Razvoj web-aplikacija pomoću aplikacijskog okvira Sinatra

Pleša, Manuela

SVEUČILIŠTE U ZAGREBU

PRIRODOSLOVNO–MATEMATIČKI FAKULTET

MATEMATIČKI ODSJEK

Manuela Pleša

RAZVOJ WEB-APLIKACIJA POMOĆU APLIKACIJSKOG OKVIRA SINATRA

Diplomski rad

Voditelj rada:
prof. dr. sc. Robert Manger

Zagreb, rujan 2021.

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, koji su bili uz mene cijelo moje školovanje te šefu i zaposlenicima firme Devōt, koji su mi omogućili korištenje njihovog iskustva i opreme pri izradi ovog diplomskog rada

Sadržaj

Uvod

Svakodnevno smo sve više smo okruženi web-aplikacijama. Aplikacijama čiji je aplikacijski program pohranjen na nekom udaljenom poslužiteljskom računalu, a potrebne podatke nam dostavljaju preko Interneta kroz sučelje preglednika. Obično tok web-aplikacije započinje tako da korisnik pošalje zahtjev web-poslužitelju na Internetu se može slati preko web-pretraživača ili preko korisničkog sučelja aplikacije. Nakon toga, web-poslužitelj primljeni zahtjev pošalje poslužiteljskom računalu web-aplikacije koji je namjenjen za rješavanje tog zahtjeva te on izvrši traženi zadatak. Dobiveni rezultat poslužiteljsko računalo web-aplikacije šalje natrag web-poslužitelju koji zatim taj rezultat prikaže korisniku koji je taj zahtjev i poslao. Iz prikazanog toka rada web-aplikacije vidimo da rad nije ograničen operacijskim sustavom ili računalom na kojem ju pokrećemo već je za rad aplikacije bitno da se otvara na pretraživaču koji ju podržava. Osim toga, kao prednost web-aplikacija možemo navesti i to da ne troše memoriju tvrdog diska s obzirom da ne moraju biti instalirane na računalu kako bismo ih mogli koristiit. Stoga nije čudno što se javlja sve više aplikacijskih okvira za razvoj web-aplikacija, a jedan od njih je i Sinatra.

Prvo ćemo detaljnije opisati što je okvir za web-aplikacije, te kakve sve tipove aplikacijskih okvira možemo susresti. Nadalje, opisat ćemo aplikacijski okvir Sinatra kojeg je dizajnirao i implemetirao Blake Mizerany, a baziran je na programskom jeziku Ruby, kojim ćemo se također baviti u jednom od poglavlja. Vidjet ćemo njegov razvoj te ga usporedit s još nekim aplikacijskim okvirima koji su bazirani na programskom jeziku Ruby, kao na primjer Ruby on Rails ili Padrino, ali i s nekim koji su bazirani na drugim programskim jezicima. Također ćemo općenito opisati kako izgleda razvoj web-aplikacije u Sinatri, od ideje do prvog pokretanja aplikacije na poslužiteljskom računalu.

Zatim ćemo na studijskom primjeru pokazati kako na vrlo jednostavan način u Sinatri koja je DSL (eng. *Domain Specific Language*) kreirati MVC (eng. *Model-View-Controller*) arhitekturu te ćemo vidjeti kako se kao rezultat toga dobila aplikacija Cijepljenje. Riječ je o aplikaciji koja se na prvi pogled čini prekompleksna za Sinatru koja se predstavlja kao aplikacijski okvir za jednostavne i male aplikacije, no Sinatra je ipak tome dorasla. Opisat ćemo sve alate i metode koji su bili potrebni za razvoj aplikacije iz studijskog primjera. I za kraj ćemo vidjeti neke prednosti i nedostatke Sinatre, gdje ju je najbolje koristiti, a gdje možda bolje izbjegavati i kako bi mogao izgledati daljnji razvoj Sinatre.

Poglavlje 1

Okviri za web-aplikacije

Kako je Sinatra jedan od primjera okvira za web-aplikacije, korisno je za početak saznati malo više o samim okvirima za web-aplikacije. Proći ćemo ukratko povijesni pregled njihovog nastanka, opisat ćemo najpoznatije tipove arhitektura, MVC i troslojnu arhitekturu te grubu podjelu okvira za web-aplikacije na poslužiteljske i klijentske.

1.1 Povijest

Kao što im samo ime kaže, okviri za web-aplikacije su dijelovi softvera koji nam nude mogućnost kreiranja i pokretanja web-aplikacija. Već u samom uvodu smo ukratko opisali što su to web-aplikacije stoga se sad možemo posvetiti samo načinu na koji ih kreiramo. No, da bi objasnili okvire za web-aplikacije, pogledat ćemo malo povijest samih web-aplikacija.

Kao početak web-aplikacija možemo navesti prvi web-pretraživač, odnosno WorldWideWeb, i to 1990. godine. U tim početcima jedini način da se nešto promijeni na web-aplikaciji bilo je da tu promijenu izvede programer koji ju je i kreirao. Zatim se 1993. godine dogodilo povezivanje takvih aplikacija s web-poslužiteljima te su tako omogućene dinamičke web-stranice te se pojavljuju i prve web-aplikacije kakve i danas znamo. Iako na prvi pogled isti pojmovi, web-aplikacija i web-stranica nisu ekvivalentni pojmovi i to možemo jednostavno objasniti definicijom web-aplikacije koristeći pojam web-stranice. Naime, web-aplikacija je web-stranica koju korisnik može kontrolirati.

Kroz godine razvijeni su razni programski jezici namijenjeni isključivo za web, kao na primjer PHP i ColdFusion. Takvi jezici imali su biblioteke za razvoj web-aplikacija, ali za neke specifične zadatke ipak su bile potrebne još neke dodatne biblioteke izvan tih jezika. Pred kraj 90-ih godina javljaju se okviri koji objedinjuju više biblioteka koje su potrebne web-aplikacijama u jedan softverski stog (eng. *software stack*) te tako olakšavaju posao web-programerima pri implementaciji web-aplikacija. Neki od najpoznatijih okvira

koji su se tada pojavili su Django, okvir baziran na programskom jeziku Python, zatim Zend Framework čija osnova je PHP te u ovom radu već spomenuti, Ruby on Rails koji se temelji na programskom jeziku Ruby.

1.2 Arhitekture okvira za web-aplikacije

Pojam arhitektura okvira za web-aplikacije predstavlja interakciju između aplikacije, baze podataka i međuslojnih (eng. *middleware*) sustava na webu. U ovom poglavlju ćemo opisati neke od češće korištenih arhitektura, a odlučili smo se da su dobri predstavnici MVC i troslojna arhitektura.

Model-View-Controller (MVC)

Već u samom uvodu smo ga spomenuli, ali je to ujedno i najpoznatiji dizajn arhitekture web-aplikacija. Značajka ovog dizajna je da razdvaja modele podataka i pravila poslovanja, odnosno logiku aplikacije od sučelja kojeg koristi korisnik. Kao što i samo ime kaže, to razdvajanje se očituje u dobivanju tri osnovna dijela koja oblikuju arhitekturu aplikacije.

Sloj modela bi bio najniži sloj što bi značilo da je odgovoran za rad s podacima. Povezan je s bazom podataka pa se dodavanje i dohvaćanje podataka radi pomoću tog sloja. Model komunicira sa slojem kontrolera (eng. *controller*) i prima njegove upite u vezi podataka spremljenih u bazi. Nakon što primi upit, model iz baze dohvati što je zatraženo upitom te rezultat vrati kontroleru. Kao što vidimo, kontroler sloj nikada ne komunicira direktno s bazom podataka već samo preko modela.

Sloj pogleda (eng. *view*) služi za prikaz podataka, stoga ovaj sloj možemo jednostavno definirati kao *front-end* dio aplikacije. Naime, to je dio aplikacije kojeg korisnik zapravo vidi i općenito možemo reći da je to HTML i CSS dio aplikacije. Podatke koje pogled prikazuje su rezultat upita model sloja prema bazi podataka.

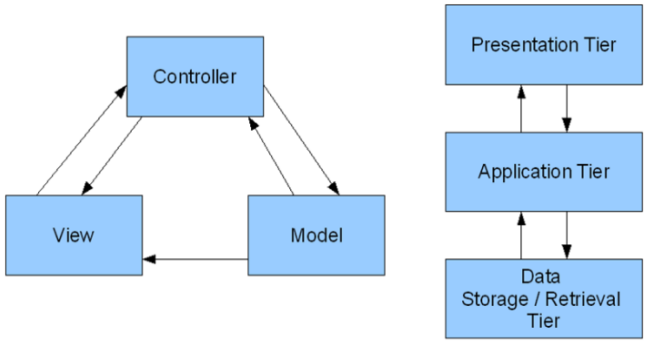
Sloj kontrolera možemo smatrati najbitnijim slojem jer nam služi kako bismo povezali model i pogled. On direktno nije odgovoran za upravljanje logikom podataka već mu je uloga slanjem upita prema modelu reći mu što treba napraviti s podacima. Isto tako ne brine pomoću čega će biti ostvaren prikaz dohvaćenih podataka na sučelju aplikacije, već samo pogledu pošalje podatke koje je primio od modela s uputom kako bi trebali biti prikazani.

Troslojna arhitektura (three-tier)

Ovaj tip arhitekture kao i MVC definira strukturu aplikacije kao tri sloja - prezentacijski odnosno korisnički, aplikacijski i sloj s podacima. Glavna značajka ove arhitekture je to što svaki od tri sloja radi na svojoj infrastrukturi. Oni mogu biti kreirani odvojeno jedan

od drugoga čak i pomoću različitih programerskih timova te se po potrebi mogu mijenjati neovisno jedan o drugome.

Ovdje treba naglasiti da nije neobično da se ova arhitektura zamijeni sa MVC arhitekturom opisanom u prošlom poglavlju. No, razlika je u tome što je MVC arhitektura u obliku trokuta, odnosno pogled pošalje promjene kontroleru, zatim kontroler ažurira model prema tim promjenama te se na kraju pogled ažurira direktno iz modela. Dok je troslojna arhitektura linearna, klijentski dio nikada ne komunicira direktno sa slojem podataka već sva komunikacija ide preko srednjeg, aplikacijskog dijela. Na slici 1.1 možemo grafički vidjeti upravo opisanu razliku između MVC i troslojne arhitekture okvira za web-aplikacije.



Slika 1.1: Razlika između troslojne i MVC arhitekture

Prezentacijski sloj (eng. *presentation tier*) je najviši sloj i predstavlja samo sučelje aplikacije u kojem korisnik komunicira s aplikacijom. On se može pokretati u web-pretraživaču, na desktopu računala ili kao grafičko korisnično sučelje (eng. *graphical user interface* - GUI). Glavna svrha mu je prikazivanje podataka i skupljanje podataka od korisnika koji koristi aplikaciju, a obično je implementiran koristeći HTML, CSS i JavaScript.

Aplikacijski sloj (eng. *application tier*) je srednji ili logički sloj. On služi kako bi obradio podatke skupljene u prezentacijskom sloju koristeći logiku i pravila aplikacije u kojoj se koristi. Osim obrade podataka, on također može dodavati, brisati ili mijenjati podatke iz sloja podataka pomoću API (eng. *Application Programming Interface*) poziva. Za implementaciju aplikacijskog sloja obično se koriste programski jezici Ruby, Python, PHP ili Java.

Sloj s podacima ili sloj baze podataka (eng. *data storage tier*) je sloj u kojemu se podaci obrađeni od strane aplikacije pohranjuju i gdje se njima upravlja. Taj sloj može biti neki od sustava za upravljanje bazama podataka kao na primjer PostgreSQL, MySQL, MariaDB ili Oracle. Isto tako može biti i neki od NoSQL poslužitelja za baze podataka kao Cassandra ili MongoDB.

1.3 Vrste okvira za web-aplikacije

Kako su se razvijale web-aplikacije, tako se javila potreba da se logika aplikacije prebaci na klijentski dio kako bi aplikacija mogla brže odgovoriti na unose i upite korisnika te aplikacije. Tako dolazimo do pojave aplikacijskih okvira namjenjenih za poslužiteljsku stranu aplikacije i onih namjenjenih isključivo za klijentsku stranu, ali u istoj aplikaciji se mogu kombinirati obje vrste ako imamo potrebu za time.

Okviri poslužiteljske strane aplikacije

Aplikacijski okviri namjenjeni poslužiteljskoj strani omogućuju alate i biblioteke koji olakšavaju uobičajene implementacijske zadatke. Između ostalog omogućuju usmjeravanje URL-ova odgovarajućim rukovateljima i komuniciranje s bazom. Također podržavaju *sessions* autentifikaciju korisnika i poboljšavaju zaštitu protiv web-napada.

Neki od najpoznatijih aplikacijskih okvira za poslužiteljsku stranu aplikacije su .NET koji je baziran na programskom jeziku C#, već ranije navedeni Django i Ruby on Rails te Symphony, aplikacijski okvir čija osnova je PHP.

Okviri klijentske strane aplikacije

Za razliku od okvira poslužiteljske strane, okviri klijentske strane se ne zamaraju logikom aplikacije već se koriste za *front-end* dio aplikacije omogućavajući programeru alate za kreiranje skalabilne i interaktivne web-aplikacije, a rade na klijentovom pretraživaču.

Neki od aplikacijskih okvira koji se koriste za klijentsku stranu aplikacije su Angular, React.JS i Ember.JS, a svi navedeni okviri kao programski jezik koriste JavaScript.

Poglavlje 2

Aplikacijski okvir Sinatra

Nakon što smo objasnili što su to općenito okviri za web-aplikacije, sada ćemo se posvetiti aplikacijskom okviru Sinatra. Prvo ćemo opisati programski jezik Ruby koji je temelj Sinatre, točnije vidjet ćemo glavne značajke i specifičnosti Ruby-ja te kako ga instalirati. Zatim ćemo opisati samu Sinatru te ju usporediti s trenutno popularnim okvirima za web-aplikacije.

2.1 Programski jezik Ruby

Programski jezik Ruby kreirao je Yukihiro Matsumoto i pustio u javnu uporabu 1996. godine. No, tek 2006. godine Ruby je doživio javno priznanje. Kombiniranjem jezika Perl, Smalltalk, Eiffel, Ada i Lisp, Yukihiro Matsumoto je dobio jezik koji je uravnotežio funkcionalno programiranje s imperativnim. Ruby je dinamički programski jezik otvorenog koda kojem je naglasak na jednostavnosti i produktivnosti, on je također i besplatan za korištenje, kopiranje, modificiranje i distribuiranje.

Ruby je objektno-orijentirani jezik, ali se razlikuje od drugih jezika takve vrste po tome što podržava samo jednostruko naslijeđivanje, ali prepoznaje koncept modula. Na taj način dodavanjem modula u klasu, klasa naslijeđuje sve metode modula. Također, postoji klasa koja je nadklasa svim klasama koje definiramo i to je klasa *Object* te sve klase koje definiramo nasljeđuju sve metode definirane u klasi *Object*.

Nadalje, jedna od bitnijih karakteristika Ruby-ja je to što je kod njega sve objekt, i klase i instance tipova, a varijable pohranjuju reference na objekte. Ako se pogleda sama sintaksa Ruby-ja, možemo vidjeti sličnosti s programskim jezicima Perl i Python, ali isto tako postoje i razlike. Kod Ruby-ja za razliku od Python-a i Perl-a sve varijable koje definiramo u nekoj klasi su privatne i može im se pristupiti izvan klase samo preko pristupnih metoda *attr_writer*, koja služi za mijenjanje vrijednosti varijable, *attr_reader* za čitanje vrijednosti i slične. To nas može podsjetiti na *getter* i *setter* metode programskih jezika C++

ili Java, ali je razlika u tome što se pristupne metode u Ruby-ju mogu pisati i kao jednolinijske metode pomoću metapogramiranja, ali i standardno kao pristupne metode u C++-u ili Java-i.

Ruby ima nekoliko različitih implementacija. Najpoznatija je vjerojatno MRI, što je kratica za Matz's Ruby Interpreter, prema nadimku samog kreatora Yukihiro Matsumotoa. Ona je još poznata i pod nazivom CRuby jer je pisana u programskom jeziku C. Ostale implemetacije su posebne po tome što omogućuju integraciju u druge jezike, okruženja ili sadrže neke mogućnosti koje MRI ne sadrži. Nabrojat ćemo neke od poznatijih:

- JRuby je Ruby na Java Virtualnom Stroju (JVM).
- Mruby je perolaki Ruby, dizajniran za povezivanje i ugrađivanje u aplikacije.
- Topaz je Ruby implementiran u Pythonu nad RPythonom, lancem alata koji pokreće PyPy.
- Fruby je Ruby programski jezik, ali na francuskom.

Instalacija

Postoji nekoliko načina instaliranja programskog jezika Ruby na računalo. Na UNIX operacijskim sustavima najlakša metoda je korištenje upravitelja paketa. Ako želimo instalirati na Debian GNU Linux ili Ubuntu operacijski sustav, korist ćemo apt upravitelj paketa:

```
sudo apt-get install ruby-full
```

Kako bi se instalirao na MacOS operacijski sustav pomoću upravitelja paketa, to ćemo obaviti pomoću Homebrew-a, kojeg prethodno treba dodati na računalo, te naredbe:

```
brew install ruby
```

Za instalaciju na Windows operacijski sustav bolji način bi bio koristiti metodu pomoću instalera kojima se može specificirati koju verziju Ruby-ja želimo. Točnije, korisit ćemo program RubyInstaller tako da se preuzmu svi potrebni paketi za instalaciju sa službene web-stranice.

Jos jedan način za instalaciju Ruby programskog jezika je korištenje asdf-vm upravitelja verzija. Asdf-vm je proširujući upravitelj verzija koji može upravljati izvedbenim verzijama više programskih jezika za svaki projekt posebno. Za instalaciju Ruby-ja pomoću

asdf-vm potreban je asdf-ruby plugin koji koristi ruby-build kako bi instalirao Ruby na računalo.

I posljednji način kako se može instalirati Ruby je instalacija pomoću izvornog koda. Potrebno je preuzeti i otpakirati *tarball* te izvrsiti naredbe:

```
./configure
make
sudo make install
```

Da bismo provjerili da je Ruby instaliran na računalu, izvršimo naredbu provjere verzije Ruby-ja:

```
ruby -v
```

Nakon što je Ruby instaliran na računalu, dokumente s `rb` ekstenzijom pokrećemo tako da navigiramo do mape u kojoj je dokument smješten te u terminalu izvršimo naredbu:

```
ruby.rb imeDatoteke
```

Specifičnosti Rubyja

Već na prvi pogled kod Ruby-ja se primjećuje blokovska sintaksa. Točnije, ne koriste se zagrade kako bi se odijelili dijelovi koda kao na primjer u programskom jeziku C u kojem se koriste vitičaste zagrade ({}). Kako onda znamo kada je kraj nekog dijela koda? Za razliku od programskih jezika kao na primjer Python gdje je to riješeno indentacijom, kod Ruby-ja je to izvedeno elegantije. Naime, svaki blokovski dio koda završava sa naredbom `end`. kao što možemo vidjeti i na primjeru if-else grananja u funkciji `provjera_broja`.

```
1 def provjera_broja(x)
2     if x > 2
3       puts 'x je veći od 2!'
4     elsif x <= 2 and x!=0
5       x += 2
6       puts 'x je bio 1 i uvećan je za 2'
7     else
```

```
8         puts 'broj je manji ili jednak 0'
9       end
10  end
```

Ruby je među rijetkim programskim jezicima koji koriste tip podataka *symbol*, a to su zapravo statički stringovi za identifikaciju. Najjednostavniji primjer korištenja simbola je *hash* ključ, ali najčešće se koriste kao reprezentacija metoda ili kao instanca imena varijabli.

```
1  hash = {a: 1, b: 2, c: 3}
2
3  # ispis vrijednosti elementa s ključem a gdje je ključ zapisan kao simbol
4  puts hash[:a]
```

Svaki simbol je jedinstven i nije promjenjiv te ih se treba koristiti kao imena ili oznake za dijelove kao što su metode, dok je obične stringove bolje koristiti kada više brinemo za podatke. Kako su stringovi i simboli povezani, postoji vrlo lagan način pretvaranja jednog tipa u drugi. Za pretvaranje simbola u *string* koristimo metodu `.to_s`, dok za obratan smjer koristimo `.to_sym`.

S obzirom da smo već spomenuli dvije vrste konverzija, sada ćemo navesti još neke koje se svakondevno upotrebljavaju. Da biste pretvorili objekt nekog tipa u tip *integer*, dovoljno je na tom objektu pozvati metodu `.to_i`. Analogan postupak se primjenjuje i na već spomenutu metodu konverzije u *string* sa `.to_s` te za konverziju u *array* sa `.to_a`. No, ako nismo sigurni hoćemo li u konverziju poslati `nil` vrijednost ili konkretan objekt, a želimo biti sigurni da dobijemo traženi tip nakon konverzije, sigurnije je koristiti metode `Array()` i `Integer()`.

Uvid u dokumentaciju programskog jezika Ruby se može obaviti na [1] iako postoje i brojne druge stranice na kojima je opisano koje sve mogućonosti Ruby ima.

Ruby Gemovi

Kao što svaki programski jezik ima bibloteke kojima se omogući korištenje specifičnih metoda tako i Ruby ima *gem*ove. *Gem* je paket koji se može preuzeti i instalirati te nakon što ga *require*amo u Ruby programu on će mu omogućiti dodatne funkcionalnosti. Većina *gem*ova je implementirana u čistom Ruby-ju, dok ih dio ima CRuby ekstenzije kako bi im se poboljšala izvedba.

Kada se govori o *gem*ovima, nezaobilazno je reći nešto i o Bundler-u. To je alat za upravljanje ovisnostima (eng. *dependency management*). Njega se koristi prilikom implementacije aplikacije u Ruby-ju. U aplikaciji koju kreiramo trebamo imati dokument

`Gemfile` u kojem navodimo sve *gem*ove koji će nam biti potrebni za izgradnju aplikacije. Zatim, kako bi se ti navedeni *gem*ovi primjenili u našoj aplikaciji potrebno je izvršiti naredbu `bundle install` čime se kreira ili izmjeni dokument `Gemfile.lock` u kojemu se može vidjeti koja verzija kojeg *gem*a je instalirana. Kako bi korištenje *gem*ova bilo jednostavnije i brže, nije potrebno svaki od njih posebno instalirati na naše računalu, već one koji nam trebaju u aplikaciji dovoljno je navesti u pripadnom `Gemfile`u.

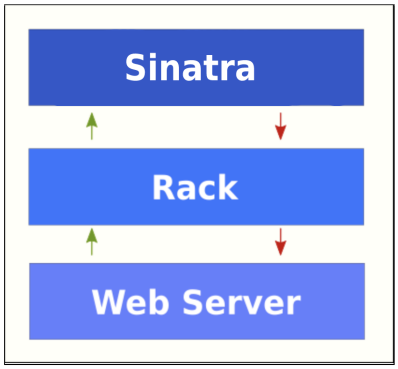
```
1  # Gemfile
2  source 'https://rubygems.org'
3
4  gem 'aasm', '~> 5.1', '>= 5.1.1'
5  gem 'activerecord', require: 'active_record'
6  gem 'activesupport'
7  gem 'bcrypt', '~> 3.1', '>= 3.1.12'
8  gem 'byebug'
9  gem 'data_mapper'
10 gem 'dm-postgres-adapter', '~> 1.2'
11 gem 'json', '~> 1.8.6'
12 gem 'pg'
13 gem 'pony', '~> 1.11'
14 gem 'require_all'
15 gem 'sinatra-activerecord'
16 gem 'bootstrap'
17
18 group :test do
19   gem 'database_cleaner-active_record'
20   gem 'rack-test'
21   gem 'shoulda-matchers', '~> 4.0'
22 end
23
24 group :production, :development, :test do
25   gem 'rake'
26   gem 'thin'
27 end
```

Popis svih Ruby *gemova*, mnoge od kojih nismo koristili, može se pogledati na [7]. Odnosno, na službenoj stranici na kojoj se nalaze svi *gem*ovi, uz svaki je dana njegova namjena te su nabrojane koje verzije tog *gem*a postoje do sada.

2.2 Rack

Rack je minimalno, modularno i prilagodljivo sučelje za razvoj web-aplikacije u Ruby-ju. Korisno ga je koristiti jer omogućuje zamjenjivost poslužitelja i aplikacijskih okvira, tako da oni postaju komponente koje možemo uključivati. U praksi bi to značilo da možemo isti poslužitelj koristiti sa Railsima i Sinatrom i bilo kojim drugim okvirom koji je kompatibilan s Rack-om.

To sučelje je vrlo jednostavni mali program u programskom jeziku Ruby koji se poziva kao dio zahtjev−odgovor kruga. On se može ponašati i kao čuvar tako da niječe zahtjeve koji nisu željeni ili pratiti odgovore koji su spori. Najčešće se koristi za prijavu u aplikaciju, *sessione*, *caching* ili sigurnost aplikacije.



Slika 2.1: Grafički prikaz uloge Rack sloja

U studijskom primjeru, Rack nije korišten kod kreiranja aplikacije već kod testiranja. Stoga smo za korištenje testne bibloteke bazirane na Rack-u trebali u projekt dodati *gem* `rack-test` te u svaki testnih dokumenata dodati redak u kojem uključujemo Rack::Test::Methods. Sam način testiranja i testove koji su korišteni u studiijskom primjeru će biti opisani u poglavlju 3.8.

2.3 Glavne značajke aplikacijskog okvira Sinatra

Iz samog imena može se zaključiti da je aplikacijski okvir Sinatra ime dobio prema poznatom pjevaču Frank Sinatri. Prvotno, Sinatra je kreirana kako bi bila alternativa web-aplikacijskim okvirima poput Ruby on Rails, Camping, Nitro i Merb. No, za razliku od Rails-a čija implementacija ima oko 100 000 linija koda, implementacija Sinatre ima manje

od 2 000 linija zbog čega je pogodna za male aplikacije i API-je koji ne zahtjevaju potpuno razvijene web-okvire.

Slika 2.2: Logo aplikacijskog okvira Sinatra

Kao što je ranije napomenuto, Sinatra aplikacijski okvir je zapravo DSL za izradu web-stranica, web-servisa i web-aplikacija koristeći programski jezik Ruby. To što je Sinatra DSL znači da je namjenjen za rješavanje posebne domene problema. Kod Sinatre domena problema je izrada web-aplikacija i aplikacija koje komuniciraju s pretraživačima pomoću HTTP-a. HTTP je jedan od protokola za prijenos informacija na Internetu. Točnije, to je *request-response* protokol kojim komuniciraju poslužitelj, odnosno server i klijent. Neke od najkorištenijih *request* metoda su GET, POST, PUT i DELETE, a *response* poruke protokola sastoje se od koda statusa odgovora, nula ili više polja zaglavlja odgovora, prazne linije i opcionalnog tijela poruke.

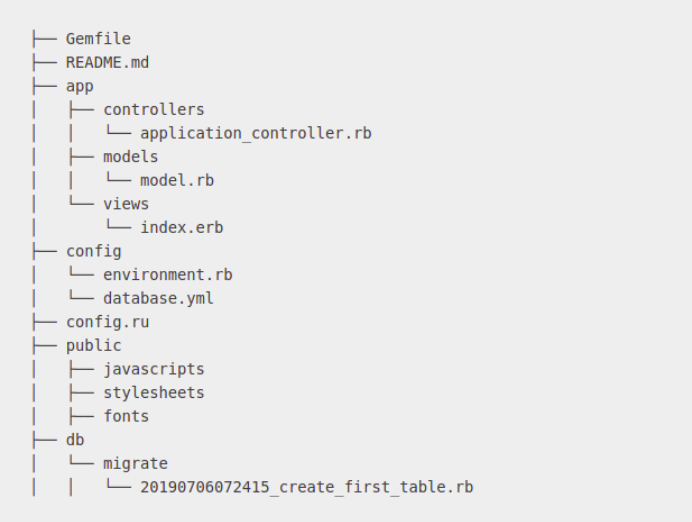
Nadalje, kao jednu od istaknutijih značajki Sinatre navest ćemo njegovu jednostavnost. Ako želite kreirati funkcionalnu aplikaciju pomoću samo jednog jedinog dokumenta, Sinatra je aplikacijski okvir koji vam je potreban. To je okvir kreiran nad Rack slojem, a što je Rack smo objasnili u poglavlju 2.2 te je perolaki, odnosno sadrži samo ono najnužnije i prilično je fleksibilan.

2.4 Kreiranje aplikacije u Sinatri

Kao i kod razvoja svake aplikacije jako važan je prvi korak, a to je odlučiti kakvu aplikaciju će se razvijati. Naravno, prilikom procesa implementacije ta ideja će se razvijati i mijenjati, ali u startu ju treba dovoljno razraditi da bi se moglo s nečim početi. Potrebno je odrediti tko će biti akteri u toj aplikaciji, tko će imati od njih kakve ovlasti, kakav je slijed pojedine akcije u aplikaciji i slično.

Nakon što je određeno kakvu aplikaciju će se razvijati, već se zna odluka za idući korak, a to je odrediti strukturu aplikacije. Naime, već na par mjesta je napisano kako je Sinatra jako dobar aplikacijski okvir za male aplikacije, čak i za one koje se sastoje od samo jednog dokumenta i nemaju povezanost s bazom podataka. Također, bit će pokazano da se u Sinatri može razviti i kompliciranija aplikacija koja će se povezivati s bazom podataka. Stoga odgovor na pitanje, koju strukturu aplikacije uzeti, daju nam odluke donešene u prvom paragrafu. Jer već nakon odluke kakva aplikacija će se kreirati, zna se hoće li to biti moguće izvesti u jednom dokumentu ili će ipak trebati definirati strukturu mapa i dokumenata za tu aplikaciju. Kako je interesantije promatrati razvoj veće aplikacije, ovdje ćemo opisati kako bi okvirno izgledalo kreiranje takve aplikacije u Sinatri, od ideje do prvog pokretanja na serveru. Konkretni primjer i primjene svega navedenog ovom poglavlju pokazat ćemo na studijskom primjeru, odnosno aplikaciji Cijepljenje u poglavlju 3.

Kako smo odlučili raditi veću aplikaciju, trebamo se odlučiti i za arhitekturu. Kao najbolji izbor se nametnula MVC arhitektura. Sada treba odrediti koliko će biti modela, koje sve atribute će ti modeli imati i kako će međusobno biti povezani. Zatim, koliko će biti pogleda, odnosno, što sve će trebati prikazati korisniku. I konačno, koliko ćemo imati kontrolera, oni se većinom rade za svaki od modela, ali kako je Sinatra DSL onda radimo kontrolere za one modele za koje će nam trebati API pozivi. Točnije, definiramo one API pozive koje ćemo trebati pozivati u aplikaciji. Kada je i to odlučeno, može se napraviti okvirna struktura mapa i dokumenata koji će činiti aplikaciju. Neka najstandardnija sturktura koja sadrži samo osnovno da bi se aplikacija mogla pokrenuti prkazana je na slici 2.3.



Slika 2.3: Stuktura aplikacije koja koristi MVC arhitekturu

Sada slijedi malo detaljniji opis što nam je na slici zapravo prikazano. `Gemfile` je dokument koji smo objasnili u potpoglavlju 2.1. `README.md` je dokument koji nije nužan u projektu, ali ga je lijepo imati kako bismo nekome tko uzme kod aplikacije objasnili kakva je to aplikacija, što ona radi, kako ju pokrenuti i naveli neke specifičnosti koda ako postoje. Mapu *app* nije potrebno objašnjavati jer smo detaljno objasnili što su *Model*, *View* i *Controller* u potpoglavlju 1.2.

Mapa *config* je zapravo jedna od bitnijih mapa jer se u njoj nalazi `environment.rb` koji omogućava da se sve komponente koje su implementirane povežu s aplikacijom kako bi ih ona mogla koristiti. Osim `environment.rb`, vidimo da je tu i dokument `database.yml`. Taj dokument je potreban samo u slučaju ako se aplikacija povezuje s bazom podataka kako bi se definiralo koju bazu podataka koristimo u kojem okruženju. Također jedan od bitnih dokumenata je i `config.ru` dokument koji služi za konfiguraciju Rack-a, što je zapravo sučelje između web-poslužitelja i web-aplikacije koje je implemetirano u Ruby-ju, a oblašnjeno je u potpoglavlju 2.2. U sljedećim isječcima koda možemo vidjeti općenite primjere kako bi dokumenti `config.ru` i `environment.rb` trebali izgledati.

```ruby
# config.ru

require_relative './config/environment'
run AppController
```

```ruby
# environment.rb
ENV['SINATRA_ENV'] ||= 'development'

require 'bundler/setup'
require 'sinatra'
Bundler.require

require './app/controllers/app_controller'
require_all 'app'
```

U mapi *public* nalaze se sve potebne mape i dokumenti koji služe za dizajn dijela aplikacije kojeg koristi korisnik. U mapi *db*, koja se dodaje kada se radi s bazom podataka što i je slučaj u našem primjeru, se nalaze migracije koje nam služe kako bismo kreirali potrebne tablice i veze između tih tablica u bazi podataka, a tablice nam predstavljaju modele od kojih se sastoji naša aplikacija. Nakon što izvršimo definiranu migraciju, napravljene promjene možemo vidjeti u `scheme.rb`. To je dokument koji nam prikazuje sturkturu

baze podataka, od kojih tablica se sastoji, koje stupce ima pojedina tablica te odnose među tablicama. Osim `scheme.rb` korisno je dodati još i dokument `seed.rb` kako bismo pri inicijalnom pokretanju aplikacije već imali početne podatke s kojima možemo nešto raditi. U sljedećem isječku koda možemo vidjeti migracijski dokument kojim se kreira tablica *admins* koja predstavlja korisnika aplikacije koji ima ulogu administratora.

```ruby
# 20210609114731_create_admin.rb
class CreateAdmin < ActiveRecord::Migration[6.1]
  def change
    create_table :admins do |t|
      t.string :email, null: false
      t.string :password_digest, null: false
      t.timestamps
    end
  end
end
```

Sada kada imamo definiranu strukturu i nakon što smo odlučili koje sve modele i veze ćemo imati, sada je dobar trenutak za odrediti koju bazu podataka ćemo koristiti u aplikaciji kako bismo znali popuniti `database.yml`. Kada se radi o izradi aplikacije u Ruby-ju, najčešće se izbor radi izmedu sqlite i PostgreSQL baze podataka. Naravno, na programeru je da odluči koja mu se čini jednostavnija za ono što je potrebno odraditi. U našem primjeru se koristi baza PostgeSQL kako bismo olakšali kasnije *deploy*anje aplikacije na platformu Heroku koja je kompatibilna s tom bazom. U sljedećem isječku koda prikazan je primjer popunjavanja dokumenta `database.yml`.

```yaml
# database.yml
test:
  adapter: postgresql
  database: db/test_cijepljenje

development:
  adapter: postgresql
  database: db/development_cijepljenje

production:
  adapter: postgresql
  database: db/production_cijepljenje
```

Kada smo kreirali najbitnije dokumente, dodali u mapu *controllers* barem dokument `app_controller.rb` i popunili ih sve na okvirno opisani način, sada ćemo objasniti i kako ju pokrenuti. Aplikaciju, odnosno poslužitelj na kojemu će se aplikacija vrtjeti može se pokrenuti na više načina, a neke od njih se može vidjeti u [6]. Zbog načina na koji smo mi postavili projekt, odnosno aplikaciju, mi ćemo poslužitelj pokretati eksplicitno pozivajući dokument `config.ru` koji onda pozove potreban controller stoga ćemo upotrijebiti naredbu:

```
bundle exec thin -p 4567 -R config.ru start
```

Upravo pokrenutu aplikaciju može se vidjeti na web-stranici `http://localhost:4567`. Odnosno na portu 4567 koji koristi TCP protokol (eng. *Transmission Control Protocol*), inače jedan od osnovnih protokola unutar IP grupe protokola. Specifičnost tog protokola je što nudi pouzdanu isporuku podataka u kontroliranom slijedu. Iz toga zaključujemo da će svi podaci poslani unutar aplikacije između pošiljatelja i primatelja biti sigurno isporučeni.

2.5 Usporedba s drugim aplikacijskim okvirima

U ovom poglavlju usporedit ćemo aplikacijski okvir Sinatru s nekim od najpoznatijih i najkorištenijih aplikacijskih okvira poput Ruby on Rails, Express, Django i Laravel.

Sinatra vs. Ruby on Rails

Prirodno je prvo usporediti Sinatru, koja je bazirana na programskom jeziku Ruby, s najpoznatijim aplikacijskim okvirom baziranim na Ruby-ju, a to je Ruby on Rails. Ruby on Rails je softver otvorenog izvornog koda koji je besplatan za korištenje te je moguće sudjelovati u njegovom poboljšavanju. Njegova arhitektura je zadano MVC te u skladu s tim potencira se korištenje ostalih dobro poznatih softverskih obrazaca kao na primjer DRY (eng. *don't repeat yourself*) kojemu je cilj izbjegavati ponavljanje bilo kojeg dijela koda te *active record* obrazac što je zapravo način rada s podacima u bazi podataka.

Kao prvu usporedbu Railsa sa Sinatrom možemo komentirati perolakost Sinatre u smislu jednostavnosti koda. Ako pogledamo Rails, on je prepunjen raznim funkcionalnostima te dolazi s velikom količinom koda, dok je kod Sinatre slučaj da je vrlo mala što se tiče količine koda. Kao posljedica toga, u Railsu je vrlo lako implemetirati aplikaciju u ograničenom vremenenu, ali zbog lakoće u količini koda takva aplikacija u Sinatri je ipak brža nego u Rails-u. Dalje, kada pogledamo strukturu aplikacije u Sinatri i u Rails-u,

možemo vidjeti da je u Rails-u aplikacija strogo strukturirana i zna se točno što se gdje nalazi. S druge strane, kod Sinatre to nije slučaj jer skoro ništa nije definirano te se svaka aplikacija može strukturirati na način koji nam u tom trenutku odgovara. S obzirom da su oba okvira bazirana na programskom jeziku Ruby, bilo bi neobično da nemaju ništa osim Ruby-ja zajedničko. Kao još jednu zajedničku karakteristiku možemo navesti da i Sinatra i Ruby on Rails u pozadini koriste Rack. Što je Rack objašnjeno je u potpoglavlju 2.4.

Iz navedenih sličnosti i razlika se može zaključiti da je Sinatra jako dobar izbor ako se odlučimo raditi manju aplikaciju. Dok ako želimo nekakavu kompliciraniju aplikaciju, iako ju se može implementirati i u Sinatri, ipak je bolji odabir Rails.

Sinatra vs. Padrino

Padrino je još jedan primjer aplikacijskog okvira koji je baziran na Ruby programskom jeziku, ali puno interesantnije je to što je Padrino okvir koji je baziran na Sinatri. Kreiran je kako bi proširio Sinatru da bi mogla podržavati kompliciranije web-aplikacije. Dobra karakteristika Padrina je to što se svaka od bitnijih komponenti može ili ukomponirati u postojeću Sinatra aplikaciju ili može zajedno s ostalim komponentama služiti kao nadogradnja Sinatre.

Već iz kratkog opisa Padrina vidimo sličnost sa Sinatrom jer Pardino je zapravo aplikacijski okvir koji je kreiran kako bi se na jednostavan način implementirala web-aplikacija pri čemu se pridržava karakteristika koje čine dobru stranu Sinatre. Kod odabira koji je okvir pogodan za koju aplikaciju treba se voditi time da je Sinatra uvijek dobar odabir kod aplikacija za koje bismo htjele da su lagane, u smislu količine koda. Dok je Padrino logičniji odabir za neke naprednije aplikacije.

Sinatra vs. Express

Express je aplikacijski okvir koji je pisan u JavaScript-u, besplatan je i ima otvoreni izvorni kod koji je pod licencom MIT-a. Jedan je od najpoznatijih okvira i smatra se standardnim okvirom za Node.js. Ono što je kod njega neobično je to što je inspiriran Sinatrom, točnije karakteristikama jezika Ruby. Kao osnovni opis Express-a navodi se njegova minimalnost i brzina, a osim toga je i fleksibilan te podržava REST API-je, odnosno aplikacijska programska sučelja koja su kreirana prema REST principu.

Usporedbu možemo započeti proučavanjem same implemetacije. Kod oba okvira je za početak rada s njima zahtjevati ih (eng. *require*) u fileovima gdje se koriste. Nadalje, kod Express okvira potrebno je dodijeliti varijable, funkcije i portove koji će slušati *request-response* upite, dok kod Sinatre ta dodjela nije potrebna. Zatim kao iduću veću razliku možemo navesti to da je Sinatra odlična za kreiranje malih aplikacijskih servisa, dok Express omogućava kreiranje aplikacija prilagodljivih na različite veličine ekrana te skalabilnih aplikacija. I za kraj usporedit ćemo način integracije baze podataka u oba od

navedenih okvira. Kod Expressa je spajanje na bazu podataka riješeno tako da se učitaju potrebni Node.js *driver*i za bazu podataka. Dok s druge strane, da bi se pri korištenju Sinatre povezali s bazom potebno je posebno ju dodati, odnosno specificirati na koju bazu se u kojem okruženju spaja.

Kao zaključak usporedbe možemo reći da ako želimo kreirati API za opću namjenu koji radi osnovne operacije s bazom podataka onda je Sinatra odlučan izbor za to. No, ako bismo htjeli skalabilnu aplikaciju koja treba operacije poput učitavanja dokumenata ili web-sockete, onda je ipak bolji izbor Express.

Sinatra vs. Django

Django je web-aplikacijski okvir čija osnova je programski jezik Python, a poštuje MTV (model-template-view) arhitekturu. Već smo upoznati sa *Model*om i *View*om, pa ćemo kratko objasniti što je *Template*. To je *text* dokument koji definira strukturu i izgled dokumenta, na primjer HTML stranice, s praznim mjestima koja predstavljaju sadržaj, a *View* dio aplikacije dinamički kreira samu HTML stranicu koriteći dani uzorak (eng. *template*).

Neke od karakteristika okvira Django su da je iznimno brz, odnosno kreiran je da se od koncepta do same aplikacije dođe što je brže moguće. Zatim, vrlo je siguran time što pomaže programerima izbjeći standardne sigurnosne greške te je izrazito skalabilan.

Zajednička osobina okvira Sinatra i Django je to što su oba namjenjena poslužiteljskoj odnosno *back-end* strani aplikacije. Kao razliku možemo navesti to što je Django po definiciji REST okvir koji je moćan i fleksibilan za jedonstavno kreiranje API-ja, dok je Sinatra DSL. Također kao još jednu od razlika možemo navesti situacije kada programeri odabiru jedan odnosno drugi okvir. Kada je cilj kreirati *browsable* API-je koji omogućuju da ih koristimo bez nekog dodatnog proučavanja dokumentacije, onda je bolji odabir Django, ali ako želite kreirati laku aplikaciju, u smislu koda, onda je ipak bolji izbor Sinatra.

Poglavlje 3

Studijski primjer - Cijepljenje

Kako bi se sve do sad opisano pokazalo u praksi, implementirali smo web-aplikaciju, naziva Cijepljenje, u aplikacijskom okviru Sinatra. U ovom poglavlju ćemo prvo detaljnije opisati što aplikacija radi. Nakon toga, ćemo objasniti MVC dizajn aplikacije tako da prvo opišemo modele korištene u aplikaciji, a zatim i kontrolere. Poglede (eng. *view*) odnosno sučelja aplikacije ćemo pokazati prema dijelu aplikacije kojem pripada. Opisat ćemo i kako se razvoj aplikacije temeljio na principu razvoj aplikacije vođen testovima (eng. *Test-Driven Development*), te će na kraju biti objašnjeni neki od alata i metoda korištenih za razvoj aplikacije Cijepljenje čiji cjelokupan kod se može vidjeti na [2].

3.1 Opis aplikacije

Aplikacija je nastala potaknuta trenutnom situacijom s cijepljenjem protiv COVID-19. Primarni cilj aplikacije je prijavljivanje građana na cijepljenje i to na tri različita načina. Na cijepljenje se može prijaviti svaki stanovnik Republike Hrvatske popunjavajući formu za prijavu, može ga prijaviti njegov liječnik te ako je dio neke od trenutno prioritetnih skupina, može ga prijaviti predstavnik te prioritetne skupine. Spomenuti liječnik i predstavnik prioritetnih skupina, kao i administrator za korištenje aplikacije, trebaju obaviti prijavu kako bi ih se autentificiralo.

Liječniku aplikacija, osim prijave pacijenat na cijepljenje, omogućuje dodjeljivanje termina zahtjevima koji su u statusu čekanja termina cijepljenja i nisu do sada primili niti jednu dozu cjepiva. Osim toga liječnik može provjeriti fazu cijepljenja za pojedinog pacijenta te promijeniti neke od podataka. U sustavu postoji i korisnik koji je cijepitelj, odnosno on je osoba koja obavlja cijepljenje. On ima uvid u sva cijepljenja koja trebaju biti obavljena unutar njegovog radnog vremena te radi unos podataka o obavljenom cijepljenju. Aplikacijom upravlja administrator koji ima ovlasti dodavanja liječnika, predstavnika prioritetnih skupina, cijepitelja, novog cjepiva te mjesta na kojima se sve obavlja cijepljenje.

3.2 MVC arhitektura

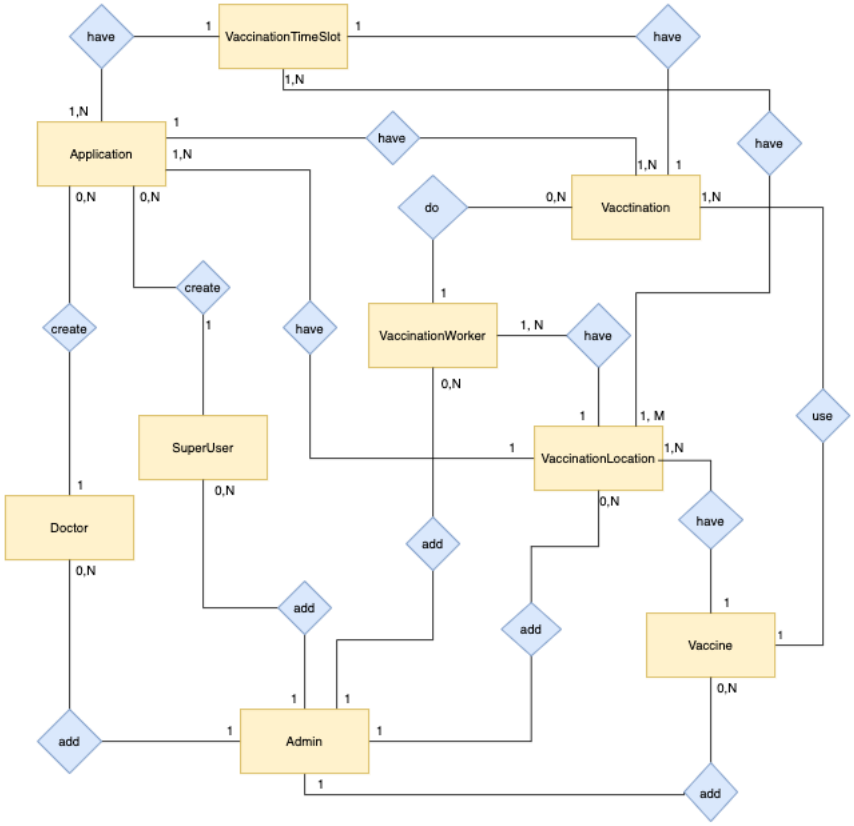
Modeli

U potpoglavlju 1.2. smo objasnili ukratko kakva je to MVC arhitektura okvira za web-aplikacije i usporedili ju sa troslojnom arhitekturom. Ovo je prvo od niza poglavlja u kojima ćemo razraditi opis MVC arhitekture na primjeru aplikacije Cijepljenje koja je kreirana u aplikacijskom okviru Sinatra. Modeli koje smo koristili u studijskom primjeru nabrojani su i opisani u sljedećoj tablici:

Model	Opis
admin	Administrator koji ima mogućnost dodavanja korisnika aplikacije, cijepiva i mjesta cijepljenja.
doctor	Liječnik koji ima mogućnost slanja prijave za cijepljenje, rješavanje zahtjeva u obradi, pregledavanje i mijenjanje podataka u zahtjevima.
superuser	Predstavnik prioritetne skupine koji ima mogućnost prijaviti za cijepljenje veći broj ljudi tako da im se odmah pridodijeli najveća razina prioriteta za dodjelu termina.
vaccination worker	Osoba koja evidentira obavljeno cijepljenje i ima pravo vidjeti sva cijepljenja koja se trebaju obaviti u toku njegovog radnog vremena.
vaccine	Cjepivo koje se trenutno može koristiti.
vaccination location	Mjesta na kojima se može obavljati cijepljenje.
vaccination time slot	Datum i vrijeme kada se može obaviti cijepljenje.
application	Zahtjev za cijepljenjem koji sadrži sve potrebne podatke o pacijentu koji se želi cijepiti.
vaccination	Obavljeno cijepljenje, a objekte ovog modela kreira cijepitelj nakon odrađenog cijepljenja.
location and time slot	Datum i vrijeme cijepljenja povezano s mjestom cijepljenja.

Tablica 3.1: Tablica modela u aplikaciji Cijepljenje

Nabrojani modeli su dobiveni iz ER (eng. *Entity Relation*) dijagrama u kojemu smo vidjeli koje sve objekte ćemo trebati promatrati i u kakvim relacijama će biti. Na sljedećoj slici je prikazan spomenuti dijagram u kojemu možemo vidjeti odnose entiteta aplikacije Cijepljenje koji su navedeni kao modeli u tablici 3.1.



Slika 3.1: Dijagram odnosa modela

Kako bismo relacije iz ER dijagrama definirali i u bazi, morali smo dodati *gem*ove `activerecord` i `sinatra-activerecord` koji omogućuju kreiranje migracija pomoću kojih se kreiraju potrebne tablice i veze u bazi podataka. Pokazat ćemo na primjeru modela liječnika kako izgleda dokument s migracijom te što se dogodi nakon što tu migraciju i izvršimo. Da bismo kreirali novi migracijski dokument potrebno je u terminalu izvršiti naredbu:

```
bundle exec rake db:create_migration NAME=create_doctors
```

Nakon što smo kreirali novi dokument, popunimo ga sa stupcima koje želimo da tablica *doctors* sadrži, odnosno navedemo atribute i tipove za te atribute koje bi model *doctors* trebao imati. Kako je to izvedeno u aplikaciji Cijepljenje, prikazano je u sljedećem isječku koda.

```
# 20210610084807_create_doctor
class CreateDoctor < ActiveRecord::Migration[6.1]
  def change
    create_table :doctors do |t|
      t.string :email, null: false
      t.string :password_digest, null: false
      t.belongs_to :admin, foreign_key: true
      t.string :first_name, null: false
      t.string :last_name, null: false
      t.timestamps
    end
  end
end
```

Sljedeće što se treba napraviti je pokrenuti kreiranu migraciju kako bismo tablice dodali u bazu podataka te tako izmijenili `schema.rb` dokument koji sadrži schemu baze podataka prema migracijama koje smo izvršili. Na sljedeća dva isječka koda ćemo vidjeti primjer `shcema.rb` dokumenta i naredbu s kojom smo dobili takav izgled dokumenta.

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

```
# schema.rb
ActiveRecord::Schema.define(version: 2021_06_10_084807) do
  create_table "admins", force: :cascade do |t|
    t.string "email", null: false
    t.string "password_digest", null: false
    t.datetime "created_at", null: false
    t.datetime "updated_at", null: false
```

```
8   end
9
10  create_table "doctors", force: :cascade do |t|
11    t.string "email", null: false
12    t.string "password_digest", null: false
13    t.bigint "admin_id"
14    t.string "first_name", null: false
15    t.string "last_name", null: false
16    t.datetime "created_at", null: false
17    t.datetime "updated_at", null: false
18    t.index ["admin_id"], name: "index_doctors_on_admin_id"
19  end
20
21  add_foreign_key "doctors", "admins"
22 end
```

Zbog modela koje smo kreirali, morali smo dodati jos dva *gem*a. Prvi od njih je *gem* `activesupport` koji se koristio kako bismo na jednostavniji način dodali validacije za atribute koji su jednaki u pojedinim modelima. Drugi *gem* je `bcrypt`, jer postoje korisnici aplikacije koji za korištenje pojedinih funkcionalnosti trebaju unijeti email i lozinku, a lozinke je potrebno zaštiti od moguće krađe podataka. Taj *gem* omogućava sigurno spremanje povjerljivih podataka u bazu i to tako da ih *hash*ira.

Kontroleri

Ranije smo već opisali koja je uloga kontrolera, ali kod kreiranja modularne aplikacije, kao što je Cijepljenje, u Sinatri, kontroleri imaju ulogu `routes.rb` dokumenta iz aplikacijskog okvira Rails. Odnosno, u samim kontrolerima moramo definirati API pozive koji će biti dostupni u aplikaciji. U aplikaciji se pojavljuju kontorleri s različitim ulogama.

- Glavni kontroler iz kojeg se pozivaju svi ostali:
  - *AppController*
- Kontroleri koji se koriste za autentifikaciju korisnika:
  - *AdminsSecurityController*
  - *DoctorsSecurityController*
  - *SuperUsersSecurityController*
  - *VaccinationWorkersSecurityController*

- Ostali kontroleri:
  - *AdminsController*
  - *DoctorsController*
  - *SuperUsersController*
  - *VaccinationWorkersController*
  - *ApplicationsController*
  - *PassportsController*
  - *VaccinationLocationsController*
  - *VaccinationTimeSlotsController*
  - *VaccinationsController*
  - *VaccinesController*

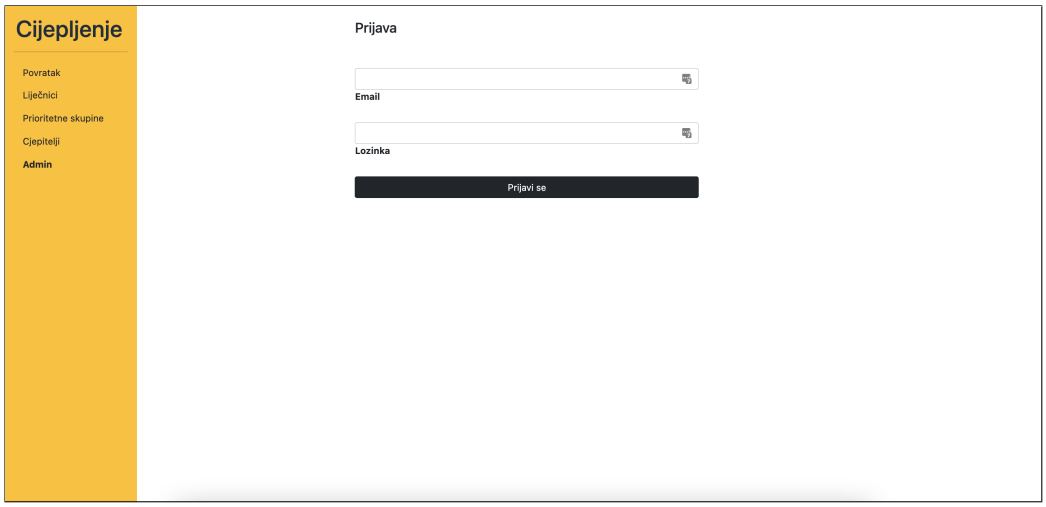
Zbog korisnika koji za korištenje funkcionalnosti trebaju prijavu, nekako se moralo osigurati postojanje *sessiona*. Kod razvoja aplikacije u okviru Sinatra to se može implementirati tako da se u glavnom kontroleru, što je u ovom slučaju *AppController*, doda naredba `enable:sessions`, te kod prijave nekog od korisnika, zabilježimo ga u polje `session`. Kako je to implementirano može se vidjeti u isječku koda kod prijave i odjave administratora aplikacije.

```ruby
#admins_security_controller.rb
class AdminsSecurityController < AppController
  set :views, File.expand_path('../../views', __dir__)

  get '/admin/login' do
    @title = 'Cijepljenje'
    erb :'admin/login'
  end

  post '/admin/login' do
    begin
      admin = Admin.find_by(email: params[:email])
    rescue ActiveRecord::RecordNotFound => e
      e.message
    else
      if admin&.authenticate(params[:password]) && !logged_in_admin?
        session[:admin_id] = admin.id
```

```
18          redirect '/admin'
19        else
20          @error = true
21          flash[:warning] = SchemeMain::ALERT_MESSAGE[:somebody_already_logged_id]
22          redirect '/admin/login'
23        end
24      end
25    end
26
27    get '/admin/logout' do
28      session[:admin_id] = nil
29      redirect '/admin/login'
30    end
31  end
```

3.3 Sučelje administratora

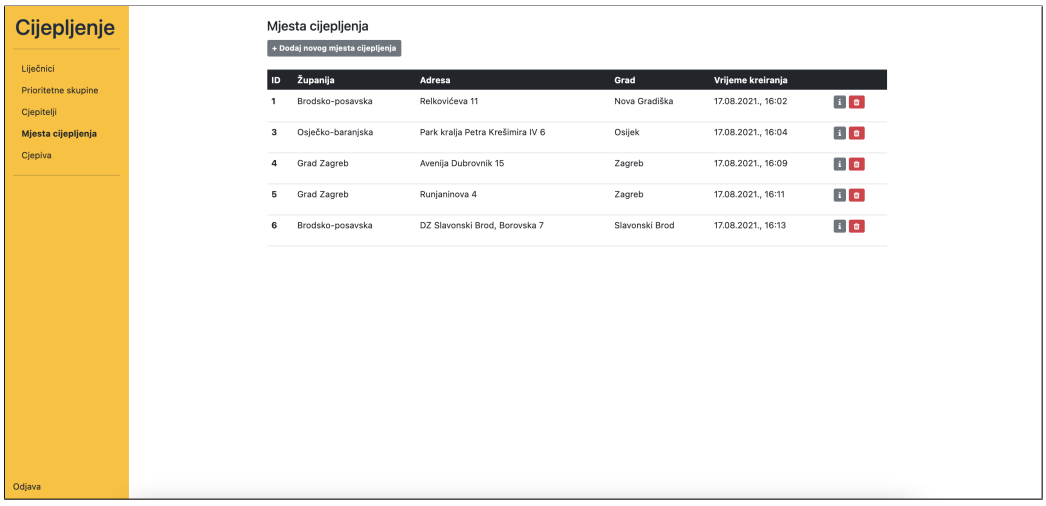
Administrator je bitan korisnik aplikacije jer on brine za funkcioniranje aplikacije, a kako bi došao do tih funkcionalnosti on se mora prvo prijaviti unoseći pripadni email i lozinku u formu za prijavu koja se može vidjeti na sljedećoj slici.



Slika 3.2: Stranica za prijavu administratora aplikacije

Oblikovanje prikazane forme, a i svake od sljedećih izvedeno je korištenjem HTML.ERB dokumenta zajedno s mogućnostima Bootstrap-a. U isječku koda može se vidjeti implementacija prikazane forme, točnije *partial*a koji sadrži formu, dok su ostale forme implementirane na vrlo sličan način.

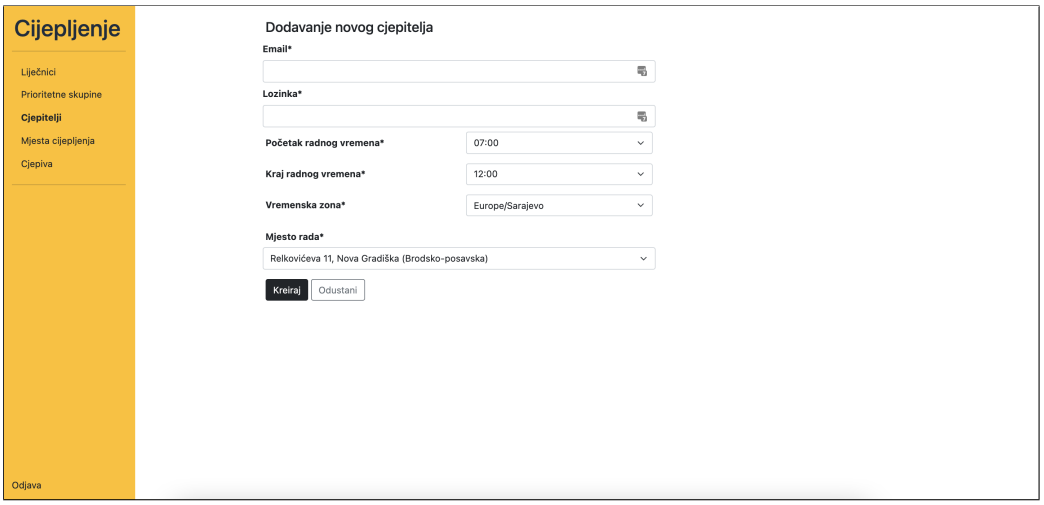
```
#login_form.rb - partial
<div class="form-outline mb-4">
<input type="email" id="email" name="email" class="form-control" required/>
<label class="form-label" for="email">Email</label>
</div>

<div class="form-outline mb-4">
<input type="password" id="password" name="password"
       class="form-control" required/>
<label class="form-label" for="password">Lozinka</label>
</div>

<button type="submit" class="btn btn-dark btn-block">Prijavi se</button>
```



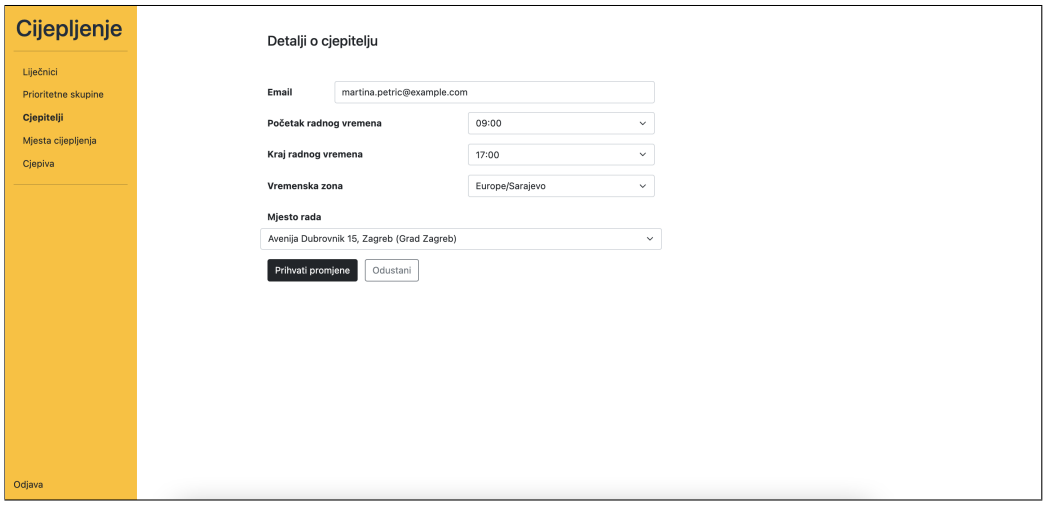
Slika 3.3: Tablica svih mjesta cijepljanje

Nakon prijave, administrator klikom na određenu skupinu objekata može vidjeti popis svih do sad dodanih liječnika, predstavnika prioritetnih skupina, cjepitelja, mjesta cijepljenja i

cjepiva. Na slici 3.3 se može vidjeti kako izgleda stranica sa popisom svih mjesta cijepljenja, vrlo slično njoj izgledaju i ostale stranice sa popisom svih objekata.



Slika 3.4: Dodavanje novog cjepitelja

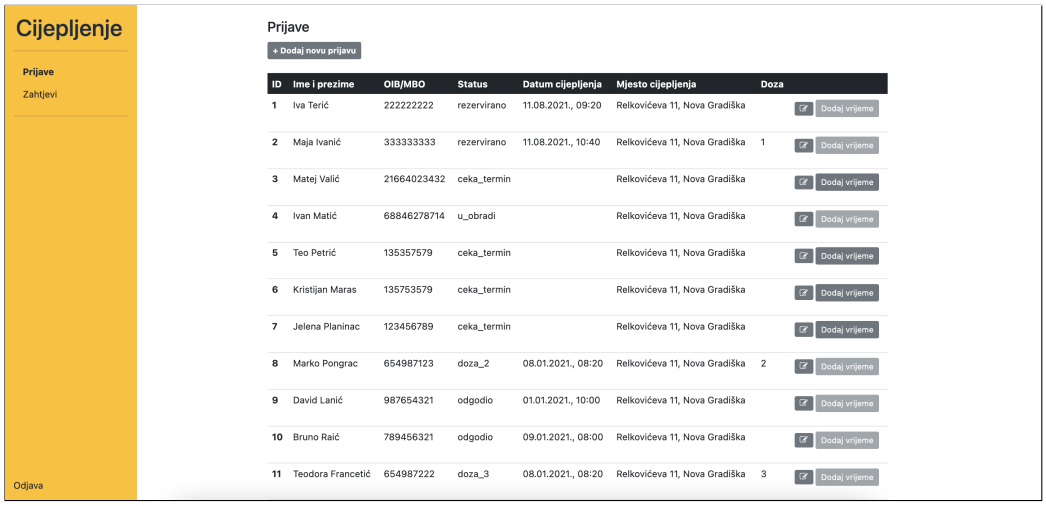


Slika 3.5: Prikaz detalja o cjepitelju

Administrator ima i mogućnost dodavanja novih objekata te pregled i uređivanje podataka o pojedinom objektu. Sučelje za ove funkcionalnosti vidljivo je na slikama 3.4 i 3.5 koje prikazuju dodavanje i pregled podataka o cjepitelju.

3.4 Sučelje liječnika

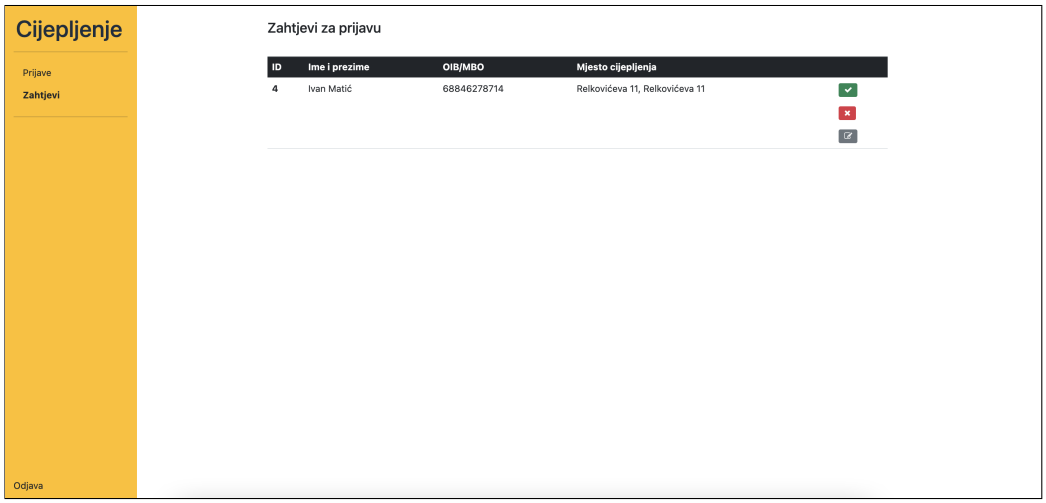
Liječnik za korištenje svojih funkcionalnosti treba biti prijavljen kao i administrator te sučelje za prijavu izgleda jednako kao na slici 3.2. Nakon prijave, liječnik dobije popis svih do sad poslanih prijava za cijepljenje, a sučelje za tu funkcionalnost je prikazano na slici 3.6.



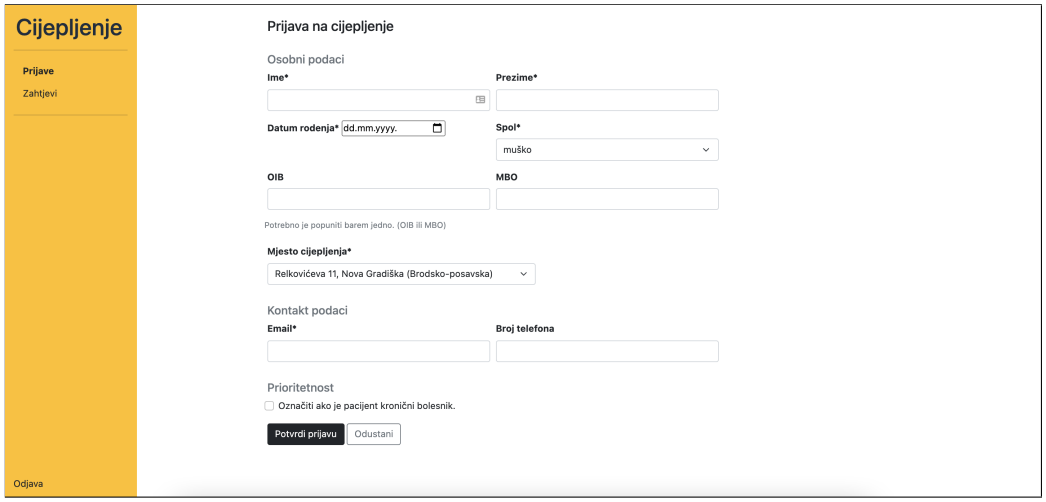
Slika 3.6: Popis svih zahtjeva

Osim popisa svih zahtjeva, liječnik može vidjeti i popis svih prijava koje čekaju na njegovu potvrdu. Naime, ako građanin ili predstavnik prioritetne skupine pošalje zahtjev za cijepljenje i pri tome označi da je osoba koja je prijavljena kronični bolesnik, onda taj zahtjev treba potvrdu liječnika prije nego uđe u proces cijepljenja. Na slici 3.7 ćemo vidjeti liječnikovo sučelje sa popisom svih zahtjeva koji čekaju potvrdu.

Kao jedan od korisnika aplikacije, liječnik ima mogućnost slanja zahtjeva za cijepljenje, pri tome svi zahtjevi koje pošalje liječnik odmah preskaču status čekanja potvrde u slučaju ako je označeno da je osoba kronični bolesnik. Kako izgleda forma za prijavu pacijenta na cijepljenje može se vidjeti na slici 3.8.

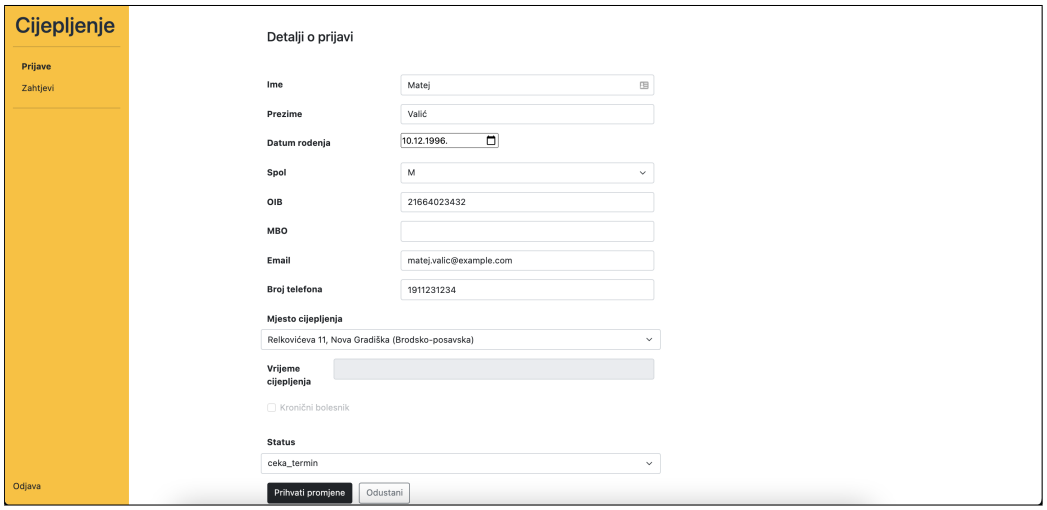


Slika 3.7: Popis svih zahtjeva koji čekaju odobrenje

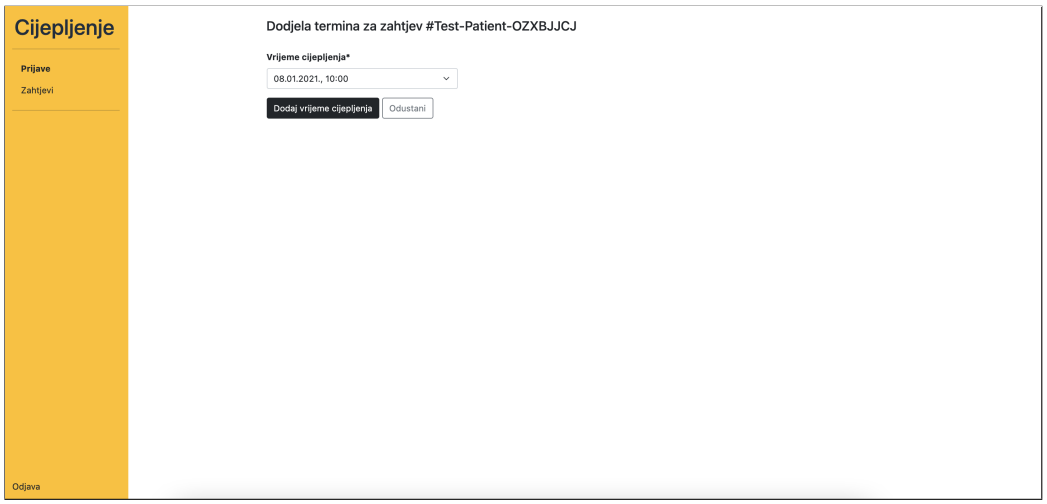


Slika 3.8: Liječnikova forma za slanje zahtjeva za cijepljenje

Osim navedenih funkcionalnosti, liječnik ima i mogućnost pregleda podataka o pojedinom zahtjevu te ako je zahtjev u statusu čekanja termina i osoba nije primila niti jednu dozu cjepiva, ima mogućnost odabrati jedan od slobodnih termina cijepljenja i dodijeliti ga tom zahtjevu.



Slika 3.9: Prikaz detalja o zahtjevu za cijepljenje



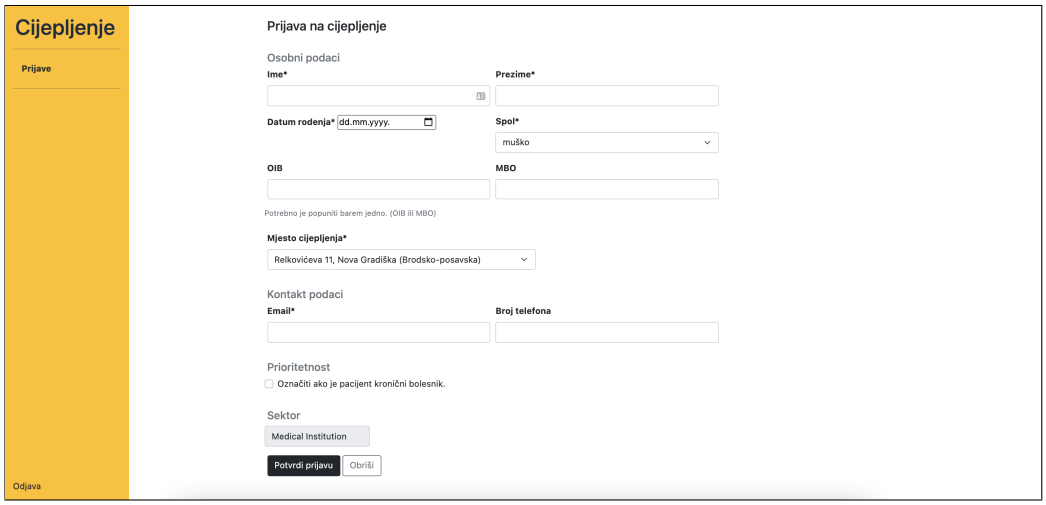
Slika 3.10: Dodjeljivanje termina cijepljenja

U sljedećem isječku koda može se vidjeti kontroler i API poziv pomoću kojeg je definirano dohvaćanje slobodnih termina koje liječnik može dodijeliti ovisno o lokaciji i zahtjevu.

```ruby
#vaccination_time_slots_controller.rb
class VaccinationTimeSlotsController < AppController
  include ControllerHelper

  set :views, File.expand_path('../views', __dir__)

  get '/doctor/vaccination_time_slots/:location_id' do
    redirect_if_not_logged_in?(:doctor_id)

    @title = 'Cijepljenje'
    if VaccinationLocation.find_by(id: params[:location_id]).nil?
        @message = SchemeMain::ERROR_MESSAGES_VACCINATION_LOCATION
        [:vaccination_location_does_not_exist]
    end
    return erb :'/errors/404_doctor' if @message

    @available_time_slots = fetch_available_vaccination_time_slots(
                            params[:location_id])
    @application = Application.find_by(id: params[:application_id])
    @message = SchemeMain::ERROR_MESSAGES_APPLICATION
    [:application_does_not_exist] if @application.nil?
    return erb :'/errors/404_doctor' if @message

    erb :'/application/doctor_side/vaccination_time_slots'
  end
end
```

3.5 Sučelje predstavnika prioritetne skupine

Predstavnik prioritetnih skupina, koji je u implementaciji nazvan *superuser* jer ima drukčije mogućnosti od običnog korisnika, nakon prijave dobije formu za slanje zahtjeva za cijepljenje. Ono što je kod tog tipa korisnika posebno je to da svakom zahtjevu koji on pošalje kao sektor rada se pridodijeli njegov sektor. Zbog toga zahtjev koji on pošalje ulazi u one zahtjeve koji kod automatske dodjele termina cijepljenja dobivaju najveći prioritet. Implementaciju automatske dodjele termina cijepljenja ćemo objasniti u potpoglavlju 3.9.



Slika 3.11: Sučelje aplikacije za predstavnika prioritetne skupine

Kako je implementirano kreiranje novog zahtjeva za cijepljenje od strane predstavnika prioritetne skupine može se vidjeti u sljedećem isječku koda. Pozivanjem funkcije iz prvog retka osigurava se da je predstavnik prioritetne skupine zaista prijavljen u aplikaciji, a nakon kreiranja ako je zahtjevu dodijeljen status da čeka termin, pacijentu se pošalje email s identifikacijskim brojem zahtjeva.

```
1  #appplication_controller.rb
2  post '/super_user/applications' do
3      redirect_if_not_logged_in?(:super_user_id)
4
5      unless check_params.nil?
6        flash[:danger] = check_params
7        redirect "/super_user/applications/new"
8      end
9
10     begin
11       application = Application.create!(
12         first_name: first_name,
13         last_name: last_name,
14         birth_date: birth_date,
15         gender: gender,
16         oib: oib,
```

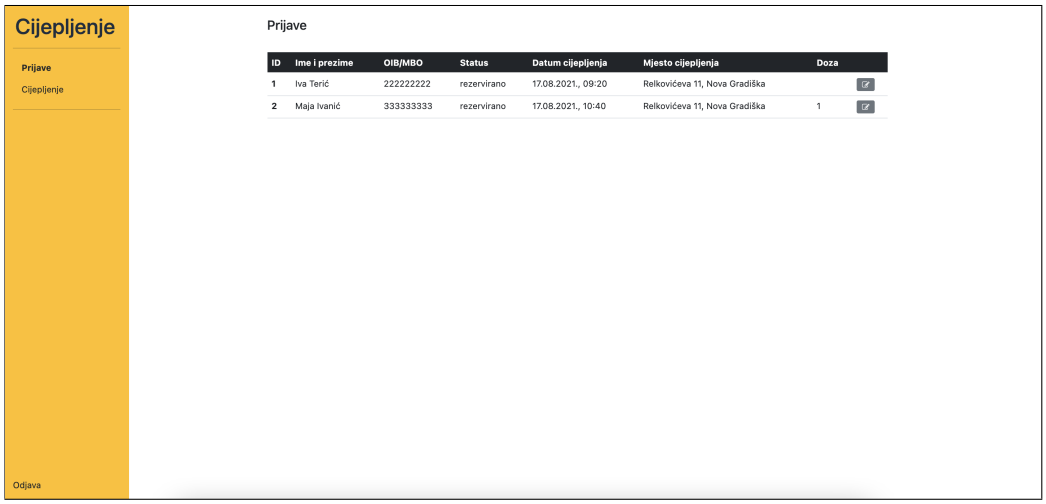
```
17          mbo: mbo,
18          email: email,
19          sector: sector,
20          phone_number: phone_number,
21          chronic_patient: chronic_patient,
22          status: application_status(),
23          vaccination_location_id: params[:vaccination_location_id],
24          author_id: params[:author_id],
25          author_type: params[:author_type],
26          reference: reference
27        )
28      rescue ActiveRecord::RecordInvalid => e
29        flash[:danger] = SchemeMain::ALERT_MESSAGE[:unsuccessfull]
30        redirect '/super_user/applications/new'
31      else
32        flash[:success] = SchemeMain::ALERT_MESSAGE[:successfull]
33        if application_status() == Application.statuses[:ceka_termin]
34          send_email_with_application_id(application.email, message_body(application))
35        end
36        redirect 'super_user/applications/new'
37      end
38  end
```

3.6 Sučelje cjepitelja

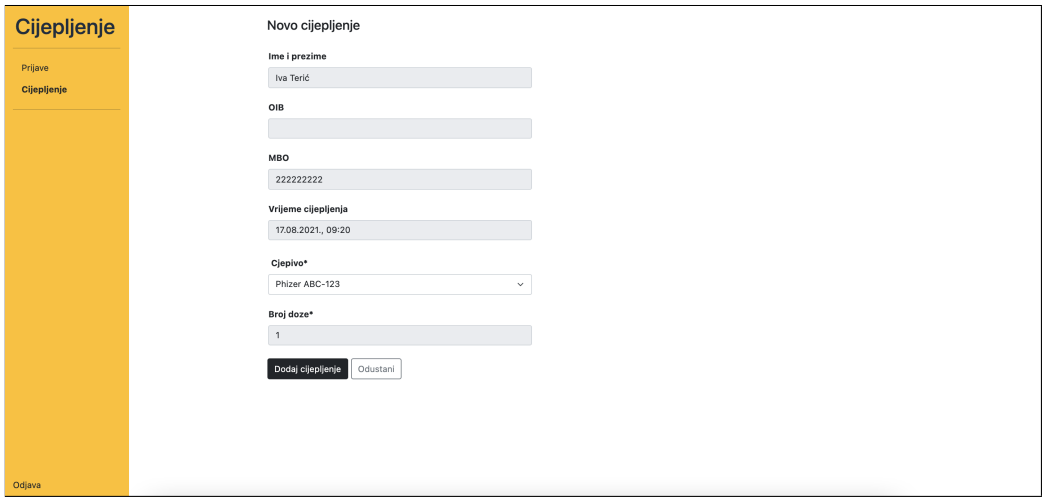
Kao što je već navedeno, cjepitelji su također korisnici aplikacije koji se trebaju prijaviti da bi im bile dostupne njihove funkcionalnosti, a stranica za prijavu izgleda jednako kao i kod administratora, liječnika i predstavnika prioritetne skupine. Njihova uloga je da unose informacije o obavljenom cijepljenju u bazu podataka, odnosno moraju kreirati novi objekt *vaccination*.

Nakon što se prijavio, cjepitelj vidi popis svi zahtjeva za cijepljenje koji se trebaju obaviti tokom tog dana kada se prijavio unutar njegovog radnog vremena. Sučelje u kojem je prikazan popis svih takvih zahtjeva vidi se na slici 3.12.

Cjepitelj nakon što klikne na neku od prijava, vidi detalje o toj prijavi, ali ima i mogućnost dodati novo cijepljenje. Nakon što je novo cijepljenje dodano, prijava se miče sa popisa cijepljenja koja se trebaju obaviti taj dan. Na slici 3.13 je prikazan izgled forme za dodavanje upravo obavljenog cijepljenja za pojedinu prijavu.

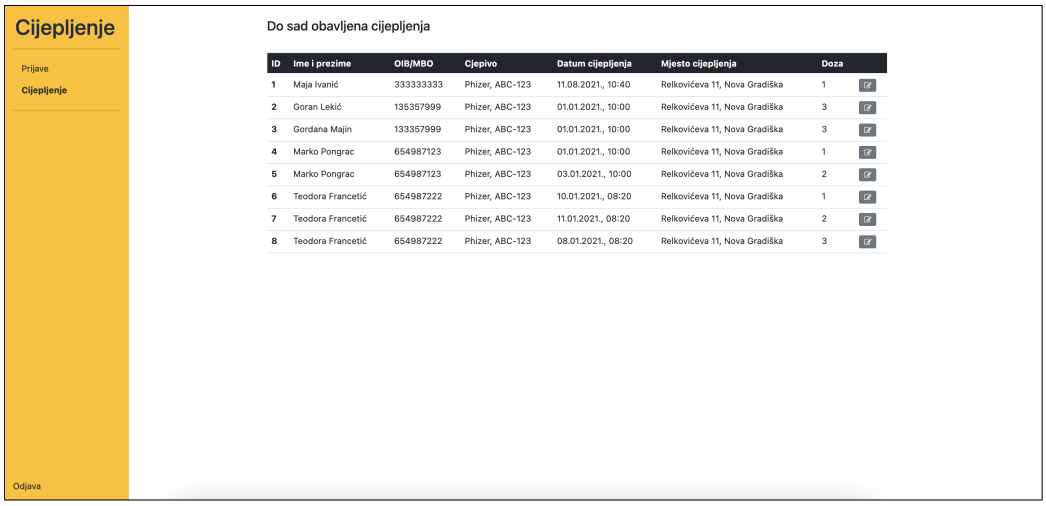


Slika 3.12: Popis zahtjeva s vremenom cijepljenja tijekom radnog vremena cjepitelja



Slika 3.13: Dodavanje novog obavljenog cijepljenja

Osim tih funkcionalnosti, cjepitelj ima mogućnost vidjeti sva cijepljenja koja je on do sada obavio, kao što je prikazano na slici 3.14. Klikom na pojedino cijepljenje može vidjeti detalje o njemu i neke od njih promijeniti.



Slika 3.14: Popis cijepljenja koja je obavio prijavljeni cjepitelj

3.7 Javno sučelje

Osim dijelova aplikacije za korisnike koji za pristup određenim funkcionalnostima trebaju obaviti prijavu, postoji i dio aplikacije za koji nije potrebna prijava i zovemo ga javni dio. Javni dio aplikacije Cijepljenje omogućava svakom građaninu da se prijavi na cijepljenje. Svaki zahtjev poslan na taj način, a u slučaju da je osoba označila da je kronični bolesnik, odlazi u obradu i čeka potvrdu liječnika kako bi se proces cijepljenja započeo, inače odmah odlazi u status čekanja termina. Forma za prijavu je jako slična ranije prikazanim pa ju nećemo prikazivati.

Nakon što zahtjev dođe u status čekanja termina, građanin dobije email s porukom da je zahtjev prihvaćen i jedinstveni identifikacijski broj zahtjeva, odnosno njegovu referencu koja se generira prilikom kreiranja zahtjeva u bazi. Za samo slanje email-ova dodan je *gem* `pony` koji omogućuje jednostavno slanje email-ova pomoću samo jedne naredbe.

Građanin, odnosno pacijent, u svakom trenutku ima pravo zatražiti COVID putovnicu. U aplikaciji je to implementirano tako da građanin unese identifikacijski broj, koji je dobio u potvrdi o primljenom zahtjevu, u formu za kreiranje COVID putovnice, koju se može vidjeti na slici 3.15. Ako je osoba primila sve potrebne doze, putovnica će se generirati kao QR code, a za to nam je poslužio *gem* `rqrcode` i metode definirane u [5]. Dok u slučaju da nisu sve doze primljene, aplikacija vrati obavijest da proces cijepljenja nije obavljen u potpunosti.

Slika 3.15: Forma za unos identifikacijskog broja zahtjeva

Slika 3.16: Izgenerirani QR code za pripadni zahtjev

3.8 Testiranje - RSpec

RSpec je DSL alat napisan u programskom jeziku Ruby koji se koristi za testiranje koda napisanog također u Ruby-ju. To je alat za razvoj vođen ponašanjem (eng. *behavior-driven development*) koji se koristi u produkcijskim aplikacijama. Primjenjuje se u programiranju vođenom testiranjem (eng. *test-driven development* - TDD) kojemu je osnovna značajka to

da se prilikom implementacije neke aplikacije prvo pišu testovi, a zatim samo toliko koda koliko je dovoljno da napisani testovi prođu.

Postoje tri metode koje daju formu testnim dokumentima odnosno samim testovima, a prva od njih je `describe()`. Ona opisuje klasu, metodu ili grupu primjera, odnosno to je vanjski blok koji sadrži testni kod. Sljedeća metoda je `context()` koja služi kako bi se opisao kontekst u kojem je korištena klasa ili metoda opisana u *describe* bloku. I zadnja od metoda je `it()` i ona služi za opisivanje specifikacije uzorka u kontekstu.

RSpec ima nekoliko vrsta testova, a nama su najinteresantiji *model specs* i *request specs* čije primjene ćemo vidjeti u sljedećim potpoglavljima.

Testiranje modela

Model specs su testovi kojima su u studijskom primjeru testirani ponašanje i svojstva pojedinog modela. U isječku koda ćemo vidjeti test kojim su testirane relacije i validacije za model *Vaccination*.

```ruby
#vaccination_spec.rb
require 'spec_helper'

RSpec.describe Vaccination, type: :model do
  describe 'associations' do
    it { is_expected.to belong_to(:application) }
    it { is_expected.to belong_to(:vaccine) }
    it { is_expected.to belong_to(:vaccination_time_slot) }
    it { is_expected.to belong_to(:vaccination_worker) }
  end

  describe 'validations are correct' do
    let(:vaccination_worker) { create(:vaccination_worker) }
    let(:vaccination_time_slot) { create(:vaccination_time_slot) }
    let(:vaccine) { create(:vaccine) }
    let(:application) { create(:application_with_mbo) }

    subject {
      described_class.create(
        application: application,
        vaccination_time_slot: vaccination_time_slot,
        vaccine: vaccine,
        vaccination_worker: vaccination_worker,
```

```
24          dose_number: 1
25        )
26      }
27
28      it { is_expected.to validate_presence_of(:application_id) }
29      it { is_expected.to validate_presence_of(:vaccination_time_slot_id) }
30      it { is_expected.to validate_presence_of(:vaccine_id) }
31      it { is_expected.to validate_presence_of(:vaccination_worker_id) }
32      it { is_expected.to validate_presence_of(:dose_number) }
33    end
34  end
```

Request testovi

Request specs su testovi koji služe da bi se testiralo što aplikacija radi, ali ne i kako radi. U primjeru aplikacije Cijepljenje ta vrsta testova korištena je za testiranje API poziva definiranih u svakom od kontrolera. Jedan *request* test ćemo vidjeti u primjeru testiranja dohvaćanja svih mjesta cijepljenja od strane administratora.

```
1   #vaccination_location_controller_spec.rb
2   require 'spec_helper'
3
4   RSpec.describe 'VaccinationLocationsController' do
5     include Rack::Test::Methods
6
7     def app
8       VaccinationLocationsController.new
9     end
10
11    let(:admin) { create(:admin) }
12    let(:vaccination_location) { create(:vaccination_location) }
13    let(:id) { vaccination_location.id.to_s }
14    let(:session_params) { { 'rack.session' => { 'admin_id' => admin.id } } }
15    before do
16      admin
17      vaccination_location
18    end
19
20    describe 'GET /admin/vaccination_locations' do
```

```
21      context 'when admin is logged in' do
22        context 'when try to get all vaccination_locations' do
23          let(:response) { get '/admin/vaccination_locations', {}, session_params }
24          before do
25            response
26          end
27
28          it 'returns view with all vaccination locations' do
29            expect(last_response.body.include?('Mjesta cijepljenja')).to eq true
30          end
31
32          it 'returns status 200 OK' do
33            expect(last_response.status).to eq 200
34          end
35        end
36      end
37
38      context 'when admin is not logged in' do
39        context 'when try to get all vaccination locations' do
40          let(:response) { get '/admin/vaccination_locations' }
41
42          before do
43            response
44          end
45
46          it 'returns status 302 and redirect' do
47            expect(last_response.location.split('//').last).to eq 'example.org/admin/login'
48            expect(last_response.status).to eq 302
49          end
50        end
51      end
52    end
53  end
```

3.9 Korišteni alati i metode

Kao što se može vidjeti iz prethodnih poglavlja, studijski primjer je poprilično kompleksna aplikacija stoga je za njezin razvoj bio potreban cijeli niz alata i metoda. Neke bitnije

ćemo ukratko opisati u sljedećim potpoglavljima kako bi se dobio dojam koliki skup alata i metoda je potreban za razvoj web-aplikacije slične Cijepljenju u aplikacijskom okviru Sinatra.

Rake

Rake je popularan pokretač zadataka u programskom jeziku Ruby. Zadatak koji on pokreće može biti izrada rezervne kopije baze podataka, pokretanje testova i slično. Prednost u pokretanju takvih zadataka pomoću Rake-a za razliku od pokretanja kao manje metode jer u tome što su ovako svi zadaci definirani na jednom mjestu, a ne u puno manjih dokumenata. Već smo pokazali primjere Rake zadataka prilikom kreiranja baze podataka i izvođenja migracija u poglavlju 3.2.

Kako aplikacija Cijepljenje kao jednu od funkcionalnosti ima dodjeljivanje termina cijepljenja zahtjevima koji cekaju termin, tako se našla dobra prilika za iskoristiti Rake zadatke. U samoj aplikaciji su kreirana tri Rake zadatka koji bi se trebali pokretati svaki dan u 20 sati navečer točno određenim redom. Prvo se pokrene zadatak koji provjeri koji od zahtjeva za cijepljenje je u potpunosti završio proces cijepljenja i kako bi mu dodijelio status da je gotov. Nakon toga se izvršava zadatak koji provjeri ima li novih cjepiva, ako ima onda kreira termine cijepljenja za pripadno cjepivo i mjesto cijepljenja. I kao zadnji se izvrši zadatak dodjele termina koji dodjeli termine prema ranije definiranim prioritetima koji se kroz vrijeme mogu mijenjati. U sljedećem isječku koda se može vidjeti kako se definira jedan Rake zadatak i to na primjeru dodjeljivanja slobodnih termina zahtjeva te neke od korištenih metoda koje su implemetirane u Ruby-ju. Naravno, postoji jos niz pomoćnih metoda koje su bile potrebne, ali radi količine i sličnosti koda nisu prikazane ili su prikazane kao primjeri u drugim poglavljima.

```
1  #scheduler.rake
2  namespace :scheduler do
3    desc 'Assign vaccination time slot to applications that waits for one'
4    task :assign_vaccination_time_slot_to_application do
5      # 1. dohvati sve aplikacije sa statusom 'odgodio'
6      applications_with_status_postpone = get_applications_with_status_postpone()
7      unless applications_with_status_postpone.empty?
8          assign_vaccination_time_slot_for_postpone(
9              applications_with_status_postpone)
10     end
11
12     # 2. dohvati sve aplikacije sa statusom 'ceka termin'
13     # i s barem jednom primljenom dozom
```

```
14      applications_that_needs_another_dose =
15          get_applications_that_needs_another_dose()
16      unless applications_that_needs_another_dose.include?(nil)
17          assign_vaccination_time_slot_for_another_dose(
18              applications_that_needs_another_dose)
19      end
20
21      # 3. dohvati sve aplikacije sa statusom 'ceka termin'
22      # i niti jednom primljenom dozom cjepiva
23      applications_with_status_wait_time_slot =
24          get_applications_with_status_wait_time_slot()
25      unless applications_with_status_wait_time_slot.empty?
26          assign_vaccination_time_slot_by_priorities(
27              applications_with_status_wait_time_slot)
28      end
29    end
30  end
```

```
1   # assign_vaccination_time_slot_to_application_helper.rb
2   def assign_vaccination_time_slot_for_postpone(applications)
3       # dodjeljuje se termin barem tjedan dana od starog termina
4       applications.each do |application|
5         # promijenit status iz 'odgodio' u 'ceka termin'
6         application.needs_to_continue_vaccination
7         location = application.vaccination_location
8         vaccination = Vaccination.find_by(application_id: application.id)
9         vaccine = vaccination.vaccine unless vaccination.nil?
10
11        # pronademo termin
12        old_time_slot = VaccinationTimeSlot.find_by(
13          id: application.location_and_time_slot.vaccination_time_slot_id)
14
15        location_and_time_slot_to_assign =
16          find_available_time_slot(vaccine, location, old_time_slot).first
17
18        if !location_and_time_slot_to_assign.nil?
19          assign(application, location_and_time_slot_to_assign)
20        else
```

```
21          # provjerimo slobodne termine za iducih 5 dana
22          location_and_time_slot_to_assign =
23              LocationAndTimeSlotRepository.available_time_slot_in_next_five_days(
24                  vaccine, location, old_time_slot.date_and_time.change(hour: 8),
25                  old_time_slot.date_and_time.change(hour: 19, min: 40))
26          unless location_and_time_slot_to_assign.empty?
27            assign(application, location_and_time_slot_to_assign.first)
28          end
29        end
30      end
31  end
32
33  def find_available_time_slot(vaccine, location, old_time_slot)
34      if vaccine.nil?
35        LocationAndTimeSlotRepository.available_time_slot_without_vaccine(
36          location, old_time_slot.date_and_time.change(hour: 8),
37          old_time_slot.date_and_time.change(hour: 19, min: 40))
38      else
39        LocationAndTimeSlotRepository.available_time_slot_depending_on_vaccine(
40          vaccine, location, old_time_slot.date_and_time.change(hour: 8),
41          old_time_slot.date_and_time.change(hour: 19, min: 40))
42      end
43  end
44
45  def assign(application, location_and_time_slot_to_assign)
46      application.location_and_time_slot_id = location_and_time_slot_to_assign.id
47      application.time_slot_assigned
48      application.save
49      time_and_date = date_and_time_format(
50          location_and_time_slot_to_assign.vaccination_time_slot.date_and_time)
51      address = application.vaccination_location.address
52      city = application.vaccination_location.city
53      message_body = "<h2>Termin cijepljenja za zahtjev "\
54      "##{application.reference}</h2><p>Vrijem cijepljenja: "\
55      "#{time_and_date} <br>Adresa mjesta cijepljenja: #{address}, #{city}</p>"
56      send_email_with_application_id(application.email, message_body)
57  end
```

Prikazani rake zadatak pokrećemo naredbom u terminalu:

```
bundle exec rake scheduler:create_location_and_time_slots
```

Repozitorij dizajn i Active Record

U svakoj većoj aplikaciji postoji veliki broj upita na bazu te vrlo brzo se može doći do problema da promjena jedne funkcionalnosti povlači za sobom mijenjanje desetak upita. Kako bi se taj problem jednostavno riješio, korisno je uvesti dizajn repozitorija. To je dizajn aplikacije u kojem se svi upiti na bazu smjeste u posebne dokumente, a glavni od njih nazovemo `repository.rb`. Sada smo dobili na jednom mjestu sve upite na bazu i vrlo lako možemo promijeniti bilo koji dio bilo kojeg upita.

Sljedeći korak poboljšanja korištenja upita rješava problem koji se pojavi kada se u kratkom vremenu pojavi puno upita na bazu podataka i ona je zapravo usko grlo aplikacije, odnosno usporava njezin rad. Za rješenje tog problema iskorištena su svojstva *Active Record Model*a koji je kompatibilan s dizajnom repozitorija. Naime, *Active Record* ima mogućnost pisanja upita ekvivalentih onima u jeziku SQL. Na taj način smo u samo jednom pozivu na bazu dohvatili sve podatke za čiji dohvat bi nam, bez korištenja tog svojstva, možda bilo potrebno desetak upita na bazu. Primjer kako izgleda jedan takav upit na bazu vidjet ćemo u isječku koda za dohvat slobodnih termina za cijepljenje ovisno o cjepivu.

```
1  # location_and_time_slot_repository.rb
2  class LocationAndTimeSlotRepository
3    def self.available_time_slot_depending_on_vaccine(vaccine, location,
4    new_date_and_time_start_day, new_date_and_time_end_day)
5      LocationAndTimeSlot.where(vaccination_location_id: location.id)
6          .where(vaccine_id: vaccine.id)
7          .left_outer_joins(:vaccination_time_slot)
8          .where('vaccination_time_slots.date_and_time BETWEEN ? AND ?',
9                         new_date_and_time_start_day, new_date_and_time_end_day)
10         .joins('LEFT JOIN applications ON
11             applications.location_and_time_slot_id=vaccination_time_slots.id')
12   end
13 end
```

Što bi bilo jednako upitu u SQL-u:

```
SELECT location_and_time_slots.* FROM location_and_time_slots
LEFT OUTER JOIN vaccination_time_slots
ON vaccination_time_slots.id = location_and_time_slots.vaccination_time_slot_id
LEFT JOIN applications
ON applications.location_and_time_slot_id = vaccination_time_slots.id
WHERE location_and_time_slots.vaccination_location_id = location.id
AND (vaccination_time_slots.date_and_time
BETWEEN new_date_and_time_start_day AND new_date_and_time_end_day);
```

Algoritam za validaciju OIB-a

Za prijavu na cijepljenje u formi se mora unijeti OIB ili MBO stoga je bilo potrebno uvesti nekakvu validaciju tih brojeva. Kako MBO nema alogirtam, barem ne onaj koji je javno poznat, za njega je validacija samo da se sastoji od devet znamenki. No, za razliku od MBO, OIB ima algoritam po kojem se generira i taj algoritam je poznat kao Luhnov algoritam ili Luhnova formula. Osim za generiranje OIB-a, Luhnov algoritam se koristi i za genriranje velikog spektra identifikacijskih brojeva kao što su broj kreditne kartice ili broj osigurane osobe u Kanadi.

Luhnov algoritam se sastoji na računanju kontrolnog broja, pa tako u OIB-u, koji se sastoji od 11 zamenki, prvih 10 su slučajno generirane znamenke dok je 11. znamenka zapravo kontorlni broj. Računanje konrolnog broja se može svesti na četiri koraka:

1. Uzme se dobiveni slučajno generirani broj, u ovom slučaju OIB, ali bez poslijednje znamenke, i krene se od najdesnije znamenke prema lijevo tako da se vrijednost svake druge znamenke (krenuvši od najdesnije) udvostruči

2. Na svakoj poziciji se dobiveni broj zamijeni brojem koji se dobije nakon zbrajanja znamenki tog broja

3. Zbroje se znamenke na svim pozicijama

4. Kontrolni broj se dobije nakon računanja izraza $(10 - (zbroj\_znam \bmod 10))$

U primjeru aplikacije Cijepljenje za validaciju vrijednosti OIB-a, korišten je algoritam koji je ranije implementiran, a može se naći na [4]. Na istoj stranici mogu se pronaći implementacije provjere kontrolnog broja za brojne druge programske jezike osim Ruby-ja.

Dodatni Ruby gemovi

Do sad je naveden veliki broj Ruby *gemo*va koji su korišteni pri izradi aplikacije Cijepljenje, ali postoje još neki koji su korišteni za neke možda ne tako očite dijelove.

Kako zahtjevi za cijepljenje imaju određene statuse u kojima se mogu nalaziti, prelazak iz jednog u statusa u drugi je definiran pomoću *state machine*. Za implementaciju te funkcionalnosti potreban je bio *gem* `aasm`.

Prilikom testiranja aplikacije korištene su neke naprednije mogućnosti RSpec testova zbog kojih su bili potrebni gemovi:

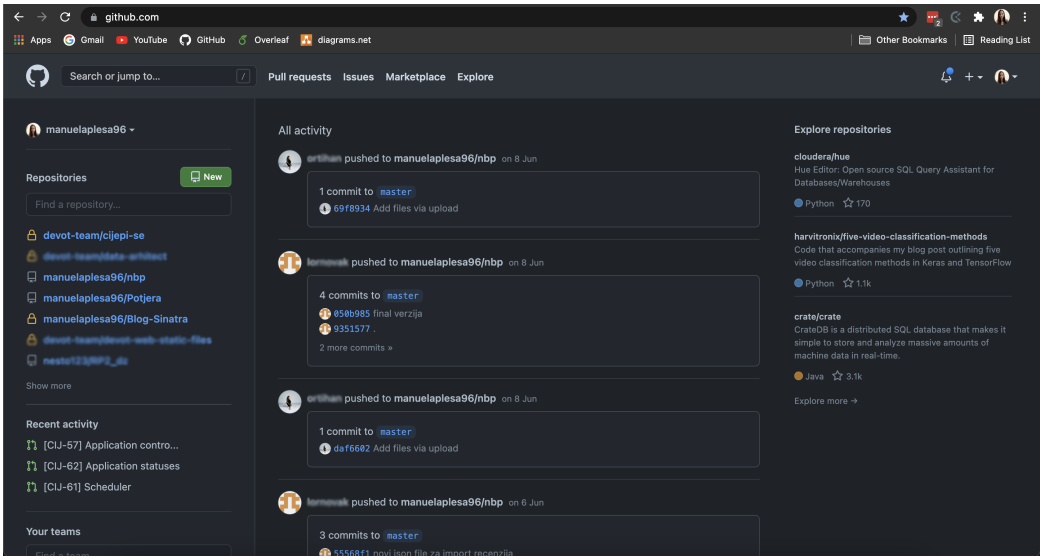
- *gem* `database_cleaner-active_record` kako bi se nakon izvršenog testa svi stvoreni podaci u testnoj bazi obrisali
- *gem* `factory_bot_rails` koji se koristi kako bi se kreirale faktorizacije koje omogućuju brže kreiranje pojedinih objekata unutar testova
- *gem* `shoulda-matchers` kako bi se olakšalo pisanje testova

I *gem*ovi koje još vrijedi spomenuti su oni korišteni za kreiranje sučelja aplikacije, a to su `bootstrap` i `sinatra-flash` koji omogućuje pojavu *flash* poruka o greški ili uspjehu.

Github

Kako bismo razumjeli što je GitHub i za što služi, prvo ćemo objasniti sam pojam Git. Git je besplatan distribuirani sustav za brzu i učinkovitu kontrolu verzija izvornog koda nastao 2005. godine. On omogućuje praćenje promjena u bilo kojem skupu dokumenata, ali najčešće se koristi u procesu softverskog programiranja među programerima koji rade na istom kodu. Kao neka od važnijih svojstava možemo nabrojati distribuiranost, usklađenost s postojećim kodom te efikasnost u radu s velikim projektima. Neki od najpoznatijih pružatelja usluge Git su GitHub koji je korišten u studijskom primjeru te Bitbucket, GitLab Cloud, Azure DevOps Git i Atlassian Stash.

Kao što je navedeno, GitHub je jedan od najpoznatijih pružatelja usluga za Git. Točnije, to je najveća i najnaprednija platforma na svijetu za programere koja služi kao pružatelj internetskih usluga (eng. *Internet hosting*) i za kontrolu verzija koristeći sustav Git. GitHub ima besplatnu verziju koja nudi neograničen broj privatnih repozitorija, te dozvoljava neograničen broj kolaboratora u tim repozitorijima uz ograničeni broj minuta GitHub Actiona mjesečno. Osim besplatnog plana, na GitHubu je moguće imati račun s planom koji se naplaćuje s čime se naravno dobivaju određene pogodnosti.



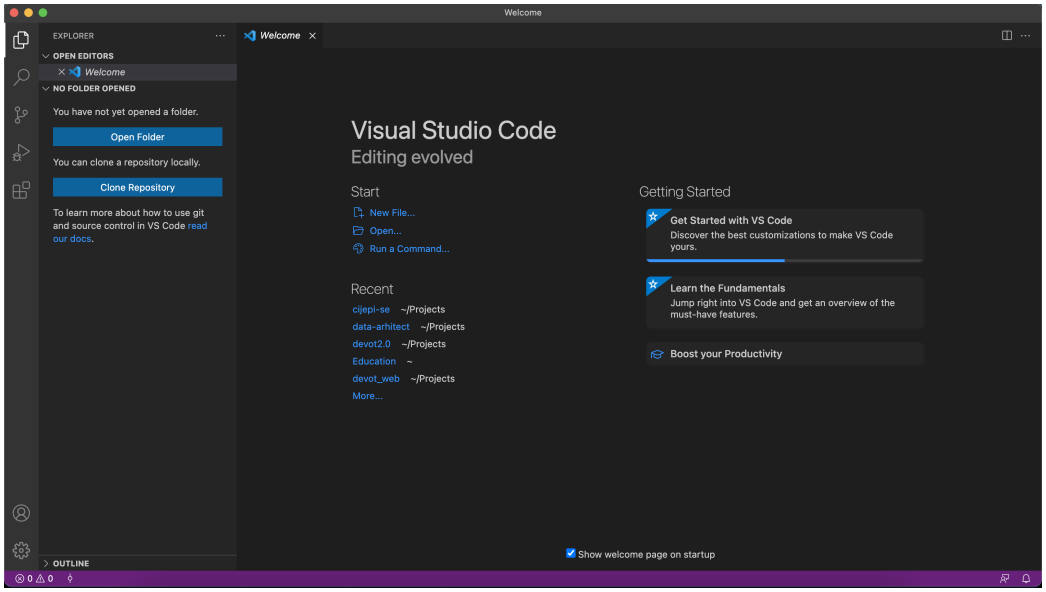
Slika 3.17: GitHub sučelje

U studijskom primjeru koji smo opisali, GitHub je košten u svrhu lakšeg praćenja razvoja aplikacije te kako bi se u slučaju pogreške lako moglo vratiti u neku od ranijih verzija koda. Osim toga, iz ovakvog načina dokumentiranja razvoja aplikacije može se vrlo lako za neku buduću aplikaciju sličnog tipa uzeti neki dio koda koji se bude smatrao primjerenim za tu namjenu.

Visual Studio Code

Visual Studio Code je perolaki, ali snažan *source-code* program kreiran od strane Microsoft-a za Windows, macOS i Linux. Sadrži ekstenzije za brojne programske jezike, između ostalog i za Ruby, C++, C#, Java, Python i PHP te ima ugrađenu podršku za JavaScript, TypeScript i Node.js. Neke od značajki programa su podrška za debuggiranje, pocrtavanje sintakse, dovršavanje i refaktoriranje koda te ima ugrađenu podršku za Git i GitHub koje smo objasnili u prethodnom potpoglavlju.

Umjesto projektnog sustava, Visual Studio Code omogućava korisnicima otvaranje više direktorija koji se mogu spremiti u radno mjesto (eng. *workspace*) za kasnije ponovno korištenje. Visual Studio Code se može proširivati pomoću već spomenutih ekstenzija koje se dodaju preko *extensions* odnosno *plug-in*ova koji su dostupni kroz centralni repozitorij.



Slika 3.18: Visual Studio Code sučelje

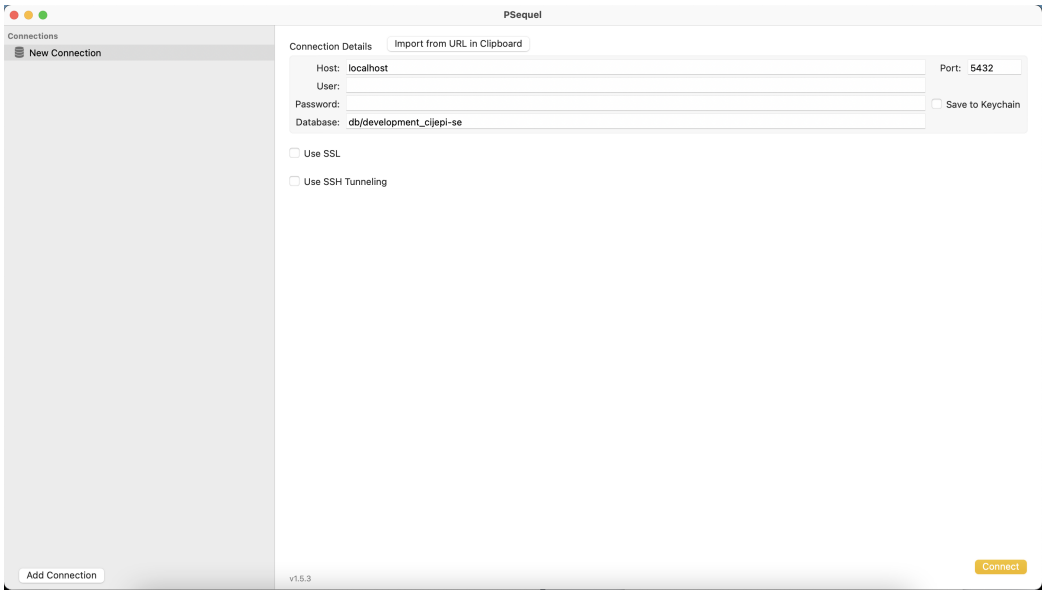
U studijskom primjeru, Visual Studio Code je korišten kao editor u kojem je pisan kod cijele aplikacije te je bio povezan s repozitrijem na GitHubu kako bi se lakše pratile izmjene koda. Visual Studio Code morali smo proširiti ekstenzijama za programski jezik Ruby, HTML i CSS. Također, dodana je ekstenzija rubocop koja služi kako bi se lakše održavala konzistentnost kod pisanja implementacije te kako bismo imali potpuno iskustvo korištenja editor.

PSequel

PSequel je GUI (*graphical user interface*) odnosno sučelje za korištenje PostgeSQL-a trenutno dostupan samo za macOS 10.10+. PostgreSQL je sustav za upravljanje bazama podataka otvorenog koda koji poštuje ACID principe pri izvođenju transakcija. ACID principi podrazumijevaju svojstva:

- Atomarnost (eng. *atomicity*) - da ako se neka od radnji vezanih za transakcije završi greškom, cijela transakcija staje i baza ostaje nepromijenjena.
- Konzistentnost (eng. *consistency*) - baza transakcijama prelazi iz jednog konzistentnog stanja u drugo.

- Izolacija (eng. *isolation*) - osigurava da ako se dvije transakcije obavljaju paralelno, rezultat bude isti kao kada bi se odvijale jedna za drugom.
- Izdržljivost (eng. *durability*) - svojstvo zbog kojeg znamo da su promjene izvršene transakcijom postojane i u slučaju nestanka struje ili sličnog kvara na bazi podataka.



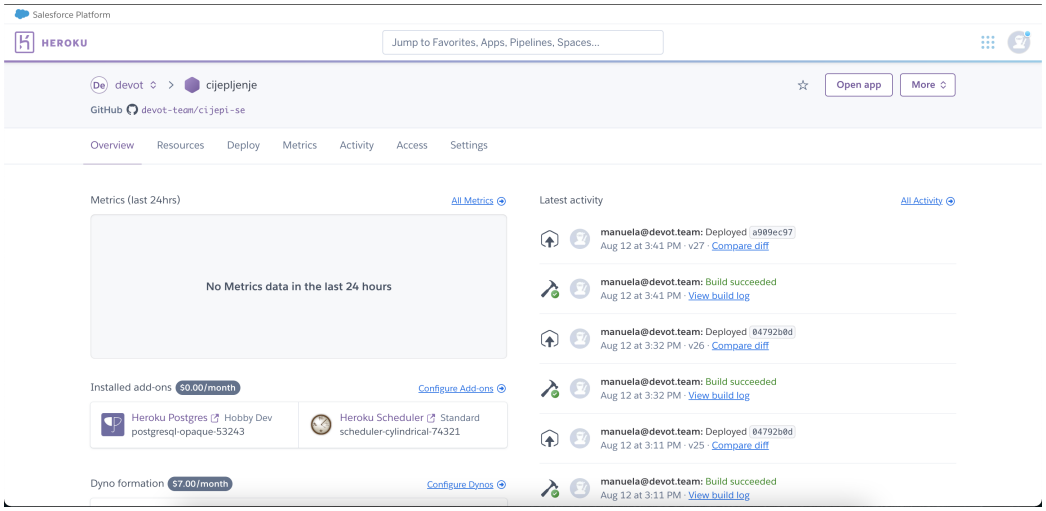
Slika 3.19: PSequel sučelje

Kako je u studijskom primjeru korištena baza podataka PostgreSQL, tako se nametuo i PSequel kao sučelje za praćenje promjena u bazi. S obzirom da Sinatra kao aplikacijski okvir nema ugrađenu konzolu pomoću koje se može pristupiti bazi podataka, za razliku od Ruby on Railsa, sučelje je korišteno najviše tijekom testiranja back-end dijela aplikacije. Također, u svakom trenutku i nakon što je aplikacija u produkcijskoj fazi, PSequel može poslužiti kao pomoć kod debuggiranja ako se pojavi neplanirano ponašanje s podacima.

Heroku

Heroku je *platform as a service* (PaaS), odnosno platforma koja služi kao servis i programerima omogućava izgradnju, pokretanje i upravljanje aplikacijom u oblaku (eng. *the cloud*). Kada se pojavio među programerima, kao jedna od prvih platformi za tu namjenu, podržavao je samo programski jezik Ruby, a sada Heroku podržava također i programske jezike Java, Node.js, Scala, Clojure, Python, PHP i Go. Kao baza podataka se koristi

Heroku Postgres baza čija osnova je PostgrSQL baza podataka koju smo opisali u potpoglavlju 3.9.



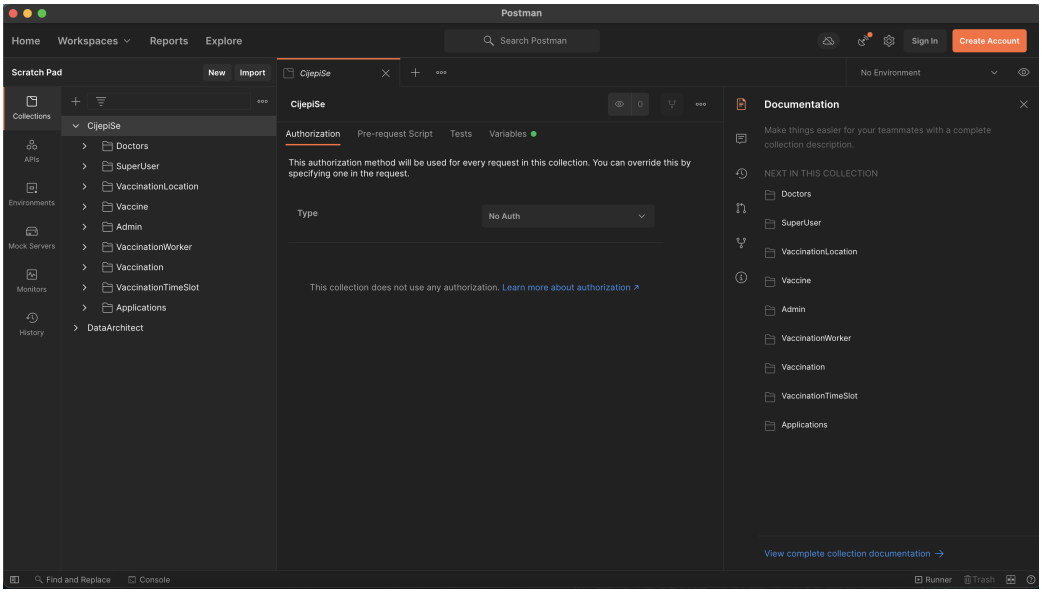
Slika 3.20: Heroku sučelje

Nakon što je aplikacija Cijepljenje u potpunosti implenetirana, postavljena (eng.*deploy*) je na platformu Heroku kako bi se mogla koristiti u produkcijskom okruženju te bila lakše dostupna. Također kreirana je baza podataka te je dodan *scheduler* pomoću kojeg se mogu izvršavati definirani rake zadaci za mijenjanje statusa zahtjeva, kreiranja novih termina cijepljenja te dodavanja termina cijepljenja zahtjevima koji ih čekaju.

Postman

Postman je platforma za API programere koja olakšava i ubrzava implementaciju API-ja. Ona programerima omogućava kreiranje, dijeljenje, testiranje i dokumentiranje API-ja tako da nudi mogućnost kreiranja, spremanja i dijeljenja kako jednostavnih tako i kompleksnih HTTP zahtjeva, ali i odgovora na njih. Programeri ga odabiru kao alat pri kreiranju aplikacije jer pomoću jednostavnog sučelja omogućava lako otkrivanje pogrešaka kod API poziva definiranih u aplikaciji.

Kako je Sinatra okvir koji nema API pozive definirane na jednom mjestu nego se definiraju u svakoj klasi posebno, u studijskom primjeru Postman je bio vrlo koristan alat kako bi se lakše moglo pratiti je su li svi API pozivi dobro definirani. Osim toga, kako postoji opcija spremanja i dijeljenja kreiranih API poziva, to otvara mogućnost za daljnji razvoj aplikacije unutar tima više programera.



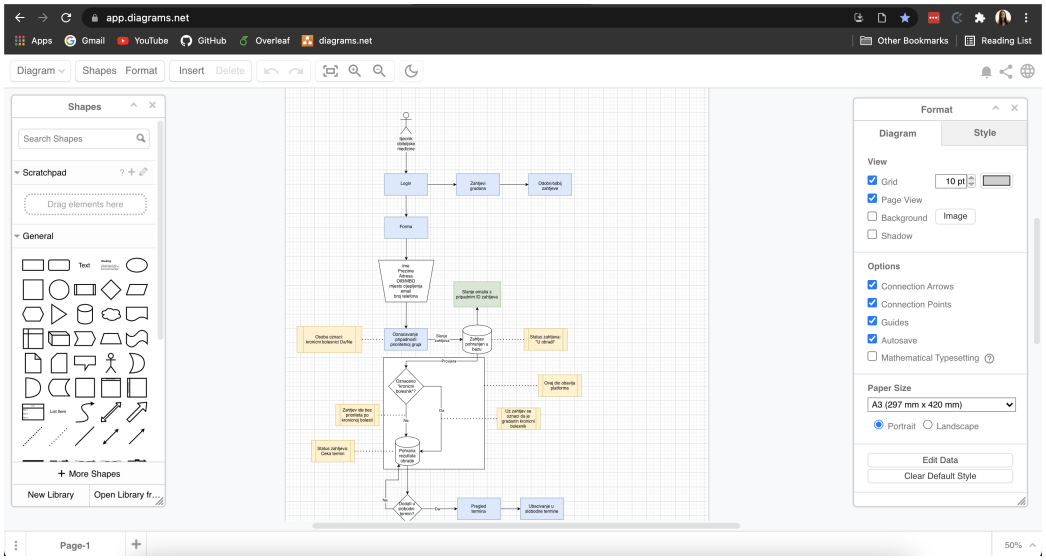
Slika 3.21: Postman sučelje

Draw.io

Draw.io ili diagrams.net je besplatan online softver za crtanje dijagrama. Može se koristiti za razne dijagrame, između ostalog i za UML i ER dijagrame koji se koriste pri razvoju nekog softvera ili aplikacije. Koristan je i već na samom početku razvoja dok je aplikacija tek ideja kako bi se lakše opisao sam slijed odvijanja akcija u pojedinom slučaju upotrebe (eng. *use case*). Prihvaćen je od strane timova za razvoj aplikacije jer ima mogućnost izvoza kreiranog dijagrama te dijeljenja s ostalim članovima tima koji ne moraju imati pristup samom softveru.

U studijskom primjeru Draw.io je korišten od samog početka kako bi se vidjelo koje sve funkcionalnosti treba implementirati u aplikaciji. Kasnije je korišten prilikom nastanka dokumenata sa tehničkim dizajnom koji pobliže opisuju svaki od većih dijelova aplikacije.

Dijagrami koji su korišteni su ER dijagrami kako bi se vidio odnos entiteta koje imamo u sustavu te UML dijagrami kako bi se prikazao odnos klasa koje su kasnije implementirane. U samim dokumenata sa tehničkim dizajnom, UML dijagrame smo nazvali CRUD dijagrami (eng. *Create, Read, Update, Delete*) jer su služili kako bi se vidjelo koje od CRUD akcija klase imaju prema klasama s kojima su povezane.



Slika 3.22: Draw.io sučelje

HTML

HTML ili punim nazivom *Hyper Text Markup Language* je standardni jezik označavanja za dokumenate dizajniran za prikazivanje u web-preglednicima. Najčešće se koristi u kombinaciji sa CSS-om (eng. *Cascading Style Sheets*) i skriptnim jezicima kao što je JavaScript. CSS-om oblikujemo izgled i prikaz sadržaja, dok dodavanjem JavaScript-a definiramo ponašanje i sadržaj web-stranice.

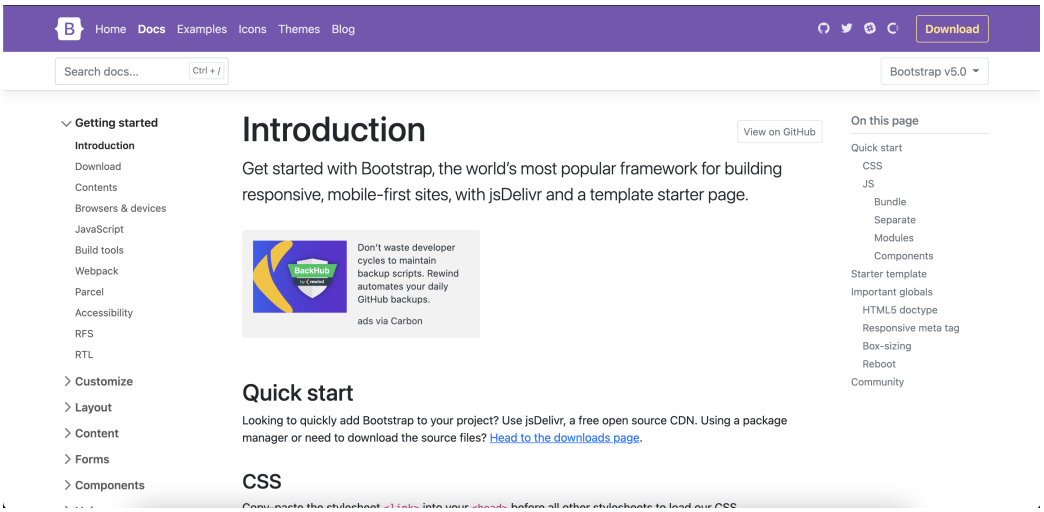
HTML elementi su gradivni blokovi HTML stranica. Oni se opisuju *tag*ovima koji se pišu koristeći šiljate zagrade, na primjer <p> ili
. Preglednici prilikom prikaza stranice ne prikazuju HTML *tag*ove, ali ih koriste za interpretaciju sadržaja stranice.

```
1 <!DOCTYPE html>
2 <html lang="en">
3     <head>
4       <title>Aplikacija</title>
5     </head>
6     <body>
7       <p>Jedini paragraf na ovoj stranici.</p>
8     </body>
9 </html>
```

Kod implementacije *front-end* strane aplikacije iz studijskog primjera korištena je posebna vrsta HTML-a, HTML.ERB. HTML.ERB je HTML pomiješan s programskim jezikom Ruby koristeći HTML *tag*ove što se može vidjeti već i iz ekstenzije .ERB koja označava *Embedded Ruby*, što u prijevodu znači ugrađeni Ruby. To znači da u dokumentu čija ekstenzija je html.erb unutar posebnih HTML *tag*ova možemo koristiti standardnu sintaksu Ruby programskog jezika. Postoje dvije vrste *tag*ova koje se koriste, < %% > koji se koristi samo ako se želi koristiti neki dio koda napisan u Ruby-ju, ali bez ispisa na ekran, dok se tag < % = % > koristi ako se želi vrijednost varijable ispisati na pripadnom mjestu na front-end-u.

```
1 <ul>
2     <% @products.each do |p| %>
3         <li><%=  @p.name %></li>
4     <% end %>
5 </ul>
```

Bootstrap



Slika 3.23: Dokumentacija Bootstrap-a

Bootstrap je besplatan CSS okvir otvorenog koda usmjeren na front-end web-razvoj koji sadrži dizajnerske predloške bazirane na CSS-u i JavaScript-u za forme, gumbe, navigaciju i ostale komponente sučelja. Možemo čak reći da je jednostavniji od samog CSS-a

jer kod CSS-a ne postoje unaprijed definirane klase i dizajni već se sav kod mora pisati otpočetka. Da bi se mogao koristiti, Bootstrap se mora instalirati, odnosno dodati kao *gem* u `Gemfile`, dokument koji smo objasnili u poglavlju 2.1.

Poglavlje 4

Zaključak

Kao što se može vidjeti, aplikacijski okvir Sinatra može biti poprilično moćan aplikacijski okvir za razvoj kompleksnih web-aplikacija. Kako je kompatibilan sa svim alatima koje nudi Ruby, u njemu možemo koristiti većinu alata koji se koriste ili već dolaze sa okvirom Ruby on Rails, okvirom koji se uvijek prvi spominje kada Sinatru uspoređujemo s nekim okvirima baziranim na programskom jeziku Ruby. Nadalje, kao pozitivnu stranu Sinatre možemo navesti i to da je kreiranje ruta za API pozive vrlo intuitivno. Naime, uspoređujući opet s Rails-om gdje se rute moraju definirati prema ranije definiranoj sintaksi aplikacijskog okvira, kod Sinatre je to riješeno elegantije tako da se definiraju metode koje nazovemo kao željeni API poziv.

Iako su pokazane brojne pozitivne strane Sinatre, ipak postoje i neke ne tako dobre. Prva od njih je to što kreirana aplikacija ne bude automatski povezana s nekom bazom, već se moraju dodati posebne postavke kako bi omogućila ta funkcionalnost, a to povezivanje nije uvijek jednsotavna radnja. Također, kao još jednu od mana možemo nabrojati i to što u slučaju kreiranja modularne aplikacije kao što je Cijepljenje koja ima veći broj kontrolera, svaki od kontrolera moramo posebno uključiti kako bi se rute definirane u njemu mogle koristiti.

Kada se pogledaju prednosti i nedostaci, može se zaključiti da nije greška ako se Sinatru uzme kao okvir u kojemu se razvija neka web-aplikacija čak i u slučaju da je ona kompleksnija. To smo zaključili iz toga što, iako nema neke predefinirane funkcionalnosti i alate koji dolaze s njom kao što to ima Rails, postoji veliki broj Ruby gemova pomoću kojih možemo na teži ili lakši način sve te funkcionalnosti i alate dodati u Sinatra aplikaciju. To se vidi iz primjera korištenja *gem*ova kao što su `sinatra-activerecord`, `activerecord`, `activesupport`, `rspec-rails` i mnogi drugi.

Iako je ovaj rad započet s opisom Sinatre kao aplikacijskog okvira za male, a čak i aplikacije od samo jednog dokumenta, definitivno možemo reći da se u svijetu razvoja web-aplikacija aplikacijski okvir Sinatra može naći i na popisu okvira koji se koriste za

razvoj kompleksnijih aplikacija. Također, kao potpora toj izjavi može se uzeti i to da se Sinatra svakodnevno razvija, što se može vidjeti i u pripadnom repozitoriju na GitHubu na [3]. To bi značilo da svakim danom Sinatra postaje sve moćniji aplikacijski okvir pa samo možemo maštati do koje razine se može razviti. Iako možda je realnije pretpostaviti da će suradnici u razvoju Sinare i dalje pokušavati održati sliku Sinatre kao aplikacijskog okvira za male web-aplikacije.

Kao zaključak, važno je istaknuti da je ovaj rad baziran ponajprije na funkcionalnostima Sinatre koje su korištene pri razvoju aplikacije Cijepljenje. No to ne znači da je to sve što Sinatra ima za ponuditi. Dapače, postoji još cijeli niz mogućnosti kojima se čak i aplikacija Cijepljenje može poboljšati, ali također i za razvoj neke druge aplikacije. Stoga je svakako korisno na [8] pogledati što sve nudi aplikacijski okvir Sinatra.

Bibliografija

[1] *Dokumentacija programskog jezika Ruby*. URL: https://www.ruby-lang.org/en/documentation/.

[2] *Git repozitorij aplikacije Cijepljenje*. URL: https://github.com/manuelaplesa96/cijepljenje.

[3] *GitHub repozitorij aplikacijskog okvira Sinatra*. URL: https://github.com/sinatra/sinatra.

[4] *GitHub repozitorij s algoritmom za validaciju OIB-a*. URL: https://github.com/domagojpa/oib-validation.

[5] *GitHub repozitorij za generiranje QR koda*. URL: https://github.com/whomwah/rqrcode.

[6] Haase K. Harris A. *Sinatra: Up and Running*. O'Reilly Media, Inc., 2011.

[7] *Web stranica koja sadrži sve Ruby gemove*. URL: https://rubygems.org/.

[8] *Web stranica službene dokumentacije aplikacijskog okvira Sinatra*. URL: http://sinatrarb.com.

Sažetak

U ovom radu obrađen je pojam okvira za web-aplikacije, povijest nastanka tog pojma, moguće arhitekture te vrste samih aplikacijskih okvira. Također, predstavljen je i programski jezik Ruby te aplikacijski okvir Sinatra za razvoj web-aplikacija. Dane su osnovne značajke Sinatre te je napravljena usporedba Sinatre s nekim danas popularnim aplikacijskim okvirima. Na kraju je samo korištenje Sinatre pokazano u studijskom primjeru aplikacije Cijepljenje čime je dokazano kako dodavanjem Ruby *gem*ova i korištenjem javno dostupnih alata za razvoj web-aplikacija možemo poprilično povećati mogućnosti Sinatre.

Summary

This paper deals with the concept of web application frameworks, the history of this concept, possible architectures and the types of application frameworks. Also, the Ruby programming language and the Sinatra application framework for web application development were presented. The basic features of Sinatra are given by comparing Sinatra with some of today's popular application frameworks. In the end, use of Sinatra is shown in the case study of application Cijepljenje (Vaccination) which proves that by adding Ruby gems and using available web application development tools, we can significantly increase Sinatra's capabilities.

Životopis

Rođena sam 10. prosinca 1996. godine u Novoj Gradišci. Osnovnu školu sam pohađala u gradu Nova Gradiška te sam nakon završene osnovne škole ondje upisala opću gimnaziju. Srednjoškolsko obrazovanje sam završila 2015. godine nakon čega sam upisala preddiplomski sveučilišni studij Matematike na Matematičkom odsjeku Prirodoslovno-matematičkog fakulteta u Zagrebu. Godine 2019. završila sam preddiplomski studij te sam upisala diplomskij studij Računarstva i matematike na istom fakultetu.