Testiranje Rails web aplikacije

Mijatović, Vedran

SVEUČILIŠTE JURJA STROSSMAYERA U OSIJEKU

FAKULTET ELEKTROTEHNIKE, RAČUNARSTVA I INFORMACIJSKIH TEHNOLOGIJA

Stručni studij

TESTIRANJE RAILS WEB APLIKACIJE

Završni rad

Vedran Mijatović

Osijek, 2018.

Sadržaj

1. UVOD **1**

2. TESTIRANJE PROGRAMSKOG KODA **2**

2.1. Metode testiranja 3

2.2. Razvoj upogonjen testiranjem 5

2.3. Razvoj upogonjen ponašanjem 6

3. TESTIRANJE RAILS WEB APLIKACIJA **8**

3.1. Ruby on Rails 8

3.2. Aplikacija za deploy 9

3.3. Rspec 12

3.3.1. Kreiranje i testiranje modela 15

3.3.2. Kreiranje i testiranje kontrolera 17

3.4. Cucumber 23

3.5. Automatizacija testiranja grafičkog sučelja 25

4. Zaključak **27**

LITERATURA **28**

SAŽETAK **29**

ABSTRACT **30**

ŽIVOTOPIS **31**

1. UVOD

U današnje vrijeme ubrzani stil života iziskuje veću efikasnost u izvršavanju svakodnevnih zadataka. Samim time čovjek poseže za tehnologijom i programskim riješenjima koji će mu život učiniti "lakšim" kako bi se fokusirao na probleme koji iziskuju njegovo vrijeme i kreativnost.

Broj programskih riješenja koji su nam dostupni u sve bržem je rastu i takvo tržište postalo je pravo bojište za sve one koji žele ugrabiti svoj dio kolača. Brzina i efikasnost u izradi takvih riješenja nikada nisu bili bitniji te se uvijek teži automatizaciji procesa izrade. Testiranje je jedan od bitnijih dijelova procesa izrade i u projektima manjeg opsega najčešće se radi ručno ali kako funkcionalnosti projekta rastu pojavljuje se potreba za automatizacijom testiranja.

Cilj ovog završnog rada je analizati načine automatskog testiranja u Ruby on Rails aplikacijama i postaviti temelje za daljnji razvoj konkretne aplikacije u smjeru automatizacijskog testiranja već postojećih funkcionalnosti kako bi se razvojni tim što više mogao fokusirati na probleme koji dolaze u budućim proširenjima. Upoznat ćemo se sa *Ruby on Rails* razvojnim okruženjem koje je već duže vrijeme među popularnijima u izradi web aplikacija kao i samom aplikacijom koja je u ovoj fazi potpuno funkcionalna ali koju očekuju dodatna proširenja u budućim verzijama. Iz tog razloga već postojeće funkcionalnosti će biti potrebno testirati prilikom svake nove implementacije. Testiranja će se obavljati alatima koji su standardni u većini organizacija i razvojnih timova koji koriste *Ruby on Rails* razvojno okruženje.

2. TESTIRANJE PROGRAMSKOG KODA

Testiranje je evaluacija programske podrške ili njezinih komponenti s namjerom da se ocjeni ispunjavaju li postavljene zahtjeve[3]. U procesu testiranja programski kôd se izvršava i dobiveni rezultati se uspoređuju sa očekivanim rezultatima. Dva su moguća rezultata testiranja, izvršeni test može biti uspješan ako se dobiveni i očekivani rezultati podudaraju ili neuspješan ukoliko postoji razlika. Dvije su vrste grešaka koje mogu rezultirati neuspješnim testiranjem kao i neprihvatljivom funkcionalnošću aplikacije:

- **Sintaksna pogreška** - ukoliko programer prilikom pisanja koda ne poštuje pravila programskog jezika ili razvojnog okruženja u kojem razvija aplikaciju (npr. zatipak, ne zatvoreno tijelo funkcije ili zaboravljeni zarez).
- **Semantička pogreška** - ukoliko algoritam nije ispravno logički napisan i u nekim ili svim slučajevima ne daje tražene rezultate (npr. vraća tražene podatke ali u krivom redosljedu ili vraća sve zapise a očekuje se zadnjih pet)

U slučaju neuspješnog testiranja programski kôd se redizajnira te se ponovno podvrgava testiranju. To je zatvoreni krug koji se ponavlja dokle god ne dobijemo željene rezultate tj. dok naša aplikacija ne ispunjava postavljene zahtjeve (slika 2.1.).

Slika 2.1. Ciklus testiranja

2.1. Metode testiranja

Testiranje se najosnovnije može podijeliti na:

- **Ručno testiranje** - kada osoba koja je zadužena za testiranje preuzima korisničku ulogu, samostalno pokreće program ili njegove dijelove i provjerava ispravnost rezultata. Ovaj proces vođen je planom testiranja kako bi se "pokrile" sve funkcionalnosti, scenariji i granični slučajevi. Prednosti ovakve metode su ne pisanje automatiziranih testova koji u nekim slučajevima iziskuju gotovo pola količine vremena koje je potrebno za pisanje samog programskog koda.
- **Automatizirano testiranje** - kada razvojni tim osim samog programa razvija i sustav za testiranje funkcionalnosti i odziva pojedinih dijelova programa. Osim testiranja ispravnosti često se testira i efikasnost programa mjerenjem vremena koje je potrebno za izvršavanje zadatka. Ispravnost programa koji ima razvijeni sustav za testiranje u svakom trenutku možemo provjeriti brzo i pouzdano te je samim time uloženo vrijeme za ovakvo testiranje dugoročno isplativije.

Prije izrade same aplikacije potrebno je odlučiti koju od navedenih metoda testiranja koristiti što ponekad nije lak posao. Prije same odluke potrebno je odgovoriti na nekoliko ključnih pitanja[3].

Radi li se o velikom i kompleksnom projektu? Ponajprije, ima li promatrani projekt tendenciju rasta i očekuju li se dodatna proširenja u budućnosti. Pri implementaciji nove funkcionalnosti ponekad nije očito da ona utječe na drugu i moguć je previd koji bi se automatiziranim testiranjem eliminirao.

Potrebno li je učestalo testiranje aplikacije i njezinih dijelova? Jedan od najčešćih razloga koji iziskuje često testiranje je proširivanje već postojeće aplikacije. Nakon svakog novog proširenja potrebno je testirati kompatibilnost sa već postojećim starim funkcijama što znači da svako proširenje rezultira testiranjem cijele aplikacije. Isto tako nakon poboljšavanja već postojećeg koda, zbog efikasnosti ili lakše čitljivosti, aplikaciju je potrebno ponovo testirati.

Mijenjaju li se često zahtjevi koje ispunjava aplikacija? Ako su očekivane učestale izmjene zahtjeva koje je aplikacija potrebna izvršavati isto tako možemo očekivati i česte izmjene sustava testiranja što zahtjeva dodatne resurse.

Potrebno li je testirati procesne mogučnosti aplikacije? Dobro je biti upoznat sa ograničenjima i kapacitetima aplikacije kako bi smo na vrijeme poduzeli korake tehničkih poboljšanja. Ako očekujemo veliki broj korisnika ili korisničkih zahtjeva upućenih našem poslužitelju potrebno je testovima simulirati veliki broj korisnika ili njihovih zahtjeva te promatrati rad aplikacije.

Može li se osigurati potrebno vrijeme za razvijanje testnog sustava? Vjerojatno i najključnije pitanje u odluci o načinu testiranja. S obzirom na već spomenutu činjenicu da razvoj testnog sustava u ekstremnim slučajevima može trajati gotovo 50% vremena razvoja same aplikacije, vrijeme može biti ključno u slučajevima kada su potrebna brza programska rješenja.

Testiranje programske podrške razlikujemo i po metodama[1]:

- **Crne kutije** - u kojoj testeri nemaju uvid u kôd aplikacije nego isključivo promatraju izlazne rezultate.
- **Bijele kutije** - ovu metodu najčešće provode članovi razvojnog tima koji imaju mogućnost pregleda programskog koda aplikacije. Za vrijeme samog testiranja pokušavaju pronaći grešku u sustavu ukoliko je test neuspješan.
- **Statičko testiranje** - pregledava se programski kôd bez njegovog pokretanja. Ova metoda se primjenuje u najranijim fazama projekta.
- **Dinamičko testiranje** - program se pokreće sa raznim vrijednostima ulaznih parametara nakon čega testeri dobivene rezultate uspoređuju sa očekivanima.
- **Testiranje korisničkog grafičkog sučelja** - pregledavaju se elementi grafičkog sučelja (format teksta, veličina slova, boje, raspored elemenata, obrasci itd.). Ova metoda je često vremenski zahtjevna te se nerjetko prepušta vanjskim suradnicima.

Također, testiranja možemo podijeliti i po razinama na kojima se izvršavaju[1]:

- **Jedinično testiranje** - testiraju se najmanji sastavni dijelovi programa pojedinačno kao što su finkcije ili klase.
- **Testiranje komponenti** - provjerava se ispravnost komponenti sastavljenih od više jedinica koje zajedno ispunjavaju definirane zahtjeve.
- **Integracijsko testiranje** - testira se međusobna interakcija gore navedenih komponenti.
- **Sistemsko testiranje** - cijelokupni sustav podvrgava se testiranju sa različitim scenarijima kako bi se uvjerili da sve radi kako je predviđeno.
- **Alfa testiranje** - vrši se unutar razvojnog tima projekta gdje članovi tima preuzimaju ulogu krajnjeg korisnika.
- **Beta testiranje** - ujedno i zadnja faza testiranja prije samog izdavanja programske podrške. Najčešće se određenom broju korisnika daje mogućnost korištenja gotovog proizvoda sa ciljem prikupljanja povratnih informacija stvarnog korisnika.

2.2. Razvoj upogonjen testiranjem

Odluku o tome potrebno li je testiranje aplikacije koja se planira razvijati nužno je donijeti u samom početku projekta jer to će oblikovati proces same izrade aplikacije. Razvoj upogonjen testiranjem ili TDD (engl. "*Test driven developement"*) način je u kojem se prije kreiranja nove programske jedinice kreira test a tek nakon toga programski kôd koji riješava zadani problem a samim time i rezultira uspješnim testom. Razvoj upogonjen testiranjem odvija se u nekoliko jednostavnih koraka pisanih u imperativu:

1. Dodaj test za novu programsku jedinicu
2. Pokreni sve testove i uvjeri se da je novi test neuspješan
3. Napiši kôd za novu programsku jedinicu
4. Pokreni testove i uvjeri se da su svi uspješni
5. "Uljepšaj" kôd programske jedinice
6. Kreni ispočetka

U prvom koraku dodaje se novi test kojim se definira što se od nove programske jedinice očekuje i nakon toga pokreću se svi napisani testovi. Još nije implementiran programski kôd pa se samim time i očekuje da će zadnje dodani test biti neuspješan. Zatim se pristupa pisanju programskog koda kojem je u ovom koraku cilj da zadovolji zahtjeve testa. Nakon što je test zadovoljen kôd se redizajnira kako bi zadovoljio standarde najčešće definirane unutar razvojnog tima. Nakon svake izmjene koda pokreću se testovi kako bi bili sigurni da je kôd i dalje semantički i sintaksno ispravan. Ovakav način rada osigurava kvalitetu napisanog koda kao i razumjevanje istog unutar tima.

2.3. Razvoj upogonjen ponašanjem

Za razliku od TDD razvoj upogonjen ponašanjem ili BDD (engl. *"behaviour driven development"*) metoda testiranja koristi pogled na sustav izvana i sastoji se od tri dijela koje zajedno nazivamo scenarij. Ne testiraju se programske jedinice pojedinačno nego funkcionalnosti aplikacije. Ovo testiranje možemo shvatiti kao metodu crne kutije u kojoj se vrši testiranje bez saznanja o unutrašnjim podprocesima promatranih funkcija. Ovakav pristup omogućava jednostavniju izmjenu programskih jedinica jer su BDD testovi fokusirani na "što ono radi" a ne na "kako ono radi". Kako bi razliku između BDD i TDD bila što jasnija poslužiti će sljedeći primjer:

Testira se sustav za prijavu putnika u vozilu gradskog prijevoza koja se vrši odgovarajućom karticom. Kartica sadrži jedinstveni kôd putem kojega se nakon prijave naplaćuje cijena vožnje, koja je zbog pojednostavljenja primjera jednaka za sve brojeve prijeđenih stanica.

TDD testiranje za ovaj primjer izgledalo bi ovako:

1. Pronalaženje zapisa pomoću jedinstvenog koda
    - Daje li ova programska jedinica očekivanu izlaznu vrijednost ukoliko kôd nije ispravno očitan?
    - Pronađen li je točno traženi zapis u slučaju ispravno učitanog koda?
2. Stanje kredita na kartici
    - Daje li ova programska jedinica očekivanu vrijednost u slučaju da ne sadrži vrijednost kredita potrebnih za minimalno jednu vožnju?
    - Dobijamo li očekivani rezultat u slučaju da je prisutno dovoljno kredita?

3. Naplata vožnje
    - Je li umanjen ukupni broj kredita za točno odgovarajući iznos cijene vožnje?
    - Je li dobivena odgovarajuća vrijednost na izlazu nakon uspješne naplate?

BDD testiranje:

1. Uspješna naplata vožnje
    - Početno stanje - putnik posjeduje karticu sa dovoljnim brojem kredita za vožnju
    - Događaj - putnik prislanja karticu na čitač
    - Odziv - uređaj signalizira uspješnu naplatu
2. Neuspješna naplata vožnje
    - Početno stanje- putnik posjeduje karticu sa nedovoljnim brojem kredita
    - Događaj - putnik prislanja karticu na čitač
    - Odziv - uređaj signalizira neuspješnu naplatu

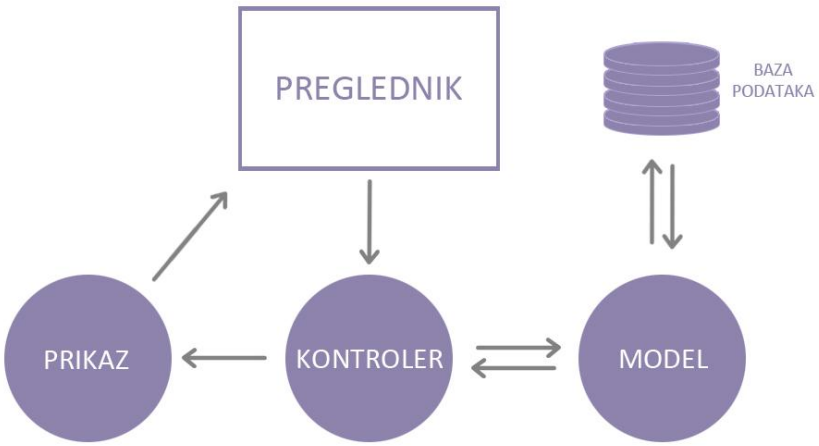
Cilj BDD testiranja je pisanje testova koji neće biti razumljivi samo unutar razvojnog tima nego i klijentu, zato takvi testovi više podjsećaju na smislene rečenice nego programski kôd. Iz navedenog primjera jasno je vidljivo da se ova metoda bavi isključivo promatranjem izvana i ne ulazi u tehničke detalje samog sustava kao TDD. Ukoliko dođe do izmjene u kojoj se putnicima dozvoljava dugovanje u iznosu od jedne vožnje BDD sustav testiranja nije potrebno izmjenjivati dok je TDD test *"Daje li ova programska jedinica očekivanu vrijednost u slučaju da ne sadrži vrijednost kredita potrebnih za minimalno jednu vožnju?"* potrebno izmjeniti kako bi svi testovi bili uspješni. Takva vrsta "otpornija" je na izmjene koje ne mijenjaju smisao sustava nego samo tehničku izvedbu. Ukoliko postoji greška u sustavu i on ne daje očekivane izlazne vrijednosti TDD sustav je puno zahvalniji jer preciznije locira izvor greške zbog svoje granularne strukture.

3. TESTIRANJE RAILS WEB APLIKACIJA

3.1. Ruby on Rails

Ruby on Rails je razvojno okruženje upogonjeno programskim jezikom *Ruby* i baziran je na MVC (engl. *"model, view, controller"*) načinu rada[5]. Sastoji se od velikog broja alata i gotovih riješenja koja korisnicima *Ruby on Rails*a olakšavaju rad i pružaju mogućnost brze izrade internetskih aplikacija. Razvijanje se uglavnom sastoji od pisanja *CSS, HTML, JavaScript, JSON* i *Ruby* datoteka koje poput kotačića stroja ovo razvojno okruženje objedinjava u jednu cjelinu koju nazivamo aplikacija. Mnoge uspješne internetske aplikacije "napisane" su u ovom razvojnom okruženju, kao što su *GitHub, Airbnb, Shopify, Twitch* pa čak *Twitter* i *Sound Cloud* u svojim počecima. Osim brzine izrade aplikacija jedna od važnijih odlika ovog okruženja je i sam *Ruby* programski jezik na kojem se bazira jer je lako čitljiv i razumljiv.

MCV način rada koji je temelj rails okruženja sastoji se od tri cjeline koje međudjelovanjem čine rails aplikaciju funkcionalnom (slika 3.1.). Modeli su definirani programskim klasama uz pomoć kojih *ActiveRecord* sustav komunicira i upravlja bazom podataka. Njih možemo shvatiti kao resurse kojima aplikacija upravlja. Upravljanjem objektima modela indirektno se upravlja bazom podataka. Ovaj aspekt rails razvojnog okruženja ujedno je i najbolji jer omogućava pisanje složenih *SQL* naredbi jednostavnim i kratkim *Ruby* metodama.



Slika 3.1. MVC način rada

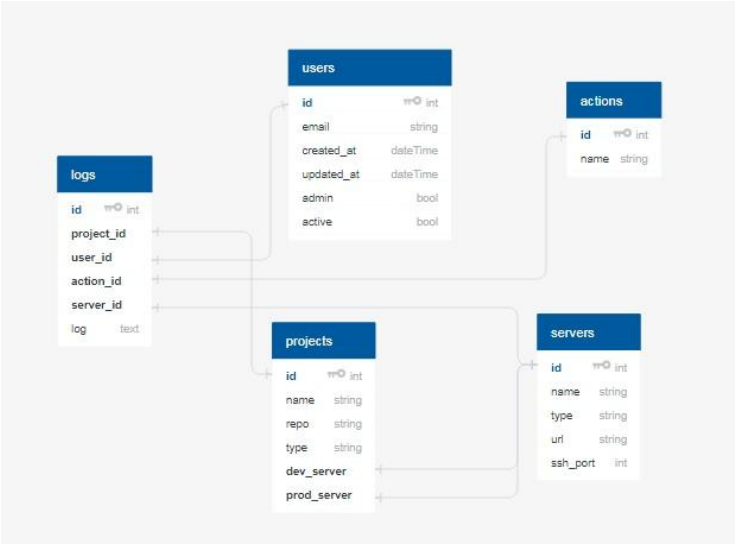
Kontroler je upravljački programski kôd koji možemo opisati kao sponu između resursa i krajnjeg korisnika. U njima definiramo metode putem kojih korisnih sprema, dohvaća, izmjenjuje i briše zapise u bazi. Prikazi (engl. *"views"*) su način na koji aplikacija komunicira sa krajnjim korisnikom. Najčešće se sastoje od *"html.erb"* proširenog tipa HTML dokumenta koji osim standardnog HTML koda prepoznaje i *Ruby* programski kôd.

3.2. Aplikacija za deploy

Nakon izrade internetske aplikacije, kako bi ona bila dostupna krajnjim korisnicima, potrebno ju je postaviti (eng. *"deploy"*) na internetski poslužitelj. Ovaj postupak iziskuje unos velikog broja konfiguracijskih naredbi i ubrzo postaje vremenski zahtjevan sa većim brojem aktualizacija aplikacije. Pošto se skup naredbi razlikuje u samo nekoliko dinamičkih parametara ovaj proces moguće je automatizirati.

Cilj aplikacije koja se obrađuje i testira u ovom radu je olakšati postavljanje aplikacija na internetski poslužitelj pokretanjem sekvencijalnog slijeda naredbi kojima dinamički prosljeđujemo ključne parametre. Jedni od takvih parametara su lokacija izvornog koda aplikacije (u ovom slučaju poveznica na git repozitorij) i lokacija internetskog poslužitelja na koji je aplikaciju potrebno postaviti.

Radi što boljeg razumjevanja spomenute aplikacije poslužit će i njezin dijagram baze podataka (slika 3.2.) koji je nužno kreirati prije početka razvijanja programskog koda.

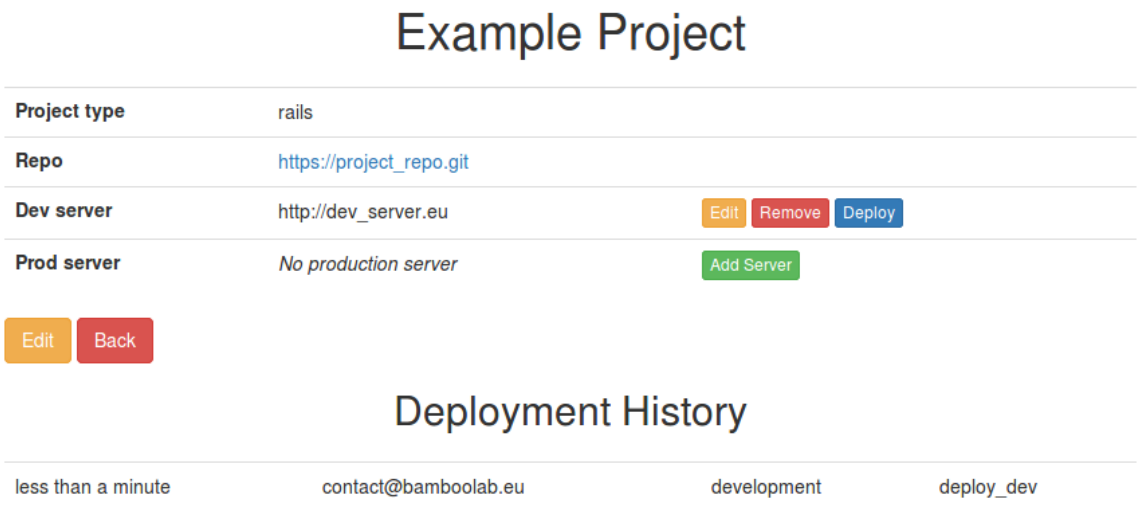


Slika 2.2. Dijagram baze podataka

Aplikaciji će moći pristupati samo članovi razvojnog tima stoga je potrebno pohraniti podatke o korisnicima radi sustava autentikacije.

Kako će aplikacija služiti za postavljanje većeg broja aplikacija, podaci o aplikacijama pohranjuju se u tablicu **`projects`** koja sadrži ime, adresa lokacije izvornog koda te vanjske ključeve produkcijskog i razvojnog poslužitelja. Podaci o poslužiteljima će biti pohranjeni u tablici **`servers`** i informacije o bitnim korisničkim aktivnostima biti će bilježene u tablici **`log actions`**.

Nakon što korisnik pohrani važeće podatke o novom projektu imati će mogućnost odabiranja poslužitelja na koji će projekt biti postavljen (slika 3.3.). Dvije su vrste poslužitelja koje možemo dodjeljivati aplikacijama a oni su razvojni i produkcijski. Razvojni poslužitelji služe za alfa i beta testiranja koja iziskuju sitne i česte izmjene unutar aplikacija i to je jedan od glavnih razloga zašto se pojavila potreba za projektom koji promatramo. Nakon što se razvojni tim uvjeri da aplikacija na razvojnom poslužitelju ispunjava dogovorene zahtjeve, ona se postavlja na produkcijski poslužitelj gdje je dostupna krajnjem korisniku. Nakon postavljanja aplikacije na produkcijski svaka nova aktualizacija aplikacije se najprije testira na razvojnom poslužitelju.

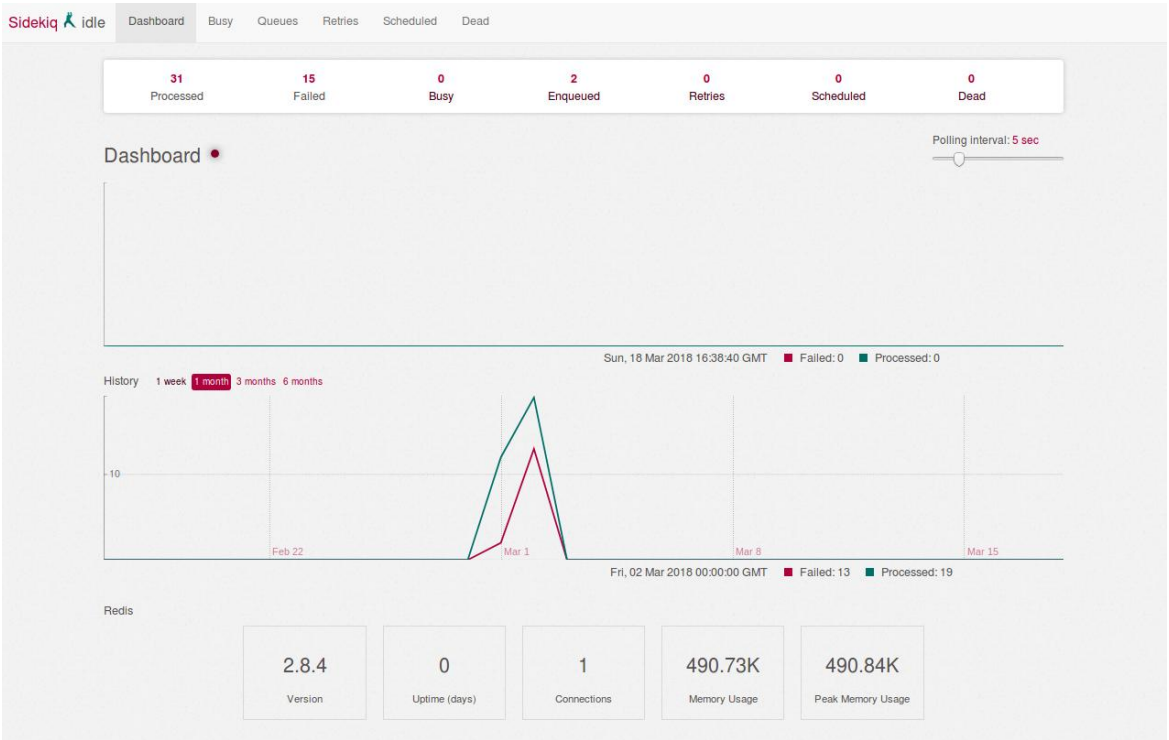


Slika 3.3. Prikaz pojedinog projekta

Nakon dodijeljivanja poslužitelja projektu korisnik ima mogućnost pokrenuti proces postavljanja aplikacije pritiskom na tipku *"Deploy"*. Nakon što je korisnik podnio zahtjev za postavljanjem aplikacije na željeni poslužitelj pokreće se pozadinski proces koji korisniku najprije šalje obavijest da je zahtjev zaprimljen. Ovakav način rada korisnika ne obvezuje da bude prisutan tokom cijelog procesa postavljanja nego on može slobodno napustiti aplikaciju i ne mora čekati kraj procesa.

Nakon što je podnesen zahtjev za postavljanje proces se prosljeđuje *Redis* sustavu za manipulaciju zahtjevima. Ovaj sustav zadužen je za uređivanje rasporeda po kojem će se pozadinski procesi obavljati. Pozadinski procesi izvršavaju se *Sidekiq* "radnicima" koji poslove prosljeđene od strane *Redisa* izvršavaju. U slučaju da je podneseno nekoliko zahtjeva za postavljanje oni će se izvršavati redom po kojem su i pristigli. O uspješnom ili neuspješnom postavljanu krajnji korisnik biti će obaviješten.

Sidekiq sustav je gotovo riješenje koje pruža mogućnost pregleda zahtjeva koji su izvršeni ili na listi čekanja (slika 3.4.). Sastoji se i od grafičkog prikaza broja aktivnosti po jedinici vremena.



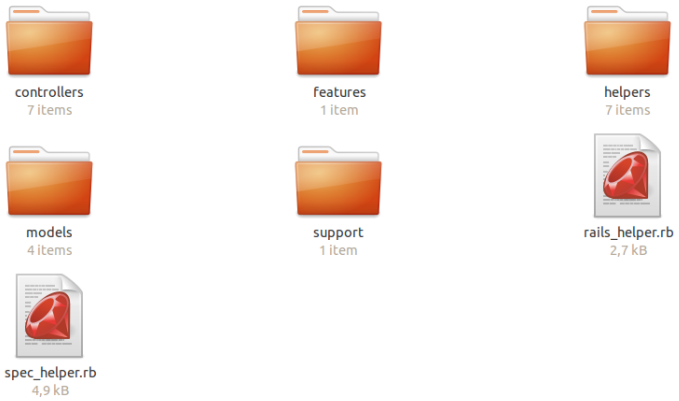
Slika 3.4. Sidekiq prikaz rasporeda pozadinskih procesa

Prilikom svakog zahtjeva podaci o korisniku, vremenu podnošenja i ulazni parametri biti će pohranjeni u tablici **`log actions`** kako bi imali uvid u aktivnosti aplikacije. Bilježenjem aktivnosti korisnici će moći u svakom trenutku ustanoviti kada je zadnji puta obavljeno postavljanje na određeni poslužitelj ("*Deployment History*").

3.3. Rspec

Rspec je alat koji se koristi u izgradnji sustava testiranja ove aplikacije. Njega možemo zamisliti kao programski jezik više razine koji je napisan uz pomoć *Ruby* programskog jezika i zbog toga testovi napisani u *Rspecu* su lako razumljivi čak i ne stručnjacima. *Rspec-rails* proširenje je prilagođeno *Ruby* aplikacijama i kroz sustav za proširenja jednostavno se implementira uvrštavanjem u *"Gemfile"* datoteku koju možemo najjednostavnije shvatiti kao datoteku za manipuliranje proširenjima.

Instalacijom *Rspec-rails* proširenja kreiran je *"spec"* direktorij u kojem će biti smješteni svi napisani testovi (slika 3.5.).



Slika 3.5. Spec organizacija testova

Testovi su organizirani u nekoliko direktorija:

- **Controllers** - sadržava testove kojim se provjerava rad metoda kontrolera aplikacije.
- **Models** - testovi za *Rails* modele i njihove metode.
- **Helpers** - testovi za pomoćne metode. Pomoćne metode u *Rails* aplikacijama koriste se kako bi se uklonile logičke operacije sa HTML stranica.

- **Features -** testovi kojima testiramo cijelokupne funkcionalnosti aplikacije (npr. korisnička prijava).
- **Support -** pomoćne metode koje se često koriste u testnom sustavu.

Datoteke *"rails_helper.rb"* i *"spec_helper.rb"* konfiguracijske su prirode i u njima razvojni tim definira željene postavke testnog sustava kao što su nasumični poredak izvršavanja testova, imenovanje 10 najsporijih testova na kraju svakog testiranja i sl.

Pošto je predviđeno da ovu aplikaciju koriste samo registrirani korisnici potreban je sustav autentikacije. Za ovu funkcionalnost koristit će gotovo riješenje koje se implementira uz pomoć *"Gemfile"* datoteke kao što je to učinjeno sa *Rspec* proširenjem. Naziv ovog proširenja je *Devise* i ono je najprihvaćenije proširenje takvoga tipa od strane članova R*ails* zajednice. Kako bi smo bili sigurni da naš sustav radi kako je i predviđeno potrebno je kreirati test koji to provjerava (slika 3.6.).

Kada korisnik u svome internetskom pregledniku pokuša pristupiti aplikaciji, točnije početnoj stranici aplikacije, ukoliko nije prijavljen predviđeno je da ga aplikacija usmjeri na stranicu za korisničku prijavu. Takvo ponašanje aplikacije osigurano je pomoćnom metodom *Devise* proširenja **`authenticate_user!`** koju glavni kontroler poziva prilikom svakog korisničkog zahtjeva. Primjetit ćemo da se svaki napisani test sastoji od **`context`** bloka koji opisuje funkcionalnost koju testiramo i **`it`** blokova koji opisuju kakav ishod testa se u pod kojim uvjetima očekuje.

```
require 'rails_helper'

RSpec.describe PagesController, type: :controller do
  context 'user authentication' do
    it 'redirects to login if not logged in' do
      get :home
      expect(response).to redirect_to(new_user_session_path)
    end

    it 'returns a success response if logged in' do
      user = User.create(email: "example@mail.com", password: "multipass")
      sign_in user
      get :home
      expect(response).to be_success
    end
  end
end
```

Slika 3.6. Test ispravnosti autentikacije

U prvom bloku pozvana je metoda za prikazivanje početne stranice aplikacije bez prethodne prijave korisnika i očekivani ishod je presmjeravanje na stranicu za prijavu.

U drugom bloku prije metode za dohvaćanje početne stranice kreira se testni korisnički račun i prijavljuje se u sustav. U ovom testnom bloku očekuje se početna stranica aplikacije kao odgovor od strane poslužitelja.

Testiranje aplikacije pokreće se naredbom *"rspec"* u terminalu unutar direktorija aplikacije i pokrenuti testovi su prikazani zelenom, odnosno crvenom bojom ovisno o tome jesu li uspješni ili ne (slika 3.7.).

```
PagesController
  user authentication
    redirects to login if not logged in
    returns a success response if logged in

Finished in 0.1121 seconds (files took 2.84 seconds to load)
2 examples, 0 failures
```

Slika 3.7. Rezultat provedenog testiranja

Nakon što je sigurno da sustav za autentikaciju korisnika radi kao što je predviđeno, može se krenuti u izradu aplikacije. Najprije će se pristupiti kreiranju svih potrebnih modela aplikacije o kojima se raspravljalo prilikom izrade dijagrama baze podataka. Generiranjem modela pokreće se *Rspec* metoda koja generira datoteku u direktoriju *"spec/models"* koja je predviđena za pisanje testova spomenutog modela. Ako se u ovom trenutku pokrene testiranje dobit će se obavijest kako test još nije definiran.

3.3.1. Kreiranje i testiranje modela

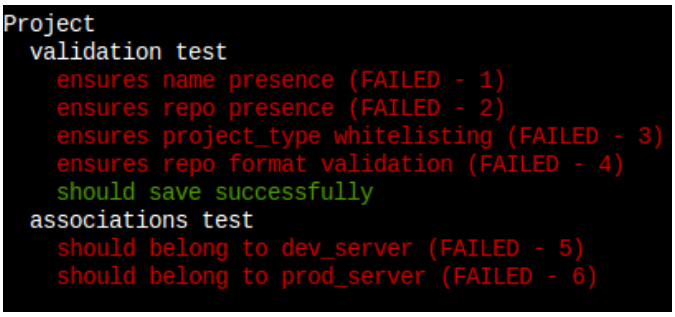
Koristeći TDD korake najprije se pristupa pisanju testova koji će definirati ograničenja i pravila novog modela (slika 3.8.).

```
require 'rails_helper'

RSpec.describe Project, type: :model do

  context 'validation test' do
    it 'ensures name presence' do
      project = Project.new(name: nil, repo: 'http://example_repo.git').save
      expect(project).to eq(false)
    end

    it 'ensures repo presence' do
      project = Project.new(name: "Example Project", repo: nil).save
      expect(project).to eq(false)
    end

    it 'ensures project_type whitelisting' do
      project = Project.new(name: "Example Project", repo: "http://example_repo.git", project_type: "not whitelisted").save
      expect(project).to eq(false)
    end

    it 'ensures repo format validation' do
      project = Project.new(name: "Example Project", repo: "not valid address").save
      expect(project).to eq(false)
    end

    it 'should save successfully' do
      project = Project.new(name: "Example Project", repo: 'https://example_repo.git', project_type: "rails").save
      expect(project).to eq(true)
    end
  end

  context 'associations test' do
    it "should belong to dev_server" do
      assc = described_class.reflect_on_association(:dev_server)
      expect(assc.macro).to eq(:belongs_to)
    end

    it "should belong to prod_server" do
      assc = described_class.reflect_on_association(:prod_server)
      expect(assc.macro).to eq(:belongs_to)
    end

  end

end
```

Slika 3.8. Testovi project modela

Dva su testna *"context"* bloka koja su napisana za projekt model, u prvom se provjeravaju metode kojima se definira valjanost zapisa a u drugom je li dobro definirana veza sa modelom poslužitelja (engl. *"server"*).

Nakon što se u prvom koraku kreiraju potrebni testovi drugi korak je pokretanje istih. Očekuje se da rezultati testiranja budu negativni, pošto nisu definirane metode koje ih trebaju ispunjavati, kako bi bilo sigurno da oni zaista funkcioniraju (Slika 10).



Slika 3.8. Rezultat provedenog testiranja

Sljedeći korak je definiranje metoda koje će zadovoljiti napisane testove. Metode koje su potrebne za provjeru valjanosti zapisa su metoda koja provjerava prisutnost imena projekta (*"name"*) i lokacije izvornog koda (*"repo"*), metoda kojom se dozvoljavaju samo predefinirani nazivi tipa projekta (*"project_type"*) te metoda koja provjerava format unešene adrese lokacije izvornoga koda. Metoda koja provjerava format zapisa koristi se *regularnim izrazom* (engl. *regex*) kojim zahtjeva da tekst *"https://"* mora biti na početku i *".git"* na kraju unešene adrese. Također je potrebno definirati i vezu između projekt modela i modela poslužitelja od kojih će se razlikovati dva tipa - razvojni i produkcijski. Nakon što su definirane sve potrebne metode i veze (slika 3.9.), ponovno pokrenuti testovi trebali bi dati pozitivne rezultate.

```
class Project < ActiveRecord::Base

  belongs_to :dev_server, class_name: "Server", foreign_key: "dev_server_id", optional: true
  belongs_to :prod_server, class_name: "Server", foreign_key: "prod_server_id", optional: true
  has_many :log_actions

  extend Enumerize
  enumerize :project_type, in: [:rails, :static, :react]

  validates_presence_of :name, :repo
  validates_format_of :repo, with: /\Ahttps:\/\/.*\.git\Z/

end
```

Slika 3.9. Project model

Dovršavanjem projekt modela dovršen je jedan cijeli TDD ciklus. Sada je model projekt jasno definiran u smislu da ne dozvoljava spremanje zapisa koji nisu u skladu sa predefiniranim pravilima i ispravno je povezan sa modelom poslužitelja.

3.3.2. Kreiranje i testiranje kontrolera

Nakon modela potrebno je kreirati kontroler putem kojeg će korisnik manipulirati zapisima projekata. Generiranjem kontrolera za projekte *Rspec* generira i datoteku *"spec/controllers/projects_controller_spec.rb"* u kojoj će se pisati potrebni testovi. Standardni *Rails* kontroler sastoji se od 7 metoda:

- **`index`** - vraća HTML dokument sa prikazom svih zapisa modela kojem pripada (u ovom slučaju sve zapise projekata)
- **`show`** - dohvaća pojedinačni zapis i prikazuje njegove detalje HTML dokumentom (ime, poveznica na izvorni kôd, pripadajuće poslužitelje itd.)
- **`new`** - inicijalizira novu instancu klase i prodsljeđuje je HTML dokumentom koji sadrži obrazac za definiranje vrijednosti njezinih atributa.
- **`create`** - prima vrijednosti atributa definiranih obrascem navedenim u prethodnoj metodi te kreira novi zapis ukoliko su parametri u skladu sa pravilima modela. Ne prosljeđuje HTML dokument nego preusmjerava na prikaz pojedinog zapisa (**`show`**)
- **`edit`** - dohvaća traženi zapis i prikazuje vrijednosti njegovih atributa u poljima obrasca unutar HTML dokumenta.
- **`update`** - prima i sprema izmjene zapisa i preusmjerava na **`show`** metodu.
- **`destroy`** - prosljeđeni zapis briše iz baze podataka i preusmjerava na **`index`** metodu.

Metode *Rails* kontrolera ne moraju "odgovarati" prosljeđivanjem HTML dokumenta, iste metode svoje rezultate mogu slati u obliku JavaScript naredbi ili JSON datoteke. Isto tako, gore navedeni odzivi **`create`** i **`update`** metoda navedeni su samo za slučajeve ukoliko se uspješno izvrše, u suprotnom one preusmjeravaju korisnika na **`new`** i **`edit`** metodu ukazujući na pogreške koje su spriječile uspješno spremanje zapisa.

Sada kada je poznato kako se metode kontrolera trebaju "ponašati" vrijeme je da se kreiraju i njihovi testovi. Započeti će se od **`index`** metode koja ima zadaću dohvatiti sve dosada spremljene zapise projekata i prosljediti ih u obliku HTML dokumenta korisnikovom internetskom pregledniku. Kako bi se prilikom testiranja imao pristup kontroleru prije svakog testiranja potrebno je kreirati testnog korisnika te ga prijaviti. To se lako može izvesti **`before(:each)`** blokom koji će prije svakog testnog bloka pokrenuti kôd koji u sebi sadrži.

Testom napisanim za **`index`** metodu (slika 3.10.) provjeriti će se jesu li svi zapisi projekata dohvaćeni i prosljeđen li je **`index.html`** dokument krajnjem korisniku.

```
RSpec.describe ProjectsController, type: :controller do

  before(:each) do
    user = User.create(email: "example@mail.com", password: "multipass")
    sign_in user
  end

  context 'GET #index' do
    it 'populates an collection of projects' do
      3.times do |i|
        Project.create(name: "Project #{i}", repo: "https://project_repo.git")
      end
      projects = Project.all
      get :index
      expect(assigns(:projects)).to eq(projects)
    end

    it "renders the :index view" do
      get :index
      expect(response).to render_template :index
    end
  end
```

Slika 3.10. Testiranje index metode

Sljedeći korak je napisati odgovarajuću **`index`** metodu koja ispunjava zahtjeve napisanog testa. Ova kratka metoda sastoji se samo od jedne naredbe **`@projects = Project.all`** i ne sadrži naredbu kojom prosljeđuje HTML dokument jer *Rails* razvojno okruženje traži istoimeni HTML dokument ("*index.html*") ukoliko drugi odziv nije definiran.

Metodu za prikaz pojedinog zapisa testiramo na sličan način kao i **`index`** metodu, jedina je razlika što je njezin zadatak prikazati točno traženi zapis (slika 3.11.).

```
  context 'GET #show' do
    before(:each) do
      @project = Project.create(name: "Example Project", repo: "https://project_repo.git")
    end

    it 'sets project' do
      get :show, params: {id: @project.to_param}
      expect(assigns(:project)).to eq(@project)
    end

    it "renders the :show view" do
      get :show, params: {id: @project.to_param}
      expect(response).to render_template :show
    end

  end
```

Slika 3.11. Testiranje show metode

Može se primjetiti kako se prije svakog testiranja kreiraju novi testni podaci, to je zato što se oni obrišu nakon svakog pojedinog testiranja kako ti isti podaci ne bi utjecali na sve testove koji slijede.

`new` metoda ima zadatak kreirati novu instancu klase **`Project`** i prosljediti je korisniku *"new.html"* dokumentom. Testom se poziva metoda i provjerava postoji li u njezinom odgovoru instanca **`Project`** klase kao i prikazuje li se dokument (slika 3.12.).

```
  context 'GET #new' do
    it 'initializes new instance of Project' do
      get :new
      expect(assigns(:project)).to be_an_instance_of Project
    end

    it "renders the :new view" do
      get :new
      expect(response).to render_template :new
    end
  end
```

Slika 3.12. Testiranje new metode

Kôd metode **`new`** sastoji se od samo jedne naredbe **`@project = Project.new`**. Pri kreiranju varijable ili objekta unutar kontrolera u slučaju da ih se želi prikazati korisniku u HTML dokumentu potrebno ih je imenovati sa početnim znakom "@" u suprotnom oni će biti vidljivi samo unutar metode u kojoj se nalaze.

Nakon što je korisnik unio vrijednosti atributa u obrazac i prosljedio ih metodi za kreiranje **`create`**, potrebno je unešene atribute provjeriti i sukladno tome poduzeti odgovarajuće korake. Ovo testiranje podijeliti će se u dva scenarija:

- Pokušaj spremanja sa nevažećim atributima
- Pokušaj spremanja sa važećim atributima

U prvom slučaju potrebno je testirati prepoznavanje nevažećih atributa i preusmjeriti korisnika natrag na obrazac. U drugom slučaju provjerava se spremljen li je zapis u bazu podataka i dobio li je korisnik odgovarajući odziv preusmjeravanjem na metodu **`show`**.

Testiranjem spremanja sa važećim atributima provjerava se je li se promijenio broj zapisa projekata za točno jedan zapis i izvršeno li je preusmjeravanje korisnika na pregled zadnje dodanog zapisa. Isto tako, testiranje u slučaju nevažećih atributa vrši se tako da je potrebno

uvjeriti se kako je broj zapisa ostao neizmjenjen nakon izvršene **`create`** metode te kako je korisniku ponovo prikazan obrazac (slika 3.13.).

```
describe 'POST #create' do
  context 'with valid attributes' do

    it 'creates new project' do
      project = Project.new(name: "Example Project", repo: "https://project_repo.git", project_type: "rails")
      expect {
        post :create, params: { project: project.as_json }
      }.to change(Project, :count).by(1)
    end

    it 'redirects to the new project' do
      project = Project.new(name: "Example Project", repo: "https://project_repo.git", project_type: "rails")
      post :create, params: { project: project.as_json }
      expect(response).to redirect_to(Project.last)
    end
  end

  context "with invalid attributes" do
    it "does not save the new contact" do
      project = Project.new(name: "Example Project", project_type: "rails")
      expect {
        post :create, params: { project: project.as_json }
      }.to_not change(Project, :count)
    end

    it "re-renders the new method" do
      post :create, params: { project: Project.new(name: "Example Project", project_type: "rails").as_json }
      expect(response).to render_template :new
    end
  end
end
```

Slika 3.13. Testiranje create metode

Primjećuje se kako je kôd napisanih testova lako čitljiv i razumljiv i neki dijelovi poput **`expect(response).to render_template :new`** zvuče čak i kao smislene rečenice.

Sada kada je napisan kôd za testiranje **`create`** metode pristupa se pisanju njezinog koda. Kako bi se zadovoljio prvi test potrebno je kreirati novu instancu klase **`Project`** i izjednačiti joj atribute sa vrijednostima koje su prosljeđene putem obrasca. Sljedeći korak biti će spremanje zapisa u bazu podataka te preusmjeravanje korisnika na prikaz kreiranog zapisa (**`show`**) obavještavajući ga porukom da je zapis uspješno spremljen.

Kako bi i drugi dio testa bio uspješan, definira se uvjet koji je ispunjen kada zapis nije uspješno spremljen i preusmjerava korisnika na *"new.html"*. Nakon što se svi testovi pokažu uspješnima **`create`** metoda je dovršena (slika 3.14.).

```
def create
  @project = Project.new(project_params)

  if @project.save
    redirect_to @project, notice: 'Project was successfully created.'
  else
    render :new
  end
end
```

Slika 3.14. Create metoda

Testovi metoda **`edit`** i **`update`** vrlo su slični testovima za **`new`** i **`create`**, jedina razlika je što se umjesto inicijalizacije nove instance dohvaća već spremljeni zapis kako bi korisnik izmijenio njegove atribute (slika 3.15.).

```
describe 'PUT #update' do
  before(:each) do
    @project = Project.create(name: "Example Project",
                              repo: "https://project_repo.git",
                              project_type: "rails")
  end

  context "valid attributes" do
    it "located the requested @project" do
      put :update, params: { id: @project, project:
                             { name: "Example Project",
                               repo: "https://new_repo.git",
                               project_type: "rails" }
                           }
      expect(assigns(:project)).to eq(@project)
    end

    it "changes @project's attributes" do
      put :update, params: {
        id: @project,
        project: { name: "New Name", repo: "https://new_repo.git", project_type: "static" }
      }
      @project.reload
      expect(@project.name).to eq("New Name")
      expect(@project.repo).to eq("https://new_repo.git")
      expect(@project.project_type).to eq("static")
    end

    it "redirects to the updated contact" do
      put :update, params: {
        id: @project,
        project: { name: "New Name", repo: "https://new_repo.git", project_type: "static" }
      }
      expect(response).to redirect_to @project
    end
  end
  context "invalid attributes" do
    it "located the requested @project" do
      put :update, params: { id: @project, project:
                             { name: "Example Project",
                               repo: "https://new_repo.git",
                               project_type: "rails" }
                           }
      expect(assigns(:project)).to eq(@project)
    end

    it "does not change @project's attributes" do
      put :update, params: {
        id: @project,
        project: { name: "New Name", repo: "invalid_repo", project_type: "static" }
      }
      @project.reload
      expect(@project.name).to_not eq("New Name")
      expect(@project.repo).to_not eq("https://new_repo.git")
      expect(@project.project_type).to_not eq("static")
    end

    it "re-renders the edit method" do
      put :update, params: {
        id: @project,
        project: { name: "New Name", repo: "https://new_repo.git", project_type: "invalid_type" }
      }
      expect(response).to render_template :edit
    end
  end
end
```

Slika 3.15. Testiranje uopdate metode

Preostala je još samo **`destroy`** metoda koja ima zadaću obrisati zapis iz tablice projekata i po svom završetku preusmjeriti na **`index`** metodu. Testom se želi uvjeriti kako je nakon brisanja zapisa ukupan broj preostalih umanjen za točno jedan zapis (slika 3.16.).

```
describe 'DELETE #destroy' do
  before :each do
    @project = Project.create(name: "Example Project",
                              repo: "https://project_repo.git",
                              project_type: "rails")
  end

  it "deletes the contact" do
    expect{
      delete :destroy, params: { id: @project }
    }.to change(Project,:count).by(-1)
  end

  it "redirects to contacts#index" do
    delete :destroy, params: { id: @project }
    expect(response).to redirect_to projects_url
  end
end
```

Slika 3.16. Testiranje destroy metode

Kreiranje metode po metode vođeno napisanim testovima dovršen je kontroler za projekte (slika 3.17.) za koji je sigurno da, bez da je isproban putem internetskog preglednika, radi u potpunosti. Važno je napomenuti da se u svakom koraku kreiralo odgovarajuće HTML dokumente *("index.html"*, "*show.html"* i dr.). Nakon izrade kontrolera za projekte, kreće se u izradu kontrolera za poslužitelje i cijeli proces kreće ispočetka.

Na kraju procesa izrade kontrolera prebrojavanjem redova napisanog koda uočeno je kako je za testove bilo potrebno čak 4 puta više teksta nego za kôd koji je testiran što bi mnoge razvojne timove moglo obeshrabriti u namjeri da svoje testove automatiziraju. Uzimajući u obzir kako se većina kontrolera razlikuje samo u ključnim riječima jasno je da prilikom izrade sljedećeg, vrijeme za pisanje testova će se znatno skratiti jer je ove već napisane moguće ponovno iskoristiti.

```ruby
class ProjectsController < ApplicationController
  before_action :set_project, only: [:show, :edit, :update, :destroy]

  def index
    @projects = Project.all
  end

  def show
    @deployment_history = @project.log_actions.order(created_at: :desc)
  end

  def new
    @project = Project.new
  end

  def edit
  end

  def create
    @project = Project.new(project_params)

    if @project.save
      redirect_to @project, notice: 'Project was successfully created.'
    else
      render :new
    end
  end

  def update
    if @project.update(project_params)
      redirect_to @project, notice: 'Project was successfully updated.'
    else
      render :edit
    end
  end

  def destroy
    @project.destroy
    redirect_to projects_url, notice: 'Project was successfully destroyed.'
  end

  private
    def set_project
      @project = Project.find(params[:id])
    end

    def project_params
      params.require(:project).permit(:name, :repo, :project_type)
    end
end
```

Slika 3.17. Izgled project kontrolera

3.4. Cucumber

Cucumber je sustav za testiranje više razine kojemu je cilj premostiti raskorak između klijentovih zahtjeva i tehničke izvedbe traženog programskog riješenja. Osnovno obilježje ovog sustava je da omogućava oblikovanje testova u smislene rečenice koje će ponekad nedovoljno razumljive zahtjeve "pojasniti" razvojnom timu kako bi se sa sigurnošću zadovoljile potrebe klijenta. Kada se kaže da je to sustav više razine, želi se reći kako testiranje obuhvaća složenije funkcionalnosti aplikacije. Sustavom za proširenja *Cucumber* se lako ugrađuje u *Rails* aplikaciju.

Za primjer ove metode testiranja kreirana je datoteka *"manage_projects.feature"* koja je zadužena za testiranje upravljanja projekatima (slika 3.18.).

```
Feature: Manage Projects
  In order to make a poject
  As an user
  I want to create and manage projects

  Scenario: Projects List
    Given I have projects named My First Project, My Second Project
    When I go to the list of projects
    Then I should see "My First Project"
    And I should see "My Second Project"

  Scenario: Create Valid Project
    Given I have no projects
    And I am on the list of projects
    When I follow "New Project"
    And I fill in "Name" with "Project Example"
    And I fill in "Repo" with "https://my_repo.git"
    And I press "Create"
    Then I should see "New project created."
    And I should see "Project Example"
    And I should see "https://my_repo.git"
    And I should have 1 article
```

Slika 3.18. Testiranje upravljanja projektima

Ovaj iznimno razumljivi kôd pravi je primjer BDD-a koji rečenice engleskog jezika oblikuje u kôd za testiranje funkcionalnosti. Ovakav način testiranja iziskuje predefinirane funkcije kako bi ovako napisan test mogao funkcionirati. Pomoćne funkcije najčešće se pišu uz pomoć *regularnih izraza* kojima se traže uzorci u tekstu i zamjenjuje ih se, u ovom slučaju, odgovarajućim poveznicama na stranice (slika 3.19.).

```
def path_to(page_name)
  case page_name

  when /the login page/
    new_user_session_path

  when /the homepage/
    root_path

  when /the list of projects/
    projects_path

  when /the list of servers/
    servers_path

  else
    raise "Can't find mapping from \"#{page_name}\" to a path."
  end
end
```

Slika 3.19. Prevođenje ruta za usmjeravanje

Ovakve pomoćne funkcije potrebne su kako bi sustav testiranja mogao napisane izraze pokretati u obliku programskog koda što ovaj sustav čini složenijim nego li se to na prvi pogled moglo primjetiti (slika 3.20.).

```
Given /^I have projects named (.+)$/ do |names|
  names.split(', ').each do |name|
    Project.create(name: name, repo: "https://my_repo.git")
  end
end

Given /^I have no projects$/ do
  Project.delete_all
end

Then /^I should have ([0-9]+) projects?$/ do |count|
  Project.count.should == count.to_i
end
```

Slika 3.20. Prevođenje izraza za testove

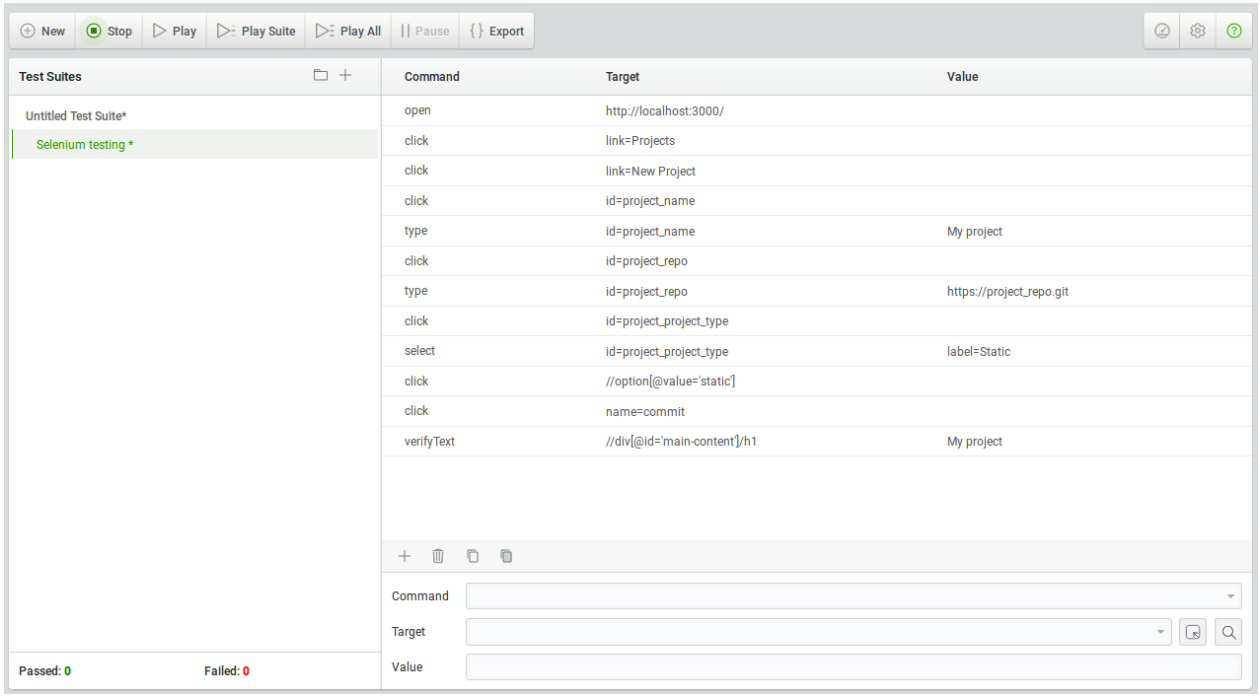
3.5. Automatizacija testiranja grafičkog sučelja

Testiranjem aplikacija pokušava se imitirati stvarnog korisnika kako bi se unaprijedilo korisničko iskustvo. Ručno i automatizirano testiranje imaju svoje očite prednosti i nedostatke i uvijek se pokušava pronaći riješenje koje će od ove dvije metode uzeti ono najbolje. Jedno od rješenja koje ove dvije metode pokušava objediniti je automatizacija testiranja grafičkog sučelja. Ovakva metoda simulirati će stvarnog korisnika u toj mjeri da će pokrenuti internetski preglednik, unositi vrijednosti u polja obrazaca i klikati pokazivačem miša po zaslonu.

Jedan od primjera sustava za takvo testiranje je i *Selenium Web Driver* koji podržava upravljanje većinom modernih preglednika (*Chrome, Firefox, Internet Explorer, Safari* i dr.). Osim standardnog ručnog unošenja redova koda za testiranje, podržava i snimanje testa koji je kasnije moguće automatski obavljati.

Selenium IDE je proširenje za web preglednik koje pokrenuto bilježi sve korisnikove aktivnosti u listu koraka (slika 3.21.). Takvu listu moguće je izvesti (engl. *export*) pisanu u Ruby programskom jeziku koju možemo pokretati *WebDriver* i *Rspec* proširenjima. Ovakav

način pruža mogućnost da se testiranje koje je obavio stvarni korisnik obavi kad god je to potrebno, brzo i jednostavno.



Slika 3.21. Sučelje za snimanje ručnog testiranja

4. Zaključak

Testiranjem aplikacija osigurava se kvaliteta proizvoda i uvjerava da je aplikacija baš onakva kako je opisana u početnim specifikacijama. Mnogo je vremena potrebno uložiti kako bi se steklo znanje pisanja kvalitetnih testova i to je ono što mnoge razvojne timove obeshrabruje u poduzimanju tog koraka.

Za promatrani projekt najidealnijom metodom pokazala se TDD metoda jer se aplikacija ne radi za klijenta nego isključivo za potrebe razvojnog tima. Pošto su predviđene buduće izmjene koje će utjecati na najmanje funkcionalnosti koda ovakvo granularno testiranje je najzahvalnije.

Zaključak je da kvalitetno napisani testovi omogućavaju bezbrižnu izmjenu i proširivanje postojećeg koda, ručno testiranje rezultira stvarnim korisničkim iskustvom, TDD usmjerava u izradi aplikacije a BDD pomaže u boljem razumjevanju korisničkih zahtjeva. Svaka od metoda iziskuje resurse i ne postoji univerzalni odgovor koja je razina testiranja u kojem slučaju najisplativija.

LITERATURA

[1] Everything You Need to Know About Software Testing Methods, https://www.thebalance.com/ (siječanj 2018.)

[2] Types and Levels of Testing in Programming, http://www.exforsys.com/ (siječanj 2018.)

[3] Software Testing Tutorial, https://www.tutorialspoint.com/ (siječanj 2018.)

[4] Test Driven Development vs. Behavior Driven Development, https://www.glowtouch.com/ (siječanj 2018.)

[5] Rails guides, http://guides.Rubyonrails.org/ (siječanj 2018.)

[6] Rspec documentation, http://rspec.info/ (siječanj 2018.)

[7] Redis documentation, https://redis.io/documentation (siječanj 2018.)

[8] Cucumber, https://cucumber.io/ (siječanj 2018.)

SAŽETAK

U ovom završnom radu osnovni cilj je bio istražiti najčešće načine testiranja programske podrške te njihove prednosti i nedostatke. Primjenjivajući metode testiranja na stvarnom projektu bilježeni su rezultati observacija i donošenja zaključaka.

Veći dio završnog rada tehničke je prirode i bavio se implementacijom programskog koda testiranja. U radu su opisane metode testiranja *razvoj upogonjen testiranjem*, *razvoj upogonjen ponašanjem* i *automatizirano testiranje grafičkog sučelja*. Opisane metode implementirane su u programskom kodu te je prikazano na koji način primjena ovih metoda potiče izradu kvalitetnije programske podrške.

Ključne riječi:

Ruby on Rails, testiranje programske podrške, Rspec

ABSTRACT

The main goal of this project was the research of the most commonly used software testing methods and their advantages and disadvantages. Testing methods were implemented on an actual software project. The observations and conclusions were drawn and noted during the testing implementation.

Majority of the project is technical by nature and is based around the implementation of the testing code. The project describes test driven development, behavior driven development and automated graphic user interface testing. The described methods are implemented in software code which showcases the ways their implementations improve the quality of the software being developed.

Keywords:

Ruby on Rails, software testing, Rspec

ŽIVOTOPIS

Vedran Mijatović rođen je 18. Listopada 1984. U Osijeku. U Elektrotehničkoj i prometnoj školi Osijek završio je smijerove elektromonter i tehničar za mehatroniku. Trenutno je izvanredni student treće godine stručnog studija elektrotehnike, smjer informatika.

Tijekom studija bio je demonstrator iz kolegija fizika i 2015. nagrađen je za postignuti uspjeh u studiranju.

Od veljače 2016. zaposlen je kao student u firmi Bamboo lab na mjestu backend developera gdje je uspješno radio na projektima kao što su izrada web stranica Gold by Waldinger, Rastavljeni, belisce.2roam i dr.