VŠB - Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra informatiky

Enterprise aplikace v prostředí PHP
Enterprise Applications based on PHP

2015 Bc. Petr Kadlec

VŠB - Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra informatiky

Zadání diplomové práce

Student: **Bc. Petr Kadlec**

Studijní program: N2647 Informační a komunikační technologie

Studijní obor: 2612T025 Informatika a výpočetní technika

Téma: Enterprise aplikace v prostředí PHP
Enterprise Applications Based on PHP

Zásady pro vypracování:

Cílem práce je vytvořit koncepční pohled na problematiku využití platformy PHP v prostředí Enterprise aplikací.

1. Zmapujte požadavky na vývoj a provoz Enterprise aplikací a definujte jejich specifika.
2. Zmapujte a zhodnoťte současnou situaci v oblasti technologií PHP z pohledu jejich využití pro vývoj Enterprise aplikací.
3. Navrhněte vhodnou architekturu a technologie tak, aby bylo možné řešit požadavky na tento typ aplikací.
4. Implementujte případové řešení, které bude prezentovat navržené řešení specifických požadavků. Zároveň bude sloužit k testování celého systému.
5. Zhodnoťte možnosti navrženého řešení s ohledem na praktické nasazení.

Seznam doporučené odborné literatury:

[1] Robin Nixon: Learning PHP, MySQL, JavaScript, CSS & HTML5. OReilly. 2014. ISBN 978-1491949467
[2] Martin Fowler: Patterns of Enterprise Application Architecture. Addison-Wesley. 2002. ISBN 978-0321127426

Formální náležitosti a rozsah diplomové práce stanoví pokyny pro vypracování zveřejněné na webových stránkách fakulty.

Vedoucí diplomové práce: **Ing. Michal Radecký, Ph.D.**

Datum zadání: 01.09.2014
Datum odevzdání: 07.05.2015

doc. Dr. Ing. Eduard Sojka
vedoucí katedry

prof. RNDr. Václav Snášel, CSc.
děkan fakulty

Prohlášení

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

V Ostravě, dne 7. května 2015

Poděkování

Děkuji Ing. Michalovi Radeckému, Ph.D. za vedení této práce.

Abstrakt

Tato práce má za cíl zhodnotit a případně navrhnout vhodné řešení pro tvorbu enterprise aplikace v PHP. Krom samotné architektury aplikace je zde řešeno vhodné vybavení serveru, použité nástroje a procesy potřebné při vývoji. Vybral jsem nástroje, které umožní vytvořit kvalitní aplikaci, a zároveň jsem naimplementoval demonstrační aplikaci, která se snaží shrnout a ověřit navrhovanou architekturu v praxi. Závěrem práce je, že PHP je vhodný jazyk pro tvorbu enterprise aplikací. Zároveň je navržen soubor nástrojů a postupů, které jsou vhodné nejen pro aplikace v PHP, ale i pro navržení velké a dále udržovatelné aplikace.

Klíčová slova

PHP, enterprise aplikace, architektura aplikace, extrémní programování, agilní metodiky, Git, Symfony 2, testování

Abstract

The goal of this thesis is to assess and propose appropriate solutions for implementing enterprise applications based on PHP. Apart from the actual architecture of the applications, the appropriate server software, the tools and the processes necessary for development are solved as well. I chose the tools to create a quality application and implemented a demo application that summarizes and validates the proposed architecture in practice. The thesis concludes that PHP is a language suitable for creating enterprise applications. It also proposes a set of tools and processes which are appropriate not only for applications based on PHP, but also for the design of large and maintainable applications.

Key Words

PHP, enterprise application, application architecture, extreme programming, agile methodology, Git, Symfon 2, testing

Seznam použitých symbolů a zkratek

DQL	Doctrine Query Language
DRY	Don’t repeat yourself
HTML	HyperText Markup Language
JSON	JavaScript Object Notation
ORM	Object-Relation Mapping
PHP	Hypertext Preprocessor
REST	Representational State Transfer
RUP	Rational Unified Process
SCRUM	Agilní metodika
SQL	Structured Query Language
WebKit	Renderovací jádro prohlížeče vyvíjené primárně firmou Apple
YAML	Ain’t Markup Language

Seznam obrázků

OBRÁZEK 1: ÚPRAVY VYCHÁZEJÍCÍ Z HLAVNÍ VĚTVE 14
OBRÁZEK 2: SLOUČENÍ ÚPRAVY S HLAVNÍ VĚTVÍ BEZ --NO-FAST-FORWARD 15
OBRÁZEK 3: SLOUČENÍ ÚPRAVY S HLAVNÍ VĚTVÍ S PŘÍZNAKEM --NO-FAST-FORWARD 15
OBRÁZEK 4: GRAF STROMU PO PROVEDENÍ REBASE 15
OBRÁZEK 5: GRAF ZÁVISLOSTI CENY OPRAVY NA DOBĚ, KDY JE OBJEVENA 17
OBRÁZEK 6: VRSTVY APLIKACE 28
OBRÁZEK 7: GIT HISTORIE DEMONSTRAČNÍ APLIKACE 41

Seznam ukázek zdrojových kódů a konfigurace

UKÁZKA 1: PŘEHLEDNÝ TEST V CODECEPTION 23
UKÁZKA 2: KONFIGURACE COMPOSERU VE FORMÁTU JSON.. 24
UKÁZKA 3: NÁVRH ENTITY DOMÉNOVÉHO MODELU 29
UKÁZKA 4: REPOSITÁŘ.. 31
UKÁZKA 5: METODA PRO VÝBĚR PUB. ČLÁNKŮ, KTEROU LZE VOLAT Z JINÝCH REPOSITÁŘŮ 32
UKÁZKA 6: SERVISA S VYTVOŘENÍM NOVÉHO UŽIVATELE 33
UKÁZKA 7: FASÁDA VYTVÁŘEJÍCÍ NOVÉ UŽIVATELE.. 34
UKÁZKA 8: TŘÍDA POUŽÍVAJÍCÍ PROPERTY INJECTION 36
UKÁZKA 9: TŘÍDA POUŽÍVAJÍCÍ CONSTRUCTOR INJECTION... 36
UKÁZKA 10: YAML KONFIGURACE SERVISNÍHO KONTEJNERU V SYMFONY 2 37

Obsah

1 Úvod ... 7

2 Enterprise aplikace ... 9

3 Zázemí aplikace ... 10

3.1 Webový server ... 10

3.2 Datové úložiště ... 10

3.3 Integrační server ... 11

4 Best practice pro samotný vývoj ... 13

4.1 Metodika vývoje ... 13

4.2 Správa verzí ... 14

4.3 Kontrola kvality zdrojového kódu - codereview ... 16

4.4 Zásady psaní zdrojového kódu ... 17

4.4.1 Nástroje pro hlídání standardů v PHP aplikaci ... 18

4.4.2 Type hinting skalárních typů ... 19

4.5 Sestavení aplikace ... 20

4.6 Testování ... 21

4.6.1 Jednotkové ... 21

4.6.2 Integrační ... 21

4.6.3 Funkcionální ... 22

5 Framework ... 24

6 Architektura aplikace ... 26

6.1 Rozvrstvení aplikace ... 28

6.1.1 Entita ... 28

6.1.2 Repositář ... 30

6.1.3 Servisa ... 33

6.1.4 Fasáda ... 34

6.1.5 Persistence a entity manager ... 35

6.1.6 Kontrolery a šablony ... 35

6.2 Propojení jednotlivých částí aplikace ... 36

6.2.1 Dependency injection a servisní kontejner 36
7 Konzistence dat 39
7.1 Datový model 39
7.2 Objektově relační mapování 39
7.3 Transakce 40
8 Demonstrační implementace 41
8.1 Instalace a sestavení 41
8.2 Architektura a návrh aplikace 42
9 Závěr 44
10 Reference 45
11 Přílohy na CD/DVD 47

1 Úvod

To, jakým způsobem by se aplikace měly vyvíjet, se liší tím, pro koho je určena a jak se bude používat. Je rozdíl vytvářet redakční systém nebo software pro banku. Každý typ aplikace má jiné požadavky na standardy, architekturu, udržovatelnost a další. Druhý typ aplikace spadá do kategorie enterprise a z hlediska návrhu a vývoje je daleko zajímavější. Již několik let se podílím na vývoji systémů, které i když nejsou přímo softwarem určeného pro banku, tak svým rozsahem a požadavky se do této skupiny řadí. Proto bych se rád podělil o své zkušenosti a zjištěná fakta, která jsem během let na těchto systémech získal. Obecně bývají enterprise aplikace psány v jazycích jako Java, C# a další, které jsou primárně určeny na tento typ aplikace. Nicméně někdy je vyžadováno třeba přímo ze zadání zadavatele, existující infrastruktury nebo kompetencemi vývojového týmu zvolit jazyk PHP, který volím i já, pro tuto práci.

Jazyk PHP nikdy nebyl primárně určen pro takovýto software a sloužil převážně pro jednodušší webové prezentace. Největší slabostí jazyka je, že patří mezi interpretované jazyky, což znamená, že zdrojový kód není přeložen do binární podoby a interpretují se při spouštění přímo zdrojové kódy. O chybách v aplikaci se tak dozvíme většinou až za běhu, což je nežádoucí. Další velkou nevýhodou je chybějící typová kontrola pro skalární typy a některé další chyby v samotném návrhu jazyka. Také díky těmto vlastnostem v základu chybí důkladná statická kontrola kódu ještě před jeho spuštěním. Nicméně jazyk jako takový díky tomu nabízí jistou dynamiku a možnost si jej upravovat podle potřeb programátorů, což při správném použití může naopak zlepšit práci v tomto jazyce.

Pokud už je třeba pro enterprise aplikaci zvolit PHP, tak aby takovéto projekty nebyly předurčeny k selhání, je třeba se na celý problém podívat ze širšího kontextu. Při vhodné volbě nástrojů, pracovních postupů a kvalitní promyšlené architektuře, lze i pomocí tohoto jazyka dosáhnout kvalitní aplikace. V posledních letech jsem pracoval na několika takovýchto systémech a zjišťoval, které postupy se osvědčily a které naopak nikoliv. Pracoval jsem s profesními špičkami v tomto oboru, ověřoval si jednotlivé technologie a principy v praxi a v této práci je shrnu do uceleného souboru informací.

V druhé kapitole bych zkráceně uvedl, co si představuji přesněji pod pojmem enterprise aplikace a jaké jsou na ní požadavky. Na to naváže třetí kapitola, kde se věnuji zázemí aplikace, pod kterým si můžeme představit, jaké další technologie budeme kromě programovacího jazyku potřebovat a jak je správně zvolit. Patří zde hlavně webový server, databáze a integrační server. Následuje kapitola sdružující seznam best practice, netýkajících se přímo architektury a programování, ale pro kvalitní vývoj aplikace se mi osvědčily nebo jsou přímo nutné pro dosažení požadované kvality aplikace. Pátá kapitola ukáže, zda volit nějaký hotový framework či nikoliv a jaké by měl případně splňovat kritéria. V šesté kapitole popíši podrobněji

architekturu aplikace, její rozdělení do vrstev jak se mi osvědčily a vysvětlím jednotlivé výhody nebo nevýhody zvolených postupů. Při mnou navrhované architektuře je kladen důraz na konzistenci dat a obecně v PHP aplikacích je to velmi podceňovaná vlastnost, proto se v sedmé kapitole zaměříme právě na konzistenci a její udržování. Nakonec je nachystána ukázková implementace, ve které se snažím navrhovanou architekturu a principy demonstrovat. Závěrem bych rád zhodnotil, zda i aplikace v PHP mohou být kvalitní a zda se hodí pro tento typ aplikací.

2 Enterprise aplikace

Abychom mohli navrhovat enterprise aplikace, je třeba si vytýčit, co to vlastně je. Jedná se o aplikace, které jsou rozsáhlé, co se funkčnosti týče, tak mohou mít třeba vysoké nároky na výkon. Na jejich vývoji se povětšinu času podílí více členů týmu, který musí být schopen pružně reagovat na požadavky, efektivně mezi sebou spolupracovat a přitom všem si zachovat vysoký standard výsledné aplikace. Navíc jakékoli chyby v enterprise aplikaci mohou mít nepříjemné dopady, takže je třeba jim předcházet například důkladným otestováním aplikace. Navíc aplikace často není jednou finálně dokončena a tím končí její vývoj, ale je neustále rozšiřována a upravována.

Přičemž nejčastější problém bývá, že po čase není možno takovéto aplikace efektivně vyvíjet a to z důvodu zvětšování chybovosti, nebo nemožnosti jednotlivé části aplikace rozšiřovat bez zásadního refaktoringu. Tomu se dá zabránit vhodným návrhem architektury a automatickým testováním. Co se týká výkonnosti, tak je třeba navrhovat zázemí aplikace s ohledem na potenciál aplikace, aby i v budoucnu bylo možno co nejvíce škálovat, neboli rozložit zátěž mezi více serverů. V případě webové aplikace se jedná jak o samotný webserver, tak o datové úložiště, které by mělo být možno efektivně využívat a případně jej cachovat do ještě rychlejších druhů úložišť.

3 Zázemí aplikace

Volba toho, na jakých technologiích bude výsledná aplikace postavena, je důležité jednak pro samotný vývoj, ale hlavně pro její další budoucí rozvoj a limity, respektive cenou za navyšování jejich limitů s tím, jak se aplikace vyvíjí a roste. U PHP aplikace jsou zde hlavní tři pilíře: webový server, datové úložiště a samotné PHP. U samotného PHP jde především o jeho verzi, kde by se samozřejmě mělo jednat o co nejvyšší stabilní verzi ať už kvůli výkonu, tak kvůli bezpečnosti a možnostem, které přinášejí nové verze jazyka. U zbylých dvou je výběr již složitější. Nakonec je třeba zvolit ještě integrační server, který sice není potřeba pro samotnou funkčnost aplikace, ale velmi usnadňuje vývoj a předchází chybám, které mohou během vývoje nastat.

3.1 Webový server

Nejčastěji používaný web server je Apache, jsou zde ale také alternativy v podobě Nginx, IIS, Lighty a mnoho dalších. Pro účely této práce jsem zvolil Nginx (1). Nginx je stále populárnější a aktuálně má podle netcraft.com v září 2014 podíl 14,3% (2) na trhu a mezi miliónem nejvytíženějšími weby dokonce 20,3%, což svědčí pro jeho výkonnost. Navíc jeho podíl stále roste.

Já jej zvolil kromě výkonu, také z důvodu jeho snadné instalace, dostatku informací na internetu a jednoduchosti konfigurace. Ve srovnání s Apache je zde například absence souboru .htaccess, který obsahuje konfiguraci konkrétní aplikace pro webserver. To považuji za výhodu, jednak pro navýšení výkonu (není třeba tyto soubory vůbec načítat) ale hlavně pro to, že tyto soubory jsou často využívány k různým činnostem, ke kterým nejsou určeny a programátoři při nutnosti upravit tento soubor buď neví, co vše ovlivní nebo dlouze prohledávají dokumentaci a diskuzní fóra na internetu.

Nutno podotknout, že webserver se dá relativně snadno zaměnit za jiný a aplikace na něj není většinou přímo vázaná. Například přechod z Apache na Nginx znamená většinou pouze přepis .htaccess do konfigurace Nginx a žádný další zákrok programátora by neměl být nutný. Často se také používají kombinace dvou webserverů, například Apache pro zpracování dynamického obsahu a veškerý statický obsah je zpracován pomocí Nginx, který je v tomto ohledu mnohem výkonnější.

3.2 Datové úložiště

Datovým úložištěm lze chápat i přímý přístup k souborům na disk, což má samozřejmě své místom, například pro obrázky a další statický obsah. Nicméně je potřeba ukládat také dynamický obsah, který je nutno různě filtrovat, propojovat a to co nejefektivněji. Nejčastěji se používají relační databáze, ale v poslední době dostávají obliby také tzv. nosql databáze a to nejčastěji v podobě key-value databáze nebo grafové databáze (3).

Key-value databáze povětšinou poskytují vyšší výkon, než běžné SQL databáze a to z důvodů, že jsou na míru přizpůsobeny k rychlému vyhledávání podle zadaného klíče. Vytváření relací mezi daty je zde také možné, ale integritu a konzistenci dat, musí hlídat samotná aplikace a při používání takto uměle vytvořených relací je degradován výkon takovéto databáze. Tento typ databáze je tedy vhodný hlavně pro data, kde nejsou nutné přímo využívat relace a aplikace si vystačí s jednoduchými dotazy, kterých může být o to více co do množství.

Grafové databáze jsou velmi specifické, ale pro určitý typ dat jsou velmi efektivní. Jako typický příklad jsou například různé sociální sítě, kde je zapotřebí zkoumat vztahy mezi lidmi a podle jejich přátel jim například doporučovat nový obsah, který by jej mohl také zajímat. Popřípadě sítě telekomunikačních operátorů a podobně. Takovéto databáze jsou přizpůsobeny hlavně na hledání vztahů mezi jejími vrcholy, nejkratších cest a podobně.

Většina aplikací, ovšem pracuje s relačními daty a na ty jsou navrženy SQL databáze. Nejznámější jsou Oracle, MSSQL, PostgreSQL, MySQL, MariaDB a další. Pro tuto práci jsem zvolil PostgreSQL, z důvodů licence a možností, které nabízí. Nicméně databázi je vhodné volit podle možností a zkušeností s konkrétní databází. Možnosti mají velmi podobné a většinou nás od přímého přístupu dělí určitá vrstva abstrakce od konkrétního přístupu zvolené databáze. Ale i přes tuto abstrakci je důležité využít potenciál databáze na maximum. Příklad uvedu v následujícím odstavci.

Většina aplikací, bude po celou dobu funkčnosti pracovat primárně s jedním druhem databáze, takže abstraktní vrstva, která nám umožňuje snadno vyměnit například PostgreSQL databázi za MySQL není potřeba. Respektive tyto abstrakce toto umožňují, ale to nemusí naši aplikaci omezovat, jelikož většinou lze tyto vrstvy rozšířit o funkcionalitu specifickou pro konkrétní databázový engine. Tudíž si můžeme dovolit používat například datový typ JSON, i když bude funkční bez dalších úprav pouze na jednom konkrétním typu databáze. Tím jsme schopni využívat lépe potenciálu zvolené databáze. Univerzálnost vyžadují pouze aplikace, které jsou navrženy pro využití v jiných aplikacích jako knihovny nebo různé frameworky, redakční systémy a podobně.

3.3 Integrační server

Integrační server slouží k průběžné integraci již hotových úprav do stabilní verze aplikace. Je součástí metodiky extrémního programování a v podstatě jde o to, že každá změna se dělá mimo stabilní verzi a kdykoli můžeme spustit na integračním serveru sestavení kterékoli úpravy a výsledkem je informace o tom, zda daná aplikace lze spustit, dodržuje všechny standardy, veškeré testy procházejí a tím pádem je úprava z technické stránky připravena stát se součástí stabilní verze aplikace.

V praxi to funguje tak, že vždy, když je do verzovacího systému nahrána nová verze nebo jakákoli změna, je tato verze sestavena se všemi kontrolami a pokud se během tohoto procesu objeví jakákoli chyba, je autor dané verze upozorněn třeba emailem, že jím nahraná verze není v pořádku. Hlavní výhodou je, že když programátor zapomene nebo si řekne, že ta drobnost, kterou přece nemůže nic pokazit, je zkontrolována a o případné chybě se dozví hned, jak to bude možné. Další nespornou výhodou je úspora času, jelikož kontroly, které trvají dlouho, se mohou spouštět pouze na integračním serveru a programátor se může věnovat něčemu jinému. Poslední výhodou je, že máme na jednom místě přehled, kdy byla která verze aplikace funkční popřípadě jiné souhrnné informace, například o pokrytí testy a jiné.

Nejpopulárnější integrační servery jsou Jenkins CI a Travis CI, ale existuje spousta jiných. Jejich výsledné použití je velmi podobné. Navíc tyto nástroje nejsou přímo vázány na PHP a lze jej využívat i pro jiné programovací jazyky a záleží jen na jejich nakonfigurování. Konfigurace toho, co má všechno integrační server na projektu dělat by ideálně měla být verzována přímo s projektem. Během vývoje aplikace totiž dochází ke změnám i v tom, co všechno a jak dělá integrační server.

4 Best practice pro samotný vývoj

Následující podkapitoly obsahují seznam doporučených technik postupů, které se netýkají samotné aplikace nebo její architektury, ale je dobré se jich držet obzvláště u rozsáhlých projektů. Tyto techniky a postupy hlavně řeší práci v týmu a udržovatelnost dalšího vývoje projektu. Aplikace může dobře fungovat i bez některých těchto technik, ale časem se její udržovatelnost stane velmi drahou nebo její kvalita bude nedostatečná. Jedná se o výčet praktik, které se mi osvědčily v praxi a dané problémy řešily nejefektivněji.

4.1 Metodika vývoje

To, jakým způsobem se celý vývoj naplánuje a jak probíhá je definováno mnoho způsoby. V minulosti se často vyvíjelo vodopádovým modelem, kdy se vše najednou vyspecifikovalo, navrhlo, zrealizovalo a nakonec najednou předalo zákazníkovi. Toto byl problém hlavně u větších projektů, mezi které rozhodně patří i enterprise aplikace. Proto byl vytvořen RUP (Rational Unified Process), který jednak klade důraz na návrh, ale zavádí také iterativní vývoj, kdy se vývojový proces skládá z více vodopádů a průběžně vidíme výsledek. Méně striktní metodikou, která klade důraz také na iterativní vývoj, a hlavně pružnou reakci na potřebné změny během vývoje, jsou agilní metodiky.

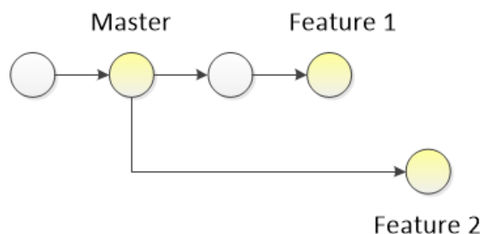
Nejznámějším zástupcem agilních metodik, který se mi osvědčil pro vývoj rozsáhlých aplikací, je SCRUM. Ten si zakládá na tom, aby co nejčastěji, v jednotkách týdnů, doručoval zákazníkovi funkční produkt rozšířený o požadované funkčnosti. V kterékoli fázi projektu máme výsledný produkt, i když ještě třeba neobsahuje veškerou funkcionalitu, kterou čekáme od finální verze produktu. Další výhodou těchto krátkých iterací je to, že již v průběhu známe rychlost vývoje a jsme tak schopni lépe odhadnout, kdy vývojový tým splní požadovaný rozsah práce. Tudíž jsme schopni reagovat na požadované termíny, popřípadě posílit tým nebo omezit funkčnost výsledného produktu ve prospěch termínu dokončení. Pokud se navíc jedná o dlouhodobý projekt, tak požadavky na systém se mohou mnohokrát změnit od jeho prvotního zadání, například novou legislativou, a díky iteracím se tyto změny dají postupně zakomponovávat do aplikace a nemusíme čekat na dokončení celé aplikace, abychom ji mohli upravovat. SCRUM si dále zakládá na denním setkání týmu, kdy si všichni členové sdělí, co dělali, co plánují dělat a s čím mají problém. Díky tomu mají členové týmu přehled, co se kde děje a řeší problémy takřka ihned nebo o nich informují a počítá se s nimi, například změnou dodaného množství nové funkcionality. Díky retrospektivám, které tým pravidelně provádí, také validuje své postupy a snaží se je neustále vylepšovat a tím se stává efektivnější. Hlavní výhodou této metodiky je, že až na pár základních pravidel je velmi variabilní a téměř každý tým si jej přizpůsobuje pro svou potřebu. Nejtěžší na této metodice je domluvit se se zákazníkem, aby i on pravidelně produkt validoval a přebíral po zvolených iteracích. Pokud s tím zákazník nemá zkušennost, tak se mu to může jevit jako spoustu času z jeho strany a nemusí ihned vidět přínosy tohoto způsobu vývoje.

4.2 Správa verzí

Důležitou součástí vývoje je správa verzí veškerých zdrojových kódů a konfigurací. Systémů pro správu verzí je několik, přičemž nejznámější jsou Git, Mercurial a Svn. Správná práce s verzovacím systémem je důležitá pro všechny členy týmu, kteří mohou jakkoli zasáhnout do zdrojových kódů. Správným použitím a definováním workflow dosáhneme toho, že tým mezi sebou dokáže efektivně spolupracovat, do stabilní verze nezaneseme změny, které nechceme a že budeme snáze hledat vznik chyb.

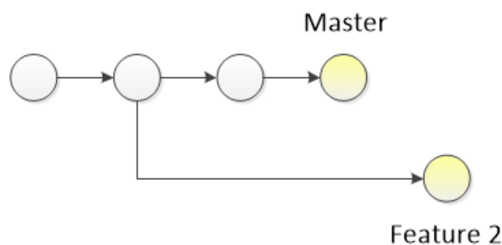
Pro tuto práci jsem si vybral Git, jakožto nástroj, který ovládám nejlépe a jeho mechanismy se mi nejvíce osvědčily v praxi. Nicméně věřím, že u ostatních nástrojů se dá použít podobné užitečné workflow.

Mé workflow spočívá v tom, že všechny změny se dělají na samostatné větvi mimo master, hlavní větev se stabilní verzí, kterou můžeme kdykoli spustit do ostrého provozu. Jednotlivé úpravy vycházejí právě z masteru a reprezentují, jak budou fungovat, pokud se začlení právě ta daná úprava do masteru. Strom takového stavu na Gitu vypadá jako na Obrázek 1.

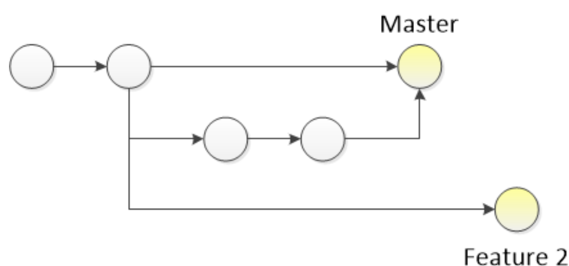


Obrázek 1: úpravy vycházející z hlavní větve

Na takovýchto větvích se provádí codereview, validace zákazníkem a další. Jakmile je úprava schválena, je možno ji sloučit s hlavní větví. Pokud provádíme sloučení ze stavu popisující Obrázek 1, nedochází již k žádné změně zdrojových kódů, takže nemůže nastat žádný konflikt a to co bylo otestováno, se skutečně stává hlavní stabilní verzí. Sloučení je třeba provést s přepínačem *--no-fast-forward*. Pokud bychom tak neučinili, vznikne strom jako na Obrázek 2. To je nežádoucí, jelikož není na první pohled zřejmé, které komity byly součástí dané úpravy. Pokud ovšem použijeme přepínač *--no-fast-forward*, tak vznikne strom jako na Obrázek 3, kde lze jasně vidět, které komity byly do stabilní verze přidány a je zde nový komit, který shrnuje celou úpravu jednak svou zprávou, tak v jediném komitu vidíme, co vše se úpravou změnilo a pokud bychom tuto změnu potřebovali použít někde jinde v budoucnu, snadno ji máme právě v tomto komitu.

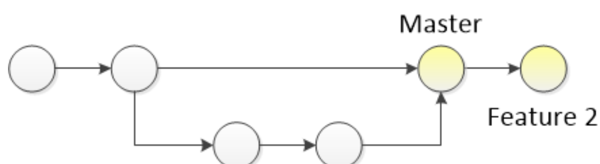


Obrázek 2: sloučení úpravy s hlavní větví bez --no-fast-forward



Obrázek 3: sloučení úpravy s hlavní větví s příznakem --no-fast-forward

Po sloučení úpravy do hlavní větve dojde ke stavu, že ostatní větve již nevychází z aktuální hlavní stabilní verze. Tato situace se dá řešit pomoci *rebase*. Nejedná se o nic víc, než že se celá větev přesune na aktuální stabilní verzi. Takže výsledek po *rebase* bude jako na Obrázek 4.



Obrázek 4: graf stromu po provedení rebase

Tento přístup pomocí *rebase* bývá často zavrhován kvůli jedinému důvodu, měnění historie změn. Toto nepodporuje ani většina nástrojů používaných při codereview a je třeba s tím počítat. Přesto si myslím, že pokud celý tým dodržuje správně pravidla, která dále zmíním, je přínos tohoto postupu daleko větší.

Prvním kladem je, že již zmíněný fakt, že při slučování do stabilní větve již nedochází ke změně zdrojového kódu a úprava vždy funguje se všemi změnami ze stabilní verze. Dále, při samotném rebase, pokud nastanou konflikty, jsou řešeny postupně. Konfliktem je myšleno, pokud v úpravě měníme stejný zdrojový kód, který se změnil ve stabilní verzi a verzovací systém jej nedokáže sám automaticky vyřešit. Přesouvá se totiž jeden komit rebasované úpravy po druhém, což umožňuje jednodušeji vyřešit případné konflikty. Konflikty také řeší autor úpravy, takže v případě řešení konfliktu může svou úpravu nejefektivněji přizpůsobit změnám ve stabilní verzi, jelikož má nastudováno, jak má úprava fungovat. Jako poslední výhodu bych zmínil právě možnost měnit historii, tím je myšleno, že například před tím, než pošle

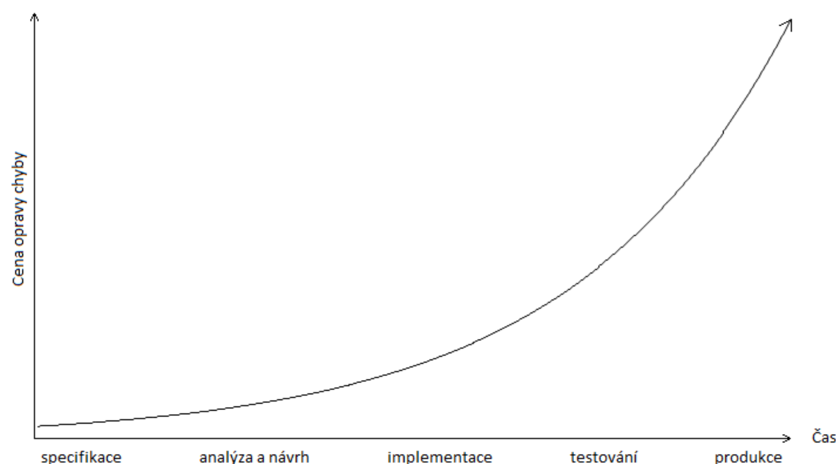
programátor úpravu ke schválení dál, upraví historii tak, aby byla přehledná, jasná a například bez zbytečných kusů kódu (například debugovací části) nebo refaktoringu, které by pouze komplikovaly následné codereview (kapitola 4.3.) Navíc, pokud je historie udržovaná, dá se v ní poměrně snadno automaticky vyhledat pomocí příkazu *bisect*, kdy se vyskytla nějaká konkrétní chyba.

Aby celý tento proces mohl fungovat, je třeba striktně dodržovat pravidlo, že každý může dělat komity pouze do svých větví. Pokud by bylo nutné tam přece jen přidat nový komit, musí na něj být vytvořena nová větev, která bude vycházet z úpravy, kde bychom komit původně potřebovali přidat. Pokud bychom toto nedodržovali, může se stát, že nového komitu si autor úpravy nevšimne a nenávratně přijdeme o všechny takovéto změny, přičemž se už to nemusíme nijak dozvědět.

4.3 Kontrola kvality zdrojového kódu - codereview

Codereview je proces, který umožňuje, aby veškerý zdrojový kód vidělo více programátorů a mohli jej připomínkovat před jeho finálním nasazením. V praxi to většinou funguje tak, že jakmile programátor dokončí úpravu, tak ještě před samotným testováním ji zasílá právě na codereview a dokud není vyřešené, úprava nemůže být testována a ani schvalována. Během codereview totiž často dochází ještě ke změnám zdrojového kódu, během kterých se může měnit funkčnost, nebo se mohou vyskytnout nové chyby. Ze zásad extrémního programování také vyplývá, že každý kus kódu by měl projít přes tuto kontrolu a já s tím souhlasím bez výjimky.

Během plánování projektu se hlavně ze začátků může jevit codereview jako neekonomické, jelikož jednu úpravu si musí nastudovat dva a více programátorů a pochopit její implementaci, což znamená více času stráveného programátory na dané úpravě. V týmech, kde jsem takto pracoval, zvýšilo časovou náročnost implementace úpravy v průměru o cca 30% celkového času. Takže je pravda, že se jedná o větší počáteční investici. Nicméně ve větším časovém horizontu to může čas naopak ušetřit. Což je způsobeno tím, že programátoři mají větší přehled o funkčnostech v aplikaci, kde nejsou přímo autory, což také zlepšuje zastupitelnost, rychlost zaučení a efektivitu implementaci dalších úprav. Chyby v implementaci bývají objeveny mnohem dříve ještě před tím, než dojde k testování, což snižuje cenu jejich odstranění, viz následující Obrázek 5.



Obrázek 5: Graf závislosti ceny opravy na době, kdy je objevena

Kromě objevení chyb se mohou odhalit daleko závažnější problémy. Tím myslím chybný návrh architektury nebo nesrozumitelný kód. Programátor provádějící codereview se na úpravu dívá s nadhledem, který autorovi úpravy chybí, jelikož ten chtěl většinou vyřešit daný problém a když jej vyřešil, už často neřeší, zda to nejde lépe. Zato jiný programátor již neřeší daný problém, pouze se dívá na hotové řešení a hledá, zda to zapadá do okolního konceptu aplikace a zda nelze něco udělat lépe. Také se minimalizuje zmíněný nesrozumitelný kód, protože pokud kontrolovaný kód nejde pochopit při jeho čtení nebo vyžaduje příliš mnoho pozornosti pro jeho pochopení, měl by být dekomponován na více jednodušších celků, které půjdou snáze pochopit. Někdy ovšem nemusí být nejlepším řešením dekomponovat část kódu, ale je pouze zajistit například vhodným komentářem, proč tomu tak je a jak daná část funguje, aby se nepochopení kódu neopakovalo jiným členem týmu nebo i stejným, ale po delším čase, kdy se budeme znovu ke kódu vracet. Proto je důležitá komunikace mezi členy týmu během samotného codereview.

Při codereview by se programátor měl zaměřit hlavně na samotnou implementaci algoritmu a jeho architekturu, takže aplikace musí být plně funkční a zdrojový kód musí splňovat všechny standardy. Pokud by tomu tak nebylo, odvádí to pozornost od toho, co má být skutečně zkontrolováno. Toho lze z velké části docílit automaticky, čemuž je věnována následující kapitola.

4.4 Zásady psaní zdrojového kódu

U takovéhoto typu projektu je důležité, aby všichni členové vývojového týmu dodržovali daná pravidla a zásady aplikace, které si jasně vymezí. Tento soubor pravidel by měl být někde jasně sepsán a v průběhu vývoje se vyvíjí společně s projektem podle potřeb.

Nicméně úplný seznam všech pravidel by v reálu nikdo nechtěl číst, tudíž nejlepší pravidla jsou ta, která jsou automaticky hlídána. K tomu máme velké množství nástrojů, které je budou hlídat za nás, a buď je za nás opraví, nebo nám řeknou, co je v nepořádku. Jako příklad, zda se

pro odsazení budou používat mezery nebo tabelátory je nejlepší, pokud nalezneme nástroj, který naši volbu zkontroluje ve všech zdrojových kódech a v případě špatného použití provede opravu a upozorní nás na to. Takovýmto způsobem si nikdo nemusí toto pravidlo číst, ale jakmile jej poruší, je na to upozorněn a přenastaví si svůj editor. A ostatní členové v týmu nemusí řešit, zda někdo toto pravidlo neporušil.

Kromě formátování zdrojových kódů zde patří i další věci, jako například, kdy můžeme považovat úpravu z technického pohledu za hotovou. Podmínkou například může být, že všechny třídy v dané vrstvě aplikace musí být stoprocentně pokryty testy, vše je řádně zdokumentované, úprava splňuje všechny akceptační kritéria zadání a sestavení aplikace (více kapitola 4.5) je bez chyby. Jako další věci, co mohou tato pravidla obsahovat, je například ještě struktura aplikace nebo procesy při vývoji.

4.4.1 Nástroje pro hlídání standardů v PHP aplikaci

Existuje mnoho nástrojů na kontrolu standardů v PHP aplikaci, nicméně každý takovýto nástroj dělá něco trochu jiného a není jeden komplexní nástroj na kontrolu všeho. Dále je si třeba uvědomit, že z podstaty tak dynamického jazyka jako PHP, nelze statickou analýzou, kterou tyto nástroje dělají odhalit veškeré chyby dřív, než dojde ke spuštění konkrétní části kódu. Nicméně i tak mohou poskytnout velmi kvalitní základy kontroly před spuštěním aplikace.

Základem je kvalitní vývojové prostředí (známé také jako IDE – Integrated Development Environment), které programátorovi ukáže mnoho nedostatků okamžitě při psaní kódu, takže je může ihned opravit a časem si je zautomatizuje. Bezkonkurenčně nejlepším vývojovým prostředím pro PHP je PhpStorm, nicméně často se také používá NetBeans. Obecně vývojové prostředí mají od ostatních nástrojů ty výhody, že znají kontext všech souborů v aplikaci. PhpStorm jde ale dál a snaží se odhalit chyby, které jiné vývojové prostředí nekontrolují, protože nepočítají s dynamickým aspektem jazyka. Například upozorní na volání metody třídy z jiného souboru, která neexistuje, ale v případě že třída používá magickou metodu *__call()*, tak naopak upozornění potlačí. Jako nesporná výhoda je použití této statické analýzy i samostatně z příkazové řádky, takže ji například můžeme spustit na integračním serveru, takže se o problému programátor dozví i v případě, kdy během psaní zdrojového kódu si nevšimne upozornění editoru.

Další skupina nástrojů slouží také ke statické analýze, ale už neudržuje kontext s ostatními soubory. Proto se dají jednoduše psát vlastní pravidla, která se mají kontrolovat. Existují stovky hotových pravidel. Uvedu například zákaz přímého přístupu k superglobálním proměnným, které umožňují přístup ke vstupu uživatele, což je neošetřený vstup a mělo by se k němu přistupovat přes třídy k tomu určené, pravidla na minimální délku názvu proměnných (vedou k popisnějším názvům a čitelnějšímu kódu), ale také zajímavějším, jako maximální

cyklomatická komplexita. Ta zjednodušeně určuje množství rozhodovacích členů v bloku kódu. Takovýmto omezením zamezíme složitým metodám a donutíme kód dekomponovat na jednodušší a čitelnější části. Mezi tyto nástroje se řadí například PHP Mess Detector a PHP Code Sniffer. Pravidla, která se dají automaticky opravit, jako například nepoužívané importy tříd, správné odsazení a podobně, provedou programy jako PHP Code Beautifer nebo PHP Code Fixer.

Do samostatné skupiny bych zařadil detektor duplicity kódu, kde patří hlavně PHP Copy/Paste Detector, který hledá duplicitní kód a to i nehledě na pojmenování proměnných a úroveň odsazení kódu. Pomáhá tak odhalit kód, který je duplicitní a programátor je donucen vytvářet znovupoužitelné komponenty a lépe tak dodržuje princip DRY.

4.4.2 Type hinting skalárních typů

PHP před verzí 7.0 nepodporuje type hinting skalárních typů. Tím je myšlena kontrola vstupních argumentů, zda jsou opravdu typu *int*, *string*, *float* nebo *boolean*. Tato volnost jazyka mu přidává na jednoduchosti, ale zároveň představuje velké riziko chybovosti a nekonzistence.

Jelikož toto není řešeno na úrovni jazyka, snaží se toto řešit různými způsoby. Nejčastěji vlastním error handlerem (4), což nepovažuji za dobré řešení. Nastává zde problém při práci s namespace, mnoho knihoven s tím není kompatibilní a celkové snížení výkonu při neustálém parsování textu chybových zpráv. Další řešení nabízí knihovna Sentry od Václava Purcharta, kterou popisuje ve své diplomové práci (5). Využívá zde zápisu anotací a generování proxy tříd, takže z hlediska výkonu by neměl být problém a řešení je navrženo čistě. Bohužel knihovna není aktuálně veřejně dostupná.

Z řešení, které jsem nalezl, mi vyplívá, že aktuálně není dostupné vhodné řešení a je třeba s touto nativní vlastností jazyka počítat a jakmile bude vydána stabilní verze PHP 7, tak přejít na tuto verzi, která tento problém řeší na úrovni jazyka a není třeba jej složitě obcházet. V případně nutnosti volit řešení HHVM (6), které tuto podporu má, ale omezuje jazyk jinak a spousta knihoven s ním není kompatibilní.

4.5 Sestavení aplikace

Sestavením aplikace je myšleno její plné zprovoznění z čisté instalace. Během tohoto procesu probíhá mnoho akcí a liší se podle typu aplikace, nicméně většinou se jedná o následující kroky:

- stažení všech závislostí/externích knihoven
- vytvořit potřebné složky – logy, cache
- nastavit složkám a souborům správná oprávnění
- vytvoření konfigurace – připojení k databázi, mailovému serveru, url aplikace
- kompilace javascriptu a kaskádových stylů (pokud používáme například LESS, CofeeScript nebo jiné)
- vytvoření struktury databáze
- nahrání základního nastavení do databáze
- zahřátí cache

Celý tento proces se vyvíjí společně s aplikací a obsahuje relativně hodně kroků. Proto je dobré toto zautomatizovat, aby nemohlo dojít k tomu, že při ručním provádění jsme na něco zapomněli. Obecně se doporučuje takzvané sestavení na jeden klik (6), což znamená, že spustíme sestavení a provede se vše potřebné automaticky, maximálně se nás aplikace doptá například na konfiguraci nebo prostředí, pokud jej ještě nezná.

Sestavení se nepoužívá pouze ke spuštění nové instance aplikace, ale také k aktualizaci a to jak na produkci, tak při vývoji. Na produkci se jedná o zamčení aplikace, aby se během provádění se systémem nepracovalo, nahrání změněných souborů, spuštění databázových migrací neboli provedení změn ve struktuře databáze a promazání cache. Při vývoji ovšem celé sestavení usnadňuje práci programátorům hlavně při stažení změn ostatních programátorů, protože k těm může docházet poměrně často a po každé změně by mělo dojít k opětovné čisté instalaci aplikace s tím rozdílem, že nám stačí stáhnout změny oproti předchozí verzi. Navíc při vývoji jsou kontrolovány standardy a spouštěny testy.

Pro samotné nastavení, co se má při sestavení aplikace dělat, existuje celá řada nástrojů. Mě se osvědčil nástroj Phing (7). V přehledné konfiguraci navolíte, jaké akce se mají spustit. Těmto akcím se dají nastavovat závislosti, takže jde například nastavit, že před vložením demonstračních dat do databáze musí dojít k vytvoření samotné databáze a její struktury. Akce jsou většinou prosté příkazy z příkazové řádky, ale existují předpřipravené akce pro nejpoužívanější aplikace, sloužící právě k testování, kontrolám standardů nebo podmínky pro provedení akcí a podobně. Samozřejmostí je možnost spouštět pouze jednotlivé akce samostatně podle potřeby.

4.6 Testování

Kvalitní aplikace by měla obsahovat testy, které zaručují a zároveň hlídají funkčnost aplikace. Pokud by testy neobsahovala, tak jednak nevíme, zda opravdu funguje jak má a hlavně při dalších úpravách aplikace nevíme, zda vše ostatní funguje tak jak se očekává a nemůžeme si snadno ověřit, zda jsme nezměnili funkčnost něčeho jiného. To ale neznamená, že aplikace, která má testy a všechny úspěšně prochází, neobsahuje žádné nestandardní chování nebo chybu. Testy nikdy nedokáží pokrýt všechny kombinace vstupních dat, nicméně by měly pokrýt většinu standardních scénářů a to jak úspěšných tak neúspěšných a mezní hodnoty. Navíc je dobrým pravidlem napsat test pokaždé, když se objeví chyba, aby se již nikdy v budoucnu neobjevila.

Abychom vůbec mohli testovat, tak je třeba mít kód aplikace napsán tak, aby byl testovatelný. Pokud totiž chceme testovat nějakou část aplikace, která je příliš složitá nebo má mnoho závislostí, stávají se testy neúměrně složité, ne-li nerealizovatelné. Abychom dosáhli testovatelnosti, je třeba dodržovat single responsibility principle (více v kapitole 6). Třídy také nesmí být závislé na nějakém globálním stavu, ale vše co ke své činnosti potřebují, musí explicitně vyžadovat jako svou závislost. Pokud by tomu tak nebylo, nelze se v testu spolehnout na její chování, protože globální stav, který implicitně využívá, nepostřehneme nebo jej nemůžeme ovlivnit. Tuto problematiku řeší striktní dodržování inversion of control (více v kapitole 6.2.1).

4.6.1 Jednotkové

Každá public metoda třídy, která obsahuje doménovou logiku, by měla obsahovat test. Tomuto testu se říká jednotkový test a kontroluje funkčnost nejmenší jednotky systému, která obsahuje logiku. Tyto testy jsou základem pro všechny další testy, jelikož bez funkčního základu nemůže fungovat ani vyšší vrstva. Navíc tyto testy jsou a musí být rychlé, aby je programátoři mohli spouštět co nejčastěji a chyby mohli řešit okamžitě.

Proto je třeba si i říct, které vrstvy aplikace chceme takto testovat. Primárně ve struktuře, kterou popisuji v této práci, jsou jednotkové testy určeny pro entity (kapitola 6.1.1) a servisy (kapitola 0.) Naproti tomu repositáře (kapitola 6.1.2) nejsou často pokryty jednotkovými testy, protože testování metody, která vrátí z databáze uživatele na základě jeho jedinečného identifikátoru, není zcela relevantní. Navíc bychom museli, kvůli přístupu například k databázi, vytvářet takzvané mocky, což jsou objekty, které kontrolovaně simulují jiný typ objektu a vydávají se za něj. Díky mockování jsme schopni odstínit závislosti testované třídy a přesně definovat, jak se tyto náhrady závislostí mají chovat. Nejpoužívanější nástroj pro testování v PHP je PHPUnit (8).

4.6.2 Integrační

Integrační testy již netestují jednu konkrétní třídu, ale to, jak spolu více tříd spolupracuje jako celek. Tyto testy se používají již pro libovolnou vrstvu, ale jejich spouštění trvá podstatně

déle, pokud se během testu využívá databáze. Což znamená ještě problém s tím, že testy by se navzájem neměly ovlivňovat. To znamená, že pokud test pracuje s nějakým persistentním uložištěm, měl by nehledě na jeho výsledek jej vrátit do výchozího stavu. Nejčastěji se jedná o databázi, kde se dá většinou celý test uzavřít do transakce a po skončení testu provést rollback.

Kromě samotného času vykonávání testů, je zde větší režie na údržbu těchto testů. Pokud máme třídy, na kterých je často nějaká závislost, typicky třída s lokalizací aplikace nebo entita uživatele a podobně, tak při její změně nebo rozšíření je třeba všechny testy ji využívající upravit a překontrolovat.

Zápis testů je velmi podobný jednotkovým testům, proto je víc než vhodné použít na ně stejný nástroj jako pro jednotkové testy.

4.6.3 Funkcionální

Funkcionální testy, někdy také nazývané jako akceptační testy jsou posledním a také nejdražším druhem testu a to jak do doby trvání testu, tak do jeho údržby. Testují poslední prezentační vrstvu aplikace, která ale závisí na všech ostatních vrstvách, takže zároveň testuje i je. Probíhají tak, že pokládají aplikaci běžné požadavky a kontrolují odpověď aplikace. Odpovědí může být v jednodušším případě JSON data, ale v případě HTML kódu stránky bývá kontrola odpovědi složitější. Naštěstí existují nástroje, které toto usnadňují a vytvářejí srozumitelnou abstrakci pro kontrolu HTML odpovědí.

První typ nástrojů neprovádí skutečné požadavky přes webový server, nýbrž pouze vytváří objekt požadavku s url, http hlavičkami a další, pošle jej aplikaci a testuje objekt reprezentující odpověď. Tento přístup má jasnou výhodu v tom, že zcela vynechává webový server, čímž zrychluje délku testů. Navíc je možné přistupovat k samotné aplikaci, takže můžeme testovat stav jednotlivých objektů v aplikaci nebo kontrolovat změny provedené v databázi. Také obsahují nástroje pro kontrolu HTML, zda odpověď obsahuje požadovaný element se zadaným obsahem a podobně. Naopak jako jasná nevýhoda je absence samotného vykreslování obsahu, spouštění JavaScriptu a požadavků na externí obsah stránky. Navíc takovýto nástroj musí být buď součástí frameworku, který jsme si zvolili nebo bychom si jej museli naprogramovat sami. Mezi hotové řešení patří například Symfony Test Client (9).

Druhým přístupem je provádět reálné požadavky na webový server. Toto umožňuje například Selenium (10), které využívá prohlížeče nainstalované v operačním systému. Nebo je zde PhantomJS (11), který využívá jádra WebKit a emuluje prohlížeč pouze v paměti, takže může běžet na pozadí a bez grafického rozhraní. Nicméně toto jsou nástroje, které lze použít pro libovolnou webovou aplikaci, kterou chceme otestovat včetně PHP aplikace.

Existují nástavby nad těmito nástroji, abychom mohli pohodlně psát testy právě pro aplikace v PHP. Nejvíce se mi osvědčil nástroj Codeception (12), kde styl zápisu je velmi přirozený a mohou je psát i technicky zdatnější testeři. Ukázka 1 demonstruje přehlednost testu přihlášení v Codeception.

```
class SecretLoginCest {

	public function testLogin(AcceptanceTester $i) {
		$i->wantTo('login on secret zone');
		$i->amOnPage('/login/');
		$i->see('Secret zone');
		$i->fillField('login[username]', 'MyNick');
		$i->fillField('login[password]', 'password123');
		$i->click('Login');
		$i->see('Welcome boss');
	}

}
```

Ukázka 1: přehledný test v Codeception

Existují ještě další nástavby, kdy tyto testy zapisují přirozeným jazykem, většinou anglicky, do obyčejných textových souborů. Tento přístup je vhodný pro zadavatele, který takto může rovnou při specifikaci nové funkčnosti na ni vytvořit test. Nicméně toto se hodí většinou pro menší projekty a pro enterprise aplikace se mi tento přístup neosvědčil kvůli komplexnosti požadovaných testů.

5 Framework

Framework je ucelený soubor funkcí, tvořící jednotný přístup pro vytváření aplikace nebo řešení problémů. Díky frameworku nemusíme řešit každý problém úplně od základu, ale použít již existující kód, na kterém se podílí a testují jej tisíce dalších programátorů. Pár let zpět bylo velmi populární nepoužívat hotové frameworky, ale vytvářet vlastní řešení. Tato doba je naštěstí pryč a hotové frameworky se staly standardem.

Navíc přichází doba takzvaných devstacků, což je soubor knihoven, které jsou mezi sebou sladěny a spolupracují spolu. To si uvědomují i vývojáři frameworků a snaží se frameworky rozdělit na samostatné funkční celky ze kterých lze poskládat celý framework. Má to tu výhodu, že je možné si z každého frameworku vzít to, co opravdu potřebujeme. Navíc když nevyužíváme všechny jeho součásti, tak jednoduše nepotřebnou součást odebereme a tím odlehčíme naší aplikaci.

Pro toto je hojně používán nástroj Composer (13), který slouží ke správě závislostí na knihovnách třetích stran. V konfiguračním souboru uvedeme, jaké knihovny vyžadujeme a v jaké verzi viz Ukázka 2: konfigurace Composeru ve formátu JSON. Důležité je také rozlišovat, jaké knihovny vyžadujeme ve vývojovém prostředí, protože například knihovny pro testování na produkci nepotřebujeme. Composer umožňuje více věcí, jako je stahování knihoven přímo z repositářů, spouštění příkazů po instalaci nebo aktualizaci a podobně.

```
{
	"require": {
		"php": ">=5.6.8",
		"symfony/symfony": "2.6.*",
		"doctrine/orm": "~2.2,>=2.2.3,<2.5",
		"doctrine/dbal": "<2.5",
		"doctrine/doctrine-bundle": "~1.2",
		"twig/extensions": "~1.0",
		"symfony/assetic-bundle": "~2.3",
		"symfony/swiftmailer-bundle": "~2.3",
		"symfony/monolog-bundle": "~2.4",
		"sensio/distribution-bundle": "~3.0,>=3.0.12",
		"sensio/framework-extra-bundle": "~3.0,>=3.0.2",
		"incenteev/composer-parameter-handler": "~2.0"
	},
	"require-dev": {
		"sensio/generator-bundle": "~2.3"
		"phpunit/phpunit": "4.5.1",
	},
}
```

Ukázka 2: konfigurace Composeru ve formátu JSON

V okamžiku, kdy se rozhodnete aktualizovat aplikaci na novější verzi knihoven, nastává problém v tom, že každou knihovnu musíme aktualizovat zvlášť. Je to o něco více práce, než kdybychom vše provedli najednou, ale možnost mít pod kontrolou jednotlivé knihovny to vyvažuje.

I když je zde možnost si aplikaci takto poskládat, tak je dobré si zvolit některý z hotových frameworků jako základ, na kterém budeme stavět základy aplikace. Primárně by měl být zvolen Framework, který tým dobře zná, protože přechod na jiný framework sice zkušený tým zvládne relativně rychle, ale nebude znát vnitřní principy frameworku a to kromě délky zaučení bude vést k tomu, že nebudou vždy voleny úplně optimální způsoby řešení. Dalším kritériem a u enterprise aplikace důležitějším, je, že Framework musí umožňovat navrhovanou architekturu. Tím je myšleno, že pokud jsme se rozhodli nepoužívat návrhový vzor Active Record, ale nějakou formu ORM, je víc než vhodné zvolit Framework, který toto už v základu umožňuje. I když většinou můžeme vyměnit způsob práce s daty, tak většina částí frameworku je tomu přizpůsobena a do budoucna budeme nuceni jej nepřirozeně ohýbat nebo mnoho věcí řešit zcela vlastním způsobem. Dalším důležitým kritériem při volbě frameworku je, že aplikuje princip inversion of control a také jak jej aplikuje. V PHP aplikacích se mi nejvíce osvědčil dependency injection (více v kapitole 6.2.1).

Pro tuto práci a ukázky jsem zvolil framework Symfony 2 (14), se kterým mám zkušenosti a osvědčil se mi v mnou navrhované architektuře. Vyniká rozsáhlou zahraniční komunitou, která jej neustále vyvíjí, včetně množství rozšíření, kde se lze minimálně inspirovat, jak se dají jednotlivé problémy řešit. Jako nespornou výhodu bych uvedl několik let dopředu známá roadmapa, kde se jasně dozvíme, kdy je naplánováno jednotlivé vydání verzí, a hlavně které verze budou označeny jako LTS (Long Term Support). Jedná se o verze, kde je po dobu tří respektive pěti let zaručena její aktualizace respektive oprava bezpečnostních chyb. Toto je důležité právě s ohledem na enterprise aplikace.

V této práci jsem zvolil pro datový model Doctrine 2 (15), jelikož je velmi dobře integrována právě do Symfony 2 a hlavně zapadá do konceptu navrhované architektury, kde objektově relační mapování mi vychází jako nejvhodnější způsob pro rozsáhlejší aplikace. Doctrine 2 vychází z ověřené implementace a návrhu frameworku Hibernate ORM v Javě.

6 Architektura aplikace

Aplikace musí mít strukturu jasně definovanou a musí dodržovat několik základních pravidel, aby bylo možné ji udržitelně dále vyvíjet. Z mého pohledu by se měla rozvrstvit do více vrstev, které budu popisovat níže a navíc dodržovat zejména DRY, SOLID a neporušovat obecně principy OOP.

DRY (don't repeat yourself) jak už z názvu vyplývá, značí, že konkrétní logika v aplikaci by měla být pouze na jednom místě a neměla by se kdekoli vyskytovat duplicitně. V případě, že tomu tak není, přináší to problémy, kde v případě jakékoli změny nebo opravy chyby je třeba provést změnu na více místech. To ale přináší riziko, že na některé z výskytů se zapomene a také je to zbytečná práce navíc. Dalším problémem při porušení tohoto pravidla je, že řešení stejného problému může být na jiném místě řešeno jinak, přičemž by jej vždy měl řešit jeden stejný ideální algoritmus.

SOLID je soubor následujících pěti principů:

- Single responsibility principle – jedna třída v aplikaci by měla dělat pouze jednu věc a nic víc. Díky tomu jsme schopni ji jednoduše otestovat a hlavně znovu používat v různých částech aplikace a dodržovat tak výše uvedené pravidlo DRY.
- Open closed principle – měly bychom být schopni jednotlivé části rozšiřovat a přitom nebýt nuceni upravovat jejich stávající kód. Zde je důležité, aby i externí knihovny, včetně částí frameworku, toto umožňovali, jinak budeme časem nuceni knihovny reimplementovat podle našich představ.
- Liskov substitution principle – nahrazením rodiče jeho potomkem nesmíme vytvořit nefunkční aplikaci, pouze bychom měli měnit její chování.
- Interface segregation principle – rozhraní by měly popisovat co nejmenší potřebné chování. V konkrétní implementaci můžeme pro každé chování mít zvlášť třídu nebo jedna třída může plnit úlohu více rozhraní. Dosáhneme tím lepších podmínek právě pro výše uvedený Single responsibility principle
- Dependency inversion principle – třída nemá být závislá na jiné konkrétní implementaci, ale pouze na daném rozhraní či jeho abstrakci. Umožňuje nám to lepší znovu použitelnost a testovatelnost kódu.

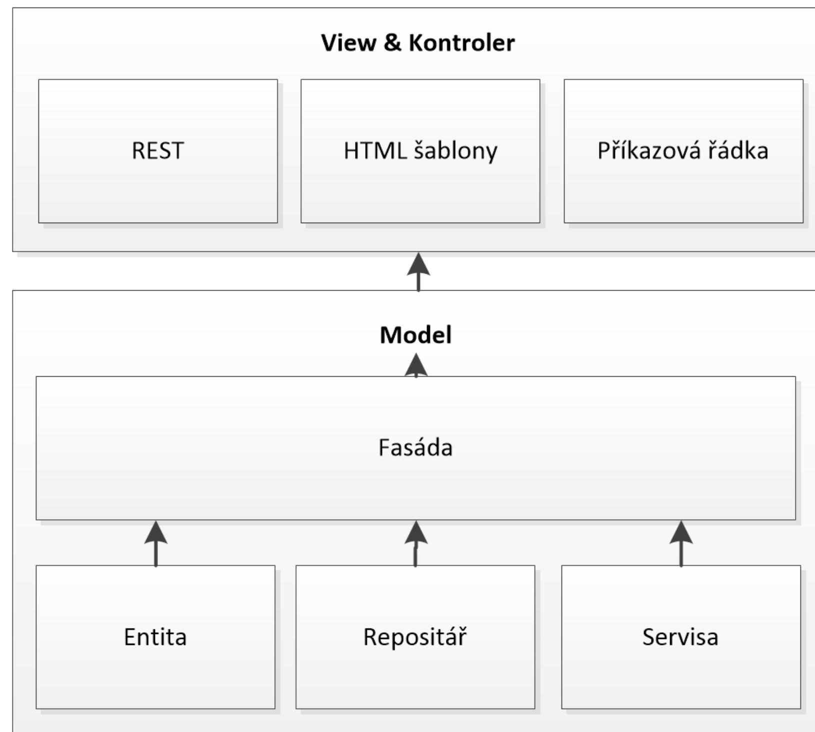
Pokud budeme dodržovat tyto pravidla, vznikne systém složený ze spousty menších částí a ty je dobré nějak seskupovat a určit mezi nimi jasná pravidla pro jejich použití. MVC je jeden z nejpoužívanějších architektonických vzorů, který aplikaci rozděluje na tři základní vrstvy:

- Model – veškerá aplikační logika, práce s úložišti
- View – formátování dat na výstupu
- Controller – zpracování vstupu, jeho předání modelu, který vrátí data pro výstup do view

Nejzajímavější je právě vrstva modelu, kterou si často lidé pletou pouze s reprezentací dat například z databáze a samotnou aplikační logiku zanášejí do kontroleru. Přitom vše co má být znovu použitelné musíme přenést právě do modelu. Potom není problém jednoduše zaměňovat vnější API aplikace, protože například pro vytvoření článku můžeme mít kontroler pro webové rozhraní, jiný pro REST API a další pro mobilní aplikaci. Kontroler pouze přetransformuje vstup na data požadující model, který provede danou akci. Proto se v následující kapitole budeme převážně věnovat architektuře této modelové vrstvy a ukážeme si, jak ji ideálně můžeme rozdělit a dále s ní pracovat.

6.1 Rozvrstvení aplikace

Architektura modelové vrstvy je pro další vývoj nejdůležitější, a proto se jí budeme věnovat nejvíce a rozdělíme ji na další části. Patří zde entity, repositáře, servisy a fasády, jak vidíme na Obrázek 6: vrstvy aplikace.



Obrázek 6: vrstvy aplikace

U každé části aplikace by mělo být na první pohled zřejmé, do které vrstvy patří. Toto je možné jednak adresářovou strukturou, tak samotným pojmenováním. Adresářovou strukturou je vhodné řešit tento problém pro minimum věcí a slouží spíše pro oddělení šablon prezentační vrstvy nebo statický obsah. Samotný zdrojový kód aplikace by měl ideálně tvořit tématické celky, takže v jedné složce, či namespace, by měly být třídy vztahující se k jedné věci. Například namespace s názvem *User* obsahuje jak entitu, repositář, servisu a fasádu. Proto je třeba dbát na názvosloví a přidávat například postfix vrstvy, do které třída spadá.

6.1.1 Entita

Entita reprezentuje datovou jednotku, která většinou reprezentuje objekt ze skutečného světa a operace nad tímto objektem. Často se tak setkáváme, že entity jsou používány pouze jako hloupé data objekty bez logiky. Jedná se o takzvaný anemický model a již delší dobu je považován za anti-pattern (16), ale přesto jej často v aplikacích vídáme.

Přičemž entita by měla obsahovat logiku s ní spjatou. Například objednávka sama spočítá, kolik obsahuje položek a které ještě nejsou vyexpedované. Další důležitou věcí je, že veškeré

změny, které na entitě provádíme, by měly představovat konkrétní případy užití. To znamená, že entita neobsahuje automaticky na každý atribut setter, ale má metody vyjadřující konkrétní scénář, ke kterému je určena. Ideální entitu vidíme v následující Ukázka 3.

```
class Article {

    private $author;

    private $title;

    private $text;

    private $createdAt;

    private $publishedAt;

    public function __construct(User $author, $title, $text) {
       $this->author = $author;
       $this->title = $title;
       $this->text = $text;
       $this->createdAt = new DateTime();
       $this->publishedAt = null;
    }

    public function edit($title, $text) {
       $this->title = $title;
       $this->text = $text;
    }

    public function puslish() {
       if ($this->publishedAt !== null) {
              throw new Exception();
       }

       $this->publishedAt = new DateTime();
    }

    // All getters attributes

}
```

Ukázka 3: návrh entity doménového modelu

V ideálním případě by ještě měly být ošetřeny skalární typy, zda například titulek je opravdu typu string a navíc nepřekračuje-li povolenou délku. Jelikož toto aktuálně není možné na úrovni jazyka pro skalární typy, nabízí se možnost toto explicitně kontrolovat. Nicméně z hlediska efektivity toto řešení běžnými způsoby není příliš výhodné a je lepší nechat tuto kontrolu na jiné vrstvě – například validační pravidla na formuláři nastavené automaticky podle databázového schématu. PHP od verze 7 by již mělo mít podporu type hintingu skalárních typů, takže základní kontrola bude vyřešena.

Na příkladu lze tedy vidět, že entita je již od jejího vzniku v konzistentním stavu. Také všechny metody, které na ní lze volat ji stále udržují v konzistentním stavu. Toto má velkou výhodu, protože kdykoli s entitou pracujeme, je ve funkčním stavu. Takovéto objekty navíc dodržují zapouzdření jakožto jeden z principů OOP.

Poslední věc, kterou by měla entita splňovat, je, že nesmí být závislá na ničem jiném, než jsou jiné entity nebo jiné deterministické závislosti. Těmi můžou být myšleny funkce nebo statické metody. Jakmile by entita měla závislost na prostředí, je hůře znovupoužitelná a špatně testovatelná. Navíc to často svádí k porušování single responsibility principle. Návrhový vzor active record přesně toto dělá, a proto jej považuji za anti-pattern. Entita jednak reprezentuje daný objekt, ale zároveň se stará o svou persistenci, přičemž už předem ví, kde a jak bude uložena. To by ale entita neměla vědět a vrstva nad ní by měla rozhodovat, zda ji uloží do databáze, souboru nebo s ní pracuje pouze v paměti.

6.1.2 Repositář

Návrhový vzor repository zahrnuje jednak odkud a jak se mají data načítat a ukládat, tak i mapování na samotné entity. Mapování většinou řeší již hotové knihovny samy, tudíž jej většinou přímo nebudeme provádět. Repositář by neměl navenek jakkoli sdělovat, kde jsou data skutečně uložena a mít natolik obecně navržené rozhraní, že jej můžeme kdykoli nahradit jiným repositářem se stejným rozhraním, akorát bude data získávat ze souboru, nikoliv databáze. Tato abstrakce úložiště je důležitá v pozdějších fázích projektu z několika důvodů. Pokud potřebujeme měnit strukturu uložených dat, tak stačí v repositáři transformovat nový formát dat do zatím neupraveného doménového modelu. Důležitou roli hrají také při optimalizaci výkonu, protože nám poskytují ideální místo, kde jsme schopni načítat data z různých cache.

Co se týká návrhu jeho rozhraní, opět by veřejné metody měly co nejvíce reprezentovat opět konkrétní případy užití a neměly by navenek poskytovat žádnou možnost dynamicky skládat svůj výsledek. Toto typicky umožňují různé formy query builderů. Ty lze používat, ale pouze interně v repositáři a navenek by toto nemělo být umožněno.

Důležitým prvkem jsou také argumenty, které přijímá. Velmi časté je, že pokud chceme například pouze články nějakého autora, předáváme jeho jedinečný identifikátor – ID. Což samo o sobě stačí ke všemu, co aktuálně potřebujeme. Doporučuji si předávat celou entitu autora. Je to jednak z důvodů absence nativní podpory typové kontroly skalárních typů, takže budeme mít zaručeno, že dostáváme opravdu to, co definujeme, tak z důvodů, že případné chyby objevíme daleko dříve, než by tomu bylo v opačném případě. Například budeme požadovat články od autora s konkrétním ID. Ale takový uživatel již nemusí z nějakého důvodu existovat. Jelikož před samotným výpisem musíme podle ID najít entitu tohoto autora, může nám jiný repositář který to má na starost vyhodit výjimku, kterou může některá z vyšších vrstev uživateli zobrazit třeba jako, že daný autor již neexistuje místo stránky, která skončí například bez jediného nalezeného článku. Zde lze namítnout, že zde vyžadujeme něco, co není třeba a zbytečně se třeba dotazujeme na databázi. To je pravda, nicméně ze zkušenosti mi to přijde stále jako lepší řešení i za cenu jednoho jednoduchého dotazu na databázi navíc a nedodržoval bych to pouze v případě nutnosti mikrooptimalizací. V Ukázka 4 je návrh repositáře, kde pouze poslední

metoda splňuje navrhované ideální rozhraní, které by měl repositář poskytovat. První dvě metody by se v něm neměly vůbec objevit z důvodů výše uvedených.

```
class ArticleRepository {

    /**
     * @param string $column
     * @param mixed $value
     * @return Article[]
     */
    public function findBy($column, $value) {
       $query = $this->entityManager->createQuery('
              SELECT a
              FROM ' . Article::class . ' AS a
              WHERE a.'column' = :value
       ');
       $query->setParameter('value', $value);

       return $query->execute();
    }

    /**
     * @param int $authorId
     * @return Article[]
     */
    public function findByAuthorId($authorId) {
       $query = $this->entityManager->createQuery('
              SELECT a
              FROM ' . Article::class . ' AS a
              WHERE a.author = :authorId
       ');
       $query->setParameter('authorId', $authorId);

       return $query->execute();
    }

    /**
     * @param User $author
     * @return Article[]
     */
    public function findByAuthor(User $author) {
       $query = $this->entityManager->createQuery('
              SELECT a
              FROM ' . Article::class . ' AS a
              WHERE a.author = :author
       ');
       $query->setParameter('author', $author);

       return $query->execute();
    }
}
```

Ukázka 4: repositář

V ideálním případě by repositáře neměly být mezi sebou nijak závislé. Zvyšuje to přehlednost při čtení jednotlivých dotazů na databázi. Jakmile je dotaz skládán na více místech v různých souborech, není na první pohled jasné, jaký dotaz bude výsledkem. Existují výjimky, kde například dotaz pro výběr publikovaných článků je základem pro mnoho jiných dotazů, které již patří do jiných repositářů a abychom logiku výběru publikovaných článků neopakovali, vytvoříme si závislost na hlavním repositáři článků. Musí být ale zjevné, jaký alias používá základní dotaz pro publikované články pro entitu článku. Takto využívaná metoda potom může vypadat jako v Ukázka 5: metoda pro výběr pub. článků, kterou lze volat z jiných repositářů. V případě takovéto potřeby je ale třeba dávat pozor, aby takovéto metody nebyly používány nikde jinde, než z jiných repositářů.

```
public function getPublishedArticlesQueryBuilder($articleAlias) {
	return $this->entityManager->createQueryBuilder();
		->select($articleAlias)
		->from(Article::class, $articleAlias)
		->where($articleAlias . '.publishedAt IS NOT NULL');
	}
}
```

Ukázka 5: metoda pro výběr pub. článků, kterou lze volat z jiných repositářů

6.1.3 Servisa

Toto je vrstva, kde by měla být uložena veškerá aplikační logika, která se netýká přímo některé z entit nebo se jí týká, ale má další závislosti, které by entita neměla mít. Typicky také obstarává logiku nad více entitami. Servisy mohou být mezi sebou navzájem závislé, ale je důležité, aby nebyly jakkoli závislé na persistenci. Pokud by tomu tak nebylo, tak by byly méně znovupoužitelné a hlavně takto tvořené servisy tvoří společně s entitami základ aplikace a je třeba je mít důkladně pokryté testy, abychom se na ně mohli spolehnout. Jakmile jsou závislé na něčem jiném než na jiných servisách, stávají se hůř testovatelné.

Proto je důležité, aby veškerá potřebná data pro provedení akce dostala servisa jako parametry volání metody. Jako příklad může sloužit vytváření nového uživatele, kdy je třeba kontrolovat, zda už takový uživatel neexistuje a pokud ano, skončit řízenou chybou. V Ukázka 6: servisa s vytvořením nového uživatele lze vidět, že pro provedení registrace, kdy kontrolu existence uživatele se stejným emailem provádíme v této servise, aniž bychom se dotazovali databáze. Na první pohled se jedná o zbytečný krok, proč se rovnou nemůžeme podívat do databáze? Protože tím přebíráme zodpovědnost jiné třídy a vrstvy. Navíc nového uživatele sice vytvoří, ale už neřeší jeho uložení. Díky dodržování těchto pravidel by na tuto třídu měl být schopný napsat test i začátečník v testování. Lze ale namítat, že i když bude uživatel se stejným emailem existovat, nemusíme jej zde zadat. To už ale není problém této třídy ale jejího použití. Při volání metody je zřejmé, že tento údaj je třeba a pokud se jej rozhodneme ignorovat nebo nastane chyba, je to záležitost vyšší vrstvy, která by měla toto případně testovat.

```
class UserService {

	/**
	 * @param UserFormData $formData
	 * @param User|null $userWithSameEmail
	 */
	public function registerNewUser(
		UserFormData $formData,
		User $userWithSameEmail = null
	) {
		if (
			$user !== null
			&& $userWithSameEmail->getEmail() === $formData->email
		) {
			throw new DuplicateEmailException($formData->email);
		}

		return new User(
			  $formData->firstname,
			  $formData->lastname,
			  $formData->email
		);
	}

}
```

Ukázka 6: servisa s vytvořením nového uživatele

6.1.4 Fasáda

Návrhový vzor fasáda, jak už název napovídá, vytváří rozhraní pro vnější komunikaci aplikace. Nejedná se ještě o konkrétní formát jako REST nebo HTML, ale soubor metod představující všechny případy užití, které se systémem můžeme provádět. Všechny vyšší vrstvy by měly přistupovat přímo pouze k této vrstvě. Komponuje totiž dohromady entity, repositáře a servisy do funkčních celků. Entity je třeba chápat s rezervou, protože často se s nimi pracuje i ve vyšších vrstvách, více o této výjimce v kapitole 7. Navíc tato vrstva pracuje s persistencí, takže zde dochází k ukládání všech dat, které změnily nebo vytvořily servisy.

Tato vrstva by neměla obsahovat žádnou aplikační logiku, protože tu obsahují vrstvy pod ní. V ideálním případě by neměla obsahovat vůbec žádnou logiku v podobě rozhodovacích členů nebo cyklů. Toto ale v některých případech například z výkonnostních důvodů nelze dodržet. V případě hromadného zpracování, které je třeba provést v aplikační vrstvě, je třeba dávkově získávat data z databáze a postupně je nechávat zpracovat pomocí servis a následně je také dávkově ukládat. Takovéto výjimky je poté třeba řádně otestovat, protože se nejedná o standardní použití fasády.

Ve většině případů se ale obejdeme bez další logiky a dochází pouze na základě skalárních parametrů získat entity z repositářů, nechat je zpracovat servisami a persistovat nový stav do databáze. Může to potom vypadat jako v Ukázka 7: fasáda vytvářející nové uživatele, kde lze vidět celý případ užití registrace nového uživatele.

```
class UserFacade {

	public function registerNewUser(UserFormData $userFormData) {
		$userWithSameEmail = $this->userRepository->findUserByEmail(
			$userFormData->email
		);
		$newUser = $this->userService->registerNewUser(
			$userFormData,
			$userWithSameEmail
		);
		$this->entityManager->flush($newUser);

		return $newUser;
	}

}
```

Ukázka 7: fasáda vytvářející nové uživatele

Jelikož by veškerá komunikace s aplikací měla procházet právě fasádou, začnou se v ní objevovat metody pouze volající stejnou metodu na repositář. Typicky metody získávající například uživatele podle jedinečného identifikátoru. Nicméně toto umožňuje jednoznačně určit, s čím můžeme ve vyšších vrstvách pracovat. Umožňuje to snazší změny v již existujícím kódu a vyšší vrstvy aplikace mají méně závislostí, protože jim stačí pouze fasáda a již nepotřebují navíc repositář a podobně.

6.1.5 Persistence a entity manager

Veškeré změny, které během požadavku provedeme, je třeba nějak trvale uložit, protože v základu si v PHP nelze mezi požadavky uchovávat žádná data. Toto by měla zajišťovat samostatná vrstva. V mém případě jsem zvolil již hotové řešení, kde využívám již hotového řešení ORM Doctrine 2 (15) a lze jej vidět v ukázkách výše. Tudíž tuto vrstvu jako takovou není třeba navrhovat, pokud použijeme hotové řešení, které funguje podle našich představ. V případě, kdybychom nechtěli použít hotové řešení, je třeba se připravit na návrh vlastního mapování datového modelu na databázi, přístupu k databázi pomocí query builderu, ošetřování vstupů, lazy loading a mnoho dalšího. Proto si nemyslím, že má dnes smysl budovat vlastní tuto vrstvu a nepoužít ověřené hotové řešení.

Mnou zvolená knihovna Doctrine 2 poskytuje zároveň takzvaný entity manager, který v sobě uchovává informace o všech entitách, se kterými v aplikaci pracujeme a jakmile provedeme všechny změny, řekneme právě entity manageru, aby uložil aktuální stav. Ten neudělá nic jiného, než že překontroluje všechny entity, zkontroluje v nich změny a ty uloží do databáze. Všechny tyto změny samozřejmě zaobalí do transakce, takže pokud by došlo k chybě, nedojde k žádné změně. Usnadňuje také práci při získávání dat, jednak již zmíněným query builderem, kde je možné dynamicky skládat dotaz například při filtrování. Také vytváří svůj vlastní jazyk DQL (Doctrine Query Language), který vychází ze standardního SQL, ale abstrahuje a zjednodušuje jeho zápis.

6.1.6 Kontrolery a šablony

Kontroler a šablony jsou sice dvě rozdílné vrstvy, nicméně jsou tak úzce spojené a z hlediska architektury již né tolik významné, že jej uvádím společně. Jedná se již o poslední vrstvy aplikace, které představují něco, jako klienta se kterým komunikuje uživatel.

Typů kontrolerů můžeme mít více, pro představu mohou být zvlášť pro webové rozhraní, REST api nebo příkazovou řádku. V případě kontroleru se jedná pouze o zpracování dat požadavku, kontrola jejich správnosti z hlediska formátu vstupu, předání těchto dat fasádě aplikace a zpracování výsledku fasády pro odpověď. V případě webové stránky se často jedná o získání dat z formuláře, jejich validace v rámci formuláře (ne v rámci datového modelu), zpracování fasádou a zobrazení hlášky o úspěšném nebo neúspěšném provedení akce. K samotnému zobrazení v případě webové stránky bude využita ještě vrstva s šablonami.

Důležité je, aby zde nebyla žádná aplikační logika. V implementaci kdy tomu tak je, bychom logiku z jednoho typu kontroleru museli duplikovat v jiném typu kontroleru a tím bychom nedodrželi princip DRY a také bychom tuto logiku museli testovat nejdražšími funkcionálními testy.

6.2 Propojení jednotlivých částí aplikace

Všechny části aplikace se snažíme navrhovat tak, aby prováděli pouze to, k čemu jsou určeny a na vše další si vytvářeli závislosti. Jde o to, aby si třída jako taková nevytvářela sama vše, co potřebuje, ale explicitně si řekla o to, co potřebuje ke své správní funkčnosti. Tento problémem se podrobněji řeší jako Inversion of Control (17), který navrhuje několik řešení. Nejlepším z nich se mi jeví právě Dependency injection.

6.2.1 Dependency injection a servisní kontejner

V PHP se dá Dependency injection řešit několika způsoby. Prvním způsobem je method injection, což znamená, že závislosti jsou předávány jako parametry metod, které je vyžadují. Toto je nejpracnější způsob a v praxi jsem jej ještě neviděl implementován. Druhým způsobem je property injection, kdy závislosti jsou po vytvoření objektu nastaveny public atributům třídy nebo pomocí setteru. Takovýto přístup můžeme vidět v Ukázka 8: třída používající property injection, tento přístup se již sice dá zautomatizovat v případě, že všechny takovéto property nějak označíme nebo vyjmenujeme. Avšak toto řešení stále umožňuje vytvořit objekt v nekonzistentním stavu a není funkční, dokud nenastavíme správně závislosti. V případě přidání nové závislosti se navíc o tom nedozvíme a můžeme zapomenout ji někde přidat a způsobit tím fatální chybu v aplikaci.

```
class UserFacade {
	private $userRepository;
	private $userService;

	public function setUserRepository(UserRepository $userRepository) {
		$this->userRepository = $userRepository;
	}

	public function setUserService(UserRepository $userService) {
		$this->userService = $userService;
	}
}
```

Ukázka 8: třída používající property injection

```
class UserFacade {
	private $userRepository;
	private $userService;

  public function __construct(
		UserRepository $userRepository,
		UserService $userService
	) {
		$this->userRepository = $userRepository;
		$this->userService = $userService;

	}
}
```

Ukázka 9: třída používající constructor injection

Ideální řešení je constructor injection, kdy veškeré závislosti potřebné pro plnou funkčnost třídy jsou vyžadovány v konstruktoru. Tím zajistíme, že ihned po vzniku takovéhoto objektu, je plně funkční, navíc ještě před tím než jej vytvoříme, zjistíme, zda máme vše potřebné pro jeho vznik a můžeme jej tak vytvořit. Třída poté vypadá jako v Ukázka 9: třída používající constructor injection.

Vytváření takovýchto objektů se většinou automatizuje, o což se stará servisní kontejner. Servisní proto, že všechny takto vytvářené objekty jsou služby neboli servisy (neplést si se servisou v kapitole 6.1.3). Kontejner v základu stačí nakonfigurovat, jaké služby může poskytovat a jakmile si o některou z nich v kódu požádáme, tak vytvoří všechny její závislosti a následně požadovanou službu. Služby se také vyznačují tím, že je vytvořena jen jedna jejich instance a pokud si o ni zažádáme znovu, je nám vrácena stejná instance jako při prvním volání. Toto chování se nejlépe demonstruje na připojení k databázi, kdy se chceme jednou připojit a nadále používat již vytvořené spojení. Toto chování lze samozřejmě změnit, že nadefinujeme, aby se vytvářeli pro konkrétní službu vždy nová instance třídy. Služby také mohou mít závislost na něčem jiném, než na jiných službách. Typicky to jsou cesty ke složkám a souborům, ale také například seznam nějakých entit a podobně. Závislost na statické konfiguraci se dá jednoduše řešit v konfiguraci, ale složitější závislosti jsou řešeny pomocí návrhového vzoru továrny, kde i tyto továrny jsou součástí kontejneru.

Celé řešení závislostí pomocí konstruktoru má nevýhodu u cyklických závislostí, kdy jsou služby na sobě navzájem závislé a to nejen napřímo, ale také transitivně přes některé své jiné závislosti. Toto se většinou dá vyřešit rozdělením některé ze tříd na více menších s méně závislostmi. Tudíž je toto většinou až následkem špatného návrhu.

```yaml
user_service:
  class: UserService

user_repository:
  class: UserRepository
  arguments:
    - @database

user_facade:
  class: UserFacade
  arguments:
    - @user_service
    - @user_repository
```

Ukázka 10: YAML konfigurace servisního kontejneru v Symfony 2

Tímto způsobem většina moderních frameworků v základu řeší závislosti jednotlivých služeb. Bohužel konfigurace kontejneru roste úměrně s velikostí aplikace, kdy udržovat tisíce řádků konfigurace není ideální řešení. Toto řeší autowiring, automatické drátování závislostí. Jde o to, že nekonfigurujeme, na čem je třída závislá, jelikož toto jsme již jednou nadefinovali v konstruktoru dané třídy. Kontejner si automaticky zjistí z definice třídy na jaké argumenty má

třída v konstruktoru a hledá, zda některá ze služeb v kontejneru nedopovídá požadovanému rozhraní. Toto řešení funguje pouze v případě, že požadované rozhraní se v kontejneru nachází pouze jednou. Pokud bychom měli v základu dvě připojení k databázi, autowiring neví, které z nich má automaticky předat jako závislost a je třeba tuto výjimku explicitně konfigurovat, jinak dojde k chybě.

I přes to je konfigurace stále rozsáhlá, musí se udržovat a přitom u většiny tříd pouze říká, že se jedná o službu s daným názvem. Pokud bychom šli dál, tak proč by kontejner nemohl rovnou použít třídu, kterou potřebujeme jako závislost? Takováto třída existuje, a i když není zaregistrovaná v kontejneru, tak můžeme vytvořit její instanci (její závislosti rekurzivně vytvořit úplně stejně) a předat ji jako závislost. Problém je, že by se takto moha stát službou i třída, která službou nemá být, například entita. Proto je třeba tyto třídy určené jako služba nějak označit, což v konfiguračním souboru je nepraktické. Ideální řešení takovéto situace, je přenést tuto konfiguraci přímo do definice třídy, kde buď anotací, nebo implementací nějakého konkrétního rozhraní bez funkčnosti, které udělá ze třídy službu. Kontejner tak nebude hledat pouze ze služeb v kontejneru, ale rovnou požadovanou třídu, a jakmile je označena anotací nebo implementuje požadované rozhraní, vytvoří z ní novou službu. Ruční konfigurace tím není zcela odstraněna, ale pouze značně zjednodušena a kontejner nejdřív hledá právě zde. To je z důvodu, kdy potřebujeme nadefinovat nějakou výjimku, která nemůže být automaticky vyřešena, nebo chceme změnit nějakou službu zaměnit za jinou.

7 Konzistence dat

V předchozích kapitolách jsem často poukazoval na konzistenci, a aby objekty byly vždy validní a funkční. Pokud bychom se nemohli na toto spolehnout, nemůžeme se spolehnout na to, že aplikace funguje podle očekávání. V PHP díky povaze jazyka a jeho dynamičnosti, kterou často i využíváme, nemůžeme zajistit vždy úplnou konzistenci všech objektů. Přesto bychom se měli snažit o co největší kontrolu všech vstupů a datových typů, které požadujeme.

7.1 Datový model

Tak jak jsme si rozdělili aplikaci do vrstev a definovali, které vrstvy mohou se kterými komunikovat, tak by v ideálním případě měla mít každá vrstva svůj datový model a mělo by se s ním pracovat pouze v dané vrstvě. Během komunikace mezi vrstvami by mělo docházet k přemapování z jednoho datového modelu na druhý, přičemž tyto datové modely by byly často identické.

Toto pravidlo už ze své podstaty porušují entity, se kterými se pracuje napříč všemi vrstvami. Takže nic nebrání v šabloně zavolat například změnu hesla uživatele. Většinou si ovšem vystačíme s ještě jedním datovým objektem, který obsahuje vnější vstup, typicky například data z formuláře, které mohou prakticky obsahovat libovolná data. Pomocí tohoto data-objektu vytváříme a editujeme již konkrétní entitu, nicméně vždy než tak učiníme, jsou data validována. Pokud nejsou validní, vrací se chybová zpráva na vstup, aby byla data upravena. Lze namítnout, že validační pravidla budeme uvádět duplicitně, jednou pro entitu a podruhé pro data-objekt vyšších vrstev. Nicméně tyto validační pravidla, které jsou platné pro entitu, se dají automaticky použít i na data objekt a v případě data objektu ještě upravit pro konkrétní případ užití k čemu je určen. Typicky email je sice povinný, ale uživatel si jej nesmí změnit, tudíž nemůže dojít k jeho vyplnění, administrátor jej editovat může a entita jej má jako povinný. Navíc tím, že entity nemají metodu pro editaci každého atributu zvlášť, ale pro jednotlivé případy užití, tak minimalizujeme možnosti, že někdo bude chtít například z šablony jen tak změnit jméno autora, když neexistuje případ užití, kdy editujeme pouze jeho jméno, ale i ostatní atributy.

7.2 Objektově relační mapování

Další problém, který může zanést nekonzistenci do datového modelu, je pokud jako zde používáme ORM a ve dvou různých částech aplikace během jednoho požadavku požadujeme od databáze stejnou entitu. Pokud bychom měli pokaždé jinou instanci entity, reprezentující ten stejný objekt, tak v případě, že budeme chtít aktuální stav uložit, nastane problém, která z instancí se má použít pro persistování. Naštěstí tento problém většina ORM včetně Doctrine 2 řeší. V případě že z databáze vracíme již načtenou entitu, nevytváří se její nová instance, ale je vrácena již v minulosti vytvořená instance dané entity, tudíž i se všemi změnami, které jsme zatím v ní provedli.

Větší problém nastává při změně vazeb mezi entitami. Pokud bychom měli například vazbu mezi autorem a jeho články a tato vazba by byla oboustranná, tak v případě, že změníme autora článku, musíme tuto změnu provést na obou stranách vazby. Kdybychom například změnili autora pouze u článku, tak autor má stále načtené své články včetně toho, který jsme upravili, a už není jeho autorem. Tuto situaci lze řešit oboustrannou synchronizací všech změn, ale nejedná se o triviální problém. Návrh řešení pomocí synchronizace popisuje ve své diplomové práci Václav Purchart (5). Situace lze řešit také bez synchronizace a to tak, že vazby budeme definovat jako jednosměrné, kdy je určena strana vlastníka, který tuto vazbu udržuje. V praxi to znamená, že článek zná svého autora, ale autor nezná své články. To ovšem neznamená, že nemůžeme zjistit články daného autora, můžeme je jednoduše vyhledat, pouze není je získat pomocí lazy loadingu přímo od autora. Ve speciálních případech můžeme nechat oboustrannou vazbu, a to tehdy, kdy se vazba za žádných okolností nemůže měnit, pouze odstranit a to v případě zániku jedné ze stran jejím smazáním. Toto nastává, když kromě autora budeme chtít u článku uložit, kdo jej jako první vytvořil. Tato vazba je jednou na začátku vytvořena a už nikdy nedojde k její změně z povahy jejího účelu.

7.3 Transakce

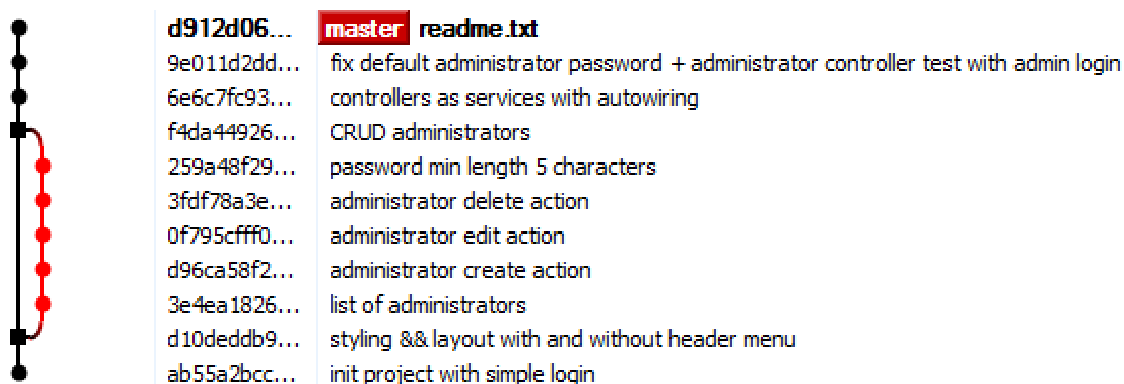
Jak jsem již zmínil v kapitole 6.1.5, všechny změny nad databází jsou prováděny v transakci. Nicméně existuje prostor mezi načtením dat a jejich uložením ještě dostatek prostoru k jejich změně během jiného požadavku a tím k zneplatnění operace, kterou chceme provést. Zjednodušeně v případě odečtení částky z nějakého účtu, pokud budou odeslány dva požadavky ve stejný čas, tak oba dva načtou dostatečný zůstatek na účtu, odečtou požadovanou částku a její výsledek uloží do databáze. První požadavek provede korektní operaci, ale druhý již nezohlední první operaci a uloží chybný zůstatek na účtu, popřípadě by k dané operaci nemělo vůbec dojít, jelikož již nebyl dostatečný zůstatek na účtu, k provedení operace.

Toto se samozřejmě dá řešit tak, že se veškerá potřebná data hned po načtení uzamknou do transakce i pro čtení a v případě modelového příkladu s účty jsme toto schopni ošetřit pro tento konkrétní případ užití. Problém nastává, když během řešení nějakého problému opomeneme a systém to za nás neřeší automaticky. Systémově toto řešit lze tak, že celé zpracování požadavku uzavřeme do transakce, což by znamenalo, že by požadavky často čekali, než se dokončí ty, které byly před nimi. Mělo by neblahý vliv na výkonost a škálovatelnost řešení by se tím značně ztížila. Proto tento přístup nedoporučuji, a pokud se nejedná o kritické operace, řešit ten problém individuálně.

8 Demonstrační implementace

V příloze 1 této práce jsem vytvořil ukázkovou implementaci aplikace, kde se snažím navrhovanou architekturu a nástroje demonstrovat. Aplikace dodržuje navrhovanou architekturu, ale neimplementuje všechny nástroje a návrhy, jelikož jejich konkrétní implementaci, kterou jsem tvořil nebo se podílel na jejich vývoji, je vlastnictvím třetích stran, které na ni mají výhradní právo. Tyto části nemají vliv na samotnou architekturu ani funkčnost, pouze by zjednodušili vývoj a zpřehlednily zdrojový kód.

Z funkční stránky obsahuje aplikace pouze přihlášení a kompletní správu uživatelských účtů. Nicméně toto dostačuje k tomu, abych mohl ukázat, jak funguje instalace, sestavení aplikace, komunikace mezi jednotlivými vrstvami aplikace a testování. Součástí je také git repositář, kde lze vidět, jak byla aplikace vyvíjena. Díky rebasování vznikl přehledný graf, viz Obrázek 7: Git historie demonstrační aplikace. Lze z něj na první pohled vidět, že správa administrátorů byla realizována mimo hlavní větev a až po jejím dokončení s ní byla sloučena. Také v každé verzi je aplikace plně funkční a schopna svého sestavení, takže je možné v ní vyhledávat, kdy se nějaká chyba poprvé vyskytla pomocí *git bisect*.



Obrázek 7: Git historie demonstrační aplikace

8.1 Instalace a sestavení

Instalace samotné aplikace by měla být co nejjednodušší, jednak to zrychlí nové instalace a čím více automatizace a méně manuálního zásahu je třeba, tím menší prostor je pro vznik chyby. Aplikace má soubor *readme.txt*, kde je popsán celý proces instalace ve čtyřech krocích. Druhý krok stáhne veškeré závislosti včetně zvoleného frameworku a knihoven v přesných verzích a dotáže se na základní konfiguraci jako připojení k databázi. Jakmile je vše staženo, tak už stačí podle třetího kroku aplikaci sestavit. Tato akce kromě vytvoření potřebných složek a souborů také zkontroluje, zda aplikace splňuje veškeré standardy a zda jsou všechny testy v pořádku. Tuto akci provádíme vždy, když si chceme toto vše ověřit a také by se měla provádět vždy, když dojde k aktualizaci aplikace tak jak popisuje kapitola 4.5.

Vše co se během tohoto kroku provádí, je uloženo v souboru *build.xml* a ve složce *build* jsou uloženy konfigurace a pravidla pro nástroje, které provádí kontrolu a opravu zdrojových kódů. Tyto akce se provádějí pouze na zdrojové kódy samotné aplikace, které se nacházejí ve složce *src*, takže zdrojové kódy třetích stran nemají na tuto kontrolu vliv. Jakmile bychom ale v nějakém našem souboru měli například proměnnou o jednom znaku, tato akce selže, popřípadě kdybychom ji spouštěli na integračním serveru, ihned se to dozvíme. Pro účely ladění při vývoji je možné tyto akce spouštět samostatně. Jejich seznam zjistíme příkazem „*phing -l*“ a můžeme je spouštět pomocí „*phing {název akce}*“.

8.2 Architektura a návrh aplikace

Architekturu a jednotlivé vrstvy nejlépe reprezentuje model *Administrator*, který se nachází v cestě *src/Kadlec/AppBundle/Model/Administrator*. Nalezneme zde entitu administrátora, ale hlavně *AdministratorFacade*, která představuje rozhraní pro veškerou komunikaci s tímto modelem, tudíž nikde jinde není využívána servisa ani repositář tohoto modelu. Využití tohoto modelu lze vidět na dvou místech. A to v kontroleru AdministratorController, kde je prohlížení, vytváření, editace a mazání administrátorů. A jelikož v demonstračních datech, které se nacházejí v *AdministratorDataFixture,* je třeba také vytvářet administrátora, používá se zde úplně stejná metoda pro jeho vytvoření. Tím je zajištěno, že aplikace se v různých místech vstupu chová identicky.

Tento model obsahuje jedinou část, která obsahuje doménovou logiku a to je servisa *AdministratorService*, tudíž právě na tuto jedinou třídu je napsán jednotkový test *AdministratorServiceTest*. Právě díky oddělení persistence od této vrstvy jsme schopni snadno otestovat všechny důležité části, jako nemožnost vytvořit dva administrátory se stejným jménem nebo změnu hesla pouze v případě, že je během editace heslo skutečně vyplníme. Repositář obsahuje natolik základní operace, že testy není třeba na tuto třídu vytvářet.

Všechny vlastní třídy v aplikaci dodržují principy DRY a SOLID a jejich závislosti jsou řešeny pomocí dependency injections, konkrétně constructor injection. Je zde také použit autowiring, nicméně řešení stále vyžaduje povinně konfigurační soubor. Při pohledu na tento konfigurační soubor *src/Kadlec/AppBundle/Resources/config/services.yml* je zřejmé, že již teď při tak malém rozsahu obsahuje spoustu relativně zbytečného obsahu. Proto navrhované řešení na konci kapitoly 6.2.1, které by většinu této konfigurace generovalo automaticky, by práci značně usnadnilo a zpřehlednilo.

Během implementace správy administrátorů také vznikla v aplikaci chyba (záměrně pro demonstraci řešení chyby), že se nebylo možné s administrátorem přihlásit, jelikož neměl v demonstračních datech vyplněné heslo. Tato chyba byla odstraněna v komitu *9e011d2* a zároveň byl napsán test, který kontroluje mimo jiné i přihlašování do aplikace pod testovacím účtem, tudíž podobná chyba by již neměla nastat. Jedná se o funkcionální test, který již testuje

aplikaci ve zpracování celého požadavku. Pro tento účel jsem zde použil Symfony Test Client (9), který nevyžaduje žádné speciální softwarové vybavení serveru a pro tento účel zcela postačuje. Navíc se u takovýchto testů dají jednotlivé požadavky zaobalit do transakce a není tak třeba pro každý jednotlivý test nahrávat celou databázi znovu.

9 Závěr

Mnoho principů zde uvedených neplatí pouze pro jazyk PHP a dají se uplatnit nehledě na zvolený jazyk. Některé věci se netýkají ani přímo programování, ale řeší vybavení serveru nebo procesy, které jsou pro vývoj enterprise aplikace nezbytné. Poukazuji na nedostatky PHP, přičemž právě chystaná verze PHP 7 některé z nich bude řešit a také by měla přinést znatelný nárůst výkonu. Ale i bez této poslední verze PHP, jsem navrhl architekturu a celý ekosystém okolo, který lze v PHP realizovat a díky němu lze vyvíjet složité aplikace, které je možné dále dlouhodobě rozvíjet a pracovat s nimi.

Demonstrační implementace slouží pouze jako demonstrace možností a ověření některých principů. Postupy, které nejsou zahrnuty v demonstrační implementaci, mám nicméně ověřené z reálných projektů a jsou také realizovatelné. Proto mohu považovat PHP za jeden z vhodných jazyků pro tvorbu enterprise aplikace a to také díky nástrojům a ostatnímu softwaru, který máme k dispozici, ale také pro dostatečnou robustnost jazyka, ve kterém lze realizovat kvalitní a udržovatelnou architekturu.

10 Reference

1. *Nginx.* [Online] 2015. [Citace: 19. duben 2015.] http://nginx.org.

2. October 2014 Web Server Survey. *Netcraft.* [Online] 2014. [Citace: 11. listopad 2014.] http://news.netcraft.com/archives/2014/10/10/october-2014-web-server-survey.html.

3. Úvod do grafové databáze NEO4J. *Webexpo.* [Online] 2012. [Citace: 20. prosinec 2014.] http://webexpo.cz/praha2012/prednaska/uvod-do-grafove-databaze-neo4j/.

4. Type Hinting v PHP, má to řešit framework. *Nette fórum.* [Online] 2011. [Citace: 16. duben 2015.] http://forum.nette.org/cs/6934-type-hinting-v-php-ma-to-resit-framework.

5. **Purchart, Václav.** *Archív diplomových prací ČVUT.* [Online] 2014. [Citace: 11. duben 2015.] https://dip.felk.cvut.cz/browse/pdfcache/purchva1_2014dipl.pdf.

6. *HHVM.* [Online] 2015. [Citace: 3. květen 2015.] http://hhvm.com.

7. **Spolsky, Joel.** The Joel Test: 12 Steps to Better Code. [Online] 2000. [Citace: 16. duben 2015.] http://www.joelonsoftware.com/articles/fog0000000043.html.

8. *The Phing Project.* [Online] [Citace: 16. duben 2015.] https://www.phing.info.

9. *PHPUnit.* [Online] [Citace: 16. duben 2015.] https://phpunit.de.

10. Working with the Test Client. *Symfony Documentation.* [Online] 2015. [Citace: 16. duben 2015.] http://symfony.com/doc/current/book/testing.html#working-with-the-test-client.

11. *Selenium.* [Online] 2015. [Citace: 16. duben 2015.] http://www.seleniumhq.org.

12. *PhantomJS.* [Online] 2015. [Citace: 16. duben 2015.] http://phantomjs.org.

13. *Codeception.* [Online] 2015. [Citace: 16. duben 2015.] http://codeception.com.

14. *Composer.* [Online] 2015. [Citace: 17. duben 2015.] https://getcomposer.org.

15. *Symfony.* [Online] 2015. [Citace: 18. duben 2015.] http://symfony.com.

16. Doctrine 2 ORM's documentation. [Online] 2015. [Citace: 18. duben 2015.] http://doctrine-orm.readthedocs.org/en/latest/.

17. **Fowler, Martin.** AnemicDomainModel. [Online] 2003. [Citace: 10. říjen 2014.] http://martinfowler.com/bliki/AnemicDomainModel.html.

18. —. Inversion of Control Containers and the Dependency Injection pattern. [Online] 2014. [Citace: 18. duben 2015.] http://martinfowler.com/articles/injection.html.

19. —. Application Facades. *Martin Fowler.* [Online] [Citace: 16. leden 2015.] http://martinfowler.com/apsupp/appfacades.pdf.

20. —. *Patterns of enterprise application architecture.* Boston : Addison-Wesley Professional, 2003. ISBN 0321127420.

21. **Arvin, Troels.** Comparison of different SQL implementations. [Online] 2014. [Citace: 20. prosinec 2014.] http://troels.arvin.dk/db/rdbms/.

22. The Principles of OOD. *UncleBob.* [Online] 2005. [Citace: 10. listopad 2014.] http://www.butunclebob.com/ArticleS.UncleBob.PrinciplesOfOod.

23. Examining the Agile Cost of Change Curve. *Agile Modeling.* [Online] [Citace: 14. duben 2014.] http://www.agilemodeling.com/essays/costOfChange.htm.

11 Přílohy na CD/DVD

1. Ukázková implementace aplikace na navrhované architektuře – DemoApp
2. Tato práce ve formátu *.docx – kad0019_DP.docx
3. Tato práce ve formátu *.pdf – kad0019_DP.pdf