Univerza v Ljubljani
Fakulteta za računalništvo in informatiko

Matej Škrlep

Testno voden razvoj v programskem ogrodju Symfony2

diplomsko delo

visokošolski strokovni študijski program prve stopnje računalništvo in informatika

Mentor: izr. prof. dr. Viljan Mahnič

Ljubljana 2014

Besedilo je oblikovano z urejevalnikom besedil LaTeX.

Fakulteta za računalništvo in informatiko izdaja naslednjo nalogo:

Testno voden razvoj v programskem ogrodju Symfony2

Tematika naloge:

Proučite postopek razvoja programske opreme po metodi Scrum in opišite, kako se vanj vklaplja testiranje razvitih programov. Opredelite različne vrste testov in predstavite postopek testno vodenega razvoja. Opišite orodja, ki se uporabljajo za testno voden razvoj v okviru ogrodja Symfony2, in prikažite postopek testiranja na primeru iz prakse.

Izjava o avtorstvu diplomskega dela

Spodaj podpisani Matej Škrlep, z vpisno številko **63020153**, sem avtor diplomskega dela z naslovom:

Testno voden razvoj v programskem ogrodju Symfony2

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 18. september 2014 Podpis avtorja:

Zahvaljujem se mentorju izr. prof. dr. Viljanu Mahniču za skrbno pomoč pri diplomskem delu.

Zahvalil bi se staršem za življenjske modrosti, ki sem jih bil deležen tekom odraščanja.

Hvala ženi Jani, ki me je podpirala, spodbujala in vedno stala ob strani.

Hvala Agati in Lovru, ker mi vsak dan narišeta nasmeh na obraz.

Jani, Agati in Lovru.

Kazalo

Seznam uporabljenih kratic

kratica	angleško	slovensko
BDD	behaviour driven development	vedenjsko voden razvoj
IDE	integrated development environment	integrirano razvojno okolje
TDD	test driven development	testno voden razvoj
XML	extensible markup language	razširljiv označevalni jezik
AJAX	asynchronous javascript and XML	asinhroni javascript in XML
CSS	cascading style sheets	kaskadne stilske predloge
JSON	javascript object notation	javascript objektna notacija
API	application programming interface	spletni programski vmesnik
SQL	structured query language	strukturiran povpraševalni jezik
RWD	responsive web design	odzivni spletni design
HTML	hypertext markup language	označevalni jezik za oblikovanje dokumentov
CI	continuous integration	sprotna integracija
CD	continuous delivery	sprotna dostava

Povzetek

Z napredkom tehnologij, ki se uporabljajo pri razvoju programske opreme, postajajo aplikacije vedno bolj napredne in kompleksne. Klasične metodologije razvoja programske opreme pa zaradi svoje rigidnosti vedno večja ovira. Zato v zadnjih letih na priljubljenosti pridobivajo agilne metodologije razvoja, ki pa od vpletenih zahtevajo veliko komunikacije. Rezultat tega je boljše poznavanje projekta vseh vpletenih in verjetnost, da končna aplikacija ne bo zadovoljila naročnika posledično toliko manjša. S kratkimi razvojnimi cikli agilne metodologije zahtevajo hiter razvoj rešitev. Testno voden razvoj pa nam pripomore pri zagotavljanju kvalitete kode, ki zahteva od razvijalcev, da že pred implementacijo spišejo test.

Diplomsko delo se osredotoča na agilno metodologijo Scrum in uporabo testno vodenega razvoja v spletnem razvojnem ogrodju Symfony2 za programski jezik PHP, ki zaradi modularnosti in dobre podpore skupnosti postaja vedno bolj priljubljen. S primeri iz realnega projekta so predstavljeni načini uporabe orodij za testiranje v ogrodju Symfony2 in prednosti, ki smo jih s takim pristopom pridobili na našem projektu.

Ključne besede: testno voden razvoj, vedenjsko voden razvoj, ogrodje Symfony2, Scrum, programski jezik PHP, PHPUnit, Behat, Mink.

Abstract

With the advance of technologies used in the software development process, the applications become more and more advanced and complex. Classic methodologies of software development are becoming a big hurdle because of their rigidity. As a consequence, agile software development methodologies are becoming more popular. They demand more communication between the customer and the development team. As a result, both parties better understand the application which leads to better customer satisfaction. With short development cycles, agile methodologies require fast production of code. With test driven development, where developer must first produce a test for the new functionality, we can assure better quality of code.

This thesis focuses on agile methodology Scrum and the use of test driven development principles in Symfony2 framework for the development language PHP. Symfony2 framework is becoming more popular because of its modularity and good support from the community.

With real life examples we will demonstrate how to use tools for software testing in Syfmony2 framework and advantages we gained using test driven development on our project.

Keywords: test driven development, behaviour driven development, Symfony2 framework, Scrum, PHP programming language, PHPUnit, Behat, Mink.

Poglavje 1

Uvod

Hiter napredek tehnologije nam omogoča, da razvijamo vedno kompleksnejše in naprednejše rešitve. Trg postaja vedno bolj zahteven, saj poleg tega zahteva tudi hitro realizacijo in cenovno ugodne ter kvalitetne rešitve. S klasičnimi metodami razvoja programske opreme je to zelo težko dosegljivo. Pogosto se pojavi problem že pri specifikacijah, ki so pri klasičnih metodah ključne. Lahko postanejo prezahtevne ali pa preohlapne. Naročnik dolgo čaka na prvo testno različico aplikacije, ki je nato še preobsežna za temeljito testiranje. Tudi naknadne spremembe postanejo težavne, sploh če zahtevajo večje posege v arhitekturo. Zato se sedaj vedno bolj pogosto, pri razvoju programske opreme, pojavljajo agilne metodologije [1], ki omogočajo veliko mero fleksibilnosti.

Scrum [2], ena najbolj priljubljenih agilnih metodologij, daje poudarek na dobro spisani kodi, ki nadomešča specifikacije. Na začetku projekta se določi okvirne funkcionalnosti. Skupaj z naročnikom se določijo prioritete, ki se potem bolj natančno definirajo in ovrednotijo. Razvoj poteka v kratkih iterativnih ciklih, ki lahko variirajo od nekaj dni do nekaj tednov. Scrum zahteva veliko interakcije med razvojno ekipo in naročnikom. Cilj tega pa je, da lahko po koncu vsakega cikla razvojna ekipa ustvari razširitev, ki je uporabna in potencialno pripravljena za izdajo.

Pri vsem tem pa je zelo pomembno, da razvojna ekipa dobro opravi delo. Da proizvaja lepo kodo in predvsem brez ali brez njih. Tu nam priskoči na pomoč testno voden razvoj programske opreme z avtomatskimi testi. Ta zahteva, da preden se dejansko lotimo razvoja nove funkcionalnosti, najprej za to napišemo

test. Potem z minimalno količino kode zagotovimo, da se test uspešno izvede. Šele na koncu pride na vrsto optimizacija kode. Moramo pa se zavedati, da ni potrebno testirati vsake funkcionalnosti, da na koncu ne posvečamo več pozornosti testom, kot pa sami kodi.

Ker pa tak način razvoja prinaša kar nekaj prednosti tako za naročnika kot izvajalca, smo se odločili, da v tem diplomskem delu predstavimo testno voden razvoj v kombinaciji z agilno metodologijo Scrum v PHP programskem ogrodju Symfony2 [3]. Prikazali bomo, kako s testno vodenim razvojem programske opreme lahko poskrbimo, da ohranjamo kvaliteto kode na visoki ravni.

Zato bomo v naslednjem poglavju predstavili agilno metodologijo razvoja programske opreme Scrum ter tehniki testno vodenega razvoja (angl. Test Driven Development ali TDD) in vedenjsko vodenega razvoja (angl. Behaviour Driven Development ali BDD). V tretjem poglavju bomo predstavili lastnosti ogrodja Symfony2 in modulov PHPUnit ter Behat, ki dodata ogrodju učinkovite funkcionalnosti za testiranje. V četrtem poglavju bomo predstavili uporabo teh orodij v praksi in pokazali nekaj primerov testov za modula PHPUnit in Behat. Omenili bomo tudi prednosti in slabosti testno vodenega razvoja, ki smo jih opazili. Za konec pa bomo povzeli ugotovitve in priporočila za čim bolj učinkovito rabo teh orodij.

Poglavje 2

Agilna metoda Scrum in testno voden razvoj

2.1 Agilna metoda Scrum

Scrum je iterativna agilna metodologija razvoja programske opreme. Definira fleksibilno strategijo razvoja, kjer majhna razvojna ekipa deluje enotno za skupni cilj. Ekipi omogoča samoorganizacijo in spodbuja sodelovanje in dnevno komunikacijo z naročnikom.

Osnova Scruma je Manifest agilnega razvoja programske opreme [4], ki daje poudarek sledečim vrednotam:

posamezniki in interakcije imajo prednost pred procesi in orodji,

delujoča programska oprema pred vseobsežno dokumentacijo,

sodelovanje s stranko pred pogodbenimi pogajanji,

odziv na spremembe pred togim sledenjem načrtom.

Manifest pravi tudi: "Četudi cenimo dejavnike na desni, vseeno bolj cenimo tiste na levi."

Bistvo Scruma je spoznanje, da se med projektom zahteve naročnika lahko spremenijo. Pri tradicionalnih metodah razvoja programske opreme spremembe med razvojem lahko povzročijo veliko težav. Zato se pri Scrumu fokusiramo raje

na hitro realizacijo zaključenega nabora zahtev, kar omogoča hiter odziv na prihajajoče zahteve.

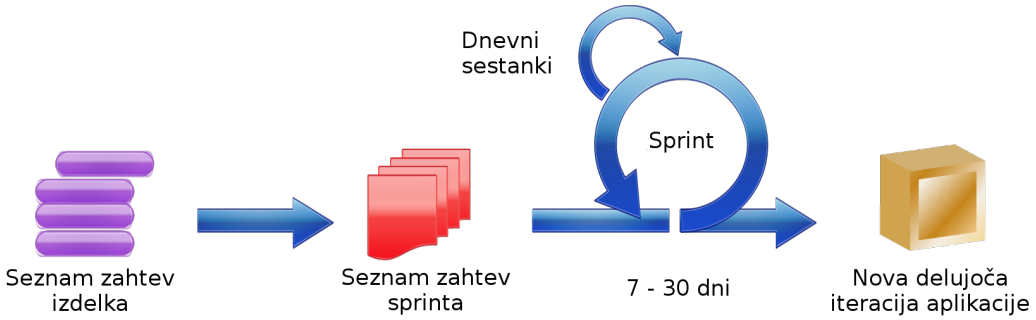
Scrum pa ima tudi svoje omejitve. Ni primeren za večje razvojne ekipe (20 ali več članov), ker to močno otežuje načrtovanje. Metodologija mora imeti podporo pri vodstvu in pri naročniku, kajti zahteva veliko komunikacije med vsemi vpletenimi in znotraj iteracij ne dovoli velikih sprememb načrtov. Potrebno je tudi dobro časovno ocenjevanje nalog, kajti drugače se naloge lahko podaljšajo v naslednji sprint. Zato je zelo pomembno dobro in temeljito definirati naloge.

2.1.1 Način razvoja po metodologiji Scrum

Kakor si lahko pogledamo na sliki 2.1, moramo na začetku projekta vse zahteve v obliki uporabniških zgodb (angl. user stories) združiti v seznam zahtev izdelka (angl. product backlog). Naročnik je odgovoren za seznam uporabniških zgodb, njihovo vsebino in prioriteto. Seznam zahtev ni nikoli dokončan in se razvija sproti z razvojem produkta, ko naročnik vidi potrebo po novih funkcionalnostih.

Pred začetkom vsake iteracije ali sprinta se organizira sestanek za načrtovanje sprinta (angl. sprint planning meeting). Dolžina enega sprinta se lahko močno razlikuje. Traja lahko od enega tedna pa do enega meseca. Načrt sestavlja celotna ekipa z naročnikom produkta. Glede na ocenjeno hitrost ekipe (angl. velocity), ki je odvisna od števila članov razvojne ekipe in od njihovega tehničnega znanja, se glede na prioriteto izberejo uporabniške zgodbe, katerih ocenjena zahtevnost je manjša od ocenjene hitrosti ekipe. Določi se tudi cilje, ki bodo doseženi v tem sprintu.

Razvojna ekipa potem uporabniške zgodbe razdeli na naloge. Med sprintom se ne uvaja sprememb, ki bi vplivale na zadane cilje. Lahko pa pride do prekinitve le-tega, če se v tem obdobju spremeni usmeritev naročnika ali če se spremeni trg ali tehnologija. To možnost ima le naročnik. Razvojna ekipa se vsak dan med sprintom dobiva na dnevnih sestankih (angl. Daily Scrum). To so kratki, do petnajst minut trajajoči sestanki, ki so namenjeni uskladitvi ekipe. Ti sestanki se priljubljeno imenujejo tudi "daily stand-up", ker je zaželeno, da vsi člani stojijo z namenom, da se sestanek hitreje odvija. Na tem sestanku vsak član poroča o delu, ki ga je opravil pretekli dan in o delu, ki ga ima načrtovanega za danes. Poroča tudi o potencialnih težavah ali predlogih, če jih ima. Na koncu sestanka ekipa



Slika 2.1: Shema sprinta

pogleda, če jim uspeva delo opravljati v skladu z načrtom za ta sprint.

Na koncu sprinta se odvijeta revizija (angl. Sprint Review) in retrospektiva sprinta (angl. Sprint Retrospective). Na prvem sestanku razvojna ekipa z naročnikom preveri, kaj je bilo v tem obdobju končano, katere stvari so ostale odprte in s kakšnimi težavami so se soočili. Opravljene naloge skupaj z naročnikom preverijo in tudi potestirajo. Potem preverijo seznam zahtevkov in prečrtajo opravljene uporabniške zgodbe in skupaj pogledajo nove. Rezultat tega sestanka je popravljen seznam zahtev produkta, ki je pripravljen za naslednji sprint. Namen retrospektive sprinta je preveriti in oceniti opravljeno delo in poiskati možne izboljšave za bolj učinkovito delo. Ocenjevanje se fokusira na:

- člane ekipe,
- odnose med člani,
- procese,
- orodja,
- pozitivne stvari in potencialne izboljšave,
- načrt za uvedbo izboljšav.

Cilj retrospektive je določiti izboljšave, ki bodo uvedene v naslednjem sprintu.

Slika 2.2: Shema testa črne škatle

2.2 Vrste testov

Na najvišjem nivoju poznamo dve vrsti testiranja. To so tako imenovani testi bele škatle (angl. white-box tests) ter testi črne škatle (angl. black-box tests).

Testi bele škatle so namenjeni testiranju interne strukture in delovanja aplikacije. V tem primeru mora pisec testov dobro poznati in razumeti celotno aplikacije. Testi bele škatle se lahko nanašajo na testiranje enot, integracije in sistema. Najpogostje se uporablja ravno za testiranje enot. Lahko testiramo povezave med enotami, povezave med enotami med integracijo in med podsistemi med sistemskim testiranjem.

Testi črne škatle pa po drugi strani tretirajo kodo kot "črno škatlo", kar nam prikazuje slika 2.2, in so namenjeni testiranju funkcionalnosti brez poznavanja same implementacije. Za pisca testov je dovolj, da ve, kaj aplikacija dela, ne pa, kako to dejansko izvede.

2.2.1 Testi enot

Cilj testiranja enot [5] je izolirati vsak del aplikacije in dokazati, da se vsak individualni del izvaja pravilno. Posamezna enota predstavlja najmanjši del kode, ki ga je še možno testirati. Testi enot določajo stroga pravila, ki jim mora določena koda zadostiti. Tu gre za teste tipa bele škatle, saj se testi osredotočajo na notranjo strukturo testirane enote. Testi enot so primerni za TDD, ker se jih lahko redno izvaja in tako sproti preverjamo pravilnost delovanja aplikacije.

2.2.2 Integracijski testi

Integracijski testi [6] so namenjeni preverjanju funkcionalnih in performančnih zahtev ter zahtev o zanesljivosti, ki so določene za posamezne zaključene sklope

enot. Namen integracijskih testov je preverjati komunikacijo med enotami znotraj različnih procesov. Bistven namen integracijskih testov je, da dobimo seznam preverjeno delujočih procesov znotraj aplikacije.

2.2.3 Ročno testiranje

Ročno testiranje [7] je osnoven proces, ki ga razvojniki redno uporabljamo. Če delamo na projektu, ki se ne poslužuje avtomatskega testiranja, pa je to edini način, da preverimo pravilno delovanje funkcionalnosti. Prav nam pride tudi v primeru, ko ne uspemo zagotoviti kriterijem testa te enote. Glavni problem takega testiranja je, da je časovno zelo potratno in nekonsistentno, saj je težko zagotoviti vedno enake pogoje in postopek izvedbe testa. Je pa ročno testiranje nepogrešljivo pri testiranju izgleda uporabniškega vmesnika. Edino tako lahko preverimo, ali je postavitev elementov pravilna, ali so barve pravilne ipd.

2.2.4 Testi funkcionalnosti

Teste funkcionalnosti [8] uvrščamo med teste črne škatle. Njihov namen je testiranje funkcionalnosti preko zunanjih vmesnikov aplikacije (spletni servisi, uporabniški vmesnik itd.) in nas samo notranje delovanje ne zanima. Testi na določen vhod posredujejo vhodne parametre in potem preverjajo pravilnost odgovora. Tako dejansko simuliramo uporabo posameznih sklopov aplikacije.

2.2.5 Regresijski testi

Regresijski testi [9] so tesno povezani s TDD. Pridejo v poštev, ko v koraku TDD preoblikovanje kode (angl. refactor) optimiziramo delovanje nove funkcionalnosti s čiščenjem odvečne kode in uporabe že obstoječih funkcionalnosti. S preoblikovanjem tvegamo, da katera od obstoječih funkcionalnosti ne deluje več pravilno oziroma pride do napake ali regresije. Za pravilno regresijsko testiranje potrebujemo velik nabor testov, ki pokriva večino funkcionalnosti aplikacije. Zato je smotrno take teste poganjati zelo pogosto, da takoj zaznamo napako in jo odpravimo še preden bi koda prišla do uporabnikov. Najbolje pa je, da proces regresijskega testiranja popolnoma avtomatiziramo.

2.2.6 Sprejemni testi

Sprejemni testi so pri agilnih metodah razvoja povezani z uporabniškimi zgodbami [10]. Le-te se definirajo v sodelovanju z naročnikom in opisujejo določene funkcionalnosti aplikacije. Če se sprejemni test za uporabniško zgodbo izvede uspešno, pomeni, da je funkcionalnonst realizirana v skladu s specifikacijami uporabniške zgodbe. Iz tega lahko sklepamo, da ustreza željam naročnika.

Pri klasičnih metodah razvoja pa pride do izvedbe sprejemnih testov šele na koncu razvoja in preden naročnik prevzame aplikacijo. Ko je aplikacija pripravljena za produkcijsko okolje, naročnik preveri skladnost aplikacije s specifikacijami.

2.3 Testno voden razvoj (TDD)

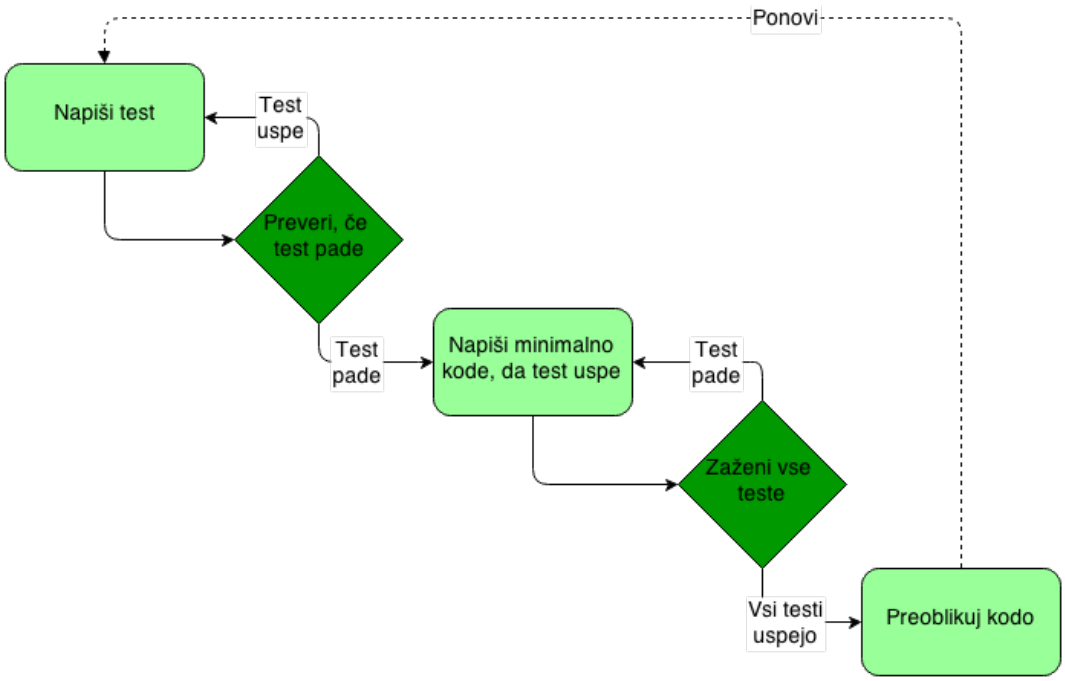
TDD je tehnika razvoja programske opreme, ki temelji na ponavljanju kratkih razvojnih ciklov 2.3. Bistvo te tehnike je, da pred implementacijo funkcionalnosti zanjo napišemo test. Zato je že pred začetkom zelo pomembno, da dobro razmislimo o arhitekturi in načinu izvedbe. Posledično ni namen pisanja testov samo v validaciji programske kode, ampak je poudarek tudi na specifikaciji. Zato ta način dobro sovpada z agilnimi metodologijami razvoja programske opreme.

2.3.1 Proces TDD

Kakor si lahko pogledamo na sliki 2.3, proces razvoja nove funkcionalnosti vedno pričnemo s pisanjem testa. Razvijalec mora dobro premisliti, kako bo nalogo realiziral, da bo znal spisati primeren test. Potem test zaženemo in rezultat mora biti seveda negativen.

V drugem koraku se lotimo realizacije testa. Napišemo minimalno količino kode, ki je potrebna, da zadostimo zahtevam trenutnega testa. Taka rešitev verjetno ni najbolj optimalna, vendar je namen tega koraka samo to, da se test izvede uspešno.

Zadnji korak je preoblikovanje (angl. refactor) kode. V tem koraku poskrbimo, da odstranimo odvečno kodo in poizkusimo uporabiti ali delno prilagoditi že obstoječe funkcionalnosti. S tem ohranimo kodo čisto in lepo strukturirano.



Slika 2.3: Shema testno vodenega razvoja

Tu je ključno dobro poznavanje projekta, da s spremembami v obstoječi kodi ne povzročimo novih napak.

Na koncu je pomembno, da izvedemo regresijske teste in tako zagotovimo delovanje celotne aplikacije.

2.3.2 Prednosti TDD

Ker razvoj pri TDD poteka v kratkih ciklih, lahko dosežemo višjo produktivnost. Razvijalci se tako osredotočajo na manjše probleme, katerim se lahko temeljito posvetijo. To poveča osredotočenost in pripomore k boljši strukturi aplikacije in čistejši kodi. Dobro spisani testi nam lahko pomagajo pri razumevanju tuje kode in lahko postanejo del dokumentacije. Z rednim testiranjem zagotovimo tudi pravilno delovanje aplikacije, kar zmanjša potrebo po naknadnem razhroščevanju kode.

2.3.3 Slabosti TDD

Največja slabost TDD je, da pisanje testov delno upočasni razvoj. Posledično se povečajo tudi začetni stroški projekta in je zato pomembna podpora vodstva. Zato tak pristop ni najbolj primeren za manjše in tehnično manj zahtevne projekte. Ob prehodu na TDD je ponavadi potrebno veliko privajanja na nov način dela. To mnoge odvrača od uporabe TDD ali pa ga med razvojem opustijo.

2.4 Vedenjsko voden razvoj (BDD)

BDD je tehnika, ki temelji na TDD in le-tega nadgrajuje [12]. Razvil jo je Dan North, ki je med poučevanjem agilnih metodologij in TDD zasledil ponavljajoče se težave in nerazumevanja. Razvojniki so se srečevali s sledečimi problemi:

- kje začeti projekt,
- kaj testirati in česa ne,
- kako poimenovati teste,
- razumevanja, zakaj test ne deluje.

Zato BDD popolnoma spremeni pristop k testiranju. Namesto testiranja implementacije, se BDD fokusira na testiranje obnašanja sistema. Testi se pišejo v obliki stavkov in so osnovani na uporabniških zgodbah.

2.4.1 Prednosti BDD

Glavna prednost BDD je v tem, da se testi pišejo s celimi stavki, kar nam omogoča lažje odkrivanje napak. Ker so napisani v človeku prijazni obliki, so tudi tehnično nepodkovanemu kadru dobro razumljivi. Če imamo na projektu dobro definirane uporabniške zgodbe, se te lahko prevedejo v korake BDD testov. In ker BDD ni vezan na samo implementacijo, so testi posledično manj občutljivi na spremembe v kodi.

2.4.2 Slabosti BDD

Pomembno pri BDD je dobra kvaliteta uporabniških zgodb. Razvojni ekipi je v veliko pomoč, če je naročnik sposoben pripraviti celovite uporabniške zgodbe. Za to pa je potrebna velika angažiranost in dnevno sodelovanje naročnika z razvojno ekipo, kar lahko predstavlja velik problem. Podobno kot pri TDD, uporaba BDD ni vedno smiselna na vseh projektih, ker lahko na koncu porabimo več časa za pripravo testov, kot samo realizacijo projekta.

Poglavje 3

Orodja

3.1 Ogrodje Symfony2

Symfony2 ogrodje je priljubljeno odprto kodno MVC spletno razvojno ogrodje za programski jezik PHP. Avtor ogrodja Fabien Potencier, ga je razvil leta 2004 za projekte podjetja SensioLabs. Leto kasneje je bilo ogrodje izdano v brezplačno javno rabo pod licenco MIT Open Source.

Namen ogrodja Symfony2 je spodbujati in promovirati profesionalni pristop, najboljše prakse, standardizacijo in interoperabilnost aplikacij. Rezultat tega je pospešen razvoj in zmanjšana potreba po vzdrževanju aplikacij. Z ogrodjem je možno postaviti robustne aplikacije, primerne tudi za korporacijsko okolje. Razvojni ekipi pa omogoča popolno kontrolo nad konfiguracijo.

Osnovo ogrodja Symfony2 sestavlja nabor 29 komponent PHP, kot so:

- ClassLoader
- Console,
- Dependency Injector,
- Event Dispatcher,
- Form,
- Routing,

- YAML.

Komponente so med sabo popolnoma neodvisne, zato se jih lahko brez težav uporabi tudi samostojno na drugih projektih. Bolj poznani projekti, ki slonijo na uporabi teh komponent so Behat, Drupal, EasyBook, Laravel, phpBB in še mnogi drugi.

Ogrodje Symfony2 pa uporablja tudi veliko obstoječih odprtokodnih projektov PHP. To so:

- Doctrine, Database Abstraction Layer (DBAL) in Object relational mapper (ORM),
- testno ogrodje PHPUnit,
- knjižnica za delo z elektronsko pošto Swift Mailer,
- orodje za izdelavo predlog Twig.

Priljubljenost ogrodju dviguje tudi velika in aktivna skupnost, ki skrbi za velik nabor kvalitetnih modulov. Ti so na voljo na spletnih portalih packagist.org, ki ponuja module za različna ogrodja, ali pa na knpbundles.com, ki je specializiran za Symfony2 ogrodje.

3.2 Composer

Composer [13] je orodje, ki skrbi za odvisnosti (angl. dependency manager) med različnimi moduli PHP. V preteklosti je bilo ob uporabi različnih modulov potrebno ročno poskrbeti za module ali knjižnice, od katerih so bili odvisni oziroma so jih potrebovali za svoje delovanje. In seznam potrebnih knjižnic je bilo zato težko ročno nadzorovati in preverjati. Sploh če na enem projektu deluje večja razvojna ekipa. Orodje Composer je bilo zasnovano po zgledu orodja npm za Node.js ali Bundlerja za programski jezik Ruby.

Za to nadležno opravilo danes poskrbi orodje Composer. Composer sam preko spleta pridobi potrebne module, preveri njihove odvisnosti od drugih in avtomatično namesti samo manjkajoče. Tudi uporaba Composerja je zelo preprosta. Ko ga namestimo na naš sistem, znotraj projekta kreiramo datoteko `composer.json`, ki je konfiguracijska datoteka.

Najpomembnejša kategorija v konfiguraciji je `require`:

```
"require": {
    "php": ">=5.4.3",
    "symfony/symfony": "2.6.x-dev",
    "doctrine/orm": "~2.2,>=2.2.3",
    "doctrine/doctrine-bundle": "dev-master",
    "twig/extensions": "~1.0",
    "symfony/assetic-bundle": "~2.3",
    "symfony/swiftmailer-bundle": "~2.3",
    ...
}
```

Vsi moduli znotraj kategorije require, se bodo namestili na vseh okoljih projekta. Poleg imena napišemo tudi, katero verzijo modula želimo uporabljati. Tako imamo res lahko vse pod nadzorom in sami kontroliramo, kdaj bomo nadgradili posamezen modul. Kakor vidimo iz primera, Composer poskrbi tudi za namestitev ogrodja Symfony2.

Poleg tega, pa nam omogoča, da določene module namestimo samo na specifična okolja. Mi smo na razvojno okolje naložili module, ki jih potrebujemo za testiranje, česar na drugih okoljih ne potrebujemo. To nam omogoča kategorija `require-dev`:

```
"require-dev" : {
    "sensio/generator-bundle": "~2.3",
    "fzaninotto/faker": "v1.2.0",
    "doctrine/doctrine-fixtures-bundle": "2.1.*@dev",
    "behat/behat": "3.*@dev",
    "behat/mink": "1.6.*@dev",
    ...
}
```

V osnovi Composer uporablja repozitorij paketov packagist.org, kjer pridobi informacije o modulih, lahko pa mu definiramo svojega. Za to uporabimo kategorijo `repositories`:

```
"repositories": [
{
    "type": "vcs",
    "url": "https://github.com/dlabs/FriendlyContexts.git"
    }
]
```

Poleg tega imamo še kategorije za opis konfiguracije, lahko definiramo skripte, ki se izvedejo pred in po namestitvi modulov.

Ko imamo konfiguracijsko datoteko pripravljeno, pa si lahko iz ukazne vrstice pomagamo z naslednjimi ukazi:

composer install - namesti vse manjkajoče module;

composer update - posodobi vse module ali samo določenega, če ga dodamo kot parameter;

composer require - doda nov modul v konfiguracijsko datoteko;

composer remove - odstrani modul iz konfiguracijske datoteke;

composer self-update - Composer preveri, če zanj obstaja nova verzija in se posodobi;

3.3 JetBrains PhpStorm

PhpStorm [14] je integrirano razvojno okolje (angl. Integrated Development Environment ali IDE) za programski jezik PHP od verzije 5.3 naprej. PhpStorm je močan urejevalnik, ki omogoča označevanje kode, podrobno konfiguracijo oblikovanja kode, sprotno preverjanje napak in dokončevanje kode (angl. code completition). PhpStorm sprotno preverja in analizira kvaliteto kode. Omogoča odpra-

vljanje napak s pomočjo orodja XDebug in ima integrirano podporo za izvajanje PHPUnit avtomatičnih testov. Omogoča integracijo s sistemi verzioniranja kode, kot so: Git, Subversion, TFS in drugi.

Kot mnoga druga podobna orodja ima PhpStorm podporo za različne vtičnike. Za delo s Symfony projekti je priporočljiva uporaba vtičnika Symfony2 Plugin. S tem vtičnikom razširimo podporo za ogrodje Symfony2 in za njegove najpopularnejše razširitve. Tako dobimo izboljšano podporo za delo s Symfony Forms, z orodjem za izdelavo predlog Twig, podporo za Yaml in konfiguracijske datoteke XML, Doctrine QueryBuilder itd.

3.4 PHPUnit

PHPUnit je napredno testno ogrodje za avtomatično testiranje enot za programski jezik PHP [19]. Ogrodje je instanca xUnit arhitekture, ki izvira iz testnega ogrodja enot za programski jezik Smalltalk SUnit.

Namen tega ogrodja je testiranje majhnih odsekov kode, čimprej kot je to mogoče. To nam omogoča hitro odkrivanje napak in posledično tudi učinkovito odpravljanje. Za preverjanje pravilnosti delovanja testov enot, PHPUnit uporablja trditve (angl. assertions).

Arhitektura xUnit sloni na sledečih komponentah:

- test runner (program, ki zaganja teste in posreduje poročilo o testiranju),
- test case (osnovni razred, iz katerega dedujejo vsi testi),
- test fixture (nabor pogojev, da se testi uspešno izvedejo),
- test suites (nabor testov, ki si delijo iste pogoje za izvedbo),
- test execution (metodi setup() in teardown(), ki poskrbita za testne pogoje),
- test result formatter (orodje, ki oblikuje rezultate testov v XML),
- assertions (funkcije, ki preverjajo obnašanje testa).

3.5 Behat

Behat je odprtokodno vedenjsko usmerjeno razvojno orodje za PHP [20]. Avtor Konstantin Kudryashov se je pri razvoju zgledoval po projektu Cucumber, ki je spisan v programskem jeziku Ruby. Behat omogoča testiranje aplikacij PHP z uporabo človeku prijazne sintakse, ki se imenuje Gherkin. Z njo napišemo funkcionalnosti in scenarije, ki opisujejo pričakovano obnašanje aplikacije.

Vsak korak v scenariju, ni nič drugega kot PHP funkcija, ki je opisana s ključno besedo, z regularnim izrazom in s funkcijo s povratnim klicem. Vsak izraz v scenariju bo preslikan v posamezen korak in funkcija za ta korak definira, kakšen je pričakovan rezultat. Sam Behat nam ponuja velik nabor preddefiniranih korakov, če pa želimo dodati lastne, moramo ustvariti razred FeatureContext, ki deduje od razreda BehatContext.

3.6 Mink

Ena najpomembnejših stvari na spletu je brskalnik. Brskalnik je aplikacija, preko katere uporabnik komunicira s svetovnim spletom. Torej za testiranje naše spletne aplikacije moramo prenesti uporabnikove akcije v korake scenarija.

Za uspešno poganjanje takih testov moramo simulirati brskalnik. Korak v scenariju bo sedaj lahko simuliral uporabnika in emulator brskalnika, preko katerega uporabnik uporablja spletno aplikacijo. Poznamo dva tipa emulatorjev:

- Headless browser emulators (brskalnik, ki deluje preko konzole in omogoča le poizvedbe HTTP in emulira brskalnik na visokem nivoju) - PhantomJS [16].
- In browser emulators (emulator, ki deluje s pravimi brskalniki). Tako lahko pripravimo realno okolje z brskalnikom, ki nima omejenih funkcionalnosti. Zato lahko delamo s CSS, javascriptom in klici AJAX. Za to funkcionalnost potrebujemo posebne gonilnike, kot sta na primer Selenium2 ali Sahi, ki znata s pomočjo PhantomJS simulirati popolno uporabniško izkušnjo brskalnika.

Ker za hitro in uspešno testiranje potrebujemo oba tipa brskalnika, nam je tu v pomoč Mink [15]. Mink je abstraktna plast emulatorja brskalnika, ki skrije razlike

brskalnikov za enoten spletni programski vmesnik (angl. Application Programming Interface ali API).

3.7 Razširitve Symfony2

Kakor sem že omenil, je Symfony2 popolnoma modularno ogrodje. Za lažje delo bom predstavil nekaj modulov (angl. bundles), ki so nam v pomoč pri testiranju.

3.7.1 FriendlyContexts

Modul FriendlyContexts [17] podjetja KnpLabs nam obogati nabor korakov za delo z Behatom. Tako nam lahko prihrani veliko časa, ki bi ga potrebovali za izdelavo lastnih. Poleg tega pa so to že optimizirane rešitve. Stavki se delijo v skupine:

- za testiranje enot,
- za testiranje tabelarične strukture,
- za nove funkcionalnosti za delo z brskalnikom,
- za preverjanje strukture trenutnega naslova URL,
- za testiranje klicev API ,
- za integracijo razširitve alice.

3.7.2 nelmio/alice

Zelo pomemben aspekt avtomatskega testiranja so kvalitetni testni podatki (angl. fixtures). Če testiramo nad pomanjkljivim naborom podatkov, lahko spregledamo različne nepričakovane napake. Napake, kot so performančne težave in vizualne napake, zaradi manjkajočih ali neprimernih vsebin, ki bi se potencialno pojavile, ko bi se naša aplikacija začela uporabljati v produkcijskem okolju.

Najboljši način za reševanje teh težav, je priprava res kvalitetnih testnih podatkov, kar pa lahko vzame veliko časa. Zato se to opravilo pogosto odlaša in posledično lahko testi ne dajejo optimalnih rezultatov.

Tu nam je potem v veliko pomoč modul alice [21]. Omogoča nam preprosto kreiranje testnih podatkov v datotekah yaml. Podatki se morajo sklicevati na objekte v naši aplikaciji, alice pa omogoča, da med njimi lahko delamo relacije. Doctrine in FriendlyContexts pa poskrbita, da se ti podatki zapišejo v podatkovno bazo. Tako lahko sedaj po potrebi pred vsakim testnim scenarijem poskrbimo za svež nabor podatkov. Primer uporabe modula na portalu www.rockpamperscissors.co.uk:

```
RPS\ServiceBundle\Entity\Salon\Stylist:
    stylist-louise:
        id: 1
        fullName: Louise
        slug: louise
        summary: null
        displayTitle: null
        status: active
    stylist-frankie:
        id: 2
        fullName: Frankie
        slug: frankie
        summary: null
        displayTitle: null
        status: active
RPS\ServiceBundle\Entity\Salon\Seniority:
    seniority1:
        seniority: Senior stylist
        id: 1
        stylists: [@stylist-louise]
    seniority2:
        seniority: Stylist
        id: 2
        stylists: [@stylist-frankie]
RPS\ServiceBundle\Entity\Salon:
    architect-hair:
        id: 38
        name: 'Architect␣Hair'
        slug: architect-hair
        strapline: '<sentence($nbWords␣=␣6)>'
        description: '<paragraph($nbSentences␣=␣3)>'
        aboutinfo: '<sentence($nbWords␣=␣6)>'
        postcode: 'LS6␣2AL'
        town: Leeds
        street_address: '52␣Otley␣Road'
        street_address_two: ''
```

```
locality: Headingley
location_lat: 53.823427
location_lng: -1.57972
status: published
stylists: [@stylist-louise, @stylist-frankie]
seniorities: [@seniority1, @seniority2]
hotslotsenabled: true
hotslotsincrement: 15
```

Zgornji primer predstavlja zaključeno celoto entitet, ki so med sabo odvisne. Različne entitete so lahko razdeljene po različnih .yml datotekah lahko pa jih združimo v eno samo. Pri relacijah med entitetami je pomembno, da mora biti entiteta vedno najprej definirana, šele potem se lahko sklicujemo nanjo. Zato sem v tem primeru najprej definiral stilista, potem njuni delovni mesti, ker se sklicujeta na stilista, in šele na koncu salon, ki ima relacije na obe entiteti. Če pa imamo podatke razdeljene med več datotek, pa moramo biti pozorni, kako jih navedemo pri uporabi.

3.7.3 fzaninotto/Faker

Faker [22] je knjižnica, ki za nas generira in oblikuje lažne podatke. Knjižnica je narejena po zgledu podobnih knjižnic za Perl in Ruby. Lahko jo uporabi pri avtomatskem testiranju spletnih obrazcev, za generiranje vsebin z modulom alice ali pa če potrebujemo bazo z veliko količino podatkov za obremenitvene teste.

Knjižnica lahko generira sledeče sklope podatkov:

- števila,
- tekst različne dolžine,
- datume,
- imena in priimke oseb,
- hišne naslove,
- telefonske številke,

- elektronske naslove, spletne naslove, številke IP ipd.
- podatke za kreditne kartice ...

Poglavje 4

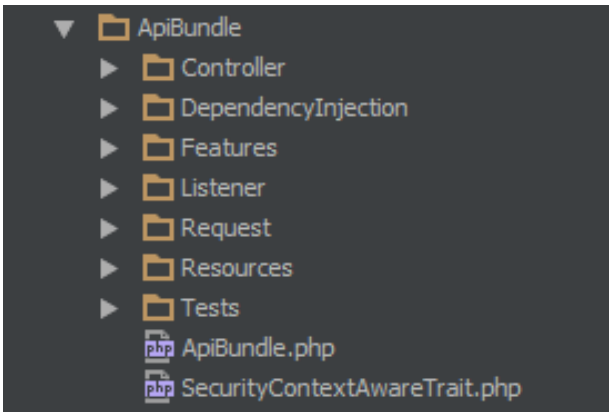
Testiranje v praksi

4.1 Predstavitev spletnega portala rockpamperscissors.co.uk

Spletni portal www.rockpamperscissors.co.uk (v nadaljevanju RPS) je spletni rezervacijski sistem za frizerske in lepotne salone. Portal izdelujemo za zagonsko podjetje (angl. startup) RPSMedia iz Velike Britanije.

Razvoj portala se je začel junija 2013, v uporabo pa je bil predan v novembru istega leta. Ob objavi je imel portal predstavitvene strani za salone, kjer se je uporabnik lahko seznanil z njihovo ponudbo, rezervacijskim sistemom in iskalnikom po salonih. Poleg tega pa so bile tu še uporabniške strani in strani za administracijo. Do danes pa smo portal obogatili še z možnostjo ocenjevanja opravljenih storitev, dodali možnost izbire različnih popustov (študenti, prvič obiskovalec salona, dnevni popusti itd.), dodali smo predstavitvene strani za stiliste, razvili API za mobilno aplikacijo, možnost spremebe termina preko SMS sporočil, preoblikovali rezervacijski sistem, ki sedaj podpira rezervacijo več storitev hkrati itd.

Naročnik je za začetni trg portala izbral mesto Leeds v Veliki Britaniji, do danes pa se je razširil še v Manchester. Začeli so z majhno bazo salonov, ki so se jim lahko posvetili in tako razvili funkcionalnosti, ki jih saloni in tudi končni uporabniki dejansko potrebujejo. Tak pristop se je izkazal za pravilnega, kar dokazuje iz meseca v mesec naraščujoče število opravljenih rezervacij.



Slika 4.1: Struktura ApiBundle modula

4.1.1 Struktura portala

Portal sledi smernicam razvoja, ki jih narekuje ogrodje Symfony2. Programska koda je razdeljena v več modulov.

Prvi je ServiceBundle, ki vsebuje večino poslovne logike, ki jo razdelimo v različne storitve (angl. services). Potem sledi DataBundle, ki vsebuje migracijske datoteke za podatkovno bazo ter nabor testnih podatkov. Migracijske datoteke vsebujejo stavke SQL, ki se morajo izvesti ob nadgranji aplikacije. Vsebujejo tudi stavke SQL za povrnitev v prvotno stanje. ApiBundle vsebuje vstopne točke za mobilno aplikacijo. Potem pa imamo še predstavitvene module. To so WebBundle, AdminBundle in UserBundle, ki vsak vsebuje svoje vstopne točke, datoteke s stili, kodo javascript, predloge Twig itd. Moduli imajo podobno strukturo, tako so tudi testi razdeljeni po modulih, kar nam prikazuje slika 4.1. Tako ima vsak modul svojo mapo Tests za teste PHPUnit in mapo Features za teste Behat.

Pri delu uporabljamo veliko odprtokodnih rešitev, ki nam pri razvoju prihranijo veliko časa. Pri razvoju poizkušamo čim bolj slediti dobrim praksam, ki jih narekujejo tehnologije, ki jih uporabljamo.

Osnovni podatki o infrastrukturi:

- za gostovanje portala uporabljamo Amazonovo oblačno storitev Amazon Web Services ali AWS [24], saj nam omogoča fleksibilnost in dobro skalabilnost,

- portal teče na strežniku Ubuntu Server [25] s spletnim strežnikom Nginx [26],
- uporabljamo podatkovno bazo PostgreSQL [27],
- repozitorij izvorne kode GitHub [28],
- portal razvijamo v programskem jeziku PHP na ogrodju Symfony2.

Razvoj programske opreme poteka po principu agilne metodologije Scrum. Imamo štirinajst dni dolge razvojne iteracije. Za vodenje seznama zahtev produkta in seznama zahtev sprinta na projektu uporabljamo spletno aplikacijo JIRA podjetja Atlassian [29].

4.2 Ročno testiranje

Ročno testiranje je še vedno pomemben del razvoja programske opreme. Iz izkušenj na projektu, bi ga lahko razdelili v sledeče sklope:

- testiranje po končani funkcionalnosti, sploh če se zaradi okoliščin nismo odločili spisati testa enote,
- celovito testiranje nove funkcionalnosti in celotnega sistema s strani testne ekipe,
- testiranje izgleda uporabniškega vmesnika,
- testiranje naročnika pred prevzemom nove verzije aplikacije.

Po končanem razvoju nove funkcionalnosti, se je vedno smiselno prepričati z ročnim testom nove kode. Če pa se zaradi določenih razlogov nismo odločili za pristop TDD in nismo spisali testa, ki bi to preveril avtomatično, pa je ročno testiranje obvezno. Na odločitev, da preskočimo pisanje testa enote, lahko vpliva več dejavnikov:

- funkcionalnost je preprosta za izvedbo,
- funkcionalnost je relativno nepomembna,
- smo časovno omejeni in je prioriteta hitrost izvedbe ...

Izvedba takih testov je ponavadi mogoča kar preko uporabniškega vmesnika. Najbolje je, da si na podlagi uporabniške zgodbe pripravimo nekaj mejnih testni scenarijev in preprosto preizkusimo funkcionalnost v našem razvojnem okolju. Če pa testiramo neko funkcionalnost npr. za API, pa si lahko pomagamo z orodji, kot je Postman [18], kjer na posamezno vstopno točko pošljemo podatke in potem preverimo pravilnost odziva.

V razvojnih podjetjih obstaja praksa, da imajo organizirano svojo testno ekipo. Testno ekipo lahko sestavljajo inženirji, ki imajo dovolj tehničnega znanja, da lahko tudi sami kreirajo teste PHPUnit in Behat. Lahko pa so to samo napredni uporabniki, ki s svojimi izkušnjami znajo predvideti obnašanje uporabnikov v aplikacijah. Testerji morajo na podlagi uporabniških zgodb znati pretestirati določeno funkcionalnost. Osnovni pogoj je, da so te zgodbe dobro definirane in morajo vsebovati tudi mejne primere uporabe funkcionalnosti.

Posebno poglavje je testiranje izgleda aplikacije. To je še največji problem pri spletnih produktih. Po statistiki tržnega deleža spletnih brskalnikov [30] imamo štiri prevladujoče brskalnike:

- Chrome z 38 odstotnim tržnim deležem,
- Internet Explorer z 21,1 odstotnim tržnim deležem,
- Safari s 15,5 odstotnim tržnim deležem,
- Firefox s 15,5 odstotnim tržnim deležem.

Čeprav vsi stremijo k čimboljšem pokrivanju standardov, med njimi še vedno prihaja do razlik pri prikazovanju stilov CSS in interpretaciji javascript kode [31]. Še največ težav tu povzroča Internet Explorer, ki ima samosvoj pristop pri upoštevanju spletnih standardov. Čeprav se je z verzijama 10 in 11 to izboljšalo, je še veliko uporabnikov, predvsem v poslovnem okolju, ki uporabljajo vezijo 8. Razlog je v uporabi operacijskega sistema Microsoft Windows XP, ki ne dovoljuje nadgradnje tega brskalnika, in nezmožnosti namestitve alternativnih aplikacij zaradi internih omejitev in pravil. Poleg tega pa prihaja do razlik tudi znotraj istih brskalnikov na različnih operacijskih sistemih.

V zadnjih letih je v porastu uporaba spletnih strani in portalov tudi preko mobilnih naprav, kot so pametni telefoni in tablice. Za čimboljšo uporabniško

izkušnjo, se razijalci poslužujejo odzivnega spletnega designa (angl. Responsive web design ali RWD) [23], ki prilagodi izgled strani glede na velikost zaslona. Poleg tega pa je potrebna podpora tudi za naprave, ki se upravljajo z dotikom.

Posledično ima ekipa testerjev veliko dela pri testiranju izgleda uporabniškega vmesnika. Testirati je potrebno na različnih mobilnih napravah ko so: iPhone, iPad in mobilni telefoni in tablice z operacijskim sistemom Android. Poleg tega pa še vse naštete brskalnike na osebnih računalnikih, po možnosti na različnih operacijskih sistemih.

Pred objavo nove verzije aplikacije, testiranje ponavadi izvede še naročnik. Pri uporabi agilnih metodologij razvoja programske opreme, se zaradi kratkih iteracij to izvaja sproti in ko pride do objave, ni potrebno dodatno obsežno testiranje. Drugače pa je pri klasičnih metodah razvoja. Tu ima naročnik ponavadi prvi stik z aplikacijo šele po tem, ko je večino stvari že dokončanih. Tako se znajde pred obsežno in zapleteno nalogo testiranja aplikacije, ki mu je še popolnoma tuja.

4.3 Testi PHPUnit

Ogrodje PHPUnit spremeni ogrodje Symfony2 v močno testno okolje. Integracija obeh ogrodij je relativno preprosta, saj Symfony2 omogoča preprosto integracijo.

PHPUnit najlažje dodamo z orodjem Composer tako, da dodamo nov vnos v datoteko composer.json:

```
"phpunit/phpunit": "4.2.*".
```

Potem pa z ukazom “composer install” poskrbimo, da se ogrodje namesti v naše okolje.

Osnovna konfiguracija za PHPUnit, ki se nahaja v datoteki `phpunit.xml.dist` že omogoča izvajanje testov, če smo se držali standardne strukture in smo teste umestili v prave mape. Teste lahko združujemo v zbirke in jim lahko tudi določimo, katero verzijo PHP naj uporabi pri izvajanju:

```
<phpunit>
    <testsuite name="RPS␣Test␣Suite">
        <directory phpVersion="5.3.0" phpVersionOperator=">=">src/*/*Bundle/
            Tests</directory>
        <directory phpVersion="5.3.0" phpVersionOperator=">=">src/*/Bundle/*
            Bundle/Tests</directory>
        <directory phpVersion="5.3.0" phpVersionOperator=">=">src/RPS/*/*
            Bundle/Tests</directory>
    </testsuite>
</phpunit>
```

Teste lahko tudi združimo v več skupin. S pomočjo anotacije `@group`, ki jo nastavimo testu, lahko specificiramo, v katero skupino testov ta test spada.

```
<phpunit>
    <groups>
        <include>
            <group>api</group>
        </include>
        <exclude>
            <group>web</group>
        </exclude>
    </groups>
</phpunit>
```

Isti učinek dosežemo, če izvedemo klic:

```
    phpunit --group api --exclude-group web
```

Konfiguracija omogoča še:

- nastavitve strežnika Selenium, ki določijo, kateri brskalnik naj uporabi in na katerih vratih,
- aktiviranje izračuna pokritosti kode s testi in definiranje mape, nad katerimi naj se ti testi poganjajo
- nastavljanje nastavitev PHP INI, konstant in globalnih spremenljivk,
- shranjevanje rezultatov testov (angl. logging) ...

4.3.1 Primer testa PHPUnit

Naša naloga je, da razvijemo razred `PriceCalculator`, ki bo za salon zgeneriral nabor popustov, ki jih ponuja. Popusti se za vsak salon nastavijo v administracijiskem modulu portala, vendar je pomembno, da se uporabniku vedno pokažejo najbolj ugodne ponudbe.

Ker smo razred `PriceCalculator` umestili v modul `ServiceBundle`, moramo znotraj modula ustvariti mapo `Tests`, v kateri bo ogrodje PHPUnit poiskalo teste. Za lepšo strukturiranost lahko teste umestimo v različne podmape. Znotraj te strukture potem ustvarimo razred `PriceCalculatorTest`, ki deduje razred `PHPUnit_Framework_TestCase`.

Za lažje testiranje, testu najprej z anotacijo `@group` določimo skupino. Tako ga lahko sedaj izoliramo od preostalih testov in poženemo samostojno. Vsak test napišemo kot funkcijo, katere ime se mora začeti z besedo test. Drugače jih PHPUnit ne izvede. Zato ga poimenujemo `testDiscountedPriceOnlySalon`:

```
/** @group calc  */
public function testDiscountedPriceOnlySalon() {

    $salonService = $this->getContainer()->get("rps.service.salon");
    $salon = $salonService->loadSalon(["id" => 1]);

    $calculator = new PriceCalculator($salon);

    $prices = $calculator->getPrices();
```

```
    $this->assertEquals(true, $prices['First_time']->getDiscountAvailable())
        ;
    $this->assertEquals(true, $prices['First_time']->getDiscountVariable());
    $this->assertEquals(0.5,  $prices['First_time']->getDiscountAmount());
    $this->assertEquals(50,   $prices['First_time']->getDiscountAmount(true)
        );
}
```

V funkciji najprej pridobimo iz testne baze podatkov salon, za katerega bomo preverjali popuste. Sedaj kreiramo nov objekt razreda `PriceCalculator` in mu kot parameter podamo salon. Naloga funkcije `getPrices()` pa je, da glede na tip popusta izračuna nove cene za vse storitve in stiliste.

Sedaj prvič poženemo test z ukazom:

```
phpunit --group calc
```

Ker razreda `PriceCalculator` še nismo spisali, seveda pride do napake.

```
PHPUnit 4.2 by Sebastian Bergmann.

Configuration read from /var/www/projects/rps/repo/phpunit.xml.dist

PHP Fatal error:  Class 'RPS\ServiceBundle\Tests\Service\PriceCalculator'
    not found in /var/www/projects/rps/repo/src/RPS/ServiceBundle/Tests/
    Service/PriceCalculatorTest.php on line 45
```

Ko spišemo funkcionalnost in zopet poženemo test, je verjetnost, da rezultati ne bodo pravilni, velika. Primer poročila PHPUnit, ko pričakovani rezultat ni ustrezal realnim podatkom.

```
PHPUnit 4.2 by Sebastian Bergmann.

Configuration read from /var/www/projects/rps/repo/phpunit.xml.dist

F

Time: 27.68 seconds, Memory: 46.00Mb

There was 1 failure:

1) RPS\ServiceBundle\Tests\Service\PriceCalculatorTest::
    testDiscountedPriceOnlySalon
Failed asserting that '0.5' matches expected 0.6.

/var/www/projects/rps/repo/src/RPS/ServiceBundle/Tests/Service/
    PriceCalculatorTest.php:55

FAILURES!
Tests: 1, Assertions: 3, Failures: 1.
```

V tretji vrstici nam poročilo grafično predstavi, koliko testov se je izvedlo. Naš test se ni izvedel uspešno, zato se je izpisala črka F (angl. fail). Če test uspe, se izpiše pika. V naslednji vrstici vidimo informacijo o potrebnem času za izvedbo testov in koliko pomnilnika je bilo porabljenega. Veliko časa za izvedbo testa je bilo namenjeno predvsem pripravi testnih podatkov.

V peti vrstici izvemo, koliko testov se ni izvedlo pravilno. Nato sledi seznam vseh napak, ki so se zgodile. Opozoriti velja, da se prikaže vedno samo prva napaka znotraj testa, ker PHPUnit v tistem trenutku prekine izvajanje testa in preide na naslednjega. Dobimo pa informacijo, v kateri vrstici testa se je zgodila napaka, kakšna je bila pričakovana vrednost in kakšno nam je vrnila aplikacija.

Na koncu se izpiše povzetek. Najprej obvestilo, da je prišlo do napak. V zadnji vrstici pa vidimo še statistiko. Koliko testov smo izvajali, koliko preverjanje se je zgodilo in koliko testov se je izvedlo neuspešno.

Na koncu, ko odpravimo vse napake, ponovno preverimo, če se test sedaj izvede uspešno.

```
PHPUnit 4.2 by Sebastian Bergmann.

Configuration read from /var/www/projects/rps/repo/phpunit.xml.dist

.

Time: 28.17 seconds, Memory: 45.75Mb

OK (1 test, 4 assertions)
```

V tretji vrstici imamo namesto črke F sedaj samo piko. V zadnji vrstici pa sedaj prikaže vsa štiri preverjanja, ki jih izvaja naš test, ker so se vsa izvedla pravilno.

4.4 Testi Behat

Podobno kot PHPUnit modul, tudi modula Behat in Mink dodamo v ogrodje Symfony2 s pomočjo orodja Composer, tako da dopolnimo datoteko `composer.json` in izvedemo ukaz composer install:

```
    "behat/behat":              "3.*@dev",
    "behat/mink":               "1.6.*@dev",
    "behat/mink-extension":     "dev-master",
    "behat/mink-goutte-driver": "dev-master",
    "behat/mink-sahi-driver":   "*",
    "behat/symfony2-extension": "dev-master",
```

S temi moduli omogočimo pisanje testov BDD v ogrodju Symfony2.

Prva konfiguracijska datoteka `behat.yml` se nahaja na prvem nivoju mapne strukture Symfony2 projekta. V tej konfiguraciji nastavimo privzete vrednosti za delo s temi moduli.

Definiramo lahko zbirke testov:

```
suites:
    web:
        bundle: WebBundle
        type: symfony_bundle
        contexts:
            - Behat\MinkExtension\Context\MinkContext
            - Knp\FriendlyContexts\Context\AliceContext
            - Knp\FriendlyContexts\Context\EntityContext
            - RPS\Web\WebBundle\Features\Context\BookingContext
    api:
        bundle: ApiBundle
        type: symfony_bundle
        contexts:
            - Knp\FriendlyContexts\Context\AliceContext
            - RPS\Api\ApiBundle\Features\Context\FeatureContext
            - Knp\FriendlyContexts\Context\EntityContext
```

Primer demonstrira dve različni zbirki testov, web in api. Za vsako moramo definirati modul, v katerem se nahaja, tip in zbirke definicij korakov, ki jih bomo uporabili pri testiranju. Pri zbirki web smo uporabili zbirko korakov Mink, pri obeh pa smo definirali tudi lastni zbirki.

Nastaviti je potrebno tudi dodatne module, ki delujejo s pomočjo Behata:

```
extensions:
    Behat\MinkExtension:
        sessions:
            default:
                symfony2: ~
            javascript:
                sahi: ~
        browser_name: phantomjs
        show_auto: false
        base_url: http://test.rps.lan/
    Behat\Symfony2Extension: ~
    Knp\FriendlyContexts\Extension:
        alice:
            fixtures:
                Users: src/RPS/DataBundle/DataFixtures/users.yml
                Stylists: src/RPS/DataBundle/DataFixtures/stylists.yml
```

```
            Salons: src/RPS/DataBundle/DataFixtures/salon.yml
        dependencies: ~
```

Tu smo predvideli uporabo Sahi gonilnika za javascript, definirali osnovni spletni naslov za teste in omogočili kreiranje testnih podatkov s pomočjo alicea.

Na projektu RPS smo potem definirali še dodatno konfiguracijsko datoteko `behat.yml`, ki se nahaja v mapi `app/config`. Ta vsebuje bolj podrobne nastavitve za vsak nabor testov. Primer za web nabor:

```
web:
    filters:
        tags: "@booking"
    extensions:
        Behat\Symfony2Extension\Extension:
            bundle:      WebBundle
            mink_driver: true
            kernel:
                env: test
                debug: true
        Behat\MinkExtension\Extension:
            browser_name:    phantomjs
            default_session: symfony2
            javascript_session: sahi
            base_url:        http://test.rps.lan/
            sahi:
                wd_host: "http://127.0.0.1:4444/wd/hub"
                capabilities: {"browser": "phantomjs", "version": "*", "
                    acceptSslCerts": true}
```

Definirali smo filter, da se bodo izvedli samo testi z anotacijo @booking. Omogočili smo razširitev Mink in nastavili vse potrebno za delovanje gonilnika Sahi.

4.4.1 Testiranje uporabniškega vmesnika

Testi uporabniškega vmesnika so tipični primeri testov črne škatle. Tu nas ne zanima potek notranjih procesov, ampak je poudarek na dejanski uporabi apli-

kacije. Torej nas zanima le to, da za nabor vhodnih podatkov dobimo pravilne rezultate. V spodnjem primeru bomo preučili primer testa, ki z uporabo Minka testira delovni proces (angl. workflow) rezervacije termina (angl. booking process) v salonu.

V datoteki `booking.feature` imamo nabor testov, ki testirajo proces rezervacije termina. Za lepo strukturo modula je priporočljivo, da se datoteke s testi nahajajo znotraj modula, ki ga testirajo, v mapi `Features`. Ker je proces rezervacije termina dokaj zapleten proces in ima tri vstopne točke (kako začnemo proces), smo spisali pet različnih testov. To so definirani scenariji:

Scenario: Booking a service without knowing what service and stylist I want,

Scenario: Booking a service by choosing a stylist,

Scenario: Booking a service by choosing a service first,

Scenario: Booking a service by changing an already chosen service,

Scenario: Booking a service by changing an already chosen service and stylist.

Nabor testov, ki pokrivajo določen sklop, po definiciji BDD vedno začnemo z opisom funkcionalnosti, ki sloni na uporabniški zgodbi. Naša uporabniška zgodba je, da kot uporabnik lahko izberem salon, kjer želim opraviti rezervacijo storitve. Primer:

```
Feature: Booking process
In order to book a service
As a user
I need to be able to choose a salon I want to book the service at
```

Potem sledijo scenariji. Tu bomo predstavili scenarij, ko pridemo na začetek booking procesa brez predizbranega stilista ali storitve. Pred scenarijem specificiramo potrebne anotacije. Ker rezervacijski proces uporablja veliko javascript kode in AJAX klicev, potrebujem orodje Mink in z anotacijo `@javascript` omogočimo, da se rezervacijski proces pravilno izvede. Nato pa sledi opis samega scenarija.

Prvi korak: Give I am on "/salon/shrine-salon-spa-headingley/booking"pove Behatu, naj naloži vsebino spletne strani na podanem naslovu. Vsebina znotraj narekovajev, se poda funkciji kot parameter. V naslednjem koraku, pa želimo preveriti vsebino strani. Kakor nam že opis pove, želimo naslove vseh glavnih sklopov na strani. Ker je to dokaj specifična zahteva, je bilo treba spisati funcijo, ki bo znala to preveriti. Zato v mapi `Features/Context` definiramo svoj razred `BookingSubContext`, ki razširja BehatContext. Tu notri potem napišemo definicije korakov, ki jih potrebujemo za lastne teste.

Drugi korak:

```
Then I should see step headlines:
| OK, LET'S BOOK AN APPOINTMENT WITH THE BAND AT SHRINE SALON AND SPA |
| CHOOSE FROM THE PLAYLIST |
| WHO'S GOING TO BE PERFORMING? |
| WHEN CAN YOU MAKE IT? PICK A DATE & A TIME SLOT. |
```

Najprej sledi opis koraka. Ostale vrstice pa predstavljajo tabelarično strukturo podatkov, ki se funkciji poda kot parameter tipa `TableNode`. V tem primeru imamo enodimenzionalno tabelo s štirimi elementi.

Ko se izvaja korak, se s pomočjo regularnih izrazov poišče prava definicija koraka. Potem se izvede logika. V tem primeru s strani preberemo vsebino posameznih elementov, ki jih pridobimo s pomočjo selektorja CSS. S klicem lokalne funkcije `assertTexts()` preverimo vsebino istoležečih elementov in če se popolnoma ujemajo, se ta korak v testu uspešno izvede.

Definicija drugega koraka:

```
/**
* @Then /^I should see step headlines:$/
*/
public function iShouldSeeStepHeadlines(TableNode $table) {
$elements = $this->getSession()->getPage()->findAll("css", self::
    STEP_HEADLINE);
$this->assertTexts($table, $elements);
```

```
}
```

Tako se izvedejo vsi koraki do konca. In če se izvedejo uspešno, pomeni, da je tudi scenarij zaključen uspešno. Iz priloženega testa je vidno, da BDD zelo pomaga pri čitljivosti in razumenju. Pri tem so zelo v pomoč predlogi Given, Then, When in And, ki so namenjeni zgolj za lažje razumevanje [32].

Primer celotnega testa:

```
@javascript
Scenario: Booking a service without knowing what service and stylist I want
Given I am on "/salon/shrine-salon-spa-headingley/booking"
Then I should see step headlines:
| OK, LET'S BOOK AN APPOINTMENT WITH THE BAND AT SHRINE SALON AND SPA |
| CHOOSE FROM THE PLAYLIST |
| WHO'S GOING TO BE PERFORMING? |
| WHEN CAN YOU MAKE IT? PICK A DATE & A TIME SLOT. |
And I should see main categories:
| HAIR |
And I should see secondary categories:
| TREATMENTS |
And I should see 113 services available
Then I choose "Mens␣cut" service
And I should see step headlines:
| OK, LET'S BOOK AN APPOINTMENT WITH THE BAND AT SHRINE SALON AND SPA |
| FOR MENS CUT |
| WHO'S GOING TO BE PERFORMING? |
| WHEN CAN YOU MAKE IT? PICK A DATE & A TIME SLOT. |
And I should see stylists:
| MONIX |
| DEBBIE |
| LAUREN |
| DEANA |
And I choose stylist "Debbie"
Then I should see step headlines:
| OK, LET'S BOOK AN APPOINTMENT WITH THE BAND AT SHRINE SALON AND SPA |
| FOR MENS CUT |
| WITH DEBBIE FOR THE TOTAL 32 |
| WHEN CAN YOU MAKE IT? PICK A DATE & A TIME SLOT. |
And I choose available day
And I should see the available day in "#book-step4␣.booking-row-elements␣.h2
```

```
    " element
And I fill in "requirements" with "Latex␣alergy"
And I press "proceed"
And I should be on "/salon/shrine-salon-spa-headingley/booking/info"
Then I fill in new user data:
| Test User |
| test.user@dlabs.si |
| test.user@dlabs.si |
| 1234567890 |
And I submit new user data form
And I should be on "/salon/shrine-salon-spa-headingley/booking/confirm"
And booking should be saved identified by "email" with value "test.
    user@dlabs.si",
 holding data:
| rel_salon.name | Shrine Salon and Spa |
| rel_service.name | Mens cut |
| rel_stylist.fullName | Debbie |
| price | 32.00 |
| healthConcerns | Latex alergy |
| fullname | Test User |
| email | test.user@dlabs.si |
| phonenumber | 1234567890 |
```

Problem s testi uporabniškega vmesnika je, da nam lahko vzamejo kar nekaj časa, da jih napišemo. Dobro je tudi, da pokrijemo čim več mogočih scenarijev. Poleg pozitivnih, tudi negativne, da preverimo, če validacija deluje pravilno. Vendar posledično lahko količina testov preseže neko razumno mejo. Problemi se pojavljajo tudi, če se zgodijo spremembe v strukturi HTML strani. V primeru zgornjega testa, pa je dovolj že sprememba vsebine.

Pri razvoju portala RPS, smo se pogosto srečevali s temi težavami. Od manjših slogovnih popravkov in popravkov v besedilu, do večjega preoblikovanja procesa, kar je pomenilo večkratno popravljanje testov. To sčasoma postane moteče, saj so ravno taki rutinski popravki za razvojnike ponavadi najbolj nehvaležni.

Taki avtomatizirani testi uporabniškega vmesnika sicer preverjajo strukturo spletne strani in preverijo tudi sam delovni proces, kar močno pripomore k zanesljivosti in kvaliteti aplikacije. Še vedno pa ni načina, da bi lahko avtomatično preverili pravilni izgled strani. Zato je ključno, da izvedemo tudi temeljito ročno testiranje aplikacije.

4.4.2 Testiranje aplikacijskega vmesnika

Za potrebe mobilne aplikacije ROCKit [33] smo pripravili API. Mobilna aplikacija se razvija za Applovo platformo iOS in je namenjena za lastnike salonov in stiliste. Torej za zaprt krog uporabnikov.

Za programski vmesnik smo postavili nov Symfony2 modul ApiBundle, kamor smo postavili vso potrebno logiko. Za pisanje testov smo uporabili uporabniške zgodbe, ki jih je pripravil naročnik.

Napisali smo dva testa za potrjevanje opravljenih rezervacij. Prvega pozitivnega in drugega negativnega, kjer je čas rezervacije že v preteklosti.

```
@reset-schema @api @accept @alice(Users) @alice(Stylists) @alice(Salons)
    @alice(Galleries) @alice(Images) @alice(Bookings) @alice(Services)
    @alice(Categories)
  Scenario: Accept the appointment, all good.
    Given I specified the following request headers:
      | X-ACCESS-TOKEN | c3646f0aa91caasdoij9dde5884d3e1 |
    Given I prepare a POST request on "/api/v1/salons/architect-hair/
        appointments/7/accept/"
    When I send the request
    Then I should receive a 200 response

  @api @accept
  Scenario: Accept the appointment, appointment is in the past.
    Given I specified the following request headers:
      | X-ACCESS-TOKEN | c3646f0aa91caasdoij9dde5884d3e1 |
    Given I prepare a POST request on "/api/v1/salons/architect-hair/
        appointments/3/accept/"
    When I send the request
    Then I should receive a 400 response
```

Pri prvem scenariju smo uporabili anotacijo `@reset-schema`, ki počisti testno bazo podatkov. Z anotacijami `@alice()` pa smo poskrbeli, da smo za oba testa pripravili svež nabor testnih podatkov. Pri obeh smo dodali anotacijo `@api`, da smo jih združili v skupni nabor testov in `@accept`, ki nam omogoča filtriranje testov. Pri drugem nam ni bilo potrebno počistiti baze, ker vemo, da prvi test ne bo vplival na nabor podatkov, ki jih potrebujemo za izvedbo drugega testa. S tem

smo prihranili približno dvajset sekund pri izvajanju testov.

Teste zaženemo z ukazom:

```
php ./bin/behat --suite api --tags accept
```

S tem ukazom povemo, da naj se izvede nabor testov api in naj jih sfiltrira glede na anotacijo `@accept`. Lahko bi zagnali Behat samo s parametrom `--tags`, vendar potem rabi dlje časa, da najde prava testa.

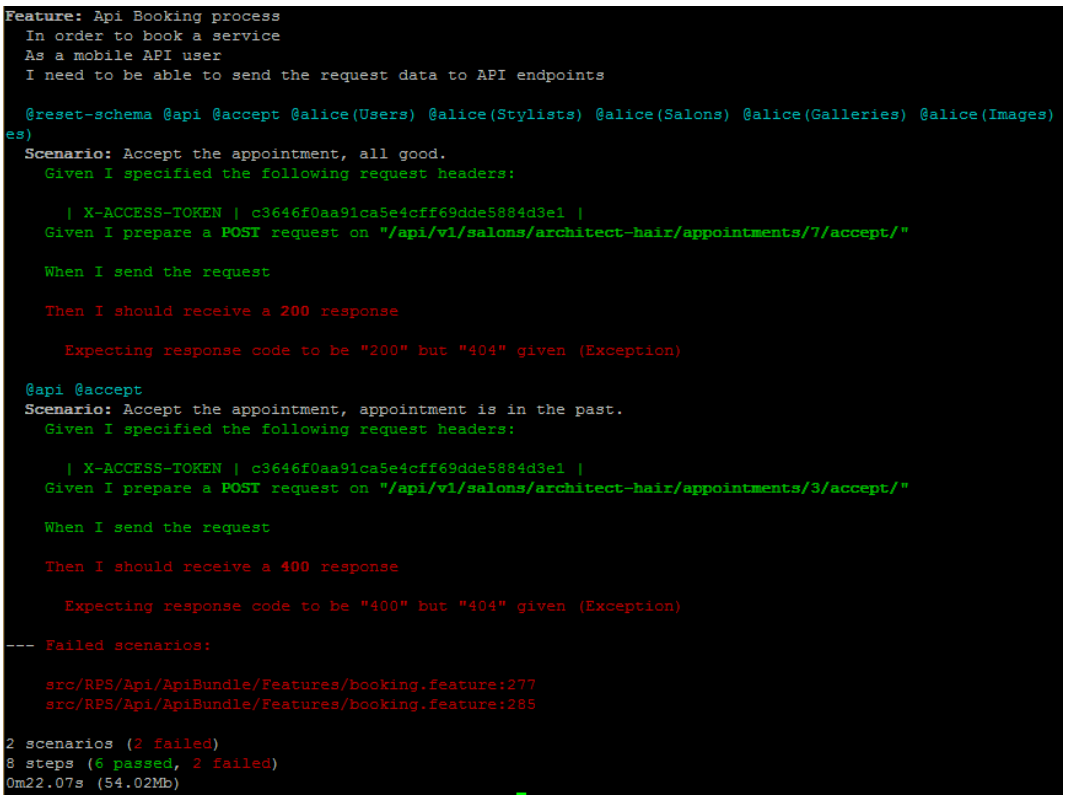
Na sliki 4.2 lahko vidimo rezultat prvega zagona. Oba vrneta odgovor 404, kar pomeni, da ta vstopna točka ne obstaja. Kar je povsem pravilno, saj se še nismo lotili realizacije.

Znotraj modula ApiBundle v mapo Controller kreiramo nov razred AppointmentController. Notri napišemo akcijo, ki bo veljala kot vstopna točka za naša testa:

```
/**
 * @Route("/v1/salons/{salon}/appointments/{appointment}/accept/", name="
     rps_api_salons_appointment_accept")
 * @Method({"POST"})
 */
public function acceptAppointment(Request $request, Salon $salon, Booking
    $appointment) {

    /** @var $bookingManager \RPS\ServiceBundle\Manager\BookingManager */
    $bookingManager = $this->get("rps.manager.booking");

    try {
        $bookingManager->acceptBooking($appointment);
    } catch(\Exception $e) {
        $this->error("Appointment can't be accepted!", 400);
    }
    return $this->success(new \stdClass(), 200);
}
```



Slika 4.2: Prikaz rezultatov neuspešne izvedbe testov Behat

```
Feature: Api Booking process
  In order to book a service
  As a mobile API user
  I need to be able to send the request data to API endpoints

  @reset-schema @api @accept @alice(Users) @alice(Stylists) @alice(Salons) @alice(Galleries) @alice(Images)
  Scenario: Accept the appointment, all good.
    Given I specified the following request headers:
      | X-ACCESS-TOKEN | c3646f0aa91ca5e4cff69dde5884d3e1 |
    Given I prepare a POST request on "/api/v1/salons/architect-hair/appointments/7/accept/"
    When I send the request
    Then I should receive a 200 response

  @api @accept
  Scenario: Accept the appointment, appointment is in the past.
    Given I specified the following request headers:
      | X-ACCESS-TOKEN | c3646f0aa91ca5e4cff69dde5884d3e1 |
    Given I prepare a POST request on "/api/v1/salons/architect-hair/appointments/3/accept/"
    When I send the request
    Then I should receive a 400 response

2 scenarios (2 passed)
8 steps (8 passed)
0m21.66s (44.03Mb)
```

Slika 4.3: Prikaz rezultatov uspešne izvedbe testov Behat

Funkcija mora v dokumentaciji vsebovati anotacijo `@Route`, ki pove ogrodju Symfony2, na katerem naslovu pričakuje klice in anotacijo `@Method`, ki opredeli tip zahtevka. Ko dokončamo programiranje te funkcionalnosti, ponovno poženemo test in na sliki 4.3 vidimo, da sta se testa uspešno izvedla.

Poglavje 5

Testi in sprotna integracija

5.1 Sprotna integracija

Sprotna integracija (angl. Continuous Integration ali CI) [34] je praksa pri razvoju programske opreme, ki zahteva od članov razvojne ekipe, da svojo izvorno kodo večkrat dnevno poenotijo s skupnim repozitorijem. Glavni namen CI je preprečiti težave z integracijo, če je le ta preredka

Prvotno je bil CI mišljen v kombinaciji z avtomatičnim testiranjem enot, napisanih po tehniki razvoja programske opreme TDD. Razvojnik je na svojem razvojnem okolju pognal avtomatične teste in šele, ko so vsi uspešno prestali testiranje, je spremembe potisnil na glavno razvojno vejo.

Preden se razvojnik loti dela na novi funkcionalnosti, naredi novo razvojno vejo v repozitoriju. Med razvojem drugi člani ekipe integrirajo svoje funkcionalnosti v glavno razvojno vejo. Tako prihaja do večjih razlik v kodi med razvojnikovo vejo in glavno vejo. Poleg kode, lahko glavna veja dobi tudi nove module, ki ustvarjajo nove odvisnosti med njimi, kar lahko potencialno pripelje do težav. In daljše kot je obdobje neskladja, večja je verjetnost, da se bodo ob integraciji pojavile težave. Zato je priporočljivo, da razvojnik svojo vejo, kjer razvija novo funkcionalnost, vsak dan uskladi z glavno razvojno vejo. Zato CI spodbuja zgodnjo sprotno integracijo. Tako se lahko ognemo predelavam in prihranimo čas in sredstva na projektu.

Dopolnilna praksa pri CI je, da razvojnik pred integracijo svojih sprememb na

glavno vejo, lokalno zgradi postavitev in požene teste.

5.2 Sprotna dostava

Sprotna dostava (angl. Continuous delivery ali CD) [35] je praksa, ki avtomatizira in nadgrajuje proces objavljanja programske opreme. Tehnike kot so avtomatično testiranje, CI in CD pripomorejo k višjim standardom programske opreme, ki jo lahko hitro objavimo na testnem ali celo produkcijskem okolju. To nam omogoča, da lahko zanesljivo in pogosto objavljamo posodobitve in popravke stranki z minimalno količino tveganja.

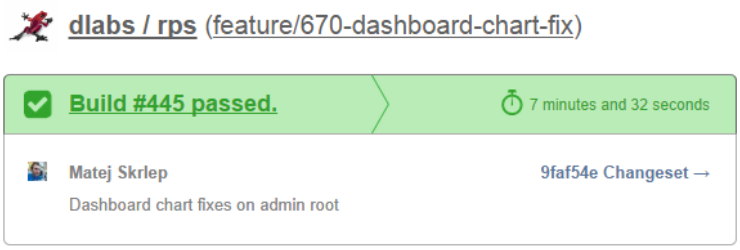
5.3 Travis CI

Travis CI [36] je spletna storitev, ki omogoča CI in CD. Obstajajo tudi druge podobne rešitve, kot so:

- Jankins CI,
- CircleCI,
- Codeship ...

Storitev Travis CI povežemo z repozitorijem na GitHubu. Travis CI dobi obvestilo, da je v našem repozitoriju prišlo do posodobitve izvorne kode. Potem pridobi izvorno kodo in zažene skripto za izgradnjo postavitve. Po končani izgradnji pa zažene še avtomatske teste. Na spletni strani storitve lahko v realnem času preverjamo stanje postavitev in vidimo, kje se je zgodila napaka. Na voljo imamo tudi zgodovino vseh postavitev, ki jih lahko tudi ponovno sprožimo. Travis CI omogoča tudi obveščanje preko elektronske pošte, ko se postopek graditve konča in testi izvedejo, kar lahko vidimo na sliki 5.1.

Izvajanje avtomatskih testov na Travis CI lahko precej zavleče objavo posodobitev na izbranem okolju. To lahko postane moteče, predvsem če želimo hitro objaviti pomemben popravek na produkcijskem okolju. Na projektu RPS trenutno cel proces postavitve in testiranja traja blizu desetih minut. Iz meritev smo ugotovili, da ima pri tem velik delež zagotavljanje primernih testnih podatkov. Zato



Slika 5.1: Prikaz Travis CI poročila preko elektronske pošte

je priporočljivo, da neodvisne teste in teste, ki ne posegajo v podatke, poganjamo nad obstoječim naborom podatkov, kar pomeni, da baze ne čistimo ob vsakem testu.

Na projektu RPS se vse posodobitve izvorne kode testirajo na Travis CI. Za nove funkcionalnosti vedno kreiramo novo vejo, ki jo potem vodja razvoja po opravljenem pregledu kode (angl. code review), integrira v glavno razvojno vejo. Tako na večih nivojih skrbimo za kvaliteto naše kode in posledično našega portala. Ker na glavno razvojno vejo integriramo samo dokončane funkcionalnosti, imamo omogočen tudi CD. CD poskrbi, da se zažene postopek objave na okolju peskovnik (angl. sandbox), kjer potem naročnik lahko hitro preizkusi posodobitve in poda komentarje na opravljeno delo.

Poglavje 6

Zaključek

V diplomskem delu smo preverili prednosti razvoja programske opreme po priljubljani agilni metodologiji Scrum. Ogledali smo si prednosti testno vodenega razvoja programske opreme v teoriji in praksi. Posvetili smo se tudi nadgradnji tega pristopa, ki se imenuje vedenjsko voden razvoj, ki se je razvil iz potrebe po boljšem razumevanju in lepši strukturi testov.

Agilna metodologija Scrum se je na tem projektu izkazala za zelo primerno. Omogočila je naročniku relativno hitro objavo portala, ki je že vseboval osnovne funkcionalnosti. Na tej osnovi smo potem načrtovali nadaljno smer razvoja portala RPS in dodajali funkcionalnosti, po katerih so spraševali uporabniki.

V začetnem obdobju testno vodenega razvoja, smo se srečevali s precejšnjimi težavami pri vpeljevanju le-tega. Predvsem, kaj moramo testirati in kje določiti mejo, kdaj je dovolj testov. Pri samem pisanju testov se je pojavljal še problem s kvalitetnimi testnimi podatki. Dogajalo se nam je, da smo z novimi testi potrebovali tudi nove podatke, ki so potem vplivali na že obstoječe teste. Zato je bilo potrebno nekajkrat popravljati večjo količino testov. Tu nam je zelo prav prišla knjižnica alice, ki omogoča pregledno pisanje testnih podatkov.

Vendar so se pozitivni učinki takega pristopa kmalu začeli kazati. Izboljšala se je struktura kode, saj smo bili pri pisanju testov prisiljeni temeljito premisliti, kakšen pristop moramo izbrali pri izvedbi. Pogosto se je v to vključila celotna razvojna ekipa, kar je pripomoglo k boljšim odločitvam in posledično k boljši kvaliteti kode. Pa tudi poznavnje obsežnega portala se je izboljšala pri vseh razvojnikih, kar je prinesla predvsem pogosta komunikacija.

Vpeljava avtomatičnih testov se je že večkrat izkazala za zelo koristno, saj je preprečila objavljanje nepopolne kode. Ena od slabosti pa je dolgo izvajanje teh testov. Pogosto se zgodi, da se razvojnik loti dela na drugi nalogi, potem pa se mora vračati. Tu imamo še nekaj prostora za optimizacijo, predvsem pri tem, kolikokrat moramo zagotoviti nov nabor testnih podatkov, saj to opravilo traja predolgo.

Literatura

[1] Laurie Williams, "Agile Software Development Methodologies and Practices", Elsevier Inc., 2010

[2] Ken Schwaber, Jeff Sutherland, "Scrum Guide™", *Scrum.org*, 2013. Dostopno na: `https://www.scrum.org/Portals/0/Documents/Scrum%20Guides/2013/Scrum-Guide.pdf`

[3] Symfony2, sept. 2014. Dostopno na: `http://symfony.com/`

[4] Manifest agilnega razvoja programske opreme, sept. 2014. Dostopno na: `http://agilemanifesto.org/iso/sl/`

[5] Wikipedia, Unit testing, sept. 2014. Dostopno na: `http://en.wikipedia.org/wiki/Unit_testing`

[6] Wikipedia, Integration testing, sept. 2014. Dostopno na: `http://en.wikipedia.org/wiki/Integration_testing`

[7] Wikipedia, Manual testing, sept. 2014. Dostopno na: `http://en.wikipedia.org/wiki/Manual_testing`

[8] Wikipedia, Functional testing, sept. 2014. Dostopno na: `http://en.wikipedia.org/wiki/Functional_testing`

[9] Wikipedia, Regression testing, sept. 2014. Dostopno na: `http://en.wikipedia.org/wiki/Regression_testing`

[10] Wikipedia, Acceptance testing, sept. 2014. Dostopno na: `http://en.wikipedia.org/wiki/Acceptance_testing`

[11] Wikipedia, Test-driven development, sept. 2014. Dostopno na: http://en.wikipedia.org/wiki/Test-driven_development

[12] Dan North & Associates, Introducing BDD, sept. 2014. Dostopno na: http://dannorth.net/introducing-bdd/

[13] Composer, sept. 2014. Dostopno na: https://getcomposer.org/

[14] JetBrains PhpStorm, PHP IDE that evolves with you, sept. 2014. Dostopno na: http://www.jetbrains.com/phpstorm/

[15] Mink - Behat, sept. 2014. Dostopno na: https://github.com/Behat/en-mink.behat.org/blob/master/index.rst

[16] PhantomJS, Full web stack, sept. 2014. Dostopno na: http://phantomjs.org/

[17] KnpLabs - FriendlyContexts, sept. 2014. Dostopno na: https://github.com/KnpLabs/FriendlyContexts

[18] Postman REST Client, sept. 2014. Dostopno na: http://www.getpostman.com/

[19] Sebastian Bergmann, PHPUnit, sept. 2014. Dostopno na: https://phpunit.de/

[20] Konstantin Kudryashov, Behat, sept. 2014. Dostopno na: http://docs.behat.org/

[21] nelmio, alice, sept. 2014. Dostopno na: https://github.com/nelmio/alice

[22] fzaninotto, Faker, sept. 2014. Dostopno na: https://github.com/fzaninotto/Faker

[23] Wikipedia, Responsive web design, sept. 2014. Dostopno na: http://en.wikipedia.org/wiki/Responsive_web_design

[24] Amazon Web Services (AWS), Cloud Computing Services, sept. 2014. Dostopno na: http://aws.amazon.com/

[25] Ubuntu Server, sept. 2014. Dostopno na: http://www.ubuntu.com/server

[26] Nginx, sept. 2014. Dostopno na: `http://nginx.org/`

[27] PostgreSQL, sept. 2014. Dostopno na: `http://www.postgresql.org/`

[28] GitHub, Build software better, together, sept. 2014. Dostopno na: `https://github.com/`

[29] Atlassian JIRA, Plan, track, work smarter and faster, sept. 2014. Dostopno na: `https://www.atlassian.com/software/jira`

[30] W3Counter August 2014 Market Share, sept. 2014. Dostopno na: `http://www.w3counter.com/globalstats.php`

[31] HTML5 test, sept. 2014. Dostopno na: `http://html5test.com/`

[32] Martin Fowler, GivenWhenThen sept. 2014. Dostopno na: `http://martinfowler.com/bliki/GivenWhenThen.html`

[33] App store, ROCKit, sept. 2014. Dostopno na: `https://itunes.apple.com/en/genre/ios/id36?mt=8`

[34] Wikipedia, Continuous integration, sept. 2014. Dostopno na: `http://en.wikipedia.org/wiki/Continuous_integration`

[35] Wikipedia, Continuous delivery, sept. 2014. Dostopno na: `http://en.wikipedia.org/wiki/Continuous_delivery`

[36] Travis CI, sept. 2014. Dostopno na: `https://travis-ci.com/`