Sustav za kontinuirani razvoj aplikacija

Šokčević, Josip

Master's thesis / Diplomski rad

2018

Degree Grantor / Ustanova koja je dodijelila akademski / stručni stupanj: **Josip Juraj Strossmayer University of Osijek, Faculty of Electrical Engineering, Computer Science and Information Technology Osijek / Sveučilište Josipa Jurja Strossmayera u Osijeku, Fakultet elektrotehnike, računarstva i informacijskih tehnologija Osijek**

Permanent link / Trajna poveznica: https://urn.nsk.hr/urn:nbn:hr:200:457224

Rights / Prava: In copyright / Zaštićeno autorskim pravom.

Download date / Datum preuzimanja: **2024-04-25**

Repository / Repozitorij:

Faculty of Electrical Engineering, Computer Science and Information Technology Osijek

SVEUČILIŠTE U OSIJEKU
FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA OSIJEK

Sustav za kontinuirani razvoj aplikacija

Josip Šokčević

Osijek, srpanj 2018.

FERIT
FAKULTET ELEKTROTEHNIKE, RAČUNARSTVA I INFORMACIJSKIH TEHNOLOGIJA OSIJEK

Obrazac D1: Obrazac za imenovanje Povjerenstva za obranu diplomskog rada

Osijek, 28.05.2018.

Odboru za završne i diplomske ispite

Imenovanje Povjerenstva za obranu diplomskog rada

Ime i prezime studenta:	Josip Šokčević
Studij, smjer:	Diplomski sveučilišni studij Računarstvo, smjer Procesno računarstvo
Mat. br. studenta, godina upisa:	D-589R, 17.10.2017.
OIB studenta:	04466103327
Mentor:	Doc.dr.sc. Josip Balen
Sumentor:	
Sumentor iz tvrtke:	
Predsjednik Povjerenstva:	Doc.dr.sc. Alfonzo Baumgartner
Član Povjerenstva:	Dr.sc. Bruno Zorić
Naslov diplomskog rada:	Sustav za kontinuirani razvoj aplikacija
Znanstvena grana rada:	**Programsko inženjerstvo (zn. polje računarstvo)**
Zadatak diplomskog rada:	U teorijskom dijelu rada potrebno je opisati tehnologije za razvoj sustava za kontinuirani razvoj aplikacija poput Jenkins CI, Git, itd. U praktičnom dijelu rada potrebno je dizajnirati i implementirati sustav za kontinuirani razvoj aplikacija s visokom raspoloživošću. Sustav treba pokrenuti proces testiranja jedinice (engl. unit test), kompajliranje projekta te testiranja ispravnosti preko funkcionalnih testova. Ako svi testovi prođu, takav sustav počinje posluživati korisnike.
Prijedlog ocjene pismenog dijela ispita (diplomskog rada):	Izvrstan (5)
Kratko obrazloženje ocjene prema Kriterijima za ocjenjivanje završnih i diplomskih radova:	Primjena znanja stečenih na fakultetu: 3 bod/bodaPostignuti rezultati u odnosu na složenost zadatka: 3 bod/bodaJasnoća pismenog izražavanja: 3 bod/bodaRazina samostalnosti: 3 razina
Datum prijedloga ocjene mentora:	28.05.2018.

Potpis mentora za predaju konačne verzije rada u Studentsku službu pri završetku studija:	Potpis:
	Datum:

FERIT
FAKULTET ELEKTROTEHNIKE, RAČUNARSTVA
I INFORMACIJSKIH TEHNOLOGIJA **OSIJEK**

IZJAVA O ORIGINALNOSTI RADA

Osijek, 16.07.2018.

Ime i prezime studenta:	Josip Šokčević
Studij:	Diplomski sveučilišni studij Računarstvo, smjer Procesno računarstvo
Mat. br. studenta, godina upisa:	D-589R, 17.10.2017.
Ephorus podudaranje [%]:	2

Ovom izjavom izjavljujem da je rad pod nazivom**: Sustav za kontinuirani razvoj aplikacija**

izrađen pod vodstvom mentora Doc.dr.sc. Josip Balen

i sumentora

moj vlastiti rad i prema mom najboljem znanju ne sadrži prethodno objavljene ili neobjavljene pisane materijale drugih osoba, osim onih koji su izričito priznati navođenjem literature i drugih izvora informacija. Izjavljujem da je intelektualni sadržaj navedenog rada proizvod mog vlastitog rada, osim u onom dijelu za koji mi je bila potrebna pomoć mentora, sumentora i drugih osoba, a što je izričito navedeno u radu.

Potpis studenta:

SVEUČILIŠTE JOSIPA JURJA STROSSMAYERA U OSIJEKU
FAKULTET ELEKTROTEHNIKE, RAČUNARSTVA I INFORMACIJSKIH TEHNOLOGIJA OSIJEK

IZJAVA

Ja, Josip Šokčević, OIB: 04466103327, student/ica na studiju: Diplomski sveučilišni studij Računarstvo, smjer Procesno računarstvo, dajem suglasnost Fakultetu elektrotehnike, računarstva i informacijskih tehnologija Osijek da pohrani i javno objavi moj **diplomski rad**:

Sustav za kontinuirani razvoj aplikacija

u javno dostupnom fakultetskom, sveučilišnom i nacionalnom repozitoriju.

Osijek, 16.07.2018.

_____
potpis

Sadržaj

1 UVOD **1**

2 INFRASTRUKTURA KONTINUIRANOG RAZVOJA APLIKACIJA **3**

2.1 Git – sustav kontrole verzija . . . 3

2.1.1 Vrste sustava kontrole verzija . . . 3

2.1.2 Osnove Git-a . . . 5

2.1.3 Korištenje Git-a i GitHub . . . 6

2.1.4 Git grane . . . 8

2.2 Jenkins . . . 9

2.2.1 Jenkins arhitektura . . . 11

2.3 Docker . . . 12

2.3.1 Docker komandni program . . . 14

2.3.2 Docker servis . . . 16

2.4 Nginx . . . 18

2.4.1 Nginx konfiguracija . . . 19

2.5 Go programski jezik . . . 20

2.5.1 Go rutine i kanali . . . 21

2.6 Cypress . . . 22

3 KONTINUIRANI RAZVOJ NA PRIMJERU WEB APLIKACIJE **24**

3.1 Aplikacija za izračun Fibonaccijevog broja . . . 24

3.2 Jenkins posao za izgradnju i objavu Docker slike . . . 27

3.3 Servis za menadžment . . . 29

3.3.1 Arhitektura servisa za menadžment . . . 31

3.3.2 Rad servisa . . . 33

3.4 Proces kontinuiranog razvoja aplikacije . . . 35

3.5 Razvoj nove verzije Fibonacci aplikacije . . . 36

3.6 Tipični problemi s objavom nove verzije aplikacije . . . 38

3.7 Vraćanje na staru aplikaciju . . . 40

4 ZAKLJUČAK **42**

LITERATURA **43**

SAŽETAK **45**

ABSTRACT **45**

ŽIVOTOPIS **46**

1. UVOD

Razvoj aplikacija uvelike se promijenio s razvojem novih tehnologija, a pogotovo u zadnjem desetljeću s razvojem oblaka (*engl. cloud*). Sve je veći broj tvrtki koje se bave razvojem aplikacija te imaju potrebu što prije objaviti njihovu nadogradnje. Stare metode objavljivanja aplikacija traju dugo te vrijeme potrebno za objavu raste s rastom broja inženjera zbog kompleknosti koda i programa. Osim što su skupe, podložne su i ljudskim greškama. Stoga se sve više tvrtki odlučuje za kontinuirani razvoj aplikacija.

Korištenjem kontinuiranog razvoja aplikacija te brzog razvoja (*engl. agile development*) moguće je objavljivati nadogradnje više puta dnevno bez potrebe za inženjerom za objavu aplikacija (*engl. release engineer*) [1]. Takav sustav mora biti u mogućnosti otkriti promjene na sustavu za reviziju koda, pokrenuti testove jedinice, kompajlirati aplikaciju, provesti dodatna testiranja te poslužiti takvu aplikaciju krajnjim korisnicima. Sustav mora biti i u mogućnost pravovremeno vratiti prethodnu verziju aplikacije.

Kontinuirani razvoj aplikacije zahtjeva određeni proces i arhitekturu. Potreban je sustav za reviziju koda te standardiziran tijek rada, odnosno izmjene aplikacije. Bez standardiziranog tijeka, nije moguće u potpunosti automatizirati proces izgradnje nove verzije. Zatim, potreban je i alat za kontunuiranu integraciju koji pokreće određene zadatke nakon svake izmjene aplikacije. Isto tako, potreban je i sustav koji može pokrenuti novu verziju aplikacije ili je napraviti dostupnom, ovisno o tipu iste. U slučaju mrežne aplikacije, potrebno je prethodnu verziju aplikacije ukloniti nakon određenog vremena.

Korporacija Facebook objavila je članak o korištenju kontinuiranog razvoja u kojem nepobitno dokazuju da brzina i kvaliteta koda ne opada s povećanjem broja inženjera razvojnih timova [2]. Broj grešaka (*engl. bugs*) u kodu linearno raste s brojem objavljivanja koda, kao što je i očekivano. Također, neovisno o broju inženjera, u prosjeku se pojavljuje jedna greška srednjeg prioriteta na oko 3000 pojedinačnih objava koda. Facebook tvrdi da je jedan od razloga upravo kontinuirani razvoj aplikacija koji sadrži komponentu testiranja.

U drugom poglavlju opisana je korištena infrastruktura kontinuiranog razvoja aplikacija. Uspoređeni su tipovi sustava za reviziju koda te njihov povijesni razvoj. Prikazane su osnove korištenja i arhitektura izabranog sustava za reviziju koda – Git. Zatim je opisan sustav za kontinuiranu integraciju te osnove alata Jenkins. U istom poglavlju opisan je Docker – alat za upravljanje kontejnerima te je uspoređen s virtualizacijom, Nginx – web server, Go – programski jezik te Cypress – alat za integracijsko testiranje. U trećem poglavlju opisan je proces i izrada jednostavne internet aplikacije u Go programskom jeziku zajedno s testovima jedinice i testovima funkcionalnosti. Opisana je izrada Jenkins posla koji pokreće testove i kompajliranje te izradu i spremanje Docker slike, program za menadžment Docker kontejnera koji je zadužen za pokretanje zadnje verzije aplikacije, korištenje Nginx programa zaduženog za posluživanje HTTP (*engl. Hypertext Transfer*

Protocol) zahtjeva, proces objavljivanja nove verzije aplikacije te proces vraćanja na prethodnu verziju. Četvrto poglavlje zaključuje rad te opisuje moguće daljnje nadrogradnje rada.

2. INFRASTRUKTURA KONTINUIRANOG RAZVOJA APLIKACIJA

Za izradu sustava za kontinuiran razvoj aplikacije potrebna je infrastruktura koja to omogućuje. Postoji velik izbor aplikacija i servisa koji se mogu koristiti u tu svrhu. Na primjer, za verzioniranje koda može se koristiti Git, Subversion, Mercurial, Microsoft TFS, a za kontinuiranu integraciju programskog koda Jenkins, Hudson, TeamCity. U ovom poglavlju opisani su:

- različite vrste kontrole verzije te izabrani sustav Git
- alat za kontinuiranu integraciju Jenkins
- Docker alat za upravljanje kontejnerima te razlika između kontejnera i virtualizacije
- Nginx web servis
- Go programski jezik razvijen od strane Google
- Cypress – alat za *end-to-end* testiranje.

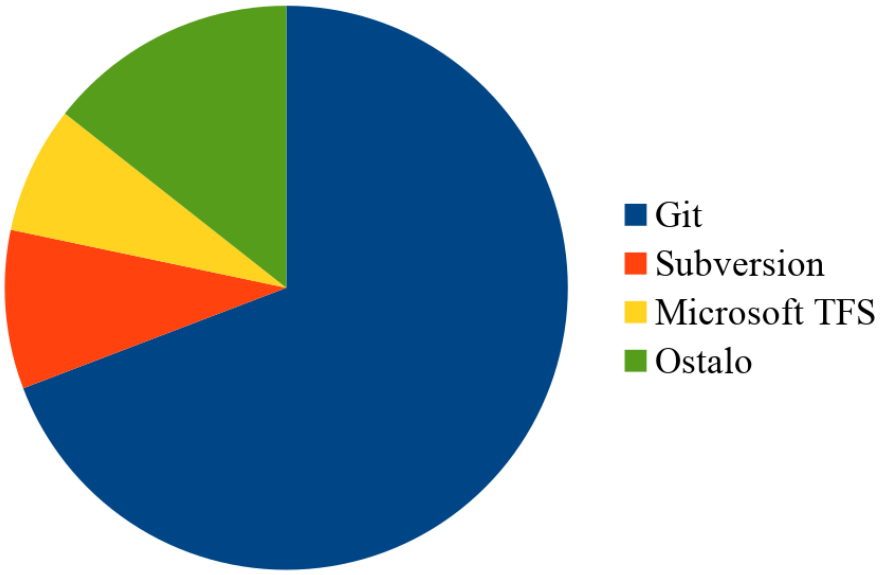
2.1 Git – sustav kontrole verzija

Sustav kontrole verzija je sustav koji sprema promjene datoteka kroz vrijeme. Najčešće se koristi za pohranjivanje programskog koda, no takav sustav može se koristiti i za druge stvari kao što su znanstveni radovi, dokumentacije, dizajni i slično [3]. Prilikom izgradnje programske podrške u pravilu je potrebno imati sustav verzija, pogotovo ako više inženjera radi na istom projektu. Korisnik u bilo kojem trenutku može zapisati trenutno stanje mape (*engl. directory*) u sustav. Na primjer, korisnik može zapisati kada je određena cjelina aplikacija isprogramirana, kada je popravljena greška (*engl. bug*) ili kada su dodani novi testovi jedinica. Korisnik može pregledati povijest u bilo kojem trenutku, vratiti mapu u prijašnje stanje, usporediti izmjene između zapisa. To je samo mali dio operacija koje korisnik može odraditi. Prema anketi Stack Overflow iz 2017, od 30.730 ispitanika, 69,2% se izjasnilo da koristi Git, a prati ga Subversion s 9,1% i Microsoft TFS s 7,3% [4]. Rezultati ankete prikazni su slikom 2.1.

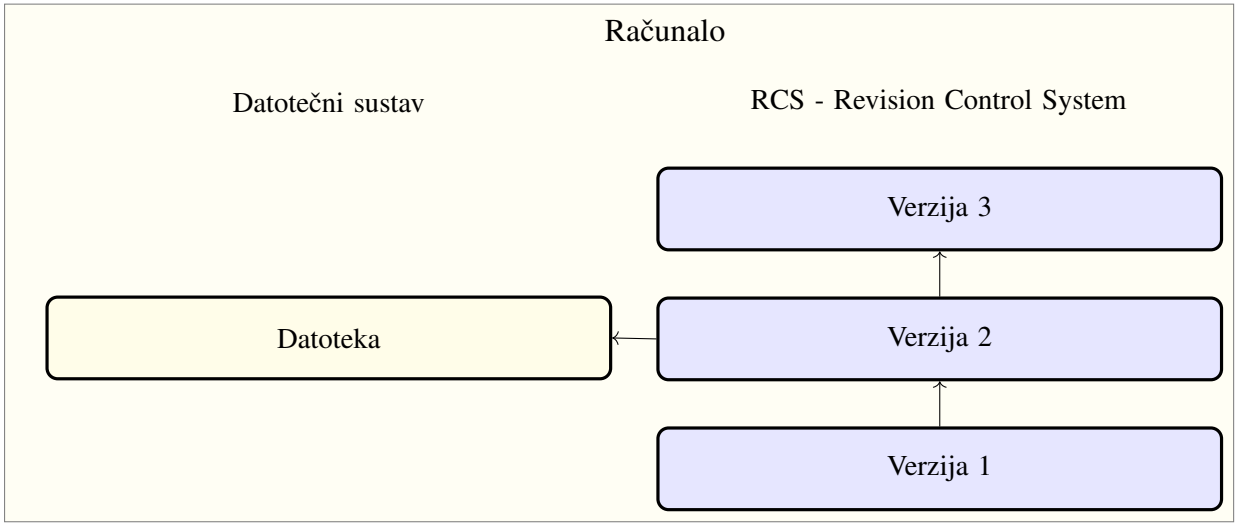
2.1.1. Vrste sustava kontrole verzija

Prije nego što su ovakvi sustavi postali popularni i industrijski standard, korisnici su ručno kopirali mapu ili datoteku te ju spremali kao rezervu (*engl. backup*). Takav proces podložan je greškama te ne pruža puno funkcionalnosti. Jedno od prvih dostupnih rješenja je RCS (*engl. Revision Control System*) [5]. RCS omogućava pohranjivanje razlike datoteka (*engl. patch set*) između revizija. Korisnik može vratiti datoteku u prijašnje stanje tako da sustav primjeni razliku datoteka, kao što je prikazano na slici 2.2.

Problemi se javljaju ukoliko je potreban rad na više računala, to jest ukoliko više inženjera radi na projektu. Kako bi se riješio taj problem, osmišljen je centralizirani sustavi kontrole verzija



Slika 2.1: Rezultati ankete o korištenju kontrole sustava

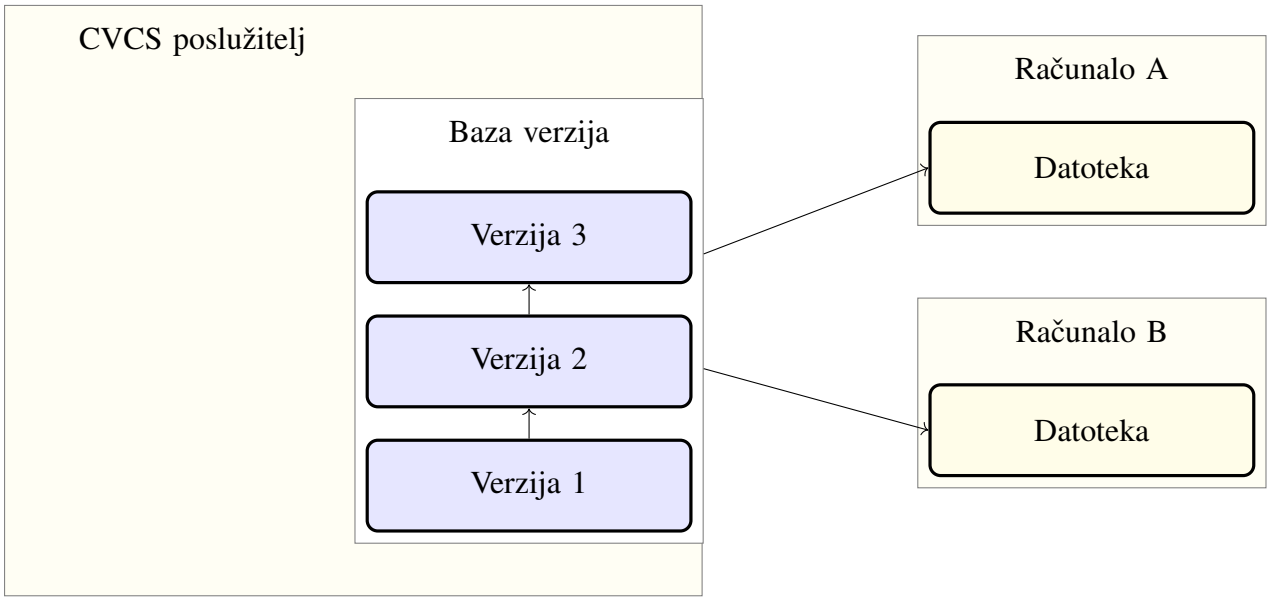


Slika 2.2: Primjer lokalnog sustava kontrole verzija

(CVCS, *engl. Centralized Version Control System*), prikazan na slici 2.3. Takvi sustavi, kao što su CSV, Subversion, Perforce, imaju centralno računalo koji sadržava sve verzije datoteka. Klijenti se spajaju na sustav kako bi kopirali datoteke ili pohranili novu verziju. Administratori u pravilu mogu ograničiti radnje po korisničkom računu.

Centrali sustavi kontrole verzija imaju velike nedostatke. Kako se radi o centralom sustavu, ispadom centralnog računala, sustav postaje nedostupan te korisnici ne mogu raditi. U slučaju gubitka podataka na centralnim računalu, primjerice zbog kvara tvrdog diska (*engl. hard drive*), informacije o svim revizijama su uništene, te jedini podatci koji su dostupni su oni na lokalnim računalima korisnika. Zbog takvih nedostataka osmišljeni su distribuirani sustavi kontrole verzija.

Distribuirani sustav kontrole verzija (DVSC, *engl. Distributed Version Control System*) je sustav u kojemu korisnik dohvaća datoteke, ali i svu povijest. Stoga, ukoliko poslužitelj postane ne-



Slika 2.3: Primjer centraliziranog sustava kontrole verzija

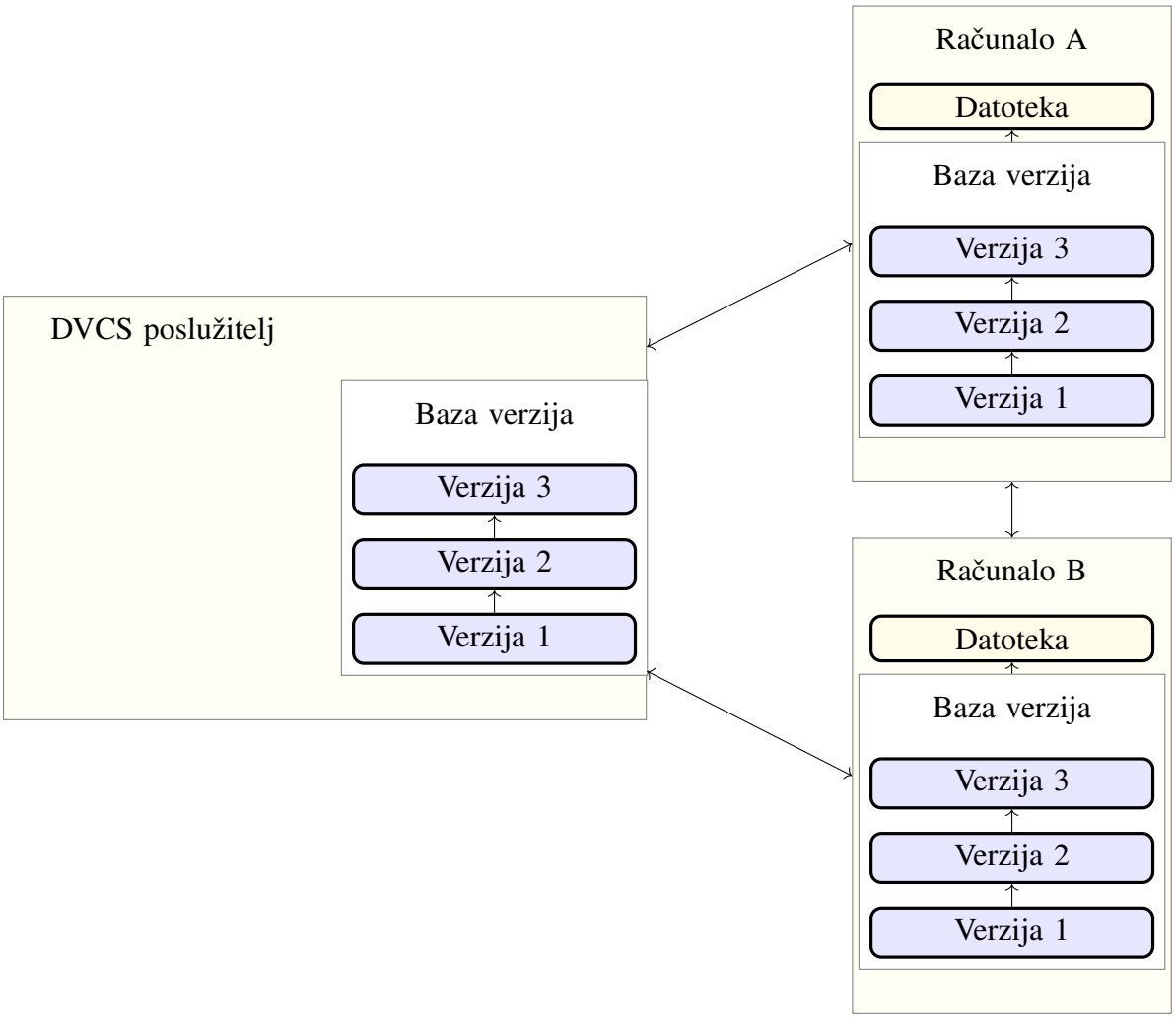
dostupan, korisnik i dalje može koristiti sve funkcije takvog sustava. Isto tako, ukoliko poslužitelj postane trajno nedostupan, korisnik može pokrenuti vlastiti poslužitelj sa podatcima spremljenim na lokalnom računalu bez ikakvih gubitaka. Arhitektura takvog distribuiranog sustava dana je slikom 2.4.

2.1.2. Osnove Git-a

Git je distribuirani sustav otvorenog koda za kontrolu verzije. Za razliku od većine drugih VSC, Git sprema cijele datoteke koje su promijenjene od zadnje verzije. Na primjer, ako korisnik promjeni datoteku tako što doda novu liniju koda, klasični VSC sustavi će spremiti informaciju o toj razlici, dok će Git spremiti cijelu datoteku. Spremanje cijele datoteke zauzima diskovni prostor, no s druge strane omogućuje brže operacije pregledavanja povijesti te sadržaja datoteka u bilo kojem trenutku. Zbog svoje distribuirane naravi Git je dostupan korisnicima i kada računalo nije mrežno povezano. Većina operacija, kao što su `git-commit`, `git-merge`, `git-checkout`, `git-log`, izvršavaju se na lokalnom računalu. Kada je računalo ponovno povezano na mreži korisnik može odabrati dohvaćanje novih datoteka ili pak slanje izmjena.

Git ima tri moguća stanja datoteka: predan (*engl. committed*), izmijenjen (*engl. modified*) i priređen (*engl. staged*). Predane datoteke su one datoteke koje su spremljene u lokalnu bazu, odnosno bazu na lokalnom računalu. Izmijenjene datoteke su one datoteke koje su izmijenjene na lokalnom računalu, no nisu spremljene u lokalnoj Git bazi podataka. Priređene datoteke su izmijenjene datoteke koje se dodaju u sljedećem unosu u Git bazu.

Jedna od značajki Git-a su grane (*engl. branch*) koje omogućuju razvoj novih funkcija programa van glavne grane. Prilikom razvoja, manji dijelovi koda se često spremaju (`tzv. git-commit`) i povremeno šalju na server (tzv. `git-push`). Kada je nova funkcija isprogra-



Slika 2.4: Primjer distribuiranog sustava kontrole verzija

mirana i testirana, takva grana se spaja (tzv. `git-merge`) na glavnu granu. U slučaju sukoba datoteka (*engl. conflict*), korisnik je obaviješten te mora ručno riješiti takav sukob i ponovno testirati aplikaciju.

Korištenje Git uvelike je pojednostavljeno ukoliko se korisnik odluči koristiti servis kao što je GitHub, Gitlab ili Bitbucket. Takvi servisi omogućuju jednostavno stvaranje Git baze te operacije spajanja `git-merge`. Također, većina integriranih sustava za razvoj (IDE, *engl. Integrated Development Environment*) podržava Git operacije, tako da korisnik ne mora koristiti komandnu liniju.

2.1.3. Korištenje Git-a i GitHub

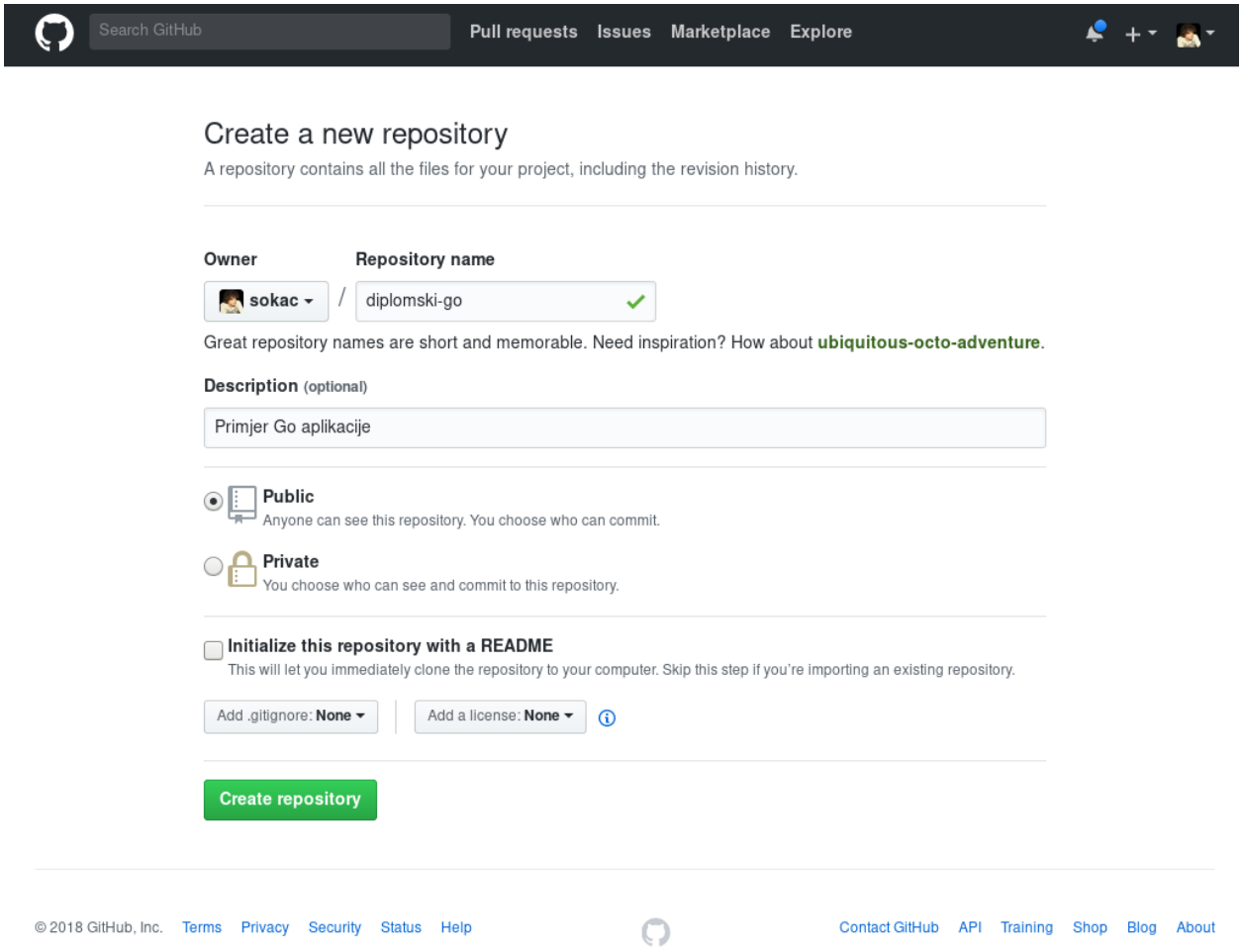
Git operacije se često izvode preko komandne linije. Kreiranje novog Git projekta obavlja se preko komande `git-init`. Git-init stvara praznu Git bazu podataka. Korisnik nakon toga može prirediti datoteke koje će se spremiti u bazu preko operacije `git-add`. Kada je sve spremno za trajno spremanje u Git bazu, pokreće se operacija `git-commit` koja predaje datoteke u lokalnu

Git bazu. Primjer stvaranja Git projekta te spremanja prikazan je programskim kodom 2.1.

```
git init
git add main.go // Pretpostavka da main.go postoji unutar direktorija
git add LICENSE // Pretpostavka da LICENSE postoji unutar direktorija
git commit -m "Prvi commit"
```

Programski kod 2.1: Kreiranje Git projekta i prvo predavanje

Takva baza spremljena je na lokalnom računalu te je samim time nedostupna drugim računalima. Kako bi takva baza postala dostupna drugim računalima, potrebno je poslati bazu na Git poslužitelj. Jedan od najpopularnijih servisa koji pruža Git poslužitelj kao uslugu je GitHub. U travnju 2017. godine GitHub je objavio da je registrirano skoro 20 milijuna korisnika i 57 milijuna Git projekata [6]. Korisnik se mora registrirati na github.com kako bi mogao koristiti usluge. Nakon uspješne registracije, Git projekt se može kreirati kao što je prikazano na slici 2.5.



Slika 2.5: Registracija Git projekta na github.com

Kada je napravljen projekt na GitHub-u, u lokalnu bazu registrira se udaljen Git poslužitelj pomoću `git-remote` operacije te se lokalna baza može poslati na udaljeno računalo pomoću

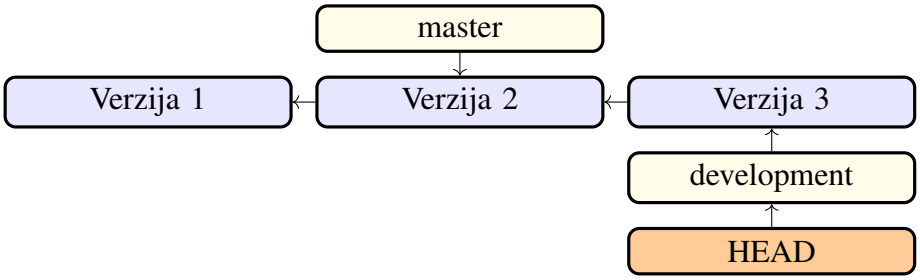
operacije `git-push`. Važno je zapamtiti da operacija `git-commit` ne šalje izmjene na udaljeno računalo, nego isključivo `git-push`. Operacija `git-push` šalje samo razliku između lokalnih predanih datoteka te udaljenih datoteka. Primjer dodavanja udaljenog servera i slanje izmjena dan je programskim kodom 2.2

```
git remote add origin git@github.com:sokac/diplomski-go.git
git push -u origin master
```

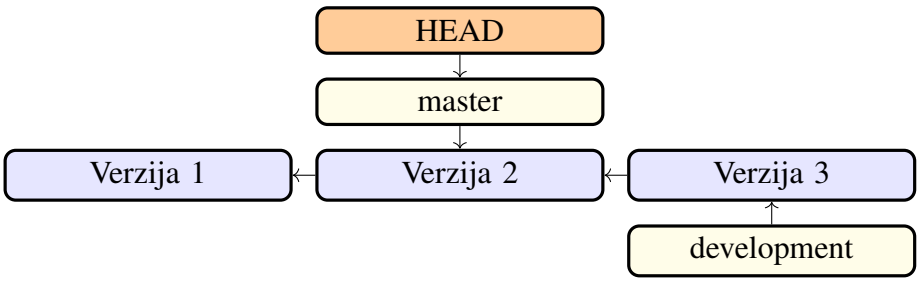
Programski kod 2.2: Dodavanje poslužitelja i slanje izmjena

2.1.4. Git grane

Većina VSC sustava podržava neku vrstu grananja. Grananje omogućava korisniku da izađe iz glavne grane razvoja, često nazvanu *master*, te nastavi razvoj u potpuno drugoj grani. Korisnik u bilo kojem trenutku može izabrati u kojoj grani želi biti. Također, korisnik može spojiti dvije grane u bilo kojem trenutku. Git ima prednost nad drugim VSC sustavima jer su stvaranje i mijenjanje grana izuzetno brze operacije. Samim time, Git potiče na često korištenje grana. Nova grana stvara se operacijom `git-branch` te ima iste datoteke kao i grana prije nego što je operacija stvaranja grane bila pozvana. Operacija `git-checkout` koristi se za prebacivanje u drugu granu. Trenutna grana zapisana je u specijalnoj referenci koja se zove *HEAD*. Na slici 2.6 prikazan je Git projekt s dvije grane, *master* i *development*, gdje grana *development* sadrži više predanih izmjena. Prilikom pokretanja `git checkout master`, *HEAD* pokazivač se pomiče na *master* granu te se datoteke mijenjaju na sadržaj *master* grane, kao što je prikazano na slici 2.7. Ukoliko korisnik preda izmjene unutar grane *master*, tada dolazi do razdvajanja grana, jer grana *master* ima datoteke koje ne postoje u povijesti grane *development* i obrnuto. Git operacija `git-merge` omogućava spajanje grana i u takvom slučaju. Osim standardnih operacija, Github servis nudi uslugu pregleda koda (*engl. code review*). Inženjeri mogu međusobno pregledavati kod prije nego što se spoji s glavnom granom. Također je moguće koristiti i integracijske testove kako bi se osiguralo ispravno funkcioniranje novog koda.



Slika 2.6: Primjer Git grana



Slika 2.7: Primjer pomicanja Git grane

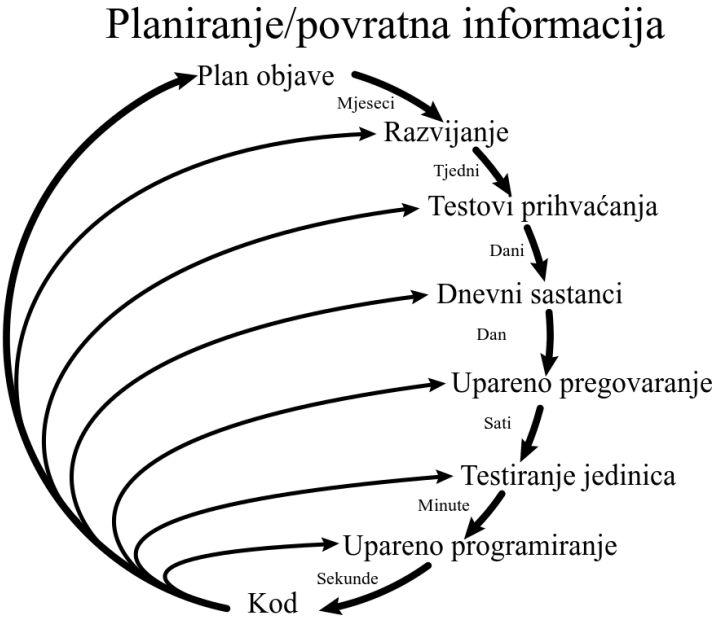
2.2 Jenkins

Timovi koji koriste stare metode, kao što su Grantt dijagrame i model vodopada (*engl. waterfall model*), moraju odvojiti poveći dio vremena za fazu integracije. U toj fazi razvoja računalne podrške inženjeri i timovi spajaju dijelove koda na kojima su radili mjesecima. Prilikom spajanja, često dolazi do sukoba datoteka i programskih sučelja koji inženjeri ručno moraju riješiti. Taj proces zna potrajati tjednima jer je potrebno riješiti sukobe i ponovno testirati program. U takvom je sustavu faza integracija izuzetno stresna za inženjere i voditelje timove. Ta faza je često uzrok odgađanja objavljive nove verzije, a samim time i gubitak povjerenja krajnjih korisnika. Iz tih je razloga nastala potreba za kontinuiranom integracijom.

Kontinuirana integracija (CI, *engl. Continuous Integration*) je način razvoja aplikacije gdje se radne verzije programa spajaju s glavnom verzijom barem jednom dnevno [7]. Ekstremnija verzija takvog načina razvoja nazvana je ekstremno programiranje (XP, *engl. Extreme Programming*), gdje se radna verzija spaja u glavnu verziju barem jednom u dva do tri sata [8], kako je prikazano na slici 2.8. Kako bi to bilo moguće potrebna je dobra infrastruktura koja omogućava brzo otkrivanje grešaka prilikom razvoja. Ona također pomaže prilikom dijagnostike te omogućava brzu izgradnju aplikacije. Svaki bi profesionalni razvojni tim trebao koristiti takav sustav kako bi bio što učinkovitiji i konkurentniji.

Pojednostavljeno, sustav kontinuirane integracije nadgleda sve promjene nad sustavom kontrole verzija. Kada je promjena očitana, sustav pokreće proces testiranja jedinica i izgradnje aplikacije (*engl. compile*). U slučaju greške, sustav obavještava inženjera o pogrešci kako bi se mogla ispraviti što prije. Osim toga, sustav mora biti u mogućnosti proizvesti izvješća o kvaliteti koda, kao što je pokrivenost testiranja jedinice, razne statistike poput vrijeme izgradnje, učestalost grešaka. Ako je sustav kontinuirane integracije pouzdan, može se automatizirati i sustav objavljivanja aplikacija. Ukoliko se takva aplikacija automatski objavljuje i dostavlja krajnjim korisnicima, radi se o sustavu za kontinuirani razvoj i objavu aplikacija (*engl. Continuous Deployment*). Kod takvog sustava iznimno je bitno da je proces nadogradnje aplikacije jednostavan, a poželjno i skriven od krajnjeg korisnika. Stoga se najčešće primjenjuje na mrežnim aplikacijama kao što su internet aplikacije i stranice.

Jenkins, izvorno nazvan Hudson, je alat za kontinuiranu integraciju otvorenog koda napisan

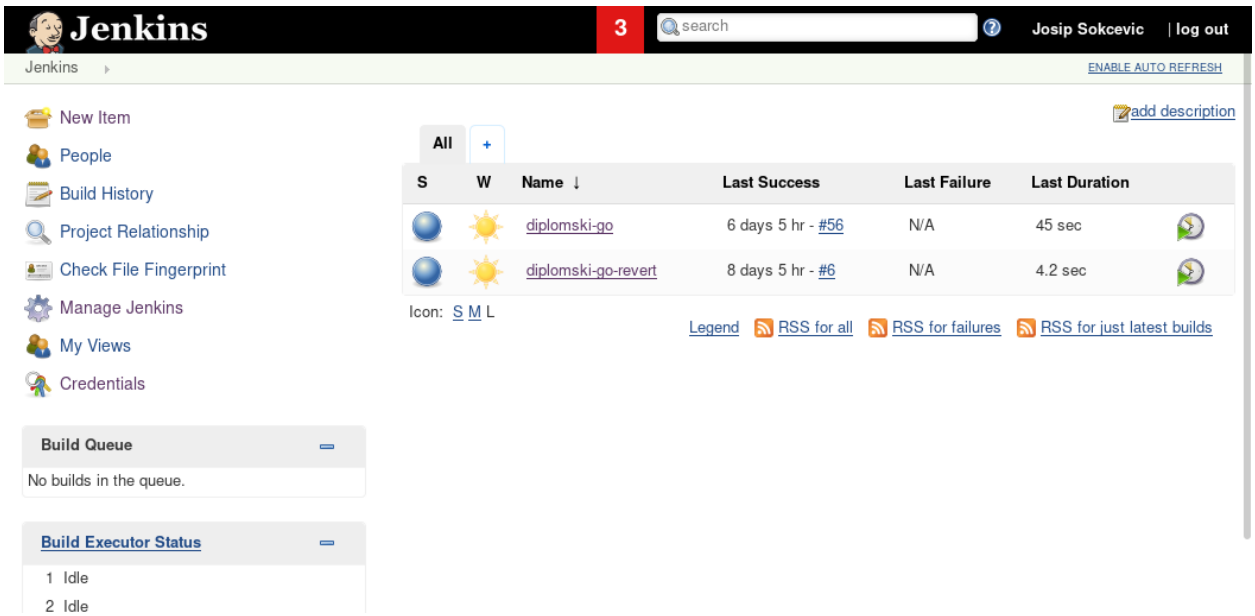


Slika 2.8: Extremno programiranje

u programskom jeziku Java. Postao je iznimno popularan zbog jednostavnosti korištenja, jednostavnog grafičkog sučelja, mogućnosti proširenja preko dodataka (*engl. plugins*) te mnogih drugih značajki. Primjer početne stranice Jenkinsa prikazan je na slici 2.9.

Jenkins alat može se postepeno uvesti u tvrtkama koji nemaju nikakav sustav kontinuirane integracije [9]. Prvo je potrebno uvesti sustav koji će na dnevnoj bazi izgrađivati aplikaciju. Druga faza je uvođenje automatskog testiranja prije izgrađivanja aplikacije, također na dnevnoj razini. Sljedeća faza je uvođenje metrika za kvalitetu koda (pokrivenosti jedinica, brzina testova i izgradnje) te automatska izgradnja dokumentacije. Četvrta faza je dodatno testiranje, poput testiranja funkcionalnosti, a ona se odvija nakon što je aplikacija izgrađena. Peta faza je automatsko objavljivanje aplikacije koja je prošla dodatno testiranje. Takvu aplikaciju mogu koristiti i osobe koje nisu inženjeri, kao što su timovi za testiranje kvalitete. Zadnja faza sustava je kontinuirano objavljivanje aplikacije krajnjim korisnicima. Jenkins se sastoji od:

- poslova (*engl. Jobs*)
- građe (*engl. build*)
- parametara
- linije (*engl. Pipeline*)
- dodataka (*engl. plugins*).

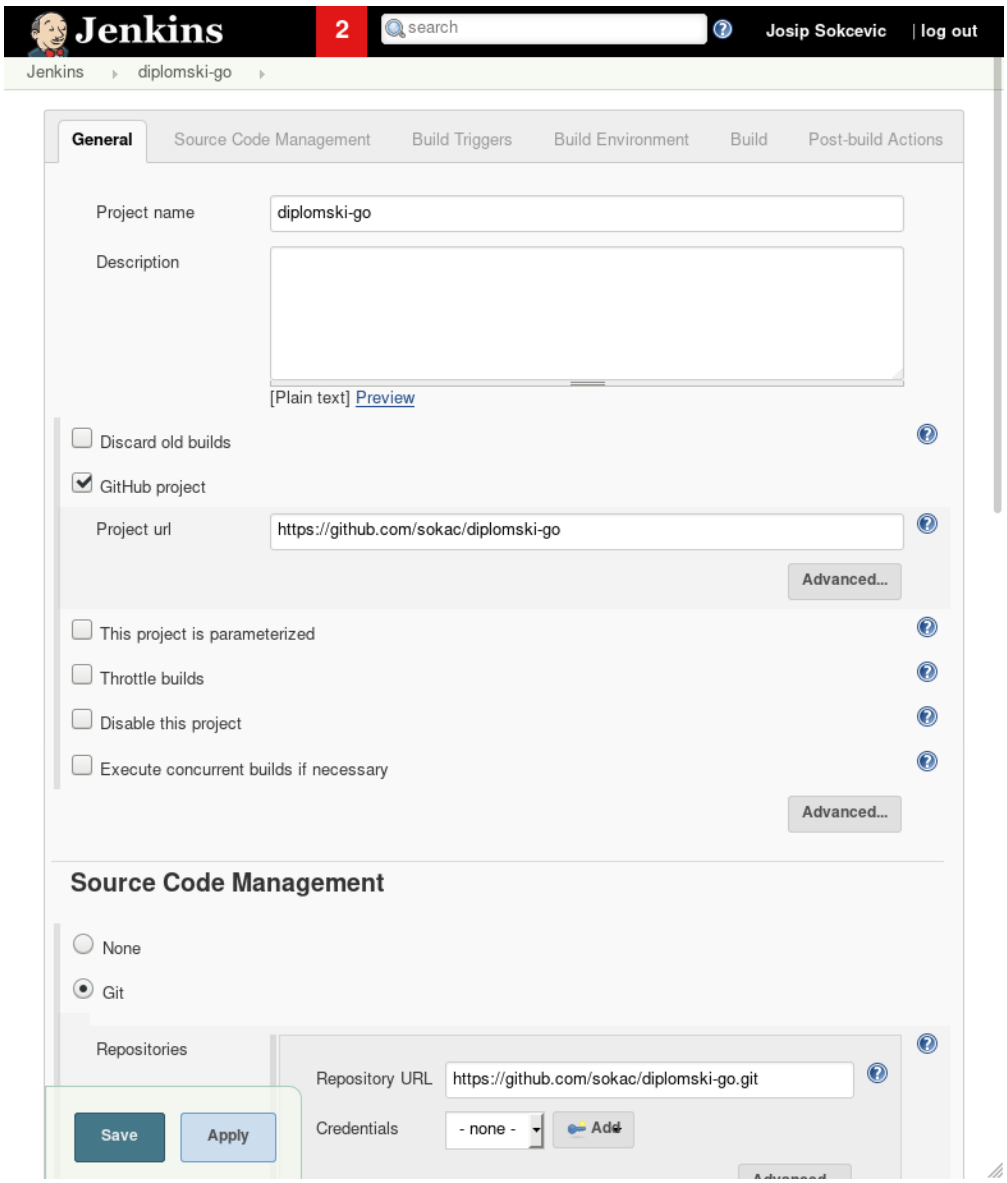


Slika 2.9: Jenkins početna stranica

2.2.1. Jenkins arhitektura

Jenkins posao je skup pravila prema kojemu se izgrađuje aplikacija. Sastoji se od naziva posla, opisa, definiranih parametara potrebnih za pokretanje izgradnje aplikacije, instrukcija za izgradnju aplikacije te listu akcija nakon što je posao završen [10]. Jenkins posao ne ovisi o programskom jeziku aplikacije koja se izgrađuje niti o VCS. Posao se može programirati, ali i izgraditi koristeći grafičko web sučelje. Primjer izgradnje posla preko web sučelja prikazan je slikom 2.10.

Jenkins parametri često se koriste za Jenkins poslove. Jenkins podržava velik broj tipova parametara, kao što su riječi, brojevi, izbornici, jedno-bitni podatci (da/ne odgovor). Parametri se mogu automatski predavati poslu ili ručno. Primjer automatskog predavanja parametara je kada jedan Jenkins posao pokreće drugi te predaje informacije o prethodnom izvršavanju. Primjer ručnog predavanja parametara je kada korisnik odabere verziju koju želi pokrenuti. Većina Jenkins poslova sadrže barem jednu Jenkins građu. Jenkins građa je jedinica posla koja se izvršava kada se Jenkins posao izvršava. Može biti jednostavna Linux Bash ili Windows batch naredba, a preko dodataka može izvršavati i kompleksne stvari kao što su Python skripte, Groovy, preuzimanje Android dodataka, itd. Primjer Jenkins posla s jednom građom prikazan je slikom 2.11. Jedan Jenkins posao može imati više Jenkins građi. Jenkins linija omogućava umrežavanje više Jenkins poslova. Linija može pokrenuti Jenkins poslove sekvencijalno ili paralelno te može sadržavati logiku. Na primjer, Jenkins linija može odlučiti da se linija prekida ukoliko jedan od paralelnih poslova bude neuspješan. Pomoću akcija nakon građe (*engl. Post-Build*) korisnici mogu biti obaviješteni o rezultatima građe. Jedna od velikih prednosti Jenkins alata jest upravo u dodatcima. Dodatci mogu mijenjati vizualno web sučelje, mijenjati način povezivanja Jenkins gazde s Jenkins slugom, dodati podršku za nove programske jezike i VSC. U službenom Jenkins repositoriju registrirano je preko tisuću različitih dodataka [11].

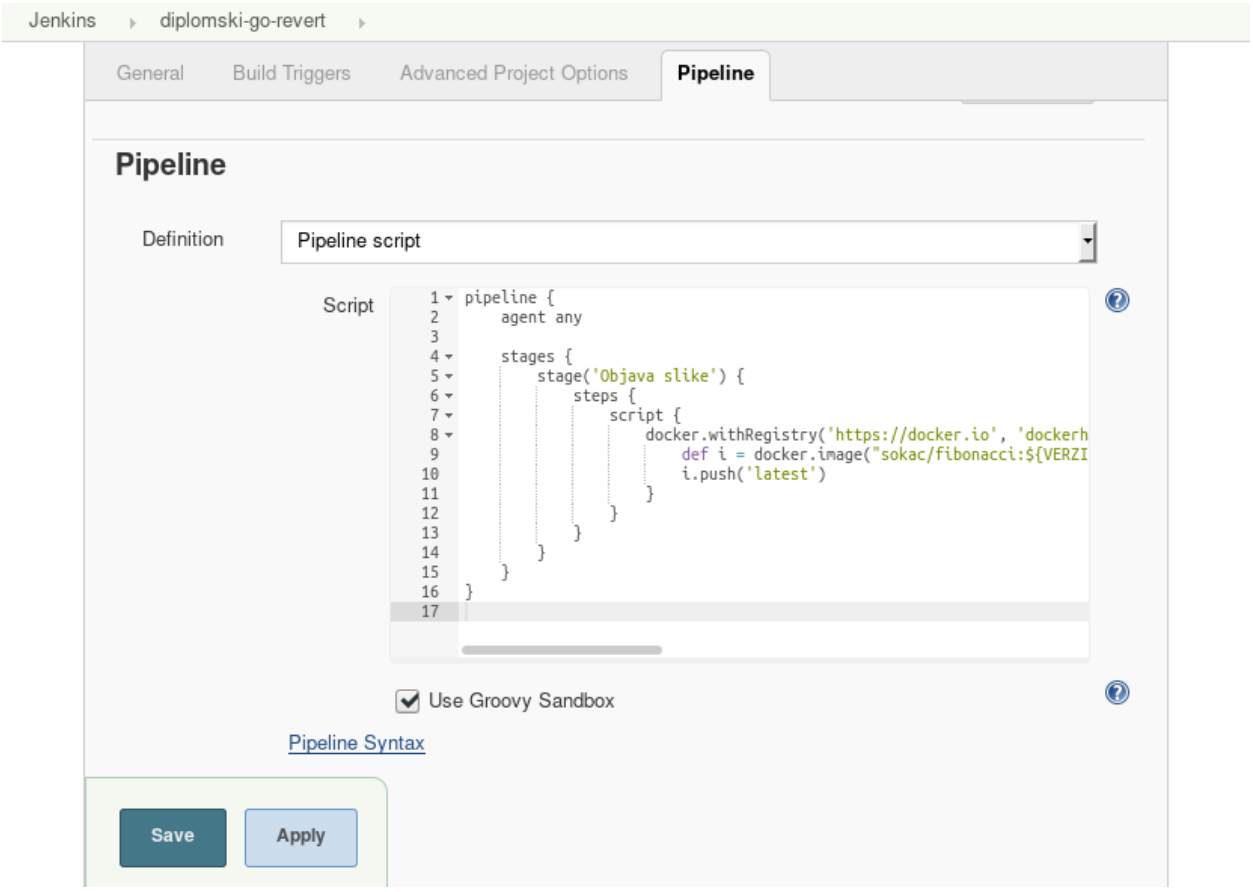


Slika 2.10: Izgradnja Jenkins posla preko grafičkog sučelja

2.3 Docker

Docker je projekt otvorenog koda napisan u Go programskom jeziku koji pojednostavljuje objavljivanje programa tako što koristi programske kontejnere (*engl. Software Containers*). Unutar `Unix` okruženja često se koristio izraz zatvor (*engl. jail*) prilikom izmjene okoline radi sigurnosne izolacije procesa i resursa. S izlaskom Solaris Linux 2005. godine i izmjene u sustavu izolacije počinje se upotrebljavati izraz kontejneri [12]. Po zadanoj vrijednosti (*engl. default value*) kontejneri nemaju pristup nikakvih resursima, što uključuje diskovne, memorijske, sabirničke te mrežne resurse. Korisnik može dozvoliti određene resurse ukoliko je to potrebno. Iako kontejneri postoje već dugi niz godina, korištenje istih nije bilo jednostavno sve do pojave Docker projekta.

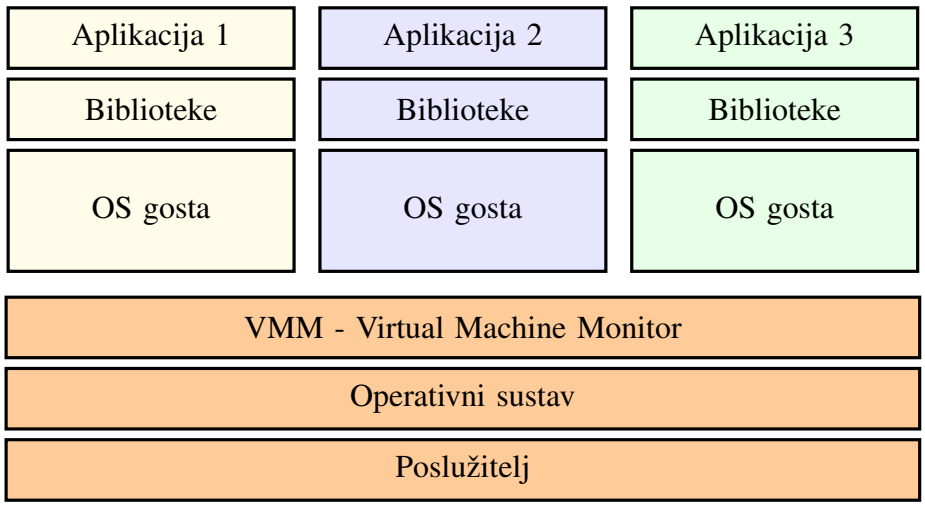
Docker je konceptno sličan virtualizaciji, kao što je VMWare i VirtualBox. No, za razliku od virtualizacije, prednost Dockera je u tome što je brži i koristi manje resursa [13]. Docker koristi



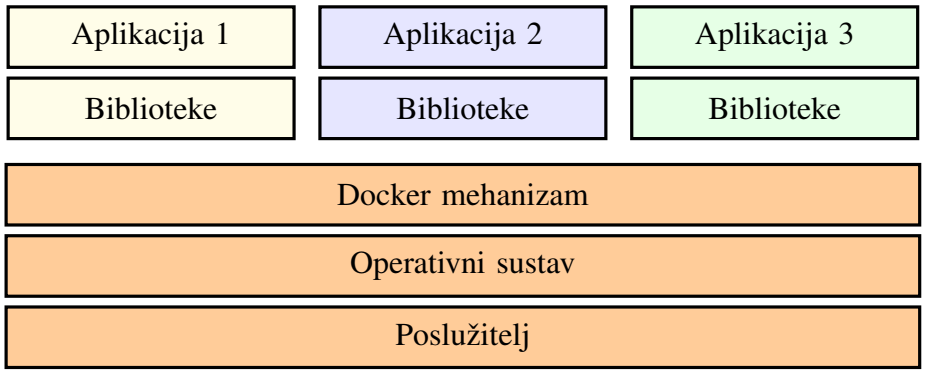
Slika 2.11: Jenkins građa pomoću Pipeline skripte

Linux jezgru glavnog računala. Na slici 2.12 prikazana je arhitektura kod tipične virtualizacije, dok je na slici 2.13 prikazana Docker arhitektura.

Docker slika (*engl. Docker Image*) je preslika datotečnog sustava koja se sastoji od niza Docker slojeva (*engl. Docker layers*). Slojevi predstavljaju promjene u datotečnom sustavu naspram prijašnjeg sloja. Docker kontejner koristi Docker sliku te pokreće određenu aplikaciju koja se već nalazi unutar Docker slike. Promjene unutar Docker kontejnera, kao što je zapis novih datoteka, ne mijenjaju Docker sliku korištenu za pokretanje tog kontejnera. Kada se Docker kontejner zaustavi i obriše, svi podatci su također obrisani. Docker slika pakira program unutar cjelokupnog datotečnog sustava koji ima sve potrebne komponente kako bi se takav program mogao izvoditi - od dinamičkih modula, biblioteka, sve do programskog sučelja operativnoga sustava. Stoga Docker garantira da će izvršavanje takvog sustava bit uvijek jednako, neovisno o računalo na kojemu se izvodi. Slike mogu se spremiti u Docker repositorij koji može biti privatni ili javni. Jedan od najčešće korištenih je službeni repositorij kompanije koja stoji iza Docker-a - Docker, Inc, a koji se nalazi na Web stranici *https://hub.docker.com*. Kako bi se Docker slika pokrenula, potrebno je imati pokrenuti Docker servis (*engl. daemon*) te imati Docker komandni program. Korisnik zatim pokreće Docker komandni program koji potom komunicira s Docker servisom. Docker servis može biti pokrenut na lokalnom ili udaljenom računalu.



Slika 2.12: Arhitektura virtualizacije



Slika 2.13: Docker arhitektura

2.3.1. Docker komandni program

Izgradnja Docker slike pokreće se pomoću naredbe `docker build`. Ta naredba učitava datoteku *Dockerfile* u zadanom direktoriju te šalje naredbu Docker servisu. *Dockerfile* je tekstualni dokument koji sadrži instalacijske instrukcije potrebne za izgradnju Docker slike [14]. Na primjeru danim kodom 2.3 prikazana je instalacija Jenkins alata.

U danom primjeru korištene su četiri naredbe. Naredbna `FROM` definira Docker sliku na kojoj će se nova slika temeljiti. Na primjeru ona se temelji na *openjdk* slici pod oznakom *8-jdk-alpine*. Naredba `RUN` pokreće program koji je definiran iza te naredbne. U danom primjeru pokrenuta je komanda `addgroup` koja stvara Jenkins groupu, naredba `adduser` koja stvara Jenkins korisnika te `curl` koja dohvaća Jenkins alat s udaljenog računala. Naredba `USER` definira pod kojim korisnikom se pokreće Docker kontejner. U danom primjeru riječ je o korisniku *jenkins*. Zadnja korištena naredba `ENTRYPOINT` definira koji će se program pokrenuti unutar Docker kontejnera, a u danom primjer je `java -jar /usr/share/jenkins/jenkins.war`.

Nakon što je Docker slika izgrađena, korisnik može pokrenuti Docker kontejner koristeći novoizgrađenu sliku pomoću naredbe `docker run`. Prilikom pokretanja takve naredbe, korisnik

```
FROM openjdk:8-jdk-alpine

RUN addgroup -g 1000 jenkins
RUN adduser -u 1000 -G jenkins jenkins

RUN curl -fsSL \
   https://repo.jenkins-ci.org/public/org/jenkins-ci/main/jenkins-war/2.117/
      jenkins-war-2.117.war \
   -o /usr/share/jenkins/jenkins.war

USER jenkins

ENTRYPOINT ["java", "-jar", "/usr/share/jenkins/jenkins.war"]
```

Programski kod 2.3: Jenkins Dockerfile

može podesiti mrežne postavke, montirati direktorije, ograničiti procesorske, diskovne i memorijske resurse, izmjeniti sigurnosne postavke te mnoge druge parametre.

Jedan od najčešće korištenih parametera za mrežne aplikacije jeste parametar `-p`. Prilikom pokretanja Docker kontejnera, mreža je podešena u tzv. *mostovnom* načinu rada te samim time pristup Docker kontejneru nije moguć u lokalnoj mreži. Parametar `-p` naređuje Docker sustavu da objavi mrežni port unutar kontejnera na glavnom računalu. Na primjer, parametar `-p 8080:80` preusmjerava mrežne zahtjeve porta 8080 na glavnom računalu prema mrežnom portu 80 unutar Docker kontejnera. Promjena mrežnog načina rada moguća je preko parametra `--network`.

Svi podatci koji su spremljeni unutar kontejnera su trajno obrisani pri brisanju istoga. Montiranje direktorija koristi se ukoliko aplikacija sprema bitne podatke na tvrdi disk te nije dopušteno brisanje. Svi zapisi na montirani direktorij spremaju se na tvrdi disk glavnog računala, a Docker kontejner ima puni pristup njima. Na primjer, parametar `-v /glavno/racunalo:/data` montira direktorij glavnog računala pod nazivom `/glavno/racunalo` na direktorij Docker kontejnera pod nazivom `/data`. Kada Docker kontejner napravi izmjenu unutar `/data` direktorija, takva izmjena bit će automatski dostupna unutar glavnog računala pod direktorijem `/glavno/racunalo`.

Zaustavljanje pokrenutog Docker kontejnera izvršava se pomoću naredbe `docker stop` koji prihvaća listu Docker kontejnera koje se želi ugasiti. Docker servis prvo šalje `SIGTERM` signal aplikaciji. Ukoliko aplikacija nije zaustavljane unutar perioda milosti (*engl. grace period*), tada Docker šalje `SIGKILL` signal. Zaustavljeni Docker kontejner može se ponovno pokrenuti naredbom `docker start`. Naredbom `docker rm` briše se kontejner zajedno sa svim podatcima spremljenima unutar kontejnera.

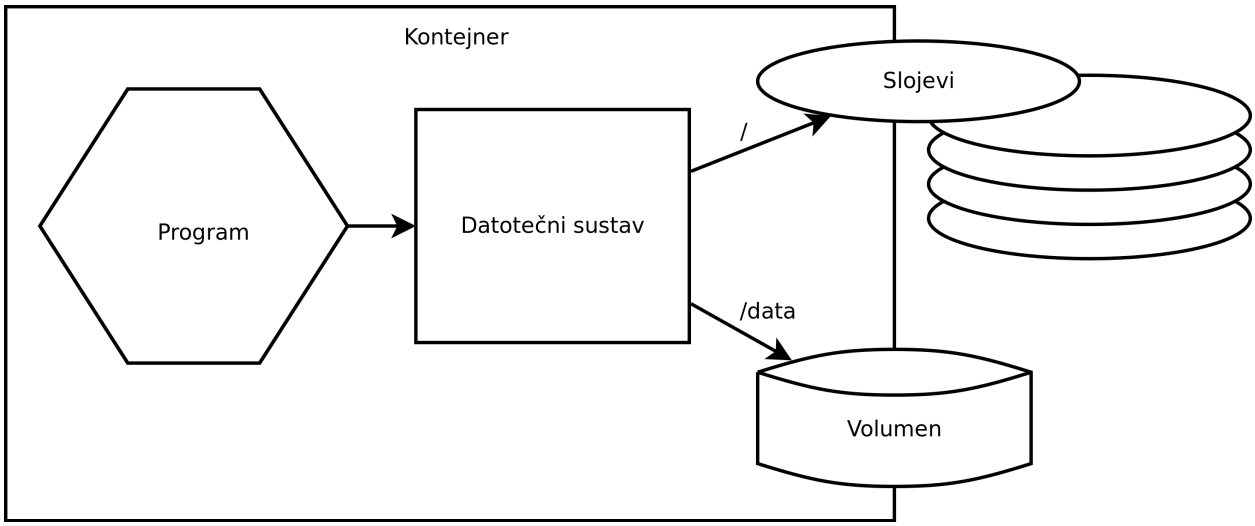
2.3.2. Docker servis

Docker servis sadrži algoritame za izgradnju slika, pokretanje kontejnera i njihovo upravljanje. Docker komandna linija omogućava korisniku jednostavnu komunikaciju s Docker servisom, koja se odvija preko RESTful API koji je dokumentiran i otvoren. Docker također podržava SDK (*engl. Software Development Kit*) za Go i Python programski jezik.

Postoje tri grupe izolacije pristupa pojedinih Docker kontejnera:

- podatkovni pristup
- mrežni pristup
- resursni pristup

Aplikacije koje spremaju podatke, poput baze podataka, moraju trajno spremiti dokumente na tvrdi disk. Ukoliko je Docker kontejner s takvom aplikacijom pokrenut bez podatkovnog pristupa, svi dokumenti spremaju se unutar radnog sloja Docker kontejnera. Docker slojevi spremljeni su na tvrdi disk, a njihov format može izabrati sam korisnik. Neke od opcija su `aufs`, `devicemapper`, `overlay2`, `zfs`. No, ukoliko korisnik želi promjeniti Docker sliku zbog nadogradnje ili promjene postavke, potrebno je pokrenuti novi kontejner. Prebacivanje podataka s jednog kontejnera na drugi nije trivijalno, stoga je preporuka korištenje Docker volumena (*engl. Docker Volume*). Docker volumen može se montirati na bilo koji direktorij, a na slici 2.14 prikazana je montaža na `/data` direktorij.

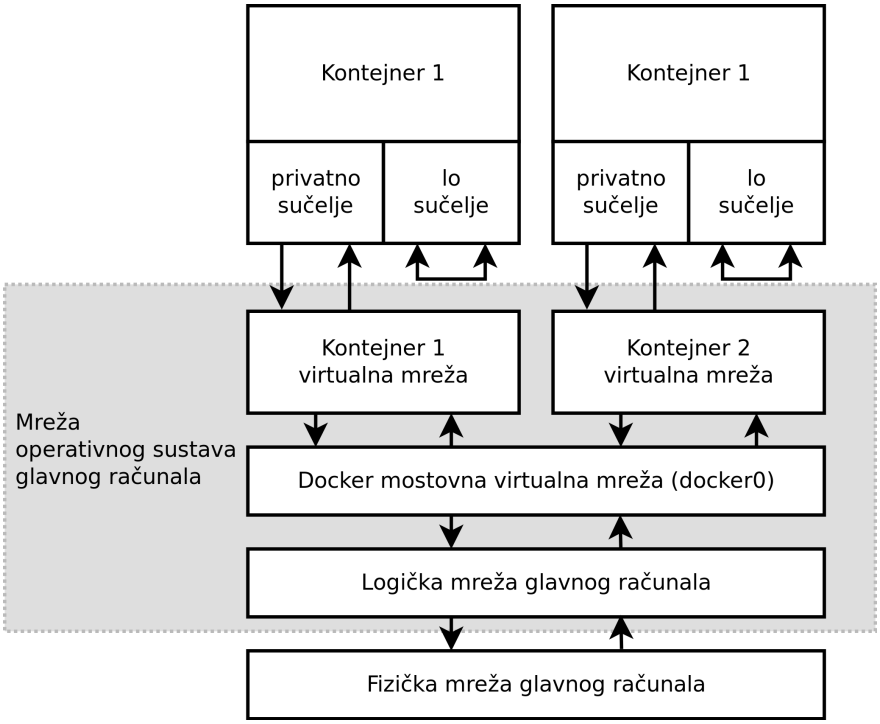


Slika 2.14: Docker volumen montiran na */data*

Postoje dvije vrste volumena: volumen vezne montaže (*engl. Bind mount volume*) i Docker upravljani volumen (*engl. Docker-managed volume*). Oba tipa volumena zahtjevaju montažni direktorij (*engl. mount point*) unutar Docker kontejnera, no razlika je u tome gdje se podatci spremaju. Volumen vezne montaže je vrsta volumena koja je direktno povezana s direktorijem na glavnom računalu. Takvi volumeni korisni su ako glavno računalo mora imati direktan pristup podatcima, u slučaju ako korisnik koristi Docker kontejner za pokretanje web poslužitelja i želi

biti u mogućnosti izmjeniti HTML datoteke. Montaža se odvija prilikom pokretanja Docker kontejnera s parametrom `-v /glavno/racunalo:/kontejner`. Za razliku od volumena vezne montaže, Docker upravljani volumeni su stvoreni i upravljani preko Docker servisa. Također se koristi paramtar `-v`, no, za razliku od vezne montaže, zadaje se samo lokacija na kontejneru. Na primjer, za naredbeni parametar `-v /direktorij` Docker će stvoriti volumen te će ga montirati na kontejner unutar `/direktorij`. Docker volumeni mogu se monitrati samo za čitanje, te isti mogu biti montirani na više Docker kontejnera u istom trenutku.

Aplikacije koje imaju potrebu za mrežnim pristupom, poput web servisa ili servisa elektroničke pošte, mogu se pokrenuti unutar Docker kontejnera. Najčešće se koristi virtualna mreža po kontejneru (*engl. single-host virtual network*), prikazana na slici 2.15. To omogućava pojedinom kontejneru pravo pristupa na određene mrežne portove. Docker omogućava i povezivanje više Docker kontejnera u virtualnoj mreži (*multi-host network*) kod koje Docker kontejneri mogu slobodno međusobno komunicirati kao u lokalnoj mreži.



Slika 2.15: Docker virtualna mreža

Docker kontejneri imaju četiri vrste mrežne izolacije:

- zatvoreni kontejneri koji nemaju mrežni pristup
- mostovni kontejneri koji imaju ograničeni mrežni pristup
- spojeni kontejneri koji imaju ograničeni mrežni pristup vanjskoj mreži i neograničeni pristup između kontejnera
- otvoreni kontejneri koji imaju neograničeni mrežni pristup preko logičke mreže glavnog računala.

Docker omogućava ograničavanje pristupa resursima kao što su radna memorija, procesor, vanjski uređaji. Prilikom pokretanja Docker kontejnera korisnik može postaviti parametar `-m` koji ograničava količinu radne memorije. Na primjer, `-m 1g` ograničava radnu memoriju na 1 GB za taj kontejner. Bitno je naglasiti da taj parametar ne rezervira memoriju te memorija nije garantirana. Ograničenje pristupa procesoru postiže se parametrom `--cpu-shares` i parametrom `--cpuset-cpus`. Parametar `--cpuset-cpus` prihvaća listu CPU jezgara koje kontejner može koristiti. Češće korišten parametar `--cpu-shares` omogućava korisniku da dodjeli relativnu količinu CPU resursa kontejneru. Na primjer, ukoliko su pokrenuta dva kontejnera; jedan s `--cpu-shares 512`, a drugi s `--cpu-shares 1024`, tada će drugi kontejner dobiti dva CPU ciklusa za svaki CPU ciklus prvog kontejnera. Takva relacija može se opisati matematičkom jednadžbom 2.1.

$$r_k = \frac{cpu_k}{\sum_{i=1}^{n} cpu_i} \tag{2.1}$$

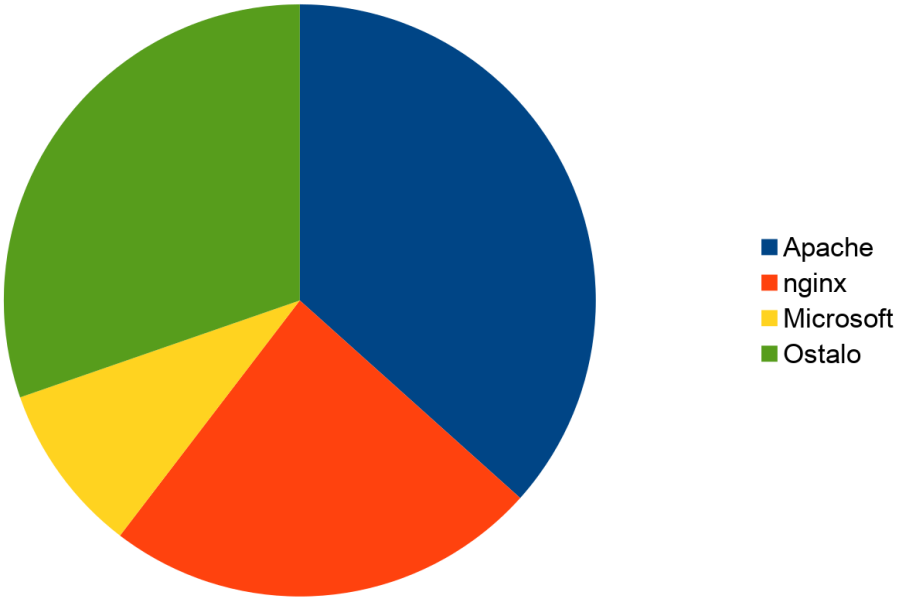
gdje je:

- r_k omjer procesorskih ciklusa kontejnera k u intervalu $[0, 1]$
- cpu_k parametar `--cpu-shares` za kontejner k.

2.4 Nginx

Nginx je web poslužitelj otvorenog koda napisan 2004. godine u C programskom kodu koji se često koristi kao balansiranje opterećenja (*engl. Load Balancer*) i obrnuti posrednik (*engl. reverse proxy*). Nastao je kao zamjena za Apache web poslužitelj s boljim performansama,a prema analizi iz 2017. godine, Nginx i dalje ima bolje performanse [15]. Kompanija Netcraft koja se bavi istraživanjem web stranica i servisa, obavlja istraživanje svakih mjesec dana. U veljači 2018. godine svaka četvrta internet stranica bila je posluživana preko Nginx poslužitelja [16]. Udio pojedinih web servera prikazan je slikom 2.16.

Nginx se sastoji od glavnog procesa koji raspodjeljuje posao radnicima. Radnik je proces koji prima naredbe od glavnog procesa i baziran je na asinkroniziranom principu [17] te se oslanja na Linux sistemske pozive `epoll/select/poll`. Stoga radnik poslužuje više zahtjeva odjednom s izuzetno malo dodatnih resursa. Zbog toga se Nginx često koristi kao balanser opterećenja. Balansiranje opterećenja je potrebno ukoliko jedan poslužitelj aplikacije nema dovoljno resursa kako bi poslužio sve klijente [18]. Nginx podržava tri algoritama balansiranja, a na samom korisniku je da utvrdi koji je najbolji za aplikaciju. Prilikom korištenja balancera opterećenja, Nginx služi kao obrnuti posrednik; klijenti se spajaju na Nginx koji zatim odlučuje, ovisno o definiranim pravilima, gdje će proslijediti takav zahtjev. Nginx ima mogućnost dodati i uređivati određene informacije na takav zahjev. Nakon primitka odgovora, Nginx također ima mogućost promjene informacija. Takav se obrađeni odgovor zatim šalje klijentu.



Slika 2.16: Udio pojedinih web servera

2.4.1. Nginx konfiguracija

Pri pokretanju Nginx poslužitelja potrebno je zadati konfiguracijsku datoteku koja sadrži informacije o mrežnim protokolima i mrežnim portovima. Ukoliko se Nginx koristi kao balanser opterećenja ili obrnuti posrednik, korisnik mora definirati gdje će se zahtjevi proslijediti. Primjer minimalne Nginx konfiguracije prikazan je kodom 2.4

Parametar `worker_process` je broj radnih procesa koje će Nginx poslužitelj pokrenuti. Svaki proces izvršava se samo na jednoj procesorskoj jezgri pa se preporuča vrijednost jednakoj broju dostupnih procesorskij jezgri. Blok `events` definira kako će se obraditi korisnički zahtjevi. Unutar toga bloka, `worker_connections` definira maksimalan broj paralelnih zahtjeva po radniku.

Unutar `http` bloka nalazi se `server` blok koji naređuje Nginx da prihvaća zahtjeve na određenom mrežnom portu. U danom primjeru to je port 80. `server_name` je naziv domene za koju će zahtjev biti obrađen. Domena je dostupna preko `HTTP` zaglavlja naziva `Host` definiranog standardom *RFC 2616* [19]. Nadalje, `location` blok definira kako će se svi zahtjevi obraditi. Pomoću naredbe `proxy_pass` moguće je preusmjeriti zahjev na proizvoljni poslužitelj. Na primjeru danom kodom 2.4, svi `HTTP` zahtjevi prosljeđeni su na `www.ferit.unios.hr` server.

Prilikom pokretanja Nginx proces učitava konfiguracijsku datoteku. Ukoliko dođe do izmjene konfiguracijske datoteke potrebno je ponovno učitati konfiguracijsku datoteku. To se može izvršiti pokretanjem nginx procesa s parametrom `-s reload`, ponovnim pokretanjem servisa pomoću `systemd reload nginx` ili slanjem Unix signala `SIGHUP`. Novu konfiguraciju moguće je učitati pomoću gašenja i ponovnog paljena Nginx servisa, no primjenom takve metode aplikacije će nakratko biti nedostupna.

```
worker_processes 1;

events {
   worker_connections 1024;
}

http {
   server {
      listen    80;
      server_name localhost;

      location / {
         proxy_pass https://www.ferit.unios.hr;
      }
   }
}
```

Programski kod 2.4: Nginx konfiguracija

2.5 Go programski jezik

Go je programski jezik otvorenog koda razvijen od strane kompanije Google. Projekt je nastao 2007. godine, dok je prva verzija javno objavljena 2009. godine. Go jezik inspiriran je C programskim jezikom te manje poznatim Alef i Oberon-2 jezicima [20]. Od C programskog jezika Go je naslijedio sintaksu, način kontrole toka programa, osnovne tipove podataka, pokazivače te brzo izvođenje programa i neovisnost o trenutnom operativnom sustavu. Od Oberon-2 naslijeđena je sintaksa za programske pakete, uključivanje istih te deklaraciju metoda i funkcija. Od jezika Alef naslijeđeni su kanali kao način komunikacije između procesa, bez potrebe za dijeljenom memorijom. Primjer jednostavne *Hello, World* aplikacije dan je kodom 2.5.

```
package main

import "fmt"

func main() {
   fmt.Println("Pozdrav, svijete")
}
```

Programski kod 2.5: Go primjer

Kako je Go prevođeni jezik, potrebno je prevesti Go kod u računalni program. Naredba `go build` pokreće Go prevoditelj i, ukoliko je prevođenje uspješno, stvara binarnu datoteku koja se može pokrenuti. Go je organiziran u pakete (*engl. packages*), što bi u drugim programskim

jezicima bile biblioteke ili moduli. Go paket se sastoji od jednog ili više Go izvornih datoteka unutar istog direktorija. U danom primjeru, naziv paketa je `main`. Go standardna biblioteka sadrži preko 100 paketa za uobičajne zadaće kao što su ulazi/izlazi, manipuliranje teksta, sortiranje, kriptografske funkcije, itd. Paket `fmt` korišten u primjeru sadrži funkcije za obradu ulaza te ispis formatiranog izlaza. Funkcija `Println` ispisuje jednu ili više vrijednosti odvojenih zarezom na standardni izlaz s novim redom na kraju ispisa. Go prevoditelj je strog oko pravila pisanja izvorne datoteke. Program `gofmt` koristi se za preoblikovanje izvorne datoteke. Na primjer, ukoliko je korišten razmak umjesto tabulatora, `gofmt` će zamjeniti razmake sa tabulatorom.

2.5.1. Go rutine i kanali

Često je prilikom izgrade aplikacija, pogotovo mrežnih, potrebno obrađivati više zahtjeva u isto vrijeme. Go rješava taj problem s Go rutinama. Go rutina je slična dretvama operacijskog sustava, no za razliku od njih koristi manje resursa. Moguće je pokrenuti na tisuće Go rutina bez usporavanja aplikacije. Pomoću ključne riječi `go` pokreće se funkcija u posebnoj Go rutini, kao što je prikazano na primjeru koda 2.6.

```
package main

import (
   "fmt"
   "time"
)

func pozdrav(str string) {
   fmt.Println("Pozdrav", str)
}

func main() {
   go pozdrav("iz Go rutine")
   pozdrav("iz glavne funkcije")
   time.Sleep(1 * time.Millisecond)
}

/*
Ispisuje:
Pozdrav iz glavne funkcije
Pozdrav iz Go rutine
*/
```

Programski kod 2.6: Go rutine

Na danom primjeru pokrenute su dvije Go rutine. Prva Go rutina je glavni program, a druga je

pokrenuta eksplicitno s linijom `go pozdrav("iz Go rutine")`. Kao što je napisano unutar koda, prva linija koja će biti ispisana je "Pozdrav iz glavne funkcije". Prilikom korištenja ključne riječi `go` rutina se stavlja u registar. Izvršavanje započinje tek kada je jedna od procesorskih jezgra slobodna. Bez linije `time.Sleep` funkcija `main` bi završila prije nego što bi se izvršila eksplicitna go rutina, a samim time i program te poruka "Pozdrav iz Go rutine" ne bi bio ispisana.

Kanali omogućuju komunikaciju i sinkronizaciju između Go rutina. Podatci se mogu slati i primati preko kanala. Ključna riječ `chan` koristi se za stvaranje kanala, a potrebno je pri tome i odredit tip podatka. Postoje dvije vrste kanala: kanali bez memorije (*unbuffered channels*) i kanali s memorijom (*engl. buffered channels*). Kanali bez memorije su kanali koji blokiraju program sve dok neka Go rutina ne pročita takvu poruku. S druge strane, kanali s memorijom dopuštaju nastavak programa nakon zapisa u kanal, pod uvjetom da kanal ima dovoljno memorije. Primjer korištenja kanala i Go rutine prikazan je kodom 2.7.

```
package main

import "fmt"

func main() {
   // Stvaranje kanala bez memorije, tipa "string"
   messages := make(chan string)

   // Slanje poruke na kanala
   go func() { messages <- "bok" }()

   // Primanje poruke s kanala
   msg := <-messages
   fmt.Println(msg)
}

/*
Ispisuje:
bok
*/
```

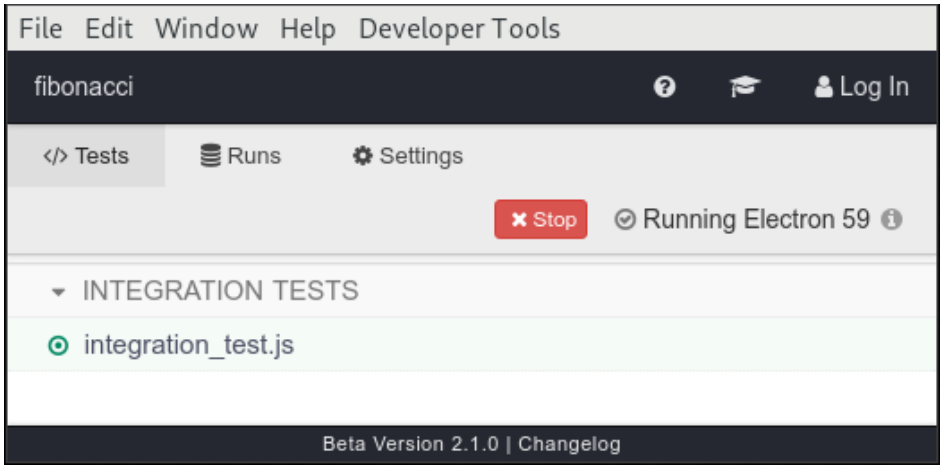
Programski kod 2.7: Go kanali

2.6 Cypress

Cypress je alat za integracijsko i *end-to-end* testiranje web stranica i aplikacija. Za razliku od sličnih alata kao što je Selenium, pokretanje Cypressa je vrlo jednostavno, ima potpuni pristup pregledniku koji vrši testiranje te sadrži pojednostavljeno programsko sučelje [21]. *End-to-end*

testiranje web stranica služi kao simuliranje stvarnih korisnika. Primjerice, ukoliko jedan od elemenata nije prikazan unutar internet preglednika, jedino *end-to-end* testiranje može otkriti takvu grešku. No, *end-to-end* testiranje je vremenski zahtjevno za pisanje, a pogotovo za održavanje. Bilo kakva vizualna promjena na aplikaciji zahtjeva promjenu *end-to-end* testova, a promjena u funkcionalnosti zahtjeva potpuno nove testove. Preporuka je Googleovih inženjera 70% testova jedinice, 20% integracijskih testova, a svega 10% *end-to-end* testova [22].

Otvaranje Cypress alata na lokalnom računalu odvija se preko naredbe `cypress start` koja otvara program za pokretanje testova, kao što je prikazano slikom 2.17. Preko aplikacije korisnik može pokrenuti testove te vidjeti rezultate. Cypress alat prati izmjene konfiguracija na tvrdom disku te ih automatski učitava ako dođe do izmjene. Cypress testovi se mogu pokrenuti i preko komandne linije, što je poželjno prilikom pokretanja na Jenkins poslužitelju. Pokretanje testa izvršava se naredbom `cypress run`. Cypress zatim pokreće preglednik u takozvanom *headless* načinu rada, odnosno bez grafičkog sučelja. Nakon završetka rada Cypress izlazni rezultat sadrži informaciju o uspješnosti testa. Cypress može spremiti sve rezultate u junit XML formatu. Primjer uspješnog rezultata dan je kodom 2.8.



Slika 2.17: Cypress aplikacija

```
<?xml version="1.0" encoding="UTF-8"?>
<testsuites name="Mocha Tests" time="1.321" tests="2" failures="0">
  <testsuite name="Fibonacci test" timestamp="2018-03-19T03:14:07" tests="2"
     failures="0" time="1.321">
    <testcase name="Fibonacci test Ispravna forma" time="0.744" classname="
       Ispravna forma"></testcase>
    <testcase name="Fibonacci test Neispravna forma" time="0.577" classname="
       Neispravna forma"></testcase>
  </testsuite>
</testsuites>
```

Programski kod 2.8: junit XML rezultat

3. KONTINUIRANI RAZVOJ NA PRIMJERU WEB APLIKACIJE

U prijašnjem su poglavlju opisani programi i infrastruktura potrebna za kontinuirani razvoj aplikacije. U ovom poglavlju je opisana implementacija sustava za kontinuirani razvoj na primjeru web aplikacije za izračun Fibonaccijevog broja koja je spremljena na Git projektu nazvanom *diplomski-go*. Opisan je proces izrade aplikacije pomoću Jenkins posla te objavljivanje aplikacije pomoću servisa za menadžment. Zatim je opisan i proces vraćanje na prethodnu verziju aplikacije.

3.1 Aplikacija za izračun Fibonaccijevog broja

Za testiranje sustava kontinuiranog razvoja izrađen je web program za izračun Fibonaccijevog broja u Go programskom jeziku. Fibonaccijev broj definiran je formulom 3.1. U prvoj iteraciji programa koristi se naivno, rekurzivno rješenje vremenske i memorijske kompleksnosti $O(2^N)$. Prilikom primitka HTTP zahtjeva, program pokreće računanje Fibonaccijevog broja, kao što je prikazano programskim kodom 3.1.

$$f(n) = \begin{cases} 0 & n = 0 \\ 1 & n = 1 \\ f(n-1) + f(n-2) & n > 1 \end{cases} \tag{3.1}$$

```
func fibonacci(n uint64) uint64 {
  if n == 0 {
    return 0
  } else if n == 1 {
    return 1
  } else {
    return fibonacci(n-1) + fibonacci(n-2)
  }
}
```

Programski kod 3.1: Fibonacci v1

Funkcija Fibonacci prima broj kao parametar. Ukoliko je broj nula, vraća vrijednost nula. Ukoliko je broj jedan, vraća vrijednost jedan. U protivnom vraća zbroj dvije Fibonacci funkcije. Prva funkcija ima parametar $broj - 1$, dok druga funkcija ima broj $broj - 2$. Na primjer, ako je zatražen broj pet, funkcija vraća zbroj dvije Fibonacci funkcije, jedna s parametrom četiri, a druge s tri. Kako bi se provjerila ispravnost koda, napisani su jednostavni testovi jedinice, prikazano kodom 3.2

Za sustav testiranja korišten je $testing$ paket koji se nalazi u standardnoj Go biblioteci. Testovi jedinice se sastoje od jednog testa s tri provjere gdje se provjerava vrijednost petog, šestog i

```
func testEqual(t *testing.T, expected, actual uint64) {
  if expected == actual {
    return
  }
  t.Errorf("Greska: \%d (ocekivano) != \%d (dobiveno)", expected, actual)
}

func TestFibonacci(t *testing.T) {
  testEqual(t, uint64(5), fibonacci(5))
  testEqual(t, uint64(8), fibonacci(6))
  testEqual(t, uint64(89), fibonacci(11))
}
```

Programski kod 3.2: Testiranje jedinice

jedanaestog Fibonaccijevog broja. Unutar svakog testa poziva se pomoćna funkcija koja prima tri vrijednosti: trenutni test, očekivana vrijednost te dobivena vrijednost. Ukoliko dobivena vrijednost nije jednaka očekivanoj vrijendosti, funkcija će označiti takav test netočnim.

Za pozivanje Fibonacci funkcije kreirana je web aplikacija, kao što je prikazano slikom 3.1. Web aplikacija se sastoji od obrasca koji sadrži prostor za unos teksta te gumb za izvršavanje operacije. Pritiskom na gumb izvršava se HTTP XHR zahtjev te se po završetku istoga prikazuje rezultat ili greška ako je do nje došlo. U glavnoj funkciji koja se poziva prilikom pokretanja programa, `main`, pokrenut je HTTP servis zadužen za posluživanje HTTP zahtjeva. Pokretanje HTTP servisa dan je kodom 3.3.

Slika 3.1: Izgled Fibonacci web aplikacije

```
func main() {
  http.HandleFunc("/", homePageHandler)
  http.HandleFunc("/api/fibonacci", fibonacciHandler)
  log.Fatal(http.ListenAndServe(":8888", nil))
}
```

Programski kod 3.3: HTTP servis

Unutar $main$ funkcije stvorena su dva HTTP rukovoditelja (*engl. handlers*). Prvi rukovoditelj obrađuje zahtjeve za početnu stranicu te, neovisno o bilo kojim argumentima, poslužuje statičnu

datoteku koja se nalazi na lokaciji `static/index.html`. Funkcija rukovoditelja je prikazana kodom kodom 3.4.

```
func homePageHandler(w http.ResponseWriter, r *http.Request) {
 http.ServeFile(w, r, "./static/index.html")
}
```

Programski kod 3.4: Rukovoditelj za statičku HTML datoteku

Drugi rukovoditelj prikazan kodom 3.5 obrađuje zahtjeve za stranicu `/api/fibonacci`. Preko GET parametra očekuje varijablu n te ju pretvara u ne-negativni cijeli broj. Ukoliko program nije u mogućnosti to napraviti, vraća grešku korisniku. U protivnom poziva Fibonacci funkciju s dobivenim brojem te vraća vrijednost korisniku u tekstualnom obliku.

```
func fibonacciHandler(w http.ResponseWriter, r *http.Request) {
 n, err := strconv.ParseUint(r.URL.Query().Get("n"), 10, 64)
 if err != nil {
  w.WriteHeader(http.StatusBadRequest)
  w.Write([]byte("Parametar n mora biti prirodni broj."))
  return
 }
 w.WriteHeader(http.StatusOK)
 w.Write([]byte(strconv.FormatUint(fibonacci(n), 10)))
}

func homePageHandler(w http.ResponseWriter, r *http.Request) {
 http.ServeFile(w, r, "./static/index.html")
}
```

Programski kod 3.5: Rukovoditelj za izračun Fibonaccijevog broja

End-to-end testovi prikazani kodom 3.6 sastoje se od dva testa. Prvi test je test u kojem se zadaje ispravan argument i zadaje izračun 11. Fibonaccijevog broja. Cypress alat otvara internet preglednik te otvara zadanu web stranicu. Nakon uspješnog otvaranja u tekstualnu kućicu unosi broj 11 te pritišće gumb "Računaj". Zatim očekuje da će stranica prikazati broj 89, koji je traženo rješenje. U drugom testu zatražen je -1. Fibonaccijev broj. Kako to nije moguće izračunati, očekivanje je da će stranica prikazati grešku da broj mora biti prirodan.

Ukoliko Cypress ne može pronaći traženi element unutar zadanog intervala, takav test je označen kao neuspješan. Na primjer, ukoliko na stranici ne postoji element s identifikacijom `number`, test će biti neuspješan. Test će također biti neuspješan ukoliko HTTP XHR zahtjev ne bude uspješan. Cypress testovi se mogu pokrenuti na lokalnom računalu pomoću naredbe $cypress\ open$. Izmjena testova će automatski biti dostupna unutar grafičkog sučelja Cypress. Testovi se također

mogu pokrenuti naredbom $cypress\ run$ koja pokreće testove te izlazi po završetku svih. Rezultati su dostupni preko standardnog izlaza i izlaznog stanja.

```
describe('Fibonacci test', function() {
   it("Ispravna forma", function() {
      cy.visit('/');
      cy.get('#number').type('11');
      cy.get('form').contains('Racunaj').click();
      cy.contains('89');
   });
   it("Neispravna forma", function() {
      cy.visit('/');
      cy.get('#number').type("-1");
      cy.get('form').contains('Racunaj').click();
      cy.contains('Parametar n mora biti prirodni broj.');
   });
})
```

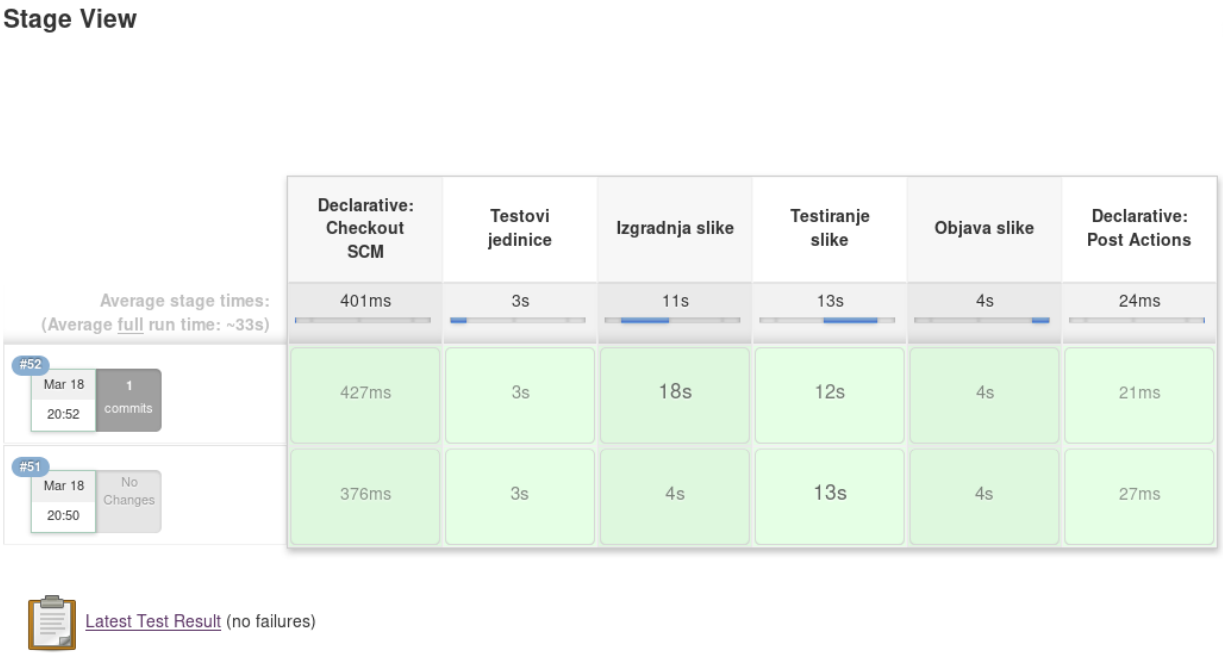
Programski kod 3.6: *End-to-end* testiranje

3.2 Jenkins posao za izgradnju i objavu Docker slike

Za Jenkins posao koji izrađuje Docker sliku i objavljuje u Docker repositorij odabrana je Jenkins linija. Jenkins linija je isprogramirana i spremljena u datoteci `Jenkinsfile` unutar Fibonacci projekta, djelomično prikazana kodom 3.7. Jenkins linija se sastoji od šest faza: dohvaćanje koda preko Git projekta, pokretanje testova jedinica, izgradnja Docker slike, pokretanje *end-to-end* testova, objava Docker slike u javni Docker repositorij te objava rezultata. Na slici 3.2 prikazana je Jenkins linija.

```
pipeline {
   agent any
   stages {
      stage('Testovi jedinice') { /* implementacija */ }
      stage('Izgradnja slike') { /* implementacija */ }
      stage('Testiranje slike') { /* implementacija */ }
      stage('Objava slike') { /* implementacija */ }
   }
   post { /* implementacija */ }
}
```

Programski kod 3.7: Struktura *Jenkinsfile*



Slika 3.2: Jenkins linija za izgradnju i objavu slike

Jenkins linija podešena je da periodički provjerava Git projekt, točnije svake minute. Ukoliko je došlo do promjene u Git projektu, pokreće Jenkins liniju. U prvoj fazi Jenkins kopira Git projekt koji se poslužuje preko Github servisa. Git projekt javno je dostupan, stoga nije potrebno podesiti autorizaciju. Nakon što je Git projekt kopiran, Jenkins uspoređuje razliku između prijašnje i trenutnačne verzije te ih prikazuje korisniku. U drugoj fazi pokreću se testovi jedinice unutar Docker kontejnera, prikazano kodom 3.8. Fibonacci aplikacija sastoji se od jednog testa jedinice koji ispituje tri ulaza. Ukoliko dođe do greške, Jenkins linija se prekida. Pomoću *junit* skripte Jenkins objavljuje rezultate testova, kao i vrijeme njihovog izvođenja.

```
steps {
   sh 'docker build -f fibonacci/Dockerfile.test -t go_test fibonacci'
   sh 'docker run --rm go_test > tests.xml'
}
```

Programski kod 3.8: Testiranje jedinice unutar *Jenkinsfile*

U trećoj fazi aplikacija se kompajlira i sprema unutar Docker slike, što je prikazano kodom 3.9. Druga i treća faza dijele istu baznu Docker sliku. Stoga inženjer može biti siguran da su sve biblioteke dostupne ukoliko su pokrivene testovima jedinice. Docker slika se označava s trenutnom verzijom izgradnje (*engl. build number*) te se objavljuje na Docker repositorij. Kako bi se slika mogla objaviti, potrebno je podesiti Docker autorizaciju unutar Jenkins sustava. U četvrtoj fazi pokreće se *end-to-end* testiranje pomoću Cypress alata, prikazano kodom 3.10. Jenkins pokreće Docker kontejner koji pokreće zadnju verziju aplikacije izgrađenu u prethodnom koraku. Zatim u drugom Docker kontejernu pokreće Cypress alat, koji dalje pokreće testove koji komuniciraju s

novom verzijom aplikacije. Rezultati se objavljuju pomoću *junit* skripte.

```
steps {
   script {
      docker.withRegistry('https://docker.io', 'dockerhub') {
         def i = docker.build("sokac/fibonacci:${BUILD_NUMBER}", '-f
            fibonacci/Dockerfile fibonacci/')
         i.push()
      }
   }
}
```

Programski kod 3.9: Izgradnja docker slike unutar *Jenkinsfile*

```
steps {
   script {
      docker.withRegistry('https://docker.io', 'dockerhub') {
         def i = docker.image("sokac/fibonacci:${BUILD_NUMBER}")
         i.withRun { c->
            sleep 1 // cekaj sekundu
            def ii = docker.build('sokac/fibonacci_integration', '-f
               fibonacci/Dockerfile.integration fibonacci/')
            ii.inside("--shm-size=1g --link ${c.id}:app") {
               sh 'cd fibonacci && CYPRESS_baseUrl=http://app:8888/ /
                  node_modules/.bin/cypress run -r junit'
            }
         }
      }
   }
}
```

Programski kod 3.10: Pokretanje *end-to-end* testova unutar *Jenkinsfile*

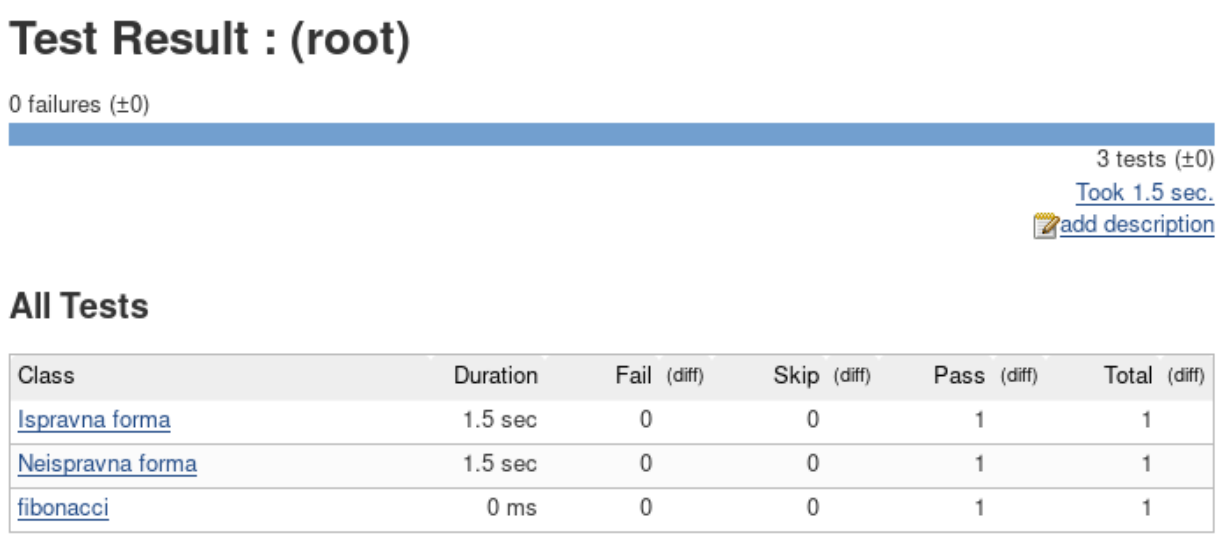
U petoj fazi, prikazanom kodom 3.11 Docker slika se označava sa specijalnom oznakom *latest*. Ta oznaka označava da je Docker slika spremna za uporabu od strane klijenata. Tu sliku dohvatit će servis za menadžment. U zadnjoj fazi, koja se bezuvjetno izvršava, prikupljaju se rezultati testova jedinica i *end-to-end* testiranja. Rezultati su vidljivi korisniku preko Jenkins web sučelja kao što je prikazano na slici 3.3.

3.3 Servis za menadžment

Servis za menadžment je servis zadužen za preuzimanje zadnje Docker slike objavljene na Docker repositoriju. Prije pokretanja servisa korisnik mora podesiti konfiguracijku datoteku. Konfiguracijska datoteka je datoteka u *JSON* formatu koja sadrži informacije o Docker slici, lokaciji

```
steps {
   script {
      docker.withRegistry('https://docker.io', 'dockerhub') {
         def i = docker.image("sokac/fibonacci:${BUILD_NUMBER}")
         i.push('latest')
      }
   }
}
```

Programski kod 3.11: Objava Docker slike unutar *Jenkinsfile*



Slika 3.3: Jenkins linija za izgradnju i objavu slike

Nginx PID datoteci, lokaciji Nginx datotečnoj konfiguracije te vremenu čekanja prilikom rotiranja verzije aplikacije. Primjer konfiguracije prikazan je kodom 3.12.

```
{
   "dockerImage": "sokac/fibonacci",
   "nginxPIDfile": "/run/nginx.pid",
   "nginxConfiguration": "/tmp/nginx.conf",
   "versionOverlapDuration": 30
}
```

Programski kod 3.12: JSON konfiguracija

Informacija o nazivu Docker slike spremljena je pod ključem `dockerImage`. Servis zahtjeva da zadnja inačica slike ima oznaku *latest*, što je ujedno i Docker standard. Nginx PID datoteka, spremljena pod ključem `nginxPIDfile`, sadrži indentitet Nginx procesa. Pomoću tog indentiteta moguće je poslati zahtjev Nginx procesu da ponovno učita konfiguracije datoteke bez prekida posluživanja korisnika. Nginx datotečna konfiguracija sprema postavke potrebne da Nginx može

proslijediti informacije aplikaciji, to jest Docker kontejneru. Vrijednost je spremljena pod ključem `nginxConfiguration`. Primjer generirane konfiguracije prikazan je kodom 3.13.

```
# Managed by Manager
upstream manager_backend {
   server 127.0.0.1:45105;
}
```

Programski kod 3.13: Nginx konfiguracija

3.3.1. Arhitektura servisa za menadžment

Servis za menadžment sastoji se od četiri glavne komponente:

- komponenta za konfiguraciju
- upravitelj *Unix* signala
- dohvatitelj Docker slike
- upravitelj Docker slike

Komponenta za konfiguraciju zadužena je za učitavanje korisničke *JSON* datoteke te provjeru njene ispravnosti. Za učitavanje datoteke s tvrdog diska korišten je `io/ioutil` paket koji služi za ulazno–izlazne operacije. Kako je konfiguracijska datoteka u *JSON* formatu, korišten je `encoding/json` paket koji se nalazi u standardnoj Go biblioteci i baziran je na standardu *RFC 7159* [23]. Ukoliko jedna od vrijednosti konfiguracije nije ispravna, komponenta vraća grešku.

Upravitelj signala je zasebna Go rutina koja čeka dva *Unix* signala: `SIGINT` i `SIGTERM` koji se zaprimaju pomoću specijalnog kanala. Ukoliko servis zaprimi jedan od ta dva spomenuta signala, započinje s milosrdnim gašenjem servisa pomoću zatvaranja specijalnog kanala. Upravitelj signala je potreban kako bi sustav mogao biti sigurno ugašen i u trenutku kada je nova verzija aplikacije objavljena.

Dohvatitelj Docker slike je također zasebna Go rutina. Ova komponenta je zadužena za povremenu provjeru zadnje Docker slike na udaljenom poslužitelju. Za komunikaciju s Docker servisom koristi službeni Docker SDK za Go jezik. Komponenta svakih trideset sekundi dohvaća zadnju sliku s udaljenog poslužitelja te provjerava identifikator Docker slike s prijašnjom verzijom. Ukoliko postoji izmjena u verziji, tada šalje vrijednost identifikatora u kanal koji se očitava preko komponente upravitelja Docker slike. Algoritam dohvaćanja nove slike prikazan je kodom 3.14.

Komponenta upravitelja Docker slike je isto tako zasebna Go rutina. Za razliku od dohvatitelja, ona isključivo čeka na poruku od kanala. Kada je identifikator Docker slike zaprimljen kroz kanal, pokreće se algoritam za pokretanje nove aplikacije. Algoritam prvo mora pronaći slobodan TCP port na kojem nova aplikacija može slušati, što je prikazano kodom 3.15. Korištenje statičkog

```
func (c *DockerFetcher) fetchLatest() {
  ctx := context.Background()
  defer ctx.Done()

  refStr := fmt.Sprintf("%s:latest", c.image)
  o, err := c.cli.ImagePull(ctx, refStr, types.ImagePullOptions{})
  if o != nil {
    defer o.Close()
  }
  if err != nil {
    log.Println("Couldn't pull the latest image", err.Error())
    return
  }
  inspect, _, err := c.cli.ImageInspectWithRaw(ctx, refStr)
  if err != nil {
    log.Println("Couldn't pull the latest image", err.Error())
    return
  }
  if c.latestID != inspect.ID {
    log.Println("New image found", inspect.ID)
    c.ch <- inspect.ID
    c.latestID = inspect.ID
  }
}
```

Programski kod 3.14: Algoritam dohvatitelja Docker slike

porta nije moguće jer bi prethodna verzija aplikacija slušala na tom portu. Ukoliko se stara verzija aplikacija ugasi radi pokretanja nove verzija, tada će takva aplikacija biti nedostupna onoliko koliko je potrebno da se aplikacija pokrene, što nije prihvatljivo za moderne aplikacije.

Nakon što je slobodan port pronađen, pokreće se Docker kontejner sa zadnjom verzijom aplikacije. Kontejner se pokreće u mostovnom mrežnom načinu rada te prosljeđuje zathjeve sa slobodnog porta glavnog računala prema aplikaciji koja se izvršava unutar Docker kontejnera. Također se zadaje naredba Docker servisu da obriše sliku po zaustavljanju Docker kontejnera. Nakon pokretanja Docker kontejnera, obavještavaju se svi pretplatnici. Pretplanik je struktura opisana `Subscriber` sučeljem, koji je prikazan kodom 3.16. Pretplanik može izvšiti bilo kakvu operaciju nakon što je obavješten, kao što je slanje elekroničke pošte za novu verziju, slanje *IRC* poruke, promjena konfiguracije web servisa, rotacija log datoteka, itd.

U ovom radu isprogramiran je samo jedan pretplanik - Nginx. Nginx pretplatnik zapisuje novu konfiguraciju unutar datoteke zadanu konfiguracijom `nginxConfiguration`. Konfiguracijska datoteka prikazana je kodom 3.13. Nakon što je datoteka zapisana, učitava se procesni identifikator nginx procesa preko *PID* datoteke. Zatim se šalje Unix signal `SIGHUP` nginx procesu

```
func getFreePort() (int, error) {
  tcpAddr, err := net.ResolveTCPAddr("tcp", "127.0.0.1:0")
  if err != nil {
    return 0, err
  }
  l, err := net.ListenTCP("tcp", tcpAddr)
  if err != nil {
    return 0, err
  }
  defer l.Close()
  return l.Addr().(*net.TCPAddr).Port, nil
}
```

Programski kod 3.15: Algoritam za pronalazak slobodnog porta

```
type Subscribers interface {
  NewContainer(dockerPort int)
}
```

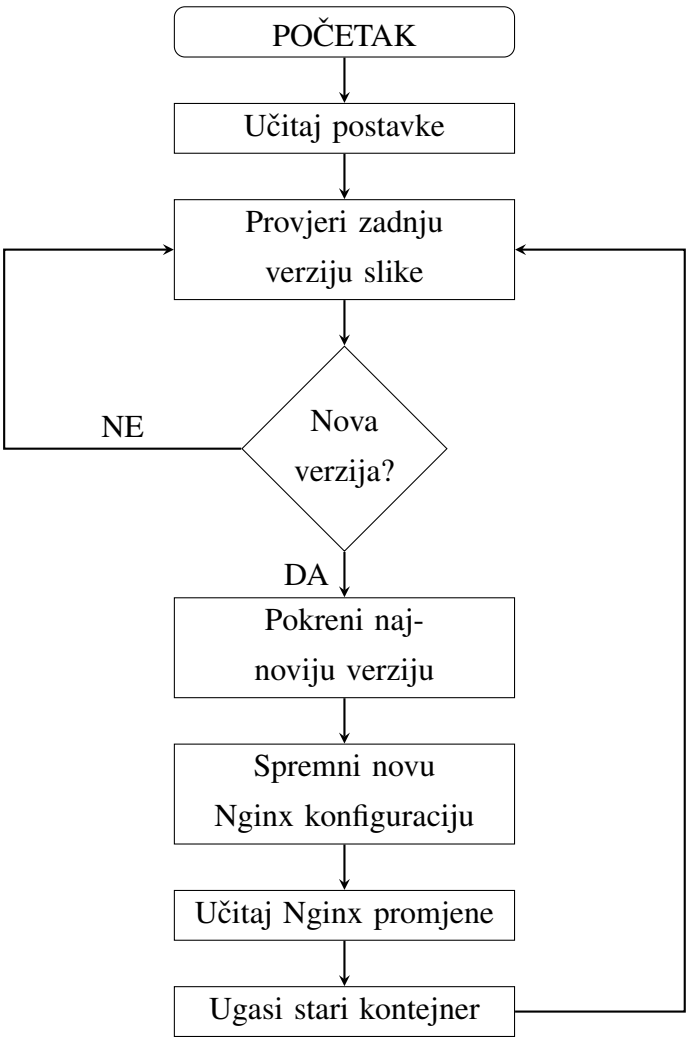
Programski kod 3.16: Sučelje pretplatnika

pomoću `syscall` paketa koji obavještava Nginx da ponovno učita konfiguracijske datoteke. U tom trenutku Nginx počinje posluživati korisnike s novom aplikacijom.

Nakon što su svi pretplatnici obavješteni i izvršili potprogram, upravitelj Docker slike čeka zadano vrijeme `versionOverlapDuration` te zatim gasi stari Docker kontejner. Stari Docker kontejner prima signal `SIGTERM` koji omogućava milosrdno gašenje aplikacije. Ukoliko se kontejner ne ugasi nakon deset sekundi, šalje se `SIGKILL` koji nasilno gasi aplikaciju i kontejner, koji se zatim briše.

3.3.2. Rad servisa

Prilikom pokretanja servisa učitava se konfiguracijska datoteka te se provjerava njena ispravnost. Ukoliko je konfiguracija datoteka neispravna, program ispisuje pogrešku te prekida rad. Ako je konfiguracijska datoteka ispravna, servis počinje periodički provjeravati izmjene na udaljenom repositoriju. Ukoliko je došlo do promjene slike s oznakom *latest*, servis pokreće novi Docker kontejner koji prihvaća zahtjeve na nekorištenom *TCP portu*. Zatim zapisuje novu Nginx konfiguraciju na tvrdi disk te šalje *SIGHUP Unix* signal Nginx procesu. Takav signal služi za učitavanje promjena konfiguracije. Nakon korisnički definiranog vremena, stari Docker kontejner se gasi te automatski briše. Pojednostavljeni dijagram toka prikazan je slikom 3.4. Generirana Nginx konfiguracija sadrži samo informacije o Docker kontejeru te kao takva nije potpuna. Glavna Nginx konfiguracija priključuje takvu konfiguraciju pomoću direktive `include`. Primjer glavne Nginx konfiguracije prikazan je kodom 3.17.



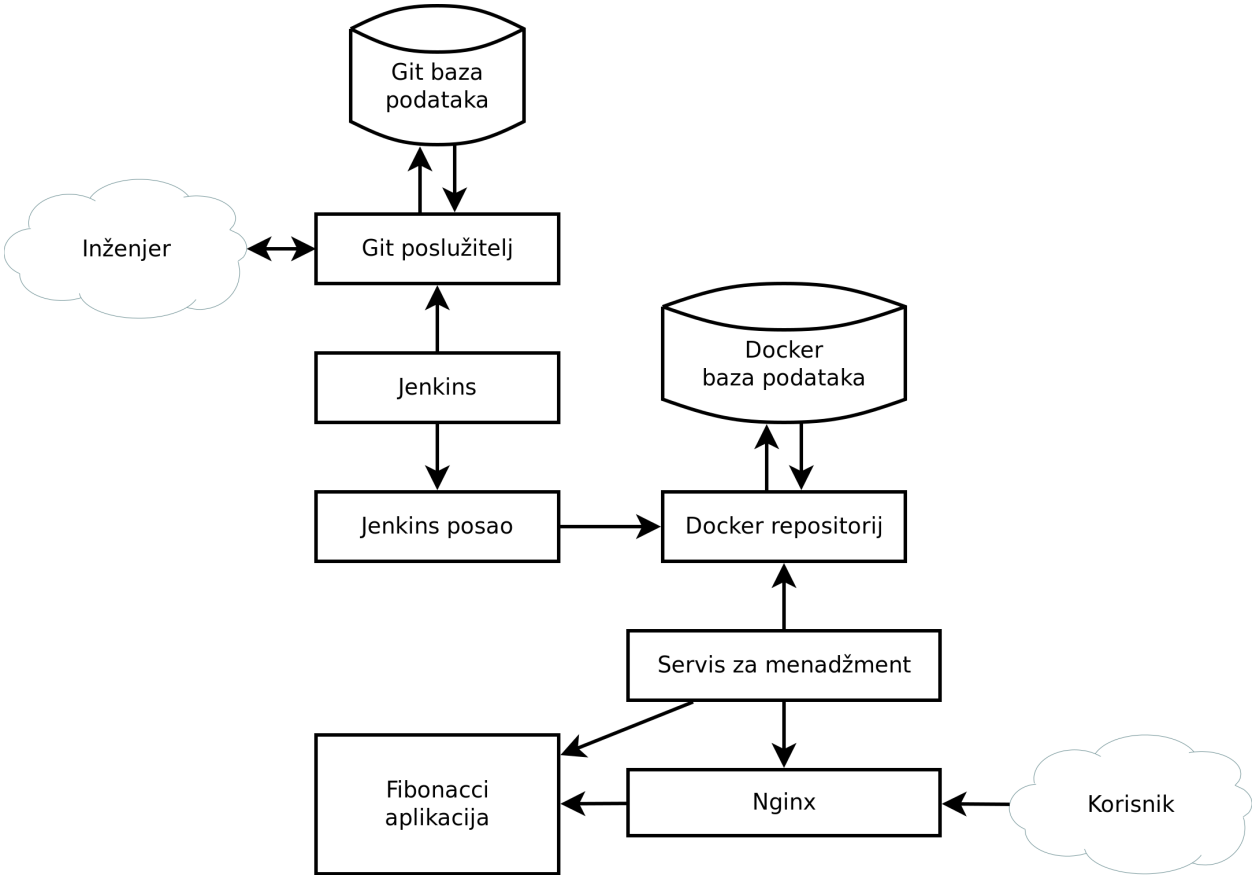
Slika 3.4: Pojednostavljeni dijagram toka servisa za menadžment

```
worker_processes 1; # broj radnika
events {
   worker_connections 1024; # broj veza po radniku
}
http {
   include /tmp/nginx.conf; # Lokacija generirane Nginx konfiguracije
   server {
      listen    80; # Port slusanja
      server_name localhost; # Domena
      location / {
         proxy_pass http://manager_backend; # Proslijedi zahtjev
      }
   }
}
```

Programski kod 3.17: Nginx konfiguracija

3.4 Proces kontinuiranog razvoja aplikacije

Proces objave nove aplikacije započinje s izmjenama unutar Git repositorija. Jenkins periodički provjerava izmjene unutar *master* grane Git repositorija te na svakoj izmjeni pokreće Jenkins posao zadužen za testiranje i objavu nove verzije aplikacije. Nakon što Jenkins objavi novu verziju, servis za menadžment ju dohvaća te pokreće algoritam za izmjenu verzija. Na slici 3.5 prikazan je pojednostavljani proces.



Slika 3.5: Proces objave nove aplikacije

Iz slike je vidljivo da inženjer komunicira isključivo s Git poslužiteljem. Takva komunikacija može biti pomoću Git komandne linije i naredbe `git-push` ili preko grafičkog sučelja programa za pregleda koda, kao što su *Github Pull Request*, *Gerrit Code Review* i *Phabricator Differential*. Iako je sustav potpuno automatizitran potrebno je pratiti proces objave nove aplikacije. Ukoliko neki od testova nije bio uspješan, na inženjeru je da vrati aplikaciju u prethodno stanje ili ispravi grešku. Također, postoji mogućnost i da svi testovi budu uspješni, a da aplikacija ne radi ispravno. Iz slike je vidljivo i da krajnji korisnik komunicira isključivo s Nginx poslužiteljem koji komunicira isključivo s Fibonacci aplikacijom. Korisnik neće biti u mogućnosti korisitit aplikaciju samo ako ili Nginx poslužitelj ili Fibonacci aplikacija nije dostupna. Ispad bilo kojeg drugog sustava ili servisa ne utječe na dostupnost aplikacije, već samo na mogućnost objave nove verzije.

3.5 Razvoj nove verzije Fibonacci aplikacije

U drugoj verziji poboljšana je vremenska kompleksnost tako što je uklonjena rekurzija te dodana memoizacija. Nova vremenska kompleksnost je $O(N)$, dok je memorijska kompleksnost $O(1)$. Nova verzija prikazana je kodom 3.18. U novoj je verziji također izmijenjen dizajn kao što je prikazano slikom 3.6

```
func fibonacci(n uint64) uint64 {
  if n == 0 {
    return 0
  }
  a := uint64(0)
  b := uint64(1)

  for n > 1 {
    tmp := a + b
    a = b
    b = tmp
    n--
  }
  return b
}
```

Programski kod 3.18: Fibonacci v2

Slika 3.6: Novi dizajn Fibonacci aplikacije

Nakon što su napravljene izmjene unutar aplikacije, potrebno je spremiti izmjene unutar Git repositorija te poslati izmjene na servis za pregled koda. Primjer za spremanje izmjena i slanje na udaljeni Git poslužitelj pomoću komandne linije prikazan je kodom 3.19. Kada je kod pregledan i odobren tada se prebacuje u *master* granu. Sustavi za reviziju koda u pravilu to automatski odrađuju nakon što je kod potvrđen za spajanje. Ručno prebacivanje prikazano je kodom 3.20.

```
# nova git grana pod nazivom optimizacija
git checkout -b optimizacija

# dodaj datoteku main.go koja sadrzi izmjene
git add main.go

# dodaj izmjenu
git commit -m "Optimizacija Fibonacci funkcije"

# posalji izmjene na server i prati udaljenu granu
git push -u origin optimizacija
```

Programski kod 3.19: Spremanje i slanje izmjena na Git poslužitelj

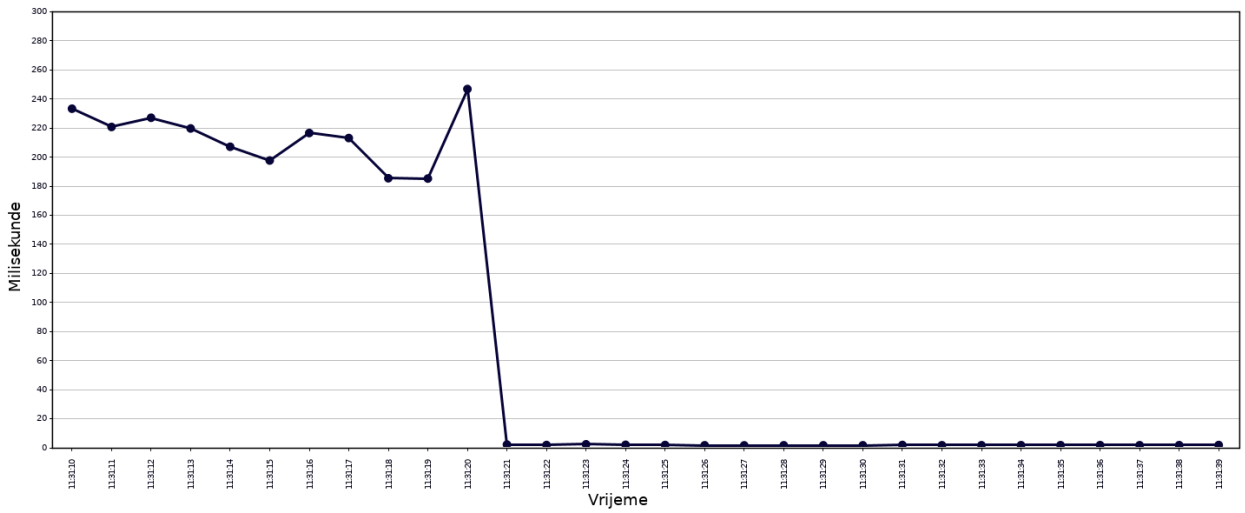
```
# prelazak u master git granu
git checkout master

# spajanje grane optimizacija u granu master
git merge optimizacija

# posalji izmjene na server, master grana
git push origin master
```

Programski kod 3.20: Spremanje i slanje izmjena na Git poslužitelj

Jenkins pokreće izgradnju nove aplikacije čim je nekakva izmjena dostupna na Git poslužitelju unutar glavne, *master* grane. Ukoliko su svi testovi uspješni objavljuje se Docker slika. Zatim servis za menadžment dohvaća novu verziju te počinje posluživati korisnike s novom verzijom. Prethodnu verziju aplikacije potrebno je ukloniti tek nakon što su svi zahtjevi obrađeni. Na primjer, ukoliko je potrebno dvije sekunde za obradu nekog zahtjeva, tada je potrebno pričekati barem toliko vremena prije nego što se prethodna verzija aplikacije može potpuno ugasiti. U protivnom zahtjev korisnika će biti prekinut i neće biti potpuno ispunjen. Testiranje stabilnosti aplikacije prilikom izmjene koda provedeno je alatom *Jmeter*. Jmeter je alat za stresno testiranje aplikacije, a razvija ga neprofitna organizacija *Apache*. Može se koristiti za stresno testiranje FTP, LDAP, HTTP, baza podataka preko JDBC, i mnogih drugih protokola. Za testiranje infrastrukture ovog projekta korišten je HTTP protokol. Izvršeno je 10.000 zahtjeva za 34. Fibonaccijev broj. U trenutku pokretanja stresnog testa korištena je stara verzija Fibonacci aplikacije, a potom je objavljena nova verzija koja sadrži optimizaciju. Svi zahtjevi uspješno su izvršeni te je na slici 3.7 prikazan vremenski odaziv prve i druge verzije aplikacije.



Slika 3.7: Vrijeme odaziva aplikacije

3.6 Tipični problemi s objavom nove verzije aplikacije

Bilo koja izmjena aplikacije može uzrokovati neočekivanu pogrešku, stoga svaka izmjena aplikacije predstavlja rizik. Osim logičkih grešaka unutar aplikacije, koje se mogu otkriti testovima jedinice i *end-to-end* testovima, moguće se i greške koje se događaju samo prilikom objavljivanja nove verzije. Primjerice, aplikacija može zapisivati podatke unutar baze podataka u određenom formatu koji nije kompatibilan s novom verzijom aplikacije. Vraćanje na prethodnu verziju aplikacija može dovesti i do dodatnih komplikacija ukoliko korisnik u trenutku ima novu verziju s novom bazom podataka.

Tipični problemi s objavom nove verzije aplikacija su:

- nekompatibilnost privremene memorije (*engl. cache*)
- nekompatibilnost baze podataka
- nekompatibilnost programskog sučelja klijenta i poslužitelja

Nekompatibilnost privremene memorije javlja se kada nova verzija aplikacije promjeni strukturu privremene memorije te očita neočekivan podatak generiran iz prijašnje verzije aplikacije. Situacija se može dodatno zakomplicirati ako su u danom trenutku pokrenute dvije verzije. Kako bi se izbjegla takva situacija najčešće se preporuča novi ključ za privremenu memoriju ili progresivna izmjena podataka privremene memorije. Prilikom korištenjem novog ključa nova aplikacija piše u novu privremenu memoriju te ne ovisi o prethodnoj verziji. Ovakvo rješenje ima problem s hladnom predmemorijom (*engl. cold cache*) koje uzrokuje s povećanim vremenom odaziva. Druga opcija je progresivna izmjena podataka koja zahtjeva dodatni kod kako bi stari i novi unos bio kompatibilan s obje verzije aplikacije. Potrebne su barem dvije verzije i neko vrijeme kako bi se mogla ispustiti podrška za prethodnu strukturu predmemorije.

Na primjeru Fibonacci aplikacije, zamislimo da koristimo sustav za memoriju, kao na primjer

memcached. U trenutnoj verziji aplikacija sprema vrijednost N-tog Fibonaccijev broja pod ključem *fibonacci-N*. U trenutnoj verziji aplikacije odlučeno je da je prvi Fibonaccijev broj nula, a drugi broj jedan. No u novoj verziji aplikacije odlučeno je koristiti originalnu verziju algoritma, tako da su i prvi i drugi Fibonaccijevi brojevi jednaki broju jedan. Ukoliko se rezultati memorije ne uklone ili spreme pod drugim ključem, tada će nova verzija Fibonacci aplikacije i dalje vraćati stare rezultate.

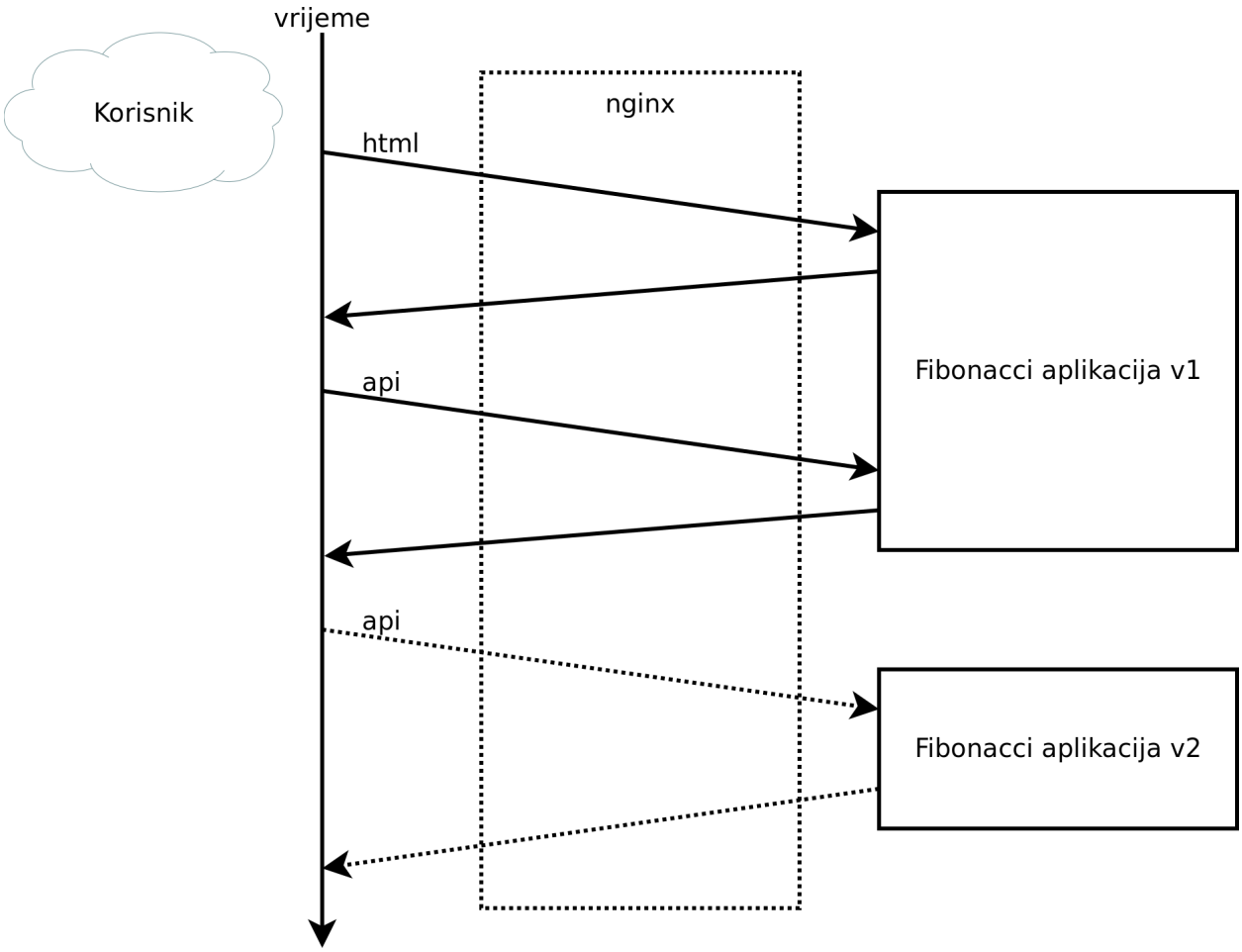
Nekompatibilnost baze podataka je slična nekompatibilnosti privremene memorije. No za razliku od privremene memorije, nije moguće izabrati novi ključ jer u tom slučaju stari podatci neće biti dostupni korisniku. Inženjer može izabrati migraciju sa stankom aplikacije ili bez stanke. Migracija sa stankom aplikacije je u pravilu jednostavnija izvedba. No, takva stanka je negativno korisničko iskustvo. Kompanije poput Google i Facebook uvijek koriste migracije bez stanke. Kako bi se migracija mogla napraviti bez stanke potrebno je nekoliko faza i verzija aplikacija. Često se takva migracija svodi na:

- izmjene sheme baze podataka, ali takve izmjene da su kompatibilne sa prethodnom verzijom aplikacije
- izmjene aplikacije koja omogućava zapis u izmijenjenu bazu podataka bez da utječe na prethodnu verziju aplikacije
- prijepis povijesnih podataka u novu shemu (*engl. backfill*)
- izmjene aplikacije tako da se ne zapisuju podatci u staru shemu baze podataka
- izmjene sheme baze podataka gdje se stare vrijednosti uništavaju.

Kod nekompatibilnosti programskog sučelja klijenta i poslužitelja dolazi prilikom objave nove aplikacije i promjenom sučelja na poslužitelju, to jest aplikaciji. Slika 3.8 prikazuje pojednostavljeni primjer korisničke interakcije s Fibonacci aplikacijom prilikom objave nove verzije aplikacije.

Korisnik prvo zahtjeva početnu HTML stranicu koja sadrži formu i klijentski Javascript kod koji određuje gdje će se zahtjev poslati nakon što korisnik upiše broj i pritisne gumb "Računaj". Na slici, prvi zahtjev je prema aplikacijskom sučelju (API) originalne verzije. No, drugi zahtjev upućen je novoj verziji aplikacije. Ukoliko nova verzija ima drugačije sučelje, takav zahtjev neće uspjeti te će korisniku biti poslana greška. Jedini način da web aplikacija ponovno proradi je tako da korisnik ručno osvježi (*engl. refresh*) web stranicu, što je negativno korisničko iskustvo.

Kako bi se to izbjeglo, potrebno je zadržati kompatibilnost programskog sučelja trenutne i nove verzije aplikacije. Nakon određenog vremena se može očekivati da će korisnici imati novu klijentsku verziju aplikacije te se kompatibilnost može maknuti. U praksi se zna i pitati korisnika za osvježavanje stranice.

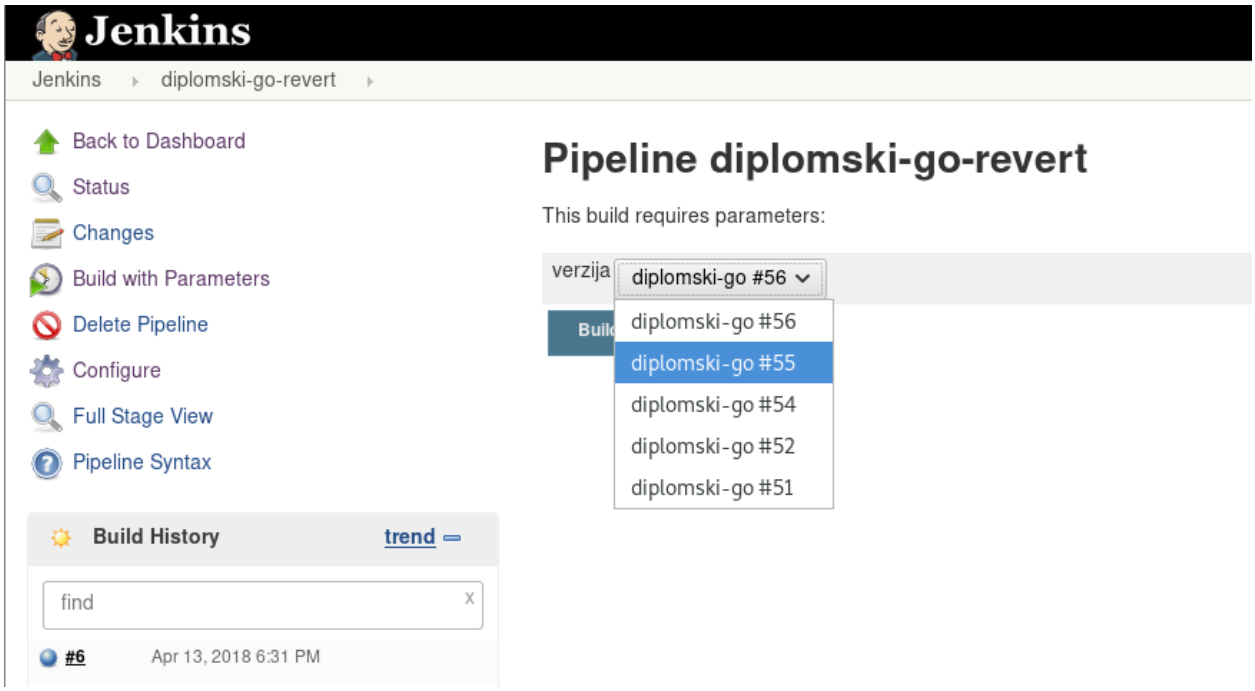


Slika 3.8: Objava nove verzije aplikacije

3.7 Vraćanje na staru aplikaciju

Kao što je rečeno u prethodnom poglavlju, svaka izmjena aplikacije može uzrokovati s neočekivanom pogreškom. Korištenjem testova jedinica i *end-to-end* testova povećana je vjerojatnost otkrivanja greške prilikom razvoja aplikacije. No, ukoliko novoobjavljena verzija aplikacije sadrži neku grešku, potreban je dobar sustav kako bi se takva pogreška mogla brzo otkloniti. Često je najbrže rješenje vraćanje na staru verziju aplikacije (*engl. version rollback*), gdje se neispravna verzija zamjenjuje s prethodno objavljenom, stabilnom verzijom. Zatim je potrebno ukloniti grešku u kodu te dodati prikladne testove nakon čega se može objaviti nova, ispravljena verzija aplikacije.

Proces vraćanja na staru verziju mora biti brz i intuitivan. U ovom radu korišten je parametrizirani Jenkins posao za vraćanje na prethodno objavljenu verziju, prikazan slikom 3.9 i kodom 3.21. Prilikom pokretanja takvog posla korisniku je dan padajući izbornik sa svim verzijama aplikacije. Nakon što korisnik izabere željenu verziju, Jenkins označava tu verziju aplikacije kao zadnju. Unutar trideset sekundi servis za menadžment pokreće novoizabranu verziju aplikacije i počinje ju posluživati. U tom trenutku verzija aplikacije s greškom se više ne poslužuje korisnicima.



Slika 3.9: Jenkins posao za vraćanje verzije aplikacije

```
pipeline {
   agent any
   stages {
      stage('Objava slike') {
         steps {
            script {
               docker.withRegistry('https://docker.io', 'dockerhub') {
                  def i = docker.image("sokac/fibonacci:${VERZIJA_NUMBER}")
                  i.push('latest')
               }
            }
         }
      }
   }
}
```

Programski kod 3.21: *Jenkinsfile* za vraćanje na prethodnu verziju

4. ZAKLJUČAK

Razvojem računalnog oblaka i aplikacija otvorenoga koda sve je veći broj tvrtki koje se bave razvojem aplikacija, a pogotovo web baziranih aplikacija. Uvođenje sustava za kontinuirani razvoj aplikacija omogućava bržu i konkurentniju izradu. Inženjeri koji rade s takvim sustavima dobivaju povratnu informaciju češće i brže te se smanjuje kontekstno prebacivanje koje smanjuje produktivnost. Samim time tvrtka kao cjelina postaje efektivnija i produktivnija. Moderne su tvrtke dokaz da korištenje takvih tehnologija ubrzava izradu novih funkcionalnosti. Tvrtke i projekti koji nemaju sustav za kontinuirani razvoj aplikacija u opasnosti su od konkurencije jer takvoj tvrtci treba puno više vremena i resursa za izgradnju iste funkcionalnosti. Na primjer, kompanija Facebook bila je u mogućnosti brže razvijati nove funkcionalnosti nego konkurentna tvrtka MySpace.

Korisnik ima na izbor velik broj komercijalnih i besplatnih, otvorenih rješenja. U ovome radu opisano je i isprogramirano samo jedno od rješenja koje se temelji na otvorenome kodu. Takvo rješenje može se koristiti za aplikacije otvorenog i zatvorenog (komercijalnog) koda u bilo kojem programskom jeziku te na bilo kojoj Unix platformi. Opisani će sustav vjerojatno zadovoljiti nove korisnike te projekte s manjim brojem inženjera. Povećanjem kompleksnosti projekta i broja inženjera potrebno je doraditi sustav kako bi i dalje bio brz, konkurentan i adekvatan za tvrtku. Na primjer, uvođenje sustava za plavo-zeleni razvoj (*engl. blue-green deployment*) omogućava pokretanje dvije verzije aplikacija u istome trenutku te slanje određenog postotka zahtjeva na jednu od tih verzija. Kako je ova tema izuzetno široka, nemoguće je projektirati idealno rješenje koje će zadovoljiti sve korisnike.

LITERATURA

[1] Pekka Abrahamsson, Outi Salo, Jussi Ronkainen, i Juhani Warsta. Agile software development methods: Review and analysis. *arXiv preprint arXiv:1709.08439*, 2017.

[2] Chuck Rossi, Elisa Shibley, Shi Su, Kent Beck, Tony Savor, i Michael Stumm. Continuous deployment of mobile software at facebook (showcase). U *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, stranice 12–23. ACM, 2016.

[3] Yasset Perez-Riverol, Laurent Gatto, Rui Wang, Timo Sachsenberg, Julian Uszkoreit, Felipe da Veiga Leprevost, Christian Fufezan, Tobias Ternent, Stephen J Eglen, Daniel S Katz, et al. Ten simple rules for taking advantage of git and github. *PLoS computational biology*, 12(7):e1004947, 2016.

[4] Stack Overflow, Stack Overflow Developer Survey 2017, 2017. dostupno na: `https://insights.stackoverflow.com/survey/2017` [5.3.2018.].

[5] S. Chacon i B. Straub. *Pro Git*. Apress, 2014.

[6] B. Firestine, Celebrating nine years of GitHub with an anniversary sale, 2017. dostupno na: `https://goo.gl/1pbVMh` [4.3.2018.].

[7] Martin Fowler i Matthew Foemmel. Continuous integration. *Thought-Works*, 122:14, 2006.

[8] Kent Beck. Embracing change with extreme programming. *Computer*, 32(10):70–77, 1999.

[9] John Ferguson Smart. *Jenkins: The Definitive Guide: Continuous Integration for the Masses*. O'Reilly Media, Inc., 2011.

[10] Nikhil Pathania. *Learning Continuous Integration with Jenkins*. Packt Publishing Ltd, 2016.

[11] Jenkis, Jenkins Plugins, 2018. dostupno na: `https://plugins.jenkins.io/` [6.3.2018.].

[12] Jeff Nickoloff. *Docker in action*. Manning Publications Co., 2016.

[13] Rafal Leszko. *Continuous Delivery with Docker and Jenkins*. Packt Publishing Ltd, 2017.

[14] Muhamad Fitra Kacamarga, Bens Pardamean, i Hari Wijaya. Lightweight virtualization in cloud computing for research. U *International Conference on Soft Computing, Intelligence Systems, and Information Technology*, stranice 439–445. Springer, 2015.

[15] Van Nam Nguyen. A comparative performance evaluation of web servers. Technical report, VNU-UET Technical Report, 2017.

[16] Netcraft, February 2018 Web Server Survey, 2018. dostupno na: https://news.netcraft.com/archives/2018/02/13/february-2018-web-server-survey.html [5.3.2018.].

[17] Will Reese. Nginx: the high-performance web server and reverse proxy. *Linux Journal*, 2008(173):2, 2008.

[18] Rahul Soni. Load balancing with nginx. U *Nginx*, stranice 153–171. Springer, 2016.

[19] Roy Fielding, Jim Gettys, Jeffrey Mogul, Henrik Frystyk, Larry Masinter, Paul Leach, i Tim Berners-Lee. Hypertext transfer protocol–http/1.1. Technical report, 1999.

[20] Alan AA Donovan i Brian W Kernighan. *The Go programming language*. Addison-Wesley Professional, 2015.

[21] Cypress, Cypress.io Documentation | Key differences, 2018. dostupno na: https://docs.cypress.io/guides/overview/key-differences.html#Debuggability [18.3.2018.].

[22] Mike Wacker, Just say no to more end-to-end tests, 2015. dostupno na: https://testing.googleblog.com/2015/04/just-say-no-to-more-end-to-end-tests.html [18.3.2018.].

[23] Tim Bray. The javascript object notation (json) data interchange format. 2017.

Sustav za kontinuirani razvoj aplikacija

Sažetak

Diplomski rad istražuje sustave kontunuiranog razvoja aplikacija te njihovu primjenu u tvrtkama i projektima. U teorijskome dijelu rada opisani su svi alati i procesi koji su potrebni za izgradnju takvog sustava. Teorijski dio započinje s usporedbom različitih sustava za reviziju koda, njihov povijesni razvoj te osnove korištenje Git sustava za reviziju koda. Zatim je opisan sustav za kontinuiranu integracija i alat Jenkins i sustav za upravljanje kontejnerima – Docker. Teorijski dio sadržava i osnove programskog jezika Go korištenog za izradu dijela sustava kontunuiranog razvoja. U praktičnome djelu je opisana arhitektura i implementacija web aplikacija za izračun fibonaccijevog broja. Zatim je takva aplikacija izgrađena pomoću sustava kontunuiranog razvoja aplikacije. U diplomskom radu razvijen je cijeli sustav za kontinuirani razvoj aplikacije koji se sadrži od preuzimanja novog koda aplikacije, pokretanja testova jedinica i *end-to-end* testova, stvaranja Docker slike i pokretanja iste te presmjeravanje prometa na novu aplikaciju. Razvijeni sustav je testiran pomoću Jmeter alata.

Ključne riječi: CI, Cypress, Docker, Git, Go, Golang, Jenkins, Kontinuirani razvoj aplikacija, Nginx

Continuous deployment

Abstract

This master thesis explores continuous deployment, its usage and impact on corporations and projects. The theoretical part starts by comparing different code revision tools, their historical development and covers Git basics. Furthermore, Jenkins, a tool for continuous integration, is described in details as well as Docker – a tool for managing containers. It also contains basics of Go programming language used in this thesis. The pratical part describes the architecture of a Web application for calculating fibonacci number. Such application is built and deployed by a continuous deployment system designed and programmed as part of this thesis. The system is tested using Jmeter tool.

Keywords: CI, Continuous deployment, Cypress, Docker, Git, Go, Golang, Jenkins, Nginx

ŽIVOTOPIS

Josip Šokčević rođen je 8. rujna 1990. godine u Vinkovcima te je tamo završio osnovnu i srednju tehničku školu, smjer mehatronika. Tijekom osnovnoškolskog i srednjoškolskog obrazovanja nastupao je na brojnim natjecanjima. Višestruki je prvak županije u matematici i informatici te ima brojne pehare s teniskih natjecanja. Josip se 2009. godine upisuje na Elektrotehnički fakultet u Osijeku, smjer računarstvo, kako bi usavršio svoje znanje u računarnoj znanosti. Studij prekida 2011. godine na godinu dana kako bi otišao u tvrtku The Backplane, Inc na poziciju inženjera programske podrške. Godine 2013. stječe status prvostupnog inženjera Računarstva te iste godine nastavlja studij upisom Sveučilišnog diplomskog studija Računarstva. Josip opet studij prekida 2014. godine kako bi otišao u tvrtku The Backplane, Inc. Godine 2015. odlazi u kompaniju Thumbtack, Inc gdje i dalje radi.

Josip je dobitnik nagrade Vidi Web Top 100 u kategoriji "Mediji i novosti" za stranicu blogeri.hr i Vidi Web Top 100 u kategoriji "Znanost, obrazovanje i kultura" za stranicu linuxzasve.com. Dobitnik je i rektorove nagrade za sudjelovanje u projektu ETFOS mobi s izradom Android aplikacije.

Josip Šokčević