UNIVERZA V LJUBLJANI

FAKULTETA ZA RAČUNALNIŠTVO IN INFORMATIKO

Rok Razinger

Vzpostavitev okolja za avtomatsko testiranje programske opreme

DIPLOMSKO DELO

VISOKOŠOLSKI STROKOVNI ŠTUDIJSKI PROGRAM PRVE STOPNJE RAČUNALNIŠTVO IN INFORMATIKA

UNIVERZA V LJUBLJANI

FAKULTETA ZA RAČUNALNIŠTVO IN INFORMATIKO

Rok Razinger

Vzpostavitev okolja za avtomatsko testiranje programske opreme

DIPLOMSKO DELO

VISOKOŠOLSKI STROKOVNI ŠTUDIJSKI PROGRAM PRVE STOPNJE RAČUNALNIŠTVO IN INFORMATIKA

MENTOR: viš. pred. dr. Igor Rožanc

Ljubljana, 2015

To delo je ponujeno pod licenco *Creative Commons Priznanje avtorstva - Deljenje pod enakimi pogoji 2.5 Slovenija* (ali novejšo različico). To pomeni, da se tako besedilo, slike, grafi in druge sestavine dela kot tudi rezultati diplomskega dela, lahko prosto distribuirajo, reproducirajo, uporabljajo, priobčujejo javnosti in predelujejo pod pogojem, da se jasno in vidno navede avtorja in naslov tega dela in da se v primeru spremembe, preoblikovanja ali uporabe tega dela v svojem delu, lahko distribuira predelava le pod licenco, ki je enaka tej. Podrobnosti licence so dostopne na spletni strani creativecommons.si ali na Inštitutu za intelektualno lastnino, Streliška 1, 1000 Ljubljana.

Izvorna koda diplomskega dela, njeni rezultati in v ta namen razvita programska oprema je ponujena pod licenco *GNU General Public License*, različica 3 (ali novejša). To pomeni, da se lahko prosto distribuira in/ali predeluje pod njenimi pogoji. Podrobnosti licence so dostopne na spletni strani http://www.gnu.org/licenses.

Fakulteta za računalništvo in informatiko izdaja naslednjo nalogo:

Tematika naloge:

V diplomski nalogi predstavite postopno vzpostavitev okolja, ki omogoča avtomatsko testiranje programskih rešitev za podjetje s specifičnimi zahtevami. Okolje naj vsebuje ustrezno orodje za nadzor programov, sistem nadzora različic, potrebne aplikacijske strežnike za izvajanje izbranih storitev, rešitve za prevajanje oziroma izgradnjo kode ter slednjič proženje avtomatskih testov. Posamezne dele okolja izberite na podlagi ustreznih kriterijev. Nalogo zaključite s prikazom namestitve in analizo učinkovitosti okolja.

IZJAVA O AVTORSTVU DIPLOMSKEGA DELA

Spodaj podpisani Rok Razinger sem avtor diplomskega dela z naslovom:

Vzpostavitev okolja za avtomatsko testiranje programske opreme

S svojim podpisom zagotavljam, da:

- sem diplomsko delo izdelal samostojno pod mentorstvom viš. pred. dr. Igorja Rožanca,
- 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 16. novembra 2015 Podpis avtorja:

Kazalo

Kazalo slik

Seznam uporabljenih kratic

kratica	angleško	slovensko
GUI	graphical user interface	grafični uporabniški vmesnik
DB	database	podatkovna baza
SVN	subversion	sistem za nadzor različic
WAR	web application archive	datoteka, ki vsebuje spletne storitve
CI	continuous integration	stalno povezovanje
BAT	batch file	paketna datoteka

Razlaga uporabljenih tujih izrazov

Izraz	angleško	Pomen v slovenščini
repozitorij	repository	namenski prostor na lokalnem računalniku, strežniku ali oblaku za dinamično izmenjavo zadnjih verzij programske opreme
skripta	script	zaporedje definiranih ukazov za izvajanje
opravilo	task	posamezno opravilo v aplikaciji
vozlišče	node	računalnik, na katerem se izvajajo testne aplikacije (pomen v primeru naše aplikacije)
storitev	service	izvedljive kode, ki se izvajajo na aplikacijskem strežniku
testno okolje	test suite	okolje za zbiranje oziroma grupiranje testnih primerov
vtičnik	plug-in	dodatki za programske rešitve ali brskalnike
razvejanje	branching	metoda za pravilno obvladovanje sistema različic programske opreme

Povzetek

Cilj diplomske naloge je vzpostavitev okolja za avtomatsko testiranje programske opreme. Postopek je razdeljen na več korakov, v katerih je treba izbrati naslednje dele okolja: orodje za nadzor programov in drugih aplikacij, sistem nadzora različic, aplikacijske strežnike za izvajanje storitev, rešitve za prevajanje oziroma izgradnjo javanske kode ter avtomatske teste, ki zagotavljajo pravilno delovanje. Pri vsakem koraku je treba določiti nekaj primernih potencialnih programskih rešitev in jih glede na njihove prednosti in slabosti med seboj primerjati po smiselnih kriterijih. Izbrane rešitve so celovito okolje, ki omogoča avtomatsko testiranje programske opreme.

Za nadzor programov in aplikacij je zaradi svoje enostavnosti izbran Jenkins. Na področju sistema nadzora različic smo zaradi naprednih funkcij izbrali GIT. Med aplikacijskimi strežniki je s svojo zmogljivostjo in dostopnostjo izstopal Wildfly. Najcelovitejša rešitev za prevajanje in izgradnjo javanske kode je bila Gradle. Za avtomatsko testiranje spletnih aplikacij je kot najzanesljivejši izbran Selenium, za funkcijsko testiranje pa, zaradi točnosti rezultatov, SOAP UI. Tako sestavljeno okolje je opisano na praktičnem primeru tako za osnovno, kot tudi za naprednejšo uporabo. Okolje je bilo preverjeno v praksi, kjer se je dobro izkazalo, vendar bi ga bilo mogoče izboljšati z uporabo zanesljivejšega, a plačljivega aplikacijskega strežnika ter boljšega avtomatskega testiranja.

Ključne besede: avtomatizacija testiranja, Jenkins, GIT, Selenium, Wildfly, Gradle, SOAP UI

Abstract

The aim of this thesis was the establishment of an environment for automatic software testing. The process was split into several steps in where the following environment components had to be chosen: a tool for managing the programs and other applications, a concurrent version system, an application server to run web services, an approach for translation or assembly of the Java code and automatic tests which ensure correct operation. With every step a few potential software solutions had to be checked and compared using a set of criteria based on their strengths and weaknesses.

Jenkins was selected for the management of programs and applications for its simplicity. For the concurrent version system GIT was the selected solution due to its advanced functions. Among application servers Wildfly was above the rest because of its abilities and accessibility. For the translation and assembly of the Java code the best solution was Gradle. Selenium, selected for its reliability, was used for automatic testing, and SOAP UI for its accurate results. This environment was described using a practical case for basic and advanced use as well. The environment was tested in practice where it performed admirably. However it could be still improved using a commercial and more reliable application server along with better automatic testing software.

Key words: automation of testing, Jenkins, GIT, Selenium, Wildfly, Gradle, SOAP UI

Poglavje 1 Uvod

Cilj diplomskega dela je prikaz izbire posameznih delov okolja, ki omogoča avtomatsko testiranje programske opreme. Avtomatsko testiranje je v zadnjih letih postalo veliko pomembnejše, saj občutno prihrani čas in hkrati zagotavlja dosledno izvedbo testnih primerov. Na začetku je potrebna količina časa za vzpostavitev takšnega sistema dokaj velika, saj je izbor rešitev odvisen od narave zahtevane programske rešitve. Zaradi hitrih sprememb, popravkov in novih zahtev prihaja do stalnih sprememb in nadgradenj programske opreme [1]. Tu lahko pripomorejo agilne metodologije razvoja, ki v ospredje postavljajo hitre iteracije razvoja programske opreme. Tu je poudarek na delujoči kodi, ki se lahko hitro spreminja ali popravlja, količina dokumentiranja pa je zaradi velike količine del navadno zgolj minimalna [2].

V želji po hitrem razvoju programskih rešitev je potrebno posvetiti vedno več časa področju testiranja programskih rešitev, saj le-te ob vsakem popravku zagotavljajo, da obstoječe, na novo dodane in popravljene funkcionalnosti delujejo, kot je zahtevano in zaželeno. Danes je problem, pa tudi izziv slediti stalni potrebi po spremembah in napredku v programski opremi, ki zahteva vedno nove funkcionalnosti, želji uporabnikov ter hkrati zadostitvi zahtevam zakonodaje in predpisov.

Bistvena prednost avtomatiziranih testov je v tem, da se samo dodajajo novi ali prilagajajo obstoječi testi, da se po nepotrebnem ne izgublja časa za testiranje zadev ter se izloči človeški faktor, ki je navadno najšibkejši in najbolj problematičen del testiranj.

Programske rešitve postajajo kompleksnejše, zato bo teh testov vedno več, njihovo izvajanje bo zahtevalo več časa in posledično višje stroške. Zato je potreba po avtomatiziranih testih vedno večja.

V podjetju, ki bi želelo uvesti takšno avtomatizirano delovanje in testiranje, bi bil ključni faktor porabljen čas oziroma prihranek le-tega. V diplomski nalogi želimo prilagoditi rešitev za slovensko podjetje, ki že izdeluje ravno takšne sisteme. Uspešnost rešitve bomo pokazali z časovno primerjavo delovanja brez in z novim okoljem.

Z okoljem je bilo potrebno pokriti nekaj osnovnih procesov:

- Podpora celotni izvedbi postopka od pridobitve kode iz repozitorija do obveščanja o uspešnosti delovanja avtomatiziranih testov,
- popolnoma avtomatizirano delovanje testov,
- zagotavljanje zanesljivosti rezultatov,
- enostavna uporaba programske opreme za enostavno spreminjanje testnih scenarijev.

V nadaljevanju diplomskega dela bomo predstavili programske rešitve, ki so na voljo za vsak korak testiranja. Preverili bomo tako odprtokodne rešitve, kot tudi plačljivo programsko opremo.

Kot primer bomo v diplomski nalogi vzeli aplikacijo, ki temelji na javanskih storitvah (`WAR`). Ko se vse storitve uspešno namestijo, delujejo kot spletna aplikacija, nad katero se bodo izvajali avtomatski testi.

Na osnovi novo dodane funkcionalnosti programske rešitve se z v nadaljevanju opisanimi orodji najprej pripravijo skripte za izvajanje potrebnih procedur. S tem se lahko začnejo izvajati avtomatizirani testi.

Sami testi simulirajo vpis podatkov v predvidena polja tako, kot bi jih vpisovali končni uporabniki programske rešitve. Odvisno od zahtevane vrste podatkov za definirano polje, se vanj vpišejo ustrezni teksti ali vrednosti.

Že pri pripravi skripte je treba imeti dobro koordinacijo s programerjem, ki je napisal novo funkcijo, saj nam poda točne zahteve končnega uporabnika. Tako lahko upoštevamo vse osnovne in morebitne dodatne funkcionalne zahteve, ki jih zahteva končni uporabnik.

Če so za programsko rešitev že pripravljeni avtomatizirani testi, se nov test samo doda k že obstoječim testom, v nasprotnem primeru pa je potrebno narediti postopek, ki je opisan v nadaljevanju.

Ko je okolje postavljeno in ko je definiran posamezni test, se testiranja lahko začnejo.

Kadar programer dobi obvestilo, da programska rešitev ne deluje pravilno, je potreben popravek rešitve. Možno je tudi, da je napačno pripravljena testna skripta in da je napaka v tem delu testiranja (napaka v kodi, manjkajo vtičniki, nepravilne verzije vtičnikov ali podobno). Napako najprej odpravlja program za avtomatizirane teste, če pa je le-ta na programerjevi strani, jo odpravi on.

Ko vse poteka brezhibno sistem ne javi napake, s tem pa dobimo potrditev, da je programska rešitev delujoča.

V poglavju 2 smo preučevali možne rešitve na področju nadzora programov in aplikacij. Iskali smo tako, ki bo najlažje združljiva z ostalimi, enostavna za uporabo in zanesljiva.

Na področju sistema nadzora različic, ki smo ga obravnavali v poglavju 3, smo potrebovali rešitev, ki bi kodo hranila na enem mestu, hkrati pa bi dovoljevala združevanje popravkov.

V poglavju 4 je bilo potrebno izbrati rešitev na področju aplikacijskih strežnikov. Ko smo dali izbrani storitveni paket v pogon, je moral zagotoviti delovanje spletnih storitev.

Po izbiri vseh potrebnih orodij in prožilcev akcij smo v poglavju 5 obravnavali programe za prevajanje in izgradnjo javanske kode. Ob vsaki posodobitvi kode je namreč potrebno tvoriti nove storitvene pakete.

V poglavju 6 smo obravnavali več območij delovanja avtomatskih testov. Na tem področju smo iskali dve rešitvi. Prva je pokrivala področje spletnih aplikacij, druga pa je bila namenjena funkcijskemu testiranju.

Ko smo izbrali vse delne rešitve, jih je bilo treba združiti v celoto in preizkusiti delovanje na praktičnem primeru, kar smo naredili v poglavju 7.

Poglavje 2 Orodje za nadzor programov in drugih aplikacij

Za nadzorovanje aplikacij, ki jih bomo v naslednjih poglavjih izbrali za uresničitev ciljev, je potrebno imeti neko univerzalno orodje, ki lahko vse te aplikacije nadzoruje oziroma pridobi uspešne rezultate.

Bistveno je, da lahko takšna aplikacija deluje kot neke vrste nadzornik nad pod-operacijami in lahko na podlagi pridobljenih stanj sproži ustrezno in vnaprej definirano dejavnost ali akcijo:

- naloga uspešno izvedena: aplikacija se premakne na oddelek »stabilno«,
- naloga prekinjena ali neuspešna: osebi, ki je povzročila izpad oziroma je kriva za napako, se pošlje obvestilo o napaki.

Za zahtevane naloge so primerne naslednje programske rešitve:

- Jenkins CI [3] (zelo široko razširjen, licenciran pod MIT licencami, tedenske nadgradnje in popravki, na trgu od leta 2011, primeren še danes zaradi številnih vtičnikov za nove in stare aplikacije).
  - Prednosti:
    - odprtokodna rešitev,
    - preprosta uporaba,
    - dodajanje novih računalnikov – elementov znotraj istega omrežja je enostavno,
    - na voljo je veliko podpore za različne programe (SVN [4], GIT [5], ANT [6], Groovy [7], Gradle [8], …),
    - nadgradnje in izboljšave 2 – 3 krat mesečno,
    - zelo velika skupnost razvijalcev.
  - Slabosti:
    - vtičnike (angl. plugins) razvijajo prostovoljni razvijalci, zato lahko pride do težav ali popolne odpovedi delovanja, kar povzroči neuspešno izvedenost naloge,
    - pri nadgradnji se določeni vtičniki ne namestijo pravilno oziroma so odstranjeni. Te vtičnike sicer lahko ponovno prenesemo in namestimo, vendar nenehno prenašanje in pregledovanje lahko vzame veliko časa, ki bi ga lahko namenili izboljšavam,
    - občasno ga je potrebno ponovno zagnati, saj porabi velike količine pomnilnika tudi v mirovanju.

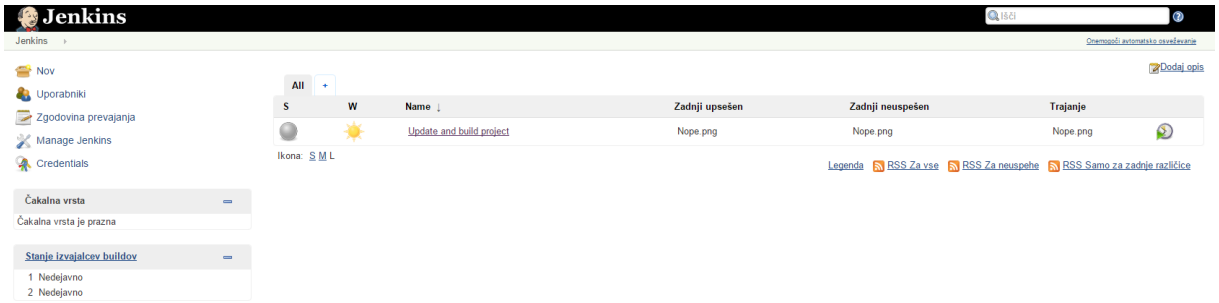
- GitLab CI [9] (šele prihaja v uporabo, v lasti podjetja Gitlab, stalne izboljšave, izdan leta 2012).
  - Prednosti:
    - enojni vpis – enotna vstopna točka tako za GitLab [10] kot tudi za GitLab CI [9]. Zaradi enega vpisa uporabniškega imena se lažje določi, kdo je sprožil neko akcijo,
    - prevajanje se sproži takoj, ko nekdo spremeni ali doda kodo.
  - Slabosti:
    - relativno omejen nabor vtičnikov,
    - praktičen samo, če se poleg tega uporablja tudi GitLab-ov sistem za nadzor različic,
    - dokaj majhna skupina razvijalcev, kar posledično pomeni manj podpore.

Poleg navedenih obstaja še vrsta drugih rešitev (Apache Continuum [11], CruiseControl [12] itd.) vendar so zastarele ali pa podpirajo premajhno število operacij, zato so deloma ali povsem neuporabni.

Kriterij za izbiro najprimernejšega kandidata:
Pri tem koraku je bilo najpomembnejše enostavnost uporabe, združljivost z drugo programsko opremo, nizka cena, možnost uporabe brez licence in podpora uporabniku.

Glede na navedene lastnosti sem se odločil za uporabo rešitve Jenkins CI, saj je dobra tako za začetnike kot tudi za napredne uporabnike in najbolj ustreza zahtevam predstavitvene naloge.

2.1 Osnovno delovanje



Slika 1: Nadzorna plošča

Naloge v Jenkinsu so poimenovane kot opravila (angl. tasks) (slika 1).

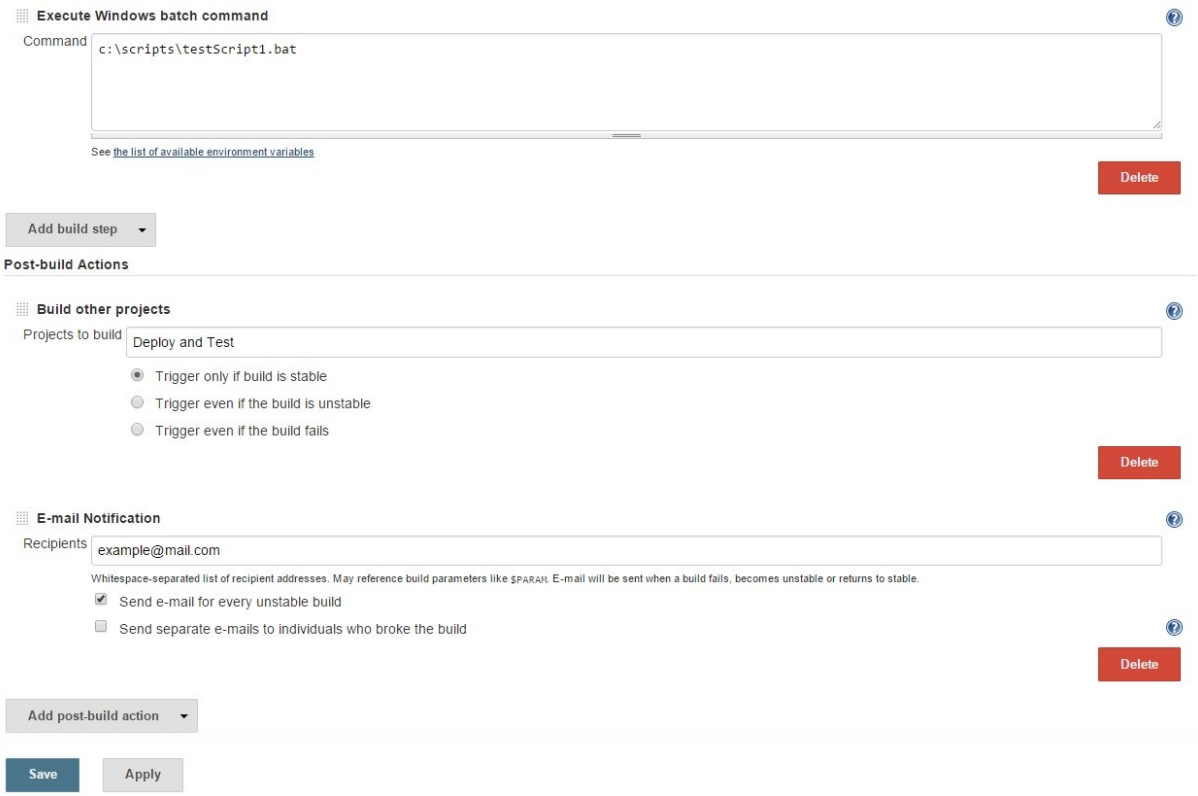
Vsako opravilo lahko opravi eno ali več točno definiranih nalog (posodobi sistem za nadzor različic, izvedi skripto, pošlji elektronsko pošto).

Osnovna namestitev omogoča dokaj omejen nabor ukazov kot je izvajaje ukazov v ukazni vrstici Windows ali Linux operacijskih sistemov, posodobitev SVN repozitorija in izvajanje ANT skript.

Po tem orodju je veliko povpraševanja, zato se je razvila skupnost, ki se trudi integrirati uporabo vseh pomembnejših in razširjenih programov (GIT, Gradle, Groovy, …) v samo rešitev. Takšnim razširitvam pravimo vtičniki in so ključni za celovito pokritost naše procedure avtomatizacije. Jenkins ponuja več kot 1000 uradnih in preverjenih vtičnikov.

Ko prenesemo in namestimo vse želene vtičnike, postane Jenkins zelo vsestransko orodje, ki podpira vse operacije, ki jih potrebujemo za uspešno izvedbo naloge.

Na prikazanem primeru želimo simulirati izvedbo Windows `bat` skripte, proženje drugega opravila in poročanje o uspešnosti izvedenega opravila (slika 2). Ob uspešni izvedbi se sproži opravilo `Deploy and Test`.



Slika 2: Nastavitve na Jenkins opravilu

Ker navedene skripte ni mogoče najti, nam Jenkins pošlje sporočilo, da je pri tem opravilu prišlo do napake.

2.2 Napredne nastavitve opravil

Pogostost izvedbe je odvisna od načina dela podjetja, zato je na voljo več možnosti nastavljanja intervalov testiranja. Časi opravljanja posameznih opravil se lahko nastavijo na več načinov:

- točen čas izvedbe (ob 6:00),
- periodično izvajanje (vsakih 15 minut),
- pregledovanje stanja datotek (na določen časovni interval se preveri, če je bila koda v repozitoriju spremenjena; v primeru da je prišlo do spremembe, se proži izvedba opravila).

Ko opravilo zaključi s svojo predvideno dejavnostjo, se obarva glede na uspešnost izvedbe. Možni rezultati:

- prozorno: opravilo se še nikoli ni izvedlo,
- modro: opravilo je stabilno oz. se je uspešno izvedlo,
- rumeno: opravilo je nestabilno, vendar se izvede,
- rdeče: prišlo je do kritične napake.

Poleg opisanih naprednih nastavitev je na voljo še veliko drugih, ki pa za naš primer niso bile relevantne.

Poglavje 3 Sistem nadzora različic

Običajno več razvijalcev želi istočasno spreminjati isti del kode, zato lahko nastanejo težave z njenim osveževanjem. Tipičen pojav je, ko prvi razvijalec odstrani popravke, ki jih je pred nekaj minutami naredil drugi razvijalec. Izziv nastane tudi v primeru, ko nek razvijalec popravi del kode, ne da bi naredil varnostno kopijo obstoječe, delujoče kode. V ta namen so bili razviti sistemi za nadzor različic [13].

Na voljo je več različnih načinov sistemov izvedb in sicer:

- spletno gostovanje sistema nadzora različic: podjetje ali oseba zakupi določen prostor za hrambo podatkov in tam hrani datoteke,
- lokalna namestitev sistema nadzora različic: za zbirališče se določi lokalni računalnik v omrežju, na njem pa se namesti sistem.

Na tem področju sta v ospredju dva sistema in sicer:

- SVN (SubVersion) [4] (zelo razširjen in priljubljen v malih in srednjih računalniških podjetjih, v lasti fundacije Apache, od leta 2000 dalje, zmeraj več uporabnikov prehaja na GIT [5]).
  - Prednosti:
    - uporaba je zelo preprosta (skupaj s programom TorouiseSVN [14]); zahtevnejše koncepte delovanja se lahko naučimo sproti, niso pa nujni za osnovno uporabo,
    - pokrije vse osnovne zahteve in težave, ki smo jih predhodno opisali.
  - Slabosti:
    - če ni dostopa do interneta, ni mogoče »objaviti« spremenjene kode in posledično ni vidna ostalim programerjem. Če dva razvijalca občutno spremenita isto datoteko in jih SVN ne uspe združiti, lahko pride do konflikta,
    - vse je locirano na eni fizični lokaciji. Če želimo dostop izven omrežja, v katerem je strežnik, moramo ustrezno nastaviti omrežne naprave. S tem lahko naredimo varnostno luknjo v omrežju,
    - ni nadaljnjega razvoja.

- GIT (ponudnik GitLab) [5] (vedno bolj priljubljena rešitev, v lasti podjetja Gitlab, vpeljan leta 2011, najprimernejša za današnje projekte in način razvoja) (razširjenost, lastnik, vzdrževanje, čas obstoja, primeren še danes).
    - Prednosti:
        - omogoča delo v nepovezanem načinu,
        - deluje kot neke vrste nadgradnja SVN (možno ga je uporabljati tako s povezavo kot tudi brez nje),
        - datoteke hranijo na spletu,
        - stalno se nadgrajuje in izboljšuje.
    - Slabosti:
        - na začetku se je težje privaditi na decentraliziran sistem, saj so za isti osnovni ukaz potrebni različni ukazi glede na verzijo sistema (`commit` – lokalno zapiše datoteke, `push` – zapiše datoteke na glavni strežnik),
        - potrebno je poznati koncepte uporabe, kot je razvejanje (angl. branching).

Kriterij za izbiro najprimernejšega kandidata:
Pri tem koraku je bilo najpomembnejše enostavnost uporabe, možnost uporabe brez licence, podpora uporabniku in uveljavljen način dela (postopek združevanja kode).

V tem primeru je najboljši kandidat Git, saj ponuja največ možnosti razvoja. Sicer je potrebno nekaj dodatnega izobraževanja, vendar se kasneje izkaže kot bolj vsestranska rešitev.

Na spletu je kar nekaj brezplačnih in plačljivih ponudnikov gostovanja GIT. Problem nastane, ko želimo omejiti število gledalcev datoteke. Veliko ponudnikov, ki brezplačno ponuja gostovanje, to stori samo pod pogojem, da je izvorna koda vidna vsem. Če želite omejiti krog ljudi, ki lahko urejajo datoteke, je za tak servis potrebno mesečno plačilo. Sistem deluje tako, da si razvijalec prenese datoteke iz centralne veje in si ustvari svojo vejo. Ko konča z razvojem in popravki, si mora najprej posodobiti svojo vejo s centralno, šele potem lahko `shrani/objavi` svoje spremembe v glavni veji.

Poglavje 4 Aplikacijski strežniki za izvajanje storitev

Naš primer uporabe predvideva izvajanje storitev na strežniku, zato za realizacijo naloge potrebujemo dobro in preverjeno rešitev. Na spletu je velik nabor ponudnikov aplikacijskih strežnikov, vendar so nekateri bolj razširjeni in uspešnejši kot drugi. Nekaj teh sem na podlagi opisov in ocen izbral in pregledal.

- JBoss [15] (drugi najbolj uporabljen aplikacijski strežnik (2015), v lasti RedHat, podpora ukinjena z migracijo na Wildfly, začetki segajo v leto 1999, preimenovan v letu 2013).
  - Prednosti:
    - odprtokodna rešitev,
    - preprosta uporaba brez registracij.
  - Slabosti:
    - ni zagotovil o delovanju (ne prevzemajo odgovornosti za nedelovanje),
    - ukinjena podpora leta 2012.
- Wildfly [16] (naslednik JBoss-a) (v tej obliki od leta 2013, mesečna izdaja novejših različic).
  - Prednosti:
    - odprtokodna rešitev,
    - preprosta uporaba brez registracij,
    - stalne izboljšave in odpravljanje napak.
  - Slabosti:
    - ob daljši uporabi lahko nekontrolirano zasede vedno več glavnega pomnilnika in program je potrebno ponovno zagnati,
    - ni zagotovil o delovanju (ne prevzemajo odgovornosti za nedelovanje),
    - ob nepravilni zaustavitvi se odstranijo vse obstoječe storitve (`.war` datoteke). Do teh težav ne pride, če zagotovimo aktivno varnostno kopijo.
- WebLogic [17] (tretji najbolj uporabljen aplikacijski strežnik (2015), v lasti organizacije Oracle, nadgradnje manj pogoste, na trgu od leta 2008, zaradi zanesljivosti je uporaben še danes).
  - Prednosti:
    - v ceno je vključena tehnična podpora,
    - zagotovila o delovanju na nivoju resnejših aplikacij.
  - Slabosti:
    - licenčni produkt,
    - zapleten uporabniški vmesnik.

- WebSphere [18] (manjšinski delež uporabnikov, izdalo ga je podjetje IBM, redke izboljšave, prvič uporabljen leta 2000, danes ni več aktualen).
    - Prednosti:
        - v ceno je vključena tehnična podpora,
        - zagotovila o delovanju na nivoju resnejših aplikacij.
    - Slabosti:
        - licenčni produkt,
        - zapleten uporabniški vmesnik.

Kriterij za izbiro najprimernejšega kandidata:
Pri tem koraku je bilo najpomembnejše enostavnost uporabe, možnost uporabe brez licence, podpora uporabniku preko forumov in zanesljivost delovanja pri dolgotrajni uporabi.

Zaradi nezadostne dodane vrednosti nakupa plačljive programske opreme je najboljši kandidat Wildfly [16], ki ga razvija skupnost Red Hat [20]. Poleg tega je tu govora o strežniku za izvajanje testne platforme in ne produkcijske različice.
Namestitev je zelo preprosta. Ko si prenesemo najnovejšo različico programa iz uradne spletne strani, jo preprosto razširimo na želeno mesto in zaženemo.

Strežnik lahko zaženemo na več načinov:

1. način: zaženemo skripto `standalone.bat`, ki zažene strežnik v načinu ukaznega poziva (angl. Command prompt),
2. način: če predpostavimo, da bo strežnik deloval na Windows platformi, ga je možno nastaviti kot Windows storitev (angl. Windows service).

Ko imamo delujoč aplikacijski strežnik, lahko naše produkte zaženemo preko Wildfly uporabniškega vmesnika ali pa preko fizičnega premika `.war` datotek v mapo `deployments`. Wildfly samodejno zazna, da so bile v mapo postavljene datoteke, ki jih je mogoče zagnati in jih samodejno obdela. Ker pa lahko med delovanjem pride do nevšečnosti, se dogajanje beleži v `log` datoteko. Seveda pa se ne moremo izogniti nekaterim težavam, ki se pojavljajo. Če pride do nenadne spremembe oziroma se strežnik na nepravilen način ugasne, lahko izgubimo vse datoteke in nastavitve, ki smo jih imeli. Ena izmed najbolj zaskrbljujočih težav pa se pojavi, ko po daljšem časovnem intervalu strežnik zahteva vse večje količine glavnega pomnilnika za delovanje tako v mirovanju, kot tudi ob polni obremenitvi, ne glede na to, da je za potrebe testiranja to več kot dovolj uporabno orodje.

Poglavje 5 Prevajanje in izgradnja javanske kode

Ko imamo pripravljena vsa potrebna orodja in prožilce akcij, se lahko posvetimo programom za prevajanje kode, saj moramo ob vsaki posodobitvi tvoriti nove storitvene pakete (`.war` datoteke). Na tem področju izstopajo tri programske rešitve. Vse so odprtokodne in so med seboj dokaj različne:

- ANT [6] (zelo razširjen, spada pod findacijo Apache, redna izdaja posodobitev, v uporabi od leta 2000, zaradi stalne uvedbe novih knjižnic je še danes aktualen).
  - Prednosti:
    - dobra ureditev datotek (XML format),
    - dobra podpora za druga orodja (MySQL, Java, ..,).
  - Slabosti:
    - težje razumevanje in tvorjenje konfiguracijskih datotek,
    - ne vsebuje nekaterih ključnih vtičnikov, ki jih potrebujemo za naš primer.

- Groovy [7] (razširjen in popularen zaradi podrobnosti z javansko kodo, lastnik je fundacija Apache, na voljo od leta 2003, še vedno v uporabi zaradi novih knjižnic).
  - Prednosti:
    - ukazi so podobni javanskih kodi,
    - velika skupnost uporabnikov.
  - Slabosti:
    - zaradi zahtevnosti sistema se pri zahtevnejših opravilih pozna na učinkovitosti ali hitrosti delovanja.

- Gradle [8] (zelo razširjen, lastnik je podjetje Gradle, redne nadgradnje in popravki, začetki v letu 2007, zaradi vsestranskosti vedno bolj priljubljen in bolj pogosto uporabljen).
  - Prednosti:
    - preprosti ukazi,
    - vsebuje vtičnike za ANT in Groovy.
  - Slabosti:
    - ker se še razvija, je lahko na nekaterih področjih primanjkljaj dokumentacije ali podpore.

Pri izbiri najprimernejšega kandidata je bil kriterij zastavljen na naslednji način: enostavnost uporabe, združljivost z orodjem za nadzor programov, možnost uporabe dodatnih knjižnic in podpora uporabniku.

Čeprav je Gradle najmlajša in teoretično najmanj samostojno dodelana rešitev, ta vključuje tudi elemente iz ANT-a in Groovy-ja, zato se odločimo za ta produkt. Predvsem nam nudi veliko možnosti in načinov, ki jih lahko uporabimo, ko želimo rešiti določen problem.

5.1 Uporaba in delovanje

Gradle so ustvarili z namenom, da poenostavijo prevajanje javanske kode. Ker pa so predvideli in spoznali, da imajo drugi to tehnologijo že v uporabi, so se odločili, da najprej omogočijo uporabo njihovih metod in sproti dodajajo svoje. Predvsem so zelo osredotočeni na dokumentiranje tako osnovnih, kot tudi naprednejših operacij, ki so na voljo na njihovi spletni strani. Omogoča tudi uporabo dodatnih zunanjih knjižnic (tako lokalnih, kot tudi spletnih) in odvisnosti (odvisnost ene procedure od druge). Če te odvisnosti niso izpolnjene, se skripta ne more ustrezno prevesti in izvesti. Ima tudi možnost deklaracije spremenljivk v datotekah oznake `.properties` (slika 3).

```
# Login information
username=myusername
password=mypassword
# File location
RequeredFileLocation="c:/directory/target"

# Login requirement
isLoginRequired=true

# Database connection
databaseConnection=jdbc:mysql://server_IP:3306/thesisDatabase
```

Slika 3: Primer datoteke gradle.properties

Tako kot pri vsakem jeziku je najbolje začeti pri osnovah, kot so izpisi teksta na zaslonu (slika 4).

```
task printText{
    println 'Hello world'
}
```

Slika 4: Primer izpisa "Hello world"

Če želite opravila narediti bolj pregledna jih lahko razdelite na manjša opravila, ki so med seboj odvisna (slika 5).

```
task subtask{
    doLast{
        copy{
            from 'c:/thesis/files'
            into 'c:/target/'
        }
        println 'Copying done'
    }
}

task mainTask(dependsOn: subtask){
    doLast{
        commandLine 'cmd', '/c', "${rootDir}/runProgram.cmd";
        println 'Script started'
    }
}
```

Slika 5: Primer odvisnosti opravil

Ena glavnih funkcij je posodobitev SVN repozitorija, v katerem se nahaja koda. Za ta namen je bila narejena knjižnica SVNKit[19], ki vsebuje funkcije za klic vseh pomembnih metod (slika 6).

```
def svnUpdate(filePath) {
    def myFile = file(filePath);
    if (myFile.exists()) {
        def didIt = false;
        while (true != didIt) {
            try {
                ISVNOptions options = SVNWCUtil.createDefaultOptions(true);
                SVNClientManager clientManager = SVNClientManager.newInstance(options,username,password);
                SVNUpdateClient updateClient = clientManager.getUpdateClient();
                updateClient.doUpdate(myFile, SVNRevision.HEAD, true);
                didIt = true;
                sleep (15*1000)
            } catch (Exception e) {
                println "SVN has encountered an unexpected error: ${e}";
                sleep (15*1000)
            }
        }
    }
}
```

Slika 6: Primer razreda, ki uporablja zunanje knjižnice

Znotraj Gradle-a kličemo procedure programskih rešitev ANT ali Groovy. To je še posebej pomembno zato, ker na spletu že obstajajo obstoječi razredi, ki so uporabni za naše izzive. Na naslednji sliki je prikazana uporaba SQL skripte na podatkovni bazi (slika 7).

```
configurations {
    sql
}
dependencies {
    sql("mysql:mysql-connector-java:5.1.13")
}

def runDBScript(scriptName, url, user, password) {

ant.sql(classpath: configurations.sql.asPath,
        src:scriptName,
        driver:'com.mysql.jdbc.Driver',
        print:false,
        url:url,
        userid:user,
        password:password,
        onerror:'continue'
    )
}
```

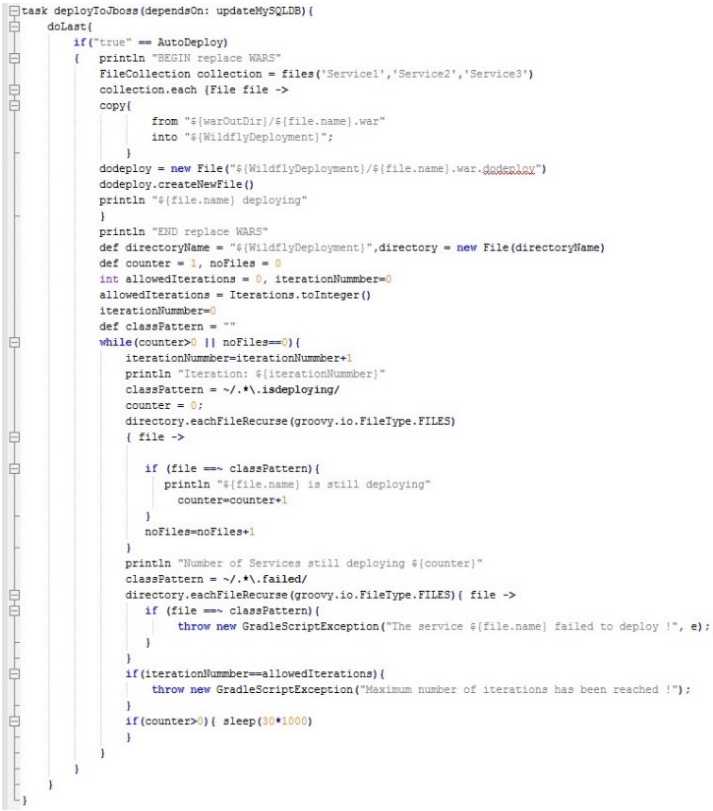
Slika 7: Primer ANT skripte v rešitvi Gradle

Ker je potrebno odstraniti tudi nekaj obstoječih datotek z določenim tipom, uporabimo Groovy skripto (slika 8).

```
task deleteCurrentFiles()  <<{
    println "BEGIN delete"
        def directoryName = "c:/thesis"
        def directory = new File(directoryName)
        def classPattern = ~/.*\.deployed/
        try
        {
            directory.eachFileRecurse(groovy.io.FileType.FILES){ file ->
              if (file ==~ classPattern)
              {
                println "Deleting ${file}..."
                file.delete()
              }
            }
        } catch (Exception e) {
            println "Error: Could not find any .deployed files ${e}"
        }
    println "END delete"
}
```

Slika 8: Razred v Groovy, ki odstrani datoteke tipa `.deployed`

Ko predhodne datoteke odstranimo, jih želimo nadomestiti z novimi, pravkar ustvarjenimi storitvami (slika 9).



```
task deployToJboss(dependsOn: updateMySQLDB){
    doLast{
        if("true" == AutoDeploy)
        {   println "BEGIN replace WARS"
            FileCollection collection = files('Service1','Service2','Service3')
            collection.each {File file ->
            copy{
                    from "${warOutDir}/${file.name}.war"
                    into "${WildflyDeployment}";
                }
            dodeploy = new File("${WildflyDeployment}/${file.name}.war.dodeploy")
            dodeploy.createNewFile()
            println "${file.name} deploying"
            }
            println "END replace WARS"
            def directoryName = "${WildflyDeployment}",directory = new File(directoryName)
            def counter = 1, noFiles = 0
            int allowedIterations = 0, iterationNummber=0
            allowedIterations = Iterations.toInteger()
            iterationNummber=0
            def classPattern = ""
            while(counter>0 || noFiles==0){
                iterationNummber=iterationNummber+1
                println "Iteration: ${iterationNummber}"
                classPattern = ~/.*\.isdeploying/
                counter = 0;
                directory.eachFileRecurse(groovy.io.FileType.FILES)
                { file ->

                  if (file =~ classPattern){
                    println "${file.name} is still deploying"
                      counter=counter+1
                  }
                  noFiles=noFiles+1
                }
                println "Number of Services still deploying ${counter}"
                classPattern = ~/.*\.failed/
                directory.eachFileRecurse(groovy.io.FileType.FILES){ file ->
                  if (file =~ classPattern){
                      throw new GradleScriptException("The service ${file.name} failed to deploy !", e);
                  }
                }
                if(iterationNummber==allowedIterations){
                    throw new GradleScriptException("Maximum number of iterations has been reached !");
                }
                if(counter>0){ sleep(30*1000)
                }
            }
        }
    }
}
```

Slika 9: Primer razreda v Gradle, ki kopira in aktivira storitve

Kljub temu, da se še uveljavlja, bo v prihodnosti verjetno vse bolj v uporabi, saj je res vsestranska in omogoča nešteto možnosti.

Poglavje 6 Avtomatski testi, ki zagotavljajo pravilno delovanje

Na spletu najdemo veliko različnih načinov uporabe in izdelave aplikacij oz. programov, zato je temu primerno tudi število načinov preverjanja delovanja. Nekaj najbolj pogostih je opisanih v nadaljevanju:

Testiranje GUI

Pri aplikacijah, ki so fizično nameščene na uporabniških računalnikih, je potrebno preveriti delovanje vmesnika s simulacijo klikov uporabnika. Primerna orodja za to so:

- Smartbear TestComplete[21] (zaradi svoje vsestranskosti zelo razširjen, v lasti podjetja SmartBear, stalna podpora in popravki, na voljo od leta 1999, danes je še vedno eden vodilnih ponudnikov.
  - Prednosti:
    - Primeren tako za začetnike kot tudi za napredne uporabnike,
    - poleg fizičnega klikanja ima tudi možnost primerjave slik in stanj v podatkovni bazi,
    - dobra podpora in primeri uporabe,
    - poleg plačljive podpore je velika skupnost tudi na spletu.
  - Slabosti:
    - Drage licence za uporabo,
    - v določenih primerih nepravilno zazna gradnike na obrazcih.

Poleg tega je na voljo še vrsto drugih rešitev:

- TestPartner [22]
- SilkTest [23]
- AutoIt [24]

Spletna aplikacija

V to kategorijo spadajo aplikacije, do katerih se dostopa preko spletnega brskalnika (Google Chrome, Mozilla Firefox, Opera, Safari, Internet Explorer). Tu uporabnik ne potrebuje namestitve na svojem računalniku, pač pa le dostop do podatkov, ki se nahajajo na nekem strežniku. Za preverjanje delovanja so na voljo naslednja orodja:

- SeleniumHQ[25] (zaradi možnih vtičnikov zelo razširjen, registriran pod licenco Apache 2.0, nadgradnja na voljo na mesečnem ciklu, projekt je bil ustvarjen leta 2004, zaradi velike skupnosti je vedno bolj priljubljen).
  - Prednosti:
    - Odprtokodna rešitev,
    - preprosta metoda »snemanja« testov preko brskalnika Firefox, ki lahko pretvori iz oblike HTML v več možnih različic: XML, JUnit4, …,
    - možnost pisanja testov v programskem jeziku Java,
    - velika podporna skupnost.
  - Slabosti:
    - V primeru slabe odzivnosti na testni platformi lahko povzroči nepravilne rezultate,
    - težava pri izboru opcije za nalaganje datoteke na stran (deluje, če je gradnik tipa input, ne deluje če je gradnik tipa button ali karkoli podobnega).

Poleg tega pa lahko uporabimo tudi naslednja orodja:

- SOAtest [26]
- Test Studio [27]
- TestComplete [21]

Funkcijsko testiranje

Tu v poštev pridejo aplikacije, ki delujejo na način odjemalec-strežnik. Komunikacija poteka z uporabo vnaprej določenih razredov, ki delujejo na način zahteva-odziv (angl. request-response). Med bolj priljubljenimi so:

- SOAPUI [29] (Zelo priljubljena za takšna testiranja, brezplačna različica je odprtokodna, plačljiva pa je v lasti podjetja SmartBear, redne izboljšave in posodobitve, na voljo od leta 2005, danes je še vedno aktualen).
  - Prednosti:
    - za uporabo ni potreben nakup licence,
    - dobro definiranje pričakovanih rezultatov,
    - podpira večino postavljenih zahtev iz definirane naloge.
  - Slabosti:
    - relativno zapleten uporabniški vmesnik,
    - če pričakujemo dinamične rezultate, le-ti niso mogoči,

- v primeru napake se ne zapiše razlog ampak samo to, da ni bilo pričakovanih rezultatov,
- če na istem projektu dela več ljudi, se lahko uniči glavna datoteko za projekt, ki hrani vse testne definicije in postopke.

Poleg predstavljene rešitve so na voljo tudi naslednje:

- Flask [30] in programski jezik Python [31]
- TestRail [32]

Kriterij za izbiro najprimernejšega kandidata:
Pri tem koraku je bilo pomembno izbrati način preverjanja delovanja, ki ga potrebujemo za našo izbrano aplikacijo.

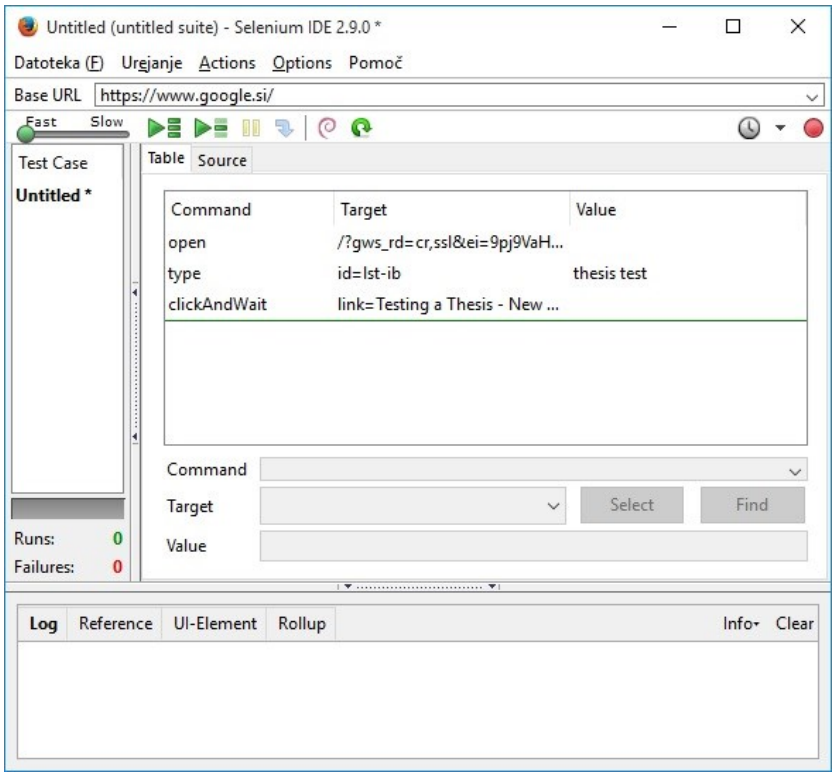
V nadaljevanju sta predstavljeni izbiri za dva primera aplikacij.

6.1 Testiranje spletnih aplikacij

Glede na lastnosti teh rešitev in naravo našega izziva je tu najbolj primeren kandidat Selenium, saj je usmerjen na preverjanje delovanja aplikacij, ki delujejo na brskalnikih. Zaradi različnih načinov delovanja pa je uporaben tudi ko ima programer omejeno znanje na področju pisanja testov. Ena najboljših lastnosti te aplikacije je stalni razvoj in dopolnjevanje funkcij. Nove verzije izhajajo v dvomesečnih intervalih.

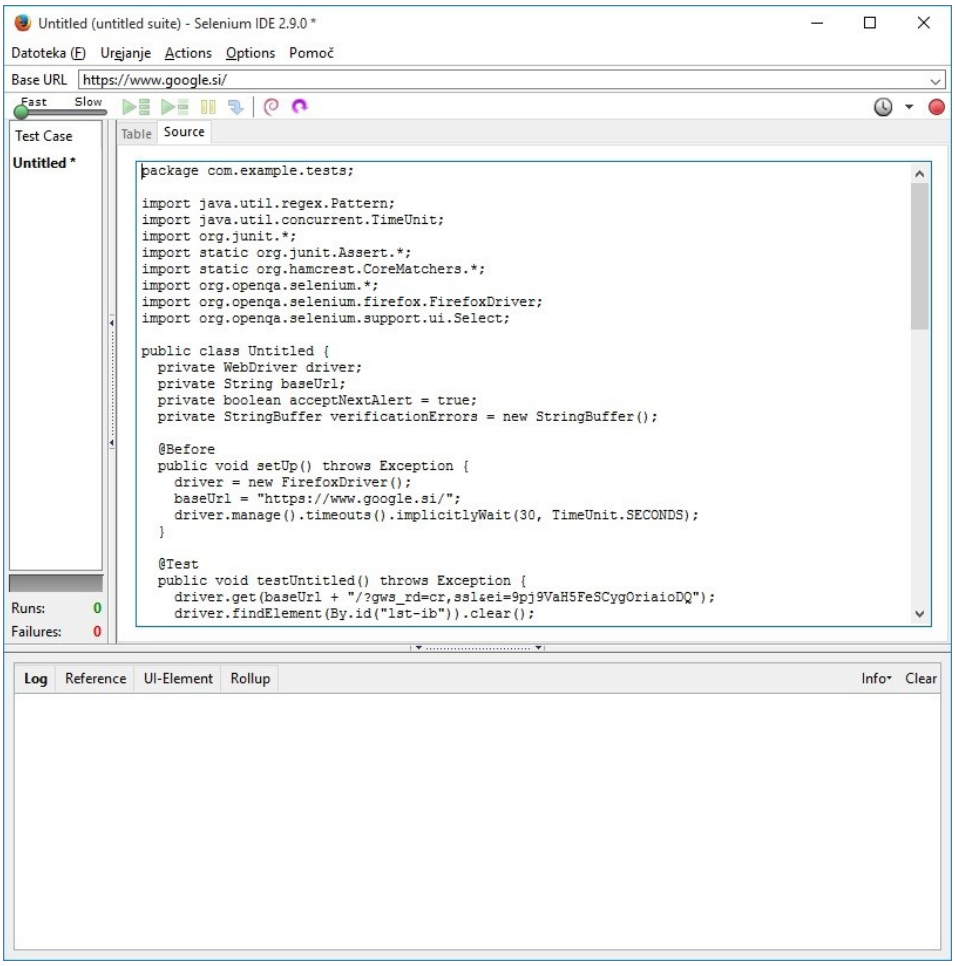
6.1.1 Snemanje in pretvorba testov

Kot že omenjeno je največja prednost v »snemanju« testov preko brskalnika Mozilla Firefox, saj vsebuje uporabniški vmesnik, ki pomaga pri urejanju ukazov (slika 10).



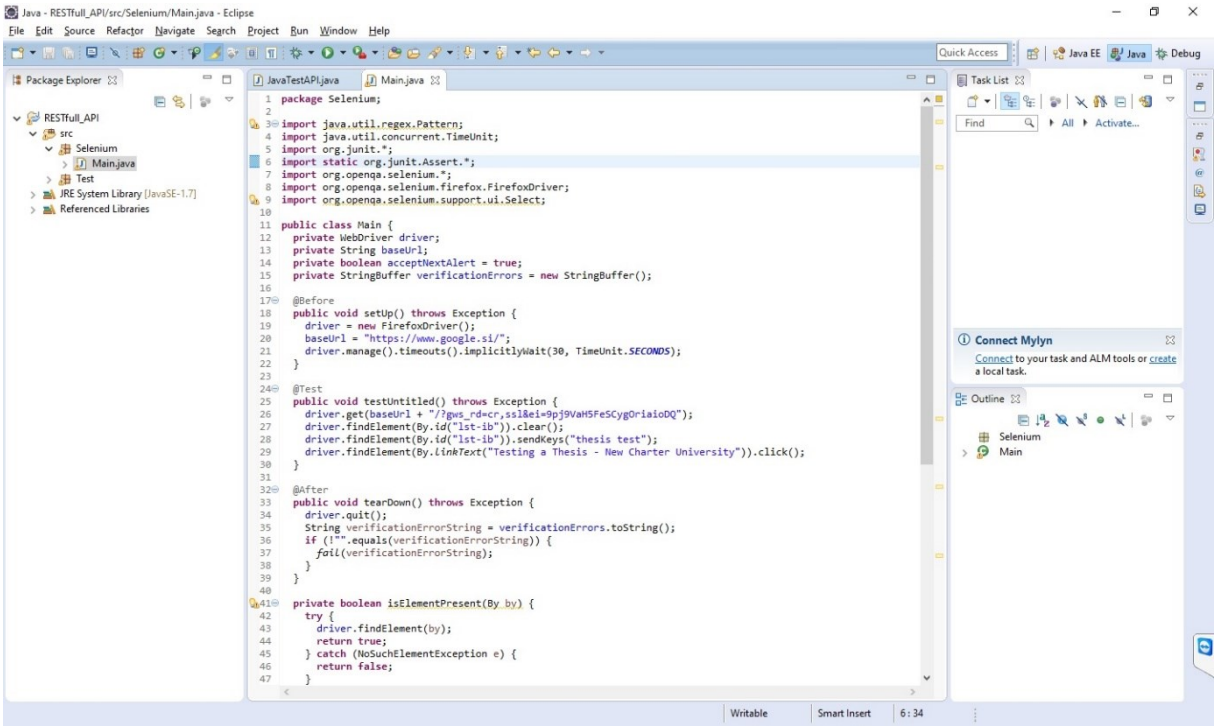
Slika 10: Selenium IDE v brskalniku Firefox

Ko test uspešno posnamemo, ga moramo pretvoriti v nam uporabno obliko. V tem primeru bomo pretvorili v JUnit4 način (slika 11).



Slika 11: Pretvorba iz oblike HTML v JUnit4

Sedaj imamo test pripravljen za izvajanje, vendar nimamo okolja, v katerem bi teste poganjali. Temu primerno v Java urejevalniku (Eclipse [33], NetBeans [34], …) pripravimo projekt in dodamo vtičnike, ki jih Selenium potrebuje za delovanje.



Slika 12: Prenos kode v Eclipse

Ko imamo test pripravljen, ga lahko izvedemo in poizkusimo, če deluje v načinu, kot smo želeli (slika 12).

6.1.2 Zagotavljanje zanesljivosti testov

Ko smo izvedli osnoven postopek snemanja in izvajanja testov, gremo naprej na bolj zanesljive načine izvedbe obstoječih testov. Tu se moramo osredotočiti predvsem na enoličnost gradnikov. Če ima gradnik stalne lastnosti, ga preprosto lociramo in tako smo prepričani, da bo pod določenimi oznakami vedno dosegljiv. Poleg lastnosti pa se lahko spremeni tudi lokacija gradnika. Da bolje razumemo kje je težava, moramo poznati nekaj osnovnih načinov definicije lokacije elementa.

Načini lociranja elementa na spletni strani (gumb za iskanje v iskalniku Google):

- HTML (ne deluje v Selenium)
  `<input type="submit" jsaction="sf.chk" name="btnK" aria-label="Iskanje Google" value="Iskanje Google">`,
- `ID` (identifikator)
  `By.id("btnK")`,
- `XPATH` (zapiše celotno pot do gradnika)
  `/html/body/div/div[3]/form/div[2]/div[3]/center/input[1]`,

- MODIFICIRAN XPATH (spremenjena oz. skrajšana različica XPATH-a). Če se v zgornjem primeru lokacija nekoliko spremeni je element nedosegljiv. Zato moramo sami zagotoviti, da lahko element lažje lociramo oz. mu določimo enolični identifikator
  `/html/body/div/div/form/div/div/center/input[contains(@value,'Iskanje Google')]`
  Lahko pa gremo še stopnjo dlje in odstranimo vse nepotrebne gradnike
  `//input[contains(@value,'Iskanje Google')]`.

Primer:

1. Googlova iskalna vrstica ima lokacijo, ki je enolično določena (`lst_ib`) in zato ne more priti do napake.
2. Ikona na strani YouTube pa takšnega identifikatorja nima, ima le ime, ki se lahko spremeni. `//a[@id='logo-container']/span`. V takem primeru potrebujemo drug način, da lahko z gotovostjo rečemo, da imamo pravi gradnik.

Seveda pa ima tudi svoje pomanjkljivosti. Najbolj izstopa možnost prenosa datotek na stran ali strežnik, saj ne zna uporabljati gradnikov tipa `button`. Delovanje je omejeno samo na gradnike `input`. Če je platforma preveč obremenjena, lahko pride tudi do odstopanj v rezultatih. Zato lahko Selenium prekorači čas, ki je namenjen čakanju na odziv spletne strani in gradnik se ne prikaže pravočasno. Ta problem rešimo tako, da med gradnike, ki povzročijo, da se test ne izvede do konca, vrinemo neko časovno zakasnitev (slika 13). Dolžina zakasnitve je odvisna od zahtevnosti predhodne operacije (da zagotovimo delovanje je povprečno dovolj od 1 do 5 sekund).

```
@Test
public void testUntitled() throws Exception {
  driver.get(baseUrl + "/?gws_rd=cr,ssl&ei=oLncVZH8MYO4swHq0pqIAg");
  driver.findElement(By.id("lst-ib")).clear();
  driver.findElement(By.id("lst-ib")).sendKeys("Gradniki");
  driver.findElement(By.linkText("64123 Gradniki sistemov vodenja - Univerza v Ljubljani")).click();
  driver.findElement(By.linkText("64620 Gradniki v tehnologiji vodenja - Univerza v Ljubljani")).click();
  driver.findElement(By.xpath("//div[@id='blueBarNAXAnchor']/div/div/div/div/h1/a/i")).click();
  Thread.sleep(3000L); //Zakasni delovanje za 3000 milisekund
}
```

Slika 13: Metoda za zakasnitev delovanja

Pomanjkljivost Seleniuma je v načinu preverjanja obstoja elementov v določeni tabeli. Tu je prednost v veliki skupnosti razvijalcev, ki se je s tem problemom že srečala, ga uspešno rešila in ga objavila na forumih (slika 14).

```
public void FindElementInTable(String Id, String Result) {
    destColumn = 0; // Desired column if found is bigger than 0
    destRow = 0; // Desired row if found is bigger than 0
    int rowCount = driver.findElements(By.xpath(Id)).size();
    rowCount++;
    for (int row = 1; row < rowCount; row++) {
        // Gets all columns inside every row
        int columnCount = driver.findElements(By.xpath(Id + "[" + row + "]/td")).size();
        columnCount++;
        for (int column = 1; column < columnCount; column++) {
            // fetches the cell value based on row & column
            String cellValue = driver.findElement(By.xpath(Id + "[" + row + "]/td[" + column + "]")).getText();
            if (cellValue.equals(Result)) {
                destColumn = column;
                destRow = row;
                return;
            }
        }
    }
}
```

Slika 14: Metoda za iskanje gradnikov v tabeli

Lahko se zgodi, da se kljub varnostim ukrepom kakšen klik ne more izvesti v prvem poizkusu. V takem primeru potrebujemo metodo, ki zna izvesti določeno število poizkusov klika elementa in nam v primeru napake ustrezno poroča o stanju (slika 15). Gre za zelo preprosto metodo, kjer se zanka izvede v določenem številu iteracij.

```
public void LoopForEffect(String Id, String Text, int type) {
    // 1 - By id with a value
    // 2 - By XPath with value
    // 3 - By XPath as click
    int attempts = 3;
    while (attempts >= 0) {
        if (attempts == 0) {
            throw new ElementNotVisibleException("This element is not avalible or is not correctly defined: " + Id);
        } else {
            try {
                switch (type) {
                    case 1:
                        WebElement elId = driver.findElement(By.id(Id));
                        elId.clear();
                        elId.sendKeys(Text);
                        attempts = -1;
                        break;
                    case 2:
                        WebElement elXPath = driver.findElement(By.xpath(Id));
                        elXPath.clear();
                        elXPath.sendKeys(Text);
                        attempts = -1;
                        break;
                    case 3:
                        WebElement clXPath = driver.findElement(By.xpath(Id));
                        clXPath.click();
                        attempts = -1;
                        break;
                    case 4:
                        WebElement clId = driver.findElement(By.id(Id));
                        clId.click();
                        attempts = -1;
                        break;

                    default:
                        break;
                }
            } catch (Exception ex) {
            }
            attempts--;
            try {
                Thread.sleep(1000);
            } catch (InterruptedException e) {
            }
        }
    }
}
```

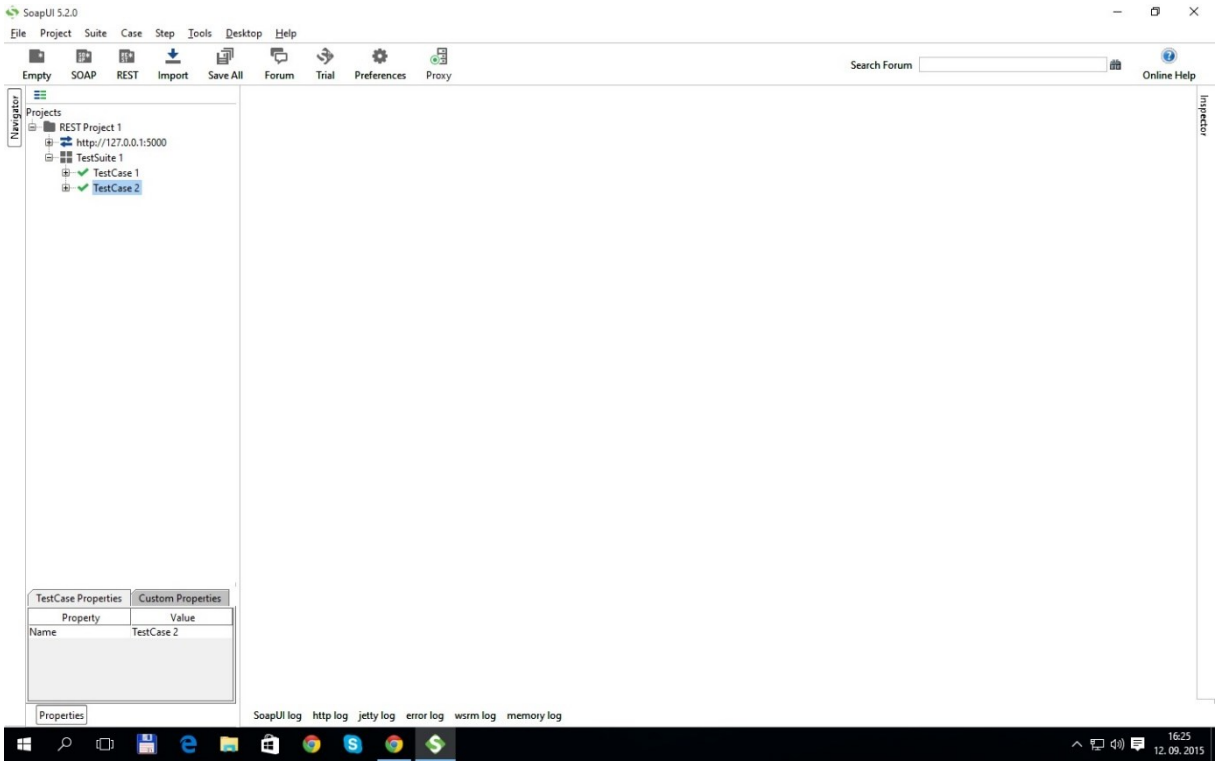
Slika 15: Metoda, ki ponavlja ukaz do izpolnitve

Ko smo pokrili vse osnovne in nekaj naprednih konceptov, lahko dokaj samozavestno dodajamo nove teste v prepričanju, da je naša rešitev zanesljiva in odporna na spremembe.

6.2 Funkcijsko testiranje

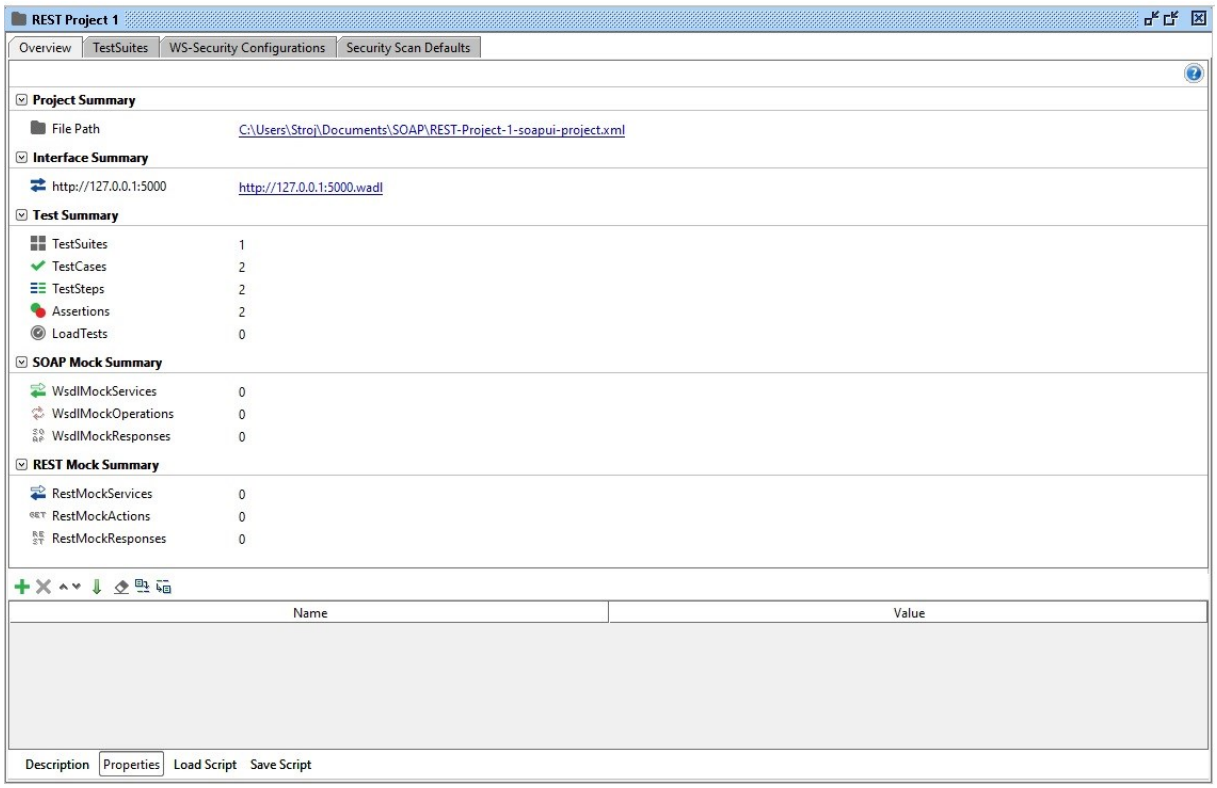
Drugi primer vzpostavitve avtomatskega testiranja je na platformi RESTFul storitev [28]. Te delujejo tako, da na vprašanje v dogovorjeni obliki (XML, JSON, tekst) vrnejo ustrezen odgovor.

Kot smo ugotovili že z analizo, je za naše testiranje najbolj primerna programska rešitev orodje SOAP UI [24]. Ker osnovna različica že zadovolji našim potrebam, nam ni potrebno plačevati licence za uporabo.



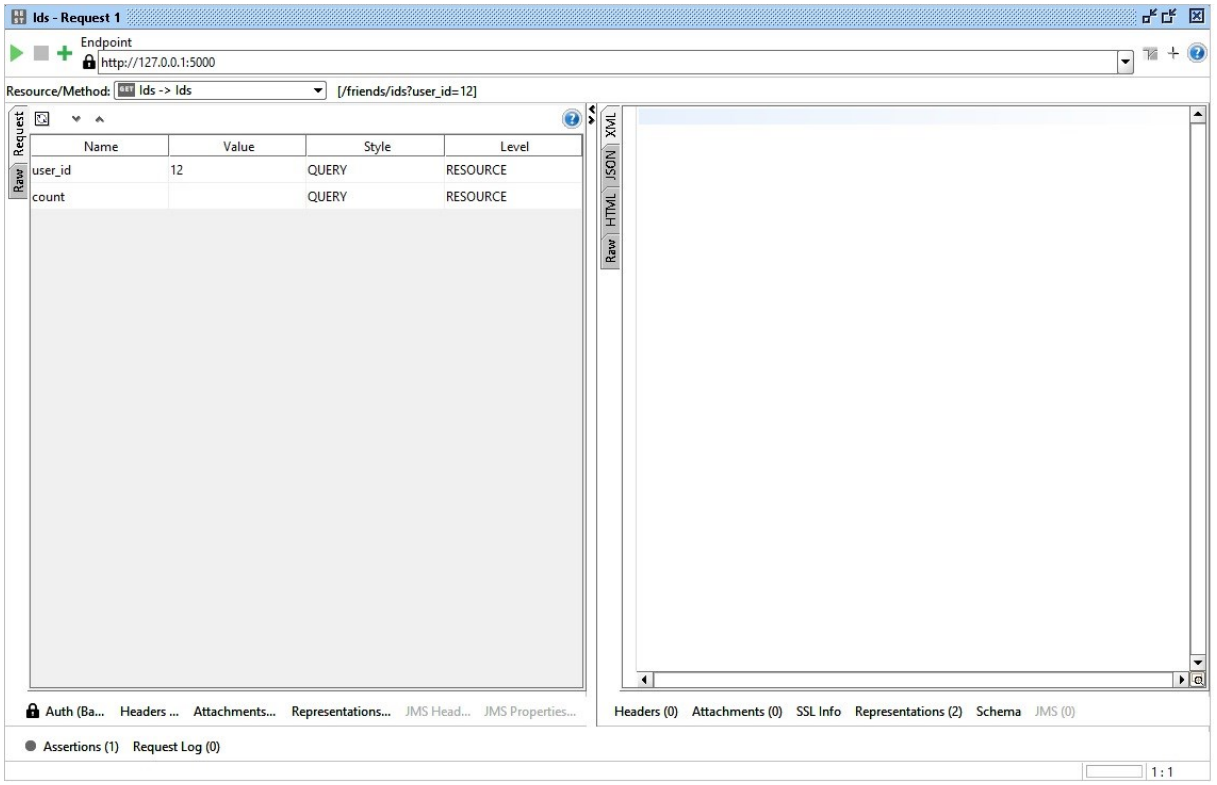
Slika 16: Nadzorna plošča v SOAP UI

Pri kratkem pregledu programske opreme je za neizkušenega programerja uporabniški vmesnik relativno nepregleden (slika 16). Samo dodajanje povezave na strežnik (URI) in ime projekta je relativno preprosto. Glede na naše želje izberemo tip projekta SOAP ali REST (v temu primeru izberemo REST, saj na takšen način delujejo naše storitve) (slika 17).



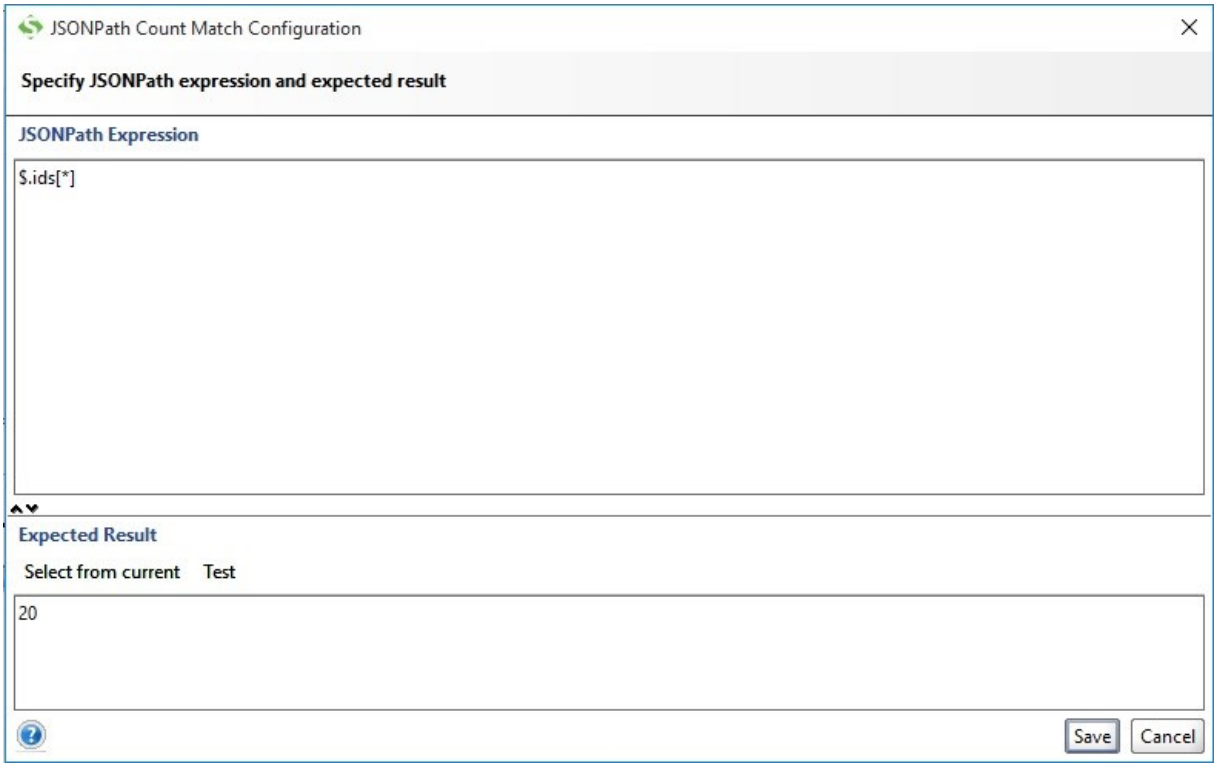
Slika 17: SOAP UI projekt

Ko imamo povezavo pripravljeno (v tem primeru `localhost:5000` oziroma `127.0.0.1:5000`), moramo ustvariti testni paket (ang. test suite). Tu bomo grupirali teste glede na področje testiranja. Znotraj teh skupin bomo definirali le testne primere, ki se bodo izvajali na spletnih storitvah.



Slika 18: SOAP UI testni primer

V zgoraj navedenem primeru pričakujemo določene vrednosti spremenljivk (slika 18). Ker pa včasih to ni dovolj, dodamo še točno določeno število vrednosti, ki jih pričakujemo. To storimo tako, da v možnost `Assertion` zapišemo pričakovano število rezultatov (slika 19).



Slika 19: Primer pričakovanega števila rezultatov

V našem primeru pričakujemo natanko 20 številskih vrednosti.

Ta rešitev ima omejitve na področju določanja dinamičnih rezultatov (drug uporabnik lahko dobi za isti primer drugačne rezultate), zato je potrebno zelo podrobno definirati rezultate, ki so konstantni. Poleg tega pa je opis pri napakah zgolj obvestilo, da se test ni izvedel po pričakovanjih brez definiranih podrobnosti kaj točno ni ustrezalo zahtevam.

Poglavje 7 Predstavitev celote

Vse gradnike, ki smo jih izbrali, združimo v celovito okolje. Najprej jih je potrebno namestiti za posamezne dele.

7.1 Namestitev aplikacijskega strežnika in orodja Jenkins

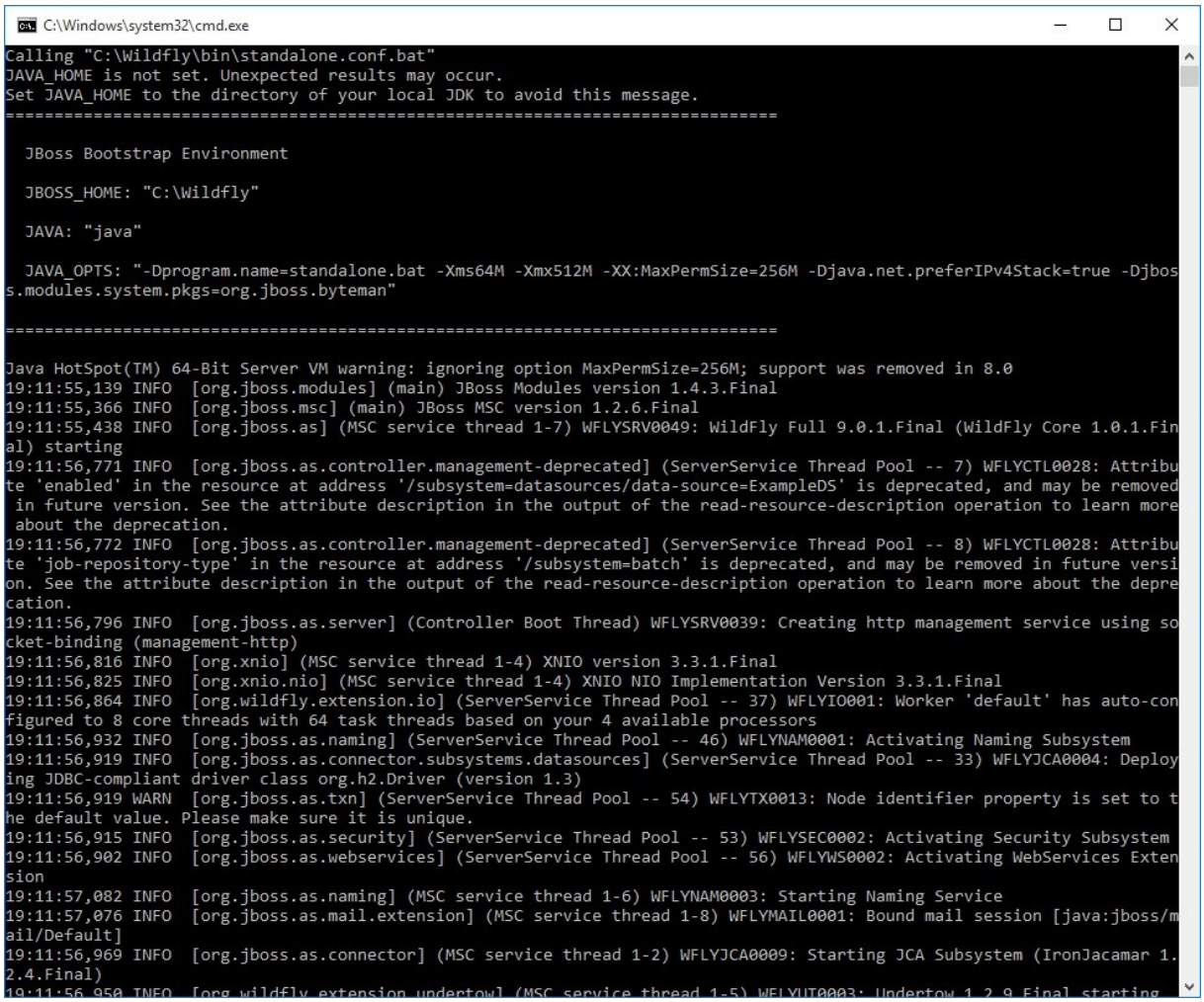
Iz uradne strani prenesemo najnovejšo stabilno različico strežnika.

To datoteko razširimo v namensko mapo (primer `C:/Wildfly`), saj bomo tako lažje spisali opravila v naslednjih korakih v Gradle-u. Wildfly za delovanje potrebuje Javo (JRE ali JDK), zmogljivost pa je odvisna od različice Jave:

- 32 bitna (največ 2GB pomnilnika, v primeru prekoračitve uporabe pomnilnika pride do odpovedi sistemov ob uporabi, edina rešitev je ponovni zagon aplikativnega strežnika),
- 64 bitna (tu ni problema z veliko porabo pomnilnika, vendar Wildfly ne odstani vseh začasnih datotek v pomnilniku; posledično se zasedenost pomnilnika nenehno povečuje in sčasoma je potreben ponovni zagon).

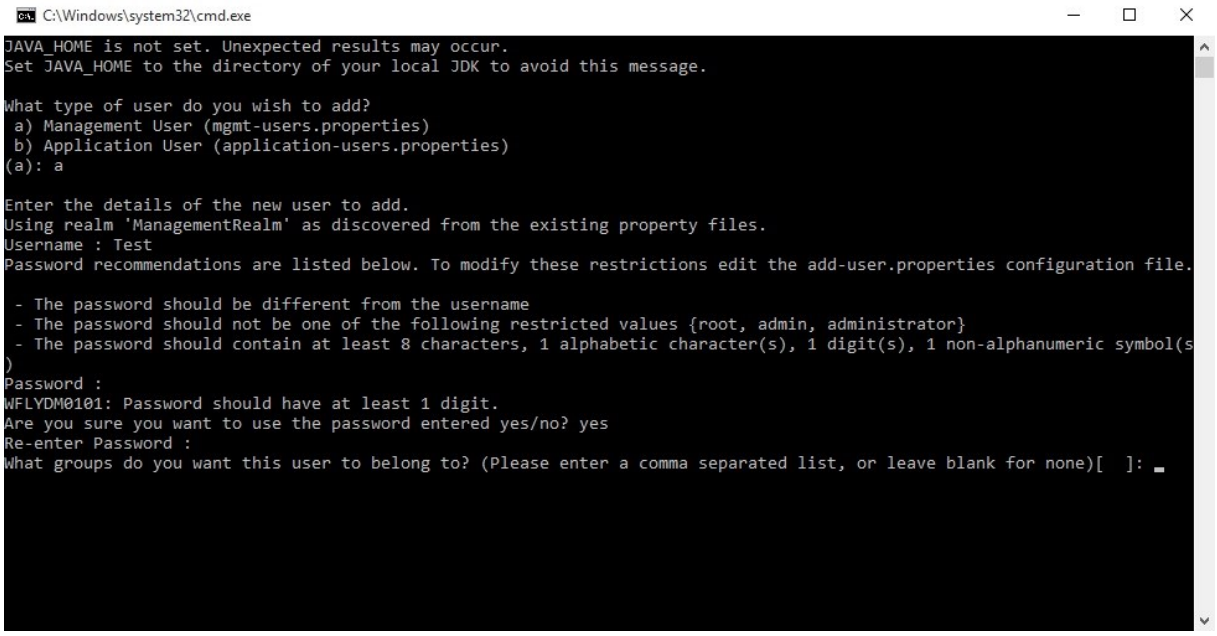
Glede na nameščeno različico Jave in količino pomnilnika, nastavimo največjo količino namenjenega pomnilnika v datoteki `standalone.conf.bat` vrednost `-Xmx` (privzeto 512MB, priporočeno 4096 MB).

Ko končamo z nastavitvami, lahko poženemo datoteko `standalone.bat`. V nekaj trenutkih se pojavi ukazni poziv, ki pripravlja strežnik (slika 20).



Slika 20: Izpis ukaznega poziva pri Wildfly

Pred začetkom urejanja strežnika moramo dodati vsaj enega uporabnika, ki se bo lahko vpisal na strežnik. To je omogočeno preko skripte `add-user.bat`. Sledimo navodilom na ukaznem pozivu in si ustvarimo uporabnika (slika 21).

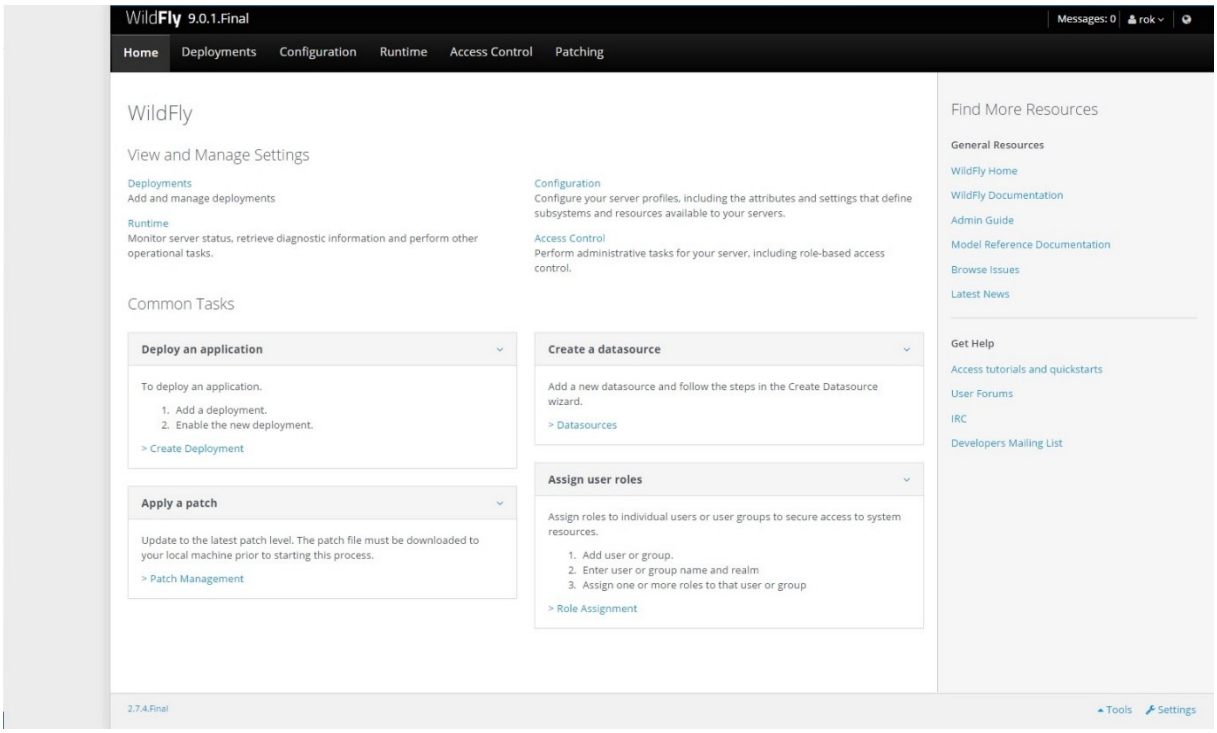


Slika 21: Dodajanje uporabnika v Wildfly

Z ustvarjenim uporabniškim računom lahko preverimo delovanje tako, da odpremo brskalnik in v vnosno polje vnesemo vrednost `localhost:8080`. Če je vse pravilno nastavljeno, bi se morala prikazati stran za vstop (slika 22).

Slika 22: Wildfly začetna stran

S tem smo se prepričali, da naš aplikacijski strežnik deluje. Za grafični prikaz delovanja izberemo možnost `Administration Console`. V tem koraku vnesemo izbrano uporabniško ime in geslo ter vse skupaj potrdimo. Ob uspešni prijavi se odpre domača stran zavihka Wildfly nadzorne plošče (slika 23).

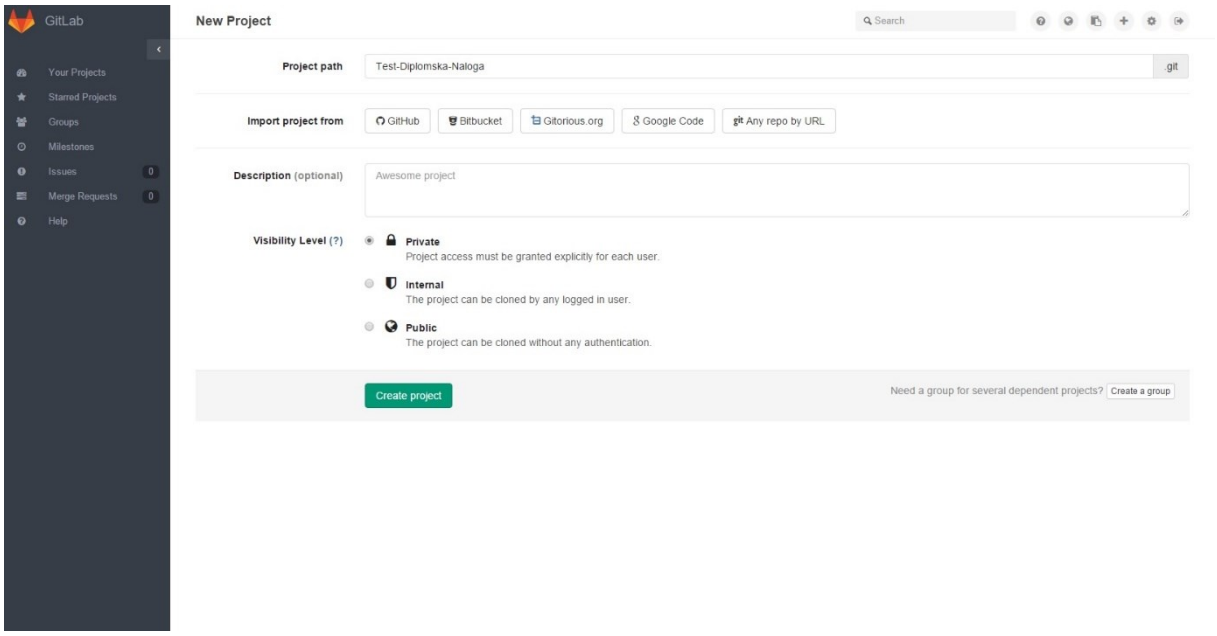


Slika 23: Wildfly nadzorna plošča

7.2 GIT sistem za nadzor različic

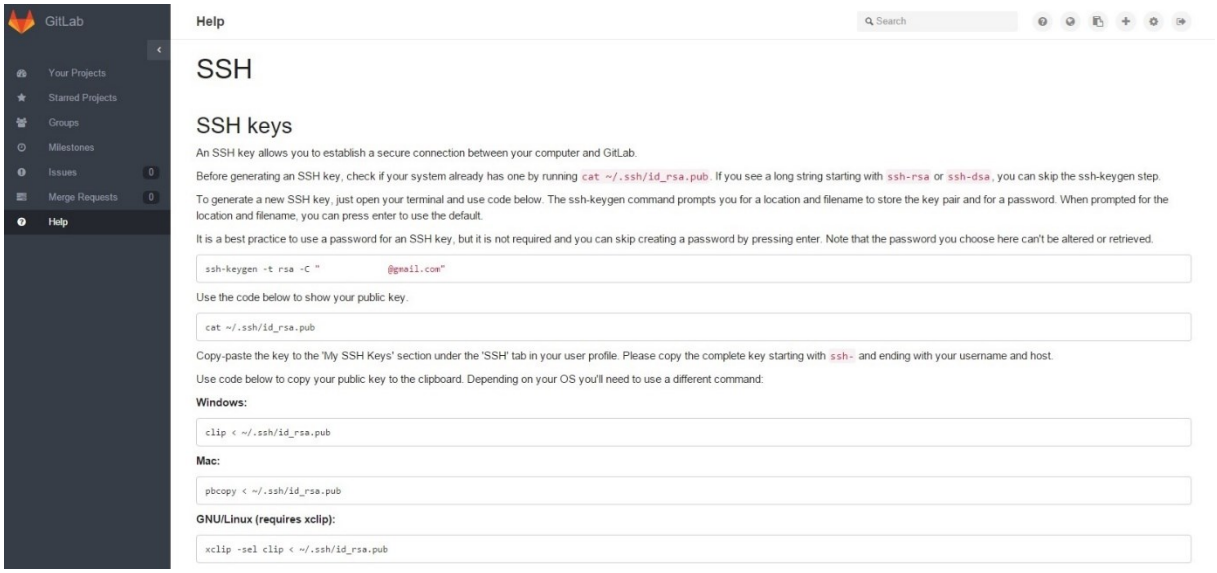
Preden lahko uporabljamo GitLab, si moramo na računalniku z aktivnim repozirotijem, namestiti Git. To storimo tako, da gremo na njihovo uradno stran in si prenesemo odjemalca. Pri namestitvi je pomembno, da izberemo tudi konzolni način, saj ga bomo kasneje potrebovali.

Sedaj se moramo registrirati na strani GitLab (določimo uporabniško ime in geslo). To storimo tako, da gremo na njihovo glavno stran in kliknemo gumb `Sign in`. Tam vnesemo obvezne podatke. V nekaj minutah na vaš izbran email naslov prispe pošta, ki vas poziva, da potrdite vašo prijavo v GitLab. Ko imamo potrjeno uporabniško ime in geslo, lahko ustvarimo projekt, kjer bomo hranili izvorno kodo naše aplikacije in potrebnih testov (lahko se hranijo posebej, ampak je bolj pregledno, če so skupaj, le da so v posebni mapi) (slika 24).

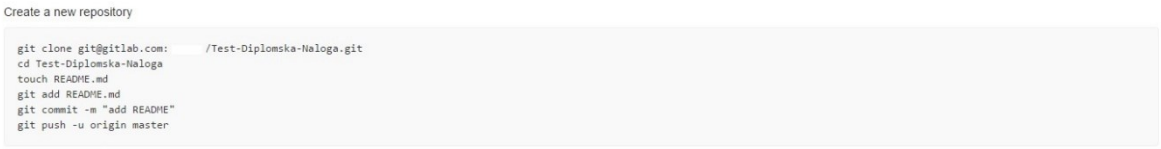


Slika 24: Dodajanje novega projekta v GITLab-u

Tu je najbolj pomembno, da izberemo stopnjo vidljivosti (angl. Visibility level) `Private`, saj s tem onemogočimo dostop nepooblaščenim osebam. Da bomo lažje nadzirali stanje na računalniku, kjer bo koda lokalno hranjena je najbolje, da si prenesemo neko programsko opremo za nadzor GIT-a. Ena od rešitev je TortoiseGit saj je enostavna in pregledna. Preden lahko pričnemo dodajati kodo ali datoteke, si moramo ustvariti SSH ključ, po katerem bo znano, da je to naš računalnik. V ta namen nam že ponudijo navodila za kreiranje in uporabo SSH ključa (slika 25). Ko to naredimo, moramo repozitorij prvič povezati ročno (slika 26).



Slika 25: Ustvarjanje SSH ključa za povezavo z GITLab-om



Slika 26: Povezava repozitorija na določeno mapo

To nam bo preneslo vse obstoječe podatke na izbrano lokacijo in mapo obravnavalo kot repozitorij, nad katerim se izvajajo operacije.

Sedaj lahko obstoječe datoteke kopiramo v mapo in jih takoj uveljavimo.

7.3 Uporaba Gradle in Gradle skript

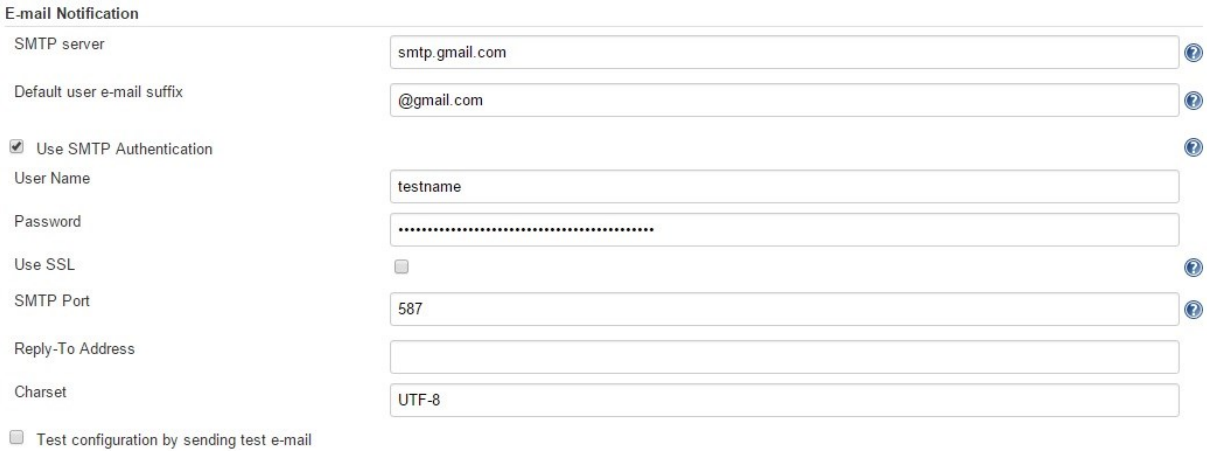
Za uporabo tega orodja moramo najprej prenesti program, ki namesti vse potrebne knjižnice in funkcije za delovanje. Obrnemo se na uradno spletno stran, kjer prenesemo namestitveni program. Ko datoteko prenesemo, jo razširimo v želeno mapo in nastavimo sistemsko spremenljivko `Path`. Tako je uporaba Gradle možna preko ukaznega poziva.

7.4 Jenkins in vtičniki

Z uspešno izvedenimi predhodnimi koraki se lotimo postavitve Jenkins-a na aplikacijski strežnik. Ko se aplikacija vzpostavi, za delovanje potrebnih podprogramov (Gradle, ANT, Groovy, SVN, GIT,…) izberemo potrebne vtičnike znotraj Jenkinsa. Za namestitev vseh potrebnih vtičnikov je potrebno okoli 30 minut in ponovni zagon za zagotovitev pravilnega delovanja. Do težav lahko pride ob nadgradnji orodja Jenkins. Nekateri vtičniki se samodejno odstranijo in jih je potrebno ponovno namestiti.

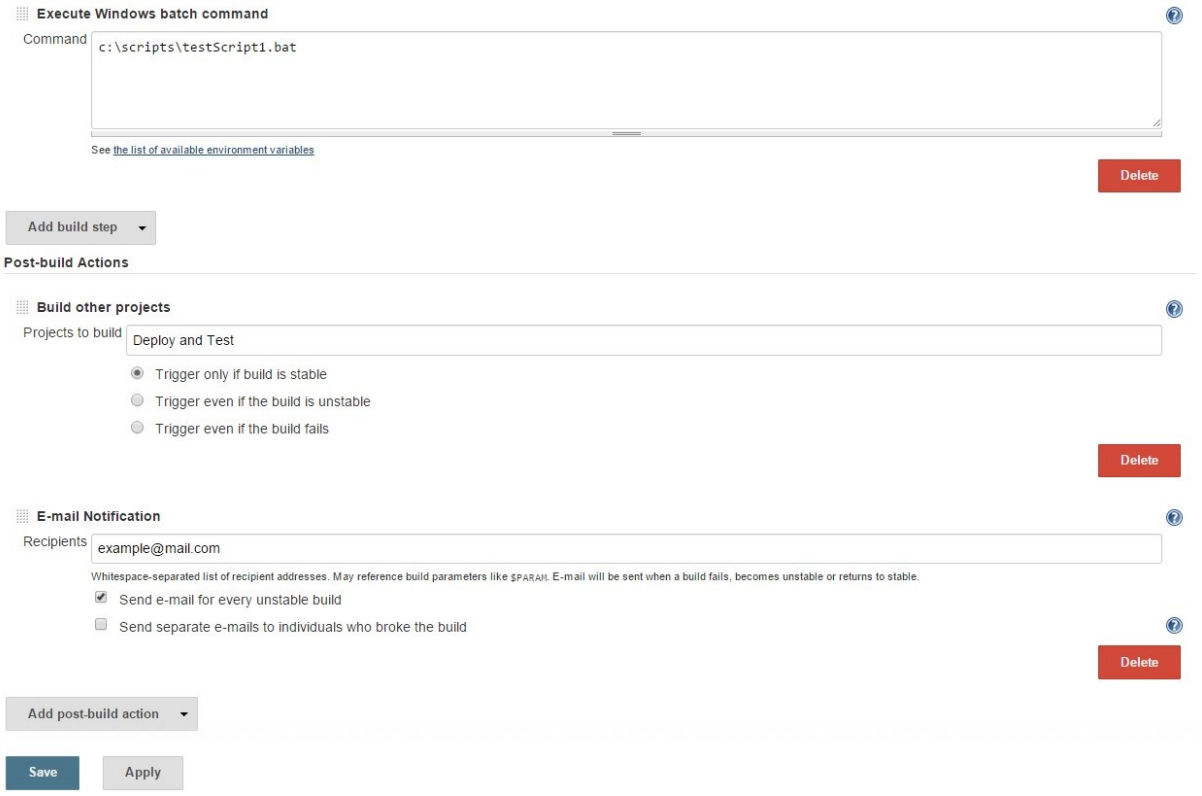
7.5 Poročanje o rezultatih preko elektronske pošte

Proces naj bi bil popolnoma avtomatiziran, zato želimo, da se poročila o uspešnosti pošljejo v ustrezni obliki. Seveda pa ni dovolj, da dobimo le obvestilo o izvedbi procesa, ampak želimo podroben seznam in opis uspešnosti izvedbe. V primeru nadaljnje uspešne izvedbe ni potrebno pošiljati obvestil, saj se predvideva, da programska rešitev deluje pravilno. Za samo namestitev lahko v Jenkinsu pride do zapletov, saj zahteva vpis v poštni predal z uporabniškim imenom in geslom (slika 27). V ta namen je najbolje ustvariti namenski poštni predal, ki pošilja obvestila. Je pa na to temo odprtih veliko tem na forumih in na straneh za podporo.



Slika 27: Nastavitve spletne pošte v Jenkinsu

Ko smo uspešno nastavili poštni predal, lahko na opravilo dodamo obveščanje (slika 28). Oblika podatkov je določena z dodatki k objavami rezultatov.



Slika 28: Nastavitve v Jenkins opravilu

Poglavje 8 Časovna primerjava testiranja povprečnega programa

Največja prednost avtomatskih testov je prihranek časa. Veliko število programov porabi do 50% razvojnega časa za testiranje [35]. V zadnjih letih so bili narejeni veliki koraki k temu, da se ta odstotek občutno zmanjša.

Tudi v slovenskem podjetju, ki bi želelo uvesti takšno avtomatizirano delovanje in testiranje, bi bil ključni faktor porabljen čas oziroma prihranek časa. V ta namen je bila narejena časovna primerjava ročnega in avtomatskega testiranja.

Ob ročnem preverjanju delovanja je z napakami najbolj izstopal človeški faktor, saj je bila izvedba testov odvisna od poznavanja programske opreme, hitrosti vnosa besedila in posameznih klikov. Poleg vseh teh faktorjev pa je potrebno upoštevati še zmotljivost (napačen vnos besedil), pozabljivost (nekaj testov ni bilo izvedenih) in opuščanje ponovnega testiranja v predhodnih fazah že izvedenih testov (pomanjkanje časa).

Tu se je dobro zavedati, da je napaka, odkrita znotraj podjetja, veliko manj škodljiva kot napaka, ki jo odkrije stranka (zaupanje v pravilno delovanje aplikacije pade, s tem pa se tudi izgublja dobro ime podjetja).

Z vsemi naštetimi morebitnimi težavami je bilo ocenjeno, da je povprečen čas testiranja vsake različice celotne aplikacije od 50 do 70 minut. Ta številka se morda ne zdi velika, vendar je tu pomembna informacija, da se nove različice aplikacije z manjšimi ali večjimi spremembami tvorijo večkrat dnevno (povprečno trikrat na dan).

Opis faze	Ročne faze postopka	Avtomatizirane faze postopka
Objava kode programerja	Programer napiše in objavi kodo (dodatek, sprememba, nova koda)	Programer napiše in objavi kodo (dodatek, sprememba, nova koda)
Izgradnja kode	Programer na osnovi kode programerja izdela novo različico programske opreme	Skripta v ozadju izdela novo različico programske opreme in jo namesti na strežnik
Testiranje starih funkcionalnosti	Na osnovi specifikacij starih (nespremenjenih) funkcionalnosti mora programer ročno preveriti	Izvedejo se vsi obstoječi testi na starih (nespremenjenih) funkcionalnostih brez izjeme oz. v

	delovanje vseh do tedaj delujočih funkcionalnosti	celoti
Napake na starih funkcionalnostih	V primeru napak pristopi k odpravljanju le teh, saj so nove programske spremembe povzročile napake pri že delujočih funkcionalnostih	V primeru napak pristopi k odpravljanju le teh, saj so nove programske spremembe povzročile napake pri že delujočih funkcionalnostih
Testiranje novih funkcionalnosti	Ročno mora preveriti delovanje novo programiranih funkcionalnosti	Dodajo se novi oz. spremenijo se obstoječi avtomatizirani testi
Napake na novih funkcionalnostih	Programer poišče napako in jo odpravi ter zopet izvede celotno proceduro od začetka	Programer poišče napako in jo odpravi ter zopet izvede celotno proceduro od začetka

Načeloma obstajata dve glavni možnosti izvedbe testiranj in sicer:

- stalno testiranje ob vsaki novi različici in sprotno odpravljanje napak (časovno in stroškovno potratno, vendar nujno za zagotavljanje nivoja zanesljivosti in delovanja programske rešitve),
- testiranje pred zaključno izdajo nove različice (nevarnost v primeru prevelikega števila napak in možnost nedelovanja programske rešitve do določenega roka izdaje).

Da bi se izognili veliki porabi časa, bi lahko izvedli teste samo ob načrtovani nadgradnji progama, vendar takrat ne bi imeli časa za odpravo morebitnih napak.

Dokler dela na posameznem projektu samo en programer, je zadeva stroškovno še mogoče sprejemljiva (kakovost testov je subjektivna stvar). Takoj, ko pri projektu sodeluje več programerjev, pa so časovne izgube za testiranje in s tem stroški, pomnoženi s številom sodelujočih pri projektu.

Če podatke izračunamo za posameznega programerja, to dnevno pomeni povprečno 150 – 210 minut manj porabljenega časa, ki bi ga lahko izkoristil za razvoj na drugem področju.

Predlagani sistem je bil tudi dejansko uveden in v obdobju enega meseca vpeljan v redno izvajanje. S tem se je dnevno porabljen čas, namenjen testom, iz 150 minut zmanjšal na 10 minut (zgolj manjši popravki) oziroma 30 minut (če se dodaja teste za nove funkcionalnosti).

Zasedenost programerja dnevno	Ročno	Avtomatizirano
Manjši popravki in testi	150 minut	10 minut (93 % prihranek)
Večji popravki in testi	210 minut	30 minut (85 % prihranek)

Kot vidimo iz tabele, se na dnevni ravni ta razlike občutno poznajo, hkrati pa smo lahko prepričani, da ni prišlo do napak oziroma izpuščenih testov. Kljub temu, da je razlika velika, je potrebno upoštevati, da je za vpeljavo avtomatiziranega sistema potrebno vložiti kar veliko časa (približno en mesec za osnovne teste, odvisno od velikosti programske rešitve).

Čeprav imajo avtomatski testi veliko prednosti, uspešnost uvedbe le-teh ni vedno zagotovljena. Ob nepravilnem načrtovanju in izbiri neustreznih rešitev se lahko zgodi, da projekt zaradi izgubljene investicije celo propade [36].

Poglavje 9 Sklepne ugotovitve

Cilj diplomske naloge je bil predstavitev popolne avtomatizacije postopka testiranja programske opreme. Pri vsakem koraku je bilo vloženega veliko truda v iskanje ustrezne programske opreme, ki je primerna tako za izvedbo koraka kot tudi za medsebojno združljivost s programskimi rešitvami pri drugih korakih. Osredotočili smo se na aktualne sisteme, ki jih podjetja izdelujejo kot spletne rešitve. Vsak del postopka je bil podrobno prikazan, na koncu pa predstavljen kot del celovitega okolja. Ko so bile te rešitve ustrezno pregledane in preučene, je sledil postopek povezovanja korakov med seboj. Ko so vsi koraki delovali pravilno in opravljali svoje naloge, je bila celovita rešitev pripravljena za uporabo na praktičnem primeru. Želeni dosežki so bili optimizacija delovanja programske rešitve in bistven prihranek časa za preverjanje delovanja aplikacije.

Takšno okolje je v dejanskem podjetju že v uporabi. Sistem je relativno stabilen, vendar brez vsakodnevnega nadzora (na 10-30 minut) ne deluje popolnoma pravilno. Kljub manjšim težavam je prihranek časa velik, okolje pa takoj obvešča tako o neuspehih kot tudi o korakih, kjer je prišlo do napake (primer: napačna deklaracija, napačna pretvorba objekta, …). Vodilni v podjetju so z okoljem zadovoljni, programerji pa z uporabo navodil lahko relativno enostavno dodajajo ali spreminjajo teste.

Kljub našim dosežkom bi bilo možno na področju aplikacijskih strežnikov zagotoviti večjo stabilnost z uporabo plačljivih rešitev, saj so narejene za uporabo na poslovnem nivoju. Prav tako bi bilo potrebno takoj, ko bo na voljo, uporabiti boljšo rešitev na področju aplikacijskih testov.

Uporabljeni viri

[1] Scrum - agilna metodologija razvoja programske opreme [Online]. Dosegljivo na http://scrummethodology.com/ (november 2015)

[2] Agilne metodologije razvoja programske opreme [Online]. Dosegljivo na http://agilemethodology.org/ (november 2015)

[3] Jenkins CI [Online]. Dosegljivo na https://jenkins-ci.org/ (november 2015)

[4] SVN – Subversion sistem za nadzor različic (angl. version control system) [Online]. Dosegljivo na https://subversion.apache.org/ (november 2015)

[5] GIT – Sistem za nadzor različic [Online]. Dosegljivo na https://git-scm.com/doc (november 2015)

[6] ANT orodje za nadzor aplikacij [Online]. Dosegljivo na http://ant.apache.org/ (november 2015)

[7] Groovy programski jezik [Online]. Dosegljivo na http://www.groovy-lang.org/ (november 2015)

[8] Gradle programski jezik [Online]. Dosegljivo na http://gradle.org/ (november 2015)

[9] GitLab CI [Online]. Dosegljivo na https://about.gitlab.com/gitlab-ci/ (november 2015)

[10] GitLab ponudnik podatkovnega gostovanja[Online]. Dosegljivo na https://about.gitlab.com/ (november 2015)

[11] Apache Continuum [Online]. Dosegljivo na https://continuum.apache.org/ (november 2015)

[12] CruiseControl [Online]. Dosegljivo na http://cruisecontrol.sourceforge.net/ (november 2015)

[13] YUILL, Simon. Concurrent versions system. *Software Studies: A Lexicon*, 2008, 64.

[14] Tortoise SVN [Online]. Dosegljivo na http://tortoisesvn.net/about.html (november 2015)

[15] JBoss aplikacijski strežnik [Online]. Dosegljivo na http://www.jboss.org/ (november 2015)

[16] Wildfly aplikacijski strežnik [Online]. Dosegljivo na http://wildfly.org/about/ (november 2015)

[17] WebLogic aplikacijski strežnik [Online]. Dosegljivo na https://www.oracle.com/middleware/weblogic/suite.html (november 2015)

[18] WebSphere aplikacijski strežnik [Online]. Dosegljivo na http://www.ibm.com/software/websphere (november 2015)

[19] SVNKit knjižnica za uporabo SVN metod [Online]. Dosegljivo na http://svnkit.com/ (november 2015)

[20] Red Hat [Online]. Dosegljivo na http://www.redhat.com/en (november 2015)

[21] Smartbear TestComplete [Online]. Dosegljivo na http://smartbear.com/product/testcomplete/overview/ (november 2015)
[22] TestPartner [Online]. Dosegljivo na http://microfocus.com/products/testpartner/ (november 2015)
[23] SilkTest [Online]. Dosegljivo na www.borland.com/en-GB/Products/Software.../Silk-Test (november 2015)
[24] AutoIt [Online]. Dosegljivo na https://www.autoitscript.com (november 2015)
[25] Selenium [Online]. Dosegljivo na http://www.seleniumhq.org/ (november 2015)
[26] SOAtest [Online]. Dosegljivo na https://www.parasoft.com/product/soatest/ (november 2015)
[27] TestStudio [Online]. Dosegljivo na www.telerik.com/teststudio (november 2015)
[28] Richardson, Leonard, and Sam Ruby. RESTful web services. O'Reilly Media, Inc. 2008.
[29] Smartbear SOAP UI [Online]. Dosegljivo na http://www.soapui.org/ (november 2015)
[30] Flask [Online]. Dosegljivo na http://flask.pocoo.org/ (november 2015)
[31] Python [Online]. Dosegljivo na https://www.python.org/ (november 2015)
[32] TestRail [Online]. Dosegljivo na www.gurock.com/testrail/ (november 2015)
[33] Eclipse [Online]. Dosegljivo na https://eclipse.org/ (november 2015)
[34] NetBeans [Online]. Dosegljivo na https://netbeans.org/ (november 2015)
[35] Dustin, Elfriede, Thom Garrett, and Bernie Gauf. *Implementing automated software testing: How to save time and lower costs while raising quality*. Pearson Education, 2009.
[36] HOFFMAN, Douglas. Cost benefits analysis of test automation. *STAR West*, 1999, 99.