Univerza v Ljubljani
Fakulteta za računalništvo in informatiko

Nina Krmavnar

Sprotna dostava programske opreme

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 dela:

Za podjetja, ki se ukvarjajo z razvojem programske opreme, je pomembno, da čim bolj skrajšajo čas, potreben za predajo novih izdaj v produkcijo. Kot rešitev tega problema se v zadnjem času predlaga ti. cevovodni sistem, ki avtomatizira postopke gradnje, testiranja in postavitve. V svoji nalogi proučite značilnosti tega pristopa in realizirajte ustrezno rešitev za podjetje, kjer ste zaposleni.

Mentor: izr. prof. dr. Viljan Mahnič

Izjava o avtorstvu diplomskega dela

Spodaj podpisana Nina Krmavnar, z vpisno številko **63070116**, sem avtorica diplomskega dela z naslovom:

Sprotna dostava programske opreme

S svojim podpisom zagotavljam, da:

- sem diplomsko delo izdelala 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 18. aprila 2015 Podpis avtorja:

Zahvaljujem se mentorju izr. prof. dr. Viljanu Mahniču za vso strokovno pomoč pri izdelavi diplomskega dela ter družini, prijateljem in sodelavcem za potrpežljivost in izkazano podporo.

Mojim najdražjim.

Kazalo

Povzetek

Glavni namen diplomske naloge je predstavitev enega izmed možnih pristopov k avtomatizaciji procesa sprotne dostave, tako da le-ta najbolj ustreza določenemu tipu aplikacije. V uvodnem delu so predstavljeni najpomembnejši razlogi za izbor teme skupaj z realnimi primeri, ki nazorno pokažejo, da je uporaba opisanega procesa v današnjem svetu skorajda obvezen dodatek v ohranjanju konkurenčnosti pri razvoju programske opreme. Poglavja, ki sledijo, predstavijo glavne karakteristike dostave programske opreme, začenši z upravljanjem konfiguracije in izvorne kode, kar je izredno pomembno pred izvedbo prve integracije. V nadaljevanju sledijo poglavja o (avtomatskem) testiranju, cevovodnem procesu postavitve ter postavitvi in dokončni izdaji programske opreme. V povezavi s procesom sprotne integracije je omenjen še integracijski strežnik Jenkins, ki ima izredno pomembno nalogo pri avtomatizaciji procesov in je poleg tega uporabljen tudi v praktičen delu diplomske naloge. Osrednji del pa je opis uporabe procesa sprotne dostave v praksi, od same integracije pa do končne izdaje v produkcijsko okolje, ki poteka sočasno z razvojem nove spletne aplikacije. Namen je pokazati največje prednosti pri uporabi celotnega procesa ter možne izboljšave, s katerimi bi se željenemu stanju čimbolj približali.

Ključne besede: sprotna dostava programske opreme, sprotna integracija, razvoj programske opreme, izdaje programske opreme.

Abstract

The main purpose of the thesis is the demonstration of one of the best possible approaches to an automated continuous delivery process as it relates to certain application types. In the introductory part, the main reason for choosing the subject is presented, along with a few examples of why nowadays - in order to keep pace with the competition - such an approach seems necessary. Following chapters discuss the basics of software delivery, starting with configuration and version control management, both necessary before the first integration process. Continuing discussion deals with (automated) testing, build pipeline process and - last but not least - deployment and application release. In connection with the continuous integration process, Jenkins is presented, an extensible open source continuous integration server, proved in practice as an important tool in the automation of any process that can or should be automated. The central part of the thesis is a presentation of the continuous software delivery process in practice, which is mainly oriented towards front-end application development and consequently to its integration and final release. With the intention of moving towards the desired state, the advantages of such practices and possible future improvements are explored.

Keywords: continuous software delivery, continuous integration, software development, application releases.

Poglavje 1

Uvod

Ko po naročilu neke stranke razvijamo programsko opremo, se na dnevni ravni srečujemo s težavami pri hitrih izdajah v produkcijsko okolje. Zaradi izjemno pomembnih povratnih informacij na spremembe v aplikaciji, je potrebno stranki novosti čimprej predstaviti, a izdaja nove verzije aplikacije v produkcijsko okolje običajno traja več dni, kar v veliki večini primerov ni sprejemljivo. Kako ukrepati? Take in podobne situacije, ki se v praksi pojavljajo prepogosto, niso neizogibna posledica v procesu razvoja programske opreme. So zgolj jasen znak, da je nekaj v omenjenem postopku narobe.

Kot opisujeta avtorja v [1], se je v poznih devetdesetih za velik uspeh pri razvoju programske opreme štelo že to, da je bilo možno nove spremembe pri določenem programju postaviti v produkcijsko okolje (*deploy to production*) enkrat dnevno, običajno v nočnih urah. Kasnejša možnost redne postavitve v produkcijsko okolje je prinesla marsikatere prednosti: popravki programja niso nekoristno čakali do naslednje postavitve, možen je bil izredno hiter odziv na težave, ki so se pri tem pojavile, hkrati pa je vse to vodilo k občutno globljemu in boljšemu odnosu med naročniki, izvajalci ter posledično končnimi uporabniki.

Izdaja programja (*software release*) mora biti hiter in ponovljiv proces. Dandanes mnoga podjetja izdajajo svoje izdelke večkrat dnevno, kar je izvedljivo tudi v primeru zelo zapletenih sistemov ali visoko kompleksnih projektov. Pojavlja se pomembno vprašanje - *Koliko časa bi potrebovali za izdajo spremembe v aplikaciji, ki vključuje popravek le ene vrstice izvorne kode?* Sprotna integracija (*continuous integration*) je temelj za ta pristop in ohranja celotno ekipo složno že samo s tem,

da omogoči popolno odstranitev zakasnitev, ki so posledica težav z integracijo. Kljub vsemu pa je sprotna integracija le prvi korak. Programje, ki je bilo uspešno integrirano v glavni tok izvorne kode, kljub vsemu še vedno ni programje, ki bi bilo postavljeno v produkcijsko okolje in opravljalo svoje delo. Da dosežemo nemoten potek procesa sprotne dostave (*continuous delivery*), moramo v to vložiti ogromno truda, a prednosti, ki jih s tem pridobimo, so neprecenljive. Dolge in naporne izdaje programske opreme tako postanejo preteklost. S stališča strank to pomeni, da od ideje do integrirane nove funkciolnosti v produkcijsko okolje preteče minimalen čas, in posodobljena programska oprema je brez prekinitev ves čas na voljo za uporabo. Najbolj pomembno pa je verjetno to, da s tem pristopom odstranimo enega največjih faktorjev stresa pri razvoju programske opreme. Nihče se namreč med vikendi ne želi ukvarjati z naporno in stresno izdajo nadgradenj programske opreme, ki mora seveda biti ves ta čas v delujočem stanju.

V praksi se pojavljata dve skrajnosti postavitve delujoče programske opreme v produkcijsko okolje. Pri prvi čas cikla merimo v tednih ali mesecih, pojavljajo se tudi časi, ki jih lahko merimo celo v letih, postopek izdaje pa definitvno ni ponovljiv, niti zanesljiv. Večkrat se zgodi, da gre pri celotnem postopku za popolnoma ročno integracijo in postavitev že na testno ter pred-produkcijsko okolje, kaj šele na produkcijsko. Po drugi strani pa se pri enako zahtevnih in kompleksnih projektih čas cikla lahko zmanjša na nekaj ur ali celo minut v primeru kritičnih popravkov. S pomočjo avtomatskega, ponovljivega in zanesljivega procesa je to možno, saj s tem vse ročne korake, ki pripeljejo do uspešne postavitve v produkcijsko okolje, nadomestimo z avtomatskimi, ki za izvedbo potrebujejo le “pritisk” na gumb. Cilj je poskrbeti, da je dostava delujoče programske opreme iz rok razvijalcev v produkcijsko okolje zanesljiv, predvidljiv, viden in popolnoma avtomatiziran proces z dobro znanimi, merljivimi tveganji. Ena od največjih prioritet pri razvoju je tako zadovoljiti kupca s hitro in sprotno dostavo kvalitetne programske opreme.

Namen diplomske naloge je pokazati, kako čimbolje povezati teme, kot so upravljanje konfiguracije, nadzor izvorne kode, načrtovanje izdaj programske opreme, revizija, skladnost in integracija v avtomatizacijo gradnje, testiranja ter procesa postavitve, ter jih uspešno pretvoriti v skupek korakov, ki pripeljejo do uspešno izvedenega procesa sprotne dostave. Vse te aktivnosti so izredno pomembne - nekateri menijo, da so pri razvoju programske opreme po prioriteti celo na drugem

mestu, takoj za razvijanjem ([1]). Glede na izkušnje pa zahtevajo predvsem veliko vloženega časa in truda ter so kritične za uspešno izvedbo procesa sprotne dostave programske opreme. V nasprotnem primeru se lahko pojavijo neželene posledice, kot so visoki stroški, ki lahko celo presežejo začetni vložek, ki je bil sprva mišljen zgolj za razvoj, zato del diplomske naloge vključuje tudi strategije, kako se takim situacijam izogniti. Namen je prikazati, kako se vse te aktivnosti – upravljanje konfiguracije, avtomatsko testiranje, sprotna integracija in postavitev, upravljanje s podatki, posameznimi okolji ter izdajo same aplikacije - med seboj dopolnjujejo.

Poglavje 2

Dostava programske opreme in sprotna integracija

2.1 Problemi pri dostavi programske opreme končnim uporabnikom

Dejstvo je, da pri uspešni dostavi programske opreme končnim uporabnikom ne gre le za dober izbor pristopov v procesu razvoja. Veliko vlogo pri tem igrajo še drugi vidiki, ki so večkrat nepremišljeno odrinjeni na stran, brez njih pa je nemogoče poskrbeti za zanesljivo in hitro izdajo programske opreme brez nezaželenih tveganj. Zanimivo je videti, kaj se zgodi v trenutku, ko so vse zahteve dokončno opredeljene, rešitve oblikovane, fazi razvoja in testiranja pa sta pri koncu. Kako se vsi ti koraki povežejo v smiselno celoto in funkcionirajo tako dobro, da je celoten proces od razvoja do izdaje programske opreme kar se da zanesljiv in učinkovit. Vzorec, ki nosi odgovor na vsa ta razmišljanja, imenujemo cevovodni proces postavitve (*deployment pipeline*), in je podrobno predstavljen v [1]. Gre za avtomatiziran postopek implementacije gradnje, postavitve, testiranja in procesa izdaje neke programske opreme.

Vsaka sprememba v aplikaciji (pa naj bo to sprememba konfiguracije, izvorne kode, okolja ali podatkov) sproži kreiranje nove instance cevovoda. Precejšen delež v pomembnosti samega procesa nosijo testi, ki potrdijo, da je kandidat za izdajo za to tudi primeren. Vsak uspešno prestan test nam da še večje zaupanje v to, da

bo kombinacija vseh teh delov zanesljivo in dobro delovala tudi kot celota. Ko so uspešno izvedeni vsi testi, je programska oprema pripravljena na izdajo. Namen cevovodnega procesa postavitve programske opreme je prvenstveno, da je vsak posamezen korak v tem procesu viden vsem sodelujočim, kar izboljša medsebojno sodelovanje. Poleg tega zagotavlja boljše povratne informacije, ki posledično do vseh vpletenih pridejo hitreje, kar omogoči takojšnjo odpravo težav, ki se tekom razvoja pojavijo. Najpomembnejša pridobitev pa je zagotovo, da ima celotna ekipa možnost postavitve in izdaje katerekoli verzije programske opreme na katerokoli okolje s pomočjo popolnoma avtomatiziranega procesa.

2.2 Pogosti primeri slabe prakse in možne izboljšave

Ročna postavitev programske opreme. Postavitev večine današnjih aplikacij je, ne glede na njihov obseg, zahteven postopek in vključuje veliko podkorakov. Še vedno je pogost pojav, da se ta postopek izvaja ročno, kar pomeni, da je vsak od njih izveden s strani nekega posameznika oziroma dela ekipe. Vsak tak korak, izveden ročno, je izpostavljen človeškim napakam. Glavni pokazatelji omenjene slabe prakse so: obsežna dokumentacija, ki opisuje korake za izvedbo postavitve; pogosti klici s pojasnili, zakaj je prišlo do težav s postavitvijo na dan izdaje; pogosti popravki v postopku izdaje, ki nikoli ne traja le nekaj minut. Vsi ti pokazatelji so zadosten razlog, da bi razvojne ekipe morale stremeti k popolnoma avtomatiziranemu procesu postavitve. Ročna v takem primeru ostaneta le še dva koraka: izbor verzije in okolja ter pritisk na gumb za postavitev. Prednosti so sledeče: dobimo ponovljiv in zanesljiv postopek, ki nam prihrani čas z odkrivanjem napak pri postopku postavitve; ni potrebe po vzdrževanju dokumentacije z opisi posameznih podkorakov, medtem ko so skripte za postavitev vedno posodobljene in hkrati služijo še za dokumentacijo; ni zanašanja na točno določenega strokovnjaka, ki te postopke obvlada, in na to, da je v trenutku, ko gre nekaj narobe, dosegljiv; prihranimo čas in denar, saj je avtomatiziran postopek hitrejši, cenejši ter preprost za testiranje.

Prva postavitev v produkcijsko okolje šele potem, ko se razvoj zaključi.

Glavni pokazatelji: odgovorni za postavitev se z novo izdajo prvič srečajo, ko se ta izda v pred-produkcijsko okolje, še posebej, če ne sodelujejo pri samem razvoju; pred-produkcijsko okolje sploh nikoli ni bilo vzpostavljeno, saj je sama postavitev okolja predraga in zaradi tega dostopnost omejena, ali pa se s tem enostavno nihče ni ukvarjal; sodelovanja med ekipami, odgovornimi za celoten postopek, je zelo malo ali nič. V primeru, da postopek postavitve vsaj v testno okolje nikoli ni bil preizkušen, v ključnem trenutku pogosto prihaja do napak. Dokumentacija je nepopolna in ekipa, ki naj bi poskrbela za postavitev, lahko le ugiba, kaj je bil namen razvojne ekipe. Vse to prinese popravke v zadnjem trenutku, stresne situacije, delo pod ogromnim pritiskom, kar razumevanje in izredno pomembno uspešno sodelovanje med ekipami le še poslabša. Postavitev je v predvidenem času praktično neizvedljiva. Integracija testiranja, postavitve in aktivnosti v zvezi z izdajo v sam proces razvoja nepredstavljivo zmanjša tveganje za zgoraj omenjene zaplete. Ko vse to enkrat postane del vsakdana, je trenutek izdaje v produkcijsko okolje veliko manj stresen, kot bi bil običajno, saj je scenarij preizkušen in uspešno izveden že neštetokrat in vse to je zagotovilo, da gre le malo stvari lahko narobe.

Ročno upravljanje konfiguracije produkcijskega okolja. Glavni pokazatelji: postavitev v pred-produkcijsko okolje je uspela že neštetokrat, medtem ko postavitev v produkcijsko okolje ne uspe; različni pripadniki gruče se obnašajo drugače, npr. ena veja zdrži manjše obremenitve kot druga ali pa potrebuje več časa za procesiranje zahtevkov; priprava okolja za izdajo traja zelo dolgo časa; hitra sprememba konfiguracije na tisto izpred nekaj dni je nemogoča; na strežnikih v gruči so, nenamenoma, različne verzije operacijskih sistemov, ostale infrastrukture, knjižnic, ipd. Tako kot vsa izvorna koda in skripte za postopek postavitve bi morala biti tudi konfiguracija posameznih okolij pod nadzorom različic, kar zagotavlja hitro in nemoteno pripravo novega okolja za postavitev in izdajo aplikacije, kadarkoli je to potrebno. To lahko dosežemo tudi tako, da dovolimo upravljanje in spreminjanje konfiguracije teh okolij le preko avtomatiziranih procesov.

2.3 Želeni cilji in njihove prednosti

Glavni cilj je dostava visoko kakovostne, dragocene programske opreme na najbolj učinkovit, hiter in zanesljiv način. Za dosego teh ciljev (nizek čas cikla in

visoka kakovost), je pomembno pogosto izvajanje avtomatiziranega procesa izdaje programske opreme.

- *Avtomatiziran proces.* Če faze od razvoja, gradnje, postavitve, testiranja do izdaje aplikacije niso avtomatizirane, niso niti ponovljive, kar pomeni, da vsaka nova izvedba lahko prinese drugačen rezultat, možnost napak je večja in skorajda nemogoče je slediti, kaj se je med postopkom zares dogajalo.

- *Pogosto izvajanje.* Če so izdaje pogoste, je razlika med eno in drugo izredno majhna, kar zmanjša tveganja in olajša vrnitev v prejšnje stanje v primeru kakršnihkoli težav. Vse to prinaša tudi hitrejše povratne informacije, ki so zelo pomembne.

Poznamo tri vrste meril, ki klasificirajo povratne informacije kot izredno pomemben del razvoja: vsaka sprememba naj sproži proces pridobivanja povratnih informacij, pridobivanje povratnih informacij mora biti kar se da hitro in ekipa, ki skrbi za dostavo, mora dobiti te povratne informacije ter se nanje tudi odzvati.

Vsaka sprememba naj sproži proces pridobivanja povratnih informacij. Vsakič, ko pride do spremembe v izvorni kodi, je pomembno, da se sproži proces gradnje (*build process*) in testiranja. Da imamo lahko vse to pod nadzorom, morata biti omenjena postopka avtomatizirana. Praksi gradnje in testiranja programske opreme ob vsaki spremembi izvorne kode pravimo sprotna integracija. Poleg sprememb v izvorni kodi, so pomembne so tudi spremembe v konfiguraciji okolij, v okolju samem ter v podatkih; vsaka od teh sprememb mora prav tako sprožiti omenjen avtomatiziran proces. Postopek pridobivanja informacij vključuje testiranje vsake spremembe s pomočjo avtomatiziranega postopka, ki gre toliko v podrobnosti kolikor le mogoče. Glede na sistem, ki je v uporabi, se načini testiranja seveda razlikujejo, a vsi morajo vključevati vsaj prevajanje izvorne kode, izvedbo testov enot (*unit tests*), preverjanje izpolnjevanja kriterijev kakovosti, izvedbo nefunkcijskih in funkcijskih testov sprejemljivosti, izvedbo raziskovalnega testiranja ter predstavitev aplikacije strankam in izbranim uporabnikom.

Pridobivanje povratnih informacij mora biti kar se da hitro. Hitro pridobivanje povratnih informacij je pogojeno predvsem z avtomatizacijo. Takoj, ko je le en postopek potrebno izvršiti ročno, se proces podaljša, saj je potrebno počakati,

da oseba, odgovorna za to delo, opravi nalogo. To vzame več časa, možnost napak se poveča in ni mogoče natančno preveriti, če so bili vsi koraki, pomembni za nalogo, pravilno izvedeni.

Ekipa, ki skrbi za dostavo, mora dobiti povratne informacije (feedback) in se nanje tudi odzvati. Pomembno je, da so vsi, ki so vpleteni v dostavo programske opreme – razvijalci, testerji, skrbniki podatkovnih baz, itd. - ves čas vključeni v proces pridobivanja povratnih informacij. Brez pomena je, če se nanje ne odzovejo, vse to pa zahteva visoko disciplino in dobro načrtovanje. Vsaki informaciji mora biti določena tudi prioriteta, ki jo je potem potrebno dosledno upoštevati.

Glavna prednost pristopa, ki je osrednja tema diplomske naloge, je doseči proces izdaje, ki je ponovljiv, zanesljiv ter predvidljiv. Posledično prinese krajše čase ciklov in omogoča hitro implementacijo novih funkcionalnosti ter popravkov, ki so potem takoj na voljo uporabnikom. Hkrati s tem pridobimo še:

- močnejše, bolj samozavestne in povezane razvojne ekipe,
- odpravo stresnih situacij,
- fleksibilnost pri postavitvi v novo okolje, kjerkoli in kadarkoli, ter
- izpopolnjevanje do popolnosti pri uporabi postopka v praksi.

2.4 Načela pri dostavi programske opreme

Ponovljiv in zanesljiv postopek izdaje programske opreme je prvo načelo in ga je mogoče doseči s tem, da avtomatiziramo, kar se le da, in da hranimo vse, kar potrebujemo za gradnjo, postavitev, testiranje in izdajo aplikacije v nadzoru različic (*version control*). Postopek postavitve programske opreme vključuje tri korake: oskrbovanje in upravljanje okolja, v katerem bo tekla aplikacija (konfiguracija strojne opreme, sama programska oprema, infrastruktura in zunanje storitve), namestitev pravilne verzije aplikacije v to okolje ter konfiguracija same aplikacije, vključno z vsemi podatki in stanji, ki jih potrebuje.

Naslednje načelo, avtomatizacija vsega, je pojem, ki se morda sliši nemogoč za izvedbo v praksi, a je v resnici seznam postopkov, ki ne morejo biti avtoma-

tizirani, precej manjši, kot mislijo mnogi. V splošnem mora biti proces dostave programske opreme avtomatiziran vse do točke, ko so potrebna le še specifična človeška navodila oziroma sprejetje neke odločitve. V interesu vseh sodelujočih na projektu bi moralo biti, da se avtomatizira, kar je le možno. Najbolj pogost razlog proti je, da se vzpostavitev tega procesa zdi kar nekako zastrašujoče opravilo. A avtomatizacija je predpogoj za uveljavitev cevovodnega procesa postavitve, saj se le tako lahko zagotovi, da bo rezultat na koncu tisto, kar res želimo.

Tretje načelo pravi - vse, kar potrebujemo za gradnjo, postavitev, testiranje in izdajo aplikacije, bi moralo biti hranjeno v eni izmed oblik nadzora različic. To vključuje vse potrebne dokumente, skripte, izvorno kodo, konfiguracijske datoteke, dokumentacijo, knjižnice, itd. Potrebno je, na primer, zagotoviti novemu članu ekipe, da na svoji delovni postaji naredi kopijo verzioniranega repozitorija ter požene en sam ukaz, ki mu omogoči gradnjo in postavitev aplikacije v katerokoli okolje, vključno s svojim razvojnim. Integracija je večkrat precej zahteven proces, kar najlažje rešimo tako, da integriramo bolj pogosto. Podobno je s testiranjem – če se problem pojavi tik pred izdajo, ga ne preložimo na kasneje, temveč testiramo pogosteje že vse od začetka razvoja. Vse to velja za vse faze od razvoja do končne izdaje programske opreme. Postopno delo na omenjenem pristopu lahko prinese ogromne koristi.

Naslednje načelo govori o tem, kako je zgodnje odkrivanje napak izredno pomembno, predvsem pa stane manj, ker se jih lahko odpravi še preden dejansko pridejo do uporabnika. Pri odpravi napak je potrebno upoštevati prioriteto, saj le tako te ne gredo v pozabo zaradi zasedenosti razvijalcev z drugimi nalogami.

Pri koncu razvoja nove funkcionalnosti večkrat rečemo, da je le-ta končana, a v resnici je nova funkcionalnost zares dokončana, ko prinaša neko uporabno vrednost uporabnikom. Pravimo pa tudi, da končano lahko pomeni že izdano v produkcijskem okolju. V splošnem, uporaba ocen za določitev časa, potrebnega za dokončanje neke naloge, lahko vodi do obtoževanj in kazanja s prstom, ko se izkaže, da ta ni bila najbolj točna. Načelo ima zanimivo posledico – da je nekaj končano, ni v moči le ene osebe, temveč celotne ekipe, ki skrbi za dostavo programske opreme. Zato je toliko bolj pomembno, da vsi člani medsebojno sodelujejo že vse od začetka razvoja. Navsezadnje ekipa uspe ali ne le kot ekipa in ne kot posameznik. Ko gre nekaj narobe, se večkrat zgodi, da člani ekipe porabijo več časa

za iskanje krivca, kot za dejansko odpravo napake. Potrebno je torej vzpodbujati boljše in kakovostnejše sodelovanje med vsemi vpletenimi v dostavo programske opreme z namenom izdajanja dragocenejšega programja uporabnikom hitreje in bolj zanesljivo.

2.5 Konfiguracija programske opreme

Pojavlja se mit ([1]), ki pravi, da spreminjanje konfiguracije predstavlja manjše tveganje kot spreminjanje izvorne kode. V resnici pa je v primeru omogočenega dostopa do obeh enako lahko ustaviti sistem s spreminjanjem enega ali drugega. Pri spreminjanju izvorne kode smo nekoliko zaščiteni pred samim sabo, če ne drugače, s pomočjo prevajalnika ali pa avtomatskih testov. Po drugi strani pa je večina informacij o konfiguraciji podana v prosti obliki in netestirana, kar lahko z določeno napačno spremembo predstavi problem šele v fazi, ko aplikacija že teče. Uporaba konfiguracije sama po sebi ni nevarna, a poskrbeti je treba, da se z njo upravlja previdno in konsistentno.

Konfiguriranje aplikacije je najbolje izvesti med samim postopkom postavitve. Na primer, če je konfiguracija za čas delovanja aplikacije shranjena v podatkovni bazi, je pomembno poskrbeti, da se podatki za povezavo z bazo podajo že ob času postavitve, da je potem vse potrebno na voljo takoj, ko se aplikacija zažene. Ne glede na način, ki ga izberemo, je dobro upoštevati, da se za konfiguriranje aplikacije oziroma okolja, v katerem živi, skozi celoten proces ves čas uporablja isti mehanizem. Vedno to ni mogoče, a takrat ko je, to pomeni, da spremembo parametrov konfiguracije lahko uredimo na enem mestu.

2.6 Sprotna integracija

Uporaba sprotne integracije v praksi se za dobro delovanje zanaša na določene predpogoje, ki morajo biti izpolnjeni.

- *Nadzor različic.* Pri vsakem projektu, pa naj bo še tako majhen, je pomembna uporaba sistema za nadzor različic, in vse, kar je kasneje del tega projekta, mora biti hranjeno tam (od izvorne kode, testov, do skript v zvezi s podatkovnimi bazami, skript za gradnjo in postavitev, ...).

- *Avtomatiziran proces gradnje.* Ne glede na mehanizem, ki se za to uporablja, mora biti vsakemu članu ekipe ali zgolj računalniku omogočeno, da izvede gradnjo, teste in postopek postavitve v neki avtomatizirani obliki preko ukazne vrstice. Kljub temu, da razna razvojna orodja omogočajo poganjanje takšnih in drugačnih postopkov direktno iz orodja samega, je pomembno, da je za začetek vsako tako skripto možno uspešno zagnati iz ukazne vrstice, šele kasneje z uporabo dodatne programske opreme.

- *Strinjanje ekipe.* Celoten proces ne bo dobro deloval, če hkrati nimamo podpore in zaupanja razvojne ekipe, ki mora vpeljavo določenega postopka podpirati in pri tem upoštevati pomembnost maksimalne discipline, ki obsega, na primer, večkratno pošiljanje čim manjših sprememb v sistem za nadzor različic ter razumevanje prioritet pri delu v primeru, da neka sprememba ustavi delovanje aplikacije, ko je najbolj pomembno to težavo takoj odpraviti.

2.7 Predpogoji za uporabo

Sprotna integracija ne bo sama po sebi izboljšala trenutnega procesa gradnje na projektu. Da bo postopek kar se da učinkovit, je dobro že pred samim začetkom poskrbeti, da bodo določene prakse razumljive in posledično redno upoštevane.

- *Sprotno pošiljanje sprememb v sistem za nadzor različic.* Vsaj enkrat na dan, še bolje pa večkrat. Spremembe so tako razdeljene na manjše enote, ki jih je lažje razumeti in manj verjetno je, da bi povzročile nedelovanje aplikacije, po drugi strani pa je, če se to kljub vsemu zgodi, veliko lažje napako najti in jo odpraviti. Na ta način se bolj verjetno izognemo tudi konfliktom z izvorno kodo ostalih razvijalcev, hkrati pa je v primeru kakšnih nezaželenih težav, kot je nenameren izbris samih datotek ali le sprememb znotraj njih, izguba že opravljenega dela manjša.

- *Zbirka celovitih avtomatskih testov.* Tudi brez testov je možno uspešno ponavljati postopek sprotne integracije, a dobrodošlo je, da z njihovo pomočjo povečamo zaupanje v to, da naša aplikacije dejansko deluje, kot mora. Najbolj pomembni testi, ki jih je dobro poganjati ob vsakem postopku sprotne

integracije, so testi enot, testi komponent ter testi sprejemljivosti (*acceptance tests*). Testi enot so namenjeni testiranju obnašanja posameznih enot aplikacije v popolni izolaciji od preostalih in se lahko uporabljajo tudi brez dejanskega delovanja celotne aplikacije. Namenjeni so testiranju obnašanja brez uporabe podatkovne baze, datotečnega sistema ali omrežja, in izredno pomembno je, da so hitri. Testi komponent so posvečeni testiranju obnašanja večih komponent in običajno za poganjanje ne potrebujejo aplikacije v delovanju. Za razliko od testov enot, lahko uporabijo podatkovno bazo, datotečni ali kak drug sistem, in se običajno izvajajo dlje. Testi sprejemljivosti pa so namenjeni testiranju funkcionalnih in nefunkcionalnih kriterijev sprejemljivosti, in se običajno uporabljajo med delovanjem aplikacije, v okolju, ki je primerljivo produkcijskemu, trajajo pa najdlje.

- *Kratek postopek gradnje in testiranja.* V primeru (pre)dolgih procesov nihče več ne bo upošteval dogovorov glede gradnje, testiranja in ostalih korakov sprotne integracije, celotna procedura pa bo postala neuporabna, saj časa za čakanje na izvedbo le-teh enostavno ni. Omenjeni postopki ne bi smeli vzeti več kot pet minut; idealen čas je sicer okoli devetdeset sekund, kar z današnjimi orodji ne bi smelo predstavljati prevelikih težav.

2.8 Integracijski strežnik Jenkins

Jenkins je strežnik, ki se uporablja za sprotno integracijo, in omogoča avtomatsko nadzorovanje izvorne kode v repozitorijih, gradnjo programske opreme in izvajanje testov ([12]). Sama namestitev in že vključena podpora za uporabo gruč omogočata preprost začetek pri izboljševanju kakovosti programske opreme ali pa le dodajanje nadzora za avtomatizacijo dnevnih nalog. Vključuje obsežno knjižnico z več kot 300 vtičniki, ki se lahko uporabljajo za gradnjo, testiranje, beleženje, analizo in še veliko drugih nalog, ki so pomembne za dostavo kakovostne programske opreme. Integracijski strežnik Jenkins je odprtokodno orodje za sprotno integracijo, napisano v programskem jeziku Java.

Ponuja storitve sprotne integracije za razvoj programske opreme in je strežniško osnovan sistem, ki teče na strežniku, kot je na primer Apache Tomcat, ter deluje po načelu zahteva-odgovor. Podpira tudi orodja za upravljanje konfiguracije pro-

Slika 2.1: Logotip integracijskega strežnika Jenkins.

gramske opreme, npr. AccuRev, CVS, Subversion, Git, Mercurial, in še mnoga druga, ter zna poganjati projekte, osnovane na tehnologijah Apache Ant in Apache Maven, prav tako dobro kot skripte, namenjene izvrševanju v ukaznih lupinah, ali pa "Windows batch"ukaze. Gradnje so lahko zagnane na različne načine, vključno z avtomatskimi sprožilci, ki nadzorujejo spremembe v določenem repozitoriju nekega sistema za nadzor različic preko periodičnih opravilnih mehanizmov, na osnovi drugih uspešno zaključenih nalog ali ob zahtevku za določen internetni naslov za gradnjo.

2.9 Uporaba sprotne integracije v praksi

Bistvene prakse, ki jih je potrebno/dobro upoštevati:

- *Prepovedano pošiljanje sprememb v nadzor različic v primeru nedelovanja aplikacije.* Prioritetna naloga v takem primeru je najprej odpraviti težavo, ki je povzročila nedelovanje aplikacije. S pošiljanjem dodatnih sprememb v nadzor različic se iskanje težave le oteži in posledično traja dlje, da se leta odpravi. Z upoštevanjem pravila si olajšamo razvoj in posledično s tem poskrbimo, da je proces gradnje čimvečkrat uspešno izveden, aplikacija pa ves čas v stanju delovanja.

- *Pred pošiljanjem sprememb v nadzor različic naj se vsi testi poženejo lokalno ali s pomočjo integracijskega strežnika.* Gre pravzaprav za nekakšen test racionalnosti naših misli, ki hkrati poskrbi, da tisto, kar verjamemo, da deluje, resnično deluje. Ni namreč vedno nujno, da težave povzročijo le spremembe enega člana ekipe, lahko pride do težav pri kombinaciji sprememb z večih strani, in s tem se le zavarujemo, da odkrijemo pomanjkljivosti, še preden steče proces intregacije.

- *Pred nadaljevanjem preverimo stanje izvedenih testov.* Vsak posameznik je po pošiljanju sprememb v nadzor različic odgovoren za rezultat, ki ga kasneje vrne proces gradnje. V primeru, da se proces ne izvede, kot je pričakovano, mora odgovorna oseba poskrbeti, da se stanje ponovno normalizira. Šele potem je smiselno nadaljevati s preostalimi nalogami.

- *Procesa gradnje, ki ni uspel, nikoli ne pustimo v takem stanju.* Treba se je zavedati dejstva, da tako dejanje lahko povzroči precejšnje težave preostalim članom ekipe, še posebej, če to pomeni, da se mora naslednji dan s tem spopasti nekdo, ki za to sploh ni odgovoren. Boljši način je morda ta, da si pred zadnjim dnevnim pošiljanjem sprememb v nadzor različic vzamemo dovolj časa, kar pomeni, da omenjene situacije lahko rešimo še v okviru normalne ure za odhod domov, v nasprotnem primeru pa raje vse to prihranimo za naslednji delovni dan.

- *Vrnitev na prejšnjo verzijo mora biti vedno omogočena.* Pričakovati, da bo vsaj enkrat določen član ekipe odgovoren za neuspešno izveden proces gradnje, je povsem realno. V primeru bolj zahtevnih projektov se take stvari večkrat dogajajo tudi dnevno, čeprav jih z marsikaterim pristopom lahko uspešno preprečimo. Včasih pa se vendarle zgodi, da je težava pri določeni spremembi večja, kot je bilo sprva pričakovati. V takem primeru je najbolje vrniti aplikacijo v prejšnje stanje (*rollback*) in se podrobno posvetiti spremembi, ki je povzročila težave. S tem preostalim članom omogočimo nemoteno nadaljnje delo, sami pa v miru raziščemo okoliščine, ki so pripeljale do nastale situacije.

- *Čas, dovoljen za reševanje težav pred vrnitvijo na prejšnjo verzijo aplikacije.* Običajno govorimo o približno desetih minutah, in če v tem času težava ni odpravljena, je dobro poskrbeti za delujoče stanje z vrnitvijo na prejšnjo verzijo.

- *Neuspelih testov ne ignoriramo.* Najbolj pomembno je ugotoviti, kaj je šlo narobe, nato poskusiti z odpravo težave, šele po tem lahko določene teste izločimo iz postopka, vendar samo v primeru, da ugotovimo, da test ne ustreza več zahtevani funkcionalnosti.

- *Razvoj, ki temelji na testih.* Hitre povratne informacije so možne le z dobro pokritostjo avtomatskih testov, kar je najpomembnejši izid sprotne integracije. To v praksi pomeni, da se za določeno funkcionalnost najprej ustvari test, ki prikazuje želeno delovanje kode, šele nato se začne z dejanskim razvojem funkcionalnosti.

- *Uporaba ekstremnega programiranja.* Ekstremno programiranje (*extreme programming*, [14]) je v kombinaciji s sprotno integracijo zelo učinkovito. Gre za metodologijo razvoja programske opreme, katere namen je izboljšanje kakovosti in odzivnosti na spremembe zahtev stranke. Zagovarja pogoste izdaje programske opreme v kratkih razvojnih ciklih.

2.10 Porazdeljeni sistemi za nadzor različic

Naraščanje porazdeljenih sistemov za nadzor različic (*distributed version control systems, DVCS*) spreminja način sodelovanja med ekipami. Njihova glavna značilnost je ta, da vsak repozitorij vsebuje celotno zgodovino projekta, kljub temu pa ponujajo še dodatno funkcionalnost, ki jih razlikuje od preostalih sistemov, in sicer, da gredo v primeru sprememb te najprej na lokalni repozitorij, preden jih je možno poslati na ostale. Prav tako pridejo vse posodobitve najprej iz ostalih repozitorijev na lokalnega, šele nato se posodobi tudi lokalna delovna kopija. Potrebno je izpostaviti, da je možna uporaba običajnega modela v nadzoru različic, in prav tako uspešno vpeljati sprotno integracijo z uporabo enega izmed porazdeljenih sistemov za nadzor različic (Git, Mercurial, ...). Eno izmed vej na repozitoriju poimenujemo "master", strežnik za sprotno integracijo pa sproži določen postopek vsakič, ko zazna spremembo na omenjeni veji. V splošnem v svetu računalniškega programiranja, porazdeljen nadzor revizij, poznan tudi pod imenom porazdeljen nadzor različic ali decentraliziran nadzor različic, omogoča veliko razvijalcem programske opreme delo na danem projektu, ne da bi si za to morali deliti skupno omrežje. V diplomski nalogi smo za te namene uporabili Git ([2]).

Poglavje 3

Cevovodni proces postavitve in dokončna izdaja aplikacije

3.1 Splošno

Vsaka sprememba aplikacije gre na poti do dejanske izdaje skozi zapleten proces, ki mu pravimo cevovodni proces postavitve (*deployment pipeline*). Ta proces vključuje gradnjo programske opreme z vsemi podprocesi gradenj od testiranja do postavitve. Cevovodni proces postavitve omenjeni postopek modelira, sam pa je del tako sprotne integracije, kot tudi orodja za upravljanje izdaj, kar omogoča spremljanje ter nadzor pri napredku ob vsaki spremembi, ko gre le-ta od nadzora različic preko množice testov in postavitev do izdaje in posledično rok uporabnikov.

Za dosego zavidljivega stanja s pomočjo cevovodnega procesa postavitve, je potrebno avtomatizirati zbirke testov, ki presojajo, če je kandidat za izdajo za to tudi primeren. Prav tako je pomembna avtomatizacija procesa postavitve v testno, pred-produkcijsko in produkcijsko okolje, saj si s tem prihranimo ročno ponavljanje težkih korakov, ki so večkrat izpostavljeni človeškim napakam.

3.2 Uporaba v praksi

Cevovodni proces postavitve je pravzaprav avtomatiziran proces dostave programske opreme. Da bi čim bolj izkoristili prednosti, ki jih uporaba tega procesa

prinaša, je dobro upoštevati naslednje prakse:

Binarne datoteke naj gredo čez proces gradnje le enkrat. Prevajanje kode se zgodi večkrat v različnih kontekstih: med procesom pošiljanja sprememb v nadzor različic, med testiranjem sprejemljivosti, med testiranjem zmogljivosti ter med vsakim procesom postavitve. Binarne datoteke, ki gredo v procesu postavitve na produkcijsko okolje, morajo biti popolnoma enake kot tiste, ki so prestale teste sprejemljivosti. Če jih vedno znova na novo kreiramo, tvegamo, da s tem pridobimo še kako spremembo, ki je v tej fazi nismo želeli. Le tiste datoteke, ki so prestale teste sprejemljivosti, so pripravljene na izdajo, do takrat pa so hranjene nekje na datotečnem sistemu, od koder jih v zadnji fazi procesa brez težav pridobimo in uporabimo.

Proces postavitve naj bo enak za vsa okolja. Izredno pomembno je, da uporabljamo enak postopek za postavitev programske opreme na katerokoli okolje, saj smo le tako lahko prepričani, da bo vse učinkovito delovalo tudi med zadnjim, najpomembnejšim korakom - postavitvijo v produkcijsko okolje. Vse to je odvisno tudi od tega, da so nastavitve, ki so različne za vsako posamezno okolje, hranjene ločeno, pridobljene pa v trenutku, ko so potrebne glede na okolje, s katerim delamo. Uporaba skupnega postopka pomeni, da v trenutku pred izdajo vemo, da je bil ta preizkušen neštetokrat, in da je posledično tveganje pri izdaji programske opreme veliko manjše.

Hitri test takoj po uspešni postavitvi. Najbolje kar preko avtomatske skripte, ki naredi hitri test in s tem preveri, da je proces postavitve uspel ter da aplikacija teče, hkrati pa preveri tudi, da so na voljo vse storitve, ki jih aplikacija potrebuje za delovanje.

Postavitev v okolje, ki je identično produkcijskemu. Največja napaka pred izdajo je, da se produkcijsko okolje bistveno razlikuje od testnega in razvojnega. Celoten postopek bi moral vseskozi teči v okoljih, ki vsebujejo čim manj razlik v primerjavi s produkcijskim, najbolj idealno pa kar na kopijah produkcijskega okolja, če je to mogoče. Predvsem je pomembno, da ni razlik v infrastrukturi (topologija omrežja, konfiguracija požarnega zidu, ...) in v konfiguraciji operacijskega sistema.

Vsaka sprememba mora takoj sprožiti postopek cevovodne postavitve. To pomeni, da sistem za sprotno integracijo ob vsakem uspešno izvedenem postopku

cevovodne postavitve znova preveri, če je prišlo do sprememb v repozitoriju, ki je za ta konkreten primer pomemben. Če je odgovor *da*, se postopek z vključenimi najnovejšimi spremembami, ponovi. Vse to je pomembno za tiste faze, ki so popolnoma avtomatizirane.

Če katerakoli faza v cevovodnem procesu postavitve ne uspe, mora biti nadaljnje izvajanje na čakanju. V primeru, da postavitev v neko okolje ne uspe, je za to odgovorna celotna ekipa, ki mora takoj prekiniti z delom in se najprej posvetiti odpravljanju nastale težave, da postopek lahko ponovno neovirano steče.

3.3 Implementacija

Kot je omenjeno v [1], je za implementacijo cevovodnega procesa postavitve dobro uporabiti inkrementalni pristop. Opisani koraki so osnova za uvedbo procesa od samega začetka do popolnega cevovoda.

Modeliranje toka vrednosti in kreiranje delujočega ogrodja. Potrebno je poznati korake, ki pripeljejo od osnutka do izdaje aplikacije in njihove pretečene čase ter čase dodane vrednosti. Za začetek je dovolj upoštevati najpomembnejše faze: fazo prenosa sprememb v nadzor različic pred postopkom gradnje, izvedbo osnovnih meritev in testov enot, fazo izvajanja testov sprejemljivosti in fazo postavitve aplikacije v okolje, ki je identično produkcijskemu za namen prikaza osnovnih idej. Ko je diagram toka vrednosti dokončan, lahko nadaljujemo z modeliranjem procesa v orodjih za sprotno integracijo in upravljanje izdaj. Katerakoli faza, ki bo v prejšnjih korakih ustvarjeno binarno datoteko skušala postaviti v okolje, ki je identično produkcijskemu, za namen ročnega testiranja ali izdaje, zahteva ročni zagon, in za postavitev uporabi izbrano verzijo. Naslednji korak je kreiranje delujočega ogrodja, kar vključuje najmanjši možen napor, ki pripelje vse ključne elemente na pravo mesto. V primeru popolnoma novega projekta je vse te korake dobro izpeljati še preden se začne z razvojem (iteracija nič), če za razvoj uporabimo iterativen postopek.

Avtomatizacija gradnje in procesa postavitve. Proces gradnje vzame kot vhod izvorno kodo in kot izhod vrne binarne datoteke, izvesti pa se mora vsakič, ko je zaznana sprememba v nadzoru različic. Ko le-ta uspešno steče in je v delujočem stanju, je potrebno avtomatizirati še proces postavitve aplikacije v izbrano oko-

lje. Potrebujemo okolje, ki bo čimbolj podobno produkcijskemu, rečemo mu predprodukcijsko okolje, in bo prav tako vzpostavljeno ter vzdrževano z avtomatiziranim procesom. Ko so vse zahteve izpolnjene, definiramo proces postavitve, ki mora biti kar se da zanesljiv, saj je njegova uspešna izvedba predpogoj za avtomatizacijo testov sprejemljivosti.

Avtomatizacija testov enot in analiziranja kode. Za izvedbo testov enot ne potrebujemo posebno zapletene vzpostavitve, saj sami testi za izvajanje ne potrebujejo aplikacije, ki teče, in so prav zaradi tega tudi izredno hitri. Priporočeno je, da po izvedbi vseh teh testov uporabimo orodje za analizo aplikacije in tako pridobimo uporabne informacije v zvezi s pravilnostjo formata kodiranja, pokritostjo kode s testi, sklopljenostjo, itd.

Avtomatizacija testov sprejemljivosti. Teste sprejemljivosti razdelimo na funkcionalne in nefunkcionalne. Že v prvo razvojno fazo je pomembno uvesti nefunkcijsko testiranje, če ne zaradi drugega, vsaj zaradi zmogljivosti, saj le tako vemo, če je nefunkcionalnim zahtevam ves čas zadoščeno. Uporaba testov sprejemljivosti ves čas razvoja lahko pomembno vpliva na zgodnje odkrivanje težko ponovljivih težav in omogoča, da jih še pravočasno obvladamo.

Avtomatizacija izdaje. Zadnja faza je avtomatizacija izdaje, ki zaključi proces. Cevovodni proces postavitve je proces, ki ves čas živi. Sprotno delo na izboljšanju procesa izdaje ter dostave vključuje vzdrževanje cevovodnega procesa postavitve ter skrb za to, da je ves čas v delujočem stanju.

3.4 Strategija izdaje

Za vsako strategijo izdaje, še posebno za izdajo prve verzije ob začetku projekta, je dobro, da vsebuje naslednje:

- ekipo, odgovorno za postavitve v različna okolja in prav tako za izdaje,
- strategijo za upravljanje s sredstvi in konfiguracijo,
- opis tehnologije, ki bo v uporabi za postavitev,
- načrt, kako implementirati cevovodni proces postavitve,

- različna okolja, namenjena za testiranje sprejemljivosti, zmogljivosti, integracije in uporabniške sprejemljivosti, ter postopek, s katerim bodo binarne datoteke kot rezultat gradnje, postavljene v eno od teh okolij,
- opis postopkov za postavitev v testno in produkcijsko okolje,
- vse potrebno za nadzorovanje delovanja aplikacije,
- razpravo o časovnih zahtevnostih za postavitev in izdajo aplikacije ter kako bo to povezano z avtomatizacijo procesa postavitve,
- opis integracije s katerokoli zunanjo storitvijo,
- podrobnosti o beleženju sprememb v stanju aplikacije,
- načrt za obnovo po katastrofi,
- sporazum na ravni storitev programske opreme glede tehnik, ki bodo potrebne,
- obseg produkcijskega okolja in načrt glede izpolnjevanja zahtev o zmogljivosti,
- strategijo arhiviranja, da bodo produkcijski podatki, ki niso več v uporabi, na voljo za kasnejši pregled ali namene vzdrževanja,
- postopek postavitve v produkcijsko okolje,
- načrt za izvedbo hitrih in nujnih (kritičnih) popravkov v produkcijskem okolju,
- načrt za izvedbo nadgradenj v produkcijskem okolju ter načrt za migracijo podatkov, ter
- načrt za vzdrževanje aplikacije.

Najbolj pomemben del strategije za izdajo je izvedba prve izdaje, ki ponavadi nosi največja tveganja, zato potrebuje skrbno izdelan načrt. Ta mora vsebovati:

- opis korakov, ki so potrebni za prvo postavitev aplikacije,

- načrt za hitri test aplikacije in vseh storitev, ki jih uporablja,
- načrt, kako ravnati v primeru težav in vrniti aplikacijo v prejšnje stanje,
- načrt za kreiranje varnostne kopije trenutnega stanja za primer, ko je potrebna obnova,
- korake za nadgradnjo aplikacije, brez spreminjanja stanja, v katerem trenutno je,
- korake za ponoven zagon ali ponovno postavitev aplikacije v primeru težav,
- lokacijo, kamor se beležijo podatki o delovanju aplikacije, in opis informacij, ki jih vsebujejo,
- načine nadzorovanja aplikacije,
- korake za izvedbo migracije podatkov, ki so potrebni kot del procesa izdaje, ter
- način beleženja težav, ki so se pojavile pri prejšnjih postopkih postavitve, in njihove rešitve.

3.5 Izdaje z nično časovno prekinitvijo

Izdaja z nično časovno prekinitvijo (*zero-downtime release*) je izdaja, pri kateri se dejanski postopek zamenjave iz ene verzije na drugo zgodi v trenutku, brez prekinitev. Prav tako je pomembna hitrost pri obratnem postopku, v primeru, da gre nekaj narobe. Glavna značilnost takih izdaj je nepovezanost različnih delov pri samem procesu izdaje, kar pomeni, da se lahko izvedejo povsem neodvisno eden od drugega, kolikor je to le mogoče.

3.6 Sprotna postavitev

Ob upoštevanju fraze ekstremnega programiranja ([14]) – če je le mogoče, delaj to čimbolj pogosto – je logična posledica nova postavitev v produkcijsko okolje ob vsaki spremembi, ki uspešno prestane avtomatske teste. Temu procesu rečemo

tudi sprotna postavitev, in ključna točka pri tem je prav to, da gre za postavitev v produkcijsko okolje. Pridemo pa tudi do trenutka, ko novih funkcionalnosti nočemo takoj poslati v produkcijsko okolje. Dober primer so recimo podjetja z omejitvami glede skladnosti, kjer je pred postavitvijo v produkcijsko okolje potrebna odobritev.

Najbolj pogost razlog proti je, da je tak način preveč tvegan, a kljub vsemu vemo, da je manj tvegano prav to, kar ves čas uporabljamo v praksi, pa naj bo to postavitev v testno okolje ali pa produkcijsko – za oba uporabljamo isti postopek, iste skripte, zato ne v enem ne v drugem primeru ne bi smelo biti nič več tveganja. V primeru pa, ko skozi proces izdaje pošljemo skoraj vsako spremembo, ki se zgodi, je to le še lažje, saj manjše spremembe vedno s seboj prinesejo manjša tveganja in so v primeru težav lažje obvladljiva. Celoten proces je nemogoče uveljaviti brez avtomatiziranih postopkov gradnje, postavitve in izdaje, prav tako je pomembno, da je zbirka avtomatskih testov celovita in zanesljiva. Tudi če naš namen ni sprotna izdaja vseh sprememb, je pomembno, da postopek tako zastavimo, in ga kot takega potem uporabimo kadarkoli želimo.

Poglavje 4

Od razvoja do priprave na izdajo

4.1 Odločitev za uporabo postopka sprotne dostave

Dosledno upoštevanje pravil pri uporabi celotnega postopka sprotne dostave od samega razvoja, sprotne integracije do končne izdaje ni enostavno in včasih v celoti skorajda nemogoče. Projekti se med seboj razlikujejo, prav tako želje strank, najbolj pa seveda to, kaj si želimo med samim razvojem olajšati, koliko je to sploh potrebno, in kaj sledi po zaključku razvoja. Vprašanj je sicer veliko, a z uporabo vsaj delnega, če že ne celotnega postopka sprotne dostave, si zelo olajšamo razvoj, odprava težav je takojšnja, kar pa posledično prinese manj stresa v obdobje, ko je čas za izdajo. Zato smo se za tak pristop odločili tudi mi.

4.2 Začetek razvoja

Ker smo se v podjetju odločili za spremembe v konceptu samega razvoja, je bil to pravi čas tudi za uporabo novega postopka postavitve v produkcijsko okolje. Sočasno z začetkom razvoja po prenovljenem principu (razvoj spletne aplikacije), smo s pomočjo [1], kjer Jez Humble in David Farley predstavita povsem drugačen način razmišljanja v zvezi s sprotnim dostavljanjem programske opreme, zasnovali

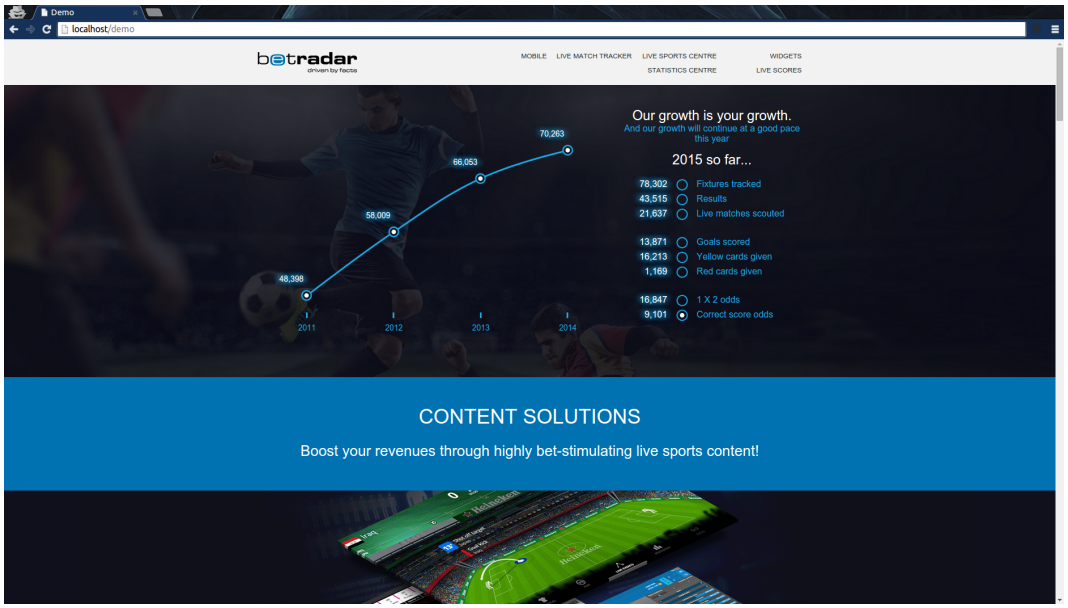
svoj, malce prilagojen proces, ki seveda potrebuje še marsikatero izboljšavo, da se čimbolj približa opisanemu postopku v knjigi.

Pri izbiri sistema za nadzor različic smo se odločili za Git ([2]), saj gre za brezplačen odprtokoden porazdeljen sistem za nadzor različic, ki je namenjen hitremu in učinkovitemu obvladovanju majhnih do zelo obsežnih projektov. Je enostaven za razumevanje in performančno izredno zmogljiv, s funkcionalnostmi kot so poceni lokalna vejitev (*cheap local branching*), priročni načini za hranitev sprememb, ki še niso primerne za verzioniranje (*convenient staging areas*), ter večje število delovnih tokov (*multiple workflows*). Z vsem tem prekaša ostala orodja za upravljanje konfiguracije programske opreme (SCM), kot so npr. Subversion, CVS, Perforce in ClearCase.

Glavno razvojno vejo smo poimenovali *develop*. Tu poteka razvoj vse do prve postavitve v produkcijsko okolje, kasneje pa se orientacija preusmeri na razvoj novih funkcionalnosti oziroma na vzdrževanje. Za nadzor nad samo kodo, smo uporabili statični analizator JSHint ([3]), ki je namenjen odkrivanju napak (predvsem v sintaksi) in možnih težav s programskim jezikom Javascript, ter na ta način celotno razvojno ekipo prisili v vnaprej določen format kodiranja. Standarde, ki jih je potrebno upoštevati, ekipa določi sama s pomočjo konfiguracijske datoteke, pravilnost pa se preveri ob vsakem poskusu pošiljanja sprememb v nadzor različic (doseženo s pomočjo dodatnih nastavitev med postavitvijo razvojnega okolja). V primeru, da pride do napak pri preverjanju formata kodiranja, je pošiljanje v nadzor različic prekinjeno, in je onemogočeno, dokler se omenjene težave ročno ne odpravijo. Ko težave odpravimo, ponovimo korak pošiljanja sprememb v nadzor različic, in če so bile vse napake odpravljene, se operacija zaključi uspešno. S tem si zagotovimo dokaj nemoteno sprotno integracijo med razvojem, saj na ta način preprečimo težave zaradi osnovnih napak pri kodiranju, ki se, predvsem pri uporabi skriptnih programskih jezikov, lahko velikokrat pojavijo povsem nenamerno.

4.3 Sprotna integracija

Za ponovljivo izvajanje postopka sprotne integracije smo izbrali integracijski strežnik Jenkins, ki nam je omogočal vse, kar smo potrebovali. Prvi korak je nadzorovanje razvojne veje in integracija sprememb na izbrani razvojni strežnik – na ta način

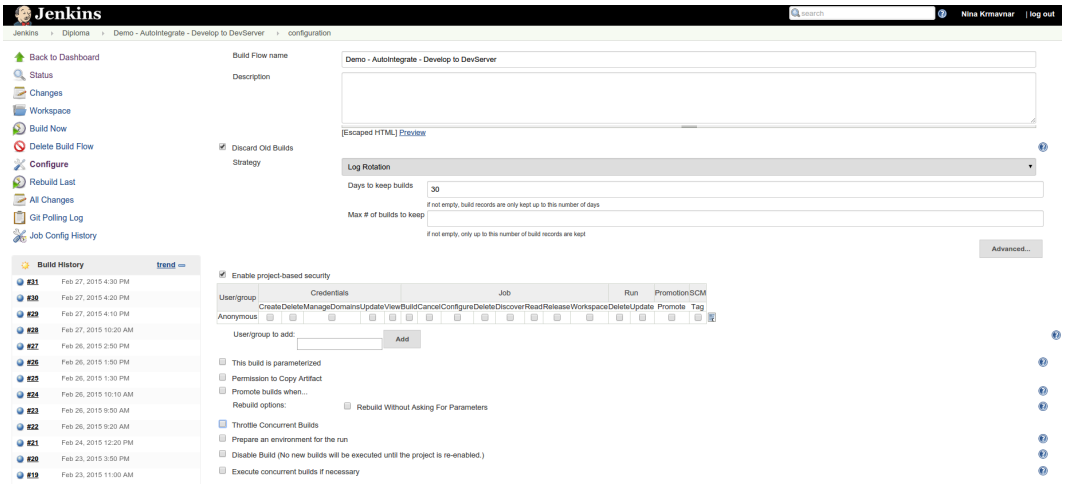


Slika 4.1: Testna spletna aplikacija, postavljena na lokalnem strežniku.

imamo vedno pregled nad trenutnim stanjem aplikacije, hkrati pa v celoti ves čas uporabljamo vse skripte za gradnjo, ki se v naslednjih fazah uporabijo tudi za postavitev v pred-produkcijsko in produkcijsko okolje (le s spremenjenimi parametri). Tako vemo, da v naslednjih fazah ne bi smelo prihajati do prevelikih težav, vsaj ne s strani postopka sprotne integracije in postavitve.

Postavili smo testno spletno aplikacijo, ki je bila na začetku še v prvih fazah razvoja in je kasneje pridobivala na vsebini ter funkcionalnostih (Slika 4.1). Sledi vzpostavitev opravila na integracijskem strežniku Jenkins, ki bo poskrbelo za sprotno integracijo spletne aplikacije v razvoju na izbrani testni strežnik. Izbran tip opravila se imenuje *tok gradnje* (*build flow*), ki kljub hierarhičnem izvajanju opravil (v večini primerov gre za dvo-nivojsko hierarhično strukturo), omogoča vpogled v celoten tok dogodkov s pomočjo zabeleženih sprememb, ki so se zgodile. Prvo opravilo je namenjeno nadzorovanju razvojne veje v časovnem intervalu desetih minut (Slika 4.2), ki nato v primeru odkritih sprememb, sproži osnovno opravilo za integracijo.

Slika 4.3 prikazuje, kako definiramo nadzorovanje razvojne veje v določenem repozitoriju, ki preverja stanje na nek določen časovni interval. V tem primeru

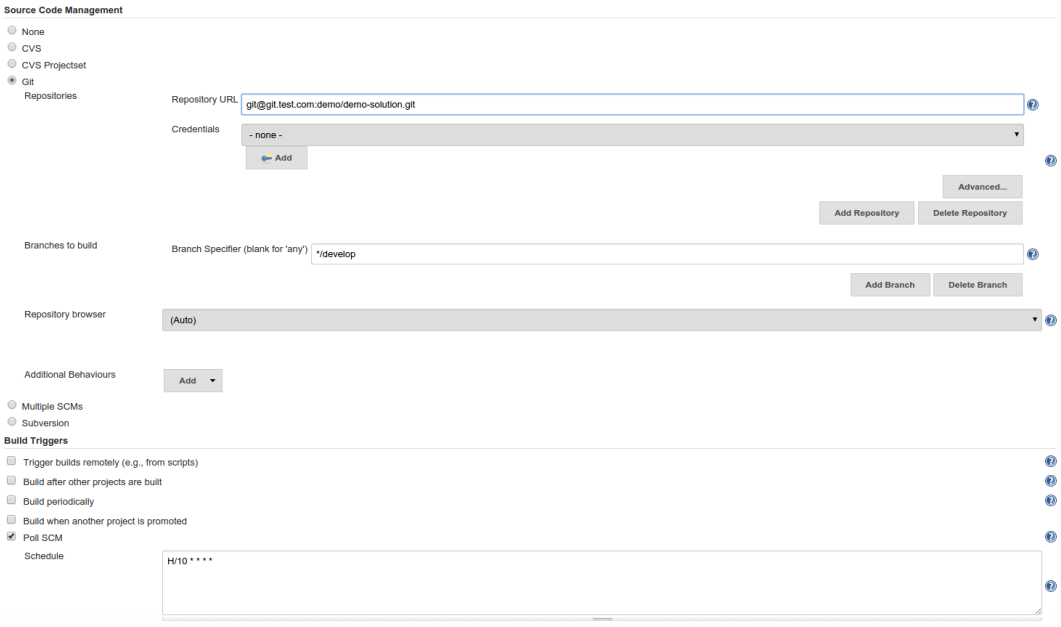


Slika 4.2: Nastavitve začetnega opravila za sprotno integracijo razvojne veje na testni strežnik.

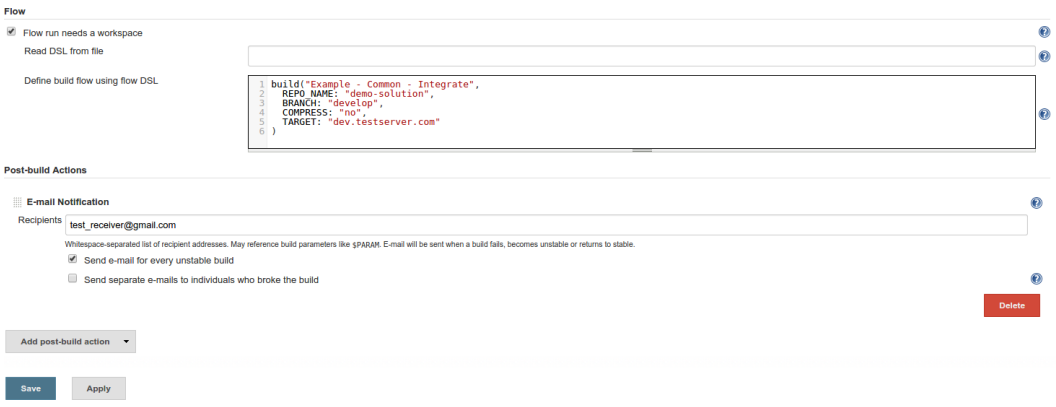
gre za razvojno vejo poimenovano *develop*, ki se nahaja v repozitoriju na lokaciji *git@git.test.com:demo/demo-solution.git* (zaradi občutljivosti informacij so pravi podatki prikriti). V naslednjem koraku definiramo še nadaljevanje toka izvajanja, ki sproži splošno opravilo sprotne integracije, ter elektronski naslov, kamor se pošlje opozorilo v primeru, da gre pri izvajanju opravila karkoli narobe (Slika 4.4).

Splošno opravilo za sprotno integracijo (Slika 4.5) in postavitev predefinirane veje podanega repozitorija na izbrani strežnik za svoje delovanje potrebuje določene vhodne parametre: ime repozitorija (repo name – obvezen parameter), ime veje v izbranem repozitoriju (branch - obvezen parameter), nastavitev opcije, ki skrbi za to, da se ob izvajanju skript za gradnjo sproži tudi stiskanje datotek, ki so pomembne za postavitev na izbrani strežnik (compress – privzeta vrednost je *ne*), ter ključ, preko katerega ugotovimo, kakšna je ciljna destinacija v primeru postavitve na nek strežnik (target – opcijski parameter). Primer nastavitve vhodnih parametrov je viden na Sliki 4.6.

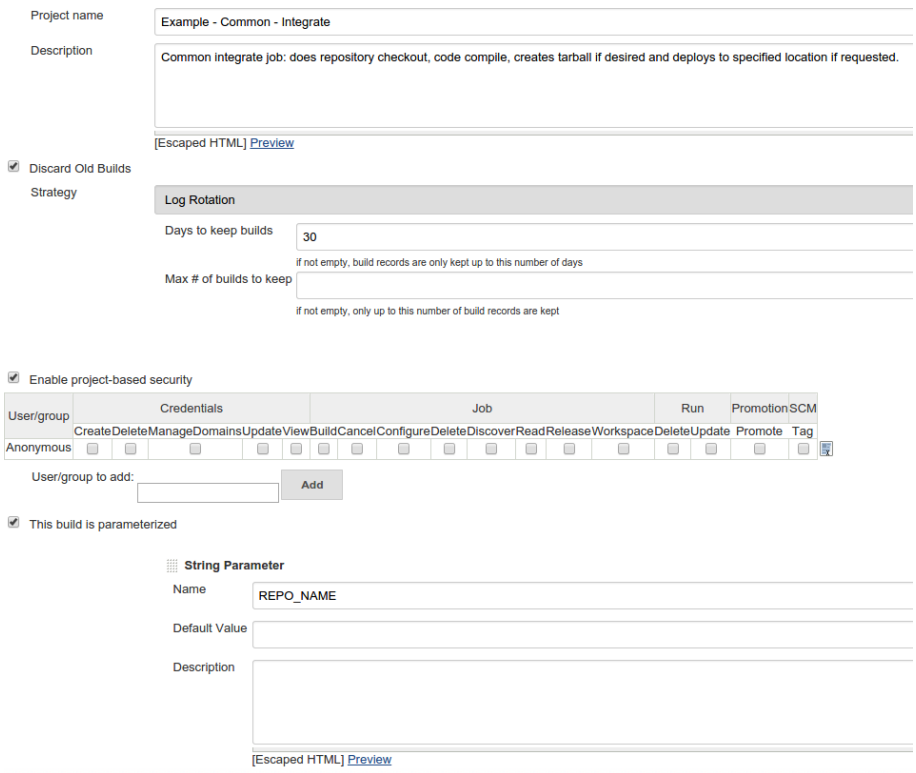
Ker je glavni del splošnega opravila izvršitev integracijske skripte za gradnjo, je potrebno definirati repozitorij, v katerem se nahajajo (Slika 4.7), v zadnjem koraku pa dejansko izvesti klic skripte za gradnjo z ustreznimi parametri. Ker gre v osnovi za sprotno integracijo razvojne veje na testni strežnik, kot parametre podamo ustrezen repozitorij ter ime razvojne veje, opcijo, ki pove ali potrebujemo



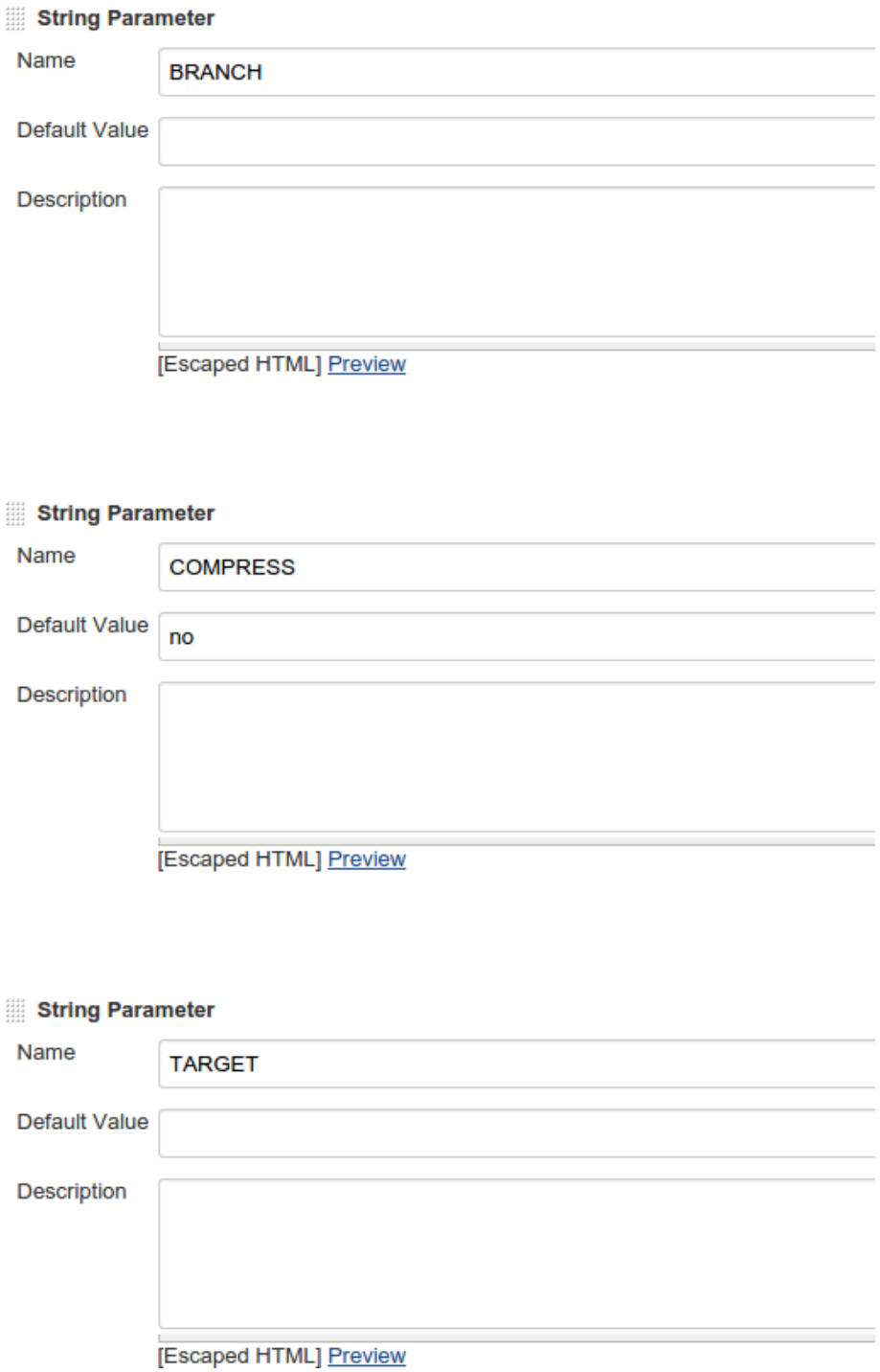
Slika 4.3: Začetno opravilo za sprotno integracijo razvojne veje na testni strežnik - nastavitve za nadzorovanje repozitorija in razvojne veje ter interval preverjanja.



Slika 4.4: Začetno opravilo za sprotno integracijo razvojne veje na testni strežnik – nastavitve za nadaljevanje toka izvajanja.



Slika 4.5: Nastavitve splošnega opravila za sprotno integracijo predefinirane veje na izbrani strežnik.



Slika 4.6: Splošno opravilo za sprotno integracijo predefinirane veje na izbrani strežnik – vhodni parametri.

stiskanje datotek ali ne, ter ciljno destinacijo za postavitev (testni strežnik). Dodatno lahko specificiramo še delovni prostor (workspace), ki je pomemben pri delu z repozitorijem (predefiniran v nastavitvah integracijskega strežnika Jenkins) ter številko gradnje (build number – avtomatsko dodana strežniku, zaporedna številka izvajanja opravila), ki nam lahko služi kot unikaten dodatek pri poimenovanju datotek, saj se vsakič inkremenitira in se zato ne more ponoviti (Slika 4.8).

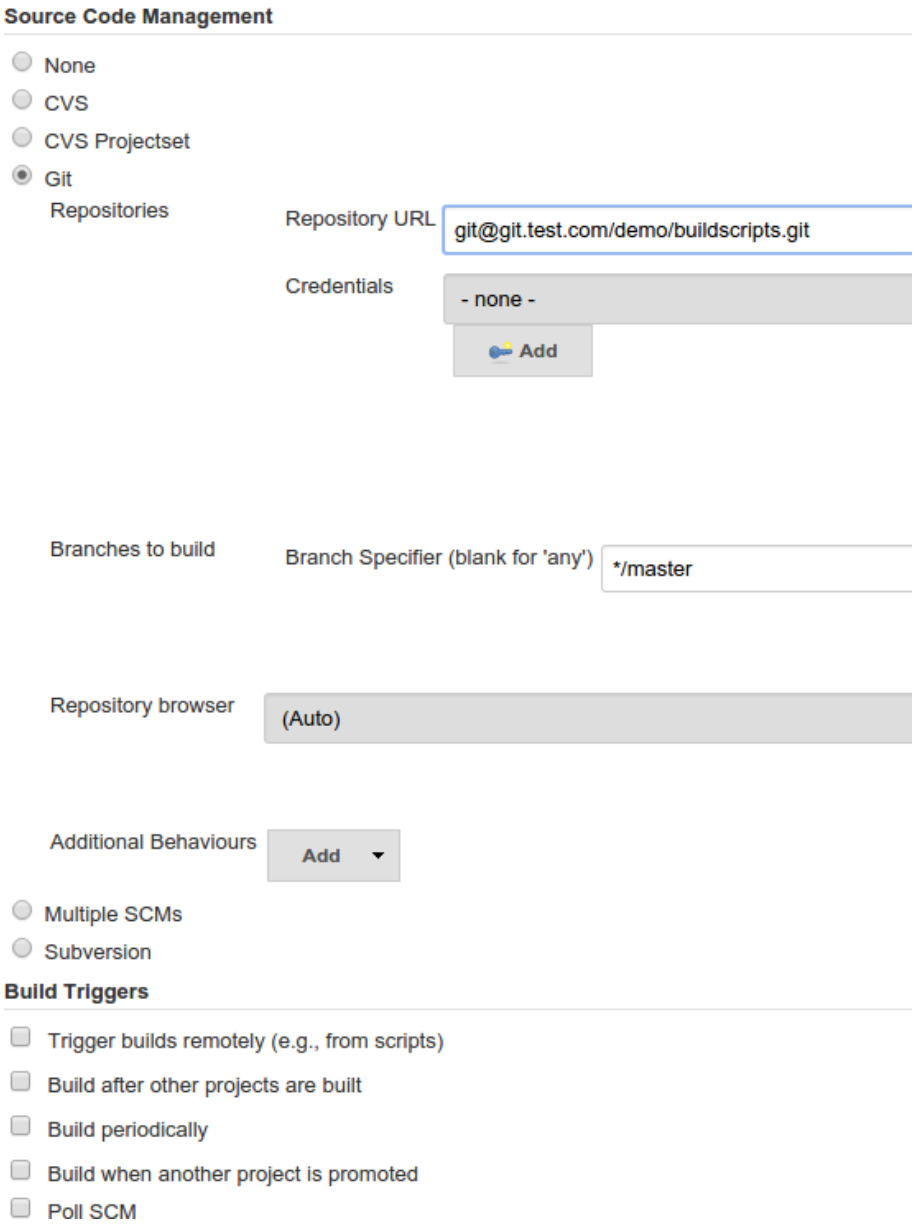
4.4 Skripte za gradnjo

Splošno opravilo na integracijskem strežniku Jenkins za sprotno integracijo v zadnjem koraku začne z izvajanjem integracijske skripte za gradnjo. Skripta integrate.sh (Dodatek A, Slika A.1), ki jo je mogoče pognati tudi iz ukazne vrstice, je ključna v celotnem procesu sprotne integracije in dostave. Skripta najprej prebere podane vhodne parametre in preveri, če so vrednosti v želenem naboru, oziroma če so sploh prisotne (kar je pomembno pri obveznih parametrih); če ni napak, določimo delovno mapo, ki bo namenjena upravljanju izvorne kode in izvrševanju vseh pomembnejših operacij. Šele po tem pripravimo vso potrebno direktorijsko strukturo. Ko je končana priprava imen za datoteke, ki jih v nadaljevanju potrebujemo, je na vrsti delo z izvorno kodo.

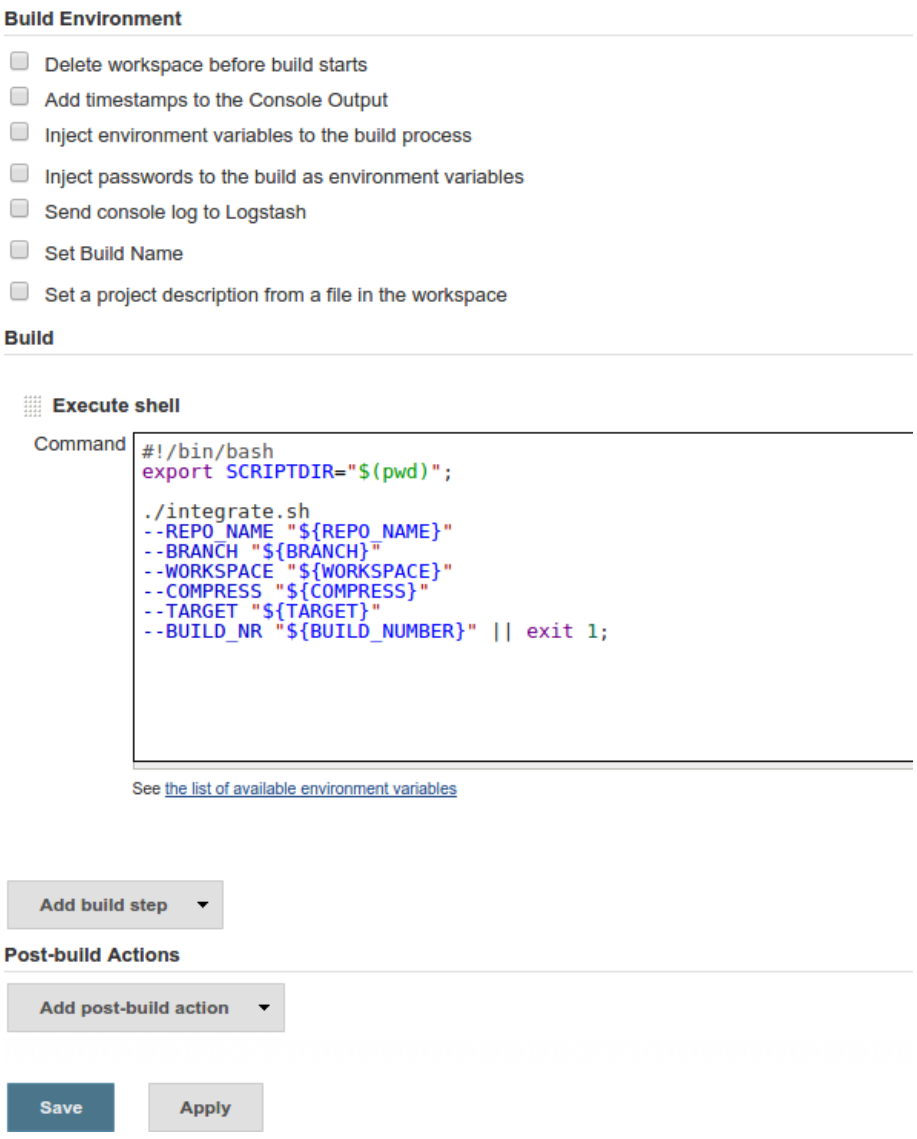
Izvajati se začne skripta checkout.sh (Dodatek A, Slika A.2), ki na začetku prav tako prebere podane vhodne parametre in preveri njihovo pravilnost. Če se skripta izvaja prvič, v nadaljevanju skopira repozitorij v delovno mapo, drugače pa le uporabi prejšnje stanje, ki se posodobi. Preusmerimo se na izbrano vejo (skripta to vrednost dobi kot vhodni parameter), jo posodobimo in pripravimo vse potrebno za kasnejšo uporabo.

Sledijo operacije, potrebne pred prevajanjem sredstev, ki jih naša spletna aplikacija uporablja (slike, javascript datoteke, itd.). S tem se pripravi vse potrebno pred klicem skripte za prevajanje, ki na osnovi izvedenih operacij ve kaj mora prevesti. Vse to prinese izjemen prihranek na času, še posebej v primeru, ko ponovno prevajanje sploh ni potrebno.

Eden izmed vhodnih parametrov integracijske skripte je tudi opcija, s katero, če to želimo, sprožimo stiskanje datotek, ki so pomembne za postavitev aplikacije na izbrani strežnik. Skripta compress.sh (Dodatek A, Slika A.3) se izvede le v



Slika 4.7: Splošno opravilo za sprotno integracijo predefinirane veje na izbrani strežnik – nastavitev repozitorija in veje, kjer se nahajajo skripte za postavitev.



Slika 4.8: Splošno opravilo za sprotno integracijo predefinirane veje na izbrani strežnik – klic integracijske skripte za gradnjo z ustreznimi parametri.

primeru, ko stiskanje želimo izvesti, potrebuje pa še pot do stisnjene datoteke, v katero se bo želena vsebina stisnila. Preden nadaljujemo, izvedemo čiščenje vsebine začasnega direktorija, saj dolgoročno hranimo vse datoteke stare do največ 14 dni, a minimalno število le-teh mora vedno biti vsaj enako 10, tudi če to pomeni, da nekatere od njih ne ustrezajo prvemu pogoju. Ko je čiščenje končano, izvedemo stiskanje datotek, ter jih s tem pripravimo na možnost prenosa na strežnik.

Zadnji, najbolj pomemben korak pri postopku sprotne integracije, je še postavitev spletne aplikacije v izbrano okolje. Ker smo v tem trenutku še v fazi razvoja, potrebujemo podatke o testnem strežniku, kamor se bo izvedla postavitev. Skripta deploy.sh (Dodatek A, Slika A.4) je na podobnem nivoju kot integracijska, saj je popolnoma uporabna tudi kot samostojna enota (na primer pri postavitvi v produkcijsko okolje). Podobno tudi ta najprej prebere podane vhodne parametre in preveri njihovo pravilnost. Sledi priprava podatkov, ki omogočajo dostop do poljubne storitve (v našem primeru dostop do testnega strežnika) in priprava poti do ciljne destinacije. Za prenos datotek na testni strežnik uporabljamo orodje *rsync* ([4]), ki je hiter in izredno vsestranski pripomoček, namenjen kopiranju datotek. Ko je prenos končan, izvedemo sprostitev delovne mape, kar pomeni, da je direktorij z izvorno kodo na voljo drugim procesom, in postopek integracije je končan.

4.5 Zagotavljanje kakovosti

Že med samim razvojem je potrebno konstantno preverjanje kakovosti programske opreme, ki jo razvijamo. Po principu, opisanem v [1], bi to pomenilo vključiti razne avtomatske teste že v sam proces sprotne integracije, a smo v našem primeru ta del izpustili ter ročno izvedbo procesa zagotavljanja kakovosti vključili na konec omenjenega postopka, ki se ponovljivo izvaja po uspešni postavitvi programske opreme v testno okolje. Za te potrebe je del ekipe zadolžen le za sprotno ročno testiranje aplikacij, ki jih razvijamo. Glavni razlog za to je tip aplikacije, saj gre za spletne produkte, kjer je naša ekipa zadolžena le za razvoj uporabnikom vidnega dela (front-end), za procese, ki se izvajajo v ozadju, pa skrbi druga ekipa in za to vodi ločen proces sprotne integracije. Testi enot, komponent, itd. tako v tem primeru ne pridejo v poštev, je pa zato večji poudarek na ročnem preverjanju

kakovosti, ki nam omogoči sprotno in čimprejšnje odkrivanje napak, ter zagotavlja, da uporabnikom dostavimo programsko opremo na najvišji možni ravni.

Poglavje 5

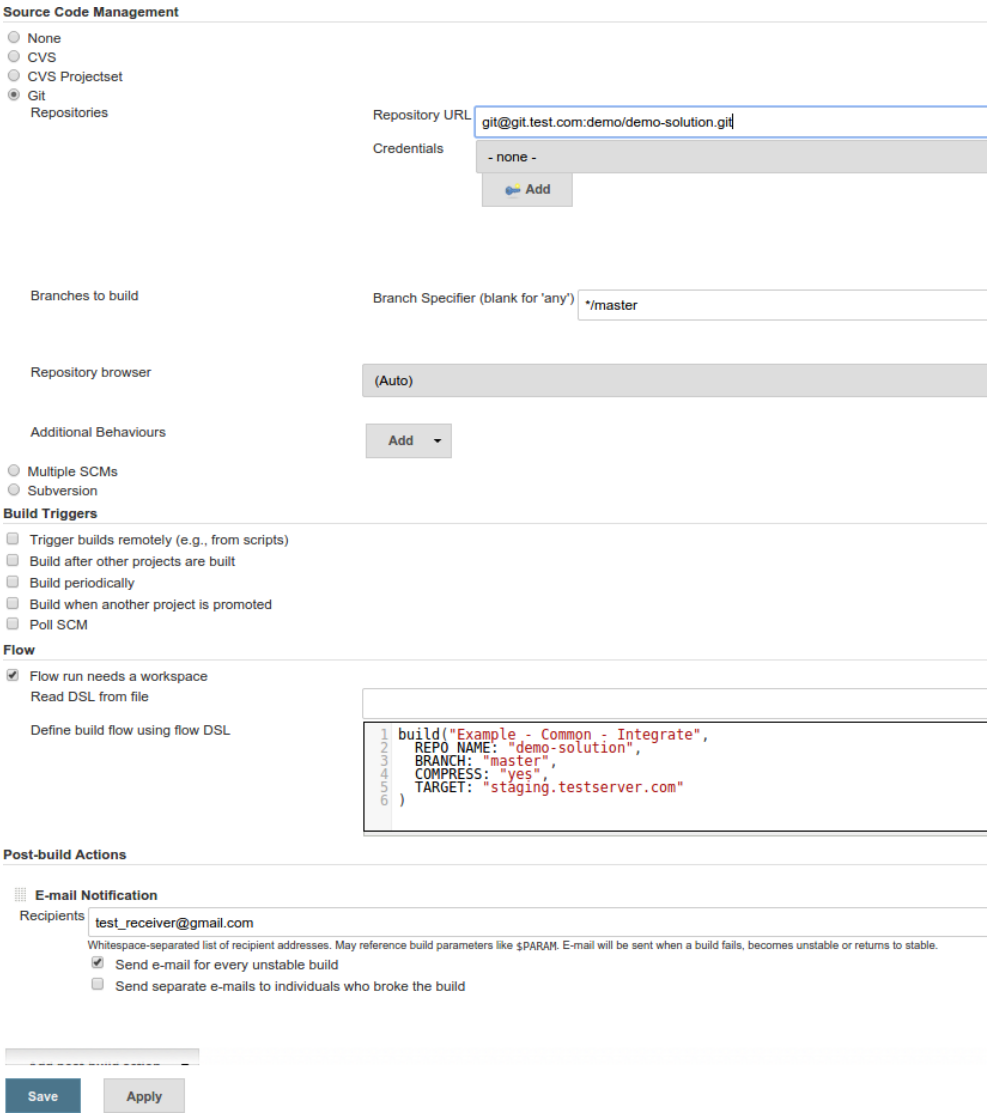
Izdaja programske opreme

5.1 Priprava na izdajo

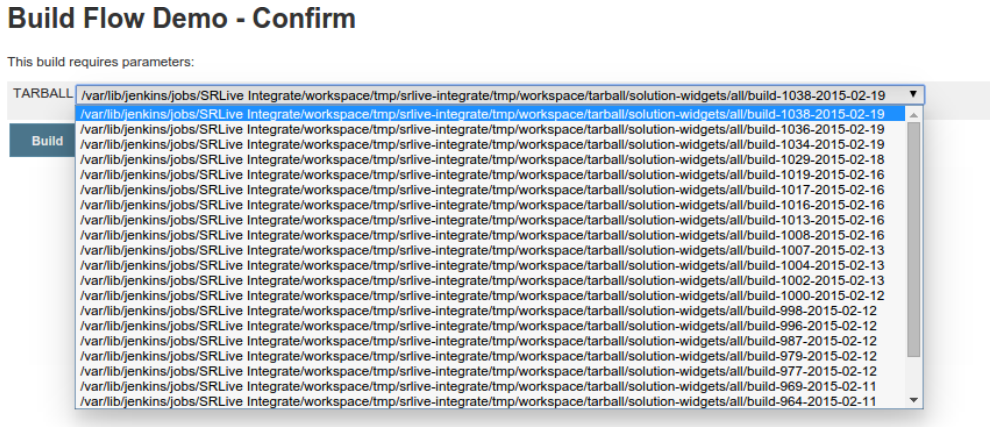
Za pripravo na prvo izdajo oziroma postavitev v pred-produkcijsko okolje je ponavadi določen datum, do katerega je potrebno poskrbeti, da je to možno izvesti. Takrat se izvede združitev razvojne veje (*develop*) v glavno (*master*). Proces se izvede ročno in potrebuje določeno mero discipline, saj zlahka pride do težav zaradi nepazljivega izvajanja ali površnosti. V ozadju na integracijskem strežniku Jenkins teče opravilo, ki ves čas nadzoruje glavno vejo, in se izvede, če zazna spremembe. Po združitvi v glavno vejo se opravilo sproži, in ima povsem enake karakteristike kot opravilo, opisano v poglavju 4.3. Najpomembnejša razlika je, da v tem koraku stisnemo datoteke in si ta skupek nekam shranimo, saj s tem zagotovimo, da kasneje za postavitev v produkcijsko okolje uporabimo točno to različico, ki mora prej uspešno prestati proces priprave na izdajo. S Slike 5.1 je razvidno še, da se spremeni tudi ciljna destinacija postavitve.

5.2 Kandidat za izdajo

Po uspešno izvedenem postopku postavitve v pred-produkcijsko okolje, mora aplikacija prestati obsežno uporabniško testiranje sprejemljivosti, in ko vse to prestane, je na vrsti določitev kandidata za izdajo. To storimo tako, da na integracijskem strežniku Jenkins zaženemo opravilo za potrditev kandidata za izdajo, za kar mo-



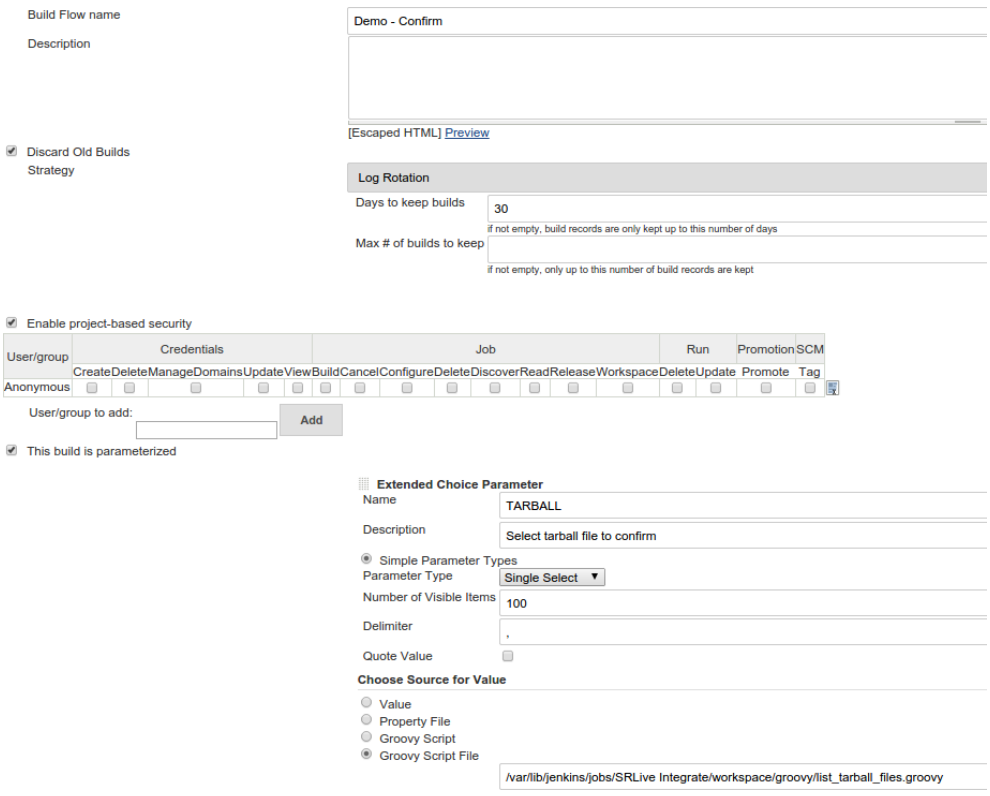
Slika 5.1: Pomembnejše nastavitve opravila za sprotno integracijo glavne veje v pred-produkcijsko okolje.



Slika 5.2: Opravilo za potrditev kandidata za izdajo – izbor stisnjene datoteke, primerne za postavitev v produkcijsko okolje.

ramo najprej iz seznama ponujenih stisnjenih datotek (Slika 5.2) izbrati tisto, ki je primerna za postavitev v produkcijsko okolje, torej tisto, ki je nazadnje uspešno prestala uporabniško testiranje sprejemljivosti. Seznam se dinamično napolni s pomočjo skripte, napisane v programskem jeziku Groovy (Slika 5.3).

Ko zaključimo izbor in zaženemo opravilo, se pot do izbrane stisnjene datoteke shrani v okoljsko spremenljivko *params* integracijskega strežnika Jenkins pod imenom, ki smo ga definirali v nastavitvah opravila (tarball). Tok izvajanja se preusmeri na splošno opravilo za potrditev kandidata za izdajo (Slika 5.4), ki v nadaljevanju izvede skripto confirm.sh (Slika 5.5). Kot je razvidno s Slike A.5 (Dodatek A), skripta najprej prebere vse vhodne parametre in preveri njihovo pravilnost, ter razpakira izbrano stisnjeno datoteko v že vnaprej pripravljeno mapo. V razpakirani mapi poiščemo datoteko revision.html. Ta vsebuje unikatno vrednost, ki kaže na določeno lokacijo v repozitoriju, kar kasneje potrebujemo zaradi kreiranja zaznamka z verzijo, s katero bomo po koncu izvajanja skripte označili omenjeno lokacijo. S pomočjo prebrane lokacije stanje v lokalno prenešenem repozitoriju vrnemo na mesto, kamor ta kaže, ter preberemo zadnjo verzijo oblike vX.Y.Z, ki je bila označena do tistega trenutka. Gre za semantični način verzioniranja (*semantic versioning*), kjer X predstavlja veliko spremembo in ne zagotavlja skladnosti s starejšimi različicami, Y predstavlja manjšo spremembo in se poveča ob običajnih postavitvah v produkcijsko okolje, Z pa se spreminja le v primeru nujnih poprav-



Slika 5.3: Nastavitve opravila za potrditev kandidata za izdajo – dinamično polnjenje seznama za izbor stisnjene datoteke za potrditev.

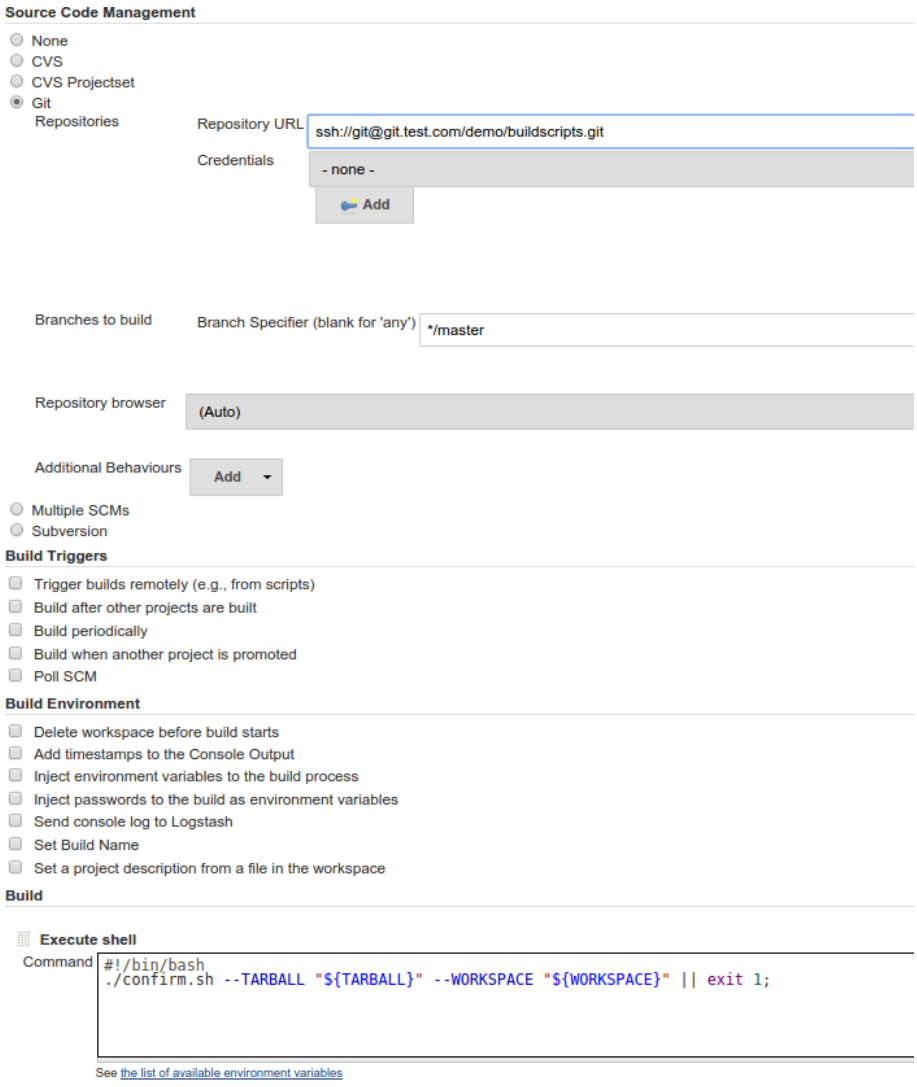
Slika 5.4: Nastavitve opravila za potrditev kandidata za izdajo – nadaljevanje toka izvajanja na splošno opravilo za potrditev kandidata za izdajo.

kov po že izvedeni postavitvi v produkcijsko okolje. Ker pri trenutnem postopku uveljavljamo manjšo spremembo, za 1 povečamo le Y (Y=Y+1), Z pa ponastavimo na 0. V primeru nujnih popravkov bi povečali le parameter Z (Z=Z+1). Novo, spremenjeno obliko verzije (vX.(Y+1).0) povežemo s prebrano lokacijo v repozitoriju (prebrana iz datoteke revision.html), in prenesemo spremembe na repozitorij še globalno. Stisnjeno datoteko, s katero smo delali, prenesemo v mapo za potrjene datoteke (*confirmed*), pred tem pa njenemu imenu na konec dodamo še novo obliko verzije.

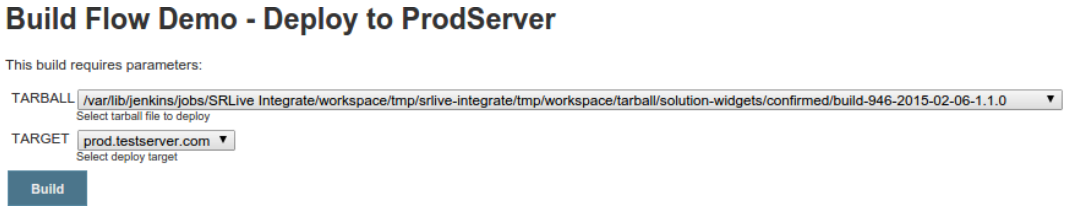
5.3 Izdaja in postavitev v produkcijsko okolje

Po končanem postopku izbire kandidata za izdajo, nas do končne izdaje loči le še zadnji korak - postavitev v produkcijsko okolje. Ena izmed operacij, ki se izvedejo v fazi izbiranja kandidata za izdajo, je tudi prenos izbrane stisnjene datoteke v mapo potrjenih datotek (*confirmed*), kar posledično pomeni, da je le-ta primerna ter na voljo za izdajo in postavitev v produkcijsko okolje.

Postopek se začne z izbiro konkretne stisnjene datoteke iz mape potrjenih da-



Slika 5.5: Nastavitve splošnega opravila za potrditev kandidata za izdajo – klic skripte confirm.sh, ki izvede potrebne operacije za potrditev kandidata za izdajo.

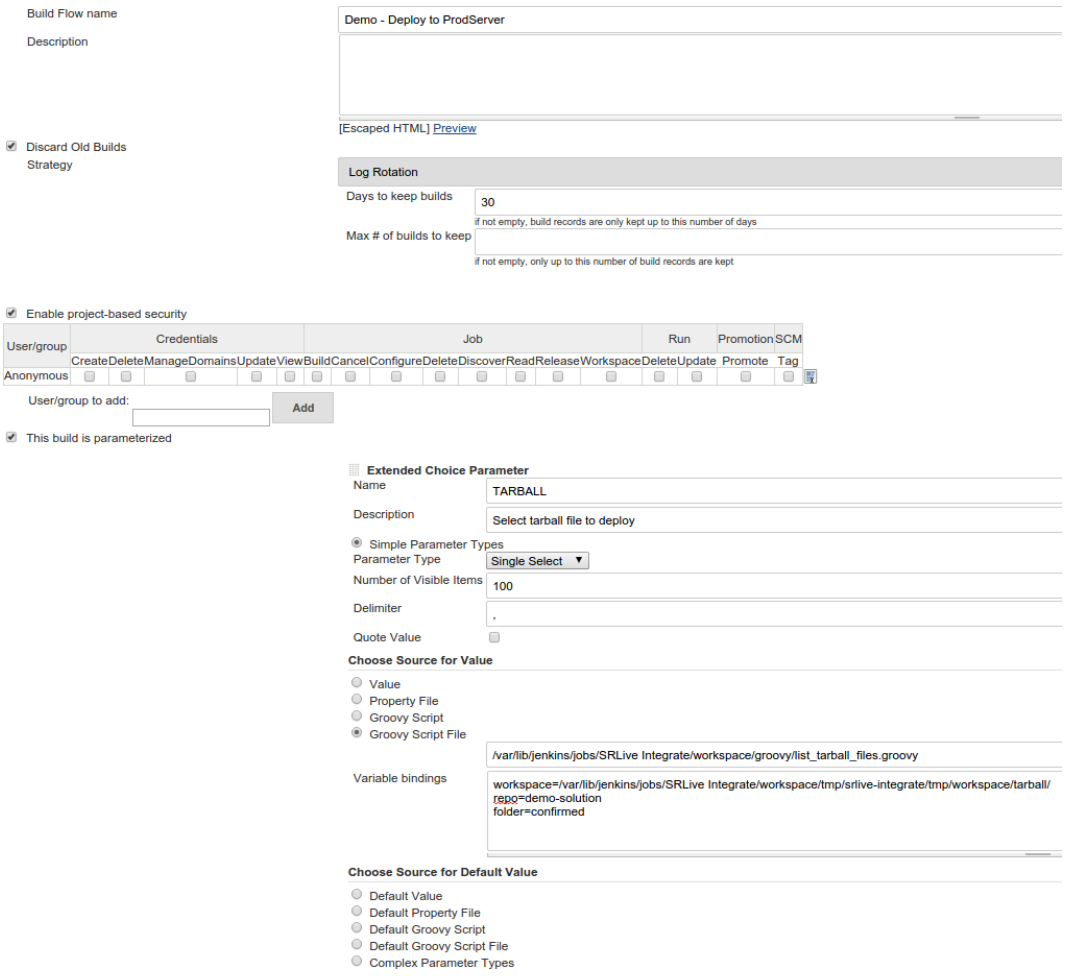


Slika 5.6: Sprožitev opravila za postavitev v produkcijsko okolje - izbor stisnjene datoteke in ciljne destinacije za postavitev.

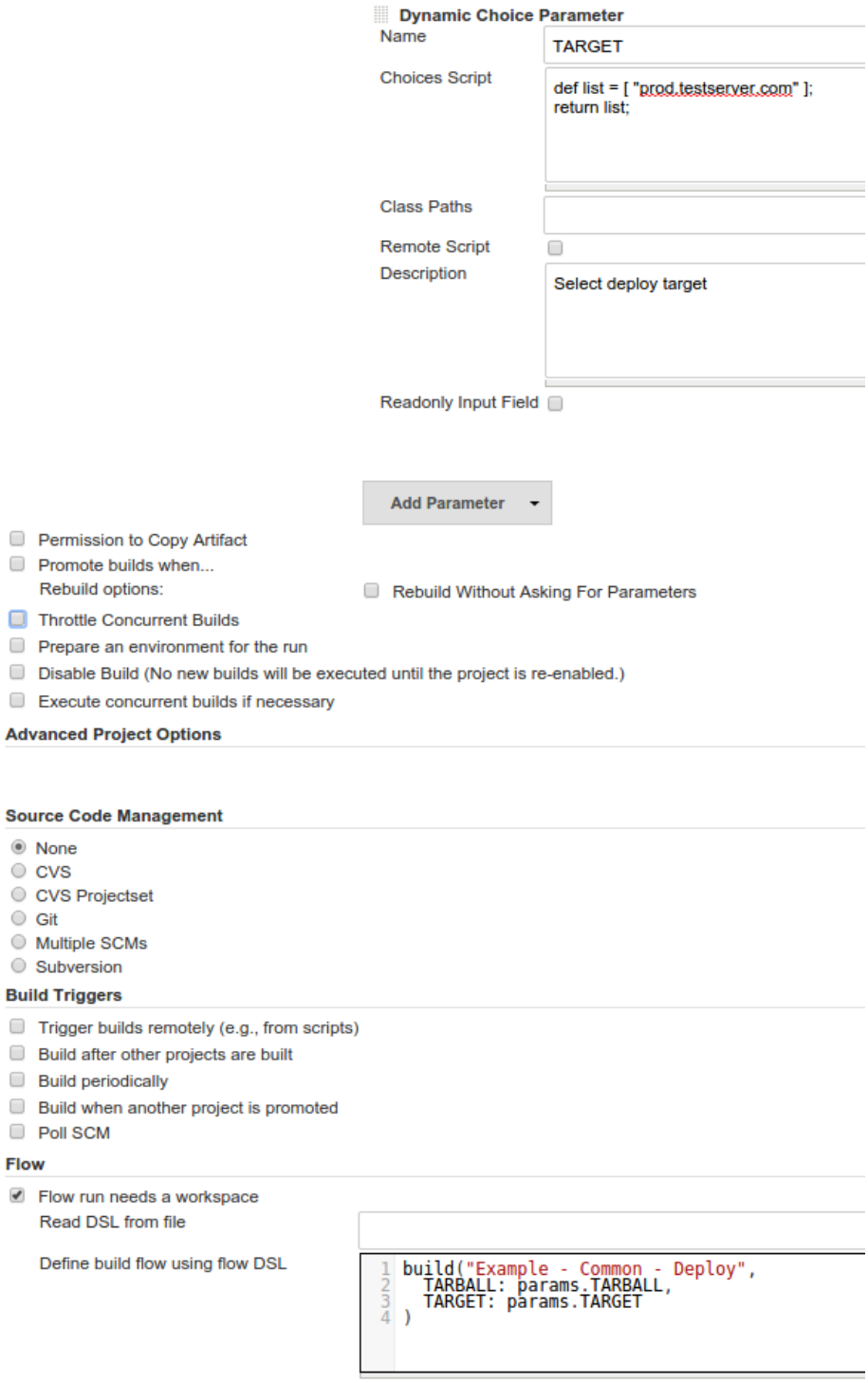
totek, ki jo želimo uporabiti za postavitev (Slika 5.6). Z izbiro datoteke in ciljne destinacije, kamor želimo postaviti aplikacijo, sprožimo opravilo na integracijskem strežniku Jenkins, ki prebere izbrane vhodne parametre in preveri njihovo pravilnost (Slika 5.7). Sledi nadaljevanje toka izvajanja s sprožitvijo splošnega opravila za izdajo in postavitev (Slika 5.8), ki izvede običajno začetno preverjanje parametrov, ipd. zaključi pa z izvedbo skripte za postavitev deploy.sh (Slika 5.9). Postopek, ki se izvede, je povsem enak tistemu, ki je opisan v poglavju 4.4 (del, ki govori o postavitvi spletne aplikacije v izbrano okolje), uporabljena skripta pa identična tisti na Sliki A.4 (Dodatek A). Edina večja razlika pri postavitvi v testno ali produkcijsko okolje je, da pri slednji za postavitev uporabimo izbrano stisnjeno datoteko, ki je primerna za izdajo, pri postavitvi v testno okolje pa gre vedno za najbolj posodobljeno stanje, ki je na voljo v ustreznem repozitoriju. Skripta izvede še vse dodatne operacije, ki so potrebne pri postavitvi, in če se vse zaključijo uspešno, je postopek postavitve v produkcijsko okolje končan, s tem pa tudi prva uspešna izdaja nove programske opreme.

5.4 Odpravljanje težav

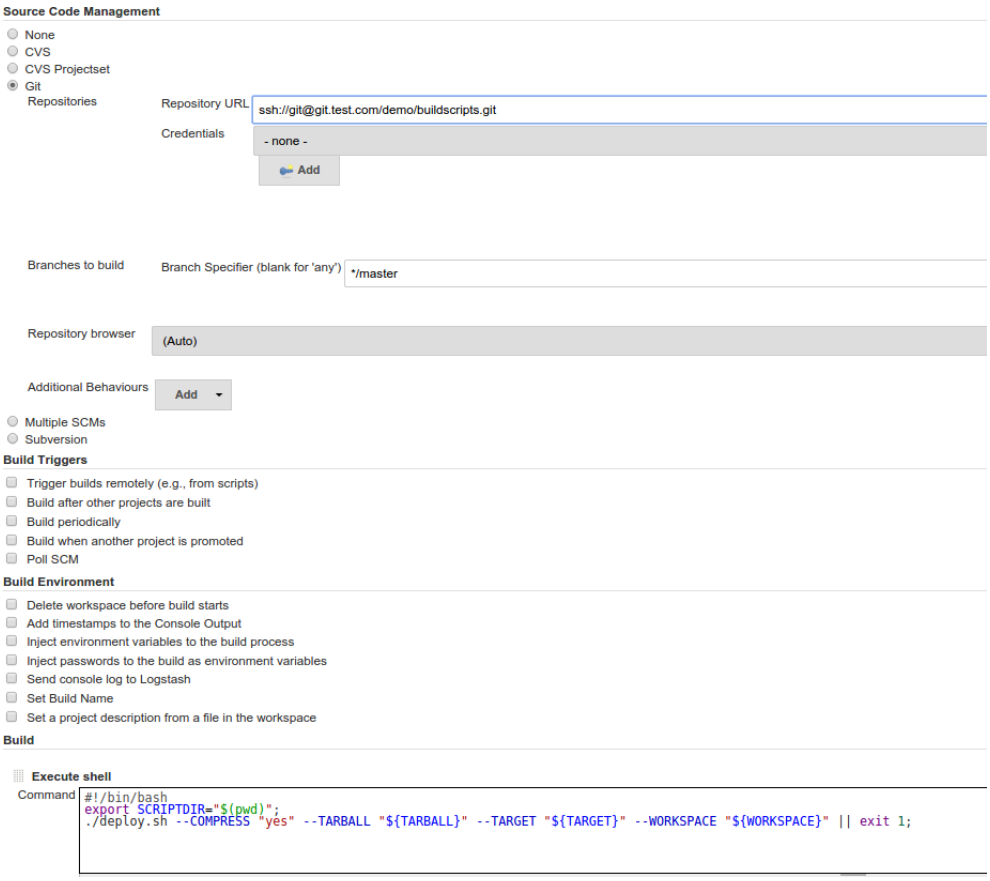
Pri prvi ali pa vsaki naslednji izdaji aplikacije v produkcijsko okolje si seveda želimo, da bi se celoten postopek zaključil brez težav, a vedno temu ni tako. Prav tako sploh ni nujno, da gre karkoli narobe pri samem postopku, temveč lahko šele po dejanski (na videz uspešni) izdaji opazimo določene pomanjkljivosti ali nepravilnosti v delovanju aplikacije, ki žal, po nekem čudnem spletu okoliščin, niso bile odkrite pravočasno. Za rešitev situacije imamo na voljo dva pristopa. V obeh



Slika 5.7: Nastavitve opravila za postavitev v produkcijsko okolje – dinamično polnjenje seznama za izbor stisnjene datoteke.



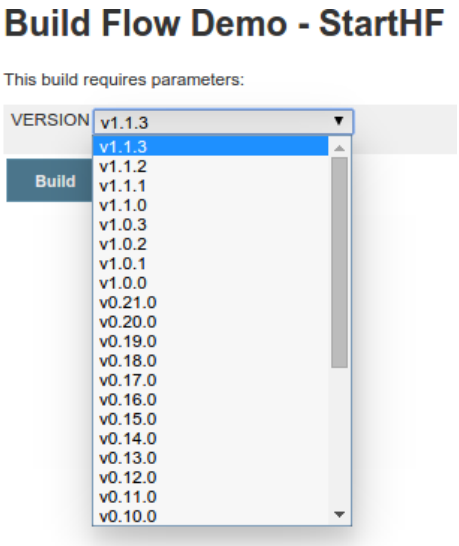
Slika 5.8: Nastavitve opravila za postavitev v produkcijsko okolje – nadaljevanje toka izvajanja na splošno opravilo za postavitev v produkcijsko okolje.



Slika 5.9: Nastavitve splošnega opravila za postavitev v produkcijsko okolje – klic skripte deploy.sh (Dodatek A, Slika A.4), ki izvede potrebne operacije za postavitev izbrane stisnjene datoteke v ciljno (produkcijsko) okolje.

primerih je možno aplikacijo brez težav vrniti na zadnjo delujočo verzijo s ponovno izvedbo koraka za postavitev v produkcijsko okolje, paziti moramo le, da izberemo ustrezno stisnjeno datoteko in s tem verzijo aplikacije, na katero se bomo vrnili (pravimo, da izvedemo *rollback*).

Druga možnost pa je, da v primeru manjših potrebnih popravkov sprožimo postopek hitrega popravka (*hotfix*). Ta se začne z izbiro verzije, na kateri želimo izvesti omenjen postopek (Slika 5.10). Opravilo za začetek izvajanja postopka hitrega popravka najprej prebere vhodne parametre (verzijo aplikacije v obliki vX.Y.Z) in preveri njihovo pravilnost (Slika 5.11). Sledi nadaljevanje toka izvajanja do splošnega opravila za omenjeni postopek (Slika 5.12), ki po koncu običajnih preverjanj sproži izvajanje skripte start-hf.sh (Slika 5.13). Skripta na Sliki A.6 (Dodatek A) na začetku prebere vse vhodne parametre in preveri njihovo pravilnost. Če pri tem ni bilo težav, se na integracijski strežnik Jenkins prenese kopija ustrezne verzije aplikacije iz podanega repozitorija (vse te informacije dobimo z vhodnimi parametri). Iz dobljene delovne mape ustvarimo novo vejo, ki jo poimenujemo glede na verzijo, ki smo jo izbrali, dodamo pa še oznako "HF", iz katere je takoj razvidno, da gre za vejo hitrega popravka (vX.Y.Z-HF). Če se veja uspešno ustvari, naredimo spremembo le še globalno vidno in postopek je zaključen. Sedaj se lahko lokalno, na svoji delovni postaji, preusmerimo na pravkar ustvarjeno vejo, in izvedemo popravek, ki je potreben za pravilno delovanje aplikacije. Ko s tem zaključimo, je na vrsti zaključitev hitrega popravka. Na Sliki 5.14 vidimo, da ta pred sprožitvijo zahteva izbiro veje, na kateri smo naredili popravek (vX.Y.Z-HF), ter ciljno vejo, na katero želimo omenjeni popravek prenesti. Glavni namen je namreč združiti HF vejo s ciljno *master* vejo. Opravilo na integracijskem strežniku Jenkins za zaključitev hitrega popravka prebere izbrane vhodne parametre ter preveri njihovo pravilnost (Slika 5.15), tok izvajanja pa se nato preusmeri na splošno opravilo za zaključitev hitrega popravka (Slika 5.16), ki začne z izvajanjem skripte finish-hf.sh za dokončanje postopka hitrega popravka (Slika 5.17). Kot običajno, tudi ta najprej opravi vse potrebno v zvezi z vhodnimi parametri, nato pa stanje delovne mape ponastavi na najnovejše možno iz podane ciljne veje. Sledi združitev veje s popravkom z omenjeno ciljno vejo. V primeru težav postopek prekinemo in vrnemo stanje nazaj na začetno, če pa se je združitev zaključila brez težav, spremembe prenesemo še na globalno raven in odstranimo vejo, na kateri smo iz-

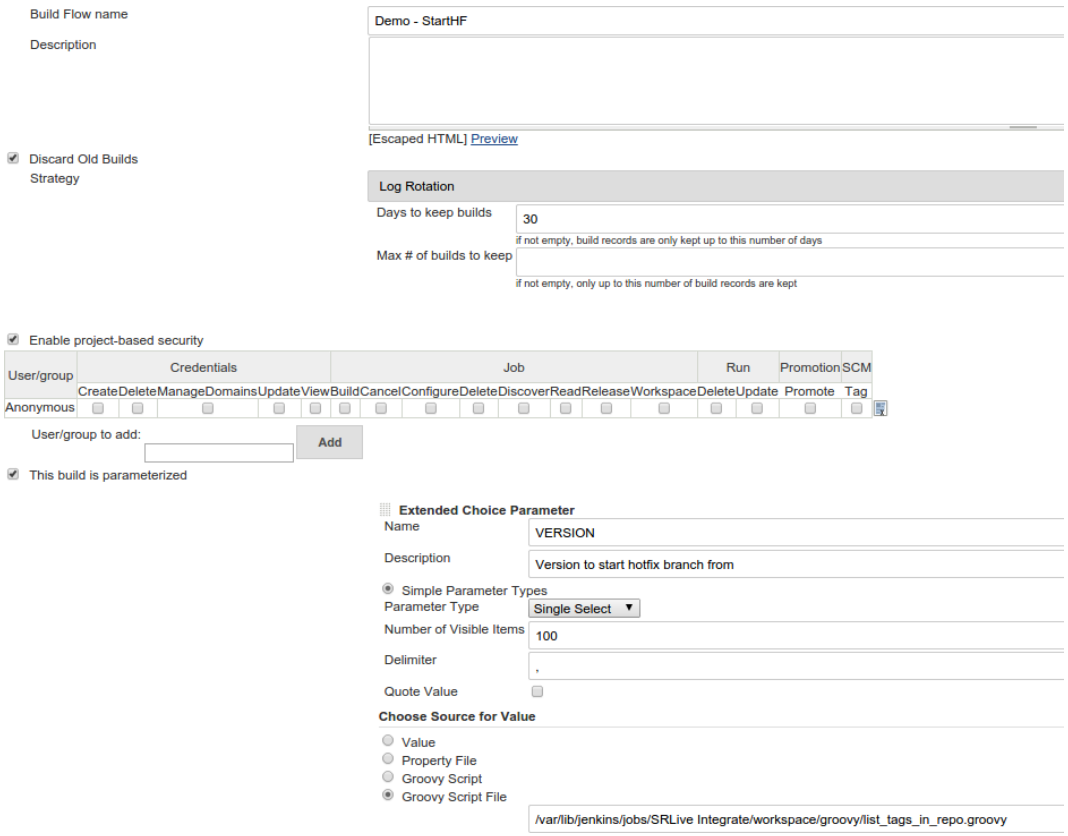


Slika 5.10: Sprožitev opravila za izvedbo hitrega popravka - izbor verzije aplikacije, na kateri želimo izvesti hitri popravek.

vedli popravek. Celotna skripta, ki se izvede, je vidna na Sliki A.7 (Dodatek A). Od tu naprej je postopek identičen kot pri običajni postavitvi - najprej izvedemo postavitev na pred-produkcijsko okolje in nato na produkcijsko, ali pa, v primeru, da tako želimo, kar direktno na produkcijsko okolje, kar sicer ni priporočljivo.

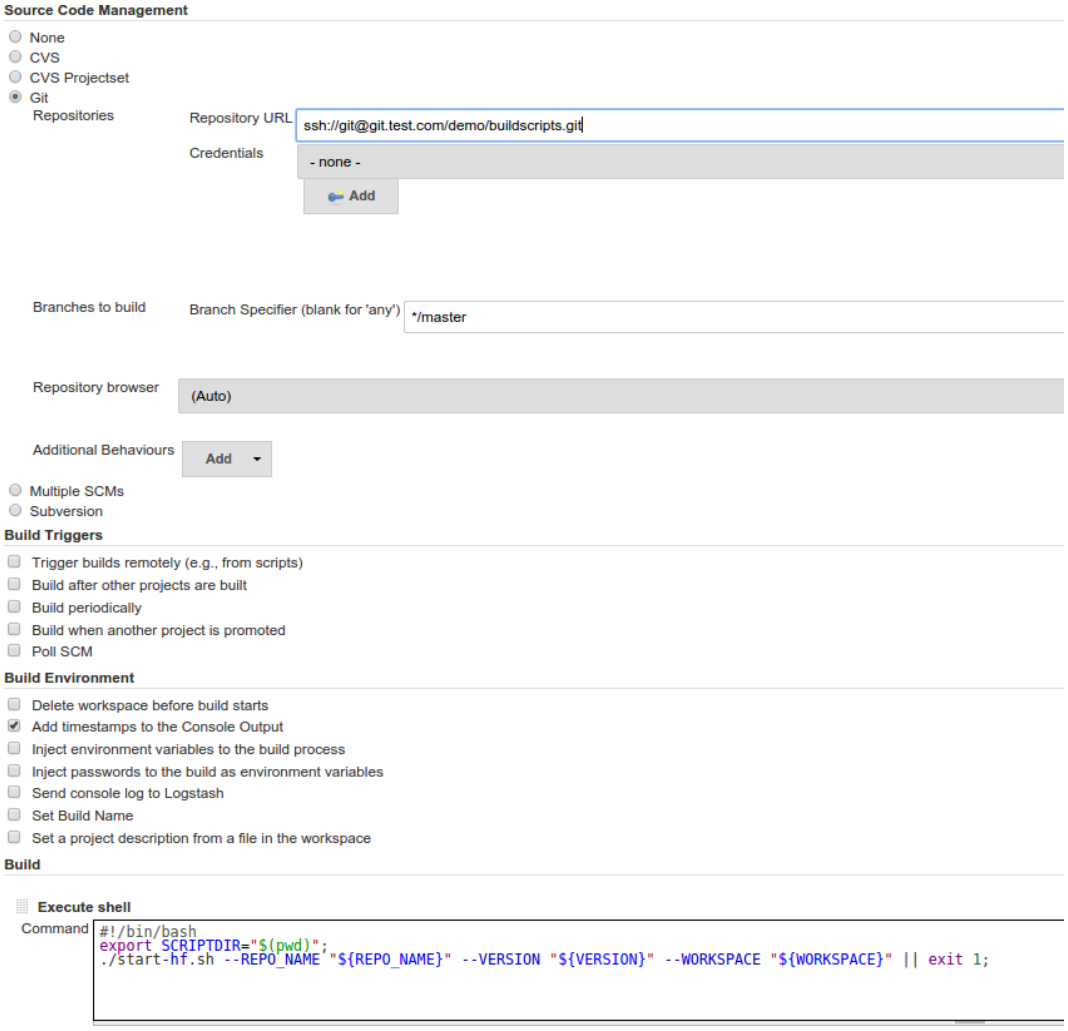
5.5 Vzdrževanje

Pri našem tipu aplikacij je najbolj pomembno ohraniti enako kakovost programske opreme tudi po koncu razvoja za vse stranke. Kljub temu, da to pomeni, da gre pri posameznih strankah večinoma le za razlike v izgledu aplikacije, je ravno zato veliko več možnosti, da se struktura strani poruši, prepišejo slogi oblikovanja, ipd. Ker je naročnikov veliko, je težko zagotoviti enako kakovost vsakemu posebej ob vsaki novi izdaji, zato smo v pomoč ekipi za zagotavljanje kakovosti razvili preprosto aplikacijo, ki pokaže, če so bile zaznane kakršnekoli razlike v izgledu strani, ki je v produkciji, in v izgledu strani, ki je pripravljena, da le-to nadomesti. Za te namene uporabljamo orodje PhantomCSS ([7]), ki izvaja regresijsko testiranje slogov oblikovanja (Cascading Style Sheets, [11]) s pomočjo modula CasperJS

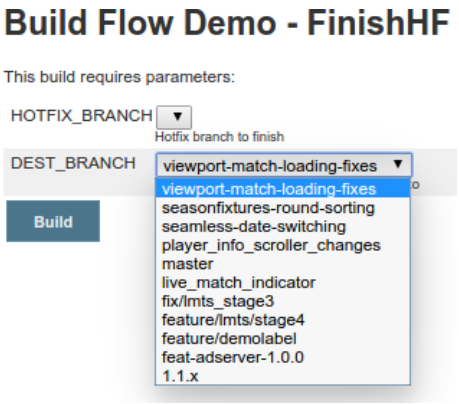


Slika 5.11: Nastavitve opravila za izvedbo hitrega popravka – dinamično polnjenje seznama za izbor verzije aplikacije, na kateri bomo izvedli hitri popravek.

Slika 5.12: Nastavitve opravila za izvedbo hitrega popravka – nadaljevanje toka izvajanja na splošno opravilo za izvedbo hitrega popravka.



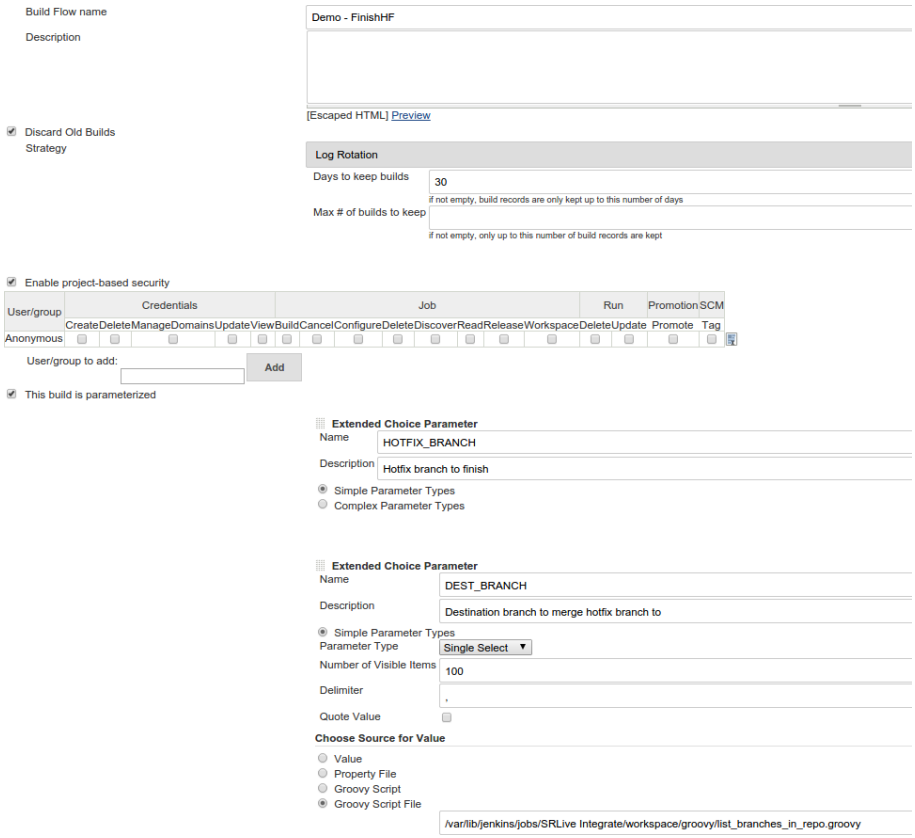
Slika 5.13: Nastavitve splošnega opravila za izvedbo hitrega popravka – klic skripte start-hf.sh (Dodatek A, Slika A.6), ki izvede potrebne operacije za izvedbo hitrega popravka na izbrani verziji aplikacije.



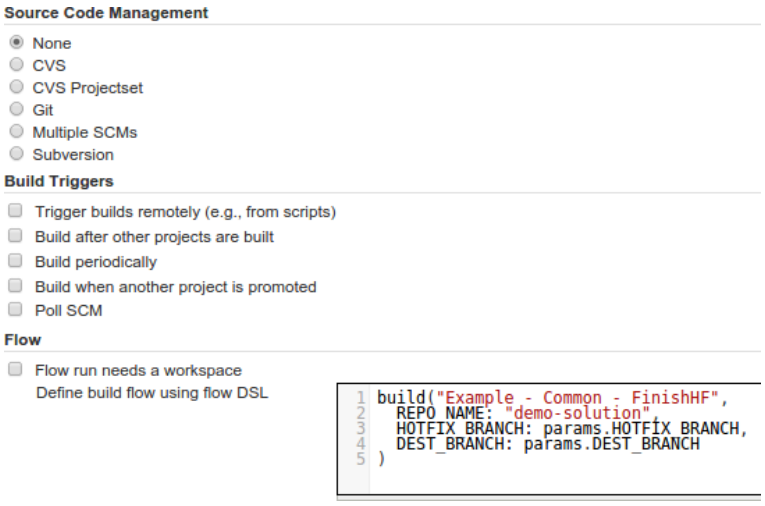
Slika 5.14: Sprožitev opravila za izvedbo zaključitve hitrega popravka - izbor veje, na kateri bomo naredili popravek (vX.Y.Z-HF), ter izbor ciljne veje, na katero želimo omenjeni popravek prenesti.

([9]), ki je namenjen izvajanju avtomatskih regresijskih vizualnih testov skupaj s knjižnicama PhantomJS ([8]) in Resemble.js ([10]). Pri regresijskem testiranju gre za testiranje sprememb v programski opremi v primerjavi s prejšnjo različico le-te. S tem si zagotovimo, da s stališča združljivosti, te ne bodo povzročile težav. Kot je navedeno v [13], naj bi se tak način testiranja uporabljal pri vseh običajnih razvojnih procesih. Pred vsako novo izdajo programske opreme je zato potrebno izvesti teste, ki trenutne rezultate primerjajo z novimi rezultati, pridobljenimi pri testiranju spremenjene verzije aplikacije. Rezultat, ki ga dobimo na koncu pove, ali so nove spremembe združljive s starimi ali ne. Velikokrat se namreč ob spreminjanju doda neka programska koda, ki nenamerno vpliva še na določene druge komponente, in s tem povzroči nepravilno delovanje ali celo nedelovanje aplikacije.

Kako deluje? PhantomCSS vzame posnetek zaslona strani za primerjavo, ki jih priskrbi CasperJS. Te posnetke nato primerja med sabo s pomočjo knjižnice Resemble.js, ki izvede test iskanja razlik v RGB slikovnih pikah. PhantomCSS nato na podlagi rezultatov vizualno prikaže razlike v posnetkih, kar omogoči lažje odkrivanje napak. Tak način regresijkega testiranja je najbolj uporaben, ko je uporabniški vmesnik aplikacije predvidljiv, kar pomeni, da se lahko v primeru razlik v vsebini strani pojavijo manjše težave in posledično testiranje ne služi več svojemu namenu.



Slika 5.15: Nastavitve opravila za izvedbo zaključitve hitrega popravka – dinamično polnjenje seznama za izbor veje, na kateri bomo naredili popravek, ter za izbor ciljne veje, na katero želimo omenjeni popravek prenesti.

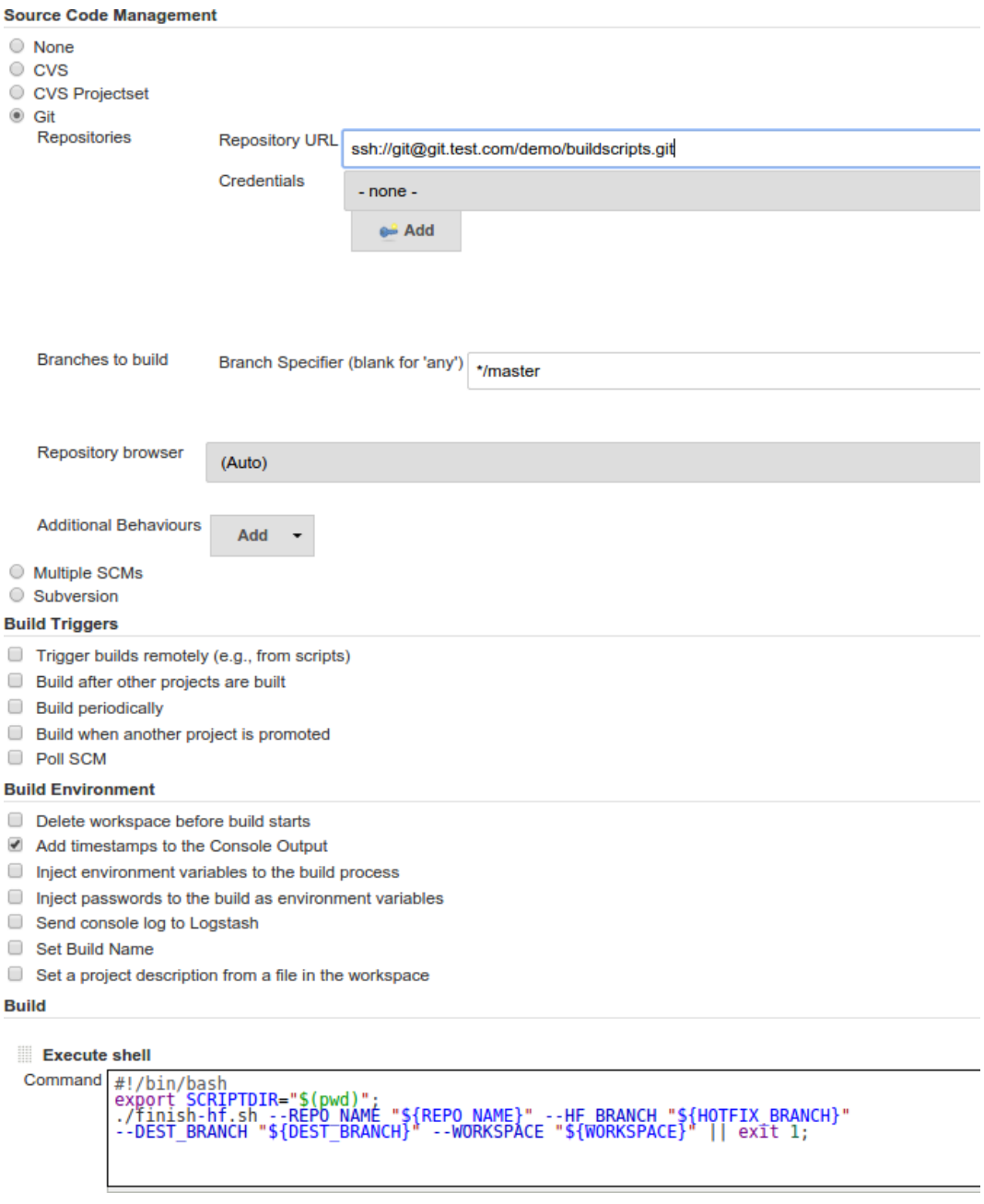


Slika 5.16: Nastavitve opravila za izvedbo zaključitve hitrega popravka – nadaljevanje toka izvajanja na splošno opravilo za izvedbo zaključitve hitrega popravka.

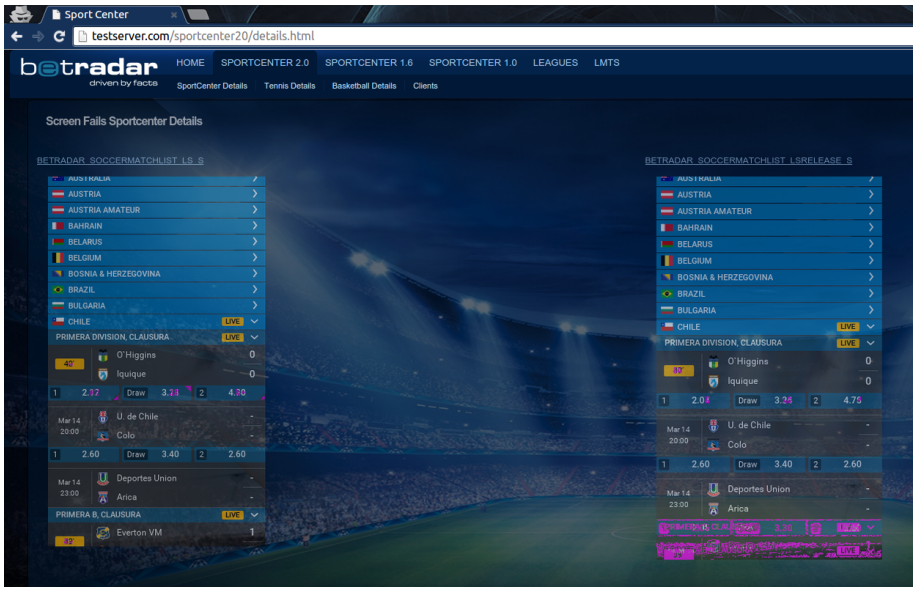
Prikaz v praksi Slika 5.18 prikazuje odkrite razlike s pomočjo PhantomCSS orodja, ki so na desnem posnetku označene z roza barvo. Primer uspešno prestanega testa pa je na Sliki 5.19, kjer je prikazana primerjava določenih komponent treh različic razvite spletne aplikacije.

5.6 Rezultati

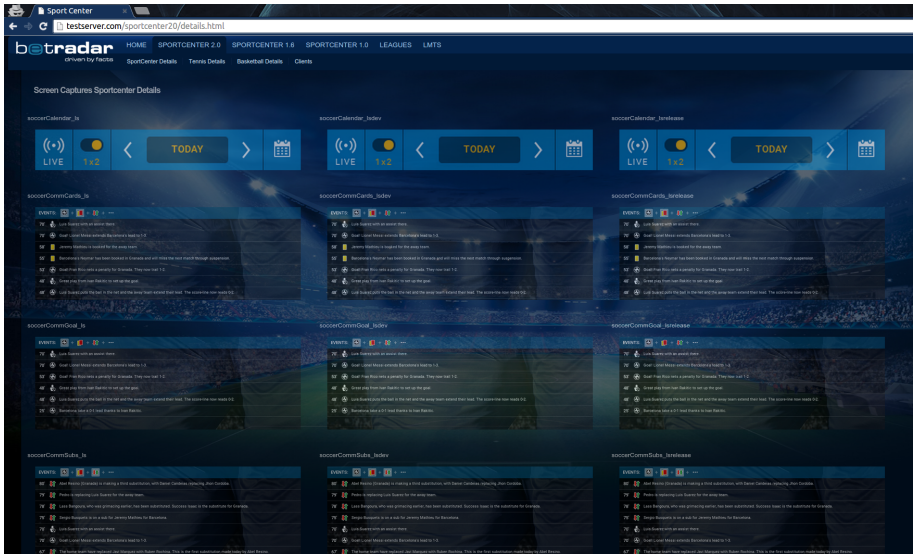
Spremenjen proces sprotne dostave, opisan v prejšnjih poglavjih, je bil vzpostavljen na osnovi že obstoječega, ki se še vedno uporablja za zagotavljanje sprotne dostave starih rešitev (Slika 5.20). Razvoj (ali vzdrževanje) ves čas poteka na razvojni veji (*develop*), iz katere se vsak ponedeljek naredi nova veja, namenjena izdaji (*release*). V naslednjem koraku se izvede integracija *release* veje v pred-produkcijsko (*staging*) okolje – najprej prevajanje kode in nato še postavitev. Vse do srede na tej stopnji poteka podrobno preverjanje in testiranje. V primeru odkritih težav oziroma napak obstaja možnost izvedbe hitrih popravkov, ki se nato še vedno lahko vključijo v sredino izdajo. V sredo dopoldne se izvede proces izdaje, ki vključuje ponovno prevajanje kode ter postavitev v produkcijsko okolje.



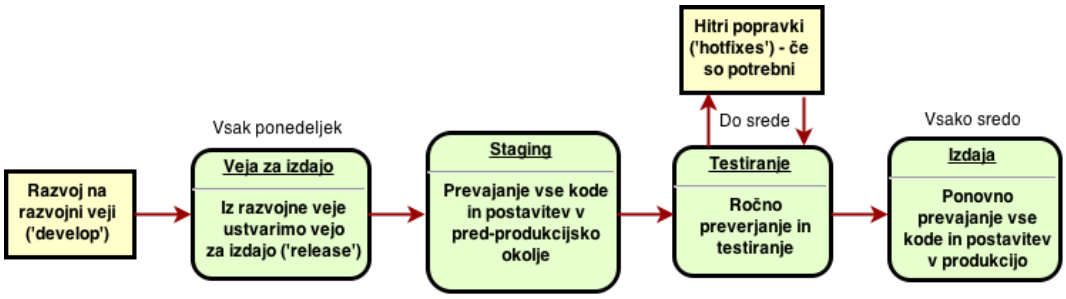
Slika 5.17: Nastavitve splošnega opravila za izvedbo zaključitve hitrega popravka – klic skripte finish-hf.sh (Dodatek A, Slika A.7), ki izvede potrebne operacije za izvedbo zaključitve hitrega popravka.



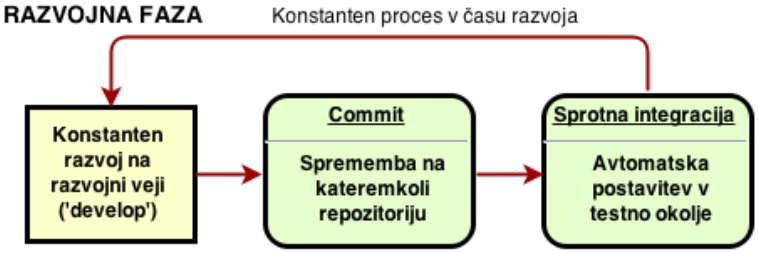
Slika 5.18: Odkrite razlike s pomočjo orodja PhantomCSS označene z roza barvo na desnem posnetku zaslona.



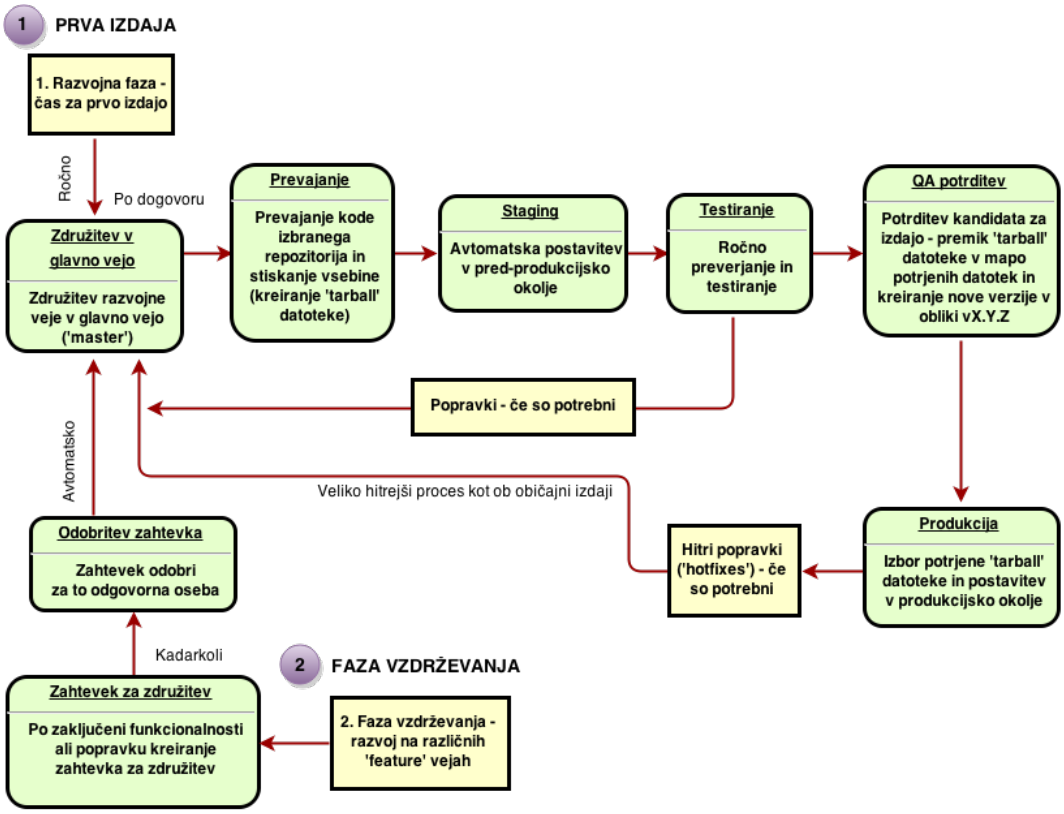
Slika 5.19: Primerjava določenih komponent treh različic razvite spletne aplikacije z orodjem PhantomCSS.



Slika 5.20: Prejšnji način implementacije cevovodnega procesa sprotne dostave.



Slika 5.21: Novi način implementacije cevovodnega procesa sprotne dostave - razvojna faza.



Slika 5.22: Novi način implementacije cevovodnega procesa sprotne dostave - podrobno.

Med starim in novim načinom seveda obstajajo podobnosti. Na Sliki 5.21 je prikazan osnovni proces sprotne integracije v za to pripravljeno testno okolje, s Slike 5.22 pa je razviden nadaljnji proces, ki izhaja iz starega (Slika 5.20), le da je razdrobljen na več manjših korakov (podroben opis je na voljo v prejšnjih podpoglavjih). Z uporabo novih, a vedno istih skript lahko izbran repozitorij postavimo na katerokoli okolje želimo povsem neodvisno od preostalih. S tem smo pridobili večjo fleksibilnost pri izdajah, saj so le-te praktično možne kadarkoli to želimo oziroma kadarkoli je potrebno.

Pridobitve z novim procesom:

- možne pogostejše izdaje zaradi razbitja aplikacije na več repozitorijev,
- hitrejši postopek sprotne dostave (približno 1-minutna pohitritev) predvsem zaradi le enkratnega prevajanja ter zaradi prevajanja samo tistega dela kode, kjer je to res potrebno,
- uvedba preverjanja kvalitete kode z uporabo zahtevkov za združitev (*merge requests*),
- hitrejša odprava napak in manjše tveganje pri prvi izdaji zaradi konstantnega integriranja že med samim razvojem.

Slabosti pri uporabi novega procesa:

- večja kompleksnost zaradi razbitja aplikacije na več repozitorijev,
- še vedno ni avtomatskega testiranja.

5.7 Možne izboljšave

Opisani proces je le približek tistemu, ki ga najdemo v [1], saj se je vseh korakov sprotne dostave pri razvoju spletne aplikacije težko dosledno držati, pojavlja pa se tudi vprašanje, če je to sploh smiselno. Ker v proces nismo vključili avtomatskega testiranja, razen preverjanja sintakse, kar pri skriptnih jezikih še vedno ni podprto tako, kot bi si želeli, veliko več odgovornosti pade na ekipo za zagotavljanje

kakovosti, ki mora tako pazljivo ves čas izvajati faze ročnega testiranja, da čimveč napak odkrije pravočasno in ne, ko so spremembe že v produkciji. Prva možna izboljšava je torej vzpostavitev postopka avtomatskega testiranja.

Po tem, ko se konča razvoj in začne vzdrževanje programske opreme, razvojna veja običajno ne služi več svojemu namenu tako kot med samim razvojem. Potrebno je še vedno ohraniti nekakšen sistem, ki dobro funkcionira, zato je pametno uvesti različne veje, ki vedno izhajajo iz glavne, na katerih se razvijajo razne nove funkcionalnosti, izboljšujejo obstoječe, ali pa le odpravljajo še vedno prisotne napake. Vsaka dokončana naloga na ločeni veji, ki se običajno poimenuje po tej nalogi, se posledično združi z glavno vejo, kar pa ni več ročen postopek. Ob zaključku je namreč potrebno oddati zahtevek za združitev (za te namene se lahko uporabi GitLab ([5])), ki ga lahko odobrijo le člani ekipe, ki so za to zadolženi, nikoli pa oseba zahtevka ne odobri sama sebi. Ko je zahtevek s strani odgovorne osebe odobren, se izvede avtomatska združitev ločene veje v glavno. Od tu dalje je postopek sprotne dostave enak.

Zadnja predlagana izboljšava pa je uporaba Selenium testov ([6]). Testi se lahko uporabijo, ko je razvita programska oprema že v fazi vzdrževanja, kar je dobra dopolnitev obstoječega načina preverjanja sprememb v izgledu aplikacije (gre za preverjanje DOM (Document Object Model) strukture spletne aplikacije v primeru kakšne nepredvidene spremembe, ki to strukturo lahko poruši).

Poglavje 6

Zaključek

V diplomski nalogi je predstavljen proces, ki je bil zasnovan po zgledu iz [1], na osnovi omenjene knjige pa celotna naloga tudi temelji. Po večkrat videnih in doživetih težavah pri zagotavljanju sprotne in nemotene dostave programske opreme strankam ter končnim uporabnikom, je bila vpeljava avtomatizacije celotnega postopka neizogiben korak naprej. Opisani praktični del uporabe postopka sprotne dostave se trenutno še z nekaterimi dodanimi izboljšavami, konstantno uporablja v praksi in prinaša mnoge prednosti. Ene izmed najpomembnejših so seveda izdaje z nično časovno zakasnitvijo ter možnost neopaznega sestopa na prejšnjo verzijo v primeru težav.

Eden izmed glavnih razlogov proti uporabi opisanega postopka sprotne dostave v praksi je ponavadi, da gre za proces, ki ga enostavno ni možno popolnoma izkoristiti, ter da je to zgolj idealistična ideja, ki lahko uspe le v popolnem svetu. Kot avtorja v [1] velikokrat omenita, cilj ni do potankosti slediti vsem korakom in navodilom, temveč se jim le čimbolj približati ter iz njih glede na tip aplikacije, ki jo razvijamo, potegniti najboljše, in predvsem tisto, kar resnično potrebujemo. Glede na dosedanje izkušnje je bil proces zastavljen dovolj dobro, da smo s tem olajšali in pohitrili celoten postopek od prve integracije do končne izdaje. Ker je kar nekaj korakov avtomatiziranih, člani razvojne ekipe lahko veliko več svojega časa namenijo izboljšanju kakovosti programske kode ter razvoju novih in boljših funkcionalnosti same aplikacije. Na voljo je še veliko prostora za izboljšave (na primer vpeljava cevovodnega procesa postavitve z uporabo posebnega tipa opravil na integracijskem strežniku Jenkins), kar je tudi del načrta za prihodnje mesece v

fazi vzdrževanja. Nov, glavni cilj pa je tudi, da se ideja predstavi še vsem ostalim razvojnim ekipam v podjetju, ki avtomatiziranega procesa sprotne dostave še ne uporabljajo v praksi.

Literatura

[1] J. Humble, D. Farley, *Continuous delivery: reliable software releases through build, test, and deployment automation.* Addison-Wesley, 2011.

[2] Git, "http://git-scm.com".

[3] JSHint, "http://jshint.com/about/".

[4] rsync, "http://linux.die.net/man/1/rsync".

[5] GitLab, "http://about.gitlab.com/".

[6] SeleniumHQ - Browser Automation, "http://www.seleniumhq.org/".

[7] PhantomCSS, "http://github.com/Huddle/PhantomCSS".

[8] PhantomJS, "http://github.com/ariya/phantomjs/".

[9] CasperJS, "http://casperjs.org/".

[10] Resemble.js, "http://huddle.github.io/Resemble.js/".

[11] Cascading Style Sheets (CSS), "http://en.wikipedia.org/wiki/CSS".

[12] Jenkins, an extensible open source continuous integration server, "http://jenkins-ci.org/".

[13] Regression Testing, "http://searchsoftwarequality.techtarget.com/definition/regression-testing".

[14] Extreme Programming, "http://goo.gl/1YCcQW".

Dodatek A

A Skripte za gradnjo

Podroben prikaz skript za gradnjo, ki se uporabljajo skozi celoten proces sprotne integracije, postavitve ter izdaje:

- integrate.sh (Slika A.1)
- checkout.sh (Slika A.2)
- compress.sh (Slika A.3)
- deploy.sh (Slika A.4)
- confirm.sh (Slika A.5)
- start-hf.sh (Slika A.6)
- finish-hf.sh (Slika A.7)

```
#!/bin/bash
set -ex;

COMPRESS="no" # not mandatory, default value 'no'

#####################################################
################## SCRIPT PARAMETERS ################
#####################################################
OPTS=`getopt -o rbwctn: --long REPO_NAME,BRANCH,WORKSPACE,COMPRESS,TARGET,BUILD_NR: -n 'parse-options' -- "$@"`
if [ $? != 0 ] ; then echo "Failed parsing options." >&2 ; exit 1 ; fi

## Read parameter values.
while true; do
  case "$1" in
    -r|--REPO_NAME)
        case "$2" in
            "") REPO_NAME= ; shift 2 ;;
            *) REPO_NAME=$2 ; shift 2 ;;
        esac ;;
    -b|--BRANCH)
        case "$2" in
            "") BRANCH= ; shift 2 ;;
            *) BRANCH=$2 ; shift 2 ;;
        esac ;;
    -w|--WORKSPACE)
        case "$2" in
            "") WORKSPACE= ; shift 2 ;;
            *) WORKSPACE=$2 ; shift 2 ;;
        esac ;;
    -c|--COMPRESS)
        case "$2" in
            "") COMPRESS='no' ; shift 2 ;;
            "yes") COMPRESS="yes" ; shift 2 ;;
            "no") COMPRESS="no" ; shift 2 ;;
            *) shift 2 ;;
        esac ;;
    -t|--TARGET)
        case "$2" in
            "") TARGET= ; shift 2 ;;
            *) TARGET=$2 ; shift 2 ;;
        esac ;;
    -n|--BUILD_NR)
        case "$2" in
            "") BUILD_NR= ; shift 2 ;;
            *) BUILD_NR=$2 ; shift 2 ;;
        esac ;;
    --) shift ; break ;;
    *) break ;;
  esac
done
## Check mandatory parameters. Exit if not set.
if ! [[ -n "$REPO_NAME" ]]; then echo "'REPO_NAME' param not set!"; exit 1; fi
if ! [[ -n "$WORKSPACE" ]]; then echo "'WORKSPACE' param not set!"; exit 1; fi
if ! [[ -n "$BRANCH" ]]; then echo "'BRANCH' param not set!"; exit 1; fi
if ! [[ -n "$BUILD_NR" ]]; then echo "'BUILD_NR' param not set!"; exit 1; fi
#####################################################
#####################################################
#####################################################

export TMPDIR="$WORKSPACE/tmp/srlive-integrate/tmp";
if [[ ! -d "$TMPDIR" ]]; then mkdir -p "$TMPDIR"; fi

source "$SCRIPTDIR/repo-factory.sh" || exit 1;

REPOSITORY=`getRepo $REPO_NAME`;

# Preapre tarball path if compress param is set to yes.
if [[ "$COMPRESS" = "yes" ]]; then
    TARBALL_MAIN="$TMPDIR/workspace/tarball/${REPO_NAME}";
    if [[ ! -d "$TARBALL_MAIN" ]]; then mkdir -p "$TARBALL_MAIN"; fi # prepare tarball folders
    if [[ ! -d "$TARBALL_MAIN/all" ]]; then mkdir -p "$TARBALL_MAIN/all"; fi
    if [[ ! -d "$TARBALL_MAIN/confirmed" ]]; then mkdir -p "$TARBALL_MAIN/confirmed"; fi

    DATE=$(date +"%Y-%m-%d");
    TARBALL_PATH="$TARBALL_MAIN/all/build-${BUILD_NUMBER}-${DATE}";
fi

TARGET_ORIG=$TARGET;
# Prepare target parameters, needed for deploy.
TARGET_PARAMS=`/bin/bash "$SCRIPTDIR/target-parser.sh" --TARGET "$TARGET" || exit 1`;
TARGET=$(echo "$TARGET_PARAMS" | awk -F'|' '{print $1}');
if [[ "$TARGET" = "nothing" ]]; then TARGET=; fi

LOG_FILENAME="$REPO_NAME-$BRANCH";
SRC="$TMPDIR/workspace/src/deploy";
SOURCEPATH="$TMPDIR/workspace/src/$REPO_NAME";
LOCK_FILE_PATH="$TMPDIR/workspace/src/$REPO_NAME.lock";

if [[ "$FORCE" = "yes" ]]; then
    echo "Forcing sourcepath and lock file clean up...";
    if [[ -d "$SOURCEPATH" ]]; then rm -rf "$SOURCEPATH"; fi
    if [[ -f "$LOCK_FILE_PATH" ]]; then rm "$LOCK_FILE_PATH"; fi
fi

if [[ ! -d "$SRC" ]]; then mkdir -p "$SRC"; fi

#################################
########## TRAP HANDLING ########
function clean_up {
    echo "Exception caught. Cleaning up...";
    if [[ -f "$LOCK_FILE_PATH" ]]; then rm "$LOCK_FILE_PATH"; fi
    if [[ -f "$TARBALL_PATH" ]]; then rm "$TARBALL_PATH"; fi
```

```
    trap - EXIT;
    exit 1;
}
trap clean_up SIGHUP SIGINT SIGTERM EXIT;
##################################
##################################

# If compressed tarball file already exists, exit.
if [[ -f "$TARBALL_PATH" ]]; then
    echo "Tarball file already exists. Exiting...";
    exit 0;
fi

# Check if repository is being used by another script.
if [[ -f "$LOCK_FILE_PATH" ]]; then
    echo "Repository is in use by another script, please try again later.";
    exit 1;
else
    # Lock repository so other scripts can't change it while current one runs.
    echo "Repository locked." > "$LOCK_FILE_PATH";
fi

# Clone repository & checkout desired branch.
/bin/bash "$SCRIPTDIR/checkout.sh" -r "$REPOSITORY" -s "$SOURCEPATH" -b "$BRANCH" || exit 1;

# Compile for deploy.
/bin/bash "$SCRIPTDIR/assets-cacheing.sh" -c "yes" -s "$SOURCEPATH" || exit 1; # avoid unnecessary assets compiling
/bin/bash "$SCRIPTDIR/compile-for-deploy.sh" -s "$SOURCEPATH" -c "$COMPRESS" -w "$TMPDIR" -l "$LOG_FILENAME" -r "$REPO_NAME" -b "$BRA

# Remember last commit hash.
cd "$SOURCEPATH";
COMMIT_HASH=$(git rev-parse $BRANCH);
DATETIME=$(date +"%Y-%m-%d-%T");
echo "Writting '$COMMIT_HASH-$DATETIME' into revision.html file...";
echo "$COMMIT_HASH-$DATETIME" > "$SOURCEPATH/htdocs/revision.html";

# Compress if COMPRESS parameter says so and TARBALL_PATH is not null
if [[ "$COMPRESS" = "yes" && -n "$TARBALL_PATH" ]]; then
    /bin/bash "$SCRIPTDIR/compress.sh" -t "$TARBALL_PATH" -s "$SOURCEPATH" || exit 1;
fi
echo "Created tarball file: $TARBALL_PATH";

# Deploy to target.
if [[ -n "$TARGET" ]]; then
    echo "Deploying...";
    /bin/bash "$SCRIPTDIR/deploy.sh" --COMPRESS "$COMPRESS" --SOURCEPATH "$SOURCEPATH" --TARBALL "$TARBALL_PATH" --TARGET "$TARGET_OR
fi

/bin/bash "$SCRIPTDIR/assets-cacheing.sh" -c "no" -s "$SOURCEPATH" || exit 1; # avoid unnecessary assets compiling

# Free repository so other scripts can use it from now on.
if [[ -f "$LOCK_FILE_PATH" ]]; then rm "$LOCK_FILE_PATH"; fi

trap - EXIT;
```

Slika A.1: Integracijska skripta integrate.sh.

```
#!/bin/bash
set -ex;

while getopts "r:s:b:" opt; do
    case $opt in
        r) REPOSITORY="$OPTARG"
        ;;
        s) SOURCEPATH="$OPTARG"
        ;;
        b) BRANCH="$OPTARG"
        ;;
    esac
done

if ! [[ -d "$SOURCEPATH" ]]; then
    git clone "$REPOSITORY" "$SOURCEPATH";
fi

# Build and update.
cd "$SOURCEPATH";
git fetch --all --prune --tags;
git checkout "$BRANCH";
git pull origin "$BRANCH";

make partial || exit 1;
php composer.phar install --optimize-autoloader || exit 1;
make assets-compiling || exit 1;
```

Slika A.2: Skripta checkout.sh - delo z izvorno kodo.

```
#!/bin/bash
set -ex;

while getopts "t:s:" opt; do
    case $opt in
        t) TARBALL="$OPTARG"
        ;;
        s) SOURCEPATH="$OPTARG"
        ;;
    esac
done

TARBALL_DIR=`echo $(dirname "$TARBALL")`;

if ! [[ -d "$TARBALL_DIR" ]]; then
    mkdir "$TARBALL_DIR";
fi

# Clean up.
/bin/bash "$SCRIPTDIR/cleanup.sh" -t "$TARBALL_DIR" -d "14" -n "10" -f "all";
/bin/bash "$SCRIPTDIR/cleanup.sh" -t "$TARBALL_DIR" -d "-1" -n "10" -f "confirmed";

# Compress
if hash tar >/dev/null; then
    echo "Trying to compress into $TARBALL_DIR.";
    cd "$SOURCEPATH" && time tar --exclude-from "application/exclude-from-deploy.txt" -pczf "$TARBALL" .;
else
    echo "Error: 'tar' is currently not installed but is required for this script to finish succesfully.
    You can install it by typing: 'sudo apt-get install tar'." >&2 ; exit 1 ;
fi

echo "File compressing finished succesfully."
```

Slika A.3: Skripta compress.sh - stiskanje delovne mape.

```
#!/bin/bash
set -ex;

#####################################################
################## SCRIPT PARAMETERS ################
#####################################################
OPTS=`getopt -o scwbt: --long SOURCEPATH,COMPRESS,WORKSPACE,TARBALL,TARGET: -n 'parse-options' -- "$@"`
if [ $? != 0 ] ; then echo "Failed parsing options." >&2 ; exit 1 ; fi
## Read parameter values.
while true; do
  case "$1" in
      -s|--SOURCEPATH)
        case "$2" in
            "") SOURCEPATH= ; shift 2 ;;
            *) SOURCEPATH=$2 ; shift 2 ;;
        esac ;;
      -c|--COMPRESS)
        case "$2" in
            "") COMPRESS= ; shift 2 ;;
            *) COMPRESS=$2 ; shift 2 ;;
        esac ;;
    -w|--WORKSPACE)
        case "$2" in
            "") WORKSPACE= ; shift 2 ;;
            *) WORKSPACE=$2 ; shift 2 ;;
        esac ;;
    -b|--TARBALL)
        case "$2" in
            "") TARBALL= ; shift 2 ;;
            *) TARBALL=$2 ; shift 2 ;;
        esac ;;
    -t|--TARGET)
        case "$2" in
            "") TARGET= ; shift 2 ;;
            *) TARGET=$2 ; shift 2 ;;
        esac ;;
    --) shift ; break ;;
    *) break ;;
  esac
done
## Check mandatory parameters. Exit if not set.
if [[ "$COMPRESS" = "no" && -z "$SOURCEPATH" ]]; then echo "'SOURCEPATH' param not set!"; exit 1; fi
if [[ "$COMPRESS" = "yes" && -z "$TARBALL" ]]; then echo "'TARBALL' param not set!"; exit 1; fi
if ! [[ -n "$TARGET" ]]; then echo "'TARGET' param not set!"; exit 1; fi
if ! [[ -n "$WORKSPACE" ]]; then echo "'WORKSPACE' param not set!"; exit 1; fi

export TMPDIR="$WORKSPACE/tmp/srlive-integrate/tmp";
if [[ ! -d "$TMPDIR" ]]; then mkdir -p "$TMPDIR"; fi

WS="$TMPDIR";
SRC_TAR="$WS/workspace/confirm-temp";

if [[ -n "$TARBALL" ]]; then
    REPO_FILE_NAME=`echo "$TARBALL" | awk -F'/' '{print $(NF-2)}'`; # get repo folder name
    echo "Repo folder name, extracted from tarball path is '$REPO_FILE_NAME'";
elif [[ -n "$SOURCEPATH" ]]; then
    REPO_FILE_NAME=`echo "$SOURCEPATH" | awk -F'/' '{print $NF}'`; # get repo folder name
    echo "Repo folder name, extracted from sourcepath is '$REPO_FILE_NAME'";
fi
LOCK_FILE_PATH="$WS/workspace/src/$REPO_FILE_NAME.lock";

# Prepare target parameters, needed for deploy.
TARGET_ORIG=$(echo "$TARGET" | awk -F'-' '{print $1}');

TARGET_PARAMS=`/bin/bash "$SCRIPTDIR/target-parser.sh" --TARGET "$TARGET" || exit 1`;
TARGET=$(echo "$TARGET_PARAMS" | awk -F'|' '{print $1}');
MCO_PRODUCT=$(echo "$TARGET_PARAMS" | awk -F'|' '{print $2}');
MCO_PROJECT=$(echo "$TARGET_PARAMS" | awk -F'|' '{print $3}');
MCO_ROLE=$(echo "$TARGET_PARAMS" | awk -F'|' '{print $4}');
MCO_MEMCACHE=$(echo "$TARGET_PARAMS" | awk -F'|' '{print $5}');
CLEAR_MEMCACHE=$(echo "$TARGET_PARAMS" | awk -F'|' '{print $6}');
if [[ "$TARGET" = "nothing" ]]; then TARGET=; fi
if [[ "$MCO_PRODUCT" = "nothing" ]]; then MCO_PRODUCT=; fi
if [[ "$MCO_PROJECT" = "nothing" ]]; then MCO_PROJECT=; fi
if [[ "$MCO_ROLE" = "nothing" ]]; then MCO_ROLE=; fi
if [[ "$MCO_MEMCACHE" = "nothing" ]]; then MCO_MEMCACHE=; fi
if [[ "$CLEAR_MEMCACHE" = "nothing" ]]; then CLEAR_MEMCACHE=; fi

###################################
########## TRAP HANDLING ##########
function clean_up {
    echo "Exception caught. Cleaning up...";
    if [[ -f "$LOCK_FILE_PATH" ]]; then rm "$LOCK_FILE_PATH"; fi
    if [[ -d "$SRC_TAR" ]]; then rm -rf "$SRC_TAR/*"; fi
    trap - EXIT;
    exit 1;
}
trap clean_up SIGHUP SIGINT SIGTERM EXIT;
###################################
###################################

if [[ "$COMPRESS" = "yes" && ! -f "$TARBALL" ]]; then echo "Tarball file does not exist!"; exit 1; fi

# Check directory structure.
if ! [[ -d "$WS/workspace" ]]; then mkdir "$WS/workspace"; fi
if ! [[ -d "$WS/workspace/confirm-temp" ]]; then mkdir "$WS/workspace/confirm-temp"; fi
```

```
if ! [[ -d "$SRC_TAR" ]]; then mkdir "$SRC_TAR"; else rm -rf "$SRC_TAR/*"; fi

# Extract tarball into $SRC.
if [[ "$COMPRESS" = "yes" ]]; then
    if [[ -n "$SOURCEPATH" ]]; then
        # this happens in case integrate job runs deploy and wants to create tarball so it's best to use existing workspace
        SRC_TAR="$SOURCEPATH";
    else
        # extract given tarball (used in case of deploy only)
        cd "$SRC_TAR" && time tar -xzf "$TARBALL";
    fi
else
    # use existing workspace (used in case of complete integrate process)
    SRC_TAR="$SOURCEPATH";
fi

SSHPARAMS=
RSYNC_DIRECTORY="$MCO_PRODUCT/$MCO_PROJECT/$MCO_ROLE";

RSYNC_TARGET="$TARGET:$RSYNC_DIRECTORY/";
ssh $SSHPARAMS $TARGET "mkdir -p $RSYNC_DIRECTORY";

echo "Deploying to '$TARGET:$RSYNC_DIRECTORY/'...";

if [[ "$COMPRESS" = "yes" ]]; then
    if [[ -n "$SOURCEPATH" ]]; then
        time rsync -az --out-format='%n%L' --stats --delete-after -e "ssh $SSHPARAMS" --exclude-from
        "$SRC_TAR/application/exclude-from-deploy.txt" "$SRC_TAR/" "$RSYNC_TARGET";
    else
        time rsync -az --out-format='%n%L' --stats --delete-after -e "ssh $SSHPARAMS" "$SRC_TAR/" "$RSYNC_TARGET";
    fi
else
    time rsync -az --out-format='%n%L' --stats --delete-after -e "ssh $SSHPARAMS" --exclude-from
    "$SRC_TAR/application/exclude-from-deploy.txt" "$SRC_TAR/" "$RSYNC_TARGET";
fi

if [[ -n "$MCO_ROLE" ]]; then
    if hash mco >/dev/null; then
        time mco rpc deploy_new deploy_parameterized parameter=$MCO_PRODUCT -W project=$MCO_PROJECT -W role=$MCO_ROLE --dt 15;
    else
        echo "The program 'mco' is currently not installed but is required for this script to finish succesfully.
        You can install it by typing: 'sudo apt-get install mcollective-client'." 1>&2;
    fi
fi

# Memcache clear.
/bin/bash "$SCRIPTDIR/memcache-clear.sh" -p "$MCO_PROJECT" -r "$MCO_ROLE" -c "$CLEAR_MEMCACHE";
# Varnish clear.
CLEARBACKVARNISH='/varnish/';
CLEARFRONTVARNISH='/frontvarnish/';
VARNISHPARAM="purge domain=$TARGET ORIG";
/bin/bash "$SCRIPTDIR/varnish-clear.sh" -b "$CLEARBACKVARNISH" -f "$CLEARFRONTVARNISH" -p "$VARNISHPARAM";

trap - EXIT;
```

Slika A.4: Skripta deploy.sh za postavitev na izbran strežnik.

```
#!/bin/bash
set -ex;

#####################################################
################## SCRIPT PARAMETERS ################
#####################################################
OPTS=`getopt -o tw: --long TARBALL,WORKSPACE: -n 'parse-options' -- "$@"`
if [ $? != 0 ] ; then echo "Failed parsing options." >&2 ; exit 1 ; fi

## Read parameter values.
while true; do
  case "$1" in
    -t|--TARBALL)
        case "$2" in
            "") TARBALL= ; shift 2 ;;
            *) TARBALL=$2 ; shift 2 ;;
        esac ;;
    -w|--WORKSPACE)
        case "$2" in
            "") WORKSPACE= ; shift 2 ;;
            *) WORKSPACE=$2 ; shift 2 ;;
        esac ;;
    --) shift ; break ;;
    *) break ;;
  esac
done
## Check mandatory parameters. Exit if not set.
if ! [[ -n "$TARBALL" ]]; then echo "'TARBALL' param not set!"; exit 1; fi
if ! [[ -n "$WORKSPACE" ]]; then echo "'WORKSPACE' param not set!"; exit 1; fi
#####################################################
#####################################################
#####################################################

export TMPDIR="$WORKSPACE/tmp/srlive-integrate/tmp";
if [[ ! -d "$TMPDIR" ]]; then mkdir -p "$TMPDIR"; fi

REPO_FILE_NAME=`echo "$TARBALL" | awk -F'/' '{print $(NF-2)}'`; # get repo folder name
echo "Repo folder name, extracted from tarball path is '$REPO_FILE_NAME'";

SRC_TAR="$TMPDIR/workspace/confirm-temp";
LOCK_FILE_PATH="$TMPDIR/workspace/src/$REPO_FILE_NAME.lock";

#################################
########## TRAP HANDLING ########
function clean_up {
    echo "Exception caught. Cleaning up...";
    if [[ -d "$SRC_TAR" ]]; then rm -rf "$SRC_TAR/*"; fi
    if [[ -f "$LOCK_FILE_PATH" ]]; then rm "$LOCK_FILE_PATH"; fi
    trap - EXIT;
    exit 1;
}
trap clean_up SIGHUP SIGINT SIGTERM EXIT;
#################################
#################################

# Check if repository is in use by another script.
if [[ -f "$LOCK_FILE_PATH" ]]; then
    echo "Repository is in use by another script, please try again later.";
    exit 1;
fi

# Check directory structure.
if ! [[ -d "$TMPDIR/workspace" ]]; then mkdir "$TMPDIR/workspace"; fi

if ! [[ -d "$SRC_TAR" ]]; then mkdir "$SRC_TAR"; else rm -rf "$SRC_TAR/*"; fi

# Extract tarball into $SRC.
cd "$SRC_TAR" && time tar -xzf "$TARBALL";

# Get revision (commit hash) from a static file revision.html.
REVISION=`cat "$SRC_TAR/htdocs/revision.html"`;
# Get only commit hash, cut off date-time part.
COMMIT_HASH=`echo "$REVISION" | awk -F'-' '{print $1}'`;

# Get tarball file name
TAR_NAME=`echo $(basename "$TARBALL")`;
TYPE="other";
# Check if tarball name contains HF.
# If it does, we have a hotfix, set TYPE to 'hf'.
set +e;
echo $TAR_NAME | grep "HF" 1>/dev/null;
if [[ "$?" = "0" ]]; then
    TYPE="hf";
fi
set -e;

# Get first tag (vX.Y.Z) reachable from a given commit.
# Increase Y for 'other' (rc) confirm, increase Z for 'hf' confirm.
# X should always be increased manually.
```

```
NEW_TAG='v0.1';
## Variant one
SOURCEPATH="$TMPDIR/workspace/src/$REPO_FILE_NAME";
cd "$SOURCEPATH";
# For safety reasons, first delete all local tags so you're sure you get only those on remote
git tag -d $(git tag);
git fetch --all --prune --tags; # find/hide also most recently added/deleted tags
set +e; # git describe exits in case no tag is found; returned error handling at the end of this part
git describe --tag --match 'v*.*.*' "$COMMIT_HASH" 2> /dev/null; # added quiet mode in case of fatal error
if [[ "$?" = "0" ]]; then
    RECENT_TAG=`git describe --tag --match 'v*.*.*' "$COMMIT_HASH"`;
    echo "First tag, reachable from a given commit $COMMIT_HASH, equals $RECENT_TAG. Creating new tag...";

    VERSION_TAG_PART=`echo $RECENT_TAG | awk -F'v' '{print $2}'`;
    VERSION=`awk -F- '{print $1}' <<< $VERSION_TAG_PART`;
    COMMITS_SINCE=`awk -F- '{print $2}' <<< $VERSION_TAG_PART`;

    MAJOR=`awk -F'.' '{print $1}' <<< $VERSION`;
    MINOR=`awk -F'.' '{print $2}' <<< $VERSION`;
    PATCH=`awk -F'.' '{print $3}' <<< $VERSION`;
    NEW_TAG=$VERSION;

    if [[ "$COMMITS_SINCE" = "" ]]; then
        echo "Commit is already tagged. Using existing version ${MAJOR}.${MINOR}.${PATCH}"
    else
        # If hotfix branch exists, don't allow tagging. Allow it only if we're confirming HF branch.
        if [[ "$TYPE" != "hf" ]]; then
            POSS_HF_BRANCH=v$MAJOR.$MINOR.$PATCH-HF;
            git ls-remote --heads origin | grep $POSS_HF_BRANCH 1>/dev/null;
            if [[ "$?" = "0" ]]; then # Check only remote branches
                echo "Hotfix branch still exists. Tagging not allowed.";
                exit 1;
            fi
        fi

        # Tag last commit if everything ok.
        if [[ "$TYPE" = "hf" ]]; then
            # confirming 'hotfix' > increasing patch version
             NEW_PATCH=$(($PATCH+1));
             NEW_TAG=v$MAJOR.$MINOR.$NEW_PATCH;
        elif [[ "$TYPE" = "other" ]]; then
            # confirming 'other' > increasing minor version, patch set to 0
            NEW_MINOR=$(($MINOR+1));
            NEW_TAG=v$MAJOR.$NEW_MINOR.0;
        fi

        git tag -a $NEW_TAG -m "version $NEW_TAG" $COMMIT_HASH 2> /dev/null;
        if [[ "$?" = "0" ]]; then # tagging succeeded, continue
            git push origin $NEW_TAG;
            if [[ "$?" = "0" ]]; then
                echo "Tagged commit $COMMIT_HASH as $NEW_TAG.";
            else
                echo "Something went wrong when pushing tag $NEW_TAG"
                exit 1;
            fi
        else
            echo "Tagging not allowed because it already exists or because of some other problem. Please try again.";
            exit 1;
        fi
    fi

    # Move confirmed tarball file to confirmed folder
    TARBALL_NEW_FILENAME="`echo $(basename "$TARBALL")`-$NEW_TAG";
    TARBALL_DIR=`echo $(dirname "$TARBALL")`;
    TARGET_FILE="$TARBALL_DIR/../confirmed/$TARBALL_NEW_FILENAME"
    if ! [[ -f "$TARGET_FILE" ]]; then
        echo "Moving confirmed tarball file to confirmed folder with new name $TARGET_FILE";
        mv "$TARBALL" "$TARGET_FILE";
    else
        echo "Confirmet tarball file already exist at $TARGET_FILE";
    fi
else
    echo "No recent tags found. At least initial tag must exist. Create one, before integrating/confirming.";
    exit 1;
fi
set -e;

trap - EXIT;
```

Slika A.5: Skripta confirm.sh za potrditev kandidata za izdajo.

```
#!/bin/bash
set -ex;

#########################################################
################## SCRIPT PARAMETERS ####################
#########################################################
OPTS=`getopt -o vrw: --long VERSION,REPO_NAME,WORKSPACE: -n 'parse-options' -- "$@"`
if [ $? != 0 ] ; then echo "Failed parsing options." >&2 ; exit 1 ; fi

## Read parameter values.
while true; do
  case "$1" in
      -v|--VERSION)
        case "$2" in
            "") VERSION= ; shift 2 ;;
            *) VERSION=$2 ; shift 2 ;;
        esac ;;
      -r|--REPO_NAME)
        case "$2" in
            "") REPO_NAME= ; shift 2 ;;
            *) REPO_NAME=$2 ; shift 2 ;;
        esac ;;
    -w|--WORKSPACE)
        case "$2" in
            "") WORKSPACE= ; shift 2 ;;
            *) WORKSPACE=$2 ; shift 2 ;;
        esac ;;
    --) shift ; break ;;
    *) break ;;
  esac
done

## Check mandatory parameters. Exit if not set.
if ! [[ -n "$REPO_NAME" ]]; then echo "'REPO_NAME' param not set!"; exit 1; fi
if ! [[ -n "$VERSION" ]]; then echo "'VERSION' param not set!"; exit 1; fi
if ! [[ -n "$WORKSPACE" ]]; then echo "'WORKSPACE' param not set!"; exit 1; fi
#########################################################
#########################################################
#########################################################

export TMPDIR="$WORKSPACE/tmp/srlive-integrate/tmp";
if [[ ! -d "$TMPDIR" ]]; then mkdir -p "$TMPDIR"; fi

LOCK_FILE_PATH="$TMPDIR/workspace/src/$REPO_NAME.lock";
SRC="$TMPDIR/workspace/src/$REPO_NAME";

###################################
########## TRAP HANDLING ##########
function clean_up {
    echo "Exception caught. Cleaning up...";
    if [[ -f "$LOCK_FILE_PATH" ]]; then rm "$LOCK_FILE_PATH"; fi
    trap - EXIT;
    exit 1;
}
trap clean_up SIGHUP SIGINT SIGTERM EXIT;
###################################
###################################

# Check if repository is being used by another script.
if [[ -f "$LOCK_FILE_PATH" ]]; then
    echo "Repository is in use by another script, please try again later.";
    exit 1;
else

    # Lock repository so other scripts can't change it while current one runs.
    echo "Repository locked." > "$LOCK_FILE_PATH";
fi

source "$SCRIPTDIR/repo-factory.sh" || exit 1;
REPOSITORY=`getRepo $REPO_NAME`;
if ! [[ -d "$SRC" ]]; then git clone "$REPOSITORY" "$SRC"; fi
cd "$SRC";
git fetch --all --prune --tags;

git checkout "${VERSION}"; # checkout specific version
if [[ `echo $?` = "0" ]]; then
```

```
    echo "Checkouting '$VERSION' version...";
    # Create HF branch.
    HF_BRANCH_NAME="${VERSION}-HF";
    git checkout -b "$HF_BRANCH_NAME";
    if [[ `echo $?` = "0" ]]; then
        # New hotfix branch created.
        echo "Created hotfix branch '$HF_BRANCH_NAME'.";
    else
        # If branch already exists, only checkout (normally shouldn't happen).
        git checkout "$HF_BRANCH_NAME";
        if [[ `echo $?` = "0" ]]; then
            echo "Checked out hotfix branch '$HF_BRANCH_NAME'.";
        else
            echo "Problem during creating/checkouting hotfix branch '$HF_BRANCH_NAME'. Aborting...";
            exit 1;
        fi
    fi
    git push origin "$HF_BRANCH_NAME";
else
    echo "Problem during '$VERSION' version checkout. Aborting...";
    exit 1;
fi

# Free repository so other scripts can use it from now on.
if [[ -f "$LOCK_FILE_PATH" ]]; then rm "$LOCK_FILE_PATH"; fi

trap - EXIT;
```

Slika A.6: Skripta start-hf.sh za izvedbo prvega dela pri postopku hitrega popravka.

```
#!/bin/bash
set -ex;

#####################################################
################ SCRIPT PARAMETERS ##################
#####################################################
OPTS=`getopt -o rhdw: --long REPO_NAME,HF_BRANCH,DEST_BRANCH,WORKSPACE: -n 'parse-options' -- "$@"`
if [ $? != 0 ] ; then echo "Failed parsing options." >&2 ; exit 1 ; fi

## Read parameter values.
while true; do
  case "$1" in
      -r|--REPO_NAME)
        case "$2" in
            "") REPO_NAME= ; shift 2 ;;
            *) REPO_NAME=$2 ; shift 2 ;;
        esac ;;
    -h|--HF_BRANCH)
        case "$2" in
            "") HF_BRANCH= ; shift 2 ;;
            *) HF_BRANCH=$2 ; shift 2 ;;
        esac ;;
    -d|--DEST_BRANCH)
        case "$2" in
            "") DEST_BRANCH= ; shift 2 ;;
            *) DEST_BRANCH=$2 ; shift 2 ;;
        esac ;;
    -w|--WORKSPACE)
        case "$2" in
            "") WORKSPACE= ; shift 2 ;;
            *) WORKSPACE=$2 ; shift 2 ;;
        esac ;;
    --) shift ; break ;;
    *) break ;;
  esac
done
## Check mandatory parameters. Exit if not set.
if ! [[ -n "$REPO_NAME" ]]; then echo "'REPO_NAME' param not set!"; exit 1; fi
if ! [[ -n "$HF_BRANCH" ]]; then echo "'HF_BRANCH' param not set!"; exit 1; fi
if ! [[ -n "$WORKSPACE" ]]; then echo "'WORKSPACE' param not set!"; exit 1; fi
if ! [[ -n "$DEST_BRANCH" ]]; then echo "'DEST_BRANCH' param not set!"; exit 1; fi
#####################################################
#####################################################
#####################################################

export TMPDIR="$WORKSPACE/tmp/srlive-integrate/tmp";
if [[ ! -d "$TMPDIR" ]]; then mkdir -p "$TMPDIR"; fi

LOCK_FILE_PATH="$TMPDIR/workspace/src/$REPO_NAME.lock";
SRC="$TMPDIR/workspace/src/$REPO_NAME";

#################################
########## TRAP HANDLING #########
function clean_up {
    echo "Exception caught. Cleaning up...";
    if [[ -d "$SRC" ]]; then rm -rf "$SRC"; fi
    if [[ -f "$LOCK_FILE_PATH" ]]; then rm "$LOCK_FILE_PATH"; fi
    trap - EXIT;
    exit 1;
}
trap clean_up SIGHUP SIGINT SIGTERM EXIT;
#################################
#################################


# Check if repository is being used by another script.
if [[ -f "$LOCK_FILE_PATH" ]]; then
    echo "Repository is in use by another script, please try again later.";
    exit 1;
else
    # Lock repository so other scripts can't change it while current one runs.
    echo "Repository locked." > "$LOCK_FILE_PATH";
fi

source "$SCRIPTDIR/repo-factory.sh" || exit 1;
```

```
REPOSITORY=`getRepo $REPO_NAME`;
if ! [[ -d "$SRC" ]]; then git clone "$REPOSITORY" "$SRC"; fi
cd "$SRC";
git fetch --all --prune --tags;

git checkout $DEST_BRANCH;
git pull;
make all || (make && make all) || exit 1;
set +e;
git merge --ff-only origin/$HF_BRANCH;
if [[ `echo $?` = "0" ]]; then
    echo "Git merge to $DEST_BRANCH branch from branch $HF_BRANCH completed successfully.";
else
    echo "Problem during git merge. Aborting...";
    #git merge --abort;
    git reset --hard HEAD;
    exit 1;
fi
set -e;
git push origin $DEST_BRANCH;

# Remove hotfix branch.
git push origin :$HF_BRANCH; # remove a remote branch

# Free repository so other scripts can use it from now on.
if [[ -f "$LOCK_FILE_PATH" ]]; then rm "$LOCK_FILE_PATH"; fi

trap - EXIT;
```

Slika A.7: Skripta finish-hf.sh za izvedbo zaključitve postopka hitrega popravka.