokumenta koji generira `main()`.

Datoteka *main.scala.html* predstavlja predložak, odnosno okvir unutar kojeg se dodaje sadržaj koji se mijenja s pozivima različitim metodama, odnosno akcijama iz kontrolera. Slijedi sadržaj datoteke *main.scala.html*.

```
1  @*
2   * This template is called from the ‘index‘ template. This template
3   * handles the rendering of the page header and body tags. It takes
4   * two arguments, a ‘String‘ as the title of the page and an ‘Html‘
5   * object to insert into the body of the page.
6   *@
7  @(title: String)(content: Html)
8
9  <!DOCTYPE html>
10 <html lang="en">
11     <head>
12         @* Here's where we render the page title ‘String‘. *@
13         <title>@title</title>
14         <link rel="stylesheet" media="screen" href="@routes.Assets.versioned("stylesheets/
               main.css")">
15         <link rel="shortcut icon" type="image/png" href="@routes.Assets.versioned("images/
               favicon.png")">
16         <script src="@routes.Assets.versioned("javascripts/hello.js")" type="text/
               javascript"></script>
17     </head>
18     <body>
19         @* And here's where we render the ‘Html‘ object containing
20          * the page content. *@
21         @content
22     </body>
23 </html>
```

main.scala.html sadrži kostur *html* dokumenta, zaglavlje u koje su uključene *javascript* i *css* datoteke, te <body> element unutar kojeg se generira sadržaj parametra `content`, odnosno sadržaj koji smo proslijedili metodi `main()`. U ovom slučaju to je samo jedan



Slika 2.6: Početna stranica predloška Play aplikacije

<p> element u kojem je ispisana poruka koju je proslijedila metoda `index()`. Taj dio *html* koda se 'nalijepi' unutar <body> elementa i mijenja se s pozivom različitih akcija iz kontrolera. Svaka akcija prikaže neki pogled, ali pogledi nisu nužno jedinstveni za svaku akciju. Više akcija može prikazati istu datoteku, ali s različitim parametrima, pa će se sadržaj pogleda ipak razlikovati. To je jedna od karakteristika MVC modela, ponovno korištenje koda.

Metode iz upravljača pozivaju se preko URL ruta. Mapiranje metode i odgovarajuće rute radi se u datoteci *routes*. Naziv metode ne mora odgovarati nazivu iz rute, a on ne mora biti jedinstven.

U svakom retku, s lijeve strane stoje rute, a s desne strane metode koje oni pozivaju, zajedno s paketom iz kojeg je metoda. U ruti su navedeni vrsta zahtjeva, naziv i parametri potrebni za poziv metode, međusobno odvojeni znakom /. Oblik rute je sljedeći:

VRSTA ZAHTJEVA /naziv/:parametar1/:parametar2

Pokraj metode se u zagradama navode tipovi parametara. Slijedi primjer:

```
GET   /show/:id          controllers.HomeController.showPerson(id: Int)
```

Dakle, kada se uputi GET poziv na URL `http://localhost:9000/show/5`, pozvat će se metoda `showPerson()` s vrijednošću parametra `id` pet, unutar upravljača `HomeController` koji se nalazi u paketu `controllers`.

Ako rute nisu dobro konfigurirane, npr. broj parametara ne odgovara onom na mjestu implementacije ili se navede krivi naziv metode, *SBT* će javiti pogrešku pri kompajliranju. Pri izvođenju aplikacije, *Play* za konkretni URL traži u ovoj datoteci odgovarajuću rutu. Ako ne nađe redak koji odgovara nazivom i parametrima, javlja grešku u pregledniku. U slučaju da više ruta odgovara, *Play* će odabrati prvu na koju naiđe, zato redoslijed upisivanja ruta treba biti od najbitnije prema manje bitnima.

Poglavlje 3

Web aplikacija Redomat

3.1 Motivacija

Kao primjer aplikacije nastale korištenjem razvojnog okvira *Play*, napravljena je aplikacija Redomat. Redomat je aplikacija koja služi generiranju rednih brojeva za različite usluge. Takva aplikacija može pronaći primjenu u bankama, bolnicama, ustanovama za izradu dokumenata (policijske uprave, razni zavodi), referadama i slično.

Generiranje brojeva tradicionalno obavljaju uređaji postavljeni u ustanovama i kojima se ti brojevi koriste. Takvi uređaji ispisuju brojeve na listiće koje korisnici predaju kada dođu na red za traženu uslugu. Nekada ispisuju samo broj i vremensku oznaku kada je on generiran, dok su neki napredniji, pa odmah ispišu i koliko je brojeva trenutno u redu za obavljanje te usluge.

Glavni cilj aplikacije koja bi služila u istu svrhu bio bi omogućavanje korisnicima da se klikom na određenu uslugu pretplate na tu uslugu. To znači da će dobiti redni broj, samo ovaj puta ne na papiru, nego na uređaju s kojeg je pristupio aplikaciji. Taj broj treba biti jedinstven jer korisnik s njim, umjesto s brojem na papiru, dokazuje svoj red za obradu. Aplikacija za generiranje brojeva također treba u svakom trenutku omogućiti korisniku da ažurira stanje reda, odnosno da vidi koliko je još korisnika prije njega u redu. Korisnik koji se pretplatio putem aplikacija mora imati sve informacije kao i korisnik koji je u tom trenutku u konkretnoj ustanovi.

Prednost pretplaćivanja na uslugu putem ovakve aplikacije, umjesto u samoj ustanovi, je očita - korisnik to može napraviti i kada nije fizički u samoj ustanovi. Time korisnik štedi vrijeme, a u ustanovama su manje gužve, pogotovo u onima u kojima se pretplaćuje puno korisnika. Također, smanjuje se broj neiskorištenih brojeva jer korisnik može prije pretplaćivanja na uslugu provjeriti koliko korisnika je već u redu i procijeniti odgovara li mu da čeka toliko. S druge strane, korisnik putem aplikacije može provjeravati koliko je još korisnika u redu prije njegovog rednog broja i tako ne propustiti svoj broj.

Funkcionalnosti aplikacije

Osnovne funkcionalnosti aplikacije su pretplaćivanje na uslugu, pregledavanje rednih brojeva za usluge na koje se korisnik već pretplatio, zatim provjera reda za određenu uslugu i na kraju obrada korisnika, odnosno otpuštanje broja.

Dodatna funkcionalnost je biranje lokacije ustanove u kojoj se korisnik želi pretplatiti. Na primjer, ako aplikacija omogućuje pretplaćivanje na usluge u nekoj banci, nudi mu se mogućnost biranja konkretne poslovnice. To je omogućeno na prikazu poslovnica na karti, gdje se poslovnice ističu kao lokacije na *Google karti* ili na popisu u obliku tablice.

Aplikacija ima dva tipa korisnika. Prvi tip je obični korisnik koji se ne ulogirava i koji se koristeći svoj uređaj može pretplatiti na ponuđenu uslugu. Drugi tip je administrator koji se ulogirava i ima mogućnost otpuštanja brojeva, odnosno poništavanje pretplate (uslijed obrade korisnika ili njegovog nepojavljivanja).

Kako bismo pokazali navedene funkcionalnosti aplikacije, napravljena je aplikacija koja omogućava pretplatu za usluge u nekoj banci. U ovoj aplikaciji područje je ograničeno na područje grada Zagreba.

3.2 Izgled aplikacije

S obzirom da se radi o web aplikaciji, koristiti se može na bilo kojem uređaju koji ima instaliran preglednik i pristup internetu. Kako bi se aplikacija dobro prikazivala na različitim uređajima, korišten je web predložak izgleda, odnosno dizajna aplikacije preuzet sa stranice w3layouts.com.

Svi predlošci na ovoj stranici imaju *css* datoteke koje definiraju klase s obzirom na uređaj na kojem je aplikacija otvorena. Klase određuju veličinu i položaj elemenata tako da krajnji izgled bude što pravilniji. To znači da nam predložak osigurava da se elementi ne preklapaju i da su svi dijelovi aplikacije vidljivi na većini uređaja. Cjelokupni kod aplikacije dostupan je na GitHub-u.

Pravilo u *css-u* koje ispituje na kakvom je uređaju aplikacija prikazana je `@media`. Ono može ispitivati razna svojstva uređaja na kojem se sadržaj prikazuje. Koje svojstvo ili svojstva je potrebno ispitati ovisi o dizajnu, odnosno o tipovima elemenata i njihovom međusobnom odnosu. U predlošku preuzetom za ovu aplikaciju bitna je bila širina ekrana, pa se to provjerava `@media` pravilom.

Pravila se definiraju na sljedeći način: u oblim zagradama nakon selektora `@media` navodi se maksimalna širina ekrana za koju će vrijediti pravila uređivanja koja se nalaze u produžetku u vitičastim zagradama. Slijedi dio datoteke *theme-style.css* s objašnjenim pravilom:

```
...
/*------Start-Media-quries------*/
/*----start-media-quries-for-1440px-Laptops-----*/
```

```
@media (max-width:1440px){
 .header-note h1 {
  font-size: 4em;
 }
 .header-note p {
  font-size: 1.32em;
 }
 .header-note {
  padding-top: 6.5em;
 }
...
}
/*----//End-media-quries-for-1440px-Laptops-----*/
/*----start-media-quries-for-1366px-Laptops-----*/
@media (max-width:1366px){
 .header-note h1 {
  font-size: 4em;
 }
 .header-note p {
  font-size: 1.32em;
 }
 .header-note {
  padding-top: 4.5em;
 }
...
}
/*----//End-media-quries-for-1366px-Laptops-----*/
...
```

Izgled aplikacije na različitim uređajima može se provjeriti u svakom pregledniku uključi li se opcija *Alati za razvojne programere*. S obzirom da se administratori moraju prijaviti kako bi koristili aplikaciju, a ostali korisnici se ne prijavljuju, bilo bi dobro ne prikazivati formu za prijavu svim korisnicima. Osim odvojenih početnih stranica za administratore i ostale korisnike, efektan način za to je *css*.

Naime, korisnici koji se pretplaćuju za neku uslugu trebaju donijeti uređaj s kojeg su se pretplatili i pokazati broj na istome. Stoga je za očekivati da će to biti neki prijenosni uređaj, pametni mobitel ili tablet. S druge strane, administratori će uvijek raditi na stolnim računalima. Prema tome, moguće je odrediti minimalnu širinu ekrana na kojoj će se forma prikazivati. Jednostavno se definira klasa koja je vidljiva kada je širina ekrana veća od određene, a sakrivena kada je širina ekrana manja od te vrijednosti. Taj efekt postiže se pomoću `@media` pravila, što je već objašnjeno.

3.3 Pogledi

U prethodnom poglavlju prikazano je kako izgledaju pogledi aplikacije napravljene u razvojnom okviru *Play* i objašnjeno je da se prilikom prikazivanja nekog pogleda pozove metoda koja prikaže sadržaj tog pogleda unutar *scala.html* datoteke koja nosi isti naziv kao i metoda.

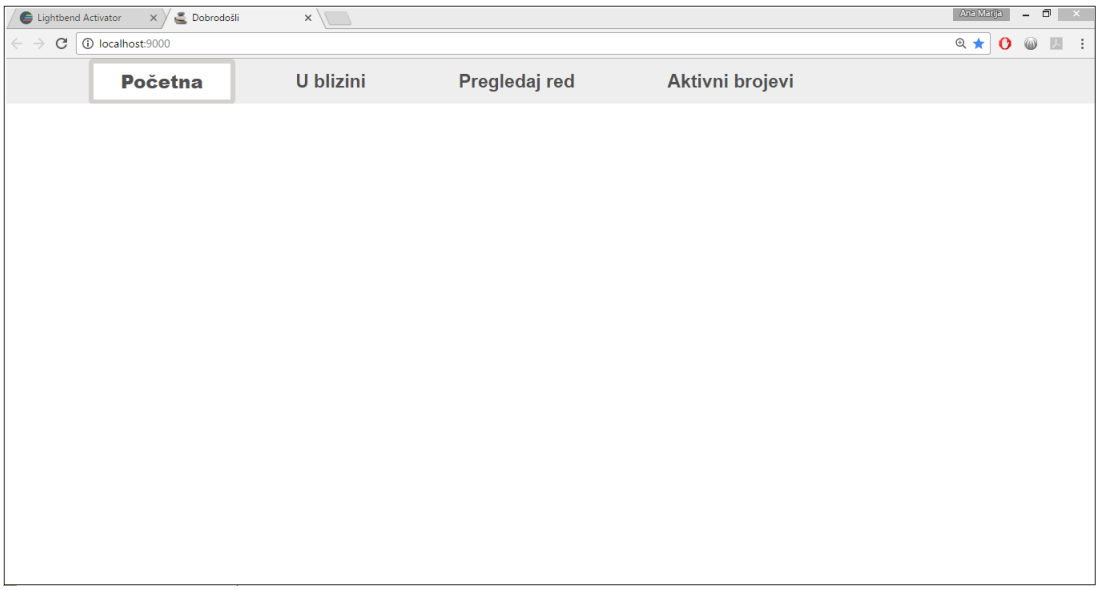
S obzirom da se radi o aplikaciji koja čini jednu cjelinu, u svim pogledima želimo

imati nešto zajedničko, npr. navigacijski panel. Na njemu između ostaloga treba uvijek biti i poveznica na početnu stranicu aplikacije kako bi se korisnik u bilo kojem trenutku mogao vratiti na početak. *MVC* obrazac također nalaže da postoji neki okvir ili kostur u kojem postoji dio koji se mijenja pozivom različitim metodama i generiranjem odgovarajućeg pogleda. U tom okviru definiran je `<html>` selektor unutar kojeg se dodaju drugi *html* elementi.

U razvojnom okviru *Play* to je izvedeno tako što se u svakom pogledu definira koji će okvir biti generiran. U početnom primjeru to je bila datoteka *main.scala.html*, a pozvana je funkcijom `@main()`. Prilikom poziva, mogu se definirati parametri i tijelo funkcije. Tijelo je *html* kod koji se unutar okvira generira na mjestu poziva funkcije `@content`. Parametri se spremaju u varijable, a mogu biti podaci proslijeđeni od strane servera ili obične poruke koje ne ovise o stanju na serveru. Konkretno, u aplikaciji Redomat koriste se npr. za prikaz korisničkog imena kada se administrator prijavi. Također, proslijeđuje se naslov pogleda koji se generira unutar selektora `<title>`. Slijedi primjer okvira unutar kojeg se generira početna stranica aplikacije. Naziv datoteke je *main.scala.html*.

```
1  @(title: String)(content: Html)
2
3
4  <!DOCTYPE html>
5  <html lang="en">
6   <head>
7    <meta charset="utf-8">
8    <meta http-equiv="X-UA-Compatible" content="IE=edge">
9    <meta name="viewport" content="width=device-width, initial-scale=1">
10
11   <title>@title</title>
12   <link href="@routes.Assets.versioned("css/bootstrap.css")" rel='stylesheet' type='text/
         css' />
13   <link href="@routes.Assets.versioned("css/maps.css")" rel='stylesheet' type='text/css' /
         >
14   <!-- jQuery (necessary for Bootstrap's JavaScript plugins) -->
15   <script src="https://ajax.googleapis.com/ajax/libs/jquery/1.12.4/jquery.min.js"></script
         >
16   <script src="@routes.Assets.versioned("js/bootstrap.min.js")"></script>
17   <script src="@routes.Assets.versioned("js/storage-data.js")"></script>
18   <script src="@routes.Assets.versioned("js/customer.js")"></script>
19   <script src="@routes.Assets.versioned("js/date.js")"></script>
20   <script src="@routes.Assets.versioned("js/chosen.jquery.min.js")"></script>
21   <!-- Custom Theme files -->
22   <link rel="stylesheet" href="@routes.Assets.versioned("css/theme-style.css")" />
23   <link rel="stylesheet" href="@routes.Assets.versioned("css/font-awesome.min.css")" />
24   <link rel="stylesheet" href="@routes.Assets.versioned("css/chosen.min.css")" />
25   <link rel="icon" href="@routes.Assets.versioned("images/favicon.png")" type="image/x-
         icon" />
26  </head>
27  <body>
28   <!--start-top-nav-->
29   <nav class="subMenu navbar-custom navbar-scroll-top" role="navigation">
30    <div class="container">
31     <div class="navbar-header page-scroll">
32      <button type="button" class="navbar-toggle" data-toggle="collapse" data-target=".
           navbar-main-collapse">
```

```
33          <img src="@routes.Assets.versioned("images/menu-icon-grey.png")" title="drop-down-
                menu" />
34        </button>
35       </div>
36       <div class="collapse navbar-collapse navbar-main-collapse top-nav">
37        <ul class="nav navbar-nav full-nav-top">
38         <li>
39         <a id="s1" class="subNavBtn" href="@routes.HomeController.index">Početna</a>
40         </li>
41         <li>
42         <a id="s2" class="subNavBtn" href="@routes.UserController.map">U blizini</a>
43         </li>
44         <li>
45         <a id="s3" class="subNavBtn" href="@routes.UserController.queue">Pregledaj red </a>
46         </li>
47         <li>
48         <a id="s4" class="subNavBtn" href="@routes.UserController.active">Aktivni brojevi</a
               >
49         </li>
50        </ul>
51       </div>
52      </div>
53     </nav>
54     <!--//End-top-nav-->
55     @content
56
57   </body>
58 </html>
```



Slika 3.1: Kostur aplikacije

Za prikaz stranica se osim *html-a* i *css-a* koristi i *javascript*. Glavnina *javascript* koda nalazi se u posebnim *js* datotekama koji se nalaze u direktoriju *public/js*. Manji dio nalazi

se u samim *html* datotekama. Razlog tome je dodavanje svojstava elementima u aplikaciji (uglavnom gumbima), odnosno povezivanje događaja s pozivima funkcija čiji se kod nalazi u *js* datotekama.

Tijelo funkcije ne stavlja se u *html* datoteku iz više razloga. Prvi razlog je činjenica da većina tih funkcija neće biti izvršena pri svakom prikazu *html* datoteke. Stoga nije poželjno da se kod nepotrebno učitava pri svakom pozivu tog konkretnog pogleda. Drugi razlog je ponovno korištenje koda. Naime, često se događa da je istu funkcionalnost potrebno implementirati na više mjesta u aplikaciji. Odvajanjem koda u posebnu *js* datoteku omogućeno je korištenje funkcionalnosti iz više različitih pogleda, a da se kod pritom ne duplicira. Zadnji i ne manje važan razlog je organizacija koda. Odvajanje implementacije funkcionalnosti u zasebne datoteke rezultira preglednijim kodom koji je lakše održavati, nadograđivati i proučavati.

Izdvajanje koda je također jedna od karakteristika *MVC* oblikovnog obrasca upravo iz navedenih razloga. U aplikaciji Redomat se *javascript* kod nalazi u nekoliko datoteka, od kojih je većina preuzeti kod koji dolazi s dodacima koji su korišteni u aplikaciji. Dodaci su zapravo *javascript* biblioteke koje olakšavaju rukovanje događajima i olakšavaju korištenje *javascript* tipova podataka i funkcija. Te biblioteke organizirane su u API-ije - skupove funkcija i procedura koje omogućavaju jednostavnije korištenje *javascript* funkcija i ugrađenih tipova podataka. Željene funkcionalnosti dobivaju se s manje koda i konačni kod je pregledniji i kompaktniji.

Dva najraširenija dodatka, koja su ukomponirana i u aplikaciju Redomat, su *jQuery* i *Bootstrap*. *jQuery* omogućuje jednostavnije upravljanje *html* datotekama, događajima, animaciju elemenata u aplikaciji te vrlo jednostavno korištenje *Ajax-a*. *Bootstrap* je skup *css* pravila i *javascript* svojstava koji zajedno oblikuju izgled *html* elemenata. Zato *Bootstrap* nije obični dodatak već čitav okvir za prikaz *html* stranica krajnjem korisniku. S obzirom da omogućuje korištenje standardiziranog prikaza u većini preglednika, nalazi primjenu u web-aplikacijama gotovo bez iznimke.

Od ostalih dodataka korišten je još *date.js* za obradu vremenskih žigova i *chosen.js* za animirani padajući izbornik.

Date

Dodatak *date.js* korišten je za obradu vremenskih žigova. Čim korisnik pristupi aplikaciji, inicijalizira se struktura u koju se spremaju podaci o dodijeljenim rednim brojevima. Ona se učitava svaki put kada se učita datoteka *main.scala.html* i provjerava stanje.

Ako struktura nije inicijalizirana u *local storage-u* [13] preglednika s kojeg je aplikacija otvorena, ona se inicijalizira s trenutnim datumom. Ako je struktura već inicijalizirana, to znači da već postoje neki podaci u istoj, pa treba provjeriti datum.

U slučaju da je datum nije jednak trenutnom, struktura se reinicijalizira jer redni brojevi

[13] Objekt ugrađen u preglednik u koji se mogu spremati podaci u JSON obliku.

od nekog drugog datuma više ne vrijede. Kada datum odgovara trenutnom, neće se ništa promijeniti. Izgled ove strukture mijenja se po potrebi, odnosno ovisno o tipu usluga koje ustanova nudi. Slijedi sadržaj datoteke *storage-data.js*.

```
1  $('document').ready(function(){
2  var services = JSON.parse(localStorage.getItem('BankServices'));
3
4  if (localStorage.getItem('BankServices') == undefined || localStorage.getItem('
       BankServices') == 'null' )
5  {
6   setInitial();
7  }
8  else
9   if (services["Datum"] != null)
10  {
11   var date = Date.today();
12   today = date.toString('dd-MMM-yyyy');
13   date = new Date(services["Datum"]);
14   var serviceDate = date.toString('dd-MMM-yyyy');
15   if (today != serviceDate)
16   setInitial();
17  }
18 });
19
20 function setInitial()
21 {
22  var initial = {};
23  // fill the localStorage data structure with the desired services
24  initial["Transakcije"] = [];
25  initial["OsobniBankar"] = [];
26  initial["Studenti"] = [];
27  initial["Datum"] = new Date().getTime();
28  localStorage.setItem('BankServices', JSON.stringify(initial));
29 }
```

Osim navedene upotrebe, dodatak *Date* se koristi i pri ispisivanju točnog vremena kada se korisnik pretplatio na uslugu. Naime, iz vremenskog žiga nije moguće jednostavno dobiti samo vrijeme ili samo datum običnim *javascript-om*. Zato se koriste razni dodaci kao što je ovdje opisani *Date*.

U nastavku slijedi prikaz korištenja padajućeg izbornika iz dodatka *chosen.js*, te korištenje funkcije `ajax` [14] iz dodatka *jQuery*. Funkcija *ajax* omogućava jednostavno slanje XMLHttp zahtjeva serveru, a za komunikaciju sa serverskim skriptama (u *MVC-u* su to metode upravljača) koristi različite formate, kao što su npr. *JSON* i *html*. Više o tipu podataka koje upravljači šalju u razvojnom okruženju *Play* piše u sljedećem potpoglavlju.

[14] Asynchronous JavaScript and XML

Chosen

Dodatak *chosen* koristi se na *html* elementu `<select>`, kojem se definira klasa `chosen`. Opcije se mogu dodati dinamički iz strukture koju proslijeđuje upravljač. U pogledu *freeNumber.scala.html* prikazuje se popis poslovnica kojima administrator može upravljati u obliku padajućeg izbornika. Podaci o poslovnicama proslijeđeni su u obliku klase `BankData`, pa na samom početku dokumenta stoji linija:

```
@import services.BankData
```

kojom uvozimo klasu čiji je kod spremljen u direktoriju *app/services*. Sami podaci proslijeđeni su kao rezultat odgovarajuće metode koja je pozvala ovaj pogled. Rezultat je oblika liste instanci klase *BankData*, spremljen u varijablu `result`. Pogledu *freeNumber.scala.html* proslijeđeni su još neki podaci, a svi oni spremljeni su u varijable u liniji koja slijedi odmah iza definicije svih uvoza:

```
@(user: String)(message: String)(result: List[BankData])
```

Preostalo je pokazati *javascript* kod koji dodaje opcije iz liste u padajući izbornik:

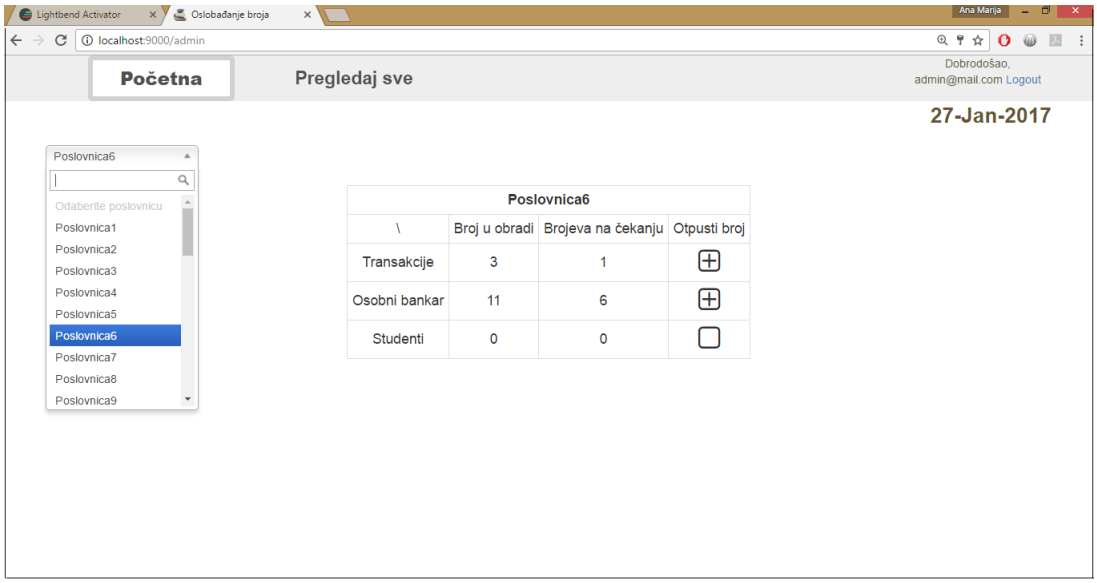
```
1  for (var i = 0; i < offices.length; i++) {
2   $('.chosen').append($('<option>', {value: (i + 1), text:offices[i]['name']})); }
```

te samu inicijalizaciju dodatka *chosen*:

```
$('.chosen').chosen();
```

i dodavanje događaja svakoj od tih opcija - odabirom određene poslovnice, prikažu se podaci o redu u svakoj od njih.

```
1  $('.chosen').on('change', function(evt, params) {
2    printOneOffice(offices[params.selected-1], params.selected);
3  });
```



Slika 3.2: Korištenje elementa chosen

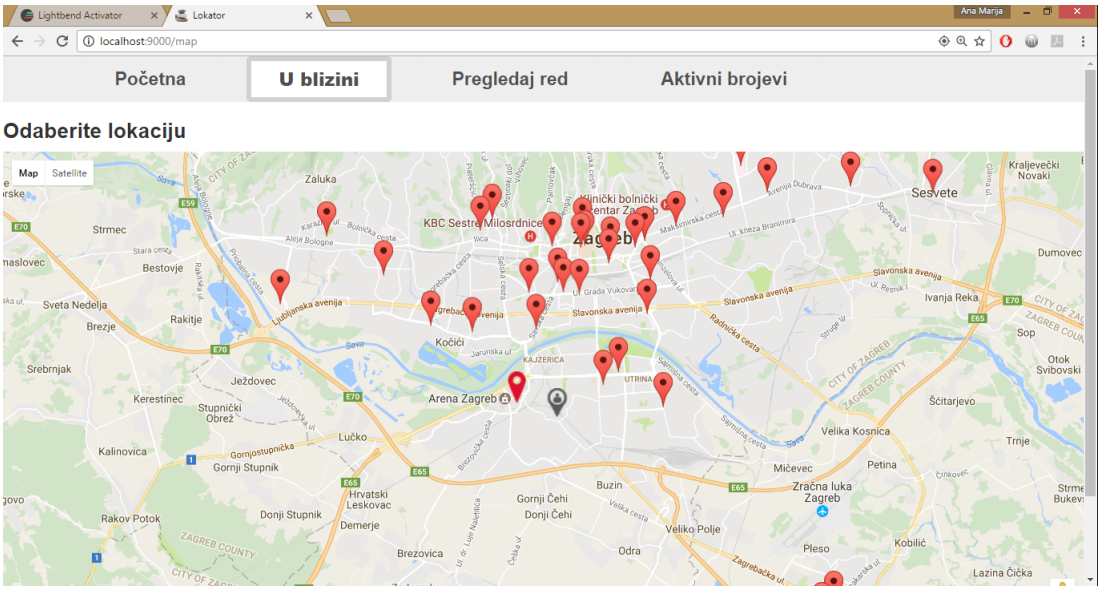
jQuery ($.ajax)

Kada se prikažu podaci o redovima u konkretnoj poslovnici, potrebno je omogućiti administratoru da otpušta brojeve. U tu svrhu korištena je funkcija `ajax` kojom se poziva metoda iz upravljača *AdminController*. Ta metoda promijeni stanje u bazi i vrati nove podatke o stanju u toj poslovnici koji se trebaju prikazati.

```
1  $.ajax({
2    type: 'POST',
3    url: 'freeNumber/' + service + "/" + id + "/" + len,
4    dataType: "text",
5    success: function(data) {
6      offices[id-1] = JSON.parse(data);
7      printOneOffice(offices[id-1], id);
8    },
9    error : function(data) {
10     console.log('Error : freeNumber');
11   }
12 });
```

Google Maps

Osim navedenih dodataka, korišten je i *Google Maps*, API za prikaz *Google karti*. On se koristi u prikazu poslovnica na karti za korisnike kako bi im se omogućilo brže pronalaženje poslovnice u kojoj se žele pretplatiti. Osim samog prikaza poslovnica na karti, posebno se istakne poslovnica najbliža trenutnoj lokaciji korisnika kako bi se lakše snašao. *Javascript* kod za prikaz *Google karti* nalazi se većim dijelom u datoteci *map.scala.html*, kao i poveznica na API. Posljedica toga je da nije dodavan nikakav *javascript* dodatak u direktorij *public/js*, već se kod API-ja pri svakom prikazu pogleda *map.scala.html* učitava s adrese u zadanoj poveznici.



Slika 3.3: Prikaz Google karte u aplikaciji

3.4 Upravljači

Aplikacija se sastoji od tri upravljača koji sadrže metode za generiranje pogleda namijenjenih administratorima (*AdminController*), korisnicima (*UserController*) te i jednima i drugima (*HomeController*).

Rute

Kao što je već objašnjeno, u datoteci *routes* definirane su metode koje se pozivaju unosom određenog URL-a. Slijedi dio sadržaja datoteke *routes*.

```
# Routes
# This file defines all application routes (Higher priority routes first)
# ~~~~

# An example controller showing a sample home page
GET     /                           controllers.HomeController.index
# getUser
POST    /login                      controllers.AdminController.getUser

...

# chooseService action in UserController
GET     /chooseService/:id          controllers.UserController.chooseService(id: Int)
# getNumber action in UserController
POST     /getNumber/:service/:id/:activeNum/:activeId    controllers.UserController.
getNumber(service: String, id: Int, activeNum: Int, activeId: Int)

...

# Map static resources from the /public folder to the /assets URL path
GET     /assets/*file               controllers.Assets.versioned(path="/public", file: Asset)
```

Sve metode u upravljačima moraju u konačnici vratiti neki pogled. Tri su načina za to: vraćanje predefinirane metode `ok()`, vraćanje druge metode (odnosno pogled koji ona vraća) ili vraćanje metode `redirect()`.
Metoda `ok()` prima kao parametar metodu `render()` nad nekim pogledom, kako je prije opisano. Slijedi primjer.

```
1  public Result index() {
2   return ok(index.render("Obavite sve i ušedite vrijeme", "",  formFactory.form(User.class)
       ));
3  }
```

U slučaju da želimo promijeniti tok metode ovisno o nekim podacima koji su joj proslijeđeni, vratit ćemo metodu na koju želimo preusmjeriti tok. Na primjer, u metodi `getUser()` provjeravamo na koji je gumb korisnik kliknuo i ovisno o tome kao rezultat vratimo jednu od metoda `saveUser()` ili `authenticate()`. One će obaviti neki dio poslovne logike i u konačnici vratiti pogled na jedan od tri navedena načina. Tako se mogu više puta ugnježđivati pozivi metodama iz upravljača. Slijedi primjer.

```
1  public Result getUser() throws SQLException {
2   DynamicForm requestData = formFactory.form().bindFromRequest();
3   if ("SignUp".equals(requestData.get("action")))
4    return saveUser(requestData);
5   else
6    if ("Login".equals(requestData.get("action")))
7     return authenticate(requestData);
8   return badRequest("This action is not allowed");
9  }
```

Kada je potrebno preusmjeriti korisnika sa serverske strane, koristi se metoda `redirect()`. Njoj se proslijedi `string` - naziv rute, te ostali parametri ako su potrebni. Na taj se način u korisnikovom web pregledniku nađe nova ruta, a metoda koja je zapisana u datoteci *routes* pod tim nazivom generira novi pogled. Slijedi primjer.

```
1  public Result logout() {
2   session().remove("connected");
3   return redirect("/");
4  }
```

Metode koje će biti pozivane s klijentske strane moraju imati modifikator pristupa `public`. One koje će biti pozivane samo sa serverske strane, odnosno iz druge metode, mogu imati modifikator pristupa `private`. Ako je potrebno metodu zvati iz drugih upravljača, modifikator pristupa može biti `protected` (pristup toj metodi onda imaju svi iz paketa `controllers`).

HomeController

HomeController sadrži jednu metodu. To je metoda `index()` koja generira početnu stranicu aplikacije (istu za administratore i korisnike).

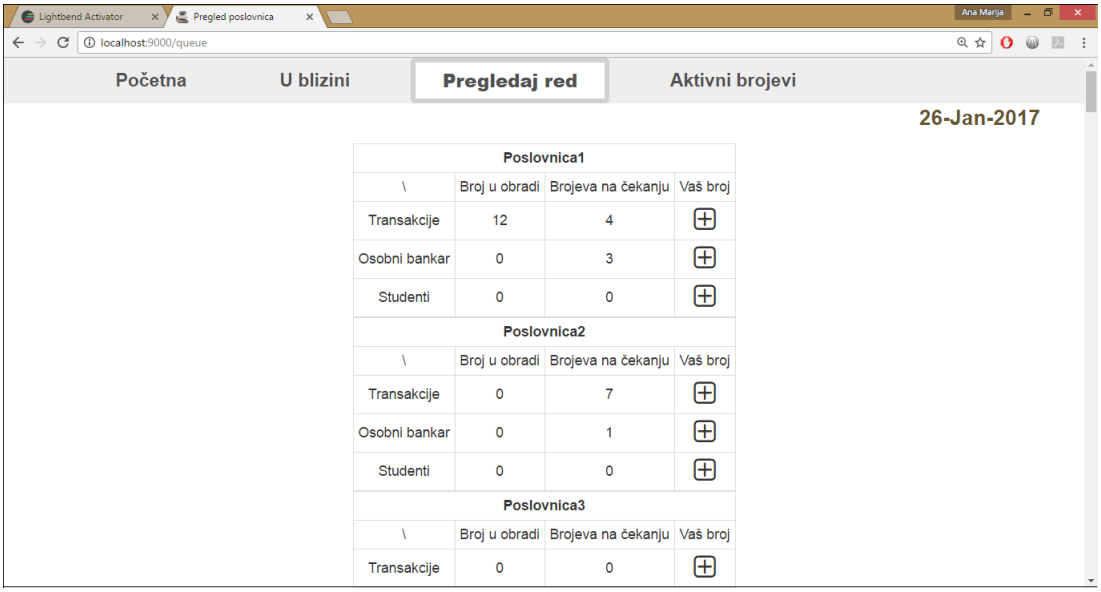
UserController

UserController sadrži metode potrebne za implementaciju funkcionalnosti koje su dostupne korisnicima koji se ne prijavljuju. Korisnik ima tri osnovna pogleda u aplikaciji, a to su *Google karta*, prikaz redova po svim poslovnicama u obliku tablice, te prikaz brojeva za usluge na koje je pretplaćen. Broj, dakle, može uzeti na dva načina.

Prvi način je pronalaženje željene poslovnice na karti koju prikazuje metoda `map()`. Do nje dolazi klikom na poveznicu *U blizini*. Klikom na konkretnu poslovnicu prikazuje se novi pogled u kojem odabire uslugu na koju se želi pretplatiti (metoda `chooseService()`).

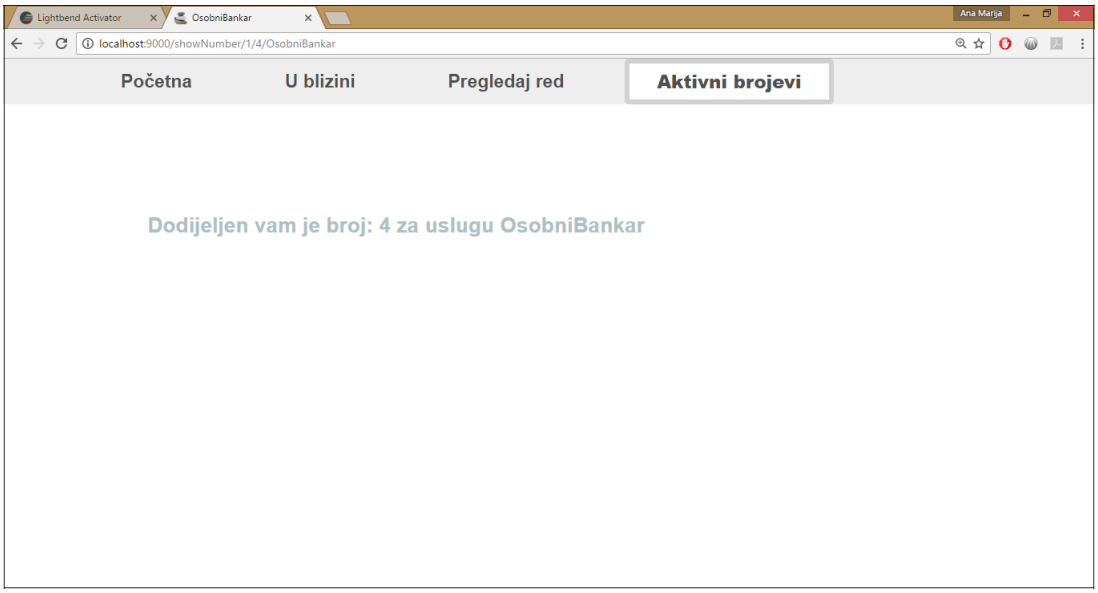
Drugi način je pronalaženje poslovnice u tablici i odabir usluge. Tablicu prikazuje metoda `queue()`, a poziva se klikom na poveznicu *Pregledaj red*.

Korisniku je onemogućeno uzeti broj za uslugu za koju već ima valjani broj, u bilo kojoj poslovnici. Prilikom prikazivanja tablice, kod usluge za koju nije trenutno pretplaćen stoji znak plus, koji signalizira da se korisnik može pretplatiti za broj. Klikom na znak plus, poziva se metoda `getNumber()` koja ažurira bazu i sprema broj u *local storage* korisnikovog



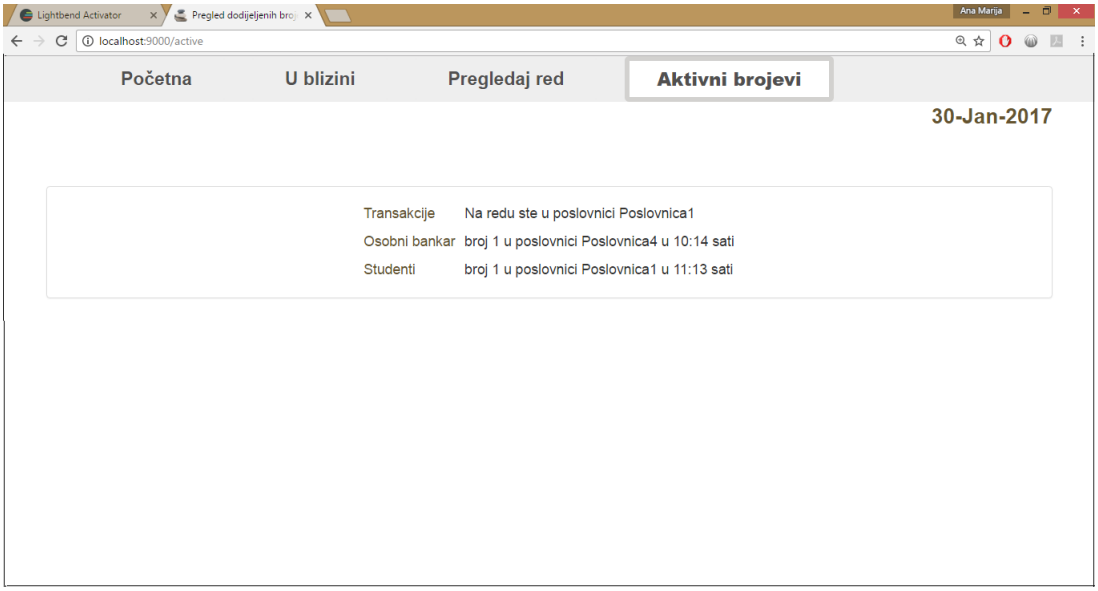
Slika 3.4: Prikaz redova u svim poslovnicama na jednom mjestu

preglednika. Kada završi metoda `getNumber()`, korisniku je prikazuje obavijest o broju koji mu je dodijeljen, a taj pogled vraća metoda `showNumber()`. Ako je nešto krenulo krivo, npr. došlo je do greške u bazi, metoda `numberError()` će obavijestiti korisnika kako njegov zahtjev nije proveden.



Slika 3.5: Obavijest o dodijeljenom broju

Kada se korisnik pretplati na uslugu, ne mora držati otvorenu stranicu sve do obavljanja usluge, već u svakom trenutku može pregledati brojeve koji su mu dodijeljeni. Klikom na poveznicu *Aktivni brojevi*, prikazuje mu se tablica s podacima o poslovnicama i brojevima za usluge koje ustanova (u aplikaciji Redomat konkretno banka) nudi.



Slika 3.6: Pregled dodijeljenih brojeva

AdminController

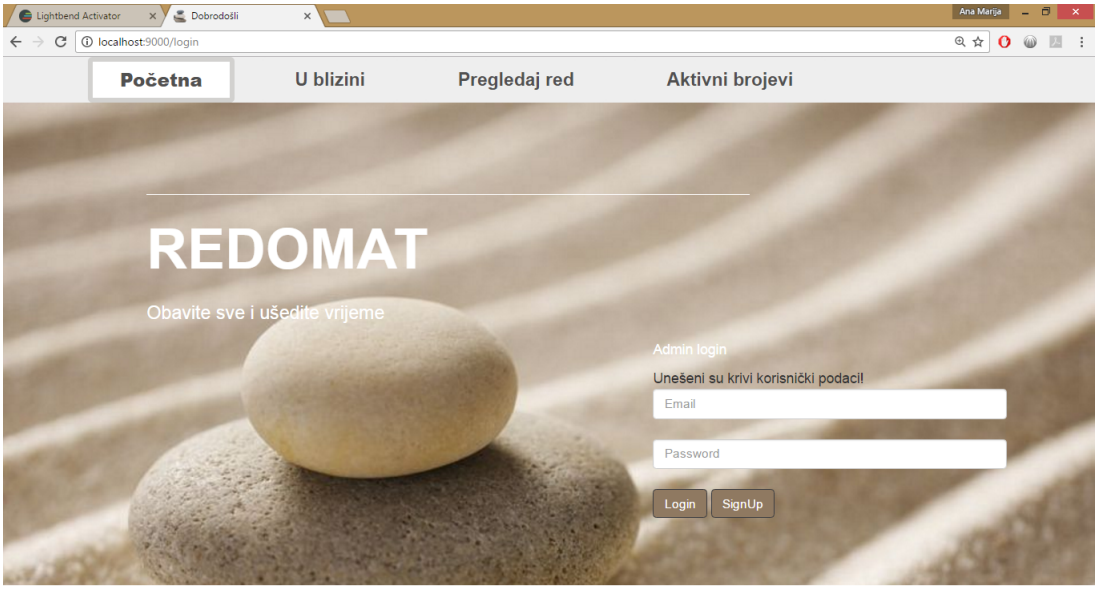
AdminController sadrži metode potrebne za implementaciju funkcionalnosti administratora. Administrator se mora prijaviti kako bi mu se omogućili drugačiji pogledi nego običnom korisniku koji se pretplaćuje na usluge. Forma za prijavu nalazi se na početnoj stranici aplikacije.

Ispod forme nalaze se dva gumba. Jedan je za prijavu, a drugi za registraciju. U stvarnom svijetu, ne bi bilo potrebno imati mogućnost registracije, štoviše, ne bi imalo smisla. Administratori bi već morali biti u sustavu ustanove u kojoj se primjenjuje aplikacija i ne bi se posebno registrirali. Međutim, kako je Redomat aplikacija koja ima obrazovnu svrhu, stavljena je mogućnost registracije. Unosom lozinke putem forme za prijavu, ista se sprema u bazu metodom *hashiranja*, odnosno enkripcije. Tako je moguće simulirati dekriptiranje lozinke, koja u bazi nikad ne smije biti spremljena u nekriptiranom obliku, u metodi koja provjerava akreditaciju prilikom prijave. U nastavku će biti objašnjen postupak.

Klikom na jedan od gumba ispod forme, poziva se metoda `getUser()` koja dohvati podatke iz forme i pozove drugu metodu, ovisno o tome na koji je gumb kliknuto.

Ako se radilo o registraciji, akcija se preusmjerava funkcijom `redirect` na metodu `saveUser()`. Ta metoda kreira novi objekt klase `User` koji sadrži *email* i enkriptiranu *lozinku*. Kreirani objekt se zatim sprema u bazu, čime je dodan novi administrator.

Nakon registracije, moguća je prijava administratora. Metoda `getUser()` ovaj puta preusmjerava akciju na metodu `authenticate()`. Ta metoda provjerava postoji li administrator s unešenom akreditacijom i ako postoji, preusmjerava akciju na početnu stranicu za administratora. Ako unešeni podaci nisu ispravni, obavještava korisnika o istome.



Slika 3.7: Greška pri prijavi

Osim provjere podataka, metoda `authenticate()` mora nekako zabilježiti podatak o tome da je netko ulogiran i tko je ulogiran. *Play* ima ugrađen sustav gdje se ti osnovni podaci mogu spremiti. Aplikacija kreira *cookie* [15] s nazivom `PLAY_SESSION` koji se sprema u korisnikov preglednik. Postoje dva načina prosljeđivanja podataka *HTTP* zahtjevima u *Play-u*, a razlika je u njihovom dosegu. `session()` će biti valjan dok god traje sesija korisnika (odnosno dok korisnik ne ugasi preglednik ili se odjavi), a `flash()` samo za idući *HTTP* zahtjev (odnosno poziv sljedeće metode). Podaci se ne spremaju nigdje na serveru, već se pri prosljeđivanju svakog zahtjeva dodaju u zaglavlje koristeći kreirani *cookie*.

U aplikaciji Redomat korišten je `session()`. Podaci se spremaju slično kao u strukturi `Map`, postoje ključ i vrijednost koji oba moraju biti tipa `String`. Količina podataka je ograničena na oko četiri kilobajta s obzirom da se spremaju u *cookie*. Slijedi kod metode `authenticate()`.

[15] Mala količina tekstualnih podataka koju izmjenjuju preglednik i aplikacija na poslužitelju.

```
1  private Result authenticate(DynamicForm requestData) throws SQLException {
2
3   String email = requestData.get("email");
4   String password = requestData.get("password");
5
6   if (User.existsInDatabase(email, password)) {
7    session("connected", email);
8    return redirect("/admin");
9   }
10
11  return ok(index.render("Obavite sve i ušedite vrijeme", "Unešeni su krivi korisnički
        podaci!", formFactory.form(User.class)));
12 }
```

Početna stranica za administratora generira se metodom `admin()`. Ona sadrži jedan padajući izbornik u kojem administrator može odabrati poslovnicu u kojoj želi otpuštati brojeve. Podaci o poslovnici se učitavaju odabirom jedne od poslovnica u tom padajućem izborniku. Već opisanom funkcijom `$.ajax`, pozove se metoda `viewOne` koja vrati potrebne podatke. Administrator, slično kao i korisnik, klikom na znak plus stavlja idući broj na redu u obradu. Poziva se metoda `freeNumber()` koja ažurira stanje u bazi i vraća podatke o tome kako bi se ažurirao pogled. *Pogled* samo prikazuje podatke koji su mu proslijeđeni, pa tako ne zna je li obrađeni broj bio zadnji u redu za tu uslugu ili ih ima još. Tu informaciju mu proslijeđuje upravljač jer je to dio poslovne logike, po principu *MVC* obrasca. Informaciju o tome ima li još brojeva (odnosno korisnika) spremnih za obradu, upravljač dobiva iz modela. O tome malo detaljnije u idućem potpoglavlju.

Administrator može pregledati i stanje u svim poslovnicama odjednom. Klikom na poveznicu *Pregledaj sve*, poziva se metoda `viewAll()`. Ona generira pogled u kojem su prikazani podaci o svim poslovnicama, sličan pogledu običnog korisnika. U tom načinu prikaza, administrator također može pozivati metodu `freeNumber()` klikom na znak plus. Kada je gotov s radom, administrator se odjavljuje klikom na poveznicu *Logout*. Poziva se metoda `logout()` koja briše *cookie* i administrator je odjavljen.

3.5 Model

Modeli u razvojnom okviru *Play* su klase u *Javi* koje opisuju podatke koje aplikacija prikazuje i sadrže metode za upravljanje istima. Te klase su spremljene u direktoriju *app/services*. U aplikaciji Redomat korišteno je pet modela, a to su Place, User, BankData, Queue i DatabaseService. Nije korišten *ORM* alat s obzirom da nema puno tablica u bazi. Također, implementacija poslovne logike je takva da se podaci iz tablica najviše pretražuju i čitaju, a unose se parcijalno. Stoga ne bi bilo velike koristi od primjene *ORM* alata.

Place

Model predstavljen klasom `Place` omogućuje spremanje podataka o poslovnicama kao što su naziv i geografska lokacija. Ti podaci se u upravljaču dohvate iz baze, a zatim se proslijeđuju pogledu koji ih prikaže u obliku lokacija na *Google karti*.

User

Za rukovanje podacima za prijavu administratora služi model predstavljen klasom `User`. Ova klasa ima ugrađene i dodatne metode osim onih za postavljanje i dohvaćanje vrijednosti. Zbog potrebe enkripcije i dekripcije lozinke, dodana je varijabla u koju se sprema ključ enkripcije i odgovarajuće metode za kreiranje tog ključa i samu enkripciju i dekripciju. Slijedi dio koda klase `User`.

```
package services;
import ...

public class User {
 String email;
 String password;
 String salt;

 public User(){ }

 public User(String email, String password) {
  this.email = email;
  this.salt = BCrypt.gensalt();
  this.password = BCrypt.hashpw(password, salt);
 }

 public String getEmail(){ ... }

 public String getSalt(){ ... }

 public String getPassword(){ ... }

 private static Boolean comparePasswords(String hashedPassword, String plainPassword,
     String salt) {
  return hashedPassword.equals(BCrypt.hashpw(plainPassword, salt));
 }

 public static Boolean existsInDatabase(String email, String password) throws SQLException
      {
  ResultSet rs = DatabaseService.executeQuery("SELECT password, salt FROM Users WHERE
      username LIKE ?", email);
  if (rs != null && rs.next()) {
   String fetchedPass = rs.getString("password");
   String salt = rs.getString("salt");
   return comparePasswords(fetchedPass, password, salt);
  }
  return false;
 }

 public static Boolean insertToDatabase(String email, String password) throws SQLException
      { ... }
```

```
}
```

Za enkripciju lozinke korištena je biblioteka `org.mindrot.jbcrypt` i klasa `BCrypt`. Radi se o implementaciji Blowfish algoritma u svrhu enkriptiranja (*hashiranja*) lozinki. Ostale metode služe za spremanje i provjeru podataka iz baze.

Sve metode su statičke, što znači da nije potrebno instancirati objekt klase `User` kako bi ih se koristilo. Ovakav pristup ima smisla jer se u metodama upravljača pojavljuje jedino potreba za spremanjem u bazu ili provjerom postoje li neki podaci u bazi. Nema potrebe za proslijeđivanjem objekta tipa `User` ili slično, za što bi bilo potrebno kreirati instancu te klase. Prikazanom implementacijom sva funkcionalnost sadržana je u samom modelu, čime se poštuje konvencija objektno orijentiranog jezika.

BankData

Za razliku od klase `User`, klasu `BankData` je potrebno instancirati pri korištenju. Jedna instanca klase `BankData` predstavlja podatke o stanju u jednoj poslovnici. Kontroleri kreiraju i proslijeđuju objekte ove klase pogledima koji ih zatim prikažu korisniku i omogućuju njihovo uređivanje. Administratori i ostali korisnici imaju različite mogućnosti uređivanja tih podataka, kako je opisano dosada u ovom radu. Slijedi kod klase `BankData`.

```
package services;
import ...

public class BankData {
 private int id;
 private String name;
 private String address;
 private List<Queue> info;

 private BankData(){ }

 public void addBankData(int id, String name, String address){ ... }

 public void put(String service, List<Integer> newList){
  this.info.add(new Queue(service, newList));
 }

 public int getId(){ ... }
 public String getName(){ ... }
 public String getAddress(){ ... }
 public List<Queue> getInfo(){ ... }

 public static List<String> getServices() throws SQLException { ... }

  public static List<BankData> getAllBanksWithData() throws SQLException {
   List<BankData> banks = new ArrayList<BankData>();

   ResultSet rs = DatabaseService.executeQuery("SELECT id, name, address FROM Banks ORDER
       BY id");
   while (rs != null && rs.next()) {
    BankData bank = new BankData();
    bank.addBankData(rs.getInt("id"), rs.getString("name"), rs.getString("address"));
```

```
     Queue.getBankQueues(rs.getInt("id"), bank);
     banks.add(bank);
    }
    return banks;
   }

   public static BankData getBankWithData(int id) throws SQLException {
    BankData bank = new BankData();
    ResultSet rs = DatabaseService.executeQuery("SELECT name, address FROM Banks WHERE id=?
        ", id);
    if (rs != null && rs.next()) {
     bank.addBankData(id, rs.getString("name"), rs.getString("address"));
     Queue.getBankQueues(id, bank);
    }
    return bank;
   }
}
```

Pored nekoliko metoda koje služe punjenju podataka, ostale metode su opet statičke. Razlog tome je što je za dohvat svih potrebnih podataka za punjenje instance klase `BankData` potrebno kombinirati više upita na bazu. Više upita na bazu obično signalizira jednako toliko različitih metoda. S druge strane, u jednom upitu je moguće dohvatiti dio podataka za sve poslovnice odjednom, npr. nazive i adrese poslovnica. Sada je jasno da za punjenje liste poslovnica ima puno više smisla pozvati statičku metodu koja će dalje delegirati posao ostalim metodama i na kraju vratiti željenu listu.

Queue

Klasa `Queue` opisuje podatke o redovima u poslovnicama. Informacije koje je potrebno prikazati korisnicima su broj koji je trenutno u obradi te suma onih koji su na čekanju za uslugu. Te informacije su usko vezane za poslovnicu, ali i u samoj poslovnici postoje različiti redovi za različite usluge. Stoga je bilo prirodno odvojiti ove podatke metode koje ih obrađuju u posebnu klasu. Slijedi kod klase `Queue`.

```
package services;
import ...

public class Queue {
 private String service;
 private List<Integer> details;

 public Queue(String service, List<Integer> details) { ... }

 public String getService(){ ... }
 public List<Integer> getDetails(){ ... }

 public static int getServiceId(String serviceName) throws SQLException { ... }

 public static void getBankQueues(int id, BankData bank) throws SQLException { ... }

 public static int inProcess(int id, String service) throws SQLException { ... }

 public static int lastOrdinal(int id, String service) throws SQLException { ... }
```

```
 public static void update(int id, String service, int number) throws SQLException {
  int serviceId = getServiceId(service);

  if (number > 0)
  {
   DatabaseService.executeUpdate("UPDATE Transactions SET finish=? WHERE bank_id=?" +
    " AND service_id=? AND finish IS NULL AND process IS NOT NULL",
    new Timestamp(System.currentTimeMillis()), id, serviceId);
   }
  else
  {
  DatabaseService.executeUpdate("UPDATE Transactions SET process=? WHERE id IN " +
   "(SELECT id FROM Transactions WHERE bank_id=? AND service_id=? " +
   "AND finish IS NULL AND process IS NULL ORDER BY entry ASC LIMIT 1)",
   new Timestamp(System.currentTimeMillis()), id, serviceId);
  }
 }

 public static int entry(int id, String service) throws SQLException {
  int serviceId = getServiceId(service);

  int ordinal = lastOrdinal(id, service);

  ordinal += 1;
  DatabaseService.executeUpdate("INSERT INTO Transactions (bank_id,service_id,ordinal,
      entry,process,finish)" +
   " VALUES (?, ?, ?, ?, NULL, NULL)", id, serviceId, ordinal, new Timestamp(System.
       currentTimeMillis()));
  return ordinal;
 }
}
```

Metoda `getBankQueues()` puni podatke iz baze o redovima u poslovnicama, odnosno puni instance klase `Queue`. Metode `inProcess()` i `lastOrdinal()` dohvaćaju redom, broj koji je trenutno u obradi te zadnji izdani broj u poslovnici sa zadanim `id-om` i za zadanu uslugu.
Metode `entry()` i `update()` ažuriraju stanje u bazi prilikom korisničkih akcija. `entry()` kreira novi redak u tablici *Transactions* koji predstavlja novododijeljeni broj. `update()` ažurira odgovarajuće retke u istoj tablici što predstavlja završetak obrade trenutnog broja ili početak obrade idućeg.

DatabaseService

Zadnja klasa koja je spomenuta je `DatabaseService`, čije metode koriste sve gore navedene klase. Sadrži dvije statičke metode koje kreiraju konekciju na bazu i obavljaju upit koji se slaže koristeći klasu `PreparedStatement`. Jedna metoda služi za izvođenje upita s povratnom vrijednosti (`SELECT`), a druga za izvođenje ostalih upita (`INSERT, UPDATE`). Upit se sastoji od `String` varijable koja čini tijelo upita, a na mjestima vrijednosti parametara nalazi se znak *?*. Metodama koje izvršavaju upite proslijeđuje se varijabilan broj parametara. Ti parametri, zajedno s tijelom upita dalje se proslijeđuju metodi `updateStatement()`.

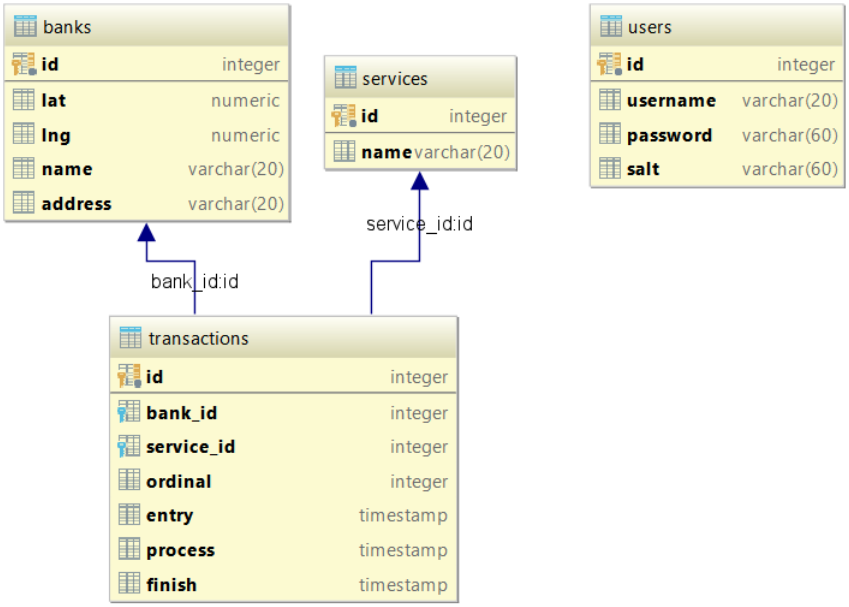
U toj se metodi slaže upit s vrijednostima parametara. Korištenjem `PreparedStatement` umjesto obične `Statement` klase, onemogućavaju se SLQ Injection[16] napadi. Slijedi kod klase `DatabaseService`.

```
package services;
import ...

public final class DatabaseService {
 public static final String CONNECTION_STRING = "jdbc:postgresql://localhost/Redomat";
 public static final String USERNAME = "postgres";
 public static final String PASSWORD = "1234";

 private static PreparedStatement updateStatement(PreparedStatement preparedStatement,
     Object... values) throws SQLException {
  for (int i = 0; i < values.length; i++) {
   preparedStatement.setObject(i + 1, values[i]);
  }
  return prepardStatement;
 }

 public static ResultSet executeQuery(String query, Object... values) throws SQLException
     {
  Connection conn = null;
  Statement stmt = null;
  try {
   conn = DriverManager.getConnection(CONNECTION_STRING, USERNAME, PASSWORD);
   stmt = conn.prepareStatement(query);
   stmt = updateStatement(stmt, values);
   ResultSet rs = stmt.executeQuery();
   conn.close();
   return rs;
  }
  catch (SQLException e) {
   System.out.println(e.toString());
  }
  finally {
   if (conn != null) {
    conn.close();
   }
  }
  return null;
 }

 public static void executeUpdate(String query, Object... values) throws SQLException {
     ... }
}
```

3.6 Baza podataka

Baza podataka sastoji se od četiri tablice, `Users`, `Banks`, `Services` i `Transactions`. Slijedi prikaz modela baze podataka. U tablici `Users` čuvaju se podaci za prijavu admi-

[16]Tehnika ubacivanja zlonamjernog SQL koda kroz obrasce za unos podataka ili putem URL-a gdje se iskorištava direktno korištenje tih podataka pri izvođenju SQL naredbi.



Slika 3.8: Model baze

nistratora. Stupac `salt` sadrži jedinstveni ključ po kojem je enkriptirana lozinka u odgovarajućem stupcu. Za potrebe enkripcije lozinki pomoću `BCrypt` klase, u aplikaciju je dodana mogućnost registracije, pa se u ovu tablicu mogu dodavati retci za vrijeme rada aplikacije.

U tablici `Services` nalazi se popis usluga koje banka nudi. U aplikaciji Redomat vrste usluge su *Transakcije*, *OsobniBankar* i *Studenti*. Sadržaj ove tablice se ne mijenja za vrijeme rada aplikacije. Ukaže li se potreba za novom uslugom ili izbacivanjem postojeće, potrebno je promjenu napraviti upravo u tablici `Services`. Ta bi izmjena bila dovoljna za cijelu aplikaciju, bez potrebe za dodatnim većim preinakama. Trebalo bi još jedino uskladiti strukturu koja sprema podatke o dodijeljenim brojevima u `local storage-u`.

Tablica `Banks` sadrži osnovne podatke o poslovnicama banke, kao što su naziv i geografska lokacija. Ti su podaci također statični i u toj se tablici ne rade izmjene tijekom rada aplikacije.

Zapisi koji predstavljaju izdane brojeve nalaze se u tablici `Transactions`. Svaki broj vezan je za konkretnu poslovnicu i uslugu, zato tablica `Transactions` ima dva strana ključa (`id` banke i `id` poslovnice). Nadalje se u tablici nalazi vrijednost samog broja u stupcu `ordinal`. Ono što određuje je li broj samo dodijeljen korisniku, trenutno u obradi ili je već obrađen, jest popunjenost zadnja tri stupca. U njima se nalaze vremenske oznake svakog od tih događaja, a ako se nisu još dogodili, vrijednost im je `NULL`. Dakle, ako je broj tek dodijeljen korisniku i čeka na obradu, u retku koji predstavlja taj broj popunjen

je samo prvi stupac, naziva `entry`. Kada broj dođe na red za obradu, vremenska oznaka se otisne u idući stupac, `process`. I na kraju, kada završi obrada korisnika s tim brojem, redak se popuni do kraja, s vremenskom oznakom u stupcu `finish`.

Glavna svrha aplikacije je dodjela rednih brojeva za neku uslugu. Određivanje koji je idući broj koji treba generirati ili koji broj administrator obrađuje, postiže se upravo pretraživanjem tih redaka. Pretražuju se retci s konkretnim `id-ovima` poslovnice i usluge. Idući broj koji treba generirati je za jedan veći od onog s najvećom vremenskom oznakom u stupcu `entry`. Idući u obradu ide onaj koji ima najmanju vremensku oznaku u stupcu `entry`, a u zadnja dva stupca vrijednost `NULL`. S obradom završava jedini redak koji ima vremenske oznake i u stupcu `entry` i u stupcu `process`, a u zadnjem vrijednost `NULL`. Na ovaj način se sprema i povijest svih transakcija, odnosno iste se ne moraju brisati na dnevnoj bazi da bi sustav funkcionirao.

Bibliografija

[1] *Design Patterns Explained Simply*. https://sourcemaking.com/design_patterns, posjećena u siječnju 2017.

[2] *Design Patterns in Java*. http://www.tutorialspoint.com/design_pattern/, posjećena u siječnju 2017.

[3] *Java™ Platform, Standard Edition 7. API Specification*. http://docs.oracle.com/javase/7/docs/api/index.html, posjećena u siječnju 2017.

[4] *Lightbend Activator*. https://www.lightbend.com/activator/download, posjećena u siječnju 2017.

[5] *Model-view-controller*. https://en.wikipedia.org/wiki/Model%E2%80%93view%E2%80%93controller, posjećena u siječnju 2017.

[6] *Play Documentation*. https://www.playframework.com/documentation/2.5.x/Home, posjećena u siječnju 2017.

[7] *Play Framework*. https://en.wikipedia.org/wiki/Play_Framework, posjećena u siječnju 2017.

[8] Eckstein, Robert: *Java SE Application Design With MVC*. http://www.oracle.com/technetwork/articles/javase/index-142890.html, posjećena u siječnju 2017.

[9] Lindholm, Tim, Frank Yellin, Gilad Bracha i Alex Buckley: *The Java Virtual Machine Specification: Java SE 7 Edition*. http://docs.oracle.com/javase/specs/jvms/se7/html/index.html, posjećena u siječnju 2017.

[10] Čupić, Marko: *Programiranje u Javi*. Skripta Fakulteta elektrotehnike i računarstva, Sveučilište u Zagrebu, rujan 2015.

Sažetak

Pri razvoju aplikacija, razlikuju se tri međusobno povezane cjeline. Jedna se brine za čuvanje podataka, druga za implementaciju poslovne logike same aplikacije, a treća prikazuje sadržaj korisniku i omogućava interakciju s njim. Prva cjelina je u većini slučajeva baza podataka, dok se druga cjelina nalazi samo u većim aplikacijama koje rade kompliciranе radnje nad podacima.

U ovom radu predstavljen je razvojni okvir *Play* koji služi za izradu web aplikacija. On omogućava oblikovanje prezentacijskog sloja aplikacije prema MVC oblikovnom obrascu. Izrada aplikacije u *Play-u* odvija se po principu konvencija prije konfiguracije. To znači da se prati obrazac koji propisuje *Play* gdje su neki elementi predefinirani. Na primjer, mapiranje ruta, stvaranje sesija, ORM alati... *Play* koristi programski jezik Java za serverski dio, Scala za generiranje *html* stranica te je omogućena jednostavna integracija Javascript i CSS elemenata. Najpoznatiji korisnici *Play-a* su LinkedIn i Coursera.

Struktura aplikacije napravljene u *Play-u* sastoji se od četiri odvojene cjeline. To su model, upravljači, pogledi i dodaci. Model se sastoji od Java klasa koje predstavljaju podatke iz baze u obliku prikladnom za prikaz korisniku. Upravljači sadrže metode čiji su rezultati različiti pogledi. Pogledi su generirane *html* stranice koje se prikazuju korisniku aplikacije. Prilikom generiranja pogleda, koriste se dodaci za stilsko uređivanje *html* elemenata i obradu događaja kao što su klik na gumb, odabir vrijednosti unutar padajućeg izbornika, itd.

Za potrebe ovog rada napravljena je aplikacija Redomat. Ona omogućava pretplatu na uslugu uzimanjem broja, te administriranje obrada tako generiranih rednih brojeva. Izrađena je u *Play-u* i pritom je korišten SBT alat za razvoj, kompajliranje i pokretanje aplikacije. Model koji je razvijen za potrebe aplikacije Redomat omogućava dodavanje ili brisanje usluga u aplikaciji bez prilagođavanja samog koda - dovoljno je dodati ili izbrisati redak u odgovarajućoj tablici u bazi. Svi pogledi se generiraju dinamički i tu glavnu ulogu ima Javascript koji iz danog modela generira strukturu za prikaz podataka. Poslovna logika sadržana je u prezentacijskom sloju. Baza podataka organizirana je tako da omogućuje spremanje podataka u obliku pogodnom za kreiranje modela koji se prikazuje korisniku.

Aplikacija Redomat može naći primjenu u svim institucijama u kojima postoji red za obradu. Primjeri su banke, policijske uprave gdje se izrađuju dokumenti, zavodi za zdrav-

stveno ili mirovinsko osiguranje, pa i same bolnice i domovi zdravlja. Na mjestima na kojima bi uslugu koristio širok krug ljudi, među kojima možda ima dosta onih koji nemaju pristup internetskoj mreži, mogli bi se dodatno koristiti aparati koji ispisuju brojeve na papir kao i dosad. U tom slučaju bi jedna aplikacija generirala sve brojeve, pa bi se onemogućili konflikti. Uz minorne promjene aplikacija Redomat bi podržavala takav način rada. Tako bi usluga postala dostupnija i informatiziranija, ali bi se korisnicima omogućilo i pretplaćivanje za uslugu na stari način.

Moj dojam o *Play-u* na kraju je vrlo dobar. Sama izrada aplikacije bila je ponešto otežana uslijed izdavanja nove verzije *Play-a*, zbog čega je dokumentacija bila djelomično nepotpuna. Moje dotadašnje iskustvo u izradi web aplikacija uključivalo je Microsoftov razvojni okvir ASP.net. I u njemu je veoma naglašen princip konvencije prije konfiguracije, te je *Play* čak i rađen na temeljima tog okvira. Zbog navedenog mi je bilo prilično lako prilagoditi se *Play-u* i shvatiti strukturu aplikacije koju on definira. S druge strane, prednost jednog takvog uhodanog razvojnog okvira pred *Play-om* je puno detaljnija i točnija dokumentacija te puno više informacija o istome na webu. Kao početniku u Javi, problem mi je stvarao i odabir biblioteka koje su kompatibilne s *Play-om*, a imaju funkcionalnost koja mi je potrebna. U tome uvelike pomaže odabir pravog IDE-a.

Summary

There are three cooperating layers when developing applications. The first one takes care of data storage, the second one implements business logic of the application itself, and the third displays the content to the user and allows him to interact with it. The first layer is usually the database, while the second layer is found in larger applications that run complex operations with data.

This thesis presents the Play framework for developing web applications. It allows designing the presentation layer that implements MVC design pattern. Development of applications in the Play framework follows the principle of convention over configuration. This means that while creating application, one follows Play prescribed pattern, using predefined elements. For example, routing, creating sessions, using ORM tools, etc. Play allows using Java programming language for backend services, Scala to generate frontend (HTML pages) and enables easy integration with JavaScript and CSS. The most known Play users are LinkedIn and Coursera.

Application developed using Play consists of four separate units. These are model, controllers, views and accessories. The model consists of Java classes that represent the data from the database in a form suitable for displaying to the user. Controllers implement methods whose results are different views. Views are generated HTML pages that are displayed to the user. Accessories consists of images, CSS and Javascript elements which are used to style the views and process events such as clicking on a button, selecting a value within drop-down menu, etc.

Application Redomat was made for this thesis. It allows one to subscribe to a service by taking a number and also administration of servicing those numbers. Development, compiling and running the application is done using SBT. The model that was developed allows one to add or remove a service in the application without rewriting the source code - it suffices to insert or delete a row in the corresponding table in database. All views are generated dynamically using Javascript which generates the model in a structure for presenting the data. The presentation layer implements business logic. The database stores data in a form suitable for creating a model that is displayed to the user.

Application Redomat can be used in all institutions where there is a queue for processing. Examples include banks, police departments where personal documents are issued,

health and pension insurance offices, and even hospitals and other health centers. At places where the service would be used by a wide range of people, among which there may be many who do not have access to the Internet, we could also use devices that print numbers on a piece of paper. In that case, one application would generate all the numbers to prevent conflicts. Application Redomat would require minor adjustments to work this way. Thus, services would become more accessible and digitalized but still enable users to subscribe for them the old way.

In the end my impression of Play is very good. The development of Redomat application using Play framework was somewhat inconvenient considering its incomplete documentation. The documentation is incomplete due to the recent release of a new version of Play. My previous experience in web development includes Microsoft's ASP.net framework. It follows the principle of convention over configuration and it inspired Play's evolvement. Therefore, it was quite easy for me to adapt to Play and understand the structure of the application which it defines. On the other hand, the advantage of a well-established development framework such as ASP.net over Play is the more detailed and accurate documentation and more information about it on web. As a Java beginner, the problem I encountered was choosing a library that is compatible with Play and that provides the functionality that was needed. Another lesson I learned was that choosing the right IDE greatly helps with development.

Životopis

Ana Marija Karlović rođena je 15.08.1990. godine u Odžaku, BiH. U osnovnu školu krenula je u Vojskovi, općina Odžak 1997. godine. Godine 2000. seli se u Zagreb. Tamo završava osnovnu školu 2005. godine. Iste godine upisuje XV. Gimnaziju u Zagrebu. Sve razrede završava s odličnim uspjehom i 2009. godine upisuje preddiplomski sveučilišni studij na Prirodoslovno-matematičkom fakultetu, Matematički odsjek. Godine 2014. završava preddiplomski studij i upisuje diplomski studij Računarstva i matematike na istom fakultetu. Tijekom studiranja radi studentske poslove, kasnije i u struci. U trenutku diplomiranja radi kao mlađi programski inženjer u jednoj IT tvrtki u Zagrebu koja se između ostalog bavi razvojem web aplikacija.