

Vysoká škola ekonomická v Praze

Fakulta informatiky a statistiky

Katedra informačních technologií

Studijní program: Aplikovaná informatika

Obor: Informační systémy a technologie

Dockerizace Continuous Integration řešení pro aplikace vyvíjené v programovacím jazyce Java

DIPLOMOVÁ PRÁCE

Student : Bc. Michal Okosy

Vedoucí : Ing. Jarmila Pavlíčková, Ph.D.

Oponent : Ing. Martin Batelka

2017

Prohlášení:

Prohlašuji, že jsem diplomovou práci zpracoval samostatně. Použitou literaturu a podkladové materiály uvádím v příloženém seznamu literatury.

V Praze dne 11. prosince 2017

.....

Michal Okosy

Poděkování

Na tomto místě bych rád poděkoval vedoucí práce, Ing. Jarmile Pavlíčkové, Ph.D., za nasměrování v úvodní části a dále především za užitečné rady a připomínky při vedení této práce.

Dále bych zde rád poděkoval své rodině za neutuchající podporu během celého studia.

Abstrakt

Diplomová práce „*Dockerizace Continuous Integration řešení pro aplikace vyvíjené v programovacím jazyce Java*“ se zabývá problematikou Continuous Integration a technologií Docker. Jejím hlavním cílem je analyzovat aplikace a technologie vhodné při použití CI u Java projektů a pomocí platformy Docker implementovat konkrétní řešení, které bude postavené na těchto technologiích.

V teoretické části je popsán a analyzován pojem Continuous Integration a s ním související pojmy – Continuous Delivery, Continuous Deployment a DevOps. V souvislosti s Continuous Integration je zde také popsán model kvality softwaru dle norem řady ISO/IEC 250nn, přičemž největší důraz je kladen na jeho dvě charakteristiky – udržitelnost a přenositelnost.

Praktická část je rozdělena do dvou kapitol. První z nich vychází z poznatků v teoretické části. Na jejich základě jsou identifikovány typy aplikací a nástrojů vhodných při aplikování CI na Java projekty. V rámci každého typu jsou aplikace vzájemně porovnány a dále je vybrána ta nejvhodnější. Pro takto vybrané aplikace jsou vytvořeny a propojeny Docker obrazy. Praktickým výstupem práce je CI řešení postavené na technologii Docker vhodné především pro Java projekty.

Klíčová slova

Continuous Integration, Continuous Delivery, Continuous Deployment, DevOps, Docker, Jenkins, GitLab, SonarQube

Abstract

Diploma thesis „*Dockerization of Continuous Integration for applications developed in Java programming language*“ deals with Continuous Integration and with technology Docker. The main aim of this thesis is to analyze applications and technologies suitable for implementing CI in Java projects and using Docker platform to implement solution, that will be based on analysed technologies.

In the theoretical part of the thesis a concept of Continuous Integration is described and analysed together with other similar concepts – Continuous Delivery, Continuous Deployment and DevOps. In the theoretical part there is also described software quality model according to ISO/IEC 250nn, with greatest emphasis being given to its two characteristics – maintainability and portability.

The practical part consists of two chapters. First of the two chapters is based on the knowledge described in the theoretical part – there are identified types of applications and tools that are useful in applying CI to Java projects. Within each type, the applications are compared to each other and the most appropriate one is selected. For these selected applications Docker images are created and linked together. Practical output of the work is CI solution based on Docker technology, suitable especially for Java projects.

Keywords

Continuous Integration, Continuous Delivery, Continuous Deployment, DevOps, Docker, Jenkins, GitLab, SonarQube

Obsah

Obsah	iv
Seznam obrázků.....	vii
Seznam tabulek.....	viii
Seznam výpisů programů	ix
1 Úvod	10
1.1 Cíle práce.....	10
1.2 Cílová skupina	10
1.3 Použité metody	11
1.4 Struktura práce	11
2 Komentovaná rešerše informačních zdrojů	13
2.1 Odborné publikace	13
2.2 Online zdroje	13
2.3 Akademické práce	14
3 Continuous Integration	17
3.1 Vymezení pojmu Continuous Integration	17
3.2 Přínosy Continuous Integration	18
3.2.1 Měřitelnost	18
3.2.2 Včasné odhalení chyb.....	19
3.2.3 Zvýšení kvality kódu.....	19
3.2.4 Vždy nasaditelná aplikace	21
3.2.5 Přínosy dle studie	22
3.3 Proces a součásti Continuous Integration	22
3.3.1 Základní proces Continuous Integration	23
3.3.2 Inicializace sestavení	24
3.3.3 Kompilace	25
3.3.4 Integrace databáze	25
3.3.5 Testování	27
3.3.6 Inspekce	30
3.3.7 Poskytnutí okamžité zpětné vazby.....	32
3.3.8 Nasazení.....	34
3.3.9 Rozšířený proces Continuous Integration.....	34
3.4 Pojmy související s Continuous Integration	36
3.4.1 Model kvality softwaru dle norem řady ISO/IEC 250nn.....	36
Charakteristika: Funkční vhodnost (Functional suitability)	40
Charakteristika: Výkonnostní účinnost (Performance efficiency)	40

Charakteristika: Kompatibilita (Compatibility)	40
Charakteristika: Použitelnost (Usability)	40
Charakteristika: Spolehlivost (Reliability)	41
Charakteristika: Bezpečnost (Security)	41
Charakteristika: Udržovatelnost (Maintainability)	41
Charakteristika: Portabilita (Portability)	42
3.4.2 Metriky pokrytí kódu testy	42
3.4.3 Cyklomatická složitost.....	45
3.4.4 Continuous Delivery, Continuous Deployment.....	47
3.4.5 DevOps.....	48
4 Porovnání a výběr aplikací a nástrojů	50
4.1 Metoda hodnocení a výběru	50
4.2 Continuous Integration servery.....	53
4.2.1 Kritéria a stanovení vah	54
Aktivita na projektu	54
Funkce.....	55
Licence	55
Možnost placené podpory	55
Předpřipravený podporovaný Docker obraz.....	55
Rozšiřitelnost.....	56
Velikost uživatelské základny	56
Stanovení vah kritérií.....	56
4.2.2 Jenkins.....	57
4.2.3 Hudson	58
4.2.4 GitLab CI.....	59
4.2.5 GoCD.....	60
4.2.6 Hodnocení.	61
4.3 Systémy správy verzí	61
4.3.1 Kritéria a stanovení vah	63
Funkce.....	63
Výkon.....	63
Stanovení vah kritérií.....	64
4.3.2 GitBucket	64
4.3.3 Gogs (Gitea)	65
4.3.4 GitLab	66
4.3.5 RhodeCode.....	66
4.3.6 Hodnocení	67
4.4 Nástroje pro analýzu kódu	68
4.4.1 Kritéria a stanovení vah	69
Funkce.....	69
Aktivita na projektu	69
Integrace s Jenkins	69

Licence	70
Možnosti kastomizace	70
Stanovení vah kritérií	70
4.4.2 Cobertura	71
4.4.3 Codacy	71
4.4.4 FindBugs	72
4.4.5 JaCoCo	72
4.4.6 JavaNCSS	73
4.4.7 PMD	73
4.4.8 SonarQube	74
4.4.9 Hodnocení	74
5 Praktické řešení	76
5.1 Technologie Docker	76
5.2 Jenkins Docker obraz	78
5.3 GitLab Docker obraz	80
5.4 SonarQube Docker obraz	82
5.5 Uživatelská příručka administrátora	83
5.5.1 Základní obsluha a změna konfigurace	83
5.5.2 Obsluha	84
5.6 Uživatelská příručka vývojáře	85
5.6.1 Základní otestování běžící aplikace	85
5.6.2 Příklad práce na ukázkovém projektu	85
5.6.3 Rozšíření o další projekty	89
6 Závěr	90
Terminologický slovník	91
Použité informační zdroje	94

Seznam obrázků

Obrázek 1: Schéma procesu CI (zdroj: Duval et al., 2007, s. 5)	23
Obrázek 2: Schéma rozšířeného procesu CI (zdroj: Duval et al., 2007, s. 224)	35
Obrázek 3: Divize norem řady ISO/IEC 25000 (zdroj: ISO/IEC 25010, 2011)	37
Obrázek 4: Kauzální řetězec kvality v životním cyklu softwarového produktu (zdroj: ISO/IEC 25010, 2011).....	40
Obrázek 5: Srovnání CI, CDL, CDP (zdroj: RightScale 2014, s. 8).....	47
Obrázek 6: Životní cykly Docker obrazů a kontejnerů (zdroj: Mazin 2014)	77
Obrázek 7: Výsledky junit testů (zdroj: autor).....	86
Obrázek 8: Komentář ke GitLab merge requestu ze SonarQube analýzy (zdroj: autor)	87
Obrázek 9: Část výsledku SonarQube analýzy pro ukázkový projekt (zdroj: autor).....	88
Obrázek 10: Detail pokrytí testy třídy TaskController (zdroj: autor)	88

Seznam tabulek

Tabulka 1: Normy spadající do řady norem ISO/IEC 25000 (zdroj: International Organization for Standardization, 2017)	38
Tabulka 2: Kombinace možností testování podmínky z výpisu 5 (zdroj: autor)	44
Tabulka 3: Příklad preferencí kritérií (zdroj: autor)	52
Tabulka 4: Výpočet vah jednotlivých kritérií (zdroj: autor)	52
Tabulka 5: Váhy jednotlivých kritérií CI serverů (zdroj: autor)	57
Tabulka 6: Hodnocení CI serverů (zdroj: autor)	61
Tabulka 7: Váhy jednotlivých kritérií VCS (zdroj: autor)	64
Tabulka 8: Hodnocení VCS (zdroj: autor)	68
Tabulka 9: Váhy jednotlivých kritérií nástrojů pro analýzu kódu (zdroj: autor)	70
Tabulka 10: Hodnocení SCA nástrojů (zdroj: autor)	75

Seznam výpisů programů

Výpis 1:	Příklad programu Hello World (zdroj: autor).....	20
Výpis 2:	Příklad programu Hello World 2 (zdroj: autor).....	20
Výpis 3:	Užití frameworku Mockito při unit testování (zdroj: autor)	28
Výpis 4:	Komponentový test (zdroj: autor)	29
Výpis 5:	Testovaná metoda pro výpis odměn zaměstnancům (zdroj: autor).....	43
Výpis 6:	Ukázková metoda pro určení cyclomatické složitosti (zdroj: autor).....	46
Výpis 7:	Metoda pro součet složených čísel (zdroj: Hummel, 2014).....	47
Výpis 8:	Seznam souborů pro vytvoření Jenkins Docker obrazu (zdroj: autor).....	78
Výpis 9:	Dockerfile pro Jenkins Docker obraz (zdroj: autor)	79
Výpis 10:	Seznam souborů pro vytvoření Jenkins GitLab obrazu (zdroj: autor)	81
Výpis 11:	Dockerfile pro Gitlab obraz (zdroj: autor)	81
Výpis 12:	Dockerfile pro SonarQube obraz (zdroj: autor)	82
Výpis 13:	Seznam souborů pro vytvoření Jenkins GitLab obrazu (zdroj: autor)	83

1 Úvod

V této diplomové práci se zabývám především tématem Continuous Integration (dále jen CI). V teoretické části je tento pojem vymezen a dán do souvislosti s dalšími s ním spojenými pojmy. Druhým tématem, kterým se v práci také zabývám, je platforma Docker, která je alternativou k plným virtualizačním technologiím.

Tato dvě témata jsou spojena v praktické části, kde jsou nejprve porovnány a vybrány nejvhodnější aplikace vhodné pro aplikování CI u Java projektů. Tyto aplikace jsou pak součástí praktického výstupu z práce – CI řešení pro Java projekty.

Osobní motivací k výběru tohoto tématu je můj hlubší zájem o technologii Docker, která je v současné době stále více užívána. Druhým důvodem je to, že mi jako Java vývojáři chybí hotové přenositelné komplexní CI řešení pro projekty, které bych mohl jedním kliknutím nainstalovat a provozovat na vlastním VPS serveru.

1.1 Cíle práce

Hlavním cílem této práce je analyzovat aplikace a technologie vhodné při použití CI u Java projektů a pomocí platformy Docker implementovat konkrétní řešení, které bude postavené na těchto technologiích. Tento hlavní cíl sestává ze tří dílčích podcílů:

- 1 Objasnit pojem CI a zasadit jej do širšího kontextu – především pak v souvislosti s pojmy, jako jsou Continuous Delivery (dále jen CDV), Continuous Deployment (dále jen CDP) a DevOps.
- 2 Analyzovat typy aplikací a technologií vhodných a běžně užívaných při CI u Java projektů. Jednotlivé aplikace, jež jsou zdarma a bez omezení použitelné i na komerčních projektech, v rámci typu vzájemně porovnat a vybrat nejvhodnější aplikaci, která bude součástí CI řešení.
- 3 Za použití výstupů z předchozího podcíle implementovat integrované řešení pro CI. Nutnou podmínkou pro splnění tohoto podcíle je, aby všechny aplikace daného řešení běžely v Docker kontejnerech.

1.2 Cílová skupina

Cílovou skupinou této práce jsou především Java vývojáři, kteří mají zájem aplikovat CI u svých, i komerčních, projektů a přitom nechtějí, nebo nemohou, utrácet peníze za existující placená řešení v této oblasti. Nejsou zde přitom nutné velké zkušenosti s technologií Docker.

Výsledné řešení bude schopen použít, nainstalovat a spustit i člověk, který má jen menší zkušenosti s používáním linuxové příkazové řádky.

1.3 Použité metody

Pro splnění cílů je v teoretické části použita rešerše a analýza zdrojů dostupných a zabývajících se popisovanými tématy. Především na základě analýzy CI jsou identifikovány skupiny aplikací vhodných pro provozování CI u Java projektů. V každé z takto identifikovaných skupin jsou pak vybrány nejvíce rozšířené aplikace, které jsou použitelné v rámci omezení plynoucích z cílů práce. Vybrané aplikace jsou pak pomocí rozšířené metody Fullerova trojúhelníku vzájemně porovnány a vybrána ta nejvhodnější. Pro tyto aplikace jsou vytvořeny Docker obrazy (pojem Docker obraz je blíže vysvětlen v kapitole 5) a skripty, pomocí nichž je možné je nainstalovat spuštěním jediného příkazu na dostatečně výkonných strojích, kde už je nainstalovaný Docker. Aplikace je pak možné ihned používat bez nutnosti dalšího manuálního nastavení.

1.4 Struktura práce

Diplomová práce je rozčleněna do šesti kapitol. Po této, první úvodní kapitole, kde jsou definovány cíle, metody a struktura práce, jsou ve druhé kapitole představeny existující zdroje a absolventské práce věnující se podobné tematicce – jsou zde obsaženy především zdroje, které se zabývají CI a technologií Docker.

V kapitole 3 (Continuous Integration) je splněn první cíl – je zde popsáno CI a zasazeno do širšího kontextu, jsou zde probrány i pojmy, které s CI úzce souvisejí – Continuous Deployment, Continuous Delivery, DevOps. Je zde rovněž popsán model kvality softwaru dle norem řady ISO/IEC 250nn. U něj je větší pozornost věnována především dvěma charakteristikám – udržitelnosti a přenositelnosti, jelikož ty s tématem práce souvisí nejvíce.

V kapitole 4 jsou porovnávány jednotlivé aplikace za pomoci rozšířené metody Fullerova trojúhelníku; důvody pro užití rozšířené verze této metody jsou specifikovány na počátku kapitoly. Výstupem z této kapitoly je seznam nejvhodnějších aplikací, které dohromady mohou tvořit komplexní CI řešení.

V první části kapitoly 5 je popsána technologie Docker v rozsahu nutném k pochopení praktické části. Jen menší důraz je věnován ryze teoretickým základům této technologie, jelikož, jak vyplynulo z rešerše, teoretickými základy a principům fungování Dockeru se zabývalo menší množství i v Česku psaných absolventských prací. V páté kapitole jsou dále popsána specifika vytvořených a upravených Docker obrazů. V závěrečné části kapitoly jsou

dvě příručky – příručka administrátora a programátora. V první z nich je popsáno, jak dané řešení instalovat, spouštět a konfigurovat. Druhá je věnována pohledu ze strany běžného Java vývojáře, jenž Java kód – menší MVC aplikaci, jejíž zdrojové kódy jsou též součástí přílohy – nahraje do úložiště, a poté se chce podívat na výsledky testů a analýz, které na kódu byly provedeny.

V závěrečné šesté kapitole je shrnuto splnění cílů a jsou popsány přínosy této diplomové práce.

2 Komentovaná rešerše informačních zdrojů

Problematika CI je rozebírána ve značném množství odborných i akademických publikací. Rovněž technologie Docker není okrajové téma a je hlavním tématem mnoha prací. Kombinace těchto dvou témat je však relativně vzácná, nevzniklo mnoho publikací zkoumajících právě tato dvě témata.

V této kapitole jsou vypracovány rešerše obdobných publikací. V podkapitole 2.1 jsou popsány knihy věnující se dané tematice. V podkapitole 2.2 jsou online zdroje. V rozsáhlejší podkapitole 2.3 jsou shrnuty vzniklé akademické práce zabývající se obdobnými tématy.

2.1 Odborné publikace

Při popisu základních principů, výhod a tzv. „best practices“ v oblasti CI bude základním literárním pramenem, ze kterého budu vycházet, publikace „Continuous Integration: Improving Software Quality and Reducing Risk“ (Duval et al., 2007). V této knize nejsou popsány jen teoretické základy, ale především v kapitole 4 (Building Software at Every Change) a v kapitole 6 (Continuous Testing) je shrnuto mnoho poznatků a konkrétních doporučení ohledně CI, ze kterých bude vhodné čerpat i v praktické části práce.

Doplňkovým literárním pramenem bude publikace „DevOps for Developers: Integrate Development and Operations the Agile Way“ (Hüttermann, 2012), především pak kapitola 2, která je věnována vysvětlení pojmu DevOps.

2.2 Online zdroje

V teoretické části budu při popisu CI, kromě výše zmíněné tištěné literatury, vycházet z článku Continuous Integration od Martina Fowlera (Fowler, 2006). V tomto článku jsou popsány nejen důvody, proč CI v projektech zavádět, ale jsou zde i rady, jak s CI začít, či další doporučení a pravidla, jež je vhodné dodržovat. Tento článek je tak ideálním doplňkem k hlavní odborné publikaci uvedené výše, ke které mimochodem Martin Fowler napsal úvodní slovo.

V první části implementační části práce, kde budou porovnávány alternativy v jednotlivých skupinách softwaru, se budu snažit vycházet vždy z oficiální dokumentace k danému softwaru. V případech, kde bude dokumentace nedostatečná, či zcela chybějící, budu vycházet z odborných (akademických či neakademických) článků. Za zdroj neakademických odborných článků uvádím například server root.cz.

V implementační části práce, kde se budu zabývat instalací CI prostředí za použití technologie Docker, bude hlavním podkladem především oficiální dokumentace (Docker, 2017a).

2.3 Akademické práce

Vzhledem k tomu, že CI není žádnou novinkou a de facto se stává standardem ve vývoji softwaru, a především v agilně řízených projektech, se lze bez implementovaného CI obejít jen stěží (Radigan, 2017), není překvapivé, že již vzniklo několik bakalářských a diplomových prací zabývajících se tématy souvisejícími s CI.

Svým zaměřením ohledně CI nejbližší, přesto od mé práce stále vzdálená, je bakalářská práce „Kontinuální integrace při vývoji webových aplikací v PHP“ Martina Hujera z Fakulty informatiky a statistiky, VŠE (Hujer, 2011). Zde v praktické části práce autor implementuje konkrétní řešení pro kontinuální integraci PHP projektů. K tomu autor používá build server Jenkins, který dále integruje s dalšími PHP nástroji – PHP CodeSniffer, PHP CPD, PHP LOC, PHP Depend, PHP MD, PHPUnit atd.

Tato práce se od mé liší hned v několika ohledech – autor nepoužívá virtualizaci ani kontejnerizaci. Dále celé řešení je soustředěno především na PHP projekty, na rozdíl od mé práce, která se soustředí na vývoj Javových projektů. Součástí této práce rovněž nebyla instalace a následná integrace jednotného úložiště zdrojových kódů ani systému pro řízení závislostí. Chybí zde také porovnání alternativ – například Jenkins je zde srovnán s Hudsonem jen z hlediska aktivity na GitHubu, nikoliv z hlediska nabízených funkcí; ostatní alternativy (např. Atlassian Bamboo) jsou zde pouze zmíněny bez dalšího srovnání.

V diplomové práci „Use of continuous integration in web application development“ (Mráz, 2013) se autor zabývá více teoretickou a procedurální stránkou CI. Autor zde nejprve popisuje rigorózní metodiky na konkrétním příkladu jedné z nich – RUP. Dále obecně popisuje agilní vývoj, detailněji pak agilní metodiku Scrum. V podkapitole věnované agilnímu vývoji vyzdvihuje důležitost a přínosy CI. V kapitole 3 pak autor na čtyřech stranách popisuje samotný proces a praktiky CI. V praktické části je rozebrán proces a jednotlivé nástroje (mj. TeamCity jako build server), které byly používány při jeho práci pro společnost Seznam.cz, a.s., konkrétně pro jeho divizi, která byla zodpovědná za vývoj internetového portálu „Proženy.cz“.

Jelikož má práce spojuje dvě hlavní témata – praktiky CI a ekosystém Docker – je nutné shrnout současný stav poznání i ve světě Dockeru. Ačkoliv se jedná o poměrně novou technologii – první verze Dockeru byla veřejnosti představena na konferenci PyCon 2013 (Hykes, 2013) – vzniklo i v Česku již několik akademických prací i na toto téma.

Pro čtenáře, který o ekosystému Docker ani Linuxových kontejnerech obecně mnoho neslyšel a rád by se v tomto směru lépe orientoval, doporučuji diplomovou práci

„Virtualizace pomocí platformy Docker“ (Jurenka, 2015). Autor této anglicky psané práce, Vladimír Jurenka, v ní ekosystém Docker porovnává s možná rozšířenějšími nástroji pro virtualizaci – Oracle VM Virtualbox a VMware Workstation player, které umožňují tzv. plnou virtualizaci. Dále se pak autor soustředí na popis ekosystému Docker, ozřejmuje zde základní pojmy jako Docker Container, Docker Image, Docker Hub a další nástroje pro orchestraci – Docker Swarm, Docker Compose a Docker Machine. Při popisu principu fungování Dockeru zde autor jde poměrně do hloubky a popisuje zde Dockerem využívané relativně nové funkce Linuxového jádra – mj. namespaces (česky jmenné prostory) a cgroupy (z angl. control groups, česky kontrolní skupiny).

V implementační části autor na jednoduchém příkladu ukázal, jak jde mezi běžícími kontejnery kopírovat soubory pomocí standardního unixového programu scp. Mimo to vytvořil pull request do oficiálního Docker úložiště na Github s novým příkazem pro aktualizaci Docker obrazů. Bohužel autor neposkytl odkaz na daný pull request, a ten nebyl před dokončením práce ještě schválen. Z nynější oficiální dokumentace to ale vypadá, že pull request schválen nebyl. Příkaz docker update totiž dělá něco zcela jiného, než co implementoval autor ve své práci (Docker, 2017a).

Bakalářská práce (Tomášek, 2015) s podobným názvem jako předchozí práce – „Virtualizace serverů a aplikací pomocí Docker“ od Patrika Tomáška – nabízí také podobně podrobný popis platformy Docker včetně popisu a porovnání typů virtualizace. Na rozdíl od předchozí práce ale nerozebírá orchestrační nástroje ekosystému Docker. Právě proto vnímám předchozí práci jako vhodnější zdroj týkající se teoretické stránky a popisu technologie Docker. V této práci je ovšem zajímavější praktická část, ve které autor vytváří jednoduchý Docker obraz s nástrojem pro správu logů Octopussy. Tento obraz je následně nahrán do veřejného úložiště – Docker Hubu – odkud je stále dostupný.

Bakalářská práce „Systém kontinuální integrace pro rychlý vývoj webových aplikací“ od Jaroslava Fedora (Fedora, 2016) má na první pohled podobné zaměření jako moje diplomová práce, proto jí zde budu věnovat více pozornosti.

Vzhledem k rozsahu práce (necelých 19 normostran, včetně obsahu a bibliografie) zde chybí generická teoretická část. Celá práce se týká konkrétní implementace CI řešení pro firmu Software Development Europe, s.r.o. V úvodních kapitolách jsou popsány požadavky firmy a na základě nich vybrány technologie, které budou použity. Co zde postrádám a na co se tedy budu ve své práci více soustředit, je bližší srovnání jednotlivých technologií a aplikací v daných kategoriích použitelných pro CI. Například jako VCS nástroj autor bez bližšího zdůvodnění vybral GitLab.

Dalším aspektem, kde práce není příliš dotažena do detailu, je integrace celého řešení. Implementované řešení nainstaluje na cloud od firmy Open Nebula několik Docker kontejnerů (ve výchozím nastavení tři kontejnery – kontejner s GitLabem a k němu v samostatných kontejnerech dvě databáze – PostgreSQL a Redis; ostatní kontejnery s Jenkinsem, Jirou a Redminem jsou dle přiložených zdrojových kódů vypnuté). Ve

zdrojových kódech, v nichž je kvůli absenci komentářů ztížená orientace, zcela chybí napojení Jenkins na VCS či jakákoliv zmínka v práci, jak je propojit. Aby byl pokryt běžný případ užití CI, je zde nutno provést další manuální, v práci nepopsané kroky.

Jelikož se Jaroslav Fedor nesnažil vytvořit CI řešení pro projekty na platformě Java, pochopitelně zde vůbec neřešil specifické potřeby takovýchto projektů (např. jak nasadit aplikaci na testovací či funkční prostředí). Právě ty naopak rozeberu v této práci já.

Tematikou virtualizace, ale také i kontejnerizace, se zabývá Oliver Stríž ve své diplomové práci „Virtuálne laboratórium pre vývoj Business Intelligence riešení“ (Stríž, 2016). V teoretické části je opět popsána virtualizace obecně, kde je také porovnána s ekosystémem Docker. Implementační část – vytvoření zabezpečeného virtuálního prostředí pro vývojáře firmy Joyful Craftsmen, s.r.o. – v této práci není plněna pomocí Dockeru, nýbrž za využití plné virtualizace – konkrétně se autor rozhodl pro využití virtualizační platformy Hyper-V od Microsoftu.

3 Continuous Integration

Tato teoretická kapitola se věnuje naplnění prvního cíle této práce – je zde nejdříve představen a vysvětlen pojem CI, dále jsou pak rozebírány přínosy jeho zavádění. V dalších podkapitolách je podrobněji popsán samotný proces CI včetně možného rozšíření. V poslední podkapitole jsou rozebírány pojmy, které s CI úzce souvisí – CDV a CDP, je zde také popsán model kvality softwaru dle norem řady ISO/IEC 250nn – větší prostor je zde věnován především dvěma charakteristikám – udržitelnosti a přenositelnosti, jelikož tyto souvisí s předmětem této práce nejvíc.

3.1 Vymezení pojmu Continuous Integration

Pojem CI (z angl. „*Continuous Integration*“) se česky překládá jako „*průběžná integrace*“. Nejedná se o nový pojem, už v květnu roku 2006 publikoval na svém blogu Martin Fowler článek s prostým názvem *Continuous Integration* (Fowler, 2006). Zde autor CI definuje jako „...*software development practice where members of a team integrate their work frequently, usually each person integrates at least daily – leading to multiple integrations per day. Each integration is verified by an automated build (including test) to detect integration errors as quickly as possible*“ (česky: „...*techniku vývoje softwaru, kde členové týmu integrují svoji práci často, obvykle každý vývojář integruje svoji část alespoň jednou denně, což vede k více integracím za den. Každá integrace je verifikována automatizovaným sestavením (včetně spuštění testů) za účelem odhalení chyb v integraci co nejdříve to je možné*“, přeložil autor). Tato definice je patrně nejznámější a nejvíce rozšířená. Je relativně obecná, a tak nechává dostatek volného prostoru pro konkrétní implementaci. Říká totiž, jen co se má dělat, nediktuje ale, jak se to má udělat. Na druhou stranu je tato definice dostatečně konkrétní, zmiňuje například i potřebu automatizace sestavení i testování nebo i jeho četnost. V článku Martin Fowler samozřejmě zmiňuje i důvody, proč CI zavádět. Tyto důvody a přínosy zavádění CI rozebírám v této práci v kapitole 3.2.

Definice CI a pojem jako takový ovšem Martin Fowler nevymyslel. Prvním, kdo s termínem CI přišel a kdo bývá často uváděn jako jeho autor, je Grady Booch. Ten již v roce 1991 začíná termínu užívat. V roce 1996 pak napsal: „*At regular intervals, the process of ‘continuous integration’ yields executable releases that grow in functionality at every release... It is through these milestones that management can measure progress and quality, and hence anticipate, identify, and then actively attack risks on an ongoing basis.*“ (Booch, 1996, s. 75). Užití termínu CI je zde ještě volnější, než je definice CI od Martina Fowlera. Booch zde už ani nemluví přímo o integraci na denní bázi jako Martin Fowler. Spíše zde zdůrazňuje důležitost existence včasné integrace jako takové. V tomto významu jde tedy spíše o určitou podobu interaktivního modelu, který má zamezit tomu, aby se integrace v podobě tzv. velkého třesku provedla jen jednou, na konci projektu. Je tedy proto nutné zde

zdůraznit, že takto volně CI již dnes nebývá vnímáno. Pod pojmem kontinuální bývá vnímána integrace častá – někdy i provedená několikrát denně, ne integrace řešení pouze na konci každé iterace (např. u projektů, které jsou vyvíjeny iterativně – např. za pomoci metodiky RUP).

Pro úplnost zde zmíním definici, která je v jistém smyslu více konkrétní a která mluví v jazyku bližším vývojářům než obě dvě definice uvedené výše. CI dle (Fitzgerald a Stol, 2015) je: *„A typically automatically triggered process compromising inter-connected steps such as compiling code, running unit and acceptance tests, validating code coverage, checking coding standard compliance and building deployment packages. While some form of automation is typical, the frequency is also important in that it should be regular enough to ensure quick feedback to developers“*. Tato definice oproti Fowlerově definici obsahuje konkrétnější kroky, které dávají jasnější představu i při vývoji Java projektů, kterými se primárně zabývá tato diplomová práce. Kompilace kódu, spouštění Unit a akceptačních testů, kontrola pokrytí kódu testy a kontrola kódu vůči standardům, to vše by v ideálním případě mělo pomáhat při vývoji softwaru nejen kvůli časnému odhalení chyb díky tzv. funkčním testům, ale také k odhalení případného technického dluhu na projektu.

3.2 Přínosy Continuous Integration

Nyní, když jsem vymezil CI jako pojem, přejdu k přínosům a případně nedostatkům aplikace užívání CI na softwarovém projektu.

3.2.1 Měřitelnost

Dle Martina Fowlera (Fowler, 2006) je jedním z největších přínosů snížení rizika neúspěchu projektu. V projektech, kde se neintegruje průběžně, ale pouze na konci před dodáním zákazníkovi, nastává zcela zásadní problém – nikdo nemá nejmenší ponětí o tom, jak dlouho může taková integrace na konci projektu trvat. A když se na konci projektu integrovat začne, je obtížné odhadnout, jak daleko v procesu integrace se projekt nachází a kolik času přibližně ještě může zbývat. Tento „*slepý bod*“, jak jej označuje Fowler, navíc nastává v nejnevhodnější dobu – v závěrečné fázi, kdy je většina projektů oproti plánu stejně opožděna.

Podobný přínos ze zavádění CI uvádí i sám autor pojmu Grady Booch (Booch, 1996, s 75), viz citace v předchozí kapitole 3.1. Ten vyzdvihuje především to, že management takto může měřit průběh a kvalitu a díky tomu lépe pracovat s nastalými riziky.

Zavedení CI s řešením tohoto problému jednoznačně pomůže. Protože v případě, že se integruje již od začátku, je v každém okamžiku zřejmé, co funguje a co se ještě musí doprogramovat.

3.2.2 Včasné odhalení chyb

Dalším přínosem CI je dle Fowlera (Fowler, 2006) včasné odhalení chyb v kódu (tzv. bugů). Aby toto platilo, je ovšem nutné, aby bylo do CI zakomponováno i automatizované spouštění testů. Je klíčové, aby tyto testy byly dostatečně podrobné. Pokud je testů dostatek a kód je v rozumné míře pokryt Unit testy, na jejichž úspěšné vykonání navazuje spuštění dalších typů testů – např. tzv. *smoke testů* a dále *regresních testů*, vede skutečně zavedení CI k dřívějšímu odhalení chyb. Toto ovšem nevede nutně k jejich odstranění, ale minimálně to tomu napomáhá. Je mnohem snadnější opravit chybu v kódu brzy po jejím vzniku než ji opravovat poté, co je již měsíce součástí kódu. Mimo to má včasné odhalování chyb a jejich oprava silný psychologický efekt – pro vývojáře je snadnější opravovat vždy několik málo chyb průběžně, po dobu celé práce na projektu, než aby v závěrečné fázi projektu zjistil, že před nasazením do produkce je potřeba doopravit sto drobných chyb, které až nyní odhalily testy (Fowler, 2006). Stejný přínos – včasné odhalení defektů – zmiňuje i (Duval et al., 2007, s. 53-55). Zde je zmíněna nejen potřeba použít automatizované regresní testy, které odhalí chyby jako takové, ale zasadit se o to, že testy jsou vůbec napsané v dostatečné míře. K tomu slouží jednoduše měřitelná tvrdá metrika – pokrytí kódu testy (angl. *code coverage*). I když se rozhodně nedá říct, že vysoké pokrytí kódu Unit testy automaticky znamená kvalitní kód, a ani se jednoduše nedá určit, co to znamená „vysoké pokrytí“, je obecně vnímaná osmdesáti až devadesáti procentní hranice jako více než dostačující. Hodnoty blízké se ke stu procentům bývají naopak spíše až podezřelé a je pak ke zvážení kontrola těchto testů. Při ní se pak dá kontrolovat například to, zda tyto Unit testy vůbec něco testují – např. zda obsahují volání metod typu *assertEquals* (Fowler, 2012).

3.2.3 Zvýšení kvality kódu

(Duval et al., 2007, s. 57-61) uvádí jako další z přínosů CI také zvýšení kvality kódu. Zde uvádí hned tři konkrétní příklady problémových situací, kde CI může pomoci se zvýšením úrovně softwaru a kódu. Za prvé může dohlížet nad dodržováním základních kódovacích standardů – aby i kód psaný různými vývojáři splňoval stejná pravidla. Ať už jde například o stejné odsazení, řazení importů, modifikátorů proměnných apod. Viz Výpis 1 a Výpis 2, které zobrazují významově tentýž program, pokaždé však dodržující jiné stylistické standardy.

Výpis 1: *Příklad programu Hello World (zdroj: autor)*

```
1 package eu.okosy.dp.example.hw;
2
3 class HelloWorldApp {
4
5     private static final String MESSAGE = "Hello World!";
6
7     public static void main(String[] args) {
8         System.out.println(MESSAGE);
9     }
10
11 }
```

Zde je většina bloků oddělena prázdným řádkem, otevírací složené závorky nejsou na samostatném řádku, rovnítko je odděleno mezerou, bloky jsou odsazeny čtyřmi mezerami, název konstanty je psán velkými písmeny a modifikátory proměnných jsou seřazeny dle doporučení specifikace jazyku Java (Gosling et al., 2015).

Výpis 2: *Příklad programu Hello World 2 (zdroj: autor)*

```
1 package eu.okosy.dp.example.hw;
2 class HelloWorldApp
3 {
4     final static private String message="Hello World!";
5     public static void main( String[] args )
6     {
7         System.out.println(message);
8     }
9 }
```

Ve výpisu 2 je vše naopak: bloky odděleny prázdným řádkem nejsou, složené závorky na samostatném řádku jsou, mezi rovnítkem není mezera, odsazení bloku činí dvě mezery, název konstanty je malým písmem a modifikátory proměnné seřazeny jinak.

Když se tyto příklady zkompilují, tak jejich bajtkód sice nebude identický, nicméně při spuštění se oba dva programy budou chovat naprosto stejně. I přes to, že na chod programu to nemá žádný vliv, jsou mezi nimi zásadní rozdíly v čitelnosti. Toto ovšem není objektivní kritérium, každému programátorovi může přijít snáze čitelné něco jiného. V rámci projektu je ale vhodné, aby všechny kód odpovídal jednomu standardu. Není rozumné, aby několik tříd mělo např. otevírací složené závorky na prázdném řádku a několik ne. Jsou zde v zásadě dva přístupy, jak dodržování stejných standardů vynutit. První možností je pro projekt sepsat pravidla do dokumentu a například při revizích kódu (z angl. *code review*) manuálně dohlížet na to, že jsou dodržována. Zde je problém manuálního ručního kroku, ne vždy si „kontrolor“ musí všimnout odchylky od dokumentu a schválí pak někdy něco, co by neměl. Druhou možností je přidat do CI další krok, který by sám automaticky kontroloval standardy místo těchto ručních kontrol. K tomuto existuje mnoho nástrojů, kde už rozsáhlá pravidla jsou ve výchozím nastavení, a tak není třeba trávit mnoho času jejich konfigurací. Pro jazyk Java lze jmenovat SonarQube, Checkstyle, nebo PMD. Tyto nástroje jsou obvykle dále

konfigurovatelné. Pokud tedy některá pravidla neodpovídají potřebám projektu, je možné dopsat vlastní, případně upravit či vypnout ta, která tam jsou již od instalace.

Dalším častým neduhem, který souvisí s kvalitou kódu a kterým trpí mnoho projektů, je duplikování kódu napříč aplikací (Duval et al., 2007, s. 60-61). Toto nejde jednoduše zakázat a především ani jednoduše manuálně kontrolovat. I zde jako v předchozím případě však existuje celá řada nástrojů, které mohou kód kontrolovat. Jako příklad je uveden Simian nebo součást PMD – CPD (Copy Paste Detector). Z vlastní zkušenosti mohu uvést, že podobné kontroly se dají nastavit i v nástroji SonarQube. I zde platí, že pokud těmito nástroji bude kontrolován všechny nový kód a tato kontrola bude součástí CI, odstraní se i problém s duplikovaným kódem relativně snadno a nebude tak vznikat větší technický dluh.

S kvalitou kódu může souviset i dodržování pravidel stanovených architektem projektu. Ten by mohl zavést pravidlo, že controllery by neměly přímo modifikovat datovou vrstvu. Toto pravidlo opět nelze jednoduše ručně kontrolovat, ale opět existují nástroje, které lze nastavit, aby se o tuto kontrolu staraly automaticky. Jako příklad (Duval et al., 2007, s. 60) uvádí nástroje JDepend, případně NDepend pro projekty založené na platformě .NET. Z informací na veřejném úložišti zdrojového kódu nástroje JDepend na GitHubu je však patrné, že projekt je již nepodporovaný. V poslední době byl změněn jen Readme soubor, všechny ostatní soubory byly naposledy aktualizovány před 7 lety (Clarkware, 2017).

Pojem kvalita softwaru je blíže rozebrán v kapitole 3.4.1, kde je blíže popsán model kvality softwaru dle norem ISO/IEC 25000.

3.2.4 Vždy nasaditelná aplikace

V praxi mnohdy nastává problém, kdy je vyvíjená aplikace úzce spojená s nastavením systému či přímo vývojového prostředí jejích vývojářů. Když pak nastane moment, kdy je aplikaci potřeba nasadit do produkčního prostředí, nikdo neví, co všechno je potřeba nainstalovat, spustit a nastavit. Někdy se také může stát, že doimplementovaná funkcionality jednomu vývojáři na jeho pracovní stanici funguje, jinému ne. Stejně tak jednomu vývojáři mohou testy procházet a druhému nikoliv. Aby se co nejvíc odstínil vlivy nastavení jednotlivých vývojářů, je vhodné v rámci CI aplikaci nasazovat a vůči ní regresní testy pouštět na neutrálním prostředí. To se pak stane jediným zdrojem pravdy v případě konfliktů typu „ale na mém počítači to funguje“. Usnadní to samozřejmě i nasazení aplikace do produkce, protože skripty, které umožňují sestavení a nasazení dané aplikace na vývojové či testovací prostředí, již musí být hotovy a součástí CI (Duval et al., 2007, s. 49-53). V ideálním případě je pak jedinou akcí, která je od vývojářů při nasazení nové verze do produkce potřeba, kliknutí na správné tlačítko v CI serveru. Někdy ani toto není nutné a nová verze je nasazována automaticky po úspěšném proběhnutí všech testů, které byly puštěny opět automaticky po přidání kódu do úložiště. V tomto významu se už CI možná posouvá až na hranici CDV nebo CDP. Rozdíly a vztah mezi těmito třemi úzce spojenými termíny jsou podrobněji rozebrány v kapitole 3.4.4.

3.2.5 Přínosy dle studie

Názory a zkušenosti autorů z předchozích čtyř kapitol je vhodné podpořit (nebo vyvrátit) reálnými daty z reálných projektů. Tímto se zabývají Ståhl a Bosch (Ståhl a Bosch, 2013) ve své studii. V ní zkoumají celkem čtyři velké softwarové produkty. Tyto produkty jsou vyvíjeny čtyřmi různými, na sobě nezáviselými, týmy. Některé z těchto týmů mají zkušenosti s CI delší, některé CI zavedly nedávno. V práci autoři na základě analýzy literatury stanovují čtyři hypotézy, jejichž validitu se pak snaží prokázat na základě rozhovorů s dříve zmíněnými týmy. Tyto hypotézy jsou:

1. CI podporuje agilní přístupy k testování: vytváření automatizovaných akceptačních testů zákazníkem i psaní Unit testů ve spojení s psaním kódu.
2. CI přispívá ke zlepšení komunikace uvnitř i vně týmu.
3. CI přispívá ke zvýšení produktivity vývojářů jednak díky tomu, že mohou jednodušeji pracovat na více projektech současně, a jednak díky tomu, že již nemusejí software kompilovat a testovat na lokálních strojích.
4. CI zvyšuje předvídatelnost projektu díky tomu, že chyby v kódu jsou nacházeny průběžně a tím dříve (Ståhl a Bosch, 2013).

Za povšimnutí stojí, že se tyto hypotézy z velké části překrývají s přínosy, které uvádějí (Booch, 1996, s 75), (Duval et al., 2007, s. 49-53) a (Fowler, 2006) a které byly popsány v kapitolách 3.1-3.4. V práci se na základě rozhovorů s členy týmů první hypotézu potvrdit nepodařilo. Druhá hypotéza byla potvrzena s dodatkem, že zkušenosti v rámci týmů byly odlišné. Třetí hypotéza byla validována jen částečně – podařilo se potvrdit jen její první část. Čtvrtou hypotézu se podařilo validovat v celém jejím rozsahu. To je zajímavé kvůli tomu, že sám autor pojmu CI Grady Booch zmiňoval jako důvod pro zavádění CI právě předvídatelnost a měřitelnost. Přínos včasného odhalování chyb na projektu díky zavedení CI uvádí i (Fowler, 2006) a (Duval et al., 2007, s. 53-55) a je podrobněji rozebrán v kapitole 3.2.2.

3.3 Proces a součásti Continuous Integration

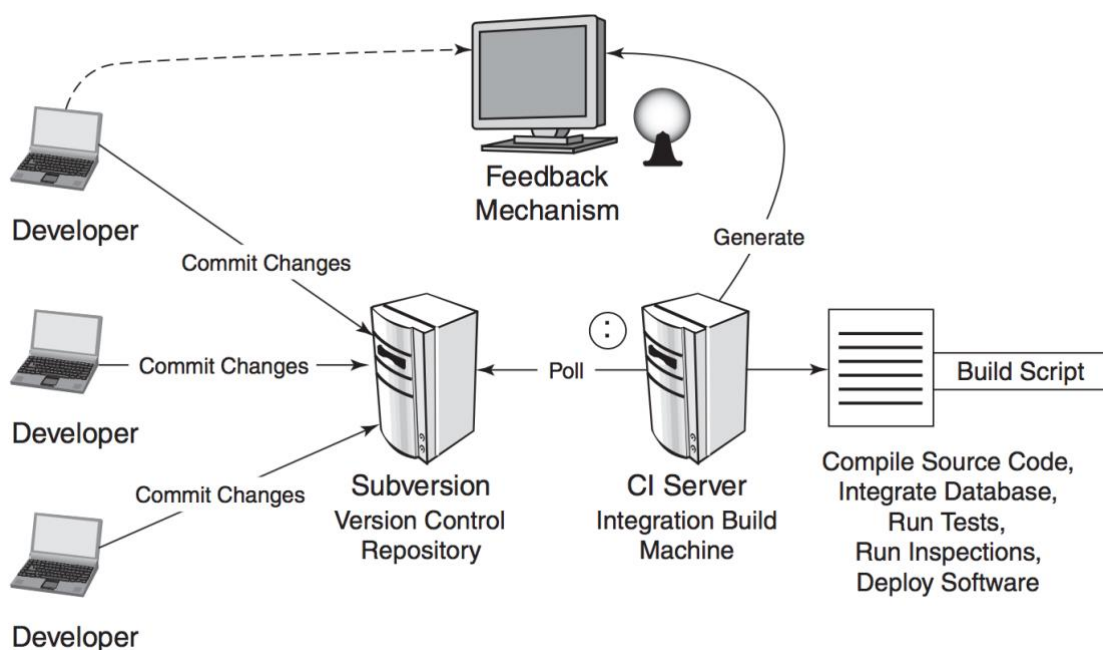
V kapitole 3.1 je sice rozebráno, co CI znamená dle definic různých autorů, přesto může být dojem z definic poněkud abstraktní. Pro konkrétnější představu je v této kapitole popsán samotný proces CI podrobněji a konkrétněji.

3.3.1 Základní proces Continuous Integration

Velmi zjednodušeně může celý proces CI probíhat nějak takto:

1. Vývojář nahraje nový zdrojový kód do úložiště.
2. CI server spustí sestavení.
3. Po skončení sestavení je vývojáři doručena zpětná vazba (např. výsledky testů).
4. Nová verze aplikace je nasazena (nepovinně).

Na obrázku 1 je tento proces zachycen.



Obrázek 1:

Schéma procesu CI (zdroj: Duval et al., 2007, s. 5)

Jedná se opravdu o značné zjednodušení. Například druhý krok – „CI Server spustí sestavení“ skýtá mnohé možnosti a způsoby inicializace tohoto sestavení – od manuálních, kdy se vývojář přihlašuje na CI server, po plně automatizované způsoby, kdy CI server sám nějakým způsobem komunikuje s úložištěm a na základě této komunikace se rozhodne spustit, nebo nespustit sestavení. Rovněž tak „sestavení“ je širší pojem a samo může sestávat z většího množství kroků. Podrobněji je proto tento proces rozebrán v kapitolách 3.3.2-3.3.8.

3.3.2 Inicializace sestavení

Vše začíná tím, že vývojář nahraje nový nebo změněný kód do sdíleného tzv. VCS – systému pro správu verzí (z angl. *Version Control System*). Ve většině případů nezáleží na tom, zda se jedná o distribuovaný (nejčastěji Git, nebo Mercurial) nebo centralizovaný (nejčastěji Subversion) VCS, protože většina CI serverů (viz další krok) by měla umět pracovat s oběma typy VCS.

V dalším kroku přichází na řadu již zmíněný CI server. To je hlavní část celého CI. CI server je zodpovědný za tzv. sestavení (angl. *build*). (Duval et al., 2007, s. 80-81) uvádí celkem čtyři možnosti, kdy může být sestavení inicializováno. Úplně nejprostší je sestavení „*on-demand*“, tzn. že každé sestavení je inicializováno manuálně – typicky se vývojář připojí na CI server, zadá parametry a klikne na příslušné tlačítko. Výhoda a zároveň nevýhoda tohoto postupu je právě v možnosti upravovat parametry sestavení. Pokud například vývojář ví, že jen změnil něco v databázi nebo jen přidával testy, nemusí zaškrtnout, že chce kód rekompilovat. Problémy tohoto přístupu však převažují. Z pohodlnosti či nezkušenosti může například opakovaně přeskakovat testy nebo krok *inspekce* kódu (viz dále). Dalším problémem je, že vývojáři se patrně zapomenou na CI server vůbec připojit a sestavení spustit, nebo to nebudou dělat po každé změně. Tento problém řeší další možnost inicializace sestavení – periodické sestavení.

Periodické sestavení probíhá v předem stanovený čas, například každou minutu, hodinu, nebo vždy o půlnoci. Výhody tohoto přístupu jsou zjevné na první pohled – se spouštěním je daleko méně práce. Nemůže se tedy stát, že by sestavení někdo zapomněl inicializovat, proběhne totiž vždy ve stanovený čas. Pokud je sestavení včetně spouštění testů a dalších kroků časově náročné, je možná vhodnější ho posunout na noční až ranní hodiny, kdy může být vytížení serveru menší než běžně přes den. Nevýhodou tohoto přístupu inicializace je neflexibilita. Na začátku projektu se někdo rozhodne, že sestavení bude probíhat například každou hodinu. Pokud ale pak vývojář svůj kód někdy nahraje do úložiště v 10:05, musí 55 minut čekat, než se mu aplikace sestaví a otestuje. To není příliš přívětivé. V lepším případě se pak alespoň může sám připojit na CI server a sestavení spustit ručně, stejně jako v předchozím případě. Mnohem větší problém však je, že sestavení proběhne vždy, bez ohledu na to, zda se v úložišti změnil kód či nikoliv. Proto jsou intervaly tohoto sestavení obvykle nastaveny na delší dobu, než je jedna minuta. V případě náročných sestavení by sestavování každou minutu bylo zbytečné přetěžování zdrojů.

Mnohem vhodnější je proto třetí způsob inicializace – CI server může periodicky pouze aktivně kontrolovat, zda se ve VCS objevila nová verze kódu (typicky může kontrolovat bez problémů klidně i každou minutu – například pokud je používán Git, tak stačí zavolat jednoduchý příkaz `git fetch origin master`). Sestavení CI server pak spustí pouze tehdy, pokud se dozví, že se v úložišti změnil zdrojové kódy. Tímto se odstraní problémy, které s sebou přinášely předchozí dva způsoby inicializace – vývojář nemusí, kromě nahrání kódu do úložiště, dělat nic ručně. Přesto se hned po nahrání tohoto kódu sestavení spustí takřka

ihned. Navíc zde nedochází k žádným zbytečným sestavením v případě nezměněného kódu jako v předchozím případě.

Poslední možnost způsobu inicializace, kterou (Duval et al., 2007, s. 81) udává, je událostmi řízená (angl. *event driven*) inicializace. Zde se užívá tzv. háku (častěji se užívá angl. verze výrazu – „hook“), kdy CI každou minutu sám nic nekontroluje jako v předchozích dvou případech, ale pokud se změní zdrojový kód, tak mu VCS sám předem definovaným způsobem sdělí, že tato změna kódu nastala. Tato možnost se se svými výhodami, respektive nevýhodami, téměř neliší od možnosti předchozí. Také v tomto případě je sestavení plně automatické a nedochází k němu ve zbytečných případech – když se zdrojový kód nezmění. Je zde však drobná výhoda v tom, že CI server nemusí aktivně čekat a každou minutu se dotazovat VCS a tím zatěžovat oba systémy. To je výhodné především v případě, že by ve společnosti existoval jeden CI server a každou minutu by se úložiště zbytečně dotazoval na řádově stovky projektů v něm spravovaných. Pro vývojáře je vhodné uvést, že tato možnost „čekání“ je velmi podobná návrhovému vzoru pozorovatel (angl. *observer*) ze světa objektově orientovaného programování, podrobnější informace o návrhových vzorech lze nalézt např. v (Gamma et al., 1994).

Jakmile se CI server rozhodne (ať už na základě ručního požadavku vývojáře, uplynutí definovaného časového úseku, nebo na základě změny kódu ve VCS) o potřebě sestavení, provede obvykle hned několik kroků, které jsou popsány dále.

3.3.3 Kompilace

Prvním z kroků sestavení obvykle bývá kompilace zdrojových kódů do strojového kódu. I kompilaci lze považovat za určitý druh otestování aplikace. V případě, že je kód nekompilovatelný, např. kvůli přebytečné závorce, vynechanému středníku, je již ihned možné podat vývojářům zpětnou vazbu. Takové případy se v praxi často pravděpodobně nedějí, protože na podobné nedostatky v kódu jsou obvykle vývojáři upozorněni již vývojovým prostředím. Co se ale může projevit, je, že na stroji chybí knihovna, kterou aplikace potřebuje pro svůj chod. Na svůj stroj si ji vývojář mohl nainstalovat sám, ručně, ale už mohl zapomenout upravit např. instalační skript, který by danou knihovnu instaloval během sestavení či před ním. Tento krok nutně nemusí proběhnout jako první. Může probíhat souběžně nebo např. až po statické analýze kódu nebo integraci databáze. Z logických důvodů ale musí kompilace proběhnout např. před spouštěním testů.

3.3.4 Integrace databáze

V případě, že se nejedná o bezstavovou aplikaci, může být obvykle další součástí sestavení integrace databáze – CDBI, z angl. *continuous database integration* (česky kontinuální integrace databáze). Tento krok nebývá až tak častý. Obvyklé může být naopak to, že

aplikační kód a skripty pro přípravu „žijí ve zcela odlišných světech“ (Duval et al., 2007, s. 107). Vývojáři často nemusí mít ani dostatečná práva přistupovat do databáze, aby ji naplnili daty. Tím spíše ani nemusí mít práva databáze vytvářet nebo modifikovat schéma či přístupová práva. Všechny tyto úkoly pak v projektech obvykle plní k tomu pověřená osoba – databázový administrátor. Není až tak neobvyklé, že když vývojář potřebuje změnit nějakou drobnost v databázi – například znovu přehrát původní testovací či vstupní data – zadá požadavek právě na databázového administrátora. Ten má samozřejmě mnoho dalších povinností a než se k tomuto jednoduchému úkolu dostane, uplyne zbytečně mnoho času. Navíc užívat zkušeného databázového profesionála na tyto rutinní úkoly jistě není nejvhodnější. Zcela jistě se nalezne lepší způsob, jak využívat jeho dovednosti. Zde ale nejde jen o zbytečné plýtvání jeho časem a tím pádem penězi společnosti. Mnohem větší problém je ale to, že databázový specialista může být zahlcen podobnými drobnými rutinními úkoly a stane se tak úzkým hrdlem projektu (Duval et al., 2007, s. 110), což může ve výsledku vyústit ve zpoždění celého projektu.

Aby se tomuto zbytečnému plýtvání předešlo, je vhodné zavést CDBI. To sestává z volání různých SQL skriptů. Konkrétně lze dle (Duval et al., 2007, s. 111) obvykle zautomatizovat následující kroky:

1. *„Odstranění databáze;*
2. *vytvoření databáze;*
3. *naplnění databáze vstupními daty;*
4. *naplnění databáze testovacími daty;*
5. *migrace databáze a schémat (pokud je tvořena databáze na základě již existující databáze);*
6. *nastavení více instancí databáze (pokud je podporováno více prostředí);*
7. *úprava sloupců a podmínek (angl. constraint);*
8. *úprava testovacích dat;*
9. *úprava procedur;*
10. *získání přístupu do jiných prostředí;*
11. *záloha databáze.“* (Duval et al., 2007, s. 111).

Kromě výše uvedeného je také pravděpodobné, že součástí skriptů bude i nastavování přístupu k tabulkám či databázím, ať už pouze pro testovací účely, nebo i pro účely běžného používání aplikace.

Aby z CDBI bylo vytěženo maximum, je nutné mít co nejvíce zásahů (ideálně všechny) do databáze zautomatizovaných. Jen tím, že se při každém sestavení dá databáze automaticky

opět do funkčního stavu bez potřeby dalších ručních zásahů, se zamezí tomu, aby se z databázového specialisty stalo opět úzké hrdlo projektu. S tím souvisí i rozšíření práv zasahovat do databáze na vývojáře, což může ohrozit stabilitu sestavení. Proto je nutné dodržovat pravidlo, že pokud sestavení skončí chybou, stane se nejvyšší prioritou tuto chybu napravit, a tím nejen opravit sestavení, ale postarat se o to, aby tato chyba nenastala opět v budoucnu (Duval et al., 2007, s. 124).

3.3.5 Testování

Vzhledem k tomu, že jedním z největších přínosů zavedení CI je včasné odhalení defektů (viz kapitola 3.2.2), je logické, že jedním z důležitých kroků při CI je testování. Tomuto kroku bude v této kapitole věnována zvýšená pozornost i vzhledem k tomu, že součástí jednoho z podcílů této práce je popsat vztah mezi CI a automatizací testování.

Jelikož je v ideálním případě celý proces CI co nejvíce zautomatizovaný, je vhodné automatizovat i testy. Možná právě proto uvádí (Duval et al., 2007, s. 132-144) jen automatizované testy. V kapitole věnované testování jsou rozlišeny celkem čtyři typy testů (autoři nerozlišují striktně mezi pojmy *typ* a *úroveň* testů):

- unit testy (česky také jednotkové testy),
- komponentové test,
- systémové testy,
- funkční (funkcionální) testy.

Někdy jsou pojmy unit test a komponent test vzájemně zaměňovány a rozdíly mezi nimi nivelizovány, např. (Doležel, 2017). (Duval et al., 2007) však mezi nimi rozlišuje.

Unit testy by měly skutečně testovat jen danou jednotku, co nejmenší kus kódu, který má smysl být pohromadě, například v případě Javy je touto jednotkou jedna třída. Pokud je tedy třída závislá na jiné třídě (a její instance je jí předávána například jako parametr konstruktoru) a i v testech je testované třídě v konstruktoru předávána *skutečná* instance jiné třídy, nemůže se jednat o unit test, ale o komponentový test. Aby se jednalo o opravdový jednotkový test, musí být místo skutečných závislostí tříd předávány tzv. „mocky“ (někdy též „stuby“, český překlad pojmu se zatím neustálil). Tak se označují objekty, které splňují stejný kontrakt jako závislosti testované třídy, ale jejich chování je definováno v testu.

I přes to, že popis typů a úrovní testů přímo nenaplnuje cíle práce, považuji za vhodné zde uvést příklad unit testu, vzhledem k tomu, že z mých zkušeností se v tom často chybí, a také vzhledem k tomu, že mezi těmito úrovněmi rozlišuje i (Duval et al., 2007, s. 132-134).

Představíme proto třídu *Car*, která je závislá na rozhraní *Engine* prostřednictvím konstruktoru. Vytvoření takového „mocku“ je na první pohled relativně snadné. Stačí uvnitř

testu definovat třídu *EngineMock*, která dané rozhraní implementuje. Autor testu má pak tuto závislost zcela pod kontrolou. Sám v testu definuje, co se stane, když se zavolá jakákoliv metoda této závislosti, a tím pádem v testu skutečně testuje jen testovanou třídu. Proto se jedná o jednotkový test, a ne test komponentový. Samozřejmě, že tento příklad je zjednodušený a v praxi by asi bylo obtížné takto testovat – s vytvářením „mocků“ – implementací rozhraní, na kterých je testovaná třída závislá, by takto bylo příliš práce. Třída by klidně mohla mít desítky metod, které by bylo nutné naimplementovat (i kdyby měly být prázdné a případně vracet null objekty). V případě, že by i tato třída byla závislá na třídách dalších, a ty na dalších atd. je situace poněkud komplikovanější. Tento problém naštěstí řeší tzv. „mockovací framework“. Mezi nejznámější patří JMockit, Mockito, EasyMock a PowerMock (Slant, 2017a). Díky těmto frameworkům je možné u „mocků“ implementovat jen metody, které jsou skutečně během testu volány, a implementaci ostatních metod vynechat. Viz výpis 3, kde je naznačeno užití frameworku Mockito.

Výpis 3: *Užití frameworku Mockito při unit testování (zdroj: autor)*

```
1 import org.junit.Test;
2 import static org.mockito.Mockito.*;
3 import static org.junit.Assert.*;
4 ...
5 @Test
6 public void unitTestStartingCarWithEngineShouldPass(){
7     Engine engine = mock(Engine.class);
8     when(engine.init()).thenReturn("initialized");
9     Car car = new Car(engine);
10    boolean started = car.start(); // uvnitř start() je volána metoda engine.init()
11    assertTrue(started); // pokud engine "initialized" mělo by být auto nastartováno
12 }
```

Ve výpisu jsou nejzajímavější řádky sedm a osm. Na nich je definován a inicializován samostatný „mock“, tedy minimalistická implementace rozhraní *Engine*. Když je pak kdykoliv volána metoda *init* této instance, je vrácen příslušný řetězec („initialized“). Pokud bychom nedefinovali, co se má vrátit, dostali bychom hodnotu *null*, což by v metodě *start* bylo vyhodnoceno jako neúspěšné nastartování motoru. Metoda by následně patrně vrátila hodnotu *false* a celý test by tedy skončil neúspěšně.

Komponentové testy dle (Duval et al., 2007, s. 134-135) testují větší jednotky než unit testy. Když se budeme držet předchozího příkladu *Car* – *Engine*, už nebude třeba vytvářet „mocky“, nýbrž se vše bude testovat dohromady, viz výpis 4.

Výpis 4: Komponentový test (zdroj: autor)

```
1 import org.junit.Test;
2 import static org.mockito.Mockito.*;
3 import static org.junit.Assert.*;
4 ...
5 @Test
6 public void componentTestStartingCarWithEngineShouldPass(){
7     Engine engine = new EngineImpl("DOHC V8");
8     Car car = new Car(engine);
9     boolean status = car.start();
10    assertTrue(status);
11 }
```

Test sice vypadá velmi podobně jako předchozí a na první pohled se může zdát, že testuje jen třídu Car. Ve skutečnosti je však tranzitivně testována i třída EngineImpl. Pokud by z její metody init byla vyhozena výjimka, test by skončil neúspěšně. Test tedy netestuje třídu Car jako samostatnou jednotku bez závislosti na ostatním kódu, ale naopak ji testuje jako komponentu s jejími všemi závislostmi.

Stejně tak jako jsou si podobné unit testy a komponentové testy, tak i systémové a funkční testy vypadají na první pohled do určité míry velmi podobně. Samozřejmě záleží na tom, podle které normy či metodiky je na typy testů nahlíženo, zde se ale budu držet rozdělení testů dle (Duval et al., 2007, s. 136-138). Dle něj je pro oba typy testů potřeba, aby celá aplikace, respektive celý systém (aplikace, databáze...), byl plně nainstalován a běžel, jinými slovy aplikace musí být někde nasazena. Rozdíl je ale v tom, že funkční testy jsou mnohem bližší akceptačním testům, (Duval et al., 2007, s. 137) dokonce prosazuje, že funkční testy jsou totožné s akceptačními testy. Rozdíl mezi funkčními a systémovými testy je ten, že funkční testují systém přesně tak, jak jej bude uživatel používat.

Při testování webové aplikace, na kterou by měli uživatelé přistupovat prostřednictvím internetového prohlížeče, bude tedy užíván internetový prohlížeč. Jako příklad, jak programově ovládat skutečný prohlížeč (včetně nejrozšířenějších – Chrome, Firefox a Internet Explorer), je uveden nástroj Selenium. Díky němu je možné např. otevřít prohlížeč, jít na libovolné URL, zadat jméno a heslo do formuláře (pole je nutné lokalizovat, např. přes XPath), simulovat stisknutí klávesy enter a následně zkontrolovat, že se na nové stránce nachází např. text „Uživatel úspěšně přihlášen“.

Systémový test by se od tohoto funkčního (akceptačního) testu lišil v tom, že by k testování stránky nepoužíval internetový prohlížeč (ani by ho neovládal skrze dříve zmíněné Selenium), ale chování prohlížeče by do jisté míry simuloval sám. Jako příklad testovacího frameworku pro tento druh testů Duval uvádí Framework JWebUnit.

Toto členění není jediné, existují další způsoby členění testů. Mezi nejrozšířenější způsob rozdělení testů patří akronym FURPS či jeho variace s menšími či většími úpravami. Například dle normy (ČSN ISO/IEC 9126-1, 2002) lze systém rozdělit na šest charakteristik jakosti (a takto se pak často rozdělují i typy testů):

- F – funkčnost (z angl. functionality),
- U – použitelnost (angl. usability),
- R – bezporuchovost (angl. reliability),
- P – přenositelnost (angl. portability),
- S – udržitelnost (angl. supportability),
- účinnost (angl. efficiency).

Pro potřeby CI je však vhodnější dělení testů na unit – komponentové – systémové – funkční (akceptační). Důležité je zachování tohoto pořadí, a to především z toho důvodu, že běh dříve zmíněných testů trvá obvykle kratší dobu než následujících. Proto je vhodné při nastavování CI nejdříve pustit unit testy a komponentové testy. Pokud ty skončí neúspěšně, nemá smysl spouštět další. Nemá to smysl ale také proto, že systémové a akceptační testy potřebují k běhu již nasazenou celou aplikaci (samozřejmě že aplikaci není vhodné nasadit na produkční prostředí a testy pouštět až tam, ideální je mít vedle prostředí testovací), kdežto unit testy lze pustit samostatně. Až když unit a komponent testy procházejí, má smysl nasadit aplikaci na testovací prostředí.

Všechny testy rozebírané v této kapitole byly myšleny jako plně automatizované. Jen takto se dá hovořit o CI. Kdyby testy nebyly plně automatizované, vykonávání testů by pravděpodobně bylo omezeno a neprobíhalo by při každé integraci zdrojového kódu. Tím by celé CI do určité míry postrádalo smysl, jelikož by přišlo o jeden z hlavních důvodů zavádění – včasné odhalení defektů. To ovšem neznamená, že zavedení CI znemožňuje manuální testování. Je tomu dokonce naopak. Dokáže šetřit čas manuálních testerů tím, že nemusí stále dokola provádět ty stejné testy (např. „vyplň políčko, klikni na tlačítko přihlásit, zkontroluj, že je uživatel přihlášen“). Ti pak tento nově nabytý čas mohou věnovat důkladnějšímu provádění testů, které zautomatizovat nejdou. Jako příklad těchto neautomatizovatelných testů mohu uvést např. ad-hoc testování, nebo průzkumné testování (angl. exploratory testing). Tyto se od předchozích testů liší právě tím, že nemají dopředu pevně daný průběh – nelze tedy naskriptovat, a tak ani automatizovat. Tester provádí testy víceméně na základě svých zkušeností, ne na základě scénáře, který by mu přesně říkal, co má dělat (Bach, nedatováno). Tyto oba přístupy je vhodné proto kombinovat a provádět průzkumné testování například až ve chvíli, kdy se aplikace blíží nasazení do produkce.

3.3.6 Inspekce

(Duval et al., 2007, s. 17) uvádí jako samostatný krok něco, co označuje jako *inspekci*. Díky ní lze odhalit mnoho problémů v kódu, které by testy neodhalily. Tyto problémy se často ani nemusí týkat přímo funkcionality, spíš mohou odhalit a měřit tzv. technický dluh v kódu (některé nástroje tento dluh převádí na představitelné jednotky – výstupem z nich může být

sdělení, že kód obsahuje tři člověkodny technického dluhu, tzn. jeden programátor by napravováním těchto „chyb“ strávil tři pracovní dny). Do technického dluhu patří problémy, jako je kopírování bloků kódu napříč projektem, nízké pokrytí kódu testy, chybějící dokumentace aj.

S technickým dluhem souvisí i pojem „pachy v kódu“ (častěji se užívá původní angl. výraz „code smells“). Toto sousloví vymysleli a zpopularizovali Fowler a Beck v knize Refactoring (Fowler a Beck, 1999). Pojem „pachy v kódu“ označuje určité indikátory toho, že v systému může být přítomen hlubší problém. Příkladem můžou být dlouhé metody, složité metody, velké třídy, duplikovaný kód, řetězení zpráv atd., podrobněji viz (Fowler a Beck, 1999, s. 63-71). Je nutné podotknout, že tyto „pachy“ ne vždy nutně indikují hlubší problém.

Testování může odhalit problémy při špatném chování programu. Těžko ale může odhalit skrytý technický dluh uvnitř kódu nebo již dříve zmíněné „pachy kódu“. K tomu obvykle běžně slouží revize prováděné kolegy (angl. peer reviews). Při nich všechen kód, který jde do „hlavní větve“ kódu („trunk“ u Subversion, „master branch“ u Git), musí být předem schválen jedním či více členy týmu; záleží na tom, jak se tým domluví. Tento ruční postup má však několik nedostatků. Jedním z nich je to, že lidé často dělají chyby, do kódu by se vlivem přehlédnutí dříve nebo později dostalo něco, co by nemělo. Ať již by mělo jít o detail typu prohřešku proti kódovacím standardům (např. uvozovací složená závorka na novém řádku), nebo o závažnější problémy – např. metoda s příliš vysokou cyklomatickou složitostí. S tím souvisí další aspekt těchto revizí, a to aspekt psychosociální. Kolega schvalující revizi nebude při ní příliš důsledný. Důvodů může být mnoho, ať už případný strach z toho, že mu kolega, jehož kód je nyní kontrolován, „vrátí“ toto „přísné“ chování při další revizi, kdy role budou prohozeny. Nebo i jen určitá forma autocenzury při hodnocení kódu seniornějšího kolegy („když použil *tuto* nekorektní techniku, asi k ní měl důvod“). Pro tyto případy je vhodnější spouštění tzv. statické a dynamické analýzy kódu, což by mělo být součástí tohoto kroku CI – inspekce. Program je nekompromisní ke všem členům týmu stejně a rovněž je v plnění některých úkolů neomylný. Další výhodou je, že jeho spuštění takřka nic nestojí – cena energií, resp. procesorového času je zanedbatelná ve srovnání s dnešní hodinovou mzdou běžného programátora. Vyšší náklady budou jen v počáteční fázi projektu, kdy bude třeba vše nainstalovat a nastavit. Rovněž pokud budou používány nástroje, jejichž užití není zdarma, je potřeba počítat s poplatky za licence. Jako příklad nástrojů, které mohou být ke statické a dynamické analýze kódu používány, (Duval et al., 2007, s. 167-183) uvádí nástroje:

- JavaNCSS (pro počítání cyklomatických složitostí metod);
- PMD (kódovací standardy);
- PMD-CPD (hledání kopírovaného kódu napříč projektem);
- Cobertura, Clover, EMMA (pokrytí kódu testy).

Pro podrobnější rozbor nástrojů provádějících inspekci vizte kapitolu 4.4. Na závěr této podkapitoly je vhodné rovněž zmínit, že na rozdíl od testování lze statickou analýzu kódu (JavaNCSS, PMD...) spouštět i před jeho kompilací. Jinými slovy takto uvedené pořadí v rámci sestavení rozhodně není závazné.

3.3.7 Poskytnutí okamžité zpětné vazby

Poskytnutí zpětné vazby je dalším z kroků CI. Není to krok vyloženě nezbytný. I bez něj se dá mluvit o CI, které bude stále plnit svoji funkci. V případě, že by ale nastal nějaký problém (typickým příkladem může být spadnutí sestavení či neprůchozí testy) a nikdo se o tom okamžitě nedozví, nebude moct ihned začít pracovat na nápravě tohoto problému. Je zde totiž velká šance, že si výsledky sestavení nebo testů někdo ručně zkontroluje až poté, co už několik dní padají. Následné řešení problému s takovýmto časovým odstupem od jeho vzniku je pak samozřejmě náročnější. Pokud do úložiště přibude kód od více vývojářů, je daleko těžší najít změnu, kvůli které padají testy, než kdyby se o pádu vývojář dozvěděl ihned.

Aby byla zpětná vazba v CI účinná, měla by splňovat následující čtyři podmínky (Duval et al., 2007, s. 205-220):

1. měla by být směřována správným lidem;
2. ve správný čas;
3. vhodným komunikačním kanálem;
4. měla by obsahovat relevantní informace.

Tyto podmínky jsou relativně obecné a těžko si pod nimi představit konkrétní kroky, proto nyní podrobněji rozeberu, co vlastně ve skutečnosti znamenají.

Na softwarovém projektu obvykle nepracují jen vývojáři, ale například i testéři, architekti, lidé z byznysu nebo projektoví manažeři. Každý z nich potřebuje pro svou práci jiné informace. Vývojář by měl být informován co nejdříve o padajícím sestavení. O padajícím sestavení není patrně nutné informovat všechny vývojáře, ale jen ty, kteří naposledy něco měnili v kódu. Stejně tak výsledky statické analýzy nemusí zajímat všechny, ale převážně vývojáře. Podobně by o padajících testech měli být primárně informováni testéři, ti pak mohou problém dále investigovat a zjistit, zda je problém v testech, nebo skutečně v kódu. Manažery naopak budou nejspíš zajímat různé metriky typu rozsah pokrytí kódu testy nebo celková stabilita sestavení. Naopak není vhodné posílat všem členům týmu všechny reporty po každé integraci. Ve výsledku by to nakonec vedlo k tomu, že by všechna zpětná vazba byla ignorována. Lidé by si například nastavili filtry na e-maily z CI serveru, a ty by v lepším případě končily v samostatné složce, kterou by nikdo nečetl, v horším by byly rovnou mazány.

To, že by zpětná vazba měla být poskytována ve správný čas, rozhodně neznamena, že by měla být poskytována vždy a okamžitě. Když zůstaneme u předchozího srovnání, tak například přehledy a reporty pro manažera mu nemusí být posílány po každé změně kódu. Procentuální pokrytí kódu testy se ze dne na den nijak zásadně nezmění. Zde opět hrozí riziko jako v předchozím bodě, a to, že pokud by zpětné vazby bylo příliš a obtěžovala by, lidé by našli způsob, jak ji všechnu ignorovat. Proto je vhodné tyto druhy reportů zasílat na pravidelné bázi, např. jednou týdně, případně na konci tzv. „sprintu“, pokud tým pracuje podle metodiky Scrum. Naopak to, že padá sestavení či testy, by se členové týmu měli dozvědět ihned. Pro všechny členy týmu by se pak mělo stát prioritou tento stav co nejdříve napravit.

Zpětná vazba by měla probíhat vhodným způsobem – vhodným komunikačním kanálem. Jako základní komunikační kanál se v dnešní době jeví e-mail. (Duval et al., 2007, s. 212) ho srovnává s SMS zprávami. Jako výhody e-mailu uvádí, že mohou být delší a obsahovat tak více informací než znakově omezená zpráva. Na druhou stranu uvádí, že SMS zprávu si člověk může přečíst vždy, kdy u sebe má mobilní telefon a člověk tak nemusí stále být na e-mailu, potažmo připojen k internetu. To byla patrně výhoda SMS oproti e-mailům platná v době vydání této publikace – před deseti lety. Nyní, dle dat z Českého statistického úřadu (Český statistický úřad, 2016), je připojeno přes mobilní telefon k internetu 42 procent všech obyvatel České republiky (z toho 11 procent používá pouze wifi a 31 pouze data nebo kombinaci). V budoucnu se dá očekávat, že se počet připojených lidí ještě zvýší, vzhledem k tomu, že v roce 2012 bylo k internetu v České republice prostřednictvím mobilního telefonu připojeno pouhých 11 procent obyvatel. Jako další možnosti, jak informovat členy týmu, uvádí (Duval et al., 2007, s. 214-220) doplňky do prohlížeče, ikonku do Windows lišty. Ty by patrně měly zelenou barvu, pokud by vše probíhalo v pořádku, nebo červenou, pokud by někde byl problém. Zde je ale nevýhoda, že takto by stěží šlo monitorovat více projektů. Lepší možnost je mít v místnosti, kde tým pracuje, k tomuto účelu dedikovanou televizi nebo větší monitor. Zde by mohlo být čitelně zobrazeno více ukazatelů i z více projektů, pokud by jeden tým pracoval na více projektech – testy, sestavení, průměrný čas integrace nebo pro výstrahu i jméno člověka, jehož poslední příspěvek do kódu způsobilo nějaký problém. Agresivnější nastavení by mohlo v případě problému spouštět zvukový signál a tím zajistit, že se na nápravě bude brzy pracovat. Další způsob, který se trochu podobá e-mailům, je zpráva do nějakého chatovacího nástroje – např. Slack nebo Telegram, který tým běžně používá pro rychlejší komunikaci, než která by byla vedena klasicky přes elektronickou poštu. Ve firmách, kde je kladen větší důraz na procesy, by bylo možné zpětnou vazbu integrovat přímo s kolaboračními nástroji. Při pádu sestavení by pak bylo možné rovnou člověku, který jej způsobil, zadat úkol do takového nástroje (např. Jira, Redmine), ve kterém je firma zvyklá úkoly přiřazené pracovníkům běžně udržovat.

To, že by měla zpětná vazba obsahovat relevantní informace, souvisí především s bodem jedna – s tím, že by zpětná vazba měla být směřována správným lidem. Pro každou roli pracující na projektu je totiž „relevantní“ něco jiného. Zde je vhodné znova zdůraznit, že

není žádané všechny členy zahlcovat informacemi, protože by to pravděpodobně nakonec vedlo jen k tomu, že by zpětnou vazbu všichni ignorovali.

3.3.8 Nasazení

Jako poslední krok CI uvádí (Duval et al., 2007, s. 189-201) nasazení (angl. deployment). Tento krok však jako součást CI nevnímají všichni autoři. Pokud je nasazení součástí automatizovaného procesu navazujícího na CI, hovoří se pak o CDV případně až o CDP. Rozdílům a vztahu mezi těmito třemi pojmy je pro přehlednost věnována samostatná kapitola 3.4.4, kde je vše popsáno podrobněji.

(Duval et al., 2007, s. 191) uvádí šest obecných kroků, které jsou obvykle součástí nasazení bez ohledu na použité technologie či konkrétní platformu:

- Označení konkrétní verze zdrojového kódu v úložišti.
- Příprava čistého prostředí.
- Na základě označené verze kódu v úložišti vytvoření a označení sestavení a jeho následná instalace na stroj z předchozího bodu.
- Spuštění a úspěšný běh testů oproti aplikaci nainstalované z předchozího kroku.
- Vytvoření reportů ze sestavení.
- V případě nutnosti (např. neprůchozích testů) vrácení se na předchozí funkční verzi.

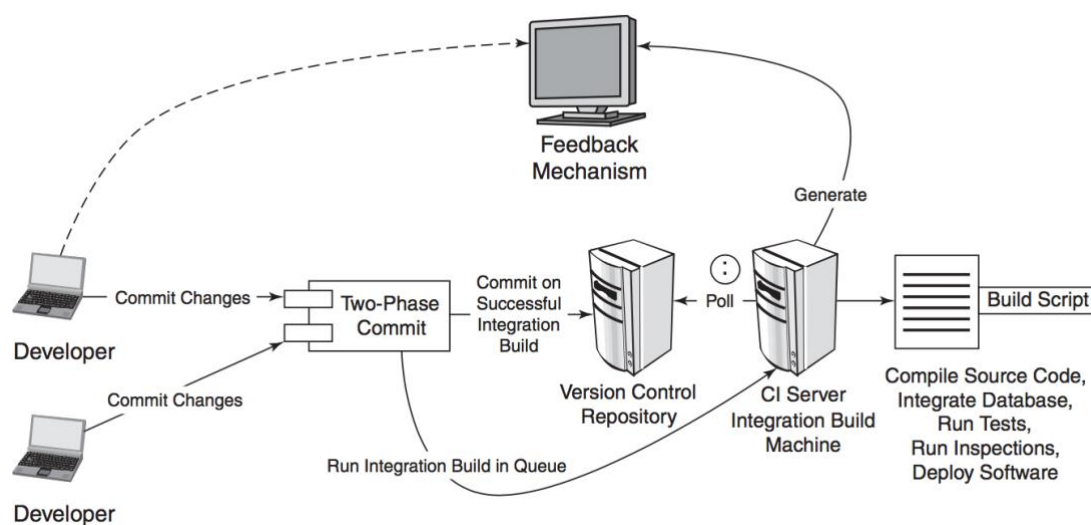
Jak je ze seznamu vidno, některé kroky nasazení, které Duval uvádí, se do určité míry překrývají s kroky, které uvádí obecně u CI – např. sestavení a spuštění testů. Je tu však jeden podstatný rozdíl. Při nasazení jsou navíc přítomny mezikroky v podobě označení (případně „*otagování*“) verzí – ať už se jedná o označení konkrétní verze, respektive kombinace verzí souborů v úložišti, či o označení sestavení. To má jednu hlavní výhodu – při reportování o výsledcích testů je možné zmínit konkrétní verzi, oproti které případný test neprošel. A v případě nasazení nefunkční verze do produkce je díky označování verzí možné „vrátit se“ k poslední funkční verzi.

3.3.9 Rozšířený proces Continuous Integration

V podkapitole 3.3.1 byl popsán základní proces CI a v podkapitolách 3.3.2–3.3.8 byly rozebrány jednotlivé kroky CI včetně jejich pořadí. Dodržování takto nastavených pravidel však nezajistí to, že sestavení z aktuální hlavní linie kódu (u Gitu většinou větev označována jako „*master*“) vždy dopadne úspěchem. Rozbití „hlavní linie kódu“, ať už tým používá jakýkoliv verzovací systém, může mít nepříjemné důsledky především v tom případě, že na projektu nepracuje pouze jeden člověk. Dejme tomu, že Alice provede změny kódu v hlavní

větvi úložiště. Kód je následně zkompilován, otestován unit testy, nasazen na testovací prostředí a tam otestován systémovými testy. Kdykoliv v průběhu je navíc spuštěna statická analýza kódu. Pak je Alici automaticky poslána zpětná vazba v podobě výsledků testů a analýzy kódu. Až nyní Alice případně zjistí, že systémové testy neprošly a že je tedy její kód rozbil a bude muset být opraven. Nyní si představme, že než Alice svůj kód opraví, Bob (její kolega) dokončí svůj úkol a také ho nahraje do hlavní větve v úložišti. Celý kolotoč CI se spustí znova a testy opět samozřejmě neprojdou (kvůli Alicině chybě). Co dělat nyní? Nezbyde než počkat, až Alice svůj kód opraví – co když je ale chyba i v Bobově kódu? Alice může zoufale měnit celý svůj kód ve snaze ho opravit, jenže testy nyní padají kvůli Bobovým změnám.

Aby se předešlo podobným situacím, bylo by vhodné nějak zajistit, aby se do hlavní větve úložiště dostal jen kód, který nerozbije sestavení, ani testy. (Duval et al., 2007, s. 223-224) proto základní proces CI, který je popsán v předchozích podkapitolách, dále rozšiřuje a zavádí nový pojem – „dvoufázový commit“ (commit je v Gitu příkaz pro uložení současné verze kódu – vytvoření „snapshotu“ (Atlassian, nedatováno)). Na obrázku 2 je zachycen tento rozšířený proces.



Obrázek 2:
Schéma rozšířeného procesu CI (zdroj: Duval et al., 2007, s. 224)

Z obrázku 2 je patrné, že pokud se vývojář snaží nahrát nový kód do úložiště (přesněji pokud se ho snaží nahrát do hlavní linie kódu), není mu to ihned povoleno. CI server nový kód zkompile a rovnou nasadí na testovací stroj. Kromě jednotkových testů a statické analýzy tak může navíc oproti takto běžící aplikaci spustit systémové testy. Až pokud vše proběhne v pořádku (sestavení proběhne úspěšně, statická analýza kódu neobjeví žádné chyby a testy projdou), CI server zpraví úložiště o tom, že je vše v pořádku, a až pak je možné, aby se kód stal součástí hlavní linie.

Díky tomuto rozšíření CI je prakticky nemožné, aby nastal problémový scénář s Alicí a Bobem z počátku kapitoly. Kdyby Alicin kód vedl k nekorektnímu chování aplikace, nebyl by jednoduše „vpuštěn“ do úložiště, respektive do jeho hlavní linie.

Takové bezpečné chování není ovšem zadarmo. Cena za takovouto stabilitu je poněkud zpomalený celý proces integrace. Aplikace je patrně vždy nahrávána na „čistý“ stroj. Vždy tak musí být znova vytvořena databáze, přehrána testovací data a nakopírována aplikace. Zde záleží na rozsahu aplikace a celého řešení a robustnosti systémových testů. Pokud základní proces CI trval například řádově jednotky minut, tak tento rozšířený proces se může vyšplhat až k desítkám minut. Další problém nastává, pokud chce svůj kód ve stejný čas integrovat více lidí. Vytvoří se klasická fronta (FIFO) a ten vývojář, který přijde později, musí počkat, než proběhne nasazení a otestování kódu na testovacím serveru toho vývojáře, který svůj kód nahrál byť o pár sekund dříve. Tím se přichází o jednu z podmínek účinné zpětné vazby jmenované v kapitole 3.3.7 – o její správné načasování, v tomto případě včasnost. Na druhou stranu i tento problém lze řešit. Je možné přidat další testovací stroj a kód integrovat paralelně.

3.4 Pojmy související s Continuous Integration

V této kapitole budou popsány pojmy úzce související s CI. V kapitole 3.4.1 je popsán model kvality softwaru dle norem řady ISO/IEC 250nn s důrazem na charakteristiky udržitelnost a přenositelnost, které se zaměřením práce souvisí nejvíce. V kapitole 3.4.2 jsou popsány různé metriky měření pokrytí kódu testy. V kapitole 3.4.3 je rozebrán pojem cyklomatická složitost. V kapitole 3.4.4 je popsán vztah mezi CI, CDV a VCP, které jsou často zaměňovány. Poslední kapitola 3.4.5 je věnována pojmu DevOps, který má k předchozím třem také blízko.

3.4.1 Model kvality softwaru dle norem řady ISO/IEC 250nn

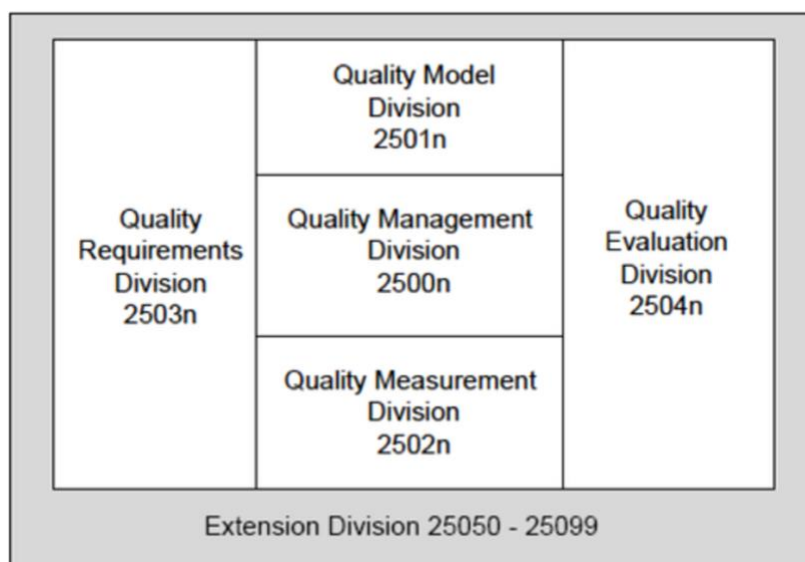
I přes to, že cílem práce není analyzovat modely kvality softwaru či obecné modely kvality, tak tato problematika s CI úzce souvisí, což je zřejmé z přínosů CI, které byly rozebrány v kapitole 3.2, především pak v podkapitole 3.2.3. Modelů kvality softwaru existuje velké množství, např. (Pavličková, 2014, str. 9) uvádí celkem sedm nejznámějších modelů kvality:

- „*McCallův model kvality*,
- *Boehmův model kvality*,
- *Dromeyův model kvality*,
- *FURPS+ model kvality*,

- *Model kvality ISO/IEC 9126,*
- *Model kvality ISO/IEC 25000,*
- *Model kvality konsorcia pro kvalitu softwaru“ (Pavličková, 2014, str. 9).*

Modely kvality FURPS a ISO/IEC 9126 jsou zmíněny v kapitole 3.3.5 věnované testování, nicméně nejsou podrobněji rozebírány. Naopak model kvality softwaru dle norem ISO/IEC 25000 je podrobněji popsán v této kapitole. Tento model jsem pro podrobnější rozbor zvolil proto, že je mezinárodně uznávaným standardem a ve srovnání s normou ISO/IEC 9126 je novější (ve skutečnosti ji nahrazuje).

Úvodem je vhodné zdůraznit, že se nejedná pouze o jednu ISO/IEC označenou 25000, ale o celou řadu norem, které jsou seskupeny celkem do šesti divizí, viz obrázek 3:



Obrázek 3:

Divize norem řady ISO/IEC 25000 (zdroj: ISO/IEC 25010, 2011)

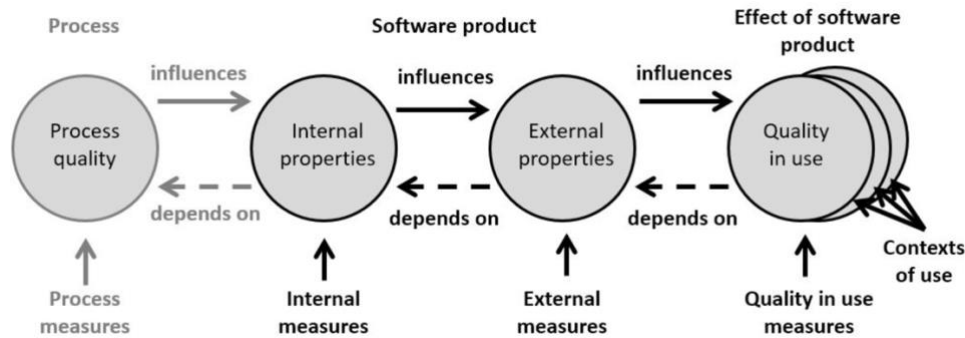
Každá z divizí obsahuje několik norem. V době psaní této práce (léto 2017) je na stránkách Mezinárodní organizace pro normalizaci www.iso.org publikováno celkem sedmnáct platných norem patřících do těchto šesti divizí. Seznam všech těchto norem, včetně označení a plného názvu, uvádím v *tabulce 1*.

Tabulka 1: Normy spadající do řady norem ISO/IEC 25000 (zdroj: International Organization for Standardization, 2017)

Divize	Označení normy	Plný název normy
Quality Management	ISO/IEC 25000:2014	Systems and software engineering – Systems and software Quality Requirements and Evaluation (SQuaRE) – Guide to SQuaRE
	ISO/IEC 25000:2014	Systems and software engineering – Systems and software Quality Requirements and Evaluation (SQuaRE) – Planning and management
Quality Model	ISO/IEC 25010:2011	Systems and software engineering – Systems and software Quality Requirements and Evaluation (SQuaRE) – System and software quality models
	ISO/IEC 25012:2008	Software engineering – Software product Quality Requirements and Evaluation (SQuaRE) – Data quality model
Quality Measurement	ISO/IEC 25020:2007	Software engineering – Software product Quality Requirements and Evaluation (SQuaRE) – Measurement reference model and guide
	ISO/IEC 25021:2012	Systems and software engineering – Systems and software Quality Requirements and Evaluation (SQuaRE) – Quality measure elements
	ISO/IEC 25022:2016	Systems and software engineering – Systems and software quality requirements and evaluation (SQuaRE) – Measurement of quality in use
	ISO/IEC 25023:2016	Systems and software engineering – Systems and software Quality Requirements and Evaluation (SQuaRE) – Measurement of system and software product quality
	ISO/IEC 25024:2015	Systems and software engineering – Systems and software Quality Requirements and Evaluation (SQuaRE) – Measurement of data quality
Quality Requirements	ISO/IEC 25030:2007	Software engineering – Software product Quality Requirements and Evaluation (SQuaRE) – Quality requirements

Divize	Označení normy	Plný název normy
Quality Evaluation	ISO/IEC 25040:2011	Systems and software engineering – Systems and software Quality Requirements and Evaluation (SQuaRE) – Evaluation proces
	ISO/IEC 25041:2012	Systems and software engineering – Systems and software Quality Requirements and Evaluation (SQuaRE) – Evaluation guide for developers, acquirers and independent evaluators
	ISO/IEC 25045:2010	Systems and software engineering – Systems and software Quality Requirements and Evaluation (SQuaRE) – Evaluation module for recoverability
Extension	ISO/IEC TR 25060:2010	Systems and software engineering – Systems and software product Quality Requirements and Evaluation (SQuaRE) – Common Industry Format (CIF) for usability: General framework for usability-related information
	ISO/IEC 25062:2006	Software engineering – Software product Quality Requirements and Evaluation (SQuaRE) – Common Industry Format (CIF) for usability test reports
	ISO/IEC 25063:2014	Systems and software engineering – Systems and software product Quality Requirements and Evaluation (SQuaRE) – Common Industry Format (CIF) for usability: Context of use description
	ISO/IEC 25064:2013	Systems and software engineering – Software product Quality Requirements and Evaluation (SQuaRE) – Common Industry Format (CIF) for usability: User needs report

V normách ISO/IEC řady 25000 je cesta ke kvalitnímu softwaru vnímána jako kauzální řetězec – kvalita procesu → interní vlastnosti → externí vlastnosti → kvalita z hlediska uživatelů softwaru. V normě ISO/IEC 25010:2011 je pro kvalitu klíčové zejména to, zda jsou naplňovány potřeby stakeholderů (patrně především zákazníků – uživatelů softwaru): „Kvalita systému je určena mírou, do jaké systém splňuje stanovené a očekávané potřeby jednotlivých zúčastněných stran, a tím poskytuje hodnotu.“ (Pavličková, 2014). Naplňování těchto potřeb a poskytování hodnoty zákazníkům je ovlivněno externími vlastnostmi softwaru. Externí vlastnosti softwaru jsou dále ovlivněny interními vlastnostmi softwaru, přičemž za interní vlastnosti softwaru je v tomto kontextu považována především kvalita zdrojového kódu. Kauzální řetězec kvality v životním cyklu produktu je znázorněn na obrázku 4.

**Obrázek 4:**

Kauzální řetězec kvality v životním cyklu softwarového produktu (zdroj: ISO/IEC 25010, 2011)

Aby bylo možné charakterizovat interní a externí kvalitu, je v modelu kvality produktu normy ISO/IEC 25010:2011 definováno celkem osm charakteristik, které jsou dále členěny na další podcharakteristiky, jichž je celkem třicet jedna. Abych se příliš nevzdálil hlavnímu tématu této práce, podrobněji rozeberu pouze charakteristiky, které přímo souvisí s tématem CI či virtualizace; ostatní charakteristiky, včetně jejich podcharakteristiky, pro úplnost pouze zmíním.

Charakteristika: Funkční vhodnost (Functional suitability)

- Úplnost funkcionality (Functional completeness),
- Správnost funkcionality (Functional correctness),
- Vhodnost funkcionality (Functional appropriateness).

Charakteristika: Výkonnostní účinnost (Performance efficiency)

- Časové vlastnosti (Time behaviour),
- Utilizace zdrojů (Resource utilization),
- Kapacita (Capacity).

Charakteristika: Kompatibilita (Compatibility)

- Koexistence (Co-existence),
- Interoperabilita (Interoperability).

Charakteristika: Použitelnost (Usability)

- Rozpoznatelnost vhodnosti (Appropriateness recognizability),
- Naučitelnost (Learnability),

- Snadnost užití (Operability),
- Ochrana před chybami uživatele (User error protection),
- Přívětivost uživatelského rozhraní (User interface aesthetics),
- Přístupnost (Accessibility).

Charakteristika: Spolehlivost (Reliability)

- Zralost (Maturity),
- Dostupnost (Availability),
- Odolnost vůči chybám (Fault tolerance),
- Obnovitelnost po nastalých chybách (Recoverability).

Charakteristika: Bezpečnost (Security)

- Důvěrnost (Confidentiality),
- Integrita (Integrity),
- Nepopiratelnost (Non-repudiation),
- Odpovědnost (Accountability),
- Autenticita (Authenticity).

Charakteristika: Udržovatelnost (Maintainability)

- Modularita (Modularity),
- Znovupoužitelnost (Reusability),
- Analyzovatelnost (Analysability),
- Modifikovatelnost (Modifiability),
- Testovatelnost (Testability) (ISO/IEC 25010, 2011), (Pavlíčková, 2014, str. 22-23).

Ze všech charakteristik souvisí udržovatelnost s CI a s touto prací nejvíce. Jedním z přínosů při zavádění CI je totiž zvýšení kvality kódu, viz kapitola 3.2.3. A právě kvalitní kód automaticky splňuje podcharakteristiky obsažené v této charakteristice. Je modulární a obvykle by mělo být možné relativně jednoduše vyměnit jeden modul (např. pro práci s konkrétní databází) za jiný modul (např. pro práci s jinou databází). S touto podcharakteristikou souvisí i snadná modifikovatelnost. Úprava kódu v jedné části aplikace

by neměla rozbít kód jinde. Rovněž modifikovatelnost a snadná znovupoužitelnost může být zajištěna jen tehdy, je-li kód čitelný i pro jiné vývojáře než jen pro jeho autora v době psaní. Testovatelnost s kvalitou kódu také přímo souvisí. U softwaru, který není testovaný vůbec, je velice obtížné jeho kvalitu prokázat. Navíc to, že kód je obtížně testovatelný či vůbec netestovatelný, je obvyklé u kódu, který je úzce provázaný (angl. *tight coupled*). To nutně nemusí znamenat, že by byl nekvalitní nebo nefunkční, jen je pak obtížně rozšiřitelný a modifikovatelný, a to už je vlastnost, jež je u kódu nevídaná.

Charakteristika: Portabilita (Portability)

- Adaptabilita (Adaptability),
- Instalovatelnost (Installability),
- Nahraditelnost (Replaceability) (ISO/IEC 25010, 2011), (Pavličková, 2014, str. 23).

Portabilita je další z charakteristik, která úzce souvisí se zaměřením této práce, ale poněkud odlišně než předchozí charakteristika. Ani tolik příliš nesouvisí s konceptem CI, ale spíše s virtualizací – a s technologií Docker. Přenositelnost byla totiž jednou z hlavních motivací práce proč postavit CI řešení na platformě Docker. Takto je možné zajistit to, že výsledné řešení bude moci sloužit různorodé skupině uživatelů. Bude zcela jedno, na jakém operačním systému, na jaké verzi či distribuci řešení poběží, pokud na systému poběží Docker. Nebude nutné doinstalovávat další programy, aplikace, prostředí, ovladače, certifikáty, ... Vše bude připravené v Docker obrazech (častěji angl. *image*, pojem podrobněji popsán v kapitole 5), a ty stačí spustit a poběží stejně na všech prostředích s Dockerem.

Nejvíce jsou tedy pokryty podcharakteristiky instalovatelnost a adaptabilita. Téměř totiž nezáleží na prostředí, kde bude CI řešení běžet. Bude možné jej nainstalovat všude, kde bude dostupný Docker. Podcharakteristika nahraditelnost je ale také naplněna. Díky tomu, že v řešení bude každá aplikace běžet v samostatném Docker kontejneru, je nahraditelnost jednotlivých komponent (modulů) relativně snadná a i za běhu ostatních komponent proveditelná. Na CI serveru se například jen nastaví jiná URL adresa nového úložiště (který poběží v samostatném kontejneru) a vše bude dále fungovat bez větších komplikací. Díky tomuto přístupu je možné bezpečně provádět nové aktualizace – vytvoří se nový obraz s novou, aktualizovanou verzí úložiště, CI server se na tuto novou verzi přesměruje a pokud vše poběží bezchybně, vše bude v pořádku a stará verze bude odstraněna. Naopak, pokud při testování nastanou chyby, vše se přesměruje zpátky na starou verzi a nová verze bude odstraněna, nebo opravena.

3.4.2 Metriky pokrytí kódu testy

V kapitole 3.3 věnované procesu CI, a především v podkapitolách 3.3.5 a 3.3.6 věnovaných testování a inspekci, bylo zmíněno pokrytí kódu testy (angl. častěji *code coverage*). Pod

tímto pojmem si většina programátorů pravděpodobně představí pouze to, kolika řádky program během testování prošel v poměru k celkovému počtu řádků programu. Ačkoliv je tato metrika nejznámější, rozhodně není jediná a někdy může být i zavádějící a dodávat falešný pocit bezpečí, jak bude ukázáno na příkladu.

Jak již bylo řečeno, existuje více metrik pro měření pokrytí kódu testy. (Potton, 2002) uvádí tři nejpoužívanější. Začnu s popisem té nejznámější – analýza příkazů a řádků programu (angl. *statement coverage*). V té se monitoruje, jestli každý příkaz v kódu programu byl proveden (Potton, 2002). Vzoreček pro výpočet je jednoduchý, buď lze rovnou udávat poměr: $\frac{\text{počet provedených příkazů (řádků)}}{\text{celkový počet příkazů (řádků)}}$, nebo lze tento poměr pro větší přehlednost převést na procenta: $\frac{\text{počet provedených příkazů (řádků)}}{\text{celkový počet příkazů (řádků)}} \times 100$. Zde je nutné si uvědomit, že stoprocentní pokrytí řádků kódu opravdu neznamená stoprocentní otestování kódu. Uvažujme, že chceme otestovat metodu pro výpočet výše vánočních odměn, jejíž kód je vidět na výpisu 5. Metoda jako parametr přijímá částku, kterou je třeba rozdělit mezi zaměstnance. Výstupem metody je textový řetězec, který obsahuje jméno zaměstnance a odměnu. Výše této vánoční odměny zaměstnance je počítána jako poměr celkové částky ku počtu zaměstnanců. Metoda by neměla vypisovat nic, pokud je celková výše odměn nekladná nebo pokud nejsou Vánoce (není prosinec).

Výpis 5: *Testovaná metoda pro výpis odměn zaměstnancům (zdroj: autor)*

```

1 public static String getChristmasRewards(float reward) {
2     String[] employees = null;
3     StringBuilder output = new StringBuilder();
4     if (reward > 0 && Calendar.getInstance().get(Calendar.MONTH) == 11 )
5         employees = new String[] { "Alice", "Bob" };
6     float employeeReward = reward / employees.length;
7     Stream.of(employees).forEach( employee -> {
8         output.append(String.format("Reward for %s is %f.%n", employee, employeeReward));
9     });
10    return output.toString();
11 }

```

Kdyby byla tato metoda testována jednoduchým Unit testem, který by jako vstup dal částku „1000“, a zároveň by buď byl prosinec, nebo by Unit test ovlivnil chování časového elementu např. za užití *mocku* (viz kapitola 3.3.5), bylo by dosaženo stoprocentního pokrytí. Toto by mohlo vést k domněnce, že je otestováno dostatečně a není metodu potřeba dále testovat na další vstupy.

Dle další metriky, kterou Ron Patton označuje jako „úplnou analýzu větvení programu“ (Patton, 2012, s. 103) (angl. *branch coverage*), by však toto otestování jako stoprocentní označeno nebylo. V této metrice je monitorováno, zda jsou otestovány ne všechny řádky programu, ale všechny větve programu. Ve výpisu 5 totiž na řádce 4 dochází k větvení programu. Pokud je celková rozdělovaná částka větší než nula nebo není prosinec, chová se program jinak, než pokud tomu tak není – jsou zde tedy dvě větve a otestována je pouze jedna.

Vzoreček pro výpočet metriky je podobný jako v předchozím případě, jen se nepočítá s řádky kódu, ale s větvemi: $\frac{\text{počet testovaných větví}}{\text{celkový počet větví}}$. Podobně pokud nechceme výsledek jako poměr, ale jako procento, stačí zlomek vynásobit stem: $\frac{\text{počet testovaných větví}}{\text{celkový počet větví}} \times 100$. Pokud i v této metrice chceme dosáhnout sta procent, je nutné zajistit, aby byl otestován i průchod druhou větví. Stačí tedy zadat částku menší než nula, což povede k tomu, že při tomto průchodu sice nebudou otestovány všechny řádky, nicméně oba testy otestují oba průchody. Dle prvního vzorečku tak metrika nabyde hodnoty 2/2, respektive 100 %.

Test, který bude testovat druhý průchod, v tomto konkrétním případě dokonce odhalí chybu – na řádku 6 bude vyhozena výjimka `java.lang.NullPointerException`, jelikož pole `employees` je naplněno hodnotami jen v případě, že podmínka z řádku 4 je vyhodnocena jako pravda. V opačném případě je pole v nedefinovaném stavu – null – a není tudíž možné na něm přistupovat k atributu `length`. I kdyby byl řádek 6 vykonáván jen v případě neprázdného pole `employees`, výjimka `java.lang.NullPointerException` by byla vyhozena o řádek dále. Pro správný průchod by mohl být celý blok řádků 6–10 posunut tak, aby byl vykonáván jen v případě splnění podmínky. V opačném případě by rovnou mohl být vrácen řetězec informující o nulových výplatách odměn v tomto měsíci.

Existuje ještě jedna metrika týkající se pokrytí kódu testy, podle které ani při těchto dvou průchodech není otestování kódu stoprocentní. Metrika je označována jako „úplná analýza podmínek“ (Patton, 2012, s. 103). Při ní by podmínka na řádku 4 výpisu 5 měla být otestována celkem čtyřikrát – měly by být otestovány všechny vzájemné kombinace obou částí podmínek – odměna větší než nula, odměna menší nebo rovna nule v kombinaci s měsícem prosincem a měsícem jiným, než je prosinec. V tabulce 2 jsou vidět všechny čtyři kombinace této složené podmínky.

Tabulka 2: Kombinace možností testování podmínky z výpisu 5 (zdroj: autor)

Vyhodnocení první části podmínky	Vyhodnocení druhé části podmínky	Vyhodnocení celé podmínky	Větev	Řádky
0 [-5000]	0 [10]	0	1.	1, 2, 3, 4, 6, chyba
1 [10000]	0 [9]	0	1.	1, 2, 3, 4, 6, chyba
0 [0]	1 [11]	0	1.	1, 2, 3, 4, 6, chyba
1 [10000]	1 [11]	1	2.	1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11

Pro stoprocentní hodnotu u metriky testování příkazů by stačil jeden test, který by odpovídal čtvrtému řádku tabulky. Pro stoprocentní ohodnocení i u analýzy větvení by stačily testy odpovídající čtvrtému a libovolnému řádku z řádků jedna až tři tabulky 2. V případě úplné analýzy podmínek je pro stoprocentní naplnění metriky nutné spustit celkem čtyři testy,

přičemž každý by měl odpovídat jinému řádku z tabulky 2. V případě, že by do složené podmínky přibyla další podmínka (např. `&& reward >= Balance.getCurrent()`), která by ještě navíc brala v potaz, zda je na účtu dostatek prostředků pro vyplacení odměn, bylo by nutné navíc testovat i různé kombinace předchozích součástí podmínky s touto. Celkem by pak bylo nutné provést osm testů. To vše jen pro otestování této metody, která má celkem pouhých jedenáct řádků. Při přidání další části podmínky by to bylo šestnáct testů, počet různých stavů podmínky roste exponenciálně s počtem jejích částí. Počet nutných testů totiž odpovídá následujícímu vzorci: $\text{počet testů} = 2^{\text{počet částí podmínky}}$.

V praxi patrně rozpočet na testování nebude dostačující, aby se tato metrika blížila stu procentům. (Patton, 2012, s. 104) považuje stoprocentní splnění u složitějších podmínek doslova za nemožné. V praxi se ani stu procentům nebude blížit první metrika – analýza příkazů a řádků programu. Například Google v roce 2014 podrobil analýze svých šest set padesát aktivních projektů (čítajících celkem přes 100 000 příspěvů do úložiště), které byly psány v různých jazycích (C++, Java, Go, JavaScript, Python). Z analýzy vyplynulo jednak to, že u Googlu jsou značné rozdíly v pokrytí řádků kódu testy – 56,6 procent u C++ oproti 84,2 procentům u Pythonu. Hlavní, s ohledem na obsah této kapitoly, je ale zjištění, že ani u Javy se pokrytí řádků kódu testy neblíží stu procentům – u Javy pokrytí činí pouhých 61,2 procenta (Ivanković, 2014). Toto průměrně relativně nízké číslo může být způsobeno několika projekty, které netestují vůbec. Možná více vypovídající hodnotou než průměr je medián, ten však v článku byl uveden pouze dohromady za všechny jazyky a byl roven 78 procentům (Ivanković, 2014). To znamená, že polovina testovaných projektů má nižší než 78procentní pokrytí testy a polovina projektů má pokrytí vyšší. Za zmínku stojí, že i přes to, že v článku je analyzováno pouze pokrytí řádků kódu testy, tak v diskuzi pod článkem jsou příspěvky, které volají také po zveřejnění analýzy pokrytí větví kódu; jsou však bez odpovědi.

3.4.3 Cyklomatická složitost

Pro měření složitosti kódu bývá často používána metrika, která je označována jako cyklomatická složitost, nebo McCabeova cyklomatická složitost (Brooks, nedatováno). Tuto metriku poprvé představil Thomas McCabe v roce 1976 v článku (McCabe, 1976).

Tato metrika se většinou používá pro určování toho, jak jsou složité jednotlivé metody, někdy i pro to, jak složité jsou celé třídy, v původní práci (McCabe, 1976) pracuje autor s pojmem modul, který je ale v daném kontextu bližší spíše metodám než třídám.

Pokud je metoda jednoduchá, je snadnější na pochopení, je více udržitelná a je menší šance, že v ní bude chyba (Brooks, nedatováno). Naopak u složitějších metod je daleko složitější pochopit, co dělají, a proto jsou i náchylnější k tomu, aby v nich byly chyby. Dalším problémem složitých metod je to, že je obtížné, někdy až nemožné, je pořádně otestovat. (Desikan, 2006, s. 67) uvádí, že metody s cyklomatickou složitostí pod 10 jsou snadno testovatelné, metody se složitostí 10-20 se testují obtížněji, metody se složitostí 20-

40 jsou už velmi obtížně testovatelné a metody se složitostí větší než 40 jsou už netestovatelné.

To, jak je daná metoda složitá, závisí dle této metriky zjednodušeně řečeno na tom, kolik existuje možných průchodů touto metodou. Pokud metoda neobsahuje žádné podmínky, cykly apod., má jen jeden možný průchod a cyklomatickou složitost jedna. Pokud obsahuje jednu podmínku, je cyklomatická složitost dva atd. Nejlépe to je vidět na následujícím příkladu ve výpisu 6.

Výpis 6: Ukázková metoda pro určení cyklomatické složitosti (zdroj: autor)

```
1 public static void workOut(Machine machine, int reps, Person person)
2     throws InterruptedException {
3     while(machine.isOccupied()) {
4         Thread.sleep(10000);
5     }
6     for(int i = 0; i < reps; i++) {
7         if(!person.tired())
8             person.trainOn(machine);
9     }
10 }
```

Tato metoda má cyklomatickou složitost rovnou čtyřem. Metoda začíná se složitostí jedna. Za `while` na řádce tři, `for` na řádce šest a `if` na řádce sedm získává dle této metriky vždy po jednom bodu, celkem tedy čtyři body.

Úkolem této metriky je ukázat, jak je daný kód složitý, a to pomocí jednoho čísla, což se však ne tak úplně daří ve všech případech. (Hummel, 2014) uvádí příklad metody, jejímž cílem je převedení vstupního parametru – čísla 0 do 11 – na jméno měsíce. Tato metoda je implementovaná pomocí příkazu `switch` a má tedy 12 možných průchodů pro validní čísla a další pro nevalidní. Metoda je to velmi jednoduchá, avšak metrika by ji označila jako středně komplikovanou metodu s cyklomatickou složitostí 14 (Hummel, 2014). Stejný zdroj (Hummel, 2014) uvádí i druhý příklad metody pro sčítání složených čísel, jež uvádím na výpisu 7.

Výpis 7: *Metoda pro součet složených čísel (zdroj: Hummel, 2014)*

```

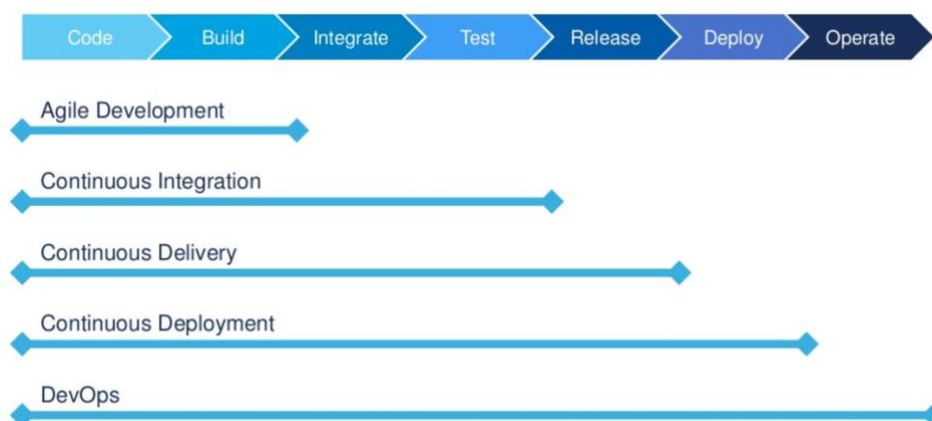
1 int sumOfNonPrimes(int limit) {
2     int sum = 0;
3     OUTER: for (int i = 0; i < limit; ++i) {
4         if (i <= 2) {
5             continue;
6         }
7         for (int j = 2; j < i; ++j) {
8             if (i % j == 0) {
9                 continue OUTER;
10            }
11        }
12        sum += i;
13    }
14    return sum;
15 }

```

Metoda má dle metriky složitost 5, tedy velmi nízkou. Na druhou stranu, kdyby bez názvu byla dána programátorovi ke zjištění toho, co má vykonávat, měl by s tím daleko větší potíže než s předchozí metodou pro převod čísel na měsíce (jež měla cyklomatickou složitost 14, tedy střední). Proto je nutné i tuto metriku brát s rezervou, podobně jako metriky předchozí. Rovněž jako stoprocentní pokrytí kódu testy neznamená, že testovaný software je bez chyb, tak nízké skóre v cyklomatické složitosti vždy nemusí znamenat, že metoda není příliš složitá a není proto nutné ji přepsat.

3.4.4 Continuous Delivery, Continuous Deployment

S pojmem CI úzce souvisí dva další pojmy CDV a CDP. Tyto pojmy bývají často zaměňovány a je obtížné říci, že panuje obecně uznávaná shoda na jejich definicích. Jedním z uznávaných paradigmat je přesvědčení, že rozdíl mezi CDV a CDP je v počtu úkonů, které zastřešují. Kdybychom tedy měli pojmy seřadit podle rozsahu úkonů, které jsou při nich vykonávány, vypadal by řetězec takto: CI → CDV → CDP. Podrobnější rozdělení činností je vidět na obrázku 5.



Obrázek 5:
Srovnání CI, CDL, CDP (zdroj: RightScale 2014, s. 8)

Z obrázku vyplývá, že do CI spadá „pouze“ část automatizace sestavení, integrování a otestování. Automatizace nasazení na libovolné neprodukční prostředí spadá pod CDV a automatické nasazení zákazníkovi je až součástí CDP.

Tomuto pojetí odpovídají i definice CDV a CDP dle (Fitzgerald a Stol, 2015): „*Continuous delivery is the practice of continuously deploying good software builds automatically to some environment, but not necessarily to actual users*“ a „*Continuous deployment implies continuous delivery and is the practice of ensuring that the software is continuously ready for release and deployed to actual customers*“. Rozdíl mezi CDV a CDP je tedy v tom, zda automatizace nasazení funkčního sestavení probíhá pouze do nějakého prostředí (CDV), nebo zda je automatizované i nasazování na produkční prostředí – zákazníkům (CDP).

Oproti tomuto pojetí stojí pojetí další, ne méně rozšířené a uznávané. V něm se tolik nepoužívá termín CDP. Jako společný termín pro obě dvě funkcionality je zde používáno CDV. Tento postoj uznává i autor rozšířené knihy Continuous Delivery – Jez Humble (Humble, Farley, 2010). Ten při jeho prezentaci na nedávné Agile konferenci v Orlandu použil tuto definici CDV: „*The ability to get changes – features, configuration changes, bug fixes, experiments – into production or into the hands of users safely and quickly and in a sustainable way.*“ (Humble, 2017). Z této definice tedy vyplývá, že do CDV spadá i zautomatizované nasazení na produkční prostředí, což je v rozporu s předchozím přístupem.

Jak vidno, nepanuje konsenzus na přesné definici CDV a CDP, a tak je nutné např. při studiu odborné literatury věnovat těmto pojmům zvýšenou pozornost. Rovněž tak v praxi při jednání se zákazníkem je třeba brát na zřetel, že každá ze zúčastněných stran si může pod těmito pojmy představit zcela odlišné plnění.

Ze své osobní zkušenosti mohu sice potvrdit, že jsem se častěji setkával s druhým pojetím – s tím, že CDV je vnímáno jako automatizace celého procesu nasazení, nebo možnosti nasazení na produkční prostředí, to však neznamená, že je obecně rozšířenější. Avšak podle mého osobního názoru toto pojetí dává větší smysl i z významového hlediska užitých slov – „*delivery*“ – dodávka (zákazníkovi), tedy na produkční prostředí. V praxi je ale možné setkat se s oběma pojetími a je tedy potřeba s tím počítat.

3.4.5 DevOps

DevOps je dalším z termínů, které mají blízko k CI, podobně jako CDV a CDP. Při jejich vzájemném srovnání v předchozí kapitole bylo dokonce na obrázku 5 DevOps vnímáno jako další krok automatizace po CDP, což může být možná na první pohled nelogické. Dále v kapitole bude vysvětleno, proč tomu tak není.

Slovo DevOps je slovo složené ze dvou – „*development*“ (vývoj) a „*operations*“ (provoz). Při vývoji software obecně existují dva základní přístupy. Prvním z nich je ten, že vývojářský tým se stará jen o vývoj dané aplikace. Když je ta považována za hotovou, je předána jinému týmu, který se stará o její nasazení k zákazníkovi (někdy tyto úkony může

provádět ještě také původní vývojářský tým) a o její provoz. Vývojářský tým se tak v tomto případě stará jen o psaní kódu – např. v Javě – a vůbec neřeší problémy typu na jakém stroji je pak aplikace nasazena, jak jsou logy zálohovány, případně když nastane problém, nemusí se na daný stroj přihlásit a aplikaci restartovat, nebo ho jinak řešit (Hüttermann, 2012, s. 16–20). Typicky dokonce ani k danému stroji nemá přístup. Často se při aplikaci tohoto přístupu stává, že když v aplikaci nastane problém, tak místo toho, aby byl urychleně vyřešen, začne vzájemné obviňování obou týmů ze zapříčinění tohoto problému (Hüttermann, 2012, s. 21–25).

Oproti tomuto přístupu odděleného vývoje a provozu stojí hnutí DevOps, které se naopak snaží o to, aby tyto dva týmy spolu spolupracovaly co nejvíce; (Gartner, nedatováno) definuje DevOps takto: *„DevOps represents a change in IT culture, focusing on rapid IT service delivery through the adoption of agile, lean practices in the context of a system-oriented approach. DevOps emphasizes people (and culture), and seeks to improve collaboration between operations and development teams. DevOps implementations utilize technology — especially automation tools that can leverage an increasingly programmable and dynamic infrastructure from a life cycle perspective.“*. Z této definice je patrný větší důraz na spolupráci obou týmů oproti předchozímu, možná i běžnému přístupu. Také je z definice daleko více patrná spojitost s CI (a rovněž s CDV a CDP), a to díky důrazu na používání automatizačních nástrojů. Pokud je tedy třeba alespoň částečně automatizovat proces od kompilace kódu po jeho nasazení zákazníkovi, je hlubší komunikace mezi oběma týmy nutná – vývojářský tým se při aplikaci CDP bez znalostí od týmu běžně zajišťujícího provoz obejde jen stěží.

Z větší mezitýmové spolupráce mohou těžit oba týmy – vývojářský tým může totiž psát software, který se dá snadněji provozovat – z vlastní zkušenosti vím, že už jen takový detail, jako je větší pozornost věnována podrobnějším zprávám v logu, může značně usnadnit hledání chyb při nestandardním chování aplikace. Oba týmy by totiž měly mít stejný finální cíl – fungující software. V DevOps by totiž všichni členové týmu měli mít povědomí i o širším kontextu a nekoukat jen slepě na „svoji“ část práce (Hüttermann, 2012, s. 16–20).

Závěrem k termínu DevOps považuji za vhodné ještě zmínit, že nutně neznamená ani nepřikazuje automatizaci celého procesu od nahrání kódu do úložiště po jeho nasazení zákazníkovi. Také dle definice DevOps nemusí znamenat zrušení týmu starajícího se o provoz s tím, že o provoz se nyní bude starat vývojářský tým. Znamená to jen větší spolupráci obou týmů, a to i s možností toho, že se oba týmy spojí. S CI má DevOps společné především to, že jak CI, tak DevOps kladou důraz na automatizaci.

4 Porovnání a výběr aplikací a nástrojů

V této kapitole jsou analyzovány typy aplikací a nástrojů, které jsou vhodné pro zavedení CI u Java projektů. Východiskem pro požadavky na funkcionalitu u jednotlivých typů aplikací bude především předchozí kapitola 3, kde jsou kromě procesu CI rozebrány mj. nejlepší praktiky a přínosy CI. S ohledem především na tyto rozebírané přínosy budou v rámci každého typu aplikací (a nástrojů) vybrány ty nejvhodnější aplikace (a nástroje) pro závěrečnou praktickou část této diplomové práce.

V první podkapitole je popsána metoda hodnocení aplikací a jejich následného výběru pro závěrečné řešení. Další podkapitoly jsou vždy věnovány jednomu typu aplikace (nástroje). V úvodní části kapitol jsou obecně popisovány typy aplikací a jejich základní hlavní funkce, a to především z pohledu CI. Následně jsou popisovány konkrétní aplikace a shrnuty jejich výhody a nevýhody. V závěru každé kapitoly je pak na základě metody popsané v kapitole 4.1 vzájemné porovnání jednotlivých aplikací a vybrána jedna z nich. Ta pak bude součástí praktického výstupu z této práce – integrovaného CI řešení pro Java projekty, založeného na technologii Docker.

4.1 Metoda hodnocení a výběru

V této kapitole je popsán způsob výběru aplikací a popsána metoda, na základě které budou jednotlivé aplikace v rámci typu aplikace mezi sebou srovnány. Na základě procesu CI, který je blíže popsán v kapitole 3, a především na základě schématu na obrázku 1, na kterém je tento proces zachycen, byly vybrány celkem tři skupiny nástrojů a aplikací. Ty jsou dále vzájemně porovnávány a z nich je vybrána právě jedna, která bude součástí praktické části práce.

Skupinou, která se logicky nabízí jako první, jsou CI servery (kapitola 4.2). Jak už z názvu vyplývá, jsou pravděpodobně hlavní částí CI celku a mají toho na práci relativně nejvíce. Mají na starost komunikaci s dalšími aplikacemi a nástroji, které jsou také popsány v samostatných kapitolách. CI servery komunikují například se systémy správy verzí – VCS (kapitola 4.3). Z něj dostávají nejnovější verze kódu do něj nahraných, na nichž spouští např. Unit testy nebo statickou analýzu kódu (kapitola 4.4). Samozřejmostí je i kompilace kódu např. do souboru typu .jar. Ten pak může být nahrán na testovací prostředí, kde probíhají integrační testy s jinými aplikacemi, po úspěšném otestování může také testovanou aplikaci rovnou nasadit na produkční prostředí.

Aplikace pro praktickou část řešení jsou vybírány na základě vícekritériálního rozhodování. Jednotlivá kritéria jsou v každé skupině samozřejmě jiná, nemá smysl porovnávat, zda nástroj pro testování umí pracovat jak se Subversion, tak i s Gitem. Stejně tak pozbývá smyslu vyžadovat od systému správy verzí, aby počítal statistiky o pokrytí kódu testy nebo

komplexitě metod. Jsou ale i kritéria, která se budou vyskytovat u více typů analyzovaných aplikací. Je jím například existence možnosti oficiální placené podpory nebo „živost“ projektu – to, jestli je projekt stále aktivně podporován a vyvíjen, nebo do něj nikdo už několik měsíců nebo let nepřispěl řádkou kódu.

Každá aplikace bude v každém kritériu bodově ohodnocena jedním až pěti body. Slovní ohodnocení bodů je následující:

1. velmi špatné,
2. spíše špatné,
3. neurčitelné,
4. spíše dobré,
5. velmi dobré.

Více bodů znamená lepší ohodnocení. Tři body neznamenají průměrné ohodnocení v rámci kritéria, nýbrž nemožnost ohodnocení aplikace v tomto kritériu vlivem např. nedostatečné dokumentace. Takto by však aplikace měly být hodnoceny jen opravdu ve výjimečných případech.

Podkladem pro hodnocení aplikací je primárně dokumentace, dále pak veřejná demo aplikací. V neposlední řadě ale také odborné publikace a články, které aplikace zkoumají, případně i porovnávají.

Jako metoda, která byla vybrána pro hodnocení vah jednotlivých kritérií, byla určena tzv. modifikovaná Fullerova metoda párového porovnání, někdy též známa jako metoda Fullerova trojúhelníku (Smrž, 2006, s. 19). Při ní se kritéria K_1 až K_n zanesou do Fullerova trojúhelníku. Kritéria se pak ve dvojicích porovnají každé s každým a označí se vždy to významnější ze dvojice. Váha daného kritéria je pak rovna zlomku, kde v čitateli je počet označení daného kritéria jako důležitějšího a ve jmenovateli celkový počet porovnání. Takto se váha kritérií počítá v základní Fullerově metodě. V rozšířené verzi metody se k čitateli přičte jednička a ke jmenovateli celkový počet kritérií. Tím se zamezí tomu, aby kritérium, které je sice důležité, ale ani v jednom vzájemném porovnání nebylo preferováno, mělo celkovou nulovou váhu (Friebelová, nedatováno). Uvedme to na příkladu. Mějme kritéria K_1 až K_4 , jejichž vzájemné preference jsou zaneseny zvýrazněním do Fullerova trojúhelníku v tabulce 3.

Tabulka 3: Příklad preferencí kritérií (zdroj: autor)

Kritéria	Preference		
K_1	K_1 K_2	K_1 K_3	K_1 K_4
K_2	×	K_2 K_3	K_2 K_4
K_3	×	×	K_3 K_4

Z tabulky vyplývá, že kritérium K_1 je preferováno před kritérii K_2 a K_4 a není preferováno před kritériem K_3 . Kritérium K_1 je tedy preferováno celkem dvakrát. Analogicky lze zjistit vzájemné preference i u ostatních kritérií. Z tabulky je také vidět, že kritérium K_4 nebylo preferováno ani jednou. To by při použití nerozšířené Fullerovy metody znamenalo, že by při porovnávání nebylo vůbec bráno v potaz. Právě z tohoto důvodu bylo rozhodnuto o použití modifikované Fullerovy metody. Konkrétní výpočet vah jednotlivých kritérií je zanesen v tabulce 4.

Tabulka 4: Výpočet vah jednotlivých kritérií (zdroj: autor)

Kritérium	Počet preferencí	Počet porovnání	Výpočet – základní Fullerova metoda	Absolutně vyjádřená hodnota	Výpočet – rozšířená Fullerova metoda	Absolutně vyjádřená hodnota
K_1	2	6	$\frac{2}{6}$	0,33	$\frac{2+1}{6+4}$	0,30
K_2	1	6	$\frac{1}{6}$	0,17	$\frac{1+1}{6+4}$	0,20
K_3	3	6	$\frac{3}{6}$	0,50	$\frac{3+1}{6+4}$	0,40
K_4	0	6	$\frac{0}{6}$	0,00	$\frac{0+1}{6+4}$	0,10
Kontrola součtem	1,00	×	1,00	1,00	1,00	1,00

V tabulce je vidět, že kritérium K_4 by mělo při použití nerozšířené metody nulovou váhu, kdežto při použití rozšířené metody má váhu 0,10. V posledním řádku je kontrola. Součet jednotlivých vah by totiž vždy měl dát dohromady 100 procent neboli 1,00.

Tímto způsobem získanými váhami bude násobeno bodové hodnocení jednotlivých kritérií (1-5), které získala každá aplikace. Výsledkem tak bude skóre, v němž bude započítáno to,

jak dobře aplikace daná kritéria splnila, ale bude bráno v potaz i to, že některá kritéria jsou důležitější než jiná. Aplikace, která v každé kategorii dosáhne nejvyššího skóre, bude následně součástí praktické části diplomové práce.

4.2 Continuous Integration servery

CI server (z Continuous Integration Server – server pro průběžnou integraci), někdy též označován jako *build server* (server pro sestavování) (Rouse, nedatováno), je základním prvkem CI. Označení *build server* je patrně mírně zavádějící a označuje trochu jiný typ aplikací než širší označení CI server. Možnost sestavení softwaru je totiž pouze jedna z funkcí CI serveru. Aby se ale jednalo o plnohodnotný CI server, měl by toho umět daleko více – na základě kapitoly 3 a v ní popsaném procesu CI se dají odvodit následující minimální požadavky na funkcionalitu:

- možnost obousměrného propojení s verzovaným úložištěm kódu,
- sestavení kódu,
- spuštění jednotkových testů,
- poskytování zpětné vazby vývojářskému týmu např. formou e-mailu,
- nasazení softwaru na testovací případně produkční prostředí,
- spuštění systémových a integračních testů,
- integrace s nástroji pro statickou analýzu kódu,
- možnost plánování, konkrétně např.:
 - periodické (např. každý den o půlnoci) spouštění libovolného výše jmenovaného kroku,
 - možnost podmíněného spouštění dalších kroků v závislosti na výsledcích kroků předchozích,
- tvorba reportů z průběhu a výsledků jednotlivých kroků

CI server je tedy stěžejním prvkem jakéhokoliv CI řešení. Sám sice neposkytuje dostatečnou funkcionalitu pro komplexní CI řešení, je ale nutné, aby uměl požadavky na toto řešení delegovat na další aplikace, výsledky interpretovat a dále s nimi pracovat, nebo je rozumnou formou zobrazit uživatelům.

CI serverů existuje velké množství. Jen na stránce alternativeto.net je více než dvacet produktů, které se dají označit jako „build“ či dokonce i CI servery (Alternativeto, 2017a).

Podobně rozsáhlý seznam je vidět i v anketě užívání CI serverů na serveru InfoQ (Humble, 2014).

S ohledem na zaměření této práce budou porovnány jen ty nejvíce rozšířené a užívané. Nemělo by smysl vytvářet komplexní CI řešení, které by bylo sice nejvíce funkční, za to ale postavené na neznámé (možná i nové) a nevyužívané technologii. Řešení by sice mohlo být objektivně nejrobustnější a nabízelo by nejvíce funkcí, ovšem kvůli nerozšířenosti dané technologie by nebylo využíváno.

Rovněž některé CI servery jsou nabízeny jen jako placené SaaS (z angl. software as a service, česky software jako služba), tudíž je nemožné si je stáhnout a provozovat je na vlastním serveru v Dockeru, což je nutná podmínka pro to, aby se mohly stát součástí CI řešení praktické části práce.

Na základě těchto omezení byly vybrány tyto čtyři CI servery, které budou v dalších kapitolách analyzovány a porovnány:

- Go CD,
- Hudson,
- Jenkins,
- GitLab.

Do srovnání se nedostaly i některé, dle průzkumu (Humble, 2014), velmi rozšířené CI servery, a to většinou kvůli licenci. Společnost Atlassian u produktu Bamboo nabízí jen časově omezené demo (Atlassian, 2017). Team Foundation nabízí zdarma také jen demo (Microsoft, 2017). Travis CI je zdarma jen pro open-source projekty (TravisCI, 2017). Omezené možnosti pro komerční užití má i TeamCity od společnosti JetBrains, ta funguje na byznys modelu freemium – software je sice možné používat i pro komerční účely, instalovat jej na vlastní server, ale jen v omezené konfiguraci a je využitelný jen pro menší projekty (JetBrains, 2017).

4.2.1 Kritéria a stanovení vah

V této kapitole budou vyjmenována a popsána jednotlivá kritéria, která budou u CI serverů hodnocena. V závěrečné části kapitoly jsou kritéria porovnána podle důležitosti a na základě tohoto srovnání jim jsou přiřazeny váhy.

Aktivita na projektu

Toto kritérium odráží to, zda a nakolik je projekt stále aktivní. Za aktivní projekt je považován ten, do něž je aktivně přispíváno např. na GitHubu, pokud se jedná o projekt s otevřenými zdrojovými kódy. U projektů, které nemají otevřené zdrojové kódy a nedá se

tedy takto ověřit, jak moc se do projektu stále přispívá, bude z hlediska tohoto kritéria hodnoceno to, zda stále vycházejí nové verze, či zda už např. několik měsíců žádná nová verze aplikace nebyla vypuštěna.

Funkce

Patrně nejdůležitější kritérium. Je v něm hodnoceno, které funkce (z funkcí jmenovaných na začátku kapitoly 4.2) nabízí a které v něm chybí. Jedná se opravdu o základní požadavky na funkcionalitu CI serveru, ne nějaké drobnosti a rozšíření. Proto je toto kritérium považováno za nejdůležitější.

Licence

Ne všechny licence musejí být dostatečné pro provozování vlastního CI serveru. Některé mohou být podmíněné počtem uživatelů, kteří aplikaci mohou používat. Jiné mohou být například zdarma použitelné pouze pro nekomerční účely, což je také nevhodné. Ideální licence pro účely této práce je ta, kterou lze používat bez omezení i na komerční projekty a zároveň umožňuje zobrazení zdrojového kódu aplikace (jelikož ten je mnohdy lepším zdrojem dokumentace než dokumentace samotná).

Možnost placené podpory

Toto kritérium souvisí s kritériem široké uživatelské základny. V případě, že je aplikace užívána pro méně standardní účely, může nastat situace, že je nutné objednat placenou podporu od lidí, kteří se na danou aplikaci specializují, ideální je, pokud ji i vyvíjejí. Toto kritérium se pravděpodobně nebude týkat velké části potenciálních uživatelů, proto bude hodnoceno při vzájemném porovnání s ostatními kritérii nízko.

Předpřipravený podporovaný Docker obraz

Toto kritérium není kriticky důležité. Vždy je možné vytvořit vlastní Docker obraz (pojem Docker obraz je blíže vysvětlen v kapitole 5) s aplikací. Na druhou stranu, pokud existuje oficiálně spravovaný a aktualizovaný Docker obraz, jedná se o výhodu. Pravděpodobně je totiž robustněji otestovaný, než by byl nově vytvořený obraz. Také vývojáři dané aplikace mají větší zkušenosti a znalosti specifické aplikace, tak jim mohou být známy skutečnosti, které se ani do dokumentace nedostaly. Díky tomuto může oficiálně spravovaný Docker obraz znamenat větší stabilitu celého řešení. To, že Docker obraz existuje, však neznamená, že se svým nastavením bude stoprocentně hodit do praktické části. To neznamená nepřekonatelnou překážku. Docker obraz je vždy možné oddělit a dále rozšířit pro konkrétní potřeby projektu.

Rozšiřitelnost

Zde se hodnotí, zda je možné CI server nějak rozšířit o nové funkce. Ať už napsáním vlastního kódu, který je možné s aplikací integrovat, nebo o možnost rozšíření CI serverů prostřednictvím doplňků (pluginů). Základní funkce totiž nemusí zcela vyhovovat každému. Pokud je systém zcela uzavřen, je možné, že v průběhu projektu vyvstane potřeba nějakého nestandardního použití CI serveru. Pokud by server nebylo možné rozšířit, jedinou možností by bylo migrovat na jiný server.

Velikost uživatelské základny

I když byly předem z porovnání a hodnocení odebrány aplikace, které jsou neznámé nebo méně často užívané, kritérium, které hodnotí tuto vlastnost, zůstalo. Zde ale nejde jen o to, zda existuje dostatek zkušených potenciálních uživatelů praktického řešení této práce. Široká uživatelská základna je pro software vždy výhodou minimálně z jednoho hlediska – kdykoliv nastane jakýkoliv problém (např. vlivem nedostatečné dokumentace), je možné vznést dotaz na komunitním fóru. V případě rozšířené technologie, která je zrovna užívaná značným množstvím lidí, dostane podobný dotaz často velmi rychle několik relevantních odpovědí, což značně šetří čas. Ať už při provozování dané aplikace, nebo při její instalaci a integraci s jinými nástroji.

Stanovení vah kritérií

V předchozích kapitolách diskutovaná kritéria byla vzájemně porovnána. Výsledky tohoto porovnání jsou zaneseny v tabulce 5. V tabulce 5 jsou rovněž znázorněny výpočty vah daných kritérií. Kritéria jsou označena zkratkami CIK_1 – CIK_7 . CIK_1 odpovídá prvnímu popsanému kritériu – aktivita na projektu – a CIK_7 odpovídá poslednímu popsanému kritériu.

Tabulka 5: Váhy jednotlivých kritérií CI serverů (zdroj: autor)

Kritérium	Preference						Počet preferencí	Výpočet a výsledná váha kritéria
Aktivita na projektu (CIK_1)	CIK_1 CIK_2	CIK_1 CIK_3	CIK_1 CIK_4	CIK_1 CIK_5	CIK_1 CIK_6	CIK_1 CIK_7	2	$\frac{2+1}{21+7} \doteq 0,11$
Funkce (CIK_2)	X	CIK_2 CIK_3	CIK_2 CIK_4	CIK_2 CIK_5	CIK_2 CIK_6	CIK_2 CIK_7	6	$\frac{6+1}{21+7} \doteq 0,25$
Licence (CIK_3)	X	X	CIK_3 CIK_4	CIK_3 CIK_5	CIK_3 CIK_6	CIK_3 CIK_7	3	$\frac{3+1}{21+7} \doteq 0,14$
Možnost placené podpory (CIK_4)	X	X	X	CIK_4 CIK_5	CIK_4 CIK_6	CIK_4 CIK_7	1	$\frac{1+1}{21+7} \doteq 0,07$
Předpřipravený podporovaný Docker obraz (CIK_5)	X	X	X	X	CIK_5 CIK_6	CIK_5 CIK_7	0	$\frac{1}{21+7} \doteq 0,04$
Rozšiřitelnost (CIK_6)	X	X	X	X	X	CIK_6 CIK_7	4	$\frac{4+1}{21+7} \doteq 0,18$
Velikost uživatelské základny (CIK_7)	X	X	X	X	X	X	5	$\frac{5+1}{21+7} \doteq 0,21$
Kontrola	X	X	X	X	X	X	X	1,00

Z tabulky vyplývá, že největší váhy mají kritéria CIK_2 – funkce a CIK_7 – velikost uživatelské základny. Naopak téměř zanedbatelné váhy mají kritéria CIK_4 – možnost placené podpory a CIK_5 – předpřipravený podporovaný Docker obraz.

4.2.2 Jenkins

Jenkins je pravděpodobně nejznámější CI server ze všech porovnávaných. Dlouhodobě patří také mezi CI servery, které mají nejširší uživatelskou základnu (Smart, 2011, s. 3-4), (CloudBees, nedatováno) a (Mittal, 2015).

Vývoj Jenkinsu započal v roce 2004 Kohsuke Kawaguchi, vývojář společnosti Sun Microsystems. V této době byl však projekt známý ještě pod názvem Hudson. Postupem času byl Hudson používán více a více projekty, až v roce 2008 dostal plnou podporu vedení a Kohsuke Kawaguchi na něm mohl začít pracovat již na plný úvazek (Smart, 2011, s. 3).

Po koupi firmy Sun Microsystems firmou Oracle v roce 2009 nastaly silné neshody ohledně vedení projektu mezi vývojáři a vedením Oraculu. Oracle byl pro striktně kontrolovaný proces vývoje s pomalejším vydáním nových verzí, kdežto vývojáři se chtěli držet původního systému práce vývoje Hudsonu – flexibilního vývoje s častějším vydáním verzí. Tyto a další neshody vedly nakonec k tomu, že byly na přelomu let 2010 a 2011 projekty rozděleny (Smart, 2011, s. 3). Původnímu projektu zůstal název Hudson, jen nyní nespadá pod Oracle, ale během roku 2011 byl převeden pod Eclipse Foundation. Novému projektu bylo vybráno jméno Jenkins a jako open-source projekt je vyvíjen komunitou. Největším přispěvovatelem je stále původní autor – Kohsuke Kawaguchi (Jenkins, 2017a), respektive celá firma CloudBees, kde nyní Kawaguchi pracuje jako CTO.

Nyní bude Jenkins popsán z pohledu hodnocených kritérií. Z hlediska funkcí Jenkins splňuje všechny požadavky jmenované na počátku kapitoly 4.2 – od propojení s verzovanými úložišti kódu (už ve výchozím nastavení bez nutné instalace doplňků podporuje Git i Subversion), po nasazování kódu na produkční prostředí. Pro některé funkce je ovšem nutné doinstalovat doplňky – např. pro lepší zobrazení reportů z prováděných testů. Tím se dostáváme k jednomu z nejsilnějších aspektů používání Jenkinse jako CI serveru. Pro Jenkins existuje nepřeborné množství doplňků, které jej obohacují o nejrůznější funkce. V době psaní této práce je na oficiálních stránkách dostupno rovných 1535 doplňků pro Jenkins (Kawaguchi a Beck, 2017).

Dalším kritériem, které Jenkins plní stoprocentně, je aktivita projektu – na oficiálním úložišti na GitHubu je vidět, že do projektu je stále často přispíváno. Průměrně se jedná zhruba o dvacet příspěvků (u Gitu je používán angl. výraz *commit*) týdně (Jenkins, 2017a). Jenkins je vydáván pod velmi svobodnou licencí, konkrétně se o jedná licenci označovanou jako MIT/X11 (Jenkins, 2017b). Dalším výborně splněným kritériem je možnost placené podpory. Ta v Jenkinsu existuje buď přímo od firmy CloudBees, jež je hlavním přispěvovatelem kódu do projektu a jejímž CTO je sám Kohsuke Kawaguchi, nebo také od dalších firem, jež jsou na seznamu komerční podpory Jenkins (Kawaguchi a Gilmore, 2017). Jenkins má také oficiálně podporovaný Docker obraz, jehož zdrojový kód je na GitHubu. I zde to vypadá, že je kód vyvíjen stejně aktivně, jako je vyvíjeno jádro Jenkins CI (Jenkins, 2017c). Posledním kritériem je velikost uživatelské základny, ta je také výborná – v současné době se jedná o nejvíce využívaný CI server, podrobněji viz začátek kapitoly.

4.2.3 Hudson

Hudson a Jenkins pojí společná historie, jež byla podrobněji popsána v kapitole 4.2.2 věnující se CI serveru Jenkins. Po rozdělení a vyčlenění Jenkinsu do samostatného projektu nastaly pro Hudson horší časy. Sedmdesát pět procent uživatelů následovalo původní vývojáře a přešlo na Jenkins, třináct procent používalo Hudson a zbylých dvanáct procent používalo oba CI servery současně – a to z důvodu průběžné migrace z Hudsonu na Jenkins (Smart, 2011, s. 4). Nicméně z hlediska kritéria uživatelské základny toto není tak důležité,

jelikož se jedná o velmi podobné projekty. Uživatel Hudsonu nebude mít značné problémy přejít na Jenkins a naopak, proto i v tomto kritériu bude hodnocen Hudson pěti body stejně jako Jenkins. Z hlediska hodnocení dalších kritérií je to u některých z nich o poznání horší.

Aktivita na projektu je slabá, možná se dá říci i nulová. Poslední příspěvek do oficiálního úložiště byl proveden před více než rokem (Eclipse Foundation, 2017). Z hlediska splnění základních funkcí je na tom stejně jako Jenkins. S množstvím dostupných doplňků je na tom Hudson sice hůře, nabízí jich ve srovnání s Jenkinsem o tisíc méně. Ovšem i tak je počet tři sta osmdesát pět dostatečný a většina z nich by měla pokrýt běžné případy užití tohoto softwaru (Hudson, 2017). Navíc vzhledem k otevřenosti projektu existuje vždy možnost dopsat si doplněk vlastní. Projekt od verze tři totiž používá otevřenou licenci EPL (Eclipse Public License), dřívější verze byly šířeny pod licencí MIT (Hudson, nedatováno). Oficiální placená podpora pro Hudson ve stejné míře, jako tomu bylo u Jenkinse, není. Nicméně vzhledem k podobnosti obou projektů se v případě potřeby zcela jistě najde dostatečný počet firem, které tuto podporu pro Hudson, vzhledem k podobnosti obou projektů, nabídnou. Posledním ještě nezmiňným kritériem je existence – v tomto případě však bohužel neexistence oficiálního Docker obrazu.

4.2.4 GitLab CI

GitLab je aplikace, která je zde porovnávána dvakrát. Jednou mezi CI servery, jednou mezi VCS servery, jelikož může zastávat obě funkce. GitLab CI je součást GitLabu, která může být zapnuta nebo vypnuta a jako CI může být používán jiný server – např. Jenkins. Tato kombinace Jenkinse jako CI serveru a GitLabu jen jako VCS serveru patrně není nijak neobvyklá, jelikož je zmíněna přímo v dokumentaci (GitLab, 2017f).

Z hlediska hodnocení získává z aktivity na projektu pět bodů (GitLab, 2017a). Splňuje všechny funkce definované na začátku kapitoly 4.2. Kromě nich jej lze ale používat i jako VCS server. V GitLabu lze také využít jeho funkce pro evidenci chyb – angl. *bug tracking* (GitLab, 2017d), proto dostává v kritériu funkce také pět bodů.

Standardní licence MIT, pod kterou je GitLab šířen, zaručuje i v kritériu licence maximální možné pětibodové hodnocení. Přímou GitLab nabízí též možnost placené podpory, jejíž součástí kromě samostatné podpory jsou i některé, pro neplatící uživatele uzavřené, funkce. Je nutné zmínit, že CI funkce jsou přítomny i v základní verzi (GitLab, 2017e). Za možnost placené podpory tak GitLab získává pět bodů. Pět bodů GitLab získává i v dalším kritériu díky tomu, že v oficiálním úložišti je přítomen Docker obraz (GitLab, 2017c).

Nízké hodnocení si GitLab odnáší jen v kritériu rozšiřitelnost. GitLab sice má vlastní doplňky, avšak užití komunitních doplňků GitLab CI v současné době nenabízí. Existuje na to otevřený návrh (Trzeciński, 2016), nicméně ten nebyl ještě implementován. Proto GitLab CI v tomto kritériu dostává jen dva body. Sice v GitLabu zatím není, ale vzhledem k licenci je možné přímo zdrojový kód upravit pro svoje potřeby.

Přesně určit velikost uživatelské základny je vždy obtížné, podrobné statistiky u sebe uvádí jen Jenkins – celosvětově existuje sto tisíc aktivních instalací (Jenkins, 2015). Pro přibližné určení velikosti uživatelské základny v tomto případě bude vycházeno z uživatelských anket (Alternativeto, 2017a) a (Slant, 2017c). Z nich vyplývá, že GitLab CI sice nemá tak velkou oblíbenost jako Jenkins, ale i tak se drží na horních příčkách v oblíbenosti, a tak dostává pět bodů.

4.2.5 GoCD

GoCD je další projekt s bohatou historií. Původně vznikl jako komerční produkt „Cruise“ firmy ThoughtWorks, která stála za vývojem dříve velmi rozšířeného CI serveru CruiseControl. Název Cruise měl sloužit jako pocta tomuto softwaru, nicméně tento název působil spíše zmatení, protože lidé zaměňovali oba nástroje. Proto byl v roce 2010 přejmenován Cruise na Go. Nutno podotknout, název GoCD je také poněkud zavádějící, jelikož program nemá nic společného s jazykem Go. Ten sice vznikl dříve, ale, dle slov jednoho z ředitelů ThoughtWorks Chada Wathingtona, nebyl zdaleka tak populární, jako je tomu dnes (Fowler, 2014).

Nyní k samotnému hodnocení kritérií. Z aktivity na projektu GoCD musí dostat pět bodů, jelikož dle příspěvků na GitHubu projekt skutečně žije, za poslední rok měl mezi desíti a padesáti příspěvky týdně (GoCD, 2017a).

V kategorii funkcí dostává také pět bodů. GoCD má všechny funkce, které jsou definované v kapitole 4.2, ale i některé další, které mohou být navíc rozšířeny doplňky (GoCD, 2017b). GoCD vznikl právě proto, že CruiseControl nebylo možné jednoduše přepsat tak, aby podporoval nejen funkce CI, ale i CD (Fowler, 2014). Rozdíly mezi CI a CD byly popsány v kapitole 3.4.4.

Od roku 2014 se jedná o open-source projekt šířený pod licencí Apache 2.0 (GoCD, 2014), a tak v kritériu licence dostává také pět bodů.

V případě potřeby nabízí placenou podporu mj. sám hlavní vývojář projektu – firma Thoughtworks (Thoughtworks, 2017), což se jeví jako ideální řešení. Kdo jiný může o softwaru vědět víc než ten, kdo ho napsal a kdo ho přes deset let vyvíjí a udržuje. Za toto dostává GoCD také pět bodů.

Pět bodů získává GoCD i za to, že existují oficiálně podporované Docker obrazy – jeden pro GoCD server a jeden pro GoCD agenta (GoCD, 2017c).

Možnost rozšířit GoCD pomocí doplňků existuje. Ovšem počet doplňků, který je zdarma volně šiřitelný a užitelný, je ve srovnání s Jenkinsem omezený (GoCD, 2017d). Kromě těchto doplňků existují i placené doplňky přímo od firmy Thoughtworks, ty jsou ale dostupné jen velkým platícím zákazníkům (Thoughtworks, 2017), proto zde dostává GoCD jen čtyři body.

Pro hodnocení velikosti uživatelské základny budou opět použity uživatelské ankety (Alternativeto, 2017a) a (Slant, 2017c). Z nich vyplývá, že GoCD není tak často užívané jako Jenkins nebo GitLab, proto zde dostává jen čtyři body.

4.2.6 Hodnocení.

Hodnocení testovaných CI serverů je vidět v tabulce 6.

Tabulka 6: Hodnocení CI serverů (zdroj: autor)

Kritérium	Váha	Jenkins	Hudson	GoCD	GitLab
Aktivita na projektu (CIK_1)	0,11	5	1	5	5
Funkce (CIK_2)	0,25	5	5	5	5
Licence (CIK_3)	0,14	5	5	5	5
Možnost placené podpory (CIK_4)	0,07	5	4	5	5
Předpřipravený podporovaný Docker obraz (CIK_5)	0,04	5	1	5	5
Rozšiřitelnost (CIK_6)	0,18	5	4	4	2
Velikost uživatelské základny (CIK_7)	0,21	5	5	4	5
Součet hodnocení	X	35	26	33	33
Celkové skóre po započtení vah	X	5,00	4,15	4,61	4,46

Z tohoto srovnání tedy vychází Jenkins jako nejvhodnější CI nástroj. Ostatní srovnávané nástroje měly buďto menší uživatelskou základnu, nebo nebyly tak rozšiřitelné. Je zde ale nutné znovu zdůraznit, že existuje větší množství nástrojů, které nebyly součástí srovnání vlivem nevhodné licence – Travis CI, TeamCity, Bamboo a Team Foundation. Konkrétní důvody jsou uvedeny na počátku kapitoly 4.2.

4.3 Systémy správy verzí

Pojem systém správy verzí (dále jen VCS) pochází z anglického označení – version control system. V anglicky psané literatuře není však terminologie pro tento druh aplikací sjednocena – často je možné se setkat s označením *revision control system*, *source control management*, nebo jen *source control*. Mezi těmito jmenovanými aplikacemi mohou být drobné rozdíly, v praxi se ale jimi poskytovaná funkcionality často překrývá, takže nemá

větší smysl tyto pojmy rozlišovat (O'Sullivan 2009, s. 2). V této práci bude pro všechny tyto aplikace užívána jednotná zkratka – VCS.

To, jak VCS zapadá do procesu CI, je zřejmé z obrázku 1 v kapitole 3.3.1, stejně tak i z názvu. Právě do VCS vývojáři nahrávají svoje nové verze kódu, na kterých pracovali. Samotný postup práce je prostý. Nejprve si vývojář stáhne nejnovější verzi kódu na svůj lokální počítač. Poté pracuje na změnách (přidá novou funkci, test, odstraní chyby, ...). To může trvat i delší časový úsek – dny, týdny, někdy i více. Průběžně však nový (i nedokončený) kód nahrává zpátky do úložiště. Aby se předešlo nefunkčnímu kódu v úložišti, bývá kód rozdělován do tzv. „větví“. Obvykle existuje jedna hlavní větev (často označována master, development nebo trunk), ve které by měl být nahrán pouze funkční kód. Když někdo začíná pracovat na nové funkci, vychází právě z této větve a založí si vlastní větev (např. označenou názvem nové funkce, nebo číslem chyby, kterou opravuje). Do té pak může nahrávat i rozdělanou, a tedy nefunkční práci. Až když je práce hotova, spojí tuto větev s hlavní větví. Takto může na stejném projektu bez problému pracovat více vývojářů současně, každý na svojí větvi, nebo dva na jedné větvi, další na jiné atd. Další z přínosů VCS je verzování. V každém okamžiku je možné vrátit se zpět k předchozí verzi kódu nahrané do VCS. Tak se zamezí tomu, aby kód byl nenávratně ztracen, pokud ho někdo omylem smaže a tuto změnu aplikuje do VCS. Kdyby byl v tomto případě místo VCS používán FTP server, nezbylo by než doufat, že někdo z vývojářů má zálohu starší verze.

Než budou popsány požadavky na funkce, je nutné v krátkosti vysvětlit dvě odlišné technologie VCS – Git a Subversion. Obě dvě je možné používat k témuž úkolu – verzování a sdílení kódu napříč týmem. Nicméně ovládání je různé a princip, na němž fungují, také – Git je distribuované úložiště, to znamená, že každému jednotlivému vývojáři je lokálně stažena historie všech verzí. Oproti tomu Subversion je centrální úložiště – historie verzí a informace o větvích jsou uloženy jen na centrálním serveru. Git a Subversion má mnoho dalších rozdílů (Deveo, 2017) a nelze jednoznačně říct, které je lepší (Stum, 2009). Oba systémy mají své zastánce i odpůrce – vyloženě pro Git je např. (Grudl, 2009), Subversion preferuje např. (Anon, 2016). Nicméně pro konkrétní řešení bylo třeba zvolit jednu možnost, a tou se stal Git. Rozhodnutí proběhlo na základě dvou vlastností Gitu. Jednak na Gitu je postaveno více open-source projektů a má širší uživatelskou základnu (BlackDuck, 2017) a jednak s ním lze pracovat lokálně, bez připojení k internetu nebo k firemní VPN (např. při cestě vlakem v tunelu). Svě neblahé zkušenosti s centrálním úložištěm umně popisuje David Grudl ve svém článku „Proč opustit Subversion (SVN)“ z roku 2009 (Grudl, 2009).

Ačkoliv existuje velký počet aplikací spadajících do kategorie VCS, dva základní požadavky diskvalifikují mnoho z nich. Většinu z nich včetně dvou nejznámějších – GitHub a Atlassian Bitbucket – buď není možné instalovat na vlastní server (*on premise*), nebo je tato možnost zpoplatněna. Samozřejmě pro projekty s veřejným kódem lze zdarma využívat přímo webové stránky github.com. Pokud je ale poptávka po tom provozovat zdarma VCS aplikaci na vlastním serveru, je počet možných řešení poněkud omezený. Z nyní dostupných VCS

aplikací byly pro podrobnější srovnání vybrány čtyři největší: GitBucket, Gogs/Gitea, GitLab a Gerrit (Slant, 2017b) (Radhakrishnan, 2014).

4.3.1 Kritéria a stanovení vah

VCS aplikací není tak velké množství jako CI serverů, proto jich zde není nutné tolik vyřazovat kvůli nízké uživatelské základně. Navíc zde velikost uživatelské základny není tak důležité kritérium. Jelikož všechny analyzované aplikace podporují Git, je jejich ovládání z hlediska uživatele, který do nich chce přispět, všude stejné – příkazy *git commit* a *git push* budou fungovat všude identicky. Na druhou stranu, širší báze uživatelů i zde může být přínosem kvůli možnosti dotazování se těchto uživatelů na fórech projektu. I některá další kritéria má smysl zde hodnotit, stejně jako byla hodnocena u CI serverů. Nemá ale smysl je znova rozebírat a definovat, co přesně bude hodnoceno, jelikož tak bylo učiněno v kapitole 4.2.1. Kritéria, která byla použita k hodnocení CI serverů a budou použita i zde, jsou:

- aktivita na projektu – VCS_1 ,
- funkce – VCS_2 ,
- licence – VCS_3 ,
- předpřipravený Docker obraz – VCS_4 ,
- velikost uživatelské základny – VCS_5 .

Jediné, co je nutné rozšířit, je popis kritéria funkce, jelikož ty byly popsány pouze konkrétně pro CI servery.

Funkce

Kritérium funkce se u VCS liší od stejného kritéria u CI serverů, proto je zde popsáno znova, na rozdíl od ostatních kritérií, která jsou u CI serverů a VCS stejná. Od testovaných VCS aplikací se samozřejmě očekává, že budou podporovat samotný Git. Mimo to by bylo vhodné, aby pomocí něj bylo možné provádět revize kódu – například když chce někdo spojit svoji větev kódu do hlavní větve („*master*“), mělo by to být možné přes tuto aplikaci. Také by mělo být možné nastavit, aby bylo spojení vlastní větve do hlavní povoleno jen v případě, že to odsouhlasí určitý počet vývojářů projektu.

Výkon

Jediné kritérium, které nebylo hodnoceno u CI serverů a bude hodnoceno zde, je výkonnost - VCS_6 . To znamená zhodnotit, jak dlouho trvá provedení typických příkazů pracujících se

vzdáleným úložištěm – jedná se tedy o příkazy jako `git fetch`, `git clone` a `git push`. Toto kritérium ale není tak významné ve srovnání s důležitějšími kritérii.

Stanovení vah kritérií

Jelikož ve srovnání s CI servery byla dvě kritéria odebrána a jedno přidáno, byly rovněž přepočítány váhy jednotlivých kritérií. Preference a výpočet váhy kritérií VCS je vidět v tabulce 7.

Tabulka 7: Váhy jednotlivých kritérií VCS (zdroj: autor)

Kritérium	Preference					Počet preferencí	Výpočet a výsledná váha kritéria
Aktivita na projektu (VCS_1)	VCS_1 VCS_2	VCS_1 VCS_3	VCS_1 VCS_4	VCS_1 VCS_5	VCS_1 VCS_6	3	$\frac{3+1}{15+6} \doteq 0,19$
Funkce (VCS_2)	$\times$	VCS_2 VCS_3	VCS_2 VCS_4	VCS_2 VCS_5	VCS_2 VCS_6	4	$\frac{4+1}{15+6} \doteq 0,24$
Licence (VCS_3)	$\times$	$\times$	VCS_3 VCS_4	VCS_3 VCS_5	VCS_3 VCS_6	2	$\frac{2+1}{15+6} \doteq 0,14$
Předpřipravený podporovaný Docker obraz (VCS_4)	$\times$	$\times$	$\times$	VCS_4 VCS_5	VCS_4 VCS_6	0	$\frac{0+1}{15+6} \doteq 0,05$
Velikost uživatelské základny (VCS_5)	$\times$	$\times$	$\times$	$\times$	VCS_5 VCS_6	5	$\frac{5+1}{15+6} \doteq 0,29$
Výkon (VCS_6)	$\times$	$\times$	$\times$	$\times$	$\times$	1	$\frac{1+1}{15+6} \doteq 0,10$
Kontrola	$\times$	$\times$	$\times$	$\times$	$\times$	$\times$	1,01

4.3.2 GitBucket

Patrně nejmenší z testovaných projektů vznikl v roce 2013 a i jako nejmenší projekt je stále aktivní, i když velikostí a počtem příspěvků se řadí k menším. Nová verze vychází nepravidelně v časovém rozmezí dvou týdnů až dvou měsíců (GitBucket, 2017a).

Funkce GitBucketu je možné vyzkoušet na veřejně přístupném demu na stránkách gitbucket.herokuapp.com. GitBucket splňuje všechny požadované z kapitoly 4.3.1, proto v tomto kritériu dostává pět bodů.

Licence „Apache License 2.0“ umožňuje téměř libovolné nakládání s projektem, včetně jeho zakomponování do vlastního, dále prodáváného softwaru (GitBucket, 2017b), proto i zde je aplikace hodnocena pěti body.

Oficiální Docker obraz GitBucket zatím nenabízí, nicméně existuje neoficiální obraz od autora, jenž vystupuje pod přezdívkou F99aq8ove (F99aq8ove, 2017). Vzhledem k tomu, že není oficiálně podporován původním vývojářským týmem, bude v tomto kritériu GitBucket ohodnocen pouhými dvěma body.

Velikost uživatelské základny je obtížné přesně změřit. Pro hodnocení v tomto kritériu bude vycházeno ze dvou anket, ve kterých hlasovali uživatelé (AlternativeTo, 2017b) a (Slant, 2017b). Ve srovnání s ostatními VCS aplikacemi je na tom GitBucket téměř nejhůř ze všech testovaných aplikací, proto zde dostává nízké hodnocení – dva body.

GitBucket ve srovnání s ostatními VCS aplikacemi sice není nejvýkonnější, nicméně vyloženě špatné výkony GitBucket nepodával. Proto je v tomto kritériu hodnocen čtyřmi body. Pro testování výkonnosti byl jako server použit Raspberry Pi 3B, kde bylo testováno, kolik paměti k běhu potřebuje, jak rychle se načítají stránky a v neposlední řadě jak rychle je možné úložiště klonovat nebo do něj nahrávat nové verze kódu (Kounoike, 2017).

4.3.3 Gogs (Gitea)

Další testovanou aplikací je Gogs (Gitea) – ve skutečnosti se jedná o dva samostatné projekty. Gitea se vyčlenila jako samostatný projekt (z angl. *fork*) poté, co původní autor projektu Gogs byl několik měsíců zcela neaktivní. Podobnému problému se chce Gitea vyhnout tím, že má více administrátorů, kteří mají práva rozhodovat o přijetí nového kódu – na rozdíl od projektu Gogs, kde je jeden. V současnosti jsou oba projekty aktivní (Caeliferum 2017), je ale možné, že v budoucnu se opět spojí do jednoho. Z hlediska funkcí a většiny dalších kritérií se jedná o téměř shodné projekty, proto budou hodnoceny společně.

Jak již bylo napsáno, aktivita v současné době na obou projektech je, nové verze vznikají podobně často jako u GitBucketu a počet přispěvatelů je zde dokonce mírně vyšší. Přesto se ale ve srovnání s GitLabem jedná o řádově menší projekt s nižším průměrným týdenním počtem příspěvků (Gogs, 2017a) a (GitLab, 2017a), proto je aplikace hodnocena čtyřmi body.

Z hlediska funkcí, které bylo možné prozkoumat na demech na <http://try.gogs.io> (respektive <http://try.gitea.io>), je vše také v pořádku, rovněž pět bodů i v tomto kritériu.

Užití poměrně svobodné licence MIT (Gogs, 2017b) a (Gitea, 2017a) se možná obrátilo proti původnímu autorovi. Umožnilo beztestně projekt okopírovat a pokračovat v něm. Nicméně z hlediska uživatelů aplikace se jedná o dobrou zprávu, jednak mohou teoreticky učinit totéž, ale hlavně mohou aplikace používat i k vývoji komerčních aplikací.

Drobnou výhodu oproti GitBucketu tyto aplikace nabízejí v tom, že pro ně existují oficiálně spravované Docker obrazy (Gogs, 2017c) a (Gitea, 2017b).

Kde si aplikace Gogs a Gitea vedou o poznání lépe než dříve testovaný GitBucket, je uživatelská základna. Ta je početnější. Na druhou stranu aplikace nejsou ani zdaleka rozšířené tak, jako je GitLab (AlternativeTo, 2017b) a (Slant, 2017b), proto jsou v tomto kritériu ohodnoceny čtyřmi body.

Výkon mají obě aplikace ještě lepší, než měl GitBucket (Kounoike, 2017), proto si jako jediné aplikace ze všech testovaných odnášejí hodnocení pět bodů.

4.3.4 GitLab

GitLab je největší a nejznámější VCS aplikací, která je zde analyzována. Od toho se odvíjí i hodnocení většiny kritérií včetně aktivity na projektu. Ta je, pokud jsou srovnávány jednotlivé příspěvky do báze kódu, průměrně za každý týden desetkrát vyšší, než je tomu u Gogs. Průměrný týdenní počet příspěvků během posledního roku u projektu Gogs byl přibližně čtrnáct (Gogs, 2017a), zatímco u Gitlabu tři sta dvacet (GitLab, 2017a).

Splňuje také všechny základní funkce definované v kapitole 4.3.1. Co ale nabízí GitLab navíc, jsou základní funkce CI (GitLab, 2017d). Užití Gitlabu jako CI serveru a porovnání s ostatními CI servery je popsáno v kapitole 4.2.4. Zde jsou také rozebrána kritéria společná pro CI i VCS server – licence, existence oficiálního Docker obrazu a velikost uživatelské základny. Ve všech zmíněných je GitLab hodnocen pěti body.

Jediné, kde GitLab zaostává za ostatními aplikacemi, je srovnání z hlediska výkonu. Jednak mu déle trvá zpracovat git příkazy (například `git push` trvá 75 milisekund oproti 30 milisekundám u ostatních aplikací), i tak ve výsledku je tento rozdíl zanedbatelný a běžným uživatelem těžko zpozorovatelný. Hlavním problémem ale je, že GitLab zabírá daleko více operační paměti (800 megabajtů oproti 200–300 megabajtům u ostatních testovaných aplikací). To v důsledku znamená, že pokud by celé komplexní CI řešení, jež je výsledkem této práce, mělo používat GitLab jako VCS server, tak by nemohlo fungovat na menším serveru – například na Raspberry PI. Proto v tomto kritériu získává GitLab pouze dva body.

4.3.5 RhodeCode

RhodeCode je nejmenší z testovaných aplikací. V žebříčku užití a popularity, z nichž bylo při výběru aplikací vycházeno (AlternativeTo, 2017b) a (Slant, 2017b), existují sice větší a

šířeji užívané projekty (GitHub, BitBucket, Phabricator, SourceForge, ...), nicméně ty nelze volně a zdarma užívat k privátním komerčním projektům, případně provozovat na vlastním serveru. RhodeCode byl vybrán jako největší z malých projektů pro reprezentativnost výsledků srovnání.

Jelikož zdrojový kód RhodeCode není na GitHubu a ani tam není zrcadlen, nelze přesně a přímo srovnat, jak moc je projekt ještě aktivní. Nicméně z poznámek o nových verzích (RhodeCode, 2017) lze zjistit, že nová verze vychází průměrně jednou za měsíc a vždy má zajímavé nové funkce, tak nelze označit projekt za mrtvý. Nicméně počet příspěvků do báze kódu není nijak závratný, proto si RhodeCode odnáší jen čtyři body.

Funkce RhodeCode pokrývá všechny potřebné, dokonce bez potřeby instalace dalších doplňků podporuje nejen Git, ale i Subversion, proto si v tomto kritériu odnáší pět bodů.

Licence u verze zdarma je AGPLv3, což je dostačující pro účely potřebné v této práci, proto si odnáší pět bodů i zde.

Podpora už je horší u Dockeru, zatím neexistuje oficiálně podporovaný Docker obraz, na druhou stranu existují obrazy od jiných vývojářů – např. (Codingtony, 2015). Díky tomu neodchází RhodeCode s jedním bodem, ale se dvěma.

Velikost uživatelské základny je ve srovnání s ostatními aplikacemi podobná jako u GitBucketu. Proto zde dostává stejně bodů jako on – dva.

Poslední kategorii – výkon – zde nelze přesně ohodnotit, jelikož ve srovnání, které bylo použito u ostatních, tato aplikace nebyla. Nebyla rovněž zveřejněna natolik konkrétní podoba úloh, které byly testovány, aby test mohl být reprodukován s jinou aplikací. Jediné, co mohlo být porovnáno, byly požadavky na paměť – RhodeCode pro chod požaduje minimálně 512 megabajtů paměti (Kuzminski, 2014), což jej staví někam mezi GitLab (800 MB) a ostatní (200-300 MB). I přes to, že ostatní parametry testovány a porovnány nebyly, tak s ohledem na paměťovou náročnost může hodnocení tři body (které je určeno pro hodnocení kritérií, pro něž z různých důvodů není dostatek dat, čili přesně jako v tomto případě) odpovídat realitě poměrně přesně.

4.3.6 Hodnocení

Celkové hodnocení testovaných aplikací typu VCS je vidět v tabulce 8.

Tabulka 8: Hodnocení VCS (zdroj: autor)

Kritérium	Váha	GitBucket	Gogs (Gitea)	GitLab	RhodeCode
Aktivita na projektu (VCS_1)	0,19	4	4	5	4
Funkce (VCS_2)	0,24	5	5	5	5
Licence (VCS_3)	0,14	5	5	5	5
Předpřipravený podporovaný Docker obraz (VCS_4)	0,05	2	5	5	2
Velikost uživatelské základny (VCS_5)	0,29	2	4	5	2
Výkon (VCS_6)	0,10	4	5	2	3
Součet hodnocení	$\times$	22	28	27	21
Celkové skóre po započtení vah	$\times$	3,74	4,57	4,75	3,64

Z tohoto hodnocení vychází jako nejlepší VCS GitLab. Jedná se ale o velmi těsný výsledek, těsně za GitLabem se umístila dvojice Gogs(Gitea). GitLab má o něco lepší aktivitu na projektu a širší uživatelskou základnu, naopak Gogs(Gitea) vyšly ve srovnání značně lépe, co se výkonu týče. Pokud by bylo cílem této práce vytvořit řešení, které bude bez problémů fungovat i na velmi slabých strojích, byla by pravděpodobně váha kritéria výkon daleko větší. Takto ale např. 800 MB RAM, které pro svůj běh potřebuje GitLab, není limitující, a tak bude součástí praktického řešení právě ten.

4.4 Nástroje pro analýzu kódu

Pojem analýza kódu spadá do fáze „inspekce“ procesu CI. Tato fáze byla popsána v kapitole 3.3.6. S tím související pojmy – především analýza příkazů a řádků programu (angl. *statement coverage*), úplná analýza větvení programu (angl. *branch coverage*), ale také cyklomatická složitost a další – byly rozebrány v kapitolách 3.4.2 a 3.4.3. V této kapitole budou porovnány nástroje, které jsou schopné tyto metriky automatizovaně měřit a zobrazovat. Následně bude za užití metody popsané v kapitole 4.1 vybrán jeden z nich, který bude součástí CI řešení, jež bude praktickým výstupem z této práce. Pro přehlednější zápis bude pro tyto nástroje různých kategorií používána zkratka SCA – z angl. *static code analysis*.

4.4.1 Kritéria a stanovení vah

Při volbě kritérií je zde odlišná situace, než byla v případech CI a VCS serverů. V obou případech se jednalo o komplexní nástroje, jejichž instalace, konfigurace, a především dlouhodobá správa může vyžadovat daleko větší nároky. Proto zde byl kladen důraz především na širokou uživatelskou základnu a v menší míře také na možnost placené podpory. Ve srovnání s tím jsou nástroje pro analýzu kódu jednodušší aplikace, proto zde bude kladen důraz na podporované funkce, které se mohou značně lišit – na rozdíl od CI a VCS serverů, kde základní funkcionalitu splňovaly všechny aplikace. Naopak velikost uživatelské základny zde není rozhodující, jelikož jednak zde kvůli nižší komplexnosti řešení není nutno řešit tolik nastavitelných problémů a jednak výstupem běhu těchto nástrojů je obvykle několik čísel – ukazatelů metrik, a tak zde není nutné pro uživatele se s nástrojem dlouze učit pracovat.

Funkce

V tomto kritériu bude porovnáváno, zda nástroj umí měřit pokrytí kódu testy, nejen *statement coverage* ale i *branch coverage*. Další funkcí, kterou by nástroj uměl rozpoznat, je kopírování stejného kódu na různých místech v projektu, což je známkou technického dluhu. S tím souvisí i měření cyklomatiky složitosti kódu a upozornění na metody, které stanovenou složitost přesahují. S kvalitním kódem souvisí i hledání a upozorňování na již probrané „*pachy v kódu*“ či upozorňování na možné podivné až vyloženě nevhodné praktiky. Např. pokud má metoda příliš parametrů nebo pokud je *catch* blok prázdný nebo obsahuje jen komentář `//TODO`. Dalším atributem, který by měl být měřen, je dodržování definovaných kódovacích standardů – kontrola toho, zda jsou otevírací závorky na konci řádku nebo na samostatném řádku, či kolika mezerami jsou odsazeny zanořené bloky kódu.

Aktivita na projektu

Rovněž toto kritérium bylo popsáno v předchozích kapitolách věnovaných jiným aplikacím, přesto je vhodné zdůraznit, proč je i zde důležité z hlediska nástrojů pro analýzu kódu. Je tomu tak kvůli jedné funkci – hledání nevhodných konstrukcí v kódu. Java se stejně jako každý jiný užívaný programovací jazyk stále vyvíjí. Např. v Javě 8 přibily lambdy (Konda, 2014) a bylo by nedostatečné, kdyby s nimi nástroj neuměl pracovat a odhalit v nich žádné chyby z toho důvodu, že projekt je již mrtvý a nikdo do něj podporu pro lambdy nezapracoval. Podobné převratné změny mohou nastávat i v budoucnu, proto je vhodné zvolit projekt, na kterém se stále pracuje.

Integrace s Jenkins

Jelikož v kategorii CI serverů byl vybrán Jenkins, je vhodné, aby s ním ostatní aplikace byly schopny pracovat. Tento požadavek odráží toto kritérium. Forma integrace může být různá – od podpory bez nutné instalace dalších doplňků, přes doplňky, které umožňují např.

komunikovat s nástrojem, který běží na jiném serveru, po doplňky, které přímo obsahují nástroje z kategorie SCA.

Licence

Toto kritérium není třeba podrobněji rozepisovat, jde zde o totéž jako v případě CI a VCS serverů – licence musí dovolovat používat nástroj zdarma, na vlastním serveru, a to i pro komerční účely.

Možnosti kastomizace

Toto kritérium odráží to, zda a jak moc je možné nástroj kastomizovat a dále konfigurovat, jelikož například kódovací standardy může mít každá firma jiné, je vhodné, aby jejich kontrola mohla být upravena.

Stanovení vah kritérií

Jelikož jde o zcela odlišené aplikace od CI a VCS, byly stanoveny nové váhy kritérií na základě dříve zmíněných důvodů. Preference a výpočet váhy kritérií nástrojů pro statickou analýzu kódu jsou vidět v tabulce 9. Kritéria zde jsou označena zkratkami SCA_1 – SCA_5 , přičemž čísla odpovídají pořadí, v jakém jsou jmenovány v předchozích kapitolách. SCA_1 označuje funkce a SCA_5 označuje možnosti kastomizace.

Tabulka 9: Váhy jednotlivých kritérií nástrojů pro analýzu kódu (zdroj: autor)

Kritérium	Preference				Počet preferencí	Výpočet a výsledná váha kritéria
Funkce (SCA_1)	SCA_1 SCA_2	SCA_1 SCA_3	SCA_1 SCA_4	SCA_1 SCA_5	3	$\frac{3 + 1}{10 + 5} \doteq 0,27$
Aktivita na projektu (SCA_2)	X	SCA_2 SCA_3	SCA_2 SCA_4	SCA_2 SCA_5	2	$\frac{2 + 1}{10 + 5} \doteq 0,20$
Integrace s Jenkins (SCA_3)	X	X	SCA_3 SCA_4	SCA_3 SCA_5	4	$\frac{4 + 1}{10 + 5} \doteq 0,33$
Licence (SCA_4)	X	X	X	SCA_4 SCA_5	0	$\frac{0 + 1}{10 + 5} \doteq 0,07$
Kastomizace (SCA_5)	X	X	X	X	1	$\frac{1 + 1}{10 + 5} \doteq 0,13$
Kontrola	X	X	X	X	X	1,00

4.4.2 Cobertura

Cobertura je nástroj, který ani zdaleka nepokrývá všechny požadované funkce. Jediné, co umí, a to spolehlivě, je měření pokrytí kódu testy a počítání cyklomatické složitosti. Proto dostává z tohoto důležitého kritéria jen dva body (Cobertura, nedatováno).

I přes svoji dřívější rozšířenost je to s aktivitou na projektu v současné době velmi špatné, za poslední rok nebylo do zdrojového kódu ani jedenkrát přispěno (Cobertura, 2017a), proto pouhý jediný bod v tomto kritériu.

I přes to, že se jedná o již nevyvíjený projekt, tak existuje hojně stahovaný doplněk pro Jenkins s názvem Cobertura Plugin, za poslední měsíc si jej stáhlo více než osmnáct tisíc uživatelů (Jenkins, 2017d). Proto zde dostává Cobertura plný počet bodů.

I v oblasti licence získává Cobertura pět bodů, a to proto, že je šířena pod svobodnou licencí GNU GPL v.2.0 (Cobertura, 2017b).

Možnosti kastomizace je obtížné posoudit, jelikož z hlediska omezenosti funkcí ani nemá smysl nástroj nějak upravovat dle přání uživatele (pokrytí kódu testy i cyklomatická složitost další nastavení nepotřebují), a tak to nástroj ani nenabízí. Na jednu stranu to nástroj nenabízí, ale na druhou stranu má omezené funkce, které to vlastně ani nevyžadují. Dostává tedy v kritériu kastomizace neutrální tři body.

4.4.3 Codacy

Codacy je jedním z větších nástrojů, které zde budou testovány. Z hlediska funkcí, na rozdíl od většiny ostatních aplikací, podporuje všechny požadované – od kontroly kódovacích standardů, přes počítání metrik pokrytí kódu testy, vyhledávání duplicitního kódu, po hledání špatných praktik a „*pachů v kódu*“ (Codacy, 2017a). Proto si z kritéria funkce odnáší pět bodů.

Codacy nemá otevřené zdrojové kódy, tak není možné přesně určit, na kolik je projekt aktivní. Dokumentace se ale zdá být aktuální a oficiální internetové stránky dobře udržované, proto dostává v kritériu aktivita na projektu také pět bodů.

Nástroj nabízí integraci s Jenkinsem, což je vidět z nedávného příspěvku v dokumentaci (Codacy, 2017b). Aktuální dokumentace jen potvrzuje pět bodů z aktivity na projektu, které Codacy dostalo v předchozím kritériu.

Kdyby nebylo kritéria licence, Codacy by možná mohlo i vyhrát toto porovnání a stát se součástí praktického řešení práce. Ale vzhledem k tomu, že nástroj je možné zdarma používat jen formou SaaS, a to jen na projekty s otevřeným zdrojovým kódem (pro komerční účely a instalaci na vlastní server je nutno platit) (Codacy, 2017c), dostává Codacy z kritéria licence jen jeden bod.

V posledním kritériu ale Codacy opět obstálo výborně – definovaná pravidla je možné dále upravovat nebo i vypínat Codacy, 2017d), proto Codacy dostává pět bodů.

4.4.4 FindBugs

FindBugs je dalším z nástrojů pro analýzu kódu, jenž nesplňuje všechny požadované funkce. Z názvu vyplývá, že se nezabývá počítáním metrik pokrytí kódu testy nebo upozorňováním na odchýlení se od kódovacích standardů, ale soustředí se na odhalování potenciálních chyb. Bohužel, toto je jeho jediná funkce (FindBugs, 2015), a tak si odnáší jen dva body.

S aktivitou na projektu je to také horší i přes to, že dříve byl projekt velmi rozšířen. Projekt se zdá být efektivně mrtvý, za poslední rok byly v úložišti kódu jen čtyři nové příspěvky (FindBugs, 2017), a proto si zde odnáší jen jeden bod.

Kde FindBugs konečně získává pět bodů, je integrace s Jenkinsem, existuje totiž doplněk FindBugs Plugin, který společně s doplňkem s názvem „Static Analysis Collector plug-in“ dokáže zobrazovat nalezené chyby (Jenkins, 2017e).

Licence, pod kterou je nástroj šířen, je GNU LGPL v.3 (FindBugs, 2015), proto i zde nástroj dostává pět bodů.

Pět bodů FindBugs dostává i díky tomu, že je možné psát do něj vlastní pravidla – kustomizovat jej dle potřeb projektu, i když zde to asi nemá takový smysl jako u kódovacích konvencí, které mohou být v každé firmě jiné. Pokud někdo napíše kontrolu na odhalení chyb, patrně by si ji neměl nechávat pro sebe a měl by učinit příspěvek do oficiálního úložiště.

4.4.5 JaCoCo

JaCoCo, jak už název napovídá, je nástroj pro měření pokrytí Javového kódu testy. JaCoCo dle slov autorů vznikl proto, že ostatní nástroje pro měření pokrytí kódu testy nejsou z různých důvodů dostatečné pro integraci – většinou vznikly jako doplněk určitého konkrétního nástroje a neposkytují dostatečně zdokumentované API. Dva nejrozšířenější nástroje (EMMA a Cobertura) jsou zase již nevyvíjené (JaCoCo, 2017a). Bohužel, měření pokrytí kódu testy společně s počítáním cyklomatické složitosti jsou jeho jediné funkce, a tak si z hodnocení v tomto kritériu odnáší jen dva body.

Z kritéria aktivita na projektu dostává čtyři body, projekt je sice stále aktivní, ale práce na něm není nijak vysoká. Za poslední rok na něm byly průměrně jeden až dva příspěvky týdně (JaCoCo, 2017b).

JaCoCo je kompatibilní s Jenkinsem pomocí doplňku JaCoCo plugin, který si za poslední měsíc stáhlo více než 1000 uživatelů Jenkinsu (Jenkins, 2017f), proto v tomto kritériu dostává pět bodů.

Licenci má JaCoCo také otevřenou – Apache 2.0, a tak si z tohoto kritéria odnáší pět bodů.

V kritériu možnostech kastomizace je na tom JaCoCo stejně jako Cobertura v kapitole 4.4.2 – možnosti kastomizace JaCoCo jsou nulové, ale zase opět vzhledem k omezenosti funkcionality to ani není třeba vyžadovat – proto si JaCoCo odnáší tři body stejně jako Cobertura.

4.4.6 JavaNCSS

JavaNCSS je dalším z téměř jednoúčelových statických analyzátorů kódu, které zde jsou porovnávány. Tento nástroj se soustředí především na měření různých metrik – např. na počet nekomentujících řádků kódu, od toho vznikl i název NCSS – z angl. „*non comenting source statements*“. Další vlastností, kterou JavaNCSS měří, je cyklomatická složitost, dále celkový počet tříd, metod, balíčků atd. I přes to si ale z tohoto kritéria odnáší jen dva body, protože většinu požadovaných funkcí nástroj neumí (JavaNCSS, 2017a).

Z počtu příspěvků na GitHubu za poslední rok (0) to vypadá, že projekt již nikdo nevyvíjí (JavaNCSS, 2017b), tak nástroj dostává jeden bod. I přes to, že projekt není vyvíjen, existuje funkční doplněk do Jenkinse s názvem JavaNCSS Plugin. Ovšem není ani zdaleka tak často stahován jako ostatní zde testované nástroje. Za poslední měsíc byl stažen pouze pětsetkrát (Jenkins, 2017g), i tak je ale JavaNCSS v kritériu propojení s Jenkins hodnocen pěti body.

Pět bodů dostává JavaNCSS i za užití svobodné licence GNU GPL (JavaNCSS, 2017a), která umožňuje používat jej i pro komerční účely.

Z kritéria kastomizace si odnáší jen tři body z důvodů stejných, jako tomu bylo u Cobaturity a JaCoCo – podrobnější úpravy nástroj sice nenabízí, ale vzhledem k omezeným funkcím to ani není nutné.

4.4.7 PMD

PMD je jedním z nejznámějších statických analyzátorů kódu nejen pro Javu, ale i pro další jazyky (C, PHP, Groovy, Scala, Python, Go, ...). Dokáže odhalit běžné špatné praktiky při psaní nebo i zjevné chyby (mj. např. definované, ale neužívané proměnné). Obsahuje též něco, co autoři označují jako CPD – „*copy paste detector*“ – detektor kopírovaného kódu napříč projektem. Jednou z věcí, které PMD také měří, je cyklomatická složitost (PMD, 2017a). Tyto funkce zvládá PMD výborně, k čemu ho ale použít nelze, je měření pokrytí kódu testy nebo kontrola kódovacích standardů. V praktickém řešení by musel být

kombinován s dalšími nástroji, které umí např. jen tyto funkce, proto v kritériu funkce dostává jen dva body.

Za počet příspěvků během posledního roku na GitHubu (PMD, 2017b), který je roven přibližně dvaceti příspěvkům týdně, si PMD odnáší pět bodů. Pět bodů si PMD odnáší ze všech dalších kritérií. Je možné jej integrovat s Jenkinsem pomocí doplňku PMD plugin (Jenkins, 2017h). Licence Apache 2.0, pod kterou je PMD šířeno (PMD, 2017c), je jedna z nejsvobodnějších licencí vůbec. Pět bodů dostává PMD i za to, že je možné jej upravovat a vytvářet vlastní pravidla, která má kontrolovat (PMD, 2017d).

4.4.8 SonarQube

SonarQube je jedním z nástrojů, který splňuje téměř všechny vyžadované funkce. Umí zobrazovat pokrytí kódu testy, hledat kopírovaný kód napříč projektem, počítat i cyklomatickou složitost. Potřebuje k tomu ale podklady ve formě XML, například z JaCoCo. Tyto výstupy pak integruje a zobrazuje spolu s ostatními výsledky analýz, které počítá už sám. SonarQube umí dohlížet na dodržování kódovacích standardů nebo i odhalovat špatné praktiky či přímo zjevné chyby v kódu. Nad rámec požadovaných definovaných funkcí nabízí i hledání základních bezpečnostních chyb (SonarQube, 2017a).

Aktivitu na projektu má SonarQube velmi vysokou, za poslední rok počet příspěvků za týden neklesl pod osmáct, většinou se ale držel kolem padesáti (SonarQube, 2017b). To znamená pět bodů pro SonarQube v kritériu aktivita na projektu.

Integraci SonarQube s Jenkinsem popisuje SonarQube v dokumentaci (SonarQube, 2017c), dle ní je SonarQube možné s Jenkinsem používat již od verze 1.491. I zde SonarQube dostává pět bodů.

Pět bodů dostává SonarQube i za kritérium licence, je totiž šířen pod svobodnou licenci GNU GPLv3 (SonarQube, 2017d).

Ani v posledním kritériu – možnosti kustomizace – SonarQube nedostává méně než pět bodů, jelikož je možné výchozí nastavení a pravidla upravovat, vypínat nebo přidávat vlastní (SonarQube, 2017e).

4.4.9 Hodnocení

To, jak si jednotlivé nástroje vedly při porovnání, je patrné z tabulky 10.

Tabulka 10: Hodnocení SCA nástrojů (zdroj: autor)

Kritérium	Váha	Cobertura	Codacy	FB	JaCoCo	JavaNCSS	PMD	SQ
Funkce (SCA₁)	0,27	2	5	2	2	2	2	4
Aktivita na projektu (SCA₂)	0,20	1	5	2	4	1	5	5
Integrace s Jenkins (SCA₃)	0,33	5	5	5	5	5	5	5
Licence (SCA₄)	0,07	5	1	5	5	5	5	5
Kastomizace (SCA₅)	0,13	3	5	5	3	3	5	5
Součet hodnocení	X	16	21	19	19	16	22	24
Celkové skóre po započtení vah	X	3,13	4,72	3,59	3,73	3,13	4,19	4,73

Celkově se dají testované nástroje pro analýzu rozdělit do několika kategorií. První z nich jsou ty, které počítají metriky pokrytí kódu testy a cyklomatickou složitost – sem patří Cobertura a JaCoCo. Na hraně těchto nástrojů je i nástroj JavaNCSS, který počítá cyklomatickou složitost a další, možná ne až tak důležité metriky – velikost tříd, balíčků atd. Další skupina nástrojů – PMD a FindBugs – se soustředí na odhalování různých vad v programu – především na špatné praktiky, jako jsou např. nekomentované veřejné metody, duplikování kódu na různých místech atd. Poslední skupinou jsou nástroje, které vše vyjmenované kombinují a umí i něco navíc, např. kontrolu kódovacích standardů dodržovaných v týmu – z testovaných nástrojů to je Codacy a SonarQube.

Pro praktickou část bude použit SonarQube v kombinaci s JaCoCo. SonarQube je téměř ve všech kritériích hodnocen maximálním možným počtem bodů. S podporou JaCoCa splňuje všechny potřebné funkce, je kastomizovatelný. Projekt je aktivní, je šířen pod správnou licenci a dokáže pracovat s Jenkinsem.

Projekt, který je těsně druhý v pořadí – Codacy – nesplňuje základní požadavek, a to aby byl volně užitelný i na komerční projekty.

5 Praktické řešení

V této kapitole je popsána praktická část diplomové práce – dockerizované CI řešení pro Java projekty. Kapitola je rozdělena do šesti částí.

V první kapitole je stručně popsána technologie Docker, v dalších třech kapitolách jsou popsány jednotlivé vytvořené Docker obrazy – Jenkins, Gitlab a SonarQube. Ve všech třech případech sice existovaly oficiální obrazy, nicméně bylo nutné je dále doplnit – např. doinstalovat doplňky do Jenkins, nastavit klíče, hesla a další podobné konfigurace. Důraz při vytváření obrazů byl totiž kladen především na to, aby celá instalace probíhala zcela bezobslužně a aby i přesto byly aplikace připraveny k provozu bez jakýchkoliv dalších manuálních kroků. Daní za tento přístup je však to, že jednotlivé Docker obrazy jsou relativně úzce provázané - např. v Jenkinsu jsou uloženy přihlašovací údaje do GitLabu a jsou zde nainstalovány doplňky pro integraci s GitLabem. Toto by při užití samostatného Jenkins obrazu bylo zbytečné – například pokud by se někdo rozhodl toto řešení upravit a použít místo GitLabu jiný systém správy verzí.

Čtvrtá kapitola je věnována instalaci, její případné kustomizaci a další údržbě aplikace z pohledu administrátora. Je zde uvedeno jednak jak aplikaci restartovat či bezpečně vypínat, ale jsou zde také popsána některá úskalí týkající se Dockeru, s nimiž jsem se při vývoji řešení setkal.

Poslední, pátá, kapitola je věnována užívání aplikací z pohledu vývojáře. Už z instalace jsou nastaveny a provázány tzv. *Jenkins joby* s *GitLab projektem*, do něhož stačí pomocí Gitu do větve nahrát kód (též součástí přílohy) a zažádat spojení kódu do hlavní větve. Tím se automaticky spustí různé *Jenkins joby*, které např. vkládají komentáře přímo do žádosti o spojení s upozorněním na nedodržování kódovacích standardů.

5.1 Technologie Docker

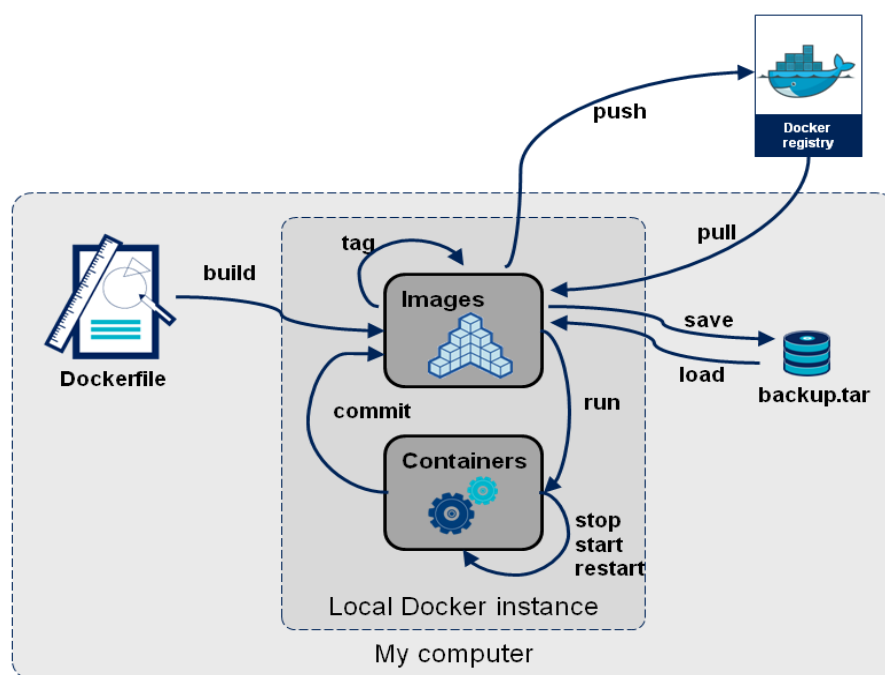
Než bude popsáno konkrétní řešení, považuji za vhodné zde v několika odstavcích představit technologii Docker. Při tomto popisu bude kladen důraz především na znalosti důležité pro pochopení praktické části. Není zde kladen důraz na to, jak přesně Docker funguje vnitřně. Pro podrobnější teoretický rozbor odkáži čtenáře na rešerše v kapitole 2 – konkrétně na absolventské práce (Jurenka, 2015) a (Tomášek, 2015).

Pro člověka, který o Dockeru vůbec neslyšel, bude nejjednodušší si jej představit jako něco podobného, jako jsou virtuální stroje – např. VirtualBox či VMware. S nimi Docker sdílí mnoho výhod – jednou z největších je pravděpodobně přenositelnost. Je relativně snadné předpřipravit obraz a nainstalovat do něj set aplikací. Když se tento obraz pak spustí na libovolném počítači, bude se chovat stejně, jelikož součástí tohoto obrazu je i instalace OS

se všemi knihovnami. Jednou z nevýhod pak je to, že výsledný obraz je relativně velký – vždy obsahuje kopii OS plus doinstalované aplikace – na virtuálních strojích tak běží dva OS najednou – hostitelský a virtualizovaný. Oproti tomu při využití Dockeru dochází ke sdílení zdrojů (např. knihoven) s hostitelským OS. Z toho tedy také plyne, že zde je o jednu úroveň abstrakce méně – aplikace běžící v Dockeru komunikují s hardwarem přímo, tedy stejným způsobem jako aplikace běžící na hostitelském stroji. V tom se liší od technologií typu VirtualBox či VMware, kde jsou ve virtuálním stroji ovladače k hardwaru emulovány (Seshachala 2014) a (Docker 2017b).

Další věc, kterou je nutné znát pro pochopení praktické části, je rozdíl mezi Docker obrazem a Docker kontejnerem, protože v další části bude hovořeno v jednu chvíli o obrazech, pak zase o kontejnerech, tak je důležité mít v těchto pojmech jasno. Dle oficiálního Docker glosáře je definice kontejneru takováto: „A container is a runtime instance of a docker image.“ (Docker, 2017c). Možná pro vývojáře bude názornější srovnání se světem objektově orientovaného programování. Je totiž možné představit si Docker obraz jako třídu a Docker kontejner jako její instanci (Cook, 2017). Může tedy existovat a současně běžet více kontejnerů toho stejného obrazu. Naopak je nesmyslné hovořit o „běžícím obrazu“.

S Docker obrazy a kontejnery souvisí také jejich životní cyklus, který je ve zjednodušené formě zobrazen na obrázku 6.



Obrázek 6:

Životní cykly Docker obrazů a kontejnerů (zdroj: Mazin 2014)

Začněme tedy obrazy. Docker obraz se musí sestavit (na obrázku 6 naznačeno příkazem **build** resp. **docker build**). Tento příkaz podle souboru Dockerfile vytvoří Docker obraz na lokálním počítači. Poté je možné jej příkazem **docker push** nahrát do úložiště Docker obrazů, z něhož je možné dostat jej zpět příkazem **docker pull**. To se využívá např. pro sdílení obrazů s komunitou nebo s kolegy. Pokud už tedy máme na lokálním počítači Docker obraz

(a je jedno, zda sestaven nebo stažen), můžeme začít pouštět kontejnery. K tomu slouží příkaz `docker run`. Pokud je potřeba kontejner předčasně ukončit či restartovat, můžeme tak učinit příkazem `docker stop`, resp. `docker restart`. Zde je potřeba zmínit jednu důležitou věc a to tu, kdy je kontejner za normálních podmínek (nikdo nezavolá příkaz podobný `docker stop`) ukončen. V kontejneru je vždy hlavní proces – ten se dá nastavit např. v Dockerfile pomocí příkazů `CMD` nebo `ENTRYPOINT`. Samotný kontejner pak běží tak dlouho, jak dlouho běží tento proces. Poté, co je kontejner ukončen, je jeho vnitřní stav za běžných podmínek ztracen. Pokud tedy chceme nějakým způsobem uchovávat stav napříč běhy kontejnerů, musíme pro to něco explicitně udělat. Asi nejjednodušší je užití parametru `-v` příkazu `docker run`, který se stará o to, že kontejner a hostitelský operační systém sdílí specifikované složky. Pokud si např. kontejner ukládá logy do adresáře `/var/logs`, a tento je pak napojen pomocí parametru `-v` např. na hostitelský adresář `/tmp/docker/container_a_logs`, je stav adresáře uchován pro další užití. Při ukončení a opětovném spuštění kontejneru pak nový kontejner má přístup k tomu, co tam předchozí zanechal. Pokud by parametr `-v` volán nebyl, byl by na začátku adresář `/var/logs` vždy prázdný (Docker, 2017a).

Poslední věc, která zde bude zmíněna, je užití příkazu `docker-compose` místo příkazu `docker`. Samotný příkaz `docker` je vhodný např. pro vývoj, kdy člověk pracuje jen s jedním kontejnerem. Pokud se ale snaží najednou spustit více kontejnerů, případně je nějak provázat, je snazší užívat `docker-compose`. Normálně by totiž člověk musel volat příkaz `docker run` (s dalšími parametry např. pro sdílení složek, odhalování portů atp.) pro každý běžící kontejner. Příkaz `docker-compose` však bere informace a dané parametry ze souboru `docker-compose.yml`, kde jsou i detaily o tom, jak získat obrazy pro jednotlivé kontejnery (zda např. na lokálním disku hledat soubory Dockerfile, a ty sestavit pomocí `docker build`, nebo zda je stahovat z úložiště obrazů příkazem `docker pull`) (Docker, 2017a).

5.2 Jenkins Docker obraz

V této kapitole bude popsán obraz pro Jenkins, jenž sestává celkem z devíti souborů, viz výpis 8.

Výpis 8: Seznam souborů pro vytvoření Jenkins Docker obrazu (zdroj: autor)

```
1 Dockerfile
2 init_groovy_scripts/001-create-user.groovy
3 init_groovy_scripts/002-add-key.groovy
4 init_groovy_scripts/003-secure-jenkins.groovy
5 init_groovy_scripts/004-install-maven.groovy
6 init_groovy_scripts/005-set-timezone.groovy
7 init_groovy_scripts/006-create-jobs.groovy
8 jenkins_at_gitlab.key
9 plugins.txt
```

Nejdůležitějším bodem pro vytvoření obrazu je v Dockeru vždy soubor s názvem Dockerfile – je v něm popsáno, od jakého jiného obrazu dědí a co se má při vytváření obrazu dít – např. stáhnout nějaký soubor z internetu, nakopírovat soubor, spustit službu atd. Díky tomu, že pro Jenkins existuje oficiálně podporovaný Docker obraz, od kterého bylo možné dědit, je obsah vlastního souboru Dockerfile velmi skromný, viz výpis 9.

Výpis 9: *Dockerfile pro Jenkins Docker obraz (zdroj: autor)*

```
1 FROM jenkins:2.60.2
2
3 ADD jenkins_at_gitlab.key /usr/share/jenkins/ref/jenkins_at_gitlab.key
4 ADD plugins.txt /usr/share/jenkins/ref/
5 RUN /usr/local/bin/plugins.sh /usr/share/jenkins/ref/plugins.txt
6 ENV JAVA_OPTS="-Dhudson.Main.development=true -Djenkins.install.runSetupWizard=false"
7 ADD init_groovy_scripts /usr/share/jenkins/ref/init.groovy.d/
```

Na prvním řádku je napsáno, že tento obraz dědí od oficiálního obrazu s označením „jenkins:2.60.2“.

Na třetím a čtvrtém řádku jsou příkazy pro nakopírování lokálních souborů `jenkins_at_gitlab.key` s privátním RSA klíčem a `plugins.txt` (soubory jsou viditelné na výpisu 8) dovnitř do Dockeru. Obsah těchto souborů bude popsán dále.

Řádek pět spouští skript `/usr/local/bin/plugins.sh` uvnitř Dockeru, který tam byl vytvořen předkem tohoto obrazu. Jako parametr bere tento skript cestu k souboru, kde jsou specifikované doplňky pro Jenkins (tento tam byl nakopírován předchozím krokem), které pak stáhne z internetu a nainstaluje.

Zde je vhodné popsat situaci kolem instalace doplňků do Jenkinse. V souboru `plugins.txt` jsou na každém řádku samostatně specifikovány doplňky, které se mají instalovat, a to ve formátu *doplňek:verze*, např. tedy „git:3.5.1“. Problém skriptu, který s tímto souborem pracuje, je to, že neinstaluje doplňky, na kterých je tento doplněk závislý. Např. doplněk git ve verzi 3.5.1 závisí na dalších sedmi doplncích, z nichž většina závisí na dalších doplncích. Tyto všechny závislosti bylo nutné rekurzivně vyřešit a otestovat, že nic nechybí. Drobný problém nastal, když různé balíčky závisely na jiné verzi téhož balíčku, jelikož není možné mít jeden balíček nainstalovaný ve více verzích. Tento problém jsem vyřešil instalací novější verze daného balíčku a otestováním, že doplněk, který vyžadoval starší verzi, funguje i na verzi nové.

Řádek šest specifikuje proměnnou prostředí `JAVA_OPTS`. Když je aplikace Jenkins spouštěna, kontroluje tuto proměnnou a upravuje své chování podle zde specifikovaných proměnných. Zde jsou specifikovány dvě proměnné – `hudson.Main.development` a `jenkins.install.runSetupWizard`. Obojí je nutné nastavit kvůli tomu, aby Jenkins při prvním spuštění nepouštěl manuálního průvodce instalací – což je výchozí nastavení. Ve výchozím nastavení se totiž při prvním přístupu z prohlížeče na URL Jenkinsu spustí několikakroková manuální instalace, kde je nutné ručně nastavovat, jaké uživatele je třeba vytvořit, jaké doplňky doinstalovat apod. S tím je možná méně práce a po chvíli cviku to zvládne každý,

nicméně celý proces vytváření a nastavování Jenkinse by pak nebyl automatizovaný. Samozřejmě by bylo možné popsat všechny manuální kroky, které je třeba vykonat – na funkcionalitu by to nemělo žádný vliv. Nejednalo by se pak ale už o bezstarostnou bezobslužnou automatickou instalaci, která v horším případě zabere pět minut, naopak manuální kroky by instalaci prodloužily a celý proces by byl daleko křehčí – administrátor by např. omylem opomněl doinstalovat jeden doplněk, a vše by přestalo fungovat.

Řádek sedm kopíruje obsah složky `init_groovy_scripts` dovnitř do Dockeru do složky `init.groovy.d`. Jakékoliv Groovy skripty nakopírované do této složky Jenkins vykoná vždy při startu aplikace. Je tedy možné Jenkins tímto způsobem dále konfigurovat – např. nastavovat uživatele, práva, vypínat či zapínat různá nastavení. Všechna tato nastavení je možné upravovat manuálně – přes prohlížeč, nicméně takto je vše automatizované.

Snažil jsem se pojmenovat všechny soubory tak, aby z jejich názvu bylo patrné, co dělají. Aniž bych příliš zabíhal do detailů, stručně popíšu, co jednotlivé skripty dělají.

Soubor `001-create-user.groovy` vytváří uživatele s administrátorskými právy se jménem *root* a s heslem *jenkinspassword*. Jméno a heslo samozřejmě nejsou napsána ve skriptu natvrdo a lze je jednoduše upravit, tato možnost je podrobněji popsána v kapitole 5.6. Ve druhém skriptu je do tzv. „*Jenkins credentials*“ přidán privátní RSA klíč, který byl do obrazu nakopírován příkazem na třetím řádku souboru Dockerfile. Díky tomuto pak bude moci být Jenkins autentizován, když bude přistupovat ke GitLabu. Třetí skript upravuje různá bezpečnostní nastavení. Po instalaci totiž Jenkins zobrazoval nemalý počet hlášek, které informovaly o různých bezpečnostních problémech, které by mohl kdokoliv zneužít. Tento skript tyto problémy s bezpečností řeší. Další skript instaluje do Jenkinse Maven, který je používán ke kompilaci zdrojového kódu a dále pro komunikaci s dalším Docker kontejnerem – se SonarQube. Pátý skript nastavuje časovou zónu, kterou Jenkins používá tak, aby odpovídala pražské časové zóně.

A konečně poslední skript slouží pro generaci ukázkových „*Jenkins jobů*“. V Jenkinsu se jako „*Jenkins joby*“ označují akce, které má smysl samostatně spouštět. Např. jedna může stáhnout ze systému správy verzí kód a pustit na něm testy, které pak zobrazí programátorům. Další může vždy v půlnoci zálohovat logy z aplikace apod. V tomto konkrétním případě se vytváří dva Jenkins joby – Test Merge Request a Merge to Master. Oba dva jsou podrobněji popsány v kapitole 5.6 na konkrétním příkladu.

5.3 GitLab Docker obraz

Obraz pro GitLab má ještě méně souborů než obraz pro Jenkins, sestává pouze ze tří souborů, viz seznam souborů na výpisu 10.

Výpis 10: Seznam souborů pro vytvoření Jenkins GitLab obrazu (zdroj: autor)

```
1 Dockerfile
2 entrypoint.sh
3 generate.sh
```

Opět, stejně jako v předchozí kapitole, zde podrobněji bude rozebrán především Dockerfile. Jelikož i v případě GitLabu existuje oficiální Docker obraz, obraz vytvářený v této práci od něj dědí, viz první řádek ve výpisu 11.

Výpis 11: Dockerfile pro Gitlab obraz (zdroj: autor)

```
1 FROM gitlab/gitlab-ce:9.5.1-ce.0
2
3 RUN apt-get update -y \
4     && apt-get install -y --no-install-recommends \
5         jq
6 ADD generate.sh /opt/gitlab/
7 ADD entrypoint.sh /opt/gitlab/
8 CMD ["/opt/gitlab/entrypoint.sh"]
```

Na řádcích tři až pět je vidět volání příkazů známých např. z Ubuntu pro instalaci balíčků – apt-get update a apt-get install, v tomto konkrétním případě instalují program jq, který slouží pro manipulaci s daty ve formátu JSON, což je využito v příkazu generate.sh. Příkazy na řádcích 6 a 7 kopírují soubory generate.sh a entrypoint.sh dovnitř do Dockeru do složky /opt/gitlab.

Pro pochopení toho, k čemu slouží příkaz na řádku osm, je nutné zopakovat něco z toho, jak fungují Docker kontejnery. Docker kontejner po nastartování běží, dokud neskončí proces specifikovaný buď příkazem CMD, nebo ENTRYPOINT (jsou i další způsoby, jak specifikovat tento proces, ale ty pro pochopení kódu nejsou podstatné). Z tohoto principu vyplývá, že tento proces může být jen jeden. Tím, že je specifikovaný příkaz CMD na řádku osm, se tedy překryl původní příkaz z předka, který tento hlavní proces specifikoval. Ze souboru Dockerfile předka verze gitlab/gitlab-ce:9.5.1-ce.0 je patrné, že původní příkaz vypadal takto: CMD ["/usr/local/bin/wrapper"] (GitLab, 2017e). Aby bylo na jednu stranu sice zachováno původní chování, ale na druhou stranu aby se určité příkazy vykonaly až poté, co naběhne kontejner s GitLabem, vytvořil jsem skript entrypoint.sh. Ten spustí na pozadí skript generate.sh a dále spustí původní skript /usr/local/bin/wrapper. Když skončí posledně jmenovaný, skončí i entrypoint.sh, a tak je původní chování zachováno.

Podobně, jak byl Jenkins v předchozím případě dále konfigurován pomocí skriptů ve složce init_groovy_scripts, byla u GitLabu také nutná další konfigurace, aby fungoval tak, jak má, hned po instalaci. V případě GitLabu ke konfiguraci slouží skript generate.sh.

Jeho obsah sem vzhledem k jeho rozsahu nebude kopírován, nicméně i tak považuji za vhodné zde popsat, co vše dělá. Nejdřív jen ve smyčce čeká, než se GitLab nastartuje. Když už je nastartován, vytvoří tři uživatele – root s heslem gitlabpassword s administrátorskými právy, uživatele jenkins s heslem jgpassword s běžnými uživatelskými právy a uživatele

alice s heslem *aliceingitlab* rovněž s běžnými právy. S uživatelem jenkins je dále spárován veřejný klíč (privátní klíč do páru byl nahrán do Jenkins Docker obrazu, jak bylo popsáno v předchozí kapitole), díky tomuto páru klíčů se může uživatel Jenkins z běžícího Jenkins kontejneru autentizovat například při volání příkazu `git` oproti GitLabu běžícímu v tomto kontejneru. Autentizaci pomocí klíče však nepodporuje GitLab u všech používaných funkcí – někde je nutný tajný token. Tento token uživatelé nemohou měnit, mohou si jen nechat vygenerovat nový. Není však možné jej získat, dokud neexistuje účet uživatele. Proto je po vytvoření uživatele *jenkins* ve skriptu voláno API GitLabu, kde je token získáván. Ten je následně do Jenkinse nahráván opět přes jeho API.

Ve skriptu je dále pomocí volání API vytvořen projekt s názvem *test*, do něhož může být pomocí Gitu nahráván zdrojový kód, jak je ukázáno v kapitole 5.6. Tento projekt je pro usnadnění inicializován prvním příspěvkem, kde je jediný soubor – README.md. Poslední, co je ve skriptu nastaveno, jsou tzv. háky (z angl. *hook*). Ty se starají o to, že když někdo nahraje nový kód úložiště, tak se kontaktuje Jenkins server a jsou spuštěny příslušné Jenkins joby – vytváření jobů bylo popsáno v předchozí kapitole 5.3 a to, co konkrétní Jenkins joby dělají, je popsáno v kapitole 5.6.

5.4 SonarQube Docker obraz

Obraz pro SonarQube je tvořen jen jediným souborem – souborem Dockerfile. Ten má pouhé tři řádky, viz výpis 12.

Výpis 12: *Dockerfile pro SonarQube obraz (zdroj: autor)*

```
1 FROM sonarqube:6.4
2
3 RUN wget https://github.com/gabrie-allaigre/sonar-gitlab-
   plugin/releases/download/2.0.1/sonar-gitlab-plugin-2.0.1.jar -P
   /opt/sonarqube/extensions/plugins
```

Opět je na prvním řádku uvedeno, ze kterého Docker obrazu tento obraz bude dědit. V tomto případě se jedná o SonarQube ve verzi 6.4, který je v době psaní této práce nejnovější. Příkaz na řádku tři stahuje z internetu doplněk pro práci s GitLabem a nahrává jej do složky pro doplňky. To, že je tento soubor s příponou `.jar` nakopírován do správné složky, je vše, co je nutné pro jeho instalaci vykonat. SonarQube při spuštění danou složku prohledá a všechny doplňky zde stažené automaticky použije. Toto však není všechna konfigurace, která je nutná pro úspěšné provozování Docker kontejneru se SonarQube. Další konfigurace je popsána v následující kapitole.

5.5 Uživatelská příručka administrátora

V této kapitole bude popsáno, jak praktické řešení spouštět, vypínat, restartovat, případně měnit konfiguraci. Dále budou popsány některé detaily řešení, které se nehodily do předchozí kapitoly, jelikož byly pro všechny obrazy společné.

5.5.1 Základní obsluha a změna konfigurace

Nejpřehlednější bude patrně začít s popisem souborů, se kterými člověk starající se o běh aplikací přijde do styku. Výpis z kořenového adresáře je vidět na výpisu 13.

Výpis 13: Seznam souborů pro vytvoření Jenkins GitLab obrazu (zdroj: autor)

```
1 gitlab/  
2 jenkins/  
3 sonar/  
4 README.md  
5 config.sh  
6 docker-compose.yml  
7 start.sh*
```

Na prvních třech řádcích jsou vidět složky s předpisy pro vytváření Docker obrazů – ty byly podrobněji popsány v kapitolách 5.2-5.4. V README.md je stručný anglicky psaný návod pro použití, jelikož zdrojové kódy k praktické části budou publikovány na mém GitHub účtu.

Soubor docker-compose.yml je speciální soubor pro použití při volání příkazu docker-compose. Jsou v něm definované parametry pro jednotlivé kontejnery, například v části **ports** u každého kontejneru je uvedeno, jak mapovat vnitřní porty kontejneru na vnější porty hostitelského stroje. Dále je zde uvedeno, jaké složky se mají sdílet s hostitelským strojem – to je v kontejnerech k vidění u části **volumes**. Důležité jsou ještě části **build**, **networks**, **restart** a **environment**. V části **build** je určeno, jak se má obraz vytvořit – jestli stáhnout z úložiště obrazů, nebo sestavit na lokálním počítači. V tomto konkrétním případě se tři obrazy sestavují (GitLab, Jenkins a SonarQube) lokálně a jeden s databází stahuje z internetu. Pro provozování SonarQube na produkčním prostředí je totiž doporučováno používat externí kontejner s PostgreSQL databází. V části **networks** je nastaveno, jak mají být běžící kontejnery viditelné pro ostatní běžící kontejnery. Pokud se např. nacházíme uvnitř GitLab kontejneru, je Jenkins dostupný pod jménem jenkins.dev. Toho je využíváno například ve skriptu **gitlab/generate.sh**, kde GitLab po naběhnutí volá Jenkins API právě přes URL jenkins.dev a upravuje některá nastavení, ukládá klíče atd. Část **restart** upravuje, co se má stát poté, co kontejner skončí. V tomto případě je všude nastaveno, že pokud skončí chybou, má kontejner opět nastartovat. A konečně část **environment** nastavuje proměnné prostředí v kontejneru. Mohou tam být nastaveny přímo hodnoty, nebo – jako v našem případě – tam mohou být předávány hodnoty, které jsou nastaveny v prostředí hostitelského počítače.

Tyto hodnoty jsou k vidění v souboru `config.sh`, např. hodnota hesla pro uživatele `alice` k Jenkins je nastavena řádkem `export JENKINS_DEVELOPER_PASSWORD=aliceinjenkins`. Výhodou tohoto oddělení je to, že všechna konfigurovatelná nastavení jsou přehledně na jednom místě a člověk je nemusí hledat možná pro někoho ve složitém souboru `docker-compose.yml`.

5.5.2 Obsluha

Zde bude popsáno, jak aplikaci spustit. Možností je více. Lidé, kteří mají obecně s Dockerem větší zkušenosti a budou tak pravděpodobně chtít mít věci více pod kontrolou, budou pravděpodobně volat přímo příkazy `docker-compose` (`build`, `up`, `down`, `stop`, `start`, ...). Jediné, co je však předtím potřeba udělat, je zavolat příkaz `source config.sh`, aby byly správně nastavené proměnné prostředí, na které se odkazuje soubor `docker-compse.yml`.

Lidé, kteří nemají s Dockerem větší zkušenosti nebo jsou jenom pohodlnější, pravděpodobně využijí připravený skript `start.sh`. Ten musí mít vždy jeden parametr z množiny `full-restart`, `restart`, `start`, `stop`, `health` a `stats`. Co dělají parametry `start`, `stop` a `restart` patrně není třeba vysvětlovat. Naopak podrobnější popis si zaslouží příkaz `full-restart`, jelikož je do určité míry nebezpečný. Kromě zastavení a smazání kontejnerů, znovuvytvoření obrazů a puštění nových kontejnerů totiž navíc maže také adresáře synchronizované s hostitelským strojem. Pokud tedy pouštíme aplikace poprvé, je bezpečné tento parametr použít, rovněž je vhodné jej použít, pokud celé řešení jen testujeme, měníme konfiguraci a nevadí nám, že mezi běhy o všechna ruční natavení (např. naklikaná přes Jenkins či GitLab UI) přijdeme.

Volání skriptu s parametry `health` a `stats` zobrazí různé detaily o běžících kontejnerech, jako je využití procesoru, paměti či „zdraví“ kontejnerů – to, jestli kontejner běží v pořádku, startuje nebo zda jsou zde nějaké problémy.

Když aplikaci pouštíme úplně poprvé, stačí zavolat `./start.sh start` a poté `./start.sh health` a kontrolovat, zda všechny kontejnery naběhly správně. První `start` může být pomalejší, jelikož se budou stahovat řádově GB dat. Když už je ale vše staženo, tak po dalším zavolání skriptu s parametrem `start` nebo `full-restart` bude vše trvat daleko kratší dobu, ovšem záleží na výkonu hostitelského počítače. Např. na stroji s RAM 16 GB a čtyřjádrovém procesoru o frekvenci 2.5 GHz od spuštění příkazu s parametrem `full-restart` po všechny správně běžící kontejnery uběhlo necelých pět minut.

Řešení bylo testováno jak na slabším stroji, který měl již dříve zmíněných 16 GB RAM a čtyřjádrový procesor, tak i na výkonnostně silnějším stroji s 144 GB RAM a 16 procesory. V obou případech řešení fungovalo bez potíží. Jediné, co bylo potřeba na menším stroji upravit, bylo, kolik RAM z hostitelského počítače může využít. Kontejner s Jenkinsem při neaktivitě okupoval něco kolem 700 MB RAM. GitLab a SonarQube pak každý potřeboval k běhu přibližně 1 GB. Kontejner s databází pro SonarQube zabíral nejméně paměti – něco

kolem 40 MB. Když bylo Dockeru dovoleno používat méně RAM než výsledné 3 GB, začal vypínat jednotlivé kontejnery. Poté, co Dockeru bylo povoleno zabrat více RAM a používat dvě jádra procesoru, nebyl při testování žádný kontejner samovolně vypnut ani jednou.

5.6 Uživatelská příručka vývojáře

Uživatelská příručka vývojáře je rozsáhlejší než uživatelská příručka administrátora, a to především kvůli tomu, že její součástí je i konkrétní příklad, kde je krok po kroku ukázáno, jak nahrát Java aplikaci (zdrojové kódy jsou též součástí praktického řešení) do GitLabu, co udělat pro to, aby na ní Jenkins spustil testy, a kde si výsledky těchto testů zobrazit. Celý návod je psán pro výchozí nastavení. Pokud tedy bude cokoliv v nastavení změněno, nemusí vše fungovat tak, jak má.

Prerokvizitou pro následování tohoto návodu je mít celé řešení spuštěné na lokálním počítači. To, jak jej spustit, je popsáno v kapitole 5.5. Z hlediska vývojáře můžeme otestovat, že vše běží tak, jak má, např. přihlášením se do jednotlivých nástrojů přes webový prohlížeč.

5.6.1 Základní otestování běžící aplikace

Pokud tedy vše máme spuštěno na lokálním počítači, můžeme se do Jenkins přihlásit přes URL <http://localhost:9999> se jménem alicie a heslem alicieinjenkins. Po přihlášení by měly být vidět dva Jenkins joby – Test Merge Request a Merge to Master. Nyní nemá smysl je spouštět, budou spuštěny automaticky, až bude kód nahrán do GitLabu.

Sonar je dostupný na portu 9000 s výchozím jménem admin a heslem rovněž admin. Po přihlášení však bude naprosto prázdný, nebude zde k vidění žádný projekt. Ten bude vytvořen, až bude automaticky spuštěn Jenkins job Merge to Master.

Do GitLabu je možné přihlásit se přes URL <http://localhost> s uživatelským jménem alicie a heslem alicieingitlab. Po přihlášení by měl být viditelný projekt s označením jenkins/test dostupný z adresy <http://localhost/jenkins/test>. Projekt je zatím téměř prázdný, obsahuje jeden soubor – README.md.

5.6.2 Příklad práce na ukázkovém projektu

Když už vše běží, je možné vyzkoušet řešení na ukázkovém Java projektu. V příloze k této práci je zabalený archiv, jehož součástí jsou dva adresáře – `docker-jenkins-gitlab-sonar` a `mvc-tasks`. Rozbalme tedy archiv a složku `mvc-tasks` přesuňme např. do domovského adresáře uživatele. Měl by tedy být dostupný na cestě `~/mvc-tasks`. Nejdříve si z lokálního GitLabu stáhneme projekt. Toho můžeme dosáhnout mnoha způsoby, já popíšu ten, který je

mi nejbližší a zároveň je univerzální – přes linuxovou příkazovou řádku. Ta je nově dostupná i na Windows, takže by nemusel být problém s jejím použitím ani na stroji s tímto operačním systémem. Z domovského adresáře tedy zavolejme jednoduchý příkaz `git clone http://alice:aliceingitlab@localhost/jenkins/test.git`. To zapříčiní, že se z lokálního GitLabu stáhne projekt do adresáře `test`. Příkazem `cd ~/test` se do tohoto adresáře dostaneme a uvidíme již dříve zmíněný soubor `README.md`. Nyní se nacházíme na větvi `master`. Pracovat přímo na hlavní větvi není dobrým zvykem, proto si vytvoříme vlastní větev příkazem `git checkout -b first-app`. Nyní už se nacházíme na větvi „`first-app`“. Můžeme tak nakopírovat obsah adresáře `mvc-tasks` `cp -r ~/mvc-tasks/* ~/test`. Trojicí příkazů `git add .`, `git commit -m „added app draft“` a `git push origin first-app` nahrajeme lokální změny do GitLabu. Tyto změny je možné si prohlédnout na URL <http://localhost/jenkins/test/branches> a kliknutím na větev `first-app`. Předpokládejme, že jsme se změnami spokojeni a chceme tuto větev spojit do `master` větve. K tomu slouží tlačítko „Create merge request“. Po kliknutí na něj se zobrazí informace s několika detaily. Zde je důležité zkontrolovat, zda souhlasí větev, kterou chceme spojit (`first-app`), a větev, s níž chceme danou větev spojit (`master`). Pokud toto souhlasí, je možné kliknout na tlačítko „Submit merge request“. Nyní mohou např. kolegové vývojáři zkontrolovat změny ve větvi, okomentovat je nebo odsouhlasit, že se změnami souhlasí a že tedy nemají problém s napojením této větve do `masteru`.

Zajímavější je ale to, že díky tomu, jak byl GitLab a Jenkins v této práci nastaven, GitLab kontaktoval Jenkins a spustil na něm job `Test Merge Request`. V něm Jenkins stáhne danou větev a spustí na ní různé testy. Samozřejmostí je spuštění přiložených `jUnit` testů a následné zobrazení, kolik z nich jich prošlo a kolik ne – jak je vidět na obrázku obrázku 7:

Test Result

0 failures (±0)

4 tests (±0)

Took 2.2 sec.

 [add description](#)

All Tests

Package	Duration	Fail	(diff) Skip	(diff) Pass	(diff) Total	(diff)
eu.okosy.thesis.controller	2.2 sec	0	0	1	1	
eu.okosy.thesis.service	0 ms	0	0	3	3	

Obrázek 7:

Výsledky jUnit testů (zdroj: autor)

Podle mého názoru je ale nejzajímavější z hlediska vývojáře to, že oproti danému kódu je puštěna také SonarQube analýza. Pokud SonarQube nalezne nějaké problémy s kódem, rovnou ke kódu do GitLabu přidá komentáře. V přiloženém projektu `mvc-tasks` bylo záměrně ponecháno několik nedostatků, aby byla tato funkce předvedena v praxi, výsledek je též vidět na obrázku 8.

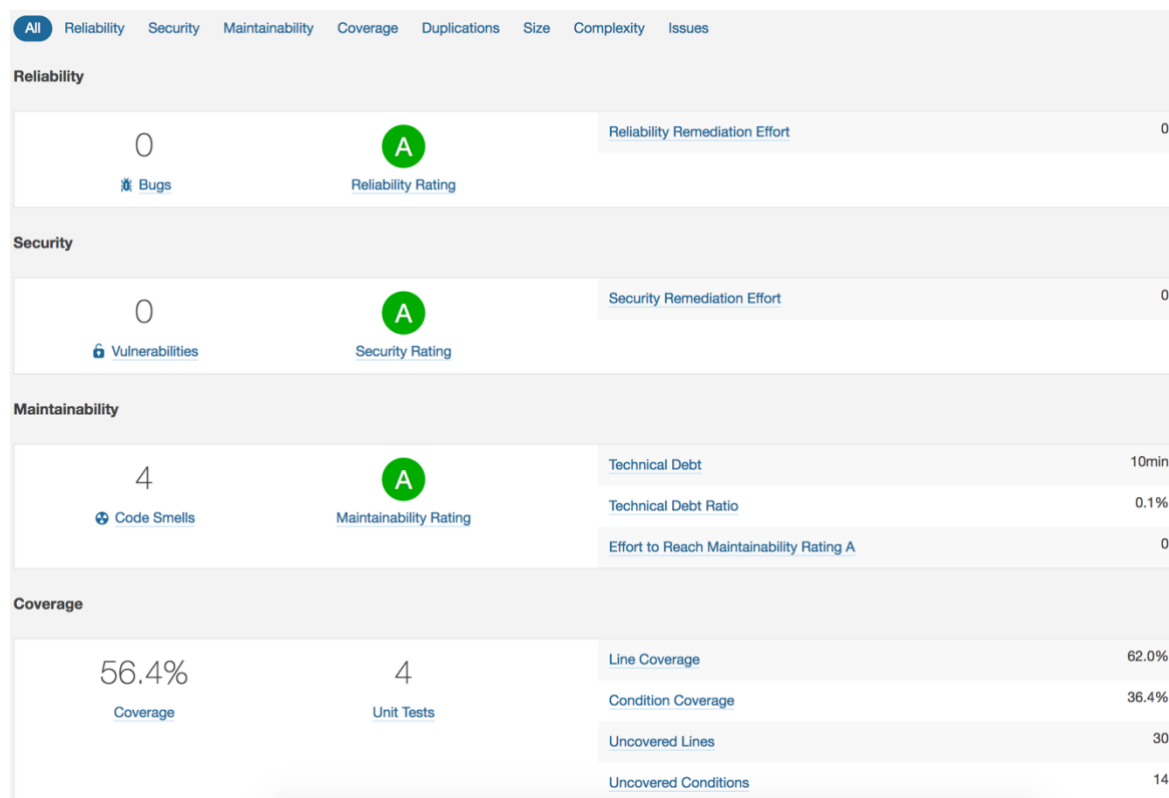
**Obrázek 8:**

Komentář ke GitLab merge requestu ze SonarQube analýzy (zdroj: autor)

Dle komentářů je tedy možné kód dále upravit a změny nahrát do GitLabu. Jenkins job bude opětovně automaticky spuštěn a nyní už nepřidá žádné komentáře. Co ale nyní bude viditelné, je trend výsledků unit testů, jelikož se jedná už o druhé spuštění tohoto jobu. Pokud tedy kolegové souhlasí a vyřešili jsme všechny problémy nalezené SonarQubem, můžeme v GitLabu kliknout na tlačítko Merge. To jednak spojí naši vytvořenou větev do větve master, ale hlavně také spustí druhý připravený jenkins job – Merge to Master.

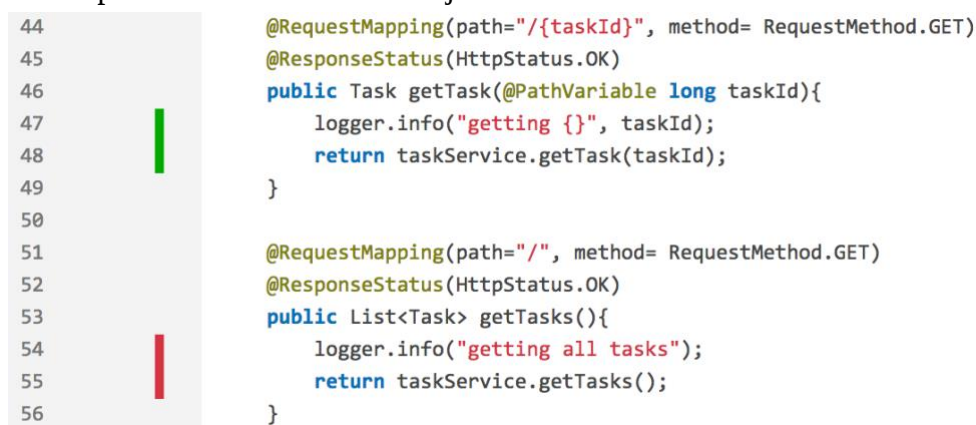
Tento Jenkins job je velice podobný předchozímu s tím rozdílem, že již nepřidává nalezené chyby jako komentáře do GitLabu – nemá to smysl, když není otevřený merge request. Na rozdíl od prvního jobu však publikuje výsledky do SonarQube. Toho je dosaženo pomocí parametru `sonar.analysis.mode` nastaveného na hodnotu `publish`. U předchozího Jenkins jobu toto nastavení bylo záměrně nastaveno na `preview`, aby výsledky z analýzy v SonarQube nebyly k vidění. To z toho důvodu, že by se stále přepisovaly, pokud by například více vývojářů současně mělo otevřený merge request. Takto výsledky viditelné v SonarQube odráží aktuální stav hlavní větve, tedy větve, která prošla jak automatizovanými kontrolami, tak např. kontrolami kolegů.

V SonarQube na URL http://localhost:9000/component_measures?id=okosy-thesis%3Amvc-tasks jsou vidět statistiky a metriky pro daný projekt. Část základního přehledu metrik je vidět na obrázku 9.

**Obrázek 9:**

Část výsledku SonarQube analýzy pro ukázkový projekt (zdroj: autor)

Metrik je opravdu mnoho a většinu z nich lze dále rozkliknout a získat tak více podrobnějších informací. Například při kliknutí na „Line Coverage“ se zobrazí seznam souborů s informacemi o pokrytí daného souboru testy. Po kliknutí na daný soubor je pak vidět celý zdrojový kód s vyznačenými otestovanými a neotestovanými řádky, jak je vidět na obrázku 10 pro soubor TaskController.java.

**Obrázek 10:**

Detail pokrytí testy třídy TaskController (zdroj: autor)

V podobném duchu by bylo možné popsat, jak SonarQube zobrazuje desítky dalších ukazatelů a metrik. Jejich popis však není cílem této práce, ani to s cíli této práce přímo nesouvisí. Podrobnější informace o SonarQube je možné nalézt v dokumentaci k tomuto nástroji a obecné informace o metrikách pro měření kvality kódu byly popsány v teoretické části práce.

Tímto je příklad u konce, a tak ještě jednou shrnu, co vše se vlastně událo. Na začátku jsme z GitLabu naklonovali prázdný projekt. Poté jsme vytvořili vlastní větev, do ní jsme nakopírovali připravené zdrojové kódy a změny jsme nahráli zpět na GitLab. Poté jsme přes UI GitLabu vytvořili tzv. merge request, který spustil Jenkins job Test Merge Request. V tomto Jenkins jobu byly spuštěny jUnit testy a SonarQube analýza. Výsledky testů jsou k vidění v detailu Jenkins jobu a některé výsledky SonarQube analýzy byly k vidění v GitLabu prostřednictvím komentářů ke kódu. Po spojení nově vytvořené větve s master větví byl automaticky spuštěn na Jenkinsu job Merge to Master. Ten opět spustil testy a analýzy, tentokrát však výsledky už publikoval přímo do SonarQube, kde jsou k nahlédnutí např. pro manažera projektu.

5.6.3 Rozšíření o další projekty

Pokud bude toto řešení používáno více projekty, není to problém. Jen je k tomu potřeba provést několik kroků, které jsou na pomezí náplně práce vývojáře a administrátora. Jednak je nutné např. pomocí Jenkins UI okopírovat oba dva Jenkins joby, a to včetně jejich nastavení. To je nutné upravit, aby komunikovalo s novým projektem na GitLabu, ne s testovacím, který byl použit v této kapitole. Dále je nutné jít do UI GitLabu a tam vytvořit nový projekt. K němu musí mít plná práva uživatel jenkins – aby jej mohl stahovat i komentovat kód. Také je nutné správně nastavit něco, co GitLab označuje jako *webhooks* – to proto, aby GitLab vždy ve správný okamžik voláním Jenkins API pustil dříve zmíněné Jenkins joby. Pro detaily tohoto nastavení doporučuji inspirovat se vytvořeným testovacím projektem na URL <http://localhost/jenkins/test/settings/integrations>. Stran SonarQube není žádné další nastavení nutné, zde je projekt automaticky vytvořen Jenkins jobem Merge to Master, pokud bude tento job správně nastaven.

6 Závěr

Hlavním cílem práce bylo analyzovat aplikace a technologie vhodné při použití CI u Java projektů a pomocí platformy Docker implementovat konkrétní řešení, které bude postavené na těchto technologiích. Tento cíl jsem rozdělil do tří dílčích cílů.

Prvním z nich bylo objasnit pojem CI a s ním související pojmy. Tohoto cíle bylo dosaženo v teoretické části, respektive v kapitole 3, kde bylo CI podrobně popsáno. V souvislosti s CI byly též v kapitole 3 popsány pojmy s CI úzce související – především Continuous Delivery, Continuous Deployment a DevOps. Dále zde byl popsán model kvality softwaru dle norem řady ISO/IEC 250nn a z něj podrobněji především dvě charakteristiky – udržitelnost a přenositelnost, jelikož ty souvisí s CI nejvíce.

Druhým dílčím cílem bylo analyzovat typy aplikací technologií vhodných při CI u Java projektů, jež je možné zdarma užívat i pro komerční účely; v rámci každého identifikovaného typu aplikace vzájemně porovnat a vybrat nejvhodnější. Tento dílčí cíl byl naplněn v kapitole 4. V té jsem na základě analýzy procesu CI z předchozí kapitoly našel tři druhy aplikací, konkrétně: CI servery, systémy správy verzí (VCS) a nástroje pro analýzu kódu. Pro dané účely specifikované zaměřením této práce byl jako nejvhodnější CI server po porovnání s ostatními vybrán Jenkins. Jako systém správy verzí byl po srovnání vybrán GitLab a jako nejvhodnější nástroj pro analýzu kódu byl vybrán SonarQube.

Posledním dílčím cílem bylo za použití výstupů z předchozího cíle implementovat integrované řešení pro CI. Nutnou podmínkou pro splnění tohoto cíle bylo, aby všechny aplikace tohoto řešení běžely v samostatných Docker kontejnerech. Splnění tohoto cíle je popsáno v kapitole 5. V té je zdokumentováno, jak byly obrazy s jednotlivými aplikacemi vytvářeny a jak je možné je používat. Zdrojové kódy, jež byly vytvořeny v rámci řešení tohoto cíle, jsou součástí elektronické přílohy této práce a jsou dostupné rovněž i na GitHubu (Okosy, 2017).

V této kapitole jsou i samostatné podkapitoly označené jako uživatelská příručka administrátora a uživatelská příručka vývojáře. První z nich je věnována případným administrátorům daného řešení – je zde ukázáno, jak z Docker obrazů spouštět Docker kontejnery, měnit konfiguraci a jsou zde popsána úskalí a omezení, na která je možné během užívání narazit. V druhé kapitole, jež je určena pro vývojáře, je na jednoduchém předpřipraveném Java projektu, jehož zdrojové kódy jsou v příloze, krok po kroku ukázáno, jak používat CI řešení. Jsou zde popsány příkazy pro naklonování téměř prázdného projektu z GitLabu, dále pro jeho úpravu (nakopírování obsahu přílohy) a následné nahrání zpět do úložiště kódu. GitLab poté přes API informuje o změnách Jenkins, který automaticky stáhne nový kód z GitLabu, spustí oproti němu přiložené jednotkové testy a další analýzy kódu za pomoci serveru SonarQube. V případě nalezených problémů jsou výsledky analýzy ihned publikovány formou komentářů zpět do GitLabu, aby tak zajistily rychlou a užitečnou zpětnou vazbu vývojářům. Pro management jsou výsledky analýz všech projektů přehledně dostupné na serveru SonarQube.

Terminologický slovník

Termín	Význam (zdroj)
CDBI	Z angl. Continuous Database Integration (česky kontinuální integrace databáze): „Continuous Database Integration (CDBI) is the process of rebuilding your database and test data any time a change is applied to a project’s version control repository,“ (Duval et al., 2007, s. 107).
CDV	Z angl. Continuous Delivery (česky průběžná dodávka): „Continuous delivery is the practice of continuously deploying good software builds automatically to some environment, but not necessarily to actual users,“ (Fitzgerald a Stol, 2015).
CDP	Z angl. Continuous Deployment (česky průběžné nasazení): „Continuous deployment implies continuous delivery and is the practice of ensuring that the software is continuously ready for release and deployed to actual customers,“ (Fitzgerald a Stol, 2015).
CI	Z angl. Continuous Integration (česky průběžná integrace): „...software development practice where members of a team integrate their work frequently, usually each person integrates at least daily – leading to multiple integrations per day. Each integration is verified by an automated build (including test) to detect integration errors as quickly as possible “ (Fowler, 2006).
Controller (mvc)	„Controller – řadič...řídí tok událostí v programu (tzv. aplikační logika) obsahuje přímý odkaz na model, aby mohl modifikovat jeho data,“ (Hordějčuk, 2017).
CPD	Součást PMD sloužící pro hledání kopírovaného kódu „...additionally it includes CPD, the copy-paste-detector. CPD finds duplicated code in Java, C, C++...“ (PMD, 2017a).
DevOps	„DevOps represents a change in IT culture, focusing on rapid IT service delivery through the adoption of agile, lean practices in the context of a system-oriented approach. DevOps emphasizes people (and culture), and seeks to improve collaboration between operations and development teams. DevOps implementations utilize technology — especially automation tools that can leverage an increasingly programmable and dynamic infrastructure from a life cycle perspective,“ (Gartner, nedatováno).
GNU GPL	„GNU GPL (GNU General Public License) je softwarová licence řešící především otázku přístupnosti a distribuce zdrojových kódů softwaru. Je koncipována tak, že autor je povinen zpřístupnit zdrojové kódy programu, pokud se rozhodne své dílo (re)distribuovat,“ (unchallenger, 2004)

Jenkins job	Jenkins job je předem definovaná posloupnost kroků, které Jenkins provádí. Mezi tyto kroky může patřit spuštění skriptu, stažení kódu z VCS, sestavní stažení kódu, volání libovolného API aj. Spuštění Jenkins jobu může být manuální, časově řízené (např. každou hodinu), událostmi řízené (např. při nahrání kódu do VCS) nebo jejich kombinací (autor).
Kastomizace	„... typové aplikace je nutné pro potřeby konkrétního podniku, resp. uživatele, upravovat, resp. kastomizovat. Kastomizace většinou probíhá na základě analýzy požadavků uživatelů...“ (Gála, Pour, Šedivá, 2009) Kastomizace je tedy úprava aplikace na základě požadavků konkrétního klienta (autor).
MVC	„Architektura MVC dělí aplikaci na 3 logické části tak, aby je šlo upravovat samostatně a dopad změn byl na ostatní části co nejmenší. Tyto tři části jsou Model, View a Controller. Model reprezentuje data a business logiku aplikace, View zobrazuje uživatelské rozhraní a Controller má na starosti tok událostí v aplikaci a obecně aplikační logiku,“ (Bernard, 2009)
PHP	„PHP (recursive acronym for PHP: Hypertext Preprocessor) is a widely-used open source general-purpose scripting language that is especially suited for web development and can be embedded into HTML,“ (PHP Group, 2017)
PMD	„PMD is a source code analyzer. It finds common programming flaws like unused variables, empty catch blocks, unnecessary object creation, and so forth. It supports Java, JavaScript, Salesforce.com Apex and Visualforce, PLSQL, Apache Velocity, XML, XSL,“ (PMD, 2017a).
RUP	„Rational Unified Process (zkráceně RUP) je komerční metodika společnosti IBM poskytující systematický přístup k vývoji SW. Obsahuje detailní návody, jak postupovat, aby byly splněny cíle a očekávání zadavatelů zakázky a zároveň byla splněna předem definovaná kvalita produktu, jeho rozsah, termín dodání a rozpočet. Popisuje fáze životního cyklu vývoje SW, definuje potřebné role, artefakty, odpovědnosti a pracovní procesy.“ (Julek, 2008)
SQL	Z angl. Structured Query Language (česky strukturovaný dotazovací jazyk): „SQL is a standard language for storing, manipulating and retrieving data in databases,“ (W3Schools, 2017b).
URL	Z angl. Uniform Resource Locator (česky jednotná adresa zdroje): „URL is an acronym for Uniform Resource Locator and is a reference (an address) to a resource on the Internet,“ (Oracle, 2017)
VCS	Z angl. Version Control System – systém správy verzí. „(VCS) je systém, který zaznamenává změny souboru nebo sady souborů v průběhu času,“ (Git, 2017).
VPS	Virtuální privátní server – „Virtuální server je samostatný vyhrazený prostor fyzického serveru, který si zákazník pronajímá do užívání. Služba vyplňuje prostor mezi běžným webhostingem a serverhostingem“ (Forpsi, 2017).

XML	<i>„XML stands for eXtensible Markup Language. XML is a markup language much like HTML. XML was designed to store and transport data. XML was designed to be self-descriptive. XML is a W3C Recommendation,“ (W3Schools, 2017a).</i>
XPath	<i>„XPath is a major element in the XSLT standard. XPath can be used to navigate through elements and attributes in an XML document,“ (W3Schools, 2017c).</i>

Použité informační zdroje

ALTERNATIVETO. Alternatives to Jenkins for all platforms with any licence. In: *AlternativeTo: Croudsourced software recommendations* [online]. 2017 [cit. 2017-07-06]. Dostupné z: <http://alternativeto.net/software/jenkins/>

ALTERNATIVETO. Alternatives to GitHub for all platforms with free licence. In: *AlternativeTo: Croudsourced software recommendations* [online]. 2017 [cit. 2017-07-15]. Dostupné z: <http://alternativeto.net/software/github/?license=free>

ANON. *Subversion vs. Git: Myths and Facts* [online]. 2016 [cit. 2017-07-08]. Dostupné z: <https://svnvsgit.com/>

APACHE. Apache Subversion. In: *Apache Subversion* [online]. 2017 [cit. 2017-10-14]. Dostupné z: <https://subversion.apache.org/>

ATTLASSIAN. *Saving changes* [online]. Nedatováno [cit. 2017-06-05]. Dostupné z: <https://www.atlassian.com/git/tutorials/saving-changes#git-commit>

ATLASSIAN. Bamboo pricing. In: *Atlassian* [online]. 2017 [cit. 2017-07-06]. Dostupné z: <https://www.atlassian.com/software/bamboo/pricing>

BACH, James. *What is exploratory testing?* [online]. Nedatováno [cit. 2017-05-27]. Dostupné z: http://www.satisfice.com/articles/what_is_et.shtml

BLACKDUCK SOFTWARE. *Compare Repositories* [online]. 2017 [cit. 2017-07-08]. Dostupné z: <https://www.openhub.net/repositories/compare>

BERNARD, Borek. Úvod do architektury MVC In: *Zdroják.cz* [online]. 2009 [cit. 2017-10-14]. Dostupné z: <https://www.zdrojak.cz/clanky/uvod-do-architektury-mvc/>

BOOCH, Grady. *Object Solutions: Managing the Object-Oriented Project*. Addison-Wesley, 1996. ISBN 978-0805305944.

BROKS, Andy. McCabe's Cyclomatic Complexity (V(G) also known as CC). In: *UNAK.is* [online]. Nedatováno [cit. cit. 2017-06-28]. Dostupné z: http://staff.unak.is/andy/staticanalysis0809/metrics/cyclomatic_complexity.html

CEALIFERUM. Gogs or Gitea – which is more active? In: *Reddit* [online]. 2017 [cit. 2017-07-15]. Dostupné z: https://www.reddit.com/r/golang/comments/5jq1cm/gogs_or_gitea_which_is_more_active/

CLOUDBEES. *Jenkins and CloudBees* [online]. Nedatováno [cit. 2017-07-07]. Dostupné z: <https://www.cloudbees.com/jenkins/jenkins-cloudbees>

CLARKWARE. JDepend [online]. 2017 [cit. 2017-05-18]. Dostupné z: <https://github.com/clarkware/jdepend>

COBERTURA. Cobertura. In: *Cobertura: A code coverage utility for Java* [online]. Nedatováno [cit. 2017-07-30]. Dostupné z: <http://cobertura.github.io/cobertura/>

COBERTURA. Commits. In: *GitHub* [online]. 2017 [cit. 2017-07-30]. Dostupné z: <https://github.com/cobertura/cobertura/graphs/commit-activity>

COBERTURA. Commits. In: *GitHub* [online]. 2017 [cit. 2017-07-30]. Dostupné z: <https://github.com/cobertura/cobertura/blob/master/LICENSE.txt/>

CODACY. Features. In: *Codacy: Automated Code review* [online]. 2017 [cit. 2017-07-30]. Dostupné z: <https://www.codacy.com/features>

CODACY. Jenkins Plugin. In: *Codacy: Automated Code review* [online]. 2017 [cit. 2017-07-30]. Dostupné z: <https://support.codacy.com/hc/en-us/articles/115000974825-Jenkins-Plugin>

CODACY. Pricing. In: *Codacy: Automated Code review* [online]. 2017 [cit. 2017-07-30]. Dostupné z: <https://www.codacy.com/pricing>

CODACY. Code Patterns. In: *Codacy: Automated Code review* [online]. 2017 [cit. 2017-07-30]. Dostupné z: <https://support.codacy.com/hc/en-us/articles/207994335-Code-Patterns>

CODINGTONY. Dockerfile for Java. In: *GitHub* [online]. 2015 [cit. 2017-07-16]. Dostupné z: <https://github.com/codingtony/docker/tree/master/rhodecode>

COOK, Joshue. *Docker for data science: Building Scalable and Extensible Data Infrastructure Around the Jupyter Notebook Server*. Santa Monica(USA): Apress, 2017. ISBN 978-1-4842-3012-1.

ČESKÝ STATISTICKÝ ÚŘAD. *Počítače a internet v českých domácnostech* [online]. Praha, 21.11.2016 [cit. 2017-06-03]. Dostupné z: https://www.czso.cz/documents/10180/49755661/csu_tk_internet_prezentace.pdf/48b0390b-bc97-4327-a0b2-92857a1e53e7?version=1.0

ČSN ISO/IEC 9126-1: *Softwarové inženýrství – Jakost produktu – Část 1: Model jakosti*. Praha: Český normalizační institut, 2002.

DEVEO. *Git vs SVN: Which version control systém should you choose?* [online]. 2017 [cit. 2017-07-08]. Dostupné z: <https://deveo.com/git-vs-svn/>

DESIKAN, Srinivasan. *Software testing: Principles and Practices*. India: Pearson Education, 2006. ISBN 978-8177581218.

DOCKER. Docker documentation. In: *Docker.com* [online]. 2017 [cit. 2017-04-01]. Dostupné z: <https://docs.docker.com/>

DOCKER. What is a Container. In: *Docker.com* [online]. 2017 [cit. 2017-09-29].

Dostupné z: <https://www.docker.com/what-container>

DOCKER. Definition of: container. In: *Docker.com* [online]. 2017 [cit. 2017-09-30].

Dostupné z: <https://docs.docker.com/glossary/?term=container>

DOLEŽEL, Michal. *Řízení kvality softwaru*. Přednáška. Praha: VŠE, 9.3.2017

DUVAL, Paul, MATYAS, Steve, a GLOVER, Andrew. *Continuous Integration: Improving Software Quality and Reducing Risk*. Boston(USA): Pearson Education, Inc., 2007. ISBN 978-0-321-33638-5.

ECLIPSE FOUNDATION. *Hudson Core* [online]. 2017 [cit. 2017-07-07]. Dostupné z:

<https://git.eclipse.org/c/hudson/org.eclipse.hudson.core.git/>

F99AQ8OVE. Docker-gitbucket. In: GitHub [online]. 2017 [cit. 2017-07-15]. Dostupné z:

<https://github.com/f99aq8ove/docker-gitbucket>

FEDOR, Jaroslav. *Systém kontinuální integrace pro rychlý vývoj webových aplikací*

[online]. Brno, 2016 [cit. 2017-03-31]. Dostupné z: http://is.muni.cz/th/422370/fi_b/.

Bakalářská práce. Masarykova univerzita, Fakulta informatiky. Vedoucí práce Tomáš Pitner.

FINDBUGS. Adding Detectors to FindBugs. In: *FindBugs* [online]. 2003 [cit. 2017-07-30].

Dostupné z: <http://findbugs.sourceforge.net/AddingDetectors.txt>

FINDBUGS. FindBugs™: Find Bugs in Java Programs. In: *FindBugs* [online]. 2015 [cit.

2017-07-30]. Dostupné z: <http://findbugs.sourceforge.net/>

FINDBUGS. Commits. In: *GitHub* [online]. 2017 [cit. 2017-07-30]. Dostupné z:

<https://github.com/findbugsproject/findbugs/graphs/commit-activity>

FITZGERALD, Brian, STOL, Klaas-Jan. *Continuous software engineering: A roadmap and agenda*. Journal of Systems and Software, 2015. Dostupné z: doi:

10.1016/j.jss.2015.06.063

FORPSI. Cloud VPS Smart In: *forpsi.cz* [online]. 2017 [cit. 2017-10-14]. Dostupné z:

<https://www.forpsi.com/virtual/>

FOWLER, Martin. *Continuous Integration* [online]. 2006 [cit. 2017-05-13]. Dostupné z:

<https://www.martinfowler.com/articles/continuousIntegration.html>

FOWLER, Martin. *Open-sourcing ThoughtWorks Go* [online]. 2014 [cit. 2017-07-09].

Dostupné z: <https://martinfowler.com/articles/go-interview.html>

FOWLER, Martin. *Test Coverage* [online]. 2012 [cit. 2017-05-14]. Dostupné z:

<https://martinfowler.com/bliki/TestCoverage.html>

FOWLER, Martin, BECK, Kent. *Refactoring: improving the design of existing code*. Reading, MA: Addison-Wesley, 1999. ISBN 978-0201485677

FRIEBELOVÁ, Jana. *Vícekritériální rozhodování za jistoty* [online]. Nedatováno [cit. 2017-07-05]. Dostupné z <http://www2.ef.jcu.cz/~jfrieb/tspp/data/teorie/Vicekritko.pdf>

GAMMA, Erich, HELM, Richard, JOHNSON, Ralph a VLISSIDES, John. *Design Patterns*. Boston(USA): Addison-Wesley, 1994. ISBN 978-0201633610

GARTNER. DevOps. In: *Gartner IT Glossary* [online]. Nedatováno [cit. 2017-06-17]. Dostupné z: <http://www.gartner.com/it-glossary/devops/>

GÁLA, Libor, Jan POUR a Zuzana ŠEDIVÁ. *Podniková informatika. 2., přepracované a aktualizované vydání*. Praha: Grada, 2009. ISBN 978-80-247-2615-1.

GIT. Správa verzí. In: *Git* [online]. 2017 [cit. 2017-09-07]. Dostupné z: <https://git-scm.com/book/cs/v1/%C3%9Avod-Spr%C3%A1va-verz%C3%AD>

GITBUCKET. GitBucket. In: *GitHub* [online]. 2017 [cit. 2017-07-15]. Dostupné z: <https://github.com/gitbucket/gitbucket>

GITBUCKET. License. In: *GitHub* [online]. 2017 [cit. 2017-07-15]. Dostupné z: <https://github.com/gitbucket/gitbucket/blob/master/LICENSE>

GITEA. License. In: *GitHub* [online]. 2017 [cit. 2017-07-15]. Dostupné z: <https://github.com/go-gitea/gitea/blob/master/LICENSE>

GITEA. Readme. In: *GitHub* [online]. 2017 [cit. 2017-07-15]. Dostupné z: <https://github.com/go-gitea/gitea/>

GITLAB. Commits. In: *GitHub* [online]. 2017 [cit. 2017-07-16]. Dostupné z: <https://github.com/gitlabhq/gitlabhq/graphs/commit-activity>

GITLAB. License. In: *GitHub* [online]. 2017 [cit. 2017-07-16]. Dostupné z: <https://github.com/gitlabhq/gitlabhq/blob/master/LICENSE>

GITLAB. Readme. In: *GitHub* [online]. 2017 [cit. 2017-07-16]. Dostupné z: <https://github.com/gitlabhq/gitlabhq>

GITLAB. GitLab Documentation. In: *GitLab* 2017 [cit. 2017-07-10]. Dostupné z: <https://docs.gitlab.com/ce/README.html>

GITLAB. Products. In: *GitLab* 2017 [cit. 2017-07-10]. Dostupné z: <https://about.gitlab.com/products/>

GITLAB. Enable or disable GitLab CI. In: *GitLab* 2017 [cit. 2017-07-10]. Dostupné z: https://docs.gitlab.com/ce/ci/enable_or_disable_ci.html

GITLAB. Dockerfile. In: *Docker Hub* 2017 [cit. 2017-08-27]. Dostupné z: <https://hub.docker.com/r/gitlab/gitlab-ce/~dockerfile/>

- GOCD. Commits. In: *GitHub* [online]. 2017 [cit. 2017-07-09]. Dostupné z: <https://github.com/gocd/gocd/graphs/commit-activity>
- GOCD. GoCD is converting to open source. In: *GoCD* [online]. 2014 [cit. 2017-07-09]. Dostupné z: <https://www.gocd.org/why-gocd/>
- GOCD. Why GoCD?. In: *GoCD* [online]. 2017 [cit. 2017-07-09]. Dostupné z: <https://www.gocd.org/why-gocd/>
- GOCD. GoCD – Docker. In: *GitHub* [online]. 2017 [cit. 2017-07-09]. Dostupné z: <https://github.com/gocd/gocd-docker>
- GOCD. Plugins. In: *GoCD* [online]. 2017 [cit. 2017-07-09]. Dostupné z: <https://www.gocd.org/plugins/>
- GOGS. Commits. In: *GitHub* [online]. 2017 [cit. 2017-07-15]. Dostupné z: <https://github.com/gogits/gogs/graphs/commit-activity>
- GOGS. License. In: *GitHub* [online]. 2017 [cit. 2017-07-15]. Dostupné z: <https://github.com/gogits/gogs/blob/master/LICENSE>
- GOGS. Readme. In: *GitHub* [online]. 2017 [cit. 2017-07-15]. Dostupné z: <https://github.com/gogits/gogs>
- GOSLING, James, JOY, Bill, STEELE, Guy, BRACHA, Gilad, BUCKLEY Alex. Field Modifiers. In: *The Java® Language Specification* [online]. 2015 [cit. 2017-05-17]. Dostupné z: <https://docs.oracle.com/javase/specs/jls/se8/html/jls-8.html#jls-8.3.1>
- GRUDL, David. *Proč opustit Subversion (SVN)* [online]. 2009 [cit. 2017-07-08]. Dostupné z: <https://phpfashion.com/proc-opustit-subversion-svn>
- HUDSON. *Hudson: Extensible continuous integration server*. Nedatováno [cit. 2017-07-07]. Dostupné z: <http://hudson-ci.org/>
- HUDSON. *Hudson 3.0 Plugin Central* [online]. 2017 [cit. 2017-07-07]. Dostupné z: <http://hudson-ci.org/PluginCentral/>
- HUJER, Martin. *Kontinuální integrace při vývoji webových aplikací v PHP* [online]. Praha, 2011 [cit. 2017-03-26]. Dostupné z: <http://theses.cz/id/qsfa0s/>. Bakalářská práce. Vysoká škola ekonomická v Praze. Vedoucí práce Jan Mittner.
- HORDĚJČUK, Vojtěch. Model-View-Controller. In: *VOHO* [online]. 2017 [cit. 2017-09-07]. Dostupné z: <http://voho.eu/wiki/model-view-controller/>
- HUMBLE, Charles. *Which CI Server Do You Use?* [online]. 2014 [cit. 2017-07-06]. Dostupné z <https://www.infoq.com/research/ci-server>

HUMBLE, Jez, FARLEY, David. *Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation*. Addison-Wesley, 2010. ISBN 978-0321601919.

HUMBLE, Jez. Continuous Delivery in Agile. In: *AgileAlliance - Agile2017* [online]. Zveřejněno 11. 07. 2017 [vid. 2017-08-11]. Dostupné z: <https://www.agilealliance.org/resources/videos/continuous-delivery-in-agile/>

HUMMEL, Benjamin. McCabe's Cyclomatic Complexity and Why We Don't Use It In: *Software Quality Blog* [online]. 2014 [cit. 2017-06-28]. Dostupné z: <https://www.cqse.eu/en/blog/mccabe-cyclomatic-complexity/>

HÜTTERMAN, Michael. *DevOps for Developers: Integrate Development and Operations the Agile Way*. Berkely(USA): Apress, 2012. ISBN 978-1-4302-4569-8.

HYKES, Solomon. The future of Linux Containers. In: *Youtube* [online]. Zveřejněno 21. 03. 2013 [vid. 2017-04-03]. Dostupné z: <https://www.youtube.com/watch?v=wW9CAH9nSLs>

ISO/IEC 25010:2011. 2011. *Systems and software engineering — Systems and software Quality Requirements and Evaluation (SQuaRE) — System and software quality models*. Geneve: ISO, 2011.

INTERNATIONAL ORGANIZATION FOR STANDADIZATION. *Standards catalogue* [online]. 2017 [cit. 2017-06-17]. Dostupné z: <https://www.iso.org/ics/35.080/x/>

IVANKOVIĆ, Marko. *Measuring Coverage at Google* [online]. 2014 [cit. 2017-05-27]. Dostupné z: <https://testing.googleblog.com/2014/07/measuring-coverage-at-google.html>

JACOCO. Mission. In: *JaCoCo documentation* [online]. 2017 [cit. 2017-07-29]. Dostupné z: <http://www.jacoco.org/jacoco/trunk/doc/mission.html>

JACOCO. Commits. In: *GitHub* [online]. 2017 [cit. 2017-07-29]. Dostupné z: <https://github.com/jacoco/jacoco/graphs/commit-activity>

JAVANCSS. JavaNCSS – A Source Measurement Suite for Java. In: *Wayback Machine: Internet Archive* [online]. 2017 [cit. 2017-07-30]. Dostupné z: <http://web.archive.org/web/20170430054456/www.kclee.de/clemens/java/javancss>

JAVANCSS. Commits. In: *GitHub* [online]. 2017 [cit. 2017-07-30]. Dostupné z: <https://github.com/codehaus/javancss/graphs/commit-activity>

JENKINS. Contributions to master, excluding merge commits. In: *GitHub* [online]. 2017 [cit. 2017-07-07]. Dostupné z: <https://github.com/jenkinsci/jenkins/graphs/contributors>

JENKINS. Jenkins. In: *GitHub* [online]. 2017 [cit. 2017-07-07]. Dostupné z: <https://github.com/jenkinsci/jenkins/blob/master/README.md>

- JENKINS. Docker official jenkins repo. In: *GitHub* [online]. 2017 [cit. 2017-07-07]. Dostupné z: <https://github.com/jenkinsci/docker>
- JENKINS. Cobertura Plugin. In: *Jenkins* [online]. 2017 [cit. 2017-07-30]. Dostupné z: <https://wiki.jenkins.io/display/JENKINS/Cobertura+Plugin>
- JENKINS. FindBugs Plugin. In: *Jenkins* [online]. 2017 [cit. 2017-07-30]. Dostupné z: <https://wiki.jenkins.io/display/JENKINS/FindBugs+Plugin>
- JENKINS. JaCoCo Plugin. In: *Jenkins* [online]. 2017 [cit. 2017-07-29]. Dostupné z: <https://wiki.jenkins.io/display/JENKINS/JaCoCo+Plugin>
- JENKINS. JavaNCSS Plugin. In: *Jenkins* [online]. 2017 [cit. 2017-07-30]. Dostupné z: <https://wiki.jenkins.io/display/JENKINS/PMD+Plugin>
- JENKINS. PMD Plugin. In: *Jenkins* [online]. 2017 [cit. 2017-07-29]. Dostupné z: <https://wiki.jenkins.io/display/JENKINS/PMD+Plugin>
- JENKINS. Jenkins Community Reaches Milestone: More Than 100,000 Active Installations Worldwide. In: *Jenkins* [online]. 2015 [cit. 2017-07-09]. Dostupné z: <https://www.cloudbees.com/press/jenkins-community-reaches-milestone-more-100000-active-installations-worldwide>
- JETBRAINS. Buy TeamCity. In: *JetBrains* [online]. 2017 [cit. 2017-07-06]. Dostupné z: <https://www.jetbrains.com/teamcity/buy/#license-type=new-license>
- JULINEK, Pavel. *Použití RUP pro malé SW projekty* [online]. Brno, 2008 [cit. 2017-10-14]. Dostupné z: <http://theses.cz/id/qjnsk2/>. Diplomová práce. Masarykova univerzita, Fakulta informatiky. Vedoucí práce RNDr. Jaroslav Ráček, Ph.D.
- JURENKA, Vladimír. *Virtualizace pomocí platformy Docker* [online]. Brno, 2015 [cit. 2017-03-26]. Dostupné z: http://is.muni.cz/th/430604/fi_m/. Diplomová práce. Masarykova univerzita, Fakulta informatiky. Vedoucí práce Filip Nguyen.
- KONDA, Madhusudhan. What's new in Java 8: Lambdas. In: *O'Reilly Media, Inc.* [online]. 2014 [cit. 2017-07-29]. Dostupné z: <https://www.oreilly.com/learning/whats-new-in-java-8-lambdas>
- KAWAGUCHI, Kohsuke, BECK, Daniel. *Plugins* [online]. 2017 [cit. 2017-07-07]. Dostupné z: <https://wiki.jenkins.io/display/JENKINS/Plugins>
- KAWAGUCHI, Kohsuke, GILMORE, Heidy. *Commercial Support* [online]. 2017 [cit. 2017-07-07]. Dostupné z: <https://wiki.jenkins.io/display/JENKINS/Commercial+Support>
- KOUNOIKE. Benchmark of GitBucket / Gogs / Gitea / GitLab on Raspberry Pi 3. In: *GitBucket News* [online]. 2017 [cit. 2017-07-07]. Dostupné z: <https://gitbucket.github.io/gitbucket-news/gitbucket/2017/03/29/benchmark-of-gitbucket.html>

KUZMINSKI, Marcin. System Requirements. In: *RhodeCode* [online]. 2014 [cit. 2017-07-16]. Dostupné z: <https://rhodecode.tenderapp.com/help/discussions/questions/4138-system-requirements>

MAZIN, Amaud. Docker Registry first steps. In: *Octo technology* [online]. 2017 [cit. 2017-08-30]. Dostupné z: <https://blog.octo.com/en/docker-registry-first-steps/>

MCCABE, Thomas. A Complexity Measure. In: *IEEE Transactions on Software Engineering*. Los Alamitos (USA): IEEE Computer Society, 1976, vol. 2, no. 4, s. 308-320. ISSN 0098-5589. Dostupné z: doi: 10.1109/TSE.1976.233837

MICROSOFT. Team Foundation Server Pricing. In: *VisualStudio* [online]. 2017 [cit. 2017-07-06]. Dostupné z: <https://www.visualstudio.com/team-services/tfs-pricing/>

MRÁZ, Martin. *Use of continuous integration in web application development* [online]. Brno, 2013 [cit. 2017-03-26]. Dostupné z: http://is.muni.cz/th/365431/fi_m/. Diplomová práce. Masarykova univerzita, Fakulta informatiky. Vedoucí práce Tomáš Pitner.

MITTAL, Nikhil. *Week of Continuous Intrusion Tools - Day 1 - Jenkins* [online]. 2015 [cit. 2017-07-07]. Dostupné z: <http://www.labofapenetrationtester.com/2015/11/week-of-continuous-intrusion-day-1.html>

O'SULLIVAN, Bryan. *Mercurial: the Definitive Guide*. Sebastopol: O'Reilly Media, Inc. 2009 ISBN 978-0-5965-5547-4

OKOSY, Michal. Thesis. In: *GitHub* [online]. 2017 [cit. 2017-09-11]. Dostupné z: <https://github.com/Pistolnik/thesis>

ORACLE. What is URL? In: *Oracle.com* [online]. 2017 [cit. 2017-10-14]. Dostupné z: <https://docs.oracle.com/javase/tutorial/networking/urls/definition.html>

PATTON, Ron. Testování softwaru. Praha: Computer Press, 2002. ISBN 80-7226-636-5.

PAVLÍČKOVÁ, Jarmila. *Model zralosti zdrojového kódu objektových aplikací* [online]. Praha, 2007 [cit. 2017-06-18]. Dostupné z: <https://vskp.vse.cz/eid/43818>. Disertační práce. Vysoká škola ekonomická v Praze. Vedoucí práce Ota Novotný.

PHP GROUP. What is PHP? In: *PHP.net* [online]. 2017 [cit. 2017-10-14]. Dostupné z: <http://php.net/manual/en/intro-what-is.php>

PMD. About PMD. In: *PMD* [online]. 2017 [cit. 2017-07-29]. Dostupné z: <https://pmd.github.io/#about>

PMD. Commits. In: *GitHub* [online]. 2017 [cit. 2017-07-29]. Dostupné z: <https://github.com/pmd/pmd/graphs/commit-activity>

PMD. License. In: *GitHub* [online]. 2017 [cit. 2017-07-29]. Dostupné z: <https://github.com/pmd/pmd/blob/master/LICENSE>

PMD. How to write a PMD rule. In: *PMD documentation* [online]. 2017 [cit. 2017-07-29]. Dostupné z: <https://pmd.github.io/pmd-5.8.1/customizing/howtowritearule.html>

RADIGAN, Dan. *Reaching true agility with continuous integration* [online]. 2017 [cit. 2017-04-01]. Dostupné z: <https://www.atlassian.com/agile/continuous-integration>

RADHAKRISHNAN, Kumar. *Top 10 Free GitHub Alternatives for Private Repositories server* [online]. 2014 [cit. 2017-07-15]. Dostupné z: <http://toppersworld.com/top-10-free-github-alternatives-for-private-repositories/>

RIGHTSCALE. Continuous Integration and Delivery in the Cloud – How RightScale Does It. In: *SlideShare.net* [online]. 2014 [cit. 2017-06-17]. Dostupné z: <https://www.slideshare.net/rightscale/rightscale-webinar-35810178>

RHODECODE. Release Notes. In: *RhodeCode Enterprise 4.8.0* [online]. Nedatováno [cit. 2017-07-16]. Dostupné z: <https://docs.rhodecode.com/RhodeCode-Enterprise/release-notes/release-notes.html>

ROUSER, Margaret. *Build server* [online]. Nedatováno [cit. 2017-07-06]. Dostupné z: <http://searchsoftwarequality.techtarget.com/definition/Build-Server>

SESHACHALA, Sudhi. Docker vs VMs. In: *DevOps.com: Where world meets DevOps* [online]. 2014 [cit. 2017-08-29]. Dostupné z: <https://devops.com/docker-vs-vm/>

SONARQUBE. Metric Definitions. In: *SonarQube Documentation* [online]. 2017 [cit. 2017-07-30]. Dostupné z: <https://docs.sonarqube.org/display/SONAR/Metric+Definitions>

SONARQUBE. Commits. In: *GitHub* [online]. 2017 [cit. 2017-07-30]. Dostupné z: <https://github.com/SonarSource/sonarqube/graphs/commit-activity>

SONARQUBE. Analyzing with SonarQube Scanner for Jenkins. In: *SonarQube Documentation* [online]. 2017 [cit. 2017-07-30]. Dostupné z: <https://docs.sonarqube.org/display/SCAN/Analyzing+with+SonarQube+Scanner+for+Jenkins>

SONARQUBE. License. In: *SonarQube* [online]. 2017 [cit. 2017-07-30]. Dostupné z: <https://www.sonarqube.org/downloads/license/>

SONARQUBE. Adding Coding Rules. In: *SonarQube Documentation* [online]. 2017 [cit. 2017-07-30]. Dostupné z: <https://docs.sonarqube.org/display/DEV/Adding+Coding+Rules>

SLANT. *What are the best mock frameworks for Java?* [online]. 2017 [cit. 2017-05-26]. Dostupné z: <https://www.slant.co/topics/259/~best-mock-frameworks-for-java>

SLANT. *What are self-hosted web-based Git repository managers?* [online]. 2017 [cit. 2017-05-26]. Dostupné z: <https://www.slant.co/topics/1440/~self-hosted-web-based-git-repository-managers>

SLANT. *What are the best continuous integration tools?* [online]. 2017 [cit. 2017-07-09]. Dostupné z: <https://www.slant.co/topics/799/~best-continuous-integration-tools>

SMART, John Ferguson. *Jenkins: The Definitive Guide: Continuous Integration for the Masses*: O'Reilly Media, Inc., 2011. ISBN 978-1-449-30535-2.

SMRŽ, Jaroslav. *Heuristické metody pro vícekritériální analýzu* [online]. 2006 [cit. 2017-07-05]. Dostupné z: http://autnt.fme.vutbr.cz/szz/2006/DP_Smrz.pdf, [cit. 10.1.2013]. Diplomová práce. Vysoké učení technické, Fakulta strojního inženýrství. Vedoucí práce Jindřich Klapka.

STUM, Michael. Why is git better than subversion? In: *Stack Overflow* [online]. 2009 [cit. 2017-07-08]. Dostupné z: <https://stackoverflow.com/questions/871/why-is-git-better-than-subversion>

STÅHL, Daniel; BOSCH, Jan. Experienced benefits of continuous integration in industry software product development: A case study. In: *The 12th iasted international conference on software engineering, (Innsbruck, Austria, 2013)*. 2013. p. 736-743.

STRÍŽ, Oliver. *Virtuální laboratorium pre vývoj Business Intelligence riešení* [online]. Praha, 2016 [cit. 2017-03-30]. Dostupné z: <http://theses.cz/id/xjne9x/>. Diplomová práce. Vysoká škola ekonomická v Praze. Vedoucí práce Jarmila Pavlíčková.

THOUGHTWORKS. Performance, support, reliability. In: *Thoughtworks* [online]. 2017 [cit. 2017-07-09]. Dostupné z: <https://www.thoughtworks.com/go/>

TOMÁŠEK, Patrik. *Virtualizace serverů a aplikací pomocí Docker* [online]. Praha, 2015 [cit. 2017-03-28]. Dostupné z: <http://theses.cz/id/x94czp/>. Bakalářská práce. Vysoká škola ekonomická v Praze. Vedoucí práce Luboš Pavlíček.

TRAVISCI. Built for every team. In: *Travis-CI* [online]. 2017 [cit. 2017-07-06]. Dostupné z: <https://travis-ci.com/plans>

TRZCIŃSKI, Kamil. GitLab runner plugins. In: *GitLab* [online]. 2017 [cit. 2017-07-10]. Dostupné z: <https://gitlab.com/gitlab-org/gitlab-ce/issues/14178>

UNCHALLENGER. GNU GPL. In: *Abclinuxu.cz* [online]. 2004 [cit. 2017-10-14]. Dostupné z: <http://www.abclinuxu.cz/slovník/gnu-gpl>

W3SCHOOLS. Introduction to XML. In: *W3Schools* [online]. 2017 [cit. 2017-10-14]. Dostupné z: https://www.w3schools.com/xml/xml_what_is.asp

W3SCHOOLS. SQL Tutorial. In: *W3Schools* [online]. 2017 [cit. 2017-10-14]. Dostupné z: <https://www.w3schools.com/sql/>

W3SCHOOLS. XML and XPath. In: *W3Schools* [online]. 2017 [cit. 2017-10-14]. Dostupné z: https://www.w3schools.com/xml/xml_xpath.asp