Česká zemědělská univerzita v Praze
Provozně-ekonomická fakulta
Katedra informačního inženýrství

Diplomová práce
Návrh CI/CD pro webovou aplikaci

Bc. Otto Klapka

ČESKÁ ZEMĚDĚLSKÁ UNIVERZITA V PRAZE

Provozně ekonomická fakulta

ZADÁNÍ DIPLOMOVÉ PRÁCE

Bc. Otto Klapka

Systémové inženýrství a informatika
Informatika

Název práce

Návrh CI/CD pro webovou aplikaci

Název anglicky

CI/CD design for a web application

Cíle práce

Cílem práce je navrhnout a implementovat průběžnou integraci (continuous integration) a její plán (continuous delivery plan) pro malou webovou aplikaci. V práci bude navržen řetězec nástrojů, který bude schopen automaticky otestovat a nasazovat aplikaci do produkčního prostředí.

Metodika

V první části bude analýza vývojového cyklu malé webové aplikace a rešerše aktuálních technologií v oblasti DevOps. V další praktické části bude po zvolení vhodných nástrojů navržen a implementován CI a CIP pro malou webovou aplikaci. Budou dodržovány všechny potřebné standardy pro dokumentaci softwarového projektování.

Doporučený rozsah práce

60 – 100 stran

Klíčová slova

DevOps; Docker; Jenkins; CI/CD; Git

Doporučené zdroje informací

PATHANIA, Nikhil. Learning Continuous Integration with Jenkins: a Beginners Guide to Implementing Continuous Integration and Continuous Delivery Using Jenkins 2. Packt Publishing, 2017. ISBN 978-1-78528-483-0.

SHARMA, Sanjeev. The DevOps Adoption Playbook: a Guide to Adopting DevOps in a Multi-Speed IT Enterprise. John Wiley & Sons, 2017. ISBN: 978-1-119-30874-4.

SONI, Mitesh, DEVOPS BOOTCAMP. PACKT Publishing Limited, 2017. ISBN 978-1-78728-596-5.

VERONA, Joakim, LEARNING DEVOPS: Continuously Deliver Better Software. PACKT Publishing Limited, 2016. ISBN 978-1-78712-661-9.

Předběžný termín obhajoby

2020/21 LS – PEF

Vedoucí práce

doc. Ing. Vojtěch Merunka, Ph.D.

Garantující pracoviště

Katedra informačního inženýrství

Elektronicky schváleno dne 7. 3. 2021

Ing. Martin Pelikán, Ph.D.

Vedoucí katedry

Elektronicky schváleno dne 7. 3. 2021

Ing. Martin Pelikán, Ph.D.

Děkan

V Praze dne 09. 03. 2021

Čestné prohlášení

Prohlašuji, že svou diplomovou práci „Návrh CI/CD pro webovou aplikaci“ jsem vypracoval samostatně pod vedením vedoucího diplomové práce a s použitím odborné literatury a dalších informačních zdrojů, které jsou citovány v práci a uvedeny v seznamu použitých zdrojů na konci práce. Jako autor uvedené diplomové práce dále prohlašuji, že jsem v souvislosti s jejím vytvořením neporušil autorská práva třetích osob.

V Praze dne 28. března 2021 _____

Poděkování

Rád bych touto cestou poděkoval doc. Ing. Vojtěchu Merunkovi, Ph.D. za cenné rady a připomínky, které mi poskytl v průběhu zpracování této práce. Také děkuji své rodině a blízkým za podporu při studiu.

Návrh CI/CD pro webovou aplikaci

Abstrakt

Tato diplomová práce se zabývá návrhem a implementací průběžné integrace a následně průběžného nasazení pro menší webovou aplikaci. Pro potřeby práce byl vytvořen projekt webové aplikace. Tento projekt představuje webový server vystavující API pro správu poznámek. Aplikace je napsaná v jazyce TypeScript a obsahuje automatické testy ověřující její funkcionalitu.

Pro produkční spuštění aplikace je třeba nejprve zdrojový kód aplikace transpilovat z jazyka TypeScript do JavaScript. Následuje ověření funkčnosti pomocí automatizovaných testů a zabalení do přenositelného virtualizačního kontejneru, který je opatřen štítkem verze. Tento kontejner je třeba nahrát do verzovacího ropozitáře. Poslední částí je připojení na cílový server pomocí SSH, odkud je stažena poslední verze kontejneru s aplikací, který je následně spuštěn.

Jako cílový server bude použit VPS. Tento VPS je Linuxový server s operačním systémem Ubuntu 18.04.5 LTS a 2GB RAM.

Navržené řešení bude automatizovat veškeré kroky včetně přenesení kontejneru s aplikací na cílový server a jeho spuštění.

Klíčová slova: DevOps, Docker, Jenkins, CI/CD, Git, GitLab

CI/CD design for a web application

Abstract

This diploma thesis deals with the design and implementation of continuous integration and continuous deployment for a small web application. A web application project was created for the purposes of the work. This project is a web server that provides a note management API. The application source code is written in TypeScript and contains automatic tests to verify its functionality.

To run the application in production, the application source code needs to be transpiled from TypeScript into JavaScript. This is followed by verification of functionality using automated tests and packaging in a portable virtualization container that has a version label. This container must be uploaded to the versioning repository. The last part is the connection to the target server using SSH, from where the latest version of the container with the application is downloaded, which is then launched.

The VPS will be used as the target server. This VPS is a Linux server with Ubuntu 18.04.5 LTS and 2GB RAM.

The proposed solution will automate all steps, including transferring the application container to the target server and launching it.

Key words: DevOps, Docker, Jenkins, CI/CD, Git, GitLab

Obsah

Seznam tabulek

Seznam obrázků

Seznam kódů

1 Úvod

S rozrůstající se oblibou agilního způsobu vývoje software sílí tlak na zrychlování doručovacích cyklů. Mnozí tedy nahrazují jejich starší monolitické vývojové modely. Bohužel některé části zastaralého vývojového modelu brzdí týmy od implementace praktik, jako průběžná integrace či průběžné doručení, a využití jejich přínosu v agilním modelu vývoje. Cílem praktiky průběžné integrace a doručení je vytvořit konstantní cyklus malých dodávek změn do produkce. Tímto je zrychlen cyklus vydávání nových verzí software, snížení nákladů a rizik plynoucích z vývoje.

Průběžná integrace je praktika při vývoji software, v níž je každá dílčí změna ve zdrojovém kódu aplikace otestována v kontextu celku, jakmile programátor uloží svoji práci. Cílem průběžné integrace je předcházet chybám vznikajících z integrace kódu programátorů nebo je odhalit co nejdříve ve vývojovém procesu.

Průběžné doručení je navazující praktikou na průběžnou integraci. Doručení je proces předání nového sestavení aplikace k rukám zákazníka. Praktika průběžného doručení se snaží o minimalizování rizik plynoucích z doručování software. Proces doručování aktualizací je notoricky komplikovaný a časově náročný. Praktika průběžného doručení snižuje rizika a náročnost tohoto procesu pomocí ověření, že každá nově přidaná funkcionalita je připravena k vydání. Ověřování umožňuje dodávat aktualizace rychleji po menších částech. Tímto způsobem je proces vydání změn méně dramatický a je možné ho provádět kdykoliv na vyžádání, jakmile je kód připraven.

2 Cíl práce a metodika

2.1 Cíl práce

Cílem práce je navrhnout a implementovat automatizovaný proces průběžné integrace a doručení. Aplikací pro kterou bude proces navržen je jednoduchý server zprostředkující API pro správu poznámek. Integrační proces musí zajistit správné běhové prostředí s NodeJS. Spustit překlad zdrojového kódu z jazyka TypeScript do jazyka JavaScript, instalovat závislosti a spustit automatické testy pro ověření funkčnosti aplikace. Následně proces doručení musí zajistit zabalení sestavené aplikace do virtuálního kontejneru. Prostředí kontejneru musí obsahovat všechny nezbytnosti pro spuštění a běh aplikace. Následně bude tento kontejner odeslán do repozitáře k verzování. Nakonec bude proces doručení zajišťovat připojení na cílový server, na který bude stažen kontejner z repozitáře a spuštěn.

2.2 Metodika práce

Práce bude rozdělena na dvě hlavní části. V první části proběhne teoretická rešerše odborné literatury týkající se podstatných témat práce.

Budou popsány aktuální postupy při vývoji software, proces průběžné integrace a jeho automatizace. Dále proces průběžného doručení, testování kódu a rešerše nástrojů pro automatizaci průběžné integrace a průběžného doručení.

V praktické části bude po zvolení vhodných nástrojů navržen a implementován CI a CD pro malou webovou aplikaci. Prostup bude detailně popsán a závěrem proběhne zhodnocení řešení.

I. TEORETICKÁ VÝCHODISKA

3 Agilní model vývoje software

Agile je soubor metod a pohledů na věc, které pomáhají týmu přemýšlet nad projektem efektivněji, pracovat efektivněji a rozhodovat se lépe.

Tyto metody se týkají všech částí tradičního vývoje software. Zahrnuje management projektu, vývoj software a jeho architektury včetně prostředí, ve kterém je vyvíjen. Každá z těchto metod sestává z praktik, které jsou optimalizovány pro jejich co nejjednodušší implementaci.

Dále dle (Stellman et al., 2015) je *agile* také myšlenkový model. Správné přemýšlení o projektu dokáže tvořit znatelný rozdíl v tom, jak tým dokáže tyto praktiky využít. Tento model pomáhá členům týmu sdílet informace, díky čemuž jsou vynášeny informovaná a společná rozhodnutí. V tomto se agilní metodika liší od tradičního modelu, kde jsou rozhodnutí vynášena managerem projektu. Agilní způsob přemýšlení má za cíl otevřít proces plánování, návrhu a vypracování pro celý tým. Agilní tým praktikuje způsob, kde všichni sdílí společné informace a každý člen má hlas v tom, jak bude postup pokračovat.

Agilní metodika vývoje software je pojem, ze kterého vychází mnoho dalších odnoží. Extrémní programování či SCRUM mají základy právě v *agile*. Agilní způsob vývoje byl definován v *agile manifesto*. Ten byl sepsán softwarovými vývojáři (a softwarovým testerem) aby vyřešili problémy, kterým čelili v praxi. Agilní způsob vývoje software se řídí 12 principy které popsané na stránkách (agilealliance, 2020).

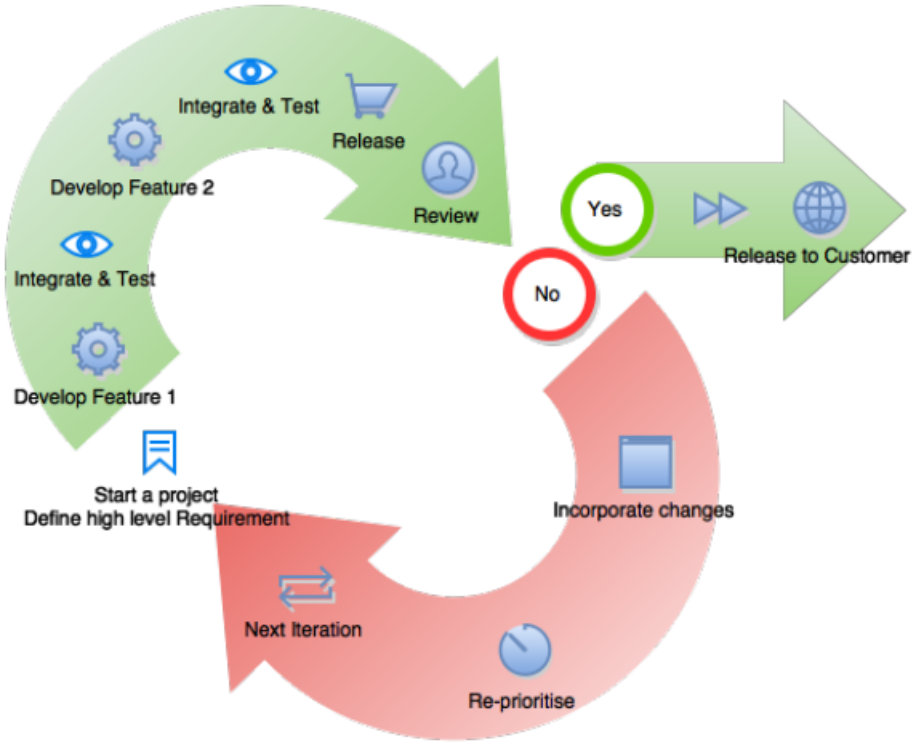
1. Nejvyšší prioritou je uspokojení zákazníka, skrze brzké a trvající dodávky kvalitního software.
2. Požadavky na změnu v zadání jsou vítané i v pozdějších fázích vývoje. Agilní procesy využívají změn ve svůj prospěch.
3. Funkční software je dodáván častěji v rozmezí několika týdnů až měsíců s preferencí menších časových úseků.
4. Nejlepší způsob sdílení informací mezi členy týmu je konverzace tváří v tvář.
5. Vývojáři spolupracují s byznysovým týmem denně na součástech projektu.
6. Motivace vývojářů je pro projekt stěžejní. Vývojářům je zajištěno pohodlné prostředí pro práci včetně všech jejich potřeb. Mají plnou důvěru, že bude produkt dodán zákazníkovi v čas.
7. Fungující software je základním měřítkem pokroku.
8. Agilní proces preferuje udržitelný vývoj. Sponzoři, vývojáři a uživatelé by měli být schopni udržet konzistentní tempo po dlouhý časový úsek.
9. Pozornost se věnuje dobrému technickému provedení. Správný návrh podporuje agilní proces.
10. Jednoduchost a umění pracovat s co nejmenším vypětím je klíčové.

11. Sebeorganizované týmy produkují nejlepší návrhy architektury a požadavky.

12. V pravidelných intervalech tým vymýšlí, jak se stát efektivnější a jak vylepšit své fungování. Chování je dle těchto návrhů upravováno.

Stránky *agilealiance.org* se zabývají propagací agilního přístupu a informacím o jeho implementaci. Autoři popisují, že agilní přístup je zaměřený na přizpůsobení měnícím se podmínkám. Zakládá se na faktu, že změny v průběhu projektu jsou nevyhnutelné a je nutné se jim přizpůsobit. Spokojenost zákazníka je na prvním místě. Jakékoliv změny návrhu jsou vítány i v pozdějších stupních vývoje. Produkt je zákazníkovi dodáván po malých částech v iteracích. V každé iteraci je vytvořena funkcionalita a předvedena zákazníkovi ke schválení. Pokud je funkcionalita zákazníkem schválena, bude nasazena do produkčního prostředí. Pokud zákazník funkcionalitu odmítne, bude zařazena do další iterace a znovu přepracována. Výhodou tohoto postupu je, že přepracování jedné malé části je mnohem lacinější, než zásah do již hotové aplikace.

Velký důraz metodika klade na týmovou práci. Každý tým je zodpovědný za určitou fázi produktu a týmy jsou sebeřízené. Role manažera ustupuje, pokud se týmu daří doručit zákazníkovi včas požadovanou funkcionalitu. Pokud tým tento úkol nezvládá, manažer vstupuje do procesu, aby týmu pomohl vypořádat se s danou situací. Hlavním úkolem manažera je zaručit, že tým bude mít vždy potřebné množství zkušeností a znalostí. (agilealliance, 2020)



Obrázek 1: Schéma agilního způsobu vývoje software (Pathania, 2017)

(Pathania, 2017) uvádí jako další klad agilní metodiky fakt, že během vývojového cyklu jsou využity všechny týmy současně. Týmy spolupracují na mnoha úrovních

jako design, plánování, vývoj nebo testování. Výsledkem takového procesu je, že žádný tým nezůstane nevyužitý jako například ve vodopádovém způsobu vývoje.

3.1 SCRUM

Agilní metodika SCRUM je heuristická. Je založena na průběžném učení a přizpůsobování měnícím se faktorům. SCRUM připouští fakt, že tým nikdy nemůže znát všechny potřebné detaily na začátku projektu a bude se vyvíjet a učit v průběhu řešení. Implementuje strukturu, která pomáhá týmu se přirozeně adaptovat na měnící se nároky klienta. Pomocí změny priorit a krátkých dodávkových cyklů zabudovaných do metodiky se tým může průběžně učit a vyvíjet. (Drumond, 2020)

Podle (Pathania, 2017), který považuje metodiku SCRUM široce využívanou je tato metodika založená na sebeorganizovaných týmech. Scrum tým je multifunkční jednotka, což znamená, že kdokoli může implementovat funkcionalitu od plánování po doručení. Model neobsahuje manažera, který by explicitně definoval funkci pracovníka nebo jak má daný problém vyřešit.

Vývojářský tým vybírá pořadí implementace funkcionalit. Takový postup je volen protože vývojáři nejlépe rozumí fungování aplikace. Jimi volený postup implementace bude proto nejlepší.

Implementaci SCRUM je možné rozdělit podle (Stellman et al., 2015) do několika následujících bodů:

- Každý sprint začíná plánováním sprintu - *sprint planning* kterého se zúčastňuje Scrum Master, Product Owner a zbytek týmu. Úkol pro pracovníka v roli Product Owner je připravit před plánováním prioritizovaný seznam úkolů, které klient nakoupil - *product backlog*. V první části schůzky Product Owner a tým dohodnou, které úkoly budou na konci sprintu předány zákazníkovi. Rozhoudujícími faktory jsou hodnota úkolů a náročnost přidělená týmem programátorů. V druhé části plánovací schůzky tým rozděluje požadované úkoly od zákazníka na jednotlivé úkony, které je třeba splnit k vytvoření funkční požadované funkcionality. Na konci plánovací schůzky se tyto úkony stanou sprintovým backlogem.

- Tým se schází každé ráno na ranních SCRUM schůzkách. Všichni členové týmu včetně Product Owner a Scrum Master se musí zúčastnit, zainteresovaní vedoucí pracovníci mohou také, zůstanou však tichými pozorovateli. Schůzka je časově ohraničena na 15 minut, takže všichni členové týmu se musí ukázat včas. Každý člen týmu zodpoví tři otázky:

  – „Co jsem dělal předchozí pracovní den?“
  – „Co budu dělat následující pracovní den?“
  – „Existují nějaké překážky, které mi brání v práci?“

  Všichni členové musí být struční, pokud některá z otázek vyžaduje diskusi, je naplánována následující schůzka zúčastněných stran.

- Každý sprint je časově ohraničený. Mnoho týmů využívá délku 30 kalendářních dnů. Tato délka se může lišit a některé týmy mohou volit dvoutýdenní délku sprintu. Při volbě délky sprintů hraje hlavní roli povaha produktu. Během sprintu tým programátorů přetaví úkoly ve sprint backlog do fungujícího produktu. Pokud některý člen týmu během sprintu zjistí, že byla práce špatně časově odhadnuta a nebude možné ji tak doručit včas, nebo naopak je práce málo a je možné přidat některý další úkol, je třeba informovat pracovníka v roli Product Manager, který je v kontaktu se zákazníkem a je oprávněn dohodnout změnu jeho očekávání. Pokud tým zjistí, že během sprintu nebude dostatek práce pro všechny, může rozšířit sprint backlog. Tým musí udržovat sprint backlog aktuální a veřejně dostupný. Ve velice neobvyklých případech a extrémních podmínkách Product Owner může ukončit sprint dříve a vyhlásit nové plánování sprintu. Takové přerušení je však extrémně neobvyklé a je třeba si uvědomit jeho dopad na důvěru zákazníka.

- Na konci každého sprintu provádí tým rekapitulaci, kde je předveden fungující software zákazníkovi a vedoucím pracovníkům. Demo může obsahovat pouze fungující části, které byly vytvořeny, otestovány a uznány pracovníkem v roli Product Owner za kompletní. Tým zodpovídá otázky zákazníka a vedení, kteří jsou poté požádáni o zpětnou vazbu. Pokud jsou vyžadovány změny, jsou zařazeny do vývoje na další schůzce k plánování sprintu. Product Owner přidá změny do seznamu product backlog a pokud jsou vysoké priority, budou vyrobeny v dalším sprintu.

- Po každém sprintu tým pořádá schůzku retrospektivy sprintu. Zúčastní se tým, Scrum Master a nepovinně Product Owner. Každý pracovník zodpovídá dvě otázky:

  - „Co se podařilo v minulém sprintu?“
  - „Co by šlo zlepšit do příštího sprintu?“

  Scrum Master si dělá poznámky o vylepšeních a mohou být přidány do seznamu sprint backlog jako ostatní požadavky.

3.2 Extrémní programování

Další populární agilní metodikou je extrémní programování. Tato metodika je založena na pěti hodnotách. Těmi jsou komunikace, jednoduchost, zpětná vazba, odhodlání a respekt. Metodika extrémního programování je velmi specifická v popisování správných vývojových postupů.

Komunikace je jednou ze zásadních hodnot. Pracovníci s propojeným zaměřením tvoří jeden tým a jejich informovanost je klíčová. Většina vývojářů se shodne, že komunikace tváří v tvář je nejužitečnější. Preferován je proto společný prostor, který nebrání fyzické komunikaci. Soukromí a klid na práci by však měly být v dosahu možností každého pracovníka.

Další důležitou praktikou je párové programování. Tato myšlenka je založena na principu, kdy je produkční kód vyvíjen dvěma programátory u jednoho stroje.

Tato praktika v praxi usnadňuje vývoj rychlejší odezvou a minimalizuje stavy zaseknutí.

Zákazník formuje požadavky do takzvaných uživatelských příběhů - *user stories*. Takový příběh popisuje chtěnou funkcionalitu nebo případ použití. Tyto příběhy jsou na začátku každého týdne hodnoceny vývojáři a je rozebírán postup, který bude zvolen k jejich dokončení. Cílem je na konci týdne vytvořit a otestovat funkcionalitu, kterou je možné prezentovat zákazníkovi. Každý týdenní cyklus by měl obsahovat některé položky s nízkou prioritou. Pokud se vývoj dostane do skluzu, tyto položky nízké priority je možné odhodit a vytvořit tak časový polštář.

Funkcionality vytvářené v týdenních cyklech by na konci čtvrtletí měly utvořit celý uživatelský příběh požadovaný klientem. Tyto cykly poskytují širší pohled na vývoj a předcházejí situaci, kdy pro stromy není vidět les. Protože se jedná o poměrně dlouhou periodu, předpokládá se, že zadání se bude měnit. Týdenní cyklus zajišťuje v tomto ohledu pružnost a přizpůsobitelnost doručováním menších funkcionalit, které je levnější přepracovat.

Autoři extrémního programování doporučují tzv. desetiminutové sestavení - *ten minute build*. Hlavní myšlenkou je sestavení aplikace a její otestování (za použití automatických testů) stihnout do deseti minut. Takové doporučení bylo zvoleno proto, že při delším sestavení a otestování je pravděpodobné, že tým nebude spouštět toto sestavení na pravidelné bázi. Tím je způsobena větší prodleva mezi odhalením chyb. Tento požadavek nutí vývojáře proces sestavení automatizovat a tím dodržovat pravidelnost. (Beck, 2004)

4 Testování kódu

V roce 2009 na XP Day konferenci v Londýně, Mark Striebeck uvedl ceny opravy chyb v kódu v závislosti na době odhalení. Podle společnosti Google, oprava chyby odhalené okamžitě po tom, co ji programátor vytvořil stojí $5. Pokud se chyba odhalí po plném sestavení aplikace, cena opravy je $50. Pokud chybu odhalí až integrační testy, cena na opravu se zvyšuje na $500. Oprava chyby odhalené systémovými testy potom stojí $5000.

Testování hraje důležitou roli ve vývoji software. Má veliký vliv na kvalitu výsledného produktu a rychlost vývoje. Neustálým opakováním cyklu test - nalezení chyb - oprava během vývoje se kvalita výsledného produktu zvyšuje. Testování je tedy proces posuzování kvality a vylepšení produktu. Obecně posuzování kvality může být rozděleno do dvou širokých kategorií, statické a dynamické analyzování.

- **Statická analýza:** Je založena na zkoumání různých typů dokumentů. Především jde o dokumenty požadavků, softwarových modelů, plánů a zdrojového kódu. Tradiční statická analýza zahrnuje *code review*, inspekci, krokování, analýzu algoritmu a dokazování správnosti postupu. Spouštění samotné aplikace se zde neprovádí. Namísto toho rozebírá kód a jeho možné chování během spuštění. Optimalizace kompileru je standardní statická analýza.

- **Dynamická analýza:** Dynamická analýza software zahrnuje spuštění programu za účelem nalezení případných chyb. Vlastnosti software jako výkon a chování jsou také podrobeny inspekci. Program je mnohokrát spouštěn s běžnými i pečlivě zvolenými argumenty pro otestování jeho chování. Často je množství argumentů pro aplikaci velice rozsáhlé. Proto jsou zvoleny pouze některé reprezentativní kroky k praktickému posouzení stavu kvality aplikace. Pečlivé zvolení těchto kroků je klíčové k posouzení kvality celého systému.

Prováděním statické i dynamické analýzy, se vývojáři snaží odhalit co nejvíce chyb, které tak mohou být opraveny v dřívějších fázích vývoje software. Statická a dynamická analýza se z podstaty doplňují a je tedy třeba provádět oba druhy pro dosažení co největší efektivnosti.

Dvěma dalšími koncepty které uvádí (Kshirasagar et al., 2008) využívané při testováni software jsou verifikace a validace. Oba koncepty jsou abstraktní a jsou realizovány několika konkrétními aktivitami.

- **Verifikace:** Tento typ aktivity v hodnocení systému určuje, zda produkt v dané vývojové fázi naplňuje požadavky zadané v předchozí části. Aktivity, které kontrolují správnost vývojové fáze, se nazývají verifikační aktivity.

- **Validace:** Tyto aktivity určují, zda produkt naplňuje podmínky za kterých má být používán. Validační aktivity míří na potvrzení zda aplikace splňuje požadavky zákazníka. Jinými slovy, validační aktivity se zaměřují na finální produkt, který je testován z pohledu zákazníka. Validace zjistí, zda produkt splňuje celkové očekávání uživatele.

Provádění validačních aktivit v pozdějších fázích vývoje software je riskantní protože může vést k navýšení nákladů na vývoj. Validační aktivity mohou být spouštěny v brzkých fázích vývoje. Příkladem takových brzkých spouštění lze nalézt v agilní metodice vývoje software extrémním programování. Během extrémního programování zákazník úzce spolupracuje s vývojářským týmem a akceptační testy jsou prováděny každou krátkou iteraci.

Verifikační proces zajistí soulad implementační fáze softwarového vývoje s jeho specifikací. Validace potom zajistí, že výsledná aplikace vyhovuje požadavkům zákazníka.

4.1 Kvalita software

Otázka „Co je to kvalita software?“ evokuje množství odpovědí. Kvalita je komplexní pojem a znamená různé věci pro různé lidi a je velmi závislá na kontextu. V článku pro časopis IEEE Software z roku 1996 (Kitchenham et al., 1996) uvádí různé pohledy na kvalitu software. Popisuje 5 pohledů na kvalitu software:

- **Transcendentní pohled:** Definuje kvalitu jako něco, co je rozeznatelné, ale je těžko popsatelné. Transcendentní pohled není specifický pro kvalitu software, ale byl použit v dalších komplexních oblastech každodenního života.
- **Pohled uživatele:** Tento pohled vnímá kvalitu podle vhodnosti k využití v určitém směru. Při posuzování kvality produktu v tomto směru je třeba si položit otázku: „Splňuje produkt potřeby uživatele a jeho očekávání?“
- **Pohled výroby:** V tomto případě je kvalita posuzována dle shody se specifikacemi. Úroveň kvality produktu je posuzována dle rozsahu shody s návrhem.
- **Produktový pohled:** V tomto pohledu je kvalita odvozena od inheritních charakteristik produktu. Inheritní znaky produktu, které jsou vnitřní kvalitou, definují jeho externí kvalitu.
- **Pohled založený na ceně:** Z tohoto pohledu je kvalita určena cenou produktu, kterou je zákazník ochoten zaplatit.

Mnoho modelů kvality software bylo vytvořeno k definování jeho kvality. Nejdůležitějšími jsou ISO 9126 a CMM. Model kvality ISO 9126 byl stanoven skupinou expertů International Organization for Standardization (ISO). Tento dokument definuje 6 kategorií charakteristiky kvality: funkcionalita, spolehlivost, použitelnost, efektivnost, udržitelnost a přenositelnost. CMM byl vytvořen Software Engineering Institute (SEI) z Carnegie Mellon University. V tomto modelu je software hodnocen na stupnici 1 - 5, známé jako úrovně 1 až 5. Úroveň 1 je nazývána inicializace a úroveň 5 optimalizován. Úroveň 5 je nejvyšším stupněm procesu vývoje aplikace.

Na poli testování software jsou dva dobře známé modely procesu. Jmenovitě *test process improvement* (TPI) model a *test maturity model* (TMM). Tyto dva modely umožňují organizacím identifikovat momentální stav testování jejich software a další logickou oblast pro posun. Pomáhají také určit plán akcí k vylepšení tohoto procesu. (Kshirasagar et al., 2008)

4.2 Testovací aktivity

Pro otestování programu je třeba aby testující pracovník provedl několik úkonů:

- **Identifikace předmětu testování:** Prvním úkonem je identifikace předmětu k testování. Předmět testování identifikuje záměr nebo důvod k sestrojení jednoho či více testovacích případů k otestování, že jsou plně implementovány programem. Jasně daný předmět musí být součástí každého testovacího případu.
- **Zvolení vstupů:** Druhou aktivitou je zvolení vstupů pro testování. Zvolení vstupů může být založeno na požadavcích, specifikaci, zdrojovém kódu nebo implicitních předpokladech. Testovací vstupy jsou vybrány na základě testovacího předmětu.
- **Definování očekávaných výsledků:** Třetí aktivitou je definování očekávaných výstupů z programu. Tyto výsledky je nejčastěji možné vyvodit z porozumění testovacímu záměru na vyšší úrovni nebo ze specifikace programu.
- **Nastavení prostředí pro spuštění:** Čtvrtým krokem je příprava prostředí pro běh programu. V tomto kroku musí být splněny všechny požadavky programu na spuštění. Takovými požadavky může být dostupnost připojení k internetu, připojení externích systémů, dostupnost periferií nebo správná data v databázi.
- **Spuštění programu:** V tomto kroku je program spuštěn v připraveném prostředí se zvolenými vstupy. K provedení testovaného případu mohly vstupy programu být vloženy z různých fyzických míst nebo v různém čase.
- **Analýza testovacího případu:** V poslední fázi testování je analyzován výstup z programu po zadání zvolených vstupů. Hlavní činností je kontrola výstupu a její shody s předpřipravenými očekávanými výstupy. Komplexnost analýzy je závislá na komplexnosti dat k posouzení. Tento výsledek může být jednoduché číslo nebo obrázek či video. Na konci analýzy je vynesen verdikt nad programem.

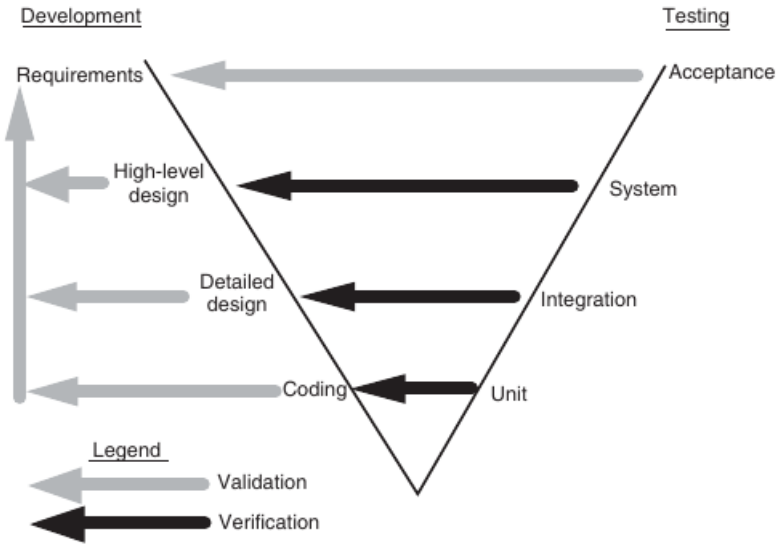
  Tři hlavní typy verdiktu jsou přijetí, chyba a neposouditelné. V některých případech není možné vynést jasný verdikt o přijetí či chybě. Například pokud během provádění vyprší časový limit pro odpověď na serveru, není možné posoudit zda došlo k chybě. V takových případech je vynesen verdikt neposouditelné a je třeba provést další testování.

  Po dokončení analýzy je sepsáno hlášení o testu. Hlavním účelem sepsání hlášení je napravení chyby pokud byla nějaká odhalena. Hlavní body hlášení jsou vysvětlení, způsob replikace chyby, popis chyby a odkaz na testovací případ, jeho vstupy a výsledky programu. (Kshirasagar et al., 2008)

4.3 Úrovně testování

Testování je prováděno na různých úrovních. Na těchto úrovních jsou zapojeny části nebo kompletní aplikace během svého životního cyklu. Softwarový produkt

prochází skrze čtyři úrovně testování během vývoje. Tyto čtyři úrovně jsou známy jako jednotkové testy, integrační a systémové testování a testování přijetí. První tři jsou prováděny různými týmy při vývoji aplikace, kdežto testy přijetí jsou prováděny zákazníkem. Čtyři úrovně testování jsou ilustrovány na obrázku 2 a tvoří písmeno V.



Obrázek 2: Vývojové a testovací fáze ve V modelu (Kshirasagar et al., 2008)

V jednotkových testech, programátor testuje jednotlivé jednotky programu. Těmi mohou být procedury, funkce, metody nebo třídy v izolovaném prostředí.

Po ověření, že tyto moduly vykazují správné jednotlivé chování, jsou seskupeny do většího konstruktu. Následuje integrační testování ve kterém je ověřena integrita těchto částí. Cílem integračních testů je zajistit solidní stabilitu větších částí systému.

Následuje testování na systémové úrovni. Tato úroveň testů zahrnuje množství způsobů testování. Těmi mohou být bezpečnostní testy, funkcionální testy, testy stability nebo zátěžové testy. Systémové testování je kritický milník při vývoji software kvůli často napjatým dodávkovým termínům. V této fázi se ověří zda aplikace funguje, splňuje požadavky a předchozí opravy nepřinesly další navazující chyby. Systémové testování zahrnuje množství aktivit. Je třeba vytvořit plán testování, připravit prostředí pro spuštění testu, spustit testy a kontrolovat jejich průběh.

Dalším typem jsou regresní testy. Tyto testy jsou prováděny kdykoliv se změní komponenta systému. Hlavním cílem regresních testů je ověření, že změna v kódu nezpůsobila další chyby. Regresní testy nejsou samostatnou úrovní v testování. Namísto toho se uvádí jako podfáze v jednotkovém, integračním a systémovém testování. V regresním testování se nevytváří nové testy. Namísto toho jsou použity stávající s upravenou prioritou k ověření, že v nové verzi aplikace nedošlo k chybám. Jedná se o finančně náročnou proceduru, je proto vhodné zvolit již stávající testy. Důležité je zvolit správné testy aby byla případná chyba odhalena.

Po testování na systémové úrovni je produkt doručen zákazníkovi. Zákazník produkt otestuje za použití svých vlastních testů. Tyto testy se často nazývají testy přijetí. Předmětem testů přijetí je kontrola kvality, na rozdíl od systémových testů,

kde se kontroluje přítomnost chyb. Klíčové měřítko v testech přijetí je naplnění očekávání uživatele. Jsou dva typy testů přijetí. Prvním je uživatelský akceptační test - *User Acceptance Test* (dále jen UAT) a byznysový akceptační test - *Business Acceptance Test* (dále jen BAT). UAT je prováděn zákazníkem pro ujištění, že systém naplňuje podepsané požadavky před zaplacením. Na druhou stranu BAT je prováděn na straně dodavatele. Jeho myšlenkou je připravení se na UAT a kontrola, jestli jím na straně zákazníka produkt úspěšně projde. (Kshirasagar et al., 2008)

4.4 Testy vedený vývoj

Test-first nebo *test driven development* je metodologií vývoje software, kdy je nejprve sestrojen test. Tento test může později představovat zákazníka. Vytváření testů je v tomto případě jakousi plánovací aktivitou. Přetavením scénáře do spustitelného automatického testu znamená vymezení přesného cíle k dosažení. Spuštěním takového testu a jeho selhání programátora vede k vyprodukování přesně tolika kódu, který je třeba aby zajistil úspěšný průchod testem a ne více. Vyjádření požadavků ve formě testů je velmi silný nástroj. Následování jednoduchého principu, napsání pouze tolika kódu aby test úspěšně proběhl, udržuje logiku programu velmi čistou. Zaručuje, že nevznikne takzvaná nechtěná složitost - *accidental complexity*. Nechtěná složitost vzniká špatným plánováním nebo přehnanou snahou programátora a vede k nepřehlednosti kódu a jeho údržby.

Předcházet nechtěné složitosti kódu lze kombinací testů, které definují chyby a nebo chybějící funkcionalitu a právě tolika kódu, aby uspokojil definované nároky. Spolu s refaktorováním kódu, aby následoval tato pravidla, jsou účinným nástrojem pro potlačení nechtěné složitosti v software. Samozřejmě vždy záleží na zkušenostech programátora a jeho schopnosti přemýšlení nad kódem. (Koskela, 2013)

4.5 Automatizované a manuální testování

Manuální testování je prováděno člověkem. Klikáním skrze aplikaci nebo interagování s prostředím za použití nástrojů. Tento proces je náročný časově a tedy i finančně. Vyžaduje od pracovníka aby připravil prostředí k práci a testy spouštěl či prováděl. Takový proces může být ze své podstaty náchylný k faktoru lidské chyby.

Na druhé straně automatizované testy provádí počítač. Pomocí předem napsaných postupů lze otestovat jednotlivé funkce tříd nebo komplexní postupy pomocí uživatelského rozhraní. Kvalita testů je závislá na jejich kódu. Obecně je jejich výhodou větší robustnost, spolehlivost a především rychlost. (Pittet, 2020)

4.6 Jednotkové testy

Jednotkové testy se vztahují k testování izolovaných jednotek programu. Pojem jednotka programu nebo kódu není přesně definovaný. V běžném překladu je termín jednotka chápán jako funkce, metoda nebo procedura. Třída v objektově orientovaném programování může být ovšem také chápána jako jednotka programu. Syntakticky je jednotkou část kódu jako metoda, procedura nebo funkce třídy, která je

volána z vnějšku a může volat další takové jednotky. Funkce definovaná jednotkou kódu nemusí mít přímou asociaci s funkcemi na systémové úrovni.

Jednotky programu jsou testovány samostatně, izolovány od ostatních. K samostatnému testování jsou dva důvody. Prvním důvodem je zúžení prostoru pro hledání chyby. Chyby nalezené při jednotkovém testu se tak vždy budou vztahovat pouze k jedné jednotce. Druhým důvodem je otestování, že každý jedinečný vstup pro jednotku programu vytvoří očekávaný výstup. Ideálně jsou otestovány všechny možné vstupy, prakticky jen tolik, kolik je možné. Volba parametrů pro test je důležitá a udává reprezentativnost provedeného testu.

Podle (Koskela, 2013) je další výhodou skládání software z jednotlivých jednotek lepší spolupráce na jednom projektu. Práce programátorů na rozsáhlé základně kódu se usnadní, zásahy do funkcionality jsou více izolovány.

Vzhledem k omezenému kontextu, ve kterém jsou jednotky kódu testovány, neexistuje garance správné funkčnosti v nadřazeném kontextu. Pokud jednotka kódu projde jednotkovým testem úspěšně, chyba se stále může vyskytnout při kooperaci s dalšími jednotkami. Tyto chyby v ideálním případě odhalí testy vyšších úrovní. V tomto případe integrační test.

Jednotkové testy jsou prováděny programátorem, který kód napsal. Autor je detailně obeznámen se zdrojovým kódem a vnitřní logikou jednotky. Cílem programátora je aby jednotka fungovala podle očekávání a bez chyb. (Kshirasagar et al., 2008; Pittet, 2020)

4.6.1 Statické jednotkové testy

Ve statickém jednotkovém testování programátor jednotku nespouští. Je zkoumán její zdrojový kód a všechny možné chování algoritmu, které mohou nastat při běhu. Ve statickém jednotkovém testování je kód jednotky hodnocen z pohledu naplnění požadavků na jednotku pomocí prozkoumání kódu. Během procesu zkoumání kódu jsou odhaleny chyby a následně odstraněny. Statické testování je levnější než dynamické testování. Jeho princip je založený na porozumění kontextu kódu. Pokud je zvolena chybná implementace postupu, dynamický jednotkový test by nemusel chybu odhalit. Tento chybný postup může mít správný výstup v rámci požadavků na jednotku, například však může nepříznivě ovlivnit výkon aplikace.

Statické jednotkové testování je součástí rozsáhlejší filosofie, podle které by měl softwarový produkt projít inspekcí na každém milníku svého životního cyklu. V tomto milníku produkt nemusí být kompletně hotový. Myšlenkou za kontrolou kódu je nalezení chyby co nejdříve po jejím vytvoření. Statické jednotkové testování využívá dvou technik. První je inspekce - *inspection*. Inspekce kódu znamená kontrolu zdrojového kódu krok po kroku. Kontroluje se použití každého kroku s ohledem na kritéria jednotky. Druhou technikou je průchod - *walkthrough*. Jedná se o posouzení, kdy autor provádí kolegy manuálním nebo simulovaným průchodem aplikace dle zvolených parametrů. (Kshirasagar et al., 2008)

4.6.2 Dynamické jednotkové testy

Jednotkové testy založené na spuštění se nazývají dynamické jednotkové testy. V tomto testování je program spuštěn v izolovaném prostředí. Toto spuštění se

od běžného spuštění liší v několika ohledech. Za prvé, jednotka je vytržena z jejího prostředí, ve kterém má být spouštěna při běžné práci. Za druhé, spouštěcí prostředí je emulováno dodatečným kódem. Emulované prostředí je třeba k samostatnému spuštění. Za třetí, jednotka je spuštěna se zvolenými parametry. Její výstup je sbírán v mnoha podobách. Nejběžnější je kontrola výstupu z obrazovky nebo kontrola logovacích souborů. Výsledky jsou porovnány s předpokládanými výsledky. Rozdíl výsledků naznačuje chybu.

Prostředí vytvořené pro spuštění jednotkového testu je vytvořeno emulováním kontextu testované jednotky. Prostředí jednotkového testu sestává ze dvou částí. Části kódu, která volá testovanou jednotku - *caller*. A všech dalších jednotek, které volá testovaná jednotka. Prostředí jednotky musí být emulováno, aby jednotka mohla být testována izolovaně. Takové prostředí musí být co nejjednodušší aby se nestalo překážkou při hledání chyby. Jednotka volající další jednotky je nazývána *test driver* a všechny emulované (volané) jednotky se označují *stubs*. Volající jednotka a všechny volané jednotky se dohromady nazývají *scaffolding*. (Kshirasagar et al., 2008; Koskela, 2013)

```
    // import knihovny pro testovani

    // definovani tridy Rectangle
    export class Rectangle {
    // konstruktor definujici dve tridni promenne width a height
    constructor(protected width: number,protected height: number){};
    // funkce zjistujici zda se jedna o ctverec
    public isSquare(): boolean {
    return this.width === this.height;
    }
    }

    describe('Rectangle class tests', () => {
    test('Should return true for square', () => {
    // vytvoreni tridy s parametry urcujici ctverec
    const rectangle = new Rectangle(5, 5);
    // ocekava se, ze funkce urci ctverec
    expect(rectangle.isSquare()).toBeTruthy();
    });

    test('Should return false when width != height', () => {
    // vytvoreni tridy s parametry pro obdelnik
    const rectangle = new Rectangle(5, 3);
    // ocekava se, ze funkce rozhodne, ze se nejedna o ctverec
    expect(rectangle.isSquare()).toBeFalsy();
    });
    });

```

Kód 1: Implementace vzorového dynamického jednotkového testu pro třídu obdélník v jazyce TypeScript, testovací framework Jest

4.7 Integrační testy

Softwarový modul je samostatný element systému. Moduly mají přesně definované rozhraní s ostatními moduly. Modul může být subrutina, funkce, metoda, třída nebo kombinace těchto prvků sestavená za účelem doručení služby vyššího řádu. Systém je kolekce modulů propojená definovanou cestou za účelem vytvoření funkcionality. V projektech, programátoři implementují moduly pro propojení a sdílení funkcionalit s prací ostatních. Jednotlivé moduly jsou individuálně testovány pomocí jednotkových testů. Jednotkové testy píší autoři jednotek a jsou zodpovědní za jejich správné chování. Při jednotkovém testování se testují jednotky zvlášť. Dalším krokem je složení modulů dohromady ve funkční systém. Složení modulů není jednoduchá úloha kvůli velkému množství rozhraní mezi moduly a chyb které mohou vyprodukovat. Cílem integračního testování je poskládat částečně stabilní systém v laboratorním prostředí. Důvody pro integrační testování jsou:

- Jednotlivé moduly jsou psané různými programátory nebo jejich skupinami. Tito programátoři mohou pracovat odděleně. V dnešní době jsou některé moduly sdíleny napříč různými aplikacemi. Ne všechny moduly tedy musí být určeny pro konkrétní aplikaci.
- Jednotkové testy jsou prováděny v emulovaném prostředí spolu s emulovanými volanými jednotkami. Emulované volané jednotky jsou určeny pro testovací účely a vrací předdefinované hodnoty. Tyto hodnoty mohou být založeny na úsudku programátora nebo na dokumentaci, která nemusela být dodržena. Obecně je kvalita jednotkových testů závislá na schopnostech autora jednotky. Z těchto důvodů není zajištěno očekávané chování ve skutečném prostředí.
- Některé moduly jsou vlastní podstatou náchylnější k chybám. Může to být dáno jejich složitostí. Takové moduly je třeba identifikovat.

Cílem integračních testů je poskládat částečně funkční systém z modulů. Integrační test by měl proběhnout po přidání každého dalšího modulu. Tak bude zajištěno správné fungování každého dalšího modulu s předešlými moduly. Jinými slovy, integrační testování je opakující se operace skládání modulů a provádění testů nad nimi za účelem odhalení chyb v integraci s ostatními. Integrační testování začíná s několika moduly, postupně modulů přibývá a končí, pokud je systém plně integrován. (Kshirasagar et al., 2008; Pittet, 2020)

```
1    describe('Notes', () => {
2    // ... inicializace testu je pro prehlednost vynechana
3
4    test('Should not delete when id not exist', async () => {
```

```
    // vytvoreni dotazu na endpoint poznamek s neexistujicim ID
    const response =
    await request.delete(`${endpoint}/${NOT_EXIST}`)
    .set('x-api-key', apiKey);

    // je ocekavan HTTP status 400 - Bad Request
    expect(response.status).toBe(400);
    // je ocekavano, ze funkce pro vyhledani poznamky dle ID
    // bude volana prave jednou
    expect(mockNoteFindByd).toBeCalledTimes(1);
    // je ocekavano ze funkce odstraneni poznamky nebude volana
    expect(mockNoteRemove).not.toBeCalled();
    });
    });

```

Kód 2: Implementace nekompletního vzorového integračního testu pro endpoint smazání poznámky v jazyce TypeScript, testovací framework Jest

4.8 Systémové testy

Cílem testování na systémové úrovni je ověření, že implementace systému odpovídá požadavkům zákazníka. Systémové testování vyžaduje množství času a zdrojů k ověření, že zadané podmínky byly naplněny dle očekávání. Množství testů je prováděno aby pokryly rozsáhlé implicitní i explicitní požadavky na systém. Protože se často jedná o systém závislý na konkrétním hardware, je třeba pohled na systém rozšířit o další specifika. Systémové testování je rozsáhlá kategorie ve které jsou spouštěny testy které mají otestovat aplikaci v mnoha ohledech.

- **Zákldaní testování:** Základní testování ověřuje, že systém je spustitelný a je možné ho podrobit dalšímu testování. Tyto testy poskytují omezené otestování hlavních funkcionalit dle specifikace. Cílem je ujistit se, že systém je dostatečně stabilní, aby se mohlo přejít k dalšímu rozsáhlejšímu testování.
- **Funkcionální testování:** Tyto testy jsou navrženy, aby otestovaly funkce systému jak nejpodrobněji je to možné. Funkcionální testy kontrolují shodu požadavků klienta na funkce systému se skutečně implementovanou funkcionalitou. Vedle jiného je kontrolováno uživatelské rozhraní, zda naplňuje očekávání uživatele nebo bezpečnost systému.
- **Testy robustnosti:** Robustnost systému označuje jeho náchylnost na neplatné vstupy. Testy této kategorie ověřují jak se systém zachová, pokud je postup uživatele chybný. Cílem testů je tedy najít situace, kdy systém spadne nebo se dostane do stavu, kdy s ním uživatel dále nemůže pracovat.
- **Testy interoperability:** V této kategorii je testování, jak systém dokáže spolupracovat se službami třetích stran. U mnoha systémů může být požadavkem

aby hardwarové zařízení bylo zaměnitelné, nebo aby systém byl zpětně kompatibilní. Systémy často obsahují možnosti nastavení, které je třeba otestovat. Operace spojené s nastavením systému se nazývají testy konfigurace. Testy interoperability zjišťují, zda systém funguje stejně na různých platformách.

- **Testy výkonosti:** Tyto testy jsou zaměřené na porovnání výkonnosti systému s požadavky zákazníka či implicitními požadavky na aplikaci. Metriky výkonnosti je třeba zvolit v závislosti na druhu aplikace. Příkladem vztažené metriky může být očekávání, že transakční systém bude mít průměrnou délku odpovědi kratší než 1 sekunda v 90% času. Během těchto testů je třeba posuzovat výsledky s ohledem na hardware, na kterém je software spouštěn.

- **Testy rozšiřitelnosti:** Testováním rozšiřitelnosti se testují hranice systému. Cílem je zjistit funkční limit systému při rozšiřování datové a jiné základny. Například jak veliký objem dat je systém schopen efektivně zvládnout.

- **Zátěžové testy:** Cílem zátěžového testu je vystavit systém náporu vyššímu, než na který byl zkonstruován. Zátěžové testy jsou vytvořeny tak, aby systém dotlačily za limit udržitelnosti. Testy tak zaručí, že se systém bude chovat předvídatelně i v extrémních podmínkách. Pokud okolnosti způsobí, že systém selže, je na místě inicializovat zotavovací mechanismus.

- **Testy stability:** Testy stability testují, zda systém zůstane funkční pod zamýšleným náporem po předpokládanou dobu. Pokud se aplikace chová požadovaným způsobem při krátkodobém horizontu spolu s vybranými testy, jeho stabilita není zaručena. Dlouhodobé spuštění bez restartování a množství uživatelů používající systém různými způsoby může mít vliv na funkcionalitu systému. Testováním stability je ověřeno očekávané chování po dlouhou dobu funkčnosti.

- **Testy spolehlivosti:** Spolehlivost je metrika k měření času, po který je systém použitelný. Stabilita systému je typicky vyjádřena průměrným časem bez selhání - *mean time to failure* (dále jen MTTF). Během testování systému se vyskytují chyby, které je třeba odstraňovat a pokračovat v testování. Během tohoto cyklu měříme čas fungování a výskytu chyby. Pokud je odhalena chyba, programátoři ji opraví. Z průměru délky trvání oprav se vypočítá průměrná doba opravy *mean time to repair* (dále jen MTTR). Nyní lze vypočítat průměrnou dobu mezi chybami *mean time between failures* (dále jen MTBF) pomocí jednoduchého vzorce MTBF = MTTF + MTTR.

- **Regresní testy:** V této kategorii se nevytváří další testy. Namísto toho jsou vybrány některé ze stávajících testů a spuštěny k ověření, že nové úpravy nezpůsobily další navazující chyby. Vybrané testy by měly reprezentovat klíčové funkcionality.

- **Kontrola dokumentace:** Je třeba ověřit, že postupy v manuálu plně odpovídají postupům v systému. Veškerá uživatelská dokumentace musí být aktuální včetně tutoriálů a online pomoci.

- **Legislativní kontroly:** V této sekci je systém plně připraven a odeslán regulačním úřadům. Hlavním účelem je získat osvědčení pro provoz od těchto autorit. (Kshirasagar et al., 2008; softwaretestingfundamentals, 2020)

4.9 Akceptační testy

Po provedení systémových testů a naplnění jejich očekávání je produkt připraven na odeslání zákazníkovi. Zákazník poté provede vlastní sadu testů zaměřenou na ověření naplnění očekávání uživatele. Je běžné, že zákazník má roli uživatele. Pokud systém splní požadavky zákazníka, může být akceptován. Zákazník si obecně vyhrazuje právo produkt odmítnout pokud doručený produkt nenaplní očekávání uživatele. Testy přijetí jsou rozděleny do dvou druhů. Test přijetí uživatelem a byznysový test přijetí (viz. Úrovně testování). UAT je prováděn na straně uživatele a ověřuje naplnění zadaných smluvních nároků na systém. BAT je prováděn dodavatelem a je ověřením, že produkt bude schopný splnit následné UAT spouštěné zákazníkem. (Kshirasagar et al., 2008)

5 Průběžná integrace

Průběžná integrace má mnoho definicí. Nejčastěji uváděnou je definice z příspěvku na blogu Martina Fowlera z roku 2006.

Průběžná integrace je praktika při vývoji software, kde členové vývojového týmu často integrují svou práci. Obvykle každý pracovník alespoň jednou denně. Každá integrace je verifikována automatizovaným sestavením aplikace a spuštěním testů pro odhalení chyb co nejdříve po tom, co byly vytvořeny. Mnoho týmů shledalo, že tento postup vede k razantnímu snížení výskytu integračních problémů a umožňuje týmům vytvářet kohezivní software rychleji. (Fowler, 2006)

Dále dle (Fowler, 2006) má výraz průběžná integrace - *continuous integration* základ v původních 12 principech extrémního programování Kena Becka v publickaci (Beck, 2004). I když princip průběžné integrace nevyžaduje žádné zvláštní nástroje, použití integračních serverů je užitečné v mnoha ohledech.

(Sharma, 2017) popisuje, že vývoj aplikace většinou sestává z několika týmů programátorů, kteří pracují na různých částech aplikace. Typicky aplikace musí komunikovat s některými externími službami nebo dalšími aplikacemi. Takové služby mohou být starší verze aplikace nebo být vyvíjené třetími stranami. Výsledkem je potřeba vývojářů integrovat svou práci s komponentami dalších týmů.

Tato potřeba dělá z integrace klíčový a komplexní úkol v životním cyklu vývoje software. Pravidelné integrování je často nazýváno jako průběžná integrace. Je jednou z hlavních praktik agilního programování.

V tradičním vývojovém procesu byla integrace druhořadým úkolem. Prováděla se až poté, co byl modul či celá aplikace úplně sestavena. Tento proces byl nákladný a nepředvídatelný. Nekompatibilita a defekty, které se dají zjistit až po integraci součástí byly odhalovány na konci vývojového procesu. Výsledkem byl typicky signifikantní nárůst oprav a práce.

Agilní způsob vývoje zařadil logický krok ke snížení tohoto rizika pomocí průběžné integrace komponent. V tomto kroku integrují vývojáři svůj kód se zbytkem týmu pravidelně.

Ve své publikaci (Duvall et al., 2008) uvádí typický průběh průběžné integrace. Tento scénář obsahuje následující kroky:

1. Nejprve programátor publikuje lokální změny do sdíleného vzdáleného repozitáře. Tento sdílený repozitář je monitorován serverem průběžné integrace.

2. Potom co jsou změny publikovány na sdílený repozitář, server průběžné integrace tyto změny detekuje. Aktuální kód je zkopírován na server průběžné integrace a ten nad ním spustí seznam definovaných úloh k provedení průběžné integrace.

3. Server průběžné integrace vygeneruje některou formu zpětné vazby. Například rozešle email s informacemi ohledně výsledků prováděných kroků interesovaným skupinám.

4. Dále server průběžné integrace pokračuje v monitorování sdíleného repozitáře kvůli dalším změnám.

5.1 Vzdálený repozitář

Typická implementace průběžné integrace zahrnuje zdrojový repozitář. Tyto repozitáře často zastupují více rolí jako build server nebo repozitář pro sestavené artefakty.

Pokud je zaznamenána změna v repozitáři, nejčastěji pokud některý z programátorů přidal či jinak upravil kód v repozitáři, je uveden do pohybu řetězec definovaných událostí. Obligátní první krok z tohoto řetězce bývá stažení nejnovější verze kódu z repozitáře a jeho kompilace, pokud je vyžadována. Pro interpretované jazyky může být spuštěn například linter. Často aplikace vyžaduje množství kódu dalších autorů, dostupných z dalších repozitářů. Pokud sestavení proběhne úspěšně, jsou nad kódem spuštěny jednotkové testy. Navazujícím krokem, pokud jednotkové testy neskončí chybou, je aplikace nahrána do testovacího prostředí. V testovacím prostředí proběhne další testování na základě kterého se může přistoupit k vydání aplikace. (Stolberg, 2009; Verona, 2016)

5.2 Server průběžné integrace

Sestavování aplikace je automatizováno pomocí některého serveru průběžné integrace. Některé populární řešení serveru jsou otevřené projekty a tedy zcela zdarma, jiné placené. Dalším rozdílem může potom být forma hostování serveru. Například otevřený projekt Jenkins (dříve Hudson) je zcela zdarma, je ale třeba vlastnit hardware na kterém poběží. Další řešení, jako Bamboo, který je serverem průběžné integrace od společnosti Atlassian jsou dostupná jako externí služba za měsíční poplatek. Důvodem zvolit konkrétní řešení může být krom nároků na provozování také kompatibilita. Například Microsoft CI server je vhodný pro práci s platformou .NET a je integrován s vývojovým prostředím Visual Studio. Serverů průběžné integrace je mnoho, každý se svými specifiky. Společnou vlastností však je implementace nástrojů pro implementaci průběžné integrace. (Rossel, 2017; Duvall et al., 2008)

5.3 Přínosy implementace

Implementace průběžné integrace má za úkol přinést některé výhody. Ze své podstaty je každý repetitivní proces prováděný člověkem náchylný na chybu. Nahradíme-li pracovníka strojem, tyto nevýhody odstraníme. (Duvall et al., 2008)

5.3.1 Snížení rizik

Pravidelnou integrací, tedy alespoň jednou denně je podle (Duvall et al., 2008) možné zmírnit rizika plynoucí z chyb kumulativního vývoje. Tímto procesem je možné usnadnit detekci chyb a měřit zdraví software.

- **Defekty jsou odhaleny a odstraněny dříve:** Protože server průběžné integrace spouští pravidelně testy a sestavení nad každým přírůstkem, je velká šance, že chyba bude okamžitě odhalena. Takto mohou být chyby odhaleny okamžitě při přidání kódu do repozitáře, namísto odhalení pozdějšími testy.

- **Zdraví software je měřitelné:** Zabudováním automatického testování a sestavování do automatické integrace je možné měřit atributy software. Takto lze například měřit průměrné časové úseky bez poruchy nebo průměrný čas za který je porucha odstraněna.
- **Chování software je předvídatelné:** Sestavování a testování software v čistém prostředí, pomocí automatizačních skriptů zaručuje předvídatelné chování software.

5.3.2 Odstranění opakujících se úloh

Množství času i peněz lze ušetřit, odstraněním repetitivních procesů, jejich automatizací. Pokud tyto opakující se úlohy provádí pracovník, jsou také náchylné ke vzniku chyb lidského faktoru. Automatizování všech aktivit zahrnujících kompilování kódu, integraci databáze, testování, inspekci kódu, nasazení aplikace a kontroly zpětné vazby se lze ujistit, že je proces vždy stejný. Proces bude vždy následovat stejné pořadí úkonů a bude spuštěn vždy po přidání nového kódu do repozitáře. (Duvall et al., 2008)

5.3.3 Aktuální software

Automatizace úkonů potřebných k sestavení aplikace je též mnohem rychlejší, než pokud je prováděno ručně. V kombinaci se strojovou pravidelností, je doručení nové verze zákazníkovi velice efektivní a rychlé. Tato skutečnost dodává vývoji flexibilitu a přizpůsobivost. Software může rychleji odrážet aktuální trendy. (Rehkopf, 2020)

5.4 Automatizované testování

Softwarové projekty obsahují dodatečný kód, který není explicitně určen k vykonávání byznys logiky programu. Tento dodatečný kód jsou testovací případy. Jsou souborem předpokladů správného chování a fungují jako záruka, že kód je bez chyb. Během vývoje jsou tyto testy spouštěny jednotlivými vývojáři pro ověření, že nově přidaný kód nezpůsobil žádné chyby. Tyto testy může také spouštět některý z nástrojů pro automatizaci tohoto testování. Nástroje pro průběžnou integraci spouští tyto testy předem definovanou akcí. Obecně pokud vývojář přidá nové úpravy do repozitáře, nástroj pro správu verzí spustí tyto automatizované testy. (Rehkopf, 2020; Soni, 2017)

5.5 Automatizace buildu

Build je artefakt vytvořený sestavením a podle typu kódu i kompilací aktuálního kódu aplikace. Výsledné artefakty jsou distribuovány zákazníkovi skrze různé kanály. Nástroje pro průběžnou integraci pomáhají řídit tento proces za použití automatických spouštěčů ve verzovacím systému. Příkladem takového spouštěče může být přidání nového kódu do produkční větve ve verzovacím systému. Nástroj potom provede sestavení aplikace a nahrání artefaktu na server kde si jej je možné stáhnout. (Rehkopf, 2020; Soni, 2017; Verona, 2016)

(Duvall et al., 2008) upozorňuje na doporučení z publikace (Beck, 2004) o agilní metodice extrémním programování, které radí držet dobu sestavení aplikace včetně testování pod deset minut. Následování takového pravidla neodrazuje programátory od častého spouštění integrace a je tak zaručena plynulá integrace. Další kód není dovoleno integrovat, dokud poslední integrace neproběhne bez chyb. Desetiminutové sestavení tak urychluje tento proces.

6 Průběžné doručení

Průběžné doručení - *continuous delivery* je dalším krokem k dokončení životního cyklu software. Implementací průběžného doručení jsou artefakty vzniklé v procesu průběžné integrace doručeny do produkčního prostředí. Průběžná integrace je tedy požadavkem k úspěšné implementaci průběžného nasazení. Samotný proces průběžného nasazení potom může být stejně jednoduchý, jako je stisknutí tlačítka.

Poté co úspěšně proběhne řetězec integračních úloh, otestování nových změn, sestavení aplikace a systémové testování, je možné software doručit zákazníkovi do produkčního prostředí. Tento proces vyžaduje mnoho specifických úkonů. Příklady těchto kroků dle (Rossel, 2017) jsou uvedeny:

- V závislosti na typu aplikace je třeba informovat uživatele, že systém bude určitou dobu nepoužitelný. Nasazení nové verze do produkčního prostředí bývá smluveno v dané časové intervaly. Samotní uživatelé však nemusí být informováni nebo mohou tuto skutečnost opomenout. V takovém případě je třeba na probíhající nasazení nové verze zákazníka upozornit profesionální cestou namísto informace z prohlížeče, že stránka neexistuje.

- Pravděpodobně dojde k zálohování stávající verze klientovi aplikace, pokud se nová verze nepodaří bezchybně nasadit, bude třeba obnovit stávající verzi. Dále musí být zaručeno, že zákazník neztratí žádná data. Data jsou nejčastěji držena ve formě databáze, její stav je tak třeba uložit před jakoukoliv změnou.

- Je třeba aplikaci připravit na běh v konkrétním prostředí. Téměř každá aplikace vyžaduje ke správnému běhu konfiguraci. Typickým příkladem je přepnutí provozu aplikace z vývojového do produkčního, kdy budou případná chybová hlášení redukována na minimum. Taková konfigurace je běžně realizována konfiguračními soubory. Takto je například zaručeno, že se zákazníkovi nepřipojí databáze s testovacími daty.

- Dále bývá často zapotřebí změnit strukturu databáze. Téměř každá funkcionalita potřebuje ukládat data a pokud některá v nové verzi přibyla, je pravděpodobné, že i v databázi bude třeba vytvořit místo pro její data. V takovém případě je tedy třeba i zaručit, že zákazníkova data zůstanou nepoškozena, pravděpodobně další zálohou.

- Některé změny v konfiguraci či prostředí mohou vyžadovat kompletní restartování platformy. Je tedy třeba znovu spustit aplikaci a připravit všechny její součásti.

Průběh doručení je typický pro každý druh aplikace. Může se jednat o rozsáhlý seznam aktivit, kde každá může selhat. Pokud jsou tyto kroky prováděny manuálně, je vypracována jejich dokumentace. Pro dokumentaci je však klíčovou vlastností aktuálnost. Pokud dokumentace není zařazena do procesu průběžné integrace, je velmi náchylná na zastarání a tedy její existence ztrácí význam. Automatizace tohoto náročného procesu usnadní úkol doručení každé nové verze zákazníkovi. Čím

složitější tento proces je, tím roste snaha programátorů o oddálení dodávky. Čím náchylnější k chybám je, tím spíše se tým snaží ujistit, že vše je připraveno a snížit tak frustraci při průběhu nasazení. Pokud je proces automatizován, výrazně se zrychlí, sníží se tlak na tým programátorů a jeho frustrace z doručování produktu. Zrychlením cyklu doručování produktu, je možné zákazníkovi produkt dodávat po menších částech což zajistí i flexibilitu při požadování změn a opravě chyb.

Naneštěstí ne vždy je průběžné doručení možné automatizovat v plném rozsahu. V případech, kdy produkční platforma není zcela v režii dodavatelské firmy, jsou možnosti omezeny. Například zákazník, který si z určitých důvodů spravuje vlastní databázi nebo na produkční platformě není dostupné internetové připojení. V takových případech stále některé kroky automatizovat jdou a čím více jich je, tím lépe. Pro první uvedený příklad je možné automatizovat vše, krom připojení k databázi, pro druhý případ je možné využít například lokální doručovací skripty.

Další překážkou v automatizaci procesu doručení může být nesouhlas zákazníka. Automatizování procesu je časově velmi náročné. Pro zákazníka tento čas znamená především peníze. Pohled na automatizaci se potom na obou stranách liší. Dodavatelská firma získává usnadnění práce, ulehčení od frustrace zaměstnanců a další výhody. Tým bývá vázán smlouvou k zákazníkovi ohledně pravidelnosti a kvalitě dodávek a v tomto ohledu by výsledek měl být nezávislý na vnitřním průběhu nasazení změn. Z pohledu zákazníka se tedy nemusí jednat o tak důležitý krok a nemusí být ochoten za implementaci zaplatit. Implementací automatizace by však zákazník měl získat rychlejší doručení a méně defektů. (Rossel, 2017; Verona, 2016)

7 Nástroje pro CI/CD

7.1 GitLab

GitLab je projekt rozdělený do dvou verzí. Verze community edition (dále jen CE) je otevřený projekt - *open source* s velmi volnou licencí MIT. Zdrojové kódy projektu jsou tedy volně přístupné a je možné s jeho vývojem pomoci. Druhou verzí projektu je enterprise edition (dále EE) určenou pro větší projekty alespoň o stovce pracovníků a obsahuje rozšířenou paletu nástrojů. Tato verze podléhá proprietální licenci a je založena na subskripcích. Placená verze krom dalších nástrojů nabízí technickou podporu.

GitLab je kompletním DevOps řešením pro celý projekt. Pro průběžnou integraci nabízí správu Git repozitářů. K tomuto účelu poskytuje množství přehledných pohledů na kód, kontrolu nad historií a řešením zabezpečení na mnoha úrovních. Rozsáhlé nastavení autorizace i autentifikace vývojářů, automatické spouštění testů pomocí *pipelines*. *Pipelines* jsou mocným nástrojem pro automatizaci. Právě pomocí tohoto nástroje lze spouštět například testy nad nově přidaným kódem. V kombinaci s kontejnerizací lze vytvářet tzv. *runner*, což je virtuální prostředí, které je možné předpřipravit pro spouštění, sestavení a testování konkrétní aplikace. Vestavěný nástroj pro řízení požadavků od zákazníka podporuje implementaci agilních metodik a poskytuje přehledný pohled na probíhající práce. Správa verzí probíhá pomocí tagů. Tag je označení verze sestavení aplikace. Může obsahovat artefakty, které byly vytvořeny během sestavování a uživatel si je tak může stáhnout přímo ze stránek gitlabu. Pro malé nebo projekty nezaložené na výdělku tak může gitlab představovat kompletní řešení produktu.

Na základě skutečnosti, že se jedná o otevřený projekt, je možné projekt GitLab CE stáhnout z oficiálního repozitáře a spustit na vlastní platformě. Toto je veliká výhoda, na rozdíl od dalších služeb jako je například GitHub, které jsou založeny pouze na měsíčních subskripcích. Vlastním hostováním odpadají omezení vyplývající ze zdrojů provozovaných poskytovatelem. Velikost úložiště nebo výkon je tak závislý čistě na platformě, na které je projekt spuštěn. (GitLab, 2020)

7.2 Jenkins

Jenkins je pravděpodobně nejoblíbenějším nástrojem průběžného doručení - *continuous delivery*. Jedná se o samostatný automatizační server. Jenkins je dostupný v mnoha verzích, jako Docker kontejner, skrze správce balíčků pro danou platformu nebo jako spustitelná aplikace schopná běžet v jakémkoliv prostředí kde je nainstalována Java (JRE). Projekt Jenkins je otevřený - *open source* a jeho zrdojový kód je dostupný na síti GitHub. Je tedy stejně jako u projektu GitLab možné kód upravovat a zapojit se do vývoje. Projekt je distribuován pod volnou licencí MIT.

Svou podstatou je Jenkins založený na rozšiřování pomocí doplňků - *plugins*. Každý doplněk přidává do základního serveru další funkcionalitu. Příkladem může být doplněk pro podporu komunikace s repozitářem hostovaným na serveru GitHub. Tímto způsobem je přidána možnost například po změně kódu v repozitáři spustit některou funkcionalitu jako je automatické testování. Tyto doplňky jsou

vytvářeny a udržovány komunitou. Jenkins nabízí společný registr, ve kterém je možné vyhledávat nové doplňky a prozkoumat dokumentaci.

Jenkins vedle vytvoření volného seznamu úkonů - *free job* nabízí i *pipelines*. Ty se staly jakýmsi standardem. Podporuje je většina CI/CD nástrojů a Jenkins není odchylkou.

Protože je Jenkins na scéně již poměrně dlouhou dobu, od roku 2005, může jeho klasické rozhraní působit staromódně. V roce 2016 na svém blogu vývojáři oznámili, že pracují na novém rozhraní nazvaném Blue Ocean. Toto rozhraní je volitelné a dostupné jako jeden z doplňků. Dnes stále ještě nepodporuje všechnu funkcionalitu a je tedy třeba se při některých krocích spoléhat na původní rozhraní. (Jenkins, 2020)

7.3 Travis

Travis CI je služba průběžné integrace. Na rozdíl od GitLab, Travis je licencovaný software a je tedy dostupný pouze skrze některou formu subskripce. Na základě vybraného plánu, bude projekt omezen na počet konkurentně běžících procesů. Na druhou stranu, projekt se zavazuje k bezplatnému použití v případě otevřených projektů - *open source*. Dle autorů se touto cestou snaží oplatit komunitě co od ní dostali.

Hlavní výhodou služby Travis CI je jednoduchost. Jediné co stačí ke spuštění *pipelines* je jednoduchým průvodcem připojit repozitář na některém z podporovaných serverů a vytvořit soubor *.travis.yml*. Definování *pipeline* soubory ve formátu *yml* je standardní krok pro většinu nástrojů. Vše je již připravené a stačí pouze definovat akce tímto standardním způsobem. Tarvis CI podporuje přes 30 jazyků a je možné požádat o zařazení dalšího. Podpora znamená, že je schopný virtuálně vytvořit prostředí, ve kterém poběží kód ve zvoleném jazyce.

Vedle spouštěčů vázaných na události v repozitáři, Travis CI umožňuje spouštění pomocí automatického plánovače úloh CRON. Tento program je standardní výbavou linuxových serverů a dokáže spouštět periodicky úkoly v závislosti na definovaném čase. (Travis CI, 2020)

8 Docker

Firma Docker Inc. začala jako poskytovatel *platform as a service* (PaaS) pod původním jménem dotCloud. Firma interně pro fungování své infrastruktury používala linuxové kontejnery. Pro správu a udržování těchto kontejnerů vytvořila interní nástroj který interně přezdívala *docker*. Odtud tedy pozdější název Docker.

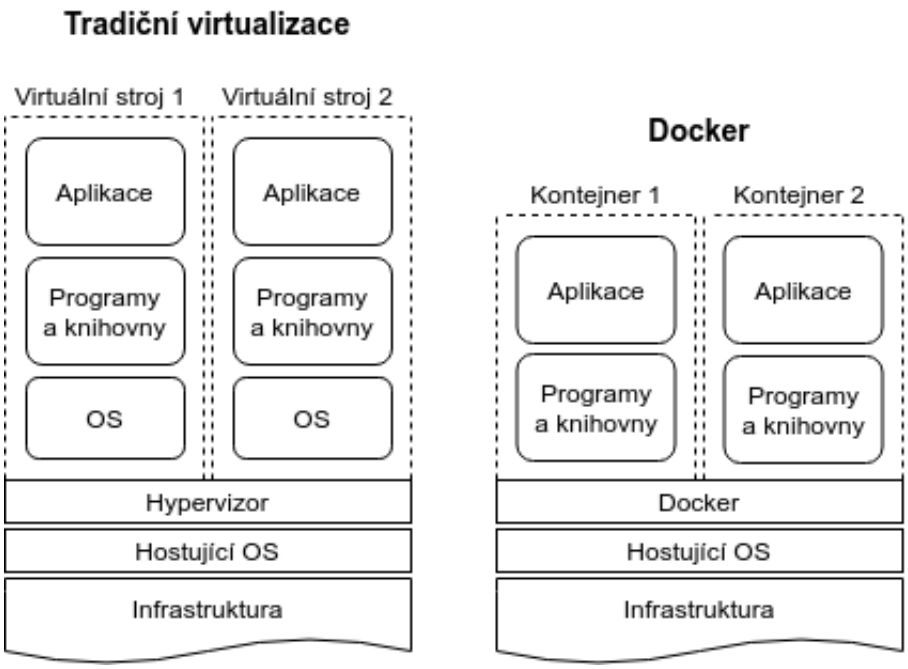
V roce 2013 firma nebyla spokojená se svým produktem a hledala nový způsob oživení výdělku. Pro tento účel najala Bena Glouba do role CEO, který firmu přejmenoval na Docker Inc. a zaměřil se na vývoj interního nástroje Docker.

Dnes je firma Docker inc. široce známa a její produkt je sponzorován některými největšími společnostmi v Silicon Valley. Téměř všechny tyto sponzorské dary byly učiněny po přejmenování firmy na Docker Inc.

Název Docker je odvozen z britského slova označujícího pracovníka v docích.

Docker je program pro automatizaci procesu kontejnerizace aplikací. Projekt byl vytvořen týmem Docker Inc. (původně dotCloud Inc.) a uveřejněn pod licencí Apache 2.0. Docker umožňuje nasadit aplikaci do virtualizovaného kontejnerového prostředí. Je vytvořený tak, aby poskytoval zdrojově nenáročné a rychlé spouštěcí prostředí pro spouštění kódu, stejně tak efektivní způsob jak dopravit aplikaci z vývojového prostředí na testovací servery či produkční prostředí.

Klíčovou výhodou kontejnerů je možnost zabalit aplikaci spolu s její konfigurací a závislostmi, čímž se stane přenositelná mezi prostředími. Kód uvnitř aplikace běží odděleně od zbytku prostředí. Vývojáři aplikace se tak nemusejí obávat o kompatibilitu prostředí, ve kterém kontejner běží. Jejich zaměření je pouze na zabalení aplikace do standardizovaného kontejneru, který poskytuje nejlepší podmínky. Stejně tak operační tým se nemusí starat o nastavení konfigurace nebo zajišťování závislostí aplikace. Vše co stačí k běhu aplikace je spuštění standardizovaného kontejneru.



Obrázek 3: Schéma tradiční virtualizace a Docker kontejneru

Technologie Docker využívá linuxové jádro a jeho funkce jako *Cgroups* nebo *namespaces* k oddělení procesů, které tak mohou běžet odděleně. Takové oddělení je záměrem kontejnerů - možnost spouštět procesy odděleně k lepšímu využité platformy a zdrojů při zachování bezpečnosti kterou by poskytly oddělené systémy.

Docker a jeho nástroje poskytují model založený na obrazech anglicky *images*. Tento model usnadňuje přenášení aplikací, kolekcí služeb se všemi závislostmi na různá prostředí. Docker také zajišťuje nasazení aplikace či kolekce procesů, ze kterých sestává do kontejneru.

Tato sada nástrojů postavených nad linuxovými kontejnery dělá Docker velmi pohodlným nástrojem k použití. Poskytuje uživatelům bezprecedentní možnosti nasazení aplikace v kontejnerizovaném prostředí a distribuci. (Red Hat, 2020; Turnbull, 2016; Poulton, 2018; Soni, 2017)

8.1 Kontejnerizace

Koncept kontejnerů není nový, Linux podporuje kontejnery již od roku 2008. Linuxové kontejnery umožňují izolaci na úrovni procesoru, paměti nebo síťových zdrojů během čehož je sdílen operační systém. Takový koncept umožňuje procesům pracovat odděleně a to s mnohem menším dopadem na zdroje, než kompletní virtuální stroj.

Kontejnery mají v dlouhou historii ve výpočetní technice. Na rozdíl od plné virtualizace, kde běží nezávislý virtuální stroj nad fyzickým hardwarem, kontejner běží v prostředí uživatele v jádře operačního systému. Kontejnerová virtualizace se proto často nazývá virtualizace na úrovni operačního systému. Technologie kontejnerizace umožňuje běh několika uživatelských prostředí na jednom hostiteli.

Jako následek jejich titulu host - anglicky *guest* operačního systému, jsou kontejnery někdy vnímány jako méně flexibilní řešení. V kontejneru může běžet pouze operační systém stejného druhu jako na hostiteli, tedy na operačním systému Ubuntu je možné spustit v kontejneru operační systém Red Hat Enterprise Linux avšak není možné spustit kontejner s Microsoft Windows [1].

Kontejnery byly také považovány za méně bezpečné než oddělené, plně virtualizované stroje. Protiargumentem může být například fakt, že kontejner má menší třecí plochu pro vznik bezpečnostních děr na rozdíl od kompletního operačního systému, který plná virtualizace vyžaduje nemluvě o samotném hypervizoru.

Navzdory těmto limitům se kontejnery používají pro mnoho účelů. Kontejnery se často užívají u produktů, kde je třeba pružně rozšiřovat a zmenšovat platformu dle momentálních požadavků. Navzdory obavám o jejich bezpečnost se využívají i jako prostředí pro běh různých procesů. Jedním z běžných použití kontejnerizace je *chroot jail*, který vytvoří izolované prostředí spolu se složkami potřebnými k běhu procesu. Pokud takové prostředí bude kompromitováno, útočník zůstane izolovaný v tomto prostředí neschopen uškodit hostujícímu systému.

Novější kontejnerizační techonologie obsahují OpenVZ, Solaris Zones a lxc. Díky těmto techologiím kontejner může vypadat jako plný virtuální počítač, namísto spouštěcího prostředí. V případě Docker kde jsou k dispozici nové linuxové jádra, které umožňují vytvořit silnou izolaci, kontejner tak má vlastní síť, paměť a řízení zdrojů.

Kontejnery jsou obecně považovány za odlehčené řešení, protože nekladou velké nároky na zdroje. Na rozdíl od tradiční virtualizace nebo paravirtualizace nevyžadují emulační vrstvu ani vrstvu hypervizoru, namísto toho využívají běžné rozhraní systémových volání. To značně snižuje nároky na hostující systém a je tak možné současný běh vícero kontejnerů.

I přes jejich dlouhou historii kontejnery nezískaly rozsáhlou oblibu. Z velké části tento neúspěch může být přisuzován jejich komplexnosti, náročnosti na údržbu a sestavení. Otevřený projekt Docker se to však rozhodl změnit. (Turnbull, 2016; Sharma, 2017; Soni, 2017)

8.2 Linuxové kontejnery

Moderní kontejnery mají kořeny v linuxovém světě a jsou produktem usilovné práce mnoha společností po dlouhou dobu. Například společnost Google LLC přispěla několika projekty zabývajícími se kontejnerizací do linuxového jádra. Bez těchto příspěvků by moderní kontejnery jak je známe dnes nemohly fungovat.

Některé z těchto technologií, které umožnili masový růst kontejnerizace v posledních letech, zahrnují *kernel namespaces*, *control groups*, *union filesystems* a samozřejmě Docker.

Existuje mnoho technologií operačních systémů, které se datují před Docker. Některé se dokonce datují až k System/360. *BSD Jail* a *Solaris Zones* a další příklady dobře známých technologií založených na unixu.

[1] Pro spuštění kontejneru s jiným jádrem než je jádro hostujícího OS je vytvořen virtuální stroj odpovídajícího jádra.

8.3 Kontejnery na platformě MS Windows

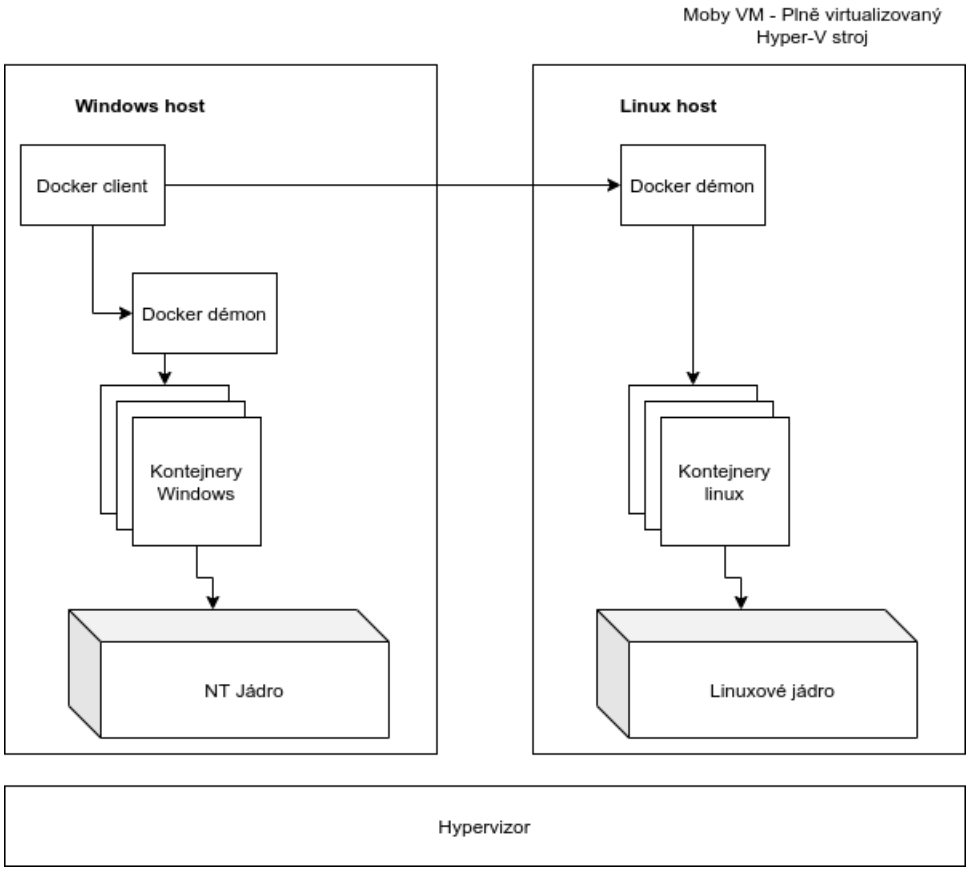
Během posledních let společnost Microsoft Corp. pracovala velmi usilovně, aby integrovala kontejnerovou technologii a Docker do svého operačního systému MS Windows. Dle dokumentace společnosti Microsoft Corp. jsou kontejnery podporovány na Windows 10 od verze 1607 a Windows Server 2016 a novější. Pro dosažení integrace kontejnerů do jádra systému Windows společnost Microsoft Corp. úzce spolupracovala se společnosti Docker Inc. a komunitou.

Základní technologie pro jádro operačního systému Windows, které jsou nezbytné k implementaci kontejnerů jsou kolektivně nazývány *Windows Containers*. Nástroj pro správu těchto *Windows Containers* je Docker. Díky této spolupráci je práce s Docker kontejnery na OS MS Windows téměř totožná jako na OS Linux. Touto cestou se administrátoři zvyklí pracovat s technologií Docker na linuxových distribucích nemusí učit žádné další příkazy nebo postupy.

Běžící kontejner sdílí jádro s operačním systémem na kterém běží. Není tedy možné nativně spustit kontejner, který využívá jádro MS Windows na systému Linux. Obecně tedy lze říci, že kontejner vyžaduje stejné prostředí jako je on sám.

Uvedené však neznamená, že na platformě OS MS Windows nelze spustit linuxový kontejner. Program od společnosti Docker, *Docker for Windows*, tedy nástroj pro spouštění standardních kontejnerů Docker, umí automaticky přepínat mezi *Windows containers* a *Linux containers*. Pokud tedy je třeba spustit kontejner jiného jádra než je hostující systém, je vytvořen virtuální stroj odpovídajícího jádra na kterém jsou kontejnery vytvořeny a spuštěny.

Docker dokáže spouštět linuxové kontejnery pod systémem MS Windows od prvního vydání v roce 2016. Stávající technologie využívá virtualizace linuxového systému na kterém dotazuje API Docker démon procesu, který se stará o vytváření kontejnerů. Linuxové kontejnery tak pod OS Windows možné spustit je, neběží však přímo na hostujícím jádře, ale uvnitř virtuálního stroje. Technologie umožňující spustit na jednom hostovacím systému kontejnery Windows a Linux zároveň je ve vývoji a je možné ji vyzkoušet jako nestabilní část software se nazývá Linux containers on Windows, která je vytvářena pomocí nástrojů LinuxKit. (Poulton, 2018; Microsoft, 2020)



Obrázek 4: Schéma virtualizace strojů pro spuštění kontejnerů rozdílných platforem na OS Windows 10 (Microsoft, 2020)

8.4 Kontejnery na MacOS

Mac OS momentálně kontejnery jako takové nepodporuje vůbec. Je však možné spouštět kontejnery pomocí nástroje *Docker fro Mac*. Tento způsob spouštění kontejnerů je založený na odlehčené virtualizaci linuxového stroje ve kterém jsou spouštěny tyto kontejnery. Navzdory faktu, že se nejedná o nativní řešení, jedná se o extrémně populární řešení pro vývojáře, kteří tak mohou pracovat s kontejnery na operačním systému Mac. (Poulton, 2018)

II. VLASTNÍ PRÁCE

9 Konfigurace serveru

9.1 SSH

9.1.1 Založení uživatele s oprávněním

Secure Shel je hlavní branou ke komunikaci se vzdáleným serverem. K autentifikaci skrze SSH je nejčastěji používáno uživatelské jméno a heslo. Je však doporučeno využití RSA klíče pro zvýšení úrovně zabezpečení. V závislosti na nastavení hesla k certifikátu potom uživatel na autentizovaném stroji nebude muset zadávat heslo, jeho nastavení je opět doporučeno. Aby se mohl uživatel připojit, musí daný uživatel existovat na tomto vzdáleném linuxovém stroji. Založení uživatele bylo provedeno pomocí příkazu *adduser*.

```
$ sudo adduser uzivatelske_jmeno
```

Kód 3: Založení nového uživatele na linuxovém stroji

Příkaz *adduser* vytvoří nového uživatele a skupinu, do které je uživatel přiřazen. Dále je vytvořen domovský adresář pro uživatele, je požadováno zadání hesla a doplňujících informací o uživateli jako například email či telefonní číslo. Po dokončení tohoto jednoduchého textového průvodce se uživatel může přihlásit a volitelně si změnit heslo. V tomto případě je zakládán uživatel jako správce, musí být tedy přiřazen do skupiny *sudo*. Přiřazení do skupiny *sudo* je provedeno stejně, jako do jakékoliv jiné skupiny příkazem *usermod*.

```
$ sudo usermod -aG sudo uzivatelske_jmeno
```

Kód 4: Přiřazení uživatele do skupiny *sudo*

Nyní je uživatel přiřazen do skupiny *sudo* a může tedy spouštět příkazy jako správce. Toto oprávnění je nezbytné pro další správu vzdáleného linuxového stroje.

Dále je třeba veřejný klíč stanice, ze které se bude uživatel ke vzdálenému serveru přihlašovat nahrát na server. Pomocí příkazu z ukázky 5 byl nahrán veřejný klíč na vzdálený server. V závislosti na nastavení, pokud uživatel nastavil heslo pro použití certifikátu bude muset nadále zadávat heslo pro přihlášení na server.

```
$ ssh-copy-id jmeno_uzivatele@server
```

Kód 5: Nahrání veřejného klíče stanice na vzdálený server

9.1.2 Zakázání přihlášení uživatele *root* skrze SSH a změna portu procesu SSH

Pokud je linuxový stroj dostupný z internetu, je zásadním krokem odepřít přihlášení uživateli *root*. Důvodů pro tento krok je několik. Nejdůležitějším je však ten, že

tento uživatel existuje na každém linuxovém stroji. Je tedy třeba uhodnout pouze heslo a tím se dramaticky sníží počet kombinací, které případný útočník musí vyzkoušet.

```
$ sudo tail -n 100 auth.log | grep 'Failed password for root'
Oct 17 11:13:28 sshd[20981]:
 Failed password for root from ...
Oct 17 11:13:30 sshd[20983]:
Failed password for root from ...
Oct 17 11:14:51 sshd[20994]:
Failed password for root from ...
Oct 17 11:16:13 sshd[21007]:
 Failed password for root from ...
Oct 17 11:17:52 sshd[21037]:
 Failed password for root from ...
```

Kód 6: Ukázka množství pokusů přihlásit se jako *root*

Z úkázky 6 lze zjistit, že během posledních 4 minut proběhlo 5 neplatných pokusů o uhodnutí hesla uživatele *root*. Tyto pokusy jsou realizovány množstvím serverů, které spouštějí nejrůznější programy, které prohledávají internet a zkouší zabezpečení strojů, na které narazí. Tyto útoky se liší od takto triviálních, jako prosté vyzkoušení seznamu primitivních hesel na uživatele po velmi komplikované.

Protože služba *Secure Shel* implicitně naslouchá na konkrétním portu, každý útočník přesně ví, kam své pokusy o uhodnutí přihlašovacích údajů směřovat. Změnou portu na kterém SSH služba naslouchá je tedy možné odstínit velké množství těchto pokusů.

9.1.3 Fail2ban

Dosavadními kroky viz. 9.1.2 byl *Secure Shel* pouze odsunut z hlavního ohniska pozornosti. Vzhledem k snadno dostupným nástrojům jako *Nmap*, je změna implicitního portu pro *Secure Shel* jenom formálním krokem. Linuxový nástroj *Nmap* oskenuje stanici a vrátí seznam otevřených portů. Identifikovat konkrétní port, na kterém běží *Secure Shel*, potom vyžaduje od útočníka jen pramálo úsilí, navíc je tento postup často automatizován.

Od další snahy útočníky nijak neodradí ani vypnutí přihlášení pro uživatele *root*. Naopak, lze logicky předpokládat že správce, tedy *root*, bude mít mnohem silnější heslo k prolomení, než obyčejný uživatel. Tento předpoklad lze ověřit i na ukázce 7. Útočník zkouší častá jména v kombinaci s často používanými hesly. Zde se jedná dokonce o více než 9 pokusů za jedinou minutu. Což je mnohem více než pokusů o přihlášení se jako *root*

```
$ sudo tail -n 100 auth.log | grep 'Failed password for invalid user'
Oct 17 13:44:30 dev sshd[22931]:
 Failed password for invalid user catharina from ...
Oct 17 13:44:31 dev sshd[22933]:
```

```
 Failed password for invalid user whitney from ...
Oct 17 13:44:45 dev sshd[22935]:
 Failed password for invalid user svn-user ...
Oct 17 13:45:00 dev sshd[22937]:
 Failed password for invalid user zizhao from ...
Oct 17 13:45:15 dev sshd[22942]:
 Failed password for invalid user sriman from ...
Oct 17 13:45:15 dev sshd[22944]:
 Failed password for invalid user anil from ...
Oct 17 13:45:35 dev sshd[22949]:
 Failed password for invalid user teamspeak ...
Oct 17 13:45:36 dev sshd[22951]:
 Failed password for invalid user service from ...
Oct 17 13:45:53 dev sshd[22953]:
 Failed password for invalid user almacen from ...
```

Kód 7: Ukázka množství pokusů o přihlášení

Linuxový nástroj *Fail2ban* představuje jednu z možností jak blokovat tyto nechtěné pokusy. Tento nástroj pravidelně kontroluje soubor *auth.log*, který obsahuje údaje o pokusech o přihlášení a odkud byly provedeny. Pokud je ze stejné ip adresy uskutečněn daný počet neúspěšných pokusů o přihlášení, je adresa zablokována po nastavenou dobu.

9.2 Firewall

Množství pokusů o zneužití serveru přístupného z internetu není zanedbatelné. Ukázky 6 a 7 zobrazují pouze pokusy o připojení skrze službu *Secure Shel*, obdobné pokusy se však snaží zneužít množství jiných služeb či aplikací. Je proto vhodné uzavřít všechny nevyužívané porty na serveru aby nedošlo k jejich zneužití. Pro účely práce byl použit *firewall ufw*. Pomocí sekvence příkazů z ukázky 8 byly zakázány veškeré příchozí požadavky a povoleny všechny odchozí.

```
$ sudo ufw default deny incoming
$ sudo ufw default allow outgoing
```

Kód 8: Zablokování všech příchozích připojení a povolení odchozích ve službě *ufw*

Vzhledem k podstatě webového serveru je třeba povolit několik portů, které umožní správné fungování webových služeb či připojení skrze *Secure Shel*. V ukázce 9 lze vidět seznam povolených portů pro účely práce.

```
$ sudo ufw status
Status: active

To                         Action      From
--                         ------      ----
```

```
80                         ALLOW       Anywhere
2222                       ALLOW       Anywhere
443                        ALLOW       Anywhere
80 (v6)                    ALLOW       Anywhere (v6)
2222 (v6)                  ALLOW       Anywhere (v6)
443 (v6)                   ALLOW       Anywhere (v6)
```

Kód 9: Seznam povolených portů pro účely práce

9.3 Docker

Celé řešení bude sestávat z několika kontejnerizovaných programů. Pro správu těchto kontejnerů bude použit nástroj *docker*, který je popsán v sekci 8.

Instalace programu *docker* neprobíhá skrze klasické registry. Pro instalaci je třeba navštívit stránku s instalačními instrukcemi na oficiálním webu *docs.docker.com*. Krom návodu na manuální instalaci je možné využít *convenience script*. Tento soubor obsahuje *bash* příkazy, které provedou sérii kroků potřebných k jeho instalaci na zařízení. Po samotné instalaci je třeba všechny uživatele, kteří budou s programem pracovat přidat do stejnojmenné skupiny *docker*. V opačném případě budou muset každý příkaz autorizovat pomocí příkazu *sudo*.

Dále je vhodné pro usnadnění spouštění kontejnerů instalovat nástroj *docker-compose*. Tento nástroj umožňuje vytvářet konfigurační soubory, které slouží jako předpis pro kontejnery a jejich nastavení. Díky těmto konfiguračním souborům není třeba zadávat konfiguraci přímo do příkazu a proces ovládání kontejnerů je mnohem čistší. Pro účely práce byl program *docker* a *docker-compose* instalován následujícími příkazy.

```
$ curl -fsSL https://get.docker.com -o get-docker.sh
$ sudo sh get-docker.sh
$ sudo usermod -aG docker jmeno_uzivatele
$ sudo curl -L "https://github.com/docker/compose \
/releases/download/1.27.4/docker-compose-$(uname -s)-$(uname -m)" \
-o /usr/local/bin/docker-compose
$ sudo chmod +x /usr/local/bin/docker-compose
```

Kód 10: Instalace programů *docker* a *docker-compose*

9.4 Nginx

Reverzní proxi server bude fungovat jako vstupní brána na server. Díky kontejnerizaci je instalace velmi jednoduchá. Nastavení proxi serveru je načítáno z konfiguračních souborů. Pro potřeby aplikace bude později konfigurace obsahovat přesměrování na aplikaci a databázi. Pomocí konfigurace v ukázce 11 bude vytvořen kontejner s *Nginx* proxi serverem který bude odpovídat na http a https dotazy.

```
version: '3'
services:
  nginx:
    image: nginx:1.17.5
    container_name: nginx
    volumes:
    - ./nginx_proxy/default.conf:/etc/nginx/conf.d/default.conf
    ports:
    - 80:80
    - 443:443
    restart: always
```

Kód 11: Konfigurační soubor *docker-compose.yaml*

9.5 Certifikát Let's Encrypt

Autorita Let's Encrypt poskytuje SSL certifikát zcela zdarma. Proces je automatizovaný a tento certifikát lze získat během několika minut. O jeho průběžnou aktualizaci se stará oficiální kontejnerová aplikace *certbot*. Pokud se tedy certifikát blíží k expiraci, je automaticky prodloužen. Pro podporu šifrované komunikace je třeba změnit konfiguraci proxi serveru tak, aby odpovídal na dotazy autorizačního serveru Let's Encrypt na portu http a všechny ostatní http požadavky přesměroval na zabezpečené připojení https. Dále je třeba naslouchat na jednotlivých doménách spolu s nastavenými certifikáty pro připojení klienta. Uvedené je vyjádřeno konfigurací na ukázce 12

```
gzip on;

server {
  listen 80;
  server_name otaklapka.cz;

  location /.well-known/acme-challenge/ {
    root /var/www/certbot;
  }

  location / {
    return 301 https://$host$request_uri;
  }
}

server {
  listen 443 ssl;
  server_name api-notes.otaklapka.cz;
  ssl_certificate /etc/letsencrypt/live/xyz.cz/fullchain.pem;
```

```
    ssl_certificate_key /etc/letsencrypt/live/xyz.cz/privkey.pem;
    include /etc/letsencrypt/options-ssl-nginx.conf;
    ssl_dhparam /etc/letsencrypt/ssl-dhparams.pem;

    ssl_protocols  SSLv2 SSLv3 TLSv1;

    location / {
      proxy_pass http://notes-api:3000;
    }
  }
```

Kód 12: Konfigurační soubor pro reverzní proxi

Pro automatizaci prodlužování certifikátu je vytvořen kontejner z oficiálního obrazu *certbot*. Tento kontejner se bude pravidelně po 12h spojovat s autoritou a dotazovat platnost certifikátu. Pokud se bude blížit expirace, bude obnoven. Kontejner je vytvořen konfiguračním souborem *docker-compose* v ukázce 13

```
version: '3'
services:
  # predesla konfigurace ...
  certbot:
    image: certbot/certbot
    container_name: certbot
    volumes:
    - ./data/certbot/conf:/etc/letsencrypt
    - ./data/certbot/www:/var/www/certbot
    entrypoint: "/bin/sh -c 'trap exit TERM; while :; \
    do certbot renew; sleep 12h & wait $${!}; done;'"
```

Kód 13: Konfigurační soubor *docker-compose.yaml*

10 Analýza požadavků

10.1 Vnější techická specifikace aplikace

Aplikace pro kterou bude navrhována automatizace je *HTTP* serverová aplikace zprostředkující *REST API* pro správu poznámek.

- Jako rozhraní aplikace využívá standardní *REST API* pro úpravu poznámek.
- Komunikace probíhá prostřednictvím *HTTP* dotazů které server obsluhuje.
- Komunikace je formátována pomocí notace *JSON* která zaručuje širokou kompatibilitu.
- Aplikace umožňuje provádět standardní *CRUD* oprace pomocí příslušných metod *HTTP* protokolu - *GET, POST, PUT, DELETE.*
- Databáze s poznámkami je uložena v externím databázovém serveru ke kterému se aplikace připojuje. Tento server je nezbytný pro fungování aplikace.
- Pro ověření klientské aplikace je zde použit pouze jednoduchý API klíč. Aplikace ověřuje přítomnost klíče v hlavičkových parametrech každého dotazu. Pokud Dotaz neobsahuje hlavičku s klíčem, dotaz nebude obsloužen.

10.2 Vnitřní technická specifikace aplikace

Vývoj aplikace probíhá na lokálním stroji. Při vývoji se nevyužívá žádná virtualizace. Každý programátor udržuje své vývojové prostředí kompatibilní s aplikací. Tento způsob je velmi jednoduchý. Vzhledem k nízkým nárokům aplikace vyplývajících z její velikosti je lokální vývoj naprosto dostačující.

- Transpilovaný zdrojový kód aplikace je v jazyce *JavaScript*. Pro její spuštění je třeba nastavené prostředí s interpreterem *Node.js*.
- Při programování aplikace je využito množství knihoven třetích stran. Pro jejich správu a verzování je použit základní nástroj pro správu balíčků *NPM (Node Package Manager)*.
- Pro pohodlí při vývoji a minimalizaci neočekávaného chování je kód aplikace napsán v jazyce *TypeScript*, který je nadmnožinou jazyka *JavaScript* a je nutné ho transpilovat, aby byl spustitelný interpreterem *Node.js*.
- Kód aplikace obsahuje množství testů. Tyto testy jsou spouštěny pomocí správce balíčků *NPM*. Pro tvorbu testů je použita testovací knihovna *Jest*.

10.3 Požadavky na produkční prostředí aplikace

Požadavky, které aplikace klade na provoz v produkčním prostředí, se mírně liší od požadavků při vývoji. Aplikace v této fázi již prošla otestováním a sestavením, množství technologií tedy již není v produkčním prostředí potřeba.

- Pro běh aplikace je třeba prostředí s instalovaným interpreterem *Node.js*
- Aplikace pro správné fungování a konfiguraci využívá množství proměnných, které musí být přítomny v běhovém prostředí.
- Jako zdroj dat aplikace využívá externí databázi. Přístup k této databázi musí být umožněn pro správné fungování.
- Ze svojí podstaty aplikace musí být dostupná z internetu pro uživatele, kteří ji budou používat.

10.4 Požadavky na řešení průběžné integrace a nasazení

Následují požadavky na vlastní řešení. Tyto požadavky představují požadavky hypotetického klienta. Výčet požadavků neobsahuje implicitní požadavky na bezpečnost či dostupnost.

- Provoz řešení musí být zdarma
- Řešení nesmí být limitováno
- Řešení musí zajišťovat autonomní cyklus od otestování po doručení na cílový server

Implementace průběžné integrace

11 Volba SCM

Základním kamenem výstavby průběžné integrace je nástroj pro správu verzí kódu. Kód aplikace se typicky rozpadá do jednotlivých větví. Každá větev obsahuje změny v kódu, které spolu logicky souvisí. Typicky větev představuje novou funkcionalitu či opravy. Pokud je práce na takové větvi dokončena, je třeba ji integrovat do hlavní větve.

Pro správu verzí zdrojového kódu aplikace existují dvě široce rozšířená řešení. Prvním a zřejmě nejpoužívanějším je *Git*. Druhým řešením je *Apache Subversion* - SVN. Obě řešení se liší v přístupu k verzovanému kódu. Rozdílná metodologie určuje výhody i nevýhody řešení, je třeba zhodnotit potřeby aplikace pro kterou je řešení určeno. Následující tabulka 1 se zabývá porovnáním vlastností obou řešení.

Funkcionalita	**Git**	**SVN**
Interakce	Chová se jako server i client. Každý vývojář má úplnou historii projektu na lokálním stroji. Git provádí změny lokálně, vývojář nemusí být po celou dobu online.	SVN má oddělený server a client. Pouze soubory, se kterými vývojář pracuje jsou dostupné lokálně. Ke komunikaci se serverem je třeba být online.
Větvení	Větve jsou odkazem na určitý *commit*. Takové řešení je velmi odlehčené a silné.	Každá větev je adresářem v repozitáři. Adresářová struktura je velmi náročná na integraci s hlavní větví kódu.
Autorizace	Git jako takový přepodkládá stejné oprávnění pro všechny vývojáře. Vzdálený repozitář však může autorizovat akce nad větvemi při synchronizaci.	Je možné nastavit oprávnění na úrovni složek i souborů.
Historie úprav	Ze své distribuované podstaty je možné, aby kdokoliv ovlivnil historii. Nepovolené změny je možné filtrovat na úrovni vzdáleného repozitáře.	Ke změně v historii je třeba přístupu na centrální server a změny jsou sledovány na úrovni souborů.
Náročnost na úložiště	Git je velice neefektivní k ukládání velkých binárních souborů.	SVN krom kódu dokáže efektivně spravovat i velké binární soubory.
Licence	GNU (General public license)	Apache License
Oblíbenost	Široce používán.	Spíše ojediněle využíváno. Tento propastný rozdíl lze ověřit například na webu *trends.google.com*

Tabulka 1: Porovnání funkcionalit Git a SVN

Pro danou aplikaci byl po zvážení kladů a záporů jednotlivých řešení zvolen SCM

Git. Hlavními faktory byly: interakce, větvení a oblíbenost.

11.1 Implementace *git*

Pro operační systém na bázi linuxu je instalace snadná, stačí jej instalovat standardní cestou ze správce balíčků. Pro platformu Mac je potom *git* dostupný skrze volitelného správce balíčků *homebrew* či jako binární instalační program. Pro Windows *git* existuje jako binární instalační program a jako přenositelná *portable* verze. Pro účely této práce byla používána linuxová stanice, SCM *git* byl tedy instalován příkazem z ukázky 14

```
$ sudo apt install git
```

Kód 14: Instalace SCM *git* na linuxové stanici

12 Volba serveru pro hostování vzdáleného repozitáře

Pro synchronizaci lokální práce s ostatními pracovníky je třeba vzdálený repozitář. Vzdálený repozitář obsahuje lokální verze které na něj byly odeslány. Zde jsou dostupné pro ostatní členy týmu.

Vzdálený repozitář dále poskytuje nástroje k další kontrole kódu. V závislosti na implementaci jednotlivých poskytovatelů se může soubor nástrojů lišit. Některé nástroje mohou být zpoplatněny. Je proto na zvážení, který poskytovatel je pro daný projekt nejvhodnější.

	Gitlab	**GitHub**	**BitBucket**
Počet kolegů	neomezeno	neomezeno	5
Počet minut CI/CD	2000 min	2000 min	50 min
Velikost úložistě	10 GB	500 MB	1 GB
Pages	ANO	ANO	ANO
Správa oprávnění	ANO	ANO	ANO

Tabulka 2: Porovnání nabízených služeb pro plán zdarma (data aktuální k 19.6.2020)

Tabulka 2 porovnává stěžejní vlastnosti nabídek vzdáleného repozitáře pro zvolený projekt. Bylo zvoleno několik nejznámějších poskytovatelů služeb. Pro srovnání bylo bráno v úvahu řešení SaaS - *Software as a service*.

Z tabulky 2 lze vyvodit, že nejvhodnější řešení, pro uvažovaný projekt nabízí služba *gitlab.com*. Tato služba také umožňuje registrovat vlastní *runner*, tedy program, který provádí úkoly zadané v *pipelines*. Tímto způsobem je možné odstranit omezení minut pro CI/CD i v plánu bez měsíčních poplatků. Toto řešení je popsáno v kapitole 13.

12.1 Implementace vzdáleného repozitáře

Za nejvhodnější volbu vzdáleného repozitáře byl zvolen server *gitlab.com*. Prvním krokem musí tedy být registrace uživatele na tomto serveru. Registrace poskytuje standardní možnosti využití některého z existujících účtů, jako například Google nebo vyplněním formuláře. Po přihlášení do systému uživatel vidí všechny své projekty spolu s možností založení dalších. Pro vytvoření nového projektu je třeba vyplnit několik nastavení a informací. Pokud je projekt označen za soukromý, nebude veřejně dostupný a jediná cesta k jeho zobrazení je přihlášení spolu s oprávněním od vlastníka.

Po založení je projekt identifikován odkazem. Odkaz na projekt existuje ve dvou formách, SSH odkaz a HTTPS odkaz. Pro využití SSH odkazu je nejprve třeba registrovat do repozitáře vygenerovaný veřejný klíč vývojové stanice. Přidání

veřejného klíče opět probíhá vyplněním formuláře a uložením. HTTPS odkaz lze využít ihned bez dalších kroků, při jeho použití je však třeba se autentifikovat kombinací jména a hesla. Naproti tomu SSH odkaz po registraci veřejného klíče bude vyžadovat pouze heslo k certifikátu je-li nastaveno. Pro účely práce je využit SSH odkaz. Dále je třeba spojit lokální složku se vzdáleným repozitářem. Spojení bylo provedeno sekvencí příkazů z ukázky 15

```
$ git init
$ git remote add origin git@...
```

Kód 15: Inicializace repozitáře

13 Volba CD serveru

Většina serverů pro správu vzdáleného repozitáře též nabízí CI/CD řešení. V tabulce 2 jsou uvedeny jednotlivé omezení. Vzhledem k požadavkům definovaným v sekci 10.4 však nejsou tato řešení dostupná protože obsahují limity pro bezplatné řešení.

Vedle poskytovatelů SaaS z tabulky 2 je další možností hostovat server průběžné integrace a doručení na vlastní platformě. Tato možnost s sebou nese jistá rizika, jako jsou problémy spojené s údržbou, zabezpečení či náročnost na zdroje serveru. Hlavní praktickou výhodou takového řešení potom je, že infrastruktura nemusí být připojená k extranetu. V situaci kdy cílový server je odříznutý od internetu, může takové řešení být vhodné.

Nejznámějšími zástupci serverů provozovaných na vlastní infrastruktuře jsou server Jenkins a GitLab CE. Produkt GitLab CE poskytuje stejné nástroje jako verze SaaS, odpadávají však omezení na zdrojích, které klade poskytovatel. Velikost kontejnerových registrů či minuty běhu *runners* jsou tedy limitovány pouze platformou na které běží. Jenkins, na rozdíl od produktu GitLab CE, neposkytuje integrované SCM či správu úkolů a jiné. Slouží pouze jako server průběžného doručení.

	Gitlab CE	**Jenkins**
Minimální RAM	4 GB	200 MB

Tabulka 3: Porovnání minimálních nároků

Z tabulky 3 lze zamítnout produkt GitLab CE. Server zvolený pro uvažovanou aplikaci nesplňuje minimální požadavky pro provoz GitLab CE. Jedinou možností z kandidátů bez omezení provozu tedy zůstává server Jenkins.

Další možností je obejít omezení serveru GitLab ve verzi SaaS pomocí zapojení vlastní infrastruktury k provádění průběžného doručení. V tomto řešení klient připojí vlastní server pomocí programu *runner*, který provádí úkony jednotlivých *pipeline*. Program *runner* potom běží přímo na serveru a využívá k provádění instrukcí zdroje serveru na kterém běží.

Obě uvedená řešení jsou pro projekt možná. Server Jenkins i řešení s GitLab *Runners* splňují požadavky. Budou tedy vyhotovena obě řešení a posléze určeno doporučené.

Pro provoz serveru Jenkins a posléze i provoz programu *runner* bude využit stejný server, který je považován za cílový. Takové řešení — využít kapacitu produkčního serveru k CI/CD není vhodné pro opravdové aplikace. Pro účely této práce by bylo zbytečné provozovat dva oddělené servery, jelikož se jedná o demonstrativní řešení.

13.1 Implementace CD pomocí serveru Jenkins

13.1.1 Instalace Jenkins serveru

Jenkins server je dostupný jako kontejnerová aplikace z oficiálního registru Docker Hub. Nejprve je tedy třeba upravit stávající soubor *docker-compose.yml* aby obsahoval konfiguraci pro spuštění nového kontejneru se serverem Jenkins. Nový záznam v konfiguraci je zobrazen v ukázce 16

```
jenkins:
  image: jenkins/jenkins:lts
  container_name: jenkins
  restart: always
  ports:
    - 8081:8080
    - 50000:50000
  # povoleni zanorenych kontejneru
  privileged: true
  user: jenkins
  volumes:
    - ./jenkins_home:/var/jenkins_home
    # namapovani vnejsiho dockeru do kontejneru
    - /var/run/docker.sock:/var/run/docker.sock
    - /usr/bin/docker:/usr/bin/docker
```

Kód 16: Nový záznam v souboru *docker-compose.yaml* s předpisem pro Jenkins

Záznam z ukázky 16 vytvoří nový kontejner podle oficiálního obrazu pro server Jenkins ve verzi *lts — long term support* tedy dlouhodobě podporované. Tento kontejner je spouštěn jako privilegovaný a je mu namapován domovský adresář, v kterém ukládá informace o doplňcích a další konfiguraci, která se tak neztratí pokud je kontejner restartován. Dále je třeba namapovat *socket* služby *docker*, takto kontejner může vytvářet další kontejnery, ve kterých budou spouštěny jednotlivé sekvence průběžného doručení. Ukázková aplikace pro sestavení bude potřebovat prostředí s platformou NodeJS. Tímto způsobem si server Jenkins, který samotný běží v kontejneru, může vytvořit další kontejner s požadovaným prostředím NodeJS, do kterého nahraje potřebné soubory a spustí nad nimi zadané operace.

Po prvním spuštění je vygenerován ve složce přiřazeného adresáře soubor s heslem, které je vyžadováno po prvním navštívení webového rozhraní serveru. Toto heslo je jednorázové a slouží pouze aby server nebyl zneužit po prvním spuštění, kdy neobsahuje žádné uživatele. Po přihlášení je spuštěn krátký průvodce nastavením serveru. Po dokončení průvodce prvního spuštění, kde si administrátor vytvoří heslo a upraví potřebné nastavení lze server začít používat.

13.1.2 Spuštění nasazení po přidání nových úprav do hlavní větve

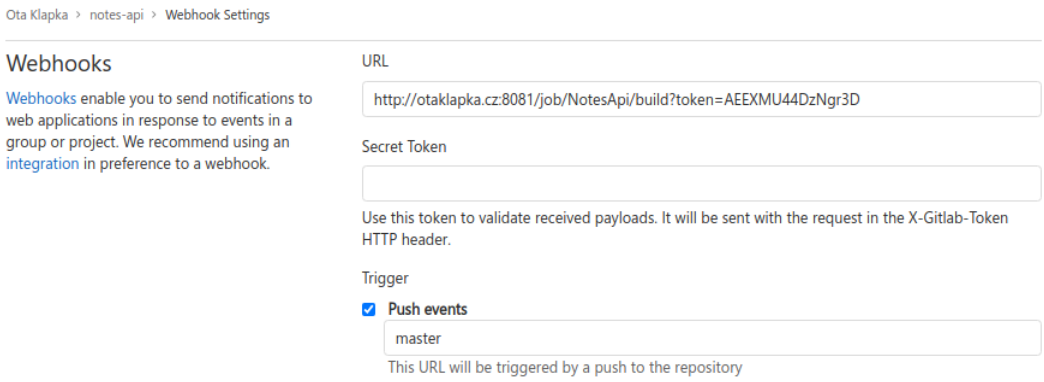
Server Jenkins umožňuje vytvořit automatizaci několika různými způsoby. Pro potřeby aplikace je třeba několika různých prostředí, která budou realizována kon-

tejnery. Těmto požadavkům vyhovuje možnost *pipelines*. Bohužel ve stávající verzi Jenkins neumožňuje napojení *pipeline* na změny provedené v repozitáři průběžné integrace. Tato možnost je prozatím pouze pro *freestyle project*.

Aby bylo dosaženo požadavku na automatické spuštění po vložení změn do hlavní větve projektu v repozitáři, je třeba nejprve vytvořit *freestyle project*. Spuštění je potom realizováno pomocí *webhooks*. Na obrázku 5 je nastavení pro *freestyle project*, které spustí daný úkol po provolání adresy spolu s tokenem.

Obrázek 5: Nastavení pro *webhook* na serveru Jenkins

Dále je třeba nastavit server vzdáleného repozitáře aby po příslušné akci zavolal server průběžného doručení. Tedy aby repozitář GitLab notifikoval server s Jenkins. Tato notifikace je zprostředkována pomocí *webhooks*. Pro aplikaci byla jako spouštěcí akce zvoleno přidání změn do hlavní větve projektu. Uvedené je reprezentováno nastavením na ukázce 6.



Obrázek 6: Nastavení pro *webhook* na serveru gitlab.com

Posledním krokem pro tento projekt je zvolení práce, která bude spuštěná po zavolání ze serveru průběžné integrace. V tomto případě bude projekt fungovat jen jako spouštěč pro *pipeline*, která vykoná hlavní práci. Projekt obsahující hlavní práci lze referovat pomocí nastavení z obrázku 7

Obrázek 7: Nastavení pro spuštění hlavního *pipeline* projektu

13.1.3 Provedení vlastního procesu

Hlavní práce je vykonávána v *pipeline* projektu. Projekt je rozdělen do několika fází, které na sebe navazují.

Příprava prostředí pro běh

Aplikace vyžaduje několik proměnných obsahující přístupové údaje do databáze a další nastavení. Tyto informace jsou obsaženy v proměnných prostředí, ze kterých si je aplikace získá. Tyto proměnné musí být dostupné při sestavování a testování aplikace a tedy přímo v průběhu *pipeline*. Proměnné prostředí jsou obsaženy v bloku *environment*. Na ukázce 17 jsou proměnné vypuštěny protože obsahují citlivé informace.

```
environment {
  ... environmentalni promenne pro aplikaci
}

agent {
  docker { image 'node:13' }
}
```

Kód 17: Kód *pipeline* pro nastavení prostředí

Aplikace k sestavení, otestování a spuštění vyžaduje NodeJS. Blok *agent* definuje, že daný proces má být spuštěn v kontejneru, který obsahuje instalovaný a nastavený NodeJS ve verzi 13. Tato sekce vyžaduje přístup k *docker* socketu, aby mohl být takový kontejner vytvořen.

Získání aktuálních kódů

Dalším krokem je získání zdrojového kódu aplikace k sestavení. Stažení aktuálních kódů je provedeno doplňkem pro server Jenkins *git*. Tento doplněk se připojí k repozitáři a pomocí přihlašovacích údajů z nastavení autorizuje požadavek na stažení zdrojového kódu. Samotné provedení stažení aktuálních změn je provedeno sekvencí na ukázce 18.

```
stage('Cloning Git') {
  steps {
    git 'https://gitlab.com/otaklapka/notes-api.git'
  }
}
```

Kód 18: Kód *pipeline* pro stažení aktuálních kódů

Otestování aplikace

Podmínkou pro další postup je průchod všech automatických testů. Pokud některý z testů selže, aplikace se nechová dle očekávání a nesmí být doručena na produkční server. Sekce otestování je proto prováděna jako první v pořadí všech úkonů. Před otestováním musí být staženy veškeré dependence pro aplikaci. Stažení dependencí je provedeno pomocí nástroje *npm*, který je rovněž obsažen v kontejneru, který byl vytvořen pro běh této *pipeline*. Samotné otestování aplikace probíhá po stažení všech závislostí opět skrze *npm* a předem definovaného skriptu. Průběh testovací fáze je popsán na obrázku 19.

```
stage('Testing code') {
  steps {
    sh 'npm install'
    sh 'npm run test'
  }
}
```

Kód 19: Kód *pipeline* pro otestování aplikace

Sestavení Docker obrazu

Pokud je otestování aplikace úspěšné, aplikaci je třeba zabalit do kontejneru, který bude posléze doručen na cílový server a spuštěn. Ovládání programu *docker* je zde opět prováděno pomocí rozšíření pro server Jenkins. Proměnná *registry* z obrázku 20 obsahuje *url* adresu Docker Hub, který je zde použit jako repozitář pro obrazy verzí aplikace.

```
stage('Building image') {
   steps{
     script {
       dockerImage = docker.build registry + ":$BUILD_NUMBER"
     }
   }
}
```

Kód 20: Kód *pipeline* pro sestavení Docker obrazu

Uložení obrazu na DockerHub

Po sestavení obrazu kontejneru s fungující aplikací uvnitř, je třeba ho uložit do registru. Tento proces je velmi usnadněný doplňkem pro manipulaci s *docker*. V sekci nastavení běhového prostředí byla vytvořena proměnná obsahující údaje pro přihlášení do Docker Hub. Nyní je předána doplňku jako proměnná *registryCredential*. Proces odeslání sestaveného obrazu do registrů je ukázán na obrázku 21.

Navazujícím procesem je odstranění vytvořeného obrazu aby bylo uvolněno místo pro další sestavení.

```
stage('Deploy Image') {
  steps {
    script {
      docker.withRegistry( '', registryCredential ) {
        dockerImage.push()
      }
    }
  }
}
stage('Remove docker image') {
  steps {
    sh "docker rmi $registry:$BUILD_NUMBER"
  }
}
```

Kód 21: Kód *pipeline* pro odeslání obrazu na Docker Hub a následné odstranění lokálního obrazu

Nasazení a spuštění kontejneru na server

Poslední fází je doručení sestaveného kontejneru z registrů Docker Hub na cílový server. Proces nasazení kontejneru má 3 fáze.

1. Připojení na cílový server pomocí SSH. Spojení je provedeno pomocí doplňku pro server Jenkins.
2. Zastavení aktuálně běžícího kontejneru a jeho odstranění. Tento krok může selhat v případě, že kontejner s předchozí verzí neexistuje. Chyba vzniklá v tomto kroku bude ignorována.
3. Přihlášení se do Docker Hub a spuštění aktuální verze kontejneru. Veškeré běhové proměnné, které aplikace potřebuje, jsou dodány z CD serveru, kde jsou zabezpečeny.

Všechny tyto fáze jsou obsaženy v kódu *pipeline* v ukázce 22

```
stage('Running docker on otaklapka.cz server') {
  steps {
    script {
      withCredentials([
        usernamePassword(
          credentialsId: 'ssh-otaklapka.cz',
          usernameVariable: 'sshUser',
          passwordVariable: 'sshPasswd'
```

```
        ),
        usernamePassword(
          credentialsId: 'dockerhub',
          usernameVariable: 'dhUser',
          passwordVariable: 'dhPasswd'
        )
      ]) {
        def remote = [:]
        remote.port = 2222
        remote.name = 'otaklapka.cz'
        remote.host = 'otaklapka.cz'
        remote.user = sshUser
        remote.password = sshPasswd
        remote.allowAnyHosts = true
        try {
          sshCommand
            remote: remote,
            command: "docker stop ${env.containerName}"
          sshCommand
            remote: remote,
            command: "docker rm ${env.containerName}"
        } catch(err) {
          sh "echo 'No previous container'"
        }
        sshCommand
          remote: remote,
          command: "docker login \
          --username ${dhUser} --password ${dhPasswd}"
        sshCommand
          remote: remote,
          command: "docker run -d -p 3030:3000
          -e NODE_ENV='${env.NODE_ENV}'
          -e PORT=${env.PORT}
          -e CORS_URL='${env.CORS_URL}'
          -e DB_NAME='${env.DB_NAME}'
          -e DB_HOST='${env.DB_HOST}'
          -e DB_PORT=${env.DB_PORT}
          -e DB_USER='${env.DB_USER}'
          -e DB_USER_PWD='${env.DB_USER_PWD}'
          -e API_KEY='${env.API_KEY}'
          --name ${env.containerName} \
          ${registry}:${env.BUILD_NUMBER}"
      }
    }
  }
}
```

Kód 22: Kód *pipeline* pro nasazení a spuštění kontejneru na cílovém serveru

13.1.4 Zhodnocení řešení

Vytvoření automatizace procesu doručení pomocí serveru Jenkins pro uvažovanou aplikaci je funkční. Mezi klady řešení patří usnadnění práce pomocí doplňků. Ty významně zkracují zápis a složitost řešení oproti klasickým *shell* příkazům. Jenkins je v komunitě již dlouhou dobu a lze tedy nalézt velké množství návodů a příkladů řešení.

Doplňky, které tvoří podstatu serveru Jenkins však mají i řadu nevýhod. Tou první je dokumentace. Každý doplněk je samostatný projekt udržovaný komunitou. Za jeho dokumentaci a co je nejdůležitější, bezpečnost, je potom odpovědný vlastník onoho projektu. Dokumentace je tak ne vždy vyčerpávající.

Dále lze zmínit nekonzistentnost nástrojů. Použitý projekt *pipelines* nepodporuje některé elementární možnosti jako spuštění pomocí *webhooks* a je nutné ho kombinovat s projekty jiného druhu pro dosažení této funkcionality. [2]

Závěrem Jenkins poskytl všechny potřebné nástroje k vytvoření funkčního řešení. Postup řešení je přehledný a snadno rozšiřitelný.

13.2 Implementace CD s využitím GitLab *Runners*

Systém GitLab umožňuje pro spouštění CI/CD sekvencí využít vlastní serverové kapacity i ve verzi SaaS. Podmínkou je pouze připojení k internetu. K využití vlastních prostředků je třeba kompatibilní rozhraní se serverem GitLab. Toto rozhraní pro instrukce je sprostředkováno pomocí *runners*. *Runner* je *open-source* program vytvořený společností GitLab. Jeho instalace je možná buď přímo na zařízení, protože je napsaný v jazyce Go, jedná se o standardní spustitelný program nebo jako obraz kontejneru. V tomto řešení byl využit oficiální kontejnerový obraz dostupný na serveru DockerHub. S využitím vlastní kapacity pro provoz *runner* lze tímto způsobem tedy obejít limit CI/CD času.

13.2.1 Vytvoření kontejneru s GitLab Runner

GitLab *Runner* je dostupný jako obraz kontejneru na serveru Docker Hub. K vytvoření kontejneru byla upravena konfigurace pro *docker-compose* s údaji na ukázce 23. Tato sekvence vytvoří a spustí kontejner s instalovaným a spuštěným programem *runner*.

```
gitlab-runner:
  image: gitlab/gitlab-runner:alpine
  container_name: gitlab-runner
  volumes:
  - ./gitlab-runner/config:/etc/gitlab-runner
```

[2] Obdobná funkcionalita může být přidána pomocí doplňku. Ten však pouze pozastaví běh již spuštěného procesu a vyčkává na dotaz. Nejedná se tedy o opravdové spuštění skrze *webhook*.

```
    # namapovani lokalniho \textit{docker} soketu
    - /var/run/docker.sock:/var/run/docker.sock
```

Kód 23: Úprava souboru *docker-compose.yml* s údaji pro nový kontejner

Kontejner bude opět vytvářet další kontejnery a je tedy třeba mu předat lokální *docker* soket. Dále je kontejneru namapována složka pro uložení konfiguraci vně kontejneru, aby zůstala perzistována i po restartování kontejneru.

Úpravy konfigurace

Při prvním spuštění kontejneru je třeba dokončit textového průvodce nastavením pro *runner*. Toto nastavení lze kdykoliv později upravit v konfiguračním souboru. Nastavení *runner* vytvořeného pro účely práce je na ukázce 24.

```
  [[runners]]
    name = "otaklapka.cz-runner"
    url = "https://gitlab.com/"
    token = "secret_token"
    executor = "docker"
    [runners.custom_build_dir]
    [runners.cache]
      [runners.cache.s3]
      [runners.cache.gcs]
      [runners.cache.azure]
    [runners.docker]
      tls_verify = false
      image = "alpine"
      privileged = true
      disable_entrypoint_overwrite = false
      oom_kill_disable = false
      disable_cache = false
      volumes = ["/cache", "/var/run/docker.sock:/var/run/docker.sock"]
      shm_size = 0
```

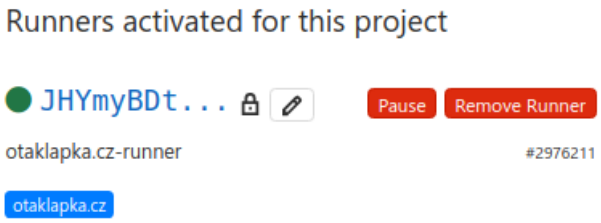
Kód 24: Soubor konfigurace pro *runner*

Z ukázky 24 je patrné, že každý *runner* je třeba pojmenovat. Dále je třeba ho přiřadit serveru, na kterém má být využit, v tomto případě SaaS verzí GitLab na serveru *gitlab.com*. Bezpečnostním prvkem je zde token, který je vygenerován na serveru a zadán do *runner*. Jako exekutor byl zvolen *docker* s implicitním obrazem kontejneru alpine. Aby byl *runner* schopen autonomně vytvářet a spravovat kontejnery, musí běžet v privilegovaném módu a je mu opět namapován soket vnějšího *docker*.

S tímto nastavením je *runner* připraven a jakmile bude připravena práce, ujme se jí.

13.3 Registrace *runner* do serveru

Aby *runner* mohl komunikovat a přijímat práci od serveru, je třeba je spojit. Toto spojení je provedeno odkazem na server v konfiguraci pro *runner* a bezpečnostním tokenem. Odkaz i bezpečnostní token je již vygenerován na stránce s nastaveními pro *runner*, stačí tak informace pouze zkopírovat. Registrovaný a připravený *runner* je zobrazen na ukázce 8



Obrázek 8: Zaregistrovaný *runner* na serveru GitLab

13.4 Vytvoření *pipelines* kódu automatizace

Pipelines na serveru GitLab jsou ovládány pomocí speciálního souboru v kořenovém adresáři repozitáře. Soubor je třeba pojmenovat *.gitlab-ci.yml* (dle základního nastavení) a jeho vnitřní struktura je ve formátu YAML. Běh a výsledek jednotlivých *pipeline* potom lze kontrolovat v grafickém uživatelském rozhraní. Toto prostředí obsahuje množství kontrolních prvků jako například opětovné spuštění či prokliknutí do kontroly výstupu jednotlivých sekcí.

Příprava běhového prostředí

Začátek instrukcí *pipeline* tvoří seznam kroků, které seskupují instrukce. Následuje určení obrazu kontejneru, který bude implicitně použit, pokud není v jednotlivých krocích definováno jinak. V tomto případě se opět jedná o kontejner s instalovaným prostředím pro NodeJs. Dále je vhodné definovat složku, kde bude uložena *cache* pěť. Tam je třeba uložit soubory knihoven, které jsou stahovány při každém sestavení aplikace, pokud se mezi verzemi nezměnily, budou použity již stažené.

```
stages:
  - build
  - test
  - build_image
  - deploy_prod

image: node:13
default:
  tags:
    - otaklapka.cz
cache:
  paths:
```

```
    - node_modules/
```

Kód 25: Kód *pipeline* pro přípravu běhového prostředí

Sestavení aplikace

Prvním krokem je opět sestavení aplikace. Je třeba stáhnout všechny dependence aplikace pomocí manažera balíčků pro NodeJS. Pokud sekvence nemá explicitně zvolený obraz kontejneru ve kterém se má spouštět, bude zvolen obraz implicitního kontejneru, kterým v tomto případě je node:13. Dostupnost programů *node* a *npm* je tedy zajištěna.

```
build:
  stage: build
  script:
    - npm install
    - npm run build
```

Kód 26: Kód *pipeline* pro sestavení aplikace

Otestování aplikace

Dalším krokem je otestování verze aplikace. Pokud tento, či předchozí krok selže, nemá smysl pokračovat a *pipeline* bude zastavena. Testování aplikace probíhá spuštěním automatických testů, které jsou předhotoveny.

```
test:
  stage: test
  script:
    - npm run test
```

Kód 27: Kód *pipeline* pro otestování aplikace

Vytvoření kontejneru

Tato fáze se liší od verze kde byl použit server Jenkins v použití registrů. GitLab nabízí vlastní registry pro verzování kontejnerů. V proměnné prostředí *CI_REGISTRY* je v tomto případě uložen odkaz na registry GitLab, nikoliv Docker Hub, který byl využit ve verzi se serverem Jenkins. Použití registrů na serveru GitLab je mnohem velkorysejší oproti Docker Hub. GitLab registry nejsou omezené počtem privátních repozitářů a každý projekt je může využívat.

Další změnou oproti předchozímu řešení je potřeba zajistit spuštění této fáze pouze pro hlavní větev. Zde je *pipeline* spouštěna při všech změnách repozitáře. Předchozí části jsou žádoucí spustit pro změnu v jakékoliv větvi, protože identifikují nefunkční kód. Každé změny, které jsou odeslány do repozitáře jsou tak

zkontrolovány a vývojář je informován o výsledku integrace. Vytvořit kontejner s aplikací je však žádoucí pouze z hlavního kódu, tedy na větvi *master*. Tato podmínka je definována blokem *only* na ukázce 28.

```
build_image:
  image: docker:19.03.12
  stage: build_image
  only:
    - master
  script:
    - >
      docker login
      -u $CI_REGISTRY_USER
      -p $CI_REGISTRY_PASSWORD $CI_REGISTRY
    - >
      docker build
      -t $CI_REGISTRY/otaklapka/notes-api:$CI_COMMIT_SHORT_SHA .
    - >
      docker push
      $CI_REGISTRY/otaklapka/notes-api:$CI_COMMIT_SHORT_SHA
```

Kód 28: Kód *pipeline* pro vytvoření obrazu kontejneru a uložení do registrů

Spuštění kontejneru na cílovém serveru

Posledním krokem je nasazení nové verze kontejneru s aplikací na cílový server. Tato operace je opět omezena pouze na hlavní větev pomocí bloku *only*.

Pro tuto část byla zvolena funkce manuálního spuštění. To znamená, že tato sekvence bude spuštěna po kliknutí na tlačítko v přehledu *pipeline*. Pro tuto sekvenci je manuální spuštění nejvhodnější. Správce tak může vložit změny do hlavního adresáře, zkontrolovat průběh sestavení, otestování a vytvoření obrazu kontejneru mimo nasazovací okno domluvené s hypotetickým klientem. Pokud dojde k chybám, je tak prostor na opravy a doladění. Později v čase, který je vymezen k aktualizaci aplikace, je spuštěna poslední sekvence, která pouze nasadí kontejner na server.

Nejprve v sekci *before_script* proběhne ověření dostupnosti SSH klienta, který je potřebný k připojení na vzdálený server. Pokud v kontejneru chybí, bude nainstalován a poté spuštěn. K povolení přístupu na server je do SSH klienta v kontejneru vložen veřejný klíč, vygenerovaný pro uživatele existujícího na vzdáleném serveru. Veřejný klíč existujícího uživatele je uložen jako proměnná prostředí a je v *pipeline* k dispozici pod proměnnou *SSH_PRIVATE_KEY*. Tímto způsobem se může kontejner vytvořený pomocí *gitlab-runner* připojit skrze zabezpečený tunel k vzdálenému serveru a spouštět na něm příkazy.

Dále následuje sekvence příkazů spouštěná na cílovém serveru. Nejprve je učiněn pokus o zastavení běžícího kontejneru. Tento příkaz může skončit chybou, která

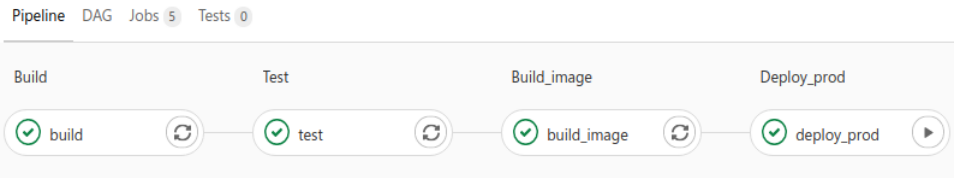
může být ignorována. Obdobně tomu je u pokusu o následné odstranění obrazu kontejneru s předchozí verzí aplikace. Následuje nasazení nové verze kontejneru.

```
deploy_prod:
stage: deploy_prod
  only:
    - master
  when: manual
  before_script:
    - >
    'which ssh-agent
    || ( apt-get update -y && apt-get install openssh-client -y )'
    - eval $(ssh-agent -s)
    - >
      echo "${SSH_PRIVATE_KEY}"
      | tr -d '\r' | ssh-add - > /dev/null
    - mkdir -p ~/.ssh
    - chmod 700 ~/.ssh
    - >
      '[[ -f /.dockerenv ]] && echo
      -e "Host *\n\tStrictHostKeyChecking no\n\n" > ~/.ssh/config'
  script:
    - >
      ssh -p 2222 gitlab@otaklapka.cz "docker stop gitlab-notes-api"
      || true
    - >
      ssh -p 2222 gitlab@otaklapka.cz "docker rm gitlab-notes-api"
      || true
    - >
      ssh -p 2222 gitlab@otaklapka.cz "docker run -d -p 3030:3000
      -e NODE_ENV='${NODE_ENV}'
      -e PORT=${PORT}
      -e CORS_URL='${CORS_URL}'
      -e DB_NAME='${DB_NAME}'
      -e DB_HOST='${DB_HOST}'
      -e DB_PORT=${DB_PORT}
      -e DB_USER='${DB_USER}'
      -e DB_USER_PWD='${DB_USER_PWD}'
      -e API_KEY='${API_KEY}'
      --name gitlab-notes-api
      $CI_REGISTRY/otaklapka/notes-api:${CI_COMMIT_SHORT_SHA}"
```

Kód 29: Kód *pipeline* pro nasazení kontejneru na server

13.4.1 Zhodnocení řešení

Řešení implementace průběžné integrace a doručení pomocí serveru GitLab je velmi pohodlné. Formát YAML je přehledný a při prvním pohledu je z kódu *pipeline* vidět co se provádí. Dokumentace GitLab je přehledná a ucelená. Kvůli velké oblíbenosti lze po internetu nalézt množství hotových řešení. Samotné webové rozhraní, ve kterém je možné soubor instrukcí pro *pipeline* upravovat, nabízí několik příkladů použití. GitLab *pipelines* podporuje velké množství instrukcí, se kterými lze pokrýt rozmanité případy použití.



Obrázek 9: Úspěšný průběh *pipeline*, poslední fáze s manuálním tlačítkem

13.5 Porovnání obou řešení a volba

Řešení pomocí serveru GitLab je po všech stránkách přívětivější než server Jenkins. Rozhraní je citelně estetičtější a přehlednější. Dokumentace je ucelenější a především na jednom místě. Není zde žádná syntetická syntaxe z doplňků, všechny příkazy jsou *shell*. GitLab nabízí ucelené řešení pro automatizovanou integraci i doručení na jednom místě. Poskytuje zdarma i registry pro verzování obrazů sestavených kontejnerů. Vše je tak ve stejném přehledu a není třeba udržovat oprávněné uživatele na několika místech. Využití vlastního *runner* funguje velice dobře.

Celkově bylo toto řešení velmi efektivní a je tedy doporučeno pro tuto demonstrativní aplikaci.

14 Závěr

Cílem práce bylo navrhnout a implementovat automatizovaný proces průběžné integrace a doručení pro webovou aplikaci. Práce byla rozdělena na dvě hlavní části. V první části proběhla rešerše odborné literatury ohledně podstatných témat práce. Byly popsány postupy vývoje software, proces průběžné integrace a doručení a jejich automatizace. Dále testování kódu a nástroje pro průběžnou integraci a doručení.

V druhé části byl navržen a implementován proces průběžné integrace a doručení pro zvolenou aplikaci. Na základě porovnání serveru Jenkins a kombinace GitLab ve verzi SaaS spolu s GitLab Runner v kapitole 13 byly vyhotoveny dvě řešení v podání obou zmíněných služeb. Obě tato řešení splňují stanovené podmínky.

Integrační proces zajišťuje díky virtualizaci správné běhové prostředí s NodeJS. Spustí překlad zdrojového kódu z jazyka TypeScript do jazyka JavaScript, instaluje závislosti a spustí automatické testy pro ověření funkčnosti aplikace.

Pokud je tento řetězec událostí úspěšně absolvován, následuje proces doručení. Ten zajistí zabalení sestavené aplikace do virtuálního kontejneru. Prostředí kontejneru obsahuje všechny nezbytnosti pro spuštění a běh aplikace. Následně je tento kontejner odeslán do repozitáře k verzování. Jako poslední krok v procesu doručení proběhne připojení na cílový server, na který se stáhne kontejner z repozitáře a ten je spuštěn.

Na základě zhodnocení v kapitole 13.5 bylo zvoleno za lepší řešení založené na serveru GitLab. Hlavními přednostmi tohoto řešení jsou vysoká míra centralizace, kdy GitLab poskytuje vzdálený repozitář, registry pro verzování a kompletní CI/CD na jednom místě. Dále potom Verzování samotného konfiguračního souboru definujícího kroky průběžné integrace a doručení. GitLab také umožňuje rozdělení oprávnění jednotlivých uživatelů. Celkově je toto řešení velmi pohodlné a přehledné. Z jednoho rozhraní je tak možné ovládat celý vývojový cyklus aplikace.

Výsledné řešení splňuje všechny stanovené podmínky.

Seznam použitých zdrojů

AGILEALLIANCE, 2020. *Agile 101* [online] [cit. 2020-06-04]. Dostupné z: `https://www.agilealliance.org/agile101/`.

BECK, Kent, 2004. *Extreme Programming Explained: Embrace Change*. John Wait. ISBN 978-0201616415.

DRUMOND, Claire, 2020. *Scrum* [online] [cit. 2020-06-05]. Dostupné z: `https://www.atlassian.com/agile/scrum`.

DUVALL, Paul M.; MATYAS, Steve; GLOVER, Andrew, 2008. *Continuous Integration: Improving Software Quality and Reducing Risk*. Pearson Education, Inc. ISBN 978-0-321-33638-5.

FOWLER, Martin, 2006. *Continuous Integration* [online] [cit. 2020-06-16]. Dostupné z: `https://martinfowler.com/articles/continuousIntegration.html`.

GITLAB, 2020. *GitLab* [online] [cit. 2020-06-06]. Dostupné z: `https://about.gitlab.com/`.

JENKINS, 2020. *Dokumentace Jenkins* [online] [cit. 2020-06-06]. Dostupné z: `https://www.jenkins.io/doc/`.

KITCHENHAM, Barbara; LAWRENCE PFLEEGER, Shari, 1996. Software Quality: The Elusive Target. *IEEE Software*, s. 12–21. Dostupné z DOI: `http://www-public.imtbs-tsp.eu/~gibson/Teaching/Teaching-ReadingMaterial/KitchenhamPfleeger96.pd`.

KOSKELA, Lasse, 2013. *Effective Unit Testing*. Manning Publications Co. ISBN 9781935182573.

KSHIRASAGAR, Naik; PRIYADARSHI, Tripathy, 2008. *Software testing and quality assurance: theory and practice*. John Wiley & Sons, Inc. ISBN 978-0-471-78911-6.

MICROSOFT, 2020. *Linux containers on Windows 10* [online] [cit. 2020-06-16]. Dostupné z: `https://docs.microsoft.com/en-us/virtualization/windowscontainers/deploy-containers/linux-containers`.

PATHANIA, Nikhil, 2017. *Learning Continuous Integration with Jenkins: a Beginners Guide to Implementing Continuous Integration and Continuous Delivery Using Jenkins 2*. PACKT Publishing Limited. ISBN 978-1-78528-483-0.

PITTET, Sten, 2020. *The different types of software testing* [online] [cit. 2020-06-05]. Dostupné z: `https://www.atlassian.com/continuous-delivery/software-testing/types-of-software-testing`.

POULTON, Nigel, 2018. *Docker Deep Dive*. Nigel Poulton. ISBN 978-1-521-82280-7.

RED HAT, 2020. *What is Docker?* [online] [cit. 2020-06-15]. Dostupné z: `https://www.redhat.com/en/topics/containers/what-is-docker`.

REHKOPF, Max, 2020. *Continuous Integration Tools* [online] [cit. 2020-06-03]. Dostupné z: `https://www.atlassian.com/continuous-delivery/continuous-integration/tools`.

ROSSEL, Sander, 2017. *Continuous Integration, Delivery, and Deployment*. Packt Publishing. ISBN 978-1-78728-661-0.

SHARMA, Sanjeev, 2017. *The DevOps Adoption Playbook: a Guide to Adopting DevOps in a Multi-Speed IT Enterprise*. John Wiley & Sons. ISBN 978-1-119-30874-4.

SOFTWARETESTINGFUNDAMENTALS, 2020. *System Testing* [online] [cit. 2020-06-05]. Dostupné z: `http://softwaretestingfundamentals.com/system-testing/`.

SONI, Mitesh, 2017. *DEVOPS BOOTCAMP*. PACKT Publishing Limited. ISBN 978-1-78728-596-5.

STELLMAN, Andrew; GREENE, Jennifer, 2015. *Learning Agile*. O'Reilly Media. ISBN 978-1-449-33192-4.

STOLBERG, S., 2009. Enabling Agile Testing through Continuous Integration. In: *2009 Agile Conference*, s. 369–37.

TRAVIS CI, 2020. *Dokumentace Travis* [online] [cit. 2020-06-06]. Dostupné z: `https://docs.travis-ci.com/`.

TURNBULL, James, 2016. *The Docker Book*. James Turnbull. ISBN 978-0-988-82020-3.

VERONA, Joakim, 2016. *LEARNING DEVOPS: Continuously Deliver Better Software*. PACKT Publishing Limited. ISBN 978-1-78712-661-9.

III. PŘÍLOHY

15 Jenkins *pipeline*

```
pipeline {
  environment {
    registry = "otaklapka/notes-api"
    registryCredential = "dockerhub"
    containerName = "notes-api-jenkins"
    NODE_ENV = "development"
    PORT = "3000"
    CORS_URL = "*"
    DB_NAME = "notes"
    DB_HOST = "otaklapka.cz"
    DB_PORT = "<port>"
    DB_USER = "<db_user>"
    DB_USER_PWD = "<pwd>"
    API_KEY = "<apikey>"
  }

  agent {
    docker { image 'node:13' }
  }

  stages {
    stage('Cloning Git') {
      steps {
        git 'https://github.com/otaklapka/notes-api.git'
      }
    }
    stage('Testing code') {
      steps {
        sh 'npm install'
        sh 'npm run test'
      }
    }
    stage('Building image') {
      steps{
        script {
          dockerImage = docker.build registry + ":$BUILD_NUMBER"
        }
      }
    }
    stage('Deploy Image') {
      steps{
        script {
            docker.withRegistry( '', registryCredential ) {
            dockerImage.push()
```

```
      }
    }
  }
}
stage('Remove Unused docker image') {
  steps{
    sh "docker rmi $registry:$BUILD_NUMBER"
  }
}
stage('Running docker on otaklapka.cz server') {
  steps {
    script {
      withCredentials([
        usernamePassword(
          credentialsId: 'ssh-otaklapka.cz',
          usernameVariable: 'sshUser',
          passwordVariable: 'sshPasswd'
        ),
        usernamePassword(
          credentialsId: 'dockerhub',
          usernameVariable: 'dhUser',
          passwordVariable: 'dhPasswd'
        )
      ]) {
        def remote = [:]
        remote.port = <ssh_port>
        remote.name = 'otaklapka.cz'
        remote.host = 'otaklapka.cz'
        remote.user = sshUser
        remote.password = sshPasswd
        remote.allowAnyHosts = true
        try {
          sshCommand
            remote: remote,
            command: "docker stop ${env.containerName}"
          sshCommand
            remote: remote,
            command: "docker rm ${env.containerName}"
        } catch(err) {
          sh "echo 'No previous container'"
        }
        sshCommand
          remote: remote,
          command:
            "docker login
            --username ${dhUser}
```

```
                    --password ${dhPasswd}"
                sshCommand
                  remote: remote,
                  command: "docker run -d -p 3030:3000
                  -e NODE_ENV='${env.NODE_ENV}'
                  -e PORT=${env.PORT}
                  -e CORS_URL='${env.CORS_URL}'
                  -e DB_NAME='${env.DB_NAME}'
                  -e DB_HOST='${env.DB_HOST}'
                  -e DB_PORT=${env.DB_PORT}
                  -e DB_USER='${env.DB_USER}'
                  -e\ DB_USER_PWD='${env.DB_USER_PWD}'
                  -e API_KEY='${env.API_KEY}'
                  --name ${env.containerName}
                  ${registry}:${env.BUILD_NUMBER}"
              }
            }
          }
        }
      }
    }
```

Kód 30: Kompletní ci/cd *pipeline* Jenkins

16 GitLab *pipeline*

```
stages:
  - build
  - test
  - build_image
  - deploy_prod

image: node:13
default:
  tags:
    - otaklapka.cz

# This folder is cached between builds
# http://docs.gitlab.com/ce/ci/yaml/README.html#cache
cache:
  paths:
    - node_modules/
```

```
build:
  stage: build
  script:
    - npm install

test:
  stage: test
  script:
    - npm run test

build_image:
image: docker:19.03.12
stage: build_image
only:
  - master
script:
  - >
    docker login
    -u $CI_REGISTRY_USER
    -p $CI_REGISTRY_PASSWORD $CI_REGISTRY
  - >
    docker build
    -t $CI_REGISTRY/otaklapka/notes-api:$CI_COMMIT_SHORT_SHA .
  - >
    docker push
    $CI_REGISTRY/otaklapka/notes-api:$CI_COMMIT_SHORT_SHA
  - >
    docker rmi
    $CI_REGISTRY/otaklapka/notes-api:$CI_COMMIT_SHORT_SHA

deploy_prod:
stage: deploy_prod
only:
  - master
when: manual
before_script:
  ##
  ## Install ssh-agent if not already installed
  ## (change apt-get to yum if you use an RPM-based image)
  ##
  - >
    'which ssh-agent
    || ( apt-get update -y && apt-get install openssh-client -y )'

  ##
  ## Run ssh-agent (inside the build environment)
```

```
    ##
    - eval $(ssh-agent -s)

    ##
    ## Add the SSH key stored in SSH_PRIVATE_KEY variable
    ## using tr to fix line endings which makes ed25519 keys work
    ## without extra base64 encoding.
    ##
    - echo "${SSH_PRIVATE_KEY}" | tr -d '\r' | ssh-add - > /dev/null

    ##
    ## Create the SSH directory and give it the right permissions
    ##
    - mkdir -p ~/.ssh
    - chmod 700 ~/.ssh
    - >
      '[[ -f /.dockerenv ]] && echo
      -e "Host *\n\tStrictHostKeyChecking no\n\n" > ~/.ssh/config'
  script:
    - >
      ssh -p 2222 gitlab@otaklapka.cz "docker stop gitlab-notes-api"
      || true
    - >
      ssh -p 2222 gitlab@otaklapka.cz "docker rm gitlab-notes-api"
      || true
    - >
      ssh -p 2222 gitlab@otaklapka.cz "docker run -d -p 3030:3000
      -e NODE_ENV='${NODE_ENV}'
      -e PORT=${PORT}
      -e CORS_URL='${CORS_URL}'
      -e DB_NAME='${DB_NAME}'
      -e DB_HOST='${DB_HOST}'
      -e DB_PORT=${DB_PORT}
      -e DB_USER='${DB_USER}'
      -e DB_USER_PWD='${DB_USER_PWD}'
      -e API_KEY='${API_KEY}'
      --name gitlab-notes-api
      $CI_REGISTRY/otaklapka/notes-api:${CI_COMMIT_SHORT_SHA}"
```

Kód 31: Kompletní ci/cd *pipeline* GitLab