Univerza v Ljubljani
Fakulteta za računalništvo in informatiko

Jaka Šušteršič

Zvezna postavitev spletnih aplikacij

diplomsko delo

univerzitetni študijski program prve stopnje
računalništvo in informatika

Mentor: izr. prof. dr. Viljan Mahnič

Ljubljana 2015

Besedilo je oblikovano z urejevalnikom besedil LaTeX.

Fakulteta za računalništvo in informatiko izdaja naslednjo nalogo:

Tematika naloge:

Proučite značilnosti zvezne integracije, zvezne dostave in zvezne postavitve aplikacij ter orodja, ki se uporabljajo v ta namen. Na podlagi tega avtomatizirajte postopek zvezne postavitve za obstoječo aplikacijo, ki je napisana v ogrodju Ruby on Rails in omogoča obveščanje uporabnikov o prometnih dogodkih na izbranem območju v realnem času. Rešitev naj obsega vse potrebne korake od oddaje spremenjene kode v repozitorij do zagona nove verzije aplikacije v produkcijskem okolju.

Izjava o avtorstvu diplomskega dela

Spodaj podpisani Jaka Šušteršič sem avtor diplomskega dela z naslovom:

Zvezna postavitev spletnih aplikacij

S svojim podpisom zagotavljam, da:

- sem diplomsko delo izdelal samostojno pod mentorstvom izr. prof. dr. Viljana Mahniča,
- so elektronska oblika diplomskega dela, naslov (slov., angl.), povzetek (slov., angl.) ter ključne besede (slov., angl.) identični s tiskano obliko diplomskega dela,
- soglašam z javno objavo elektronske oblike diplomskega dela na svetovnem spletu preko univerzitetnega spletnega arhiva.

V Ljubljani, dne 20. avgusta 2015 Podpis avtorja:

Zahvaljujem se mentorju izr. prof. dr. Viljanu Mahniču za pomoč pri izdelavi diplomskega dela. Prav tako se zahvaljujem svoji družini za vso podporo tekom celotnega študija. Zahvaljujem se tudi Manci in vsem svojim študijskim prijateljem, ki so me spodbujali in mi stali ob strani.

Družini in vsem, ki so me podpirali in spodbujali pri študiju.

Kazalo

Seznam uporabljenih kratic

kratica	angleško	slovensko
HTML	hypertext markup language	označevalni jezik za oblikovanje večpredstavnostnih dokumentov
HTTP	hypertext transfer protocol	protokol za prenos hiperteksta
IP	internet protocol	internetni protokol
JSON	JavaScript object notation	objektna notacija JavaScript
SSH	secure shell	varna lupina
SSL	secure socket layer	sloj varnih vtičnic
SQL	structured query language	strukturiran povpraševalni jezik
VPS	virtual private server	navidezni zasebni strežnik

Povzetek

Postopek izdaje programske opreme, na kateri dela več razvijalcev, je pogosto časovno potraten proces, ki obenem povečuje tveganje za vnos napak. Za hitro in zanesljivo dostavo spletnih aplikacij končnim uporabnikom se zato v praksi pogosto uporablja postopek zvezne postavitve. V diplomski nalogi smo implementirali postopek zvezne postavitve za obstoječo aplikacijo, namenjeno obveščanju o prometnih dogodkih, ki je napisana v ogrodju Ruby on Rails. Postavitev aplikacijske infrastrukture je prvotno zahtevala ročno vnašanje okrog 50 ukazov in je v povprečju trajala pol ure. S prenosom na ustrezna orodja za avtomatizacijo postopka postavitve aplikacijske infrastrukture nam je razvijalčevo delo uspelo zmanjšati na vnos zgolj enega ukaza, trajanje postavitve infrastrukture pa na okrog 10 minut. Prav tako smo z avtomatizacijo za približno tretjino zmanjšali tudi trajanje postavitve aplikacije. S tem smo razbremenili razvijalce in zmanjšali stroške režijskega dela, kar jim je omogočilo hitrejšo in pogostejšo dostavo funkcionalnosti končnim uporabnikom aplikacije.

Ključne besede: razvoj programske opreme, spletne aplikacije, zvezna dostava, zvezna postavitev.

Abstract

The process of releasing new versions of software, worked on by multiple developers, is often a time consuming and risky proposition. Continuous deployment is often used to guarantee reliable and quick delivery of web applications to end users. In our thesis we implemented continuous deployment for an existing application, used for notifying users of traffic events, which is built using the Ruby on Rails framework. The task of provisioning infrastructure originally required manual input of about 50 commands and took an average of 30 minutes to execute. By using tools which automate provisioning of application infrastructure we managed to reduce the developer's workload to only one command, which only takes about 10 minutes to execute. Using automation also reduced the time needed to deploy our application by a third. Achieving this allowed us to decrease developer's workload even further and reduce deployment overhead. Consequently, it enabled faster and more frequent deployment of new application functionality to the end user.

Keywords: software development, web application, continuous delivery, continuous deployment.

Poglavje 1

Uvod

Postopek izdaje programske opreme, na kateri dela več razvijalcev, je pogosto tvegan in časovno potraten proces [1]. V zadnjih letih zato v skupinah razvijalcev opažamo trend uporabe zvezne integracije (angl. *continuous integration*), zvezne dostave (angl. *continuous delivery*) in zvezne postavitve (angl. *continuous deployment*), ki rešujejo širok nabor izzivov v procesu razvoja in postavitve aplikacij. Uporaba navedenih praks omogoča zmanjšanje časa od oddaje novih delov programske kode v repozitorij do uporabe funkcionalnosti, ki jih ti implementirajo, v produkcijskem okolju. Avtomatizacija vseh potrebnih korakov za postavitev aplikacije vodi v pohitritev razvoja programske opreme in znižanje stroškov. Nove funkcionalnosti se v tem primeru ne izdajajo več v tedenskih ciklih, ampak takoj ko so pripravljene. Na ta način so razvijalci manj obremenjeni zaradi strogih časovnih rokov postavitve [2]. Zanesljive in manj tvegane izdaje programske opreme obenem omogočajo nenehno prilagajanje produkta (aplikacije) glede na povratne informacije uporabnikov [3], premike trga in morebitne spremembe poslovne strategije.

Za pridobitev konkurenčne prednosti omenjene prakse uporablja večina velikih tehnoloških podjetij kot so Facebook [4], Google [5], Netflix [6] in Amazon. Podjetje Amazon je razkrilo, da novo verzijo njihove spletne aplikacije v produkcijskem okolju v povprečju postavijo na vsakih 11.6 sekund, v

eni uri pa so zabeležili največ 1079 postavitev. Z uporabo zvezne postavitve so od leta 2006 zmanjšali izpade storitev zaradi postavitev za 75%, celoten čas izpadov zaradi postavitev pa za 90% [7].

V sklopu diplomskega dela želimo implementirati postopek zvezne postavitve za obstoječo spletno aplikacijo, ki je namenjena obveščanju o prometnih dogodkih. Med razvojem aplikacije smo namreč prepoznali več možnosti za izboljšavo postopkov dostave produkta končnim uporabnikom. Osredotočili se bomo predvsem na razbremenitev dela razvijalca pri nalogah postavitve infrastrukture in aplikacije. Prehod novih funkcionalnosti od oddaje izvorne kode v repozitorij do dostave končnim uporabnikom želimo čim bolj avtomatizirati in tako tudi lažje in hitreje slediti premikom trga.

V drugem poglavju bomo predstavili izbrano območje in glavna uporabljena orodja ter koncepte. Sledila bo predstavitev zgradbe obstoječe aplikacije in pa glavnih izzivov tekom razvoja. Definirali bomo tudi cilje, ki jih želimo z implementacijo zvezne postavitve doseči. Naslednje poglavje bo osredotočeno na samo implementacijo zvezne postavitve, in bo vsebovalo evalvacijo trenutnega stanja aplikacije, dopolnitve nabora testov, avtomatizacijo postavitve okolja, avtomatizacijo postavitve aplikacije, podatkovne migracije ter podroben prikaz postopkov storitve za zvezno integracijo. V predzadnjem poglavju bo predstavljena uporaba zvezne postavitve s strani razvijalcev, zadnje poglavje pa bo namenjeno zaključnim sklepom.

Poglavje 2

Predstavitev področja

2.1 Zvezna integracija

Zvezna integracija je način razvoja programske opreme, ki poskuša rešiti problem integracije izvorne kode večih razvijalcev. Kot prvo ključno pravilo izpostavlja integracijo dela vseh razvijalcev večkrat na dan. S povečevanjem pretečenega časa se namreč povečuje verjetnost, da bo vsaka nova lokalna kopija izvorne kode projekta (vsak razvijalec ima običajno svojo) v konfliktu s kopijo drugega razvijalca. Skupine razvijalcev, ki izvorno kodo integrirajo redko, imajo tako več težav, saj v sistem vnašajo večje spremembe izvorne kode. Poleg tega za odkrivanje napak porabijo več časa, vsaka integracija pa postane naporno opravilo. Ob pojavu ekstremnega programiranja (angl. *extreme programming*), ki je ena izmed priljubljenih agilnih metod razvoja programske opreme, je opisani problem dobil sinonim integracijski pekel (angl. *integration hell*) [8]. Zvezna integracija v sklopu ekstremnega programiranja predlaga integriranje več desetkrat na dan - Jez Humble, eden izmed vodilnih avtorjev del s področja zvezne postavitve, v svojem delu [10] omenja, da je "zvezno bolj pogosto, kot bi si ljudje mislili".

Drugo ključno pravilo zvezne integracije je testiranje aplikacije ob vsaki spremembi glavne izvorne kode. Namen testov je hitro in enostavno odkrivanje napak in s tem preverjanje pravilnega delovanja aplikacije. V primeru,

da je rezultat izvajanja katerega koli izmed testov neuspešen, je celotna razvojna skupina odgovorna, da v najkrajšem možnem času odpravi napake, pa čeprav to lahko pomeni prenehanje z delom na tekočih opravilih [8]. Aplikacija je tako večino časa razvoja v delujočem stanju, hkrati pa imajo razvijalci večje zaupanje v pravilnost delovanja aplikacije. Testi se najpogosteje izvajajo na temu namenjenemu strežniku za zvezno integracijo, ki zazna vsako spremembo kode in izvede zaporedje korakov, potrebnih za zagon testov in aplikacije.

2.2 Zvezna dostava

Zvezna dostava je princip razvoja aplikacij, ki zagotavlja možnost dostave aplikacije končnim uporabnikom v vsakem trenutku [9]. Izbira časovnega okvira postavitve nove verzije aplikacije tako ni več tehnični problem, ampak poslovna odločitev.

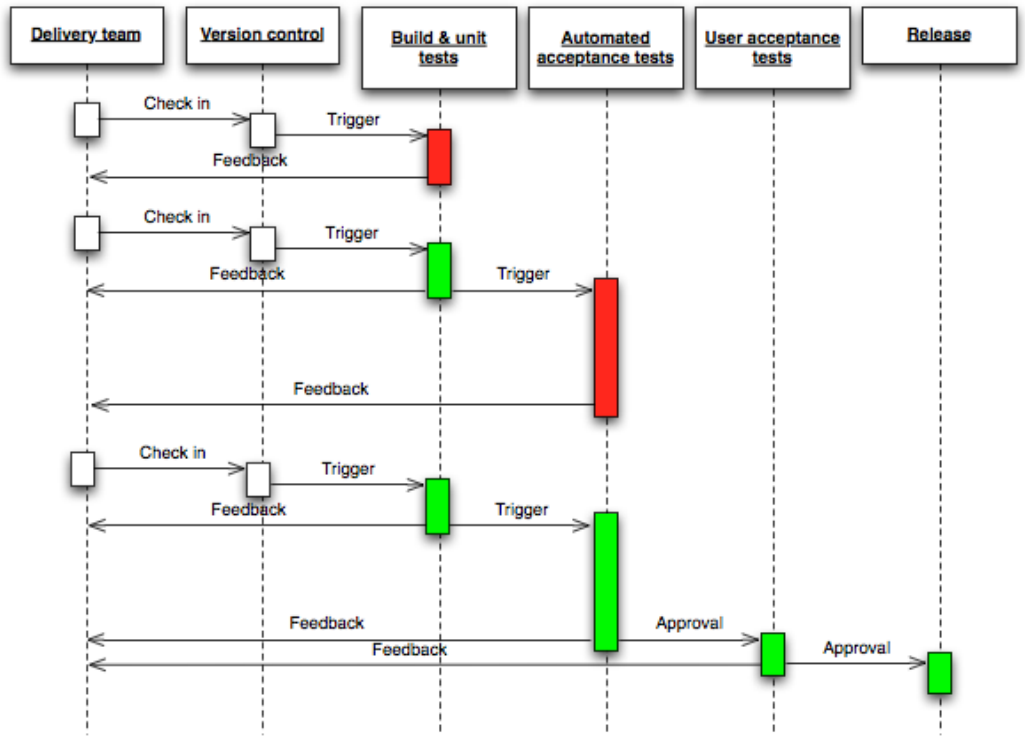
Zvezna dostava nadgrajuje zvezno integracijo z uvedbo dostavnega cevovoda (angl. *deployment pipeline*). Dostavni cevovod je abstrakcija vseh posameznih korakov prehoda na novo verzijo aplikacije od oddaje izvorne kode v repozitorij do njene uporabe v produkcijskem okolju [10]. V širšem smislu je naloga dostavnega cevovoda zaznavanje sprememb, ki bi lahko v produkcijskem okolju pripeljale do napak. Poleg preprostih programskih hroščev namreč lahko pride tudi do zmanjšanja zmogljivosti aplikacije in težav z varnostjo, ki so pogosto kritične. Dostavni cevovod mora omogočati jasen pregled nad tokom sprememb sistema in sodelovanje vseh skupin, ki so vključene v dostavo programske opreme [11].

Dostavni cevovod modeliramo z uporabo zaporedja korakov preverjanja pravilnosti (validacij), ki jih mora aplikacija prestati, preden gre lahko v novo izdajo. Uspešen prehod skozi vsako naslednjo stopnjo cevovoda poveča zaupanje v pravilnost delovanja trenutne verzije aplikacije. Stopnje dostavnega cevovoda običajno uredimo po naraščajočem času izvajanja. Zgodnje stopnje cevovoda najdejo večino problemov in nam nudijo hitro povratno informa-

cijo, poznejše stopnje pa pogosto vsebujejo počasnejše sprejemne teste ali pa celo ročno preverjanje pravilnosti delovanja aplikacije. Posamezne stopnje se lahko izvajajo avtomatizirano ali pa zahtevajo ročno avtorizacijo razvijalca.

Prva stopnja dostavnega cevovoda je namenjena izgradnji aplikacije in poganjanju testov enot (angl. *unit tests*). Naslednji dve stopnji vsebujeta integracijske (angl. *integration*) in sprejemne (angl. *acceptance*) teste. Bolj podrobni testi v naslednjih stopnjah vsebujejo tako ročne kot tudi avtomatizirane teste v vmesnem (angl. *staging*) okolju, ki naj bi bilo čim bolj podobno produkcijskemu okolju. V kolikor v nobeni od stopenj cevovoda ne pride do napake pri izvajanju testov, lahko v zadnji stopnji cevoda novo verzijo aplikacije postavimo v produkcijsko okolje.

Diagram dostavnega cevovoda s tipičnimi stopnjami prikazuje slika 2.1.



Slika 2.1: Diagram dostavnega cevovoda [10].

2.3 Zvezna postavitev

Zvezna postavitev dopolnjuje zvezno dostavo z avtomatizacijo postavitve vsake nove verzije aplikacije v produkcijsko okolje. Vsaka sprememba izvorne kode, ki uspešno pride skozi vse stopnje dostavnega cevovoda, se brez posredovanja razvijalcev postavi v produkcijsko okolje in je tako na voljo uporabnikom. Za razliko od zvezne dostave, kjer mora biti vsaka sprememba glavne izvorne kode v vsakem trenutku v obliki, ki je primerna za dostavo končnim uporabnikom, uporaba zvezne postavitve narekuje avtomatizirano postavitev aplikacije ob vsaki novi spremembi izvorne kode.

Uporaba tehnik za avtomatizirano integracijo, testiranje in postavitev omogoča razvijalcem hiter, zanesljiv ter ponovljiv postopek dostave novih funkcionalnosti končnim uporabnikom. Zaradi pogostosti izvajanja postopka ima izvedba le-tega nizko tveganje in majhne stroške režijskega dela [10]. Nove funkcionalnosti se v tem primeru ne izdajajo več v tedenskih ciklih, ampak takoj ko so pripravljene. Na ta način so razvijalci manj obremenjeni zaradi strogih časovnih rokov postavitve [2]. Zanesljive in manj tvegane izdaje programske opreme obenem omogočajo nenehno prilagajanje aplikacije glede na povratne informacije uporabnikov [3], premike trga in morebitne spremembe poslovne strategije.

Čeprav ima uvedba zvezne postavitve številne pozitivne učinke, njena uporaba mnogim razvijalcem predstavlja izziv. Zvezna postavitev namreč zahteva uvedbo avtomatizacije celotnega postopka postavitve aplikacije, kar je v praksi lahko problematično, saj običajno zajema veliko število netrivialnih korakov. Prav tako pa nenehno spreminjanje aplikacije v produkcijskem okolju zahteva pozorno spremljanje parametrov delovanja. Kljub obsežnemu testiranju delovanja aplikacije lahko namreč v produkcijskem okolju naletimo na programske hrošče, vendar lahko s spremljanjem ključnih poslovnih parametrov (na primer števila novih registracij ali števila nakupov) hitro zaznamo izpad storitve in se nanj ustrezno odzovemo.

Ko v novi verziji aplikacije odkrijemo napako, želimo imeti na voljo preprosta orodja za prehod na prejšnjo verzijo. Ta orodja morajo poleg prenosa

izvorne kode prejšnje verzije zagotavljati tudi obnovitev sheme podatkovne baze in vse korake konfiguracije strežnikov.

Pri uporabi zvezne postavitve morajo biti razvijalci pozorni na to, da se v produkcijsko okolje sproti postavljajo vse spremembe repozitorija, vključno z morebitnimi nedokončanimi funkcionalnostmi. V tem primeru je potrebna uporaba stikal funckionalnosti (angl. *feature toggles*), s pomočjo katerih je mogoče v zadnjo verzijo aplikacije integrirati tudi nedokončane funkcionalnosti, ki pa jih uporabniku še ne prikažemo [12].

2.4 Pregled uporabljenih orodij in storitev

2.4.1 Git

Git je razpršeni sistem za nadzor nad verzijami datotek, ki ga je razvil Linus Torvalds leta 2005 [13]. Od takrat se je zelo razširil in je danes najbolj uporabljan sistem za nadzor nad verzijami datotek v postopku razvoja programske opreme [14]. Za učinkovito uporabo sistema je potrebno razumevanje osnovnih konceptov sistema, opisanih v nadaljevanju.

Repozitorij

Repozitorij je osnovna lokacija direktorija za delo s sistemom za nadzor nad verzijami datotek. Poleg vseh datotek projekta vsebuje še direktorij z imenom *.git*. V njem so shranjene vse pretekle spremebe datotek, kar nam omogoča tako pregledovanje zgodovine sprememb kot tudi povrnitev datotek v stanje, zabeleženo v določeni točki zgodovine projekta.

Uveljavljena sprememba

Uveljavljena sprememba (angl. *commit*) je osnovni gradnik za vnos sprememb v repozitorij. Vsebuje seznam datotek, ki so bile spremenjene glede na zadnje shranjeno stanje, in njihovo novo vsebino v trenutku oddaje v repozitorij.

Veja

Veja (angl. *branch*) v sistemu Git predstavlja unikatno pot razvoja projekta, ki je neodvisna od izvora veje. Vejitev razvijalcu omogoča ustvarjanje kopije celotnega projekta in uveljavljanje sprememb v izvorni kodi. Nove spremembe repozitorija so zabeležene samo v ciljni veji in ne vplivajo na potek razvoja v ostalih vejah.

Združevanje

Operacija združevanja (angl. *merge*) omogoča ponovno integracijo vsebine ločene veje v izvorno vejo. Vse uveljavitve sprememb v ločeni veji, ki so bile izvedene po vejitvi, se primerjajo s stanjem datotek izvorne veje. Če med istimi deli datotek ne pride do konfliktov, se nove spremembe samodejno prepišejo nazaj v izvorno vejo. V nasprotnem primeru mora razvijalec pred združevanjem sam odpraviti konflikte.

2.4.2 Github

Github je spletna storitev za gostovanje Git repozitorijev, ki nudi vse funkcionalnosti razpršenega sistema za nadzor nad verzijami datotek in upravljanja z izvorno kodo [15]. Za razliko od programa Git, ki ga v osnovi upravljamo preko ukazne vrstice, nam Github ponuja bogat spletni grafični vmesnik in funkcionalnosti za podporo skupinskemu delu.

Zabeležke

Za lažji pregled nad stanjem projekta nam Github omogoča beleženje napak in programskih hroščev. Razvijalci lahko zabeležkam dodajajo oznake in si jih dodelijo za razrešitev. Za načrtovanje izdaj imajo na voljo postavljanje mejnikov, ki lahko vsebujejo samo specifične popravke napak in cilje.

Dokumentacija

Github za hranjenje dokumentacije projekta ponuja programski paket Wiki, ki nam omogoča vnašanje podatkov o uporabi projekta, njegovem namenu in pogojih uporabe. Z uporabo opisnega jezika Markdown lahko vsi člani prispevajo dokumentacijo in omogočijo lažje vključevanje novih članov razvojne skupine v projekt.

Zahteve za pregled

Ena izmed glavnih funkcionalnosti, ki jih ponuja Github, so zahteve za pregled (angl. *pull request*). Omogočajo obveščanje ostalih članov skupine o na novo dodanih delih kode v določeni razvojni veji. Ko razvijalec ustvari zahtevo za pregled, lahko ostali razvijalci pregledajo nabor sprememb, razpravljajo o možnih popravkih in tudi sami objavijo nove dodatke h kodi. S pomočjo avtomatskega vstavljanja spletnih povezav se lahko v komentarjih sklicujejo na zaznamke, uporabnike in dodatke h kodi. V kolikor razvijalci ne odkrijejo napak v dodani kodi, jo lahko združijo v glavno vejo preko spletnega vmesnika.

Skupine

Github za potrebe večjih razvojnih skupin ponuja orodja za razdeljevanje razvijalcev v skupine. Vsaki skupini lahko dodelimo različne pravice dostopa do repozitorija - bralne, bralno-pisalne ali pa administratorske. Na voljo je tudi funkcionalnost sklicevanja na posamezno skupino v komentarjih in avtomatsko obveščanje vseh članov skupine ob sklicevanju nanjo.

Spletni klici

V primeru dogodkov kot sta na novo dodana koda ali ustvarjanje nove veje, spletni klici omogočajo komunikacijo z zunanjimi storitvami. Github v teh primerih lahko pošlje HTTP POST zahtevo s podatki o dogodku na izbrani spletni naslov. Slednje omogoča integracijo s širokim naborom storitev in akcije, kot so sprožitev postopka zvezne integracije, dodajanje novega vnosa

v zunanji sledilec napak in posodobitev varnostne kopije datotek.

2.4.3 Github Flow

Github Flow je delovni tok, ki je osnovan na ločevanju vej razvoja in nudi podporo skupinskemu delu na projektih, ki zahtevajo pogosto izvajanje postavitve [16]. Glavna predpostavka delovnega toka je, da je vsebina glavne veje v vsakem trenutku pripravljena na postavitev. Potek dela v Github delovnem toku sestavlja zaporedje petih korakov, opisanih v nadaljevanju.

1.) Ustvarjanje ločene razvojne veje
Ustvarjanje ločenih razvojnih vej nam omogoča, da nove funkcionalnosti razvijamo v okolju, ki ne vpliva na glavno vejo. Svoje eksperimentalno delo lahko nadaljujemo brez skrbi, da bi to delo vplivalo na aplikacijo v produkcijskem okolju. Ločene veje morajo izhajati iz glavne veje, prav tako pa jih moramo ustrezno poimenovati, da ostali člani razvojne skupine takoj vidijo bistvo razvojnega dela v dani veji.

2.) Razvoj in oddajanje programske kode
Vsako spremembo datotek v ločeni veji zabeležimo z uveljavitvjo sprememb, kar omogoča sledenje napredku. Sporočila o spremembah kode ustvarijo podrobno zgodovino našega dela in ostalim razvijalcem omogočijo razumevanje našega mišljenja - kateri del izvorne kode smo dodali v danem trenutku in zakaj. Sprotno oddajanje kode v repozitorj omogoča tudi kasnejšo razveljavitev sprememb in vrnitev v prejšnje stanje v primeru, da odkrijemo programski hrošč ali pa se odločimo za drugo smer dela.

3.) Zahteva po pregledu
Zahteva po pregledu omogoči ostalim članom razvojne skupine pregled predlaganih sprememb, ki jih želi razvijalec vključiti v glavno vejo. Zahtevo po pregledu lahko razvijalec ustvari tudi, če ima zgolj idejo o funkcionalnosti in se želi z ostalimi člani skupine najprej posvetovati o

možnosti njene implementacije, lahko pa tudi, ko je funkcionalnost že implementirana in želi izvorno kodo deliti z ostalimi člani skupine.

4.) **Pregled kode**
Po objavi zahteve za pregled lahko ostali člani razvojne skupine objavijo svoje komentarje in vprašanja glede predlaganih novosti izvorne kode. Izrazijo lahko nestrinjanje z uporabljenim slogom kode, morebitnimi manjkajočimi testi enot, ali pa izrazijo svoje strinjanje z dodatkom kode.

5.) **Združitev ločene razvojne veje z glavno**
Ko se z dodatkom nove kode strinja večji del razvijalcev iz skupine, se ločeno razvojno vejo lahko združi z glavno. Funkcionalnost, ki jo implementira nova izvorna koda, je po združitvi pripravljena na postavitev v produkcijsko okolje.

2.4.4 Chef

Chef je programska oprema za avtomatizacijo procesov upravljanja z infrastrukturo [17]. Omogoča nam, da proces postavitve infrastrukture obravnavamo kot programsko opremo (angl. *infrastructure as code*). Za razvoj omenjenih procesov lahko uporabimo podobne tehnike kot za razvoj aplikacij.

Chef poudarja centralizirano modeliranje infrastrukture, uporabo primitivnih virov in doseganje želenega stanja. Konceptualno je sestavljen iz dveh komponent: agenta in centralnega strežnika. Agent je nameščen na vozliščih in se periodično povezuje s centralnim strežnikom. Izvaja poizvedbe po novih definicijah stanj in skrbi za skladnost vozlišča z le-temi. Agenti večino konfiguracijskega dela opravijo na posameznih vozliščih, kar platformi omogoča porazdeljevanje dela in skalabilnost. Centralni strežnik pa služi kot zvezdišče za pridobitev informacij o konfiguraciji. Na njem so shranjene definicije želenih stanj, podatki zaupne narave in podrobni podatki o vseh vozliščih.

V nadaljevanju podajamo opis konceptov, s katerimi Chef opisuje želena stanja sistemov.

Recepti

Recept (angl. *recipe*) je osnovni konfiguracijski element v programu Chef. Napisan je v programskem jeziku Ruby in uporablja razširjen domensko specifični jezik (angl. *DSL - domain specific language*) za upravljanje z različnimi viri kot so datoteke, direktoriji, programski paketi, ukazi in uporabniki. Z uporabo receptov lahko natančno definiramo želeno končno stanje virov na sistemu. Ko Chef izvaja recept, preveri ujemanje trenutnega stanja vsakega vira z njegovim želenim stanjem. V primeru ujemanja ga preskoči, sicer pa poskuša izvesti vse potrebne spremembe, ki bi vir postavile v želeno končno stanje. Primer izvorne kode za spreminjanje dostopnih pravic datoteke ima sledečo obliko:

```
file "/etc/cron.weekly/apt-xapian-index" do
 mode '0644'
 not_if "stat -c%a /etc/cron.weekly/apt-xapian-index| grep 644"
end.
```

Zbirka receptov

Zbirka receptov (angl. *cookbook*) je osnovna enota za razpečevanje konfiguracije in pravil. Vključuje lahko več receptov, privzetih vrednosti atributov in predloge datotek. Veljavna zbirka mora vsebovati direktorije *recipes*, *templates*, *spec* in *test*. Direktorij *recipes* lahko vsebuje različne datoteke s končnico *.rb*, ki predstavljajo podmodule recepta, privzeto pa se ustvari in izvaja datoteka *default.rb*. Teste enot in integracijske teste hranita direktorija *test* in *spec*, v direktorij *templates* pa lahko shranjujemo lastne predloge datotek. Obvezna je tudi prisotnost datoteke *metadata.rb*, ki vsebuje podatke o zbirki receptov, avtorju in morebitne odvisnosti od drugih zbirk.

Vozlišča

Vozlišče (angl. *node*) predstavlja določen tip strežnika. Konfiguriramo ga z

uporabo datotek v formatu JSON. Atribut *run list* predstavlja seznam receptov ali vlog, ki se bodo izvedle za dosego želenega končnega stanja strežnika. Navedemo lahko tudi druge opcijske atribute, do katerih lahko pozneje dostopamo v receptih. Primer JSON datoteke vozlišča ima sledečo obliko:

```
{
        "capistrano_base": "/home/deploy/promet",
        "run_list":[
                "role[general]",
                "role[postgres]",
                "role[geo]",
                "role[ruby-rails]",
                "role[redis]",
                "role[web]",
                "recipe[setup_database_yml]",
                "role[firewall]",
                "recipe[push_service_certificate]"
        ]
}.
```

Vloge

Vloga (angl. *role*) združuje več zbirk receptov, ki so običajno tematsko povezane in se lahko kot celota uporabljajo v različnih vozliščih. Podobno kot vozlišča jih konfiguriramo z uporabo JSON datotek in vsebujejo obvezen atribut *run list* ter ostale opcijske atribute. Ob izvedbi recepta se seznam zbirke receptov vozlišča in atributi združijo s seznamom zbirk receptov vloge in njenimi atributi, pri čemer atributi vozlišča prepišejo istoimenske atribute vloge.

Predloge datotek

Predloge datotek (angl. *template*) imajo končnico *.erb* in vsebujejo vgrajeno Ruby programsko kodo. To je uporabno predvsem za ustvarjanje datotek,

ki vsebujejo atribute, odvisne od ciljnega strežnika. Na primer, če imamo v neki datoteki atribut za največje dovoljeno število povezav, lahko v predlogo zapišemo ime spremenljivke, ki jo hranimo v konfiguracijski datoteki strežnika. Ob izvedbi recepta se izvede programska koda znotraj oznak <%= %>, rezultat evalvacije se vstavi na ustrezno mesto in ustvari se datoteka. Spodnji primer prikazuje predlogo datoteke, ki vsebuje dinamično določen atribut lokacije aplikacije:

```
upstream app_server { server 127.0.0.1:3000 fail_timeout=0; }
server {
        listen   80;
        root <%= node['capistrano_base']%>/current/public;
        server_name promet.me;
        index index.htm index.html;
        location / {
         rewrite ^ https://$http_host$request_uri? permanent;
        }
}.
```

Podatkovne vreče

Podatkovne vreče (angl. *databags*) predstavljajo preprost mehanizem za shranjevanje parov ključ-vrednost za uporabo v Chef receptih. Posamezni elementi so shranjeni v direktoriju *data_bags* v obliki JSON datotek. Vsaka podatkovna vreča mora vsebovati vsaj atribut *id* z enoličnim identifikatorjem podatkovne vreče, ostali atributi pa so opcijski. Spodnji primer prikazuje tipično obliko podatkovne vreče:

```
{
  "id"       : "deploy",
  "comment"  : "Deploy user",
  "home"     : "/home/deploy",
  "shell": "\/bin\/bash",
```

```
  "groups"   : ["deployers"],
  "ssh_keys" : [
        "ssh-rsa AAAAB3NzaC1yc2 Jaka@Jaka-Sustersic-MBP.local",
        "ssh-rsa AAAAB3NzaC1yc2 Codeship/uncoverd/promet"
  ]
}.
```

Podatkovno vrečo lahko šifriramo s skupno skrivnostjo, na primer z mehanizmom AES-256-CBC . To nam omogoča varno shranjevanje zaupnih podatkov (kot so na primer gesla podatkovnih baz) preko sistemov za nadzor nad verzijami datotek, saj v primeru vdora v repozitorij čistopisi niso vidni. Vsak atribut podatkovne vreče je lahko šifriran s svojim ključem, saj so v JSON formatu datotek predvideni atributi za inicializacijski vektor, verzijo protokola in izbrano kriptografsko šifro. Struktura posameznih polj za šifrirane podatkovne vreče je razvidna iz spodnjega primera:

```
{
  "id": "postgres",
  "db_name": {
    "encrypted_data": "MT/K013pBHqrOrXVk+KPs3TgnqXgzzyp+GY=\n",
    "iv": "Lc2fXdy6ZHSmluhpiatUaw==\n",
    "version": 1,
    "cipher": "aes-256-cbc"
  },
  "db_username": {
    "encrypted_data": "SpHAUDKhTrXiAVPpCr38J5bGC8zLTs\n5vvd\n",
    "iv": "CCbmN50RazZfmJg1eyAyiQ==\n",
    "version": 1,
    "cipher": "aes-256-cbc"
  },
  "db_password": {
    "encrypted_data": "Rde1f9hMtjkFRSrGmumcWIolIaEB2U\nyOJI\n",
    "iv": "kRg4sDWjAdjvWjG3soBMDA==\n",
```

```
    "version": 1,
    "cipher": "aes-256-cbc"
  }
}.
```

2.4.5 Capistrano

Capistrano je orodje za avtomatizacijo izvajanja ukazov na oddaljenih strežnikih, napisano v programskem jeziku Ruby [18]. Podpira izvajanje skript in preprostih ukazov z uporabo domensko specifične nadgradnje jezika knjižnice Rake. Capistrano vključuje širok nabor ukazov, ki poenostavljajo interakcijo z oddaljenimi strežniki in je v prvi vrsti namenjen uporabi pri postavitvi spletnih aplikacij na poljubnem številu strežnikov. Strežnike lahko razdelimo v posebne skupine, jim določimo vloge in na njih izvajamo ukaze istočasno ali pa zaporedno. Capistrano razvijalcem olajša delo tudi z avtomatizacijo skrbniških nalog, kot so upravljanje dnevniških datotek in namestitve varnostnih popravkov.

Orodje je zelo prilagodljivo, saj omogoča uporabo vtičnikov za upravljanje spletnih ogrodij, kot so Ruby on Rails, Laravel in Drupal. Razvijalec ga lahko razširja tudi sam z uporabo Capistrano opravil.

2.4.6 Amazon S3

Amazon S3 je spletna storitev za hranjenje datotek, ki razvijalcem nudi varno, skalabilno in robustno hranjenje podatkov [19].

Podatki so shranjeni kot objekti v vedrih (angl. *buckets*). Vedra so podobna datotečnim direktorijem in omogočajo natačno določanje pravic dostopa do objektov. Dodajanje, brisanje in urejanje objektov lahko omejimo na določene uporabnike, na voljo pa imamo tudi dnevniške datoteke dostopov. Geografsko regijo lokacije vedra lahko izbiramo glede na naše zahteve po odzivnem času, ceni in zakonskih zahtevah. Do podatkov lahko dostopamo z uporabo protokolov kot sta HTTP in BitTorrent. Za uporabo storitve

Amazon S3 v sklopu aplikacij so na voljo ovojnice REST storitev v številnih programskih jezikih, na primer Java, PHP, Ruby in .NET.

2.4.7 DigitalOcean

DigitalOcean je ameriški ponudnik navideznih zasebnih strežnikov (angl. *VPS - virtual private servers*), ki med drugim ponuja tudi najem strežnikov in nam s tem omogoča uporabo infrastrukture kot storitve (angl. *infrastrusture as a service*) [20]. DigitalOcean omogoča preprosto ustvarjanje novih strežnikov v večini geografskih regij, celoten postopek pa traja manj kot eno minuto. Podpira širok nabor operacijskih sistemov kot na primer Ubuntu, CentOS, CoreOS, FreeBSD, Fedora in Debian. Razvijalci imajo na voljo več različnih tipov strežnikov, ki se razlikujejo po količini pomnilnika, velikosti podatkovnega prostora na SSD diskih in številu procesorjev. Uporaba strežnikov se obračuna po urnem intervalu, cene pa se gibljejo od 5 do 500 dolarjev na mesec.

Poglavje 3

Izhodiščna aplikacija

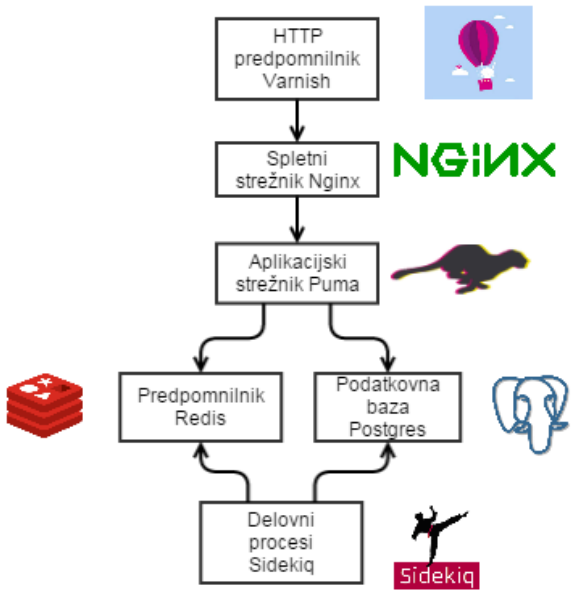
Praktično uporabo tehnik zvezne integracije, dostave in postavitve ter razloge za uporabo le-teh želimo prikazati na primeru obstoječe aplikacije.

Glavna funkcionalnost naše aplikacije je obveščanje uporabnikov o prometnih dogodkih na izbranih območjih v realnem času. Za razliko od ostalih podobnih aplikacij na trgu naša aplikacija uporabniku omogoča natačno izbiro območja na zemljevidu, za katerega želi prejemati obvestila o prometnih dogodkih. Uporabnik na ta način prejema obvestila o prometnih nesrečah, zastojih ali izrednih dogodkih na cestah, ki se nahajajo v izbranem območju. S prilagajanjem interesnega območja se lahko uporabnik izogne prejemanju zanj nepomembnih prometnih obvestil. Uporabnik lahko svojo izbrana območja ureja tako na spletni strani *https://promet.me*, kot tudi na mobilni aplikaciji *Ceste* na iOS platformi.

3.1 Zgradba aplikacije

Zaledje spletne in mobilne aplikacije je sestavljeno iz šestih večjih komponent, ki so prikazane na sliki 3.1. Uporabniške zahteve na vratih 80 in 443 sprejema programski paket Varnish, ki je zadolžen za predpomnenje pogosto zahtevanih vsebin. Slednji zahteve po vsebinah, ki niso v predpomnilniku, posreduje spletnemu strežniku Nginx. Spletni strežnik hrani statične vsebine

(npr. slike in stilne predloge) in posreduje dinamične zahteve aplikacijskemu strežniku Puma, ki izvršuje poslovno logiko. Aplikacija podatke hrani v podatkovni bazi Postgres, za predpomnenje računsko zahtevnih operacij pa uporablja programski paket Redis. Za posodabljanje prometnih podatkov je uporabljen program Sidekiq, ki je namenjen izvajanju delovnih procesov.



Slika 3.1: Diagram zaledja aplikacije.

3.2 Razlogi za implementacijo zvezne postavitve

V času razvoja aplikacije smo z večanjem števila komponent projekta in dodajanjem novih funckionalnosti naleteli na več težav, ki so oteževale in upočasnjevale razvoj. Težave, opisane v nadaljevanju, postanejo še bolj izrazite v ekipah večih razvijalcev, ki so v industriji programske opreme stalnica, in predstavljajo motivacijo za prehod na zvezno postavitev aplikacij.

3.2.1 Prenos razvojnega okolja na drug sistem

Razvoj aplikacije se je začel na operacijskem sistemu OSX. Ob prehodu na operacijski sistem Windows je pogosto prihajalo do težav zaradi razlik med konfiguracijami obeh sistemov. Za veliko programskih paketov, ki smo jih uporabili pri razvoju aplikacije, ne obstaja Windows verzija programskega paketa, ali pa so na voljo le starejše verzije. Razvijalci nekaterih obstoječih verzij programov za okolje Windows, kot sta Varnish in Redis, njuno uporabo v produkcijskem okolju celo odsvetujejo [21] [22], zaradi česar te verzije niso primerne niti za uporabo v razvojnem okolju, saj ne nudijo zagotovil o skladnosti delovanja z veljavno specifikacijo.

Največje težave pri prenosu na okolje Windows smo imeli pri namestitvi zunanjih Ruby knjižnic, ki za delovanje zahtevajo namestitev domorodnih knjižnic, napisanih v programskih jezikih C in C++. Te knjižnice so odvisne od sistema, na katerem se izvajajo, in jih je treba v Windows okolju večinoma ročno zgraditi. Do težav pride, ker je večina zunanjih knjižnic razvitih v okolju Linux, kjer so potrebne knjižnice že pred-nameščene, v Windows okolju pa ne obstajajo. Veliko razvijalcev tudi eksplicitno ne podpira uporabe njihove programske opreme na operacijskem sistemu Windows.

Do razlik med različnimi verzijami programa in platformami pride na primer pri uporabi aplikacijskega strežnika Puma. Operacijski sistem Windows 7 za razliko od sistemov OSX in Ubuntu ni skladen z družino standardov POSIX, ki določajo enoten programski vmesnik za kompatibilnost različnih UNIX sistemov. Na Windows sistemih odsotnost funkcije *fork*, ki je namenjena ustvarjanju novega procesa, onemogoča izvajanje strežnika Puma v načinu prikritega procesa. Tako izvajanje aplikacijskega strežnika poteka v ozadju in ni pod kontrolo interaktivnega uporabnika.

3.2.2 Dolgotrajna in zapletena postavitev aplikacije

Postavitev nove verzije aplikacije zahteva od razvijalca interakcijo z vsemi večjimi komponentami zaledja (naša aplikacija jih ima šest). Zato je poleg

vnosa vsaj petih ukazov potrebno tudi dolgotrajno ročno preverjanje dosegljivosti različnih sklopov aplikacije po postavitvi. Postopek je zelo občutljiv na napačen vrstni red vnosa podatkov in lahko pripelje do nezaznanih napak komponent. V primeru napake je potrebna ponovna postavitev, kar še podaljša čas postavitve na več kot 5 minut.

3.2.3 Dolgotrajna in zapletena postavitev infrastrukture

Začetna postavitev strežnika in nameščanje vseh programskih paketov je bila dolgotrajna in je zahtevala več kot 2 uri časa.

Celoten postopek je bil dokumentiran z željo, da bi nadaljnje postavitve strežnika potekale hitreje. Kljub temu pa je ročno vnašanje vseh za to potrebnih ukazov ob večini naslednjih postavitev bilo vir velikega števila napak zaradi napačnega vrstnega reda vnešenih ukazov. Ker se je konfiguracija sistema s časom spreminjala, so poleg tega določeni deli dokumentacije zastareli, s tem pa je bila otežena postavitev enakega sistema. Vsaka ponovna postavitev strežnika je zato kljub natančni dokumentaciji in vloženemu trudu trajala več kot pol ure.

Težave s postavitvijo in razreševanje konfliktov zaradi neskladnosti produkcijskega okolja z razvojnim so razvoj aplikacije podaljšali za vsaj en teden.

3.2.4 Prekinitev storitve med postavitvijo

Med postopkom postavitve se ustavi tako skupina delovnih procesov programa Sidekiq kot tudi aplikacijski strežnik Puma. Uporabnikom spletne in mobilne aplikacije storitev zato za nekaj sekund ni dostopna. Da bi zmanjšali verjetnost, da končni uporabniki opazijo izpad storitve, se postavitve aplikacij v praksi pogosto izvajajo ponoči - s tem pa razvijalci ne rešijo problema, ampak ga zgolj zaobidejo. Že v primeru naše aplikacije kljub temu med vsako postavitvijo pride do večminutne izgube podatkov, ki so bili zajeti s pomočjo zunanje storitve za zbiranje prometnih podatkov.

3.2.5 Zaznava napake postavitve in ukrepanje

Med postavitvami aplikacije je pogosto prihajalo do napak pri upravljanju strežniških komponent, saj je bilo vsakič potrebno ročno vnašanje ukazov. Zaradi napačnega vrstnega reda vnosa ukazov ali napak v kodi aplikacije je prihajalo do naslednjih tipov napak:

- Nginx strežnik se ni zagnal zaradi napake v konfiguracijski datoteki,
- aplikacijski strežnik Puma se ni zagnal zaradi nepravilne zaustavitve,
- aplikacijski strežnik Puma se ni zagnal zaradi napake v konfiguracijski datoteki,
- aplikacijski strežnik Puma se ni zagnal zaradi neveljavne spremembe v izvorni kodi aplikacije,
- delovni procesi programa Sidekiq se niso začeli zaradi nepravilne zaustavitve,
- HTTP predpomnilnik ni bil ponastavljen.

Po končanem procesu postavitve je bilo zato potrebno ročno preverjanje vseh komponent aplikacije. Razvijalec je moral najprej ročno obiskati spletno stran aplikacije in preveriti odzivnost v brskalniku. Potrebna je bila tudi osvežitev podatkov v mobilni aplikaciji in testiranje glavnih uporabniških poti. Če je razvijalec odkril napako, je moral preveriti datotečne izpise ključnih delov sistema, uporabiti orodja za pregled trenutno izvajajočih se procesov in ponovno opraviti postavitev.

Posebno pozornost smo namenili tudi pravilni ponastavitvi HTTP predpomnilnika programskega paketa Varnish. Razvijalec je moral počakati na vsaj eno osvežitev podatkov in v brskalniku preveriti, da spletni strežnik ne streže zastarelih podatkov.

3.2.6 Podatkovne migracije

Z uporabo podatkovnih migracij lahko spreminjamo shemo podatkovne baze, kar lahko predstavlja nevarnost za integriteto podatkov. Operaciji spreminjanja dolžine polj v tabelah in odstranjevanja tabel lahko privedeta do izgube podatkov. Naša aplikacija pred uvedbo zvezne postavitve ni imela implementiranega varnostnega kopiranja podatkovne baze, kar bi v primeru napak lahko vodilo v izgubo vseh podatkov o uporabnikih aplikacije. Poleg t.i. destruktivnih migracij pa obstajajo še druge nevarne migracije, ki lahko npr. z uporabo napačnih parametrov zaradi zaklepanja tabel onemogočijo dostop do podatkov za daljše obdobje in povzročijo izpad storitve.

3.3 Cilji prehoda na zvezno postavitev

V razdelku 3.2 smo opisali najpogostejše težave, s katerimi se srečujejo ekipe razvijalcev pri implementaciji netrivialnih spletnih aplikacij. Eden izmed načinov kako jih lahko v veliki meri odpravimo, je implementacija zvezne postavitve. V nadaljevanju si oglejmo v kolikšni meri in na kakšne načine lahko zvezna postavitev prispeva k reševanju omenjenih težav ter kaj so pravzaprav cilji prehoda na zvezno postavitev.

3.3.1 Razbremenitev razvijalca

Postopek postavitve, opisan v razdelku 3.2.2, predstavlja za razvijalca zahtevno nalogo, saj vsak postopek postavitve traja vsaj pet minut in od razvijalca zahteva pozornost tako med vnašanjem zaporedja ukazov kot tudi med preverjanjem delovanja funkcionalnosti po postavitvi. Kljub vnaprej znanemu naboru ukazov, potrebnih za postavitev, mora razvijalec poleg razvoja funkcionalnosti svoj čas posvetiti še previdnemu vnašanju ukazov. To za razvijalca predstavlja režijsko delo, zato se dolgotrajnemu postopku postavitve želimo izogniti. Naloge razvijalca želimo omejiti samo na oddajanje kode v repozitorij in sprejemanje sprememb članov ekipe z uporabo zahtev

za pregled. S tem želimo spodbuditi pogoste in predvsem manjše izdaje.

Eden od ciljev razbremenitve razvijalca je tudi boljši pregled nad delom razvojne skupine in izboljšanje komunikacije znotraj le-te. Brez praktične uporabe zvezne integracije, dostave in postavitve razvijalec med delom na novi funkcionalnosti nima možnosti sprejemanja komentarjev o svojem trenutnem delu s strani drugih razvijalcev.

3.3.2 Postavitev brez prekinitve delovanja

Razviti želimo postopek, ki bo omogočal nadgrajevanje izvorne kode na aplikacijskem strežniku brez za uporabnika opazne prekinitve delovanja. Izogniti se želimo tako večminutnim prekinitvam kot tudi načrtovanju postavitev v času najmanjše uporabe storitve.

3.3.3 Robustna rešitev za prehode med verzijami

Naš cilj je tudi preverjanje pravilnosti aplikacije na več nivojih, ki ga izvajamo preko različnih skupin testov. Obnašanje funkcij razredov preverjamo s testi enot, pravilnost skupkov razredov z integracijskimi testi, stanje po postavitvi pa z dimnimi testi (angl. *smoke tests*). Kljub temu lahko v produkcijskem okolju pride do nepredvidljivih napak. V takih primerih želimo imeti na voljo preprost mehanizem za prehod na prejšnjo (delujočo) verzijo izvorne kode. Želimo, da nas na napako pri postavitvi opozori orodje za postavitev, in nam predstavi možne ukaze, primerne za razrešitev dane napake.

3.3.4 Opozarjanje na nevarne podatkovne migracije

Spreminjanje podatkovne sheme baze z uporabniškimi podatki zahteva veliko mero pazljivosti. Z neustrezno rabo podatkovnih migracij lahko pride do nepovratnih sprememb ali celo do izgube podatkov. Zato si želimo avtomatiziranih programskih rešitev, ki bi razvijalce lahko opozarjale na najpogostejše tipe nevarnih migracij in s tem preprečile morebitno poslovno škodo.

3.3.5 Testiranje v vmesnem okolju

Nove funkcionalnosti želimo testirati v vmesnem okolju, ki je kopija produkcijskega (vsebuje vse podatke iz produkcijskega okolja). Konfiguracija operacijskega sistema v vmesnem okolju naj bo enaka tisti v produkcijskemu okolju, da se izognemo napakam zaradi neustreznih ali manjkajočih verzij programske opreme. Izvorna koda in postavitev aplikacije naj tečeta enako v obeh okoljih in naj ne vsebujeta funkcionalnosti, ki jih bodo uporabniki prvič preizkusili šele v produkcijskem okolju.

3.3.6 Preverjanje stanja po postavitvi aplikacije

Po postavitvi aplikacije v produkcijsko okolje se želimo z uporabo preprostih testov prepričati, da je aplikacija dosegljiva uporabnikom in da so ključne uporabniške poti delujoče. Ročno preverjanje, ki ga sicer opravijo razvijalci, pogosto ne odkrije napak in poleg tega zahteva veliko časa in truda, zato želimo avtomatizirati preverjanje stanja po postavitvi.

3.3.7 Preprosta postavitev razvojnega okolja

Prav tako želimo čim bolj poenostaviti postopek postavitve razvojnega okolja. V kolikor bodo vsi razvijalci uporabljali enako konfigurirane sisteme, se lahko izognemo pogostim napakam zaradi različnih verzij operacijskih sistemov in programske opreme. Razvojno okolje naj bo na voljo na vseh najpogosteje uporabljanih družinah operacijskih sistemov - Windows, OSX in Linux.

3.3.8 Preprosta postavitev infrastrukture

Tudi postavitev infrastrukture želimo za razvijalca čim bolj poenostaviti. Izogniti se želimo dolgotrajnemu vnašanju ukazov in spreminjanju konfiguracijskih datotek vseh komponent (npr. spletnih in aplikacijskih strežnikov). Želimo si ponovljiv in prilagodljiv postopek, ki ne bo zahteval ročnega vna-

šanja velikega števila ukazov, za dodajanje novih strežnikov v primeru povečanega števila HTTP zahtev pa ne želimo porabiti več kot pol ure.

Poglavje 4

Implementacija zvezne postavitve

Razvoj ustrezne implementacije zvezne postavitve zahteva podrobno znanje o delovanju in upravljanju vseh komponent aplikacije. Za uspešen prehod na avtomatiziran postopek postavitve aplikacije in uvedbo ostalih sprememb v razvoju aplikacije je najprej potrebna evalvacija trenutnih postopkov. Sledilo bo dopolnjevanje vseh tipov testov, saj želimo povečati raven zaupanja razvijalcev v avtomatiziran postopek. Implementaciji postavitve okolja in aplikacije, opisani v naslednjih poglavjih, predstavljata jedro diplomskega dela. Posebna pozornost je v naslednjem poglavju namenjena implementaciji ustreznih postopkov za upravljanje s podatkovno bazo, zadnje poglavje pa je namenjeno opisu uporabe storitve za zvezno integracijo.

4.1 Evalvacija trenutnega stanja

Za zagotovitev implementacije ustrezne rešitve je bila najprej potrebna evalvacija postopkov postavitve aplikacije in infrastrukture. Večina podatkov o postavitvi infrastrukture je bila zabeležena v dokumentaciji, ki pa v določenih delih ni bila v skladu z realnim stanjem sistema.

4.1.1 Postavitev infrastrukture in razvojnega okolja

Postavitev pravilno konfiguriranega strežnika je pogosto zapleten proces, ki zahteva vnašanje velikega števila ukazov. Na primer, za nastavitev strežnika naše aplikacije je potrebna namestitev najmanj sedmih večjih kosov programske opreme in sprememba najmanj petih konfiguracijskih datotek. Nastavitev strežnika v grobem zajema naslednje sklope:

- nastavitev uporabniškega računa in protokola SSH,
- nastavitev časovnega pasu,
- sinhronizacija sistemske ure,
- nadgradja seznama programskih paketov,
- namestitev upravljalnika verzij programskega jezika Ruby,
- namestitev izbrane verzije izvajalnega okolja programskega jezika Ruby,
- nastavitev sistemske verzije programskega jezika Ruby,
- namestitev programskega paketa Postgres,
- ustvarjanje direktorija za začasne podatke paketa Postgres,
- sprememba lastništva direktorija z začasnimi podatki,
- inicializacija programa Postgres,
- ustvarjanje uporabnika podatkovne baze,
- dodajanje pravic uporabniku podatkovne baze,
- ustvarjanje podatkovne baze,
- sprememba gesla podatkovne baze,
- namestitev programskega paketa Redis,

- namestitev paketa node.JS,
- namestitev GEOS paketa,
- namestitev ostalih zunanjih knjižnic, napisanih v programskem jeziku Ruby,
- nastavitev dostopnih pravic za podatkovno bazo,
- namestitev spletnega strežnika Nginx,
- nastavitev spletnega strežnika,
- namestitev HTTP predpomnilnika Varnish,
- nastavitev programa Varnish,
- nastavitev požarnega zidu.

4.1.2 Ročna postavitev aplikacije

V izhodiščni aplikaciji je moral razvijalec vsako postavitev nove verzije aplikacije opraviti ročno.

V ta namen se je moral na produkcijski strežnik povezati preko protokola SSH, ki omogoča oddaljeno izvajanje ukazov. Za prenos nove verzije izvorne kode iz glavne veje repozitorija je moral razvijalec v direktoriju aplikacije izvesti ukaz `git pull`, ki iz oddaljenega repozitorija v izbran direktorij prenese najnovejšo verzijo aplikacije.

Nato je bilo potrebno ustaviti delovne procese programa Sidekiq z ukazom `rake sidekiq:stop`. Ker so razredi omenjenih delovnih procesov serializirani in se shranjujejo v podatkovni zbirki Redis, ki je tipa ključ-vrednost, je bilo obenem potrebno izbrisati vse ključe. To je bilo potrebno storiti z ukazom `rediscli flushdb`. S tem ukazom smo poskrbeli tudi za razveljavitev vseh shranjenih rezultatov računsko zahtevnih operacij ogrodja Rails, ki podatkovno zbirko uporablja kot predpomnilnik. Za tem je bilo potrebno z ukazom `rake puma:stop` ustaviti tudi aplikacijski strežnik Puma. Po tem,

ko smo izvedli zaporedje do sedaj omenjenih ukazov, je postala večina funkcionalnosti aplikacije za uporabnika nedosegljivih. Uporabnikom so po tem bile na voljo le še vsebine spletnih naslovov, ki so bile predpomnjene z uporabo HTTP predpomnilnika Varnish. Če so za delovanje novejših verzij izvorne kode bile potrebne podatkovne migracije, jih je razvijalec moral pognati z ukazom `rake db:migrate`, nato pa je moral ponovno zagnati aplikacijski strežnik z ukazom `rake puma:start`. Zadnji korak ročne postavitve bil ponovni zagon delovnih procesov z ukazom `rake sidekiq:start`.

Ročno preverjanje pravilnosti

Po uspešno opravljenem zaporedju navedenih korakov je bila nova izvorna koda na voljo v vseh komponentah sistema. Razvijalec je nato moral preveriti še, ali je v postopku prišlo do napak. To je lahko storil preko spletnega brskalnika, in sicer najprej je preveril dosegljivost podstrani */api/v1/events*, ki je namenjena potrjevanju pravilnega delovanja aplikacijskega strežnika. Nato je iz metrik na podstrani */sidekiq* lahko razbral, ali se delovni procesi programa Sidekiq izvajajo normalno, oziroma ali je prišlo do napak pri časovni dodelitvi ali izvajanju kode.

Razvijalec je nazadnje moral postavitev še zabeležiti preko spletnega vmesnika storitve New Relic.

4.1.3 Testi v izhodiščni aplikaciji

Testi v izhodiščni aplikaciji so bili osredotočeni na preverjanje pravilnosti t.i. kontrolerjev, v katerih se nahajajo metode z večino poslovne logike. Za testiranje celotnih poti HTTP zahtev je izhodiščna aplikacija uporabljala orodje Rspec, ki deluje kot ovojnica integracijskih testov ogrodja Ruby on Rails. Testirali smo predvsem pravilnost interakcije mobilne aplikacije s strežnikom na nivoju HTTP zahtev in odgovorov, in sicer tako, da smo najprej ustvarili in poslali HTTP zahtevo strežniku, nato pa preverjali strežnikov odgovor. Tak način testiranja pa je oteževal preverjanje delovanja posameznih funkci-

onalnosti aplikacije na nivoju metod.

Po drugi strani pa je izhodiščna aplikacija vključevala veliko testov delovnih procesov - preverjalo se je tako zaželjeno kot tudi nezaželjeno delovanje.

Testi enot so zajemali samo funkcionalnosti dveh razredov - *CacheManager*, ki je zadolžen za ponastavitev HTTP predpomnilnika programa Varnish in metod razreda *User*, ki so se navezovale na stanje urnika.

Veliko testov je bilo medsebojno odvisnih, za pravilno delovanje so namreč potrebovali podatke, ki so jih ustvarili drugi testi. Posledično so bili scenariji za pripravo testov zapleteni, saj je bilo potrebno ustvarjati in uporabljati celotne objekte namesto njihovih začasnih dvojnikov.

Celoten nabor vseh opisanih testov je zajemal 1386 vrstic kode, zajetih v enajstih datotekah, skupen čas testiranja pa je bil 17 sekund.

4.2 Dopolnitev testov

Glavna cilja pri dopolnitvi nabora testov aplikacije sta bila hitrejše izvajanje testov in večje število testiranih funkcionalnosti. Pri uporabi zvezne postavitve si namreč čim prej želimo povratno informacijo o pravilnosti na novo dodane kode. Hitro izvajanje testov omogoča razvijalcem bolj pogosto testiranje in krajše prekinitve ob razvoju. Ker se postopek postavitve izvaja avtomatizirano, želimo v testiranje zajeti tudi čim večje število funkcionalnosti aplikacije.

4.2.1 Testi enot

Funkcionalnosti razreda *User*, ki so v izhodiščni verziji aplikacije urejale delo z urnikom, smo v končni verziji prestavili v nov modul *Schedule*.

Prenos logike v ločen modul ima za testiranje dve veliki prednosti. Ker implementacija metod ni več vezana na razred *User*, lahko testiranje izvedemo z nadomestnim objektom, ki je njegov začasni dvojnik. Za razliko od razreda *User*, ki hrani podatke v podatkovni bazi, je lahko dvojnik preprost Ruby razred. Z neposrednim nastavljanjem rezultatov klica metod dvojnika

z metodo *stub* se lahko izognemo uporabi podatkovne baze. Posledično na ta način prihranimo na času izvajanja testov, ker ni več potrebno vzpostavljati povezave s podatkovno bazo, prav tako pa ni potrebno posodabljati vsebine objekta v podatkovni bazi in nenazadnje še zapirati povezave s podatkovno bazo. Za vzpostavitev začasnega dvojnika smo uporabili naslednjo izvorno kodo:

```
let(:user) { (Class.new { include Schedule }).new }

before do
  user.stub(start_schedule: "7:00")
  user.stub(end_schedule: "22:00")
end.
```

Druga velika prednost uporabe ločenega modula je v tem, da nam za izvajanje testov ni več potrebno zaganjati celotnega ogrodja Ruby on Rails. Ker ne potrebujemo funkcionalnosti povezovanja s podatkovno bazo, lahko z ukazom `require` vključimo le datoteko z modulom in del ogrodja Rails, ki nam olajša računanje s podatkovnimi tipi za čas:

```
require_relative '../../app/models/concerns/schedule'
require 'active_support/all'.
```

Meritve časa izvajanja testov so pokazale, da isti testi pri izvajanju z začasnim objektom dvojnikom (0.01 sekunde na test) porabijo za velikostni red manj časa kot testi z razredom *User* (0.1 sekunde na test).

4.2.2 Integracijski testi

Večino poslovne logike, ki je bila v izhodiščni aplikaciji v kontrolerjih, smo kasneje prestavili v posamezne storitvene objekte. En storitveni objekt ustreza enemu tipu interakcije uporabnika s sistemom in običajno ureja več kot eno poslovno entiteto aplikacije. Tak način strukturiranja izvorne kode nam omogoča bolj enostavno testiranje delovanja posameznih razredov, prav tako

pa tudi bolj enostavno ustvarjanje začasnih dvojnikov objekta za namene testiranja ostalih razredov in uporabo iste funkcionalnosti na večih mestih. Na primer, storitveni objekt za registracijo novega uporabnika lahko uporabimo tako v kontrolerju spletne aplikacije kot tudi v kontrolerju mobilne aplikacije. Na ta način storitveni objekti obsegajo naslednje tipe uporabniške interakcije z aplikacijo:

- dodajanje za uporabnika zanimivega geografskega območja,
- urejanje za uporabnika zanimivega geografskega območja,
- nakup naročnine,
- registracija novega uporabnika,
- brisanje za uporabnika zanimivega geografskega območja.

V nadaljevanju podajamo primer storitvenega objekta za registracijo uporabnikov:

```
class RegisterUser
  attr_reader :status, :username, :password,
  :user_id, :access_token

  def initialize(params)
    @username = params[:username]
    @password = params[:password]
    @user_id = params[:user_id]
    @access_token = params[:access_token]

    begin
      check_parameters
      check_for_existing_username
      add_username_and_password
    rescue UserNameExists, UserParametersInvalid,
```

```
    AccessTokenInvalid, ActiveRecord::RecordNotFound
      return
    end
  end

end.
```

Storitvenim objektom inicializacijske podatke posredujemo preko atributa *params* s klicem metode *new*. Ko Ruby zazna klic te metode, se namreč poleg ustvarjanja novega objekta sproži tudi njegova inicializacija preko metode *initialize*.

Zaradi uporabe storitvenih objektov se metode kontrolerjev zelo poenostavijo:

```
def register
        result = RegisterUser.new(params).status
        render json: result
end.
```

4.2.3 Dimni testi

Dimni testi so namenjeni avtomatiziranemu preverjanju pravilnega delovanja aplikacije po postavitvi in pokrivajo samo delovanje najpomembnejših funkcionalnosti sistema [23]. Osredotočeni so na testiranje nabora preprostih operacij, ki simulirajo interakcijo uporabnika s spletno aplikacijo. Na primer, dimni testi preverjajo, ali se uporabnik z veljavnimi uporabniškimi podatki lahko prijavi v aplikacijo, ali obstaja neko besedilo na spletni strani, ali je določena podstran dosegljiva in ali so želeni HTML elementi prisotni na spletni strani. Dimni testi nam omogočajo večje zaupanje v pravilnost postopka postavitve aplikacije in na ta način spodbujajo pogostejše izdaje.

Pri implementaciji dimnih testov smo si pomagali z uporabo programskega paketa Capybara, ki omogoča uporabo visokonivojskega jezika specifične domene za pisanje testov, ki se izvajajo z vmesnikom brezobličnega

brskalnika (angl. *headless browser*) Poltergeist. Uporaba omenjenega tipa brskalnika omogoča hitrejše izvajanje dimnih testov, saj za izvajanje ne potrebuje grafičnega vmesnika. Za pisanje scenarijev v dimnih testih lahko uporabimo preproste ključne besede, ki simulirajo uporabniške akcije, na primer *visit*, *fill_in*, *find* in *click*.

Spodnja izvorna koda, s katero preverjamo, ali se uporabnik z veljavnim uporabniškim imenom in geslom lahko prijavi v aplikacijo, prikazuje tipično vsebino dimnega testa:

```
let (:seeded_username){'testuser'}
let (:seeded_password){'test1'}

it "logins with a correct username", :js => true  do
  visit '/eventmap'
  within('#loginForm') do
    fill_in 'Uporabniško ime', :with => seeded_username
    fill_in 'Geslo', :with => seeded_password
  end
  click_button 'Prijava'
  username_nav = find('#usernameNavbar')
  user_data = find('#user')

  expect(username_nav).to have_content seeded_username
  expect(user_data['data-username']).to eq seeded_username
end.
```

Za potrebe testiranja smo v podatkovno bazo produkcijskega okolja predhodno vnesli testnega uporabnika *testuser*. Testni scenarij, ki ga opisuje zgornja koda, se začne na podstrani */eventmap*. Ta podstran vsebuje prijavni obrazec s HTML oznako *#loginForm*. Test napolni obe potrebni polji z veljavnimi podatki in pritisne gumb s HTML vsebino "Prijava". Da je zgornji test uspešno opravljen, mora uporabniška vrstica vsebovati pravilno

ime uporabnika, HTML element z oznako *#user* pa mora imeti nastavljene pravilne vrednosti atributov.

Z dimnimi testi smo poleg prijave uporabnika preverjali tudi splošno dosegljivost domače strani, prikaz napake ob neveljavni prijavi, veljavno registracijo in prikaz napake pri neveljavni registraciji. Vmesnik Poltergeist pa nam še dodatno omogoča avtomatsko zaznavanje uporabniških JavaScript napak.

Celoten nabor dimnih testov se izvede po postavitvi v vmesno in produkcijsko okolje z naslednjim ukazom :

```
TARGET_ADDRESS=https://188.166.95.64 bundle exec
 rspec --pattern "**/{smoke}/*_spec.rb".
```

Argument *TARGET_ADDRESS* predstavlja naslov ciljnega strežnika, ki ima v produkcijskem okolju vrednost *https://promet.me*.

V povprečju se celoten nabor dimnih testov izvede v enajstih sekundah.

4.3 Avtomatizacija postavitve okolja

Ker našo aplikacijo želimo izvajati v večih različnih okoljih, potrebujemo zanesljiv in enostaven postopek za namestitev in nastavitev vseh potrebnih programskih paketov. Za implementacijo avtomatizacije postavitve okolja smo uporabili programsko opremo Chef, saj ta v svojih receptih omogoča uporabo poznanega jezika Ruby.

4.3.1 Uporaba programa Chef

Za namestitev programa Chef moramo predhodno namestiti naslednje programske pakete:

- izvajalno okolje programskega jezika Ruby,
- zunanjo Ruby knjižnico (angl. *gem*) *Chef*

- zunanjo Ruby knjižnico *librarian-chef* za prenos javno dostopnih zbirk receptov,
- zunanjo Ruby knjižnico *knife-solo* za upravljanje programa Chef brez centralnega strežnika,
- zunanjo Ruby knjižnico *knife-solo_data_bags* za podporo iskanju po podatkovnih vrečah brez centralnega strežnika.

Po tem, ko si izvorno kodo aplikacije prenesemo iz repozitorija, lahko manjkajoče javno dostopne zbirke receptov namestimo tako, da iz direktorija *chef* izvedemo ukaz

```
librarian-chef install.
```

Ob prvi uporabi je potrebno še skonfigurirati podatkovne vreče in skrivni ključ.

Skrivni ključ

Za uporabo šifriranih podatkovnih vreč moramo ob prvi uporabi generirati datoteko s skrivnim ključem. Za ustvarjanje skrivnega ključa smo uporabili odprtokodni programski paket OpenSSL. Ključ dolžine 512 bitov ustvarimo z ukazom

```
openssl rand -base64 512 | tr -d '\r\n' > encrypted_secret.
```

Z vgrajenim programom *tr* odstranimo vse prelome vrstic in s tem preprečimo morebitno okvaro ključa, do katere lahko pride ob prenosu na platformo z drugačnim standardom za prelome vrstic. Skrivnega ključa ne smemo hraniti v repozitoriju izvorne kode, saj bi v primeru nepooblaščenega bralnega dostopa do repozitorija napadalcu omogočili dostop do vseh gesel. Zato je priporočljivo, da na sisteme, ki izvajajo postavitve strežnikov, skrivni ključ prenesemo posebej.

Vnos podatkov v podatkovne vreče

Za delovanje vseh komponent sistema smo morali ustvariti štiri šifrirane in eno nešifrirano podatkovno vrečo. Šifrirane podatkovne vreče vsebujejo občutljive podatke kot so SSL certifikati, privatni ključi, gesla in avtentikacijski žetoni za zunanje storitve. Podatkovno vrečo ustvarimo z ukazom

```
EDITOR=nano knife solo data bag create SKUPINA_VREČE VREČA
 --secret-file encrypted_data_bag_secret.
```

Postavitev strežnika

Za postavitev in nastavitev strežnika, primernega za naš projekt, je potreben vnos ukaza

```
knife solo bootstrap -i ~/.ssh/id_rsa root@IPNASLOV
 nodes/promet_node.json,
```

kjer namesto niza IPNASLOV vstavimo pravi IP naslov našega strežnika. Postopek namestitve in konfiguriranja vseh paketov traja približno deset minut. Ker pa poleg produkcijskega potrebujemo še vmesno okolje, smo postopek ponovili na dveh ločenih strežnikih ponudnika DigitalOcean.

4.3.2 Implementacija postavitve infrastrukture

Za namestitev in konfiguracijo programskih paketov smo uporabili tako javno dostopne zbirke receptov kot tudi tiste, ki smo jih razvili sami. Ker je naš sistem sestavljen iz velikega števila komponent, prav tako pa je postopek testiranja počasen, je implementacija postavitve infrastrukture zahtevala največ časa.

Uporaba šifriranih podatkovnih vreč

Več receptov uporablja podatke iz šifriranih podatkovnih vreč. Za dešifriranje smo uporabili spodnjo programsko kodo:

```
normal_path = "/root/chef-solo/data_bag_key"
vagrant_path = "tmp/vagrant-chef-3/encrypted_secret_key"

if ::File.exists?(normal_path)
 key_file=Chef::EncryptedDataBagItem.load_secret(normal_path)
else
 key_file=Chef::EncryptedDataBagItem.load_secret(vagrant_path)
end
user_data = Chef::EncryptedDataBagItem.load
('ime-skupine','ime-vreče', key_file)

if user_data["private_key"].nil?
 raise "Private key missing from the encrypted databag."
end

certificate_content_64 = user_data["private_key"].
```

Zaradi implementacijskih razlik med orodji *knife-solo* za postavitev strežnika in *Vagrant* za postavitev razvojnega okolja, je bila sprva potrebna vejitev, da smo lahko pridobili pot do datoteke s skrivnim ključem. Z ukazom `Chef::EncryptedDataBagItem.load ('ime-skupine','ime-vreče', secret_file)` pridobimo dešifrirano vsebino celotne podatkovne vreče. V primeru, da podatkovna vreča ne vsebuje vseh potrebnih atributov, se z namenom lažjega odkrivanja napak pri postavitvi sproži izjema.

Okoljske spremenljivke

Za pravilno delovanje implementiranega sistema za varnostno kopiranje podatkovne baze oblačna datotečna storitev Amazon S3 zahteva vnos ustreznih avtentikacijskih žetonov. V podatkovni vreči *s3.json* sta zahtevana atributa *access_key_id* in *secret_access_key*, ki sta aplikaciji dostopna z uporabo istoimenskih okoljskih spremenljivk. Omenjena atributa ustvarimo z uporabo spodnje programske kode:

```
execute "add s3 access key id" do
  user "deploy"
  command "echo export ACCESS_KEY_ID=#{key_id} |
   cat - /home/deploy/.bashrc > /tmp/out &&
    mv /tmp/out /home/deploy/.bashrc"
  not_if { `grep ACCESS_KEY_ID /home/deploy/.bashrc
   | wc -w`.chomp == "2"}
end.
```

Da v sistemu obdržimo okoljsko spremenljivko, jo moramo vnesti v datoteko *.bashrc*. Okoljsko spremenljivko s preusmeritvjo vsebine v začasno datoteko dodamo na konec datoteke, vendar moramo za zagotavljanje idempotentnosti ukaza pred tem preveriti, ali je ista okoljska spremenljivka že prisotna. Pri tem si pomagamo z vgrajenima programoma *grep* in *wc*.

Nginx

Uporaba spletnega strežnika Nginx zahteva nastavitev dveh večjih sklopov - konfiguracijske datoteke in SSL certifikatov. Najprej moramo s pomočjo datotečne predloge v direktoriju */etc/nginx/sites-enabled/current* ustvariti konfiguracijsko datoteko z imenom *nginx.conf*. S konfiguracijsko datoteko lahko nastavimo naslov aplikacijskega strežnika, podatke o implementaciji SSL protokola in omrežna vrata, na katerih bo strežnik sprejemal HTTP zahteve. Ob izvedbi recepta z uporabo predloge vnesemo podatka o IP naslovu strežnika in lokaciji izvorne kode, ustvariti pa moramo tudi direktorij */etc/nginx/ssl*, kamor se shranjujejo SSL certifikati in privatni ključi.

Zaradi zagotavljanja kompatibilnosti spletne storitve z mobilno aplikacijo je spletna aplikacija prek protokola SSL dosegljiva na dveh končnih točkah. Prva končna točka je enaka IP naslovu spletnega strežnika in so jo uporabljale starejše verzije mobilne aplikacije v kombinaciji s samopodpisanimi certifikati. Druga končna točka pa je domena *promet.me*, ki uporablja certifikat, podpisan s strani izdajatelja digitalnih potrdil, in jo uporabljajo tako novejše verzije mobilne aplikacije kot tudi spletni vmesnik aplikacije.

Za pravilen zagon strežnika in delovanje SSL protokola moramo ustvariti podatkovno vrečo z imenom *nginx.json* in atributi *public_certificate*, *private_key*, *promet_me_certificate* in *promet_me_key*. Preden ključe in certifikate shranimo v podatkovno vrečo, jih moramo pretvoriti v zapis, ki uporablja Base64 kodiranje. Na ta način se izognemo morebitnim napakam pri postavitvi zaradi neveljavnih znakov ali prelomov strani. Kodiranje lahko izvedemo z uporabo modula *Base64*, ki je vključen v standardno knjižico programskega jezika Ruby, s spodnjim ukazom:

```
require 'base64'
Base64.encode64(File.read("server.crt")).
```

Varnish

Za namestitev programa Varnish smo uporabili javno dostopno zbirko receptov, ki je z izjemo predloge konfiguracijske datoteke nismo spreminjali.

Postgres

Postavitev podatkovne baze Postgres in njeno pripravo za uporabo v ogrodju Rails smo implementirali v zbirki receptov *setup_database_yml*. Nastavitve za vzpostavitev podatkovne baze se nahajajo v podatkovni vreči *postgres.json*. Ime podatkovne baze nastavimo z atributom *db_name*, uporabniško ime in geslo pa z atributoma *db_username* in *db_password*. Podatkovno bazo in novega uporabnika le-te moramo ustvariti preko ukazne vrstice `psql`. Spodnja izvorna koda prikazuje ustvarjanje novega uporabnika podatkovne baze:

```
execute "create new postgres user" do
  user "postgres"
  command "psql -c \"create user #{db_username} with password
  '#{db_password}' superuser;\""
  not_if {`sudo -u postgres psql -tAc \"SELECT * FROM pg_roles
  WHERE rolname='#{db_username}'\" | wc -l`.chomp == "1" }
end.
```

Idempotentnost ustvarjanja uporabnika podatkovne baze zagotovimo s preverjanjem, ali isti uporabnik že obstaja (SQL poizvedba `SELECT * FROM pg_roles WHERE rolname='#db_username`, kjer spremenljivka *db_username* predstavlja uporabniško ime).

Ogrodje Rails avtentikacijske podatke za povezavo s podatkovno bazo poišče v datoteki *database.yml* v direktoriju aplikacije *config*. Omenjeno datoteko ustvarimo s spodnjim zaporedjem ukazov :

```
template "#{node['cap_base']}/shared/config/database.yml" do
        source 'database.yml.erb'
        owner 'deploy'
        mode '0644'
        variables({db_name: db_name, db_username: db_username,
         db_password: db_password })
end.
```

Poleg imena predloge datoteke moramo navesti tudi parameter *variables*, ki vsebuje imena spremenljivk, ki bodo v predlogi na voljo za uporabo. Slednje je potrebno za preprečevanje podvajanja izvorne kode v receptu in predlogi datoteke.

Ker v repozitoriju izvorne kode ne želimo hraniti čistopisov gesel, je za uporabo orodij *pg_restore* in *pg_dump* potrebna še konfiguracija datoteke *.pgpass*, ki vsebuje avtentikacijske podatke za povezavo na lokalno podatkovno bazo.

Dnevniške datoteke

Za lažje odkrivanje napak in zbiranje metrik o delovanju aplikacije se pogosto zanašamo na dnevniške datoteke. Ker aplikacija neprestano teče in sproti beleži vse HTTP zahteve, lahko dnevniške datoteke presežejo predvidene velikosti in s tem zasedejo trdi disk. Da bi porabo virov omejili, smo se odločili za sistem izmenjevanja dnevniških datotek z uporabo privzeto nameščenega orodja `logrotate`. V direktoriju */etc/logrotate.d/* z uporabo predloge ustvarimo datoteko s sledečo vsebino:

```
<%= node['capistrano_base']%>/shared/log/*.log {
        weekly
        missingok
        rotate 52
        compress
        delaycompress
        notifempty
        copytruncate
}.
```

Dnevniške datoteke v direktoriju aplikacije `log` se tedensko stiskajo in arhivirajo, na trdem disku pa je v vsakem trenutku zadnjih 52 arhivov dnevniških datotek.

Dodatne programske knjižnice

Zaradi uporabe velikega števila programskih paketov smo morali namestiti določene knjižnice, ki so potrebne za delovanje omenjenih programov. Za pravilno delovanje zunanje Ruby knjižnice *Rgeo* je potrebna namestitev paketa *libgeos++-dev*, ki nudi preprost vmesnik za delo s projekcijskimi sistemi in koordinatami. Dimni testi po postavitvi aplikacije se izvajajo s programom Capybara, ki uporablja brezoblični brskalnik PhantomJS - ta za delovanje zahteva namestitev istoimenskega paketa. Zunanja Ruby knjižnica *Grit*, ki se uporablja za primerjavo različnih vej git repozitorija, pa zahteva namestitev paketa *libicu-dev*.

Požarni zid

Operacijski sistem Ubuntu ima privzeto nameščen požarni zid `iptables`. Za upravljanje pravil požarnega zidu smo uporabili javno dostopno zbirko receptov *iptables-ng*. Pravila za vhodne povezave so zapisana v JSON datoteki vloge *firewall*. Privzeto se vhodni paketi zavržejo, dovoljeni pa so paketi iz lokalnega vmesnika in že vzpostavljene povezave. Za pravilno delovanje sistema je potrebno omogočiti prehod paketom na naslednjih omrežnih vratih:

- vrata 22 (protokol SSH, za oddaljeni dostop),
- vrata 80 (protokol HTTP, uporablja spletni strežnik),
- vrata 443 (HTTPS, za spletni strežnik),
- vrata 11371 (uporablja GNU Privacy Guard za preverjanje integritete paketov ob namestitvi).

Pravila za požarni zid so shranjena v JSON datoteki v obliki, podobni nastavitvenim ukazom požarnega zidu iz ukazne vrstice:

```
"filter": {
  "INPUT": {
    "default": "DROP [0:0]",
    "loop": {
      "rule": "-i lo -j ACCEPT"
    },
    "established":{
    "rule": "-m conntrack --ctstate
    RELATED,ESTABLISHED -j ACCEPT"
    },
    "ssh": {
      "rule": "-p tcp --dport 22 -j ACCEPT"
    },
    "http": {
      "rule": "-p tcp --dport 80 -j ACCEPT"
    },
    "https": {
      "rule": "-p tcp --dport 443 -j ACCEPT"
    },
    "gpg": {
      "rule": "-p tcp --dport 11371 -j ACCEPT"
    }
```

```
  }
}.
```

Uporabniški račun

Postavitev aplikacije in ostala skrbniška opravila izvaja uporabniški račun *deploy*, ki smo ga ustvarili z receptom *deploy_user*. Slednji služi kot ovojnica za javno dostopno zbirko receptov *users*, ki omogoča umestitev uporabnika v skupino uporabnikov, izbiro izvršilne vrstice in izbiro javnih ključev uporabnikov, ki lahko dostopajo do sistema preko protokola SSH. Dostop do sistema moramo dovoliti razvijalcem in storitvi Codeship, ki bo izvajala postavitev naše aplikacije. Ker v sistem vnašamo javne ključe, ni potrebno šifriranje podatkovne vreče, vendar pa moramo v podatkovno vrečo *deploy.json* obvezno vnesti atribute *home*, *shell*, *groups* in *ssh_keys*.

APNS

APNS (Apple Push Notification Service) je storitev za pošiljanje obvestil na mobilne naprave z operacijskim sistemom iOS. V produkcijem okolju za pošiljanje obvestil potrebujemo certifikat v podatkovni vreči *push_service*, ki ima atribut *production_certificate*. Ko se izvede recept, se certifikat namesti v direktorij aplikacije *shared*.

RVM

RVM (Ruby Version Manager) konfiguriramo z zbirko receptov *rvm*. Z uporabo recepta *rvm::system* na sistem namestimo izvajalno okolje programskega jezika Ruby. Konfiguracijo programa RVM hranimo v JSON datoteki vloge *ruby-rails*. Z uporabo atributa *rubies* lahko za namestitev izberemo več različnih verzij izvajalnega okolja Ruby. Za hitrejšo namestitev novih zunanjih Ruby knjižnic pa lahko dodamo stikali `--no-ri` in `--no-rdoc`, ki preprečita namestitev in indeksiranje dokumentacije. Da prihranimo na času pri prvi postavitvi aplikacije, za globalno namestitev izberemo tudi večje zunanje Ruby knjižnice.

```
"default_attributes": {
        "rvm": {
          "install_rubies" : true ,
          "rubies": ["2.2.1"] ,
          "default_ruby": "2.2.1",
          "rvmrc_env": {
                "rvm_gem_options": "--no-ri --no-rdoc"
          },
          "gems": {
                "2.2.1": [
                  {"name": "rails", "version": "4.2.1"},
                  {"name": "rgeo"},
                  {"name": "pg"},
                  {"name": "nokogiri"}
                ]
          }
        }
  }
```

Redis

Program Redis smo namestili s pomočjo receptov *redisio* in *redisio::enable*, ki ju najdemo v javni zbirki receptov *redisio*.

4.3.3 Razvojno okolje - Vagrant

Za implementacijo postavitve razvojnega okolja poskrbita programa Vagrant in Chef. Glavna funkcija programa Vagrant je ustvarjanje in nastavitev navideznih razvojnih okolij. Uporablja se kot visokonivojska ovojnica nad programsko opremo za virtualizacijo kot sta VirtualBox in VMware. Vagrant razvijalcu omogoči hitro in enostavno ustvarjanje novih instanc razvojnega okolja na vseh družinah pogostih operacijskih sistemov - Linux, Windows in OS X.

Zaradi uporabe programa Vagrant za nastavitev želenega navideznega sistema razvijalcu ni potrebno prenašati velikih namestitvenih datotek s spleta - potrebuje zgolj vsebino direktorija *chef* in datoteko *Vagrantfile*, ki v celoti vsebuje navodila za nastavitev in namestitev razvojnega okolja. Vagrant pa omogoča tudi prenos že vnaprej pripravljenih verzij pogosto uporabljanih operacijskih sistemov, kot je Ubuntu. Želeno verzijo izberemo z ukazom

```
config.vm.box = "ubuntu/trusty64",
```

v zgornjem primeru tako nameščamo uradno 64-bitno verzijo operacijskega sistema Ubuntu 14.04.

Za lažje testiranje aplikacije lahko omrežni promet iz vrat navideznega sistema preusmerimo na gostiteljeva omrežna vrata z ukazom

```
config.vm.network "forwarded_port", guest: 3000, host: 3001.
```

Zgornji ukaz nam omogoči dostop do aplikacijskega strežnika na računalniku gostitelja na spletnem naslovu *http://localhost:3001*.

Izvorno kodo projekta v delovnem direktoriju v navidezni sistem prenesemo z ukazom

```
  config.vm.synced_folder "", "/vagrant".
```

V tem primeru je celotna izvorna koda, ki se nenehno sinhronizira, dostopna v direktoriju */vagrant* na navideznem sistemu.

Konfiguracijski blok *provision* določa tip agenta za nastavitev skupaj z ustreznimi nastavitvami za namestitev vseh programskih paketov. Agent *chef_solo* nam omogoča ponovno uporabo že napisanih receptov in vlog, ki smo jih predhodno uporabili za postavitev strežnika. Poleg določitve izbranih vlog je potrebna tudi izrecna navedba lokacij zbirk receptov, podatkovnih vreč, vlog in skrivnega ključa, kot prikazuje spodnji izsek izvorne kode.

```
config.vm.provision "chef_solo" do |chef|
    chef.cookbooks_path = "chef/cookbooks"
    chef.provisioning_path = "/tmp/vagrant-chef-3"
```

```
    chef.data_bags_path = "chef/data_bags"
    chef.encrypted_data_bag_secret_key_path = "chef/bag_key"
    chef.roles_path = "chef/roles"
    chef.add_role("general")
    chef.add_role("postgres")
    chef.add_role("geo")
    chef.add_role("ruby-rails")
    chef.add_role("redis")
    chef.add_role("web")
  end
```

Postopek konfiguracije okolja za razvoj se nekoliko razlikuje od konfiguracijskega postopka za produkcijsko okolje. V okolju za razvoj smo odstranili vlogo za nastavitev požarnega zidu in recept za varno nastavljanje podatkovne baze, saj zgornji opravili prepuščamo razvijalcu - v kolikor ju potrebuje, ju lahko na enostaven način doda sam.

Uporaba programa Vagrant

Razvijalec mora za zagon navideznega razvojnega okolja izvesti ukaz

`vagrant up,`

ki ob prvi uporabi poskrbi za inicializacijo okolja in agenta *chef_solo*. Razvojno okolje ustavimo z ukazom

`vagrant halt,`

njegovo uničenje v primeru napake v konfiguraciji pa z ukazom

`vagrant destroy.`

Razvijalec dostop do ukazne vrstice razvojnega okolja dobi z ukazom

`vagrant ssh,`

to pa mu omogoča izvajanje testov, zagon aplikacijskega strežnika in nadaljnjo uporabo poljubnih ukazov.

4.4 Avtomatizacija postavitve aplikacije

Za avtomatizacijo vseh korakov postavitve aplikacije smo izbrali program Capistrano, saj privzeto vsebuje številne razširitve za lažje delo z ogrodjem Ruby on Rails. Capistrano omogoča avtomatizirano upravljanje z različnimi verzijami izvorne kode in uporabo obstoječih ter po potrebi ustvarjanje novih direktorijev. Za postavitev naše aplikacije smo omogočili še dodatne module *bundler*, *rails/assets* in *rvm*. Tako smo v relativno majhnem številu korakov dosegli avtomatiziran prenos novih verzij izvorne kode repozitorija na strežnik, namestitev potrebnih zunanjih knjižnic, predprevajanje (angl. *precompilation*) slik, slogovnih in JavaScript datotek ter izvajanje ukazov v primernem izvajalnem okolju programskega jezika Ruby.

Postopek postavitve aplikacije določa konfiguracijska datoteka *deploy.rb*, ki se nahaja v direktoriju *config*. Večino nastavitev lastnosti postavitve opravimo s klicem metode *set*, ki kot argumenta sprejme ime spremenljivke in njeno vsebino. Po tem so spremenljivke globalno dostopne v vseh opravilih programa Capistrano.

Za uspešno postavitev moramo najprej vnesti spletni naslov in tip našega repozitorija izvorne kode, ki ju podamo v spremenljivkah *scm* in *repo_url*. Spremenljivki *deploy_to* in *tmp_dir* hranita poti do direktorijev za shranjevanje novih verzij aplikacije in začasnih datotek.

Program Capistrano na strežniku privzeto hrani zadnjih pet verzij izvorne kode aplikacije. Katera verzija aplikacije je v danem trenutku aktivna, lahko določimo tako, da v direktoriju *current* ustvarimo simbolno datotečno povezavo na datoteke z izbrano verzijo izvorne kode. Za pravilno delovanje aplikacije moramo zagotoviti dostop do datotek z izvorno kodo tudi med postopkom postavitve in med menjavanjem simbolne datotečne povezave, zato moramo ustrezno nastaviti vrednost spremenljivke *linked_dirs*. V ta namen moramo podati seznam direktorijev, ki jih želimo ohraniti med postavitvami, in bodo z uporabo simbolne povezave povezani v direktorij *shared*. Tipično so to direktoriji, ki vsebujejo dnevniške datoteke, datotečni predpomnilnik, vtičnice in datoteke z identifikatorji delujočih procesov.

Implementirana opravila za urejanje postavitve se z uporabo povratnih klicev izvedejo v različnih stopnjah le-te. Lastnosti povratnih klicev nastavljamo s pomočjo kvalifikatorjev *before* in *after*, ki kot argument sprejmeta ime stopnje postavitve in ime Capistrano opravila, ki se privzeto nahaja v direktoriju *lib/capistrano/tasks*.

Za implementacijo želene rešitve smo uporabili povratne klice različnih korakov postavitve. Ob koncu vsake postavitve (v stopnji *deploy*) se izvedeta opravili ponovnega zagona delovnih procesov in aplikacijskega strežnika. V stopnji *deploy:updated* se izvede opravilo za migracijo podatkovne baze. Za obnovitev prejšnje verzije izvorne kode aplikacije in podatkovne baze pa se po izvedbi stopnje *deploy:rollback* izvede opravilo `restore_application`.

Ciljna okolja in pripadajoče skupine strežnikov moramo definirati v datotekah *production.rb* in *staging.rb*, ki se nahajata v direktoriju *config/deploy*. Gre za nastavitveni datoteki, ki vsebujeta podatke o IP naslovih skupine strežnikov, uporabniških imenih in vlogah posameznih strežnikov. Razdeljevanje strežnikov v skupine omogoča enostavno izvrševanje nalog na poljubnem številu strežnikov.

4.4.1 Brezprekinitvena postavitev aplikacije

Pogoste postavitve novih verzij aplikacije predstavljajo velik izziv za karseda nemoteno delovanje spletne storitve. Najpreprostejša rešitev, ki vključuje začasno prekinitev delovanja, je uvedba predvidenih časovnih oken za vzdrževanje. V izbranih časovnih oknih je storitev običajno deloma nedosegljiva. V primeru zvezne postavitve, kjer se nove verzije aplikacije lahko postavljajo tudi večkrat na dan, pa slednje ne pride v poštev, saj bi prihajalo do neprestanih motenj v delovanju spletne storitve, kar močno vpliva na uporabniško izkušnjo.

Brezprekinitveno (angl. *zero downtime*) delovanje spletne storitve zagotavlja, da med prehajanjem na novo verzijo aplikacije, ki teče na strežniku, prihajajoče HTTP zahteve obravnavamo kot običajno. Posledično ne pride do izgub HTTP zahtev ali do dolgega čakanja na strežnikov odziv. Meha-

nizme za brezprekinitveno delovanje storitve lahko vzpostavimo na različnih nivojih sistema.

Ena izmed možnih rešitev je uporaba programske opreme za izenačevanje obremenitve (angl. *load balancing*) na nivoju HTTP protokola, kot je HAProxy ali Varnish. Za brezprekinitveno delovanje ju moramo postaviti pred aplikacijske strežnike, tako da jim posredujeta vse zunanje HTTP zahteve. Ta pristop zahteva preverjanje dosegljivosti vsakega izmed aplikacijskih strežnikov, saj mora izenačevalnik obremenitve ob sprejemu nove zahteve posredovati to zahtevo prvemu dosegljivemu aplikacijskemu strežniku. V postopku postavitve nove verzije aplikacije tako zaporedoma nadgrajujemo vsak aplikacijski strežnik posebej - najprej ga označimo kot nedosegljivega, nato izvedemo vse potrebne postopke, na koncu pa ga ponovno označimo kot dosegljivega.

Glavna pomanjkljivost tega pristopa je, da lahko pride do prekinitve obstoječe zahteve v kolikor je uporabnik komuniciral z aplikacijskim strežnikom, ki smo ga v istem trenutku nadgrajevali. Zato smo se raje odločili za uporabo mehanizma za brezprekinitveno delovanje na nivoju aplikacijskega strežnika Puma.

Ukaz za fazni ponovni zagon (angl. *phase restart*), ki ga glavnemu procesu Puma posredujemo s signalom *SIGUSR1*, povzroči, da aplikacijski strežnik za vsak delovni proces posebej sproži postopek nadgradnje. Pri tem upošteva, da ima proces lahko še aktivne spletne zahteve in zato počaka, da se vse zaprejo. Ta pristop zagotavlja, da je v vsakem trenutnu na voljo vsaj en delovni proces, ki obdeluje nove zahteve, in da ne prekinjamo obstoječih zahtev.

4.4.2 Zagon aplikacijskega strežnika

Ponovni zagon aplikacijskega strežnika smo implementirali s Capistrano opravilom `phase_restart_application`. Za pravilno izvedbo `rake` opravila `puma:phase_restart`, ki vsebuje izvorno kodo za interakcijo z aplikacijskim strežnikom, je potreben klic opravila znotraj blokov *on roles*, *within path* in

with RAILS_ENV. Ukaz se mora namreč izvršiti na strežnikih z vlogo *app* v trenutno aktivnem direktoriju z izvorno kodo aplikacije in v primernem izvajalnem okolju. Pravilen način klica opravila ima naslednjo strukturo:

```
desc "Phase restart application server"
task :phase_restart_application do
  on roles(:app) do
    within "#{fetch(:deploy_to)}/current/" do
      with RAILS_ENV: fetch(:environment) do
        execute :rake, "puma:phase_restart"
      end
    end
  end
end.
```

Interakcija z aplikacijskim strežnikom Puma je odvisna od trenutnega stanja strežnika, zanjo pa skrbi upravljalski program *pumactl*. Ustrezne klice in preverjanja stanja strežnika smo implementirali v `rake` opravilu `puma`, ki ima naslednjo vsebino:

```
PUMA_PID = File.expand_path("/tmp/pids/puma.pid", __FILE__)
namespace :puma do
  desc "start Puma server"
  task :start => :environment do
      puts "Puma is starting"
      system 'RACK_ENV=production pumactl start'
    end
  end

  desc "phase restart puma"
  task :phase_restart => :environment do
    if puma_is_running?
      puts "Puma is phase restarting"
```

```
      system 'RACK_ENV=production
      pumactl -P tmp/pids/puma.pid phased-restart'
    else
      Rake::Task["puma:start"].invoke
    end
  end

  def puma_is_running?
    File.exists?(PUMA_PID) &&
    system("ps -p `cat #{PUMA_PID}` 2>&1 > /dev/null")
  end
end
```

Stanje aplikacijskega strežnika preverjamo v dveh korakih. Najprej z uporabo metode *File.exists?* preverimo obstoj datoteke z identifikatorjem procesa v direktoriju *tmp/pids*, ki jo vsak proces ustvari ob zagonu. Če datoteka obstaja, preverimo, ali se trenutno izvaja proces z danim identifikatorjem. Pri tem si pomagamo z ukazom `ps`, ki kot argument sprejme vsebino datoteke z identifikatorjem procesa.

Če se aplikacijski strežnik v produkcijskem okolju ne izvaja, ga zaženemo s klicom programa *pumactl* in argumentom *start*. V nasprotnem primeru pa kot argumente uporabimo stikalo *P*, pot do datoteke z identifikatorjem procesa in tip ukaza *phased-restart*.

4.4.3 Zagon delovnih procesov

Ob vsaki postavitvi aplikacije je potrebno ponovno zagnati delovne procese zunanje Ruby knjižnice Sidekiq, ki skrbijo za osveževanje prometnih podatkov. Poleg tega moramo izprazniti podatkovno zbirko Redis, v kateri hranimo serializirano izvorno kodo posameznih Ruby razredov. Ta korak je potreben, ker se delovni procesi, ki se morajo izvesti vsako minuto, hranijo v podatkovni zbirki Redis in poznejše spreminjanje izvorne kode na njihovo izvajanje nima vpliva.

Ponovni zagon delovnih procesov smo implementirali s Capistrano opravilom `restart_workers`, ki ima naslednjo vsebino:

```
desc "Restart sidekiq workers"
task :restart_workers do
  on roles(:app) do
    within "#{fetch(:deploy_to)}/current/" do
      with RAILS_ENV: fetch(:environment) do
        execute :rake, "sidekiq:stop"
        execute "redis-cli flushdb"
        execute :rake, "sidekiq:start"
      end
    end
  end
end.
```

Da se izvede najnovejša verzija izvorne kode, je v ustreznem direktoriju najprej potrebno izvesti `rake` opravilo `sidekiq:stop`. Slednje z uporabo upravljalnega procesa *sidekiqctl* glavnemu Sidekiq procesu pošlje sistemski signal za ustavitev (*TERM*). Za tem moramo preko konzole programa Redis izvršiti ukaz `flushdb`, ki izbriše vse ključe in pripadajoče vrednosti trenutne podatkovne zbirke. Zagon delovnih procesov se nato izvede z uporabo `rake` opravila `sidekiq:start`.

4.5 Podatkovne migracije

Dodajanje, odstranjevanje in spreminjanje tabel ter stolpcev podatkovne baze se v ogrodju Ruby on Rails izvaja z uporabo podatkovnih migracij. Datoteke s podatkovnimi migracijami vsebujejo enostavna navodila za interakcijo s strukturo podatkovne baze. Migracijsko datoteko, ki ustvari tabelo za shranjevanje podatkov o mobilnih napravah, ustvarimo z ukazom

```
rails generate migration Device token:string user_id:integer.
```

Zgornji ukaz v direktoriju *db/migrate* ustvari datoteko *20150714132554_create_devices.rb*, v kateri se nahaja spodnja koda:

```
class CreateDevices < ActiveRecord::Migration
  def change
    create_table :devices do |t|
      t.string :token
      t.integer :user_id
      t.timestamps
    end
  end
end.
```

Datoteke s podatkovnimi migracijami se nahajajo v direktoriju *db/migrate* in jih lahko izvedemo z ukazom `rake db:migrate`. Ogrodje Rails preveri prisotnost vseh potrebnih tabel in njihovih stolpcev ter poskrbi za izvedbo potrebnih sprememb, ki vodijo do želene sheme podatkovne baze.

4.5.1 Zaznavanje nevarnih podatkovnih migracij

V produkcijskem okolju smo zaradi uporabe podatkovnih migracij postavljeni pred dva izziva.

Prvi je posledica dejstva, da zaradi uporabe rešitev za brezprekinitveno delovanje v danem trenutku lahko sočasno tečeta starejša in nova verzija aplikacije. Zato lahko pride do situacije, ko obe verziji istočasno dostopata do podatkovne baze. To pomeni, da morajo vse spremembe podatkovne sheme in aplikacijske kode, ki jih vnesemo s podatkovnimi migracijami ogrodja Ruby on Rails, biti kompatibilne s prejšnjo verzijo podatkovne sheme.

Drugi izziv predstavljajo morebitni izpadi storitve zaradi napačnih nastavitev podatkovnih migracij. Nekateri ukazi za delo s podatkovno bazo namreč zahtevajo previdno uporabo, saj lahko pri njihovi uporabi nad večjim naborom zapisov privedejo do dolgotrajnega zaklepanja tabel.

Za samodejno zaznavanje potencialno nevarnih migracij smo implementirali razred *MigrationDetector*, ki ga na strežniku za zvezno integracijo ob vsakem dodajanju kode v repozitorij požene ukaz `rake migration:check`. Za implementacijo samodejnega zaznavanja nevarnih podatkovnih migracij smo potrebovali še zunanjo Ruby knjižnico *Grit*, ki ponuja programski vmesnik za lokalni Git repozitorij in nam omogoča primerjanje prispevkov kode obeh vej repozitorija - glavne in trenutne veje iz zahteve za pregled. Iz rezultatov primerjave lahko z uporabo regularnega izraza */ActiveRecord::Migration/* izberemo izključno datoteke, ki vsebujejo podatkovne migracije. Dobljene datoteke so s tem pripravljene za obdelavo v naslednjih korakih zaznavanja nevarnih podatkovnih migracij, ki jih bomo predstavili v naslednjih razdelkih.

Odstranjevanje stolpca

Naš sistem kot potencialno nevarno migracijo zazna prisotnost uporabe stolpca tabele v glavni veji izvorne kode, ki ga bo podatkovna migracija iz veje z zahtevo za pregled odstranila.

Za pravilno odstranjevanje stolpca iz tabele moramo izvesti dva koraka. Najprej moramo v ločeni zahtevi za pregled poskrbeti za postavitev verzije aplikacije, ki nikjer v kodi ne uporablja stolpca, ki ga želimo odstraniti. Šele po tem lahko ustvarimo novo zahtevo za pregled, ki vsebuje podatkovno migracijo za odstranitev stolpca. Tak postopek preprečuje, da bi med postavitvijo prišlo do napake pri obstoječi spletni zahtevi uporabnika v primeru, da bi jo obdelal strežnik s prejšnjo verzijo aplikacijske kode. Podatkovne migracije se namreč izvedejo pred postavitvijo nove verzije aplikacije. Primere neustreznega odstranjevanja stolpca lahko zaznamo z ukazom `grep`, ki ga v okolju Ruby izvedemo z uporabo klica metode *popen3* modula *Open3* z argumentom

```
"grep -rl -n -E \"#{'\.' + attribute_name}|:#{attribute_name}\"
 master/app master/lib master/spec",
```

kjer spremenljivka *attribute_name* predstavlja ime stolpca za odstranitev. Zgornji ukaz rekurzivno preišče vse pomembne direktorije aplikacije za pojavitvami izbranega stolpca. V primeru ujemanja z regularnim izrazom se sproži napaka razreda *DangerousMigrations*, ki ustavi postopek postavitve in uporabniku izpiše opozorilo:

```
rake migration:check
Warning, the following files include usage of
the removed "push_token" attribute on the master branch
master/spec/controllers/apis/v1/user_controller_spec.rb
master/spec/services/purchase_subscription_spec.rb
rake aborted!
MigrationDetector::DangerousMigrations.
```

Da se v čim večji meri izognemo napačno zaznanim potencialno nevarnim podatkovnim migracijam, se rekurzivno preverjajo samo direktoriji *app*, *lib* in *spec*, ki vsebujejo izvorno kodo modelov, kontrolerjev, pogledov, storitvenih objektov, delovnih procesov in teste.

Dodajanje indeksov

Operacija dodajanja indeksov pri podatkovni bazi Postgres onemogoči spreminjanje tabele, na kateri ustvarjamo indeks. Zaklepanje tabele za pisalne dostope lahko vodi v prekinitev delovanja storitve, saj za čas operacije dodajanja indeksov ostali deli aplikacij ne morejo spreminjati podatkov iz dane tabele. Postopek indeksiranja lahko na tabelah z veliko vnosi traja več minut.

Podatkovna baza Postgres ima za zmanjševanje nedosegljivosti za ukaz `CREATE INDEX` na voljo opcijo `CONCURRENTLY`, ki indeks ustvari v dveh prehodih, brez onemogočanja pisalnih dostopov za SQL ukaze kot so `INSERT`, `UPDATE` in `DELETE`. Za uporabo omenjene opcije je bil v modulu *ActiveRecord* verzije 4 ogrodja Ruby on Rails predstavljen argument *algorithm: :concurrently* za metodo *add_index* [24]. Uporaba opcije `CONCURRENTLY` pa zahteva ustvarjanje indeksov izven transakcije, kar je v nasprotju s privzetim delovanjem modula *ActiveRecord*. Razredi podatkovnih migracij morajo zato

vsebovati še direktivo *disable_ddl_transaction!* [25]. Direktiva je omejena samo na datoteko z migracijo, v kateri je napisana. Vse ostale migracije iz drugih datotek se izvedejo v transakcijah, zato je priporočeno, da migracijski razred z omenjeno direktivo shranimo v posebno datoteko. Ustrezna podatkovna migracija mora imeti naslednjo obliko:

```
class AddIndexToDevicesActive < ActiveRecord::Migration
  disable_ddl_transaction!
  def change
    add_index :devices, :active, algorithm: :concurrently
  end
end.
```

Zaznavanje neustreznega tipa podatkovne migracije se nahaja v metodi *find_incorrect_index*. Z uporabo regularnega izraza */add_index/* poiščemo migracije, ki vsebujejo dodajanje indeksa na stolpec. Nad dobljenimi migracijami se nato izvedeta še dve preverjanji. Prvo preverjanje uporablja regularni izraz */disable_ddl_transaction!/* in v primeru zadetka sproži izjemo razreda *MissingIndexTransaction*, drugo preverjanje pa v primeru zadetkov pri regularnem izrazu */algorithm: :concurrently/* sproži izjemo razreda *MissingConcurrentIndex*.

Preskočitev preverjanja podatkovnih migracij

Implementirani način zaznavanja potencialno nevarnih podatkovnih migracij lahko kot nevarne migracije opredeli tudi tiste, ki to niso. Ker tudi lažno pozitivne migracije onemogočajo postavitev aplikacije, potrebujemo mehanizem za preskočitev njihovega preverjanja. Razvijalec lahko rezultat preverjanja, ali je migracija zaznana kot nevarna, preveri na Github strani zahteve za pregled. V kolikor je prepričan, da migracija ni nevarna, njeno preverjanje preskoči tako, da v sporočilo prispevka kode doda besedilo *[migration-detection-skip]*.

Preverjanje se preskoči tudi v primeru, da zahtevo za pregled sprejmemo in njeno vejo združimo z glavno vejo. To je potrebno, ker so vsi prispevki

kode v repozitorij testirani, in je tudi združitev z glavno vejo obravnavana kot prispevek kode.

4.5.2 Varnostne kopije podatkovne baze

Ob postavitvi aplikacije se v sklopu Capistrano koraka *deploy:updated* ustvari varnostna kopija podatkovne baze v produkcijskem okolju in se prenese v datotečno storitev v oblaku Amazon S3. Na ta način želimo zmanjšati verjetnost, da bomo izgubili uporabniške podatke, prav tako pa želimo omogočiti enostavno prehajanje na prejšnje stanje podatkovne baze v primeru napak podatkovne sheme.

Varnostno kopiranje podatkovne baze smo implementirali v Capistrano opravilu `backup_db`. Podatki za avtentikacijo podatkovne baze se preberejo iz datoteke *database.yml*, ki se nahaja na strežniku, nato pa se z uporabo programa `pg_dump` izvozi binarna datoteka z vsemi zapisi baze in podatkovno shemo. Ukaz `pg_dump` kličemo na naslednji način:

```
pg_dump -U #{db_user} -h localhost --data-only -F c
--exclude-table=schema_migrations #{db_name} > #{backup_path}.
```

Direktorij na strežniku, kamor se začasno shrani izbrana datoteka, se določi dinamično glede na nastavitve programa Capistrano, in ima sledečo obliko:

```
/home/#{fetch(:deploy_user)}/database_backups/#{backup_name}.
```

Datoteko z varnostno kopijo prenesemo s strežnika v začasni direktorij na sistemu s Capistrano ukazom `download!`, ki izvaja postavitev. Za nalaganje na datotečno storitev v oblaku S3 se nato uporabi razred *S3Uploader*. Avtentikacija z S3 storitvijo poteka z uporabo avtentikacijskih ključev, ki so na produkcijskem strežniku shranjeni v obliki okoljskih spremenljivk (*ACCESS_KEY_ID* in *SECRET_ACCESS_KEY*). Varnostna kopija podatkovne baze se najprej prenese v vnaprej določeno vedro z imenom *promet-production*, nato pa se ji nastavijo še dostopne pravice (angl. *ACL, Access Control Lists*), ki so privzeto nastavljene na javni dostop.

4.5.3 Produkcijski podatki v vmesnem okolju

Vsako novo funkcionalnost aplikacije želimo testirati na realnih podatkih - zaznati želimo dolgotrajne podatkovne migracije, robne primere uporabniških podatkov in morebitne vplive na zmogljivost aplikacije. Zato varnostne kopije podatkovne baze hranimo v produkcijskem okolju.

Obnovitev produkcijskih podatkov v vmesnem okolju smo implementirali v Capistrano opravilu `load_production_db`. Ob vsaki postavitvi v vmesno okolje se na računalnik, s katerim izvajamo postavitev, najprej prenese ustrezna binarna datoteka z varnostno kopijo podatkovne baze. Pri tem nam pomaga razred *S3Downloader*. Omenjena datoteka se nato z uporabo Capistrano ukaza `upload!` prenese na ciljni strežnik v vnaprej določen direktorij. Zasebna metoda *prepare_for_restore* najprej poskrbi za ustavitev aplikacijskega strežnika, delovnih procesov in izbris ključev programa Redis, za tem pa se z zaporedjem ukazov `rake db:drop db:migrate db:create` izvede ponovna postavitev podatkovne baze. Na ciljnem strežniku se nazadnje zažene še program `pg_restore`, ki je del programskega paketa Postgres in ga zaženemo na naslednji način:

```
pg_restore -U #{db_user} -h localhost --data-only --clean -d
#{db_name} < #{remote_path}."
```

Za pravilno izvedbo zgornjega ukaza je potrebno kot argument podati uporabnika, ime tabele in ime datoteke z varnostno kopijo podatkovne baze.

4.5.4 Ukrepanje v primeru napak

Kljub ročnemu pregledu kode pred postavitvjo s strani razvijalcev in avtomatiziranemu postopku postavitve aplikacije lahko pride do napak, ki imajo za posledico lahko tudi izpad delovanja storitve. V tem primeru si želimo imeti na voljo orodja, ki nam olajšajo in pospešijo postopek odkrivanja in razreševanja napak.

Najbolj enostaven pristop za odpravljanje napak v izvorni kodi je prehod na prejšnjo, delujočo verzijo. Program Capistrano nam to privzeto omogoča

z ukazom `deploy:rollback`. Na strežniku se v direktoriju *releases* hrani zadnjih pet izdaj izvorne kode aplikacije, kar omogoča enostavno spreminjanje trenutne verzije aplikacije zgolj s spremembo ciljne datoteke simbolne povezave iz direktorija *current*, ki ga kot vir izvorne kode uporabljajo vsi deli sistema.

Razreševanje napak zaradi podatkovne baze pa je bolj zahtevno. Težava je v tem, da vse izvedene podatkovne migracije niso obrnljive. Slednje pomeni, da po izvedbi takih migracij podatkovne sheme ne moremo enostavno povrniti v prejšnje stanje. Za prehod na prejšnjo verzijo podatkovne sheme moramo zato uporabiti varnostno kopijo podatkovne baze.

Binarne datoteke z varnostnimi kopijami podatkovne baze hranimo v direktoriju

```
/home/#{fetch(:deploy_user)}/database_backups/#{backup_name},
```

kjer spremenljivka *backup_name* predstavlja uro in datum, ko je bila varnostna kopija ustvarjena. Če razvijalec želi obnoviti prejšnje stanje podatkovne baze, mora izvesti ukaz

```
cap production deploy:rollback,
```

ki povzroči prikaz implementiranega dialoga za izbiro tipa obnovitve. V kolikor izbere obnovitev izvorne kode s podatkovno bazo, se izvede zasebna metoda *restore_database*. Najprej se na ciljnem strežniku zajame izhod programa `ls` s sledečimi argumenti:

```
ls -a /home/#{fetch(:deploy_user)}/database_backups |
 cat | tail -n +3.
```

Rezultat zgornjega klica je izpis seznama ustvarjenih varnostnih kopij podatkovne baze. Primer izpisa vidimo spodaj:

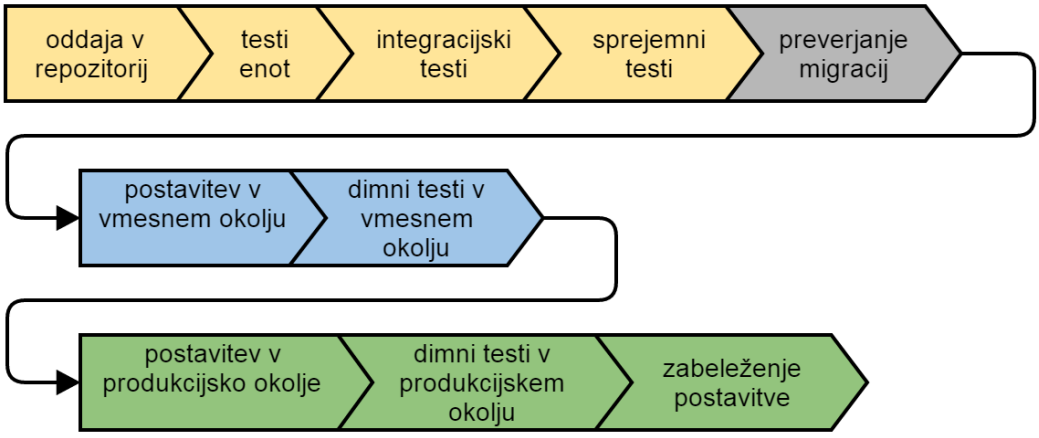
```
==========Listing database backups==========
2015-05-09@20:30
2015-05-09@20:35
```

```
2015-05-09@20:39
2015-05-09@21:05
2015-05-09@22:26
2015-05-09@22:40
production_data
==========================================
Please enter backup (name).
```

Po izbiri imena datoteke je potrebno ustaviti aplikacijski strežnik in delovne procese in na novo ustvariti podatkovno bazo. Stanje podatkovne baze nato obnovi orodje `pg_restore` na enak način, kot je opisano v poglavju 4.5.3.

4.6 Storitev za zvezno integracijo

Naša aplikacija za avtomatizirano testiranje in postavitev uporablja komercialno storitev za zvezno integracijo Codeship, ki omogoča pripravo lastnega strežnika v oblaku in izbor nabora ukazov za testiranje in postavitev aplikacije. Po avtorizaciji preko storitve Github je za delovanje storitve Codeship potrebno nastaviti tri za integracijo pomembne gradnike: okolje, testne ukaze in postavitev. Celoten dostavni cevovod je prikazan na sliki 4.1.



Slika 4.1: Implementiran dostavni cevovod.

4.6.1 Priprava okolja

Ko storitev Codeship zazna uveljavitev spremembe v repozitoriju izvorne kode, najprej izvede ukaze za pripravo sistemskega okolja. V tem sklopu je najprej potrebno izbrati verzijo programskega jezika Ruby z ukazom

`rvm use 2.1.2,`

in namestiti vse zunanje Ruby knjižnice z ukazom

`bundle install.`

Sledi ponastavitev podatkovne baze in izvedba podatkovnih migracij za testno okolje z ukazom

`bundle exec rake db:drop db:create db:migrate db:test:prepare,`

kar zagotovi, da v okolju ni zapisov prejšnjih testov. Zadnji korak pri pripravi okolja je prenos glavne veje repozitorija izvorne kode na trdi disk sistema, kar omogoči primerjavo s trenutno vejo (v kateri se nahaja na novo dodana koda) in zaznavanje morebitnih nevarnih migracij v nadaljevanju. Za prenos ustrezne veje uporabimo ukaz

```
git clone --branch 'master' --depth 50
git@github.com:uncoverd/promet.git master.
```

4.6.2 Testni ukazi

Testiranje izvorne kode aplikacije poteka v dveh delih. Najprej z ukazom

```
bundle exec rspec --pattern
"**/{models,controllers,services,workers}/*_spec.rb"
```

izvedemo vse teste enot in integracijske teste. Uporaba vzorca nam omogoča izbiro testov v direktorijih z modeli, kontrolerji, storitvami in delovnimi procesi. V kolikor pri izvedbi tega ukaza ne pride do napak, z ukazom

```
bundle exec rspec --pattern "**/views/*_spec.rb"
```

izvedemo še sprejemne teste. Skupno se oba sklopa testov izvajata približno 30 sekund. Nato izvedemo še preverjanje, ali na novo dodana koda vsebuje nevarne podatkovne migracije, in sicer z uporabo `rake` opravila

```
bundle exec rake migration:check.
```

Rezultati testiranja se nato z uporabo spletnega povratnega klica storitve Github objavijo na spletni strani zahteve za pregled, kar razvijalcem omogoča hiter vpogled v pravilnost na novo dodane kode.

4.6.3 Postavitev

Če se po izvedbi testov razvijalci odločijo za združitev na novo dodane kode v glavno vejo repozitorija, se sproži postopek za postavitev aplikacije. Najprej se izvede postavitev in izvedba dimnih testov v vmesnem okolju s parom ukazov

```
cap staging deploy
TARGET_ADDRESS=https://188.166.95.64
bundle exec rspec --pattern "**/{smoke}/*_spec.rb.
```

V kolikor med postavitvijo in izvajanjem dimnih testov ne pride do napak, se postopek postavitve ponovi še za produkcijsko okolje z ukazoma

```
cap production deploy
TARGET_ADDRESS=https://promet.me
bundle exec rspec --pattern "**/{smoke}/*_spec.rb.
```

Na koncu mora Codeship poskrbeti še za beleženje postavitve s pomočjo storitve New Relic. To nam omogoča lažje spremljanje sprememb v zmogljivosti naše aplikacije. V ta namen se zato izvede še ukaz

```
RAILS_ENV=production newrelic deployments.
```

V celotnem postopku integracije preko storitve Codeship največ časa (približno eno minuto) traja postavitev aplikacije v vmesno ali produkcijsko okolje. Časovno zahtevno je tudi izvajanje testov enot in integracijskih testov (skupaj približno 30 sekund) ter izvajanje dimnih testov (približno 11 sekund). Celoten nabor ukazov, prikazanih v prejšnjih treh razdelkih, se izvede v približno treh minutah in pol.

Poglavje 5

Primer uporabe

Razvijalci morajo za implementacijo novih funckionalnosti slediti delovnemu toku Github Flow, ki je bil predstavljen v poglavju 2.4.3.

Ustvarjanje ločene razvojne veje

Prvi korak pri razvoju je ustvarjanje nove veje repozitorija z ukazom

`git checkout -b feature_ime_funckionalnosti.`

Za lažji pregled nad večjim številom vej imajo ločene veje predpono *feature* ali *bugfix*, glede na to ali gre za implementacijo nove funkcionalnosti ali zgolj za odpravljanje programskih hroščev prejšnje verzije aplikacije.

Uveljavitev sprememb v ločeni veji

Implementacija novih funkcionalnosti se v vejo repozitorija doda z uveljavitvijo sprememb, ki predstavljajo manjše zaključene enote dela. Datoteke dodamo na seznam s spremembami za uveljavitev z uporabo ukaza `git add`. Nova uveljavljena sprememba s sporočilom se ustvari z ukazom

`git commit -m "opis spremembe".`

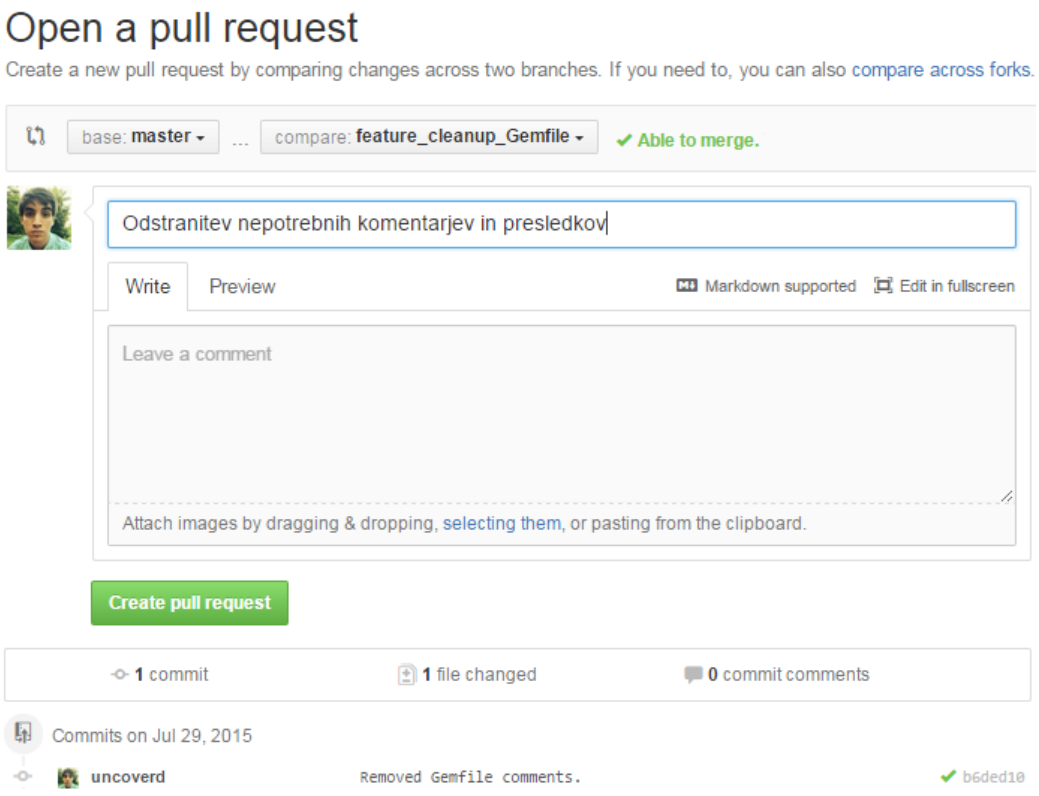
Oddajanje kode v oddaljeni repozitorij

Ko je razvijalec kodo pripravljen deliti z drugimi, mora lokalno vejo razvoja najprej dodati v oddaljeni repozitorij z ukazom

`git push -u origin feature_ime_funkcionalnosti.`

Zahteva za pregled

Po tem, ko je koda uspešno prenešena v oddaljeni repozitorij, lahko razvijalec na spletni strani Github repozitorija ustvari zahtevo za pregled, kot prikazuje zaslonski posnetek na sliki 5.1.



Slika 5.1: Ustvarjanje zahteve za pregled.

Komentiranje zahteve za pregled

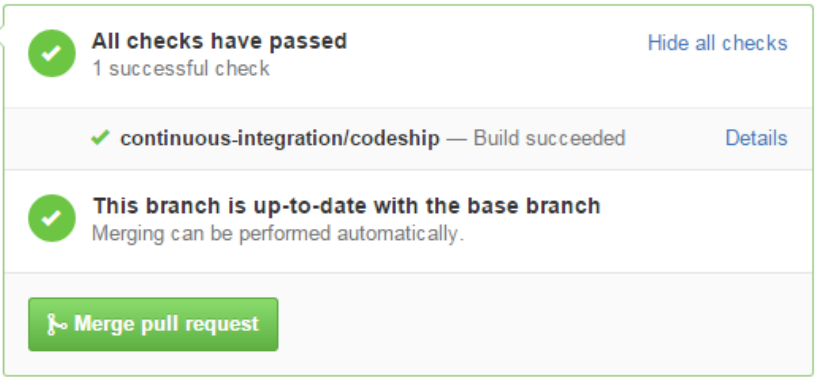
Ostali razvijalci lahko komentirajo tako celotno implementacijo nove funkcionalnosti kot tudi posamezne izseke izvorne kode. V komentarjih se lahko sklicujejo na obstoječe zahteve za pregled, zaznamke in člane razvojne skupine. Primer dialoga za vnos komentarjev prikazuje zaslonski posnetek na sliki 5.2.

Slika 5.2: Komentiranje zahteve za pregled.

Rezultati testiranja

Na spletni strani Github lahko vidimo tudi rezultate testiranja, kot prikazuje posnetek zaslona na sliki 5.3. Testi se izvedejo ob vsaki novi spremembi v veji, rezultate pa lahko razvijalec sproti spremlja na spletni strani. Poleg rezultatov izvedbe testov na integracijskem strežniku je na tem mestu možen še prikaz rezultatov zunanjih storitev za statično analizo izvorne kode.

Ročni pregled in združevanje v glavno vejo



Slika 5.3: Rezultati integracijskega strežnika.

Ko razvijalec zaključi z delom na novi funkcionalnosti, je potreben še ročni pregled pravilnosti kode, ki ga opravijo ostali člani razvojne skupine. V primeru, da ne odkrijejo nobenih napak in se vsi testi uspešno izvedejo, se lahko odločijo za uveljavitev sprememb v glavni veji. S pritiskom na gumb *"Merge pull request"* član skupine združi ločeno vejo iz zahteve za pregled z glavno vejo. Združitev vej zazna strežnik za zvezno integracijo in sproži postopek postavitve aplikacije v produkcijsko okolje, ki je opisan v poglavju 4.6.

Poglavje 6

Sklep

V sklopu implementacije zvezne postavitve na primeru obstoječe aplikacije za obveščanje o prometnih dogodkih nam je uspelo doseči vse zastavljene cilje. Trajanje postopka postavitve strežnikov in razvojnega okolja nam je uspelo zmanjšati iz pol ure na deset minut, oboje pa od razvijalca namesto vnosa množice ukazov zahteva zgolj vnos enega ukaza v ukazno vrstico.

Dosegli smo, da se postavitev nove verzije aplikacije izvede avtomatsko, ob vsaki spremembi izvorne kode v glavni veji repozitorija, in od razvijalca ne zahteva več izvajanja ukazov na oddaljenem računalniku. Opazili smo, da je pogosto izvajanje postavitve pohitrilo dostavo novih funkcionalnosti končnim uporabnikom naše spletne aplikacije.

Uvedba vmesnega okolja in dimnih testov je bistveno zmanjšala število napak v produkcijskem okolju, saj smo napake v veliki meri zaznali še pred dostavo funkcionalnosti končnim uporabnikom.

Implementacija postopka zvezne postavitve je torej za nas imela številne pozitivne učinke. Kljub temu pa se zavedamo številnih možnosti za nadaljnje delo in izboljšave. Na tem mestu želimo izpostaviti predvsem nadaljnje izboljšanje postopka postavitve infrastrukture. Implementirani postopek za postavitev strežnika namreč kljub avtomatizaciji traja okrog deset minut. Pohitritev bi lahko dosegli z uporabo predhodno pripravljene slike sistema (angl. *system image*), ki bi trajanje postopka zmanjšala na le nekaj minut.

Dodatna prednost uporabe pripravljenih slik sistema vidimo v zmanjšanju števila napak pri postavitvi - te bi se pojavljale le še zaradi neustrezne nastavitve strežnika. Uporaba nespremenljive infrastrukture (angl. *immutable infrastructure*) namreč za nadgradnjo programske opreme strežnikov ne bi več zahtevala izvajanja ukazov na strežnikih, ampak zgolj zamenjavo slike sistema.

Literatura

[1] Abdul, F. A. and Fhang, M. C. S. "Implementing Continuous Integration towards rapid application development", v zborniku: Innovation Management and Technology Research (ICIMTR), 2012 International Conference, 2009, str. 369–374.

[2] Neely, S. and Stolt, S. "Continuous Delivery? Easy! Just Change Everything (Well, Maybe It Is Not That Easy)", v zborniku: Agile Conference (AGILE), 2013, 2013, str. 121–128.

[3] Olsson, H. H. and Alahyari, H. and Bosch, J. "Climbing the Stairway to Heaven ; – A Mulitiple-Case Study Exploring Barriers in the Transition from Agile Development towards Continuous Deployment of Software", v zborniku: Software Engineering and Advanced Applications (SEAA), 2012 38th EUROMICRO Conference, 2012, str. 392–399.

[4] Facebook Release Process. [Online]. Dosegljivo: http://www.infoq.com/presentations/Facebook-Release-Process. [Dostopano 7. 5. 2015].

[5] Continuous delivery at Google. [Online]. Dosegljivo: https://air.mozilla.org/continuous-delivery-at-google/. [Dostopano 18. 4. 2015].

[6] Deploying Netflix api. [Online]. Dosegljivo: http://techblog.netflix.com/2013/08/deploying-netflix-api.html. [Dostopano 22. 3. 2015].

[7] Velocity Culture. [Online]. Dosegljivo: http://assets.en.oreilly.com/1/event/60/Velocity%20Culture%20Presentation.pdf. [Dostopano 10. 5. 2015].

[8] Jez Humble and David Farley, *Continuous Integration: Improving Software Quality and Reducing Risk*. Addison-Wesley Professional, 2007, pogl. 2, pogl. 3

[9] Lianping Chen, "Continuous Delivery: Huge Benefits, but Challenges Too", *Software, IEEE*, št. 2, zv. 32, str. 50–54, 2015.

[10] Jez Humble and David Farley, *Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation*. Addison-Wesley Professional, 2010, pogl. 3, pogl. 5.

[11] DeploymentPipeline. [Online]. Dosegljivo: http://martinfowler.com/bliki/DeploymentPipeline.html. [Dostopano 24. 1. 2015].

[12] Feature Toggle. [Online]. Dosegljivo: http://martinfowler.com/bliki/FeatureToggle.html. [Dostopano 3. 2. 2015].

[13] Git fast version control. [Online]. Dosegljivo: https://git-scm.com. [Dostopano 7. 5. 2015].

[14] Eclipse Community Survey 2014 results. [Online]. Dosegljivo: https://ianskerrett.wordpress.com/2014/06/23/eclipse-community-survey-2014-results/. [Dostopano 5. 5. 2015].

[15] GitHub. [Online]. Dosegljivo: https://github.com. [Dostopano 7. 5. 2015].

[16] Understanding the GitHub flow. [Online]. Dosegljivo: https://guides.github.com/introduction/flow/. [Dostopano 7. 5. 2015].

[17] All about Chef. [Online]. Dosegljivo: http://docs.chef.io. [Dostopano 7. 5. 2015].

[18] Capistrano. [Online]. Dosegljivo: http://capistranorb.com. [Dostopano 7. 5. 2015].

[19] Amazon S3. [Online]. Dosegljivo: https://aws.amazon.com/s3/. [Dostopano 7. 5. 2015].

[20] DigitalOcean technology. [Online]. Dosegljivo: https://www.digitalocean.com/features/technology/. [Dostopano 7. 5. 2015].

[21] Redis download. [Online]. Dosegljivo: http://redis.io/download. [Dostopano 10. 4. 2015].

[22] Installing Varnish 3.0.5 on Cygwin Windows. [Online]. Dosegljivo: https://www.varnish-cache.org/trac/wiki/VarnishOnCygwinWindows. [Dostopano 4. 4. 2015].

[23] McConnell Steve, "Daily build and smoke test", *IEEE Software*, št. 5, zv. 13, str. 144–144, 1996.

[24] Adds support for concurrent indexing in PostgreSQL adapter. [Online]. Dosegljivo: https://github.com/rails/rails/pull/9923. [Dostopano 3. 2. 2015].

[25] How to Create Postgres Indexes Concurrently in ActiveRecord Migrations. [Online]. Dosegljivo: https://robots.thoughtbot.com/how-to-create-postgres-indexes-concurrently-in. [Dostopano 5. 4. 2015].