Razvoj web aplikacija pomoću okruženja Ruby on Rails

Šumelj, Mara

Master's thesis / Diplomski rad

2016

Degree Grantor / Ustanova koja je dodijelila akademski / stručni stupanj: **University of Zagreb, Faculty of Science / Sveučilište u Zagrebu, Prirodoslovno-matematički fakultet**

Permanent link / Trajna poveznica: https://urn.nsk.hr/urn:nbn:hr:217:801001

Rights / Prava: In copyright / Zaštićeno autorskim pravom.

Download date / Datum preuzimanja: **2024-03-28**



Repository / Repozitorij:

Repository of the Faculty of Science - University of Zagreb







SVEUČILIŠTE U ZAGREBU
PRIRODOSLOVNO–MATEMATIČKI FAKULTET
MATEMATIČKI ODSJEK

Mara Šumelj

RAZVOJ WEB APLIKACIJA POMOĆU OKRUŽENJA RUBY ON RAILS

Diplomski rad

Voditelj rada:
doc. dr. sc. Zvonimir Bujanović
Suvoditelj rada:
prof. dr. sc. Robert Manger

Zagreb, Studeni, 2016.

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. _____

Mojim roditeljima.

Sadržaj

Sadržaj **iv**

Uvod **1**

1 Programski jezik Ruby **2**
1.1 Povijest . . . 2
1.2 Osnovne karakteristike . . . 3
1.3 Priprema radnog okruženja . . . 4
1.4 Paketi RubyGems . . . 6
1.5 Osnove programskog jezika Ruby . . . 8
1.6 Prednosti i nedostatci programskog jezika Ruby . . . 16

2 Okruženje Rails **18**
2.1 Povijest . . . 18
2.2 Osnovne karakteristike . . . 18
2.3 Stvaranje nove aplikacije u okruženju Rails . . . 19
2.4 RESTful arhitektura . . . 23
2.5 MVC arhitektura . . . 24

3 Razvoj web aplikacije **38**
3.1 Modeli aplikacije . . . 38
3.2 Kontroleri aplikacije . . . 41
3.3 Mogućnosti korisnika . . . 43
3.4 Superkorisnik - admin . . . 46
3.5 Proširenja . . . 47

Bibliografija **49**

Uvod

U današnje vrijeme web aplikacije su dominantna tehnologija korištena na Internetu. Njihovo ubrzano širenje proizašlo je iz usluga koje omogućuju korisnicima upotrebu, razmjenu i promjenu informacija neovisno o platformi kroz infrastrukturu Interneta. Jedno od trenutno najpopularnijih okruženja za razvoj web aplikacija je Ruby on Rails. U ovom radu cilj nam je predstaviti programski jezik Ruby i okruženje za razvoj web aplikacija Rails.

Da bismo mogli koristiti Rails potrebno je poznavati barem osnove programskog jezika Ruby. Stoga ćemo se u prvom poglavlju baviti Rubyjem, proučiti njegove glavne karakteristike te ćemo usvojiti osnove koje su potrebne da bismo krenuli s razvojem web aplikacija.

U drugom poglavlju ćemo upoznati Rails kroz izradu jedne jednostavne aplikacije s ciljem da se pokaže jednostavnost i učinkovitost ovog okruženja. Na kraju rada bit će opisana složenija aplikacija koja je izrađena u sklopu ovog rada. Ona omogućuje korisnicima (studentima) da se pretplate na primanje obavijesti za kolegije koje slušaju, sami stvaraju objave na stranicama tih kolegija te dijele sadržaje poput slika i datoteka s drugim korisnicima.

Poglavlje 1

Programski jezik Ruby

1.1 Povijest

Autor Rubyja Yukihiro "Matz" Matsumoto je pažljivo odabrao i spojio dijelove svojih najdražih programskih jezika (Perl, Smalltalk, Eiffel, Ada i Lisp) i na taj način dobio jednostavan, ali i jako moćan programerski alat. Često zna naglasiti da pokušava napraviti Ruby prirodnim, a ne jednostavnim jezikom:

> *„Ruby izgleda jednostavan, ali je jako kompleksan iznutra, baš kao ljudsko tijelo."*

Još od svog nastanka 1995. godine, Ruby stječe odane programere širom svijeta. Već 2006. Ruby je dobio svjetsku popularnost. Ruby-Talk, primarna mailing lista za raspravu o Rubyju, tada je primala prosječano 200 poruka na dan. U prvoj razvojnoj fazi Ruby je već imao mnoge sada istaknute značajke: objektno-orjentirani dizajn, klase sa svojstvom nasljeđivanja, *mixins*, iteratore, *closure* funkcije, prihvat i generiranje iznimki te automatsko čišćenje resursa (eng. *garbage collection*).

Godine 2007. je javno objavljena verzija 1.9 koja je donijela nekoliko bitnih promjena: lokalne blokovske varijable (varijable koje su lokalne u bloku u kojem su deklarirane), lambda izrazi, nove API-utičnice (podrška za IPv6), novo osiguranje sigurnosti pomoću ključne riječi *require_relative*. Rubyju popularnost sve više raste, a najveće zasluge mogu se pripisati okruženju Ruby on Rails (2005. godine). Verzija 2.0 objavljena je 2013. Najbitnije nove značajke iz te verzije su: funkcijski argumenti, nova metoda za proširenje klasa (*Module#prepend*), novi literal za kreiranje niza simbola, novi API za lijenu evaluaciju generičkog tipa *Enumerable* i nova konvencija korištenja *#to_h* za konvertiranje objekata u*Hashes*. Posljednja stabilna verzija 2.3 objavljena je u prosincu 2015. U najnovijoj verziji poboljšane su mnoge performanse,

popravljeni razni bugovi, ali i implementirane primjetne promjene, poput sigurnosnog operatora .&.

Implementacija na koju se odnosi ovaj rad se često zove i MRI (Matz's Ruby interpreter) ili CRuby jer je napisana u programskom jeziku C. Međutim, postoji nekoliko različitih implementacija. Korisne su u određenim situacijama, omogućuju integraciju s drugim jezicima ili okolinom, ili imaju neka specijalna svojstva koje MRI ne podržava. Najpoznatije implementacije su:

- JRuby predstavlja implementaciju jezika Ruby za Javin virtualni stroj (eng. *JVM - Java Virtual Machine*).
- Rubinius je implementacija Rubyja koja se temelji na načelima jezika Smalltalk.
- MacRuby je implementacija Rubyja u programskom jeziku Objective-C namjenjena za korištenje u okruženju CoreFoundation koje služi za razvoj web aplikacija na operacijskom sustavu Apple OS X.
- IronRuby je implementacija Rubyja u Microsoftovoj .NET platformi.
- MagLev je brza, stabilna 64-bitna implementacija Rubyja otvorenog kôda u VMware-ovoj GemStone/S 3.1 virtualnoj mašini.

Kao što vidimo, različite implementacije namjenjene su prilagodbi Rubyja raznim platformama i operacijskim sustavima.

1.2 Osnovne karakteristike

Ruby je dinamični programski jezik otvorenog kôda (eng. *open source*) s naglaskom na jednostavnost i produktivnost. To je objektno-orjentirani programski jezik čija jednostavna sintaksa i fleksibilnost omogućuje programerima pisanje kôda točno onako kako oni sami žele. Da je to i bio cilj potvrđuju riječi njegovog autora:

> *„Želim da su korisnici Rubyja slobodni. Želim im dati slobodu odabira. Ako postoji najbolji put među svim mogućim alternativama, želim potaknuti odabir tog puta."*

Kod mnogih objektno-orjentiranih programskih jezika brojevi i drugi primitivni tipovi podataka nisu objekti. Ruby se tu ističe jer su svim tipovima podataka pridružene instancirane varijable i metode, pa možemo reći da je sve u Rubyju objekt. Ruby je poznat kao jako fleksibilan jezik. Programerima je omogućeno da uklone ili redefiniraju osnovne dijelove Rubyja. Na primjer, zbrajanje je omogućeno pomoću operatora +, ali ako radije želimo koristiti riječ „plus" možemo promijeniti ugrađenu

klasu *Numeric*. Napomenimo još da je Ruby skriptni jezik što znači da se programer udaljava od detalja izvedbe programa na ciljnoj platformi, tj. koristi gotove komponente koje se brinu o tim detaljima.

1.3 Priprema radnog okruženja

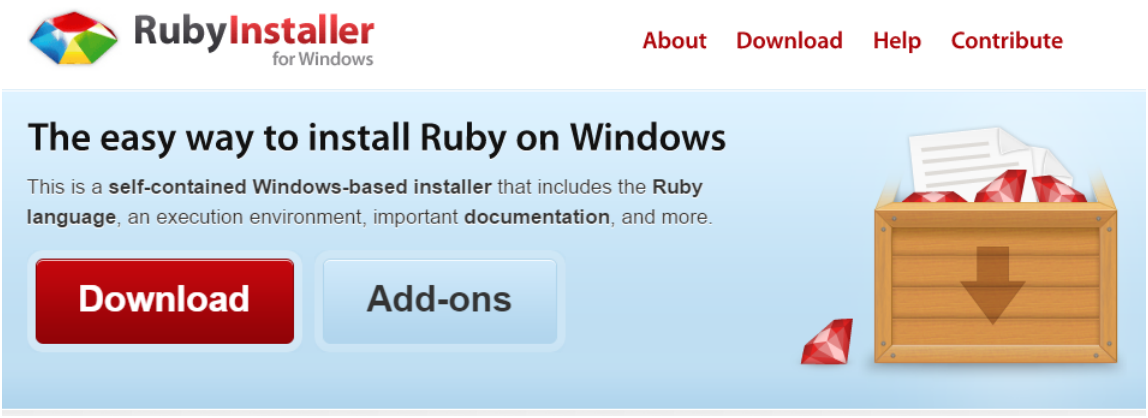
Kako bismo mogli raditi s programskim jezikom Ruby potrebno je napraviti radno okruženje koje se sastoji od interpretera i dokumentacije. Potrebno je imati i editor u kojem ćemo pisati kôd. Okruženje za razvoj programa u Rubyju sastoji se od tri komponente:

- interpreter,
- sustav za upravljanje proširenjima,
- interaktivna Ruby ljuska (irb).

Posljednje dvije komponente nisu neophodne za rad, ali su jako korisne. Interaktivna ljuska omogućuje jednostavno izvršavanje kôda bez da ga prethodno napišemo u neku datoteku. Ovisno o operacijskom sustavu kojeg koristimo postoje različiti načini instalacije programskog jezika Ruby.

Instalacija

Standardna implementacija programskog jezika Ruby može se instalirati i izvršavati na različitim operacijskim sustavima. Najjednostavniji način instalacije na operacijskom sustavu Windows je preko programa RubyInstaller tako da direktno preuzmemo potrebne pakete preko službene web stranice.



Slika 1.1: Izgled službene stranice RubyInstaller-a

Ovim načinom instaliramo interpreter za programski jezik Ruby, osnovne pakete RubyGems i osnovna proširenja te dokumentaciju.

Prilikom izrade ovoga rada Ruby je instaliran sa standardnog Ubuntu repozitorija. Prvo instaliramo sve potrebne dodatke pomoću naredbi:

```
sudo apt-get update
sudo apt-get install git-core curl zlib1g-dev build-
   essential libssl-dev libreadline-dev libyaml-dev
   libsqlite3-dev sqlite3 libxml2-dev libxslt1-dev
   libcurl4-openssl-dev python-software-properties libffi-
   dev
```

Zatim koristeći rbenv instaliramo Ruby:

```
cd
git clone https://github.com/rbenv/rbenv.git ~/.rbenv
echo 'export PATH="$HOME/.rbenv/bin:$PATH"' >> ~/.bashrc
echo 'eval "$(rbenv init -)"' >> ~/.bashrc
exec $SHELL

git clone https://github.com/rbenv/ruby-build.git ~/.rbenv
   /plugins/ruby-build
echo 'export PATH="$HOME/.rbenv/plugins/ruby-build/bin:
   $PATH"' >> ~/.bashrc
exec $SHELL

rbenv install 2.3.1
```

```
rbenv global 2.3.1
ruby -v
```

Potrebno je još instalirati Bundler pomoću naredbe:

```
gem install bundler
```

Nakon instalacije još je potrebno izvršiti naredbu:

```
rbenv rehash
```

Programska podrška za izradu aplikacija u Rubyju je spremna.

Pokretanje kôda

Klasični način pokretanja Ruby kôda je kreiranje datoteke s ekstenzijom *.rb* te pokretanje te datoteke pomoću naredbe *ruby*:

```
$ ruby pozdrav.rb
Hello world!
```

Možemo isprobavati Ruby naredbe pomoću interaktivne ljuske koja se pokreće naredbom *irb*:

```
$ irb
>>puts "Hello world!"
Hello world!
```

Imamo još mogućnost pokretanja kôda u komandnoj liniji tako da interpreter pozivamo s opcijom -e:

```
$ ruby -e "puts 'Hello world!'"
ruby -e "puts 'Hello world'"
Hello world
```

1.4 Paketi RubyGems

RubyGems služi za upravljanje proširenjima jezika Ruby. Alat koji je dizajniran da bi se na što jednostavniji način skidale i instalirale ekstenzije dio je standardne biblioteke Rubyja od verzije 1.9. RubyGems pokrećemo naredbom *gem*. Za instaliranje nekog proširenja koristimo naredbu *gem install*. Ona će nam preuzeti i instalirati odabrano proširenje i sve zavisne dodatke te generirati potrebnu dokumentaciju. Da bismo instalirali okružje Rails dovoljno je izvršiti sljedeću naredbu:

```
gem install rails -v 4.2.6
```

Svaki gem ima svoje ime, verziju i platformu. Verziju Railsa zadali smo prilikom instalacije, a u dokumentaciji možemo pronaći da mu je platforma Ruby, što znači da je kompatibilan sa svim platformama na kojima se Ruby pokreće. Proširenje se sastoji od slijedećih komponenti: kôd, dokumentacija i *gemspec*. Sva proširenja prate standardnu strukturu organizacije kôda.

```
coderay-1.1.0
├── bin
│   └── coderay
├── lib
│   ├── coderay
│   └── coderay.rb
├── Rakefile
├── README_INDEX.rdoc
└── test
    └── functional
```

Slika 1.2: Struktura gema

Na slici iznad vidljive su glavne komponente gema:

- Lib direktorij koji sadrži kôd gema.
- Test ili spec direktorij koji sadrži testove.
- Gem sadrži datoteku Rakefile koju program *rake* koristi za automatiziranje testova, generiranje kôda i sl.
- Ovaj gem sadrži datoteku u bin direktoriju koja će se nalaziti u varijabli okruženja $\$PATH$ nakon instaliranja gema.
- README_INDEX datoteka sadrži potrebnu dokumentaciju.

1.5 Osnove programskog jezika Ruby

Varijable

Varijabla je memorijska lokacija čiji se sadržaj može mijenjati tokom izvođenja programa. Ruby podržava različite tipove varijabli. Lokalne varijable imaju doseg bloka, metode, klase ili funkcije u kojoj su definirane. Ime lokalne varijable počinje znakom _. Globalne varijable su varijable definirane izvan klase, odnosno modula i imaju globalni doseg. Imena tih varijabli počinju znakom $. Pseudo varijable su varijable koje izgledaju kao lokalne varijable, a ponašaju se kao konstante. Ne možemo im pridružiti vrijednost. To su sljedeće varijable: $self$, $true$, $false$, nil, $_FILE_$, $_LINE_$. Još imamo klasne varijable čije ime počinje znakovima @@ i instancirane varijable koje počinju znakom @. Klasne varijable su korisne kada želimo zabilježiti neki podatak koji se odnosi na cijelu klasu. Stoga je doseg klasnih varijabli cijela klasa za razliku od instanciranih varijabli čiji je doseg objekt kojem pripadaju. Klasa ima značenje kao i u drugim objektno-orjentiranim programskim jezicima te ćemo malo više o njoj vidjeti kasnije. Napomenimo još da konstante mogu imati i lokalni i globalni doseg, a njihova imena počinju velikim slovom. Varijable ćemo najbolje vidjeti kroz primjer:

```
@globalna_varijabla = 1
class Class1
   @@klasna=nil
   def metoda1
       @instancirana = 2
       puts "Vrijednost globalne varijable: #
   $globalna_varijabla. Pristupiti mozemo i  varijablama
   instancirana i klasna."
   end
end

class Class2
   Konstanta = 3
   def metoda2
      puts "Konstanta ima doseg klase Class2"
   end
end
```

U Rubyju možemo pristupiti vrijednosti bilo koje varijable ili konstante tako da napišemo znak # ispred imena te varijable ili konstante. Tako smo u primjeru iznad pristupili vrijednosti globalne varijable.

Operatori

Baš kao i svi moderni programski jezici, Ruby podržava veći broj operatora, odnosno vrsta operatora. Sve o aritmetičkim operatorima, operatorima usporedbe, logičkim operatorima, operatorima pridruživanja, operatorima ranga, trojnom i *defined?* operatoru u Rubyju može se pronaći u službenoj dokumentaciji. Istaknimo samo *defined?* operator. To je poseban operator u Rubyju koji određuje da li je predani izraz definiran ili ne. Ako je izraz definiran ispisuje opis izraza, inače ispisuje *nil*.

```
defined? Class2
```

Izvršavanjem gornjeg kôda dobivamo rezultat:

```
"constant"
```

Dok izvršavanjem sljedećeg kôda dobijemo rezutlat *nil*.

```
defined? program
```

Numerički tipovi podataka

Već smo spomenuli da se brojevi promatraju kao objekti u Rubyju. To znači da svaki numerički tip podatka pripada nekoj od numeričkih klasa. Ruby ima mnogo ugrađenih numeričkih klasa, a u nastavku ćemo navesti samo neke najvažnije. Cijeli brojevi u Rubyju su sadržani u klasama *Fixnum* i *Bignum*. Kojoj klasi pripada neki cijeli broj ovisi o njegovoj veličini. Klasa *Numeric* je nadklasa klasama *Fixnum* i *Bignum* kao i svim ostalim numeričkim klasama. Zatim, imamo i klasu *Float* koja sadrži realne brojeve dvostruke preciznosti, te klasu *Rational* koja sadrži racionalne brojeve.

```
123                    # Fixnum
-500                   # Negativni Fixnum
12345678901234567890 # Bignum
123.4                  # Float
Rational(3,4)          # Rational : (3/4)
```

Da su brojevi stvarno objekti koji mogu imati metode vidimo u sljedećem primjeru:

```
2.times{puts "Sve je objekt!"}
```

Pokretanjem gornjeg kôda dobili smo izlaz:

```
Sve je objekt!
Sve je objekt!
2
```

Operatori nad numeričkim tipovima su isti kao i u ostalim programskim jezicima, uz malu razliku: inkrementalni i dekrementalni operatori (++ i −−) nisu definirani, a modularno dijeljenje je drugačije definirano nego u C-u. Pokrenimo naredbe:

```
puts (5 % -3)
puts (-5 % 3)
```

Prva naredba daje izlaz -1, a druga 1.

Kontrola toka

Navest ćemo strukture za kontrolu redoslijeda izvršavanja naredbi, tzv. naredbe za kontrolu toka.

If/else je selektivna struktura koja nam omogućuje izvršavanje jednog bloka naredbi ukoliko je uvjet ispunjen, odnosno izvršavanje drugog bloka naredbi ukoliko uvjet nije ispunjen. Pokretanjem sljedećeg koda dobit ćemo rezultat: *0 is true*. Izvršavanjem ovakvih naredbi u npr. C-u dobili bi rezultat: *0 is false*.

```
if 0 then
    puts "0 is true"
else
    puts "0 is false"
end
```

Kada imamo više mogućih kriterija, strukturu *if/else* je bolje zamijeniti strukturom *case*.

```
case @followers
    when 0 .. 20
       puts "not popular"
    when 21 .. 50
       puts "ok"
    when 51 .. 100
       puts "popular"
    else
       puts "super popular"
end
```

Ako varijabli $@followers$ pridružimo vrijednost 100, ispisat će se: *popular*.

While je upravljačka struktura zadužena za ponavljano izvođenje naredbe ili grupe naredbi. Izvršava blok dok je zadani uvjet istinit. Rad petlje možemo prekidati na četiri različita načina:

- break - u potpunosti izlazimo iz petlje,

- next - skačemo na iduću iteraciju,
- redo - ponovno pokrećemo trenutnu iteraciju,
- return - izlazimo iz petlje, ali i iz metode u kojoj se petlja nalazi.

```
while $i < $granica  do
    puts("U petlji je broj #$i" )
    $i +=1
    break if i==2
end
```

Osim *while*, **for** je najčešća ključna riječ koja se pojavljuje kod provođenja iteracija.

```
for current_iteration_number in 1..3 do
    puts "Trenutni broj je #current_iteration_number"
end
```

Pokretanjem gornjeg koda dobijemo sljedeći ispis:

```
Trenutni broj je 1
Trenutni broj je 2
Trenutni broj je 3
=> 1..3
```

Funkcije i metode

Metode imaju barem jednu klasu kojoj pripadaju. Funkcije ne moraju imati klasu kojoj pripadaju i njima se možemo služiti bilo gdje u programu. Fleksibilnost Rubyja je očita kod metoda i funkcija: ne trebamo deklarirati tip ulaznih parametara kao ni tip povratnog parametra. Ključna riječ *return* je opcionalna. Bez nje ćemo kao povratnu vrijednost dobiti zadnju izvršenu liniju koda. Definirajmo metodu *plus* u klasi *Numeric*:

```
class Numeric
    def plus(x)
        self.+(x)
    end
end
```

Sada umjesto $2 + 2$ možemo koristiti $2.plus\ 2$.

Imena metoda počinju malim slovom. Mogu završavati s znakovima '?' i '!'. Kada koristimo metode i funkcije s parametrima bitno je kod svakog poziva poslati

točan broj parametara. Stoga je uvijek dobro zadati početne vrijednosti ulaznih parametara kao u sljedećem primjeru:

```
def test(p="Ruby",p1="Smalltalk", p2="Perl", p3 ="Eiffel",
    p4="Ada", p5="Lisp")
    puts "Programski jezik je #{p}!"
 end
```

Ispis nakon pozivanja funkcije bez parametara: *Programski jezik je Ruby!*. Parametri funkcija u Rubyju se uvijek šalju po vrijednosti.

Klasa String

Klasa *String* je jedna od najbitnijih podatkovnih struktura za web aplikacije, obzirom da se naposlijetku web stranice sastoje od nizova znakova poslanih sa servera na preglednik. String literali kreiraju se korištenjem dvostrukih navodnika.

```
"string literal"
```

Konkatenaciju stringova možemo ostvariti pomoću operatora +:

```
"Ruby" + " on" + " Rails"
```

Još jedan način na koji možemo spojiti stringove je umetanjem (interpolacijom) pomoću specijalih znakova #{}. Time ćemo stvoriti novi string, a varijable koje se koriste u interpolaciji ostaju nepromijenjene.

```
first_="Ruby"
second_="on"
third_="Rails"
"#{first_} #{second_} #{third_}"
```

Za ispis stringova najviše se koristi Ruby funkcija *puts*. Funkcija *puts* ispisuje string na ekran te vraća vrijednost *nil*. Slična je i metoda *print*:

```
print "Ruby on Rails"
```

Ruby još podržava i stringove pod jednostrukim navodnim znakovima. Efektivno isti kao i stringovi s dvostrukim navodnim znakovima uz jednu bitnu razliku: Ruby ne podržava interpolaciju stringova s jednostrukim navodnim znakovima.

Stringovi imaju razne ugrađene metode, ali možemo i sami implementirati nove metode ako želimo. U sljedećem primjeru ćemo definirati metodu *prazan_string* koja ispisuje da li je string prazan ili ne.

```
class String
    def prazan_string()
```

```
        if this.empty?
            "String je prazan!"
        else
            "String nije prazan!"
         end
    end
end

puts "Ruby".prazan_string()
```

Izvršavanjem gornjeg kôda dobivamo ispis: *String nije prazan!*

Nizovi

Niz je skup podataka poredanih nekim redoslijedom. Kao i u većini programskih jezika i u Rubyju su nizovi ‚zero-offset', odnosno prvi element niza ima indeks 0. Nizovi se zapisuju u uglatim zagradama, a elementi niza su odvojeni zarezima. Svaki član niza može biti proizvoljnog tipa.

```
niz = ["prvi element je niz znakova", "treci element je
   broj", 1]
```

Pristupati članovima niza možemo koristeći uglate zagrade. Za pristup članovima niza koji se najviše koriste postoje posebne ugrađene funkcije.

```
niz = ["prvi", "drugi", 3, 4, "zadnji"]
```

Na primjer, za gore definirani niz možemo pozvati metode *first*, *last* i *count* koje vraćaju redom prvi i zadnji element niza, te broj elemenata niza.

```
niz.first
niz.last
niz.count
```

Postoje mnoge ugrađene metode za nizove koje možete detaljnije pogledati u službenoj dokumentaciji.

Hash-tablice

Hash-tablica je tip podataka sličan nizu s razlikom da za indeks možemo koristiti bilo koji tip podataka, a ne samo cijeli broj.

```
polje = {}
polje["programski jezik"] = "Ruby"
polje["okruzenje"] = "Rails"
```

U prethodnom primjeru koristili smo indekse tipa string. U Rails okruženju se najviše koriste simboli kao ključ polja. Simbol, čije ime počinje znakom : je specijalni tip podataka sličan stringu, a najčešće predstavlja neki objekt.

```
stranica = {:name => "Matematicka analiza", :description
   => "Obavezni kolegij 1. godine"}
```

Vrijednosti polja mogu biti bilo što, čak i druga polja:

```
polje[:stranica] = {:name => "Matematicka analiza", :
   description => "Obavezni kolegij 1. godine"}
```

Pristupiti atributu : *name* možemo na sljedeći način:

```
polje[:stranica][:name]
```

Hash-tablice koje za indeks imaju druge hash-tablice, kao u primjeru iznad, zovu se „ugniježđene" hash-tablice (eng. *nested hash tables*), te se često koriste u okružju Rails.

Blokovi

Zaokruženi kôd koji možemo promatrati kao cjelinu i koji se nalazi između ključnih riječi *do* i *end* zovemo blok. U drugim programskim jezicima metode i procedure pozivamo s jednim ili više argumenata koji mogu biti neki tip podataka tog jezika. Kod Rubyja možemo kao parametar zadati ne samo podatke nego i sam programski kôd. Definiramo metodu na sljedeći način:

```
def funkcija
    puts "argument funkcije je blok"
    yield
    yield
end
```

Posebnost ove metode je u tome što sadrži ključnu riječ *yield* koja izvršava blok koji se prenosi u metodu kao argument. Pozvat ćemo gore definiranu metodu navodeći kao argument blok koji se sastoji od jedne naredbe (u slučaju bloka koji se sastoji od samo jedne naredbe nije potrebno pisati ključne riječi):

```
funkcija {puts "Pozdrav iz bloka!" }
```

Rezultat izvršavanja:

```
argument funkcije je blok
Pozdrav iz bloka!
Pozdrav iz bloka!
```

Dakle, blokovi se ne tretiraju na isti način kao drugi argumenti koji se prenose funkcijama. Argumenti koji nisu blokovi evaluiraju se prije poziva funkcije, dok se blokovi evaluiraju tek prilikom poziva metode *yield* u tijelu funkcije. Obzirom da blokovi mogu primati argumente svaki put kada ih izvršavamo, najčešće se koriste za implementaciju iteratora. Prisjetimo se, iterator je posebna metoda koja se izvršava na svakom elementu spremnika podataka na kojem je definiran. Blokovi zajedno s iteratorima predstavljaju još jednu posebnost Rubyja.

```
[2, 4, 8, 10].each do |b|
    puts b*2
end
```

Pokretanje gornjeg kôda rezultirat će sljedećim ispisom:

```
4
8
16
20
```

U gornjem primjeru smo definirali niz od 4 elementa te smo zatim na svaki element pozvali metodu *each* koja za svaki član niza izvršava zadani blok pridajući mu pritom vrijednost trenutnog člana niza kao parametar.

Klase

Svaka klasa u Rubyju predstavlja novi tip podatka u programu. Instancu klase nazivamo objekt. Program se konstruira kao niz interakcija među objektima. Klasama implementiramo korisničke tipove podataka koji modeliraju objekte iz aplikacijske domene. Klase su osnove objektno-orjentiranog programiranja i kao takve su najvažniji tip podataka u Rubyju. Klase definiramo pomoću specijalne riječi *class*.

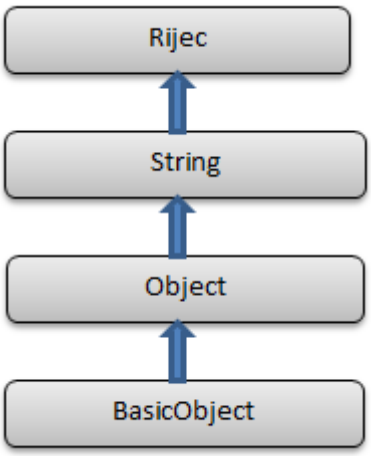
```
class Rijec < String
    def palindrom?
       self == self.reverse
    end
end
```

U prethodnom primjeru definirali smo klasu *Rijec* koja ima metodu *palindrom*?. Oznakom $Rijec < String$ smo definirali da naša nova klasa nasljeđuje već ugrađenu klasu *String*. Time smo naslijedili sve metode definirane u klasi *String*.

Kada učimo koristiti klase korisno je znati hijerarhiju naslijeđivanja. Da bismo saznali te podatke možemo koristi metodu *superclass*. Pogledajmo hijerarhiju naslijeđivanja za gore definiranu klasu *Rijec*.

```
>> str.class
=> Rijec
>> str.class.superclass
=> String
>> str.class.superclass.superclass
=> Object
>> str.class.superclass.superclass.superclass
=> BasicObject
>> str.class.superclass.superclass.superclass.superclass
=> nil
```

Iz prethodnog primjera se vidi da klasa *Rijec* nasljeđuje klasu *String* koja pak nasljeđuje klasu *Object*. I na kraju kao nadklasu klase *Object* imamo klasu *BasicObject* koja nema svoju nadklasu. Iz ovog proizlazi „sve u Rubyju je objekt". Dijagram hijerarhije nasljeđivanja vidi se na sljedećoj slici.



Slika 1.3: Hijerarhija nasljeđivanja

1.6 Prednosti i nedostatci programskog jezika Ruby

Velika prednost Rubyja je njegova prirodna sintaksa. Dok većina programskih jezika strogo definira točan pristup i način rješavanja određenih problema, Ruby podržava izražajnost i suptilnost. Većina programskih jezika limitira programere i

onemogućava korištenje svih aspekata kôda na više načina, a Ruby zbog svog dinamičnog stila pisanja pruža tu opciju. Ruby se ističe svojom prirodnom sintaksom, te već nakon prvog čitanja možemo zaključiti što koji dio kôda izvršava. Još jedna istaknuta prednost je ta što za gotovo sve postoji proširenje. Svi dodatci i proširenja su javno dostupni, te ih je jednostavno instalirati. Ruby je puno važnosti predao testiranju i automatizaciji testova. Ova činjenica pomaže da se isporuči softver visoke kvalitete.

Glavna prednost Rubyja, njegova prirodna sintaksa i dinamičnost imaju i svoju cijenu. Ruby, kao skriptni jezik, je u usporedbi s Javom i C-om dosta sporiji. Iako je dostupno mnogo implementiranih proširenja u obliku gemova, ponekad je teško pronaći dokumentaciju za njihovo korištenje. Pogotovo je teško pronaći dokumentaciju za manje popularne gemove.

Poglavlje 2

Okruženje Rails

2.1 Povijest

Autor okruženja Ruby on Rails je David Heinemeier Hansson. Na osnovu njegovog rada na popularnom menadžerskom alatu Basecamp, godine 2004. je nastalo jedno od najpopularnijih okruženja za razvoj web aplikacija, Ruby on Rails. Tada je bio dostupan kao otvoreni kôd (eng. *open source*), a u veljači 2005. bila je dostupna prva službena verzija Railsa. Prve verzije su koristile Mongrel web server, iako je bilo moguće pokretanje i na drugim tada dostupnim web serverima.

Verzija 1.2 javno je objavljena u siječnju 2007. Najvažnije promjene u toj verziji bile su usklađivanje s REST arhitekturom i uvažavanje nove HTTP generacije. Također je prilagođena instalacija za većinu dostupnih servera. Svjetsku pozornost Ruby on Rails je dobio u listopadu 2010. kada ga je Apple odlučio ukomponirati u svoj Mac OS X operacijski sustav Leopard. Iste godine Rails se spojio sa svojim konkurentom Merbom. Verzije Railsa nakon 3.0 su stoga unaprijeđene Merbovim modularnim dizajnom, stabilnim API-jem i poboljšanim performansama.

Glavni razlog sve veće popularnosti Railsa je osobina da se brzo prilagođava novim tehnikama i alatima web tehnologija. Tijekom godina Ruby on Rails je omogućio početnicima da se na vrlo lak i brz način uklope u razvoj dinamičnih web stranica. S druge strane, također je omogućio i razvoj velikih i komplesnih aplikacija kao što su Twitter, GitHub, Airbnb, Shopify, Yellow Pages.

2.2 Osnovne karakteristike

Rails je okruženje za razvoj web aplikacija napisano u Rubyju. Dizajnirano je da bi učinilo razvoj web aplikacija što jednostavnijim pretpostavljajući što svaki programer

treba da bi krenio s razvojem. Ostavljajući jako puno izbora u stilu programiranja Rails brzo stječe svjetsku popularnost.

Filozofija programiranja u okruženju Rails vođena je s dva glavna principa:

- Ne ponavljaj se (eng. *Don't repeat yourself, DRY*): Princip prema kojem svaki dio kôda mora biti jedinstven, jednoznačan i reprezentativan unatar cijelog sistema. Softver je proširiv i modularan s minimalnim potrebama za održavanjem, a kôd je održiviji i lakše proširiv ako se ne ponavljamo.
- Konvencija ispred konfiguracije (eng. *Convention over Configuration, CoC*): ako pratimo zadane konvencije možemo napraviti mnogo koraka u web aplikaciji bez velikih podešavanja konfiguracijskih datoteka. Točnije, ovaj princip omogućuje programeru da ne troši vrijeme opisujući sva ponašanja, već je potrebno opisati samo ona koja odstupaju od poznatih normi, odnosno konvencija.

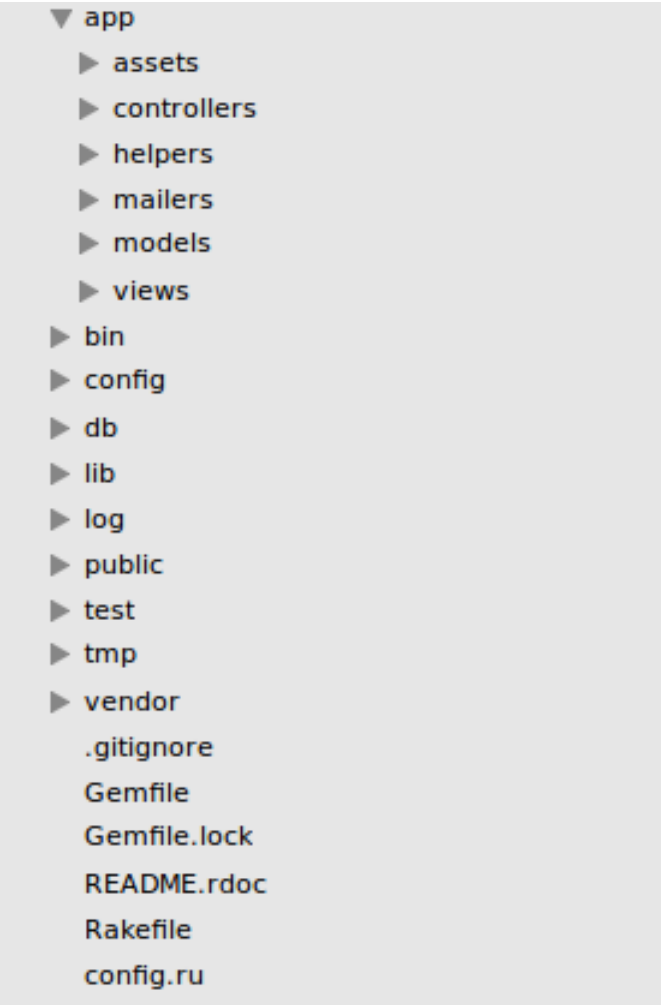
Poštujući te principe, izrada web aplikacija je brza, kvalitetna, jednostavna i učinkovita.

2.3 Stvaranje nove aplikacije u okruženju Rails

U prethodnom poglavlju prikazano je kako instalirati Rails nakon što već imamo instaliran Ruby. Sada možemo kreirati osnovnu aplikaciju.

```
$ rails new osnovna_aplikacija
```

Prethodna naredba je dovoljna da nam kreira sve potrebne datoteke i instalira sve potrebne gemove za osnovnu aplikaciju. Sada imamo stvoren direktorij *osnovna_aplikacija* čija je struktura vidljiva na sljedećoj slici.



Slika 2.1: Struktura direktorija Rails aplikacije

Vidimo da Rails ima jedinstvenu strukturu direktorija. Time povećava čitljivost i organizaciju kôda. Pogledajmo najvažnije dijelove strukture:

- app - Organizira komponente aplikacije.
- app/controllers - Poddirektorij kontrolera gdje Rails sprema klase kontrolera.
- app/helpers - Poddirektorij koji sadrži sve pomoćne klase koje se koriste za nadopunu klasa modela, pogleda i kontrolera. Pomoćne klase omogućuju da je kôd modela, pogleda i kontrolera manji, uredniji i fokusiran na bitnije dijelove.

- app/models - Poddirektorij u kojem se nalaze klase koje modeliraju podatke spremljene u bazu aplikacije.
- app/view - Poddirektorij u kojem se nalaze predlošci koji se pretvoreni u HTML prikazuju u web pregledniku.
- config - Direktorij koji sadrži malu količinu konfiguracijskog kôda koja je potrebna za funkcioniranje aplikacije. Tu spada konfiguracija baze, struktura Rails okoline i preusmjeravanje (eng. *routing*) nadolazećih web zahtjeva.
- db - U većini slučajeva Rails aplikacije imaju modele koji pristupaju relacijskim bazama podataka. Skripte za prilagodbu relacijske baze, ukoliko su potrebne, dodajemo u ovaj direktorij.
- lib - Direktorij za biblioteke.
- log - Direktorij za .log datoteke.
- public - Direktorij koji sadrži web datoteke koje se ne mijenjaju, kao što su npr. slike i JavaScript datoteke.
- test - Direktorij koji sadrži sve automatske i ručno dodane testove.
- tmp - Rails koristi ovaj direktorij za sve privremene datoteke tijekom izvođenja aplikacije.
- vendors - Sve datoteke koje su nisu stvorene od strane Railsa (sigurnosne biblioteke i sl.).
- Gemfile - Datoteka koja sadrži popis svih gemove koji su potrebni za izvedbu aplikacije.
- README - Datoteka koja sadrži osnovne detalje o Rails aplikaciji i opis strukture direktorija.
- Rakefile - Datoteka koja pomaže kod buildanja i testiranja kôda.

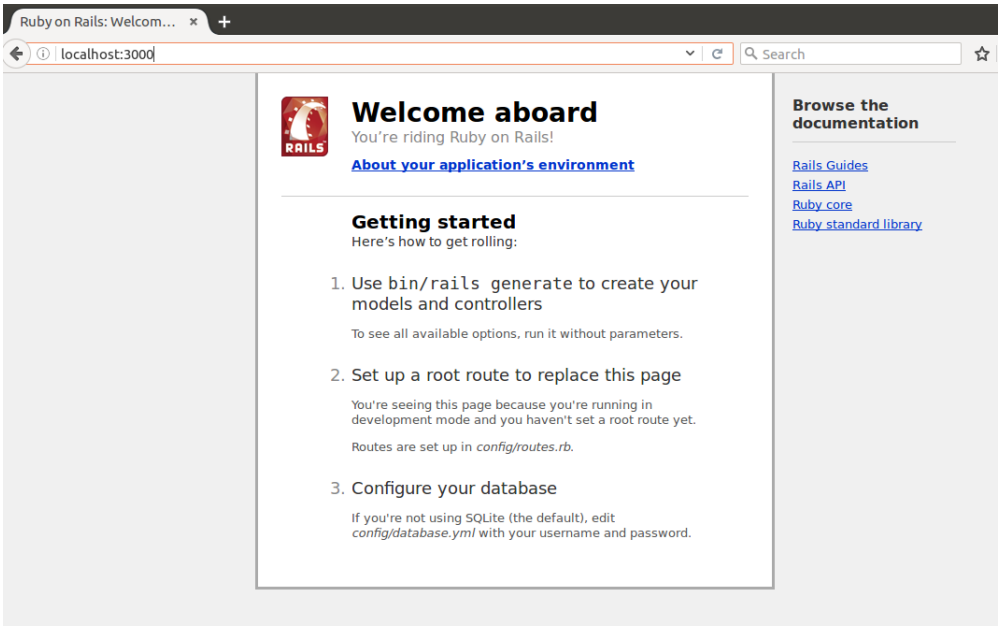
Naša aplikacija je postavljena za početni rad. Potrebno je još pokrenuti Rails server. Pozicionirani u glavni direktorij aplikacije izvršavamo naredbu:

```
$ rails server
```

Dobijemo sljedeći ispis:

```
=> Booting WEBrick
=> Rails 4.2.6 application starting in development on http
   ://localhost:3000
=> Run ‘rails server -h‘ for more startup options
=> Ctrl-C to shutdown server
```

Adresa na kojoj se nalazi aplikacija ispisana je na ekran nakon pokretanja servera: *http://localhost:3000*. U web pregledniku na toj adresi vidimo prikaz kao na Slici 2.2. Pozivom naredbe *rails new* automatski je generirana ova vrlo jednostavna, ali funkcionalna web aplikacija. Njezinim modifikacijama moći ćemo implementirati željene mogućnosti web aplikacije, ma kako složene one bile.



Slika 2.2: Početna stranica aplikacije

Instalacija aplikacijskih ovisnosti

RubyGems, koji smo objasnili u prethodnom poglavlju, nudi mnogo proširenja i dodataka, te nam omogućuje da lako instaliramo module treće strane. Jednostavno praćenje i instaliranje ovisnosti koje su potrebne za funkcioniranje aplikacije omogućava nam pomoćni alat Bundler. Prilikom stvaranja aplikacije definira se datoteka Gemfile s osnovnim ovisnostima. Ona sadrži sve RubyGems pakete koje aplikacija koristi. Nakon promjene te datoteke potrebno je pokrenuti naredbu:

```
bundle install
```

Time dohvaćamo i instaliramo sve nove pakete koji nam nedostaju. Naredba *rails new* sama pokreće instalaciju paketa definiranih u automatski stvorenoj *Gemfile* datoteci.

2.4 RESTful arhitektura

REST (eng. *representational state transfer*) je stil programerske arhitekture koji se koristi u razvoju distribuiranih, mrežnih sustava i web aplikacija. Njime je definiran način komunikacije između klijenta i poslužitelja pri korištenju mrežnih resursa pomoću HTTP protokola. Komunikacija se odvija tako da klijent pošalje zahtjev poslužitelju, poslužitelj zatim taj zahtjev obradi te na osnovu zatraženog resursa formira odgovor i šalje ga klijentu. Resurs može biti bilo što (dokument, slika, numerički podatak i sl.) i ima jedinstvenu adresu lokacije upotrebom univerzalne sintakse koja se koristi u hipermedijskim poveznicama (eng. *URL - universal resource locator*). Web servisi koji koriste REST arhitekturu zovu se RESTful web servisi. Stoga, možemo sažeti ukratko principe i ograničenja na kojima se temelje RESTful web servisi:

- Svaki resurs ima jedinstvenu adresu lokacije (URL).
- Stanja aplikacije i funkcionalnosti su reprezentirani resursima.
- Svi resursi dijele uniformno sučelje za prijenos stanja između klijenta i resursa pomoću dobro definiranih operacija i tipova podataka.
- Resursi čine mrežu resursa, sadržavajući poveznice na druge resurse.
- HATEOAS (eng. *hypermedia as the engine of applicaton state*) je ograničenje REST arhitekture gdje aplikacija dinamički daje informacije klijentima o REST sučelju kroz hipermediju koristeći linkove koje dodaje u odgovore.
- Resursi se mogu pohraniti u priručnu memoriju, informacije o roku valjanosti kopija se moraju poštovati.
- Protokol komunikacije je klijentsko-poslužiteljski, interakcija je bez pamćenja stanja, mogući su posrednici.
- Kôd na zahtjev (eng. *code on demand*) – poslužitelji mogu privremeno proširiti funkcionalnost klijenta.

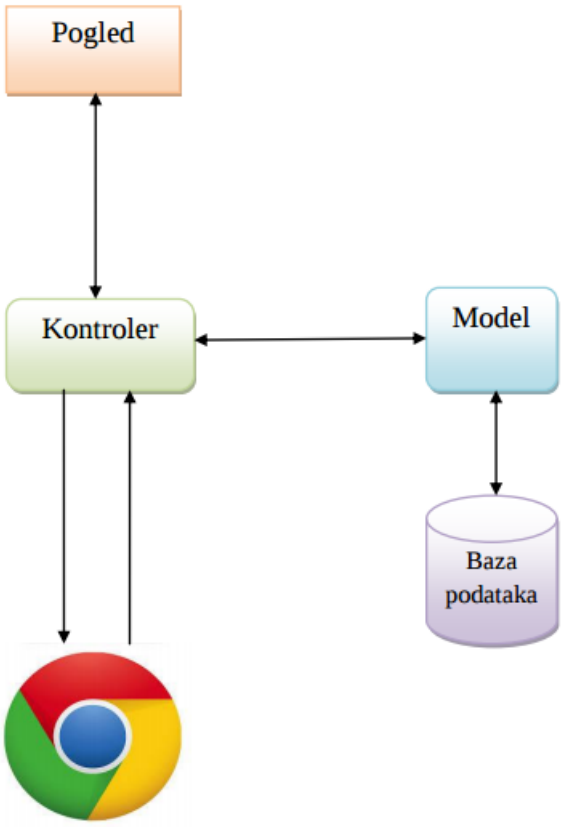
RESTful web servisi koriste uglavnom HTTP protokol za prijenos i URL mehanizam za adresiranje. Dakle, resursi se jednoznačno određuju i identificiraju s *Uniform Resourse Identifier* (URI), a za sve manipulacije s resursima koristimo HTTP metode. Da bismo dohvatili neki resurs koristimo GET, za stvaranje koristimo PUT, za ažuriranje koristimo POST, te za brisanje koristimo DELETE.

RESTful stil uklopljen u razvoj web aplikacije pomoću okruženja Rails znatno olakšava implementaciju i odabir resursa. Aplikaciju stvaramo koristeći resurse koji mogu biti kreirani, dohvaćeni, ažurirani i izbrisani. Kada pristigne HTTP zahtjev, Rails analizira zatraženi URI te ga obradi pomoću datoteke *routes.rb* koja se nalazi u direktoriju $config$. U njoj se nalaze sve potrebne informacije za pravilno usmjeravanje. U nastavku ćemo vidjeti primjer.

2.5 MVC arhitektura

Model-View-Controller (MVC) arhitektura je obrazac softverske arhitekture koji se u softverskom inženjerstvu koristi za odvajanje pojedinih dijelova aplikacije u komponente ovisno o njihovoj namjeni. MVC arhitektura sastoji se od tri komponente. Svaka komponenta je zadužena za obavljanje specifičnih funkcija koje se mogu promatrati kao cjelina. Razdvajanje pojednih dijelova aplikacije u komponente puno je bolje nego kada se sve nalazilo na jednom mjestu. S ovakvom strukturom kôda lakše je pronalaziti i ispravljati pogreške, a i sama funkcionalnost i fleksibilnost aplikacije je bolja. MVC arhitektura potiče odvajanje logike aplikacije od njene prezentacije i time dobiva tri međusobno zavisne komponente: model (eng. *model*), pogled (eng. *view*) i kontroler (eng. *controller*).

Na Slici 2.3 prikazan je princip rada MVC arhitekture u web okruženju. U komunikaciji web preglednika s Rails aplikacijom, web preglednik šalje zahtjev serveru koji ga potom prosljeđuje Rails kontroleru. Rails kontroler obrađuje primljeni zahtjev. U nekim slučajevima kontroler odmah generira novi pogled, koji je u biti predložak koji se konvertira u HTML i šalje natrag web pregledniku. Mnogo češći slučaj je da kontroler mora stupiti u interakciju s modelom. Model je Ruby objekt čija je uloga rad s podacima, odnosno komunikacija s bazom podataka. Nakon pozivanja modela, kontroler generira novi pogled i web pregledniku šalje cijelu web stranicu u HTML-u.



Slika 2.3: Shematski prikaz principa rada MVC arhitekture

Kako bismo pokazali jednostavnost izrade aplikacije u okruženju Ruby on Rails, aplikaciju *osnovna_aplikacija* nadopunit ćemo modelom, kontrolerom i pogledima. Aplikacija će biti vrlo jednostavna. Omogućit će stvaranje, prikazivanje i brisanje jednostavnih korisnika s imenom i email adresom preko web preglednika. Kroz nju ćemo bolje upoznati komponente MVC arhitekture te način njihove međusobne komunikacije.

Model

Model je komponenta MVC arhitekture odgovorna za komunikaciju s bazom podataka. Za model još kažemo da predstavlja poslovnu logiku aplikacije. Klase modela mogu biti kreirane ručno ili generirane iz baze podataka. Novi model *User* u *osnovna_aplikacija* stvorit ćemo tako da, pozicionirani u glavnom direktoriju, izvršimo naredbu:

```
rails generate model User name:string email:string
```

Jedan od rezultata ove naredbe je datoteka koja se zove *migracija*. Migracije omogućuju postepeno mijenjanje baze podataka, tako da se model može prilagođavati promijenjenim zahtjevima. U ovom slučaju migracija stvara tablicu *users* s dva stupca *name* i *email*, kao što možemo vidjeti u datoteci *db/migrate/[timestamp]_create_users.rb*:

```
1  class CreateUsers < ActiveRecord::Migration
2    def change
3      create_table :users do |t|
4        t.string :name
5        t.string :email
6
7        t.timestamps null: false
8      end
9    end
10 end
```

Migracija sadržava metodu *change* koja određuje kakva promjena će se dogoditi u bazi. U ovom slučaju metoda *change* koristi Rails metodu *create_table* da bi kreirala tablicu za spremanje korisnika u bazi. Metoda *create_table* stvara tablicu sa stupcima *name* i *email* koje smo definirali u naredbi te pomoću naredbe *t.timestamps* dodaje još dva „magična" stupca *created_at* i *updated_at* koja predstavljaju vremenske oznake kada je korisnik stvoren i uređivan. Da bi se promjene ostvarile u bazi potrebno je pokrenuti migraciju sljedećom naredbom:

```
bundle exec rake db:migrate
```

Drugi rezultat naredbe *rails generate model* je stvaranje samog modela. Kôd modela *User* koji se nalazi u datoteci *app/models/user.rb* izgleda vrlo jednostavno:

```
1  class User < ActiveRecord::Base
2  end
```

Vidimo da klasa *User* nasljeđuje klasu *ActiveRecord::Base*. Dakle, model *User* automatski ima sve metode klase *ActiveRecord::Base*. Koje sve funkcionalnosti ima naš model znat ćemo kada se malo bolje upoznamo s klasom *ActiveRecord*.

Klasa ActiveRecord

Za klasu *ActiveRecord* možemo reći da predstavlja model u MVC arhitekturi. Ona olakšava stvaranje i korištenje objekata čiji podaci zahtjevaju trajnu pohranu u bazu podataka. U klasi *ActiveRecord* objekt predstavlja i podatke i operacije nad tim podacima. Vodi se logikom da će na taj način korisnici lakše usvojiti kako čitati iz baze i pisati u nju. *ActiveRecord* je objektno-relacijsko mapiranje (ORM) za jezik Ruby.

ORM (eng. *object rational mapping*) je tehnika programiranja za pretvaranje podataka između nekompatibilnih tipova podataka u objektno-orjentiranim programskim jezicima. To zapravo stvara „virtualnu objektnu bazu podataka" koja se može koristiti unutar programskog jezika. Koristeći svojstva ORM-a veze objekata u aplikaciji mogu se spremati u bazu podataka i dohvaćati iz nje bez upotrebe SQL upita.

Stoga, *ActiveRecord* ima mnogo svojstava, od kojih su najvažniji:

- Predstavlja modele i njihove podatke.
- Predstavlja povezanost između tih modela.
- Predstavlja hijerarhiju nasljeđivanja preko povezanih modela.
- Provjerava podatke prije nego ih sprema u bazu.
- Na objektno-orjentirani način izvodi operacije nad bazom.

Inače, okruženja za ORM zahtjevaju puno konfiguracijskog kôda pri izradi aplikacija. Međutim, ako slijedimo konvencije Railsa (CoC), pri stvaranju *ActiveRecord* modela ne trebamo se puno brinuti o konfiguraciji. Ideja koja stoji iz ove konvencije je da ako je konfiguracija za naše aplikacije gotovo uvijek ista onda bi to trebao biti već ugrađeni standard.

Kod klase *ActiveRecord* bitno je poznavati konvencije imenovanja. Jednu od istih već smo mogli zamijetiti kod kreiranja modela *User*. Naš model zove se *User*, a tablica u bazi podataka se zove $users$. *ActiveRecord* još koristi konvenciju imenovanja za stupce tablica u bazi, ovisno o njihovoj svrsi u tablici.

- Strani ključ - imena stranih ključeva u tablicama trebaju slijediti pravilo : ImeTaliceUjednini_id (npr. *page_id*).
- Primarni ključ - Active Record automatski stvara i koristi stupac *id* kao primarni ključ bilo koje naše tablice.

Nadalje, *ActiveRecord* automatski kreira metode koje omogućuju aplikaciji čitanje podataka iz baze te manipulaciju s njima. Te metode su tzv. CRUD metode: stvaranje

(eng. *create*), čitanje (eng. *read*), ažuriranje (eng. *update*) i brisanje (eng. *delete*). Za stvaranje objekta imamo dvije metode: *new* i *create*. Metoda *new* stvara objekt, dok metoda *create* vraća objekt i sprema ga u bazu.

Obzirom da zasad imamo samo model, pristupiti mu možemo jedino preko *rails console*. Možemo vidjeti kako izgleda poziv funkcije *new* nad objektom *User*.

```
User.new
```

Rezultat naredbe:

```
<User id: nil, name: nil, email: nil, created_at: nil,
   updated_at: nil>
```

Ispod možemo vidjeti poziv funkcije *create* nad objektom *User*:

```
User.create(:name => "Example", :email => "example@example
   .com")
```

Rezultat naredbe:

```
   (0.3ms)  begin transaction
  SQL (0.6ms)  INSERT INTO "users" ("name", "email", "
   created_at", "updated_at") VALUES (?, ?, ?, ?)  [["name
   ", "Example"], ["email", "example@example.com"], ["
   created_at", "2016-09-09 09:36:01.173169"], ["
   updated_at", "2016-09-09 09:36:01.173169"]]
   (87.8ms)  commit transaction
```

Vidimo da je pozvana SQL naredba INSERT INTO te nam „commit transaction" potvrđuje da je podatak uspješno spremljen u bazu.

Za pristup podacima u bazi, *ActiveRecord* ima na raspolaganju mnogo metoda. Ispod ćemo navesti neke primjere korištenja tih metoda. Da bismo dohvatili sve objekte modela *User* u varijablu *users* koristimo naredbu:

```
users = User.all
```

Da bismo dohvatili objekt modela *User* kojem je $id = 1$ u varijablu *User* koristimo naredbu:

```
user = User.first
```

Da bismo dohvatili objekt modela *User* kojem je $name =$ "*Example*" u varijablu *user* koristimo naredbu:

```
user = User.find_by(name: 'Example')
```

Nakon što smo dohvatili neki objekt, možemo ga mijenjati i promjene spremati u bazu vrlo lako. Objekt modela *User* kojem je $name =$ "*Example*" ažuriramo s novim imenom sljedećom naredbom:

```
user = User.find_by(name: 'Example')
user.update(name: 'Example1')
```

Nad objektima još možemo koristiti metodu *destroy* te ih time izbrisati iz baze. Objekt modela *User* kojem je $name = "Example1"$ brišemo sljedećom naredbom:

```
user = User.find_by(name: 'Example1')
user.destroy
```

Još jedan vrlo moćan mehanizam koji nudi ova klasa je validacija objekta modela prije spremanja u bazu. Validacija podataka je vrlo bitna kod trajnog zapisivanja podataka u bazu, stoga metode *save* i *update* vraćaju $false$ i ne vrše nikakve daljnje radnje nad bazom ako validacija ne uspije. Na primjer, ako želimo da nam polje *name* ne bude prazno, ali da nema više od 50 znakova, a da polje *email* ima jedinstvenu valjanu email adresu s najviše 255 znakova to možemo ostvariti tako da datoteku *user.rb* nadopunimo naredbama:

```
1  before_save { self.email = email.downcase }
2  validates :name, presence: true, length: { maximum: 50 }
3  VALID_EMAIL_REGEX = /\A[\w+\-.]+@[a-z\d\-.]+\.[a-z]+\z/i
4  validates :email, presence: true, length: { maximum: 255 },
5  format: { with: VALID_EMAIL_REGEX },
6  uniqueness: { case_sensitive: false }
```

Sada više neće biti moguće u bazu upisati podatke koji ne zadovoljavaju kriterije koje smo naveli.

Kontroler

Kontroler upravlja dolaznim zahtjevom, daje mu smisao i stvara konačni izlaz kao odgovor na primljeni zahtjev. Iako zvuči kao najsloženija komponenta MVC arhitekture, postoje razne konvencije koje olakšavaju rad s kontrolerom. Kontroler možemo gledati i kao posrednika između modela i pogleda: pogledu daje podatke modela tako da ih pogled može prikazati korisniku, a podatke koje korisnik pošalje prosljeđuje modelu.

Kontroler je Ruby klasa koja nasljeđuje ugrađenu klasu *ApplicationController* te kao i svaka druga klasa ima svoje metode. Kada aplikacija primi zahtjev, preusmjeravanjem (eng. *routing*) se određuje koji kontroler i koju akciju treba pokrenuti. Rails stvara instancu tog kontrolera te pokreće metodu koja ima isti naziv kao akcija. Na primjer, ako korisnik pristupi stranici *users/new*, Rails će stvoriti instancu *UsersController* i pozvati metodu *new*.

Vratimo se na našu aplikaciju. Imamo model *User* i htjeli bismo da svaki objekt tog modela možemo stvoriti, prikazati i izbrisati preko web preglednika kao što smo

to do sada radili preko *rails console*. Da bismo to ostvarili potrebni su nam pogledi i kontroler. Kontroler *Users* ćemo stvoriti naredbom:

```
rails generate controller Users new show index
```

Ova naredba nam je stvorila datoteku *users_controller.rb* : u direktoriju *controllers* koja izgleda ovako:

```
class UsersController < ApplicationController
  def new
  end

  def show
  end

  def index
  end

end
```

Vidimo da *users_controller.rb* definira klasu *UsersController* koja ima metode *new*, *show* i *index*. Ako sada pogledamo u direktorij *views* vidjet ćemo da imamo novi direktorij *users* u kojem se nalaze tri nove datoteke. Svaka metoda koju imamo u kontroleru ima odgovarajući pogled. Na primjer, metoda *new* ima odgovarajući pogled *new.html.erb*. U nastavku ćemo više pisati o pogledu. Obzirom da želimo spremiti u bazu novi objekt koji kreiramo i da želimo imati opciju da objekt izbrišemo iz baze, trebat će nam još dvije metode u kontroleru: *create* i *destroy*. Njih nismo generirali kao ostale jer nam neće trebati posebni pogledi za njih.

Da bi kontroler mogao komunicirati s pogledom na ispravan način potrebno je prilagoditi datoteku *routes.rb* koja se nalazi u *config* direktoriju. Trenutni izgled *routes.rb* koji se automatski promijenio nakon stvaranja kontrolera:

```
Rails.application.routes.draw do
  get 'users/new'

  get 'users/show'

  get 'users/index'

end
```

Obzirom da mi ne želimo samo pristupati podacima, već želimo omogućiti stvaranje i brisanje korisnika u skladu s HTTP metodama prilagođavamo *routes.rb*:

```
Rails.application.routes.draw do

  root              'users#index'
  get 'new'      =>  'users#new'
```

```
5   post 'new'     =>  'users#create'
6
7   resources :users
8 end
```

Usmjeravanje pomoću resursa (eng. *resource routing*) ostvareno naredbom *resources:users* omogućava deklariranje svih putanja za dani kontroler. U samo jednoj liniji kôda deklarirali smo putanje za *index*, *show*, *new*, *edit*, *create*, *update* i *destroy* akcije. Da bi nam kôd bio pregledniji i URL adresa jednostavnija dodane su još tri definicije putanja. Na primjer, *root* postavlja kao početnu stranicu *users#index* na istom URL-u kao i prije, te nam omogućuje da u pogledu umjesto *users#index* pišemo *root_path*.

Da bismo nadopunili metode u kontroleru potrebni su nam mehanizmi za slanje i primanje parametara među komponentama. Postoje dva načina prenošenja parametara u web aplikacijama. Prvi način je da se parametri prenose preko Uniform Resource Locator (URL). Takve parametre zovemo niz string parametara. Parametrima smatramo sve što se nalazi iza znaka ? u URL-u. Drugi način se odnosi na podatke koji se prenose preko metode POST. Ove parametre obično dohvaćamo preko HTML forme koju korisnik popunjava. Takve parametre još zovemo i POST podaci jer se mogu poslati samo kao dio HTTP POST zahtjeva. Kod interpretacije parametara u Railsu nema nikakve razlike, te su oba načina dohvaćanja dostupna.

Podatke za stvaranje novog korisnika ćemo dohvaćati preko HTML forme, pa će metoda *create* koristiti POST podatke preko metode *user_params* koja će biti objašnjena u nastavku.

```
1 @user = User.new(user_params)
```

S druge strane, metoda *show* koristit će niz string parametara.

```
1 @user = User.find(params[:id])
```

U svakom kontroleru imamo metodu odgovora i metodu zahtjeva koje predstavljaju HTTP zahtjev koji se trenutno odvija. Metoda zahtjeva pokazuje na objekt zahtjeva, a metoda odgovora vraća objekt odgovora koji prikazuje što će biti poslano natrag korisniku.

Objekt zahtjeva sadrži mnogo korisnih informacija o zahtjevu kojeg je poslao korisnik. Putem ovog objekta možemo pristupiti sljedećim svojstvima:

Svojstvo zahtjeva	Namjena zahtjeva
host	Ime računala (eng. *hostname*) korišteno pri slanju zahtjeva.
domain(n=2)	Prvih n dijelova imena računala.
format	Vrsta sadržaja kojoj korisnik želi pristupiti.
method	HTTP metoda zahtjeva.
get?, post?, patch?, put?, delete?, head?	Vraća true ako je HTTP metoda GET, POST, PATCH, PUT, DELETE ili HEAD.
headers	Zaglavlja korištena u zahtjevu.
port	Port korišten pri slanju zahtjeva.
protocol	String koji sadrži ime korištenog protokola plus "://", npr. "http://".
query_string	Niz string parametara.
remote_ip	IP adresa korisnika.
url	Cijeli URL zahtjeva.

Rails skuplja sve navedene parametre koji se šalju zajedno sa zahtjevom u polju parametara.

Objekt odgovora obično se ne koristi direktno, već se koristi prilikom izvršavanja potrebne akcije kontrolera i vraćanja podataka korisniku. Međutim, dobro je znati paremetre odgovora jer ih je ponekad potrebno ručno mijenjati.

Svojstvo zahtjeva	Namjena zahtjeva
body	String podataka koji se šalju korisniku pošiljatelju zahtjeva (najčešće HTML).
status	Kôd HTTP status odgovora (npr. 404 - tražena web stranica nije pronađena na serveru.).
location	URL na koji će korisnik biti preusmjeren.
content_type	Tip sadržaja koji se šalje natrag.
charset	Raspored znakova (zadano: "utf-8").
headers	Zaglavlja korištena u odgovoru.

Kada znamo kako prenositi parametre između pogleda i kontrolera možemo nadopuniti kontroler. Prisjetimo se, želimo da korisnik unese podatke preko web preglednika. Mi ćemo te podatke spremiti u bazu te na zahtjev korisnika prikazati podatke samo jednog objekta ili svih. Također želimo iz baze izbrisati objekt kada korisnik to zatraži. Koristeći metode klase *ActiveRecord* nadopunimo metode kontrolera na sljedeći način:

```
class UsersController < ApplicationController
   def new
     @user = User.new
  end

  def show
    @user = User.find(params[:id])
  end

  def index
    @users = User.all
  end

  def create
    @user = User.new(user_params)
      if @user.save
        redirect_to @user
      else
        redirect_to new_path
      end
  end

  def destroy
    User.find(params[:id]).destroy
    redirect_to root_path
  end

  private
  def user_params
      params.require(:user).permit(:name, :email)
  end

end
```

Definirali smo jake parametre (eng. *strong parameters*) preko pomoćne metode *user_params* da bismo onemogućili korisnicima promjenu atributa koje ne bi smjeli mijenjati. Iz definicije proizlazi da su nam potrebni svi parametri (eng. *require*), a da nam je preko kontrolera dopušteno (eng. *permit*) prenijeti samo *name* i *email*. Time smo zaštitili atribut *id* od bilo kakvih promjena izvan modela. Koristili smo

još ugrađenu metodu *redirect_to* da bi nakon izvršene akcije prikazali stranice čija je putanja dana kao parametar. U nastavku ćemo definirati poglede za definirane akcije kontrolera.

Pogled

Pogled je dio MVC arhitekture namjenjen za prikaz stranica. Pogled u Rails okruženju je ERB predložak (eng. *embeded Ruby*) koji dijeli podatke s kontrolerom kroz varijable koje su im oboma dostupne. Ekstenzija datoteka koje definiraju pogled je *.html.erb*, odnosno html u koji je pomoću posebnih tagova uključen Ruby. Da bismo koristili sve mogućnosti programskog jezika Ruby unutar HTML-a omogućeno nam je korištenjem ovih oznaka:

- <% %>- izvršava Ruby kôd
- <%= %>- izvršava Ruby kôd i prikazuje rezultat

Općenito, Rails obrađuje zahtjeve pomoću klasa *ActionController* i *ActionView*. *ActionController* je zadužen za komunikaciju s bazom preko modela i izvođenje tzv. CRUD akcija, a *ActionView* je zadužen za spajanje odgovora u cjelinu.

Action View

Akcije kontrolera u Rails aplikaciji rezultiraju osvježavanjem postojećeg pogleda ili prikazivanjem novog. Za svaki kontroler unutar aplikacije postoji poddirektorij unutar direktorija *app/views* u kojem su sadržane sve datoteke predložaka koje služe za generiranje pogleda za taj kontroler. Za imenovanje pogleda u Railsu postoji konvencija imenovanja: pogledi dijele ime s akcijom kontrolera koju predstavljaju. Na primjer, ako imamo datoteku *index.html.erb* u *app/views/users* tada *users_controller.rb* treba imati akciju *index*.

Kompletan HTML koji se prikazuje klijentu sastoji se od tri Rails elementa:

- Predložak (eng. *template*),
- Dio predloška (eng. *template partial*),
- Format (eng. *layout*).

Prisjetimo se, kod generiranja kontrolera generirani su i svi potrebni predlošci za našu aplikaciju. Ako bismo htjeli dodati neki novi predložak potrebno je stvoriti datoteku u odgovarajućem direktoriju, nadopuniti kontroler metodom koja dijeli ime s datotekom predloška, te naposlijetku postaviti preusmjeravanje za novi predložak. Sada smo spremni za popunjavanje predloška *index.html.erb*.

```
<h1>Svi korisnici</h1>

<table>
  <thead>
    <tr>
      <th>Ime</th>
      <th>Email</th>
      <th colspan="3"></th>
    </tr>
  </thead>

  <tbody>
    <% @users.each do |user| %>
      <tr>
        <td><%= user.name %></td>
        <td><%= user.email %></td>
        <td><%= link_to 'Prikazi', user %></td>
        <td><%= link_to "Izbrisi", user, method: :delete,
        data: { confirm: "Jeste sigurni?" } %></td>
      </tr>
    <% end %>
  </tbody>
</table>

<br>
<%= link_to 'Novi korisnik', new_path %>
```

Dio predloška je izdvojeni dio koda iz predloška u novu datoteku te se na taj način može koristiti u više predložaka. Time poštujemo Railsov princip „Ne ponavljaj se". Da bismo demonstrirali upotrebu dijela predloška napravit ćemo dio predloška *_forma* koji će služiti za unos podataka prilikom stvaranja novog korisnika. Za skupljanje informacija od korisnika koristit ćemo Rails formu *form_for*. Forma za argument prima objekt klase *ActiveRecord* i konstruira formu koristeći atribute tog objekta. Konvencije imenovanja omogućuju kontroleru da primi podatke preko parametara. Potrebno je unutar direktorija *views/users* stvoriti novu datoteku s imenom *_form.html.erb* te ju popuniti sljedećim kôdom:

```
<%= form_for(@user) do |f| %>
    <div class="field">
    <%= f.label :name %><br>
    <%= f.text_field :name %>
  </div>
  <div class="field">
    <%= f.label :email %><br>
    <%= f.text_field :email %>
  </div>
  <div class="actions">
```

```
11     <%= f.submit %>
12   </div>
13 <% end %>
```

Pomoću naredbe *render* dio predloška uključujemo u predložak. Predložak *new* sad izgleda:

```
1 <h1>Novi korisnik</h1>
2
3 <%= render 'forma' %>
4
5 <%= link_to 'Natrag', root_path %>
```

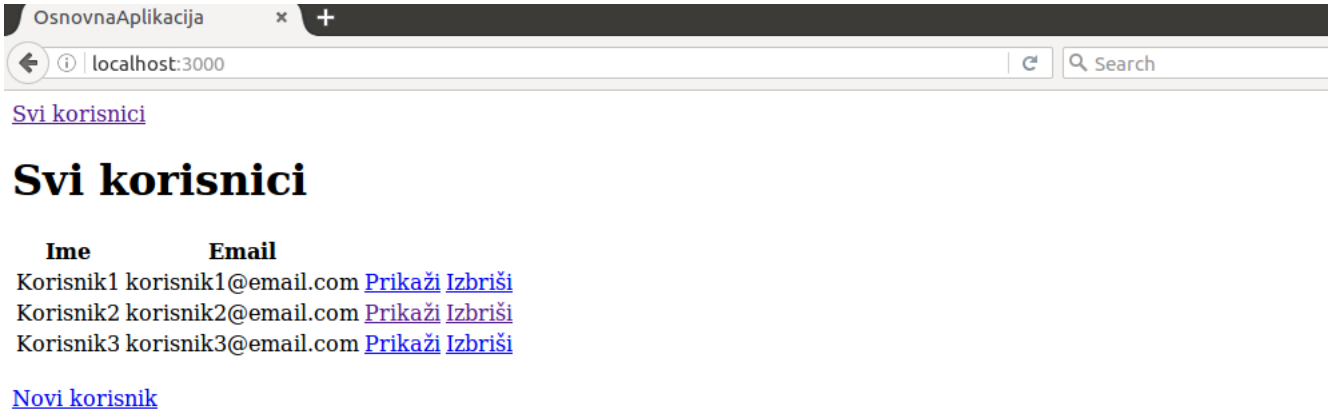
Da bismo upotpunili aplikaciju još trebamo definirati datoteku *show.html.erb* na sljedeći način:

```
1 <p>
2   <strong>Name:</strong>
3   <%= @user.name %>
4 </p>
5
6 <p>
7   <strong>Email:</strong>
8   <%= @user.email %>
9 </p>
```

Da bi pogled bio čitav još nam nedostaje format. Format je predložak koji upotpunjuje pogled tako da daje isti izgled određenim dijelovima sadržaja unutar stranica koje su logički povezane. Prilikom generiranja pogleda Rails prvo potraži datoteku s baznim imenom kontrolera u *app/views/layouts*. Ako takva datoteka ne postoji onda koristi datoteku *application.html.erb*. Ako želimo da nam svaka stranica koja se generira u web pregledniku ima neke zajedničke elemente onda te elemente definiramo u datoteci *application.html.erb*. Na primjer, ako želimo da nam osnovna_aplikacija na svakoj stranici u gornjem lijevom kutu ima poveznicu prema stranici *index* dovoljno je u *application.html.erb* dodati:

```
1 <header>
2   <%= link_to "Svi korisnici", root_path %>
3 </header>
```

Sada možemo vidjeti na slici ispod kako izgleda početna stranica naše aplikacije:



Slika 2.4: Početna stranica aplikacije osnovna_aplikacija

Dobili smo funkcionalnu web aplikaciju koja može stvarati, prikazivati i brisati objekte modela. Time smo pokrili osnove izrade web aplikacija u okruženju Rails. Prateći konvencije Railsa vidimo da se ovakva funkcionalnost može ostvariti na vrlo jednostavan način. U sljedećem poglavlju ćemo vidjeti kako izgleda aplikacija koja ima više modela.

Poglavlje 3

Razvoj web aplikacije

U ovom poglavlju ćemo vidjeti mogućnosti Railsa kroz složeniju web aplikaciju koja je dio ovog diplomskog rada. Kroz aplikaciju ćemo demonstrirati već predstavljene mogućnosti, kao i mnogobrojne koje nisu spomenute. Izvorni kôd je dostupan na CD-u priloženom uz diplomski rad.

Opis aplikacije

Aplikacija "Funda" omogućuje korisnicima (studentima) pristup informacijama o kolegijima koje slušaju te pristup datotekama i slikama koje postave drugi korisnici. Svaki kolegij ima na svojoj stranici gumb *Pretplatite se*. Klikom na taj gumb korisnik počinje pratiti kolegij. Na korisnikovoj naslovnici pojavljuju se novosti svih kolegija koje prati. Na stranicama kolegija korisnici mogu postavljati objave, datoteke i slike.

3.1 Modeli aplikacije

Modeli aplikacije:

- *user*,
- *page*,
- *micropost*,
- *relationship*.

Model *user* opisuje korisnike koji su registrirani na Fundi. Model *page* opisuje kolegije koji su stvoreni i mogu se „pratiti". Model *micropost* opisuje objave koje su korisnici

stvorili na kolegijima koje prate. I naposljetku, model *relationship* opisuje veze koje povezuju korisnike s kolegijima koje prate.

Napomenimo samo da je za ovu aplikaciju korištena mysql baza podataka. Zadana baza podataka u Railsu je SQLite. Međutim, kada instaliramo potrebne dodatke, vrlo lako je koristiti i druge baze podataka. Da bismo postavili mysql kao bazu podataka dovoljno je prilikom stvaranja projekta napraviti sljedeće:

```
rails new funda -d mysql
```

Potrebno je još samo unutar datoteke *config/database.yml* podesiti podatke za root korisnika.

Modeli *user* i *page* dodani su na isti način kao i model *user* iz prethodnog poglavlja. Obzirom da su modeli *micropost* i *relationship* povezani s modelima *user* i *page* vezama jedan na prema mnogo, osim osnovnih atributa moramo im dodati i odgovarajuće strane ključeve. Koristeći *references*, Rails automatski dodaje još jedan stupac u tablici, indeks i strani ključ u skladu s konvencijom imenovanja. Uz uvjet da već imamo stvorene modele *user* i *page*, model *micropost* stvaramo sljedećom naredbom:

```
1 rails generate model Micropost content:text user:references pages:
    references
```

Ova naredba nam još u datoteku *micropost.rb* upiše sljedeće:

```
1 belongs_to :user
2 belongs_to :page
```

Dakle, definira nam vezu u jednom smjeru, tj. da *micropost* pripada modelima *page* i *user*. Da bismo definirali i drugi smjer veze dodamo u *page.rb* i *user.rb* sljedeće:

```
1 has_many :microposts
```

Da bi model *relationship* bio reprezentativniji, sve veze modela namještene su ručno. Model je stvoren s atributima *follower_id* i *followed_id*. Indeksi, strani ključevi i odnosi među modelima *relationship*, *user* i *page* dodani su ručno pomoću istih ključnih riječi kao gore.

Sada nam je model *user* povezan s modelom *relationship* stranim ključem *follower_id*, a *page* s *followed_id*. Da bi Rails to povezao moramo umetnuti sljedeći kôd u model *relationship*:

```
1 belongs_to :follower, class_name: "User"
2 belongs_to :followed, class_name: "Page"
```

Pogledajmo sada kôd modela *page.rb* te način na koji smo definirali povezanost tog modela s modelom *relationship*.

```
1 class Page < ActiveRecord::Base
2   has_many :microposts, dependent: :destroy
```

```
3
4
5   has_many :passive_relationships, class_name: "Relationship",
6   foreign_key: "followed_id", dependent: :destroy
7   has_many :followers, through: :passive_relationships, source: :
      follower
8 end
```

Passive_relationships i *active_relationships* je još jedna konvencija imenovanja Railsa kojom nam je omogućeno da preko modela *relationship* definiramo da korisnik prati kolegije. Dakle, hoćemo reći da se objekt modela *relationship* stvara isključivo preko modela *user*. Možemo još primjetiti da svaki kolegij može imati više objava i više pratitelja. Kada izbrišemo kolegij sve objave i sve veze koje imaju odgovarajući strani ključ se također brišu.

Obzirom da je već nekoliko puta naglašeno da je Rubyju jedna od najvećih prednosti upravo prirodna sintaksa, nema potrebe puno objašnjavati kôd koji puno govori sam za sebe. Pogledajmo sadržaj datoteke *schema.rb*. Ta datoteka je automatski generirana iz trenutnog stanja baze te je iz nje moguće vidjeti koje atribute ima koji model.

```
1  ActiveRecord::Schema.define(version: 20160829162922) do
2
3    create_table "microposts", force: :cascade do |t|
4      t.text     "content",     limit: 65535
5      t.integer  "user_id",     limit: 4
6      t.integer  "page_id",     limit: 4
7      t.datetime "created_at",                 null: false
8      t.datetime "updated_at",                 null: false
9      t.string   "attachment", limit: 255
10   end
11
12   add_index "microposts", ["page_id"], name: "
       index_microposts_on_page_id", using: :btree
13   add_index "microposts", ["user_id"], name: "
       index_microposts_on_user_id", using: :btree
14
15   create_table "pages", force: :cascade do |t|
16     t.string   "name",         limit: 255
17     t.text     "description", limit: 65535
18     t.datetime "created_at",                  null: false
19     t.datetime "updated_at",                  null: false
20   end
21
22   create_table "relationships", force: :cascade do |t|
23     t.integer  "follower_id", limit: 4
24     t.integer  "followed_id", limit: 4
```

```
25      t.datetime "created_at",              null: false
26      t.datetime "updated_at",              null: false
27    end
28
29    create_table "users", force: :cascade do |t|
30      t.string   "name",                    limit: 255
31      t.string   "email",                   limit: 255
32      t.datetime "created_at",                          null: false
33      t.datetime "updated_at",                          null: false
34      t.string   "password_digest", limit: 255
35      t.string   "remember_digest", limit: 255
36      t.boolean  "admin"
37    end
38
39    add_foreign_key "microposts", "pages"
40    add_foreign_key "microposts", "users"
41  end
```

Napomenimo još da metode klase *ActiveRecord* nisu bile dovoljne da bi se ostvarile sve mogućnosti zamišljene za model *user*, stoga je klasa *User* nadopunjena novim metodama kao što je metoda *follow* koja omogućuje korisniku da se pretplati na neki kolegij i metoda *feed* koja vraća objave kolegija koje korisnik prati zajedno s njegovim objavama:

```
1  def follow(page)
2    active_relationships.create(followed_id: page.id)
3  end
4
5  def feed
6    following_ids = "SELECT followed_id FROM relationships WHERE
       follower_id = :user_id"
7    Micropost.where("page_id IN (#{following_ids}) OR user_id = :user_id",
       user_id: id)
8  end
```

Ovako definirane metode koriste se na isti način kao i metode iz klase *ActiveRecord*.

3.2 Kontroleri aplikacije

Imamo šest kontrolera aplikacije:

- *UsersController*,
- *PagesController*,
- *MicropostsController*,

- *RelationshipsController*,
- *SessonsController*,
- *StaticPagesController*.

UsersController, *PagesController*, *MicropostsController* i *RelationshipsController* su kontroleri za navedene modele i definirani su na sličan način kao i kontroler u prethodnom poglavlju, s tim da su poštovane veze koje su stvorene između modela. Aplikacija ima sesiju za svakog korisnika gdje je moguće skladištenje manje količine podataka koji će biti očuvani između uzastopnih HTTP zahtjeva. *SessonsController* je zadužen za stvaranje i uništavanje sesije te prema tome ima samo dvije metode koje to omogućuju.

Da bismo preko drugih kontrolera saznali informacije koji korisnik je trenutno prijavljen, da li je korisnik admin i sl., u pomoćnoj klasi *SessionsHelper* definirali smo metode poput ovih:

```
def log_in(user)
  session[:user_id] = user.id
end

def current_user?(user)
  user == current_user
end

def admin?
  if (current_user == nil)
    return false
  end
  if (current_user.admin == true)
    return true
  else
    return false
  end
end
```

Da bismo korisnika ispravno prijavili i koristili informacije o njegovoj prijavi u *routes.rb* definirana je putanja:

```
get 'login'  =>     'sessions#new'
```

Dakle, novi korisnik će biti prijavljen u aplikaciju preko pogleda *new* koji se nalazi u *views/sessions*.

StaticPagesController je kontroler za statične stranice, a u ovom slučaju imamo samo jednu, *home*, pa nam kontroler ima sljedeći kôd:

```
class StaticPagesController < ApplicationController
  def home
    if logged_in?
      @micropost = current_user.microposts.build
      @feed_items = current_user.feed.paginate(page: params[:page])
    end
  end
end
```

3.3 Mogućnosti korisnika

Aplikacija Funda aplikacija je zatvorenog tipa. Da bismo koristili sve mogućnosti aplikacije potrebno je imati validan korisnički račun. Neregistrirani korisnik ima uvid samo u statičnu stranicu aplikacije. Početna stranica ima *signup* i *login* forme. Klikom na gumb *Prijavite se* na početnoj stranici preusmjeravanje nas preusmjeri na stranicu *users#new* u skladu s definicijom u datoteci *routes.rb*:

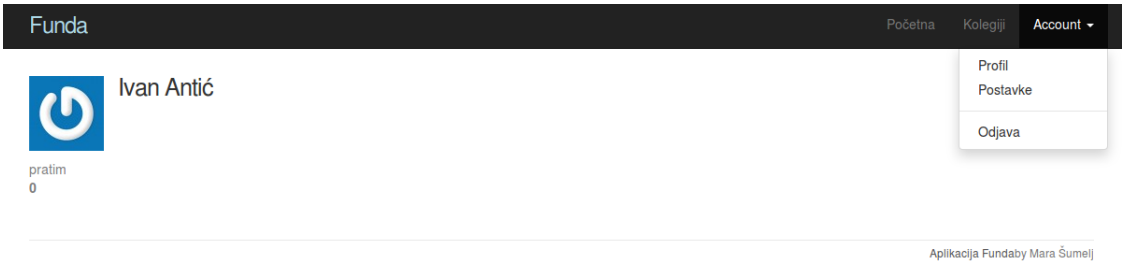
```
get 'signup'  =>  'users#new'
```

Preusmjereni smo na *sigup* formu te možemo kreirati račun.



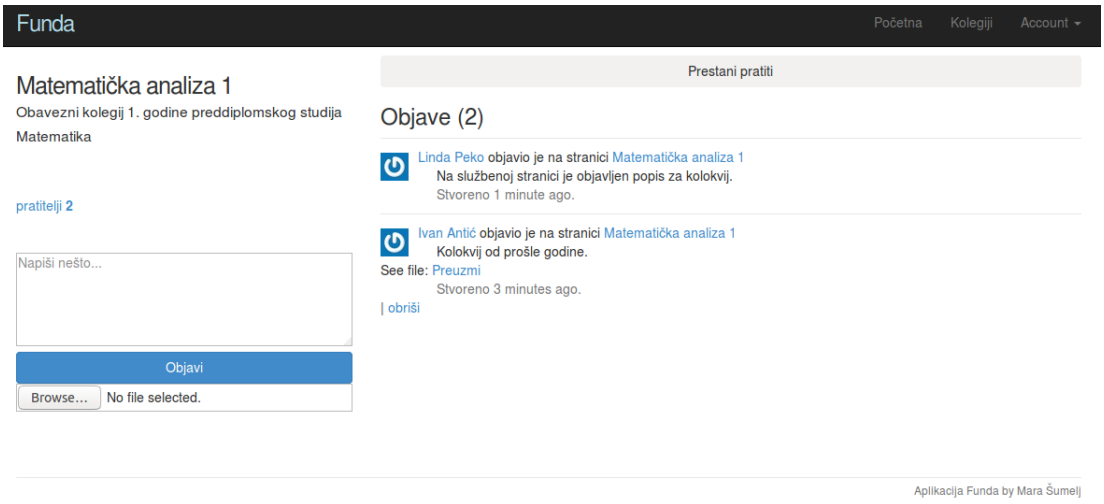
Slika 3.1: Forma za registraciju korisnika

Pomoću funkcije *create* u kontroleru *UsersController* stvaramo novog korisnika i spremamo njegove podatke u bazu, ukoliko su svi podaci ispravno unešeni. Korisnik je preusmjeren na stranicu svog profila te njegov profil sad izgleda kao na Slici 3.2.



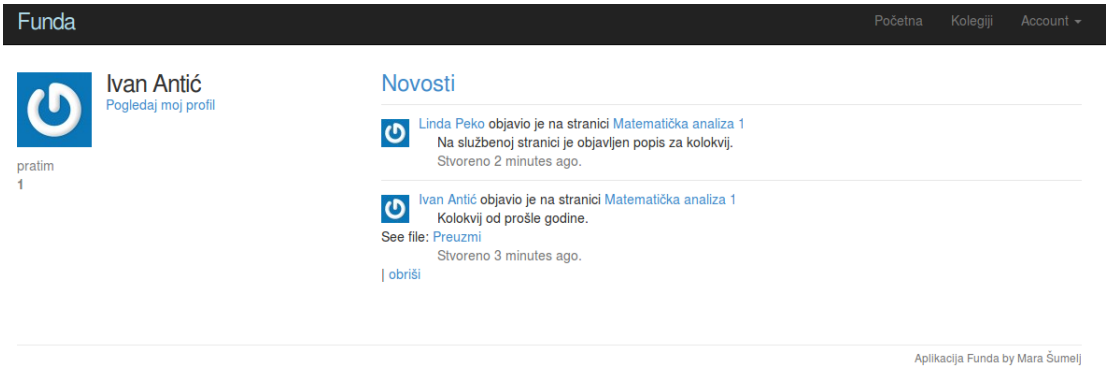
Slika 3.2: Profil korisnika

Vidimo da naš korisnik može ići na svoju početnu i na svoju profilnu stranicu, može pregledati koje stranice postoje, može ažurirati svoj profil i naravno na kraju se odjaviti. Uzmimo da našeg korisnika zanima kolegij *Matematička analiza*. Klikom na *Kolegiji* može pronaći njemu zanimljiv kolegij te pregledati stranicu tog kolegija.



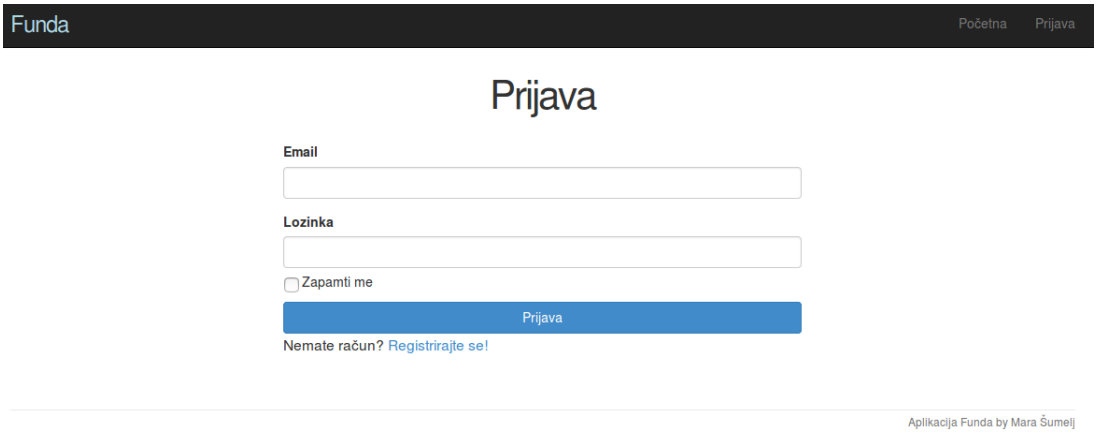
Slika 3.3: Stranica kolegija Matematička analiza

Ako korisnik odluči pratiti kolegij, kliknut će gumb *Pretplatite se* te će se na taj način na njegovoj početnoj stranici početi pojavljivati sve objave sa stranice tog kolegija.



Slika 3.4: Početna stranica nakon pretplate na kolegij Matematička analiza

Korisnik ima mogućnost pisanja objava te postavljanja datoteka i slika na stranicu. Svi sadržaji sa stranica kolegija prikazivat će se na početnoj stranici korisnika koji prate te kolegije. Korisnik se može odjaviti iz aplikacije. Da bi sljedeći put mogao pristupiti svojim podacima potrebno je unijeti email i odgovarajuću lozinku pomoću kojih je račun kreiran u odgovarajuća polja *login* forme.

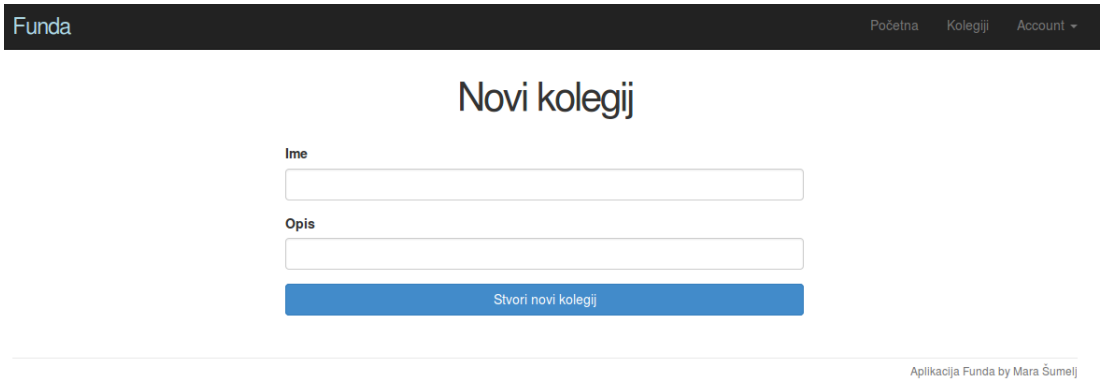


Slika 3.5: Forma za prijavu

Vidjeli smo kako izgleda korisničko sučelje aplikacije te prikazali najvažnije mogućnosti korisnika.

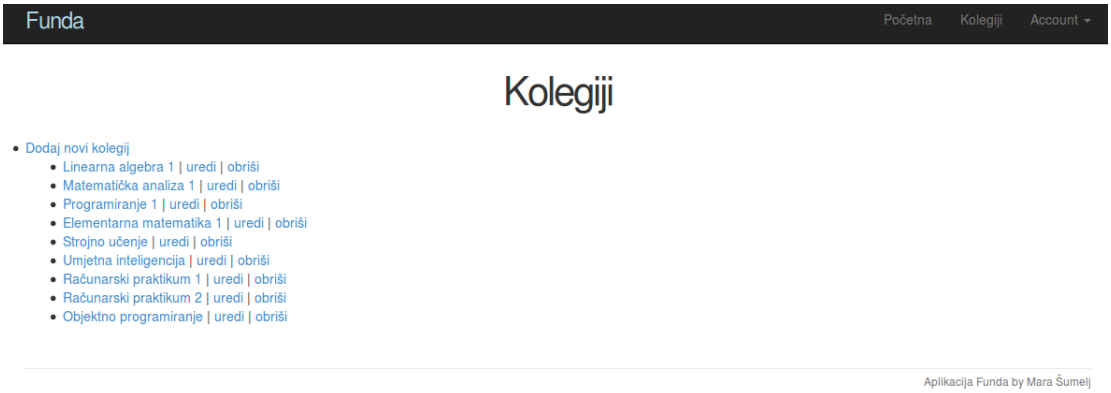
3.4 Superkorisnik - admin

Kada se obični korisnik prijavi na aplikaciju on može pronaći i pratiti već stvorene stranice. Stranice dodaje i briše superkorisnik - admin.



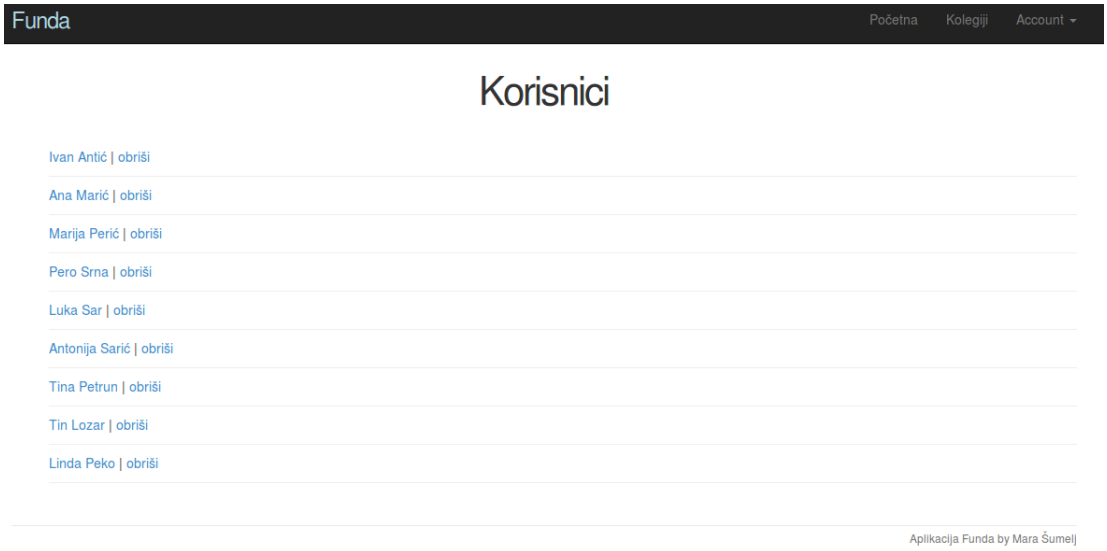
Slika 3.6: Stvaranje nove stranice

Superkorisnik može pristupiti popisu svih stranica koje postoje te ih može uređivati i brisati.



Slika 3.7: Svi kolegiji

Također, superkorisnik može pregledavati i brisati korisnike.



Slika 3.8: Svi korisnici

3.5 Proširenja

Navest ćemo proširenja koja su korištena za izradu ove aplikacije.

Za izgled aplikacije koristili smo *Bootstrap* okvir. Instaliramo ga na sljedeći način:

```
1 gem 'bootstrap-sass', '3.2.0.0'
```

Za umetanje dodatnih sadržaja u objave, poput slika i dokumenta korišten je *Uploader Carriewave*. Instaliramo ga na sljdeći način:

```
1 gem 'carrierwave', '~> 0.9'
```

Da stranice ne bi bile pretrpane podacima kada se baza podataka popuni koristili smo biblioteke *will_paginate* i *bootstrap-will_paginate*. Instaliramo ih na sljedeći način:

```
1 gem 'will_paginate', '3.0.7'
2 gem 'bootstrap-will_paginate', '0.0.10'
```

Već smo napomenuli da je mysql korištena kao baza podataka. Da bi je mogli koristiti instalirano je proširenje:

```
1 gem 'mysql2', '>= 0.3.13', '< 0.5'
```

Da bi osigurali lozinku korisnika od vanjskih napada koristili smo biblioteku *bcrypt*. Instaliramo ju na sljedeći način:

```
1 gem 'bcrypt', '~> 3.1.7'
```

Da bi popunili bazu podataka koristili smo biblioteku *faker*. Instaliramo ju na sljedeći način:

```
1 gem 'faker', '1.6.6'
```

Na službenoj stranici alata RubyGems moguće je pronaći popis svih mogućnosti proširenja, te upute kako ih uklopiti u Rails aplikaciju.

Bibliografija

[1] Službene web-stranice alata Bundler, `http://bundler.io/` (rujan 2016.)

[2] Michael Hartl, *Ruby on Rails Tutorial: Learn Web Development with Rails*, Addison Wesley, 2015.

[3] Službene web-stranice baze podataka MySQL, `https://www.mysql.com/` (rujan 2016.)

[4] Službene web-stranice programskog jezika Ruby, `https://www.ruby-lang.org/` (rujan 2016.)

[5] Službene web-stranice alata RubyGems, `https://rubygems.org/` (rujan 2016.)

[6] Službene web-stranice okruženja Ruby on Rails, `http://rubyonrails.org/` (rujan 2016.)

[7] Sandi Metz, *Practical Object-Oriented Design in Ruby: An Agile Primer*, Addison Wesley, 2015.

Sažetak

U ovom radu predstavljen je programski jezik Ruby i okruženje za razvoj web aplikacija Rails. Prikazane su osnovne karakteristike i specifičnosti Rubyja. Pokazano je na koji način Rails koristi REST i MVC arhitekture. Slijedeći konvencije Railsa, vrlo brzo dolazimo do velikog napretka u razvoju aplikacije. Budući da postoje brojna proširenja za često korištene zadatke, stvaranje web aplikacija pomoću Railsa je dodatno olakšano. Velikim brojem ugrađenih metoda programerima pojednostavljuje i skraćuje definiranje konfiguracije.

Summary

In this work we presented the programming language Ruby and web application framework Rails. Basic and specific characteristics of Ruby were described, along with the way of applying the MVC pattern and REST architecture in Rails. By following the conventions adopted by Rails, web application development can be very fast. Rails provides code extensions for common tasks, which speeds up the development even more. Built-in methods reduce and simplify code configuration.

Životopis

Rođena sam 23. prosinca 1991. godine u Splitu. Pohađala sam Osnovnu školu Zmijavci u Zmijavcima. Po završetku osnovnoškolskog obrazovanja upisala sam prirodoslovno-matematičku gimnaziju u Gimnaziji dr. Mate Ujevića u Imotskom. Maturirala sam 2010. godine te sam iste godine upisala preddiplomski sveučilišni studij Matematike na Matematičkom odsjeku Prirodoslovno-matematičkog fakulteta u Zagrebu. Godine 2013. završila sam dotadašnji studij te sam na istom fakultetu upisala diplomski sveučilišni studij Računarstva i matematike.