VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
FACULTY OF INFORMATION TECHNOLOGY

ÚSTAV INTELIGENTNÍCH SYSTÉMŮ
DEPARTMENT OF INTELLIGENT SYSTEMS

PODPORA KUBERNETES PRO QUARKUS QE TEST FRAMEWORK

KUBERNETES SUPPORT FOR QUARKUS QE TEST FRAMEWORK

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE **GEORGII TROITSKII**
AUTHOR

VEDOUCÍ PRÁCE **doc. Mgr. ADAM ROGALEWICZ, Ph.D.**
SUPERVISOR

BRNO 2024

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

Zadání bakalářské práce

154686

Ústav: Ústav inteligentních systémů (UITS)
Student: **Troitskii Georgii**
Program: Informační technologie
Název: **Podpora Kubernetes pro Quarkus QE Test Framework**
Kategorie: Analýza a testování softwaru
Akademický rok: 2023/24

Zadání:

1. Prostudujte Quarkus QE Test Framework určený pro testování aplikací nad frameworkem Quarkus.
2. Seznamte se s platformou Kubernetes a nástrojem Jenkins.
3. Navrhněte rozšíření Quarkus QE Test Frameworku o podporu testování aplikací v Kubernetes clusteru.
4. Navržené rozšíření implementujte a připravte na integraci do Quarkus QE Test Frameworku a to včetně propojení Jenkins a GitHub pomocí GitHub Pull Request Builder.
5. Demonstrujte funkčnost vašeho řešení a diskutujte možná rozšíření.

Literatura:

- Quarkus QE Test Framework: https://github.com/quarkus-qe/quarkus-test-framework
- Dokumentace projektu Kubernetes: https://kubernetes.io/docs/home/
- Dokumentace projektu Jenkins: https://www.jenkins.io/sigs/docs/

Při obhajobě semestrální části projektu je požadováno:
Body 1 a 2 zadání.

Podrobné závazné pokyny pro vypracování práce viz https://www.fit.vut.cz/study/theses/

Vedoucí práce: **Rogalewicz Adam, doc. Mgr., Ph.D.**
Konzultant: Michal Vavřík (RedHat)
Vedoucí ústavu: Hanáček Petr, doc. Dr. Ing.
Datum zadání: 1.11.2023
Termín pro odevzdání: 9.5.2024
Datum schválení: 6.11.2023

Abstrakt

Tato práce se zaměřuje na vylepšení open-source projektu Quarkus QE Test Framework, určeného pro testování aplikací postavených na frameworku Quarkus. Cílem je přidání podpory pro automatozovane spuštění testů na platformě Kubernetes. K dosažení tohoto cíle do virtuálního stroje na platformě OpenStack byl nainstalován cluster Kubernetes. Pomocí pluginu pro GitHub Pull Request Builder byla zřízena integrace mezi Jenkins a GitHub CI v Quarkus QE Test Frameworku. Tato integrace umožňuje automatické spouštění testů v Kubernetes při vytváření Pull Requestů na GitHubu po zadání specifické fráze do Pull Requestu. Toto řešení umožňuje plně automatizované spouštění testů v Kubernetes jako součást GitHub CI pipeline, čímž rozšiřuje seznam externich platform, na kterych tým Quarkus QE testuje aplikace Quarkus.

Abstract

This thesis focuses on improving the open-source Quarkus QE Test Framework project, designated for testing applications built on the Quarkus framework. The goal is to add support for automated test execution on the Kubernetes platform. To achieve this goal, a Kubernetes cluster was installed in a virtual machine on the OpenStack platform. Integration between Jenkins and GitHub CI in the Quarkus QE Test Framework was set up using GitHub Pull Request Builder plugin for Jenkins. This integration allows Kubernetes to automatically run tests when creating Pull Requests on GitHub after entering a specific trigger phrase in the Pull Request. This solution enables fully automated test execution in Kubernetes as part of the GitHub CI pipeline, expanding the list of external platforms on which Quarkus QE team tests Quarkus applications.

Klíčová slova

Quarkus, Quarkus QE Test Framework, Java, Kubernetes, K8s, cluster, kubectl, Docker, GitHub, CI, Jenkins, GHPRB, GitHub Pull Request Builder, trigger, job, pipeline

Keywords

Quarkus, Quarkus QE Test Framework, Java, Kubernetes, K8s, cluster, kubectl, Docker, GitHub, CI, Jenkins, GHPRB, GitHub Pull Request Builder, trigger, job, pipeline

Citace

TROITSKII, Georgii. *Podpora Kubernetes pro Quarkus QE Test Framework*. Brno, 2024. Bakalářská práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedoucí práce doc. Mgr. ADAM ROGALEWICZ, Ph.D.

Podpora Kubernetes pro Quarkus QE Test Framework

Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně pod vedením pana docenta Adama Rogalewicze. Další informace mi poskytl technický vedoucí z firmy Red Hat Michal Vavřík. Uvedl jsem všechny literární prameny, publikace a další zdroje, ze kterých jsem čerpal.

.....
Georgii Troitskii
8. května 2024

Poděkování

Chtěl bych poděkovat panu docentu Rogalewiczovi, který byl ochoten mou práci vést a mohl jsem s ním konzultovat obsah, prezentace i text samotné práce. Dále bych také chtěl poděkovat mému technickému vedoucímu Michalovi Vavříkovi za rady a nápady, které mi pomohly během vytváření a testování bakalářské práce. Také bych chtěl poděkovat svým přátelům, kteří mi pomohli s kontrolou technické zprávy a poskytli cennou zpětnou vazbu.

Obsah

Kapitola 1

Úvod

Tato bakalářská práce se zaměřuje na vylepšení open-source projektu *Quarkus QE Test Framework*, který je navržen pro testování aplikací postavených na frameworku Quarkus [21]. V současné fázi svého vývoje Quarkus QE Test Framework poskytuje širokou škálu možností pro testování aplikací v různých režimech zahrnujících JVM, nativní a vývojový mód na platformě OpenShift. Tato všestrannost umožňuje dosažení konzistentního chování s produkčním prostředím a usnadňuje proces nasazování, například pokud jde o kontejnerizaci databází a programovou aplikaci změn na nasazených aplikacích.

Cílem této práce je rozšířit tuto flexibilitu a funkcionalitu pro testování aplikací v prostředí cloud-native v prostředí Kubernetes, což umožní uživatelům efektivně testovat své aplikace v prostředí Kubernetes.

Instalace Kubernetes clusteru do virtuálního stroje na platformě OpenStack zajistí infrastrukturu co nejblíže té, která bude použita konečným uživatelem pro jeho vlastní aplikace ve frameworku Quarkus.

Dale práce integruje testovací pipeline Jenkins do GitHub Continuous Integration (CI) pro Quarkus QE Test Framework. Tato integrace umožní automatické spouštění testů v Kubernetes při otevírání tzv. Pull Requestů, přičemž aktivace této funkce bude prováděna prostřednictvím specifické fráze v Pull Requestu, což vyvolá automaticke spuštění Jenkins jobu. Celkový praktický výsledek práce zahrnuje pokrytí nezbytných scénářů pro testování aplikací napsaných ve frameworku Quarkus v prostředí cloud-native.

Kapitola 2

Quarkus projekt

2.1 Quarkus

[21] [25]
Quarkus je framework s otevřeným zdrojovým kódem pro vývoj aplikací v programovacím jazyce Java. Framework je optimalizovaný pro rychlejší spuštění aplikací a také, kvůli své nízké paměťové náročnosti a efektivitě využití zdrojů, je ideální pro vývoj cloud-native[1] aplikací v jazyce Java, běžících efektivně na platformách Kubernetes a Red Hat OpenShift.

2.2 Quarkus QE Test Framework

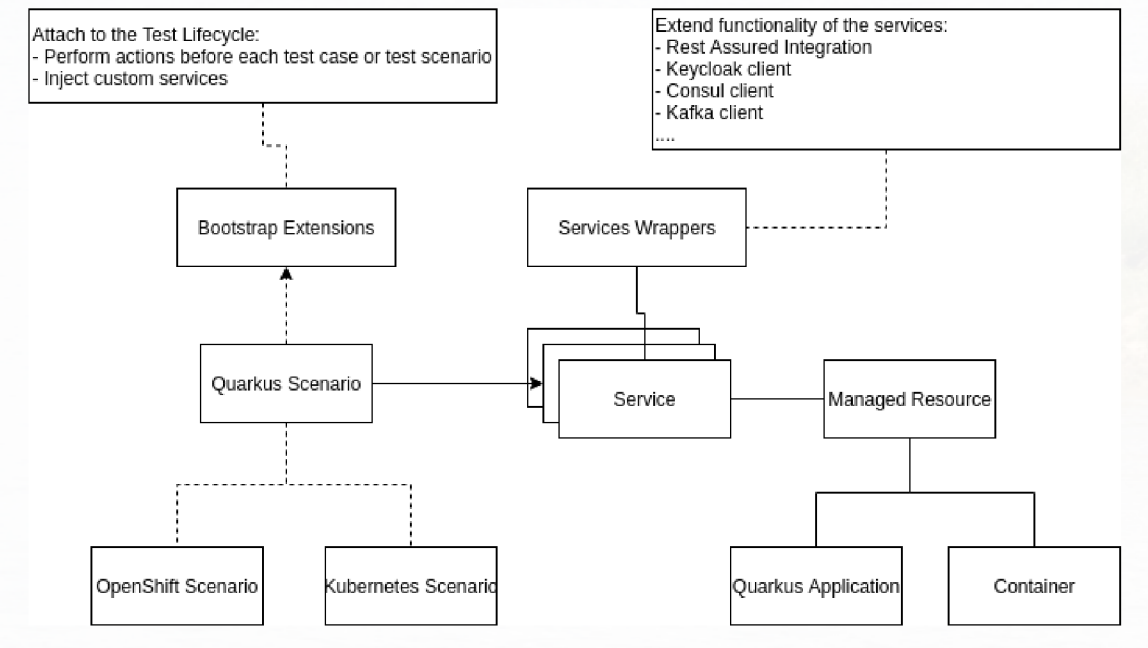
Při vývoji aplikace využívající framework Quarkus je potreba zajistit kvalitu a spolehlivost výsledného produktu. Testovací tým Quarkus QE[2] nabízí podporu v oblasti testování aplikací, vyvinutých pomocí frameworku Quarkus.

Quarkus QE Test Framework (QQE Framework) [24] je nástroj vyvinutý QE týmem za účelem všestranného testování Quarkus aplikací. Daný framework je postavený na principu Extension Model architektury, takže pro podporu dalších funkcí nebo nových možností nasazení do cloud prostředí je potřebá jen implementovat nový modul rozšíření.

Obrázek 2.1 znázorňuje architekturu Quarkus QE Test Frameworku:

- Scenario – určuje, kam budou nasazené služby (services), například `@QuarkusScenario` pro Bare-metal[3],
- Service – určuje, jakým způsobem budou testy interagovat se zdroji, nejčastěji se používá `RestService` pro volání REST endpointu,
- Managed Resource – určuje, co bude nasazeno jako zdroj, možnostmi jsou Java aplikace ve frameworku Quarkus `@QuarkusApplication` nebo Docker kontejner `@Container`.

[1] Aplikace navržené pro efektivní provoz v cloudovém prostředí, využívají kontejnerizaci a architekturu mikroslužeb
[2] QE – Quality Engineer (specialisté zabývající se kontrolou a zajištěním kvality softwarového produktu během všech cyklů vývoje)
[3] Bare-metal – softwarová aplikace běží přímo na fyzickém zařízení, bez použití virtuálních strojů



Obrázek 2.1: Architektura Quarkus QE Test Frameworku. Schéma byla převzata z [4]

Příkladem skutečného testu napsaného pomocí QQE Frameworku může být `PostgresqlDatabaseIT` (výpis 2.1). Jak je vidět podle `@QuarkusScenario` anotace, test je napsaný pro nasazení v bare-metal prostředí. Kdyby bylo potřeba spustit ten stejný test v OpenShift[4], stačí vytvořit novou třídu `OpenShiftPostgresqlIT`, která bude dědit existující test `PostgresqlDatabaseIT`. Anotace scénáře musí být změněna na `@OpenShiftScenario`.

Za spuštění testovacího scénáře je zodpovědná třída `QuarkusScenarioBootstrap`, která implementuje rozhraní JUnit Jupiter API [3]. Nejdůležitější vlastností je, že tato třída přepisuje metodu `beforeAll`, která je spuštěna ještě před testy. V ní se provádí inicializace testovacího scénáře, registrace rozšíření a spuštění služeb – kontejnerů a aplikace Quarkus. Po inicializaci a spuštění služeb dál probíhá vyhodnocení testů, označených anotaci `@Test`. Ve většině napsaných testů v Quarkus QE Test Frameworku se jedná o volání REST endpointu a kontrolování návratové hodnoty a očekávané odpovědi. Návod pro napsání aplikací ve frameworku Quarkus může být nalezen v oficiální dokumentaci projektu[5].

```
1  @QuarkusScenario
2  public class PostgresqlDatabaseIT {
3
4      private static final int POSTGRESQL_PORT = 5432;
5
6      @Container(image = "${postgresql.image}",
7              port = POSTGRESQL_PORT, expectedLog = "is ready")
8      static PostgresqlService database = new PostgresqlService();
9
10     @QuarkusApplication
11     static RestService app = new RestService()
12             .withProperty("quarkus.datasource.username", database.getUser())
```

[4]Red Hat OpenShift Container Platform – platforma pro správu kontejnerů založená na Kubernetes. Víc informace lze zjistit na https://www.redhat.com/en/technologies/cloud-computing/openshift

[5]Dokumentace Quarkus: https://quarkus.io/guides/

```
13              .withProperty("quarkus.datasource.password", database.getPassword())
14              .withProperty("quarkus.datasource.jdbc.url", database::getJdbcUrl);
15
16      @Override
17      protected RestService getApp() {
18          return app;
19      }
20
21      @Test
22      public void getAll() {
23          getApp().given()
24                  .get("/book")
25                  .then()
26                  .statusCode(HttpStatus.SC_OK)
27                  .body("", hasSize(7));
28      }
29  }
```

Výpis 2.1: Priklad testu z Quarkus QE Test Frameworku. Ukázka kódu byla převzata z [23] a [22]

Kapitola 3

Kontejnerizované prostředí a Docker

Pro ukázaní a pochopení architektury clusteru Kubernetes a základních principů této platformy je důležité pochopit základní koncepty kontejnerizace.

3.1 Kontejnery

[26]

Kontejner je izolované prostředí, které zabalí aplikace se všemi jejími závislostmi do ucelené jednotky tak, aby aplikaci bylo možné rychle a spolehlivě nasadit.

Moderní aplikace se vetšinou skládají z několika kontejnerů, kde každý plní svou určitou funkci. Velikost kontejnerů se obvykle měří na megabajty. Kontejnery nevyužívají hypervizor[1] a obecně jsou považovány za rychlejší způsob řešení izolace procesů než virtuální stroj.

Virtualní stroje hrají důležitou roli v cloud computing, emulují skutečné fyzické stroje a umožňují oddělit běhové prostředí různých aplikací. Jeden server může obsahovat více virtuálních strojů, hypervizor je pak vrstvou mezi hardwarem a virtuálním strojem a efektivně spravuje přístup ke zdrojům. Virtuální počítače obsahují vlastní operační systém, což jim umožňuje vykonávat více funkcí náročných na zdroje najednou. Větší množství prostředků, které mají virtuální počítače k dispozici, jim umožňuje abstrahovat, rozdělovat, duplikovat a emulovat celé servery, operační systémy, databáze a sítě [26]. Grafické porovnání kontejnerů a virtuálních strojů je znázorněno na obrázku 3.1

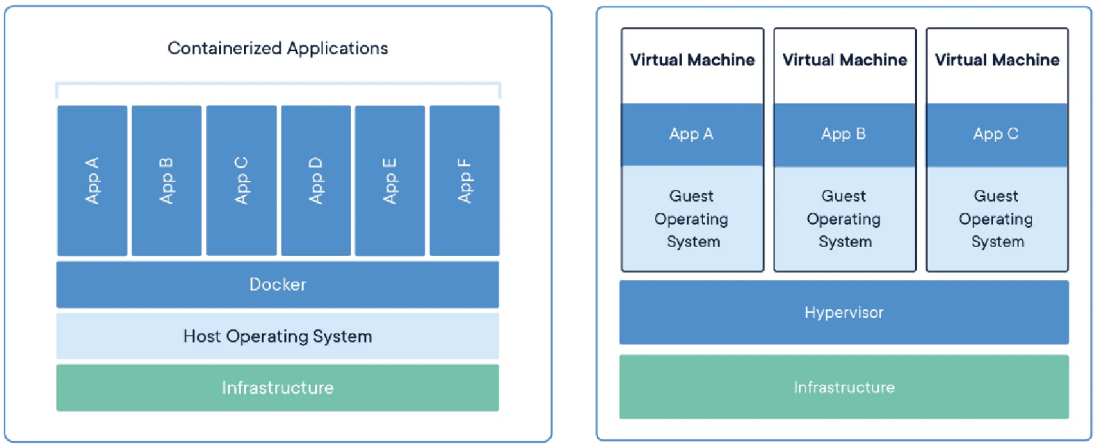
Přestože virtuální stroje se zdají mohutným a dostačujícím řešením, moderní cloud-native aplikace jsou často postavené na principu kontejnerizace, neboť' režie spojení s hostitelským operačním systémem je nižší.

Tradiční IT architektura předpokládá provozovaní cele aplikace v rámci virtuálního počítače, ačkoli mít celý kód a závislosti na jednom místě vedlo k nadměrným velikostem virtuálních počítačů, u kterých docházelo k selháním a výpadkům při zavádění aktualizací [26].

[1] Hypervizor – software, virtualizující hardware do izolovaných virtuálních strojů

Cloud-native vývoj a například i spuštění CI/CD[2] jsou možné, protože systémová zátěž je rozdělena mezi malými jednotkami, které plní specifické úkoly. Tyto jednotky jsou izolované, což přispívá k jejich snadnému vývoji, rychlému nasazování a škálování. V porovnání s tradiční architekturou, na kterou jsou často postavené monolitické aplikace, nabízí cloud-native přístup větší flexibilitu a modularitu. Moduly jsou zabaleny do kontejnerů, což umožňuje více lidem pracovat na jednotlivých částech aplikace, aniž by ovlivnili kód zabalený v jiných kontejnerech.

Produkční aplikace se často skládají z velkého množství kontejnerů a je vhodné mít nástroj pro jejich správu. Příkladem platformy pro správu orchestrace kontejnerů je Kubernetes. Daná platforma je založena na kontejnerech a používá je k balení aplikací do izolovaných jednotek, které mohou být nasazené, škálované a spravované v rámci clusteru Kubernetes.



Obrázek 3.1: Porovnání způsobu nasazení aplikací. Převzato z https://www.docker.com/resources/what-container/

3.2 Docker

[7] [6]

Docker je platforma s otevřeným zdrojovým kódem, která umožňuje vytvářet, nasazovat a spouštět kontejnery. Kontejnery lze při práci sdílet a mít jistotu, že každý, s kým je sdílíte, dostane kontejner, který funguje stejným způsobem.

Pokud je kontejner izolovaný proces, odkud získává své soubory a konfiguraci? Právě tady přicházejí na řadu obrazy kontejnerů.

Obraz (anglicky image) kontejneru je standardizovaný balíček, který obsahuje všechny soubory, binární soubory, knihovny a konfigurace pro spuštění kontejneru. Obrazy jsou neměnné. Jakmile je obraz jednou vytvořen, nelze jej měnit.

Každý soubor obrazu Docker se skládá z řady vrstev, které jsou spojeny do jednoho obrazu. Vrstva se vytvoří, když se obraz změní. Docker tyto vrstvy opakovaně používá při vytváření nových kontejnerů, což urychluje proces vytváření [7].

[2]CI/CD (Continuous Integration/Continuous Deployment) je přístup k automatizaci vývoje softwaru, která umožňuje pravidelnou integraci změn do kódu, automatizované testování a nasazování aplikací do produkčního prostředí.

Docker registr – úložiště pro obrazy Docker, které umožňuje sdílet obrazy mezi uživateli. Docker Hub je nejpopulárnější veřejný registr pro platformu Docker.

Je možné vytvářet kopii existujících úložišť Docker na vlastním serveru. Má to své výhody pro použití v prostředí, kde probíhá časté stahování (pull) obrazu z Docker Hub. Kvůli tomu, že tento registr má limity na počet požadavků pro nezaregistrované uživatele, je potřeba mít předplacený plán pro navýšení počtu požadavků nebo zprovoznit svůj vlastní registr obrazu Docker, na který se budou přesměrovávat všechny požadavky z `docker.io`. Vlastnění soukromého úložiště je výhodné pro velké softwarové firmy a je použita i v Quarkus QE pro testování aplikací v kontejnerech, využívajících veřejné obrazy jako například PostgreSQL.

3.3 Běhové prostředí kontejnerů

[7] [6]

Běhové prostředí kontejneru (container runtimes) je komponenta zodpovědná za spuštění a životní cyklus kontejneru. Běhové prostředí kontejneru lze obecně rozdělit na nízkoúrovňové a vysokoúrovňové, přičemž každý z nich nabízí jedinečné funkce a úrovně abstrakce:

- nízkoúrovňové běhové prostředí – základní, minimální běhové prostředí, které přímo spolupracuje s jádrem operačního systému,
- vysokoúrovňové běhové prostředí – zodpovědné za transport a správu obrazů kontejnerů, rozbalení obrazu a předání nízkoúrovňovému běhovému prostředí ke spuštění kontejneru.

3.3.1 Open Container Initiative (OCI)

Open Container Initiative (OCI) je otevřený projekt, podporovaný nadací Linux Foundation s konkrétním cílem vytvořit otevřené průmyslové standardy pro kontejnerové formáty a běhová prostředí. OCI spravuje tři základní specifikace, které musí splňovat všechny moderní běhové systémy kontejnerů [33]:

- vlastní specifikace obrazu kontejneru,
- způsob, jakým může běhové prostředí kontejnerů načíst obraz kontejneru,
- jak se obraz rozbaluje, připojuje do vrstev a spouští.

3.3.2 Rozhraní pro běh kontejneru

[29]

Rozhraní pro běh kontejneru (anglicky Container Runtime Interface) CRI určuje způsob komunikace mezi Kubernetes a konkrétním běhovým prostředím kontejneru. Umožňuje kubeletu[3] používat širokou škálu běhových prostředí kontejnerů, aniž by bylo nutné překompilovat součásti clusteru.

[3]Kubelet je součástí ekosystému Kubernetes. Funguje jako agent na úrovni uzlu, který pomáhá se správou kontejnerů a orchestrací v rámci clusteru Kubernetes. Všechny komponenty Kubernetes budou podrobně vysvětleny v kapitole 4

Aby mohl kubelet spouštět kontejnery, potřebujete na každém uzlu v clusteru funkční běhové prostředí kontejnerů.

Rozhraní CRI je hlavním protokolem pro komunikaci mezi kubeletem a běhovým prostředím kontejnerů.

3.3.3 Populární běhové prostředí kontejneru

[1] [27]

Každá platforma pro kontejnerizaci, která implementuje specifikaci OCI, je považována za nízkoúrovňové kontejnerové běhové prostředí. Například mezi ně patří runC a containerd.

- **runC** byl vytvořen společností Docker a darován OCI. Jedná se o výchozí běhové prostředí kontejnerů, které se univerzálně používá na zařízeních s operačním systémem na bázi Linux kernel.,
- **containerd** je na pomezí nízké a vysoké úrovně. Ve skutečnosti se více přiklání k vysokoúrovňovému běhovému prostředí kontejnerů, protože poskytuje vrstvu API nad běhovém prostředím kompatibilním s OCI. Vždy existuje externí vrstva, kterou používáte k interakci s ním, například CRI v Kubernetes,
- **Docker** je nejznámější platformou pro běh kontejnerů. Komplexní platforma pro vytváření, nasazování a provozování aplikací v kontejnerech,
- **CRI-O** byl speciálně navržený pro spouštění kontejnerů, které dodržují specifikaci CRI. Zaměřuje se na poskytování lehkého, stabilního a bezpečného běhového prostředí pro spouštění kontejnerů v rámci clusterů Kubernetes. Nabízí CRI-O funkce jako je izolace kontejnerů, správa obrazů a životního cyklu kontejnerů.

Kontejnerové běhové prostředí způsobily revoluci ve vývoji, nasazení a správě softwaru. Docker, CRI-O, containerd a runc hrají důležitou roli na IT trhu. Docker díky svému uživatelsky přívětivému rozhraní zpopularizoval kontejery. CRI-O je určen speciálně pro prostředí Kubernetes [1] [27].

Kapitola 4

Kubernetes

Kubernetes (zkráceně K8s) je systém s otevřeným zdrojovým kódem pro orchestraci kontejnerů. Společnost Google otevřela projekt Kubernetes v roce 2014. Kubernetes kombinuje více než 15 let zkušeností společnosti Google s provozováním velkých produkčních prostředí s nejlepšími nápady a postupy komunity [31].

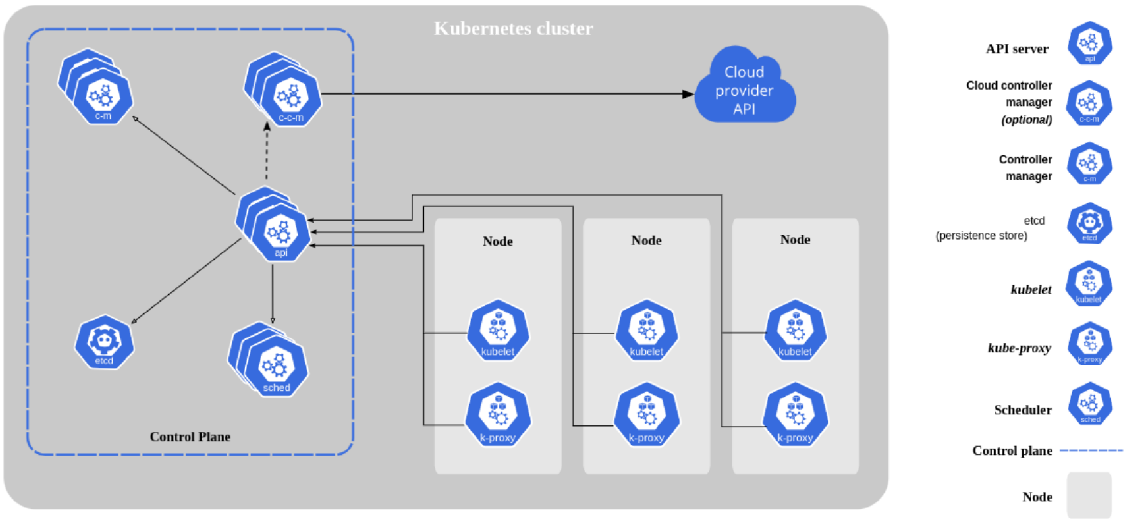
4.1 Výhody Kubernetes

Kontejnery jsou vhodným způsobem, jak aplikace spouštět a jednoduše distribuovat. Na produkční aplikace je kladen důraz na dostupnost. Při výpadku je potřeba hned nastartovat další kontejner a to způsobuje mnoho problémů, od monitoringu aktuálního stavu až po opravu vzniklé situace. Kubernetes je nástrojem, který je schopen řešit takové problémy. Může automaticky škálovat a znovu nasadit aplikace v případě poruchy, poskytuje vzory nasazení a další. Mimo jiné Kubernetes je schopný:

- vyvážení zátěže (load balancing) – pokud je provoz na kontejneru vysoký, Kubernetes dokáže vyrovnávat zátěž a rozdělovat síťový provoz tak, aby bylo nasazení stabilní,
- automatické řízení stavu kontejneru v reálném času. Kubernetes restartuje kontejnery, které selžou, nahrazuje kontejnery, zabíjí kontejnery, které nereagují na vámi definovanou kontrolu stavu, a neinzeruje je klientům, dokud nejsou připraveny k obsluze,
- horizontální škálování – aplikaci lze škálovat nahoru a dolů automaticky na základě využití procesoru a paměti přiřazené kontejnerům.

4.2 Architektura clusteru Kubernetes

Kubernetes cluster se skládá z množiny pracovních stanic (anglicky nodes), na kterých běží kontejnerové aplikace. Komponenty Kubernetes lze rozdělit na ty, které spravují jednotlivé uzly, a ty, které jsou součástí řídicí roviny (anglicky control plane) [38]. Kompletní schéma clusteru Kubernetes je znázorněno na obrázku 4.1.



Obrázek 4.1: Architektura clusteru Kubernetes. Převzato z [38]

4.2.1 Control plane

[38] [8]

Řídící rovina tvoří jádro systému Kubernetes. Komponenty řídicí roviny lze spustit na libovolném stroji v clusteru. Pro zjednodušení skripty nastavení obvykle spouštějí všechny součásti řídicí roviny na stejném stroji, ale pro náročné požadavky se doporučuje nastavení clusteru v režimu vysoké dostupnosti. Řešeni této práce vsak vystačí s clusterem s jedním uzlem, kde všechny komponenty běží na stejném serveru.
Součásti řídicí roviny Kubernetes jsou následující:

etcd je silně konzistentní, distribuované úložiště klíč-hodnota. Kubernetes používá etcd k ukládání konfiguračních dat, která představují celkový stav clusteru v daném okamžiku. Případná ztráta nebo poškození uložených dat může vést k narušení funkčnosti celého clusteru.

API server zpřístupňuje rozhraní API Kubernetes, je jedním z nejdůležitějších hlavních služeb clusteru. Umožňuje uživateli konfigurovat pracovní jednotky Kubernetes. API server je jediná komponenta, která přímo komunikuje s etcd, všechny zbylé komponenty už komunikují pomocí API serveru. Implementuje rozhraní RESTful, což znamená, že s ním může snadno komunikovat mnoho různých nástrojů a knihoven. Příkladem může být klient `kubectl`.

Plánovač přiřazuje pracovní zátěže konkrétním uzlům v clusteru. Plánovač je zodpovědný za sledování dostupné kapacity jednotlivých uzlů, aby se zajistilo, že pracovní zátěže nebudou naplánovány v rozsahu, který by překračoval dostupné zdroje.

Správce kontrolérů spravuje různé kontroléry, které regulují stav clusteru, například zajišťují správný počet replik. Při zjištění změn v systému kontrolér přečte nové informace a provede kroky pro dosazení požadovaného stavu.

4.2.2 Komponenty uzlu

[38] [8]

Komponenty uzlu představují samostatné aplikace, které běží uvnitř clusteru. Tyto aplikace jsou nezbytné pro komunikaci s hlavními komponentami clusteru, konfiguraci sítě kontejnerů a spouštění skutečných pracovních zátěží, které jsou jim přiděleny.

Běhové prostředí kontejneru je zodpovědné za spouštění a správu kontejnerů. Každá pracovní jednotka na clusteru je implementována jako jeden nebo více kontejnerů, které je třeba nasadit. Běhové prostředí kontejneru je komponentou, která spouští kontejnery definované v pracovních úlohách předaných clusteru. Běhové prostředí kontejneru je podrobně popsané v sekci 3.3.

Kubelet je agentem, který je spuštěn v každém uzlu clusteru a spravuje kontejneru, vytvořené pomoci Kubernetes.

Kube-proxy je implementace implementace síťového proxy serveru[1], umožňuje sítovou komunikaci mezi pody[2] a službami v clusteru.

4.3 Objekty a pracovní zátěže Kubernetes

[8]

Kontejnery jsou základním mechanismem pro nasazení aplikací, ale Kubernetes nad nimi přidává další vrstvy abstrakce, které nabízejí funkce jako škálování a správa životního cyklu. Místo správy jednotlivých kontejnerů uživatelé pracují s instancemi, které kombinují různé objekty Kubernetes a interagují s nimi. Dále následuje přehled základních důležitých objektů Kubernetes.

Pod je základní jednotka nasazení v Kubernetes, která se sestavuje z jednoho nebo více kontejnerů, které mají být řízeny jako jedna aplikace. Pody mají společný životní cyklus a měly by být vždy naplánovány na stejném uzlu. Obvykle obsahují hlavní kontejner a volitelné pomocné kontejnery pro úzce související úkoly. Například pod může obsahovat hlavní aplikační server a pomocný kontejner pro správu souborů. Horizontální škálování na úrovni podů se nedoporučuje, k tomu jsou určeny objekty vyšší úrovně.

Namespaces neboli prostory jmen poskytují mechanismus pro izolaci skupin prostředků v rámci jednoho clusteru. Názvy prostředků musí být jedinečné v rámci jmenného prostoru, ale ne napříč jmennými prostory [39].

Deployments neboli nasazení představují mechanismus správy sady replik aplikaci. Deployment řídí počet aktivních podů a v případě potřeby provede aktualizaci nebo ji vrátí zpět [36].

Service neboli služba je komponentou, která funguje jako základní interní vyrovnávač zátěže. Služba seskupuje pody, které vykonávají stejnou funkci, a prezentuje je jako jeden celek. Ve výchozím nastavení jsou služby dostupné pouze pomocí interně směrovatelné IP adresy `ClusterIP`. Služby lze zpřístupnit i mimo cluster změnou nasazovací strategii. Konfigurace NodePort funguje tak, že se na vnějším síťovém rozhraní každého uzlu otevře statický port. Aplikace je pak přístupná na adrese, skládající se z IP adresy clusteru a statického portu. Alternativou je typ služby LoadBalancer, který vytvoří externí vyvažovač zátěže, poskytovaný poskytovatelem cloudu [8].

[1] https://cs.wikipedia.org/wiki/Proxy_server

[2] Pod je základní jednotka nasazení v Kubernetes, která se sestavuje z jednoho nebo více kontejnerů, které mají být řízeny jako jedna aplikace

4.4 Síťový model Kubernetes

[40]

Kubernetes definuje síťový model, který pomáhá zajistit jednoduchost a konzistenci napříč různými síťovými prostředími a síťovými implementacemi. Síťový model Kubernetes poskytuje základ pro pochopení toho, jak spolu kontejnery, pody a služby v rámci Kubernetes komunikují.
Model sítě Kubernetes určuje:

- každý pod dostane svou vlastní IP adresu,
- kontejnery v rámci podů sdílejí IP adresu podů a mohou mezi sebou komunikovat,
- pody mohou komunikovat se všemi ostatními pody v clusteru pomocí IP adres podů bez NAT.

Kubernetes podporuje síťové rozhraní kontejneru (CNI[3]) pro nastavení síťové komunikace uvnitř clusteru. CNI plugin je žádoucí pro implementaci síťového modelu Kubernetes.

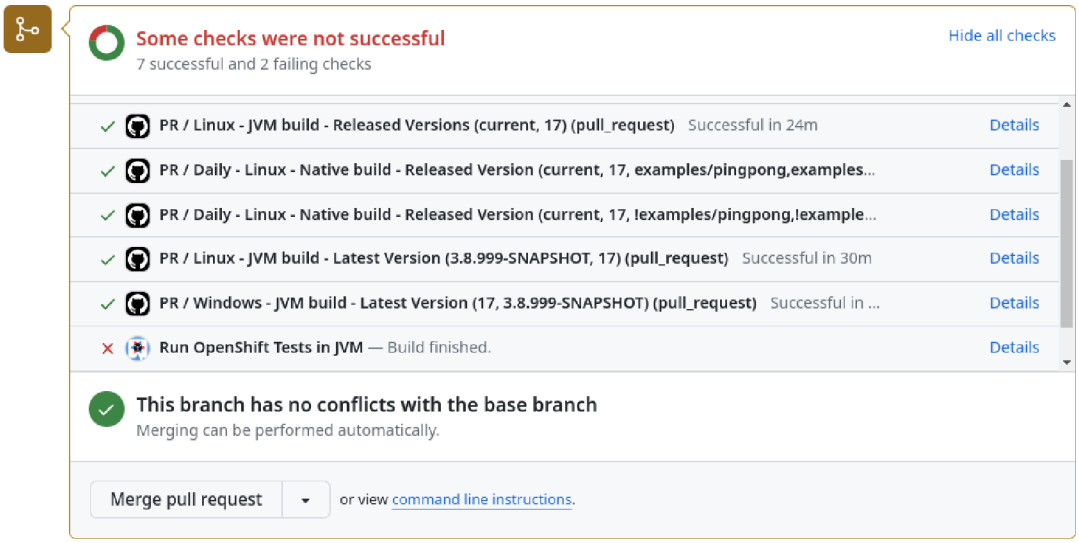
[3]Container Network Interface (CNI) – síťové rozhraní kontejneru

Kapitola 5

GitHub a Jenkins

Quarkus QE Test Framework je nástroj s otevřeným zdrojovým kódem a je umístěný na známé webové platformě pro správu verzí GitHub.

GitHub umožňuje vytvářet tak zvané Pull Requesty (PR), návrhy na sloučení sady změn z jedné větve do druhé. Po založení Pull Requestu mohou účastníci zkontrolovat a prodiskutovat navrhovanou sadu změn předtím, než je začlení do jiné větve [9]. Dobrou praxí je zprovozněni automatického testovaní po změnách v PR, příklad výsledku takového testování je na obrázku 5.1.



Obrázek 5.1: Příklad zobrazování výsledků GitHub CI. Snímek obrazovky byl udělán v PR https://github.com/quarkus-qe/quarkus-test-framework/pull/1116

Některé testy z různých důvodů nemohou být spuštěny uvnitř infrastruktury, co poskytuje GitHub a proto jsou vykonané na externí platformě, například Jenkins[1].

Jenkins používá ke své práci projekty nebo jinak joby, které jsou definované uživateli systému. Jenkins má několik typů projektu, ale v rámci této práce se používá typ `Pipeline` [20].

[1] https://www.jenkins.io/

`Jenkins Pipeline` je sada pluginů pro vytváření opakujících se automatických postupů. Pipeline je obvykle rozdělena na několik fází, obsahující nějaké menší úlohy. Například fáze sestavení, testování a nasazení aplikace.

Kód, definující pipeline se ukládá do textového souboru známém jako `Jenkinsfile`. Doporučuje se uložení tohoto souboru spolu s zdrojovým kódem projektu, například na platformě GitHub. Soubor Jenkinsfile lze napsat pomocí dvou typů syntaxe - deklarativní a skriptované [42] [19]. Při řešení této práce budou využívané skriptované pipeline z důvodu větší flexibility. Příklad skriptované pipeline, která instaluje adaptér PostgreSQL pro Python je možné vidět ve výpisu 5.1

```
node('large') {
    stage('Install Postgresql adapter with pip'){
        sh """
        python3 -m venv venv
        source venv/bin/activate
        pip install psycopg2
        """.stripIndent()
    }
}
```

Výpis 5.1: Příklad skriptováné pipeline Jenkins

V příkladu `node` určuje typ stroje, na kterém poběží pipeline. `Stage` neboli fáze je jednotka, která se zabývá instalací PostgreSQL adaptéru. Tato fáze se skládá z několika kroků, napsaných v Shell skriptu[2].

Jenkins umožňuje bezpečné ukládání přihlašovacích privátních údajů do jiných aplikací. Jakmile správce Jenkins přidá privátní udaje do systému, mohou být tyto udaje použité v pipeline. Pro maximální zabezpečení jsou údaje uložené v zašifrované podobě a v projektech pipeline s nimi pracuje pouze prostřednictvím identifikátoru.

[2] https://en.wikipedia.org/wiki/Shell_script

Kapitola 6

Návrh řešení

Téma této práce vzniklo z potřeb společnosti Red Hat a Quarkus QE týmu. V testovacím QE frameworku už skoro dva roky nefungoval Kubernetes scénář[1] a hlavním cílem této práce bylo potřeba ho opravit.

Dále je nutné zmínit, že v době hlášení zmíněné GitHub issue byly Kubernetes testy spuštěny jako součást denní GitHub Actions CI pipeline. Cluster Kubernetes se instaloval do infrastruktury GitHub pomocí nástroje Kind[2]. Požadavek Quarkus QE týmu je instalace plnohodnotného vlastního clusteru Kubernetes do interní infrastruktury Red Hat. Tím bude zaručena základní podpora pro testování aplikace Quarkus v Kubernetes. Ta bude co nejblíže scénáři, jaký používají zákazníci v produkčním prostředí. Tyto testy musí být někde automaticky spuštěné a nejlepší volbou se stal Jenkins, na kterém jsou už periodicky spouštěné testy pro QQE Framework a různé testovací sady v JVM a nativním módech. Implementace bude předpokládat aktivní Jenkins trigger job, který bude spojený s QQE Framework repozitářem v GitHub pomocí GitHub Pull Request Builder pluginu [17]. Trigger job bude každých 5 minut hledat aktivační fráze v otevřených Pull Requestech a v případě nalezení spustí všechny testy, označené **@KubernetesScenario** v QQE Frameworku. Po dokončení testování bude výsledek předán zpět do Pull Requestu.

Celkové budoucí řešení lze rozdělit do tří částí:

- změny v Quarkus QE Test Frameworku, sekce 6.1,
- instalace clusteru Kubernetes, sekce 6.2,
- Jenkins joby a propojení s GitHub, sekce 6.3.

6.1 Úprava Quarkus QE Test Frameworku

Zaprvé je nutné provést verifikaci nefunkčnosti scénáře podle postupů z GitHub issue (výpis 6.1).

```
1 # Nastartovani lokalniho clusteru Kubernetes
2 minikube start
3 # Klonovani Quarkus QE Test Framework Git repozitare
4 git clone git@github.com:quarkus-qe/quarkus-test-framework.git
5 cd quarkus-test-framework/examples/greetings
6 # Vyber release vetve pred zmenami pro Kubernetes
```

[1] GitHub issue: https://github.com/quarkus-qe/quarkus-test-framework/issues/433
[2] https://kind.sigs.k8s.io/

```
git checkout 1.4.2.Beta9
# Spusteni testu
mvn clean verify -Dts.container.registry-url=quay.io/quarkusqeteam \
-Dit.test=KubernetesRemoteDevModeGreetingResourceIT

[ERROR] Tests run: 1, Failures: 0, Errors: 1, Skipped: 0
[ERROR] io.quarkus.qe.KubernetesRemoteDevModeGreetingResourceIT
java.lang.NullPointerException: Cannot invoke
"io.fabric8.kubernetes.api.model.HasMetadata.getMetadata()" because "obj" is null
```

Výpis 6.1: Verifikace nefunkčnosti Kubernetes scénáře

Originální problém má jiný chybový výpis, související s nemožností přiřazení vnější IP adresy pro službu typu LoadBalancer, `External IP` zůstává ve stavu `Pending`

```
kubectl: NAME    TYPE          CLUSTER-IP     EXTERNAL-IP
kubectl: app  LoadBalancer   172.30.183.43   <pending>
```

Aktuálně se program ukončí chybou typu „null pointer“ a v první řadě bude potřeba provést dodatečné úpravy pro dosažení stavu chyby při vystavení služby.

V tomto okamžiku jsou definované body, které je potřeba splnit pro fungování testovacího scénáře Kubernetes:

- Upravit kód `KubectlClient` třídy pro zabránění NullPointerException,
- Provést změny v modulu `quarkus-test-kubernetes` pro externí vystavení služeb pomocí `NodePort` místo `LoadBalancer`,
- Refaktorování kódu: aktualizace metody `KubectlClient` tak, aby používala nové metody z klienta Fabric8 Kubernetes [13]. Zastaralé funkce budou odstraněny.

6.2 Instalace clusteru Kubernetes

Cluster Kubernetes bude nainstalován na virtuální stroj, běžící na platformě Red Hat OpenStack[3]. Quarkus QE tým má předpřipravené snímky pro vytváření virtuálních strojů na bázi systému Red Hat Enterprise Linux 8[4].

Jako rozhraní pro běh kontejneru (CRI) bude použit CRI-O z důvodu podpory ze strany společnosti Red Hat, stejný CRI je použity i v OpenShift. Srovnání CRI-O s dalšími alternativami je popsané v sekci „Populární běhové prostředí kontejneru“ 3.3.3.

Při výběru síťového rozhraní kontejneru CNI se rozhodovalo mezi Flannel [16] a Calico [41], které jsou jedněmi z nejpopulárnějších. Nakonec volba padla ve prospěch Flannel. Tento plugin má nízkou režii a přijatelný výkon pro malé až střední případy použití. Přestože poskytuje jen základní síťové připojení, v porovnání s Calico je pro případ spuštění testů v clusteru Kubernetes s jedním jediným pracovním uzlem dostačujícím řešením.

Pak bude potřeba nakonfigurovat přesměrování stahování obrazu z repozitáře `docker.io` do vlastního registračního serveru, provozovaného Quarkus QE týmem. Tato úprava zabrání dosažení limitu pro stažení veřejných kontejnerových obrazů z Docker Hub registru.

[3] OpenStack je open-source cloudová platforma, která umožňuje vytvářet a spravovat infrastrukturu a služby. Více lze přečíst na stránce https://www.redhat.com/en/technologies/linux-platforms/openstack-platform

[4] https://access.redhat.com/products/red-hat-enterprise-linux

Když všechny důležité komponenty jsou určeny, přichází čas na instalaci clusteru, a to pomocí nástroje `kubeadm`. Nástroj provádí činnosti nezbytné pro zprovoznění minimálně životaschopného clusteru, podpůrné pluginy se instalují potom. Návod na instalaci se nachází v sekci 7.2 kapitoly „Implementace“ 7.

Po dokončení instalace a konfigurace clusteru bude potřeba nastavit mechanizmus autentizace a práva uživatele. Spolu s Quarkus QE týmem jsme přišli s nápadem, ze použit Bearer token[5] pro autentizaci bude vhodným řešením kvůli možnosti jeho trvalého uložení mezi Jenkins credentials.

6.3 Jenkins joby a propojení s GitHub

Řešení předpokládá vytvoření dvou Jenkins jobu. První bude instalovat Kubernetes cluster, druhý bude zodpovědný za automatizované spouštění testů Kubernetes z Quarkus QE Test Frameworku při zadání specifické fráze v Pull Requestu.

Instalační skript clusteru Kubernetes je nejdůležitější částí celé práce, bez funkčního clusteru nebude možné provést další kroky testování aplikací v Kubernetes. Tento skript lze logicky rozčlenit na pipeline fáze:

- přihlášení do platformy OpenStack,
- vytváření virtuálního stroje na baží operačního systému RHEL 8,
- staženi repozitáře Kubernetes a instalace kubelet, kubeadm, kubectl,
- instalace a konfigurace rozhraní pro běh kontejnerů,
- konfigurace uzlu pro používání interního registračního serveru, který funguje jako zrcadlo (anglicky mirror) pro `docker.io`,
- inicializace řídící roviny Kubernetes pomocí nástroje `kubeadm`,
- nastavení konfigurace nástroje `kubectl` pro přístup k Kubernetes API,
- kontrola síťového připojení podů a instalace CNI Flannel,
- konfigurace API serveru clusteru pro akceptaci autentizace pomocí Bearer tokenu,
- udělení přístupu uživateli k prostředkům clusteru pomocí RBAC[6].

Další Jenkins pipeline job bude spuštěn v případě zadání aktivační fráze v Pull Requestu, otevřeném v GitHub projektu *Quarkus QE Test Framework*. Aktivace bude umožněna díky *GitHub Pull Request Builder* Jenkins pluginu[7]. Fáze této pipeline zahrnují:

- získání IP adresy clusterů Kubernetes ze seznamu instancí OpenStack,
- instalaci nástroje kubectl a obnovení jeho konfiguraci pro spojení s vytvořeným clusterem Kubernetes,
- spouštění Kubernetes testů z Quarkus Test Frameworu, obsahujících aktuální změny z Pull Requestu.

[5] Bearer token je typ tokenu, který se používá pro autentizaci a autorizaci ve webových aplikacích a rozhraních API. Token se obvykle skládá z náhodného řetězce znaků [2].

[6] Řízení přístupu na základě rolí (RBAC) je metoda řízení přístupu, která poskytuje oprávnění uživatelům na základě jejich rolí v organizaci, čímž umožňuje jednoduchou správu a bezpečný přístup ke zdrojům.

[7] https://plugins.jenkins.io/ghprb/

Kapitola 7

Implementace

Implementační část této bakalářské práce zahrnuje následující:

- úpravu modulu `quarkus-test-kubernetes` v Quarkus QE Test Frameworku,
- automatická instalace a nastavení clusteru Kubernetes pomocí Jenkins pipeline,
- automatické spuštění testů Kubernetes z QQE Frameworku, aktivované zadáním speciální fráze v GitHub Pull Requestu.

7.1 Podpora testování Kubernetes v QQE Test Frameworku

Úprava QQE Test Frameworku byla mým prvním krokem řešení této práce. Jak bylo zmíněno v sekci 6.1, řešení pro spuštění Kubernetes testů, napsaných ve QQE Test Frameworku, nefungovalo už skoro dva roky před napsáním této bakalářské práce. Cestou spuštění testu z GitHub issue[1] jsem zjistil, ze test framework dává chybu typu `Null Pointer Exception`.

Před opravou chyb je potřeba znát, jak funguje testovací scénář Kubernetes v Quarkus QE Test Frameworku. Budu využívat metody tříd z modulu jádra testovacího frameworku `quarkus-test-core` a rozšíření Kubernetes `quarkus-test-kubernetes`. Každé spuštění testovacího scénáře ve frameworku se začíná ve funkci `QuarkusScenarioBootstrap`, která se dědí od funkcí frameworku JUnit. Metoda beforeAll hledá rozšíření v napsaném testu, anotace `@QuarkusScenario, @KubernetesScenario, @OpenShiftScenario`. Nalezením Kubernetes anotace se provede volání metody `beforeAll` ve funkci `KubernetesExtensionBootstrap`, kde se zinicializuje `KubectlClient`. Stojí za zmínku, že Kubernetes anotace dovoluje určit jednu ze dvou nasazovacích strategií. První výchozí je za použití registu kontejneru, druhá používá rozšíření Quarkus Kubernetes[2].

Pro vysvětlení dalšího postupu spuštění testu bude využit příklad Kubernetes testu jako je ve výpisu 7.1.

```
1 @KubernetesScenario
2 public class KubernetesPostgresqlIT {
3
4     @Container(image = "${postgresql.image}",
5             port = 5432, expectedLog = "is ready")
6     static PostgresqlService database = new PostgresqlService();
```

[1] „Kubernetes use-cases are not working“: https://github.com/quarkus-qe/quarkus-test-framework/issues/433

[2] https://quarkus.io/guides/deploying-to-kubernetes#using-the-kubernetes-client

```
7
8       @QuarkusApplication
9       static RestService app = new RestService()
10              .withProperty("quarkus.datasource.username", database.getUser())
11              .withProperty("quarkus.datasource.password", database.getPassword())
12              .withProperty("quarkus.datasource.jdbc.url", database::getJdbcUrl);
13
14      @Override
15      protected RestService getApp() {
16          return app;
17      }
18
19      @Test
20      public void getAll() {
21          getApp().given()
22                  .get("/book")
23                  .then()
24                  .statusCode(HttpStatus.SC_OK)
25                  .body("", hasSize(7));
26      }
27  }
```

Výpis 7.1: Příklad Kubernetes testu

QQE Test Framework vždy zpracovává anotace v souboru shora dolu. `@KubernetesScenario` anotace už je zpracována. Pokračuje anotací `@Container`, kterou řídí třída `ContainerManagedResourceBuilder`. Podle kontextu testu framewrok zjistí, že se má provést vazba s `KubernetesContainerManagedResource`. Spuštěním tohoto zdroje se začíná příprava kontejneru k nasazení, dělá se to ve fázích. Zaprvé se načte kontext o kontejneru: název obrazu, port, na kterém bude běžet kontejner a volitelné další parametry. V druhé fázi probíhá příprava souboru YAML pro nasazení kontejneru. Šablonu je možné vidět na výpisu 7.2. Proměnné v hranatých závorkách (jako ve výpisu 7.2) se nahradí skutečnými hodnotami. Poslední krok před nasazením je doplnění šablony o další informace: prostor jmen, proměnné prostředí, config mapy, citlivé údaje (secrets). Tuhle činnost provádí metoda `KubectlClient#enrichTemplate(...)`. Výhodou je, ze tato obohacená šablona se uloží do složky target[3], odkud ji pak lze snadno aplikovat pomoci metody `KubectlClient#apply(...)`. Metoda používá nástroj `kubectl` pro aplikaci. Následně pomocí příkazu `kubectl expose deployment my-deployment --port container-port ...` kontejnerová služba bude vystavena v rozsahu jedné repliky.

Podobnou proces se provádí i pro `@QuarkusApplication` anotaci. Rozdíl je v tom, ze pro nasazení aplikace Quarkus existují tři různé typu správy zdrojů (anglicky managed resources). Názvy tříd mají následující formát `*KubernetesQuarkusApplicationManagedResource`:

- ContainerRegistry – základní typ, používá registr kontejneru,
- Extension – používá rozšíření Quarkus Kubernetes,
- RemoteDevMode – používá se pro správu aplikací, spuštěné ve vývojovém režimu.

Základní je zdroj, používající registry kontejneru `ContainerRegistry`. Ve funkci `doInit()` zdroje se zinicializují. Do registru kontejneru, specifikovaného pomocí property `ts.container.registry-url`, nahraje se obraz aplikace Quarkus. Příklady v této práci

[3]QQE Test Framework používá Maven pro sestavení projektu. Složka target je výchozím umístěním, kam se ukládají výstupy z procesu sestavení projektu.

používají privátní registrační platformu kontejneru Quay[4], stahovaní a nahrávaní obrazu do quay.io/quarkusqeteam repozitáře je dovoleno pouze členům týmu Quarkus QE. Pak šablona pro nasazeni služby aplikaci Quarkus `quarkus-app-kubernetes-template.yml` se rozšíří o další informace stejným způsobem, jako pro kontejnery. Dále tahle šablona bude aplikována a služba vystavena. V tomto bodě měli by být vystavené a funkční všechny služby, určené v testovací třídě. Po nastartování aplikace Quarkus probíhá vyhodnocení testu.

```
---
apiVersion: "v1"
kind: "List"
items:
- apiVersion: "apps/v1"
  kind: "Deployment"
  metadata:
    name: "${SERVICE_NAME}"
  spec:
    replicas: 1
    selector:
      matchLabels:
        deployment: "${SERVICE_NAME}"
    template:
      metadata:
        labels:
          deployment: "${SERVICE_NAME}"
      spec:
        containers:
        - image: "${IMAGE}"
          args: [${ARGS}]
          name: "${SERVICE_NAME}"
          ports:
          - containerPort: ${INTERNAL_PORT}
            name: "http"
            protocol: "TCP"
```

Výpis 7.2: Šablona YAML pro nasazení kontejnerů

Když jsem se seznámil se všemi koncepty popsanými výše, mohl jsem začít řešit nefunkčnost scénáře Kubernetes. Chyba Null Pointer pochází z metody `enrichTemplate` třídy `KubectlClient`.

```
private List<HasMetadata> loadYaml(String template) {
    return client.load(new ByteArrayInputStream(template.getBytes())).get();
}

private String enrichTemplate(...) {
    List<HasMetadata> objs = loadYaml(template);
    for (HasMetadata obj : objs) {
        obj.getMetadata().setNamespace(namespace());
        ...
    }
}
```

Na objekt typu `HasMetadata` je volána metoda `getMetadata()` a příslušný objekt je null. Problém byl ve špatném zpracování YAML souboru metodou `loadYaml(...)`. Podle

[4] https://www.redhat.com/en/technologies/cloud-computing/quay

popisu metod Fabric8 klientu[5] metoda load vrátí `NamespaceListVisitFromServerGetDeleteRecreateWaitApplicable<HasMetadata>` a to rozhraní obsahuje jen metodu `items()`. Výsledkem je změna metody `loadYaml(...)` dle výpisu 7.3.

```
private List<HasMetadata> loadYaml(String template) {
    NamespaceListVisitFromServerGetDeleteRecreateWaitApplicable<HasMetadata> load = client
        .load(new ByteArrayInputStream(template.getBytes()));
    return load.items();
}
```

Výpis 7.3: Změněná metoda KubectlClient#loadYaml

Tahle změna ale nestačí pro dosazeni stavu, zmíněném v GitHub issue[6]. Dále je nutné se zbavit špatně použitých klientů OpenShift v třídě `KubectlClient`, neboť třída se zabývá jedině Kubernetes. Promenné `DefaultOpenShiftClient masterClient` a `NamespacedOpenShiftClient client` byly nahrazené `KubernetesClientImpl client`. Kód třídy `KubectlClient` byl refaktorován pro použití jediného klienta. Pak bylo potřeba změnit závislost v pom.xml, rozšíření Kubernetes ve frameworku musí používat `kubernetes-client` z platformy Fabric8 místo `openshift-client`. Kód privátního konstruktoru `KubectlClient` byl modifikován pro podporu nového klienta a zlepšení čitelnosti (výpis 7.4). QQE Test Framework podporuje efemérní prostředí jmen (anglicky ephemeral namespaces), kde framework vytváří nové jmenné prostředí pro každý testovací scénář a neefemérní, kde se používá aktuálně dostupné prostředí jmen. V obou případech probíhá inicializace klienta Kubernetes s aktuální předanou konfiguraci.

```
private KubectlClient(String scenarioUniqueName) {
    if (ENABLED_EPHEMERAL_NAMESPACES.getAsBoolean()) {
        currentNamespace = createNamespace();
        Config config = new ConfigBuilder().withTrustCerts(true).withNamespace(
    currentNamespace).build();
        client = createClient(config);
    } else {
        Config config = new ConfigBuilder().withTrustCerts(true).build();
        client = createClient(config);
        currentNamespace = client.getNamespace();
    }
}

private static KubernetesClientImpl createClient(Config config) {
    return new KubernetesClientBuilder().withConfig(config).build()
            .adapt(KubernetesClientImpl.class);
}
```

Výpis 7.4: Modifikovaný konstruktor třídy KubectlClient

Poslední část před verifikaci změn představuje zbavení se zastaralých funkcí v třídě `KubectlClient`. Tyto funkce byli zjištěny pomocí standartního analyzátoru kódu vývojového prostředí IntelliJ IDEA[7].

První zastaralá funkce se nacházela v metodě `applyServicePropertiesUsingDeploymentConfig`, která využívala

[5] https://github.com/fabric8io/kubernetes-client/blob/main/kubernetes-client-api/src/main/java/io/fabric8/kubernetes/client/KubernetesClient.java

[6] „Kubernetes use-cases are not working“: https://github.com/quarkus-qe/quarkus-test-framework/issues/433

[7] https://www.jetbrains.com/idea/features/#intelligent-editor

`createOrReplace(deployment)` z Fabric8 klienta pro vytvoření nového nasazení. Podle dokumentace projektu `Fabric8 kubernetes-client` [11] nové je potřeba používat konstrukci `resource(deployment).create()`. Tak jsem musel zaprvé odstranit staré nasazení a vytvořit nové jak je ukázáno ve výpisu 7.5

```
client.apps().deployments()
    .withTimeout(DEPLOYMENT_CREATION_TIMEOUT, TimeUnit.SECONDS)
    .delete();
client.apps().deployments().resource(deployment).create();
```

Výpis 7.5: Vytváření nového nasazení pomocí funkcí Fabric8 klient

Druhá zastaralá funkce se nacházela v metodě `enrichTemplate`. `Serialization.yamlMapper()` podle dokumentace [14] má nahradit `Serialization.asYaml()`.

Poslední zastarala funkce `createOrReplace()` pro ConfigMap[8] se nacházela v metodě `createOrUpdateConfigMap()` a byla nahrazena způsobem, uvedeným v [12]. Změněný kód je možné vidět ve výpisu 7.6.

```
client.configMaps().resource(new ConfigMapBuilder()
        .withNewMetadata().withName(configMapName).endMetadata()
        .addToData(key, value)
        .build()).createOr(NonDeletingOperation::update);
```

Výpis 7.6: Vytvoření nového objektu ConfigMap

Po zbaveni se zastaralých funkci jsem spustil testovací scénář a už jsem viděl chybu přiřazení vnější IP adresy pro službu typu LoadBalancer, jak byla definována v GitHub issue[9]. Pro podporu testovací aplikace na platforme Kubernetes pro Quarkus QE Framework není nutné potřeba nastavovat externí stroj, který bude rozdělovat zátěž. Pro účel externího vystavení služby v Kubernetes také slouží `NodePort`[10], který jsem použil místo `LoadBalancer`.

Podpora vystaveni služeb pomocí NodePort v QQE Test Frameworku vyžadovala úpravu několik metod napříč frameworkem. V třídě `KubectlClient` byla modifikována metoda `expose()` pro použití příkazu `kubectl expose deployment deployment-name --type=NodePort`. Nově metoda `host()` vrací IP adresu clusteru a metoda `port(Service service)` nově vrátí port na kterém běží předaná parametrem služba. V šabloně pro nasazeni aplikace Quarkus `quarkus-app-kubernetes-template.yml` a třídě `ExtensionKubernetesQuarkusApplicationManagedResource` typ služby byl vyměněn za `NodePort`.

V tomto bode je potřeba verifikovat funkčnost řešení s použitím vlastního clusteru Kubernetes, postup jeho instalace je detailně popsaný v sekci 7.2. V hlavní složce projektu QQE Test Framework jsem spustil příkaz `mvn clean verify -Dts.container.registry-url=quay.io/quarkusqeteam -Dkubernetes`, který prohledává všechny moduly v složce examples. Když Maven najde integrační test, označený anotací `@KubernetesScenario`, tak ho spustí. Výsledek zkušebního spuštění dopadl dobře až na výjimku testu `KubernetesSecurityResourceIT`, který vyuziva specilani realizaci zdroje pro spusteni kontejneru pomocí frameworku - `@KeycloakContainer`. Podporu tohoto typu kontejneru pro Kubernetes bylo potřeba realizovat v modulu `quarkus-test-service-keycloak`. Stejným principem jako pro obyčejné typy kontejneru je potřeba zajistit vazbu mezi realizaci kontejneru a Kubernetes (výpis 7.7). Metoda `appliesFor`

[8] https://kubernetes.io/docs/concepts/configuration/configmap/

[9] https://github.com/quarkus-qe/quarkus-test-framework/issues/433

[10] https://kubernetes.io/docs/concepts/services-networking/service/#publishing-services-service-types

vrátí true a prováže prostředky kontejneru Keycloak a Kubernetes jen když v definici integracního testu se najde anotace `@KubernetesScenario`. Pro zajištění dynamického načítání služby `KeycloakContainer` pomocí Java SPI[11] jsem přidal plně kvalifikovaný název třídy `KubernetesKeycloakContainerManagedResourceBinding` do speciálního souboru v složce META-INF.services. Třída `KubernetesKeycloakContainerManagedResource` rozšiřuje základního správce kontejneru pro Kuberntes, třídu `KubernetesContainerManagedResource`.

```
1  public class KubernetesKeycloakContainerManagedResourceBinding
2      implements KeycloakContainerManagedResourceBinding {
3      @Override
4      public boolean appliesFor(KeycloakContainerManagedResourceBuilder builder) {
5          Annotation[] annotations = builder.getContext().getTestContext().
       getRequiredTestClass().getAnnotations();
6          return Arrays.stream(annotations)
7                  .anyMatch(annotation -> annotation.annotationType().getName()
8                          .equals("io.quarkus.test.scenarios.KubernetesScenario"));
9      }
10
11     @Override
12     public ManagedResource init(KeycloakContainerManagedResourceBuilder builder) {
13         return new KubernetesKeycloakContainerManagedResource(builder);
14     }
15 }
16
17 public class KubernetesKeycloakContainerManagedResource extends
       KubernetesContainerManagedResource {
18     private final KeycloakContainerManagedResourceBuilder model;
19
20     protected KubernetesKeycloakContainerManagedResource(
       KeycloakContainerManagedResourceBuilder model) {
21         super(model);
22         this.model = model;
23     }
24 }
```

Výpis 7.7: Vazba prostředků kontejneru Keycloak a Kubernetes

Posledním krokem přidání podpory Kubernetes pro QQE Test Framework bylo přidání chybějícího testu, který by otestoval aplikaci Quarkus spolu se známým obrazem kontejneru. Rozhodl jsem se rozšířit modul `database-postgresql` o jeden test Kubernetes, konkrétně `KubernetesPostgresqlIT`, který rozšiřuje abstraktní třídu `AbstractSqlDatabaseIT` a používá obraz databáze PostgreSQL[12]

Následně jsem spustil všechny integrační testy Kubernetes a testy proběhly úspěšně.

7.2 Instalace clusteru Kubernetes

Instalátor clusteru Kubernetes bude využívat výhody platformy pro automatizaci procesů Jenkins. Platforma umožňuje vytvářet pipeliny pro automatické spuštění periodicky opakovaného procesu. V kontextu této bakalářské práce jsou to instalace clusteru Kubernetes a spuštění testu Kubernetes z Quarkus QE Test Frameworku. Popis trigger jobu pro testy je v sekci 7.3

Automatizovaný instalator clusteru Kubernetes je realizován pomoci Jenkins pipeline jobu `jobs/infra/install_kubernetes_cluster.groovy`.

[11] https://www.baeldung.com/java-spi
[12] https://www.postgresql.org/

Instalační job (výpis 7.8) vypadá následně :

```
pipelineJob('infra-install-kubernetes-cluster') {
    description('Install Kubernetes cluster into RHOS-D OpenStack.')
    parameters {
        stringParam(
            'KUBERNETES_VERSION',
            '1.29',
            'Kubernetes version'
        )
    }
    definition {
        cps {
            script(readFileFromWorkspace('pipelines/install_kubernetes_cluster.groovy'))
            sandbox()
        }
    }
}
```

Výpis 7.8: Instalační job pro Kubernetes

Pipeline skript `pipelines/infra/install_kubernetes_cluster.groovy` (výpis 7.9) popisuje instalaci clusteru. Ve funkci `withKubernetesInstallerEnvVars` se provádí přiřazování Jenkins credentials do proměnných. Funkce příjímá argument `closure`, který v kontextu skriptu představuje všechny kroky pipeline. To znamená, že proměnné definované v bloku `withCredentials` lze použít globalně v libovolném kroku pipeliny.

`psi-quarkus-qe-service-account` obsahuje údaje, nutné pro práci s OpenStack serverem.

`hudson-private-key` je privátní klíč RSA, používá se pro autentizace během připojení k vytvořenému serveru pomoci SSH.

```
def withKubernetesInstallerEnvVars(closure) {
    withCredentials([
                usernamePassword(credentialsId: 'psi-quarkus-qe-service-account',
                    usernameVariable: 'OPENSTACK_SERVICE_ACCOUNT_USERNAME',
                    passwordVariable: 'OPENSTACK_SERVICE_ACCOUNT_PASSWORD'),
                sshUserPrivateKey(credentialsId: 'hudson-private-key',
                    keyFileVariable: 'HUDSON_PRIVATE_KEY'),
    ]) {
        closure()
    }
}

node('RHEL8&&large') {
    withKubernetesInstallerEnvVars {
        timestamps {
            stage("Clean up") {
                cleanWs()
            }
            ...
        }
    }
}
```

Výpis 7.9: Struktura instalační pipeline

Každá část instalace je Bash skript, zabalený do kroků v pipeline `stage('název_kroku')`, celková pipeline[13] obsahuje 13 kroků.

První tři kroky se zabývají přípravou prostředí pro vytvoření OpenStack instancí: vyčištění pracovního prostředí, instalace klienta a nastavení autorizačních údajů pro OpenStack. V čtvrtém kroku probíhá vytvořeni virtuálního stroje na baží operačního systému RHEL 8 (výpis 7.10).

```
export TEST_VM_NAME=kubernetes-on-jenkins
export TARGET_SNAPSHOT_NAME='__QUARKUSQE-SNAPHOT-rhel8-x86_64-DEPLOY'
openstack server create ${TEST_VM_NAME} --flavor ci.m1.large
    --image $TARGET_SNAPSHOT_NAME --nic net-id=provider_net_cci_8
    --security-group default --key-name hudson --wait
```

Výpis 7.10: Postup vytváření RHEL 8 instance na platformě OpenStack

V této fázi je dostupný virtuální stroj, na kterém budou běžet všechny části clusteru Kubernetes.
Další krok se zabývá instalací sady nástrojů: `kubeadm` pro spuštění clusteru, `kubelet` pro správu běhu kontejneru v clusteru a `kubectl` nástroj příkazového řádku pro komunikaci s clusterem prostřednictvím API serveru. Pro operační systém RHEL je nutné stáhnout Kubernetes repozitář pro yum[14] a nainstalovat balíčky [37].

Před spuštěním instalátoru `kubeadm` je nutné nainstalovat rozhraní pro běh kontejneru CRI. Můj vyber byl CRI-O z důvodu podpory ze strany společnosti Red Hat, což zajistí kompatibilní prostředí pro testování komerční verze aplikace Quarkus, takzvaný Red Hat Build of Quarkus. Navíc platforma OpenShift používá stejný CRI. Podle návodu ze zdrojů [35] a [5] byla provedena konfigurace sítě a instalace CRI-O. Je nutno také podotknout, ze konfigurace pro různé verze Kubernetes se může lišit a celé řešení této bakalářské práce předpokládá instalaci Kubernetes verze 1.29.

Obrazy kontejnerů se obvykle se stahuji z veřejného registračního serveru Docker Hub, který má limity na počet stahování. Obejití limitu je možné nakonfigurovat přesměrováním stažení z `docker.io` na vlastní registrační server. Důvody pro takové řešení jsou zmíněné v sekci 3.2.

```
cat << EOF | sudo tee /etc/containers/registries.conf
unqualified-search-registries = ['registry.access.redhat.com', 'docker.io']
[[registry]]
  prefix = 'docker.io'
  location = '$(dig +short registry.dynamic.quarkus):5000'
  insecure = true
EOF
```

Výpis 7.11: Nastavení zrcadlení Docker obrazů na vlastní registrační server

Jak je vidět ve skriptu 7.11, změny se provádí v konfiguračním registračním souboru běhového prostředí kontejneru. Určuje se registrační server pro zrcadlení požadavku. Například požadavek na stažení z `docker.io/image` se přesměruje na `registry.dynamic.quarkus:5000/image`. Dané změny se aplikují po spuštění služby CRIO.

V tomto bode příprava před instalací už byla udělána a následující důležitý krok je spuštění instalátora (výpis 7.12). Prvním krokem se stáhnou obrazy API serveru, správce kontrolérů, plánovače a další. V druhém kroku následuje instalace. Je důležité specifikovat

[13] Pipeline je sada pluginu pro vytváření opakujících se automatických postupů. Pipeline je obvykle rozdělena na několik fází, obsahující nějaké menší úlohy. Například fáze sestavení, testování a nasazení aplikace [42].

[14] https://www.ituonline.com/tech-definitions/what-is-a-yum-repository/

adresový rozsah podu a vybrané CRI. Postup instalace byl převzat ze dvou zdrojů [10], [28] a následně upraven podle potřeb.

```
sudo kubeadm config --cri-socket=unix:///var/run/crio/crio.sock images pull
sudo kubeadm init --pod-network-cidr=10.244.0.0/16 \
    --cri-socket /var/run/crio/crio.sock
```

Výpis 7.12: Instalace clusteru Kubernetes pomocí nastoje kubeadm

Když byl cluster instalován je potřeba nakonfigurovat `kubectl` pro umožnění komunikace s clusterem pomoci API (výpis 7.13). Návod je přímo daný ve výstupu `kubeadm init` příkazu. V základním nastavení příkaz `cp -i /etc/kubernetes/admin.conf $HOME/.kube/config` vyžaduje přístup k souborům z kořenového adresáře. Kvůli tomu jsem se musel připojit k účtu superuživatele pomoci ssh a nakonfigurovat kubectl pro obyčejného uživatele se jménem `hudson`.

```
ssh cloud-user@localhost \
"
    HOME='/home/hudson'
    sudo mkdir -p $HOME/.kube
    sudo cp -i /etc/kubernetes/admin.conf $HOME/.kube/config
    sudo chown hudson:hudson $HOME/.kube
    sudo chown hudson:hudson $HOME/.kube/config
"
```

Výpis 7.13: Konfigurace kubectl

První kontrolu funkčnosti hlavních komponent clusteru jsem provedl vykonáním příkazu `kubectl get pods --all-namespaces`, kde jsem si všiml, že CoreDNS pody jsou ve stavu `CrashLoopBackOff`. Dle dokumentace Kubernetes [30], řešení spočívá v specifikaci správného DNS resolveru pro kubelet.

Následuje instalace síťového rozhraní kontejneru Flannel (vykonává se jediná aplikace YAML souboru) dle návodu z [15].

Kvůli tomu, ze cluster běží na jednom jediném stroji, tak jakýkoliv nový pod bude běžet na řídící rovině (control plane) Kubernetes. Problém spočívá v tom, že Kubernetes v základu zabraňuje plánování podu v řídící rovině a je nutné ručně zrušit omezeni dle dokumentace v sekci „Control plane node isolation“ zdroje [28].

Konfigurace clusteru pokračuje nastavením mechanismu autentizace pro přístup k Kubernetes API zvenku hostujícího virtuálního stroje. Spolu s mým technickým vedoucím z Red Hat jsme rozhodli, ze nejlepší volbou pro využití Kubernetes clusteru v našem týmu bude statický Bearer token. Stačí ho jednou uložit mezi Jenkins credentials. Na rozdíl od autentizace pomoci certifikátu nebude potřeba ho pokaždé měnit při přeinstalaci clusteru, v případě vzniku problému na straně vnitřní infrastruktury Red Hat. Podle návodu z [34] jsem vytvořil soubor `tokens.csv`, obsahující sekvenci token, jméno uživatele Kubernetes a identifikátor systémového uživatele hudson. Vytvořený soubor musí byt zmíněn v konfiguraci kontejneru Kubernetes API serveru, tak jsem přidal řádek `--token-auth-file=/etc/kubernetes/pki/tokens.csv`. Po aplikaci změn `kube-apiserver` pod se restartuje, je potřeba počítat přibližně s minutovou čekací dobou, než API server stane zase plně funkční. Na výpisu 7.14 je možné vidět část skriptu pro nastavení autentizace.

```
sudo sh -c 'cat <<EOF > /etc/kubernetes/pki/tokens.csv
${KUBERNETES_API_TOKEN},qe,$(id -u hudson)
EOF'

```

```
sudo sed -i '/- kube-apiserver/ a\\
\ \ \ \ - --token-auth-file=/etc/kubernetes/pki/tokens.csv
' /etc/kubernetes/manifests/kube-apiserver.yaml
```

Výpis 7.14: Nastavení autentizace pomocí Bearer tokenu

Poslední krok této pipeline uděluje přístup uživateli k prostředkům clusteru pomocí RBAC. Použil jsem návod z oficiální dokumentace Kubernetes [32]. Modul pro testovaní Kubernetes v Quarkus QE Test Frameworku vytváří nový jmenný prostor pro každý test a to role uživatele musí umožnit. Tak objekty ClusterRole a ClusterRoleBinding jsou v tomto případě jasnou volbou, protože se nevztahují k určitému prostoru jmen. YAML soubor ve výpisu 7.15 popisuje všechna pravidla. Očekává se, že cluster budou používat jen členové týmu Quarkus QE pod stejným účtem, a cluster bude přístupný jen z interní Red Hat sítě, připadně přes VPN[15]. Z toho důvodu jsem vytvořil ClusterRole jako pro administrátora, následně ClusterRoleBinding svazuje tuto roli s uživatelem `qe`. Aplikací souboru `qe-role.yaml` se instalace clusteru úspěšně ukončí.

```
cat <<EOF | tee qe-role.yaml
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRole
metadata:
  name: qe-cluster-admin
rules:
- apiGroups: ["*"]
  resources: ["*"]
  verbs: ["*"]
---
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRoleBinding
metadata:
  name: qe-cluster-admin-binding
subjects:
- kind: User
  name: qe
  apiGroup: rbac.authorization.k8s.io
roleRef:
  kind: ClusterRole
  name: qe-cluster-admin
  apiGroup: rbac.authorization.k8s.io
EOF
```

Výpis 7.15: Nastavení RBAC pro uzivatele qe

7.3 Jenkins trigger job

Za automatické spuštění testu Kubernetes z QQE Frameworku, aktivované zadáním speciální fráze v GitHub Pull Requestu je zodpovězen pipeline job `jobs/quarkus/kubernetes_tf_githubci.groovy`. Hlavní myšlenkou tohoto jobu je provázání Pull Requestu v GitHub a automatické spuštění Jenkins jobu. Tato vlastnost může být dosažena díky GitHub Pull Request Builder (GHPRB) pluginu [17] pro Jenkins.

V Jenkins pipeline lze konfiguraci, včetně triggerů, obvykle nastavit v souboru Jenkinsfile[16], je běžně uložen v repozitáři na platformě pro správu verzí společně s kódem.

[15] VPN – virtuální privátní síť

[16] Jenkinsfile je skript, který definuje pipeline a její konfiguraci

Problém je, že Quarkus QE tým ukládá definice Jenkins jobu do soukromeho repozitáře mimo project QQE Test Framework, uloženého na GitHub. Zbavení se tohoto omezení spočívá v přidání definice GHPRB triggeru přímo do Groovy skriptu, jak je ukázané v kódu (výpis 7.16).

```
configure { node ->
    node / 'triggers' << 'org.jenkinsci.plugins.ghprb.GhprbTrigger' {
        orgslist 'quarkus-qe'
        cron 'H/5 * * * *'
        onlyTriggerPhrase 'true'
        triggerPhrase '.*(re)?run kube tests.*'
        extensions {
            'org.jenkinsci.plugins.ghprb.extensions.status.GhprbSimpleStatus' {
                commitStatusContext "Run Kubernetes tests in JVM mode"
            }
        }
    }
```

Výpis 7.16: Zjednodušená verze nastavení GitHub Pull Request Builder triggeru

Blok `configure`[17] je metoda, která umožňuje přímo měnit konfiguraci pipeline job, představující soubor XML. V části kódu `node / 'triggers'` se přistupuje k sekci triggerů. Tato sekce je doplněna o `GhprbTrigger`. Zkrácena definice tohoto triggeru obsahuje:

- `cron` – určuje, jak často se má spouštět trigger, každých 5 minut,
- `orgslist` – určuje organizace projektu GitHub, které můžou aktivovat job pomocí aktivačního komentáře v Pull requestu,
- `triggerPhrase` – text aktivační fráze,
- `commitStatusContext` – název kroku pipeline v GitHub CI,
- `onlyTriggerPhrase 'true'` – aktivace jobu jedině přes trigger frázi.

Pipeline se skládá se dvou fází: konfigurace nastroje `kubectl` a spuštění Kubernetes testu.

Během konfigurace `kubectl` se jako první nastaví URL adresa clusteru Kubernetes[18], `KUBERNETES_URL` je parametr trigger jobu.

```
kubectl config set-cluster kubernetes --server=${KUBERNETES_URL} \
    --insecure-skip-tls-verify=true
```

Nastaví se kontext clusteru[19] a Bearer token[20] `KUBERNETES_API_TOKEN` pro přístup k API, uložený mezi citlivými údaji v Jenkins. Proměnnou `KUBERNETES_API_TOKEN` Jenkins bezpečně nahradí za reálný uložený token.

```
kubectl config set-context qe-user-context --cluster=kubernetes \
    --namespace=default --user=qe
```

[17] https://github.com/jenkinsci/job-dsl-plugin/wiki/The-Configure-Block

[18] https://kubernetes.io/docs/reference/kubectl/generated/kubectl_config/kubectl_config_set-cluster/

[19] https://kubernetes.io/docs/reference/kubectl/generated/kubectl_config/kubectl_config_set-context/

[20] https://kubernetes.io/docs/reference/kubectl/generated/kubectl_config/kubectl_config_set-credentials/

```
# Set credentials for 'qe' user
kubectl config set-credentials qe --token=${KUBERNETES_API_TOKEN}
kubectl config set-context qe-user-context
```

V druhé fázi pipeline probíhá stažení GitHub repozitáře (výpis 7.17) a větve za pomocí Jenkins Git pluginu [18]. Specifikace větve byla upravována podle návodu GHPRB pluginu [17]. `ghprbPullId` se nahradí skutečný identifikátor Pull Requestu, který předá pipeline GHPRB plugin.

```
checkout([$class: 'GitSCM', branches: [[name: 'origin/pr/${ghprbPullId}/merge']],
    extensions: [],
    userRemoteConfigs: [
      [refspec: '+refs/pull/${ghprbPullId}/*:refs/remotes/origin/pr/${ghprbPullId}/*',
       url    : 'https://github.com/quarkus-qe/quarkus-test-framework']]])
```

Výpis 7.17: Načtení GitHub repozitáře a PR větve

Dále probíhá sestavení projektu Quarkus QE Test Framework a spuštění všech Kubernetes testů

```
# Build framework
mvn -B -V clean install -Pframework -Dmaven.repo.local=${MVN_REPO_LOCAL}
# Run tests
mvn -B -V clean install ${MVN_MODULES_APPEND} -fae -Pexamples,kubernetes \
    -Dmaven.repo.local=${MVN_REPO_LOCAL} \
    -Dts.container.registry-url=quay.io/quarkusqeteam \
    -Dkubernetes
```

Kapitola 8

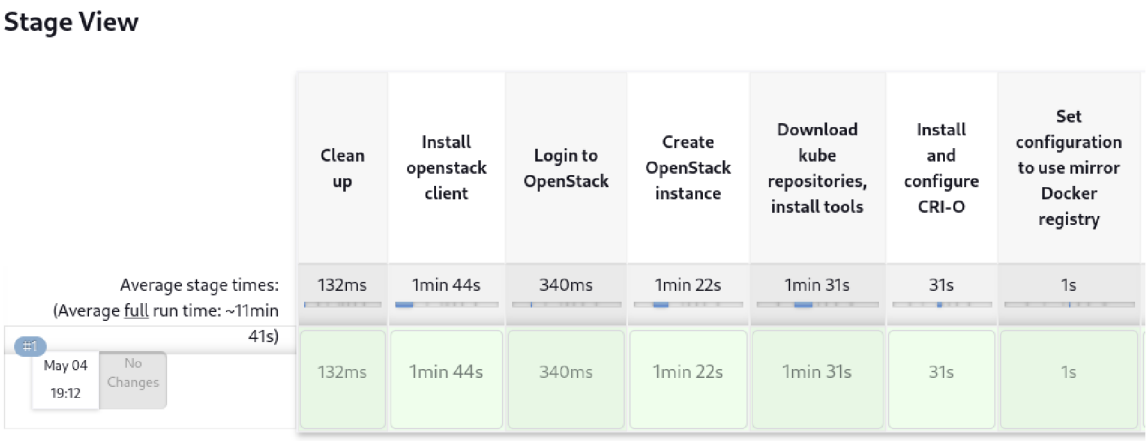
Ověření funkcionality

Když všechny komponenty práce byly implementované, je nutné ověřit jejich funkcionalitu.

Úpravy provedené v Quarkus Test Frameworku byly akceptované mým technickým vedoucím a byly sloučené s hlavni větví projektu ze dvou Pull Requestu: „Fix Kubernetes test scenario“[1] a „Change check of Kubernetes annotation for Keycloak resource binding“[2].

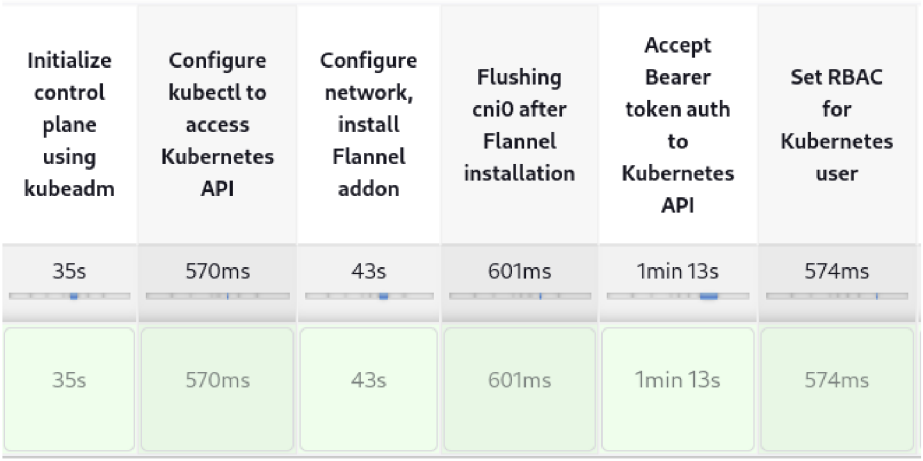
Oba Jenkins jobu se načítají z mého forku privatniho GitLab repozitare Quarkus QE. Tyto joby byly přeneseny do Jenkins pomocí speciálního jobu, která aktualizuje všechny úlohy v Jenkins při sloučení změn do privátního GitLab repozitáře.

Intalační job pro Kubernetes úspěšně vytváří cluster, stav pipeline je mozné vidět na obrázcích: 8.1 a 8.2.



Obrázek 8.1: Výsledek spuštění Jenkins pipeline pro instalaci clusteru Kubernetes č.1

[1] https://github.com/quarkus-qe/quarkus-test-framework/pull/1083
[2] https://github.com/quarkus-qe/quarkus-test-framework/pull/1096



Obrázek 8.2: Výsledek spuštění Jenkins pipeline pro instalaci clusteru Kubernetes č.2

Název virtuálního stroje, do kterého byl instalován cluster Kubernetes je fixně daný `kubernetes-on-jenkins`. Ve výpisu 8.1 probíhá vyhledávání tohoto stroje mezi servery na platformě OpenStack. Zároveň je tady vidět aktuální IP adresou clusteru `10.0.103.4`. Tato IP adresa musí být nastavena jako výchozí parametr `KUBERNETES_URL` v Jenkins trigger jobu.

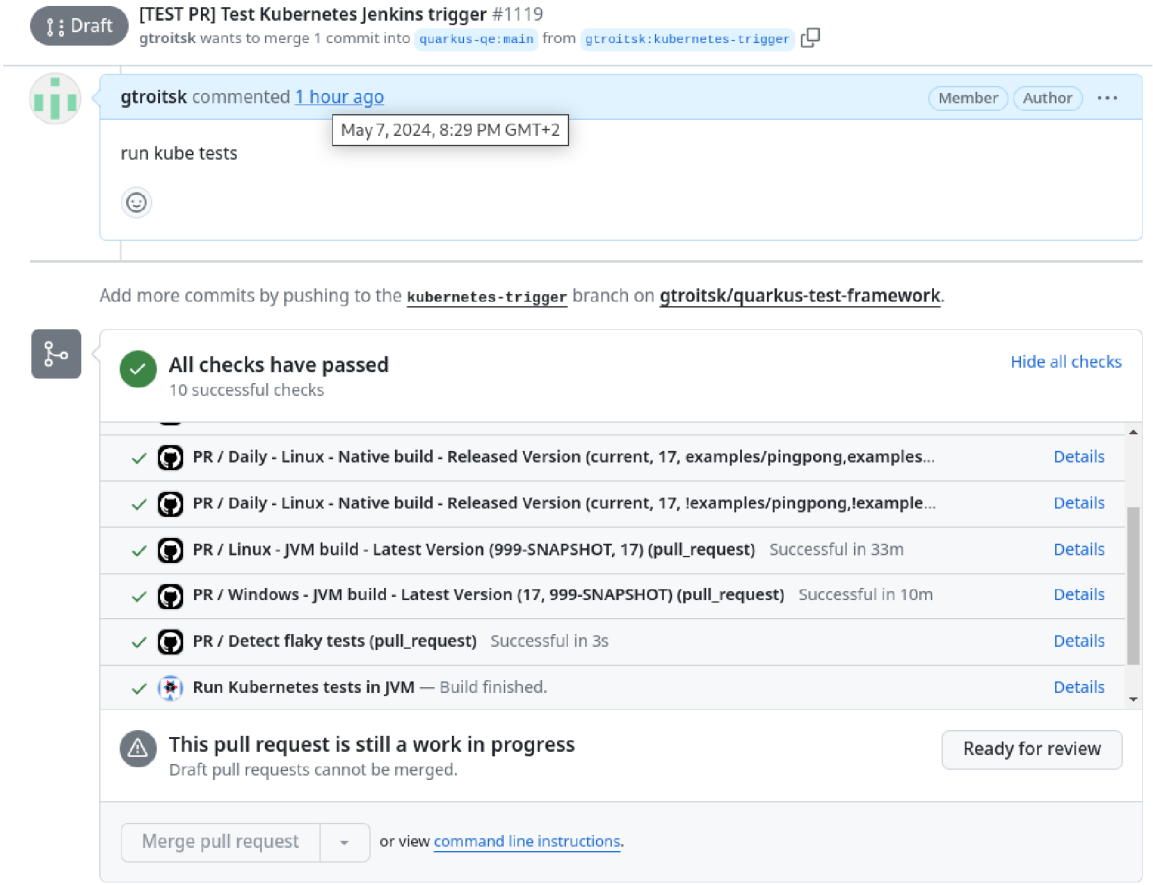
```
[gtroitsk@fedora openstack]$ openstack server list | grep "kubernetes-on-jenkins"
kubernetes-on-jenkins  | ACTIVE | provider_net_cci_8=10.0.103.4 | __QUARKUSQE-SNAPHOT-rhel8
    -x86_64-DEPLOY
```

Výpis 8.1: Manuální získání IP adresy clusterů Kubernetes

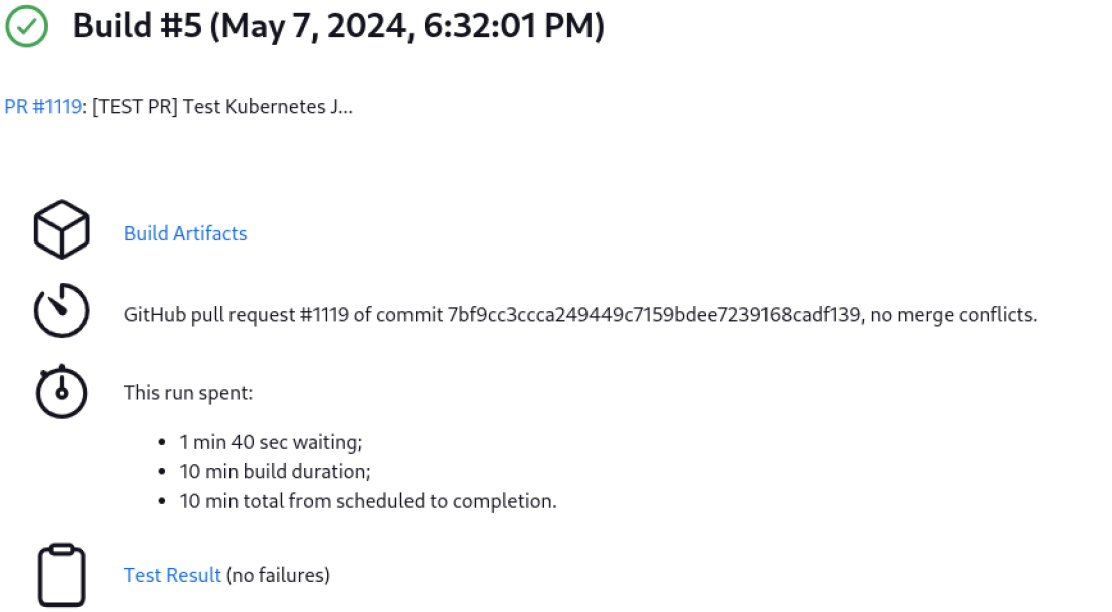
Před testováním trigger pipeline je potřeba vytvořit Pull Request[3] v projektu Quarkus QE Test Framework. Po napsání trigger frázi `run kube tests` v PR během 5 minut se má nastartovat Jenkins job. Obrázek 8.3 ukazuje stav GitHub pipeline po spuštění trigger jobu. Je vidět, ze všechny testy proběhly úspěšně.

Na obrázku 8.4 je zobrazeno úspěšné spuštění trigger jobu z Jenkins. Při porovnání snímku obrazovek GitHub a Jenkins lze říci, že existuje určitá vazba mezi Pull Requestem `#1119` a spuštěným trigger jobem. Job se aktivoval přibližně po třech minutách po zadání trigger fráze. Hodinový čas se neshoduje kvůli jinému nastaveni časového pásma v systému Jenkins.

[3] https://github.com/quarkus-qe/quarkus-test-framework/pull/1119



Obrázek 8.3: Výsledek GitHub CI po spuštění Kubernetes trigger jobu



Obrázek 8.4: Výsledek spuštění Jenkins trigger jobu

Kapitola 9

Závěr

Hlavním cílem této práce bylo rozšířeni funkcionality Quarkus QE Test Frameworku o podporu testovaní aplikaci Quarkus na platformě Kubernetes.

První část práce byla zaměřena na porozumění architektuře Quarkus QE Test Frameworku. Velký důraz byl kladen na pochopení zavádění testovacích scénářů pro spuštění aplikací Quarkus v prostředí bare-metal a cloud-native.

Další teoretická část seznamovala s principem fungování kontejnerů a nástrojů pro jejich správu. Byla prostudována jedna z komponent clusteru Kubernetes - běhové prostředí kontejnerů. Další část byla jedna z nejdůležitějších a uvádí do platformy Kubernetes. Seznamovala s architekturou clusteru, hlavními komponentami uzlů a objekty Kubernetes. Posledním důležitým bodem teorií byl úvod do platformy Jenkins, kde jsem dozvěděl o způsobech vytváření automatických pipeline.

Další část práce spojovala návrh a implementaci. Úprava QQE Test Frameworku vyžadovala aktualizaci použitých metod a změnu strategie vystavení služeb Kubernetes ve prospěch `NodePort`. Zároveň byla přidána podpora pro specifické kontejnery, jako Keycloak, co vyžadovalo studování rozhraní Java Service Provider.

V dalším kroku byla realizována instalační pipeline pro Kubernetes. Tato část byla nejsložitější ze všech, protože požadovala kombinaci několika technologií. Kubernetes cluster se instaloval do virtuálního stroje, běžícího na platformě OpenStack, když pipeline byla spuštěna v Jenkins. Navíc instalace vyžadovala nastavení odrazu registračního serveru Docker a přidání autentizace pomocí Bearer tokenu.

Poslední implementační část se zabývala zprovozněním Jenkins trigger jobu. Výsledkem bylo propojení GitHub Pull Requestu a Jenkins trigger jobu, který následně spouštěl testy Kubernetes z QQE Test Frameworku. Funkčnost celého scénáře byla úspěšně otestována a je možné říct, že požadované vlastnosti byly implementované.

Tato práce se zabývala pouze přidáním podpory Kubernetes pro Quarkus QE Test Framework spolu s nastavením nutné infrastruktury. V práci by ještě někdo mohl pokračovat tak, že by přidal nové Kubernetes testy do Quarkus QE Test Frameworku a do clusteru Kubernetes by nainstaloval UI nástroje jako Kubernetes Dashboard a nástroje pro monitorování zdrojů.

Celkově lze konstatovat, že tato bakalářská práce dosáhla svého cílů - podporu spuštění Kubernetes testu v Quarkus QE Test Frameworku. Výsledné řešení umožňuje plně automatizované spouštění testů v Kubernetes jako součást GitHub CI pipeline, čímž rozšiřuje seznam externích platform, na kterých tým Quarkus QE testuje aplikace Quarkus.

Literatura

[1] Aglave, S. *Most Popular Container Runtimes* [online]. 3. srpna 2023 [cit. 2024-04-22]. Dostupné z: https://www.cloudraft.io/blog/container-runtimes.

[2] Apidog Pte. Ltd. & Anakin AI, Inc.. *What is a Bearer Token?* [online]. 2024 [cit. 2024-04-23]. Dostupné z: https://apidog.com/articles/what-is-bearer-token/#what-is-a-bearer-token.

[3] Bechtold, S., Brannen, S., Link, J. et al. *JUnit 5 User Guide* [online]. 5.10.2. 2024 [cit. 2024-04-16]. Dostupné z: https://junit.org/junit5/docs/current/user-guide/.

[4] Carvajal, J. *Quarkus QE Test Framework* [online]. 26. května 2021 [cit. 2024-04-16]. Dostupné z: https://sgitario.github.io/quarkus-qe-test-framework/.

[5] CRI-O Project Authors. *CRI-O Installation Instructions* [online]. 11. dubna 2024 [cit. 2024-05-01]. Dostupné z: https://github.com/cri-o/cri-o/blob/main/install.md#readme.

[6] Docker Inc.. *Docker overview* [online]. 17. dubna 2024 [cit. 2024-04-21]. Dostupné z: https://docs.docker.com/get-started/overview/.

[7] Docker Inc.. *What is an image?* [online]. 18. dubna 2024 [cit. 2024-04-21]. Dostupné z: https://docs.docker.com/guides/docker-concepts/the-basics/what-is-an-image/.

[8] Ellingwood, J. *What is Kubernetes?* [online]. 3. května 2018 [cit. 2024-04-24]. Dostupné z: https://www.digitalocean.com/community/tutorials/an-introduction-to-kubernetes.

[9] GitHub, Inc. *About pull requests* [online]. 2024 [cit. 2024-05-02]. Dostupné z: https://docs.github.com/en/pull-requests/collaborating-with-pull-requests/proposing-changes-to-your-work-with-pull-requests/about-pull-requests#about-pull-requests.

[10] Jain, K., Rajeev, A. a Srinivasan, M. *Set up Kubernetes on a Red Hat Enterprise Linux system running on IBM Power servers* [online]. 17. února 2022 [cit. 2024-05-01]. Dostupné z: https://developer.ibm.com/tutorials/set-up-kubernetes-on-rhel-running-on-power/.

[11] Kolektiv autorů Fabric8 Kubernetes client. *Fabric8 Kubernetes Java Client Cheat Sheet* [online]. 6. listopadu 2023 [cit. 2024-05-02]. Dostupné z: https://github.com/fabric8io/kubernetes-client/blob/main/doc/CHEATSHEET.md#deployment.

[12] Kolektiv autorů Fabric8 Kubernetes client. *Alternatives to createOrReplace and replace* [online]. 7. února 2024 [cit. 2024-05-02]. Dostupné z: https://github.com/fabric8io/kubernetes-client/blob/main/doc/FAQ.md#alternatives-to-createorreplace-and-replace.

[13] Kolektiv autorů Fabric8 Kubernetes client. *Kubernetes and OpenShift Java Client* [online]. 2024 [cit. 2024-05-07]. Dostupné z: https://github.com/fabric8io/kubernetes-client.

[14] Kolektiv autorů Fabric8 Kubernetes client. *Serialization* [online]. 6.12.1. 2024 [cit. 2024-05-02]. Dostupné z: https://javadoc.io/doc/io.fabric8/kubernetes-client-api/latest/io/fabric8/kubernetes/client/utils/Serialization.html#yamlMapper--.

[15] Kolektiv autorů Flannel. *Deploying flannel manually* [online]. 29. března 2023 [cit. 2024-05-01]. Dostupné z: https://github.com/flannel-io/flannel?tab=readme-ov-file#deploying-flannel-manually.

[16] Kolektiv autorů Flannel. *Flannel* [online]. 2024 [cit. 2024-05-07]. Dostupné z: https://github.com/flannel-io/flannel.

[17] Kolektiv autorů GitHub Pull Request Builder pluginu. *GitHub Pull Request Builder* [online]. 13. února 2021 [cit. 2024-05-02]. Dostupné z: https://plugins.jenkins.io/ghprb/.

[18] Kolektiv autorů Jenkins. *Git* [online]. 2024 [cit. 2024-05-02]. Dostupné z: https://plugins.jenkins.io/git/.

[19] Kolektiv autorů Jenkins. *Pipeline* [online]. 2024 [cit. 2024-04-23]. Dostupné z: https://www.jenkins.io/doc/book/pipeline/.

[20] Kolektiv autorů Jenkins. *Working with projects* [online]. 2024 [cit. 2024-04-23]. Dostupné z: https://www.jenkins.io/doc/book/using/working-with-projects/.

[21] Kolektiv autorů Quarkus. *Quarkus framework* [online]. 2024 [cit. 2024-04-16]. Dostupné z: https://quarkus.io/.

[22] Kolektiv autorů Quarkus QE. *AbstractSqlDatabaseIT* [online]. 2022 [cit. 2024-05-07]. Dostupné z: https://github.com/quarkus-qe/quarkus-test-framework/blob/main/examples/database-postgresql/src/test/java/io/quarkus/qe/database/postgresql/AbstractSqlDatabaseIT.java.

[23] Kolektiv autorů Quarkus QE. *PostgresqlDatabaseIT* [online]. 2023 [cit. 2024-05-07]. Dostupné z: https://github.com/quarkus-qe/quarkus-test-framework/blob/main/examples/database-postgresql/src/test/java/io/quarkus/qe/database/postgresql/PostgresqlDatabaseIT.java.

[24] Quarkus QE tým. *Quarkus QE Test Framework* [online]. 2024 [cit. 2024-04-16]. Dostupné z: https://github.com/quarkus-qe/quarkus-test-framework.

[25] Red Hat, Inc. *What is Quarkus?* [online]. 13. ledna 2020 [cit. 2024-04-16]. Dostupné z: https://www.redhat.com/en/topics/cloud-native-apps/what-is-quarkus.

[26] Red Hat, Inc. *Containers vs VMs* [online]. 13. prosince 2023 [cit. 2024-04-16]. Dostupné z: https://www.redhat.com/en/topics/containers/containers-vs-vms.

[27] Sysdig, Inc.. *What are Container Runtimes?* [online]. 2024 [cit. 2024-04-22]. Dostupné z: https://sysdig.com/learn-cloud-native/container-security/what-are-container-runtimes/.

[28] The Linux Foundation. *Creating a cluster with kubeadm* [online]. 16. února 2022 [cit. 2024-05-01]. Dostupné z: https://v1-29.docs.kubernetes.io/docs/setup/production-environment/tools/kubeadm/create-cluster-kubeadm/.

[29] The Linux Foundation. *Container Runtime Interface (CRI)* [online]. 1. června 2023 [cit. 2024-04-21]. Dostupné z: https://kubernetes.io/docs/concepts/architecture/cri/.

[30] The Linux Foundation. Known issues. *Debugging DNS Resolution* [online]. 18. září 2023 [cit. 2024-05-01]. Dostupné z: https://v1-29.docs.kubernetes.io/docs/tasks/administer-cluster/dns-debugging-resolution/#known-issues.

[31] The Linux Foundation. *Overview* [online]. 19. září 2023 [cit. 2024-04-22]. Dostupné z: https://kubernetes.io/docs/concepts/overview/.

[32] The Linux Foundation. *Using RBAC Authorization* [online]. 5. září 2023 [cit. 2024-05-01]. Dostupné z: https://v1-29.docs.kubernetes.io/docs/reference/access-authn-authz/rbac/.

[33] The Linux Foundation. *About the Open Container Initiative* [online]. 2024 [cit. 2024-04-21]. Dostupné z: https://opencontainers.org/about/overview/.

[34] The Linux Foundation. Static token file. *Authenticating* [online]. 9. dubna 2024 [cit. 2024-05-01]. Dostupné z: https://v1-29.docs.kubernetes.io/docs/reference/access-authn-authz/authentication/#static-token-file.

[35] The Linux Foundation. *Container Runtimes* [online]. 1.29. 15. března 2024 [cit. 2024-05-01]. Dostupné z: https://v1-29.docs.kubernetes.io/docs/setup/production-environment/container-runtimes/#cri-o.

[36] The Linux Foundation. *Deployments* [online]. 14. března 2024 [cit. 2024-04-24]. Dostupné z: https://kubernetes.io/docs/concepts/workloads/controllers/deployment/.

[37] The Linux Foundation. *Installing kubeadm* [online]. 24. dubna 2024 [cit. 2024-05-01]. Dostupné z: https://kubernetes.io/docs/setup/production-environment/tools/kubeadm/install-kubeadm/.

[38] The Linux Foundation. *Kubernetes Components* [online]. 31. ledna 2024 [cit. 2024-04-24]. Dostupné z: https://kubernetes.io/docs/concepts/overview/components/.

[39] The Linux Foundation. *Namespaces* [online]. 3. dubna 2024 [cit. 2024-04-24]. Dostupné z: https://kubernetes.io/docs/concepts/overview/working-with-objects/namespaces/.

[40] Tigera, Inc.. *Kubernetes Networking* [online]. 2024 [cit. 2024-04-27]. Dostupné z: https://www.tigera.io/learn/guides/kubernetes-networking/.

[41] Tigera, Inc.. *What is Project Calico?* [online]. 2024 [cit. 2024-05-07]. Dostupné z: https://www.tigera.io/project-calico/.

[42] Zoho Corporation Pvt. Ltd.. *Understanding the Jenkins Pipeline* [online]. 2024 [cit. 2024-04-23]. Dostupné z: https://www.site24x7.com/learn/jenkins-pipelines.html.