Česká zemědělská univerzita v Praze

Provozně ekonomická fakulta

Katedra informačních technologií

Bakalářská práce

Automatické nasazování aplikací – CI/CD

Iryna Osadcha

ČESKÁ ZEMĚDĚLSKÁ UNIVERZITA V PRAZE

Provozně ekonomická fakulta

ZADÁNÍ BAKALÁŘSKÉ PRÁCE

Iryna Osadcha

Systémové inženýrství a informatika
Informatika

Název práce

Automatické nasazování aplikací – CI/CD

Název anglicky

Continuous Integration and Continuous delivery in automated aplication deployment

---

Cíle práce

Hlavním cílem práce je navržení poskytnutého procesu automatizace pro vývojáře pomocí vlastní aplikace. Dílčí cíle práce jsou:
- vytvoření metody CI/CD,
- implementace nástrojů SonarQube a Black Duck,
- vytvoření Unit a Smoke testů.

Metodika

Metodika řešené problematiky bakalářské práce je založená na studiu a analýze informačních zdrojů v oblasti moderních technologii pro automatizaci vývoje SW.

Na základě teoretických znalosti bude navrhnuto vlastní řešení problému formou ověření automatizovaného procesu nasazení aplikace.

Pomocí syntézy teoretických poznatků a výsledků praktické časti budou formulovány závěry práce.

Doporučený rozsah práce

35 – 40 stran

Klíčová slova

Continuous integration, kontejner, aplikace, automatizace, pipeline

---

Doporučené zdroje informací

Azure Container Registry documentation [online], Dostupné z https://docs.microsoft.com/en-us/azure/container-registry/

Azure DevOps documentation [online], Dostupné z https://docs.microsoft.com/en-us/azure/devops/?view=azure-devops

Belmont J.: Hands-On Continuous Integration and Delivery. Packt Publishing, 2018. ISBN 978-1789130485

Duvall, P.,Matyas S., Glover A.: Continuous integration: improving software quality and reducing risk. Upper Saddle River : Addison-Wesley, 2007. ISBN 978-0-321-33638-50-321-33638-0

Lezsko R.: Continuous Delivery with Docker and Jenkins. Packt Publishing, 2017. ISBN 978-1787125230

RedHat Interactive Learning Portal [online], Dostupné z https://learn.openshift.com/

Rossel S.: Continuous Integration, Delivery, and Deployment. Packt Publishing, 2017. ISBN 978-1787286610

Předběžný termín obhajoby

2020/21 LS – PEF

Vedoucí práce

Ing. Jan Masner, Ph.D.

Garantující pracoviště

Katedra informačních technologií

Elektronicky schváleno dne 7. 9. 2020

Ing. Jiří Vaněk, Ph.D.

Vedoucí katedry

Elektronicky schváleno dne 19. 10. 2020

Ing. Martin Pelikán, Ph.D.

Děkan

V Praze dne 07. 03. 2021

Čestné prohlášení

Prohlašuji, že svou bakalářskou práci "Automatické nasazování aplikací – CI/CD" jsem vypracovala samostatně pod vedením vedoucího bakalářské práce a s použitím odborné literatury a dalších informačních zdrojů, které jsou citovány v práci a uvedeny v seznamu použitých zdrojů na konci práce. Jako autorka uvedené bakalářské práce dále prohlašuji, že jsem v souvislosti s jejím vytvořením neporušila autorská práva třetích osob.

V Praze dne 15.3.2021 Iryna Osadcha _____

Poděkování

Ráda bych touto cestou poděkovala Ing. Janu Masnerovi, Ph.D. za ochotu a trpělivost při vedení této bakalářské práce. Dále pak rodině za podporu při napsaní práce. Nakonec bych rada poděkovala společnosti ŠKODA AUTO a.s. a to především zástupcům odděleni IT Middleware Services za poskytnutí všech zdrojů pro zpracování této bakalářské práce.

Automatické nasazování aplikací – CI/CD

Abstrakt

Práce popisuje proces automatického nasazování aplikace CI/CD pro Python web aplikace pro společnost ŠKODA AUTO a.s. Cílem práce je návrh, implementace a ověření procesu nasazování aplikace. Nezbytnou součásti bakalářské práce je vytvoření vlastní webové aplikace v Python s využitím knihovny Flask a použití Microsoft Azure Devops a Docker pro sestavení build pipeline, následně vytvoření release pipeline, která rozloží na serveru aktuální verzi kódu pomocí Docker obrazu a nahrazení předchozího Docker obrazu novým pomocí technologie Kubernetes s následným zasláním potvrzení náhrady na OpenShift.

Prvním dílčím cílem je nastavit pravidelnou analýzu kódu pomocí nástroje SonarQube, která upozorní na případné nesrovnalosti v kódu.

Druhým dílčím cílem je integrační testování – vytvoření unit a smoke testů. Pro složité, obsahující velké množství vazeb aplikace integrační testování je nezbytné z hlediska ověření, že všechny komponenty fungují správně jako celek. Pro testovací aplikace byly napsané unit a smoke testy, jejichž cílem je zajistit, že celý systém je správně rozložen a nakonfigurován na serveru, kde běží.
Provedená práce bude následně sloužit jako podklad ve vývoji.

Klíčová slova: Continuous integration, kontejner, aplikace, automatizace, pipeline

Continuous Integration and Continuous delivery in automated application deployment

Abstract

The work describes the process of automatically deploying the CI/CD application for web application in cooperation with ŠKODA AUTO a.s. The main goal of the work is the design, implementation and verification of the application deployment process. An essential part of the bachelor thesis is to create a custom web application in Python using the Flask framework and then, using Microsoft Azure Devops and Docker to create a build pipeline, then create a release pipeline that distributes the current version of the Docker image on the server and replace the previous Docker image with new one with help of Kubernetes technology, then sending the confirmation to OpenShift.

The first partial goal is to set up a regular analysis of code quality with SonarQube.

The second goal is the creation of unit and smoke tests. Integration testing is necessary to verify that all components work correctly in a huge and complex application. Unit and smoke tests have been written for test applications to ensure that the entire system is properly decomposed and configured on the server where it runs.

The solution can be used as a basis for development in future.

Keywords: **Continuous integration, container, application, automation, pipeline**

Obsah

1 Úvod

V posledních letech prošly téměř všechny oblasti podnikání, které souvisejí s vývojem softwaru, velkými změnami. Z důvodu rychlého rozvoje moderních technologií, zvyšování množství dat a možnosti jejich rychlého zpracování se společnosti snaží být agilnější, a to z důvodu inovací a rychlejší reakce na aktuální tendence. To vyžaduje změny v přístupech k vývoji, testování a vydávání nového kódu. Vzhledem k tomu, že podnikání potřebuje větší flexibilitu, byla zvolena metodologie DevOps, která zajišťuje automatizaci a využití nepřetržité integrace, dodání a testování.

V současné době aktivně narůstá trend automatizace ve vývoji webových aplikací. Mezi hlavní cíle činnosti vyplývající z principů DevOps patří eliminace rizik při předávání kódu do produkčního prostředí a nepřetržitá průběžná kontrola kódu. Na základě požadavku společnosti ŠKODA AUTO a.s. se bakalářská práce zabývá automatickým nasazováním aplikace v prostředí Microsoft Azure DevOps a Red Hat OpenShift, spolu s použitím vlastní webové aplikace, která je realizována v jazyce Python.

V teoretické části je zpracována rešerše v oblasti CI/CD, kontejnerizace, testování a principu fungování webových aplikací. Dále je zpracována rešerše v oblasti systému správce verzí a základní princip jazyku Python. V této části jsou popsány nástroje použitelné k automatickému nasazování aplikace a analýze kódu.

Praktická část se zabývá analýzou požadavku na základě technologického stacku ŠKODA AUTO a.s. Rovněž se soustředí na popis a implementaci navrženého řešení a vytváření CI/CD pipeline s následným ověřením její funkčnosti.

2 Cíl práce a metodika

2.1 Cíl práce

Hlavním cílem práce je navržení procesu automatizace pro společnost ŠKODA AUTO a.s. pomocí vlastní aplikace.
Dílčí cíle práce jsou:

- vytvoření metody CI/CD
- implementace nástrojů analýzy kódu SonarQube a Black Duck
- vytvoření Unit a Smoke testů

2.2 Metodika

Metodika řešené problematiky bakalářské práce je založená na studiu a analýze informačních v oblasti moderních technologii pro oblast automatizace vývoje SW.
Na základě teoretických znalosti bude navrhnuto vlastní řešení problému formou ověření automatizovaného procesu nasazení aplikace na testovacím prostředí ŠKODA AUTO a.s. Pomocí syntézy teoretických poznatků a výsledků praktické části budou formulovány závěry práce.

3 Teoretická východiska

3.1 DevOps jako nástroj pro automatizaci procesu vývoje

DevOps – je metodologie postavená na aktivní spolupráci odborníků z vývojového týmu a odborníků na informační technologie z provozního týmu s cílem vzájemné integrace procesů pro dosažení vyšší možné kvality práce. Sloučením dvou anglických slov Development (vývoj) a Operations (provoz) vznikl nový pojem DevOps. DevOps je aktuální metodologií pro velké společnosti zabývající se vývojem softwaru, popř. jeho dodáním.

Nástroje DevOps řeší celou řadu neefektivností, kterým čelí celý životní cyklus vývoje softwaru. Nástroje DevOps umožňují podnikům automatizovat vývoj softwaru a testování životního cyklu tím, že standardizují a automatizují pohyb a zavádění kódu v různých prostředích. Tyto nástroje pomáhají vývojářům integrovat nepřetržitou zpětnou vazbu, takže mohou zkrátit dobu odezvy a průběžně uvolňovat software na základě zpětné vazby a chování uživatelů.[1]

Historie DevOps začíná v Belgii v roce 2007 a je spojena s Patrickem Deboisem. Ten, jako osoba odpovědná za testování softwaru v rámci veřejné zakázky, pozoroval nesrovnalosti ve spolupráci mezi týmem vývoje (dev tým) a provozním týmem (ops tým). Práce mezi týmy nebyla nastavena optimálně. Mladý tester viděl problémy ve spolupráci, ale v té době ještě neměl návrh na zlepšení. V roce 2008 potkal Patrick Debois v Torontu inženýra Andrewa Schaefera, se kterým se podělil o myšlenku použití Scrum a dalších agilních metodik. Byla zformována Agile System administrace v prostředí Google Groups, kde probíhaly diskuze o agilních metodologiích a nepřetržité integraci. Následně se v Belgii a v USA konala akce DevOpsDays, která posiluje zájem o metodologii mezi mnoha odborníky po celém světě. Jednoduchá myšlenka DevOps se do roku 2015 transformovala v široce používanou strategii v Enterprisech a nadnárodních společnostech. [2]

Podle výzkumu společnosti Grand View Research, zveřejněného v roce 2018, může celosvětová velikost trhu DevOps do roku 2025 dosáhnout 12,85 miliard USD. Růst trhu budou posilovat mj. současná digitalizace podniků za účelem automatizace podnikových procesů, rostoucí vnoření cloudových technologií, zvýšení popularity agilního přístupu ve vývoji softwaru a potřeba lepší spolupráce mezi IT týmy za účelem zvýšení provozní účinnosti.[3]

Celý DevOps proces zahrnuje následující základní nástroje:

- Kódování – vytvoření a analýza kódu,
- Build – proces sestavení projektu jako celku
- Testování – za účelem nalezení případných chyb
- Zabalování – proces přípravy aplikace k vydání
- Vydání – finální zveřejnění produktu
- Monitorování – následné sledování způsobu fungování aplikace v produkčním prostředí.

3.2 Životní cyklus aplikace

Životní cyklus aplikace tvoří řada etap, kterou aplikace prochází – od plánování do následného monitorování. I když v současné době existuje velký počet metodologií a technik týkajících se vývoje softwaru, většina z nich se točí kolem mezinárodního standardu ISO/IEC/IEEE 12207, který definuje procesy nezbytné pro definování, vývoj a údržbu aplikace. [4]

Mezi hlavní kroky životního cyklu aplikace patří:

- Plánování – v této fázi se namapuje plán projektu. Definují se téma, cíle, metody, obsah projektu, stanoví se časové možnosti a rozpočtová omezení. Shromáždí se obchodní požadavky a určování uživatelských příběhu (User Story). Současně proběhne výběr jazyka a knihoven, použitých ve vývojovém procesu.
- Kódování – tato fáze zahrnuje realizaci projektu – přidělení zdrojů, pravidelnou práci na projektu, zjišťování a eliminaci bezpečnostních slabin v kódu a zároveň jeho vytváření.
- Testování – tým testuje kód podle daných požadavků s cílem zkontrolovat, zda produkt funguje v souladu s očekáváním. Provádějí se QA testy, statické a dynamické testování ve speciálním testovacím prostředí. Hodně firem v současné době zavádí automatizované testování softwaru.
- Zavedení do provozu – nejcitlivější krok v celém řetězci; software se nasazuje na produkční server.
- Následné monitorování – ve chvíli, kdy už je produkt používán zákazníkem, je nezbytností sledování uživatelských zkušeností a fungování aplikace. Má to význam pro průběžné zlepšování a plánování vývoje nových verzí aplikace. V tuto chvíli cyklus se uzavře a přejde znovu do fáze plánování.[5]

Takto tedy vypadá poněkud zobecněný životní cyklus aplikace. Primárně se táto prace bude zabývat procesem, ve kterém probíhá přenos kódu z testovacího prostředí do prostředí produkčního. S cílem zvýšení efektivity, zkvalitnění služeb a zlepšení konkurenceschopnosti používají moderní podniky speciální DevOps nástroje. Vývojáři díky nim mohou minimalizovat riziko selhání aplikace a posílat ji do produkce. [5] [6]

3.2.1 Testování

Podle standardu ANSI/IEEE 1059 je testování v softwarovém inženýrství proces hodnocení softwarového produktu s cílem zjistit, zda aktuální softwarový produkt splňuje požadované podmínky či nikoli. Testovací proces zahrnuje vyhodnocení vlastností softwarového produktu z hlediska požadavků, z hlediska případných chybějících požadavků, chyb nebo chyb zabezpečení, spolehlivosti a výkonu.[7]
Testování softwaru je důležité, protože pokud se v softwaru vyskytnou chyby, je možné je včas identifikovat a odstranit je ještě před dodáním softwarového produktu. Správně testovaný softwarový produkt zajišťuje spolehlivost, bezpečnost a vysoký výkon, což dále vede k úspoře času, efektivitě nákladů a spokojenosti zákazníků. Nedostatečná kvalita testování může naopak být příčinou finančních i lidských ztrát.
„Existuje tradiční model klasifikace testů, založený na jejich oblasti činnosti (objem kódu, který pokrývají) a jejich určení. Tento model rozděluje kód testů na modulární, integrační a systémové testy. Přidává také smoke testování, testy produktivity (performance tests) a další testy pro různé účely.“ – Python Continuous Integration and Delivery: A Concise Guide with Examples. [8]

Existují tři základní metody testování:

Metoda černé skříňky (Black Box Testing), neboli behaviorální testování, je metoda založená na testování funkčnosti bez odkazu na vnitřní strukturu testovací jednotky. Metoda je takto pojmenována proto, že osoba, která provádí test, nemá možnost dohlížet do vnitřní struktury aplikace. Metoda černé skříňky je často používána pro uživatelské testování, jehož cílem je odhalit, zda očekávané chování odpovídá dle daných požadavků chování skutečnému.[9]
Metoda bílé skříňky (White Box Testing) je metoda založená na testování funkčnosti s odkazem na vnitřní strukturu testovací jednotky. Předpokládá, že zodpovědná osoba zná

strukturu a má přístup ke zdrojovému kódu. White box testování je testování mimo uživatelské rozhraní. [10]

Metoda šedé skříňky (Gray Box Testing) je softwarová testovací metoda, která je kombinací metody Black Box Testing a metody White Box Testing. Vnitřní struktura je zde částečně známá.[11]

3.2.1.1 Modulární testy (Unit tests)

Modulární testy slouží k ověření, že každá jednotka softwaru funguje správně dle zadání. Je to automatizovaný test, který provádí vývojář. Jednotka (unit) je nejmenší testovatelnou částí softwarového produktu. Obvykle má jeden nebo několik vstupů a jeden výstup. Při procesním programování může být jednotkou individuální program, funkce, procedura atd. V objektově orientovaném programování (například v Pythonu) je nejmenší jednotkou metoda, třída nebo modul. Chyby jsou obvykle odstraněny ihned po zjištění a nejsou formálně reportovány. Unit testy se provádějí metodou White box testing.[12] Na obrázku č. 1 je uvedeny příklad unit testu v jazyce Python.

```python
import unittest
from controls import Weather


class UnitTest(unittest.TestCase):

    def setUp(self):
        pass

    def tearDown(self):
        pass

    def test_parsing(self):
        html = open('html_mock.txt', 'r').read()
        data = Weather._parse_result(html)
        self.assertNotEqual(data['data'], [])


if __name__ == '__main__':
    unittest.main()
```

Obrázek 1: Unit test v Pythonu [51]

3.2.1.2 Integrační testy (integration tests)

Integrační testy slouží k ověření, zda veškeré komponenty a jednotky fungují správně jako celek ve skupině. Existuje zde riziko, že se některé komponenty v kódu nemohou správně propojit – integrační testy slouží pro odhalení podobných konfliktů. Integrační testy se provádějí s použitím White box testování.[13]

3.2.1.3 Systémové testy (System tests)

Při provádění systémového testu se část kódu vkládá do prostředí vhodného pro testování a ověřuje se jeho fungování jako celku. Pro systémové testy se používá metodika Black box testování. V praxi je tento typ testování nejblíže k testování reálným uživatelem. Systémové testování zahrnuje následující prvky:

- Testy použitelnosti – ověřují, zdali produkt nebo aplikace mají dobrou uživatelskou zkušenost (user experience).
- Regresní testování – cílem je potvrdit, že poslední provedené změny v kódu neměly negativní dopad na fungování ostatních komponent systému.
- Load testování – ověřuje chování aplikace při konkrétním očekávaném zatížení.
- Funkční testování – provádí se pro zajištění správnosti fungování systému v souladu s uživatelskou specifikací.
- Migration testování – testování programů používaných k migraci dat z jedné aplikace do druhé, náhradní aplikace.
- Testování kompatibility – slouží k ověření, že se systém chová stejně ve všech prostředích.
- Fuzz testování – používá se k poskytování neplatných, neočekávaných nebo náhodných dat vstupům programu. [14]

3.2.1.4 Smoke testy

Smoke testování (též „Build Verification Testing“) je druh testování softwaru, který se skládá z demonstrativního souboru testů, jejichž cílem je zajistit, aby nejdůležitější funkce fungovaly. Výsledek tohoto testování se používá k rozhodnutí, zda je build dostatečně stabilní, aby se mohlo pokračovat v dalším testování. Může být také použit k rozhodnutí, zda lze oznámit vydaní produkce, nebo je-li se třeba vrátit k přepracování. Smoke testy však neslouží jako náhrada funkčních nebo regresních testů. [15]

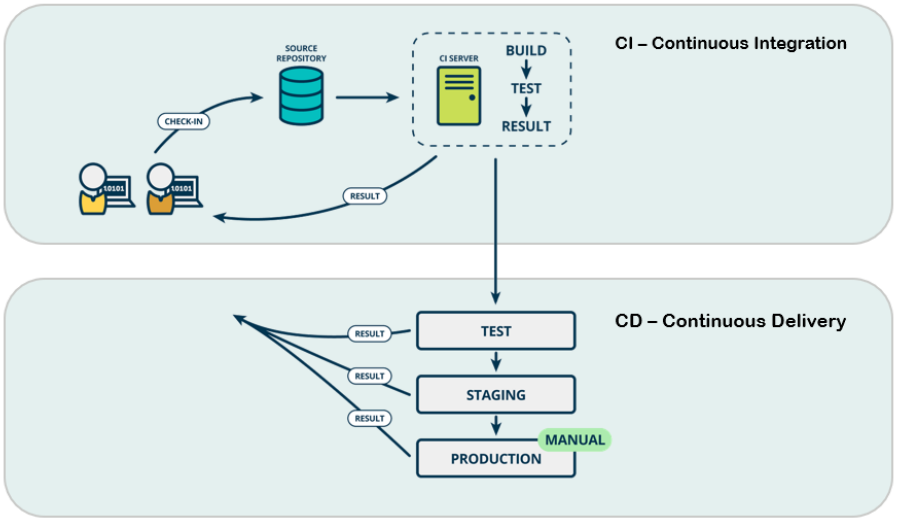
Na obrázku č. 2 uveden příklad smoke testu v jazyce Python.

```python
import unittest
from app import app


class SmokeTest(unittest.TestCase):

    def setUp(self):
        app.config['TESTING'] = True
        app.config['DEBUG'] = False
        self.app = app

    def tearDown(self):
        pass

    def test_main_page(self):
        with app.test_client() as client:
            response = client.get('/')
            self.assertEqual(response.status_code, 200)


if __name__ == '__main__':
    unittest.main()
```

Obrázek 2: Smoke test v Pythonu [52]

3.3 CI/CD

V následujících kapitolách jsou popsáné pojmy Continuous Integration (CI) a Continuous Delivery (CD). Dále se bude zabývat základními pojmy, které se přímo týkají automatického nasazení aplikace. Na obrázku č. 3 je znázorněno, jak vypadá CI/CD.



Obrázek 3: CI/CD schéma [53]

3.3.1 CI – Continuous Integration

CI (nepřetržitá integrace) je moderním, elegantním řešením problému nekorektního sloučení všech větví zdrojového kódu. Jde o to mít pod kontrolou situaci, kdy každý developer v týmu kóduje na vlastní větvi a zároveň se všichni vývojáři přepínají na produkční server, vytvářejí větev pro přidání opravy a migrují kód do produkce, čímž může dojít ke konfliktu při slučování, resp. provedené změny nebudou v souladu se změnami, které současně provádějí ostatní vývojáři. Tento způsob eliminuje chybu v aplikaci v produkčním prostředí v momentě vydání. [16]

Autorem koncepce CI je odborník v oblasti softwarového inženýrství Grady Booch. V roce 1991, kdy docházelo k bouřlivému rozvoji technologií a nutnosti automatizace procesů při vývoji aplikací, byl G. Boochem navržen nový koncept CI, který pak byl používán v extrémním programování. [17]

„Extrémní programování je *velmi „lehká“, ale disciplinovaná metodika*, která zavádí *specifické praktiky jako párové programování, refaktorizace, testy před kódováním* a další. Patří mezi nejpopulárnější a nejpoužívanější agilní metodiky.

Extrémní programování vychází z obvyklých principů a postupů, známých z vývoje softwaru, jsou však dováděny do extrémů (BECK, 2002):

- neustálé revize zdrojového kódu (párové programování),
- neustálé testování vývojáři (testování jednotek) i zákazníky (testování funkcionality),
- každodenní návrh (refaktorizace),
- to nejjednodušší, co ještě může fungovat,
- integrace a testování několikrát denně (nepřetržitá integrace),
- opravdu krátké iterace.“ [18]

V současné době umožňuje Continuous Integration vývojářům integrovat nový kód do sdíleného úložiště (např. GitHub) v průběhu celého dne. V Gitu se ukládají veškeré změny v kódu a existuje možnost sledovat změny každého vývojáře zvlášť. Jakmile jsou změny sloučeny, spouští se automatizované testy (unit testy a integrační testy), jejichž cílem je zajistit, aby aplikace po změnách nespadla. Jedná se o důkladné testování všech modulů aplikace.

Výhodou CI je úplná automatizace. V následujícím odstavci bude znázorněn rozdíl

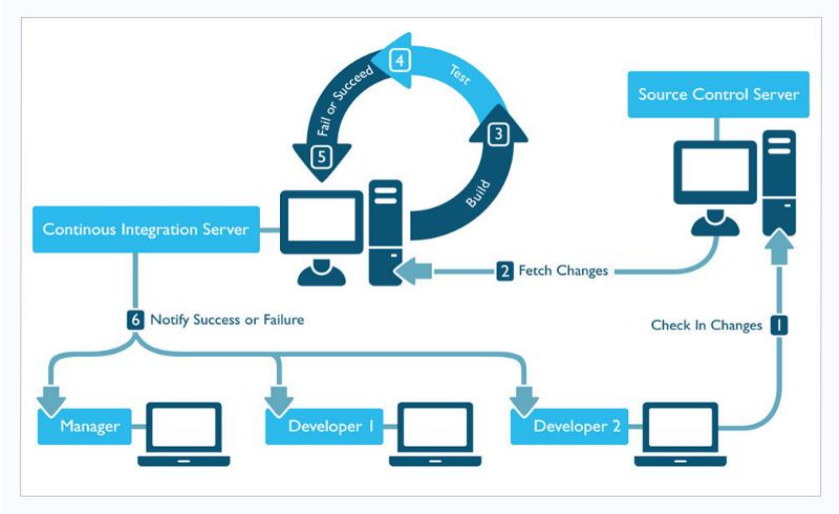
v postupech manuálního nastavení a principu fungování CI/CD. Manuální nastavení probíhá takto: aby došlo k testování, je nutné sestavit build ze zdrojového kódu a pustit ho na testovacím zařízení. Sestavení buildu lze provést ručně (i když je to dost náročné – programátor si musí pamatovat, v jakém pořadí je třeba spouštět jednotlivé soubory). Sestavení lze rovněž nastartovat pomocí speciálního softwaru (jako např. Ant, Maven, Gradle). V tomto procesu je nutná přítomnost odborníka, který napíše speciální příkaz a sestaví build. Nicméně sestavení buildu se nepovažuje za hotový build k testování – je potřeba ho ještě nastartovat. Nastartování má na starosti server. Odborník musí zkopírovat archiv se sestaveným buildem, vložit ho na server a následně na něm spustit testování.

Vyloučením odborníka z celého schématu se vytváří celkově automatizovaný proces CI. Aplikace CI „samostatně přebírá aktualizace kódu“ a lze ji nastavit dvěma způsoby:

- CI „poptává“ změny v kódu jednou za určitý časový interval.
- Repositář samostatně informuje CI o aktualizacích v kódu.

V momentě, kdy se CI dostává ke změnám, se spouští sestavení buildu a automatické testy. V případě, že automatizované testy odhalí konflikt mezi novým a stávajícím kódem, CI na to upozorní vývojáře, čímž je zajištěno, že se aplikace s neopravenými chybami nedostane do produkce. V případě, že sestavení proběhlo úspěšně, CI rozloží aplikaci na testovacím zařízení. [19][24][25]

Typicky je CI proces znázorněn na obrázku č. 4.



Obrázek 4: CI proces [54]

3.3.2 Continuous delivery

Continuous delivery (nepřetržité doručování) je softwarová vývojová praxe, kdy jsou změny kódu automaticky nasazené, otestované a připravené k vydaní do produkce. Poprvé se tento termín objevil v roce 2001 v Agile Manifest:

„*Our highest priority is to satisfy the customer through early and continuous delivery of valuable software.“* [20]

CD staví na základech automatizace sestavování a testování založeném na nepřetržité integraci. Díky nepřetržitému doručovaní změní ručně kroky použité k uvolněni sestavení softwaru do výroby na automatizovaný proces. To znamená, že kromě automatických testů je možné automatizovat realizaci procesu nasazení aplikace kdykoli, a to pouhým stisknutím tlačítka. Hlavní myšlenkou je však zajistit, aby každá změna v kódu byla připravena k přenosu do produkčního prostředí. Proces CD může být automatizován úplně, nebo částečně s možností provedení manuálních akcí v kritických bodech. V praxi lze pomocí continuous delivery nastavit release produktu s libovolnou frekvencí vyhovující obchodním podmínkám. Nejefektivněji samozřejmě funguje takový proces CD, který je spouštěn co nejdříve, a to proto, aby se zajistilo, že deploy kódu probíhá po menších částech, ve kterých je snazší odhalit problémy. [20][21][22]

3.3.3 Continuous deployment

Continuous deployment (CD) je proces vydávání softwaru, který využívá automatické testování k ověření, zda jsou změny v kódové základně správné a stabilní pro okamžitě autonomní nasazení do produkčního prostředí. Cyklus vydávání softwaru se postupem času vyvíjel. Starší proces přesunu kódu z jednoho stroje do druhého, a kontroly, jestli funguje podle očekávání nebo byl proces na prostředí náročným nebo náchylný k chybám. Nyní mohou nástroje automatizovat celý proces nasazení, což umožňuje inženýrským organizacím soustředit se na hlavní obchodní potřeby namísto režie infrastruktury. [23]

3.3.4 Rozdíl Continous Delivery/ Continuous Deployment

Continuous delivery a Continuous Deployment jsou úzce související koncepty, nicméně se někdy definují samostatně podle toho, v jaké míře dochází k automatizaci. V případě Continuous delivery jsou zpravidla změny vývojového tymu v aplikaci automaticky

testovány na chyby a nahrány do uložiště, kde je pak může operační tým nasadit do živého produkčního prostředí. Je to odpověď na problém špatné komunikace mezi vývojovými a obchodními týmy. Za tímto účelem je třeba v případě Continuous Delivery zajistit, aby nasazení nového kódu vyžadovalo minimální úsilí.

Continuous deployment na druhé straně pokrývá některé další kroky v procesu vydávání nového softwaru. Obvykle zahrnuje proces automatické publikace změn vývojáře z úložiště do produkce, kde je zákazníkem použitelný. Řeší problém přetížení operačních týmů manuálními procesy, které zpomalují proces doručovaní aplikace. Staví na výhodách Continuous Delivery a automatizaci další fáze v procesu. [23][24]

3.3.5 Kontejnerizace

Kontejnerizace je v současnosti stále větším trendem ve vývoji softwaru. Zahrnuje zapouzdření neboli zabalení softwarového kódu, včetně všech prvků a závislostí mezi nimi tak, aby mohl běžet a fungovat jako celek zcela izolovaně. Probíhá spuštění aplikace a jejích součástí, jako například systémové knihovny, ve zcela standardizovaném kontejneru, který je spojen s hostem (Host). Kontejner není závislý na zdrojích nebo architektuře hostu, na kterém běží. Veškeré komponenty potřebné k fungování jsou použitelné několikrát. Aplikace běží v kontejneru v izolovaném prostředí a nepoužívá paměť nebo procesor hostovaného operačního systému. Přínos metody kontejnerizace spočívá v tom, že v důsledku výměny prostředí mohou být odhaleny chyby v kódu, o kterých vývojář dříve nevěděl. Při tom neexistuje mezisíťová závislost, pokud jde o riziko konfliktu verzí. Každý kontejner je zvláštním „mikroservisem“ a může být aktualizován bez ohledu na potíže související se synchronizací. [26]

3.3.6 Systém správy verzí (Git)

V současném světě si nelze představit vývoj softwaru bez systému pro správu verzí. Význam tohoto systému je dán tím, že zaznamenává změny souboru nebo sady souborů v průběhu času a umožňuje se později vrátit ke konkrétní verzi. Pokud jde o webdesign, kde je třeba uložit každou verzi obrázku nebo makety, také zde pomůže systém pro správu verzí. Tato předvolba umožňuje vracet soubory do stavu, ve kterém byly před změnami, vrátit projekt do původního stavu, zobrazit změny, zjistit, kdo naposledy něco změnil (a případně způsobil problém), kdo a kdy nastavil úlohu apod.

3.3.6.1 Historie Gitu

„*V XXI. století je kvalita softwaru novým standardem vysoké úrovně profesionality, a proto je důležité hledání rychlých způsobů zavádění inovací. Git dovoluje urychlit proces vývoje a začít přinášet zákazníkům prospěch rychleji.*“ – Sid Sidgebrandige, generální ředitel GitLab. 7. dubna 2015 vydal vývojář Linuxového jádra Linus Torvalds první systém správy verzí Git. Cílem bylo vytvořit rychlý a jednoduchý nástroj s kvalitní podporou nelineárního vývoje, plně distribuovaný, který bude moct pracovat s velkými projekty. [28][41]

3.3.6.2 Větve v Gitu

Téměř každý systém pro správu verzí v nějaké formě podporuje větvení. Použitím větve se může odchýlit od hlavní linie vývoje a pokračovat v práci nezávisle na ní, aniž by zasahoval do hlavní linie. Pro různé systémy správce verzí je větvení velmi náročný proces, který často vyžaduje novou kopii zdrojového adresáře, což může být u velkého projektu časově náročné. Větvení v Gitu je jednoduché, protože operace vytvoření větve je okamžitá a přepínaní mezi větvemi je obvykle rychlé. Na rozdíl od mnoha jiných systémů, Git podporuje pracovní postup, kde dochází k větvení a slučování větví. Na obrázku č. 5 je vidět schéma, jak vypadá větvení v Git. [27][28]

Obrázek 5: Větvení v Gitu [55]

3.3.7 Pipeline

Funkční CI/CD pipeline znamená, že se aplikace může v reálním čase aktualizovat ze zdrojového kódu. Vývojář nemusí ručně nasazovat aplikaci, proces probíhá automaticky. V případě, že se v buildu objeví chyba, aplikace se dále nekompiluje a zodpovědná osoba dostane e-mail od GitHubu s ohlášením chyby.
Nasazování se aktualizuje nepřetržitě, proto konečný uživatel dostává nejnovější funkcionalitu produktu okamžitě. Existuje velký počet poskytovatelů konfigurace CI/CD pipeline, jako například GitHub Actions, CircleCI, TravisCI a Jenkins.
Pro úspěšnou konfiguraci pipeline je nutné mít workflow, ve kterém bude popsána řada kroků, které je třeba spustit nebo udělat, pokud dojde k nějaké spouštěcí události. [29][42]

3.4 Nástroje pro DevOps

Mezi nejpopulárnější moderní nástroje pro DevOps patří Jenkins, Bamboo, Nagios, Splunk, Maven, Selenium a další. V rámci této bakalářské práce budou využity následující nástroje: Docker, SonarQube, Git, Microsoft Azure DevOps a Red Hat OpenShift.

3.4.1 Microsoft Azure DevOps

Microsoft Azure DevOps je služba cloud computingu vytvořená společností Microsoft pro budování, testování, zavádění a správu aplikací a služeb prostřednictvím datových center, spravovaných společností Microsoft. Poskytuje software jako službu (SaaS – software as a service), platformu jako službu (PaaS – platform as a service) a infrastrukturu jako službu (IaaS – Infrastructure as a service). Microsoft Azure podporuje mnoho různých programovacích jazyků, nástrojů a frameworků, včetně softwaru a systémů přímo od Microsoftu a od třetích stran. [30]

3.4.1.2 Azure Pipelines

Azure Pipelines automaticky vytváří a testuje kód a zajišťuje možnost následného zpracování a publikaci zdrojového souboru. Podporuje téměř všechny programovací jazyky a typy projektů. Azure Pipelines kombinuje nepřetržitou integraci (CI) a nepřetržité dodání (CD) pro neustálé a důsledné testování a sestavení kódu a jeho následné využití. [30]
Pipeline může být nakonfigurován pomocí jazyku YAML nebo jiným vizuálním průvodcem. Společnost Microsoft dává přednost YAML, protože konfigurací kanálu vznikne další soubor, který bude existovat vedle zdrojového kódu, kde jej lze spravovat podle výběru

správy prostředků, což může automatizovat proces testování a sestavení. Spuštění sestavení ze závazku může výrazně urychlit vývojový proces. Vytvoření kanálu v YAML je dostatečně snadné pomocí azure-pipelines.yml. [29]

Příklad je uveden na obrázku č. 6.

```yaml
pool:
  name: Build
  demands: java

variables:
  ContainerRepository: 'projects-openshift-weather-proxy'

steps:
- task: DeleteFiles@1
  displayName: 'Delete configurations'
  inputs:
    SourceFolder: source/src/main/resources
    Contents: |
     application.properties
     application.yaml
     application-*.properties
     application-*.yaml

- task: sonarsource.sonarqube.15B84CA1-B62F-4A2A-A403-89B77A063157.SonarQubePrepare@4
  displayName: 'Prepare analysis on SonarQube'
  inputs:
    SonarQube: SQ
    scannerMode: CLI
    configMode: manual
    cliProjectKey: 'Openshift_weather-proxy_web'
    cliProjectName: 'weather-proxy'
    cliSources: source/app
    extraProperties: |



- task: Docker@1
  displayName: 'Build an image'
  inputs:
    containerregistrytype: 'Container Registry'
    dockerRegistryEndpoint: 'shared-registry'
    dockerFile: source/app/Dockerfile
    imageName: '$(ContainerRepository)/$(Build.Repository.Name):$(Build.BuildNumber)'

- task: Docker@1
  displayName: 'Push an image'
  inputs:
    containerregistrytype: 'Container Registry'
    dockerRegistryEndpoint: 'shared-registry'
    command: 'Push an image'
    imageName: '$(ContainerRepository)/$(Build.Repository.Name):$(Build.BuildNumber)'
```

Obrázek 6: Příklad souboru azure-pipelines.yml [56]

Výchozí konfigurační soubor Azure Pipelines zpracovává pouze základní úlohy a je potřeba soubor vyladit podle aplikace a cílového prostředí.

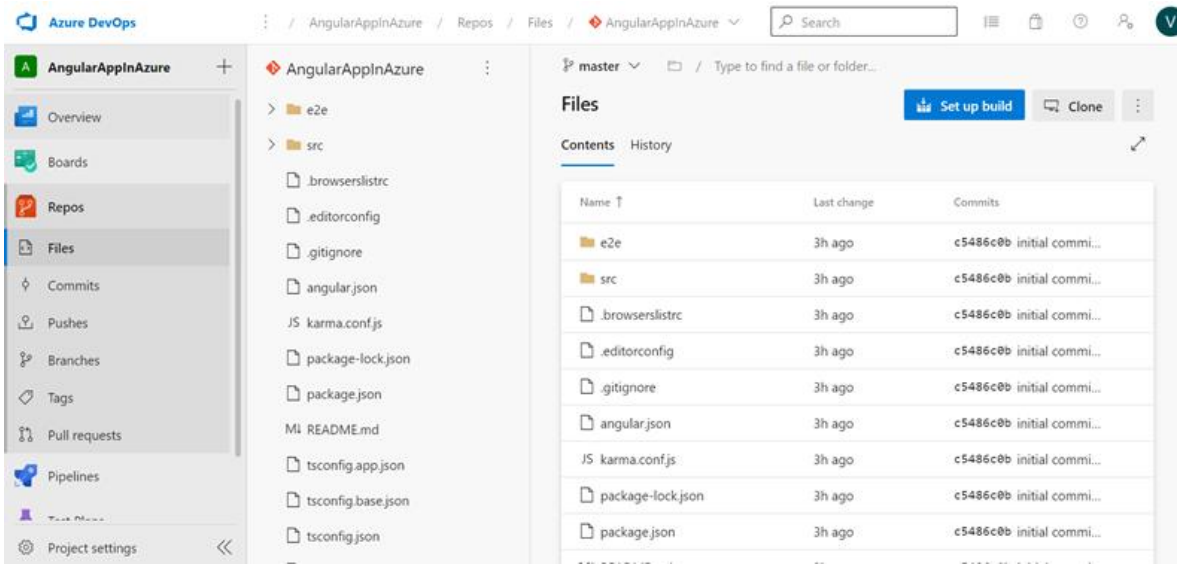
3.4.1.2 Azure Artifacts

Azure Artifacts je řešení pro správu balíku integrovaného do Azure DevOps, které umožňuje vytvoření a sdílení balíku Maven, NPM a NuGet prostřednictvím kanálů, které mohou být veřejné i soukromé, pokud jde o organizaci týmů jakékoliv velikosti. Azure Artifacts také umožňuje zahrnout protiproudový kanál do nakonfigurovaných kanálů balíčku, což umožní ukládat do cache paměti balíčky, kde jsou aplikace závislé na vlastních kanálech. To umožňuje další příjem závislosti na balíčcích, které jsou použity aplikací tehdy, když již nejsou k dispozici z původního zdrojového kanálu. Navíc Azure Artifacts umožňuje vkládat další artefakty do zdrojového kódu v tzv. univerzálních balíčcích, které lze dle potřeb dále přizpůsobit. [31]

3.4.1.3 Azure Repos

Azure Repos je sadou nástrojů pro správu verzí, které lze použít ke správě kódu. Jak bylo zmíněno v podkapitole 3.3.4, systém pro správu verzí pomáhá sledovat a koordinovat změny v kódu, provedené v rámci celého týmu [32].

Na obrázku č. 7 je znázorněn příklad prostředí Azure DevOps.



Obrázek 7: Azure Repos v aplikace Azure DevOps [57]

3.4.2 Red Hat OpenShift

Red Hat OpenShift je podniková kontejnerizační open source platforma. Softwarový produkt, který obsahuje komponenty projektů správy kontejnerů Kubernetes, a navíc zajišťuje lepší výkon a vyšší úroveň bezpečnosti. OpenShift je úspěšnou Enterprise Platform

as a service (PaaS) z pohledu vývojáře i velmi spolehlivou kontejnerizační službou (CaaS – Container as a service) z výrobního hlediska. OpenShift je založen na Kubernetes, který je open source kontejnerovým orchestračním projektem. Platforma pomáhá uživatelům spravovat clusterové skupiny hostitelů s linuxovými kontejnery, což jsou sady procesů, které obsahují vše potřebné ke spuštění v izolaci. [33]

3.5 Kubernetes

Název Kubernetes pochází z řečtiny, ze slova „pilot“. Google otevřel projekt Kubernetes v roce 2014. Kubernetes kombinuje více než 15 let zkušeností společnosti Google s výrobním zatížením v měřítku s nejlepšími nápady a postupy společnosti.

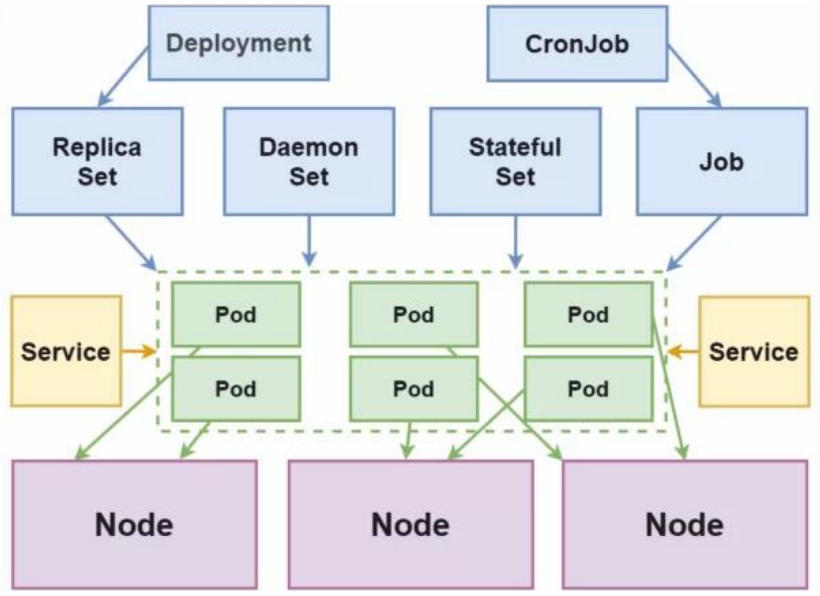
Kubernetes je přenosná, rozšiřitelná open source platforma pro správu kontejnerizovaného pracovního zatížení a služeb, která usnadňuje jak deklarativní konfiguraci, tak automatizaci. Má velký, rychle rostoucí ekosystém. Služby, podpora a nástroje Kubernetes jsou široce dostupné. Základem fungování Kubernetes je použití deklarativního přístupu. Vývojář je povinen uvést, čeho je potřeba dosáhnout, nikoli jak toho dosáhnout. Kromě toho lze v Kubernetes použít imperativní příkazy, které umožní přímo vytvářet, upravovat a mazat prostředky. K nasazení softwaru se používá základna kontejneru Linux, například Docker, Containerd nebo CRI-O, a popis toho, kolik kontejnerů bude požadováno a kolik prostředků bude potřeba. Samotné nasazení kontejnerů probíhá na základě pracovních uzlů (Node) – virtuálních nebo fyzických nástrojů.

Základní úkoly Kubernetes se skládají z následujícího:

- Nasazení kontejnerů a všechny operace potřebné ke spuštění požadované konfigurace. Patří mezi ně restartování zastavených kontejnerů, přesunutí k přidělení prostředků pro nové kontejnery a další operace.
- Škálování a běh většího množství kontejnerů současně na velkém počtu hostitelů.
- Vyrovnání více kontejnerů během spuštění. K tomu používá Kubernetes API, jehož úkolem je logicky seskupit kontejnery. To umožňuje definovat její kanály, nastavit její umístění a rovnoměrně rozložit zátěž. [34]

3.5.1 Základní komponenty Kubernetes

Kubernetes je užitečný nástroj pro orchestraci kontejneru. Toto řešení však nefunguje samo o sobě bez přípravy a dalšího nastavení. Na obrázku č. 8 jsou základní komponenty Kubernetes.



Obrázek 8: Komponenty Kubernetes [58]

3.5.1.1 Uzel (Node)

Node nebo uzel jsou virtuální nebo fyzické stroje, na kterých jsou kontejnery nasazeny a spuštěny. Kolekce uzlů tvoří cluster Kubernetes. První běžící uzel nebo hlavní uzel přímo ovládá cluster pomocí správce řadiče a plánovače. Je zodpovědný za rozhraní pro interakci s uživateli prostřednictvím serveru API a obsahuje úložiště „etcd“ s konfigurací clusteru, metadaty a stavu objektu. [34]

3.5.1.2 Namespace (Jmenný prostor)

Objekt určený k vymezení prostředku clusteru mezi týmy a projekty. Jmenný prostor znamená více virtuálních clusterů běžících na jednom fyzickém clusteru. [34]

3.5.1.2 Pod

Pod je objekt primárního nasazení a primární logická jednotka v Kubernetes. Pods jsou sadou jednoho nebo více kontejnerů pro společné nasazení v Node. Seskupování kontejnerů různých typů je vyžadováno, pokud jsou vzájemně závislé a musí běžet v jednom Node. To

umožní zejména zvýšit odezvu během interakce. Mohou to být například kontejnery, které ukládají webovou aplikaci a službu pro její ukládání do cache paměti. [34][43]

3.5.1.3 ReplicaSet (Sada replik)

Objekt zodpovědný za popis a řízení vícero instancí (replik) podů vytvořených v clusteru. Pokud je dostupná více než jedna replika, může dojít ke zlepšení odolnosti vůči chybám v škálovatelnosti aplikace. V praxi se ReplikaSet vytváří pomocí komponentů Deployment.[34]

3.5.1.4 Deployment (Nasazení)

Objekt, který ukládá popis Pods, počet replik a algoritmus pro jejich nahrazení v případě změny parametru. Řadič nasazení umožňuje deklarativní aktualizace objektů, jako jsou nody a sady replik. [34]

3.5.1.5 StatefulSet (Sada nastavení)

Stejně jako ostatní objekty (ReplicaSet nebo Deployment) umožňuje StatefulSet nastavit a spravovat jeden nebo více podů. Ale na rozdíl od nich má ID podů předvídatelné hodnoty, které přetrvávají po restartování. [34]

3.5.1.6 DaemonSet (Sada Daemon)

Objekt, který je zodpovědný za zajištění toho, že se na každém jednotlivém node (nebo na několika vybraných) spustí jedna instance vybraného podu. [34]

3.5.1.6 Job/CronJob (Úkoly / Naplánované úkoly)

Objekty pro úpravu jednorázového nebo pravidelného spuštění vybraných podů a sledování dokončení jejich práce. Řadič Job je zodpovědný za spuštění jedné úlohy, CronJob je zodpovědný za spuštění vícero úloh podle plánu. [34]

3.5.1.7 Label/Selector (Štítek / Selektor)

Značky se používají k označení prostředků. Umožňují zjednodušit skupinové manipulace. Selektory umožňují vybrat nebo filtrovat objekty na základě hodnoty štítků.
Štítky a selektory sice nejsou nezávislé objekty Kubernetes, ale bez nich nebude systém schopen plně fungovat. [34]

3.5.1.8 Servis

Servis je nástroj pro publikování aplikace jako síťové služby. Používá se mimo jiné k vyvážení provozu nebo zatížení mezi pody. [34]

3.5.1.9 Kubernetes Cluster

Kubernetes cluster je sada uzlových strojů, která zajišťuje běh kontejnerových aplikací. Spuštěním Kubernetes se spustí cluster. Cluster vždy obsahuje ovládací panel a jeden nebo více výpočetních strojů nebo uzlů. Ovládací panel je zodpovědný za udržování požadovaného stavu clusteru (např. jaké aplikace běží a jaké obrazy kontejneru se používají). Uzly skutečně spouštějí aplikace a pracovní zatížení. Cluster je jádrem klíčové výhody Kubernetes: dovoluje plánovat a provozovat kontejnery přes skupinu fyzických nebo virtuálních strojů jak v provozovnách, tak i v cloudu. Kubernetes kontejnery nejsou vázány na jednotlivé výpočetní stroje. [34][35]

3.5.1.10 Helm

Helm je package manager pro Kubernetes, který usnadňuje programátorům proces zabalení, konfigurace a rozmístění aplikace i servis přes Kubernetes cluster. Helm se používá k nasazení, konfiguraci nebo aktualizaci clusteru Kubernetes ve službě Azure Container Service, a to spuštěním příkazů Helm. Helm je nástroj, který zjednodušuje nasazení a správu aplikací Kubernetes pomocí formátu balení zvaného Charts. Helm pomůže kombinovat více manifestů Kubernetes (yaml), jako jsou služby, nasazení, konfigurační mapy a další, do jedné jednotky s názvem Helm Charts. Není potřeba vymýšlet ani používat nástroj pro tokenizaci nebo šablonování. [34][35][36]

3.6 Docker

„Docker je Open source projekt pro automatizaci nasazení aplikací jako přenosných a vlastních kontejnerů, které mohou běžet v cloudu nebo místně. Docker je také Společnost, která propaguje a vyvíjí tuto technologii a pracuje ve spolupráci s dodavateli cloudů, Linux a Windows, včetně Microsoftu.“ – Dokumentace Microsoft [40]

Hlavní úlohou Dockeru je usnadnění a zjednodušení vývoje aplikace. Docker vytváří izolované virtuální prostředí pro budování, zavádění a testování aplikace. V této kapitole budou definovány hlavní nástroje Dockeru, které zajišťují fungování celého systému.

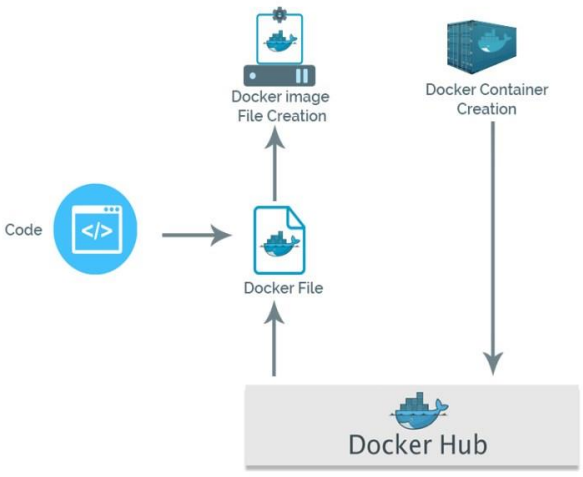
3.6.1 Dockerfile

Dockerfile je speciální seznam instrukcí (scriptů) definující vytvoření Docker obrazu. Soubor automaticky zpracuje veškeré příkazy ze seznamu a vytvoří se Docker obraz.

3.6.2 Docker obraz (Docker image)

Docker obraz (Docker image) je soubor, obsahující zdrojový kód, knihovny, závislosti, nástroje a další prvky potřebné ke spuštění aplikace. Slouží k představení aplikace a jejího virtuálního prostředí v určitém okamžiku. Umožňuje vývojářům testovat a upravovat kód ve stabilním prostředí za jednotných podmínek. Docker obraz je ve své podstatě šablonou, kterou lze použít jako základ pro sestavení kontejneru, který je běžícím obrazem.
Kontejnery Docker můžou běžet kdekoli, místně v datacentru zákazníka, v externím poskytovateli služeb nebo v cloudu v Azure. Mohou také běžet nativně v systémech Linux a Windows. Na rozdíl od VirtualBoxu jsou vyloučeny systémové náklady (overhead), potřebné pro virtualizaci zařízení. Docker slouží při vývoji a nasazení webových aplikací a služeb. Docker zajišťuje provoz služeb v pozadí, do Dockeru může být umístěna aplikace (dockerizing application), a lze také využít obrazy podobných kontejnerů k nasazení aplikace na produkci. [37][38][39]

V následujícím diagramu, který je na obrázku č. 9, je popsán zjednodušený postup Docker workflow.



Obrázek 9: Docker workflow [59]

3.6.3 Build Image

Příkaz *docker build* vytváří z dockerfile a „kontextu“ docker image (obraz). Kontextem se rozumí seznam souborů umístění, které definuje PATH nebo URL. Proces sestavení může odkazovat na libovolný soubor v kontextu. Sestavení může například použít přikaz COPY pro odkaz na soubor v kontextu. Parametr URL může odkazovat na tři druhy prostředků: Git repozitáře, předbalené tarball kontexty a jednoduché textové soubory. Když dojde k sestavení Dockerfile, vytvoří se Docker Image, který se nachází na lokálním serveru. [40]

3.7 SonarQube

SonarQube je open source platforma určená pro nepřetržitou analýzu a měření kvality kódu. Mezi hlavní možnosti, které SonarQube poskytuje, patří:

- Podpora jazyků Java, C, C++, C# Objective-C, Swift, PHP, JavaScript, Python a další.
- Poskytování reportů o duplicitě kódu, dodržování standardů kódování, pokrytí kódu modulárními testy, popř. chyby v kódu a další.
- Poskytování plně automatizované analýzy – tj. provádí nepřetržitě integraci s Maven, Ant, Gradle a jinými systémy. [44]

3.8 Python a jeho knihovny

3.8.1 Python

Python je vysokoúrovňový, dynamický, objektově orientovaný programovací jazyk, který se používá pro vývoj rozsáhlých webových aplikací, pro analýzu dat, testování a machine learning. Python je open source jazykem, což znamená, že je volně k použití, a to i pro komerční využití. Python může běžet na Macu, Windows i Unix systémech a byl také součástí Java a .NET virtuálních strojích. Standardní knihovna Pythonu obsahuje širokou škálu užitečných přenosných funkcí. K dispozici je velké množství balíčků, které po importu nabízejí inovativní řešení a možnosti nad rámec řešení úloh. Veškeré balíčky se nacházejí v otevřeném repozitáři, odkud je možné balíčky stáhnout a nainstalovat. [45]

3.8.2 BeautifulSoup

BeautifulSoup je Python knihovnou sloužící k extrahování dat z HTML a XML souborů. Podporuje jednoduché a přirozené způsoby navigace, vyhledávání a modifikace stromů

syntaktického rozboru. Ve většině případů pomůže programátorovi ušetřit hodiny, nebo i dny práce.[46]

3.8.3 Flask

Flask je framework pro vytváření webových aplikací v Pythonu. Patří do kategorie micro frameworku – minimálních koster webových aplikací, nabízejících jen základní možnosti. [47]

3.8.4 Scrapping

Jedná se o automatizovaný proces získání dat z webové stránky napsané v HTML. Princip scrapingu stojí na dvou pilířích: web crawler a web scraper.

- Web crawler, kterému se říká „pavouk“, je příkladem využití umělé inteligence, která prohlíží internet, zkoumá, indexuje a hledá obsah sledováním odkazů.
- Web scraper je specializovaný nástroj, určený k přesnému a rychlému extrahování dat z webové stránky. Důležitou součástí každého scraperu jsou lokátory dat (neboli selektory), které slouží k nalezení dat, která mají být extrahována ze souboru HTML – obvykle se aplikují xpath, css selektory, regex nebo jejich kombinace.[48]

3.9 Webová aplikace

Webová aplikace je software nebo program, který je přístupný z jakéhokoliv webového prohlížeče. Jeho front-end je obvykle vytvářen pomocí nástrojů HTML, CSS a jazyku Javascript, které jsou podporovány téměř všemi moderními webovými prohlížeči. Pro psaní back-endu je populární volbou mezi vývojáři Java, Python, PHP a Ruby.

Webové aplikace jsou navrženy tak, aby běžely na webových serverech (jako je Internetová informační služba nebo Apache) a jako uživatelské rozhraní používaly webové prohlížeče jako Microsoft Internet Explorer nebo Chrome. Webové aplikace jsou typicky klientské/serverové aplikace. Klientská část je v tomto pojetí to, co vidí uživatel v browseru, když načítá webovou stránku. Klient zapíná všechny potřebné prvky pro komunikaci se serverem. Serverovou částí je web server, který zpracovává dotazy klienta (browseru) a připravuje výsledek ve formátu webové stránky zpět klientovi.

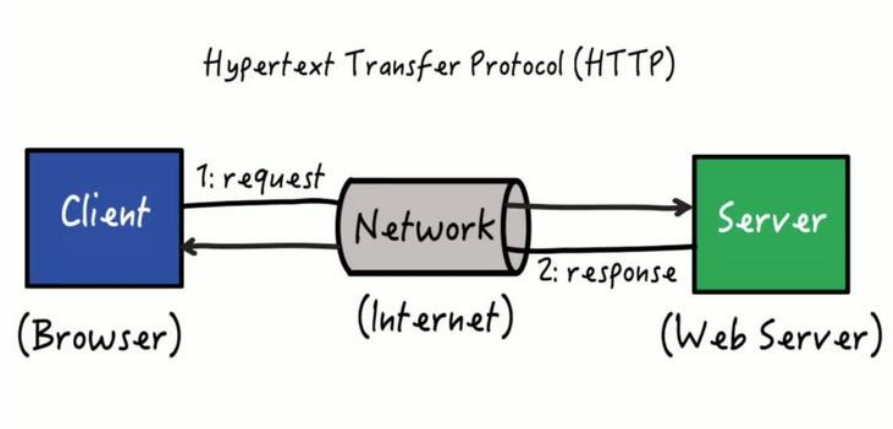
Princip fungování webových aplikací je následující:

- Uživatel přistupuje k webové aplikaci prostřednictvím webového prohlížeče nebo mobilní aplikace a spouští požadavek (request) na webový server přes internet.

- Webserver předá požadavek na webový aplikační server. Webový aplikační server provede požadovaný úkol – například dotaz na databázi nebo zpracování dat – a pak generuje výslednou odpověď na požadavek (response).
- Webový aplikační server odešle výsledky zpět na webserver.
- Webserver doručí klientovi požadované informace, které se mu zobrazí na displeji.

Základem pro komunikaci na webu je The HyperText Transfer Protokol (HTTP), který obsahuje request/response interakce.

Na obrázku č. 10 představena schéma HTTP, jak probíhají dotazy a odpovědí.



Obrázek 10: Request / Response v HTTP [60]

Kód odpovědi (stav) HTTP ukazuje, zda byl úspěšně proveden určitý HTTP požadavek. Kódy jsou seskupeny do 5 tříd:

- Informační 100–199
- Úspěšné 200–299
- Přesměrování 300–399
- Chyba na straně klienta 400–499
- Chyba na straně serveru 500–599 [49] [50]

4 Vlastní práce

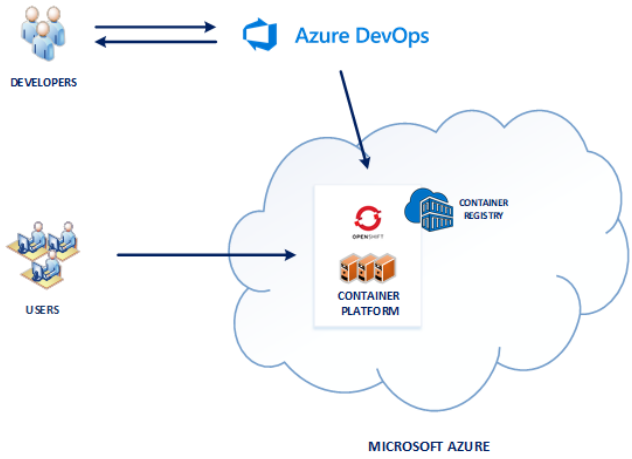
ŠKODA AUTO a.s. má ve svém vývojářském světě snahu rozvíjet nejlepší a nejefektivnější systém. Pracuje také s metodou DevOps. Tato část práce se zabývá návrhem, implementaci a ověřením automatizovaného procesu nasazení aplikace pomoci integrace CI/CD podle vnitřního stacku ŠKODA AUTO a.s.. Následně budou formulovány závěry práce, zhodnocení výzkumu a návrh postupu pro další možná zlepšení.

4.1 Analýza požadavku

ŠKODA AUTO a.s. dosud neměla technologické řešení v oblasti jazyku Python a jejího automatizovaného zpracování. Podle jednoho z požadavků bylo třeba navrhnout web aplikace v jazyce Python, který v sobě bude obsahovat Dockerfile, unit a smoke testy. Dalším krokem mělo být navržení stacku pomocí Azure DevOps, který zpracuje sadu zadání a ve výsledku vytvoří funkční Docker Container. Dále měl být pomocí platformy OpenShift vytvořen prostor pro budoucí web aplikace, která „půjde do světa“. Je zcela zásadní, aby obě platformy byly přizpůsobeny univerzálním Python aplikacím. Na konec má být přidán nástroj pro analýzu kódu, která bude analyzovat celou aplikaci, na jejímž základě dojde buď k doporučení ke schválení, nebo k upozornění na kritické momenty v kódu aplikace. Ve výsledku musí web aplikace úspěšně projít všechny operace a musí být úspěšně uložen na produkční server.

4.2 Popis technologického stacku

Na obrázku č. 11 je představen schematický proces toho, jak by měl vypadat technologický stack podle požadavku ŠKODA AUTO a.s..



Obrázek 11: Schéma technologického stacku [61]

Daný stack je sestaven z následujících částí:

- Vývoj
- Azure DevOps
- OpenShift
- Uživatelé

4.2.1 Vývoj

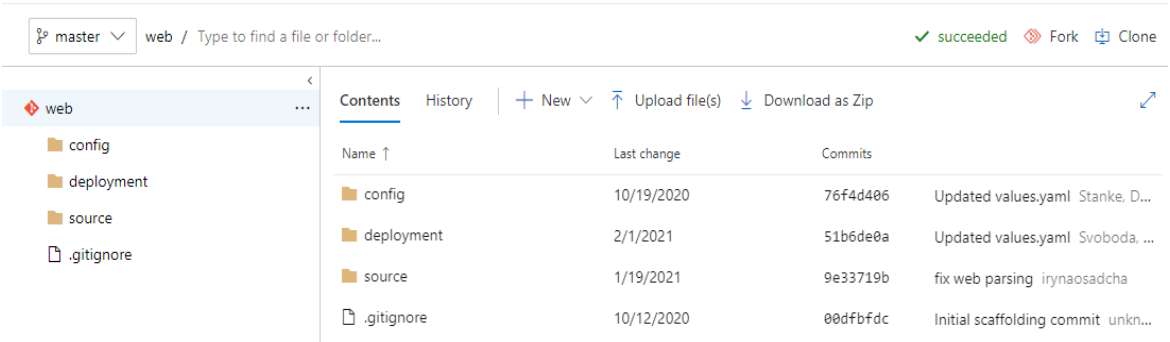
V této části stacku mluvíme o vývoji aplikace. Je to počáteční bod v hierarchii stacku, od kterého začne fungovat CI/CD. Byla zvolena jednoduchá webová aplikace, jejímž hlavním cílem je provést tzv. extrakci dat z webu – scraping, a ve výstupu potřebná data vygenerovat v podobě HTML stránky. Pro účely bakalářské práce byla aplikace napsána v jazyce Python s využitím knihoven Flask a Beautiful Soap. Dalším krokem byla potřeba integrovat do aplikace Dockerfile, který v budoucnu umožní udělat kontejner obrazu z této aplikace. Na obrázku č. 12 je ukázka nastavení Dockerfile, s propojeným unit a smoke testem. Do unit testu byl přidán fake soubor, vytvářející emulace dat, která by aplikace měla dostat. Tím se ověřuje, že logika této aplikace byla správně implementována. Smoke test umožňuje v případě zadání kontrolovat proces spuštění webové aplikace. Ve chvíli, kdy přijde odpověď HTTP Response Status Code 200 z lokálního serveru Docker Image, test proběhne úspěšně. Pokud se však jako odpověď vrátí Status Code 400, test selhal.

```
FROM python:3.8-buster

RUN mkdir /app
WORKDIR /app
ADD . /app/
RUN pip install -r requirements.txt


EXPOSE 8080
CMD ["python", "app.py"]

#Start unit test
RUN python -m unittest unit_test.py

#Start smoke test
RUN python -m unittest smoke_test.py
```

Obrázek 12: Konfigurace Dockerfile [62]

4.2.2 Nastavení Azure DevOps

Podle návrhu vystoupí tato platforma v roli továrny, která přijme aplikaci, zpracuje ji a pokusí se vypustit tu aplikaci „ven do světa“. Příjmem aplikace se rozumí nastavení systému správce verzí (Git), který obsahuje zdrojové kódy aplikace. Na obrázku č. 13 je zobrazen systém správce verzí, který je součástí Azure DevOps.



Obrázek 13: Systém správce verzí (Git) [63]

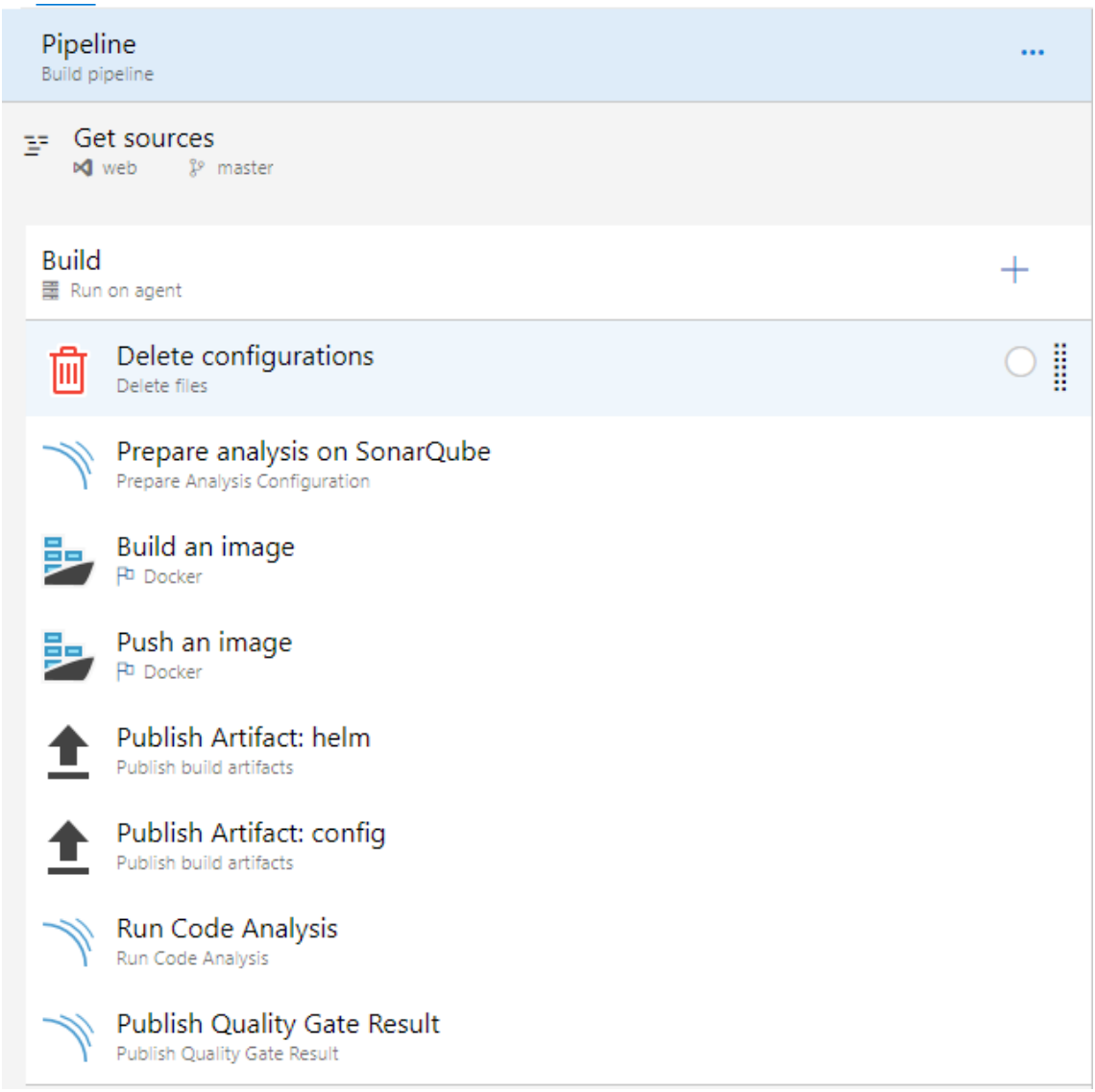
4.2.3 Build pipeline

Po doručení zdrojového kódu do Gitu, je dalším krokem, který se očekává, zpracování aplikace, představené v podobě „Build Pipeline“, která obsahuje množství nástrojů pro přípravu automatického testovaní, analýzy, kontejnerizace atd. Buildovací pipeline je entitou, jejímž prostřednictvím je definováno automatizované sestavení pipeline. Obsahuje

sadu zadání (tasks), z nichž se každý účastní procesu sestavení. Katalog umisťuje velké množství zadání, kterými lze začít. V praxi pipeline funguje tak, že Azure převezme v momentě příchozí notifikace z Gitu zdrojový kód, a řekne Agentu (virtuální prostředí, které nachází na serveru), aby ho na základě pipeline zadání zpracoval. Důležitá připomínka je, aby webová aplikace obsahovala sadu konfiguračních souborů, které popisují chovaní celé pipeline. Na obrázku č. 14 je vidět realizace Build Pipeline podle zadaného stacku:

- Delete configurations – odstranění předchozí konfigurace ze serveru.
- Prepare analysis on SonarQube – inicializace nástrojů SonarQube pro analýzu kódu.
- Build Image – zpracování aplikace podle konfigurace Dockerfile (zároveň budou spuštěny testy).
- Push Image – publikace Docker Image na Docker Hub
- Publish Artifact: helm – sestavení Artifact pro cluster Kubernetes
- Publish Artifact: config – přidání ke clusteru Kubernetes konfigurační závislosti
- Run Code Analysis – spuštění analýzy SonarQube
- Publish Quality Gate Result – publikace analýzy na server SonarQube

Každé zadání pipeline se zpracovává postupně a zvlášť, s tím, že pokud nějaký prvek vrátí chybu, tak se proces sestavení přeruší. V tomto případě se proces nachází pod přísnou kontrolou, a zajišťuje, že se sestava obsahující chybu nedostane do produkčního prostředí. Příkladem je smoke test, který byl úspěšné implementován do webové aplikace (viz obrázek č. 15). Ve finále se pipeline, sestavená Azure, ukládá jako Artifact.



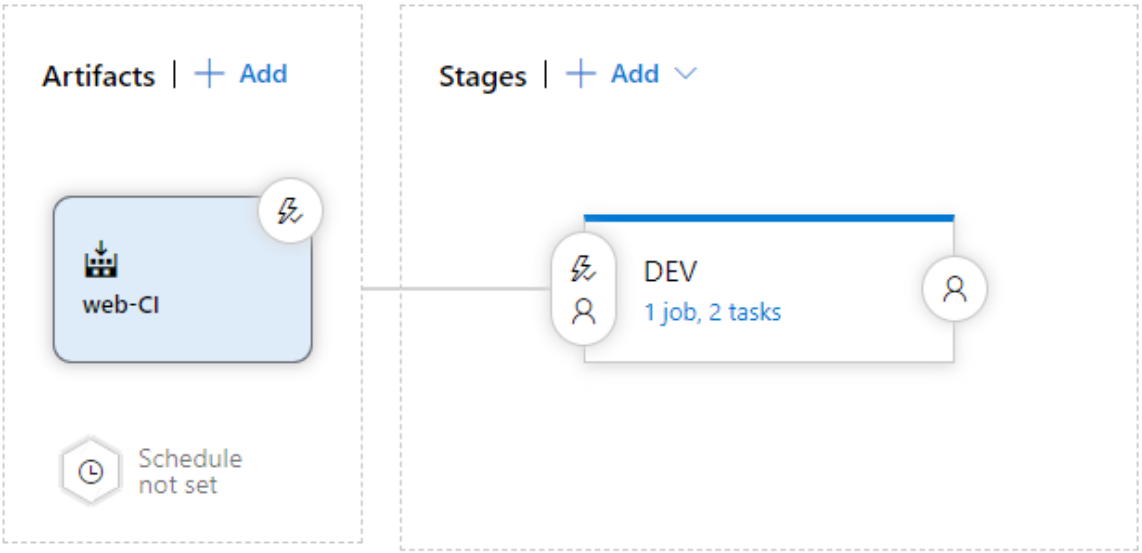
Obrázek 14: Build pipeline pro webovou aplikaci v Pythonu [64]

```
======================================================================
FAIL: test_main_page (test.BasicTests)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "/app/test.py", line 18, in test_main_page
    self.assertEqual(response.status_code, 300)
AssertionError: 200 != 300

----------------------------------------------------------------------
Ran 1 test in 0.486s

FAILED (failures=1)
```

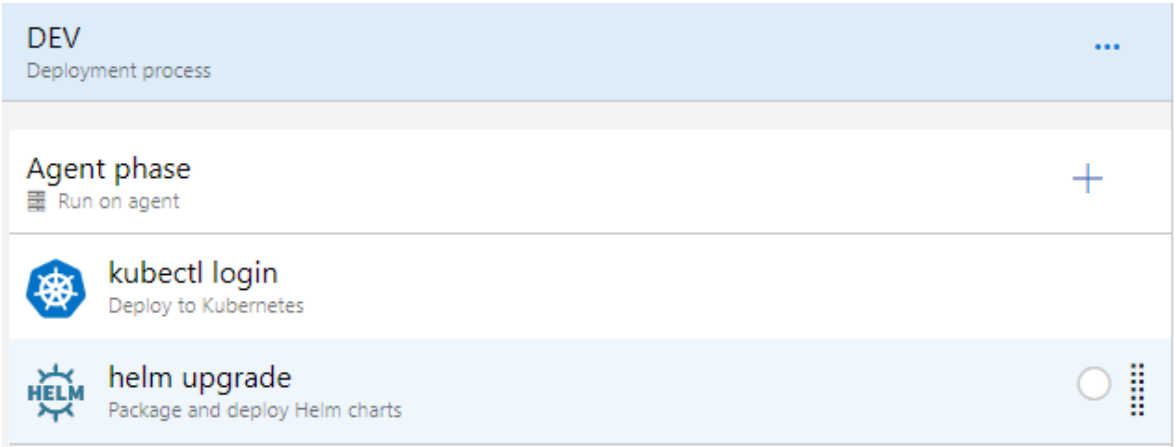
Obrázek 15: Výpis chyby z logu, který říká, že smoke test selhal [65]

4.2.4 Release pipeline

Po zpracování Build pipeline přejde Azure k release (zveřejnění, publikaci). Musí pro to být založen nový typ pipeline – tzv. Release Pipeline. Znamená to, že se vezme commitnuty kód a pošle se do produkce. Podle požadavku musí být pipeline nastavena tak, aby Artifact, který vystupuje jako trigger z Build pipeline převzal ze serveru poslední funkční Docker Image a poslal ho do produkčního prostředí, kde se ho pak pokusí rozložit. Schematicky je to vidět na obrázku č. 16. Jakmile přijde Artifact, tak se rovnou spustí konfigurační zadání, které zpracuje další Agent. Na obrázku č. 17 je představena funkční konfigurace pro Release pipeline.



Obrázek 16: Schéma struktury Release Pipeline [66]



Obrázek 17: Výchozí konfigurace pro Release Pipeline [67]

Konfigurace Release agentu sestavena ze dvou částí:

- Deploy to Kubernetes
- Package and deploy Helm charts

4.2.3.1 Deploy to Kubernetes

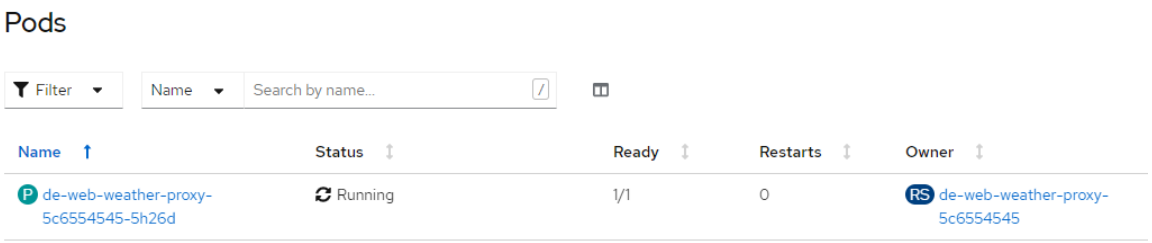
Azure Kubernetes Service spravuje hostované prostředí Kubernetes, což zrychluje a usnadňuje nasazení a správu kontejnerových aplikací. Tato služba také eliminuje zátěž probíhajícího provozu a údržby zajišťováním, upgradováním a škálováním zdrojů na vyžádání, aniž by bylo nutné přepnout aplikace do off-line módu.
Na základě vygenerovaného **deployment.yaml**, který se nachází v projektu jako konfigurační soubor, se Artifact odešle na cluster Kubernetes, který se potom zpracuje OpenShiftem. Hodně závislostí z deployment.yaml přebírá OpenShift a přidává nějaké svoje další objekty. V **deployment.yaml** se bude nacházet základní nastavení pro budoucí Release produktu, jakožto základní port, na kterém se spustí finální verze v produkci nebo metadata.

4.2.3.2 Package and deploy Helm charts

Smyslem této úlohy je nasazení, konfigurace nebo aktualizace clusteru Kubernetes ve službě Azure Container Service spuštěním příkazů Helm. Helm pomůže kombinovat více manifestů Kubernetes (yaml), jako jsou služby, nasazení, konfigurační mapy a další, do jedné jednotky s názvem Helm Charts. Není potřeba vymýšlet ani používat nástroj pro tokenizaci nebo šablonování. Ve výsledku bude tato úloha očekávat pozitivní výstup, který bude znamenat, že nasazení proběhlo úspěšně.

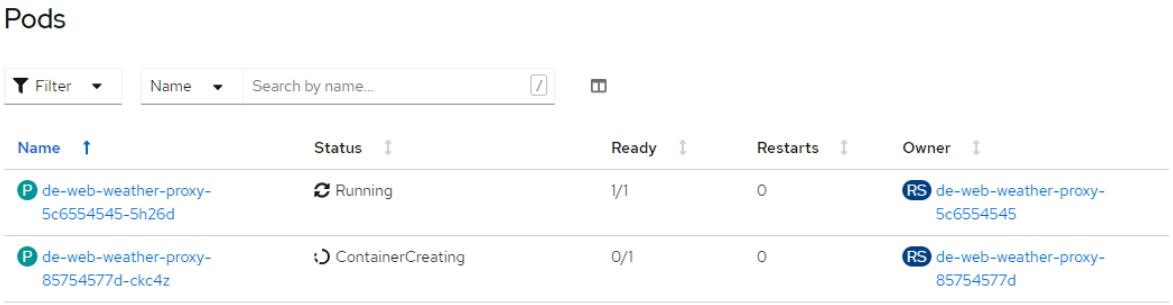
4.3 Ověření – výstup z OpenShift

Předposledního kroku požadavku se účastní platforma Red Hat OpenShift, která je zodpovědná za Continuous Delivery. Ve chvíli, kdy z Azure Release Pipeline proběhne příprava clusteru Kubernetes, se pošle signál na OpenShift s tím, že je založen Pod, ve kterém začne rozkládání Docker Image. V momentě založení Podu dojde k použití templates pro výchozí aplikace. Templates se řídí jazykem Helm, na jehož základě se vytvoří infrastruktura. Na obrázku č. 18 je vidět, jak v dané webové aplikaci běží funkční infrastruktura.



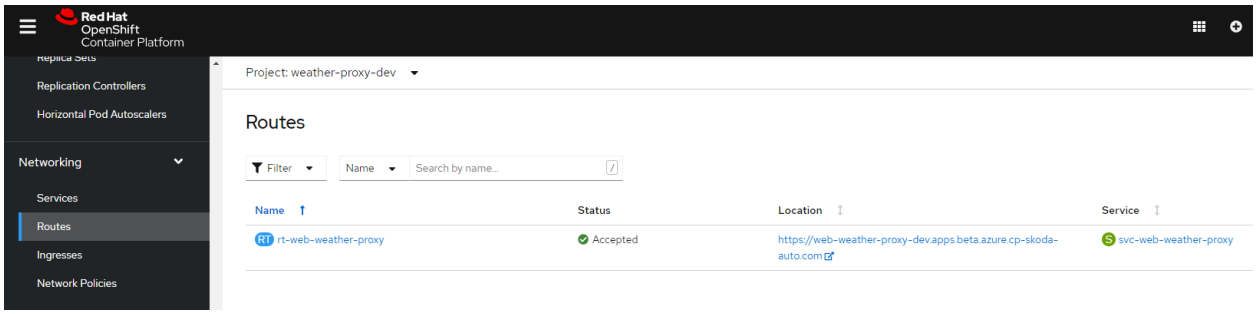
Obrázek 18: Pod na produkčním serveru pro zadanou webovou aplikaci [68]

V momentě updatu kódu nebo přidání dalšího funkcionálu se proces automaticky spustí od začátku daného stacku a OpenShift začne proces nahrazení starého produkčního Podu na nový. Na obrázku č. 19 je vidět, že se zakládá nový Pod, ve kterém dochází ke snaze spuštění další aplikace. Ve výsledku je po úspěšném spuštění nahrazen starý Pod novým, který začne běžet na produkčním serveru.



Obrázek 19: Založení nového Podu na produkčním serveru [69]

Jestliže je ve finále výsledek kladný, OpenShift vrátí na Azure Release Pipeline odpověď, že nasazení proběhlo úspěšně. Na obrázku č. 20 je odkaz na webovou aplikaci, po jejímž otevření se lze podívat na aplikaci, která běží na produkčním prostředí (viz obrázek č. 21).



Obrázek 20: Přidělená cesta webové aplikace [70]

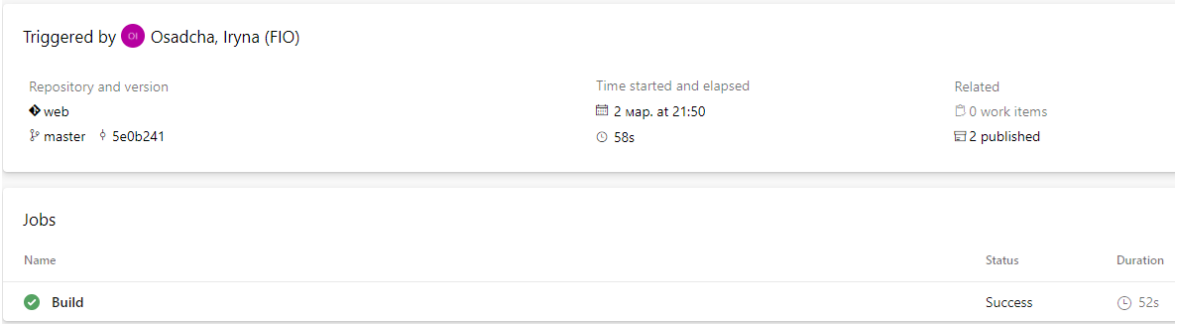
Obrázek 21: Webová aplikace Počasí, rozložena v produkčním prostředí [71]

5 Zhodnocení a doporučení

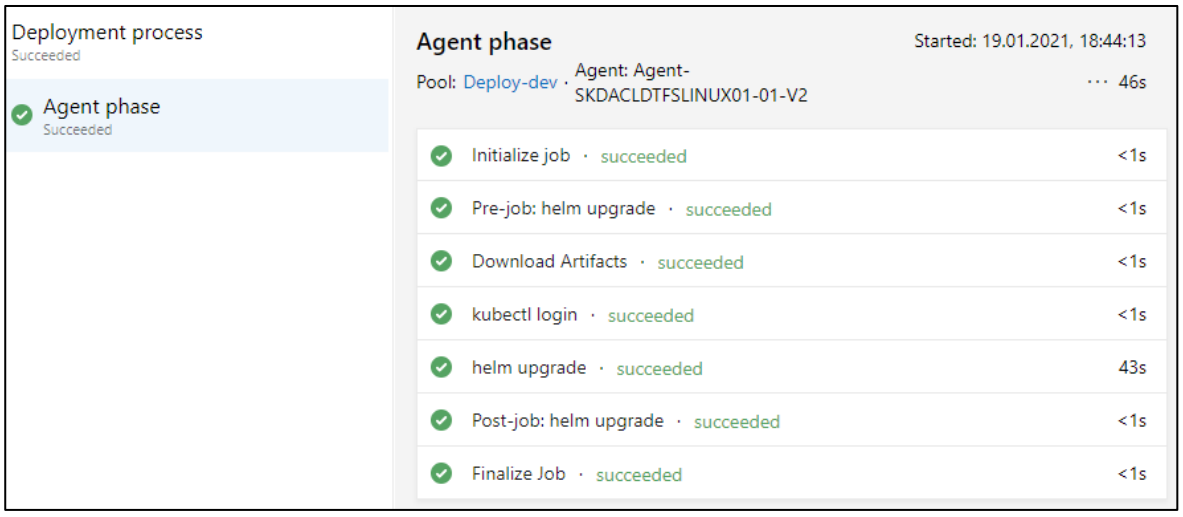
V této kapitole bakalářské práce jsou uvedné zhodnocení výsledků a základní doporučení pro budoucí vývoj daného technologického stacku.

5.1 Zhodnocení výsledků

Ve výsledku celý proces sestavy Build a Release Pipeline trvá v průměru 1-2 minuty. Konkrétní čas v každém případě je závislý na úrovní přetíženosti prostředí. Automatizace daného procesu optimalizuje dobu vykonávání úlohy a zajišťuje to, že na produkčním prostředí bude nasazená verze bez chyb.
Na obrazcích č. 22 a č. 23 představen výsledek v Azure DevOps.



Obrázek 22: Úspěšně sestaveny Build Pipeline za 52 sekundy [72]



Obrázek 23: Úspěšně sestaveny Release Pipeline za 46 sekund [73]

5.2 Doporučení ke zlepšení

5.2.1 Optimalizace testování

CI/CD do značné míry spoléhá na automatizované testování, tj. důvěřuje v kvalitu vyvíjeného softwaru. To však neznamená, že musejí být otestovány všechny představitelné scénáře. Cílem CI/CD je poskytnutí rychlé zpětné vazby a dodání softwaru uživatelům co nejrychleji (rychleji než tradiční metody). Ve stručnosti to znamená, že mezi pokrytím testu a výkonem musí vzniknout rovnováha. Pokud testování trvá příliš dlouho, lidé hledají způsob, jak ten postup obejít. Unit testy obvykle proběhnou jako první. Mohou poskytovat široké pokrytí a mohou upozornit na zjevné problémy se změnami v kódu.
Dále je možné investovat do složitějších, automatizovaných testů, jako je grafické uživatelské rozhraní nebo testy výkonu a zátěže. Všechny tyto typy testů, ať už automatizované, nebo manuální, jsou časově náročnější. Je však také třeba věnovat pozornost tomu, co představuje největší riziko pro produkt a uživatele.

5.2.2 Izolace a ochrana prostředí CI/CD

Z hlediska provozního zabezpečení lze systém CI/CD brát jako nejdůležitější infrastrukturu pro ochranu. Díky tomu, že má systém CI/CD plný přístup k databázi, kódu i účtům, které slouží rozložení v různých prostředích, je velmi důležité chránit interní data a integritu softwaru nebo produktu. Vzhledem k vysokému významu systému CI/CD je riziko útoku poměrně vysoké, a proto je velmi důležité jej co nejlépe izolovat a chránit. Systém by měl být nasazen na interních zabezpečených sítích bez externího přístupu. Doporučuje se nakonfigurovat VPN a další technologie řízení přístupu k síti tak, aby do systému měli přístup pouze ověření operátoři. Pokud sítě nejsou řádně zabezpečeny nebo izolovány, mohou útočníci s přístupem k jednomu prostředí tento přístup využít ke zneužití interní sítě a k získání přístupu k dalším serverům prostřednictvím slabých míst na serverech CI/CD.

5.3.3 Zabezpečeni kódu

Rostoucí distribuce otevřeného zdroje kódu v dnešní době zvyšuje zranitelnost komerčního softwaru. V daném stacku je silně doporučována integrace Black Duck, který má v sobě komplexní řešení pro správu rizik zabezpečení, a dodržování licence a kvality kódu, které vyplývají z použití open source kódu v aplikacích a kontejnerech. [74]

6 Závěr

Teoretická část práce obsahuje definice nezbytné k pochopení problematiky pojmů jako DevOps, životní cyklus aplikace a jeho prvky. Kapitola zabývající se popisem technologií Continuous Integration, Continuous Delivery, Continuous Deployment a kontejnerizace seznámila čtenáře se základními principy fungování automatického nasazení aplikace. Také byla prozkoumána problematika testování softwaru. Byl popsán programovací jazyk Python a vlastní knihovny použité ve zpracování aplikace. Nezbytnou součástí teoretické části se stal obecný popis fungování webové aplikace.

Cílem této bakalářské práce bylo navrhnout proces automatického nasazování aplikace CI/CD v testovacím prostředí ŠKODA AUTO a.s.. Nejprve byla zrealizována webová aplikace v jazyce Python, která byla následně integrována do systému Microsoft Azure DevOps. Připravené řešení by se mělo stát základem pro budoucí automatizaci nasazení větších projektů v Pythonu.

V souladu s dílčími cíli práce byla vytvořena CI/CD pipeline a připojená automatická analýza kódu za pomoci nástroje SonarQube, která na začátku sestavení každého buildu analyzuje kvalitu kódu a navrhuje optimalizaci a zlepšení. Následně byly napsané Unit a Smoke testy.

V průběhu zpracování bakalářské práce bylo zjištěno, že k technologickému stacku není připojena služba Black Duck, takže bylo stanoveno, že tento nástroj zabezpečení a kvality kódu nebude použit.

Bylo zjištěno, že požadavek ŠKODA AUTO a.s. bude splněn za předpokladu správných nastavení CI/CD a včasného zapojení každé potřebné služby kde probíhá úspěšná náhrada starší verze kódu za novou, a na obrazovce vývojáře se načítá výsledek. Při tom je jistota že v produkčním prostředí se následně neobjeví chyby. Pokud naopak kód chyby obsahuje, napojená Continuous Integration Pipeline nedovolí pustit aktuální verzi kódu do produkce a vývojář dostane upozornění. Nakonec bylo provedeno zhodnocení práce a byla nabídnuta sada návrhu na zlepšení stávajícího procesu.

7 Seznam použitých zdrojů

[1] Software Development Lifecycle (SDLC): What is a Software Development Lifecycle? [online]. [cit. 2021-03-07].
Dostupné z: https://www.veracode.com/security/software-development-lifecycle.

[2] Mezak, Steve. The Origins of DevOps: What's in a Name? [online]. [cit. 2021-03-07].
Dostupné z: https://devops.com/the-origins-of-devops-whats-in-a-name/.

[3] DevOps Market Size Worth $12.85 Billion by 2025 | CAGR: 18.60% [online]. [cit. 2021-03-07].
Dostupné z: https://www.grandviewresearch.com/press-release/global-development-to-operations-devops-market.

[4] ISO/IEC/IEEE 12207:2017: Systems and software engineering — Software life cycle processes [online]. [cit. 2021-03-07].
Dostupné z: https://www.iso.org/standard/63712.html.

[5] Co byste měli vědět o životním cyklu řízení projektu [online]. [cit. 2021-03-07]. Dostupné z: https://www.microsoft.com/cs-cz/microsoft-365/business-insights-ideas/resources/what-you-should-know-about-project-management-life-cycle.

[6] Software Testing Fundamentals [online]. [cit. 2021-03-07].
Dostupné z: https://softwaretestingfundamentals.com/.

[7] What is Software Testing [online]. [cit. 2021-03-07].
Dostupné z: https://sites.google.com/site/swtestingconcepts/home/what-is-software-testing.

[8] LENZ, Moritz. Python Continuous Integration and Delivery: A Concise Guide with Examples. Apress, 2019. ISBN 978-1484246894.
Dostupné také z: https://sites.google.com/site/swtestingconcepts/home/what-is-software-testing.

[9] Black Box Testing [online]. [cit. 2021-03-07].
Dostupné z: https://softwaretestingfundamentals.com/black-box-testing/

[10] White Box Testing [online]. [cit. 2021-03-07].
Dostupné z: https://softwaretestingfundamentals.com/white-box-testing/

[11] Gray Box Testing [online]. [cit. 2021-03-07].
Dostupné z: https://softwaretestingfundamentals.com/gray-box-testing/

[12] Unit Testing [online]. [cit. 2021-03-07].
Dostupné z: https://softwaretestingfundamentals.com/unit-testing/.

[13] Integration Testing [online]. [cit. 2021-03-07].
Dostupné z: https://softwaretestingfundamentals.com/integration-testing/.

[14] System Testing [online]. [cit. 2021-03-07].
Dostupné z: https://softwaretestingfundamentals.com/system-testing/.

[15] Smoke Testing [online]. [cit. 2021-03-07].
Dostupné z: https://softwaretestingfundamentals.com/smoke-testing/

[16] WALLEN, Jack. What is CI/CD? [online]. 08.10.2019.
Dostupné z: https://www.techrepublic.com/article/what-is-cicd/

[17] BASU, Soumyajit. Continuous Integration: Its History and Benefits [online]. 23.5.2017 [cit. 2021-03-07].
Dostupné z: https://dzone.com/articles/continuous-integration-and-its-whereabouts

[18] HENDRICKSON, Mike. Extreme Programming. Addison-Wesley Professional, 2000. ISBN 978-0201708424.

[19] What is CI/CD? [online].
Dostupné z: https://www.redhat.com/en/topics/devops/what-is-ci-cd#:~:text=CI%2FCD%20is%20a%20method,the%20stages%20of%20app%20development.&text=Specifically%2C%20CI%2FCD%20introduces%20ongoing,phases%20to%20delivery%20and%20deployment.

[20] Principles behind the Agile Manifesto [online]. 2001.
Dostupné z: https://agilemanifesto.org/iso/en/principles.html

[21] PITTET, Sten. Continuous integration vs. continuous delivery vs. continuous deployment [online].
Dostupné z: https://www.atlassian.com/continuous-delivery/principles/continuous-integration-vs-delivery-vs-deployment

[22] WATERMAN, Steven. GitHub is a free CI/CD/Hosting solution [online]. 24.02.2020.
Dostupné z: https://blog.scottlogic.com/2020/02/24/github-cd.html

[23] Continuous integration vs. continuous delivery vs. continuous deployment [online]. [cit. 2021-03-07].
Dostupné z: https://www.atlassian.com/continuous-delivery/principles/continuous-integration-vs-delivery-vs-deployment

[24] ROSSEL, Sander. Continuous Integration, Delivery, and Deployment.. Packt Publishing, 2017. ISBN 978-1787286610.

[25] DUVALL, Paul M. Continuous Integration: Improving Software Quality and Reducing Risk. Addison-Wesley Professional, 2007. ISBN 978-0321336385.

[26] IBM Cloud Education. Containerization [online]. 15.05.2019.
Dostupné z: https://www.ibm.com/cloud/learn/containerization

[27] Getting Started – What is Git? [online]. [cit. 2021-03-07].
Dostupné z: https://git-scm.com/book/en/v2/Getting-Started-What-is-Git%3F

[28] CHACON, Scott. Pro Git. Apress, 2014. ISBN 978-1484200773.

[29] Azure Pipelines documentation [online]. [cit. 2021-03-07].
Dostupné z: https://docs.microsoft.com/en-us/azure/devops/pipelines/?view=azure-devops

[30] Azure DevOps documentation [online]. [cit. 2021-03-07].
Dostupné z: https://docs.microsoft.com/en-us/azure/devops/?view=azure-devops

[31] What is Azure Artifacts? [online]. 2020 [cit. 2021-03-07].
Dostupné z: https://docs.microsoft.com/ru-ru/azure/devops/artifacts/overview?view=azure-devops

[32] What is Azure Repos? [online]. 2020 [cit. 2021-03-07].
Dostupné z: https://docs.microsoft.com/ru-ru/azure/devops/repos/get-started/what-is-repos?vie

[33] Interactive Learning Portal [online]. [cit. 2021-03-07].
Dostupné z: https://learn.openshift.com/

[34] Kubernetes Documentation [online]. [cit. 2021-03-07].
Dostupné z: https://kubernetes.io/docs/home/

[35] AMOS, David. Red Hat OpenShift vs. Kubernetes [online]. [cit. 2021-03-07].
Dostupné z: https://realpython.com/python-web-scraping-practical-introduction/

[36] What is Helm? [online]. [cit. 2021-03-07].
Dostupné z: https://helm.sh/

[37] LEZSKO, Rafal. Continuous Delivery with Docker and Jenkins. 2017. ISBN 978-1787125230.

[38] MOHAMMAD, Jalaluddeen. Why use docker? Introduction [online]. [cit. 2021-03-07].
Dostupné z: https://www.factweavers.com/blog/why-use-docker/

[39] What is a Container?: A standardized unit of software [online]. [cit. 2021-03-07].
Dostupné z: https://www.docker.com/resources/what-container

[40] What is Docker? [online]. [cit. 2021-03-07].
Dostupné z: https://docs.microsoft.com/en-us/dotnet/architecture/microservices/container-docker-introduction/docker-defined

[41] The complete DevOps platform [online]. [cit. 2021-03-07].
Dostupné z: https://about.gitlab.com/

[42] Creating CI/CD solutions for applications using OpenShift Pipelines [online]. [cit. 2021-03-07].
Dostupné z: https://docs.openshift.com/container-platform/4.5/pipelines/creating-applications-with-cicd-pipelines.html

[43] Create static Pods [online]. [cit. 2021-03-07].
Dostupné z: https://kubernetes.io/docs/tasks/configure-pod-container/static-pod/

[44] SonarQube Documentation [online]. [cit. 2021-03-07].
Dostupné z: https://docs.sonarqube.org/latest/

[45] Jazyk Python [online]. [cit. 2021-03-07].
Dostupné z: http://www.web2py.com/books/default/chapter/40/02/the-python-language

[46] BREUSS, Martin. Beautiful Soup: Build a Web Scraper With Python [online]. [cit. 2021-03-07].
Dostupné z: https://realpython.com/beautiful-soup-web-scraper-python/#author

[47] Flask Documentation [online]. [cit. 2021-03-07].
Dostupné z: https://flask.palletsprojects.com/en/1.1.x/

[48] AMOS, David. A Practical Introduction to Web Scraping in Python [online]. 17.08.2020 [cit. 2021-03-07].
Dostupné z: https://realpython.com/python-web-scraping-practical-introduction/

[49] Web Application [online]. [cit. 2021-03-07].
Dostupné z: https://www.pcmag.com/encyclopedia/term/web-application

[50] The Hypertext Transfer Protocol (HTTP) [online]. [cit. 2021-03-07].
Dostupné z: http://researchhubs.com/post/computing/web-application/the-hypertext-transfer-protocol-http.html

[51] Unit test v Pythonu [cit. 2021-03-13]. Zdroj: Vlastní zpracování

[52] Smoke test v Pythonu [cit. 2021-03-13]. Zdroj: Vlastní zpracování

[53] CI/CD schéma[online]. [cit. 2021-03-13]. Zdroj: ŠKODA AUTO a.s.

[54] CI proces [Online]. [cit. 2021-03-13].
Dostupné z: https://developers.redhat.com/blog/2017/09/06/continuous-integration-a-typical-process

[55] Větvení v Gitu [Online]. [cit. 2021-03-13].
Dostupné z: https://intellect.icu/varianty-ispolzovaniya-git-v-zhiznennom-tsikle-razrabotki-programmnogo-obespecheniya-9247

[56] Příklad souboru azure-pipelines.yml [cit. 2021-03-13]. Zdroj: Vlastní zpracování

[57] Azure Repos v aplikace Azure DevOps. [cit. 2021-03-13]. Zdroj: Vlastní zpracování

[58] Komponenty Kubernetes [Online]. [cit. 2021-03-13].
Dostupné z: https://eternalhost.net/blog/razrabotka/kubernetes-chto-eto

[59] Docker workflow [Online]. [cit. 2021-03-13].
Dostupné z: https://www.factweavers.com/blog/why-use-docker/

[60] Request / Response v HTTP [Online]. [cit. 2021-03-13].
Dostupné z: http://researchhubs.com/post/computing/web-application/the-hypertext-transfer-protocol-http.html

[61] Schéma technologického stacku [online]. [cit. 2021-03-13]. Zdroj: ŠKODA AUTO a.s.

[62] Konfigurace Dockerfile [cit. 2021-03-13]. Zdroj: Vlastní zpracování

[63] Systém správce verzi (Git) [cit. 2021-03-13]. Zdroj: Vlastní zpracování

[64] Build pipeline pro web aplikaci v Pythonu [cit. 2021-03-13]. Zdroj: Vlastní zpracování

[65] Výpis chyby z logu, který říká ze smoke test selhal. [cit. 2021-03-13]. Zdroj: Vlastní zpracování

[66] Schéma struktury Release Pipeline.[cit. 2021-03-13].Zdroj: Vlastní zpracování

[67] Výchozí konfigurace pro Release Pipeline .[cit. 2021-03-13].Zdroj: Vlastní zpracování

[68] Pod na produkčním serveru pro zadanou web aplikace. [cit. 2021-03-13]. Zdroj: Vlastní zpracování

[69] Založeni nového Podu na produkčním serveru. [cit. 2021-03-13]. Zdroj: Vlastní zpracování

[70] Přidělena cesta web aplikace [cit. 2021-03-13]. Zdroj: Vlastní zpracování

[71] Web aplikace Počasí rozložena na produkčním prostředí. [cit. 2021-03-13]. Zdroj: Vlastní zpracování

[72] Úspěšně sestaveny Build Pipeline za 52 sekundy. [cit. 2021-03-13]. Zdroj: Vlastní zpracování

[73] Úspěšně sestaveny Release Pipeline za 46 sekund. [cit. 2021-03-13]. Zdroj: Vlastní zpracování

[74] Black Duck by Synopsys: User Guide. In: Web Application [online]. Version 2021.2.0, s. 525 [cit. 2021-03-07].
Dostupné z: https://testing.blackduck.synopsys.com/doc/pdfs/user_guide.pdf