Univerza v Ljubljani
Fakulteta za računalništvo in informatiko

Aleksandra Bersan

Okostje za testiranje PHP aplikacij z oblačnimi storitvami

DIPLOMSKO DELO

UNIVERZITETNI ŠTUDIJ RAČUNALNIŠTVA IN INFORMATIKE

Mentor: doc. dr. Mojca Ciglarič

Ljubljana, 2016

Fakulteta za računalništvo in informatiko izdaja naslednjo nalogo:

Tematika naloge:

Oblačne storitve omogočajo cenovno učinkovito rabo računalnikov na mnogih področjih. Oblačne storitve so še posebej uporabne na področju razvoja programske opreme, kjer razvijalcem omogočajo lažje sodelovanje znotraj ekip in dostop do računskih zmogljivosti, ki presegajo tiste, ki so dostopne na običajnih delovnih postajah. Eno od področij razvoja, ki zahteva precejšnje računske zmogljivosti, je testiranje programske opreme.

Preučite možnosti uporabe oblačnih storitev za podporo testiranja in sprotne integracije. Preglejte področje testiranja, izberite in opišite ustrezne tehnologije in zgradite lastno okostje za izvajanje testiranja v oblaku. Izdelek kritično ovrednotite.

Zahvala.

Zahvaljujem se svoji mentorici doc. dr. Mojci Ciglarič ter viš. pred. dr. Igorju Rožancu za pomoč in svetovanje pri pisanju diplome. Zahvaljujem se svojim staršem za vso skrb in podporo, ki so mi jo nudili. Zahvaljujem se Roku Andree za nasvete in ideje, Vladi Semenovi in Katarini Čepon za spodbudo v pravih trenutkih, pa tudi vsem ostalim prijateljem, ki so mi pomagali tako ali drugače.

Дорогу осилит идущий.

Kazalo

Povzetek

Abstract

1 Uvod 1

2 Pregled področja 3

2.1 Ročno in avtomatizirano testiranje 4

2.2 Princip bele in črne škatle 4

2.3 Testi enot 5

2.4 Integracijski testi 5

2.5 Sistemski testi 6

2.6 Sprejemni testi 6

2.7 Regresijski testi 7

2.8 Testiranje kot storitev 7

2.9 Zvezna integracija 7

2.10 Kakovost programske opreme 8

3 Uporabljene tehnologije 11

3.1 Git in GitHub 11

3.2 PHP in Composer 13

3.3 PHPUnit 13

3.4 Codeception 15

3.5 Travis CI 17

3.6 Scrutinizer CI 19
3.7 Demo aplikacija 20

4 Implementacija **25**
4.1 Zgradba okostja 26
4.2 Uporaba okostja 29

5 Zaključek **39**

Literatura **41**

Povzetek

Naslov: Ogrodje za testiranje PHP aplikacij z oblačnimi storitvami

Cilj pričujočega diplomskega dela je ustvariti okostje odprtokodnega PHP projekta, ki vključuje podporo za avtomatsko testiranje, zvezno integracijo in analizo kakovosti. Namen okostja je enostavna integracija dodatnih načinov testiranja v že obstoječi PHP projekt ter omogočanje avtomatizirane aktivacije testiranja in analize kakovosti.

Okostje dobimo s povezavo različnih orodij in storitev. Projekt se mora nahajati na GitHub strežniku, ki ga povežemo s ponudnikom zvezne integracije Travis CI. Za analizo kakovosti uporabimo orodja, ki jih ponuja Scrutinizer, za pisanje testov pa uporabimo orodje Codeception.

Za izbrano demo aplikacijo prikažemo postopek integracije okostja, pripravimo različne teste ter preizkusimo povezane storitve.

Ključne besede: testiranje, git, GitHub, Codeception, Travis CI, Scrutinizer.

Abstract

Title: Skeleton for PHP application testing in a cloud

The goal of this diploma thesis was to create an open-source PHP project skeleton containing a prepared configuration for automated testing, continuous integration and code quality analysis. The skeleton can be used by any project hosted on GitHub. By using the skeleton, a programmer can quickly set up a testing stack supporting several types of tests which supports automatic triggering of tests and which produces a code quality report.

The skeleton uses Codeception to integrate various test types, Travis CI as the continuous integration service provider and Scrutinizer for code analysis. The skeleton is hosted on GitHub under a Free Software license. As part of this thesis, the skeleton was also tested on a demo blog application and the experience documented.

Keywords: testing, git, GitHub, Codeception, Travis CI, Scrutinizer.

Poglavje 1

Uvod

Namen pričujočega diplomskega dela je ustvariti okostje (angl. skeleton) odprtokodnega PHP projekta, ki vključuje podporo za avtomatsko testiranje, zvezno integracijo in analizo kakovosti. Izbira programskega jezika PHP je vplivala na izbor orodij in storitev, ki smo jih povezali. Okostje smo uporabili na primeru manjšega projekta, uporabno pa naj bi bilo tudi za večje projekte.

Testiranje je zelo pomemben del razvoja programske opreme, ki ima več ciljev. V fazi razvoja s testiranjem poskušamo odkriti in preprečiti napake. S tem lahko tudi izboljšamo kvaliteto programa in povečamo zadovoljstvo uporabnika oz. naročnika. Posledice testiranja se poznajo tudi po končanem razvoju, saj za kvalitetno programsko opremo običajno velja, da ima nižje stroške vzdrževanja [1].

Dandanes aplikacije postajo čedalje bolj kompleksne, zato vzpostavljanje in vzdrževanje testnega okolja postaja za podjetja čedalje večje časovno in finančno breme. S pojavitvijo oblačnih storitev so se odprle tudi nove možnosti na področju testiranja. Pojavil se je tudi nov koncept storitve – testiranje kot storitev (angl. Testing as a Service oz. TaaS) [2].

Naš cilj je povečati uporabo različnih načinov testiranja in avtomatizirati sam proces izvajanja testiranja. Zato najprej v poglavju 2 opravimo pregled področja testiranja programske opreme in predstavimo osnovne pojme, ki jih kasneje uporabljamo. Nato v poglavju 3 opišemo orodja, ogrodja in

tehnologije, ki smo jih uporabili za izgradnjo zamišljenega projektnega okostja. Predstavimo ga v poglavju 4. Prav tako je v tem poglavju na primeru manjše PHP aplikacije prikazana uporaba ustvarjenega okostja. V 5 poglavju so predstavljene ugotovitve, ki so nastale med študijem dokumentacije predstavljenih ogrodij.

Poglavje 2

Pregled področja

S testiranjem programske opreme želimo pridobiti potrditev, da je narejena v skladu s specifikacijskimi zahtevami, ter odkriti in odpraviti čim več napak in pomanjkljivosti [3]. S tem povečamo kvaliteto produkta in posledično povečamo zadovoljstvo uporabnika. Testiranje tudi prispeva k zvišanju učinkovitosti razvojnega procesa ter k zmanjšanju vzdrževalnih stroškov.

S testiranjem se trudimo odkriti čimveč napak, vendar ne moremo zagotaviti, da bomo identificirali vse obstoječe napake v preiskovani aplikaciji. To je povezano s tem, da obstajata dve vrsti napak – napake, ki nastanejo med kodiranjem, ter napake, ki nastanejo že pri načrtovanju programske opreme. Načrtovalske napake je teže odkrivati in odpravljati.

Obstaja več delitev testiranja po različnih kriterijih. Glede na način testiranja ločimo ročno in avtomatizirano. Glede na poznavanje vsebine kode govorimo o principih bele in črne škatle. Lahko delimo metode testiranja glede na to, katere elemente kode testirajo. Iz te skupine so najbolj pogosti testi enot in sprejemni testi.

Končni namen vseh metod testiranja je zagotoviti ali izboljšati kvaliteto programske opreme. Kvaliteto pa opredeljujejo funkcionalnost, zanesljivost, učinkovitost, uporabnost, vzdrževalnost in prenosljivost. In na osnovi teh faktorjev tudi lahko delimo metode testiranja v dve skupini – metode testi-

ranja, ki preverjajo funkcionalnosti aplikacije, in metode testiranja, ki preverjajo performančne lastnosti oz. zmogljivosti.

V nadaljevanju so predstavljeni načini in metode testiranja ter pojmi, ki so povezani s področjem preiskovanja te diplomske naloge.

2.1 Ročno in avtomatizirano testiranje

Glede na način izvajanja ločimo ročno in avtomatizirano testiranje. Pri ročnem testiranju tester prevzame vlogo končnega uporabnika in poizkuša identificirati napake ali nepričakovana obnašanja programske opreme. Prednost takega testiranja je v nižjih kratkoročnih stroških in večji fleksibilnosti, pomanjkljivost pa je v težji ponovljivosti nekaterih testov. Pri zelo hitrem razvoju, ko se izvorna koda dodaja tudi večkrat na dan, je ročno testiranje lahko zelo problematično.

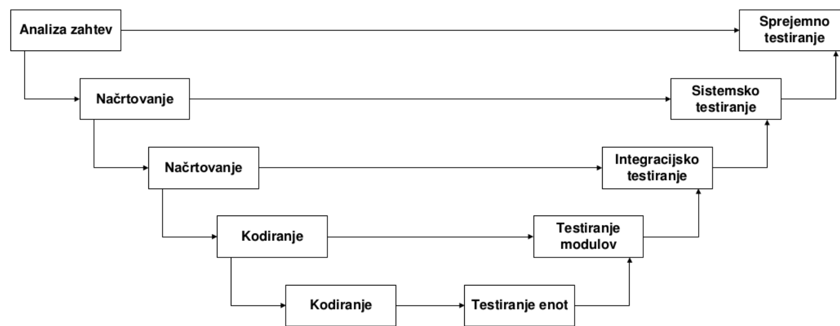
Za razliko od ročnega se avtomatizirano testiranje izvaja s pomočjo skript in dodatnih programskih orodij. Avtomatizacija testov skrajša čas in poveča efektivnost testiranja, zagotavlja ponovljivost testov in lahko generira dokumentacijo. Vendar ima tudi ta način svoje pomanjkljivosti. Kompleksna orodja za testiranje so lahko zelo draga. Za avtomatizacijo obstoječih ročnih testov je potreben dodaten čas. V programski opremi lahko obstajajo napake, ki jih je možno odkriti samo z ročnim testiranjem.

2.2 Princip bele in črne škatle

Z vidika osebe, ki testira programsko opremo, ločimo dve metodi za testiranje:

- Testiranje z metodo bele škatle (angl. white-box tests), ko ima tester vpogled v programsko kodo, pozna njeno strukturo in namen [6].

- Testiranje z metodo črne škatle (angl. black-box tests), ko tester ne pozna notranje sestave sistema in izvaja testiranje funkcionalnosti na podlagi znanih vhodnih in izhodnih parametrov [7].



Slika 2.1: Faze razvoja in testiranja.

2.3 Testi enot

Testi enot (angl. unit test) se izvajajo vzporedno s kodiranjem in predstavljajo najbolj osnovno obliko testa. S to metodo testiramo posamezne dele kode, modulov in procedur; njen cilj je pokazati, da le-ti delujejo pravilno.

Enota je po navadi najmanjši del aplikacije, ki jo je še možno testirati. Lahko kot enoto določimo na primer posamezno metodo ali razred v objektno usmerjenem programiranju, lahko pa celoten modul v proceduralnem programiranju. V tem primeru govorimo o testiranju modulov. Za pisanje tovrstnih testov potrebujemo znanje o notranjem delovanju kode, zato je najbolje, da to opravi sam programer že med kodiranjem ali pred njim.

Pri testiranju enot smo omejeni z obsegom kode in pokažemo samo pravilnost ločenih delov, ne pa aplikacije kot celote.

2.4 Integracijski testi

Integracijski testi (angl. integration test) preverjajo načrtovalno fazo razvoja programske opreme in testirajo, če komunikacija med enotami pravilno deluje. Pri tovrstnih testih se kombinirajo različni moduli in/ali komponente aplikacije in se testirajo kot skupina. Cilj testov je potrditi, da skupine enot izpolnjujejo funkcionalne, učinkovne in zanesljivostne zahteve specifikacije.

Obstajata dva pristopa k integracijskemu testiranju: od spodaj navzgor (angl. bottom-up) in od zgoraj navzdol (angl. top-bottom). Pri prvem načinu testiranje temelji na že opravljenem testiranju enot in združi te enote v smiselno skupino. Pri drugem načinu testiranja se najprej preveri delovanje celotnega modula, nato se ga postopoma 'razstavi' do velikosti enot. V praksi se uporabljata oba načina tako, da se najprej izvede integracijsko testiranje od spodaj navzgor in potem še od zgoraj navzdol, pri čemer se zastavljajo testni scenariji, ki se najbolj približujejo dejanski uporabi oz. procesu izvajanja.

2.5 Sistemski testi

Sistemski testi (angl. system tests) tudi preverjajo načrtovalno fazo razvoja programske opreme, vendar za razliko od integracijskih testov testirajo delovanje združenih komponent kot celotnega sistema. Pri klasičnem načinu razvoja programske opreme tej fazi sledijo sprejemni testi, ko vlogo testerja prevzamejo naročniki in uporabniki. Pri novejših agilnih metodologijah so sistemski testi zajeti znotraj sprejemnih in se jih redko smatra kot ločeno skupino.

2.6 Sprejemni testi

Sprejemni testi (angl. acceptance tests) v razvojnem procesu programske kode ustrezajo fazi analize zahtev. Njihov glavni namen je preverjanje aplikacije s strani naročnika in ugotavljanje, ali so funkcionalnosti narejene v skladu s specifikacijo. Za agilen način razvoja programske opreme sprejemni test predstavlja testno implementacijo uporabniških scenarijev in pri testiranju preverjamo, ali aplikacija ustreza potrebam naročnika.

2.7 Regresijski testi

Regresijski testi (angl. regression tests) so poseben sklop testov, ki se izvajajo ob večjih spremembah aplikacije, na primer, ko dodamo novo funkcionalnost. Z njihovo pomočjo ugotavljamo, ali obstoječ del aplikacije še zmeraj deluje pravilno oz. če posodobljena aplikacija kot celota deluje v skladu s specifikacijo ter ali dodane spremembe vplivajo na druge dele aplikacije. Po navadi gre za nek vnaprej določen izbor testov, ki pokrivajo vse glavne funkcionalnosti aplikacije [3].

2.8 Testiranje kot storitev

Koncept "testiranje kot storitev"je leta 2009 predstavilo podjetje Tieto v svojem poslovnem strateškem načrtu (v sodelovanju s podjetjem IBM) [4]. Testiranje kot storitev je definirano kot ponujanje statičnega ali dinamičnega testiranja na zahtevo (angl. on-demand) z uporabo oblaka in oblačnih storitev [5].

Testiranje z uporabo oblaka ponuja naslednje prednosti [2]:

- Oblak ponuja širok spekter resursov in omogoča postavitev različnih neodvisnih infrastruktur.
- Oblak lahko dodeli resurse za testiranje na zahtevo (angl. on-demand), s čim poenostavi postavljanje testnega okolja in zmanjša stroške.
- Elastičnost oblačnih storitev ponuja najboljše prilagajanje testnega okolja potrebam testiranja.
- Testiranje se lahko izvede na različnih platformah.
- Nastavitev in nadzor testnega okolja v oblaku je enostavno za uporabnika.

2.9 Zvezna integracija

Zvezna integracija je praksa razvoja programske opreme, pri kateri razvijalci večkrat na dan objavijo oz. dodajo novo kodo tekočemu projektu. Ob vsa-

kem dodajanju kode se najprej sproži proces izgradnje (angl. build), nato sledi preverjanje nastavitev, odvisnosti, zagon testov ter drugi mehanizmi za zgodnje odkrivanje napak.

Prvi je pojem zvezne integracije (angl. Continuous integration – CI) uporabil Grady Booch pri opisu Boochove metode, vendar njegova definicija ni predvidevala integracije večkrat dnevno. Današnjo definicijo pa je prvi uporabil Kent Beck pri opisu ekstremnega programiranja.

Glavna prednost zvezne integracije je urejeno in pregledno dodajanje nove kode, kar je zelo pomembno za projekte, kjer dela več razvijalcev hkrati. Ko gre za skupino razvijalcev, lahko pride do situacij, ko imamo zelo veliko sprememb in potrebujemo precej dela in časa, da te spremembe pregledamo in integriramo v projekt. Z uporabo zvezne integracije povečamo kvaliteto kode, stabilnost celotnega projekta, pospešimo celoten razvojni postopek ter izboljšamo zavedanje o stanju projeta.

Postavitev in nastavitev lastnega sistema za zvezno integracijo je kompleksna in časovno potratna. Vendar obstajajo ponudniki te storitve na oblačnih platformah [21].

2.10 Kakovost programske opreme

Za kakovost programske opreme (angl. code quality) dandanes obstaja več standardov in modelov kakovosti. Pri merjenju kakovosti naj bi merili naslednje osnovne faktorje: funkcionalnost, učinkovitost, uporabnost, zanesljivost, varnost in prenosljivost. Te faktorje pa lahko ocenjujemo z različnih vidikov bodisi z vidika razvijalca bodisi uporabnika. Zato je moč zaslediti pojma notranje in zunanje kakovosti. Notranja kakovost je povezana s tem, kako pojem kakovosti dojemajo razvijalci. Zanje je pomembna skladnost s specifikacijo, število odkritih napak na fazo, število odpovedi na časovno enoto itn. Zunanja kakovost pa posledično odraža pogled uporabnika. V tem primeru se preverja skladnost s potrebami in se meri zadovoljstvo pri uporabi programa.

Testna pokritost (angl. code coverage) je trenutno najbolj priljubljena metrika za merjenje notranje kakovosti. Z njeno pomočjo najlažje odkrijemo vrstice kode, ki niso preverjene v testih, kar nam omogoča enostavno identifikacijo robnih primerov, ki jih nismo predvideli v fazi razvoja, oz. za katere nismo pripravili testov.

Metrika testne pokritosti temelji na rezultatih izvedenih testov ter uspešnosti prevedene kode, zato pravimo, da gre za dinamično analizo. Ta analiza potrebuje veliko procesorskih in spominskih resursov in lahko upočasni graditev kode pri zvezni integraciji. Zato računanje metrik testne pokritosti izvajamo v ločeni fazi, ko se zaključi faza testiranja.

Poglavje 3

Uporabljene tehnologije

V tem poglavju bomo opisali ogrodja in tehnologije, ki jih nameravamo povezati skupaj. Na izbiro orodij je vplivala predvsem izbira programskega jezika in uporaba GitHuba.

3.1 Git in GitHub

Git je leta 2005 zasnoval Linus Torvalds, ki je sicer znan kot avtor operacijskega sistema Linux. Potreboval ga je ravno za pregled razvoja linux jedra v sodelovanju z drugimi programerji [10].

Git je sistem za upravljanje z izvorno kodo. Je distribuiran sistem, ki omogoča hitrejši razvoj, dobro integriteto podatkov in podpira vzporedne, nelinearne tokove dela. Git se lahko uporablja lokalno na posameznem računalniku in/ali preko omrežnega sistema.

Lokacija oz. direktorij, kjer se nahaja projekt, se v git-terminologiji imenuje repozitorij. Lahko gre za lokalno mapo na osebnem računalniku, nek prostor pri javnem ponudniku git-storitve ali za lokacijo znotraj privatnega git-sistema.

Git hrani celotno zgodovino sprememb na projektu v drevesni strukturi. To doseže preko posnetkov (angl. snapshot) stanja, ki jih dobimo, ko izvedemo ukaz git commit. Vsak posnetek ima lahko enega ali več staršev.

Zaradi te lastnosti git omogoča tudi 'vračanje' v eno od prejšnjih stanj projekta. Še ena dobra lastnost, ki jo s tem dobimo, je zagotavljanje pristnosti kode. V vsakem trenutku se lahko s preverjanjem hash-oznak preveri, da v preteklosti nihče ni spreminjal kode.

Še ena dobra lastnost gita so veje (angl. branch), ki ustvarjajo vzporedne tokove dela. V projektu vedno obstaja glavna veja (angl. master), lahko pa jih naredimo več. Ko se v nekem trenutku 'odcepi' veja, vsebuje posnetek stanja svojega vira. Torej: od veje se lahko odcepi nova veja, ki ima isto stanje kot je v starševski veji, ne pa tako, kot je v tistem trenutku naprimer v glavni veji. Ponavadi znotraj veje naredimo določene spremembe, npr. implementiramo novo funkcionalnost, odpravimo neko napako itn. Ko je razvoj na veji zaključen, se njena vsebina združi (angl. merge) s starševsko vejo. Z delom v ločenih vejah lahko poskrbimo, da se v glavni veji vedno nahaja delujoča koda oz. produkcijska verzija projekta [11].

Pri delu z gitom se najpogosteje uporabljajo naslednji ukazi [13]:

git init: inicializira nov git repozitorij

git status: izpiše, kateri veji pripada vsebina trenutnega direktorija in našteje datoteke, vsebina katerih se razlikuje od zadnje slike stanja (angl. snapshot) te veje

git add: zabeleži, katere datoteke naj se kasneje dodajo

git commit: ustvari nov posnetek stanja trenutnega direktorija/repozitorija

git merge: združi vsebino (razvojne) veje z vsebino starševske veje

git push: objavi lokalne spremembe v repozitorij na git-strežnik

git pull: pobere z git-strežnika obnovljeno različico repozitorija

git clone: naredi lokalno kopijo repozitorija

Ukazi se po navadi dajejo preko ukazne vrstice lupine oz. terminala. Pomanjkanje grafičnega vmesnika predstavlja za današnje uporabnike glavno slabost gita. Zato se git pogosto uporablja preko enega od ponudnikov za gostovanje git repozitorijev, ki omogočajo vizualizacijo vsebin repoziorija. Najbolj znani med njimi so GitHub, Bitbucket in GitLab. GitHub poleg

spletnega pregleda in urejenja vsebin ponuja tudi grafično aplikacijo za lokalno uporabo gita.

GitHub v bistvu ni samo ponudnik storitve git ter eden izmed največjih ponudnikov prostora za skupinske projekte, ampak je tudi socialno omrežje. Uporabniki lahko brskajo po javnih projektih, puščajo komentarje in pripombe, opozarjajo na napake in predlagajo izboljšave. S tem dobi GitHub dodatno vrednost kot odličen resurs za učenje.

3.2 PHP in Composer

Composer je program za upravljanje paketov za programski jezik PHP, ki v posebnih datotekah hrani seznam vseh nameščenih knjižnic in njihovih različic. V datoteki composer.json so zabeležena imena in različice knjižnic. Datoteka composer.lock pa ima podatke o dejansko nameščenih različicah in podrobne podatke o paketih – opis, tip licence, odvisne knjižnice, avtorji, ključne besede itn.

Izvorna koda 3.1 prikazuje primer datoteke composer.json.

Druge ključne besede, ki se lahko uporabljajo v composer.json datoteki in njihove vloge:

`require` – vsebuje naštete pakete, ki jih potrebuje (angl. dependencies)

`require-dev` – vsebuje naštete pakete, ki jih potrebuje za razvoj ali testiranje

`config` – dodatne konfiguracijske opcije

`scripts` – opcija za prilagajanje inštalacijskega postopka

`extra` – dodatne informacije, ki so potrebne za navedene ukaze znotraj ključnega polja

3.3 PHPUnit

PHPUnit je ogrodje za testiranje enot aplikacij, ki so napisane v programskem jeziku PHP. Namenjeno je testiranju manjših odsekov kode za hitro

```
{
    "name": "vendor_name/lib_name",
    "description": "One liner description",
    "keywords": ["keyword"],
    "homepage": "https://github.com/git_user/repository_name"
       ,
    "type": "library",
    "license": "MIT",
    "authors": [{
        "name": "John Doe",
        "email": "john@example.com",
        "homepage": "http://example.com"
    }],
    "require": {
        "php": ">=5.3.0"
    },
    "autoload": {
        "psr-0": {"MyLibraryNamespace": "src/"}
    }
}
```

Izvorna koda 3.1: Datoteka composer.json

odkrivanje in odpravljanje napak. Pravilnost delovanja testov enot ugotavlja na osnovi trditev (angl. assertions) [22].

Zgradbo ogrodja sestavljajo naslednje komponente:

- Test runner: program, ki zaganja teste in posreduje poročila o rezultatih testiranja.
- Test case: osnovni razred testa.
- Test fixture: množica pogojev, ki določajo uspešnost izvedenega testa.
- Test suites: množica testov, za katere veljajo isti pogoje za izvedbo.
- Test execution: sta dve metodi: setup() in teardown(), ki nastavljata testne pogoje.
- Test result formatter: orodje, ki nastavi XML obliko dobljenih rezultatov.
- Assertions: trditve, ki določajo rezultate testov.

V datoteki phpunit.xml.dist (glej izvorno kodo 3.2) se nahaja konfiguracija za PHPUnit ogrodje. Če so testi shranjeni v mapi tests v korenu repozitorija, potem je to dovolj za izvajanje testiranja.

3.4 Codeception

Codeception je kompleksno ogrodje za testiranje PHP aplikacij. S tem ogrodjem lahko izvajamo več vrst testiranja – teste enot, integracijske teste, teste funkcionalnosti, teste grafičnega vmesnika (angl. API tests) ter sprejemne teste. Codeception vsebuje več kot 20 modulov za testiranje preko razvijalskih PHP ogrodij (Symfony, Laravel itn), podatkovnih zbirk (MySQL, PostgreSQL, MongoDB itn.) ter druge module. Codeception lahko izvaja teste, ki so napisani s pomočjo orodij PHPUnit in Selenium.

```xml
<?xml version="1.0" encoding="UTF-8"?>

<phpunit backupGlobals="false"
         backupStaticAttributes="false"
         colors="true"
         convertErrorsToExceptions="true"
         convertNoticesToExceptions="true"
         convertWarningsToExceptions="true"
         processIsolation="false"
         stopOnFailure="false"
         syntaxCheck="false"
         bootstrap="tests/bootstrap.php"
>
    <testsuites>
        <testsuite name="Example Test Suite">
            <directory>./tests/</directory>
        </testsuite>
    </testsuites>

    <filter>
        <whitelist>
            <directory>./src/</directory>
        </whitelist>
    </filter>
</phpunit>
```

Izvorna koda 3.2: Primer datoteke phpunit.xml.dist

Codeception se enostavno namesti preko Composer orodja. Pomembno je, da se med inštalacijo v datoteko composer.json doda referenca na Codeception ter nastane nastavitvena datoteka .codeception.yml.

Zapis v konfiguracijski datoteki se lahko spreminja preko boostrap skript. Primer izvorne kode 3.3 prikazuje privzeto konfiguracijsko datoteko s komentarji.

3.5 Travis CI

Travis CI je ponudnik oblačne storitve za zvezno integracijo za projekte, ki se nahajajo na GitHubu. Ob določenih dogodkih na repozitoriju se sproži avtomatska izgradnja projekta, njegovo testiranje in analiza. Travis podpira veliko programskih jezikov oz. tehnologij. Storitev je brezplačna za vse odprtokodne projekte, privatnim pa zaračunajo mesečno plačilo. [18]

Travis CI ima pripravljenih več posnetkov navideznih računalnikov (angl. virtual machine oz. VM) z različnimi konfiguracijami ter naborom programske opreme. Razvijalci so našli način, pri katerem ne zaganjajo vsakič nove slike navideznega računalnika (angl. VM) preko VirtualBoxa, ampak imajo en proces, ki zažene pravi posnetek strežnika. Izbira posnetka se določa preko programskega jezika in njegove verzije. [19]

Povezava projekta s Travis CI storitvami se naredi preko datoteke .travis.yml, ki se mora nahajati v korenskem direktoriju projekta. Ta datoteka vsebuje ključne podatke – uporabljeni programski jezik ter njegovo različico, opis okolja za izgradnjo, opis okolja za testiranje, vključuje seznam dodatnih paketov (angl. dependencies) ter morebitne druge parametre. Izvorna koda 3.4 prikazuje minimalno nastavitev datoteke .travis.yml.

Glede na željeno nastavitev se avtomatska izgradnja projekta ter izvajanje testov lahko sproži ob spremembah na glavni (angl. master) ali vzporedni (angl. branch) veji. Najpogosteje se postopek veže na objavljanje sprememb (ukaz git push) ali na prevzem vsebine (ukaz git pull).

Spodaj je nekaj ključnih besed, ki se uporabljajo v konfiguracijski dato-

```
actor: Tester

paths:
    # lokacija za shranjevanje modulov
    tests: tests
    # direktorij za podatke
    data: tests/_data
    # directorij za pomozno kodo
    support: tests/_support
    # directorij za izhodne in dnevniske datoteke
    log: tests/_output
    # directory za konfiguracijo okolja
    envs: tests/_envs

settings:

    # imena vseh uporabljenih bootstrap datotek
    bootstrap: _bootstrap.php
    # nakljucni vrstni red testiranja
    shuffle: true
    # uporabi barvno paleto
    colors: true
    # omejitev spomina, ki se lahko porabi
    memory_limit: 1024M
    # ali naj PHPUnit naredi varnostno kopijo
    backup_globals: true

# Konfiguracija globalnih modulov.
modules:
    config:
        Db:
            dsn: ''
            user: ''
            password: ''
            dump: tests/_data/dump.sql
```

Izvorna koda 3.3: Primer datoteke composer.json

```
language: php
php:
  - 5.5
  - 7.0
script: phpunit
```

Izvorna koda 3.4: Primer datoteke .travis.yml

teki, z opisom možnih nastavitev:

language - nastavi tehnologijo projekta

php - seznam verzij, za katere naj se izvede testiranje

env - opcija za nastavitev dodatnih lastnosti okolja (angl. environmet), npr. 'DB=mysql'

before_install - opcija za dodatno namestitev paketov, kot so npr. Ubuntu paketi

install - seznam knjižnic (angl. dependencies), ki jih potrebuje aplikacija

before_script - definiramo skripte, ki se izvedejo pred izvajanjem testiranja

script - nastavitev za orodje testiranja, prevzeto je PHPUnit

after_script - definiramo skripte, ki se izvedejo po testiranju

Za zgled: če se v projektu uporabljajo dodatne knjižnice, ki se nastavijo preko Composer orodja, moramo v datoteko travis.yml, dodati:

```
before\_script: composer install
```

3.6 Scrutinizer CI

Scrutinizer je analitično ogrodje za zvezno integracijo, ki deluje kot storitev. Omogoča več vrst analiz: za ugotavljanje kakovosti programske opreme

```
filter:
    paths: [src/*]
checks:
    php:
        code_rating: true
        duplication: true

tools:
    external_code_coverage: true
```

Izvorna koda 3.5: Primer datoteke .scrutinizer.yml

oz. projekta, s pomočjo katerih lahko odkrijemo hrošče (angl. bugs), varnostne ranljivosti (angl. security vulnerabilities) in neupoštevanje načel dobre prakse. Omogoča izračun več metrik testne pokritosti [23].

Scrutinizer ne izvaja testov, lahko pa ga povežemo s Travis CI, tako da bo ob vsakem novem preverjanju posredoval rezultate orodju Scrutinizer, ki jih bo uporabil pri svojih analizah ter generiranju poročil.

Izvorna koda 3.5 prikazuje primer datoteke scrutinizer.yml.

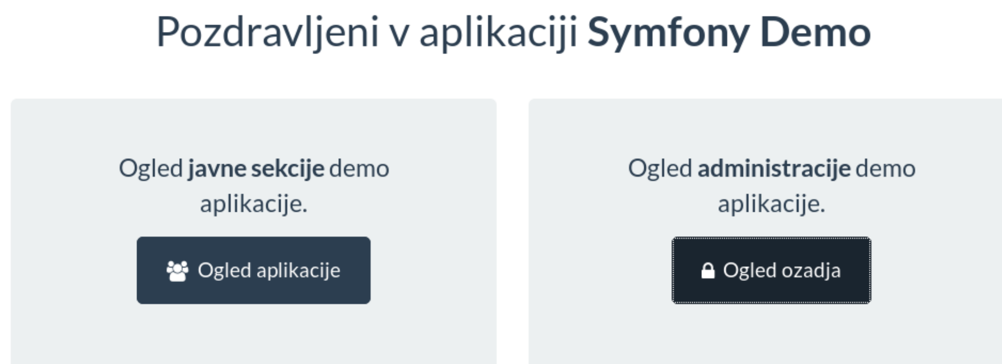
Povratno informacijo o uspešnosti in kakovosti lahko prikažemo v repozitoriju v datoteki README.md preko značk (angl. badge).

3.7 Demo aplikacija

Za namene testiranja pripravljenega okostja smo uporabili demo aplikacijo Symfony, ki je napisana v programskem jeziku PHP z uporabo ogrodja Symfony. To je enostavna blog aplikacija, ki je namenjena učenju ter testiranju novih funkcionalnosti [16].

Začetna stran (slika 3.1) nam daje dve možnosti: ogled javne sekcije aplikacije ter ogled administracijskega dela. Kasnejše strani vsebujejo gumb 'Prikaži kodo'; ob kliku nanj se prikaže pripadajoči del izvorne kode.

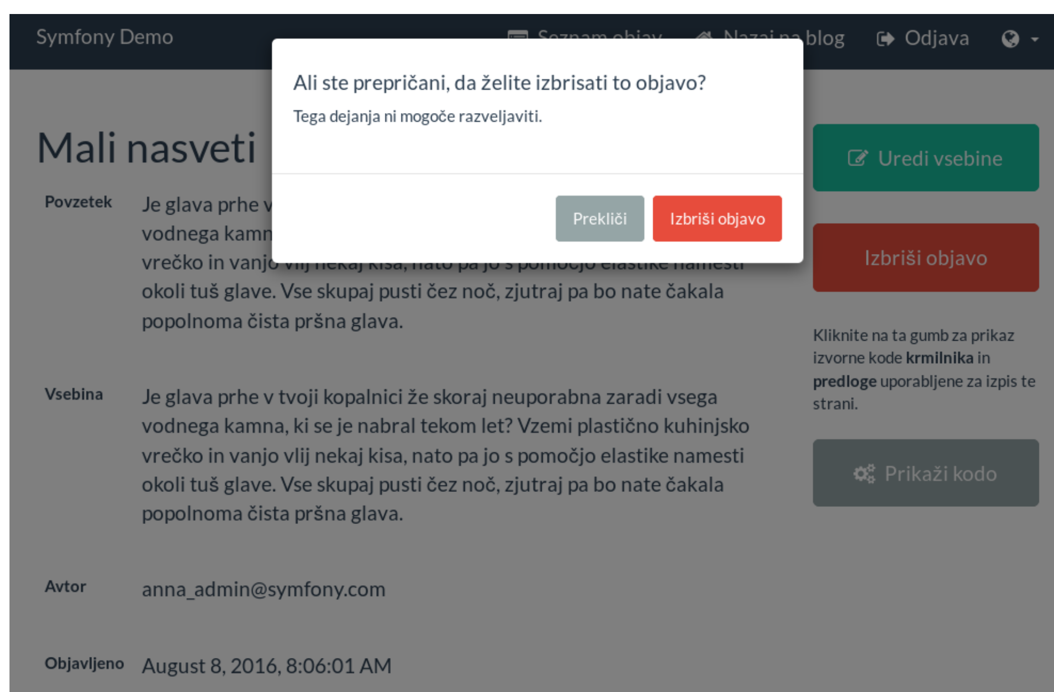
Javna sekcija aplikacije (slika 3.2) je namenjena ogledu in komentiranju zapisov.



Slika 3.1: Začetna stran demo aplikacije.

Slika 3.2: Javna sekcija demo aplikacije.

Slika 3.3: Administracijski del demo aplikacije.



Slika 3.4: Primer brisanja članka.

Preko administracijskega dela (slika 3.3) se lahko prijavijo uporabniki, ki imajo bodisi vlogo splošnega uporabnika bodisi vlogo administratorja. Splošni uporabniki lahko berejo in komentirajo članke v blogu. Administrator pa jih poleg tega lahko tudi dodaja, ureja in briše. Primer brisanja članka je prikazan na sliki 3.4 [17].

Poglavje 4

Implementacija

V skladu z namenom pričujočega dela v tem poglavju predstavimo, kako povežemo ogrodje za testiranje Codeception, ogrodje za izvajanje zvezne integracije Travis CI ter Scrutinizer CI, ki omogoča analizo kakovosti izvorne kode. Za odprtokodne projekte, ki se nahajajo na GitHubu, to pomeni, da imajo že pripravljeno okolje, preko katerega lahko brez posebnega truda opravijo avtomatsko izvajanje različnih vrst testiranja in zanje dobijo tudi analitično poročilo. V prvem delu poglavja so opisane konfiguracijske datoteke, ki sestavljajo okostje. Nato v drugem delu poglavja opisano okostje preizkusimo na demo aplikaciji, ki je bila predstavljena v poglavju 3.7. Za to aplikacijo je bilo napisanih tudi več testov. Nekaj od njih je tudi predstavljenih v tem delu poglavja.

Pripravljeno okostje poskrbi za izvajanje naslednjih korakov:

1. Z orodjem Codeception in/ali PHPUnit se pripravijo testi.

2. Spremembe z lokalne kopije repozitorija se dodajo v glavni repozitorij na GitHub.

3. Ob dodajanju sprememb se sproži izgradja projekta in avtomatsko testiranje na Travis CI.

4. Travis CI posreduje rezultate testov orodju Scrutinizer za analizo kakovosti ter vrne v repozitorij povratno informacijo o uspešnosti, ki se odraža v obliki značke.

5. Po končani analizi Scrutinizer objavi statistične rezultate ter javi v repozitorij povratno informacijo, ki se odraža v obliki značke.

6. Razvijalec na podlagi dobljenih rezultatov bodisi nadaljuje z razvojem bodisi z razhroščevanjem (angl. debug).

4.1 Zgradba okostja

V prejšnjem poglavju našteta ogrodja povežemo skupaj preko naslednjih konfiguracijskih datotek, ki se morajo nahajati v korenskem direktoriju GitHub repozitorija:

- composer.json
- codeception.yml
- .travis.yml
- .scrutinizer.yml

Obstajati mora mapa /tests, kjer se nahajajo testi. Če gre za spletno aplikacijo, narejeno s pomočjo orodja Symfony, se morata mapi z enotskimi ter funkcionalnimi testi premakniti v /src/AppBundle/tests direktorij.

4.1.1 Datoteka composer.json

Preko te datoteke javimo, da za testiranje uporabljamo orodje Codeception. Ta podatek potrebuje Travis CI, ko postavlja virtualno okolje za izgradnjo aplikacije. Zato v datoteki obvezno mora biti naslednji del kode:

```
{
    "require-dev": {
        "codeception/codeception": "~2.1"
    }
}
```

4.1.2 Datoteka codeception.yml

V tej datoteki znotraj definicije 'paths' podamo lokacijske nastavitve. Preko ključne besede 'tests' določimo, v katerem direktoriju se nahajajo testi. Preko

ključne besede 'log' nastavimo, v kateri direktorij se shranijo izhodne datoteke. Preko 'data' povemo, v kateri mapi imamo podatke.

Znotraj definicije 'settings' zapišemo, katero bootstrap skripto bomo uporabili, omogočimo barvni izpis rezultatov (v konzolo) in nastavimo spominsko omejitev. To varovalo je koristno, ker nekateri testi, po navadi - funkcijski, med izvajanjem porabijo veliko spominskega prostora.

Znotraj definicije 'modules' podamo nastavitve podatvne baze. Znotraj definicije 'coverage' pa omogočimo računanje testne pokritosti in podamo lokacijo, za katero naj se izvaja analiza.

```
actor: Tester
paths:
    tests: tests
    log: tests/_output
    data: tests/_data
    helpers: tests/_support
settings:
    bootstrap: _bootstrap.php
    colors: true
    memory_limit: 1024M
modules:
    config:
        Db:
            dsn: ''
            user: ''
            password: ''
            dump: tests/_data/dump.sql
coverage:
    enabled: true
    include:
        - src/*
```

4.1.3 Datoteka .travis.yml

To datoteko potrebujemo, da aktiviramo na Travis CI storitev zvezne integracije. S številom različic programskega jezika določimo, koliko testnih

okolij naj se postavi.

Pod ključno besedo 'install' poskrbimo, da je orodje Composer zagotovo nameščeno. Dodatno prenesemo tudi strežnik orodja Selenimum, z uporabo katerega po fazi testiranja posredujemo rezultate orodju Scrutinizer. Znotraj ključne besede 'before_script' zapišemo vse korake za namestitev strežnika. Pod ključno besedo 'script' pa zaženemo teste in nastavimo beleženje rezultatov.

```
language: php
sudo: false

php:
  - 5.5
  - 5.6

install:
  - wget http://selenium-release.storage.googleapis.com/2.42/
     selenium-server-standalone-2.42.2.jar
  - composer global require "fxp/composer-asset-plugin:~1.1.1
     "
  - composer install

before_script:
  - export DISPLAY=:99.0
  - sh -e /etc/init.d/xvfb start
  - java -jar selenium-server-standalone-2.42.2.jar -port
     4444 &
  - sleep 5

script:
  - php vendor/bin/codecept run unit --coverage-xml --env
     travis

after_script:
  - wget https://scrutinizer-ci.com/ocular.phar
  - php ocular.phar code-coverage:upload --format=php-clover
     tests/_output/coverage.xml
```

4.1.4 Datoteka .scrutinizer.yml

V tej datoteki nastavimo znotraj definicije 'tools' sprejem zunanjih rezultatov testiranja. V našem primeru bomo analizirali rezultate testov, ki so se izvajali na Travis CI. Znotraj definicije 'checks' za programski jezik PHP aktiviramo računanje ocene kakovosti kode in preverjanje duplikatov. Znotraj definicije 'filter' določimo za kateri del kode naj se izvaja analiza.

```
tools:
    external_code_coverage: true
checks:
    php:
        code_rating: true
        duplication: true
filter:
    paths:
        - src/*
```

4.2 Uporaba okostja

Na primeru demo aplikacije Symfony, ki je opisana v poglavju 3.7, prikažemo vgradnjo pripravljenega okostja za avtomatsko testiranje. Najprej opišemo korake, ki so potrebni za vgradnjo, nato podamo tudi primere testov, ki jih je možno uporabiti.

4.2.1 Postopek vgradnje

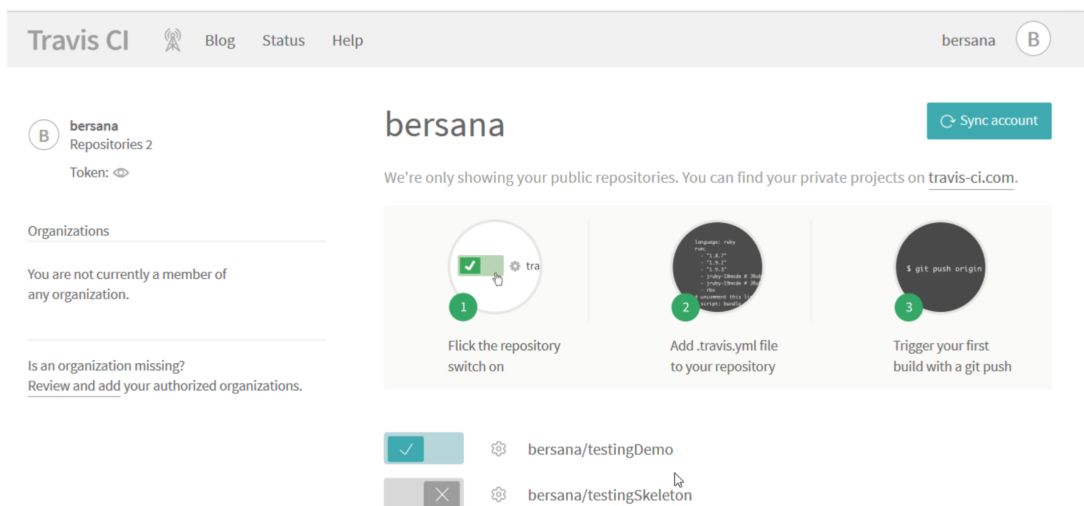
Spodaj so našteti predvideni koraki za vgradnjo okostja v aplikacijo, kjer bi radi omogočili avtomatsko zaganjanje testov in zvezno integracijo.

- Če še nimamo orodja Composer, ga namestimo.

- Namestimo Codeception:

  ```
  $ composer require codeception/codeception --dev
  ```

  S tem ukazom se v /bin direktoriju namestita Codeception in PHPUnit.



Slika 4.1: Aktivacija repozitorija na strani Travis CI.

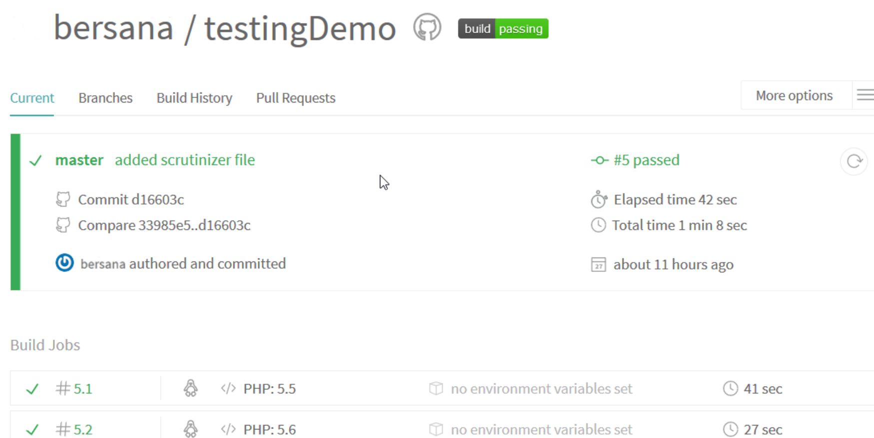
- Za Symfony projekt moramo paziti, da so v mapi /tests samo sprejemni testi, testi enot ter funkcionalni testi pa morajo biti znotraj mape src/AppBundle. Za pripravo primerne strukture direktorijev, uporabimo naslednje zaporedje ukazov, ki omogoča pisanje testov enot, funkcionalnih in sprejemnih testov:

```
$ php bin/codecept bootstrap --empty
$ php bin/codecept bootstrap --empty src/AppBundle --
   namespace AppBundle
$ php bin/codecept g:suite unit -c src/AppBundle
$ php bin/codecept g:suite functional -c src/AppBundle
$ php bin/codecept g:suite acceptance
```

  Navedeni ukazi poskrbijo za pravilno strukturo direktorijev v Symfony projektu.

- Dodati datoteko .travis.yml
- Dodati datoteko .scrutinizer.yml
- Aktivizirati repozitorij demo aplikacije na Travis-CI.com preko profilne

Slika 4.2: Dodajanje repozitorija na strani Scrutinizer-ci.



Slika 4.3: Izpis rezultatov izgradnje.

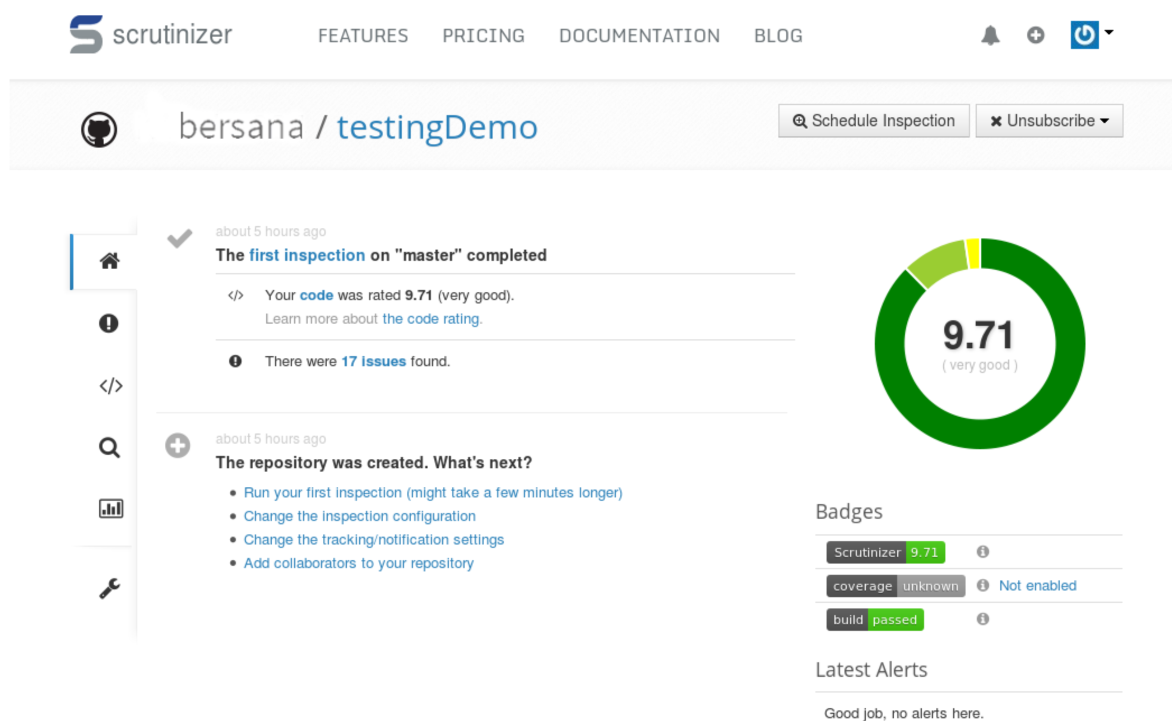
strani (glej sliko 4.1).

- Na profilni strani Scrutinizer-ci dodati repozitorij z aplikacijo (glej sliko 4.2).

4.2.2 Rezultati

Za našo aplikacijo je Travis CI vrnil rezultat, ki je podan na sliki 4.3. Na izpisu imamo zaporedno številko izgradnje, koliko časa je izgradnja trajala in koliko časa je minilo od zaključenega procesa. Navedena imamo tudi vsa testna okolja, za katera smo izvajali izgradnjo in teste. V našem primeru sta dva - okolje s PHP verzijo 5.5 in okolje s PHP verzijo 5.6. Ko je izgradnja uspešno zaključena, se izpisi obarvajo z zeleno barvo, sicer pa z rdečo.

V nastavitvah smo rezultate testov, ki so bili izvedeni na Travis CI, posredovali Scrutinizerju, ki je izračunal za nas oceno kakovosti. Rezultat analize je prikazan na sliki 4.4. To oceno kakovosti dobimo na podlagi analize pripravljenih testov. Pripravili smo 17 različnih testov. Analitična orodja so za vsak test preverila, ali dobimo napako, ali imamo podvojeno/neuporabljeno kodo in so ocenila razumljivost (angl. comprehensibility). Na podlagi teh



Slika 4.4: Scrutinizer, povzetek analize.

faktorjev so testi dobili vrednostne uteži, povprečna vrednost katerih predstavlja oceno kakovosti. Formule so nastavljene tako, da na koncu izračuna vedno dobimo vrednost med 0 in 10. Bližja ko je vrednost številki 10, višja je kakovost naše kode.

Uspelo nam je pripraviti 8 'popolnih' testov, kjer smo dobili najvišjo možno oceno, in ker 'resnost' odkritih pomanjkljivosti pri ostalih ni bila visoka, smo v skupnem seštevku dobili oceno kakovosti 9.71. Je pa Skrutinizer odkril dve datoteki s srednje 'resno' napako (angl. major severity). To sta DeleteUserCommand.php in Admin/BlogController.php, ki kličeta neobstoječe metode. S tem ko smo to odkrili, lahko preverilmo našo kodo in odpravimo napako. Pri primerih, kjer imamo podvojeno kodo, lahko preverimo, če jo res potrebujemo.

4.2.3 Primeri testov

Codeception temelji na orodju PHPUnit. Zato pri pisanju testov lahko uporabljamo bodisi osnovno PHPUnit implementacijo, pri kateri smo omejeni zgolj na teste enote, bodisi nadgrajeno verzijo, ki omogoča tudi pisanje integracijskih testov. Codeception lahko poganja teste, ki so bili napisani v PHPUnitu že pred njegovo namestitvijo.

V procesu pisanja diplomske naloge je bilo pripravljenih več različnih testov. Za ilustracijo uporabe ogrodja Codeception bomo nekaj primerov predstavili v nadaljevanju poglavja.

Osnovni test enote

Z naslednjim ukazom na enostaven način dobimo osnutek za test enote, ki deduje iz razreda PHPUnit_Framework_TestCase:

```
$ php codecept generate:phpunit unit Example
```

Tovrstni test smo uporabili za testiranje metode slugify() iz razreda Slugger:

```
<?php
namespace AppBundle\Utils;
class Slugger
{
    /**
     * @param string $string
     *
     * @return string
     */
    public function slugify($string)
    {
        return trim(preg_replace('/[^a-z0-9]+/', '-',
            strtolower(strip_tags($string))), '-');
    }
}
```

Pripravili smo razred SluggerTest (izvorna koda 4.1), ki vsebuje dve metodi. V metodi getSlugs() naštejemo testne primere. Za vsak testni primer

podamo niz, ki ga sprejme metoda slugify(), in niz, ki ga pričakujemo po izvajanju te funkcije.

```
<?php
namespace Tests\Utils;
use AppBundle\Utils\Slugger;
class SluggerTest extends \PHPUnit_Framework_TestCase
{
    /**
     * @dataProvider getSlugs
     */
    public function testSlugify($string, $slug)
    {
        $slugger = new Slugger();
        $result = $slugger->slugify($string);

        $this->assertEquals($slug, $result);
    }

    public function getSlugs()
    {
        yield ['Lorem Ipsum'     , 'lorem-ipsum'];
        yield ['  Lorem Ipsum  ' , 'lorem-ipsum'];
        yield [' lOrEm  iPsUm  ' , 'lorem-ipsum'];
        yield ['!Lorem Ipsum!'   , 'lorem-ipsum'];
        yield ['lorem-ipsum'     , 'lorem-ipsum'];
    }
}
```

Izvorna koda 4.1: Razred SluggerTest

Test enote

Z naslednjim ukazom se nam ustvari osnutek za test enote, ki deduje od razreda Codeception\TestCase\Test. Ta razred predstavlja izboljšano verzijo PHPUnita in omogoča več dodatnih možnosti:

```
$ php codecept generate:test unit Example
```

S testom, ki je podan v izvorni kodi 4.3, smo preizkusili delovanje metode hoHtml() razreda Markdown (izvorna koda 4.2).

```
<?php
namespace AppBundle\Utils;
class Markdown
{
    /**
     * @var \Parsedown
     */
    private $parser;
    /**
     * @var \HTMLPurifier
     */
    private $purifier;

    public function __construct()
    {
        $this->parser = new \Parsedown();

        $purifierConfig = \HTMLPurifier_Config::create([
            'Cache.DefinitionImpl' => null,
        ]);
        $this->purifier = new \HTMLPurifier($purifierConfig);
    }
    /**
     * @param string $text
     * @return string
     */
    public function toHtml($text)
    {
        $html = $this->parser->text($text);
        $safeHtml = $this->purifier->purify($html);

        return $safeHtml;
    }
}
```

Izvorna koda 4.2: Razred Markdown

```
<?php
namespace AppBundle\Utils;

class MarkdownTest extends \Codeception\TestCase\Test
{
    /**
     * @var Markdown
     */
    protected $markdown;

    protected function _before()
    {
        $this->markdown = new Markdown();
    }

    public function testBasicMarkdownParsing()
    {
        $this->assertEquals(
            "<p><strong>Hello</strong></p>",
            $this->markdown->toHtml("**Hello**")
        );
    }

}
```

Izvorna koda 4.3: Razred MarkdownTest

Funkcionalni test

S funkcionalnim testom smo preverili prijavo uporabnika z administratorskimi pravicami. Pri pisanju testa smo uporabili objekt \$I, ki predstavlja testerja in ima enostavne funkcije za testiranje akcij. Imena metod so dovolj slikovita, da iz testa hitro razberemo njegov namen. Izvirna koda 4.4 v našem primeru ponazarja namen prijaviti se na stran, zato se mora tester (objekt \$I) nahajati na strani 'prijava' in v prijavni obrazec vnesti svoje uporabniško ime in geslo.

```
    protected function loginAsAdmin(FunctionalTester $I)
{
    $I->amGoingTo('log in to site');
    $I->amOnLocalizedPage('/prijava');
    $I->submitForm('#main form', ['_username' => '
        anna_admin', '_password' => 'kitten']);
}
```

Izvorna koda 4.4: Primer funkcionalnega testa

Sprejemni test

Spodaj je prikazan primer sprejemnega testa, kjer tudi uporabimo objekt $I. Primer izvorne kode 4.5 prikazuje test, kjer opravimo več akcij: ko se tester nahaja na začetni strani, klikne na napis 'Browse aplication' in za stran, ki ima v naslovu niz 'blog', odpre element, ki prestavlja članek.

```
<?php
$I = new AcceptanceTester($scenario);
$I->wantTo('open blog page and see article there');
$I->amOnPage('/');
$I->click('Browse application');
$I->seeInCurrentUrl('blog');
$I->seeElement('article.post');
```

Izvorna koda 4.5: Primer sprejemnega testa

Poglavje 5

Zaključek

Testiranje programske opreme je pomemben del razvojnega procesa, pri katerem je potreben širši pogled. Žal so v praksi razvijalci pogosto pod časovnim pritiskom in opravijo le teste enot. Čeprav se morda zavedajo, da je treba uporabiti tudi druge načine testiranja, jih od tega odvrne poglabljanje v dokumentacijo in študiranje konfiguracijskih datotek.

Z razvojem oblačnih storitev so se razvile nove prakse razvoja programske opreme in so se pojavile tudi nove možnosti za izvajanje in avtomatizacijo testiranja. Tako je na primer pri uporabi načina zvezne integracije smotrno opraviti tudi testiranje in analizo programske kode. Postavitev zvezne integracije je zahteven proces. Na srečo so dandanes že na voljo oblačni ponudniki te storitve.

V sklopu diplomske naloge smo ustvarili okostje, ki poveže orodje za testiranje programske opreme Codeception, storitev za zvezno integracijo ponudnika Travis CI, uporabo orodij za analizo pri ponudniku Scrutinizer ter PHP projekt na spletni storitvi GitHub.

Uporabljena ogrodja so precej kompleksna, saj podpirajo več programskih jezikov, orodij, dodatnih modulov itn., kar poveča obseg dokumentacije in jo včasih naredi nepregledno. S tako težavo smo se soočili pri orodju Scrutinizer, ki ima sicer na prvi pogled lepo organizirano dokumentacijo, vendar na nekatera vprašanja glede nastavitev nismo dobili odgovorov. Travis CI

je glede tega boljši, vendar tudi pri njem študij dokumentacije vzame veliko časa zaradi velikega števila podprtih tehnologij in ogrodij. Največji izziv pa je bil ustvariti pravilno povezamo med orodjem Codeception in Travis CI. Testno okolje na Travis CI smo preko naših nastavitev spremenili tako, da je za testiranje uporabilo Codeception. Upamo, da bo okostje, ki je nastalo v okviru te diplomske naloge, olajšalo uporabo omenjenih spletnih storitev in jih je naredilo dostopne tudi programerjem, ki nimajo časa, da bi se poglobili v nastavljanje uporabljenih orodij.

Trenutno okostje bi lahko še dodelali tako, da bi postopek uporabe, opisan v tej nalogi, zapisali v obliki programa v enem od skriptnih jezikov. S tem bi še bolj olajšali delo pri postavljanju enotnega testnega okolja.

Literatura

[1] What are Different Goals of Software? [Online]. Dosegljivo: http://testingbasicinterviewquestions.blogspot.si/2012/03/what-are-different-goals-of-software.html. [Dostopano 20. 7. 2016]

[2] A. V. Katherine, K. Alagarsamy "Conventional Software Testing Vs. Cloud Testing", *International Journal Of Scientific & Engineering Research*, št. 9, zv. 3, str. 1–5, 2012.

[3] P. Ammann, J. Offutt. *Introduction to Software Testing*. New-York: Cambrige University, 2008.

[4] Tieto Test Tools as a Service. [Online]. Dosegljivo: http://www-05.ibm.com/lt/software_day_2010/pdf/Tieto_test_tools_as_a_service.pdf. [Dostopano 20. 7. 2016]

[5] J. Gao, X. Bai, W. T. Tsai "Cloud Testing - Issues, Challenges, Needs and Practice", *Software Engineering: An International Journal*, št. 1, zv. 1, str. 9–23, 2011.

[6] Testiranje z metodo bele škatle. [Online]. Dosegljivo: https://en.wikipedia.org/wiki/White-box_testing. [Dostopano 11. 6. 2016]

[7] Testiranje z metodo črne škatle. [Online]. Dosegljivo: https://en.wikipedia.org/wiki/Black-box_testing. [Dostopano 11. 6. 2016]

[8] Software Testing: quick guide. [Online]. Dosegljivo: http://www.tutorialspoint.com/sowtware_testing_quick_guide.htm. [Dostopano 20. 5. 2016]

[9] Software quality. [Online]. Dosegljivo: https://en.wikipedia.org/wiki/Software_quality. [Dostopano 18. 7. 2016]

[10] Git. [Online]. Dosegljivo: https://en.wikipedia.org/wiki/Git_(software). [Dostopano 2. 6. 2016]

[11] GitHub For Beginners: Don't Get Scared, Get Started. [Online]. Dosegljivo: http://readwrite.com/2013/09/30/understanding-github-a-journey-for-beginners-part-1/. [Dostopano 2. 6. 2016]

[12] Understanding the GitHub Flow. [Online]. Dosegljivo: https://guides.github.com/introduction/flow/. [Dostopano 12. 7. 2016]

[13] Git Documentation. [Online]. Dosegljivo: https://git-scm.com/doc. [Dostopano 10. 6. 2016]

[14] Composer Documentation. [Online]. Dosegljivo: https://getcomposer.org/doc/. [Dostopano 14. 6. 2016]

[15] PHPUnit. [Online]. Dosegljivo: http://phpunit.de/. [Dostopano 15. 6. 2016]

[16] Symfony demo aplikacija. [Online]. Dosegljivo: http://symfony.com/blog/introducing-the-symfony-demo-application. [Dostopano 20. 7. 2016]

[17] Symfony dokumentacija. [Online]. Dosegljivo: http://symfony.com/doc/current/index.html. [Dostopano 22. 6. 2016]

[18] Travis CI. [Online]. Dosegljivo: https://en.wikipedia.org/wiki/Travis_CI. [Dostopano 18. 6. 2016]

[19] Meet Travis CI: Open Source Continuous Integration. [Online]. Dosegljivo:
https://www.infoq.com/news/2013/02/travis-ci. [Dostopano 18. 6. 2016]

[20] Travis CI dokumentacija. [Online]. Dosegljivo:
https://docs.travis-ci.com/user/languages/php/. [Dostopano 17. 6. 2016]

[21] P. M. Duvall, S. Matyas, A. Glover. "Continuous Integration: Improving Software Quality and Reducing Risk", 2007.

[22] S. Bergmann. "PHPUnit Pocket Guide", O'Really, 2005.

[23] Scrutinizer-ci dokumentacija. [Online]. Dosegljivo:
https://scrutinizer-ci.com/docs/. [Dostopano 21. 6. 2016]