Univerza v Ljubljani
Fakulteta za računalništvo in informatiko

Matic Volk

Avtomatsko testiranje spletnih strani v okolju Laravel

DIPLOMSKO DELO

UNIVERZITETNI ŠTUDIJSKI PROGRAM PRVE STOPNJE
RAČUNALNIŠTVO IN INFORMATIKA

Mentor: izr. prof. dr. Viljan Mahnič

Ljubljana 2014

Fakulteta za računalništvo in informatiko izdaja naslednjo nalogo:

Tematika naloge:

Proučite značilnosti agilnega testiranja programske opreme ter analizirajte prednosti in slabosti avtomatske priprave testov. Nato predstavite možnosti, ki jih za avtomatizacijo testiranja nudi okolje Laravel. Postopke testiranja v tem okolju prikažite na primeru testiranja spletne strani.

Izjava o avtorstvu diplomskega dela

Spodaj podpisani Matic Volk, z vpisno številko **63100293**, sem avtor diplomskega dela z naslovom:

Avtomatsko testiranje spletnih strani v okolju Laravel

S svojim podpisom zagotavljam, da:

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

V Ljubljani, dne 16. septembra 2014 Podpis avtorja:

Zahvaljujem se mentorju izr. prof. dr. Viljanu Mahniču za pomoč in vodenje pri izdelovanju diplomskega dela ter družini za podporo pri študiju in diplomskem delu. Prav tako se zahvaljujem Katarini Golob za jezikovno pregledan izdelek ter prijateljem in ostalim, ki so mi stali ob strani in mi nudili podporo. Posebna zahvala gre očetu, ker mi je bil v času študija v ogromno pomoč in oporo.

Kazalo

Povzetek

Abstract

1 Uvod **1**

2 Agilno testiranje **5**

2.1 Prednosti avtomatizacije testov 6

2.2 Ovire avtomatizacije 7

2.3 Sprotna integracija 9

2.4 Testno voden razvoj 10

2.4.1 Testiranje enot 11

2.4.2 Integracijski testi 12

2.4.3 Sprejemni testi 12

2.4.4 Mockery 12

2.4.5 Factory 12

3 Okolje **13**

3.1 Laravel 13

3.2 PHPUnit 16

3.3 Codeception 18

4 Testiranje na primeru spletne strani **21**

4.1 Struktura in funkcionalnosti spletne strani 22
4.2 Testiranje pogleda 24
4.2.1 Sprejemni testi za prijavni obrazec 24
4.3 Testiranje kontrolerja 28
4.3.1 Priprava za testiranje kontrolerja 28
4.3.2 Testi metod kontrolerjev 29
4.4 Testiranje modela 31
4.4.1 Priprava za testiranje modela 31
4.4.2 Testiranje validacije modela 33
4.4.3 Testiranje relacij modelov 34
4.4.4 Testiranje Get/Set metod 35
4.4.5 Testiranje lastnih metod 36
4.5 Testiranje podatkovne baze 37
4.6 Testiranje pri posodobitvi kode na odlagališče (repositorij) . . . 38
4.6.1 Uporaba servisa Travis 38

5 Zaključek **41**

Povzetek

Porast števila spletnih aplikacij je vplival na uporabo naprednih metod za njihov razvoj. V ospredju so agilne metode, ki vpeljujejo razvijanje, vključno s testiranjem. Testno voden način razvoja spletnih strani zajema pisanje testov, preden se začne implementacija posamezne funkcionalnosti spletne strani. V diplomski nalogi so opisani tipi testov, ki so prikazani tudi na primeru razvoja spletne strani. Testiranje poteka po komponentah glede na vlogo, ki je komponenti namenjena. Predstavljeni so testi pogleda, modela in kontrolerja, razdeljeni na posamezne enote tako, da omogočajo natančno in učinkovito testiranje.

Ključne besede: Spletna stran, avtomatski testi, Laravel, model, pogled, kontroler, testi enot.

Abstract

Increase of web application development has affected usage of advanced development methodologies. In front of development are agile methods, which invoke testing inside software programs. Test driven development practise of website development covers writting tests before implementating functionalities of web page. In the thesis are descibed different types of tests, which are shown in practical use. Components are tested dependently on which component is currently tested. Presented tests are view, model and controller tests divided into individual units that they can be tested accurately and efficiently.

Keywords: Website, automatic tests, Laravel, model, view, controller, unit tests.

Poglavje 1

Uvod

Spletna stran je dokument z nadbesedilom (hypertext), ki omogoča prikaz različnih vsebin. Prikaz vsebine dokumenta omogočajo brskalniki. Spletne strani lahko prikazujejo:

- Besedila,
- slike,
- povezave,
- zvok,
- video.

Prvo spletno stran je leta 1990 napisal Tim Berners-Lee. Leta 1994 je bilo število spletnih strani še relativno majhno, po letu 1995 pa se je začela bitka brskalnikov in s tem tudi velika porast spletnih strani. Do leta 2001 se je število strani povečalo na $550 * 10^9$. Leta 2009 je Google Search našel 10^{12} enoličnih spletnih naslovov [12]. Pred razvojem protokolov HTML in HTTP so bile spletne strani namenjene le brskanju po dokumentih in prenašanju dokumentov s strežnika. Spletne strani današnjega časa obsegajo širok nabor funkcionalnosti. Predvsem se razširja razvoj spletnih aplikacij, ki so dostopne

na katerikoli napravi z internetnim dostopom. Zaradi velikega porasta razvoja so se tudi načini razvijanja aplikacij začeli spreminjati. Velike spremembe so nastale pri testiranju razvojnega produkta. Iz slapovnega modela, ki je vključeval fazo testiranja ob koncu razvoja produkta, se je razvoj preusmeril v iterativni model. Poznanih je več vrst iterativnih modelov. Njihova skupna značilnost je postopen razvoj, kar pomeni, da za razliko od zaporednega pristopa ne končujemo faz v celoti, ampak le delno, cel cikel pa ponavljamo, dokler aplikacija ni zaključena [6]. Z namenom povečanja produktivnosti so se vpeljale agilne metode razvoja programske opreme. Agilne metode delujejo na principu zaključene celote posameznega cikla. Cikli so časovno omejeni in vsebujejo naloge, ki se v primeru nedokončanosti prenesejo v naslednji cikel. Znotraj vsakega cikla je potrebno rezervirati tudi čas, namenjen za testiranje napisane kode. Za ročno testiranje kode je potrebno imeti testerje, ki se dobro spoznajo na zahteve stranke. Testerji morajo slediti sistematičnemu pristopu testiranja kode, da zagotovijo maksimalno odkrivanje napak. Osnovni koraki za ročno testiranje so:

- Izbrati testni načrt,
- napisati natančne teste,
- teste dodeliti testerjem,
- napisati poročilo o opravljenih testih.

Testni načrt definira metodologijo, po kateri se bo moral tester orientirati. Natančno napisani testi predstavljajo korake, ki opisujejo postopek testiranja ter pričakovane rezultate. Napisane teste se dodeli testerjem, ki jih izvršijo ter na koncu napišejo rezultate testiranja [9].

Ročno testiranje ima prednosti kot tudi pomanjkljivosti. Kratkoročno imajo ročni testi nižje stroške in so zelo fleksibilni, pomankljivosti pa se nahajajo v težavnosti ročnega testiranja ter v ponovni uporabi ročnih testov.

Določene tipe testov je zelo težko ročno uporabljati. Teste je potrebno ob spremembi kode ponovno zgraditi in preveriti pravilnost delovanja.

Agilne metode stremijo k avtomatizaciji testov. Agilni razvoj ne obravnava testiranja kot ločeno fazo razvoja, ampak kot integriran del pisanja testov skupaj z razvijanjem produkta. V naslednjem poglavju bodo predstavljene prednosti in vrste avtomatiziranih testov.

Poglavje 2

Agilno testiranje

Agilno testiranje je način programskega testiranja, ki sledi principom agilnega razvoja. Agilno testiranje vključuje vse člane razvojne ekipe, vključno s testerji, ki s svojim strokovnim znanjem zagotavljajo, da stranka prejema želene rezultate v določenih časovnih intervalih. Testerji morajo sodelovati s stranko in razvojno skupino tako, da zahteve, ki jih je podala stranka, postanejo vodič za kodiranje. Testiranje in kodiranje se izvajata inkrementalno in iterativno, dokler produkt ne doseže stopnje pripravljenosti za izid [1].

2.1 Prednosti avtomatizacije testov

Agilne metode testiranja stremijo k avtomatizaciji testov. Dobra avtomatizacija omogoča skupini programerjev hiter in kvaliteten razvoj aplikacij. Razlogi za avtomatizacijo testov [13]:

- Ročno testiranje traja dlje,
- ročno procesiranje vsebuje napake,
- avtomatizacija testov izboljša implementacijo,
- avtomatski regresijski testi dajejo občutek varnosti,
- avtomatski testi vračajo povratno informacijo,
- testi so odlična dokumentacija.

Osnovni razlog za uporabo avtomatskih testov je čas, ki ga skupina porabi za testiranje, saj s tem ko aplikacija narašča, narašča tudi čas testiranja. Skupine programerjev, ki uporabljajo agilne metode razvoja, naredijo veliko sprememb v kodi. Spremembe kode lahko nastajajo tudi dnevno, zato pri ročnem testiranju pride do problema, ko testerji ne morejo dohajati spremembam v kodi. Ročno testiranje je časovno potratno pri uporabniških vmesnikih in funkcionalnostih, ki potrebujejo podatke. Predvsem pri testiranju z določenimi podatki pride do velike verjetnosti napak zaradi obsega različnih primerov uporabe podatkov.

Testi se stalno ponavljajo v iteracijah, zato se pri ročnem testiranju hitro zgodi, da pride do napak ali spuščanja korakov pri testih. Avtomatizacija testov poskrbi, da se tovrstne napake ne dogajajo, ker se vsako testiranje izvaja na način, ki je bil določen ob implementaciji.

S pisanjem testov pred implementacijsko kodo se programer lahko bolje osredotoči na pisanje dejanske funkcionalnosti aplikacije, kar pomeni boljšo kodo ter zagotavljanje pravilnega načrtovanja aplikacije.

Pri popravljanju ali dodajanju nove kode v že obstoječo, pride velikokrat do napak v delih kode, ki je že bila napisana. Avtomatski testi poskrbijo, da se v takem primeru hitro odkrije novo nastale napake in programerjem omogoči njihovo odpravo.

Stalno preverjanje delovanja aplikacije s pomočjo testov, omogoča hitro povratno informacijo o stanju napredka. Hitro odkrite napake je programerjem lažje odpraviti kot pa tiste, ki se odkrijejo kasneje. Pomembno je, da za vsak del nove kode programer tudi preveri, če aplikacija še vedno opravi teste, saj hitro zaznane napake stanejo manj.

Napisani testi predstavljajo dokumentacijo, kako aplikacija deluje. Statično dokumentacijo je težko ohranjati posodobljeno. Potrebno je poskrbeti, da vsi testi opravijo pozitivno in tako vedno opisujejo zadnje stanje delovanja aplikacije.

Pomembna komponenta povratne informacije avtomatskih testov je način, kako so napake popravljene. Skupine, ki se zanašajo na ročne teste, odkrijejo napake kasneje in jih težje popravljajo. Pri sprotnem testiranju lahko programerji z odkritjem napake hitro ukrepajo, in če je potrebno, tudi popravijo začrtan načrt aplikacije, kar je pri ročnih testih težje.

2.2 Ovire avtomatizacije

Tako kot ročno testiranje, ima tudi avtomatizacija svoje slabosti. Orodja za testiranje, ki zajemajo celotno, oziroma pokrijejo vsaj pomembnejši del testiranja, so lahko draga investicija. Avtomatski testi so bistveno hitrejši od ročnih, vendar je potreben čas za implementacijo teh testov. So stvari, ki jih avtomatski testi ne morejo testirati, to so na primer vizualne lastnosti spletne strani, ki jih je potrebno testirati ročno, kar pomeni, da orodja za avtomatsko testiranje ne zmorejo testirati vsega.

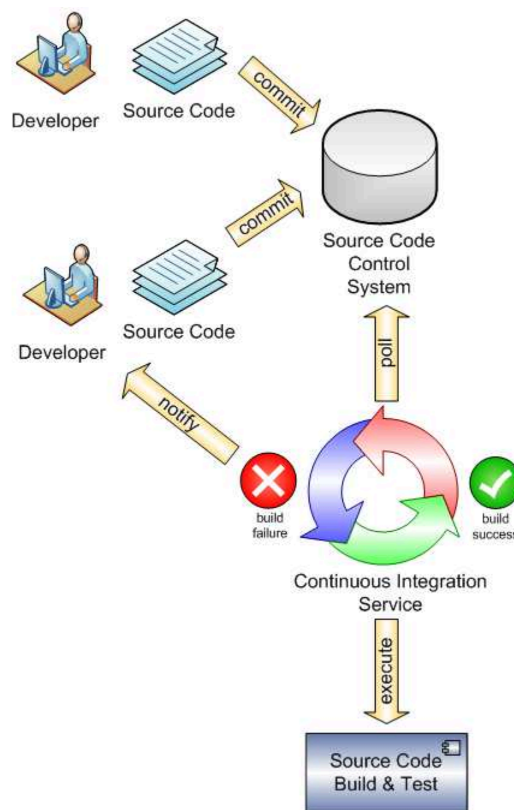
Agilni pristopi so izziv, ki ga mora sprejeti celotna razvijalska skupina. Programerji, ki se šele privajajo na avtomatiziran način testiranja, so navajeni

oddajati kodo le zato, da končajo v zadanem roku, čeprav je koda polna napak. Pri avtomatiziranem pristopu gre za premišljeno pisanje kode tako, da koda ustreza pogojem funkcionalnosti ter vsebuje čim manj napak.

V praksi se uporablja različne agilne prakse razvoja spletnih aplikacij. Vsaka praksa ima svoj način razvijanja aplikacij, zato se je potrebno, odvisno od zahtev ter potreb programerjev, odločiti za eno od praks. Različne prakse imajo različno začrtan potek dela. Tipične prakse ne zahtevajo, kako pogosto je potrebno novo napisano kodo vključevati v celoto. Programerjem povzroča tak način težave zaradi popravljanja napak in konfliktov ob posodobitvi kode. Enostavno združevanje kode postane težko in dolgo opravilo, medtem ko pa pri praksi, kot je sprotna integracija (continous integration), cikli posodabljanja kode kratki in obvladljivejši. Iz navedenih razlogov sem za namene razvoja spletne strani s pomočjo avtomatiziranih testov uporabil prakso sprotne integracije.

2.3 Sprotna integracija

Continuous integration je praksa razvoja programske opreme, ki usmerja programerje k stalnemu vstavljanju nove kode, v že obstoječo [2]. Namenjen je predvsem programerjem, ki delajo v skupinah na določenem projektu.



Slika 2.1: Continuous integration diagram

Koristi tega načina programiranja pripomorejo k hitrejšemu razvoju aplikacij, kot tudi enostavnemu dodajanju novih lastnosti k že delujoči aplikaciji. S pomočjo avtomatiziranih testov, se pospešuje tudi odpravljanje napak znotraj programa. Eden od procesov za razvijanje programske opreme z avtomatiziranimi testi je testno voden razvoj (test driven development (TDD)).

2.4 Testno voden razvoj

TDD je način razvijanja aplikacij, ki se osredotoča na razbijanje nalog na manjše enote, da je mogoče vsako enoto posamično testirati [15]. Testiranje takih enot poteka ponavljajoče ter s tem zagotavlja, da enota deluje pravilno po vsaki spremembi. Test driven development način zagotavlja, da bo vsaka enota delovala tako kot ji je namenjeno. Testiranje spletne aplikacije je lahko zelo zamudno, če se uporablja brskalnik in velikokrat se zgodi, da oseba, ki testira aplikacijo, spregleda kakšno napako v programu. Testi enot omogočijo hitrejše ter natančnejše testiranje aplikacije. Nadaljnje spremembe v programu lahko povzročijo napake v ostalih delih programa. Pri ročnem testiranju je težko dobiti napake, ki so nastale pri spremembi. Pri tem pridejo v pomoč že vnaprej spisani testi, ki točno pokažejo, kateri testi niso bili opravljeni in tako enostavno zaznajo napake v aplikaciji. Cikel razvoja aplikacije [15]:

- Priprava testa,
- programiranje kode, ki opravi test,
- optimizacija zapisane kode.

Najprej mora programer razmisliti, kakšno funkcionalnost bo napisan test preverjal. Test napiše tako, da pripravi objekte in podatke, ki so potrebni za test in zapiše, kaj pričakuje od testa. Glede na napisan test je potrebno napisati kodo, ki bo ob izvršitvi test opravila. Napisano kodo se nato v iteracijah optimizira. Napotki za učinkovitejše pisanje testov so:

- Testna koda mora biti kratka, natančna in razumljiva,
- testi se lahko izvršujejo v poljubnem vrstnem redu,
- posamezen test mora delovati neodvisno od rezultata ostalih.

Testiranje je razdeljeno na več vrst, kar pripomore k lažjemu razumevanju ter boljši organizaciji testiranja. Testi za testiranje spletne aplikacije so razdeljeni na tri osnovne vrste [15]:

- Testi enot,
- integracijski testi,
- sprejemni testi.

2.4.1 Testiranje enot

Enota je najmanjši del znotraj aplikacije, ki ga je mogoče testirati. Pri testiranju enot je potrebno paziti, da se posamezno enoto testira neodvisno od ostalih enot. Nekatere stvari je težko testirati neodvisno, ampak tudi to je mogoče, s pomočjo objektov, ki simulirajo delovanje pravih objektov. Vsak test enote mora obsegati dva testna primera. Enota lahko deluje pravilno ali napačno, zato je potrebno testirati obnašanje v obeh primerih in poskrbeti, da se enota v obeh primerih pravilno odzove. Tako kot drugi testi, imajo tudi testi enot meje sposobnosti. Testi enot so omejeni na testiranje funkcionalnosti ene enote izolirano od ostalih enot. Še vedno se lahko pojavijo napake pri funkcionalnostih, ki za delovanje potrebujejo več enot. V ta namen je potrebno napisati integracijske teste, ki preverjajo pravilno delovanje več med seboj odvisnih enot. Pomembno je hraniti spremembe v kodi, ker se lahko zgodi, da je že opravljen test ponovno zavrnjen zaradi sprememb. V takem primeru se preverijo spremembe, ki so nastale za zadnjim še opravljenim testom nad enoto, nato pa se primerno popravi test ali kodo, ki je bila spremenjena.

2.4.2 Integracijski testi

Če testiranje enot razgradi teste na posamezne enote in testira posamezno enoto neodvisno od drugih, se pri integracijskih testih enote združijo in opravijo testi nad združeno celoto. Namen integracijskih testov je preveriti funkcionalnost, zmogljivost ter zanesljivost zahtev, ki jih mora aplikacija izpolnjevati.

2.4.3 Sprejemni testi

Testi enot so potrebni, toda ne zagotavljajo delovanja aplikacije kot celote. Testi enot zagotavljajo, da posamezna enota aplikacije deluje, medtem ko sprejemni testi preverijo ali so zahteve stranke izpolnjene. Sprejemne teste pišejo programerji, ki ne poznajo notranjega delovanja aplikacije. Teste lahko pišejo programerji stranke ali tudi stranka sama. Sprejemni testi so končna dokumentacija aplikacije, ki jih je v začetnih fazah razvoja težko pisati, ampak lahko pripomorejo k boljšim odločitvam na projektu kot celoti.

2.4.4 Mockery

Mockery se uporablja pri testiranju enot za generiranje objektov, ki simulirajo delovanje objektov, ki so odvisni od testirane enote. Kljub temu, da je testirana enota odvisna od drugih objektov, se z uporabo Mockery ohrani izolirano testiranje enote.

2.4.5 Factory

Namen Factory je generiranje objektov, ki vsebujejo podatke za testiranje. Generirani objekti so napolnjeni z naključno vsebino. Factory omogoča dodeljevanje vrednosti po meri. Poleg navedbe, kateri objekt naj generira, se lahko navede tudi ime atributa in vrednost, ki jo priredi.

Poglavje 3

Okolje

Laravel okolje je eno od okolij za izdelovanje spletnih strani. Podpira tudi testiranje s pomočjo PHPUnit testnega okolja, ki omogoča uporabo testov enot. Za naprednejše teste, kot so funkcijski ali sprejemni testi, se uporablja okolje Codeception.

3.1 Laravel

Laravel je prostodostopno okolje, ki zagotavlja močna orodja za izdelovanje spletnih aplikacij. Okolje stremi k razbremenitvi programerjev pri programiranju najbolj pogostih nalog, s katerimi se srečamo pri razvoju spletnih aplikacij. Primeri pogostih nalog [8]:

- Avtentikacija,
- seje,
- usmerjanje (routing),
- predpomnjenje (caching).

Avtentikacijo sproži strežnik, njen namen pa je preveriti, ali je uporabnik, ki se želi vpisati v sistem, dejansko tisti uporabnik, za katerega se predstavlja.

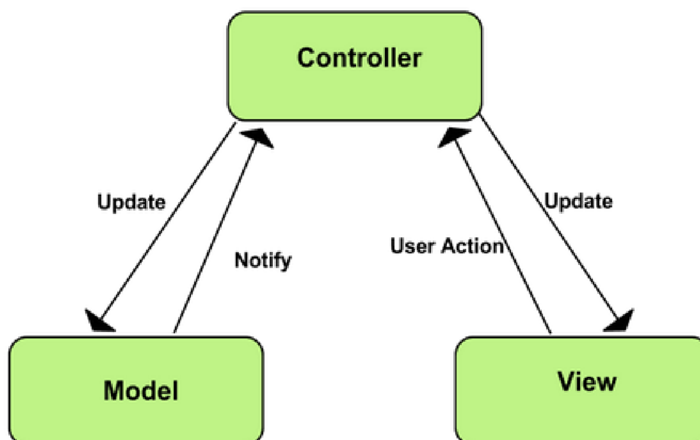
Spletna aplikacija vsebuje seje za hranjenje informacij, ki so potrebne za pravilno delovanje strani. Seja se vzpostavi v določenem času in hrani informacije, dokler se seja ne prekine. Usmerjanje se ukvarja z manipulacijo URL naslovov, ki jih uporablja celoten projekt. Gre za usmerjanje med posameznimi povezavami strani. Predpomnjenje skrbi za hranjenje podatkov, ki so pogosto v uporabi in s tem skrajša čas dostopa do njih. Pri razbremenitvi programerjev je potrebno poudariti pomembnost razmerja med svobodo razvoja aplikacij ter vgrajenimi funkcionalnostmi za razvoj. Vgrajene funkcionalnosti olajšajo delo, ampak odvzamejo svobodo realizacije aplikacije in obratno. Laravel okolje za svoje delovanje potrebuje [8]:

- PHP 5.4 ali novejšo različico,
- Composer,
- MCrypt PHP Extension.

Composer je PHP manager, ki upravlja odvisnosti med različnimi knjižnicami in skrbi za instalacijo potrebnih paketov [4]. MyCrypt je PHP knjižnica z naborom funkcij, ki jih Laravel potrebuje [8]. Ob inicializaciji projekta je poleg priloženo tudi orodje za testiranje PHPUnit, ki vsebuje metode, s katerimi je mogoče testirati kodo, napisano v PHP jeziku.

Arhitektura Laravel aplikacije temelji na osnovi modela Model-View-Controller (MVC). Model MVC je način za izgradnjo uporabniških vmesnikov. Sestavljen je iz treh komponent:

- Pogled,
- kontroler,
- model.



Slika 3.1: Model MVC

Vsaka od komponent ima določeno nalogo. Pogled je zadolžen za prikaz podatkov, ki jih uporabnik zahteva preko uporabniškega vmesnika in predstavlja vizualizacijo strukture modela. Kontroler skrbi za interakcijo med pogledom in modelom. Kontrolna enota obdeluje zahtevke, ki jih uporabnik sproži v pogledu ter vrača pogledu odgovor, glede na poslan zahtevek in po potrebi dostopa tudi do modela, kjer lahko pridobiva zahtevane podatke. Modeli so razredi, ki skrbijo za manipulacijo s podatkovno bazo, vsak model je tudi preslikava ene tabele znotraj podatkovne baze.

3.2 PHPUnit

PHPUnit je testno okolje, ki je že vključeno znotraj Laravel okolja in omogoča testiranje enot. Testi se nahajajo znotraj mape "app/tests". Vsi testi znotraj te mape se izvedejo ob ukazu phpunit. Znotraj datoteke phpunit.xml je potrebno nastaviti nekaj parametrov. Pomembno je nastaviti pot do tests mape, ki jo je mogoče nastaviti znotraj testsuite značke.

```
<?xml version="1.0" encoding="UTF-8"?>
<phpunit>
    <testsuites>
        <testsuite name="Application Test Suite">
            <directory>./app/tests/</directory>
        </testsuite>
    </testsuites>
</phpunit>
```

Razred s testnimi funkcijami vsebuje razširitev `PHPUnit_Framework_TestCase`:

```
class TestCase extends PHPUnit_Framework_TestCase {
        public function createApplication()
        {
                $unitTesting = true;

                $testEnvironment = 'testing';

                return require __DIR__.'/../../bootstrap/
                    start.php';
        }
}
```

Razširitev omogoča uporabo funkcij za testiranje. Na novo ustvarjeni testni razredi se smatrajo za teste, če vsebujejo razširitev razreda TestCase. Ime testnega razreda se mora zaključiti z besedo Test.php, saj ga drugače phpunit ne upošteva. Ravno tako se mora pri lastnih testnih funkcijah začeti ime z besedo test.

Osnovne testne funkcije so sledeče:

- AssertEquals preveri ujemanje pričakovanega in dejanskega rezultata funkcije,
- assertTrue preveri, ali dejanska funkcija vrne true,
- assertFalse preveri, ali dejanska funkcija vrne false,
- assertContains preveri, ali dejanska funkcija vsebuje podano vsebino,
- assertInstanceOf preveri, ali je objekt pravilnega tipa.

Preprost primer testa (Hello World.):

```
<?php
class HelloWorldTest extends TestCase {
        public function testHelloWorld()
        {
                $greeting = 'Hello, World.';
                $this->assertEquals('Hello, World.',
                    $greeting);
        }
}
```

Test preveri, ali se vsebina spremenljivke in pričakovana vrednost ujemata. Večina assert funkcij vsebuje tri parametre. Prvi parameter vsebuje vrednost, ki jo programer pričakuje, drugi parameter vsebuje vrednost, ki jo poda testiran del kode, tretji opcijski parameter vsebuje sporočilo, ki je posredovano programerju po izvedenem testu.

3.3 Codeception

Codeception je okolje namenjeno testiranju spletnih aplikacij. V primerjavi z PHPUnit okoljem je veliko lažji za razumevanje ter vsebuje več dodatkov. Testi so lahko napisani tudi z PHPUnit sintakso. Omogoča izvajanje [3]:

- Testov enot,
- testov funkcionalnosti,
- sprejemnih testov.

Instalacija Codeception paketa je mogoča preko Composer managerja. V composer.json datoteko je potrebno vključiti

```
"require-dev": {
        "codeception/codeception": "1.6.1.1"
}
```

in nato znotraj konzole pognati ukaz composer update. Osnovne zahteve so minimalne, in sicer Laravel okolje verzije vsaj 4.1. Po instalaciji Codeception je zaradi lažje uporabe dobro izvesti še ukaz codeception bootstrap app. Ukaz generira mape ter datoteke, ki so v pomoč programerju pri testiranju znotraj tests mape. Z ukazom codecept run se poženejo vsi testi znotraj mape app/tests. Codeception test vsebuje objekt $I, ki predstavlja programerja. Objekt vsebuje preproste funkcije za testiranje posameznih akcij programerja.

V nadaljevanju navajam nekaj preprostih funkcij za uporabo:

- $I->wantTo je opis testa,
- $I->see preveri, ali je navedena vsebina vidna,
- $I->amOnPage poda stran, ki jo je potrebno testirati,
- $I->seeCurrentUrlEquals preveri ujemanje url-a,
- $I->fillField izpolni vnosno polje z navedeno vsebino,
- $I->click izvede klik na naveden objekt.

Preprost primer sprejemnega testa:

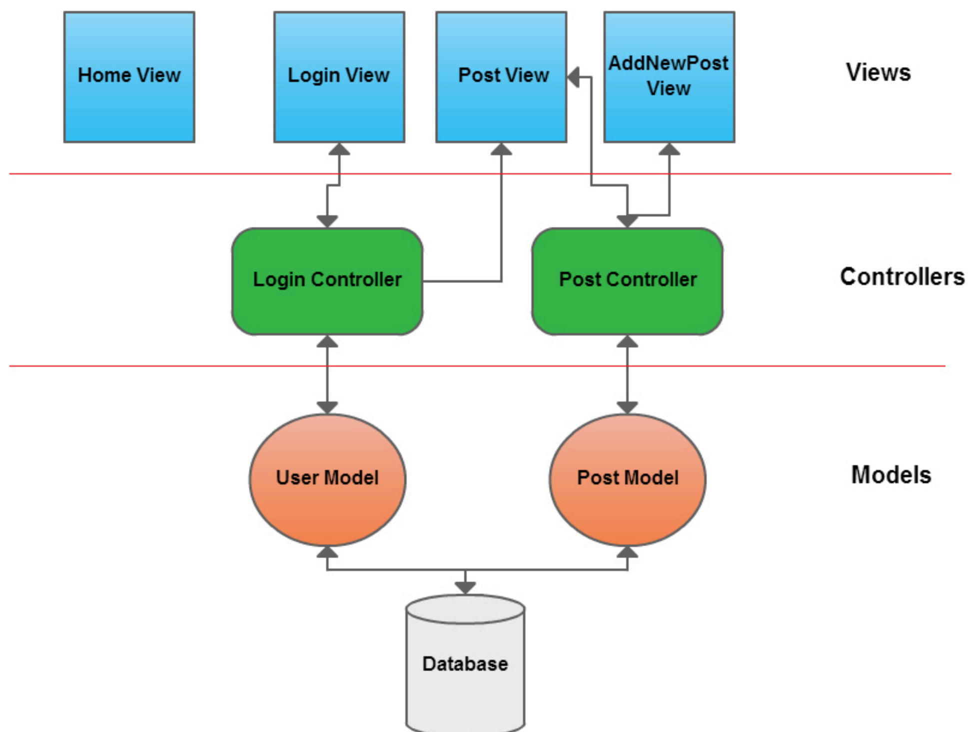
```
<?php
$I = new AcceptanceTester($scenario);
$I->wantTo('see login page');
$I->amOnPage('/Login');
$I->seeCurrentUrlEquals('/Login');
$I->see('Login');s
```

Objekt $I se nahaja na prijavni strani. Preveri, ali se url ujema s podanim ter ali se na strani nahaja Login napis.

Poglavje 4

Testiranje na primeru spletne strani

V namen praktičnega primera testiranja spletne strani je nastala spletna stran z imenom Laravel Testing. Spletna stran je preprosta, a zajema vse osnovne komponente, ki jih vsebuje vsaka spletna stran. Napisana je v okolju Laravel s pomočjo MVC modela.



Slika 4.1: Struktura spletne strani

4.1 Struktura in funkcionalnosti spletne strani

Spletna stran ima 4 poglede:

- Domov,
- Prijava,
- Objave,
- Dodaj novo objavo.

Domov je začetna stran spletnega portala, na kateri je pozdrav uporabniku, logo strani ter povezava prijava, ki uporabnika preusmeri na stran Prijava. Na strani Prijava je obrazec za prijavo uporabnika, ki zahteva vnos uporabniškega

imena ter gesla. Stran Objave vsebuje seznam vseh objav, ki jih je uporabnik kdajkoli objavil. Na strani Dodaj novo objavo je obrazec za vnos naslova ter vsebine objave, ki jih želi uporabnik dodati. Za preusmerjanje uporabnika med posameznimi stranmi skrbita dva kontrolerja.

Prijavni kontroler skrbi za preusmeritev uporabnika na prijavno ali na objavno stran v primeru pravilne prijave. Ob izpolnitvi uporabniškega imena in gesla se ob kliku na gumb prijava sproži prijavni kontroler, ki poskrbi za zahtevo uporabnika ter posreduje odgovor. V primeru napačnega uporabniškega imena ali gesla kontroler vrne uporabnika na prijavno stran z obvestilom o napaki. Če kontroler uporabnikove podatke avtenticira kot pravilne, se izvrši preusmeritev na pogled Objave.

Kontroler objav služi za preusmerjanje uporabnika med stranema Objave ter Dodaj novo objavo. V primeru, da uporabnik zbriše objavo na seznamu objav, se sproži funkcija znotraj kontrolerja objav. Funkcija poskrbi za izbris objave znotraj baze in na preusmeritev nazaj na stran objav. Če uporabnik želi dodati novo objavo s pritiskom na gumb Dodaj novo objavo, sproži preusmeritev na pogled Dodaj novo objavo, na katerem je obrazec za vnos naslova ter vsebine objave. Kontroler skrbi, da se novo dodana objava shrani v podatkovno bazo in preusmeri uporabnika na stran objav.

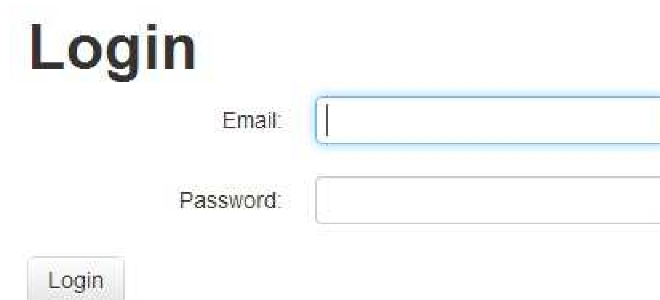
Poleg dveh kontrolerjev sta v strukturo spletne strani vključena tudi dva modela. Prvi je model uporabnika, ki skrbi za posredovanje uporabnikovih podatkov. Uporabljata ga pogled Prijava ter prijavni kontroler. Drugi je model Objava, ki posreduje podatke pogledoma Objave in Dodaj novo objavo. Za interakcijo skrbi kontroler objav.

4.2 Testiranje pogleda

Pogled predstavlja vse, kar uporabnik vidi na spletni strani. Za testiranje pogleda se uporablja sprejemne teste, ki morajo biti napisani tako, da upoštevajo strankine želje. Programer mora torej upoštevati in napisati teste, ki preverijo vse zahtevane funkcionalnosti s strani stranke. Če so sprejemni testi opravljeni, se spletno stran objavi za uporabo uporabnikom. V naslednjem poglavju je prikazan test pogleda Prijava.

4.2.1 Sprejemni testi za prijavni obrazec

Prijavni obrazec sestavljajo tri komponente. Pogled, ki ga uporabnik vidi, vsebuje vnosno polje za elektronsko pošto, vnosno polje za geslo in gumb za prijavo na spletno stran.



Slika 4.2: Prijavni obrazec

Vsako vnosno polje ima pravila, ki se jih mora uporabnik držati ob izpolnjevanju. Pravila so namenjena validaciji vnosa podatkov. Vpisani podatki se posredujejo kontrolerju, ki preveri validacijo in avtentikacijo uporabnika. Če sta obe pravilni, kontroler posreduje odgovor, ki uporabnika preusmeri na naslednjo stran. Ob napačni izpolnitvi kontroler vrne uporabnika nazaj na prijavni pogled s sporočilom o napaki. Prijavni obrazec uporablja tudi model User. Model je namenjen pridobivanju podatkov o uporabnikih s podatkovne

baze. Pri avtentikaciji uporabnika se preveri, ali uporabnik z vpisanim elektronskim naslovom in geslom obstaja v podatkovni bazi, za kar poskrbi model. Sprejemni testi za prijavni obrazec simulirajo uporabnikovo izpolnjevanje obrazca. Postopek za testiranje pravilno izpolnjenega prijavnega obrazca je naslednji:

- Test se postavi na prijavno stran,
- preveri, ali je to res prijavna stran,
- izpolne se polje za elektronsko pošto ter vpiše geslo,
- izvede se klik na gumb "Login",
- test preveri, ali se nahaja na naslednji strani z url-jem "/login"
- in ali stran vsebuje napis "You have arrived."

```
<?php
$I = new AcceptanceTester($scenario);
$I->wantTo('login with credentials');
$I->amOnPage('/');
$I->see('Login');
$I->fillField('email','matic.volk@gmail.com');
$I->fillField('password','1234');
$I->click('Login');

$I->seeCurrentUrlEquals('/login');
$I->see('You have arrived.');
```

Človek je zmotljiv, zato je potrebno preveriti, ali se aplikacija pravilno odziva tudi na napačne vnose. V ta namen je potrebno opraviti test za napačen vnos podatkov. Pri testiranju napačnega vnosa sta pomembna dva testa. Prvi test preveri pravilen odziv obrazca ob napačno izpolnjenih poljih. Če uporabnik napačno vpiše elektronski naslov ali sploh ne izpolni vnosnega polja, se sproži

napaka validacije. Test preveri, ali kontroler preusmeri uporabnika na pravilno stran ter ali se statusno sporočilo ujema.

```
<?php
$I = new AcceptanceTester($scenario);
$I->wantTo('let empty form inputs');
$I->amOnPage('/');
$I->see('Login');
$I->click('Login');
$I->seeCurrentUrlEquals('/');
$I->see('The email field is required.');
$I->see('The password field is required.');
```

Namen drugega testa je odziv na napačen vnos elektronskega naslova ali gesla uporabnika. Pri tem testu pride do napake pri avtentikaciji, ker tak uporabnik ne obstaja v podatkovni bazi, zato je potrebno uporabnika preusmeriti na prijavno stran ter ga obvestiti o napaki pri avtentikaciji. Test ob napačni avtentikaciji preveri izpis sporočila, ki se izpiše na prijavni strani.

```
<?php
$I = new AcceptanceTester($scenario);
$I->wantTo('login with wrong credentials');
$I->amOnPage('/');
$I->see('Login');
$I->fillField('email','matic.volk@gmail.com');
$I->fillField('password','1111');
$I->click('Login');

$I->seeCurrentUrlEquals('/');
$I->see('Wrong credentials.');
```

Po izvedenih testih se v konzoli izpiše stanje. Če je posamezen test bil pravilno izveden, se izpiše beseda "Ok"na koncu opisa testa.

```
$ codecept run
Acceptance Tests (3)
----------------------------------------------------------
Trying to login with credentials(LoginWithCredentialsCept)
Ok
----------------------------------------------------------
Trying to let empty form inputs(WrongFillFormErrorTestCept)
Ok
----------------------------------------------------------
Trying to login with wrong credentials(
   LoginWithWrongCredentialsCept)
Ok
----------------------------------------------------------
Time: 4.52 seconds, Memory: 8.00Mb
OK (3 tests, 10 assertions)
```

4.3 Testiranje kontrolerja

Pri testiranju kontrolerja je potrebno testirati le posredovanje podatkov in interakcij med pogledom ter modelom. Testiranje vključuje preverjanje [15]:

- Zahtevkov,
- odgovorov,
- pravilnosti poslanih spremenljivk pogledu.

Proces testiranja kontrolerja zajema [5]:

- Izolacijo vseh odvisnosti,
- sproženje želene metode,
- preverjanje odziva metode.

4.3.1 Priprava za testiranje kontrolerja

Zaradi interakcij kontrolerja z modelom in pogledom je potrebno testne enote izolirati in popraviti strukturo kontrolerja, saj je le tako zagotovljeno testiranje enot. Če kontroler nima konstruktorja, ga je potrebno dodati.

```
class PostController extends \BaseController {
        protected $post;
        public function __construct(Post $post)
        {
                $this->post = $post;
        }
        //...
```

Dodan konstruktor omogoča generiranje navideznega objekta Post, ki simulira dejanski objekt. Testni razred PostControllerTest prav tako potrebuje začetne nastavitve konstruktorja ter tearDown metodo, ki ob končanem testiranju prekine Mockery.

```
class PostControllerTest extends TestCase {

        public function __construct()
        {
                $this->mock = Mockery::mock('Eloquent', '
                   Post');
        }

        public function tearDown()
        {
                Mockery::close();
    }
    //...
```

V konstruktorju se inicializira objekt modela Post, ki ga potrebujejo testne metode.

4.3.2 Testi metod kontrolerjev

Osnoven test za prikaz strani Objava se preveri s klicem metode index v post kontrolerju. Metoda se sproži ob klicu posts z metodo GET. Test preveri, ali je bil odgovor ustrezen in ali pogled vsebuje spremenljivko posts.

```
public function testIndex()
{
        $this->call('GET', 'posts');
        $this->assertResponseOk();
        $this->assertViewHas('posts');
}
```

Preverjanje zahtevka ima, za dostop do obrazca za dodajanje nove uporabnikove objave, podoben postopek. Razlika je le v klicu, kjer se razlikuje usmerjevalni id. Za dostop do obrazca je napisana usmeritev (route) posts/create. Usmeritev zahtevka za shranitev nove objave uporabnika je testiran za dva primera. V primeru uspešno dodane objave, test generira novo objavo z naslo-

vom, ki je obvezen vnos v obrazcu in nato sproži klic metode store, ki shrani objavo v PB. Preveri se preusmeritev na index stran, kjer je seznam vseh objav uporabnika.

```
public function testStorePostSuccess()
{
        $input = ['title' => 'Hello'];
        $this->mock->shouldReceive('create')->once();
        $this->app->instance('Post',$this->mock);
        $this->call('post','posts', $input);
        $this->assertRedirectedToRoute('posts.index');
}
```

Če zahtevek ni pravilen, se objava ne shrani v PB in kontroler zavrne zahtevek ter preusmeri uporabnika nazaj na vnosni obrazec. Test za zavrnjen zahtevek generira objavo s praznim naslovom. Preveri se preusmeritev nazaj na vnosni obrazec ter ali seja vsebuje napake.

```
public function testStorePostFails()
{
        $input = ['title' => ''];
        $this->app->instance('Post',$this->mock);
        $this->call('POST', 'posts', $input);
        $this->assertRedirectedToRoute('posts.create');
        $this->assertSessionHasErrors(['title']);
}
```

Ostane le še preverjanje delovanja izbrisa objave iz PB. Pri izbrisu se sproži metoda kontrolerja imenovana delete, ki izbriše določen vnos in preusmeri uporabnika na seznam objav. Test preveri pravilnost preusmeritve.

```
public function testDeletePost()
{
        $this->call('GET','posts/{1}');
        $this->assertRedirectedToRoute('posts.index');
}
```

4.4 Testiranje modela

Model vsebuje 4 glavne sklope testiranja [15]:

- Validacije modela,
- relacije modelov,
- Get, Set metode,
- lastne metode.

Vsakega od naštetih testov uvrščamo med teste enot.

4.4.1 Priprava za testiranje modela

Za testiranje kode modela, ki potrebuje interakcijo s podatkovno bazo, je potrebno pripraviti nekaj funkcij. Testiranje je bolje opravljati znotraj testne baze, saj se s tem prepreči spreminjanje podatkov ali strukture produkcijske baze. PHPUnit omogoča uporabo podatkovne baze, ki se ob vsakem testiranju generira znotraj glavnega pomnilnika. Za uporabo take PB je potrebno dodati datoteko z imenom database.php znotraj app/config/testing mape. Datoteka database.php mora vsebovati naslednje nastavitve:

```
<?php
return array(
        'default' => 'sqlite',
        'connections' => array(
                'sqlite' => array(
                        'driver'   => 'sqlite',
                        'database' => ':memory:',
                        'prefix'   => '',
        ),
        ...
)
```

Opcija 'database' => ':memory:' generira PB v glavnem pomnilniku. V pomnilniku se ohrani, dokler je odprta povezava. Poleg nastavitev povezave podatkovne baze je potrebno dodati metodo setUp() znotraj TestCase razreda, ki poskrbi za vzpostavitev začetnega stanja PB.

```
class TestCase extends Illuminate\Foundation\Testing\
   TestCase {
        public function setUp()
        {
                parent::setUp();
                Artisan::call('migrate');
        }
```

Ob koncu testiranja je metoda tearDown tista, ki povrne stanje PB v začetno stanje. Tako kot metoda setUp, se tudi ta metoda nahaja v razredu TestCase.

```
        public function tearDown()
        {
                Artisan::call('migrate:rollback');
        }
```

Poleg nastavitev za migracijo PB se za lažje testiranje modela uporablja knjižnici ModelHelpers in Factory. Obe knjižnici se vključi na začetku testnega razreda:

```
use Way\Tests\Factory;
use Way\Tests\ModelHelpers;
```

ModelHelpers vsebuje metode za lažje testiranje relacij med modeli.

4.4.2 Testiranje validacije modela

Model vsebuje pravila, ki preprečujejo napačno vpisovanje podatkov v PB. Preden se izvede nov vnos v bazo se vnosni podatki validirajo. Testiranje validacije User modela: Model vsebuje 4 atribute:

- Id,
- name,
- email,
- password.

Pravila modela določajo obvezen vnos gesla in poštnega naslova, pri čemer mora biti poštni naslov unikaten. Testni razred modela vsebuje test pravilnosti dodajanja novega uporabnika, za katerega poskrbi metoda testValidNewUser. Metoda najprej generira novega uporabnika s pomočjo Factory, ki vrne objekt uporabnika. Podatka o elektronskem naslovu in geslu sta dodana kot parameter. Test je opravljen, če validacija nad novim uporabnikom vrne true.

```
<?php
use Way\Tests\Factory;

class ValidationUserModelTest extends TestCase {
        public function testValidNewUser()
        {
                $user = Factory::user(array('email' => '
                    matic.volk@gmail.com', 'password' => '
                    1234'));
                $user->email = 'matic.volk@gmail.com';
                $user->password = '1234';
                $this->assertTrue($user->validate());
        }
```

Testna metoda testInvalidWithoutEmail testira odziv validacije ob generiranju novega uporabnika brez elektronskega naslova. V primeru validacije uporab-

nika brez elektronskega naslova vrne false. Testna metoda preveri, ali validacija ne uspe.

```
public function testInvalidWithoutEmail()
{
        $user = Factory::user(array('email' => null));
        $this->assertFalse($user->validate());
}
```

Za testiranje unikatnosti elektronskega naslova uporabnika skrbi metoda testInvalidWithoutUniqueEmail. Metoda najprej generira uporabnika z določenim naslovom in ga shrani v PB. Nato generira še enega uporabnika z enakim elektronskim naslovom in preveri njegovo validacijo. Če validacija ne uspe, se test pravilno izvede.

```
public function testInvalidWithoutUniqueEmail()
{
        $user = Factory::user(
                array('email' => 'matic.volk@gmail.com')
        );
        $user->save();
        $this->assertFalse($user->validate());
}
```

4.4.3 Testiranje relacij modelov

Modeli si delijo relacije, ki jih povezujejo v skupno celoto. Primer relacije ena proti mnogo je relacija med uporabnikom in njegovimi objavami. Metoda testHasManyPosts preveri, če ima uporabnik več objav, kar nakazuje relacijo ena proti mnogo.

```
public function testHasManyPosts()
{
        $this->assertHasMany('posts','User');
}
```

4.4.4 Testiranje Get/Set metod

Model uporabnika vsebuje Get/Set metodi za nastavljanje in pridobivanje podatka o imenu uporabnika. Za testiranje metod sta napisana dva preprosta testa. Prvi test je namenjen testiranju metode, ki vrača ime uporabnika. Najprej se ustvari uporabnik z imenom Janez in nato preveri, ali se ime ujema z imenom, ki ga get metoda vrne.

```
public function testGetNameMethod()
{
        $user = Factory::user(array('name' => 'Janez'));
        $this->assertEquals('Janez',$user->getName());
}
```

Drugi test preveri pravilnost nastavljanja vrednosti imena uporabnika. Test ustvari novega uporabnika z naključnimi podatki. Nato kliče metodo setName, ki nastavi uporabniku ime Janez. Na koncu preveri, ali se pričakovano ime uporabnika ujema z dejanskim.

```
public function testSetNameMethod()
{
        $user = Factory::user();
        $user->setName('Janez');
        $this->assertEquals('Janez',$user->getName());
}
```

4.4.5 Testiranje lastnih metod

Lastne metode so tiste, ki jih programer implementira za pomoč pri dosegi funkcionalnosti, ki jo potrebuje. Testiranje lastnih metod se zelo razlikuje glede na posamezno metodo, predvsem pa je odvisno, kaj določena metoda naredi, temu primerno pa mora tudi test preveriti njeno delovanje. Kot primer navajam enostavno metodo, ki vrne domeno elektronskega naslova uporabnika. Test ustvari novega uporabnika z določenim elektronskim naslovom ter nato preveri, ali metoda getEmailDomain vrne domeno, ki se ujema s pričakovano.

```
public function testGetEmailDomain()
{
        $user = Factory::user(
                array('email' => 'matic.volk@gmail.com')
        );
        $this->assertEquals('gmail.com',$user->
            getEmailDomain());
}
```

4.5 Testiranje podatkovne baze

Testiranje podatkovne baze je zahtevno, ker lahko z dostopanjem do podatkovne baze v času testiranja, pride do sprememb znotraj baze. Posledično se lahko spletna aplikacija sesuje, zato je bolje testirati brez direktnih zahtevkov v podatkovno bazo ali pa s pomočjo testne podatkovne baze. Med testiranjem je potrebno poskrbeti za [15]:

- Shemo in tabele,
- vnašanje vrstic v tabele, ki so potrebne za testiranje,
- stanje PB med testiranjem,
- opravljanje testov z začetnega stanja PB.

Podatki, ki jih prejme spletna aplikacija preko PB, so globalni. To pomeni, da so lahko istočasno dostopni večkrat, kar lahko vpliva na končni rezultat testiranja, če test ni pravilno zapisan ali pa testno okolje ni pravilno pripravljeno. Časovna zahtevnost testiranja podatkovne baze je odvisna od količine podatkov, ki jih test procesira. Če je količina podatkov zelo velika, se tudi sorazmerno poveča čas testa. Časovno zahtevnost je mogoče skrajšati s testiranjem majhnih koščkov PB.

4.6 Testiranje pri posodobitvi kode na odlagališče (repositorij)

Način continuous integration razvoja aplikacij ima veliko prednosti pri hitrosti razvijanja in uporabe najnovejše verzije kode za produkcijo, vendar pride hitro do napak, ki se razširijo na vse programerje, če uporabijo na novo posodobljeno kodo z napakami. Za odpravitev tega problema obstajajo servisi, ki skrbijo, da se posodobljena koda na odlagališču tudi testira in obvesti programerje o trenutnem stanju. Travis je odprtokodni servis integriran v GitHub, ki omogoča vzpostavitev okolja za različne jezike. Istočasno lahko testira več različic okolja ter uporablja različne podatkovne baze [11], [15].

4.6.1 Uporaba servisa Travis

Travis ne deluje samostojno, ampak se registrira na posamezno odlagališče. Uporabnik se registrira na strani $https://github.com/$, ustvari novo odlagališče in naloži kodo na ustvarjeno odlagališče. Naslednji korak se izvaja na strani $https://travis-ci.org/$, kjer se je potrebno ponovno registrirati. Na novem profilu je seznam odlagališč, ki ga je treba sinhronizirati s GitHub uporabniškim računom. Vsa odlagališča na strani GitHub se sinhronizirajo in prikažejo na seznamu odlagališč Travis uporabniškega profila. Odlagališču, ki ga uporabnik želi spremljati s pomočjo servisa Travis, je potrebno preklopiti opcijo On. V korensko mapo projekta je potrebno dodati datoteko .travis.yml, ki vsebuje informacije o jeziku, različici sistema ter ostalih konfiguracijskih podatkih. Osnovna informacija, ki jo Travis potrebuje za delovanje, je jezik, v katerem je projekt napisan. Če ima datoteka .travis.yml vključenih več različic jezika, se za vsako različico posebej vzpostavi okolje in preveri delovanje projekta. Nastavi se lahko tudi skripto, ki se izvede pred glavnim delom programa in skripto, ki se izvede po zaključenem glavnem delu programa. Skripte pred in po glavnem delu programa so namenjene vzpostavljanju vseh potrebnih virov

za pravilno izvajanje in na koncu za vrnitev programa v začetno stanje.

```
language: php
php:
- 5.4
- 5.5
before_script:
- composer install --dev --prefer-source --no-interaction
- mysql -e 'create database IF NOT EXISTS myapp_test;'
script: phpunit
```

Ko so osnovne nastavitve nastavljene in datoteka .travis.yml dodana v korenski mapi, je potrebno le še posodobiti kodo na odlagališču. Po posodobitvi se avtomatsko zažene Travis in opravi testiranje. Na GitHub je vidno stanje odlagališča, ki ga testira Travis. Če je prišlo do napak pri zadnji posodobitvi kode, se ob potrditvi uporabnika odlagališče vrne na prejšnje stanje.

Poglavje 5

Zaključek

Razvoj spletnih aplikacij je v porastu, saj je do njih mogoče dostopati z vsake internetno dostopne točke. Ljudje lahko dostopajo do svojih podatkov neodvisno od naprave, ki jo uporabljajo, naj si bo mobilni telefon, tablica ali prenosni računalnik. Pomembno je, da so podatki dostopni, kar pri namiznih aplikacijah ni mogoče.

S tako velikim poudarkom na spletnih aplikacijah, se je moral prilagoditi in izboljšati tudi razvoj. Poznane so nam razne metode razvoja aplikacij, ki skrbijo, da razvijalci razvijajo produkte na enostaven, hiter ter učinkovit način. Predvsem izstopajo agilne metode razvoja aplikacij, zato sem se v diplomski nalogi osredotočil na razvoj spletne strani s pomočjo testno vodene metode (Test driven development). Do te ideje je prišlo zato, ker sem vedno imel težave z vnašanjem novo napisane kode v že obstoječo, lahko pa se je tudi zgodilo, da je z novo vnešeno kodo prenehal delovati že delujoči del aplikacije.

Cilj diplomske naloge je bil, da se seznanim z metodo Test driven development, ki je pripomogla k temu, da sem začel pisati spletno stran s pomočjo testov, ki jih napišemo, še preden začnemo z implementacijo funkcionalnosti posameznega dela spletne strani. Prišel sem do novih spoznanj in vidikov razvijanja ter opazil bistveno razliko pri razmišljanju in organizaciji razvijanja spletne strani. Najprej sem moral razmisliti o funkcionalnosti, ki jo želim te-

stirati in napisati test. Ko je bil test napisan, ga je bilo potrebno opraviti in nazadnje optimizirati kodo, ki je test opravila. Testi so mi omogočili hitro zaznavanje napak in povratno informacijo o problemih, zato sem lahko spletno stran hitreje razvijal ter z večjo mero samozavesti spreminjal kodo.

V prihodnosti bodo agilne metode še učinkovitejše, posledično se bodo tudi testi prilagodili novejšim metodam. Testna razvojna okolja se bodo izpopolnjevala vzporedno z izboljšanimi metodami razvoja. Spremljanje sprememb in upoštevanje novih testnih metod bo prineslo prednosti razvijalcem v razvoju.

Literatura

[1] Agile testing, http://en.wikipedia.org/wiki/agile_testing.

[2] Ci, http://www.versionone.com/agile101/continuous_integration.asp.

[3] Codeception, http://codeception.com/.

[4] Composer, https://getcomposer.org/.

[5] Culttt, http://culttt.com/.

[6] Iterativni modeli, http://www2.gov.si/mju/emris.nsf.

[7] Laracasts, https://laracasts.com/.

[8] Laravel, http://laravel.com/.

[9] Manual testing, http://en.wikipedia.org/wiki/manual_testing.

[10] Phpunit, http://phpunit.de/.

[11] Travis, http://docs.travis-ci.com/user/getting-started/.

[12] World wide web, http://en.wikipedia.org/wiki/world_wide_web/.

[13] Lisa Crispin and Janet Gregory. Agile testing. In *Agile Testing, A Practical Guide for Testers and Agile Teams*. Addison-Wesley, 2009.

[14] Robert Cecil Martin. Agile software development principles, patterns and practices. In *Agile Software Development Principles, Patterns and Practices*. Alan Apt Series, 2002.

[15] Jeffrey Way. Laravel testing decoded. In *Laravel Testing Decoded*. Leanpub, 2013.