Vývojářské metody – kontinuální integrace a kontinuální doručování

Bc. Tomáš Lecián

Diplomová práce
2019

Univerzita Tomáše Bati ve Zlíně
Fakulta aplikované informatiky

Univerzita Tomáše Bati ve Zlíně
Fakulta aplikované informatiky
akademický rok: 2018/2019

ZADÁNÍ DIPLOMOVÉ PRÁCE
(PROJEKTU, UMĚLECKÉHO DÍLA, UMĚLECKÉHO VÝKONU)

Jméno a příjmení:	**Bc. Tomáš Lecián**
Osobní číslo:	**A17810**
Studijní program:	**N3902 Inženýrská informatika**
Studijní obor:	**Informační technologie**
Forma studia:	**kombinovaná**
Téma práce:	**Vývojářské metody – kontinuální integrace a kontinuální doručování**
Téma anglicky:	**Development Methods – Continuous Integration and Continuous Delivery**

Zásady pro vypracování:

1. **Zpracujte literární rešerši na téma vývojářské metody kontinuální integrace (CI), kontinuálního doručování (CD) a principy verzovacích systémů.**
2. **Proveďte rozbor současných nástrojů a metod na CI/CD včetně jejich praktického nasazení.**
3. **Vytvořte vhodnou vzorovou aplikaci pro demonstraci CI/CD.**
4. **Navrhněte a zaveďte v projektu soubor vhodných testů.**
5. **Navrhněte integrační prostředí včetně jeho nastavení pro potřeby CI/CD.**
6. **Zhodnoťte nasazení CI/CD.**

Rozsah diplomové práce:

Rozsah příloh:

Forma zpracování diplomové práce: **tištěná/elektronická**

Seznam odborné literatury:

1. **HUMBLE, Jez a David FARLEY. Continuous delivery: reliable software releases through build, test, and deployment automation. Upper Saddle River, NJ: Addison-Wesley, 2010. ISBN 978-032-1601-919.**
2. **ROSSEL, Sander. Continuous Integration, Delivery, and Deployment. Packt Publishing, 2017. ISBN 9781787284180.**
3. **DUVALL, Paul M, Steve MATYAS a Andrew GLOVER. Continuous integration: improving software quality and reducing risk. Upper Saddle River, NJ: Addison-Wesley, c2007. ISBN 978-0-321-33638-5.**
4. **KAWALEROWICZ, Marcin a Craig BERNTSON. Continuous integration in .NET. Greenwich, Conn.: Manning, c2011. ISBN 9781935182559.**
5. **ROSSBERG, Joachim. Agile project management using team foundation server 2015. New York: Apress, 2016. Expert's voice in .NET. ISBN 978-1-4842-1869-3.**

Vedoucí diplomové práce: **Ing. Milan Navrátil, Ph.D.**
Ústav elektroniky a měření

Konzultant: **Ing. Filip Kovář**
Business Logic s.r.o.

Datum zadání diplomové práce: **3. prosince 2018**

Termín odevzdání diplomové práce: **15. května 2019**

Ve Zlíně dne 7. prosince 2018

doc. Mgr. Milan Adámek, Ph.D.
děkan

prof. Mgr. Roman Jašek, Ph.D.
garant oboru

Prohlašuji, že

- beru na vědomí, že odevzdáním diplomové práce souhlasím se zveřejněním své práce podle zákona č. 111/1998 Sb. o vysokých školách a o změně a doplnění dalších zákonů (zákon o vysokých školách), ve znění pozdějších právních předpisů, bez ohledu na výsledek obhajoby;
- beru na vědomí, že diplomová práce bude uložena v elektronické podobě v univerzitním informačním systému dostupná k prezenčnímu nahlédnutí, že jeden výtisk diplomové/bakalářské práce bude uložen v příruční knihovně Fakulty aplikované informatiky Univerzity Tomáše Bati ve Zlíně a jeden výtisk bude uložen u vedoucího práce;
- byl/a jsem seznámen/a s tím, že na moji diplomovou práci se plně vztahuje zákon č. 121/2000 Sb. o právu autorském, o právech souvisejících s právem autorským a o změně některých zákonů (autorský zákon) ve znění pozdějších právních předpisů, zejm. § 35 odst. 3;
- beru na vědomí, že podle § 60 odst. 1 autorského zákona má UTB ve Zlíně právo na uzavření licenční smlouvy o užití školního díla v rozsahu § 12 odst. 4 autorského zákona;
- beru na vědomí, že podle § 60 odst. 2 a 3 autorského zákona mohu užít své dílo – diplomovou práci nebo poskytnout licenci k jejímu využití jen připouští-li tak licenční smlouva uzavřená mezi mnou a Univerzitou Tomáše Bati ve Zlíně s tím, že vyrovnání případného přiměřeného příspěvku na úhradu nákladů, které byly Univerzitou Tomáše Bati ve Zlíně na vytvoření díla vynaloženy (až do jejich skutečné výše) bude rovněž předmětem této licenční smlouvy;
- beru na vědomí, že pokud bylo k vypracování diplomové práce využito softwaru poskytnutého Univerzitou Tomáše Bati ve Zlíně nebo jinými subjekty pouze ke studijním a výzkumným účelům (tedy pouze k nekomerčnímu využití), nelze výsledky diplomové/bakalářské práce využít ke komerčním účelům;
- beru na vědomí, že pokud je výstupem diplomové práce jakýkoliv softwarový produkt, považují se za součást práce rovněž i zdrojové kódy, popř. soubory, ze kterých se projekt skládá. Neodevzdání této součásti může být důvodem k neobhájení práce.

Prohlašuji,

- že jsem na diplomové práci pracoval samostatně a použitou literaturu jsem citoval. V případě publikace výsledků budu uveden jako spoluautor.
- že odevzdaná verze diplomové práce a verze elektronická nahraná do IS/STAG jsou totožné.

Ve Zlíně, dne 17. 5. 2019

Tomáš Lecián, v. r.
……………………..
podpis diplomanta

ABSTRAKT

Diplomová práce se zabývá vývojářskými metodami kontinuální integrace a kontinuálního doručování. Teoretická část popisuje dílčí kroky procesu kontinuální integrace a kontinuálního doručování s principy verzovacích systémů. Dále jsou zde popsány možnosti automatizovaného testování a rozbor současných nástrojů kontinuální integrace a kontinuálního doručování. Praktická část obsahuje vzorovou aplikaci se sadou automatizovaných testů a implementaci procesů kontinuální integrace a kontinuálního doručování vzorové aplikace pomocí platformy Azure DevOps.

Klíčová slova: Kontinuální integrace, kontinuálního doručování, automatizované testy, pipeline, Azure

ABSTRACT

This Master's thesis deals with development methods of continuous integration and continuous delivery. The theoretical part describes partial steps of the process of continuous integration and continuous delivery with principles of versioning systems. It also describes the possibilities of automated testing and analysis of current tools of continuous integration and continuous delivery. Practical part contains the sample application with a set of automated tests and implementation of continuous integration and continuous delivery the sample application with using Azure DevOps.

Keywords: Continuous integration, continuous delivery, automated testing, pipeline, Azure

Děkuji vedoucímu práce Ing. Milanu Navrátilovi, Ph.D. a konzultantovi Ing. Filipu Kovářovi za věnovaný čas a věcné připomínky. Děkuji své nejbližší rodině za podporu po celou dobu studia.

Prohlašuji, že odevzdaná verze diplomové práce a verze elektronická nahraná do IS/STAG jsou totožné.

OBSAH

ÚVOD **8**
I TEORETICKÁ ČÁST **9**
1 METODY KONTINUÁLNÍ INTEGRACE A DORUČOVÁNÍ **10**
1.1 HISTORICKÝ VÝVOJ 10
1.2 KONTINUÁLNÍ INTEGRACE 11
1.2.1 CI a vývojový proces 12
1.2.2 Verzování kódu do centrálního úložiště 13
1.2.3 Automatizované sestavení 14
1.2.3.1 Build script 14
1.2.3.2 CI server 15
1.2.4 Automatizované testy 16
1.2.4.1 Testovací prostředí 16
1.2.5 Sestavení musí být rychlé 17
1.2.6 Kontinuální zpětná vazba 18
1.3 KONTINUÁLNÍ DORUČOVÁNÍ A KONTINUÁLNÍ NASAZENÍ 19
1.3.1 Kontinuální doručování 19
1.3.1.1 Základní Deployment pipeline 21
1.3.2 Kontinuální nasazení 23
1.4 ZÁKLADNÍ ZÁSADY VÝVOJÁŘE 24
1.4.1 Sdílet (commitovat) často svůj kód 24
1.4.2 Nesdílet nefunkční kód a spouštět lokální build 24
1.4.3 Vytvářet testy pro vyvíjené části 24
1.4.4 Udržovat úspěšnosti automatizovaných testů 25
1.4.5 Okamžitá oprava neúspěšného buildu 25
2 VERZOVACÍ SYSTÉMY **26**
2.1 GIT 28
2.1.1 Pravidla verzování 29
3 AUTOMATIZOVANÉ TESTOVÁNÍ SOFTWARU **31**
3.1 PYRAMIDA AUTOMATIZOVANÝCH TESTŮ 31
3.2 FUNKČNÍ VS. NEFUNKČNÍ TESTOVÁNÍ 32
3.3 JEDNOTKOVÉ TESTY 34
3.4 INTEGRAČNÍ TESTY 34
3.5 SMOKE TESTY 36
3.6 UI TESTY 36
4 NÁSTROJE CI/CD **38**
4.1 AZURE DEVOPS SERVER A AZURE DEVOPS SERVICES 38
4.2 JENKINS 41
4.3 BITBUCKET PIPELINES 43
4.4 BAMBOO 45
4.5 TEAMCITY 47
4.6 SOUHRN 49
II PRAKTICKÁ ČÁST **52**

5 VZOROVÁ APLIKACE MINISHOP 53
5.1 POŽADAVKY NA APLIKACI 53
5.1.1 Funkční požadavky 53
5.1.2 Nefunkční požadavky 54
5.1.3 Datová specifika 54
5.2 SYSTÉMOVÝ NÁVRH 54
5.3 IMPLEMENTACE 55
5.4 AUTOMATIZOVANÉ TESTY 57
5.4.1 Jednotkové testy 59
5.4.2 Integrační testy 60
6 IMPLEMENTACE CI/CD 63
6.1 DEVOPS PROJEKT 63
6.2 NASTAVENÍ KONTINUÁLNÍ INTEGRACE 63
6.2.1 Agent 64
6.2.2 Repozitář kódu aplikace 65
6.2.3 Pipeline job 66
6.2.3.1 Npm balíčky 66
6.2.3.2 NuGet balíčky 67
6.2.3.3 Obnovení NuGet balíčků 67
6.2.3.4 Build API aplikace 68
6.2.3.5 Build Webové aplikace 69
6.2.3.6 Automatizované testy 70
6.2.3.7 Uložení artefaktů 71
6.2.3.8 Publikování artefaktů 72
6.2.4 Pipeline proměnné 73
6.3 NASTAVENÍ KONTINUÁLNÍHO DORUČENÍ 73
6.3.1 Pipeline job 74
6.3.1.1 Nasazení databáze 75
6.3.1.2 Nasazení API aplikace 75
6.3.1.3 Nasazení Webové aplikace 76
7 SHRNUTÍ NASAZENÍ CI/CD 77
ZÁVĚR 79
SEZNAM POUŽITÉ LITERATURY 80
SEZNAM POUŽITÝCH SYMBOLŮ A ZKRATEK 84
SEZNAM OBRÁZKŮ 86
SEZNAM TABULEK 88
SEZNAM PŘÍLOH 89

ÚVOD

Se zvyšujícími se nároky na softwarové systémy vznikají čím dál více rozsáhlejší a sofistikovanější softwarová řešení. To s sebou přináší inovace v oblasti vývoje softwarových projektů a přístupů k jejich procesům. Vývojové týmy musí rychle a flexibilně reagovat na měnící se požadavky zadavatele softwaru. Tyto principy reflektují agilní metodiky a techniky vývoje softwaru. Při utváření agilního přístupu k vývoji softwaru vznikly metody kontinuální integrace a doručování, kterým se věnuje tato práce.

V teoretické části je představen princip kontinuálních metod a jejich začlenění do životního cyklu aplikace. Je zde podrobně rozebrána podstata, požadavky a přínosy jednotlivých kroků, které jsou nezbytné pro vytvoření procesu kontinuální integrace a doručování. Samostatná kapitola je věnována automatizovanému testování, které tvoří významnou část procesu kontinuálních metod. Dále je součástí teoretické části rozbor nejvýznamnějších moderních nástrojů v oblasti kontinuálních metod s ukázkami praktického nasazení a jejich srovnání s ostatními nástroji.

Praktická část je zaměřena na reálné nastavení integračního prostředí pro potřeby kontinuální integrace a doručování na základě poznatků z teoretické části práce. Nasazení procesů je prováděno pomocí cloudových nástrojů Azure DevOps Services od společnosti Microsoft. Potřeby kontinuální integrace a doručení jsou prováděny pro vzorovou webovou aplikaci, která je součástí diplomové práce. Implementace webové aplikace je vytvořena pomocí technologií .NET Core a Angular.

I. TEORETICKÁ ČÁST

1 METODY KONTINUÁLNÍ INTEGRACE A DORUČOVÁNÍ

Kontinuální integrace a dodávka je spojení nástrojů a postupů informačních technologií, které umožňují vývojovým týmům často a spolehlivě uveřejňovat nové verze vyvíjeného softwaru. Automatizace úkonů, namísto neustálého manuálního opakování stejných postupů, přináší usnadnění a zefektivnění vývoje. Členům vývojových týmů to umožňuje lépe integrovat svoji práci a mít přehled o výsledku zapracovaných změnách. To se odráží do rychlosti a kvality celého vývoje softwaru.

1.1 Historický vývoj

Vše na světě prochází vývojem, nic se nezrodí přes noc. Není tomu jinak ani při samotném vývoji počítačového softwaru. Myšlenka o pravidelném vytváření buildu při vývoji softwaru se začala rodit počátkem 90. let minulého století.

Grady Booch ve své knize *Object-Oriented Analysis and Design* popisuje mikro proces jako strategii pro odhalení potenciálního problému již v rané fázi vývoje a tím snížení rizika při vývoji a jeho urychlení. Zmíněný mikro proces označil výrazem „Continuous Integration“ (CI). Jedná se o interní proces v organizaci, v rámci kterého by mělo docházet k pravidelnému vytváření mnoha dílčích verzí systému, které jsou inkrementálního rázu. Určité verze by se měly poté následně stabilizovat a brát jako produkční verze. [1]

Tyto myšlenky se dále rozvíjely v rámci extrémního programování, které je jednou z metodik agilního vývoje softwaru. To, že má CI své místo v softwarovém inženýrství, popisuje zakladatel extrémního programování Kent Beck ve své knize *Fundamental approaches to software engineering v roce 1998*. [2]

O významnou popularizaci CI se v roce 2000 zasloužil Martin Fowler. Ve svém článku popsal základní praktiky kontinuální integrace a vyzdvihuje zde benefity, které s sebou kontinuální integrace přináší. [3]

Kontinuální integrace se tak počátkem 21. století dostává do podvědomí širší společnosti a proces se pomalu stává nedílnou součástí vývoje softwaru u velkých projektů. V roce 2001 zásluhou Matthewa Foemmela vzniká první specializovaný server a nástroj pro kontinuální integraci, který je pojmenován *Cruise Control.* [4] [5]

1.2 Kontinuální integrace

Vývoj softwaru se skládá z několika fází – od návrhu přes vývoj, testování, až po uveřejnění softwaru a předání zákazníkovi. Na vývoji se může podílet jednotlivec či rozsáhlý vývojový tým, který může pracovat paralelně na různých částech a modulech softwaru. Tyto části je nutné spojovat do jednoho funkčního celku. Pro integraci kódu, předcházení chybám a plynulejšímu vývoji vedoucímu ke zdárnému dokončení se nabízí uplatnění principů kontinuální integrace. [6]

Kontinuální integrace, v češtině taktéž označována jako postupná či průběžná integrace, vychází z anglického spojení „*Continuous Integration*“. Ve spojení s moderními technologiemi je proces často označován pouze počátečními písmeny jako CI. Jednu z nejvýstižnějších definic, co je kontinuální integrace, přinesl Martin Fowler:

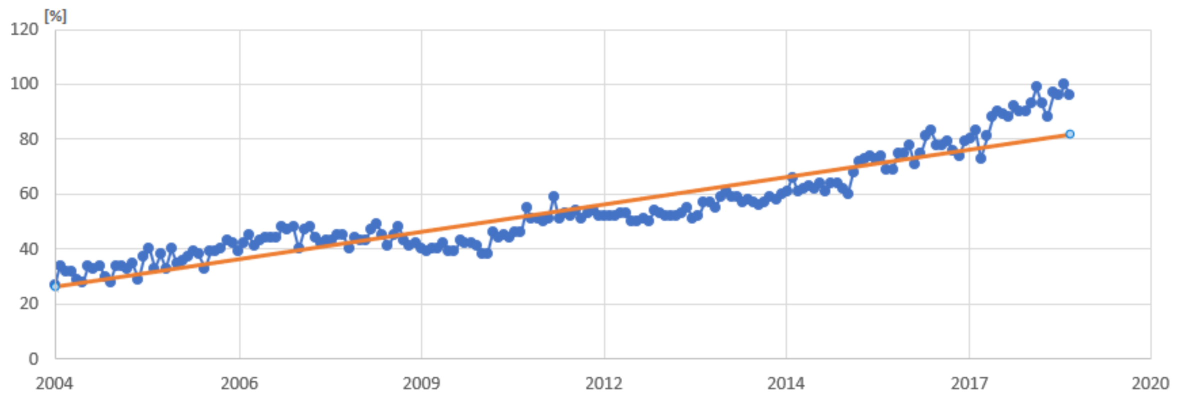
„Kontinuální integrace je technika vývoje software, při které členové vývojového týmu pravidelně integrují svou práci, obvykle každý člen týmu alespoň jednou denně – to vede k mnoha integracím denně. Každá integrace je ověřena automatickým sestavením a provedením automatizovaných testů, aby bylo možné co nejrychleji zjistit případné chyby integrace. Mnoho týmů shledalo, že tento přístup vede k výraznému snížení problémů plynoucích z integrace a umožňuje týmům tento přístup vyvinout soudržný software rychleji.“ [7]

Paul M. Duvall ve své knize *Continuous Integration Improving Software Quality and Reducing Risk*, kde se podrobně zabývá dílčími kroky CI, popisuje proces následovně:

„Kontinuální integrace je ztělesněním taktiky, která nám jako vývojářům softwaru umožňuje provádět změny v našem kódu s vědomím, že pokud rozbijeme software, budeme okamžitě informováni. Tato okamžitá zpětná vazba nám dává čas na to, abychom se správně zorientovali a rychleji přizpůsobili.“ [8]

Vývoj softwaru vyžaduje plánování, průběžné sledování a postupné usměrňování průběhu vývoje. CI je ztvárnění taktiky, která vývojářům softwaru dává schopnost provádět změny v kódu s vědomím, že pokud se z důvodu zapracovaní změn do kódu podaří rozbít software, jsou okamžitě na základě zpětné vazby o tom informováni. Tato okamžitá zpětná vazba umožňuje členům týmu mít možnost rychleji se přizpůsobit změnám. [8]

Popularita kontinuální integrace při vývoji softwaru je čím dál vyšší. Dokazují to nejen přibývající konkurenční nástroje pro snazší implementaci CI, ale i stále rostoucí trend ve vyhledávání společnosti Google (Obr. 1).



Obr. 1. Popularita vyhledávání pojmu Continuous Integration na celém světě. [9]

1.2.1 CI a vývojový proces

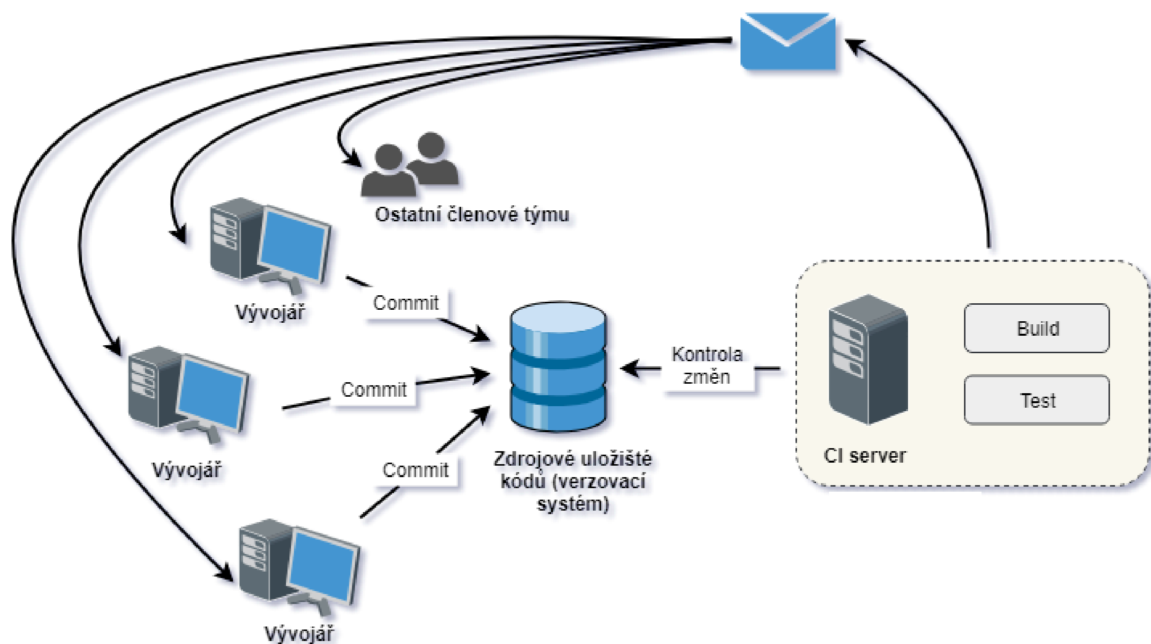
Pro implementaci kontinuální integrace nezáleží na tom, zda se vývojový tým striktně drží metodiky agilního vývoje. Naopak se preferuje spíše cesta vývoje vycházející z vodopádového modelu či kombinace obou zmíněných modelů vývoje softwaru. Pokud jde o psaní kódů, proces je obdobný a na zvolené metodice nezáleží i přes to, že kontinuální integrace původně vznikla jako součást extrémního programování. Činnost vývojáře při psaní kódu lze popsat následujícími základními kroky [10]:

1. Vývojář si stáhne potřebné zdrojové kódy z úložiště zdrojových kódů.
2. Provede změnu kódu.
3. Spustí kompilaci upraveného kódu ve svém IDE pro vytvoření buildu softwaru.
4. Je-li kompilace kódu neúspěšná, vrací se k bodu 2.
5. Spustí si jednotkové testy pro ověření zapracovaných vlastních změn i pro ověření, zda nezpůsobil svými změnami chybu v jiné části softwaru.
6. Nejsou-li všechny jednotkové testy úspěšné, vrací se k bodu 2.
7. Zkontroluje nový kód a provede případně změny vedoucí ke zvýšení čitelnosti a čistoty kódu, tzv. refactor. Tento proces vede ke zvýšení kvality celého kódu software, snižuje počet budoucích chyb a z toho plynoucí rychlejší vývoj. Pro ověření, zda při refactoru nebyly do kódu vneseny další chyby, přejde vývojář zpět k 5. kroku.
8. Následně uloží kód do centrálního úložiště kódu.

S využitím kontinuální integrace, vývojář provádí při vývoji obdobné kroky. Navíc však po uložení nového či změněného kódu do centrálního úložiště kódu jsou podniknuty následující kroky, které se neustále periodicky opakují:

9. Automatizovaný systém provádí kontrolu centrálního úložiště kódu. Detekuje-li změnu, stáhne se poslední aktuální verze kódu a provádí se následující krok.
10. Automatizovaný systém provede kompilaci a vytvoření buildu aplikace.
11. Automatizovaný systém spustí jednotkové testy.
12. Automatizovaný systém rozešle jednotlivým členům týmu zpětnou vazbu o výsledku kompilace aplikace a úspěšnosti dokončení provedených testů.

Pro názornost procesu vývojového týmu a automatizovaného systému je na obr. 2 naznačeno blokové schéma.



Obr. 2. Blokové schéma procesu kontinuální integrace. [8] [10] [20]

Z výše popsaného procesu kontinuální integrace vyplývají určité principy a zásady, které tvoří samotný proces. Konkrétní kroky kontinuální integrace a z nich plynoucí zásady jsou popsány dále. Myšlenky jsou čerpány především z Fowlerova článku [3], knihy *Continuous integration: improving software quality and reducing risk* [8] od Duvala či autorů knihy *Continuous delivery: reliable software releases through build, test, and deployment automation* [12].

1.2.2 Verzování kódu do centrálního úložiště

Softwarové projekty se skládají z velkého množství souborů, které jako celek tvoří výsledný produkt. Tyto soubory je zapotřebí spravovat. Nástroje pro verzování kódu, tj. uchovávání historie provedených změn kódů, jsou již neodmyslitelnou součástí vývoje.

Při práci s verzovacím systémem je důležité verzovat vše potřebné a nezbytné pro správné vytvoření buildu softwaru – zdrojové kódy, konfigurační soubory, testovací skripty, databázová schémata i knihovny třetích stran. Díky tomu je možné po stažení aktuálních zdrojových souborů z jednoho místa provádět kompilaci softwaru. [3] [8]

Při používání verzovacího nástroje by se nemělo zapomínat na obecné zásady správného používání verzovacího systému, viz kapitola 2.1.1. Přínosy kontinuální integrace nebudou využity, pokud například nebude možné neustále zkompilovat aplikaci z důvodů commitnutí nefunkčního kódu vývojářem. Commitnutím se rozumí potvrzení provedených změn v kódu a jejich uložení do centrálního úložiště kódu.

1.2.3 Automatizované sestavení

Nepřetržité sestavování zdrojového kódu je jednou z nejzákladnějších a nejběžnějších vlastností CI systémů. Ve skutečnosti je to téměř synonymem CI. Sestavování znamená podle předem nastavených pravidel kompilovat zdrojové kódy a vytvářet (sestavovat) build aplikace. Jednoduše řečeno: build je proces, který převádí kód čitelný člověkem, do konečné podoby, která je následně spustitelná na dané počítačové platformě. [8]

1.2.3.1 Build script

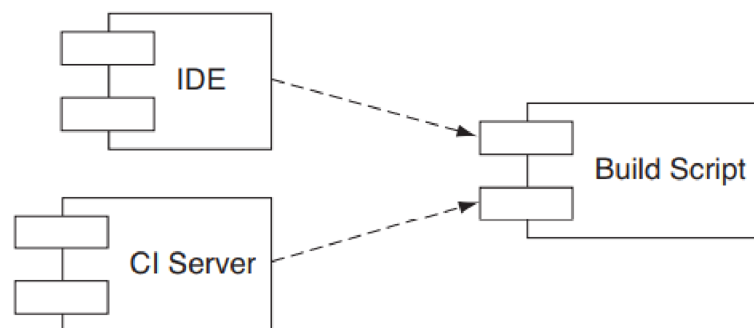
Build softwaru je prováděn pomocí jednoho buildu či pomocí celé sady build scriptů. Je možné použít skripty, které lze získat pomocí sestavovacích nástrojů, pro automatickou tvorbu build scriptů, které neposkytují CI. Mezi takovéto nástroje se řadí např. MSBuild (.NET), Ant (Java), Rake (Ruby) a další. Mnohé z nich používají IDE k samotnému buildu softwaru. Např. IDE Microsoft Visual Studio u technologie .NET využívá pro sestavování projektů MSBuild. Build script u této technologie je označen příponou .csproj. [8] [12] [13]

Duvall ve své knize [8] poznamenal, že build skripty vytvářené a používáné v rámci IDE, mají pro kontinuální integraci smysl pouze tehdy, pokud je možné je stejně dobře použít i bez IDE. Obdobný názor na téma build skripty sdílejí i autoři knihy *Continuous Delivery* [12]. Všichni autoři jsou zastánci oddělených build scriptů z následujících důvodů:

- V rámci vývojového týmu mohou vývojáři ke své práci využívat různé IDE. V takovém případě může být obtížné rozeznávat odlišnosti v konfiguraci každého IDE.

- CI Server musí provádět integrační build automaticky bez zásahu někoho z týmu. Proto by měl být používaný stejný build script pro vývojáře i pro CI server.
- Build skripty by měly být verzovány, testovány a refaktorovány stejně jako softwarový kód. S automaticky generovanými skripty pomocí IDE to však není vždy proveditelné.

Build skripty u kontinuální integrace je možné použít nejen na samotnou kompilaci softwaru, ale také pro integraci databáze, testování či závěrečné vystavení. Tyto podprocesy jsou popsány dále.



Obr. 3. Blokové schéma oddělování build scriptu od IDE. [8]

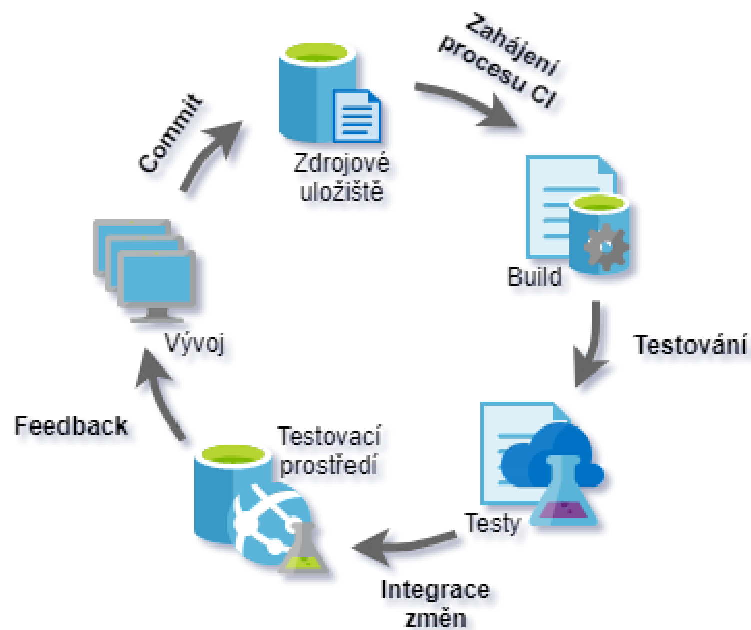
1.2.3.2 CI server

Procesy v rámci kontinuální integrace jsou prováděny pomocí kontinuálně integračního serveru, tzv. CI server. Tento server periodicky kontroluje hlavní vývojovou větev repozitáře verzovacího systému, zda někdo z vývojového týmu nenahrál do repozitáře nové změny kódu. Jsou-li detekovány změny, je automaticky spuštěn integrační build.

Integrační build je prováděn pomocí integračního sestavovacího stroje („integration build machine"), který je hostem CI serveru.

Základními úkoly CI serveru jsou:

- připojení k úložišti verzovacího systému,
- spuštění build skriptů,
- zasílání informací se zpětnou vazbou, např. e-mail, Slack notifikace, SM apod.,
- integrace změn zdrojového kódu (manuální nebo kontinuální).



Obr. 4. Ukázka základního procesu CI a funkce CI serveru. [20]

1.2.4 Automatizované testy

Nespuštěním automatizovaných testů by kontinuální integrace nebyla kontinuální integrací. Pokud dojde k úspěšné kompilaci kódu, program může běžet. Neznamená to však, že pracuje správně. Pro dosažení určité kvality vývoje je nezbytné mít vytvořenou komplexní sadu automatizovaných testů. Pomocí této sady testů jsme schopni mít pod kontrolou podstatnou část kódu a mít tak přehled o jeho stavu, což je zvláště důležité při řízení samotného projektu. [3]

Podrobnější popis a použití testů je popsáno v kapitole 3 Automatizované testování softwaru.

1.2.4.1 Testovací prostředí

Smyslem testování je nejen prověření, zda systém funguje dle zadaných požadavků, ale také snaha odhalit potenciální problémy, které by systém mohl mít v produkčním prostředí. Významným prvkem je i prostředí, ve kterém bude konečný systém pracovat. Pokud bude testování probíhat v testovacím prostředí, které se významně liší od produkčního prostředí, dopouštíme se rizika, že chování v systému v prostředí produkčním bude odlišné vzhledem k chování v prostředí testovacím.

Testovací prostředí by mělo být co nejvíce shodné s produkčním prostředím. To znamená: použít stejný operační systém i databázový server se shodnými verzemi, mít dostupné

stejné služby, nainstalované stejné knihovny atp. Řešením může být například virtuální stroj, který bude kopií produkčního serveru.

1.2.5 Sestavení musí být rychlé

Kontinuální integrace klade důraz na rychlost a plynulost procesu vývoje. Pokud samotná kompilace softwaru a spouštění testů trvá příliš dlouho, nastávají následující rizika, která uvádí autoři knihy *Continues Delivery* [12]:

- Vývojáři nebudou motivování spouštět lokální build a automatizované testy před commitnutím a uložením změn do centrálního úložiště kódu. To způsobí větší chybovost v kódu a počet neúspěšných verzí integračního buildu.
- Proces kontinuální integrace, tj. kompilace a spuštění všech testů, bude trvat tak dlouho, že před spuštěním dalšího procesu kontinuální integrace nastane ve VCS několik dalších změn od různých vývojářů. Dopadne-li integrační build nezdarem, nebude tak patrné, které změny od kterého vývojáře způsobily rozbití.
- Vývojáři budou méně často ověřovat, zda jejich sdílené změny byly správně integrovány v rámci integračního buildu a nevyžadují nějaký další zásah – opravu.

V ideálním případě by kompilace a spuštění testů nemělo trvat déle než několik minut. A to na lokálním počítači, na kterém vyvíjí člen týmu i na CI serveru, kde je prováděn integrační build. Autoři Fowler, Humble a Farley se shodují, že ideální doba trvání buildu je 10 minut a méně [3] [12].

Doba potřebná pro sestavení je závislá na velikosti projektu, který je integrován či na technické vybavenosti a vytíženosti CI serveru. Co nejkratší doba integračního buildu si taktéž rozporuje s udržováním komplexní sady automatizovaných testů nad projektem. Pro rychlejší zpracovávání testů existuje několik technik.

Na začátku je třeba zvážit, zda neexistuje nějaká možnost zrychlení samotných testů. U nejdéle trvajících testů je třeba prověřit, zda neexistuje nějaký způsob optimalizace kódu či způsob, jak funkcionalitu, kterou test kontroluje pokrýt jinak a s menší náročností. Nástroje pro automatizované testování, které se využívají pro psaní testů, např. xUnit, JUnit, JSUnit, CppUnit, Selenium a další, běžně poskytují informaci o tom, jak dlouho se jednotlivé testy zpracovávají. Tento refactor testů by měl být běžnou praxí při vývoji. [12]

Ne všechny testy je možné optimalizovat a některé testy zkrátka potřebují svůj čas. Například jednotkové testy bývají rychlé, jelikož prověřují jednoduchou základní funkčnost.

Ale integrační testy, v rámci kterých je vyžadována komunikace s databází, jsou v porovnání s jednotkovými testy pomalejší. Začne-li projekt obsahovat velké množství testů, které jsou vytvořeny za různým účelem testování, je to pravděpodobně moment, kdy bude nutné rozdělit spouštění testů do více fází za účelem plynulejšího procesu kontinuální integrace.

V první fázi by mělo docházet ke kompilaci kódu a spouštění pouze jednotkových testů, které jsou rychlé a prověřují základní jednotky. Následně by mělo dojít k vystavení vytvořeného buildu do testovacího prostředí, tzv. deploy. Kroky první fáze se často označují výrazem „Commit Build“ či „Commit Deploy“ [3] [12]. Tyto kroky by měly být rychlé a neměly by trvat déle něž několik minut.

V rámci druhé etapy by mělo dojít ke spuštění dalších testů, např. integračních, akceptačních, výkonnostních, nad vystavenou verzí z první fáze. Charakter, význam a potřeba spouštění testů je rozličná, proto je možné uvažovat rozdělení testů do více než dvou fází.

Díky rozdělení spouštění testů do fází můžeme dosáhnout rovnováhy potřeb odhalování chyb a rychlosti tak, aby byl Commit Build stabilní a vývojový tým s ním mohl pracovat. Pokud druhá fáze trvá déle než půl hodiny, je na zvážení, zda jednotlivé testovací balíky nespouštět paralelně. [3] [12]

1.2.6 Kontinuální zpětná vazba

Informace o softwarovém projektu se neustále mění, a tak je nutné sdílet informace s klientem, manažery, vývojáři, testery a dalšími členy týmu o stavu projektu. Zatímco komunikace tváří v tvář může být nejefektivnější, není příliš škálovatelná. Nabízí se v rámci kontinuální integrace zautomatizování sdílení cenných informací, kterými CI server disponuje.

Mezi běžné informace, které CI server může distribuovat patří např. výsledek aktuálně dokončeného buildu, čas dokončení buildu, výsledky testů, aktuální vystavená verze v prostředí a další. Rozesílání zpětné vazby z CI serveru všem členům projektu po každé změně, která nastane, není zcela šťastné řešení. Rozesílání množství zpráv, např. s informací, že je vše v pořádku, může u členů týmu vyvolat ignorování zpráv a zpětná vazba z CI serveru začne postrádat smysl. Je důležité klást si otázku, komu z týmu budou zaslány jaké informace, proč je dostane, kdy a jakou cestou mu budou doručeny.

Projektoví manažeři, kteří se zaměřují převážně na přidělování zdrojů (členům týmu či hardware), správě dílčích částí, stráveného času a nákladů projektu, potřebují mít v reálném čase přehled o stavu softwaru z pohledu času, nákladů, kvality či rozsahu.

Techničtí vedoucí naopak ocení informace o stavech proběhlých dílčích buildů, jelikož nahlíží na software jako na celek.

Vývojářům budou nejvíce užitečné informace o výsledku integračního buildu (Commit buildu), v rámci kterého byly zpracovávány změny kódu, které sami provedli.

Testery naopak budou nejvíce zajímat výsledky jednotlivých automatizovaných testů a kontrol.

Staré správy nejsou novinkami, a tak čím více času uplyne, např. od nějakého problému, tím si zodpovědná osoba bude méně pamatovat, co se stalo a proč. To má poté neblahý efekt na rychlost a kvalitu řešení. Zpětná vazba z CI serveru, která může napomoci k efektivnějšímu řešení situace, by měla být doručována ihned.

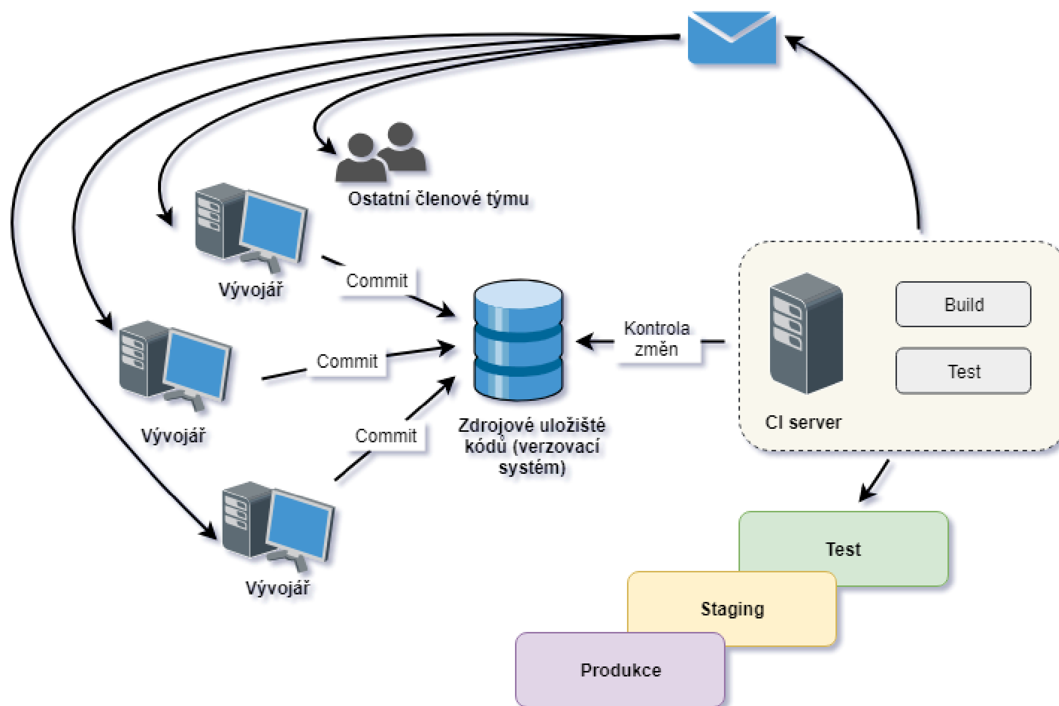
Nástroje pro kontinuální integraci mnohdy nabízejí široké možnosti způsobu zpětné vazby z CI serveru. Mezi nejběžnější se řadí zasílání emailů, SMS, vytváření příspěvků u kolaborativních nástrojů (např. Slack), dashboard nástroje pro CI a další.

1.3 Kontinuální doručování a kontinuální nasazení

Zatímco kontinuální integrace je přímočarý proces a znamená nepřetržitou integraci, která se zaměřuje na usnadnění vydání software, kontinuální doručování je proces nepřetržitého poskytování nebo nasazení softwaru do produkčního prostředí.

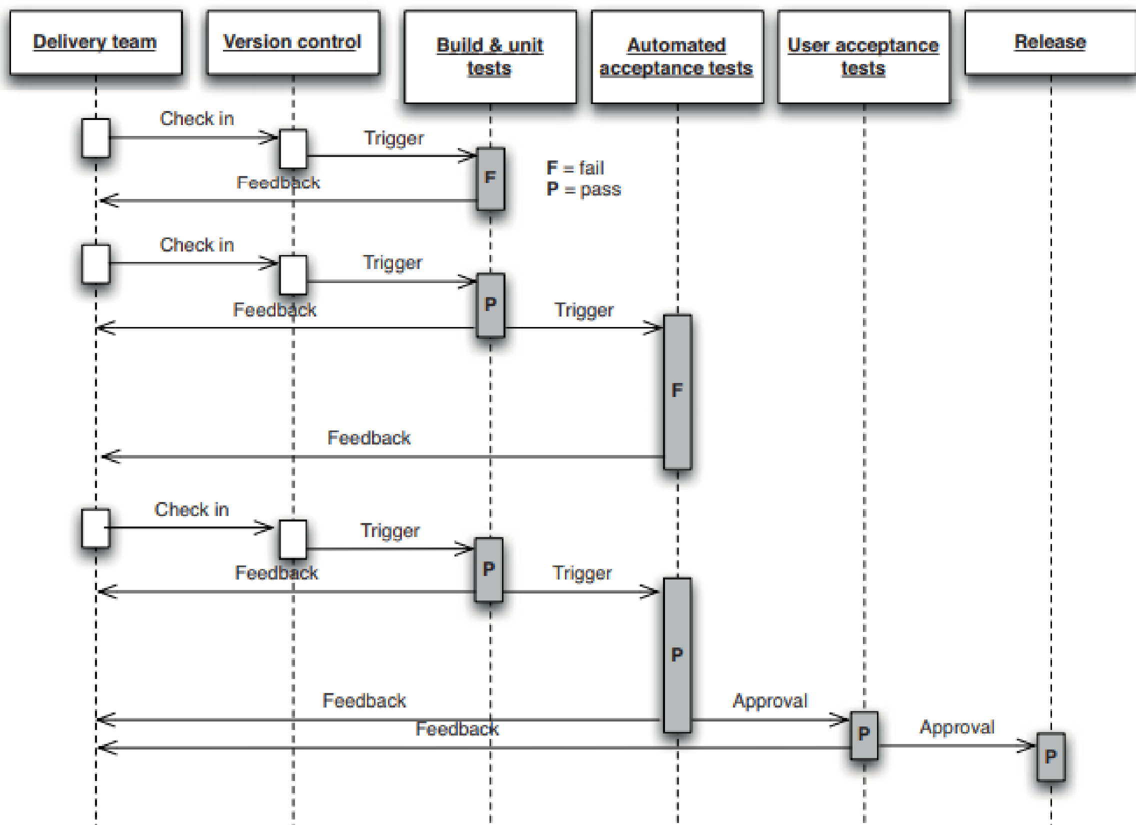
1.3.1 Kontinuální doručování

Výraz kontinuální doručování vychází z anglického *Continuous Delivery* (CD). Kontinuální doručování je rozšířením kontinuální integrace pro zajištění rychlého uvolňování nových změn softwaru udržitelným způsobem. To znamená, že proces CI/CD nejen umožňuje automatickou kompilaci a otestování nových změn v kódu po uložení, ale je také zautomatizovaný proces nasazení software, kdy je možné novou verzi uvolnit kdykoli kliknutím na tlačítko. [25]



Obr. 5. Základní blokové schéma procesu CI/CD. [8] [20]

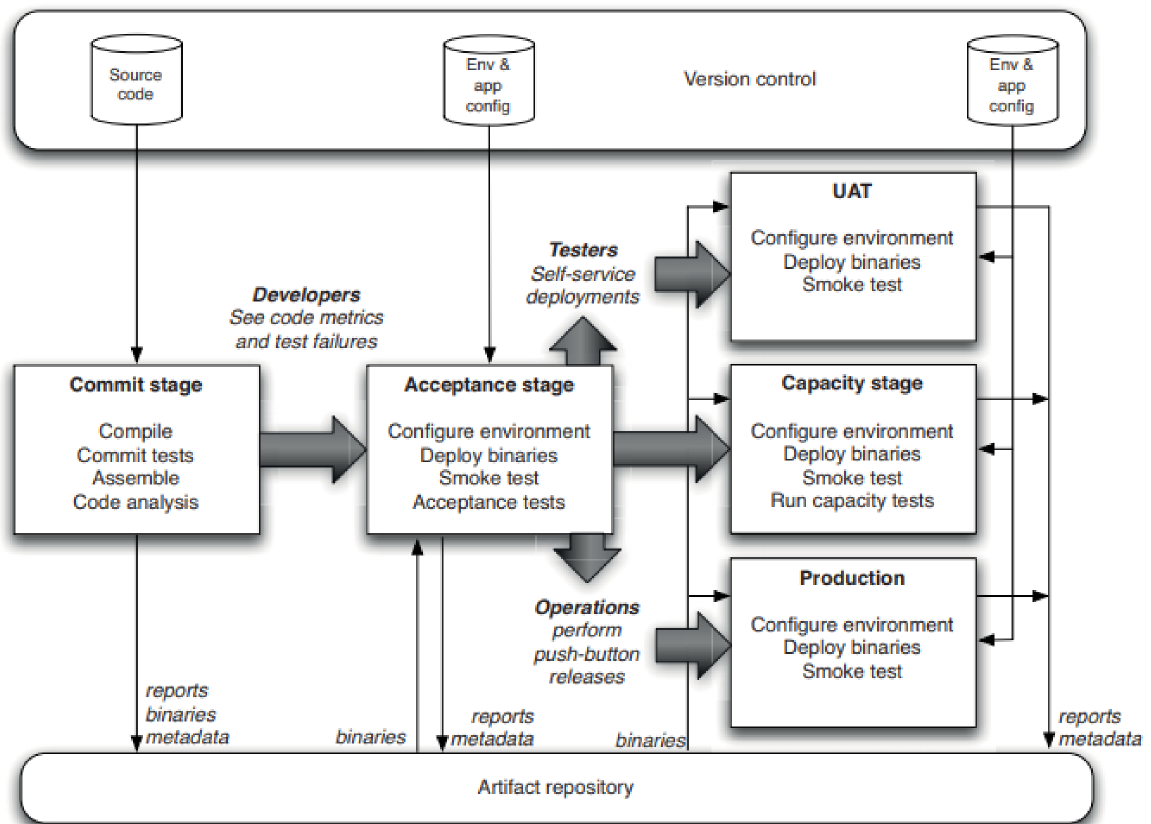
Na začátku procesu je vždy konkrétní verze softwaru ve verzovacím systému. Každá změna vytvoří nový build a nastává posloupnost kontrol pomocí testovací sady pro ověření, zda je verze softwaru vhodná pro nasazení do prostředí. Tento proces posloupnosti testovacích fází, z nichž každá vyhodnocuje úspěšnost vystavené verze z jiné úrovně posouzení, je zahájen obdobně jako proces CI, a to každou změnou ve verzovacím systému. Při procházení buildu jednotlivými úrovněmi testování se u něj zvyšuje důvěra pro vystavení na produkci. Cílem procesu je včasně odhalit neúspěšné verze buildu a doručit tak vývojovému týmu zpětnou vazbu o selhání co nejdříve. Výše popsaný proces je znázorněn sekvenčním diagramem na obr. 6. [12]



Obr. 6. Sekvenční diagram ukazující změny probíhajících v rámci procesu CI/CD. [12]

1.3.1.1 Základní Deployment pipeline

Celý proces, který překlápí integrovaný kód do produkčního softwaru je označován výrazem deployment pipeline. Na následujícím obrázku Obr. 7. je znázorněna základní podoba deployment pipeline zachycující podstatu přístupu CI/CD.



Obr. 7. Základní koncept deployment pipeline. [12]

Proces, jak je popsáno u kontinuální integrace, začíná u vývojářů, kteří provedené změny v kódu integrují do verzovacího systému. V tom okamžiku systém správy kontinuální integrace běžící v rámci CI serveru reaguje na změny vytvořením nové instance pipeline.

První fáze, neboli *Commit stage*, spuštěné pipeline provede kompilaci kódu, spustí jednotkové testy, vytvoří instalátory a provede případnou analýzu kódu. Jsou-li všechny jednotkové testy úspěšné, sestaví se spustitelný kód v binární podobě, tzv. artefakty, a uloží se do úložiště artefaktů. Moderní CI servery nabízejí možnosti pro ukládání artefaktů a jejich zpřístupnění v dalších fázích. CI servery taktéž umožňují napojení na specializované nástroje pro správu artefaktů, např. Artifactory či Nexus.

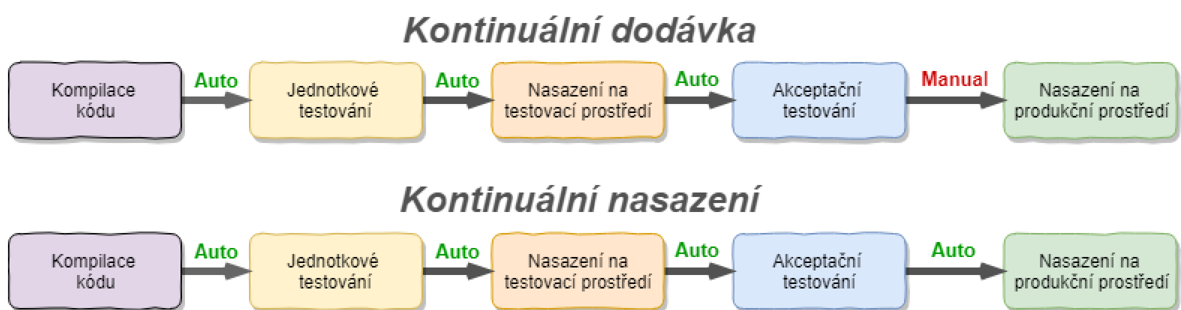
Proces druhé fáze, označován jako *Acceptance stage,* je typicky složen z všestranných automatizovaných testů. Časový předpoklad trvání této fáze je dlouhodobější, právě z důvodu náročnosti všestranných testů. Mnohé CI servery však umožňují rozdělení testů do balíčků, které pak mohou být vykonávány paralelně. Spuštění této fáze by mělo nastat automaticky po úspěšném dokončení předchozí fáze.

V tomto bodě je možné pipeline rozdělit a nasadit tak verzi buildu na různá prostředí. Prvním prostředím, které se nabízí, je testovací prostředí, tzv. UAT z anglického User Acceptance Testing, které slouží pro manuální testování na straně vývojového týmu. Druhým prostředím je Staging, ve schématu označeno jako *Capacity stage*. V rámci tohoto prostředí provádí své testování klient a může také sloužit pro nefunkcionální testování jako např. testování kapacity, tj. prověření, kolik uživatelů aplikace zvládne pojmout, dokud se výkon a stabilita aplikace stanou nepřijatelnými. Posledním typem prostředí je produkce – *Production*, což je prostředí ostrého provozu aplikace.

Po dokončení akceptačního testování je možné samoobslužné uveřejnění vybrané verze buildu. Toho lze dosáhnout pomocí automatizovaného scriptu pro vystavení. Po kliknutí na tlačítko je tak možné uveřejnit kteroukoli verzi buildu na vybrané prostředí. [12]

1.3.2 Kontinuální nasazení

Continuous deployment, neboli kontinuální nasazení, je plně automatizovaný proces kontinuální dodávky popsané výše. Každá nová změna, která úspěšně projde všemi fázemi pipeline, je automaticky uveřejněna do produkčního prostředí zákazníka. V procesu tak neexistuje žádný lidský zásah a pouze neúspěšný test zabrání novým změnám, aby byly nasazeny na produkční prostředí. Bez závěrečné manuální kontroly před vystavením na produkční prostředí je tak kladen větší důraz a odpovědnost na vývojáře, který je zodpovědný za správně implementované změny a testovací sadu pokrývající požadované funkčnosti aplikace. Kvalita testovací sady tak nepřímo určuje kvalitu vystavené verze. [12] [25]



Obr. 8. Porovnání dílčích kroků kontinuální integrace, doručování a nasazení. [20] [26]

Mezi benefity plynoucí z implementace kontinuálního nasazení patří urychlení vývoje, kdy není nutné mít pozastavený vývoj pro vystavení nové verze na produkční prostředí. Uvolnění nových změn je méně riskantní, v případě problému je snazší jeho náprava, jelikož se změny vystavují po malých dávkách. Nepřetržitým vystavování zákazníci vidí neustálý vývoj aplikace. [25]

1.4 Základní zásady vývojáře

Jsou-li na projektu implementovány zásady kontinuální integrace, je pro její efektivnost důležité, aby členové týmu, a to především vývojáři, měli zažité základní návyky vývoje softwaru. Vývojář by měl svůj kód často sdílet s ostatními, neprodleně opravit chyby, které způsobí rozbití buildu, psát automatizované testy a udržovat je v průchozím stavu, nesdílet s ostatními nefunkční či neúplný kód. Duvall ve své knize [8] doporučuje několik principů či návyků, které by se měli dodržovat při práci s kontinuální integrací. Jsou to často principy vycházející z používání verzovacích nástrojů.

1.4.1 Sdílet (commitovat) často svůj kód

Jelikož je jedním z hlavních principů kontinuální integrace pravidelné vytváření integračních buildů, a to i několikrát denně, je nezbytné, aby vývojáři pravidelně commitovali své změny kódu a ukládali je tak do centrálního úložiště kódu. Bude-li vývojář se sdílením svých změn čekat a bude vytvářet commity po několika dnech až týdnech, vzniknou konflikty při spojování verzí kódu a větší prostor pro chyby. To bude mít za následek větší časové režie na opravu buildu a podobně.

Vývojář většinou programuje na základě úkolu, kde má definováno, jaký je cíl úkolu a často také jaká je odhadovaná náročnost úkolu. Vývojář by si měl rozvrhnout práci na úkolu a commitovat změny postupně. Především jedná-li se o práci s odhadem v řádech jednotek až desítek hodin. Taktéž by měl vývojář vytvářet commity podle toho, na čem právě pracuje a neslučovat více úkolů do jednoho commitu.

1.4.2 Nesdílet nefunkční kód a spouštět lokální build

I při těch nejmenších úpravách kódů by si vývojář měl spustit lokální build v rámci IDE, ve kterém vyvíjí. Tím bude mít jistotu, že necommitne nefunkční kód. Stejně tak by vývojář neměl commitovat změny kódu, které jsou v rozpracovaném stavu a tím pádem neúplné a nefunkční.

1.4.3 Vytvářet testy pro vyvíjené časti

Jak je popsáno výše, neodmyslitelnou součástí integračních buildů softwaru je i spouštění jednotkových testů. Proto by mělo být zásadou, že při vytváření nových funkcionalit v projektu vývojář zavede i patřičné automatizované testy. Je-li k psaní testů přistupováno

zodpovědně, umožňují tak v rámci kontinuální integrace mít přehled o stavu jednotlivých komponent softwaru.

1.4.4 Udržovat úspěšnosti automatizovaných testů

Pokud je to možné, ať už z technického či časového hlediska, mělo by být pravidlem, že před commitnutím změn kódu si každý vývojář spustí jednotkové automatizované testy lokálně. Předejde tak zbytečným pádům integračního buildu. Test může dopadnout nezdarem, např. z důvodu vnesení chyby do očekávané funkčnosti kódu. V takovém případě je nutné kód opravit. Právě kvůli tomuto případu se automatizované testy používají. Neúspěšnost testu může však být způsobena i změnou očekávané funkčnosti kódu, především u agilního vývoje. V takovém případě je nutné přizpůsobit tělo testu, aby docházelo k testování očekávaného chování kódu a případný další neúspěch testu tak byl objektivní.

1.4.5 Okamžitá oprava neúspěšného buildu

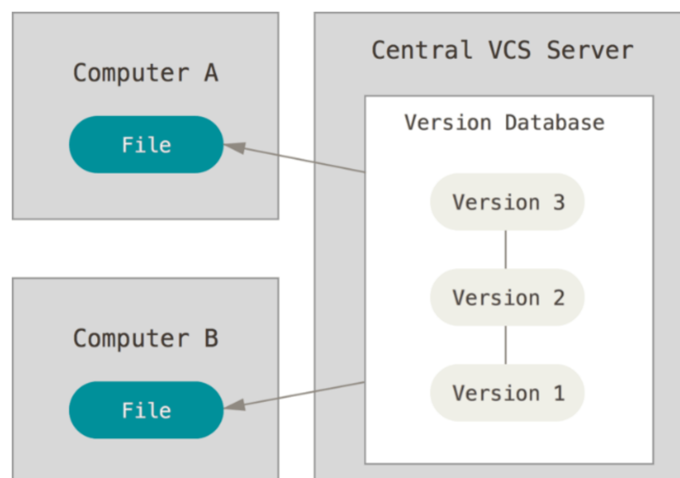
I při dodržování veškerých doposud zmíněných zásad může nastat, že integrační build skončí nezdarem. Pokud tzv. spadne integrační build, vývojář obdrží zpětnou vazbu o neúspěšném integračním buildu od CI serveru a jeho nejvyšší prioritou by mělo být sjednání nápravy. Pokud vývojář, jehož změny způsobily pád, není schopný sjednat nápravu do nějaké doby, kterou má tým stanovenou na opravu, měl by být jiný vývojář schopný vrátit jeho změny zpět, tzv. revert, aby bylo možné plynule pokračovat v pravidelném sestavování integračních buildů.

2 VERZOVACÍ SYSTÉMY

Verzovací systém, systém správy verzí (VCS) či systém řízení zdrojů je označení pro nástroje umožňující zaznamenávání historie změn v aplikacích včetně zdrojového kódu, dokumentace, definic databází, sestavovacích skriptů, testů a dalších nezbytných souborů pro vývoj softwaru. Verzovací systémy lze však použít pro správu jakýchkoli souborů.

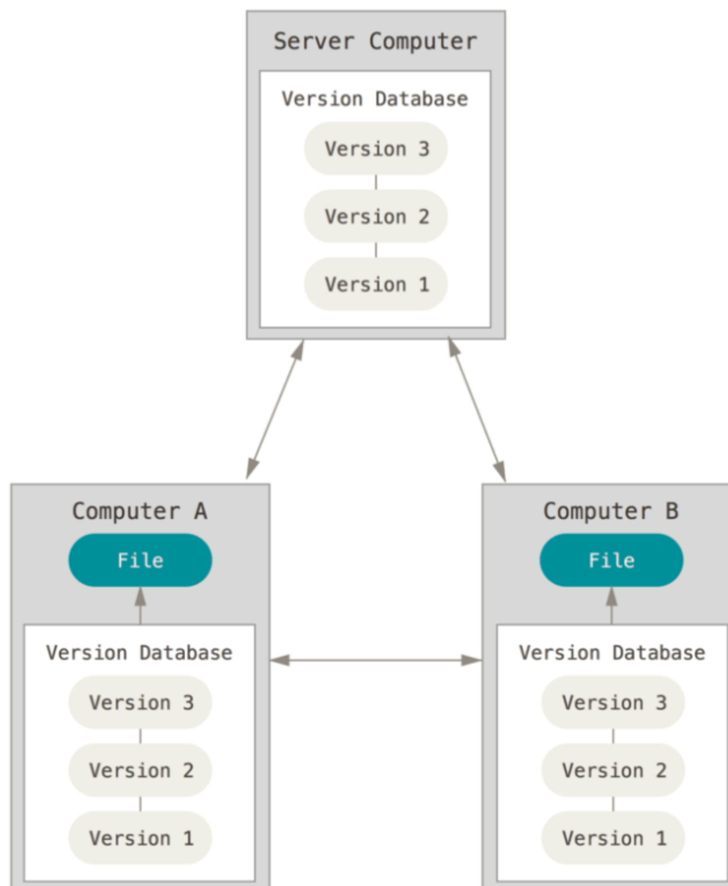
Pomocí verzovacího systému je možné vrátit vybrané soubory zpět do předchozího stavu, porovnávat provedené změny v čase, zjistit, kdo provedl které změny, prozkoumat, co mohlo být příčinou problému a další. Verzovací systémy se dělí na dva typy podle typu architektury. [27]

Centralizované verzovací systémy – tyto systémy mají jeden centrální server, který uchovává veškeré verze spravovaných souborů. Klienti se na tento server dotazují na jednotlivé verze souborů. Zástupci tohoto typu systému jsou např. Subversion či Team Foundation Version Control (TFVC).



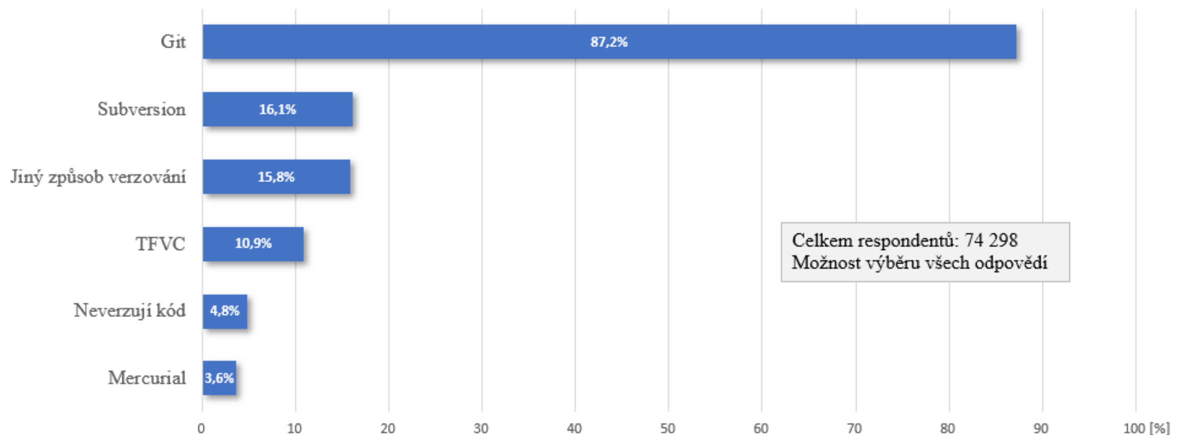
Obr. 9. Schéma principu centralizovaného verzovacího systému. [27]

Distribuované verzovací systémy – u těchto systémů se klienti nedotazují na poslední verze dílčí souborů, ale rovnou zcela zrcadlí centrální úložiště. Zástupci jsou např. Git, Mercurial, Bazaar, Fossil nebo Veracity.



Obr. 10. Schéma principu distribuovaného verzovacího systému. [27]

Verzovacích systémů je několik. Výše, v rámci rozdělení dle typu architektury systému, byly zmíněny ty nejznámější. Mezi ty nejvíce používané verzovací systémy se řadí Git, Subversion, TFVC a Mercurial. Uvedené pořadí verzovacích systémů odpovídá pořadí četností jejich používání podle průzkumu portálu StackOverflow. Tento portál provedl v roce 2018 dotazníkové šetření zaměřené na své početné komunity vývojářů. Zjišťoval používání technologií, mezi nimiž byly i verzovací systémy. Výsledek průzkumu je na obr. 11.



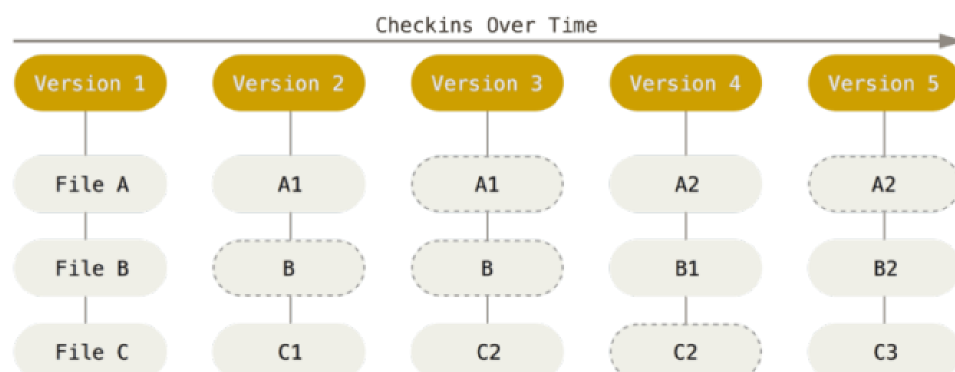
Obr. 11. Výsledky průzkumu používání verzovacích systémů při vývoji softwaru na portále StackOverflow v roce 2018. [28]

Pro značnou oblíbenost nástroje Git bude dále popsán princip fungování verzovacích systémů právě na tomto nástroji. Ostatní nástroje jsou do jisté míry v rámci obecného principu fungování obdobné a liší se převážně v uživatelské obslužnosti.

2.1 Git

První verze verzovacího nástroje Git byla uveřejněna v dubnu 2005 a stál za ní Linus Torvalds. Nástroj klade důraz na rychlost, jednoduchý design, podporu nelineárního vývoje, úplnou distributovatelnost a schopnost efektivní správy velkých softwarových projektů. Vlastnosti a principy, na kterých nástroj Git pracuje, podrobně popsali a publikovali Scott Chacon a Ben Straub ve své knize *Pro Git* [27].

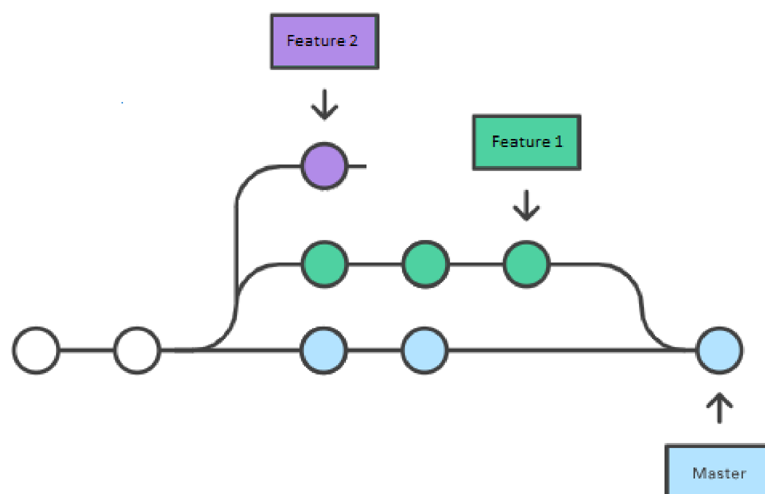
Git nahlíží na změny v datech jako na sadu snímků, tzv. *snapshots*. Při požadavku uložení změn si Git vytvoří snímky aktuálních souborů, které se změnily a uloží si odkazy na tyto snímky. Nezměněné soubory Git opakovaně neukládá, pouze si vytvoří odkaz na již dříve uložené soubory (snímky). Princip provázanosti ukládaných snímku je na obr. 12.



Obr. 12. Princip ukládání dat pomocí snímků v průběhu času. [27]

Jelikož je Git distribuovaný verzovací systém, snímky se ukládají do lokálního repozitáře na lokálním počítači, kde si Git uchovává data i potřebná metadata. Uložení nových změn do lokálního repozitáře se označuje výrazem *commit*. Každý commit má vlastní SHA–1 hash, na základě kterého je kontrolováno, zda při ukládání nedošlo k nějaké neočekávané změně či ztrátě dat. Lokální repozitář je možné kdykoli synchronizovat se vzdáleným centrálním repozitářem, označovaným jako *Origin,* a tím tak uložit veškeré změny kódu na vzdálené centrální úložiště. Stažení nových změn ze vzdáleného repozitáře je označováno jako *pull*. Opakem je uložení změn (commitů) do vzdáleného repozitáře – *push*.

Vytvářením commitů při své práci vzniká jejich souvislá linie označována jako *Master branch*. Git podporuje možnost větvení, tzv. *branching*. Větvení znamená oddělení se od aktuální aktivní větve a pokračování ve své práci nezávisle na dění v původní větvi. Mezi větvemi je možné se libovolně přepínat a také je navzájem slučovat, tzv. *Merge*.



Obr. 13. Ukázka práce s větvemi v rámci verzovacího systému. [29]

Větvení je aktivně využíváno při vývoji v týmu, kdy vývojáři tvoří svoji práci v rámci oddělených větví. Následně tyto větve slučují do jedné společné hlavní větve. Při vývoji v týmu nastávají nežádoucí situace, tzv. konflikty. To je stav, kdy více vývojářů změnilo současně stejnou část kódu a při slučování verzí není jasné, která verze má být upřednostněna. Takovou situaci je nutné manuálně vyřešit, aby mohly být verze spojeny s centrální verzí. [27]

2.1.1 Pravidla verzování

Práce s verzovacím systémem spadá do každodenní činnosti vývojáře. Pro maximální využitelnost a efektivitu by se měl vývojář držet následujících základních pravidel: [11]

1. **Pravidelné vytváření commitů** – Vytváření častých commitů prováděných změn v kódu, umožňuje častější sdílení kódu s ostatními. Takové změny je pak snazší integrovat pravidelně a vyhnout se konfliktům při slučování velkých pozměněných částí kódu.
2. **Commitovat související změny** – Každý commit by měl obsahovat pouze související změny, které jsou pro commit charakterizující. Například při opravě dvou různých chyb by měly vzniknout dva oddělené commity. To poté usnadní ostatním vývojářům pochopit změny a umožní tak snadněji vrátit změny, pokud se něco pokazí.
3. **Necommitovat rozpracovanou práci** – Měly by se vytvářet commity pouze dokončeného a funkčního kódu. To však neznamená, že je nutné dokončit celou složitou funkcionalitu před vytvořením commitu. Naopak zpracování funkcionality by mělo probíhat po logických částech, které by se měly postupně commitovat. Pokud např. nastane nutnost mít čistou pracovní kopii kódů, nabízí se uložit si změny dočasně, tzv. funkce Stash, než vytvářet commit s nefunkční částí.
4. **Otestování před vytvořením commitu** – Vývojář by neměl vytvářet commity pokud se pouze domnívá, že je vše pořádku. Svou práci by si měl otestovat a ujistit se, že provedené změny nezpůsobily nežádoucí vedlejší účinky.
5. **Výstižný popis commitu** – Popis změn by měl být výstižný a přesný. Mělo by být z popisu čitelné, z jakého důvodu se prováděly změny a jak se liší změny od předchozí implementace.
6. **Verzovací systém není zálohovací systém** – Zálohování změn na vzdáleném serveru je přínosný vedlejší efekt verzovacích systémů. Verzovací systém by se však neměl používat jako pouze zálohovací systém. Při verzování by měla být obezřetnost na souvislosti verzovaných změn, viz bod 2.
7. **Větvení** – Možnost větvení, tzv. branching, je užitečnou vlastností verzovacího systému. Pomocí větví je možné se snadno vyhnout míchání různých směrů vývoje. Větve by měly být nedílnou součástí při vývoji a vývojář by měl pracovat v oddělených větvích u každé funkcionality, bugu či jen nápadu.

3 AUTOMATIZOVANÉ TESTOVÁNÍ SOFTWARU

Testování je aktivita, kterou by se měl zaobírat celý tým a měla by být prováděna od začátku projektu. Vytváření kvalitních automatizovaných testů pro více úrovní testování, které jsou vykonávány v rámci kontinuální integrace, by mělo být zažitou praxí. Manuální testování softwaru však stále nepostrádá svůj význam, jelikož průzkumné testování a testování použitelnosti by mělo probíhat průběžně v rámci celého vývoje projektu. [12]

Při psaní automatizovaných testů by měli spolupracovat testeři, vývojáři i běžní uživatelé softwaru. Testy jsou vytvářeny dříve, než vývojář začne implementovat požadovanou funkčnost, kterou testy prověřují. Tento přístup je označován jako programování řízené testy, z anglického výrazu *Test-driven development* (TDD). Tuto techniku založenou na malých stále se opakujících krocích při práci s testy začal prosazovat Kent Beck v rámci extrémního programování [15].

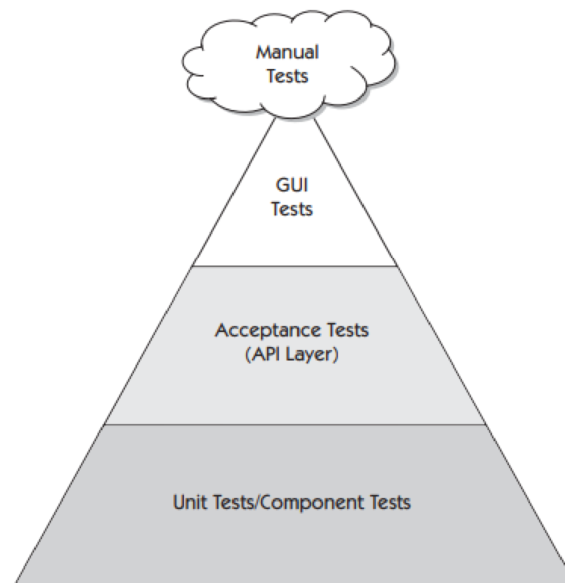
Vytvořený soubor testů tak tvoří spustitelnou specifikaci očekávaného chování a při jejím úspěšném vykonání dokazují, že funkčnost softwaru požadovaná zákazníkem je správná.

Strategie testování je důležitým procesem identifikace rizik projektu a opatření, které je zapotřebí přijmout pro jejich zmírnění. Dobrá strategie testování přináší mnoho pozitivních efektů. Díky testování si můžeme být jistí, že software funguje, jak má, což znamená méně chyb, snížení nákladů na podporu a lepší pověst.

Oblast testování není hlavním cílem této práce, a tak jsou dále naznačeny pouze základní principy a charakteristika. Oblastí testování softwaru se detailněji zabývají Lisa Crispin a Janet Gregory, kteří své poznatky prezentují v knize *Agile testing: a practical guide for testers and agile teams* [14].

3.1 Pyramida automatizovaných testů

Pyramida automatizovaných testů nabízí zjednodušený pohled na různé druhy automatizovaných testů potřebných pro vytvoření vyváženého portfolia testovcí sady. Hlavní podstatou je dosáhnout nejvyššího zastoupení nízkoúrovňových testů.



Obr. 14. Testovací pyramida. [14]

Nejnižší vrstva tvoří robustní základ pro všechny ostatní vrstvy. Je tvořena především jednotkovými testy a testy komponent. Většina automatizovaných testů této vrstvy je psána ve stejném programovacím jazyce jako samotný kód softwaru. Tyto testy bývají v porovnání s ostatními nejrychlejší, často jsou také nejméně nákladné a poskytují nejrychlejší zpětnou vazbu.

Testy střední vrstvy pyramidy jsou především funkčními testy, které ověřují, zda vytvářený software splňuje očekávání klienta. Do této vrstvy se řadí testy integrační, akceptační a jim podobné. Tyto testy pracují na úrovni API a testují funkčnost bez nutnosti grafického rozhraní.

Poslední vrstva představuje testy, které jsou úzce spjaté s prezenční vrstvou softwaru. Tyto testy by měli mít nejmenší zastoupení, jelikož mají obecně nejnižší návratnost. Správa takovýchto testů je nákladnější především z toho důvodu, že prezenční vrstva se často mění a tyto testy tak vyžadují častější správu než testy, které pracují na funkcionální či jednotkové úrovni. [14]

3.2 Funkční vs. nefunkční testování

Funkční testování je typ testování, v rámci kterého je cílem ověřit, zda chování funkcionality softwaru je v souladu se specifikací požadavků. Každá funkčnost systému je testována ověřením výstupu na základě odpovídajícího vstupu a porovnáním skutečných

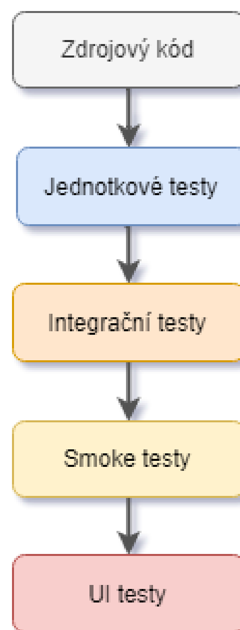
výsledků s očekávanými výsledky. Tato kategorie testování zahrnuje kontrolu uživatelského rozhraní, rozhraní API, databáze, zabezpečení či funkčnosti aplikace.

Nefunkční testy oproti funkčnímu testování spočívají v testování všech vlastností aplikace, které přímo nesouvisí s jejími funkcemi, ale zároveň jsou podstatné pro její správné fungování. Mezi nefunkční testy můžeme zařadit např. výkonové testování, kde se testuje aplikace z pohledu vytížení (větší počet uživatelů pracující s aplikací ve stejný čas). [16]

Tab. 1. Porovnání vlastností funkčního a nefunkčního testování. [7]

Funkční testování	**Nefunkční testování**
Funkční testování se provádí pomocí funkčních specifikací poskytovaných klientem. Ověřuje systém podle funkčních požadavků.	Nefunkční testování kontroluje výkonnost, spolehlivost, škálovatelnost a další nefunkční aspekty softwarového systému.
Funkční testy jsou prováděny jako první.	Nefunkční testy jsou prováděny až po dokončení funkčních testů.
Pro funkční testování může být použit proces jak manuální i automatizovaný.	Použití nástrojů pro nefunkční testování je efektivnější.
Vstupy pro funkční testování jsou obchodní požadavky.	Vstupy pro nefunkční testování jsou parametry výkonosti, např. rychlost či škálovatelnost.
Funkční testování prověřuje, co software dělá.	Nefunkční testování prověřuje, jak dobře software funguje.
Typy funkčních testů jsou např. • Jednotkové testy • Integrační testy • Smoke testy • White box testování • Black Box testování • Akceptační testy	Typy nefunkčních testů jsou např. • Výkonové testy • Zátěžové testy • Bezpečnostní testy • Instalační testy • Penetrační testy • Migrační testy

Dílčích typů testů které jsou popsány výše je velké množství. Dále jsou uvedeny typy testů, které mají největší význam v kategorii automatizované testy.



Obr. 15. Proces zpracovávaných automatizovaných testů. [20]

3.3 Jednotkové testy

Jednotkové testy, tzv. unit testy, pracují přímo proti kódu aplikace a nikoli proti uživatelskému rozhraní. Jednotkové testy ověřují chování částí kódu, tj. např. vlastností tříd či metod. Jednotkové testy by měly být psány tak, aby zcela kontrolovaly funkcionalitu vybrané části kódu s očekávanými požadavky chování softwaru. Testování pomocí jednotkových testů je příkladem tzv. White Box testování. Každý jednotkový test je sám o sobě kus kódu a ačkoli jsou typicky psané ve stejném jazyce jako kód aplikace, není to technicky nutné.

U jednotkových testů je kladen důraz na jednoduchost a rychlost. V rámci tohoto typu testů se neprovádí integrace s databází či jinými službami jako http serverem apod. [18]

3.4 Integrační testy

Integrační testy jsou určené k testování logiky celých softwarových modulů a integraci mezi těmito moduly, nikoliv k testování funkčních částí, které by měli být ověřovány v rámci zmíněných jednotkových testů. Integrační testovací scénáře se liší od ostatních testovacích scénářů v tom smyslu, že se zaměřují především na rozhraní a tok dat, přesněji na informace předávané mezi moduly.

Typický softwarový projekt se skládá z několika softwarových modulů, které byly tvořeny různými vývojáři. Integrační testování se zaměřuje na kontrolu datové komunikace mezi těmito moduly, na které je nahlíženo jako na skupinu.

Přestože každý softwarový modul může být úspěšně otestován jednotkovými testy, stále mohou v softwaru z různých důvodů nacházet chyby. [19]

- Modul je obecně navržen individuálním vývojářem softwaru, jehož porozumění a programovací logika se může lišit od ostatních programátorů. Integrační testování je nezbytné pro ověření toho, že softwarové moduly pracují v jednotě.
- V době vývoje modulu existuje velká šance na změny požadavků klientů. Tyto nové požadavky nemusí být podchyceny při testování jednotkovými testy, a proto je nutné uvažovat i o komplexnějším testování systému.
- Navržená struktura softwarových modulů se nemusí slučovat s vlastnostmi jednotlivých tabulek databáze a může tak docházet k chybám při komunikaci.
- Pokud existují externí hardwarová rozhraní, mohou postrádat či mít rozdílné konfigurační nastavení.
- Problémy může taktéž způsobit i nedostatečná manipulace s výjimkami.

Softwarový průmysl používá řadu strategií k provádění integračních testů: [19]

- **Přístup velkého třesku** (Big Bang) – Všechny moduly jsou integrovány dohromady a najednou. Poté jsou testovány jako celek.
- **Přírůstkový přístup zespodu nahoru** (Top Up) – Při tomto přístupu se testování provádí spojením dvou nebo více modulů, které jsou logicky příbuzné. Poté jsou přidávány a testovány další související moduly pro správnou funkci. Proces pokračuje, dokud nedojde ke spojení a úspěšnému otestování všech příbuzných modulů.
- **Přírůstkový přístup shora dolů** (Top Down) – Testování je obdobné jako v případě přírůstkového testování zespoda nahoru s tím rozdílem, že směr testování je opačný, tj. testování probíhá shora dolů.
- **Přírůstkový přístup sandwich** – Metoda testování, která je kombinací přírůstkového přístupu shora dolů a zespoda nahoru.

3.5 Smoke testy

Testování kouřem, v anglickém jazyce označováno výrazy „SmokeTesting“ či „*Build Verification Testing*“, je typ testování softwaru, který zahrnuje neúplný soubor testů. Testování se provádí po sestavení softwaru a jeho cílem je prověřit, zda kritické funkce softwaru fungují správně. Testování kouřem se taktéž provádí před provedením podrobných funkčních nebo regresních testů, jelikož není cílem testování provádět důkladné testování, ale kontrolu, zda daná verze softwaru je v základním funkčním stavu. Je-li testování kouřem neúspěšné je to známka toho, že verze softwaru je nestabilní a natolik poškozená, že by testovací tým ztrácel čas instalací a testováním softwaru. V takovém případě je nutné zadat požadavky na opravu softwaru a vyčkat na jejich dokončení.

Při vytváření Smoke testů je nezbytné pokládat si základní otázky pro fungování aplikace např. [21]

- Je možné přistoupit k aplikaci a přihlásit se do ní?
- Je GUI aplikace interaktivní?
- Je možné v aplikaci přecházet z jednoho okna do druhého?

Testování kouřem napomáhá odhalit nesprávnou integraci a závažné problémy na počátku cyklu testování. Lze jej provádět na nově vytvořených částech softwaru i po zapracování změn softwaru. Testování se může provádět manuálně i s pomocí automatizačních nástrojů. Spouštění probíhá v rámci procesů CI/CD.

Automatizované kouřové testy často běží rychle, což přináší výhody rychlejší zpětné vazby oproti testování rozsáhlých testovacích sad, které se provádí mnohonásobně déle. Mezi další výhody Smoke testů se řadí např. [22]

- Odhalení problémů s integrací kódu.
- Včasné odhalení problémů.
- Poskytuje jistotu, že provedené změny v softwaru nepříznivě neovlivnily hlavní funkcionality softwaru a oprávněně je možné provádět další podrobnější testování softwaru.

3.6 UI testy

Obecně jsou dva typy uživatelského rozhraní aplikací. Rozhraní příkazové řádky, tzv. „*Command line*“, u kterého na základě uživatelského zadávání textových příkazů aplikace

reaguje a zobrazuje výsledky textovou formou. Druhým typem je grafické uživatelské rozhraní, z anglického spojení Graphical User Interface (GUI), které umožňuje ovládání aplikace pomocí grafických prvků a komunikace tak probíhá pomocí grafických komponent.

Cílem při UI testování je prověření grafického uživatelského rozhraní ve vystaveném testovacím prostředí. Testování tak zahrnuje kontrolu dílčích obrazovek aplikace a jednotlivých ovládacích prvků, jako je menu, tlačítka, nástrojové panely, dialogová okna apod. Při testování se klade důraz na otestování následujících náležitostí [23]:

- Zda všechny prvky grafického uživatelského rozhraní mají správnou velikost, umístění a zda vstupy přijímají očekávané znaky nebo čísla.
- Zda je možné pomocí GUI provádět požadovanou funkci aplikace.
- Zda jsou chybové zprávy zobrazovány správně.
- Zda je použité písmo čitelné a texty jsou správně zarovnány.
- Zda jsou grafické prvky správně zarovnány.
- Zda je umístění jednotlivých prvků GUI pro různé rozlišení obrazovky.

UI testování má tři přístupy k testování [23]:

- **Manuální testování** – Při tomto přístupu je grafické uživatelské rozhraní aplikace testováno manuálně testerem.
- **Nahrání a opakované přehrání** – Pro testování se využívají automatizační nástroje a testování je prováděno ve dvou fázích – nahrávání, přehrávání. Během pořizování záznamu jsou testovací kroky zachycovány automatizačním nástrojem. Během přehrávání jsou zaznamenané kroky provedeny nad vystavenou verzí aplikace.
- **Modelové testování** – U modelového testování je cílem vytvořit model, který je grafickým popisem softwaru. Tento testovací model je možné automaticky spouštět nad vystavenou verzí aplikace v rámci procesu kontinuální integrace.

Nástrojů pro automatizované testování uživatelského rozhraní je nepřeberné množství. Příkladem mohou být nástroje Selenium, AutoHotkey, Sikuli a další. Pro orientaci mezi dostupnými nástroji pro automatizované GUI testování je možné využít podrobný seznam těchto nástrojů na Wikipedia [24].

4 NÁSTROJE CI/CD

Nástrojů pro kontinuální integraci a kontinuální doručování je v softwarovém světě již spousta. Při výběru nástroje CI/CD bychom se měli zaměřit na následující vlastnosti:

- **Možnost hostování –** Nástroje se liší v tom, zda jsou hostovány na straně poskytovatele – vyžadují menší nároky na počáteční nastavení. Nebo zda se jedná o vlastní hostování, kde je zodpovědnost za nastavení a údržbu pouze na uživateli.
- **Integrace a flexibilita** – Jde o to, jak dobře je nástroj CI/CD integrovatelný s ostatními nástroji pro vývoj software, např. nástroje pro řízení projektů nebo statické analýzy. Taktéž by měl mít nástroj dostatečnou flexibilitu v podpoře buildovacích nástrojů či systémů pro verzování kódu.
- **Použitelnost –** Možnosti a použitelnost rozhraní nástroje mohou učinit proces sestavení pipelines mnohem jednodušší.
- **Rozšíření –** Výhodou nástroje může být veřejně dostupné úložiště nejrůznějších pluginů, použitelných kroků sestavení, které jsou open-source, nebo komerčně dostupné.

4.1 Azure DevOps Server a Azure DevOps Services

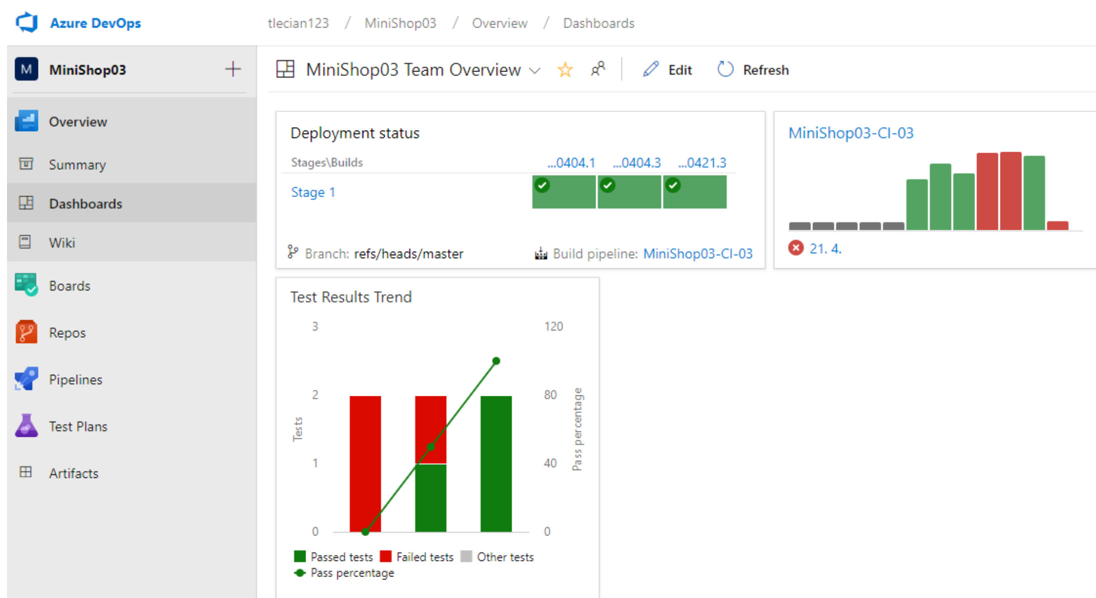
Azure DevOps Server, dříve znám pod názvem Team Foundation Server (TFS), je sada nástrojů pro spolupráci členů týmu při vývoji software. Azure DevOps Server je on-premise řešení, tj. software, který je nainstalovaný na lokálním serveru. Společnost Microsoft, která nástroje vyvíjí a licencuje, nabízí i cloudové řešení. Cloudové služby vystupují pod obchodním názvem Azure DevOps Services, dříve označovány také jako Visual Studio Team Services (VSTS). Obě řešení poskytují integrované rozhraní pro správu. Základem rozhraní je widgetový dashboard, který je uživatelsky přizpůsobitelný a umožňuje mít přehled nad využívanými nástroji Azure DevOps.

Ačkoli obě řešení poskytují stejné základní služby, při výběru mezi cloudovým a on-premise řešením jsou následující rozdíly, které mohou zásadně ovlivnit používání služeb:

- **Rozsah a škálovatelnost dat** – Azure DevOps Services umožňuje škálovatelnost pomocí organizací a projektů. Každá organizace může obsahovat několik projektů, které tvoří jednu kolekci. Uživatelé tak mohou přistupovat pouze k organizacím

ke kterým jim byl zaplacen přístup. Azure DevOps Server pak umožňuje škálovatelnost pomocí topologie nasazených serverů, kolekcí projektů a projektů samotných.

- **Ověřování uživatelů** – K službě Azure DevOps Services uživatelé přistupují přes internet a jejich autentifikace probíhá na základě účtu Microsoft nebo Azure Active Directory. U Azure DevOps Server se uživatelé připojují k vlastnímu serveru, kde ověření probíhá na základě systému Windows a prověření vlastních doménových služeb Active Directory.
- **Správa přístupů uživatelů a skupin** – Ve službě Azure DevOps Services musí být každému uživateli organizace přiřazena úroveň přístupu. Ověřování uživatelů následně probíhá na základě úrovně přístupu a předplatného. Přístup uživatelů k službě Azure DevOps Server závisí pouze na nastavené úrovni přístupu, jelikož přístupy se vztahují k licenci celé služby.
- **Bezpečnost a ochrana údajů** – Zatímco u Azure DevOps Services je bezpečnost a ochrana dat dána cloudovou službou, u Azure DevOps Server (on-premise řešení) je zcela na provozovateli serverů.
- **Reporting** – Obě varianty služeb nabízejí několik nástrojů pro sledování kvality vývoje. Azure DevOps Server (on–premise rešení) však nabízí navíc možnost nasazení SQL Server Reporting Services pro analýzu vývoje a kvality projektu.



Obr. 16. Ukázka dashboardu webového rozhraní nástroje Azure DevOps Services s widgety zobrazujícími výsledky procesu CI/CD.

Hlavním nástrojem Azure DevOps pro kontinuální integraci a doručení je Azure Pipelines. Tento nástroj umožňuje vytvoření buildu a otestování běžně používaných programovací jazyků včetně C/C++, .NET, Java, PHP, Python, Node.js, HTML, Go, Ruby, Android či iOS apps. Umožňuje nadefinovat několik předpisů (pipelines) pro vytváření buildů a nasazení na libovolnou platformu nebo cloud. Pipeline je možné integrovat s jakýmkoli Git repozitářem, Subversion či Azure Repos. Azure Repos je další službou Azure DevOps pro správu kódu. Automatizované dílčí kroky pipeline se tvoří pomocí úloh. Úloha je určitou abstrakcí pro skript nebo proceduru s předem určenými vstupy.

Azure Pipelines umožňuje vystavení softwaru přímo na virtuální stroj nebo využití libovolného správce virtuálních strojů, např. VMWare či Center Virtual Machine Manager. Azure Pipelines podporuje také vystavování na cloudové služby, např. na Azure, Google Cloud Platform, Amazon Web Services nebo pomocí rozšiřujících úloh i na další cloudové služby.

Azure Pipelines má mnoho předdefinovaných úloh pro vytváření pipelines, které významně zjednodušují vytváření předpisů pro kontinuální integraci a doručení. Detailní popis a ukázka praktického nasazení procesu CI/CD nástroje Azure Pipelines je v kapitole *6. Implementace CI/CD.*

V rámci Azure DevOps je možné využívat další služby, které jsou se službou Azure Pipelines spjaty a spolu tvoří samotnou podstavu DevOps.

- **Azure Repos** – Jsou nástroje pro správu verzí kódu, které je možné využít i pro řízení vývoje. Azure Repos poskytuje dva typy přístupů – distribuované řízení verzí pomocí Git a centralizované řízení verzí pomocí Team Foundation Version Control.
- **Azure Boards** – Je služba pro správu a řízení softwarového projektu, která nabízí funkce pro řízení agilních projektů s podporou kanbanu (přesouvání štítků), přizpůsobitelných řídících panelů a reportingu.
- **Azure Test Plans** – Služba umožňuje řízení a monitoring automatizovaných testů v životním cyklu vývoje software.
- **Azure Artifact** – Umožňuje uchovávat a spravovat artefakty pro jednotlivé pipelines kontinuální integrace a kontinuálního doručování.

Tab. 2. Základní přehled nástroje Azure DevOps Services/Azure DevOps Server.

Název	Azure DevOps Services/Azure DevOps Server (Microsoft)
Hostování	On-Premise/Cloud
Platforma	Multiplatformní
Placená služba	Ano, pro open-source projekty zcela zdarma
Zkušební verze	Ano
Podpora kontejnerů	Ano
Možnosti sestavení	MSBuild, Visual Studio, Docker, Maven, Ant, Android, Gradle, C, C++, CMake, Python, Node.js, Ruby, PHP, Perl, Go
Vznik	2006

4.2 Jenkins

Jenkins je open-source nástroj kontinuální integrace, který je kompilovaný a psaný v programovacím jazyce Java. Je to multiplatformní nástroj s konfiguracemi pro GUI rozhraní i pro příkazový řádek. Je primárně určen pro nasazení na lokální servery (on-premises) avšak může být nasazen i na cloudových serverech. Nástroj Jenkins nemá ve svém základu tolik funkcionalit jako např. Azure DevOps Server, avšak jeho funkcionality je možné jednoduše rozšířit pomocí doplňků a pluginů. Použití pluginů je téměř nevyhnutelné při nasazení procesu kontinuální integrace. Jelikož je Jenkins open-source software, s početnou komunitou, je dostupných více jak 1 000 nejrůznějších pluginů.

Obr. 17. Ukázka webového rozhraní nástroje Jenkins.

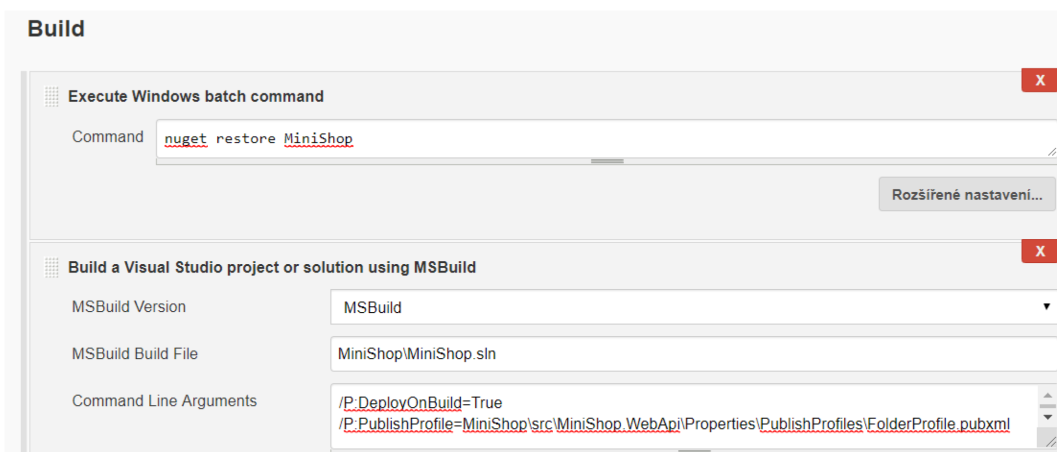
Jedna instance Jenkins serveru může zpracovávat několik vývojových projektů. Definice CI procesu neboli pipeline, se provádí pomocí tzv. Jenkins projektů, které je možné libovolně seskupovat do tzv. views. Jenkins projektů je několik a ve své podstatě definují

způsob práce s nástrojem Jenkins. Mezi základní typy pro proces CI/CD patří např. *Freestyle project* či *Pipeline*.

Jenkins v základu nabízí integraci se dvěma verzovacími systémy, Git a Subversion. Pro privátní repozitáře je možné nadefinovat vlastní SSH spojení. Jenkins pro kontrolu změn v repozitářích nabízí obdobné možnosti jako Azure Pipelines, např. periodickou kontrolu změn repozitáře či plánování konkrétních časů spuštění buildu. Pomocí pluginů je možné rozšířit možnosti spouštění buildů, např. o spuštění buildu na konkrétní akci repozitáři pomocí web hooku.

Jenkins používá architekturu Master-Slave, kde master je hlavní server, který monitoruje otroky, což jsou vzdálené počítače pro distribuci softwaru a testování. Master server se stará o plánování úloh, sleduje podřízené jednotky, nahrává a plánuje sestavení. Otroci, kteří mohou běžet na různých operačních systémech, naslouchají hlavní instanci Jenkins master a zpracovávají požadavky.

Jenkins ve svém základu neumí vytvořit build technologie .NET. Je nutné nainstalovat MSBuild plugin, který Jenkins o tuto možnost rozšíří. Obdobně je tomu i v případě, že chceme spustit jednotkové testy pomocí nástroje MSTest. Je nutné na server doinstalovat odpovídající plugin pro možnost spuštění jednotkových testů.



Obr. 18. Ukázka buildu .NetCore aplikace v rámci Jenkins Freestyle projektu.

Zatím co *Jenkins Freestyle projekt* umožňuje definování kroků kontinuální integrace pomocí předem definovaných bloků a je tak určen pro jednodušší vývojové projekty. *Jenkins projekt Pipeline* umožňuje definování celého procesu kontinuální integrace pomocí CLI příkazů vsazených do struktury pipeline. To sebou přináší možnost vytváření složitějších definic buildu.

```
1  pipeline {
2      agent any
3      stages {
4          stage('Clone') {
5              steps {
6                  bat 'git clone https://tomasLecian@bitbucket.org/tomasLecian/minishop.git'
7              }
8          }
9          stage('RestorePakages') {
10             steps {
11                 bat 'dotnet restore MiniShop/MiniShop.sln'
12             }
13         }
14         stage('Build') {
15             steps {
16                 bat 'dotnet build MiniShop/MiniShop.sln /p:DeployOnBuild=True
17                 /p:PublishProfile=MiniShop/src/MiniShop.WebApi/Properties/PublishProfiles/FolderProfile.pubxml'
18             }
19         }
20         stage('Publish') {
21             steps {
22                 bat 'dotnet publish MiniShop/MiniShop.sln'
23             }
24         }
25     }
26 }
27
```

Obr. 19.Ukázka buildu .NET Core aplikace v rámci Jenkins Pipeline projektu.

Tab. 3. Základní přehled nástroje Jenkins.

Název	Jenkins
Hostování	On-Premise/Cloud
Platforma	Multiplatformní
Placená služba	Open-source
Zkušební verze	Ano
Podpora kontejnerů	Ano
Možnosti sestavení	MSBuild, Batch Script, Docker, Maven, Ant, NAnt, Android, Gradle, C, C++, CMake, Python, Node.js, Ruby, PHP, Perl, Go
Vznik	Od roku 2004 vyvíjeno jako Hudson projekt. Od roku 2011 je Jenkins vyvíjen samostatně.

4.3 BitBucket Pipelines

BitBucket Pilelines je poměrně novým nástrojem, který integruje proces CI/CD do webové služby BitBucket. BitBucket je cloudová služba pro správu repozitářů zdrojových kódů, jejíž základem jsou verzovací systémy Git a Mercuril. BitBucket Pilelines umožňuje automaticky sestavovat, testovat a nasazovat kód aplikace na základě konfiguračních souborů ve formátu YAML uložených přímo v repozitáři.

BitBucket Pipelines spouští všechny buildy pomocí Docker image. Pro každý krok buildu je možné použít jiný Docker image. Jelikož každý krok buildu je spuštěn v odděleném

kontejneru (kontejner označuje instanci Docker image), je zapotřebí to mít na paměti a spravovat všechny artefakty, které mají být zachovány mezi dílčími kroky buildu. BitBucket Pipelines totiž nad tímto neposkytuje žádnou kontrolu.

BitBucket Pipelines neumožňuje spustit build přímo na virtuálním stroji nebo přímo na konkrétním hardware. Taktéž není zatím možné provádět build Windows, macOS nebo iOS aplikací.

```
1  image: microsoft/dotnet:latest
2
3  pipelines:
4      default:
5
6      - step:
7          name: WebApi build
8          caches:
9              - dotnetcore
10
11         script:
12             #Definition of variables
13             - export PROJECT_NAME=MiniShop
14             - export NAME_SPACE=$NAME.WebApi
15             - export PROJECT_SRC=./src/$PROJECT_NAME
16             - export ZIP_FILE_SRC=dist/$PROJECT_NAME.zip
17             - mkdir dist
18
19             #1. Restore NuGet packages
20             - dotnet restore $PROJECT_SRC/$PROJECT_NAME.sln
21
22             #2. Build
23             - dotnet build $PROJECT_SRC/$PROJECT_NAME.sln
24
25             #3. Tests
26             - dotnet test $PROJECT_SRC/tests/NAME_SPACE.tests.csproj
27
28             #4. Publish
29             - dotnet publish $PROJECT_SRC/$PROJECT_NAME.sln -c Release
30
31             #5. Utilities
32             - apt-get update
33             - apt-get -qq install apt-utils
34             - apt-get -qq install zip
35
36             #6. Zip the publish
37             - zip -j -r $ZIP_FILE_SRC $PROJECT_SRC/bin/Release/netcoreapp2.2/publish
38
39         artifacts:
40             - dist/**
```

Obr. 20. Ukázka .NET Core aplikace MiniShop v nástroji BitBucket Pipelines.

Ukázka definice jednoduchého buildu .NET Core aplikace MiniShop je na obr. 20. Soubor YAML s příponou .yml, který se umisťuje do centrální složky repozitáže obsahuje následující dělení:

- **Pipelines** – Označuje začátek celé pipeline definice.

- **Default** – Obsahuje všechny kroky, které budou spuštěny a provedeny po každém přidání nového kódu do repozitáře.
- **Step** – Každý step spouští nový Docker kontejner, který obsahuje klon repozitáře a vykonává se v rámci něj definovaný skript.
- **Script** – Sekce obsahuje příkazy, které se mají sekvenčně vykonat.

Použitý Docker kontejner *microsoft/dotnet,* prostřednictvím kterého běží build aplikace (Obr. 20), je volně přístupný kontejner na Docker Hub od společnosti Microsoft [33]. Je možné si nadefinovat vlastní kontejner a pouštět build v rámci něj.

Nástroj BitBucket Pipelines je poměrně omezený ve svých možnostech. Nabízí základní sadu funkcí pro kontinuální integraci, se kterou si vystačí malé projekty. Avšak pro složitější build sestavy může být tento nástroj nedostačující. Především pokud požadujeme vlastní reportování, statické analýzy trendů, výsledky testů v čase atd.

BitBucket Pipelines umožňuje odesílání emailů všem členům, kteří jsou součástí týmu nebo kteří sledují úložiště při neúspěšném buildu. K dispozici je integrace HipChat či Slack. Není možné však nedefinovat pravidla pro zasílání zpráv, např. komu se pošle která zpráva apod.

Tab. 4. Základní přehled nástroje BitBucket Pipeline.

Název	BitBucke Pipelines (Atlassian)
Hostování	Cloud
Platforma	Web kontejner
Placená služba	Ano, cena se odvíjí od počtu členů týmu a zprocesovaného času
Zkušební verze	Ano
Podpora kontejnerů	Ano
Možnosti sestavení	Batch Script, Docker, Maven, Gradle, C, C++, CMake, Python, Node.js, Ruby, PHP, Rust, Go, Haskell, Clojure
Vznik	2016

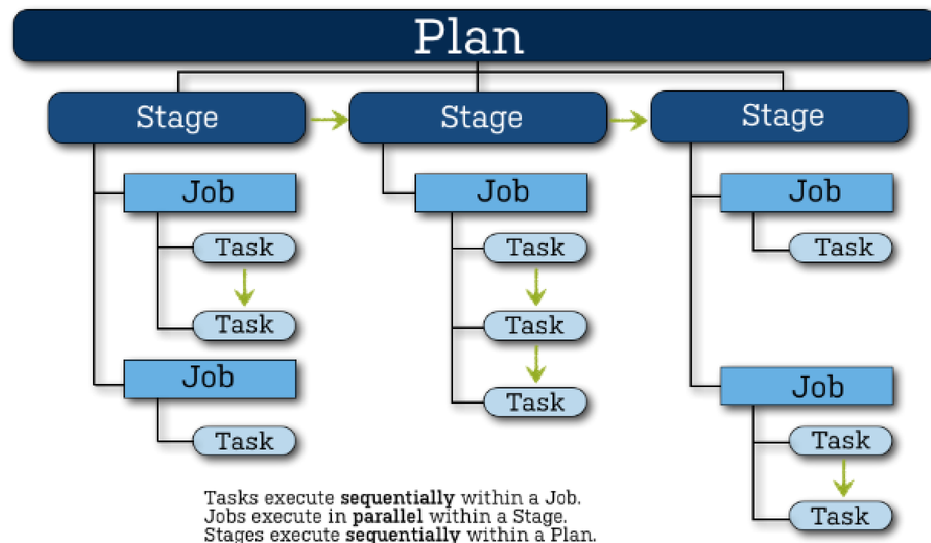
4.4 Bamboo

Bamboo je CI/CD server vyvinutý společností Atlassian v jazyce Java navržený pro on-permise řešení CI/CD. Rozvoj cloudové služby v podobě *Bamboo Cloud* společnost

Atlassian ukončila v roce 2017 [36] a zcela ji nahradila nově vzniklou službou BitBucket Pipelines, která je popsána výše.

Bamboo umožňuje propojení s verzovacími systémy Git (BitBucket, GitHub), Mercurial, Subvercion, CVS a Perforce. Sestavení, testování a nasazení se provádí pomocí úkolů – *jobs*. Úkol je možné nadefinovat pomocí sekvenčně zpracovávaných úloh – *tasks*, které spouští build definice, skripty či zpracovávají výsledky testů. Pro usnadnění práce je mnoho typových úloh předdefinováno. Práce s těmito úlohami je podobná jako v Azure DevOps. V úkolu lze pracovat pouze s artefakty, které byly vytvořeny v předchozí fázi – *stage*. Fáze se využívají k seskupování úkolů. Výchozí nastavení je však jeden úkol na jednu fázi. V případě seskupení více úkolů pod jednou fází, mohou být úkoly zpracovávány paralelně více *agenty*. Pro vetší abstrakci se fáze seskupují do tzv. plánů, u kterých se definují výchozí uložiště kódu, se kterým úkoly pracují, podmínky spuštění plánu, zasílaní zpětné vazby, uživatelské přístupy k plánům či definice proměnných. Ukázka možností členění procesu CI/CD v Bamboo je naznačena na obr. 21. Bamboo umožňuje automatické spuštění plánů na základě zaznamenaných změn v repozitáři, časových podmínek spuštění nebo v závislosti na výsledku jiných plánů.

Způsoby zasílání zpráv zpětné vazby o průběhu procesu CI/CD je obdobný jako u předchozích nástrojů CI/CD, např. zasílání emailů, zobrazování zpráv na dashboardu Bamboo prostředí, RSS zpráv či zasílání zpráv do nejrůznějších služeb jako Slack, Google Talk apod. Některé možnosti zasílání zpráv nejsou nabízeny v základních funkcionalitách nástroje Bamboo. Obdobně jako u ostatních nástrojů CI/CD je možné tyto funkcionality implementovat pomocí pluginů.



Obr. 21. Architektura členění procesu CI/CD v Bamboo. [37]

Mezi velké výhody nástroje Bamboo patří těsná vazba s ostatními produkty od společnosti Atlassian, např. BitBucket pro správu kódu či JIRA pro plánování a sledování vývoje. Je možné vytvořit komplexní nástroj pro řízení softwarového projektu. Bamboo nabídne možnost zobrazení změny kódu současně s JIRA úkoly, které slouží jako zadání pro vývojáře. Takovéto propojení umožňuje jednoduše určit, které funkcionality jsou v dané verzi softwaru a jaký je jejich stav.

Tab. 5. Základní přehled nástroje Bamboo.

Název	Bamboo (Atlassian)
Hostování	On-Premise
Platforma	Multiplatformní
Placená služba	Ano, pro open-source projekty zcela zdarma
Zkušební verze	Ano
Podpora kontejnerů	Ano
Možnosti sestavení	MSBuild, Visual Studio, Docker, Maven, Ant, Android, C, C++, CMake, Python, Node.js, Ruby, PHP, Perl
Vznik	2007

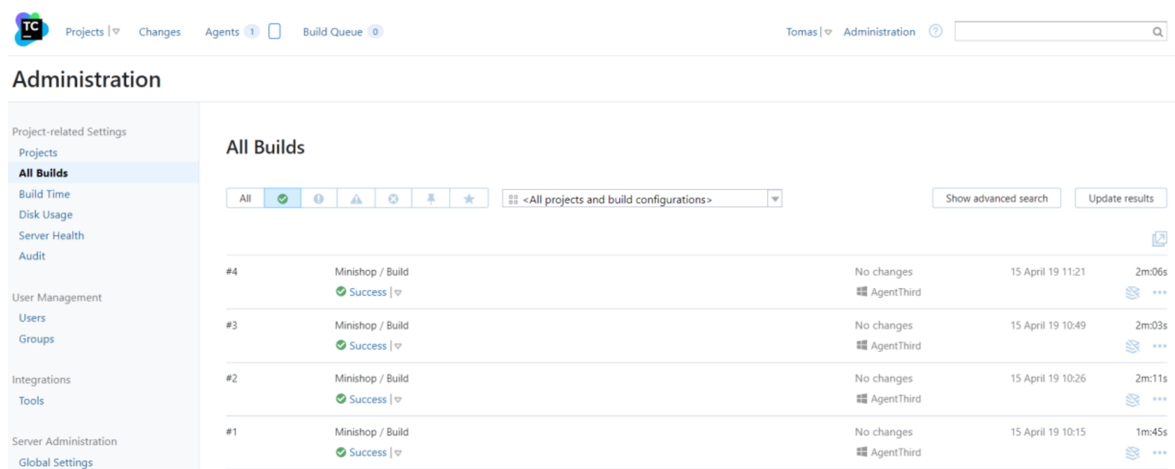
4.5 TeamCity

Nástroj TeamCity od společnosti JetBrains je dalším velkým hráčem na poli kontinuální integrace a doručování. TeamCity je spolehlivý a kvalitní multiplatformní CI/CD nástroj, který podporuje všechny nejnovější verze operačních systémů Windows, Linux a macOS.

Nástroj je určený k on-premise řešení. Jeho instalace je intuitivní a nejsou při ní vyžadovány žádné další nastavení ani přizpůsobení. Nástroj vyniká řadou unikátních funkcí jako jsou podrobné přehledy historie proběhlých procesů, okamžitá zpětná vazba o selhání testů a opakované použití nastavených definic pipelines. Díky možnosti vytváření šablon, vlastních proměnných a parametrů, TeamCity usnadňuje vytvoření flexibilní šablony pro proces CI/CD u běžných řešení, např. nasazení aplikace MVC do služby IIS. Vytvoření konfigurace pro nový typový projekt se tak stává rychlé a efektivní.

TeamCity má velkou podporu nástrojů .NET, které integruje lépe než Jenkins, Bamboo či BitBucketPipelines. TeamCity obsahuje několik nástrojů pro analýzu pokrytí kódu (dotCover, NCover nebo PartCover), .NET testovacích frameworků (NUnit, MSUnit, xUnit, MSpec nebo Gallio) či statické analýzy kódu (ReSharper).

TeamCity umožňuje získávat kód projektu pouze z repozitáře VCS na základě unikátní URL adresy. Nástroj podporuje všechny významné VCS a ty, které nejsou podporovány v základu nástroje, je možné doinstalovat pomocí pluginu. Pro rozšíření základních funkčností, TeamCity nabízí k dnešnímu dni 363 pluginů [39].



Obr. 22. Ukázka webového rozhraní nástroje TeamCity.

TeamCity poskytuje dobrou sadu vestavěných funkcí pro sestavování softwaru – podrobné informace o historii sestavení, selhání sestavení a dalších provedených změnách či řízení zdrojů. TeamCity má jednoduchý a srozumitelný mechanizmus řetězení umožňující sestavovat a nasazovat software pomocí vícenásobných řetězových závislostí. Práce při řetězení pipelines v TeamCity je obdobná jako v nástrojích Azure DevOps Services/Server.

Pro projekty, které nepodléhají žádným licencím, TeamCity nabízí cenový model v podobě bezplatné verze s plným přístupem ke všem funkcím nástroje s omezením maximálně

100 možných build definic – *jobs* a tří agentů. TeamCity pro open-source projekty je zcela zdarma a bez jakýchkoliv omezení. Podnikové licence TeamCity jsou zpoplatněné podle počtu využívaných agentů.

Tab. 6. Základní přehled nástroje TeamCity.

Název	TeamCity (JetBrains)
Hostování	On-Premise
Platforma	Multiplatformní
Placená služba	Ano, pro open-source projekty zcela zdarma
Zkušební verze	Ano
Podpora kontejnerů	Ano
Možnosti sestavení	MSBuild, Visual Studio, NAnt, Docker, Maven, Ant, Android, C, C++, CMake, Python, Node.js, Ruby, PHP, Perl
Vznik	2006

4.6 Souhrn

Azure DevOps Server a Azure DevOps Services

Azure DevOps Server a jeho cloudová podoba Azure DevOps Services je flexibilní sada nástrojů pro řízení a vývoj softwarových projektů různých velikostí. Konkrétně nástroj Azure Pipelines poskytuje výkonné a snadno použitelné služby CI/CD. Azure Pipelines umožňuje vytvářet jednoduché i komplexní CI/CD pipelines pro nasazení aplikace na širokou škálu prostředí včetně různých služeb Azure, jako jsou např. webové servery a databáze. Sestavení a nasazení pipelines je flexibilní a pomocí úloh pipeline je možné definovat téměř libovolný pracovní postup. V rámci cloudové služby Azure poskytuje předdefinované hostované agenty platforem Windows, Linux a macOS, což může ušetřit čas při nasazování procesu CI/CD. Pro nastavení pipelines nabízí Azure šablony, které napomáhají rychle nastavit proces CI/CD pro běžné scénáře.

Pro softwarové projekty postavené na Microsoft technologiích (tj. na frameworcích .NET nebo .NET Core) je nástroj Azure Pipelines, potažmo celá sada nástrojů Azure DevOps, jednoznačnou volbou jak rychle a efektivně implementovat proces CI/CD. Jedinou nevýhodu u Azure DevOps vnímám snad jen cenu, která při kombinaci nejrůznějších nástrojů a početném týmu může hrát významnou roli.

Jenkins

Jenkins je jeden z nejpoužívanějších nástrojů pro kontinuální integraci. Je to bezplatný open-source nástroj, který může ušetřit peníze při vývoji projektu. Jeho početná komunita, která vyvinula nespočet pluginů, tak dopomohla k tomu, že je Jenkins jeden z nejflexibilnějších nástrojů CI/CD a nabízí tak integraci s téměř jakýmkoli externím programem pro vývoj software. Komunita Jenkins navíc vybízí své uživatele k rozšiřování Jenkins o nové funkcionality poskytování výukových prostředků.

Hlavní slabinou nástroje Jenkins je nedostatečná dokumentace u některých pluginů, čímž se nastavení buildu stává náročnějším a déle trvajícím. Taktéž jsem v dokumentaci Jenkins např. nenalezl specifika pro softwary využívající technologii .NET. Uživatelské rozhraní nástroje se neřídí moderními principy designu. Některé ikony jsou nadměrně rozmazané a automaticky se neobnovují, když úloha skončí.

Síla nástroje Jenkins se ukáže nejvíce u velkých softwarových projektů, kde je zapotřebí nejrůznějších nastavení a úprav, které je možné provést pomocí pluginů. Množství nastavitelných možností má za následek, že implementace procesu stojí nějaký čas.

BitBucket Pipelines

Nástroj BitBucket Pipelines, který je poměrně nový mezi nástroji pro CI/CD má veliký potenciál, ale jeho současné služby jsou v porovnání s Azure Pipelines či Jenkins omezené. BitBucket Pipelines může být skvělý nástroj pro rychlé nasazení procesu CI/CD malého projektu, ale pokud je vyžadováno více pokročilých funkcí nástroje, naráží se na hranice platformy.

Velké plus si BitBucket Pipelines zaslouží za přehlednost rozhraní, zpracovanou dokumentaci k nástroji a rychlost buildů. Naopak za negativní stránky nástroje vnímám výše zmíněnou malou komplexnost. Pro projekty, které využívají technologii .NET, je zatím BitBucket Pipelines nedostačující.

Bamboo

Bamboo je flexibilní nástroj CI/CD, který podporuje široké spektrum technologií. Nástroj může být vhodnou volbou ve spojení s dalšími produkty od společnosti Atlassian (BitBucket, JIRA, Confluence), které společně tvoří komplexní nástroj pro vývoj softwaru.

Slabinou Bamboo je, že v porovnání s open-source řešením Jenkins či licencovaným Azure Pipelines má menší podporu rozšíření – pluginů. Aktuálně společnost Atlassian nabízí pro Bamboo 172 pluginů, které rozšiřují základní funkčnost [38].

TeamCity

TeamCity je nástroj umožňující pouze kontinuální integraci a kontinuální doručení. V porovnání s výše zmíněnými nástroji neumožňuje rozšíření o další nástroje pro vývoj softwaru, jako jsou např. správa verzovaného uložiště zdrojových kódů či nástroje pro řízení projektu.

TeamCity je poměrně robustní nástroj pro základní e-comerce nebo jiné menší projekty, které po počátečním spuštění nevyžadují mnoho rozvoje. Vzhledem ke své složitosti a licenční politice je vhodnější pro komplexnější projekty a větší softwarové společnosti. I přesto, že má TeamCity podrobně zpracovanou dokumentaci, jeho komplexnost při nastavování procesu CI/CD vyžaduje dostatek znalostí a času. Nástroj TeamCity je v možnostech kontinuální integrace a kontinuálního doručování konkurence schopný v porovnání s nástroji Jenkins a Azure DevOps Services/Server.

Následující tabulka Tab. 7 znázorňuje orientační pořizovací ceny nástrojů CI/CD. Ceny jsou zaokrouhleny a vychází z tabulky Tab. PI.8.

Tab. 7. Orientační ceny nástrojů CI/CD. Ceny jsou uvedeny za jeden měsíc.

Parametry	**Případ 1.**	**Případ 2.**	**Případ 3.**
Průměrná doba buildu (min)	5	5	15
Počet buildů za den	10	10	50
Počet dnů	20	20	20
Počet uživatelů	5	15	40
Počet agentů	1	1	3
Název nástroje CI/CD	**Cena (Kč/měsíc)**	**Cena (Kč/měsíc)**	**Cena (Kč/měsíc)**
TeamCity	4 400 Kč	4 400 Kč	4 400 Kč
Bamboo	20 Kč	20 Kč	6 400 Kč
BitBucket Pipelines	600 Kč	1 800 Kč	25 000 Kč
Azure Pipelines	0 Kč	800 Kč	2 400 Kč
Azure DevOps Server	900 Kč	2 700 Kč	8 900 Kč

II. PRAKTICKÁ ČÁST

5 VZOROVÁ APLIKACE MINISHOP

Pro demonstraci procesů kontinuální integrace a kontinuálního doručování je součástí této práce i samotná implementace vzorové aplikace. Vzorová aplikace ve zjednodušené formě reflektuje určité praktiky a nástroje používané při vývoji reálných webových aplikací. Aplikace je primárně určena pro demonstraci nasazení procesu CI/CD za pomocí různých nástrojů. Funkční a obsahová stránka aplikace je tak zjednodušena pouze pro demonstraci nasazení procesu CI/CD.

Účel vzorové aplikace je zjednodušený internetový obchod pro interní nákup potravin ve společnosti. Tato webová aplikace je dále označována názvem MiniShop. Zaměstnanci společnosti, neboli uživatelé aplikace MiniShop, mají možnost se do systému přihlásit. Po přihlášení je uživateli zobrazen celý sortiment produktů, který je v aplikaci nabízen. Uživatel má možnost zakoupit vybrané produkty. Po zakoupení je automaticky vytvořena objednávka na zakoupený produkt. Pro zjednodušení je možné najednou zakoupit pouze jeden množstevní kus produktu, tzn. objednávka obsahuje vždy pouze jeden produkt. Pro zobrazení zakoupených produktů má uživatel možnost zobrazit si seznam všech provedených objednávek, tj. zakoupených produktů.

5.1 Požadavky na aplikaci

Aplikaci MiniShop je možné popsat následujícími požadavky. Požadavky se dělí na funkční, tj. specifika, co by měla aplikace umět, a nefunkční požadavky definující jaká by aplikace měla být.

5.1.1 Funkční požadavky

Uživatelé

- přihlášení a odhlášení z aplikace,
- zobrazení sortimentu zboží,
- zobrazení objednávek uživatele,
- zakoupení zboží.

Zboží

- vedení seznamu zboží

Objednávky

- vedení seznamu objednávek

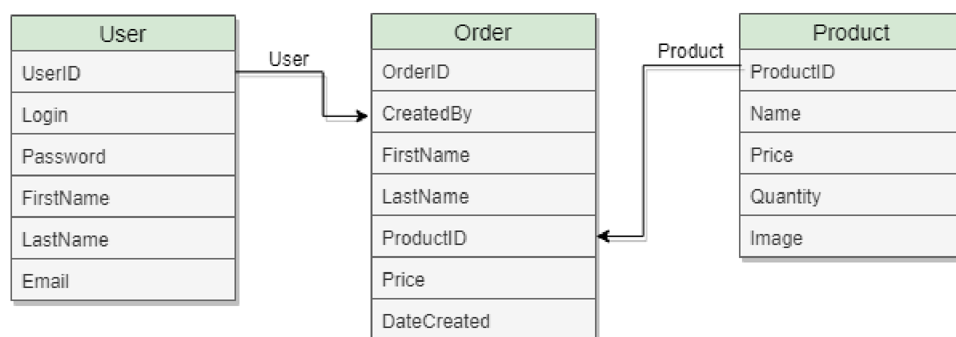
5.1.2 Nefunkční požadavky

- vícevrstvý systém obsahující minimálně datovou, logickou a prezentační vrstvu,
- využití relační databáze pro datovou vrstvu,
- možnost dostupnosti logické vrstvy třetí stranou,
- využití vhodného frameworku pro prezentační vrstvu,
- využití vhodného frameworku pro objektově-relační mapování.

5.1.3 Datová specifika

Z výše popsaných funkčních požadavků vyplývají tři základní entity, které charakterizují aplikaci po datové stránce. Jedná se o entity uživatele, produktu a objednávky. Každá z těchto entit má charakteristické atributy, které jsou naznačeny na obr. 23.

Za zmínku stojí entita objednávek (*Order*), u které se může zdát, že jsou atributy duplicitní a nadbytečné v porovnání s ostatními entitami. Objednávka však představuje produkt zakoupený uživatelem v daném čase. Pokud by např. došlo ke změně ceny produktu, na již vytvořených objednávkách se cena za zakoupený produkt nesmí změnit. To je důvodem proč entita objednávek nese dané informace a není možné je získat provázáním z ostatních entit, tj. uživatelů a produktů.



Obr. 23. Class diagram charakterizující datovou vrstvu aplikace MiniShop.

5.2 Systémový návrh

Webová aplikace MiniShop bude jako celek tvořena dvěma dílčími samostatnými aplikacemi. Tou první je aplikace MiniShopWebApi, která zastřešuje aplikační logiku a propojení datové vrstvy s relační databází. Aplikace je vytvořená v jazyce C# za použití univerzální open-source vývojové platformy *.NET Core* spravované společností Microsoft.

Aplikace MiniShopWebApi komunikuje s relační databází MS SQL, která je hostována v rámci cloudové služby Azure. Komunikační můstek mezi objektově orientovanou aplikací a relační databází je tvořen technologií Entity Framework (EF) Core.

EF umožňuje mapování objektů aplikace na relační data databáze (ORM) a dovoluje přistupovat, dotazovat a upravovat data z tabulek databáze pomocí silně typových objektů a syntaxe LINQ. EF odstiňuje od jazyka SQL a umožňuje pracovat s daty, jako by se jednalo o kolekce .NET.

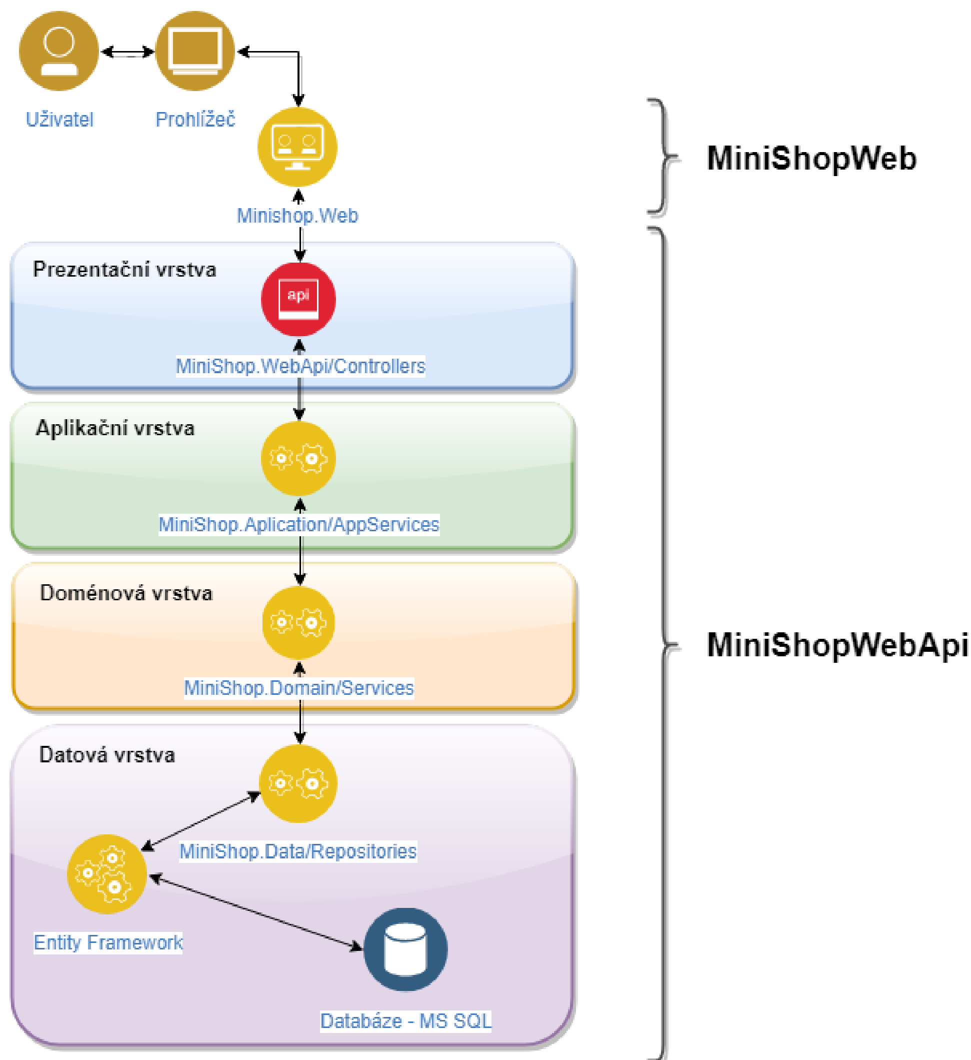
Další knihovnou třetích stran, která je v aplikaci MiniShopWebApi použita, je Autofac. Tato knihovna slouží pro vkládání závislostí, tzv. dependency injection. Spravuje závislosti mezi třídami tak, aby aplikace zůstala jednoduchá a snadno rozšiřitelná.

Druhou dílčí aplikací je MiniShopWeb. Tato aplikace je vytvořena pomocí javascriptového frameworku Angular, který je určen pro vytváření mobilních a desktopových webových aplikací. Dílčí aplikace MiniShopWeb tvoří zcela prezentační (frontendovou) vrstvu aplikace MiniShop. Aplikace je postavena na architektuře komponent se službami a je psána jazyce TypeScript.

5.3 Implementace

Dle popisu nefunkčních požadavků v kapitole 5.1.2, je aplikace MiniShop složena z několika vrstev, viz diagram na obr. 24.

Frontendová aplikace MiniShopWeb architekturu vrstev jako takovou nemá. Je složena z komponent, které se zaměřují na prezentování dat a delegování datového přístupu mezi službami. Služby namísto toho umožňují sdílet informace mezi jednotlivými komponentami, které se navzájem neznají. Komponentou v projektu je např. *Order-list.* Tato komponenta zajišťuje vykreslení produktů na stránce a volá servisu *DataService* pro získání produktů, které budou zobrazeny. Obecně jsou data získávána na základě HTTP dotazu na odpovídající přístupový bod API aplikace MiniShopWebApi v prezentační vrstvě.



Obr. 24. Vrstvy architektury aplikace MiniShop.

Prezentační vrstva aplikace MiniShopWebApi slouží pro interakci s aplikací MiniShopWeb. Tato vrstva obsahuje přístupové body, na které se aplikace MiniShopWeb dotazuje. Tyto HTTP požadavky jsou dále v rámci vrstev aplikace zpracovávány a výsledek je opětovně vracen v podobě HTTP odpovědi. Vrstva obsahuje pouze řadiče, které shlukují tématické přístupové body (endpoints) k API.

Aplikační vrstva slouží k tomu, aby prezentační vrstva byla co nejštíhlejší a nebyla v ní žádná logika. Aplikační vrstva řeší jednotlivé požadavky uživatele na systém. Logika vrstvy pouze určuje, co se má dělat, nikoli jak se má dělat. To znamená, že *AppServices* v této vrstvě volají jednotlivé funkcionality nižších vrstev a skládají dohromady výsledný požadavek uživatele. Kromě zmíněné logiky vrstva obsahuje definice vkládání závislostí či datové modely, které vracejí jednotlivé přístupové body.

Doménová vrstva obsahuje veškerou business logiku aplikace a stává se tak nejdůležitější vrstvou celého modelu. Doménová vrstva obsahuje třídy, které odpovídají modelové realitě, tj. User, Order, Product. Dále services, které zprostředkovávají komunikaci mezi třídami a obsahují business logiku. V neposlední řadě obsahují rozhraní pro přístup k datům.

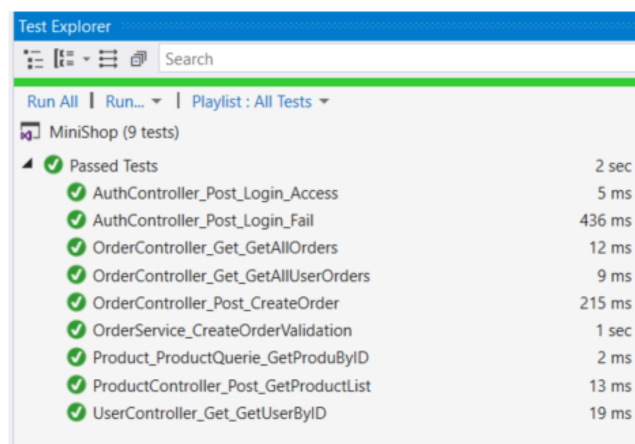
Datová vrstva zprostředkovává přístup k datům v databázi. Její součástí jsou dílčí dotazy do databáze, tzv. queries. Jelikož se tato vrstva zabývá přístupem, je její součástí konfigurace EF, přesněji konfigurace doménových tříd a definice relačního mapování tříd.

V neposlední řadě je součástí aplikace MiniShopWebApi databázový projekt, který obsahuje definice všech entit (tabulek) databáze a umožňuje je spravovat. Pomocí database projektu je možné definice entit verzovat.

5.4 Automatizované testy

Dle popisu v kapitole 3 by mělo být automatizované testování neodmyslitelnou částí vývoje softwaru. Součástí aplikace MiniShop je sada automatizovaných testů, která obnáší dva typy testů, a to jednotkové a integrační testy. Jednotkové testy prověřují správnou funkčnost převážně doménové vrstvy a konkrétní funkcionality. Integrační testy prověřují kompletní funkčnost přístupových bodů řadičů.

Jednotlivé testy jsou umístěné v odděleném projektu *MiniShop.WebApi.Tests*. Oba typy testů jsou vytvořeny pomocí specializovaných knihoven pro testování. Projekt MiniShop využívá Microsoft MSTest Framework.



Obr. 25. Přehled testů v projektu MiniShop.

Automatizované testy tvoří testovací sadu, která pokrývá datové a funkční požadavky aplikace. Cílem dílčích testů je prověřit následující scénáře:

- Test *AuthController_Post_Login_Access* – Ověřuje, že při zadání platných přihlašovacích údajů uživatele, je příslušným přístupovým bodem vrácena validní hodnota, na základě které je možné přistupovat do aplikace.
- Test *AuthController_Post_Login_Fail* – Ověřuje opačný případ předchozího testu *AuthController_Post_Login_Access*. Při zadání neplatných přihlašovacích údajů je ověřováno, že je odepřen přístup uživatel do aplikace.
- Test *OrderController_Get_GetAllUserOrders* – Ověřuje, zda přístupový bod vrací všechny objednávky pro vybraného uživatele a zda jsou z API vraceny všechny náležitosti objednávky.
- Test *OrderController_Get_GetAllOrders* – Ověřuje, zda přístupový bod vrací objednávky všech uživatelů, které jsou uložené v relační databázi.
- Test *OrderController_Post_CreateOrder* – Ověřuje celý proces vytvoření objednávky, tj. zda se na základě vstupních parametrů identifikátoru uživatele a produktu správně vyplní údaje objednávky a nastane úspěšné uložení objednávky do relační databáze. Současně se při úspěšném vytvoření objednávky ověřuje snížení dostupného množství objednaného produktu.
- Test *OrderService_CreateOrderValidation* – Ověřuje dílčí část procesu vytvoření nové objednávky, která se týká validačních pravidel. Testuje se, jestli dochází k znemožnění vytvoření objednávky, tj. vytvoření nového záznamu v relační databázi, při nesplnění předem určených kritérií, např. vytvoření objednávky na produkt, které má nulové dostupné množství.
- Test *ProductController_Post_GetProductList* – Ověřuje, zda jsou odpovědným přístupovým bodem API vráceny všechny dostupné produkty, které jsou uloženy v relační databázi a informace o produktu jsou úplné.
- Test *Product_ProductQuerie_GetProduByID* – Ověřuje, zda dotaz vytvářený do relační databáze vrací očekávaný produkt se všemi náležícími informacemi produktu.
- Test *UserController_Get_GetUserByID* – Na základě identifikátoru uživatele test ověřuje, zda jsou příslušným přístupovým bodem vráceny všechny očekávané informace uživatele evidované v relační databázi.

5.4.1 Jednotkové testy

Ukázka použití jednotkového testu v aplikaci MiniShop bude demonstrována na zjednodušené metodě *CreateOrderValidation*, která zapouzdřuje validační podmínky pro znemožnění vytvoření nevalidní objednávky. Tato metoda je volána před vytvořením samotné objednávky v databázi. Ukázka metody je na obr. 26.

```
1 public void CreateOrderValidation(Product product)
2 {
3     if (product.Quantity <= 0)
4         throw new Exception("Product not in stock");
5 }
```

Obr. 26. Metoda CreateOrderValidation v projektu MiniShop.Domain.Services.Orders

Jednotkový test je ukázán na obr. 27. Při použití nástroje MSTest je zapotřebí každou třídu, která slouží pro testování, označit atributem *TestClass* a metody *TestMethod*. Jednotkový test ověřuje, že nelze zakoupit produkt, pokud má nulové množství. Prvním krokem při testování je připravení dat pro testování. V tomto případě je vytvořena instance třídy (entita) *Produkt* a naplněna testovacími hodnotami.

Pro vykonání testu je zapotřebí vyvolat správnou metodu, která má být testována. V tomto případě je požadováno vykonat metodu v doménové vrstvě. V ukázkovém testu je využito vkládání závislostí (dependency injection). Je vytvořena nová instance třídy *DependencyResolver()*, která ve svém konstruktoru vytvoří identický kontejner závislostí (instancí objektů), jaký je vytvořen při spuštění samotné aplikace MiniShop. Následně stačí pouze získat zaregistrovanou instanci z kontejneru pomocí metody *Resolve<T>*. Problematika vkládání závislostí přesahuje rozsah této práce. Je zde popsán pouze základní princip, více informací je možné získat z dokumentace použité knihovny Autofac [34].

Poté, co je získána instance třídy *OrderService*, je možné vyvolat požadovanou metodu s předchystanými testovacími daty. Cílem testu je nasimulovat případ, kdy objednávaný produkt není na skladě a testovaná metoda by měla vyvolat odpovídající výjimku o nedostupnosti produktu. Samotné ověření, zda došlo k vyvolání výjimky, se provádí pomocí metod třídy *Assert* určené pro vyhodnocení jednotkových testů.

```
1  [TestMethod]
2  public void OrderService_CreateOrderValidation()
3  {
4      //Arrange
5      var testProduct = new Product()
6      {
7          ProductID = 0,
8          Name = "TestProduct",
9          Price = 99,
10         Quantity = 0
11     };
12
13     //Act
14     using (var resolver = new DependencyResolver())
15     {
16         var productService = resolver.Resolve<IOrderService>();
17
18         //Assert
19         var ex = Assert.ThrowsException<Exception>(() => productService.CreateOrderValidation(testProduct));
20         Assert.AreEqual("Product not in stock", ex.Message);
21     }
22 }
```

Obr. 27. Jednotkový test ověřující podmínky pro vytvoření objednávky produktu.

5.4.2 Integrační testy

Integrační testy oproti jednotkovým testují komplexnější bloky softwaru, a to především celé moduly. V aplikaci MiniShop byly integrační testy použity k testování funkčnosti přístupových bodů API. Tento typ testů ověřuje, že dochází ke správné komunikaci mezi všemi vrstvami aplikace, dochází ke správnému propojení s relační databází a v neposlední řadě, že jsou vraceny všechny požadované informace. Ukázka implantace a integračního testu bude demonstrována na přístupovém bobu API pro vytvoření nové objednávky, viz obr. 28.

Obdobně jako u jednotkových testů je prvním krokem příprava testovacích dat. Na základě znalosti dat v databázi je nejprve určen identifikátor uživatele – *userID* a produktu – *productID*, se kterými bude v testu pracováno. Následně je obdobným principem jako u jednotkových testů získána instance třída *DataContext*. Díky EF je možné pomocí této třídy vytvářet dotazy do relační databáze. Dalším krokem je získat z databáze entitu produktu podle dříve zvoleného identifikátoru produktu – *productPre*. Po získání této entity je vytvořen nový objekt objednávky – *order*, charakterizující vytvořenou objednávku ve webové aplikaci.

Po přípravě všeho potřebného test následuje zavolání přístupového body API pro vytvoření objednávky. Toto volání je zprostředkováno pomocí test serveru a příslušné metody volání. Test server je vytvořen právě pro účely integračního testování a jeho definice vytvoření je součástí projektu *MiniShop.Testing*. Parametrem volaného přístupového bodu je připravená třída objednávky – *order*. Návratová hodnota po úspěšném vytvoření

objednávky je identifikátor nové objednávky. Tento identifikátor je uložen do privátní proměnné *_orderID* testovací třídy pro účely úklidu po testu, viz dále.

Aby bylo možné ověřit správné vytvoření nové objednávky, je tato objednávka – *order* získána z databáze opět pomocí třídy *DataContext* a identifikátoru nové objednávky. Identickým způsobem je získána z databáze entita produktu, pro který byla vytvářena objednávka – *productPost*.

Následuje ověření pomocí metod třídy *Assert*, zda nově vytvořená objednávka má očekávané náležitosti a jestli množství produktu bylo po vytvoření objednávky sníženo.

```
1  [TestMethod]
2  public void OrderController_Post_CreateOrder()
3  {
4      //Arrange
5      var productPre = new Product();
6      var productPost = new Product();
7      var order = new Order();
8      var userID = 1;
9      var productID = 4;
10
11     using (var resolver = new DependencyResolver())
12     {
13         var context = resolver.GetContext();
14         productPre = context.Product.FirstOrDefault(x => x.ProductID == productID);
15     }
16
17     var newOrder = new Order()
18     {
19         CreatedBy = userID,
20         ProductID = productID
21     };
22
23     // Act
24     _orderID = _testServer.Post<int>("api/order/CreatOrder", newOrder);
25
26     using (var resolver = new DependencyResolver())
27     {
28         var context = resolver.GetContext();
29         order = context.Order.FirstOrDefault(x => x.OrderID == _orderID);
30         productPost = context.Product.FirstOrDefault(x => x.ProductID == productID);
31     }
32
33     //Assert
34     Assert.IsNotNull(productPre);
35     Assert.IsNotNull(order);
36     Assert.AreEqual(productPre.Name, order.Name);
37     Assert.AreEqual(productPre.Price, order.Price);
38     Assert.AreEqual(productPre.ProductID, order.ProductID);
39     Assert.AreEqual(userID, order.CreatedBy);
40     Assert.AreEqual(productPre.Quantity - 1, productPost.Quantity);
41 }
```

Obr. 28. Integrační test ověřující úspěšné vytvoření objednávky.

Jelikož tento integrační test neprověřuje pouze získání dat z databáze, ale dochází při vytváření objednávky ke změnám samotných dat v databázi, je posledním krokem testu

uvedení změněných dat v databázi do původního stavu. Toto tzv. vyčištění po testu se provádí pro zajištění integrity dat a pro nedůležitost pořadí spouštěných testů. Pokud by se data po dokončení testu nevrátila do původního stavu, spouštěné testy by se navzájem ovlivňovaly a nebylo by tak možné jednoznačně určit očekávaná data pro vyhodnocení testu.

Toto vyčištění po každém testu je možné jednoduše provést označením metody atributem *TestCleanup()*. Tento atribut zaručí, že metoda v příslušné testovací třídě bude vykonávána po každém testu, viz obr. 29. V ukázkovém kódu je naznačeno, že pokud došlo k vytvoření objednávky, je tato objednávka z relační databáze smazána a množství produktu, pro který byla objednávka vytvořena, navýšeno.

```
1  [TestCleanup()]
2  public void Cleanup()
3  {
4      if (_orderID.HasValue)
5      {
6          using (var resolver = new DependencyResolver())
7          {
8              var context = resolver.GetContext();
9              var order = context.Order.FirstOrDefault(x => x.OrderID == _orderID);
10             var product = context.Product.FirstOrDefault(x => x.ProductID== order.ProductID);
11
12             product.Quantity++;
13
14             context.Update(product);
15             context.Remove(order);
16             context.SaveChanges();
17         }
18         _orderID = null;
19     }
20 }
```

Obr. 29. Metoda pro navrácení dat do původního stavu před spuštěním testu testovací třídy OrderControllerTest.

6 IMPLEMENTACE CI/CD

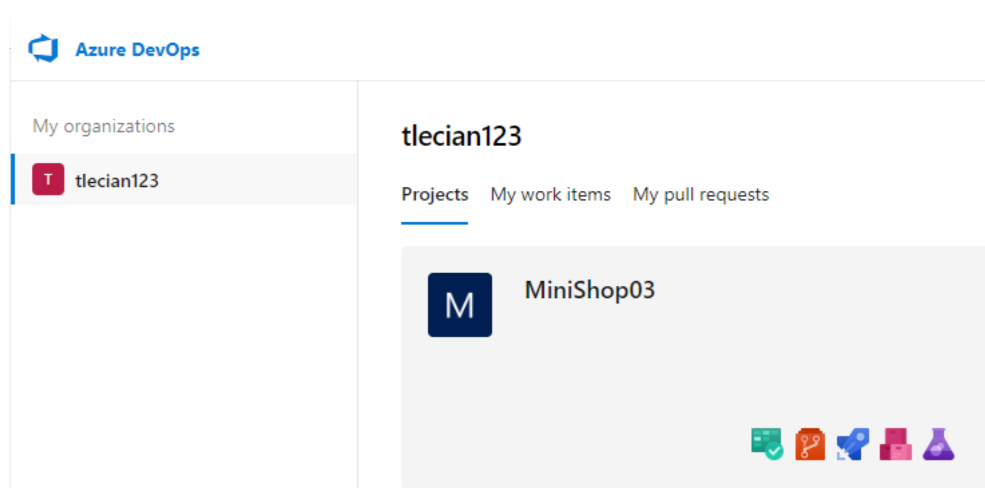
Ukázka nasazení kontinuální integrace a kontinuálního doručení bude provedena na aplikaci MiniShop popsané v kapitole *5. Aplikace MiniShop*. Pro integraci aplikace jsme zvolili nástroje Azure DevOps, jakožto technologii společnosti Microsoft.

V této kapitole jsou popsány kroky vedoucí k úspěšnému nasazení procesu CI/CD na projektu. Jsou zde specifikovány dílčí aplikace, tj. API a webové aplikace, a nástrahy, které lze očekávat při implementaci procesu integrace pro vybrané technologie.

6.1 DevOps projekt

Pokud chceme pro kontinuální integraci a doručení využívat služby Azure DevOps [31], je nezbytné mít vytvořený účet na portále Microsoft Azure [30]. Tento portál je startovacím místem pro využívání široké nabídky cloudových služeb Azure.

Na portále Microsoft Azure je zapotřebí vytvořit si organizaci a zvolit cenovou politiku pro spravování cloudové služby. Po vytvoření organizace vedou následné kroky k vytvoření tzv. DevOps projektu pod danou organizací. V rámci takového projektu je možné plně využívat možnosti služby Azure DevOps.

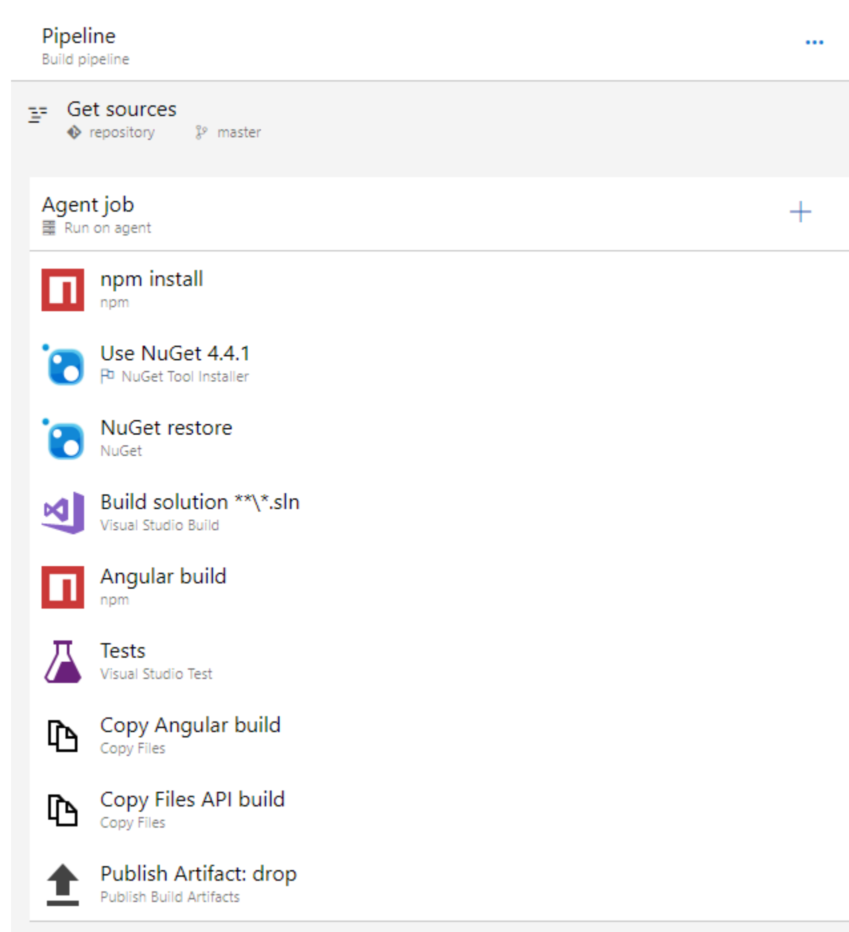


Obr. 30. Ukázka projektu MiniShop03 zařazený pod organizací tlecian123 v cloudové službě Azure DevOps.

6.2 Nastavení kontinuální integrace

Abychom docílili automaticky vytvářeného buildu aplikace, je nutné vytvořit a správně nadefinovat dílčí kroky build pipeline. U každé aplikace jsou kroky build pipeline závislé na

konkrétní vyvíjené aplikaci. Pro ukázkovou aplikaci MiniShop jsou kroky naznačeny na obr. 31. Jednotlivé kroky a způsob jejich nastavení je popsán dále v této kapitole.



Obr. 31. Dílčí kroky (úlohy) celé build pipeline.

6.2.1 Agent

První krokem v nastavení pipeline je definování a*gent pool*. V rámci agent poolu jsou definováni a spravováni dílčí agenti. Na agenta můžeme nahlížet jako na software, ve které m se prování definice jednotlivých specifikovaných pipelines. Agent běží na CI serveru a je možné jej spouštět jako virtuální stroj či přímo v kontejneru.

Azure DevOps umožňuje vytvořit si vlastního agenta (agent pool) a plně jej spravovat nebo využít tzv. default agent pools, které mají předdefinované agenty a jejich vlastnosti. Je možné vybírat mezi agenty, kteří jsou uzpůsobeni pro vytváření buildu na operačním systému Linux, Mac OS apod.

Pro účely práce jsme využili defaultní agent pool *Hosted VS2017*. Agent se spouští na virtuálním stroji s operačním systémem Windows server 2016 a je na něm nainstalováno Microsoft Visual Studio 2017.

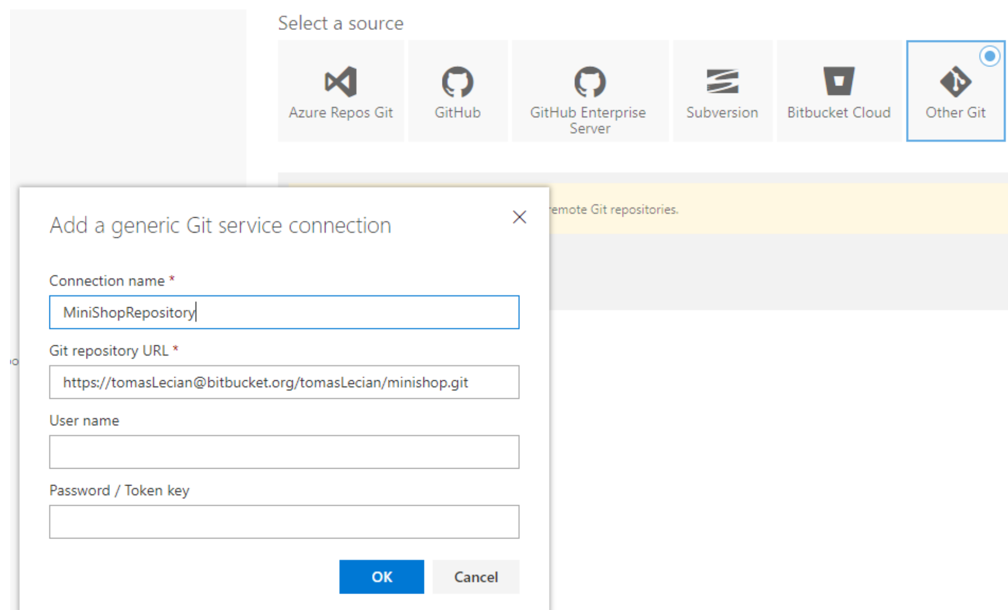
```
pool:
  name: Hosted VS2017
  demands:
  - npm
  - msbuild
  - visualstudio
  - vstest
```

Obr. 32. Ukázka nastavení Agent pool.

6.2.2 Repozitář kódu aplikace

Neodmyslitelnou součástí aplikace je repozitář centrálního úložiště kódu. Pro verzování kódu aplikace jsem využil nástroje BitBucket, který umožňuje privátní repozitáře. Nástroj BitBucket je nástavbou nad verzovacím systémem Git. Pro propojení projektu Azure DevOps s repozitáře nám proto stačí definovat URL daného repozitáře a přihlašovací jméno s heslem.

Na závěr je nutné určit větev repozitáře, z které bude kopírován kód aplikace do CI serveru a následně sestavován build.



Obr. 33. Ukázka propojení projektu DevOps s repozitářem nástroje BitBucket.

6.2.3 Pipeline job

Pipeline job definuje dílčí kroky kontinuální integrace, tzv. úlohy. Job je spouštěn prostřednictvím agenta, kterého můžeme v tomto kroku definovat. Při zpracování projektu diplomové práce nastavujeme výběr agenta podle agenta definovaného nad celou pipelinou, tj. mém případě *Hosted VS2017*.

Dalšími možnostmi tohoto kroku je definice exekučního plánu či nastavení, zda mohou skripty spouštěné v rámci jobu přistupovat na jiné služby přes OAuth token.

Obr. 34. Ukázka nastavení agent jobu v DevOps pro build pipeline.

Jelikož aplikaci MiniShop jako celek tvoří dvě dílčí aplikace – API a webová aplikace, budou v rámci pipeline jobu definovány buildy obou těchto dílčích aplikací.

6.2.3.1 Npm balíčky

Prvním opravdovým krokem pro vytvoření funkčního buildu je nainstalování Npm balíčků. Tyto balíčky jsou využívány technologií Angular a jsou potřebné pro vytvoření buildu webové části aplikace MiniShop.

Tento krok je možné provést pomocí *Npm* úlohy, kterou DevOps nabízí. Důležitým parametrem této úlohy je příkaz *install*, který se definuje, že se jedná o instalaci balíčků. Následně probíhá definice souborové cesty k adresáři aplikace, ve kterém se nachází soubor

packeage.json nezbytný pro instalaci balíčků. Tento konfigurační soubor se nachází v hlavním adresáři webové aplikace.

```
variables:
  NameSpaceWeb: '$(ProjectName).Web'

steps:
- task: Npm@1
  displayName: 'npm install'
  inputs:
    workingDir: '$(Build.SourcesDirectory)\src\$(NameSpaceWeb)'
    verbose: false
```

Obr. 35. Definice úlohy pro instalaci Npm balíčků.

6.2.3.2 NuGet balíčky

Zatímco Npm balíčky představovaly nezbytné knihovny pro webovou aplikaci, NuGet balíčky se využívají v technologii Microsoft .Net Core jakožto externí knihovny. Tyto balíčky jsou využívány pro API aplikace MiniShop, a tak je nezbytné mít tyto balíčky nainstalovány při buildu aplikace.

Úloha *NuGet Tool Installer* používané balíčky v rámci projektu stáhne a uloží na CI server. Jeho jediným parametrem je verze balíčků, která se má nainstalovat. Pro definici verze balíčků je možné použít některé matematické operandy či znak *x* pro reprezentaci libovolného čísla např. 4.x, 3.3.x, 2.8.6, >=4.0.0-0.

```
steps:
- task: NuGetToolInstaller@0
  displayName: 'Use NuGet 4.4.1'
  inputs:
    versionSpec: 4.4.1
```

Obr. 36. Definice úlohy NuGet Tool Installer pro instalaci NuGet balíčků.

6.2.3.3 Obnovení NuGet balíčků

V předchozí úloze se NuGet balíčky pouze stáhly a uložili na CI server. Dalším krokem je instalace chybějících či aktualizace zastaralých balíčků. Úloha *Nuget* s parametrem *restore* umožní obnovení všech potřebných balíčků.

```
steps:
- task: NuGetCommand@2
  displayName: 'NuGet restore'
  inputs:
    restoreSolution: '$(Parameters.solution)'
```

Obr. 37. Definice úlohy Nuget pro obnovení NuGet balíčků.

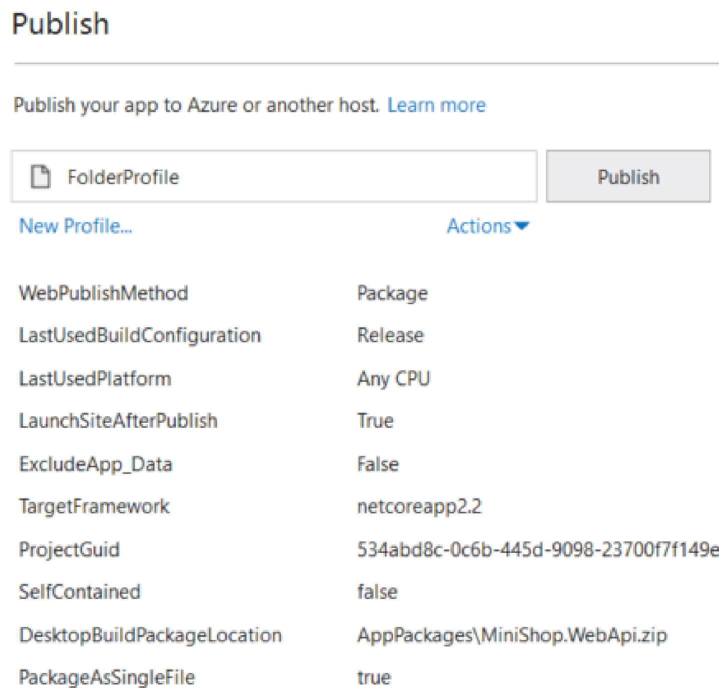
6.2.3.4 Build API aplikace

Po nezbytných přípravách v podobě stažení a obnovení balíčků je dalším krokem samotná definice buildu API aplikace. Technologie Microsoft .Net Core využívá pro kompilaci kódu buildovací nástroj MSBuild. Aby bylo možné build vytvořit, je nejprve zapotřebí nějaký předpis říkající co a jak se má kompilovat, tzv. build script. U technologie Microsoft .Net Core je build skriptem soubor označený příponou *.sln*, který je vytvářen automaticky IDE Visual Studio. Obdobným způsobem, jako lze na základě tohoto build skriptu vytvořit build samotné aplikace přímo v IDE na počítači, můžeme pomocí tohoto skriptu vytvořit také build prostřednictvím kontinuální integrace na CI serveru.

Azure DevOps nabízí úlohu *Visual Studio Build*, která se používá právě pro vytvoření buildu pomocí MSBuild nástroje. Prvním volitelným parametrem úlohy je verze Visual Studia, která se má pro využití buildu použít. V mém případě se nabízí verze IDE Visual Studio 2017, v rámci kterého jsem API aplikace vytvářel a je součástí agenta, který bude zpracovávat tento job.

Dále je v úloze zapotřebí definovat některé argumenty MSBuild skriptu. Prvním argumentem je argument *DeployOnBuild*, který je třeba nastavit na hodnotu *true*, čímž říkáme MSBuilderu, že tento webový projekt musí být zabalen/nasazen jako součást buildu.

Dalším argumentem je *PublishProfile*, kterým definujeme cestu ke konfiguračnímu souboru s příponou *.pubxml*. Je zapotřebí manuálně nadefinovat p*ublish* projektu v IDE a konfigurační soubor zaverzovat s ostatními soubory projektu. Souborová cesta k tomuto konfiguračnímu souboru je hodnotou zmíněného argumentu *PublishProfile*. Ukázka nastavení konfiguračního souboru je na obr. 38.



Obr. 38. Konfigurační soubor FolderProfile.pubxml projektu MiniShop.WebApi aplikace MiniShop.

Dalšími parametry úlohy u *Visual Studio Build*, které je zapotřebí nastavit, jsou *platforma*, pro kterou se má build vytvořit, a typ konfigurace – viz *Obr. 39*.

```
variables:
  NameSpaceWebApi: '$(ProjectName).WebApi'
  BuildPlatform: 'any cpu'
  BuildConfiguration: 'release'

steps:
- task: VSBuild@1
  displayName: 'Build solution **\*.sln'
  inputs:
    solution: '$(Parameters.solution)'
    vsVersion: 15.0
    msbuildArgs: '/p:DeployOnBuild=True
/p:PublishProfile=$(Build.SourcesDirectory)\src\$(NameSpaceWebApi)\Properties\
PublishProfiles\FolderProfile.pubxml '
    platform: '$(BuildPlatform)'
    configuration: '$(BuildConfiguration)'
    logProjectEvents: false
```

Obr. 39. Definice úlohy Visual Studio Build pro nastavení parametrů MSBuildu.

6.2.3.5 Build Webové aplikace

Pro vytvoření buildu webové aplikace postavené na technologii Angular se využijí dříve nainstalované Npm balíčky. Pro definici skriptu je možné opět využít *Npm* úlohu. V tomto případě však nebude parametr příkazu *install*, ale *custom*, který umožňuje definovat vlastní

podobu příkazů. Build Angular aplikace je možné provést pomocí příkazu *ng build* s parametrem *prod* definujícím produkční build. Tento příkaz je možné spustit lokálně pouze v rámci *Angular CLI* jobu pomocí příkazu *npn run*. Celá podoba příkazu je na obr. 40.

Příkaz *ng build* pracuje s konfiguračním souborem *package.json*. Proto dalším parametrem tohoto úlohy je souborová cesta k tomuto souboru. Soubor je uložen v hlavním adresáři webové aplikace.

```
variables:
  NameSpaceWeb: '$(ProjectName).Web'

steps:
- task: Npm@1
  displayName: 'Angular build'
  inputs:
    command: custom
    workingDir: '$(Build.SourcesDirectory)\src\$(NameSpaceWeb)'
    verbose: false
    customCommand: 'npm run ng build -- prod'
```

Obr. 40. Definice úlohy Npm pro vytvoření buildu Angular aplikace.

6.2.3.6 Automatizované testy

Pro otestování aplikace MiniShop jsme zvolil jednotkové a integrační testy. Testy jsou vytvořeny pomocí knihoven MSTest. V DevOps je možné tyto typy testů spustit pomocí úlohy *Visual Studio Test*. Je zapotřebí definovat jakým způsobem se budou testy spouštět. Zvolili jsme variantu spouštění testů pomocí vytvořených dll souborů – *test assembly*.

Tato varianta obnáší nadefinování konkrétních dll souborů testů, které se mají spouštět z pracovního adresáře agentu, viz obr. 41. Pro integrační testy, obsahující spojení s databází, je nutné mít před jejich spuštěním vystavenou oddělenou databázi, která je určena pouze pro testování. Tento krok předchází spuštění automatizovaných testů. Nastavení oddělené databáze je popsáno v kapitole 6.3.1.1

```
variables:
  ProjectName: 'MiniShop'

steps:
- task: VSTest@2
  displayName: Tests
  inputs:
    testAssemblyVer2:
     tests\**\$(BuildConfiguration)\**\$(ProjectName)*test*.dll
     !**\obj\**
     !**\MiniShop.Testing.dll
     !**\MiniShop.WebApi.dll
    vsTestVersion: 15.0
    platform: '$(BuildPlatform)'
    configuration: '$(BuildConfiguration)'
```

Obr. 41. Definice úlohy Visual Studio Test pro automatizované spuštění jednotkových a integračních tesů.

6.2.3.7 Uložení artefaktů

Během buildů dílčích aplikací se vytvořily v pracovním adresáři agentu spustitelné soubory, které tvoří aplikace MiniShop jako celek. S těmito soubory se pracuje během procesu kontinuálního nasazení, který je však vykonáván jiným jobem a agentem. Abychom o spustitelné soubory, tzv. artefakty, po ukončení jobu nepřišli, musí být tyto soubory překopírovány do nějakého úložiště.

Před uložením artefaktů na CI server se všechny artefakty nakopírují do systémového adresáře, odkud budou dále kopírovány jako balík souborů. Jedná se o obecnou praktiku při použití DevOps. K adresáři s artefakty je možné přistupovat pomocí systémové proměnné *build.artifactstagingdirectory.*

Ke kopírování souborů je určená úloha *Copy Files*, u kterého se definuje zdrojový adresář, tj. pracovní adresář agenta. Cílový adresář je systémový adresář agenta s artefakty a soubory (artefakty), které se mají kopírovat. Kopírování artefaktů probíhá ve krocích – pro API a webovou aplikaci odděleně. Detailní definice kopírování artefaktů je na obr. 42 a obr. 43.

```
variables:
  NameSpaceWeb: '$(ProjectName).Web'

steps:
- task: CopyFiles@2
  displayName: 'Copy Angular build'
  inputs:
    SourceFolder: '$(Build.SourcesDirectory)'
    Contents: '$(Build.SourcesDirectory)\src\$(NameSpaceWeb)\dist\**'
    TargetFolder: '$(build.artifactstagingdirectory)\Web'
```

Obr. 42. Definice úlohy Copy Files pro uložení artefaktů webové aplikace.

```
variables:
  NameSpaceWebApi: '$(ProjectName).WebApi'

steps:
- task: CopyFiles@2
  displayName: 'Copy Files API build'
  inputs:
    SourceFolder: '$(Build.SourcesDirectory)'
    Contents: |
     **\AppPackages\*.zip
     **\AppPackages\*.cmd
     **\AppPackages\*.xml
     **\*.dacpac
     **\Release\*.publish.xml
     **\$(NameSpaceWebApi)\appsettings.json
    TargetFolder: '$(build.artifactstagingdirectory)\WebApi'
```

Obr. 43. Definice úlohy Copy Files pro uložení artefaktů API aplikace.

6.2.3.8 Publikování artefaktů

Závěrečným krokem kontinuální integrace je uložení artefaktů na CI server. DevOps umožňuje uložení na vlastní server i na cloudové úložiště artefaktů Azure Pipelines. V našem případě jsme využili cloudového úložiště, které DevOps umožňuje, a pomocí úlohy *Publish Build Artifacts* jsme artefakty z *build.artifactstagingdirectory* uložili na CI server pro možnost použití v rámci vystavení aplikace.

```
steps:
- task: PublishBuildArtifacts@1
  displayName: 'Publish Artifact: drop'
  inputs:
    PathtoPublish: '$(build.artifactstagingdirectory)'
  condition: succeededOrFailed()
```

Obr. 44. Definice úlohy Publish Build Artifacts pro uložení artefaktů na Azure Pipelines/TFS.

6.2.4 Pipeline proměnné

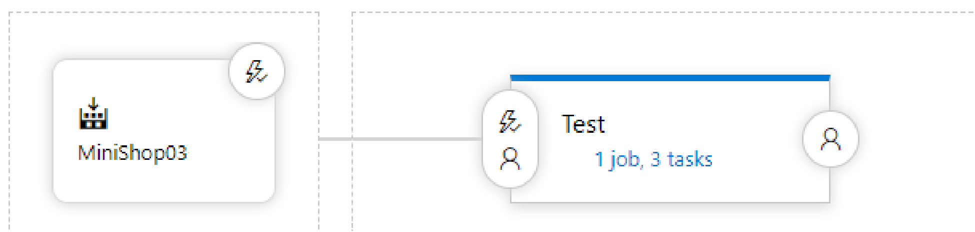
Pro usnadnění definic, snížení redundantních hodnot parametrů a možnost určité pružnosti skriptů, je možné v rámci pipeline používat vlastní proměnné. Na obr. 45 je znázorněn seznam používaných vlastních proměnných v build pipeline. Dále je možné v Azure DevOps využívat předdefinovaných proměnných [32].

Name ↑	Value
BuildConfiguration	release
BuildPlatform	any cpu
NameSpaceWeb	$(ProjectName).Web
NameSpaceWebApi	$(ProjectName).WebApi
ProjectName	MiniShop

Obr. 45. Seznam vlastních proměnných v build pipeline.

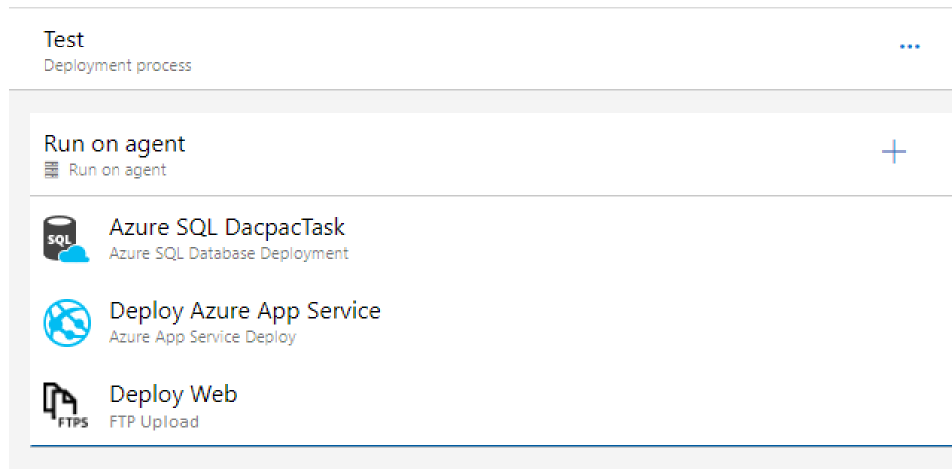
6.3 Nastavení kontinuálního doručení

Po nastavení procesu kontinuální integrace, tj. po vytvoření a uložení artefaktů, nám nebráni nic v nastavení kontinuálního doručování. Za testovací prostředí, prostřednictvím kterého je aplikace MiniShop hostována, jsme zvolili službu Azure App Service. Tato služba pak byla na portále Azure založena pod identickou organizací jako DevOps projekt.



Obr. 46. Ukázka pipeline kontinuálního doručování v Azure DevOps.

Pro ukázku kontinuálního doručovaní bude aplikace nasazována pouze na testovací prostředí. Reálný vývoj, např. webových aplikací, běžně vyžaduje 2–3 prostředí pro projekt. Princip konfigurace kontinuálního nasazení je u dalších prostředí obdobný.



Obr. 47. Dílčí kroky celé deploy pipeline.

6.3.1 Pipeline job

Obdobně jako u pipeline pro kontinuální integraci jsou i u kontinuální dodávky dílčí kroky pipeline zpracovávány v jobu, který je spouštěn agentem. Pro účely práce opět volíme agent pool *Hosted VS2017* z obdobných důvodů jako při kontinuální integraci.

Důležitým parametrem je v tomto kroce definování zdroje artefaktů, které se budou nasazovat na testovací prostředí. Při inicializaci agenta budou tyto soubory nakopírovány do pracovního adresáře agenta.

Obr. 48. Ukázka nastavení agent jobu v DevOps pro deploy pipeline.

6.3.1.1 Nasazení databáze

Při kontinuální dodávce je nezbytné myslet na spuštění databázových skriptů, jinak řečeno vystavení databáze. Databáze pro aplikaci MiniShop je obdobně jako testovací prostředí hostována v cloudových službách Azure, konkrétně ve službě Azure SQL Database.

V DevOps je pro tuto službu předdefinovaná úloha *Azure SQL Database Deployment*. Je nezbytné definovat parametry jako např. databázový server, způsob autentifikace, databázi či přihlašovací údaje. Nejdůležitější je však určit, jakým způsobem bude docházet k nasazení databáze. Pro databázi aplikace MiniShop jsme zvolili způsob datového balíčku *SQL DACPAC File*. Balíček se vytváří v buildu API aplikace a v úloze *SQL DACPAC File* je nutné definovat jeho umístění v pracovním adresáři agenta. Ukázka skriptu pro nasazení databáze je na obr. 49.

```
variables:
  LoginDB: ''
  PasswordDB: ''

steps:
- task: SqlAzureDacpacDeployment@1
  displayName: 'Azure SQL DacpacTask'
  inputs:
    azureSubscription: 'MiniShop03 - Service Endpoint'
    ServerName: MiniShopServer
    DatabaseName: MiniShop
    SqlUsername: '$(LoginDB)'
    SqlPassword: '$(PasswordDB)'
    DacpacFile: '$(System.DefaultWorkingDirectory)/MiniShop03-CI-
03/drop/src/MiniShop.DB/bin/Release/MiniShop.DB.dacpac'
  enabled: false
```

Obr. 49. Definice úlohy Azure SQL Database Deployment pro vystavení databáze.

6.3.1.2 Nasazení API aplikace

Vystavování aplikace na službu Azure App Service je v DevOps plně automatizované. V úloze *Azure App Service Deploy* stačí pouze nadefinovat konkrétní hosting a umístění balíčku (artefaktu) API aplikace v pracovním adresáři agenta. V rámci Azure App Service se smažou soubory z předchozího vystavení a nahradí se novými.

Azure App Service Deploy ⓘ

Task version 4.*

Display name *

Deploy Azure App Service

Connection type * ⓘ

Azure Resource Manager

Azure subscription * | Manage

MiniShop03 - Service Endpoint

App Service type *

Web App on Windows

App Service name *

minishop03

Deploy to Slot or App Service Environment ⓘ

Virtual application ⓘ

Package or folder * ⓘ

$(System.DefaultWorkingDirectory)/**/*.zip

Obr. 50. Základní nastavení úlohy Azure App Service Deploy v prostředí DevOps.

6.3.1.3 Nasazení Webové aplikace

Pro vystavení webové aplikace je možné využít protokol FTP pro přenos a uložení souborů na server, který hostuje službu prostředí. Určená úloha pro definici tohoto úkonu se v Azure DevOps nazývá *FTP Upload*. V úloze se definují parametry pro navázání spojení FTP, zdrojový adresář souborů, které se mají uložit na hostující server, a cílový adresář pro uložení souborů.

```
variables:
  NameSpaceWeb: '$(ProjectName).Web'

steps:
- task: FtpUpload@1
  displayName: 'Deploy Web'
  inputs:
    rootDirectory: '$(System.DefaultWorkingDirectory)/MiniShop03-CI-
03/drop/src/$(NameSpaceWeb)'
    remoteDirectory: /site/wwwroot/wwwroot
    cleanContents: true
    preservePaths: true
    trustSSL: true
  enabled: false
```

Obr. 51. Základní nastavení úlohy FTP Upload v prostředí DevOps

7 SHRNUTÍ NASAZENÍ CI/CD

Kontinuální integrace a doručení má několik výhod pro společnosti zabývající se vývojem softwaru, a to především svou plynulostí, efektivností a cenovou návratnost při produkci softwaru. CI/CD je v mnoha případech poměrně jednoduše integrovatelný proces do již existujícího vývojového prostředí. Proces CI/CD zaručuje úspěšné uveřejňování verzí softwaru, protože každá změna kódu je nejprve automatizovaně testována a reportována. Vývojový tým je tak schopen rychle a včasně identifikovat a opravit případné chyby. S implementací procesu CI/CD se zároveň utváří disciplína vývojového týmu, která zlepšuje vývojové standardy a kvalitu vyvíjeného kódu.

I přes počáteční investice může kontinuální integrace a doručení výrazně zvýšit návratnost investice, protože efektivita celého výrobního cyklu softwaru se zvýší tím, že společnost platí méně času stráveného na vývoji pro stejně kvalitní produkt.

Náklady na hardware a software potřebný pro kontinuální integraci se na počátku liší zvoleným technickým řešením. Technické řešení je často ovlivněno velikostí společnosti a jejím dosavadním řešením infrastruktury. U cloudového řešení CI/CD nejsou tak velké počáteční investice a náklady se dají považovat za pravidelné provozní výdaje, jejichž výše se odvozuje od celkového času využívání cloudové služby. Cloudové řešení je tak často využíváno malými a středně velkými společnostmi. Pro on-premise řešení, u kterého je licencovaný software CI/CD provozován v rámci lokálního hardware, je počáteční investice kapitálového charakteru.

Doba potřebná k implementaci a vybudování prostředí CI/CD není na počátku zanedbatelná a náklady u složitějších projektů mohou být významné. Správně implementovaný proces CI/CD však stačí napsat pouze jednou, což značně zkracuje dobu, kterou členové týmu tráví při ručním procesu vystavování a prodlevách při čekání testovacího týmu na novou verzi software.

Konkrétní implementace procesu CI/CD, je ukázána v kapitole 6 Implementace CI/CD. Pro demonstraci procesu CI/CD byl použit cloudový nástroj Azure Pipelines, který výbornou platformou mezi nástroji pro kontinuální integraci a doručování. Azure Pipelines je spolehlivou platformou, která nabízí pro nespočet programovacích jazyků široké možnosti nastavení procesu CI/CD. Koncept úloh, který nástroj Azure Pipelines používá pro definování pipelines, velice usnadňuje práci a přehlednost při nastavování procesů. Nabídka úloh a jejich funkčnosti jsou natolik komplexní, že podrobná dokumentace [35] je užitečným

pomocníkem při implementaci. V porovnání s populárním nástrojem Jenkins je Azure Pelines pro nastavení procesu kontinuální integrace více přehlednější a intuitivnější.

Velký potenciál nástroje Azure Pipeline vidím ve spojení s ostatními nástroji, které Azure nabízí. Je možné vytvořit jeden velký integrovaný systém, který umožní kompletně integrovat celý proces vývoje softwaru. Ve své podstatě implementovat zásady přístupu vývoje DevOps. Nástroj Azure Boards umožňuje plánování, přerozdělování a kompletní správu vývojových činností. Azure Repo nabízí hostování repozitářů a správu při verzování. Azure Test Plans umožňuje vytváření testovacích scénářů a komplexní správu testů s vývojovými úkoly. Všechny tyto nástroje je možné integrovat s IDE Microsoft Visual Studio.

ZÁVĚR

Hlavní cíl diplomové práce byl splněn provedením rozboru moderních nástrojů kontinuální integrace a kontinuálního doručování pro možnosti praktického nasazení s výběrem vhodného integračního prostředí pro nasazení potřeb CI/CD vytvořené vzorové aplikace.

V teoretické části byl představen význam kontinuální integrace a kontinuálního doručování pro vývojový tým a přínosy, které kontinuální metody přináší pro vývoj softwaru. Dále byly specifikovány kroky procesu CI/CD a vysvětlena jejich funkčnost a význam pro CI/CD. Současně s definovanými kroky byly popsány zásady, které by měly být při nasazení dodržovány, aby implementovaný proces CI/CD nabýval svého významu, např. zásady pro vytváření build skriptů, rychlost sestavení, pokrytí softwaru automatizovanými testy, ale také základní zásadami, které by měli dodržovat samotní vývojáři při vývoji softwaru. Zásady automatizovaného testování a přínos automatizovaného testování pro proces CI/CD byl popsán na konkrétních typech testů – jednotkových, integračních, kouřových testech a testech uživatelského rozhraní.

Závěr teoretické části byl věnovaný rozboru konkrétních moderních nástrojů pro implementaci procesu CI/CD na softwarový projekt. Nejprve byly uvedeny obecné zásady, na co se zaměřit při výběru nástroje CI/CD a následně byly rozebrány možnosti nástrojů a jejich specifika. U každého nástroje – Azure DevOps Services/Server, Jenkins a Bitbucket, byla ukázána základní implementace kontinuální integrace.

Praktická část byla věnována vzorové webové aplikaci MiniShop, která představuje zjednodušený internetový obchod. Byly stanoveny požadavky pro návrh aplikace a následně popsána architektura a implementace samotné webové aplikace. Pro aplikaci byla zavedena sada jednotkových a integračních testů, jejíchž implementace byla taktéž podrobně vysvětlena.

Pro vytvořenou aplikaci byl nasazen proces kontinuální integrace a kontinuální dodávky pomocí nástroje Azure Pipelines v cloudovém prostředí Azure DevOps Services. Pro kroky této implementace byl proveden rozbor a představeny možnosti dalších rozšiřujících nastavení.

V závěru diplomové práce bylo provedeno zhodnocení nasazení principů a metod kontinuální integrace a kontinuálního doručování.

SEZNAM POUŽITÉ LITERATURY

[1] BOOCH, Grady. *Object-oriented analysis and design with applications*. 2nd ed. Redwood City, Calif.: Benjamin/Cummings Pub. Co., c1994. ISBN 0-8053-5340-2.

[2] ASTESIANO, E. *Fundamental approaches to software engineering: First International Conference, FASE '98 held as part of the joint European Conferences on Theory and Practice of Software, ETAPS '98, Lisbon, Portugal, March 28-April 4, 1998: proceedings*. New York: Springer, c1998. ISBN 978-3-540-64303-6.

[3] FOWLER, Martin. *Continuous Integration (original version)* [online]. 10. 9. 2000 [cit. 2019-04-10]. Dostupné z: https://www.martinfowler.com/articles/original-ContinuousIntegration.html#TheBenefitsOfContinuousIntegration

[4] *CruiseControl* [online]. [cit. 2019-04-10]. Dostupné z: http://cruisecontrol.source-forge.net/

[5] SHARMA, Alek. *A Brief History of DevOps, Part III: Automated Testing and Continuous Integration* [online]. 1. 2. 2018 [cit. 2019-04-10]. Dostupné z: https://circleci.com/blog/a-brief-history-of-devops-part-iii-automated-testing-and-continuous-integration/

[6] DUCHO, Pavel. *Kontinuální integrace: lék na lidské chyby v deploymentu* [online]. 2. 11. 2016 [cit. 2019-04-10]. Dostupné z: https://www.root.cz/clanky/kontinualni-integrace-lek-na-lidske-chyby-v-deploymentu/

[7] FOWLER, Martin. *Continuous Integration* [online]. 1. 3. 2006 [cit. 2019-04-10]. Dostupné z: https://martinfowler.com/articles/continuousIntegration.html

[8] DUVALL, Paul M., Steve MATYAS a Andrew GLOVER. *Continuous integration: improving software quality and reducing risk*. Upper Saddle River, NJ: Addison-Wesley, c2007. ISBN 978-0-321-33638-5.

[9] *Google Trends: Continuous Integration* [online]. [cit. 2019-04-10]. Dostupné z: https://trends.google.cz

[10] KAWALEROWICZ, Marcin a Craig BERNTSON. *Continuous integration in .NET*. Greenwich, Conn.: Manning, c2011. ISBN 9781935182559.

[11] GÜNTHER, Tobias. *Version Control Best Practices* [online]. 1. 2017 [cit. 2019-04-11]. Dostupné z: https://www.git-tower.com/blog/version-control-best-practices/

[12] HUMBLE, Jez a David FARLEY. *Continuous delivery: reliable software releases through build, test, and deployment automation*. Upper Saddle River, NJ: Addison-Wesley, 2010. ISBN 978–0–321–60191–9.

[13] JONES, Mike. *Standard and custom Toolset configurations* [online]. [cit. 2019-04-14]. Dostupné z: https://docs.microsoft.com/cs-cz/visualstudio/msbuild/standard-and-custom-toolset-configurations?view=vs-2019

[14] CRISPIN, Lisa a Janet GREGORY. *Agile testing: a practical guide for testers and agile teams*. Upper Saddle River, NJ: Addison-Wesley, c2009. Addison-Wesley signature series. ISBN 978-0321534460.

[15] BECK, Kent. *Test-driven development: by example*. Boston: Addison-Wesley, c2003. ISBN 978-0321146533.

[16] JORGENSEN, Paul. *Software testing: a craftsman's approach*. 3rd ed. Boca Raton: Auerbach Publications, c2008. ISBN 0-8493-7475-8.

[17] *What is Functional Testing? Types & Examples* [online]. 28. 2. 2017 [cit. 2019-04-14]. Dostupné z: https://www.guru99.com/functional-testing.html

[18] BROCKSCHMIDT, Kraig. *A primer on unit testing and continuous integration* [online]. 28. 2. 2017 [cit. 2019-04-14]. Dostupné z: https://docs.microsoft.com/en-us/visualstudio/cross-platform/tools-for-cordova/debug-test/unit-test-primer?view=toolsforcordova-2017#using-unit-tests-and-continuous-integration-to-minimize-the-cost-of-bugs

[19] *Integration Testing: What is, Types, Top Down & Bottom Up Example* [online]. 28. 2. 2017 [cit. 2019-04-14]. Dostupné z: https://www.guru99.com/integration-testing.html

[20] *Draw.io Online* [online]. [cit. 2019-04-17]. Dostupné z: https://about.draw.io/

[21] SHARMA, Lakshay. *Smoke Testing* [online]. 9. 5. 2016 [cit. 2019-04-17]. Dostupné z: https://www.toolsqa.com/software-testing/smoke-testing/

[22] *Smoke Testing* [online]. [cit. 2019-04-17]. Dostupné z: http://softwaretestingfundamentals.com/smoke-testing/

[23] *GUI Testing Tutorial: User Interface (UI) TestCases with Examples* [online]. [cit. 2019-04-18]. Dostupné z: https://www.guru99.com/gui-testing.html

[24] *Comparison of GUI testing tools* [online]. [cit. 2019-04-18]. Dostupné z: https://en.wikipedia.org/wiki/Comparison_of_GUI_testing_tools

[25] PITTET, Sten. *Continuous integration vs. continuous delivery vs. continuous deployment* [online]. [cit. 2019-04-18]. Dostupné z: https://cs.atlassian.com/continuous-delivery/principles/continuous-integration-vs-delivery-vs-deployment

[26] *Devops: Continuous Integration vs Continuous Delivery vs Continuous Deployment* [online]. [cit. 2019-04-19]. Dostupné z: https://www.nastel.com/blog/devops-continuous-integration-vs-continuous-delivery-vs-continuous-deployment/

[27] CHACON, Scott a Ben STRAUB. *Pro Git: Everything you need to know about git, 2nd ed,* [online]. 2.1.146. 2019 [cit. 2019-04-20]. Dostupné z: https://git-scm.com/book/en/v2

[28] *StackOverflow: Developer Survey Results 2018* [online]. [cit. 2019-04-19]. Dostupné z: https://insights.stackoverflow.com/survey/2018

[29] *Merging vs. Rebasing* [online]. [cit. 2019-04-20]. Dostupné z: https://cs.atlassian.com/git/tutorials/merging-vs-rebasing

[30] *Microsoft Azure* [online]. [cit. 2019-04-22]. Dostupné z: https://azure.microsoft.com

[31] *Microsoft Azure: Azure DevOps* [online]. [cit. 2019-04-22]. Dostupné z: https://azure.microsoft.com/cs-cz/services/devops/

[32] ANDYJLEWIS, KATHRYNEE, Alex HOMER, Steve DANIELSON a David STAHELI. *Predefined variables* [online]. 20. 7. 2018 [cit. 2019-04-22]. Dostupné z: https://docs.microsoft.com/en-us/azure/devops/pipelines/build/variables?view=azure-devops&tabs=yaml

[33] *Official images for .NET Core and ASP.NET Core* [online]. [cit. 2019-04-22]. Dostupné z: https://hub.docker.com/_/microsoft-dotnet-core

[34] *Autofac's documentation* [online]. [cit. 2019-04-22]. Dostupné z: https://autofaccn.readthedocs.io/en/latest/

[35] *Azure Pipelines Documentation* [online]. [cit. 2019-04-22]. Dostupné z: https://docs.microsoft.com/en-us/azure/devops/pipelines/?view=azure-devops

[36] MAO, Andy. *Atlassian Announces End-of-Life (EOL) for Bamboo Cloud: Is it Time to Consider a Managed Cloud Hosting Solution?* [online]. 18. 8. 2016 [cit. 2019-04-22]. Dostupné z: https://elasity.io/blog/atlassian-announces-end-of-life-eol-for-bamboo-cloud-is-it-time-to-consider-a-managed-cloud-hosting-solution/

[37] *Understanding the Bamboo CI Server* [online]. 20. 6. 2017 [cit. 2019-04-22]. Dostupné z: https://confluence.atlassian.com/bamboo/understanding-the-bamboo-ci-server-289277285.html

[38] *Atlassian Marketplace: Bamboo apps* [online]. [cit. 2019-04-23]. Dostupné z: https://marketplace.atlassian.com/search?product=bamboo

[39] *TeamCity Plugins* [online]. [cit. 2019-04-23]. Dostupné z: https://plugins.jetbrains.com/teamcity

[40] *TeamCity Licensing Policy: Licensing Overview* [online]. [cit. 2019-05-15]. Dostupné z: https://confluence.jetbrains.com/display/TCD18/Licensing+Policy

[41] *Atlassian: Bamboo Licensing and Pricing* [online]. [cit. 2019-05-16]. Dostupné z: https://cs.atlassian.com/licensing/bamboo

[42] *Atlassian: BitBucket Pricing* [online]. [cit. 2019-05-16]. Dostupné z: https://bitbucket.org/product/pricing?tab=cloud

[43] *Microsoft Azure: Pricing for Azure DevOps* [online]. [cit. 2019-05-16]. Dostupné z: https://azure.microsoft.com/en-ca/pricing/details/devops/azure-pipelines/

[44] *Azure DevOps Server Pricing* [online]. [cit. 2019-05-16]. Dostupné z: https://visualstudio.microsoft.com/team-services/tfs-pricing/

SEZNAM POUŽITÝCH SYMBOLŮ A ZKRATEK

CI Kontinuální integrace (Continuous Integration). Je automatizovaný proces vytvoření buildu software a spuštění dalších služeb pro kontrolu a monitoring softwaru.

CD Kontinuální doručování (Continuous Delivery nebo Continuous Deployment). Je automatizovaný proces nasazení verze softwaru do prostředí, v rámci kterého má být software spuštěn.

VCS Systém správy verzí (Version Control System) slouží pro uchovávání historie provedených změn v datech.

.NET Označení pro soubor technologií v softwarových produktech od společnosti Microsoft.

TDD Programování řízení testy (Test-driven development). Je proces při práci s jednotkovými testy založený na malých, stále se opakujících se krocích.

GUI Grafické uživatelské rozhraní (Graphical User Interface), umožňuje ovládat software pomocí interaktivních grafických ovládacích prvků.

UI Uživatelské rozhraní (User Interface) je souhrn způsobů, jakými je možné ovládat počítačový software.

UAT Uživatelské akceptační testování (User Acceptance Testing) je proces testování softwaru před předáním zákazníkovi.

SVN Subversion je centralizovaný systém pro správu verzí.

TFVC Team Foundation Version Control je verzovací systém společnosti Microsoft.

API Rozhraní pro programování aplikací (Application Programming Interface).

IDE Software určený vývojářům pro vytváření kódů (Integrated Development Environment).

CLI Uživatelské rozhraní příkazového řádku (Command Line Interface).

TFS Produkt společnosti Microsoft určený pro vývoj, testování a management software (Team Foundation Server).

VSTS Je cloudová sada nástrojů pro spolupráci členů týmu při vývoji software (Visual Studio Team System).

URL	Řetězec znaků sloužící k přesné specifikaci umístěných zdrojů infomací na internetu (Uniform Resource Locator).
PHP	Hypertextový preprocesor (Hypertext Preprocessor) je skriptovací programovací jazyk.
DevOps	Je přístup k vývoji software, jehož hlavní podstatou je spolupráce, komunikace a integrace mezi členy týmu pomocí technologií (Development Operations).
SSH	Zabezpečený komunikační protokol (Security Shell).
YAML	Formát pro serializaci strukturovaných dat (Ain't Markup Language).
EF	Entity Framework je open-source ORM framework pro technologie .NET.
ORM	Je přístup zajišťující automatickou konverzi mezi objektově orientovaným jazykem a relační databází (Object Relational Mapping).
LINQ	Je integrovaný jazyk do technologie .NET určený k vytváření dotazů (Language Integrated Query).
SQL	Strukturovaný dotazovací jazyk používaný pro práci s dady v relačních databázích (Structured Query Language).
HTTP	Komunikační protokol pro přenos informací v internetu (Hypertext Transfer Protocol).
RSS	Je formát zpráv určených pro čtení novinek na webových stránkách (Rich Site Summary).
MVC	Je softwarová architektura, která rozděluje datový model, řídící logiku a uživatelské rozhraní do tří nezávislých komponent (Model-view-controller).
IIS	Je softwarový webový server pro operační systémy Windows (Internet Information Services).

SEZNAM OBRÁZKŮ

Obr. 1. Popularita vyhledávání pojmu Continuous Integration na celém světě. [9]...12
Obr. 2. Blokové schéma procesu kontinuální integrace. [8] [10] [20] 13
Obr. 3. Blokové schéma oddělování build scriptu od IDE. [8] 15
Obr. 4. Ukázka základního procesu CI a funkce CI serveru. [20] 16
Obr. 5. Základní blokové schéma procesu CI/CD. [8] [20].......................................20
Obr. 6. Sekvenční diagram ukazující změny probíhajících v rámci procesu CI/CD. [12] 21
Obr. 7. Základní koncept deployment pipeline. [12] 22
Obr. 8. Porovnání dílčích kroků kontinuální integrace, doručování a nasazení. [20] [26] 23
Obr. 9. Schéma principu centralizovaného verzovacího systému. [27].....................26
Obr. 10. Schéma principu distribuovaného verzovacího systému. [27] 27
Obr. 11. Výsledky průzkumu používání verzovacích systémů při vývoji softwaru na portále StackOverflow v roce 2018. [28] 28
Obr. 12. Princip ukládání dat pomocí snímků v průběhu času. [27] 28
Obr. 13. Ukázka práce s větvemi v rámci verzovacího systému. [29] 29
Obr. 14. Testovací pyramida. [14] 32
Obr. 15. Proces zpracovávaných automatizovaných testů. [20] 34
Obr. 16. Ukázka dashboardu webového rozhraní nástroje Azure DevOps Services s widgety zobrazujícími výsledky procesu CI/CD. 39
Obr. 17. Ukázka webového rozhraní nástroje Jenkins..41
Obr. 18. Ukázka buildu .NetCore aplikace v rámci Jenkins Freestyle projektu.........42
Obr. 19.Ukázka buildu .NET Core aplikace v rámci Jenkins Pipeline projektu. 43
Obr. 20. Ukázka .NET Core aplikace MiniShop v nástroji BitBucket Pipelines. 44
Obr. 21. Architektura členění procesu CI/CD v Bamboo. [37] 47
Obr. 22. Ukázka webového rozhraní nástroje TeamCity..48
Obr. 23. Class diagram charakterizující datovou vrstvu aplikace MiniShop. 54
Obr. 24. Vrstvy architektury aplikace MiniShop...56
Obr. 25. Přehled testů v projektu MiniShop. 57
Obr. 26. Metoda CreateOrderValidation v projektu MiniShop.Domain.Services.Orders 59
Obr. 27. Jednotkový test ověřující podmínky pro vytvoření objednávky produktu. ..60

Obr. 28. Integrační test ověřující úspěšné vytvoření objednávky. 61
Obr. 29. Metoda pro navrácení dat do původního stavu před spuštěním testu testovací třídy OrderControllerTest. 62
Obr. 30. Ukázka projektu MiniShop03 zařazený pod organizací tlecian123 v cloudové službě Azure DevOps. 63
Obr. 31. Dílčí kroky (úlohy) celé build pipeline. 64
Obr. 32. Ukázka nastavení Agent pool. 65
Obr. 33. Ukázka propojení projektu DevOps s repozitářem nástroje BitBucket. 65
Obr. 34. Ukázka nastavení agent jobu v DevOps pro build pipeline. 66
Obr. 35. Definice úlohy pro instalaci Npm balíčků. 67
Obr. 36. Definice úlohy NuGet Tool Installer pro instalaci NuGet balíčků. 67
Obr. 37. Definice úlohy Nuget pro obnovení NuGet balíčků. 68
Obr. 38. Konfigurační soubor FolderProfile.pubxml projektu MiniShop.WebApi aplikace MiniShop. 69
Obr. 39. Definice úlohy Visual Studio Build pro nastavení parametrů MSBuildu. ...69
Obr. 40. Definice úlohy Npm pro vytvoření buildu Angular aplikace. 70
Obr. 41. Definice úlohy Visual Studio Test pro automatizované spuštění jednotkových a integračních tesů. 71
Obr. 42. Definice úlohy Copy Files pro uložení artefaktů webové aplikace. 72
Obr. 43. Definice úlohy Copy Files pro uložení artefaktů API aplikace. 72
Obr. 44. Definice úlohy Publish Build Artifacts pro uložení artefaktů na Azure Pipelines/TFS. 72
Obr. 45. Seznam vlastních proměnných v build pipeline. 73
Obr. 46. Ukázka pipeline kontinuálního doručování v Azure DevOps. 73
Obr. 47. Dílčí kroky celé deploy pipeline. 74
Obr. 48. Ukázka nastavení agent jobu v DevOps pro deploy pipeline. 74
Obr. 49. Definice úlohy Azure SQL Database Deployment pro vystavení databáze. 75
Obr. 50. Základní nastavení úlohy Azure App Service Deploy v prostředí DevOps. 76
Obr. 51. Základní nastavení úlohy FTP Upload v prostředí DevOps 76

SEZNAM TABULEK

Tab. 1. Porovnání vlastností funkčního a nefunkčního testování. [7] 33
Tab. 2. Základní přehled nástroje Azure DevOps Services/Azure DevOps Server. ..41
Tab. 3. Základní přehled nástroje Jenkins. 43
Tab. 4. Základní přehled nástroje BitBucket Pipeline. 45
Tab. 5. Základní přehled nástroje Bamboo. 47
Tab. 6. Základní přehled nástroje TeamCity. 49
Tab. 7. Orientační ceny nástrojů CI/CD. Ceny jsou uvedeny za jeden měsíc. 51
Tab. PI.8.Přehled kritérií a orientačních cen nástrojů CI/CD. 90

SEZNAM PŘÍLOH

P I Přehled kritérií pro stanovení orientační pořizovací ceny nástrojů CI/CD.

PŘÍLOHA P I: KRITÉRA PRO STANOVANÍ CENY NÁSTROJŮ CI/CD.

Tab. PI.8.Přehled kritérií a orientačních cen nástrojů CI/CD.

	TeamCity [40]	Bamboo [41]	Bamboo [41]	BitBucket Pipelines [42]	BitBucket Pipelines [42]	Azure Pipelines [43]	Azure Pipelines [43]	Azure DevOps Server [44]
Forma řešení	On-premis	On-premis	On-premis	Cloud	Cloud	Cloud	Cloud	On-premis
Min. agentů	3	1	neomezeno (lokální)	neomezeno	neomezeno	1	1	neomezeno
Max. jobů	neomezeno	neomezeno	10	neomezeno	neomezeno	1	1	neomezeno
Max. build (min/měsíc)	neomezeno	neomezeno	neomezeno	500	50	neomezeno	1800	neomezeno
Max. uživatelů	neomezeno	neomezeno	neomezeno	5	5	neomezeno	5	1
Mim. uživatelů	neomezeno	neomezeno	neomezeno	5	neomezeno	neomezeno	neomezeno	1
Zkušebních dnů zdarma	60	30	30	x	x	x	x	90
Platnost ceny (měsíců)	12	12	12	1	1	1	1	1
Cena (Kč)	52 000 Kč	25 300 Kč	230 Kč	230 Kč	0 Kč	770 Kč	0 Kč	180 Kč