Diplomová práce

F3 Fakulta elektrotechnická
Katedra počítačů

Platformy pro kontejnerizaci aplikací a orchestraci kontejnerů

Bc. Daria Kulynychenko

Vedoucí: Ing. Jiří Šebek
Obor: Softwarové inženýrství
Studijní program: Otevřená informatika
21. května 2021

ZADÁNÍ DIPLOMOVÉ PRÁCE

I. OSOBNÍ A STUDIJNÍ ÚDAJE

Příjmení: **Kulynychenko** Jméno: **Daria** Osobní číslo: **465896**

Fakulta/ústav: **Fakulta elektrotechnická**

Zadávající katedra/ústav: **Katedra počítačů**

Studijní program: **Otevřená informatika**

Specializace: **Softwarové inženýrství**

II. ÚDAJE K DIPLOMOVÉ PRÁCI

Název diplomové práce:

Platformy pro kontejnerizaci aplikací a orchestraci kontejnerů

Název diplomové práce anglicky:

Platforms for application containerization and container orchestration

Pokyny pro vypracování:

Udělejte rešerší existujících platforem pro orchestraci kontejnerů (Kubernetes, OpenShift…)
Zaměřte se na platformy OpenShift a Kubernetes, popište jejich features a ukažte jejich použití
Prozkoumejte způsoby použití známých pluginů pro zlepšení kvality aplikace (logování, čistota kódu apod)
Na příkladě ukázkové aplikace popište nasazení na platformu Kubernetes a následně migraci do Openshift.

Seznam doporučené literatury:

Burns, Brendan, Kelsey Hightower, and Joe Beda. Kubernetes : up and running: dive into the future of infrastructure. Sebastopol, CA: O'Reilly Media, 2019. Print.
Vohra, Deepak. Kubernetes microservices with Docker. Berkeley, CA New York, NY: Apress,Distributed to the Book trade worldwide by Springer, 2016. Print.
Truyen, E.; Van Landuyt, D.; Preuveneers, D.; Lagaisse, B.; Joosen, W. A Comprehensive Feature Comparison Study of Open-Source Container Orchestration Frameworks. Appl. Sci. 2019, 9, 931.
PICOZZI, STEFANO. DEVOPS WITH OPENSHIFT : cloud deployments made easy. O'REILLY MEDIA, 2017. Print.

Jméno a pracoviště vedoucí(ho) diplomové práce:

Ing. Jiří Šebek, kabinet výuky informatiky FEL

Jméno a pracoviště druhé(ho) vedoucí(ho) nebo konzultanta(ky) diplomové práce:

Datum zadání diplomové práce: **18.02.2021** Termín odevzdání diplomové práce: **21.05.2021**

Platnost zadání diplomové práce: **19.02.2023**

Ing. Jiří Šebek podpis vedoucí(ho) práce	podpis vedoucí(ho) ústavu/katedry	prof. Mgr. Petr Páta, Ph.D. podpis děkana(ky)

III. PŘEVZETÍ ZADÁNÍ

Diplomantka bere na vědomí, že je povinna vypracovat diplomovou práci samostatně, bez cizí pomoci, s výjimkou poskytnutých konzultací. Seznam použité literatury, jiných pramenů a jmen konzultantů je třeba uvést v diplomové práci.

. | Datum převzetí zadání

Podpis studentky

Poděkování

Chtěla bych poděkovat svému vedoucímu Ing. Jiří Šebkovi za čas věnovaný této práci, za nádherné téma a morální podporu.

Prohlášení

Prohlašuji, že jsem předloženou práci vypracovala samostatně a že jsem uvedla veškerou použitou literaturu.

V Praze, 21. května 2021

Abstrakt

Tato práce se zabývá problematikou kontejnerizace aplikací a orchestrace kontejnerů. Během projektu byly prozkoumány populární orchestrátory, jejich vlastnosti a výhody. V rámci práce byly podrobněji popsány vybrané nástroje – Kubernetes a OpenShift – a taký ukázána práce s nimi. Praktická část projektu je věnována ukázce použití těchto platforem pomocí nasazení testovací aplikace. Další částí je implementace aplikace pro migraci Kubernetes clusteru do platformy OpenShift a testování fungování pomocí ukázkové aplikace.

Klíčová slova: Kubernetes, cloud, kontejnery, OpenShift, orchestrace, migrace

Vedoucí: Ing. Jiří Šebek

Abstract

This work deals with the issue of containerization of applications and orchestration of containers. Popular orchestras, their features and benefits were explored during the project. The work described selected tools - Kubernetes and OpenShift - and also shows how to work with them. The practical part of the project is devoted to a demonstration of the use of these platforms by deploying a test application. The next part is the implementation of the application for migrating the Kubernetes cluster to the OpenShift platform and testing the functioning using a sample application.

Keywords: Kubernetes, cloud, containers, OpenShift, orchestration, migration

Title translation: Platforms for application containerization and container orchestration

Obsah

Obrázky

Tabulky

Fragmenty kódu

Kapitola 1

Úvod

Cílem této práce je prozkoumat problematiku kontejnerizace aplikací a orchestrace kontejnerů, zaměřit se na dva vybrané orchestrátory a navrhnout způsob migrace aplikace mezi nimi. Důvodů proč se tímto tématem vůbec zabývat je několik. V současné době získává popularitu microservisní architektura, která pro využití všech svých výhod potřebuje co nejvíc automatizovanou správu a orchestraci jednotek aplikace. Dalším důvodem je populární motto „napsáno jednou, spuštěno kdekoliv“, které je popisem fungování systému nezávislého na platformě. Nakonec zvýšení zájmu zákazníků a jejich požadavků na dostupnost a rychlost odezvy aplikace vynucuje velké a malé společnosti vyvíjet robustní a flexibilní systémy, jejichž správa je bez automatizace do budoucna čím dál dražší. V rámci tohoto projektu chci čtenáře přesvědčit, že použití kontejnerů řeší výše uvedené problémy, ale i přináší nové. V následujících kapitolách ukážu, že přestože kontejnery mají své určité výhody, jejich použití bez kvalitního orchestrátoru zvyšuje náročnost provozu aplikace. Pro úvod do problematiky taky krátce popíšu vývoj kontejnerizace, od historie do současných trendů. V kapitole 2.4 udělám přehled populárních poskytovatelů takových služeb, popíšu jejich výhody a důležité vlastnosti. V části 3.1 se zaměřím na platformu Kubernetes, která je v současné době velmi populární a získává více a více fanoušků. Detailně popíšu architekturu orchestrátoru a jeho API. Dále prozkoumám platformu OpenShift a hlavně její rozdíly oproti Kubernetes. V praktické části vyvinu testovací aplikaci, na příkladě které ukážu nasazení do Kubernetes clusteru. Orchestrátor rozšířím o vybrané pluginy pro logování, sledování metrik apod. Nakonec vyvinu aplikaci pro migraci clusteru Kubernetes do platformy OpenShift a pomocí ukázkového projektu tuto funkčnost otestuji.

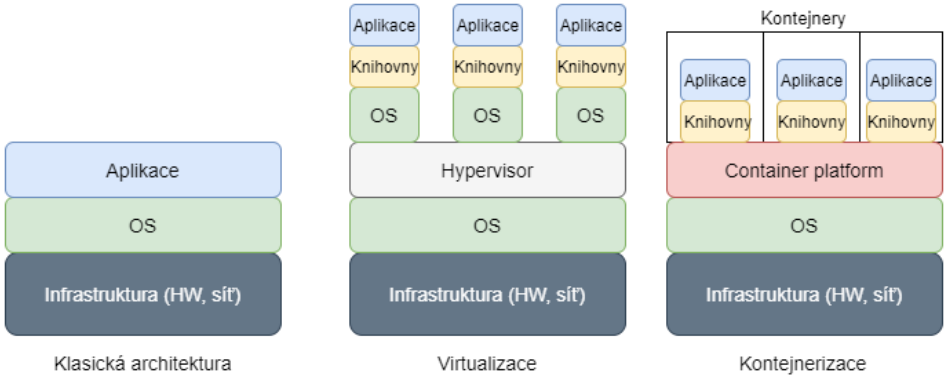
Kapitola 2

Rešerše – virtualizace, kontejnerizace a orchestrace kontejnerů

Tato kapitola slouží úvodem do problematiky. V následujících sekcích prozkoumám problém kontejnerizace aplikace a orchestrace kontejnerů. Na závěr kapitoly udělám přehled populárních platforem, jejich výhod a základních vlastností.

2.1 Historie virtualizace

První koncept aplikačních kontejnerů byl vytvořen v roce 1979 a použit v Unixu verze 7. Jednalo se o příkaz `chroot`, který sloužil pro izolaci procesu metodou změny jeho kořenového adresáře v souborovém systému. Další pokrok ve virtualizaci a kontejnerizaci byl udělán již v roce 2000, kdy byl vyvinut Free BSD Jails, koncept kterého je dost podobný současným Docker kontejnerům. Jaily poskytují větší izolaci než `chroot`, a to tím, že virtualizují nejen souborový systém, ale i množinu uživatelů a síť [1]. Stejně jako předchůdce z Unixu, Jaily dovolí změnit kořenový adresář pro proces. Navíc ale poskytují možnost pro každý kontejner zvolit jinou IP adresu a hostname, nastavit vlastní množinu uživatelů a jejich práva [2]. V roce 2001 podobná technologie se objevila i pro uživatele Linuxu, tzv. LinuxVServer. K dalšímu rozvoji kontejnerizace přispěly technologie Linux Containers (LXC) (vyvinutá v roce 2008) a „Let Me Contain That For You“ (LMCTFY) z roku 2013 od společnosti Google. Tyto dvě platformy byly základem pro současný Docker, který vznikl v roce 2013. Původně Docker používal platformu LXC, potom přešel na vlastní knihovnu *libcontainers*, která vychází a LMCTFY [3]. Právě po vzniku Dockeru kontejnerizace získala největší popularitu.



Obrázek 2.1: Porovnání klasické serverové architektury, virtualizace a kontejnerizace

2.2 Kontejnerizace a aplikační architektury

Mezi klasickou architekturou, virtualizací a kontejnerizací jsou velké rozdíly. Aplikace vyvíjená pro nasazení na tradiční serverovou architekturu je závislá na operačním systému cílového počítače. Tento fakt znamená, že migrace na jiný server s jiným OS vynutí změny v aplikaci. Velikost změn je závislá na typu aplikace a programovacím jazyku. Například v případě Javy změny budou menší, hlavně díky použití JVM, což je Java Virtual Machine, která slouží vestavěným virtuálním strojem a již garantuje platformní nezávislost. Na druhou stranu většina aplikací v jazyce C++ má být zkompilovaná pro každý cílový OS, což přináší spoustu komplikací při údržbě SW.

Virtualizace, ať už pomocí virtuálních strojů, nebo kontejnerů, řeší tento problém, stejně jako i velké množství dalších. V případě VM se nad vrstvou OS (nebo místo této vrstvy v případě nativního hypervisoru) nachází hypervisor, účelem kterého je emulace reálného HW pro použití ve virtuálních strojích. Pak každý VM se jeví jako reálný počítač se svým OS. Výhoda virtuálních strojů je zároveň i jejich nevýhodou – každý z nich potřebuje operační systém, který zabírá místo a zdroje fyzického počítače. Pro vyřešení tohoto problému se používají kontejnery, které mají trochu jiný přístup k virtualizaci. Kontejnery využívají virtualizovaný OS a v sobě mají zabaleno jen aplikaci spolu s její knihovnami a závislostmi, proto jsou kontejnery lehčí a flexibilnější. Hlavním účelem kontejnerizace je to, aby aplikace byly napsány jednou a spuštěny kdekoliv [4].

Porovnání různých způsobů virtualizace s klasickou serverovou architekturou je vidět na obrázku 2.1.

Jak bylo zmíněno dříve v této kapitole, kontejnerizace řeší následující problémy:

1. Přenos aplikace mezi různými prostředími a operačními systémy.
2. Izolace aplikace a hostitelského počítače.
3. Zabezpečení aplikace a serveru pomocí izolace souborového systému.
4. Rychlost a účinnost v porovnání s virtualizací (hlavně díky tomu, že sdílí zdroje a OS hostitelského stroje a nejsou zahlceny touto režií).

Kontejnerizace má i své nevýhody:

1. Kontejnery jsou pomalejší v porovnání s klasickou serverovou architekturou kvůli režiím souvisejícím s propojením hostitelského systému a kontejnerů.
2. Vzhledem k tomu, že Docker a jiné kontejnerizační platformy jsou poměrně nové technologie, potýkají se s bezpečnostními problémy.
3. Ukládaní dat v kontejnerizované aplikaci je mnohem komplikovanější. Kontejnery jsou navržené tak, že při jejich vypnutí se data uložená v samotném kontejneru ztrácí. Z tohoto důvodu častou praxí je ukládání dat mimo kontejner, což samo o sobě porušuje princip izolace a je bezpečnostní dírou.

Ještě jedna věc, která stojí za zmínku, je orchestrace, kterou se zabývá tato práce. Nelze jednoznačně říct, zda se jedná o výhodu nebo nevýhodu kontejnerů. Na jednu stranu orchestrace kontejnerů je komplexní záležitost, která potřebuje velké množství nástrojů. Takové pomocné nástroje řeší například logování, CI/CD, metriky, měření výkonu, kontrolu stavu kontejneru apod. Na druhou stranu orchestrace je silný nástroj, který dovolí vytvořit celý ekosystém pro aplikace různých velikostí, správa kterého bude z velké části automatizovaná a jednodušší než v případě klasické serverové architektury [5].

Kontejnerizace se v současné době často používá spolu s microservisní architekturou [4]. Architektura mikroslužeb spočívá v tom, že velká aplikace je rozdělená na několik funkčních izolovaných jednotek, které komunikují přes běžná rozhraní (REST, SOAP apod.). Každá service má svůj životný cyklus, své prostředí a někdy i odlišný programovací jazyk. Kontejnerizace přináší vývojářům více svobody: každá komponenta, zapouzdřená do samostatného kontejneru, může mít různé závislosti a knihovny, přitom to neovlivní jiné části aplikace. Navíc kontejnerizace a microservisní architektura mají stejný účel – přenos klasické „těžké“ aplikace do lehké, flexibilní, škálovatelné a snadno provozovatelné podoby.

Kromě výhod samotných kontejnerů, microservisní architektura navíc nabízí:

1. Lepší horizontální škálovatelnost aplikace.

2. Snadnou distribuci zodpovědností – například správa různých microservice různými týmy.
3. Přehlednější architekturu aplikace.
4. Snadnější rozšíření funkčnosti aplikace – ve mnoha případech se jedná o přidání nové service s minimálním zásahem do ostatních.

Microservisní architerkura má i své nevýhody (do následujícího seznamu nejsou zahrnuté minusy, které jsou společné nebo vyplývají z použití kontejnerů):

1. Komplikované zajištění konzistence dat. S příchodem microservisní architektury se klasická ACID konzistence mění na eventuální konzistenci. Mezikrokem slouží použití dvoufázového commitu (jedná se o technologii, kde se transakce považuje za úspěšně ukončenou v případě, že každý její účastník úspěšně dokončil své operace). Tento přístup ale zmenšuje propustnost systému a jeho rychlost kvůli dodatečné komunikaci, uzamykaní DB tabulek apod. Řešení tohoto problému jsou podrobněji popsána v kapitole 2.3.2.
2. Nutnost se počítat s tím, že může nastat situace, kdy některé service přestanou fungovat. Může se jednat o výpadek node, chybu v kontejneru, výpadek sítě apod. V případě monolitického systému je možný jen výpadek celé aplikace, což je určitě nepříjemná věc, ale většinou nevede ke ztratě konzistence dat.

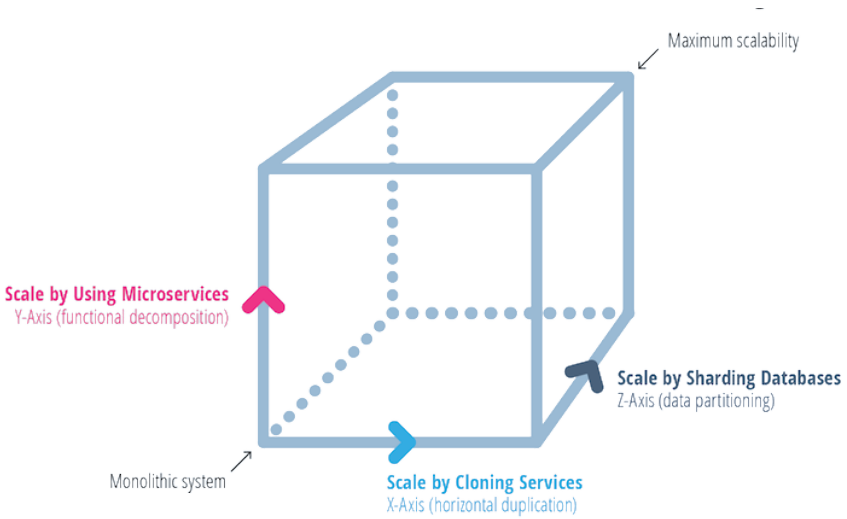
Příkladem úspěšného použití kontejnerů spolu s microservisní architekturou může být streamovací služba Netflix nebo sada aplikací od společnosti Uber [6].

2.3 Současné trendy

V současné době microservisní architektura stává čím dál populárnější. Podle výzkumu, který se prováděl společností JRebel mezi skoro 400 Java programátory, více než půlka vyvíjených aplikací v roce 2020 používala microservisní architekturu [7]. V této kapitole se proto zaměřím na populární nástroje a návrhové vzory, které se používají pro daný typ architektury.

2.3.1 Škálovatelnost v microservisní architektuře

Důležitou vlastností aplikace je její schopnost se přizpůsobovat potřebám uživatelů. Takové požadavky zahrnují i rychlou odezvy systému, jeho schopnost vydržet nárůst zákazníků a stabilní práci i při vysokém zatížení.



Obrázek 2.2: Škálovací kostka [8]

Schopnosti se rozšířit tak, aby mohla aplikace rostoucí zátěž zvládnout, se říká škálovatelnost.

Na škálovatelnost lze koukat z několika pohledů. Existuje dělení na horizontální a vertikální škálovatelnost. Tyhle přístupy se liší z hlediska samotného způsobu rozšíření systému. Takové dělení je použitelné i pro monolitní architekturu, proto je podrobněji popsáno v kapitole 2.4. V této sekci se ale zaměřím na jiné dělení škálovatelnosti – tzv. škálovací kostku[1]. Jak je vidět na obrázku 2.2, lze definovat tři způsoby škálování:

1. Horizontální duplikace, nebo škálování podle osy X. V tomto případě se jedná o replikaci, tj. vytvoření několika stejných instancí aplikace nebo její části.
2. Funkcionální dekompozice, nebo škálování podle osy Y. Při použití daného způsobu se monolitická aplikace rozdělí na součástky podle jejich funkcí. Takové škálování se používá při migraci monolitické aplikace do microservisní architektury.
3. Rozdělení dat[2], nebo škálování podle osy Z. Tento přístup je často používán při škálování databází a spočívá v tom, že se data rozdělí mezi různými instancemi aplikace podle nějakého příznaku.

Škálovatelnost podle osy X řeší problém přetížení systému tak, že se zátěž rozdělí rovnoměrně mezi jednotlivými instancemi aplikace. Je to často požívaný způsob rozšíření monolitického systému. Několik instancí aplikace

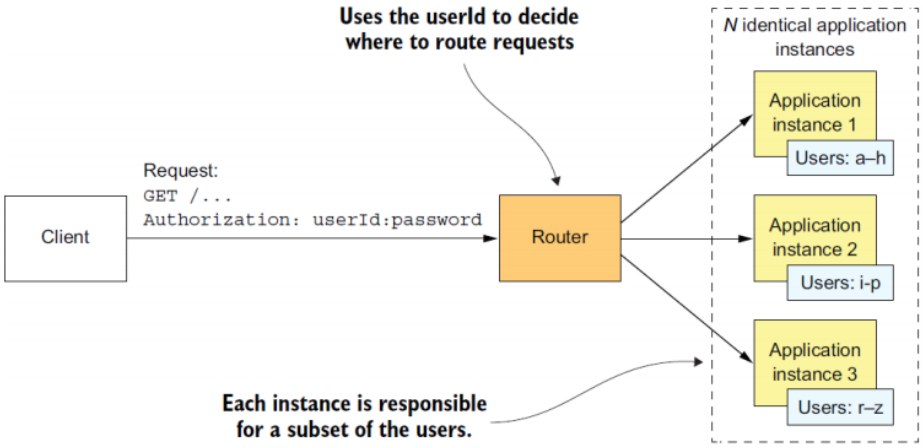
[1] angl. Scale cube
[2] angl. Data partitioning

se spustí za tzv. Load balancerem[3]. Tento nástroj rozdělí zátěž mezi různými jednotkami systému rovnoměrně, například každý požadavek bude postupně posílat na jiný server v definovaném pořadí, tak každá instance dostane stejný počet requestů. Takové rozdělení ale není vždy optimální, protože každý požadavek může trvat jinou dobu a spotřebovávat jiné množství zdrojů. Proto existují různé algoritmy pro rozdělení zátěže [9], které krátce popíšu níže:

1. **Round robin**. Jedná se o výše popsaný algoritmus, kde se požadavky rotují mezi jednotlivými instancemi.
2. **Vážený Round Robin** je modifikací předchozího algoritmu, kdy každá instance se ohodnotí na základě počtu požadavků, které může zpracovat. Používá se v případě, kdy jednotky nejsou úplně stejné, ale mají například odlišný HW.
3. **Least connections**[4]. Požadavek se předá instanci, která má nejmenší počet aktivních spojení. Předpokládá se, že spojení jsou stejně náročná na zdroje.
4. **Vážený Least connections** stejně jako vážený round robin k předchozímu algoritmu přidává ohodnocení instancí na základě jejich schopnosti zpracovat různý počet požadavků.
5. **Založený na zdrojích (adaptivní)**. Algoritmus spočívá v tom, že se load balancer před přesměrováním požadavku ptá na metriky každé instance, aby zjistil její aktuální zatížení. Jednotka s nejmenším zatížením dostává požadavek. Algoritmus vyžaduje nějaký nástroj pro sběr takových metrik.
6. **IP hash.** Tento algoritmus má různé modifikace. Někdy se pro výpočet hashe používá jen zdrojová a cílová IP adresy [10], v jiných případech základem hashe je protokol, zdrojové a cílové IP adresy a porty, sekvenční číslo paketu a podobné [11]. Hlavním cílem algoritmu je v případě selhání spojení vrátit uživatele na stejnou instanci.
7. **Rozdělení na základě požadavku**[5] bere v úvahu dobu zpracování a spotřebu zdrojů pro každý požadavek a na základě toho rozdělí requesty mezi instancemi [10].

Existují i jiné algoritmy pro load balancer, speciální pro každého poskytovatele této služby. Také někdy existuje možnost definovat vlastní pravidla na základě specifika jednotlivých aplikací [11]. Výběr vhodného algoritmu vždy záleží na konkrétním systému, poskytnutých zdrojích a vlastnostech požadavků.

[3] česky doslovně vyvažovač zátěže
[4] česky Nejméně spojení
[5] angl. Request specific distribution



Obrázek 2.3: Příklad škálování podle osy Z [5]

Škálování podle osy Z je podobné horizontální duplikaci s tím rozdílem, že rozdělení je založeno na datech, nikoliv na zatížení systému. Podobně jako v případě osy X existuje několik stejných instancí aplikace. Tyto jednotky běží ne za load balancerem, ale za routerem, který na základě konkrétních dat v requestu pošle požadavek na určitou instanci. V příkladu na obrázku 2.3 se posílá uživatelské jméno, na základě kterého router určí, na kterou instanci požadavek přesměrovat. Každá jednotka aplikace je zodpovědná za uživatele s jiným rozsahem jmen. Nevýhodou takové metody škálování je to, že většinou data nejsou rovnoměrně rozdělená. Například aplikace nemusí mít stejný počet uživatelů s loginem začínajícím se v rozsahu a až h a v rozsahu i až p. Pak takové rozdělení zatíží jednu instanci, ale druhá bude běžet naprázdno. Proto existuje několik způsobů rozdělení [12], ze kterých na základě předpokládaných dat je nutné vybrat nejlepší pro danou aplikaci:

1. **Dělení podle hash** – hashovací funkce se aplikuje na atribut nebo několik atributů požadavku a na základě výsledku se požadavek přesměruje na zodpovědnou instanci. Platí, že požadavky se stejnou hash se dostanou na stejný server.
2. **Dělení podle rozsahu**. Tenhle způsob je představen v příkladu na obrázku 2.3, kde se data rozdělí podle uživatelského jména.
3. **Dělení podle seznamu** je dost podobné předchozímu způsobu s tím rozdílem, že pro rozhodnutí se nepoužívá rozsah, ale výčet hodnot.
4. **Round robin dělení** určí pořadí dat (například v databázové tabulce) a každý i-tý záznam přiřadí $(i \bmod n)$-té instanci, kde n je počet všech instancí.

5. **Kompositní dělení**. Takový přístup kombinuje výše uvedené způsoby. Typicky se používá dělení podle rozsahu spojené s dělením podle hash.

Na rozdíl od předchozích typů škálování, které mohou být aplikovány jak na microservisní architekturu, tak i na monolitickou, funkcionální dekompozice je rozdělení monolitické aplikace na jednotlivé služby. Horizontální duplikace a data partitioning se zabývají problémy kapacity a dostupnosti aplikace, když škálování podle osy Y řeší problém zvětšení komplexity vývoje a rozšíření funkčností systému. Takový přístup dovolí nejen zpřehlednit kód, ale i distribuovat vývoj mezi několika týmy. Funkcionální dekompozice se často používá spolu s ostatními způsoby škálování, kde každá služba může mít více instancí v závislosti na předpokládané zátěži [5].

2.3.2 Vybrané návrhové vzory microservice

Škálovatelnost a rozdělení zátěže řeší problém dostupnosti aplikace. Existují ale i jiné nevýhody microservisní architektury, zmíněné v předchozí kapitole, například inkonzistence dat nebo selhání jednotlivých součástí aplikace. Pro řešení takových problémů se používají návrhové vzory.

Obecná definice návrhových vzorů říká, že jsou to šablony, které řeší ten či jiný problém [13]. Existuje navíc pojem jazyk vzorů[6], což je množina návrhových vzorů, kde každý její prvek řeší problém ve stejné specifické oblasti [14]. Taková množina je definovaná i pro microservisní architekturu a zahrnuje velké množství šablon pro zabezpečení, dekompozici, testování, dotazování, komunikaci a jiné [15]. V této kapitole se podívám na vybrané vzory z tohoto jazyku, které řeší nevýhody microservisní architektury.

Spolehlivost služeb

Spolehlivost je velkou otázkou pro microservisní architekturu. Pro připomenutí problému zopakuji jednu z nevýhod microservice z předchozí kapitoly, a to takovou, že jelikož každá služba běží jako samostatná jednotka, je dost pravděpodobný výpadek jedné nebo více service. Pro příklad si představíme, že jednou ze služeb aplikace je platební brána, která zpracovává dotazy z jiných service v systému. Najednou se může stát, že je služba přehlcena požadavky a nemá zdroje na jejich zpracování. Zdrojové service ale posílají požadavky dál, čímž nedají možnost této platební bráně se obnovit. Navíc kvůli zvětšení času odpovědí na požadavek jiné služby spotřebují více a více zdrojů, a v důsledku mohou taky zkrachovat. Takový problém kaskádového selhání celé aplikace řeší circuit breaker pattern.

[6] angl. Pattern language

V komunikační řetězec mezi jednotlivými service se doplní circuit breaker, který se chová následně: při velkém počtu selhání jedné ze služeb tento prvek přestává posílat na ni dotazy a rovnou všem jiným službám odpovídá chybou. Po nějaké uživatelem definované době circuit breaker jeden požadavek pošle na spadlou service, a pokud bude úspěšný, tak komunikaci znovu otevře. V jiném případě pokračuje v odesílání odpovědí s chybou.

Z výše uvedeného chování lze odvodit, že circiut breaker se může nacházet ve třech stavech:

1. **Zavřený**. Přestože název tohoto stavu je trochu matoucí, zavřený circuit breaker povolí libovolnou komunikaci mezi službami. Při každém požadavku ale kontroluje stav odpovědi, a pokud chyby překročily uživatelem stanovený limit, přechází do stavu **Otevřený**.
2. **Otevřený**. V tomto stavu circuit breaker místo posílání zpráv na spadlou službu vrací chybu sám (nebo vyvolá fallback funkci).
3. **Polootevřený**. Do tohoto stavu circuit breaker přejde po nějaké době ze stavu **Otevřený**. Pošle jeden nebo několik požadavků na spadlou službu, a pokud dostane chybu, tak se vrátí do stavu **Otevřený**, jinak se zavře a dovolí službám mezi sebou komunikovat.

Výhodou takového přístupu je to, že jednak zahlcená služba má čas se obnovit, ale taky jiné jednotky nebudou spotřebovávat nadměrné zdroje při čekání na odpověď [16].

Konzistence dat

V monolitické architektuře je konzistence dat většinou zajištěná pomocí ACID transakcí. Microservisní architektura má ale dva faktory, které dělají nemožným požití ACID. Prvním z nich je to, že business transakce může zahrnovat zpracování ve více službách, každá z kterých má vlastní lokální databázovou transakci. Druhým faktorem je doporučený návrhový vzor z jazyku vzorů pro microservisní architekturu [15], a to tzv. Database per service, což český znamená mít separátní databáze pro různé služby aplikace.

V předchozí kapitole jsem krátce naznačila, že řešením takového problému může být dvou-fázový commit. Tento návrhový vzor se skládá ze dvou částí. První fáze je příprava, kdy každá služba dělá vlastní logiku v lokální transakci a reportuje stav své transakce řídicímu prvku. Ve druhé fázi řídicí prvek vyhodnocuje odpovědi ze všech service, a pokud byly všechny lokální transakce úspěšné, data celé business transakce budou zapsány do databáze. Jinak celá business transakce skončí chybou a žádné změny nebudou zapsány. Dvou-fázový commit zaručí ACID vlastnosti, včetně konzistence dat, ale na druhou

stranu má své nevýhody. Hlavním problémem je to, že samotný návrhový vzor počítá se synchronním zpracováním business transakce. To znamená, že všechny účastníky transakce čekají na její dokončení. V prostředí microservice může být takové čekání dlouhé vzhledem k tomu, že služby nekomunikují v paměti, ale většinou v síti [17].

Z výše uvedených důvodů se v microservisní architektuře často používá eventuální konzistence. Hlavní myšlenkou takového přístupu je to, že systém nemusí být v každém okamžiku konzistentní a musí být schopen v takovém stavu pracovat. S eventuální konzistencí těsně souvisí jeden prvek z jazyku vzorů pro microservisní architekturu – Saga pattern.

Saga je založená na asynchronním zpracování a událostech. Hlavní myšlenkou je to, že služby nemusí čekat na dokončení lokální transakce jiné service. Místo toho každá služba reaguje na příslušné události, zpracovává data a po dokončení může publikovat vlastní událost. Pokud jedna z lokálních transakcí byla neúspěšná, vyhodí se chybová událost, na kterou zareagují ostatní služby a vytvoří tzv. kompenzační transakce (jednoduše řečeno, je to manuální rollback změn). Do momentu ukončení úlohy v poslední službě business transakce data jsou v nekonzistentním stavu, jelikož každá služba reálně zapíše data do databáze, tedy informace budou přístupné pro čtení. Proto lze říct, že Saga podporuje jen eventuální konzistenci, nikoliv ACID transakce.

Saga může být implementována dvěma způsoby: pomocí orchestrace nebo choreografie. Orchestrovaná Saga má jednu službu navíc – orchestrátor – která řídí celou business transakci a reaguje na úspěšné zpracování nebo chyby jednotlivých lokálních transakcí. V tomto případě se nepoužívají události, ale příkazy, které orchestrátor posílá jednotlivým službám. Saga s choreografií je řízená událostmi, a samotné služby implementují logiku posílání a reakci na ně [18].

Výhodou Sagy je možnost asynchronního zpracování business transakce. Služby nejsou zatížené čekáním na dokončení jiných lokálních transakcí. Na druhou stranu je Saga složitá pro implementaci hlavně tím, že u větších transakcí přibude logika navíc pro vyhodnocení událostí nebo příkazů.

Objevování služeb

Jádrem microservisní architektury je komunikace jednotlivých služeb mezi sebou. Při lokálním testování nebo nasazení na vlastní servery není problém tyto služby propojit napřímo pomocí IP adresy. Ale v případě externího hostingu propojení služeb vyžaduje dodatečné konfigurace a náklady na údržbu. Problém se zvětšuje v případě existence několika replik služby, navíc když počet těchto instancí se dynamicky mění.

S řešením přichází jeden prvek z jazyku vzorů pro microservisní architekturu, a to service discovery. Jádrem tohoto vzoru je service registry, což je databáze, kde se ukládá aktuální adresa jednotlivých služeb. Je možné říct, že service discovery funguje podobně DNS protokolu: klient se ptá na umístění konkrétní služby podle nějakého aliasu, a service registry vrátí IP adresy dostupných služeb s tímto aliasem. Jednotlivé service se registrují do service registry a s nějakým uživatelem definovaným intervalem posílají heartbeaty – jednoduché požadavky, které ukazují, že systém je funkční. Služba se vymaže ze service registry, pokud se odregistruje nebo nepošle heartbeat.

Existují dva druhy objevování služeb: z klientské nebo serverové strany[7]. V prvním případě klient napřímo komunikuje se service registry a zjistí adresy všech dostupných instancí služby. Následně vlastním algoritmem rozdělení zátěže si zvolí službu, na kterou nakonec pošle dotaz. Příkladem takového objevování služeb je Netflix OSS[8]. Jeho komponenta Netflix Eureka je service registry, která poskytuje REST API pro objevování služeb. Spolupracuje s ní Netflix Ribbon, který slouží load balancerem a rozděluje zátěž mezi instancemi aplikace. V případě objevování služeb ze serverové strany klient posílá požadavek na load balancer, který dělá dotaz na service registry, přeposílá data na jednu z instancí služeb a vrátí odpověď. Tím je klient úplně abstrahován od implementace objevování služeb a ani neví, že takových instancí je víc. Příkladem je Amazon Elastic Load Balancer [9], který je zároveň load balancerem a service registry. Ten přijímá požadavky ze vnější sítě a posílá je na jednu ze zaregistrovaných instancí aplikace. Každý z těchto přístupů má své výhody a nevýhody. V případě objevování z klientské strany plusem je to, že lze používat inteligentnější logiku rozdělování zátěže podle samotného požadavku. Na druhou stranu každý klient tuto logiku musí implementovat. Pro objevování ze serverové strany je výhodou to, že klient je od výběru instance abstrahován a rozdělování zátěže funguje jednotně pro všechny zákazníky. Minusem je to, že navíc k service registry, load balancer taky stává kritickým místem poruchy systému[10].

Registraci služeb do service registry lze taky rozdělit na dva druhy: samoregistrace a registrace pomocí třetí strany. V případě samoregistrace každá služba je zodpovědná za to, aby se zapsala do service registry a posílala heartbeaty. Při registraci pomocí třetí strany se přidává ještě jedna komponenta, která jediná napřímo komunikuje se service registry. Navíc tato služba monitoruje své prostředí a v případě objevu nové service ji automaticky zaregistruje. Zodpovědnost kontrolovat stav služeb se taky předává třetí straně [19].

[7] angl. client-side discovery, resp. server-side discovery
[8] Dostupné z: `https://netflix.github.io/` (24.12.2020)
[9] Dostupné z: `https://aws.amazon.com/ru/elasticloadbalancing/` (24.12.2020)
[10] angl. Single point of failure

2.4 Orchestrace kontejnerů

Jedna z otázek, kterou řeší táto práce, je proč používat kontejnery a orchestrátory. Předchozí kapitola byla věnována popisu kontejnerizace a její výhod, hlavně pro fáze vývoje a nasazení. Jinými důležitými částmi životního cyklu aplikace jsou údržba, podpora a rozvoj. S časem se aplikace rozšíří o nové funkčnosti a integrace, získá zájem většího množství uživatelů a přitom musí poskytovat rozumnou rychlost odezvy a dostupnost služeb. Zátěž aplikace se řeší pomocí škálování, buď vertikálního, nebo horizontálního. Vertikální škálovatelnost se dosáhne zlepšením HW, na kterém aplikace běží: použitím modernějších součástek, rozšířením operační paměti apod. Ale rozšíření HW má své limity, kdy velikost další investice nebude odpovídat výkonu, který se získá úpravou. Dalším typem rozšíření je horizontální škálovatelnost. Jedná se o přidání prvků systému pomocí replikace nebo rozdělení dat (podrobnější popis těchto typů škálování je uveden v kapitole 2.3.1).

Z pohledu údržby je vertikální škálovatelnost více limitovaná, ale jednodušeji udržovatelná. Horizontální škálovatelnost má skoro nekonečné možnosti, ale přináší problémy spojené s rozdělením zátěže mezi jednotlivými replikami. Externí služba, která komunikuje s replikovaným prvkem, musí vědět, na kterou adresu má poslat dotaz. Adresou se v tomto případě myslí IP adresa, použití které způsobí další problém. Při nasazení aplikace do jiného prostředí není zajištěno, že síť IP adres zůstane stejná, což znamená, že při libovolné migraci se musí provést zásah do aplikace.

Po vyřešení výše uvedených problémů se objeví další, například nutnost definovat chování systému v případě výpadku jedné z replikovaných jednotek nebo obecně nějaké služby. V tomto případě musí předchozí algoritmus pro rozdělení zátěže přebudovat celou síť a zároveň se musí služba obnovit. Dalším problémem může být nasazení nové verze služby, zvlášť u kritických systémů, kde dostupnost má být maximální a kde vypnutí celého systému pro nasazení jedné komponenty není možné.

Orchestrátor kontejnerů je nástroj, který řeší výše uvedené problémy. Dovolí uživateli nakonfigurovat požadované chování systému ve výše zmíněných případech a správu kontejnerů automatizuje podle konfigurace. Orchestrátory poskytují následující služby (názvy služeb se mohou lišit v závislosti na použité platformě, ale jejich účel zůstává stejný):

1. Vytvoření požadovaného počtu replik služby.
2. Load balancing, který řeší problém rozdělení zátěže mezi replikami.
3. Service discovery, který místo použití IP adres nabídne uživatelsky přívětivé názvy služeb.

4. Automatické opravy pro lepší dostupnost, které zahrnují restart vypadlých jednotek, nastartování služby na jiném HW apod.

5. Rollout update, který jako i předchozí bod přispívá vysoké dostupnosti a spočívá v tom, že při aktualizaci aplikace se služby mění postupně.

Detailnější popis některých z výše uvedených služeb se nachází v kapitole 2.3.

Kromě řešení výše uvedených problémů orchestrátory nabízejí další služby, které se liší na základě zvolené platformy. Většinou se jedná o sledování metrik, logování, integrace s externími systémy apod [5]. V další podkapitole se nachází přehled populárních platforem pro orchestraci kontejnerů.

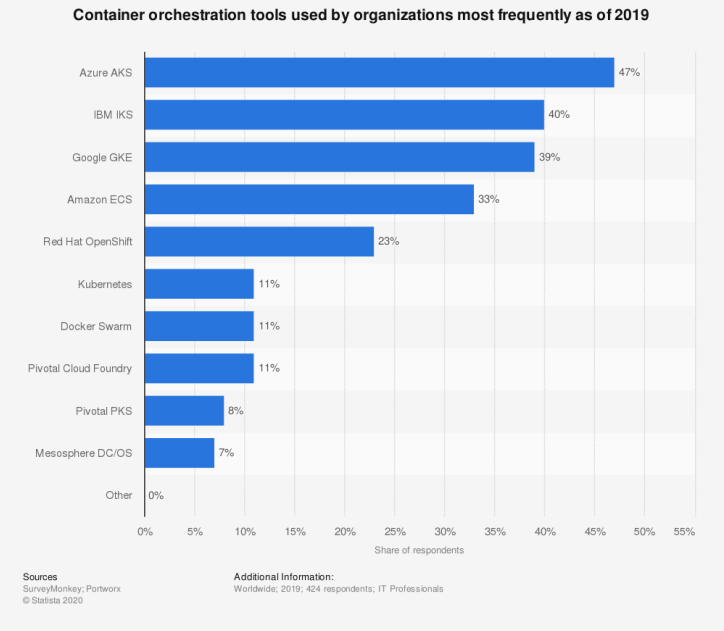
2.4.1 Přehled populárních platforem pro orchestraci kontejnerů

Podle webu Statista[11] mezi nejpopulárnějšími nástroji pro orchestraci kontejnerů mezi společnostmi v roce 2019 patří Azure AKS[12], IBM IKS[13], Google GKE[14], Amazon ECS[15], Red Hat OpenShift[16] a Kubernetes[17] (obrázek 2.4). V této kapitole se zaměřím na jejich základní architekturu, výhody a poskytované služby.

Kubernetes a Kubernetes-as-a-Service

Většina z výše zmíněných platforem (kromě AWS ECS a samotného Kubernetes) jsou poskytovatele KaaS (Kubernetes-as-a-Service). Kubernetes je platforma pro orchestraci kontejnerů s otevřeným zdrojovým kódem. Účelem tohoto nástroje je automatizace mnoha procesů zapojených do nasazení, údržby a škálování kontejnerizované aplikace, její sítě a úložišť. Detailnější popis architektury a možností Kubernetes bude následovat v kapitole 3.1. Kubernetes není tradiční systém typu Platform-as-a-Service. Poskytuje mnohem méně funkcí, například nativně nepodporuje CI/CD pipelines, nenabízí vestavěný middleware pro aplikace (databáze, frameworky) a nepodporuje komplexní konfigurace a nástroje pro správu HW. Jedním z důvodů je to, že Kubernetes pracuje na úrovni kontejnerů, nikoliv HW. Dalším důvodem je

[11] Dostupné z: `https://www.statista.com/` (04.11.2020)
[12] Dostupné z: `https://azure.microsoft.com/cs-cz/services/kubernetes-service/` (04.11.2020)
[13] Dostupné z: `https://www.ibm.com/cloud/container-service/` (04.11.2020)
[14] Dostupné z: `https://cloud.google.com/kubernetes-engine` (04.11.2020)
[15] Dostupné z: `https://aws.amazon.com/ru/ecs/` (04.11.2020)
[16] Dostupné z: `https://www.openshift.com/` (04.11.2020)
[17] Dostupné z: `https://kubernetes.io/` (04.11.2020)

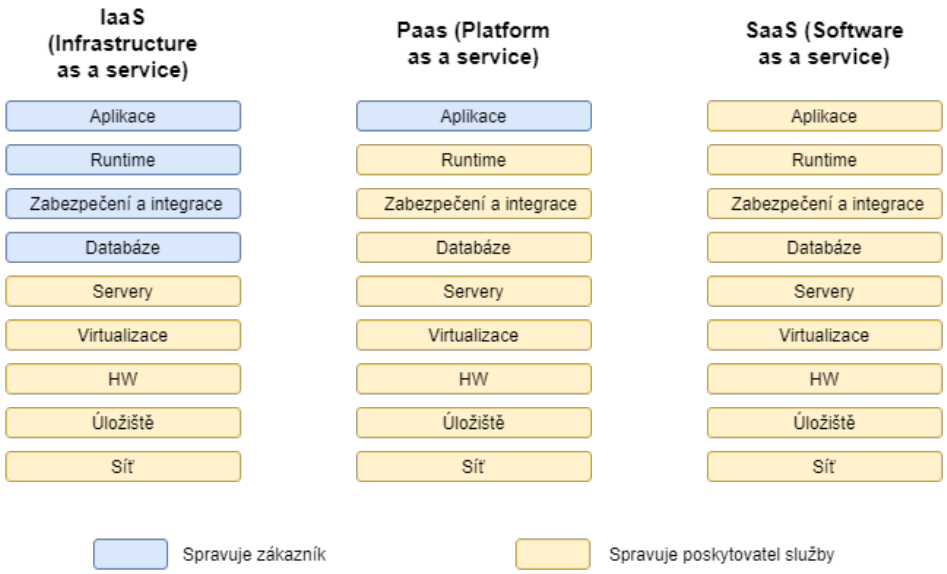


Obrázek 2.4: Nejpopulárnější nástroje pro orchestraci kontejnerů, 2019 [20]

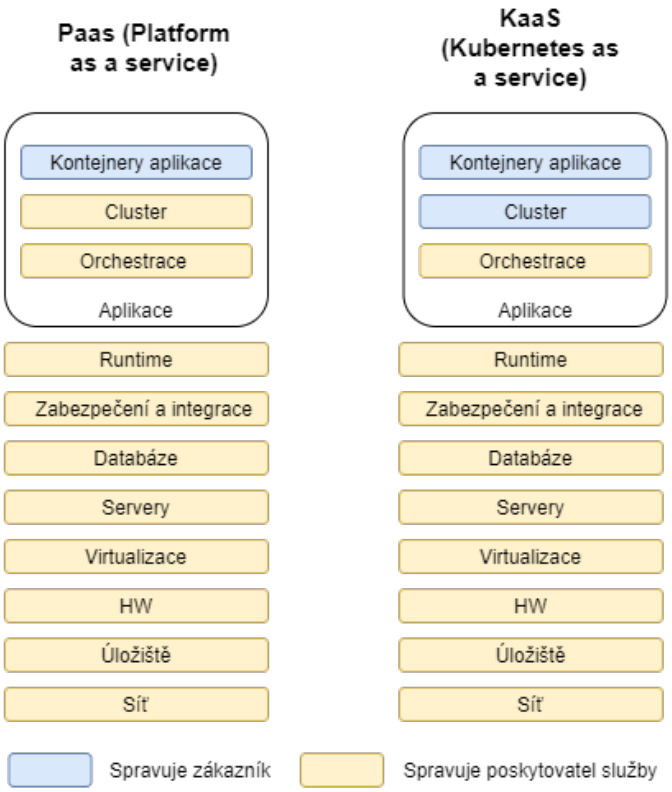
záměr vývojářů K8s poskytnout programátorům možnost mít lehký nástroj pro orchestraci kontejnerů, který se pak dá konfigurovat pomocí pluginů podle potřeb jednotlivých aplikací [21].

Kubernetes-as-a-Service je systém, který zapouzdřuje pouhý Kubernetes a poskytuje některé funkce navíc. Většinou se jedná o automatizovanou konfiguraci Kubernetes clusteru, snadné připojení pluginů, správu infrastruktury, sítě a úložišť [22]. Na rozdíl od PaaS, kde hlavní jednotkou abstrakce je aplikace, operátor KaaS spravuje clustery Kubernetes [23].

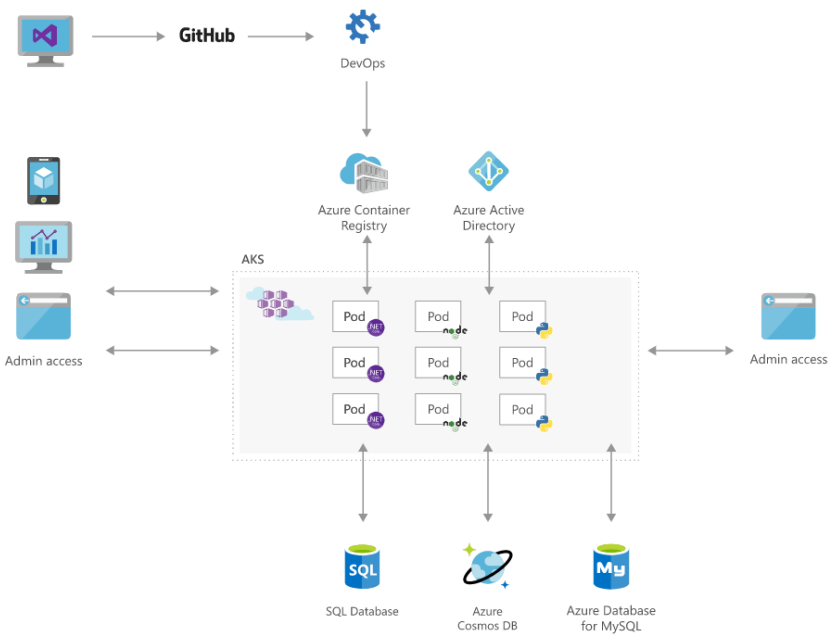
Na obrázku 2.5 je vidět klasické porovnání tří systémů: IaaS, PaaS a SaaS. Liší se tím, za kterou část infrastruktury je zodpověděn uživatel, a kterou spravuje poskytovatel služby. V případě klasické monolitické architektury v PaaS uživatel spravuje jen vlastní aplikaci, ostatní součásti infrastruktury jsou poskytnuté jako služby. Veškeré aktualizace, opravy a údržbu zaručí poskytovatel. V případě kontejnerizovaného systému se jeden malý obdélník *Aplikace* rozdělí na několik částí: kontejnery, cluster a orchestrace (obrázek 2.6). V případě PaaS (pokud provider umožní nasadit kontejnerizovanou aplikaci) uživatel má možnost spravovat jen samotné kontejnery. Obvykle cluster a orchestraci poskytuje provider a umožňuje jen minimální zásah do konfigurace. V případě KaaS provider poskytuje možnost uživateli řídit cluster a upravovat ho podle vlastních potřeb. Určitě na výše zmíněném obrázku 2.6 je možné změnit *Kubernetes* na obecné *Containers*, a stále bude porovnání platit.



Obrázek 2.5: Porovnání IaaS, PaaS a SaaS



Obrázek 2.6: Porovnání PaaS a KaaS



Obrázek 2.7: Propojení AKS a jiných systémů Microsoft Azure [25]

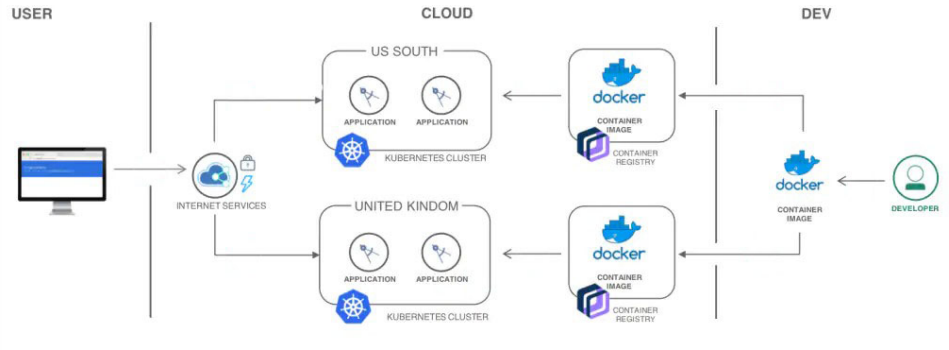
Azure Kubernetes Service (AKS)

AKS je služba typy Kubernetes-as-a-Service, která poskytuje podporu Kubernetes architektury ve svém cloudu. Jedná se o orchestrátor od společnosti Microsoft, který je součástí platformy Microsoft Azure. Mezi zákazníky patří firmy Bosch, Siemens a Hafslund [24].

Jak je vidět na obrázku 2.7, AKS se dá propojit s jinými nástroji platformy Azure. Po pushnutí kódu do GitHub service Azure DevOps[18] zahájí build aplikace a případně její testování (záleží na nastavení CI pipeline). Potom bude aplikace odeslána do Azure Container Registry[19], odkud se nasadí do clusteru. AKS Azure nabízí také různé databáze pro ukládání dat. K aplikaci uživatelé a správce mohou přistupovat přes webové rozhraní. Zabezpečení aplikace zajišťuje Azure Active Directory[20], což je nástroj pro správu identit a přístupů k různým prostředkům (jednoduše řečeno je nástrojem pro autentizaci a autorizaci). Tím Azure spolu s AKS zajišťuje celou infrastrukturu pro vývoj kontejnerizovaných aplikací [26].

Popularita Azure Kubernetes Service je dána několika faktory. Za prvé,

[18] Dostupné z: `https://azure.microsoft.com/cs-cz/services/devops/` (16.11.2020)
[19] Dostupné z: `https://azure.microsoft.com/cs-cz/services/container-registry/` (16.11.2020)
[20] Dostupné z: `https://azure.microsoft.com/cs-cz/services/active-directory/` (16.11.2020)



Obrázek 2.8: Nasazení aplikace do IBM Cloud Kubernetes clusteru [29]

Azure nabízí firmám výhodnou cenovou politiku. Společnost platí jen za node, ve kterých běží její aplikace, master node (správce infrastruktury) má potom zcela zadarmo. Za druhé, Kubernetes v AKS je plně spravovaný firmou Microsoft, proto nemusí uživatel mít hluboké znalosti architektury K8s. Nakonec Microsoft poskytuje celou infrastrukturu Azure s velkým množstvím služeb a možností jejich integrace s AKS. Řešení nabízí uživateli celou síť služeb, které zajistí snadnou údržbu, zabezpečení, správu, kontrolu a škálovatelnost aplikace.

IBM Cloud Kubernetes Service (IKS)

IBM Cloud Kubernetes Service je certifikovaným poskytovatelem Kubernetes [27]. Nabízí možnost založit vlastní K8s nebo OpenShift cluster a automatizovat jeho nasazení a správu. IKS zajistí dostupnost aplikace a samoléčení kontejnerů, poskytuje služby vyvážení zátěže, automatizovaného nasazení, správy zabezpečení a monitorování clusteru. Mezi zákazníky patří The Weather Company od IBM, Eurobits Technologies a Think research [28].

Na obrázku 2.8 je zobrazen diagram nasazení do prostředí IBM IKS. Aplikace má být kontejnerizovaná a odeslána do registru kontejnerů. Následně může být aplikace nasazená do clusteru, přičemž IBM podporuje nasazení do vícero regionů pro lepší dostupnost a rychlost připojení. Navíc IBM Cloud poskytuje dvě možnosti nasazení clusterů – klasické a VPC (Virtual Private Network[21]). Klasické clustery jsou nasazeny do veřejného cloudu, když to VPC clustery běží v izolovaném prostředí. Jejich porovnání najdete v tabulce 2.1.

IBM nabízí dvě varianty clusterů z cenového hlediska. Bezplatná verze podporuje jen K8s clustery a klasickou infrastrukturu, omezuje počet pracovních uzlů do jednoho a nabízí omezené výpočetní prostředky. Uživatel dostane na svůj účet pevný počet kreditů (měna v IBM Cloud) pro integraci s

[21] česky Virtuální privátní síť

	Klasická infrastruktura	VPC
Výběr výpočetních strojů	Podporuje všechny 3 dostupné typy strojů: virtuální stroj, bare metal[1] a softwarově definované úložiště[2].	Podporuje jenom virtuální stroje.
Zabezpečení	Nabízí velké množství zabudovaných služeb pro zabezpečení clusteru a izolaci prostředků.	VPC běží v izolovaném prostředí a v privátní síti. Navíc nabízí Network Access Control Lists (seznam pro řízení přístupů v síti) pro zabezpečení podsítí.
Síť	Pracovní uzly běží v soukromých, nebo veřejných VLAN[3] sítích. Jsou podporovány IPv4 a IPv6 adresy.	Pracovní uzly běží v soukromé podsíti VPC. Pro přístup k veřejné síti je nutné použit VPN nebo plovoucí IP. Jsou podporovány jenom IPv4 adresy.
Typy kontejnerů	Podporuje Kubernetes a OpenShift kontejnery.	
Integrace s jinými službami	Nabízí možnost integrace s celou řadou služeb IBM a třetích stran pro monitoring, zabezpečení, ukládání dat, automatizaci nasazení a logování.	Podporuje jen vybrané služby (menší počet služeb v porovnání s klasickou infrastrukturou).

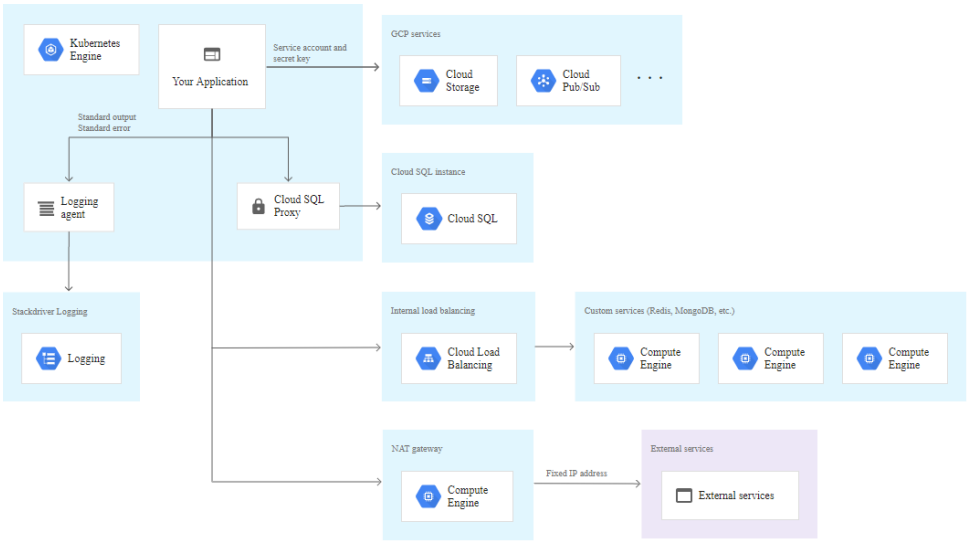
[1] Bare metal – stroj, k fyzickým prostředkům kterého má zákazník přímý přístup (například k paměti nebo k procesoru).
[2] SDS – Software defined store – fyzický stroj s pevnými disky navíc pro trvalé ukládání dat.
[3] VLAN – Virtual Local Area Network – virtuální lokální síť.

Tabulka 2.1: Porovnání typů infrastruktury v IBM IKS [27]

jinými službami. Takový účet je dostupný po dobu 30 dní a dovolí uživateli se seznámit s IBM IKS a vyzkoušet jeho možnosti. Dále IBM nabízí různé cenové plány v závislosti na požadavcích uživatele. Obecně se používá systém Pay-as-you-go[22], kde uživatel platí za použité prostředky (spotřebovaná paměť, zátěž aplikace, vybraná infrastruktura, konfigurace stroje nebo propojené služby) [30].

IBM Cloud Kubernetes Service je mezi zákazníky populární díky své jednoduchosti použití a zároveň rozmanitosti nabízených služeb a kvalitní dokumentaci. Mnozí klienti oceňují i přívětivou cenu za použití služeb [31].

[22] česky – průběžné účtování



Obrázek 2.9: Integrace GKE s ostatními systémy [33]

Google Kubernetes Engine (GKE)

Google Kubernetes Service je systém pro správu clusterů s otevřeným zdrojovým kódem, který je součástí platformy Google Cloud. Funkce GKE zahrnují automatické horizontální škálování, vyrovnání zátěže, samoléčení kontejnerů, monitorování stavu clusterů a další. Mezi zákazníky patří Forbes, GitLab, Whirlpool, Sony Music a jiné [32].

Google Cloud je velká platforma, která nabízí mnoho produktů. GKE jako součást platformy může být s těmito službami integrovaná pomocí různých nastavení (obrázek 2.9). Například pro používání většiny služeb stačí nastavit aplikaci příslušná oprávnění a předat tajný klíč, pomocí kterého bude probíhat komunikace. Pro použití vlastních aplikací je třeba nastavit load balancer, který přesměruje dotaz na příslušnou repliku služby. GKE umožní také integraci s externími systémy, buď s výchozími nastaveními, nebo s použitím pevné externí adresy a brány NAT[23] [34].

GKE nabízí několik typů clusterů. Z pohledu geografického umístění se rozlišují jednozónové, vícezónové a regionální clustery[24]. Různé typy se liší podle geografického umístění a počtu replik řídicí jednotky a pracovních node [35]. Rozdíly jsou znázorněny v tabulce 2.2. Při zakládání clusteru uživatel může taky zvolit, jakou verzi K8s bude používat. Rozlišuje se výchozí verze (vybraná na základě interního testování týmem GKE), specifická verze

[23] Podle varování Google Cloud zdrojová IP adresa záleží na použitém uzlu virtuálního počítače. Může se stát, že pakety mohou mít různé IP adresy přestože byly odeslány ze stejné aplikace. Nutnost konfigurace NAT brány záleží na možnosti externích systémů takové pakety zpracovat [34].

[24] Ve slovníku GKE zóna je podmnožinou regionu. Například, region `us-west1` obsahuje zóny `us-west1-a`, `us-west1-b` a `us-west1-c`.

	Zonální cluster		Regionální cluster
	Jednozónový	Vícezónový	
Řídicí jednotka	Jediná replika běží v jedné zóně	Jediná replika běží v jedné zóně	Má více replik, které běží ve vícero zónách jednoho regionu
Pracovní jednotka	Běží ve stejné zóně, jako příslušná řídicí jednotka	Pracovní uzly běží ve více zónách	Pracovní uzel běží v každé zóně, kde běží řídicí jednotka

Tabulka 2.2: Porovnání typů clusterů z geografického hlediska v GKE

(vybraná uživatelem) a tzv. release channel (aktualizuje verzi Kubernetes skoro hned po její vydání) [36].

Účtování služeb GKE se dělí na 2 části. První položka faktury bude cena správy clusteru, která je pevně dána a nemění se v závislosti na velikosti a topologii clusteru [25]. Druhá část účtování se vztahuje na využívání pracovních jednotek (cena záleží na konfiguraci zvoleného stroje, využití paměti, zátěži CPU apod.)[37].

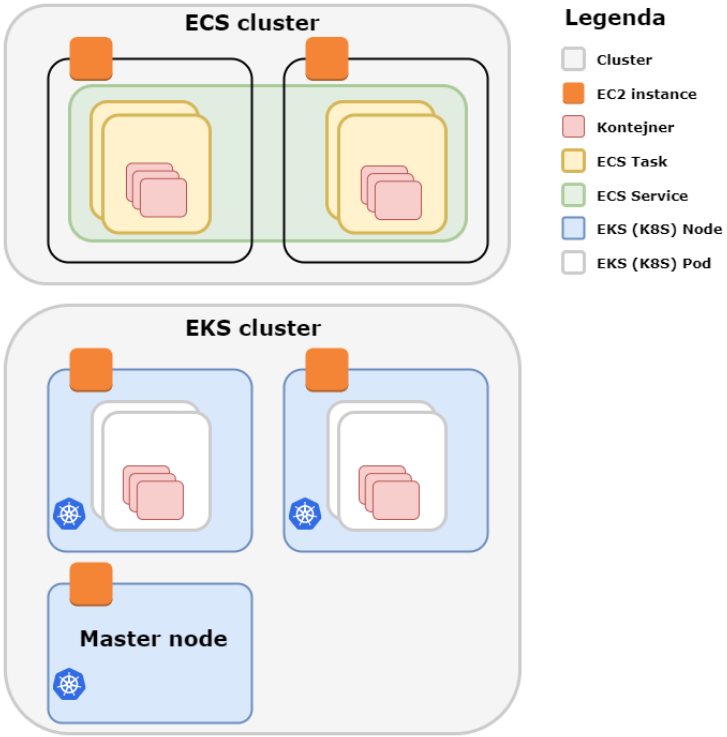
Podle recenzí na webu trustradius.com zákazníci GKE nejvíc oceňují flexibilitu služby, tj. možnost spravovat životní cyklus clusteru a regionální nastavení. Za výhodu považují i velkou nabídku služeb pro integraci z platformy Google Cloud. Uživatelé ale vytýkají GKE špatnou dokumentaci a podporu [38].

Amazon Elastic Container/Kubernetes Service

AWS nabízí 2 různé služby: EKS (Amazon Elastic Kubernetes Service) a ECS (Amazon Elastic Container Service). Jejich hlavní rozdíl je v základní platformě pro orchestraci kontejnerů. ECS je postaven na proprietárním produktu společnosti Amazon, má vlastní API a architekturu. Základem EKS je open-source Kubernetes, což díky společnému API zjednodušuje migraci z jiných K8s-based platforem. Mezi zákazníky ECS patří například společnosti Ubisoft, Duolingo nebo Autodesk [39]. Klienty EKS jsou Snap Inc., Pearson Vue a samotný Amazon.com [40].

Na obrázku 2.10 je vidět porovnání architektur clusterů ECS a EKS. Z daného schématu je na první pohled vidět důvod rozdílu cenové politiky těchto služeb. Obě platformy účtují poplatek za použití výpočetních zdrojů a paměti na EC2 instancích, ale na rozdíl od ECS Kubernetes potřebuje jednu

[25] GKE nabízí benefit v podobě osvobození jednoho zonálního clusteru od poplatků za správu. Jelikož výjimka se vztahuje na celý cluster, kde může být více zonálních node, limit pro bezplatné používání je počítán v hodinách a je roven maximálnímu počtu hodin pro určitý měsíc (`počet dní v měsici * 24`)



Obrázek 2.10: Porovnání architektur clusterů Amazon ECS a EKS

instanci navíc. Na tomto uzlu se nachází K8s master node, který řídí pracovní jednotky s kontejnery [41]. Z těchto důvodů je k poplatkům pro EKS přičtená paušální částka za správu clusteru pomocí master node. Dalším rozdílem je existence tzv. ECS Service. Tahle část clusteru spouští a řídí několik instancí stejného tasku. Tenhle princip připomíná jednoduchou replikaci: uživatel definuje, kolik stejných instancí chce mít, orchestrátor tyto jednotky spravuje a v případě selhání jedné z nich novou repliku hned nasadí. Tato služba navíc může být spuštěna s použitím load balanceru, který rozdělí zátěž mezi jednotlivými replikami [42]. Shrnutí porovnání ECS a EKS je vidět v tabulce 2.3

Amazon nabízí 2 platformy, na kterých mohou běžet kontejnery ECS a EKS: Fargate a EC2 (Elastic Computing Cloud). Liší se hlavně úrovní abstrakce při nasazení aplikace. EC2 je IaaS platforma, která dovolí uživateli pro každou instanci definovat výpočetní zdroje, velikost paměti a trvalých úložišť, nastavení sítě apod. Nabízí na výběr velké množství různých typů instancí (virtuálních serverů), které se liší velikostí paměti a úložiště, výrobcem procesoru, počtem jader a propustností sítě. Různé instance jsou rozdělené do skupin pro přehlednost, například výpočetně nebo paměťově optimalizované stroje. Z toho vyplývá jedna z nevýhod EC2 – pokud zvolená instance má nedostatek nebo nadbytek zdrojů, uživatel je vynucen ručně modifikovat typ instance. Při takové transformaci je nutné počítat například s změnou veřejné IPv4 adresy aplikace. Naopak, Fargate je CaaS platformou, která

	Amazon ECS	Amazon EKS
Open-source	Ne – proprietární produkt AWS	Ano, základem je open-source Kubernetes
Minimální jednotka pro nasazení kontejneru	Task	Pod
Složitost vývoje a nasazení	Nízká – znalost AWS API	Střední – je nutná znalost API AWS a Kubernetes
Zabezpečení	Zabudovaná kontrola identit, zabezpečená síť v rámci jednoho tasku	Zabezpečená síť v rámci podu. Pro kontrolu identit je nutná instalace doplňků
Limit počtů kontejnerů pro jeden virtuální stroj	Maximálně 120 tasků	Maximálně 750 podů
Integrace s jinými cloudovými platformami	Ne – produkt AWS	Ano
Cena	Platí se za použité prostředky	Platí se za použité prostředky + paušální cena za cluster

Tabulka 2.3: Porovnání Amazon ECS a EKS [43]

abstrahuje uživatele od správy serverů a clusterů. Vývojářovi pak dovolí definovat požadavky na HW, ale celou správu infrastruktury a virtuálních strojů včetně případného škálování převezme na sebe. Je to dobrá volba pro nenáročné projekty, navíc s proměnlivou zátěží, protože Fargate se přizpůsobí aktuálním požadavkům a uživatel pak zaplatí jen za reálně spotřebované prostředky [44].

Podle recenzí uživatelé nejvíc oceňují jednoduchost použití AWS ECS a to, že je součástí velké platformy Amazon. Na druhou stranu konfigurace služeb je poněkud dostatečně náročná a potřebuje více času [45].

Red Hat OpenShift

Pod názvem OpenShift, na rozdíl od předchozích platforem, se skrývá skupina produktů. Seznam nabízených produktů se začíná OKD, což je distribuce Kubernetes od společnosti Red Hat. Je nadstavbou K8s a nabízí navíc:

1. Větší zabezpečení (izolace kontejnerů a souborů, omezená oprávnění pro kontejnery).
2. Automatickou aktualizaci clusteru, konfiguraci pracovních jednotek a nasazení aplikace po přijetí změn z repositáře se zdrojovým kódem.
3. Vlastní registr kontejnerů.

4. Vestavěný monitoring, logování a sběr metrik [46].

Dalším produktem je OpenShift Container Platform (OCP), který je PaaS platformou pro on-premise provoz na vlastním HW nebo u certifikovaných cloudových providerů (na moment napsání této práce mezi takovými providery patří Google Cloud, Microsoft Azure a Amazon Web Services (AWS)[47]). Následujícím řešením je OpenShift Dedicated, který je dost podobný OCP s tím rozdílem, že cluster je nasazen do cloudu a plně spravován společností Red Hat. Dostupnými platformami pro provoz jsou Amazon Web Service a Google Cloud [48]. Oba řešení, OCP a OpenShift Dedicated, nabízejí několik typů přípravy aplikace k nasazení [49]:

1. Docker build – příprava aplikace pomocí kroků, definovaných v Dockerfile.
2. Source-to-Image (S2I) – definované workflow, které ze zdrojového kódu vytvoří image. Uživatel musí jen definovat vlastní skripty pro build a spuštění aplikace.
3. Vlastní build – uživateli je dostupná možnost definovat celý proces buildu aplikace.
4. Pipeline build – příprava aplikace pomocí integrovaného Jenkins pluginu.

Posledním produktem, na který se zaměřím v této podkapitole, je OpenShift Online. Je to hostingová platforma řády OpenShift, která dovolí uživateli provádět buildy, nasazovat, spravovat a škálovat kontejnerizovanou aplikaci v cloudu. Platforma je zajímavá tím, že při vytvoření projektu nabízí předpřipravené šablony pro různé jazyky programování a frameworky (například Java, NodeJS). Šablona obsahuje konfigurace a pipeliny, specifické pro daný jazyk nebo framework, které pak uživatel může libovolně modifikovat. Navíc OpenShift Online nabízí hotové kontejnery pro různé databáze. Aplikaci, pipeliny, nasazení a logování lze spravovat přes webové rozhraní nebo pomocí příkazové řádky [50].

Porovnání výše zmíněných produktů OpenShift je v tabulce 2.4.

Zákazníky OpenShift jsou například společnosti Cisco, UPS, Deutsche Bank a Slovenská sporiteľňa [52].

	OKD	OpenShift Container Platform	OpenShift Dedicated	OpenShift Online
Typ produktu	Distribuce Kubernetes	PaaS platforma pro správu a provoz aplikace	PaaS platofrma pro správu a provoz aplikace	PaaS platforma s možností hostingu
Cíl nasazení	Lokální, vlastní datacentrum, cloud	Lokální, vlastní datacentrum, cloud (AWS, Google Cloud, Microsoft Azure)	Cloud (AWS, Google Cloud)	Cloud (hosting Red Hat)
Typ provozu	v závislosti na cíli nasazení	samostatně spravovaný	spravovaný společností Red Hat	spravovaný společností Red Hat
Cena	Open-source projekt	Placený, částka je fakturována společností Red Hat za použití OpenShift produktu a případně cloudovým providerem za použití výpočetních prostředků (při nasazení do cloudu)	Placený, částka je fakturována společností Red Hat za použití OpenShift produktu a AWS nebo Google Cloud za použití a správu infrastruktury	Placený, částka je fakturována společností Red Hat za použití OpenShift produktu a hosting

Tabulka 2.4: Porovnání vybraných produktů skupiny OpenShift [51]

Kapitola 3

Analýza práce s vybranými orchestrátory

Tato kapitola je věnována práci s Kubernetes a OpenShift. V následujících částech popíšu architekturu K8s, prozkoumám základní principy ovládání clusteru a ukážu minimální konfigurační soubory, které jsou potřeba pro nasazení ukázkové aplikace. V části věnované OpenShift se zaměřím na hlavní rozdíly oproti Kubernetes a popis nových objektů API. Na závěr kapitoly bude ukázána práce s clusterem pomocí klienta příkazové řádky.

3.1 Kubernetes

Většina z platforem, které byly uvedené v předchozí kapitole, jako základ používala Kubernetes. Kubernetes, nebo K8s, je open-source platforma pro automatické nasazení, škálování a řízení kontejnerizovaných aplikací [53]. V této části práce se zaměřím na popis tohoto systému, jeho architektury a definici pojmů, a to tak, aby tato kapitola sloužila návodem pro používáni K8s i pro čtenáře bez předchozích zkušeností s touto platformou.

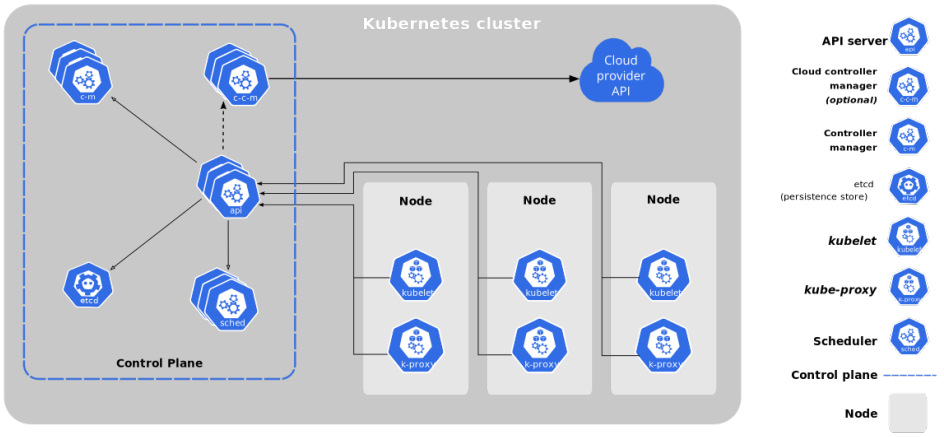
3.1.1 Architektura Kubernetes a definice pojmů

Tahle kapitola je zcela založená na dokumentaci Kubernetes [1].

Tuto část práce začnu definováním pojmů, které se budou často používat dále v kapitole k popisu architektury K8s. Celý slovník je dostupný v dokumentaci Kubernetes[2]. Níže jsou uvedené základní pojmy, které jsou nezbytné pro další práci s Kubernetes:

[1] Dostupné z: `https://kubernetes.io/docs/home/` (24.12.2020)

[2] Dostupné z: `https://kubernetes.io/docs/reference/glossary/?all=true` (26.12.2020)



Obrázek 3.1: Architektura clusteru K8s [54]

1. **Kontejner** – spustitelná jednotka, která obsahuje software a jeho závislosti.
2. **Pod** – množina běžících kontejnerů v clusterů.
3. **Node (pracovní jednotka)** – základní komponenta v architektuře Kubernetes, na které běží pody, například virtuální stroj.
4. **Cluster** – neprázdná množina nodů, ve kterých běží kontejnerizované aplikace.
5. **Control plane (řídicí jednotka)** – vrstva orchestrace, která poskytuje API pro definování, nasazení a správu životního cyklu kontejnerů.
6. **Service** – způsob jak vystavit aplikaci běžící na několika podech jako jednu službu.

Po definici pojmů se můžeme pustit do samotné architektury Kubernetes (obrázek 3.1). Cluster se skládá z množiny nodů, ve kterých jsou umístěny pody. V každém node může být umístěno více podů, v každém podu – více kontejnerů. Dále cluster obsahuje řídicí jednotku, která spravuje nody a pody v clusteru. V produkčním prostředí control plane často běží na několika strojích, protože se jedná o kritickou část clusteru. V následujících podkapitolách podrobněji popíšu každou komponentu řídicí jednotky a nodů.

Řídicí jednotka clusteru K8s a její komponenty

Jak už bylo zmíněno výše, řídicí jednotka odpovídá za správu clusteru a je jeho důležitým prvkem. Dělá veškerá rozhodnutí na úrovni clusteru (například plánování zdrojů), detekuje a reaguje na události (například spuštění nového podu). Své API řídicí jednotka poskytuje přes komponentu `kube-apiserver`.

Vystavuje REST API, které dovolí uživatelům, částem clusteru a externím systémům komunikovat mezi sebou. Rozhraní je dokumentováno pomocí OpenAPI. Pro komunikaci se používá JSON nebo Protobuf (většinou pro interní zprávy). Další možnosti použití apiserveru je `kubectl`, který dovolí provolat K8s rozhraní přes příkazovou řádku. Apiserver je horizontálně škálovatelný, tj. je možné spustit několik instancí za load balancerem pro zvětšení dostupnosti. Konfigurace a data o stavu clusteru se ukládají do `etcd`, což je key-value úložiště pro distribuované prostředí. Na rozdíl od apiserveru `etcd` není interním projektem Kubernetes. Vyvíje to společnost Cloud Native Computing Foundation. Úložiště `etcd` je taky použito například v projektech Openstack nebo CoreDNS [55].

Další důležitou částí řídicí jednotky je plánovač. Ten přiřazuje nové pody k nodům, odpovídajícím požadavkům na nasazení. Výchozím plánovačem pro K8s je `kube-sheduler`. Uživatel může upravovat výchozí chování podle svých potřeb. Plánování se skládá ze tří fází: filtrování, ohodnocení a přiřazení. Ve fázi filtrování `kube-sheduler` z množiny pracovních jednotek vybírá proveditelné, tj. ty, které odpovídají požadavkům podu na zdroje, například mají dost jader CPU, paměti RAM apod. Uživatel může definovat další filtry nebo použit existující predikáty. Pokud je množina proveditelných nodů prázdná, pod se nikam nenasadí a bude čekat na uvolnění zdrojů. V jiném případě následuje fáze ohodnocení. Během této fáze každý node dostává vlastní skóre na základě definovaných parametrů. Pro nasazení se vybírá pracovní jednotka s nejvyšším ohodnocením. Pokud existuje více node se stejným skóre, vybírá se náhodný. Při ohodnocení se berou v ohled požadavky na hardware a software, umístění dat, aktuální zátěž apod. Uživatel může upravovat logiku této fáze a přidávat vlastní podmínky. Poslední fáze je přiřazení, kdy se pod nasazuje na vybraný node. Tohle chování je interně implementováno v K8s a uživatel ho nemá možnost upravovat.

Nedílnou součástí řídicí jednotky je `kube-control-manager`, nebo správce controllerů. Controller je v Kubernetes proces, který pomocí apiserveru kontroluje stav celého clusteru a dělá změny, aby cluster posunul do požadovaného stavu. Příklady takových controllerů jsou:

1. **Node controller**, který reaguje na události, vzniklé selháním nějaké pracovní jednotky.
2. **Replication controller**, který je zodpovědný za správný počet replik pro pody.
3. **Endpoints controller**, který slouží pro vystavení API service a podů.
4. **Controller účtů a tokenů**, který vytváří výchozí účty a API tokeny pro nové služby.

Existují i další typy controllerů, které odpovídají za konkrétní stav clusteru.

Uživatel má možnost implementovat vlastní controller pro rozšíření výchozích funkčností.

Podobnou jednotkou je i `cloud-controller-manager`. Ten je taky správcem controllerů, ale na rozdíl od `kube-control-manager` spravuje controllery, které jsou specifické pro danou cloud platformu. Používá se pro integraci K8s do cloudových platforem (například AWS nebo Google Cloud). Při on-premise nasazení nebo lokální instalaci se `cloud-controller-manager` nepoužívá.

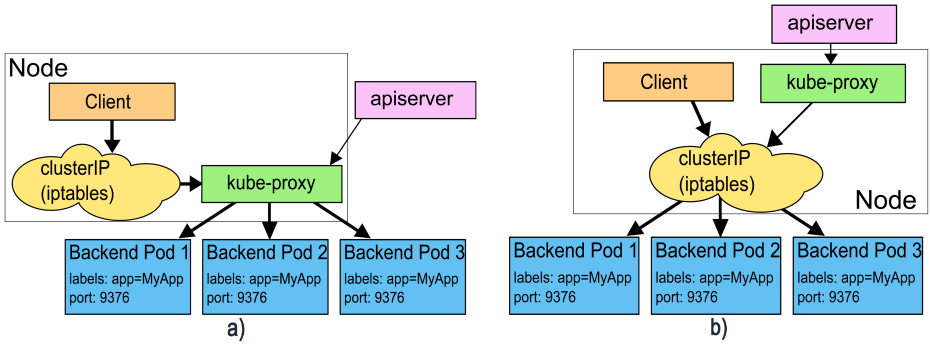
Pracovní jednotka clusteru K8s a její komponenty

Pracovní jednotka je nedílnou součástí každého clusteru. Nody obsahují pody, ve kterých běží kontejnerizovaná aplikace. Kromě podů pracovní jednotka má i řídicí prvky. Jedním z nich je `kubelet`, což je nástroj, který je zodpovědný za běh kontejnerů. Tento prvek čte konfigurační soubory podů (definované buď ve formátu YAML nebo JSON) a na základě toho spouští kontejnery a kontroluje jejich „zdraví“. Když je konfigurační soubor odstraněn ze sledované složky, `kubelet` zastaví kontejner. Těsně s `kubelet` souvisí tzv. Container runtime, což je software pro běh kontejnerů. Obsahuje programy, závislosti a knihovny pro běh konkrétní technologie kontejnerů v podu. Výchozí technologií je Docker, ale K8s podporuje i containerd[3], CRI-O[4] technologie, a navíc libovolnou uživatelskou implementaci Container Runtime Interface (CRI).

Důležitou součástí pracovní jednotky je `kube-proxy`. Je to nástroj, který spravuje síť na celém nodu. Vystavuje API kontejnerů do externí sítě, spravuje komunikaci mezi jednotlivými Service a zároveň slouží jako vestavěný load balancer. Podporuje tři různých režimy: userspace proxy, iptables proxy a IPVS proxy.

V režimu userspace `kube-proxy` reaguje na události apiserveru a při vytvoření nové service aktualizuje směrovací tabulku. Otevře připojení na náhodně vybraném portu a veškerou komunikaci na service přesměruje na tento port. Při volbě backend služby se řídí algoritmem round robin. Výhodou takového přístupu je to, že proxy zajistí, aby nevznikla kolize portů. Na druhou stranu přesměrování komunikačních paketů přes proxy je časově náročné. Navíc při selhání `kube-proxy` veškerá komunikace se zastaví. Režim iptables je dost podobný předchozímu přístupu s tím rozdílem, že komunikace není závislá na dostupnosti `kube-proxy`. Jak i v předchozím případě při vytvoření nové service proxy aktualizuje směrovací tabulky, ale již jiným způsobem. Nastaví je tak, aby klient mohl komunikovat se službou napřímo přes IP tabulky, bez použití proxy. Load balancer používá náhodný výběr backend služby. Rozdíl mezi userspace a iptables režimem je vidět na obrázku 3.2. Výhodou takového

[3]Dostupné z: `https://containerd.io/docs/` (26.12.2020)
[4]Dostupné z: `https://cri-o.io` (26.12.2020)



Obrázek 3.2: Režimy kube-proxy: a) userspace, b) iptables [56]

přístupu je to, že je rychlejší než proxy, a navíc při selhání `kube-proxy` komunikace může pro běžící service pokračovat. Třetí přístup je relativně nový (byl implementován ve verzi 1.11 [56]) a je velmi podobný iptables režimu. Má ale 2 zásadní rozdíly:

1. Místo klasických IP tabulek používá interní hash tabulky, které jsou výrazně efektivnější ve velkých clusterech (velkým clusterem se myslí cluster s například 10 000 service [57]).
2. Poskytuje více algoritmů pro load balancing, například least connections, IP hash, nejmenší očekávané zpoždění apod.

3.1.2 Ovládání Kubernetes clusteru přes příkazovou řádku

Jak už bylo zmíněno dříve v této kapitole, Kubernetes poskytuje své API přes REST rozhraní. Tento přístup ale nemusí být vhodný pro některé uživatele, kteří jsou zvyklí na používání příkazové řádky. Z tohoto důvodu je součástí platformy Kubernetes i klient příkazové řádky `kubectl`. Ten nabízí množinu příkazů, které interně transformuje do HTTP požadavků a následně převádí odpověď do uživatelsky přívětivé podoby [58]. Pro manipulaci s clusterem budu používat právě tohoto klienta, proto bude tato kapitola věnována jeho základním funkčnostem.

Obecný tvar příkazu vypadá následně:

```
$ kubectl [command] [TYPE] [NAME] [flags]
```

Význam jednotlivých proměnných je následující:

1. `command` je operace, kterou uživatel chce provést nad jedním nebo více zdroji.

2. `TYPE` je citlivý na velikost písmen typ zdroje, nad kterým bude operace provedena (například pod nebo node). Může se používat ve třech různých podobách: celý název v množném čísle, celý název v jednotném čísle nebo zkrácený název.

3. `NAME` je název konkrétního zdroje, nad kterým bude operace provedena. Pokud bude název vynechán, operace se provede nad všemi zdroji tohoto typu. Parametr je citlivý na velikost písmen.

4. `flags` jsou volitelné flagy pro rozšíření funkcionality.

Pro čtení objektů, tj. zjištění jejich aktuálního stavu, se používá operace `get`. Například následujícím příkazem získám informaci o podu s názvem *my-pod*:

```
$ kubectl get pods my-pod
```

Jak už bylo zmíněno výše, název zdroje může být vynechán, proto následující příkaz vrátí seznam všech podů:

```
$ kubectl get pods
```

Kubernetes taky nabízí možnost získat údaje o více zdrojích najednou. Uživatel může jednoduše v příkazu napsat seznam názvů těchto zdrojů, pokud jsou stejného typu, nebo seznam `TYPE/NAME`, pokud mají typ odlišný. Nakonec je možné získat data pomocí konfiguračního souboru:

```
$ kubectl get -f myPod-config.yml
```

Příkaz `get` vrací jen základní informace o zdroji. Pro získání podrobnějších dat lze použit `describe`. Ten vrací nejen detailnější popis zdroje, ale i údaje o souvisejících objektech. Navíc se příkaz liší svou syntaxi. Stejně jako `get`, může popsat zdroj pomocí souboru nebo najít údaje o všech objektech stejného typu při chybějícím názvu. Nově ale přibývá možnost zdroj vyhledat pomocí prefixu jeho názvu. Například koukneme na následující příkaz:

```
$ kubectl describe pods my-pod
```

V tomto případě pokud existuje pod s názvem *my-pod*, tak příkaz `describe` vrátí informaci o něm. Jinak vrátí detail všech podů, jejichž název se začíná na *my-pod*. Pokud bude uživatel chtít použit přesnou shodu názvu, musí vyhledávaný výraz zadat ve tvaru `TYPE/NAME` [59].

Kromě získávání údajů o zdrojích chceme taky tyto zdroje vytvářet a aktualizovat. Na to se používá příkaz `apply`, který jako parametr přijímá konfigurační soubor ve formátu JSON nebo YAML:

```
$ kubectl apply -f myPod-config.yml
```

Dalším způsobem vytvořit zdroje je příkaz `create`. Na rozdíl od `apply`, který v případě existence zdroje jeho aktualizuje, příkaz `create` smaže existující objekt a vytvoří nový. Některé typy zdrojů jako jsou pody a joby se dá vytvořit

i jiným způsobem, a to pomocí příkazu `run` s použitím flagu `--generator`. Tento způsob je pro jiné zdroje než pody zastaralý od verze 1.17 [60].

Nakonec pro smazání zdroje se používá příkaz `delete`. Dovolí odstranit objekt buď pomocí konfiguračního souboru, nebo typu a názvu. Znamená to, že následující příkazy jsou ekvivalentní:

```
$ kubectl delete -f myPod-config.yml
$ kubectl delete pods my-pod
```

Další možné příkazy lze najít v dokumentaci Kubernetes[5] nebo pomocí příkazu:

```
$ kubectl help
```

3.1.3 Nasazení clusteru Kubernetes

Častým způsobem vytvoření objektů Kubernetes je definice konfiguračních souborů ve formátu YAML nebo JSON. Každý objekt má přesně definované povinné, volitelné a zakázané parametry. V rámci této podkapitoly se podívám na nejjednodušší definici některých objektů, které potom budou použité při nasazení testovací aplikace v kapitole 5.1.

Vytvoření podů

Jak už víme z předchozích podkapitol, pod je jednotkou clusteru, ve které běží jeden nebo více kontejnerů. Je nutné zmínit, že pod je nejmenší nasaditelnou jednotkou clusteru. To znamená, že veškeré kontejnery v podu sdílí IP adresu a nutně se nachází na jednom node. Navíc není možné různě škálovat kontejnery v rámci jednoho podu, škálovací poměr bude pro ně vždy stejný. Z těchto důvodu podle doporučení Kubernetes většina podů by měla obsahovat jen jeden kontejner. Výjimka se dělá v případě, že kontejnery jsou úzce provázány a potřebují sdílet zdroje a paměť [61].

Minimální konfigurační soubor pro vytvoření podu je zobrazen ve fragmentu kódu 3.1. Pro ostatní položky se použijí výchozí hodnoty. Tento konfigurační soubor vytvoří jednoduchý pod s jedním kontejnerem, ve kterém je spuštěn image *nginx*.

```
apiVersion: v1
kind: Pod
metadata:
  name: my-pod
spec:
```

[5] Dostupné z: `https://kubernetes.io/docs/reference/kubectl/overview/` (27.12.2020)

```
containers:
  - name: container-1
    image: nginx
```

Fragment kódu 3.1: Minimální konfigurační soubor pro vytvoření podu

Uživatele ale mohou potřebovat i podrobnější definici podu. Například jedním z případů použití může být požadavek na minimální zdroje, dostupné pro aplikaci. Může se hodit nejen pro jistotu, že aplikace dostane dostatečné zdroje pro své fungování, ale i pomůže plánovači určit vhodný node pro nasazení. Požadavek na prostředky se definuje pomocí položky `resources` pro jednotlivé kontejnery. Fragment kódu 3.2 ukazuje, jak definovat minimální a maximální zdroje dostupné pro kontejner. Jak je vidět na fragmentu, množství CPU se zadává jako *500m* nebo *1000m*, což znamená 0.5, respektive 1 CPU[6].

```
...
containers:
  - name: container-1
    image: nginx
    resources:
      requests:
        cpu: "500m"
        memory: "128M"
      limits:
        cpu: "1000m"
        memory: "256M"
```

Fragment kódu 3.2: Definice zdrojů pro pod

Dalším případem použití detailnější konfigurace je definice závislostí mezi kontejnery. Například pokud se ve stejném podu nachází databáze a samotná aplikace (což jak bylo zmíněno výše není doporučeno), uživatel bude chtít spustit aplikaci jen poté, co bude databáze dostupná. Dalším příkladem může být čekání na dostupnost jiného podu nebo service. Na fragmentu 3.3 je vidět, že takové závislosti se definují pomocí položky `initContainers`. Je tam seznam kontejnerů, které se spustí sekvenčně před spuštěním hlavních kontejnerů (tj. před nastartováním podu). Pokud jeden ze základních kontejnerů selže, pod se nespustí.

```
apiVersion: v1
kind: Pod
metadata:
  name: my-pod
spec:
  containers:
    - name: container-1
      image: nginx
  initContainers:
```

[6] V Kubernetes 500m CPU znamená 500 miliprocessorů. Interní API transformuje číslo s desetinní čárkou do této podoby, proto je zápis s jednotkou *m* preferovanější [62]

```
  - name: db-container
    image: mysql
```

Fragment kódu 3.3: Definice závislostí mezi kontejnery

Pro vytvoření pokročilejších podů lze definovat další položky, například porty pro vystavení rozhraní, zásady restartu, štítky apod. Možné položky a jejich výchozí hodnoty může čtenář dohledat v dokumentaci rozhraní odpovídající verze[7].

Podle doporučení Kubernetes lepším způsobem vytvoření podů je definice nasazení[8], což je komplexní typ, kterým se v Kubernetes definuje celá verzovatelná část aplikace [61]. Detailnější popis vytvoření definice nasazení se nachází v kapitole 3.1.3

Ukládání dat

Důležitou vlastností kontejnerů a podů je to, že se při jejich vypnutí nebo restartu smaže veškerý souborový systém těchto kontejnerů. Na jednu stranu tato vlastnost zaručuje čistou instalaci aplikace. V případě, že souborový systém bude poškozen, pro jeho obnovu bude potřeba jen restartovat pod. Na druhou stranu některé aplikace potřebují perzistentní úložiště, kde by mohly uchovávat dlouhodobá data. Pro tento účel existuje v Kubernetes pojem *volumes*, což jsou zvolená místa v souborovém systému hostitele, kam má kontejner přístup [63]. Ve specifikaci podu volume musí být definován na dvou místech:

1. v položce `spec.volumes`. Tady se definují veškeré volume, dostupné pro daný pod.
2. v položce `containers.volumeMounts`. Pro každý kontejner tady uživatel definuje dostupný volume z výše uvedené specifikace. Kontejner nemusí mít přiřazená všechna definovaná úložiště, a jedno úložiště může sdílet více kontejnerů.

Ve fragmentu kódu 3.4 se vytváří volume *my-data* a přiřadí se toto úložiště kontejneru. Položka `mountPath` specifikuje mapování perzistentního úložiště na cestu v souborovém systému kontejneru.

```
apiVersion: v1
kind: Pod
metadata:
  name: my-pod
```

[7] Dostupné z: `https://v1-17.docs.kubernetes.io/docs/reference/generated/kubernetes-api/v1.17/` (28.12.2020)

[8] angl. deployment

```
spec:
  volumes:
    - name: "my-data"
      hostPath:
        path: "/path-in-node"
  containers:
      - name: container-1
        image: nginx
        volumeMounts:
          - mountPath: "/path-in-container"
            name: "my-data"
```

Fragment kódu 3.4: Definice perzistentního úložiště pro kontejner

Kromě ukládání dat přímo na node Kubernetes nabízí i jiná úložiště, například vzdálené disky jako AWS Elastic Block Store, Microsoft Azure Data Disk nebo NFS systém. Kompletní seznam všech dostupných typů úložišť je uveden v dokumentaci Kubernetes[9].

Definice nasazení

V předchozí kapitole jsem zmiňovala, že podle doporučení Kubernetes lepší způsob vytváření podů je pomocí definice deploymentu. Pod je jednoduchou jednotkou, kterou nelze samu o sobě škálovat a verzovat.Bez definice nasazení při změně image v podu se starý kontejner zastaví, smaže a nahradí novým. Deployment nabízí vestavěnou podporu verzování, škálování a rollout aktualizací. Příklad definice nasazení je uveden ve fragmentu 3.5.

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: my-deployment
spec:
  replicas: 2
  selector:
    matchLabels:
      app: my-app
  template:
    metadata:
      name: my-pod
      labels:
        app: my-app
    spec:
      containers:
```

[9]Dostupné z: `https://kubernetes.io/docs/concepts/storage/volumes` (29.12.2020)

```
  - name: container-1
    image: nginx
```

Fragment kódu 3.5: Definice deploymentu

Jak je vidět ve fragmentu kódu 3.5, deployment může spravovat jen jeden pod. Nyní detailněji popíšu součásti konfigurace definice nasazení ve výše uvedením fragmentu :

1. položka `spec.replicas` indikuje, že bude vytvořeno 2 repliky podu *my-pod*.
2. položka `spec.selector` definuje podmínku, podle které budou vybrány pody, které bude daný deployment spravovat. Pod, definovaný dále v položce `template`, musí tomuto selektoru odpovídat.
3. v položce `spec.template` se definuje pod, který bude vytvořen pomocí deploymentu. Struktura tohotu objektu je stejná jako při vytvoření pouhého podu s tím rozdílem, že se nedefinují položky `apiVersion` a `kind`.

Z konfiguračního souboru se deployment vytvoří pomocí příkazu:

```
$ kubectl apply -f deployment-config.yml
```

Tento příkaz zároveň spustí nasazení aplikace. Výchozí způsob aktualizace je rollout update, který dělá aktualizaci každé repliky postupně, což zaručuje nasazení bez prostoje.

Změna škálování deploymentu se dělá pomocí aktualizace konfiguračního souboru nebo příkazem `kubelet scale deployment/NAME --replicas=4`. Je důležité zmínit, že škálování nezpůsobí rollout update. Verzování deploymentu se dělá jen při změně v položce `spec.template`, tj. při aktualizaci podu.

Kromě nasazení nové verze deployment podporuje i evidenci historie, kterou uživatel může zobrazit pomocí příkazu `kubectl rollout history deployment NAME`. Navíc Kubernetes dovolí vrátit se k libovolné předchozí revizi následujícím příkazem:

```
$ kubectl rollout undo deployment my-deployment

$ kubectl rollout undo deployment my-deployment
 --to-revision=2
```

První příkaz znamená návrat k předchozí verzi. Druhý příkaz vrátí deployment k verzi 2.

Z této kapitoly vyplývá, že vytvoření definice nasazení je mnohem pohodlnější než vytvoření samostatného podu.

Definice service

Jak už bylo zmíněno v předchozí kapitole, deployment řídí jen jeden pod (bez ohledu na počet replik). Někdy se ale aplikace skládá z několika nezávislých komponent, které komunikují mezi sebou pomocí sítě. Pro umožnění takové komunikace v Kubernetes existují service, které můžeme brát jako obal pro množinu podů, který vystaví jejich rozhraní do sítě [63]. Definice service obsahuje 2 důležité části:

1. `spec.selector`, pomocí kterého se jednotlivé pody přiřadí ke konkrétní service. Pokud bude selektor vynechán, koncové body pro přesměrování dotazů musejí být vytvořeny ručně.
2. `spec.ports`, kde se definuje, na kterém portu service poslouchá a na jaký port podu přesměruje dotazy.

Jako selektor podů se používá tzv. `label`, což je jednoduchý záznam ve formátu klíč-hodnota, přiřazený k objektu Kubernetes [63]. Například, ve fragmentu kódu 3.6 se vytváří pod, který má 2 labely: label *app* s hodnotou *my-app* a label *env-stage* s hodnotou *prod*.

```
apiVersion: v1
kind: Pod
metadata:
  name: my-pod
  labels:
    app: my-app
    env-stage: prod
...
```

Fragment kódu 3.6: Definice labelů pro pod

Pokud budeme chtít pro tento pod vytvořit service, můžeme nastavit hodnotu `spec.selector` na `app: my-app` nebo `env-stage: prod` (příklad ve fragmentu kódu 3.7). Ve stejném příkladu je zadefinováno, že service poslouchá TCP spojení na portu 80 a přesměruje to na port kontejneru 8080.

```
apiVersion: v1
kind: Service
metadata:
  name: my-service
spec:
  selector:
    app: my-app
  ports:
    - protocol: TCP
      port: 80
      targetPort: 8080
```

Fragment kódu 3.7: Definice service

3.2 OpenShift

V kapitole 2.4.1 jsem zmiňovala, že společnost Red Hat zákazníkům nabízí produkt OKD, což je distribuce Kubernetes. Ze všech nabízených projektů platformy OpenShift se nyní soustředím právě na tento základní produkt, popíšu jeho vlastnosti a rozdíly oproti Kubernetes.

3.2.1 Rozdíly oproti Kubernetes

Prvním rozdílem, kterého si všimne uživatel po instalaci OpenShift clusteru, je existence webové aplikace pro správu clusteru. Na rozdíl od Kubernetes, se kterým uživatel pracuje většinou přes příkazovou řádku, OpenShift poskytuje vestavěné vizuální rozhraní pro tyto účely. Webová aplikace umožňuje vizualizovat práci s clusterem a spravovat jeho objekty. Navíc přes webové rozhraní se dá snadno stáhnout klienta příkazové řádky `oc` a přejít na oficiální dokumentaci.

Co se týká samotného klienta příkazové řádky, v kapitole 3.1.2 jsem ukazovala, jak se Kubernetes ovládá pomocí příkazu `kubectl`. Protože je Kubernetes jádrem OpenShift, cluster OKD taky podporuje komunikaci přes `kubectl`. Pro použití rozšířených možností ale bude muset uživatel použit klienta příkazové řádky `oc`, práce se kterým je ukázána v kapitole 3.2.3. Pro jednoduchost příkazy těchto obou klientů jsou kompatibilní, například příkazy `kubectl get pods` a `oc get pods` vrátí stejný výsledek.

Další novinkou pro uživatele OpenShift je vestavěný repositář image. Použít ho lze v případě, když uživatel nechce nahrávat image aplikací do veřejného vzdáleného repositáře. Tím taky zredukuje závislost fungování clusteru na dostupnosti externích systémů. Kromě vestavěného repositáře OpenShift nabízí také vestavěnou podporu EFK stacku, což je soubor třech aplikací pro logování a sběr metrik aplikace: ElasticSearch, Fluentd a Kibana [64]. Poslední vestavěnou funkčností, kterou zmíním v této práci, je podpora nástrojů pro správu zdrojového kódu, jako jsou třeba GitHub nebo Bitbucket. OpenShift přidává podporu reakcí na události typu nahrání nové verze do repositáře. Navíc pomocí již dříve zmíněné technologie Source-to-Image umožňuje definovat celé workflow od nahrání kódu do GitHubu až po spuštění nové verze kontejnerů [65].

Důležitou vlastností OpenShift je vylepšené zabezpečení. OpenShift na rozdíl od Kubernetes má vestavěné nástroje pro správu uživatelů. Přes své API poskytuje entitu `User`, kterou může administrátor clusteru spravovat, upravovat a mazat. Navíc OKD cluster má vestavěný OAuth server pro autorizaci uživatelů [66]. Dalším vylepšením zabezpečení je to, že cluster OKD lze nainstalovat pouze v systému s podporou SELinux, což je rozšíření Linuxu,

které slouží k větší bezpečnosti systému [67]. Poslední důležitou vlastnosti z pohledu zabezpečení je to, že s výchozími nastaveními OpenShift spouští všechny kontejnery jako procesy bez administrátorských oprávnění[10]. Tento přístup má ale své nevýhody – většina známých kontejnerů se v OpenShift s takovými nastaveními nespustí [68].

Jak vyplývá z výše uvedeného popisu, OpenShift je postaven na Kubernetes, ale přidává do orchestrátoru velké množství nových funkčností a vylepšení. V této kapitole jsem se soustředila jenom na nejdůležitější rozdíly, které mohou uživatele přesvědčit migrovat svůj cluster z prostředí Kubernetes do OpenShift.

3.2.2 Nové objekty

OpenShift je postaven na základě Kubernetes, proto disponuje stejnými druhy objektů. Má například pody, nody, deploymenty apod. Kromě těchto objektů OpenShift poskytuje i jiné, úplně nové, zdroje, které ještě více automatizují nasazení a správu aplikace.

Prvním důležitým objektem, který zmíním v této kapitole, je projekt. Podle dokumentace OKD projekt „je jmenným prostorem Kubernetes s dalšími anotacemi a je centrálním prostředkem, kterým se spravuje přístup ke zdrojům pro běžné uživatele“ [69]. Znamená to, že projekt je rozšířením namespace o další metadata a poskytuje administrátorovi více nástrojů pro řízení přístupu k objektům uvnitř jmenného prostoru.

Dalším novým objektem, který stojí za zmínku, je build. Jedná se o API zdroj, účelem kterého je ze zdrojového kódu vytvořit image. OpenShift poskytuje několik různých strategií buildu, které již byly zmíněny v kapitole 2.4.1. Tyto strategie mají společné vlastnosti, a to například logování celého procesu, publikace výsledků sestavení do repositáře, omezení použitých zdrojů apod. Fungování buildu je jednoduché: řídicí jednotka vytvoří nový pod, uvnitř kterého běží jeden kontejner. Tento kontejner má přístup k instanci Dockeru a v závislosti na strategii pomocí různých příkazů vytváří nový image (například u Docker buildu se používá příkaz `docker build`) [70].

OpenShift nabízí také další objekt, úzce spojený s buildem. Jedná se o tzv. image stream. Tento zdroj představuje abstrakci pro sledování image a jejich verzí. Umožní pak při změně tagů (pomocí kterých se většinou sledují verze image) nastartovat build nebo vynutit deployment použít novou verzi pro kontejnery. Image streamy podporují sledování změny tagů u vestavěného i vzdáleného repositáře [70].

Jak už bylo zmíněno dříve v kapitole 3.2.1, OpenShift poskytuje na rozdíl

[10] v Linuxu tzv. non-root uživatele

od Kubernetes API pro správu uživatelů. Definuje pak tři nové objekty:

1. User (uživatel) – aktér systému, který používá objekty clusteru, ke kterým má přístup.
2. Identity (identita) – objekt, který reprezentuje konkrétní způsob přihlášení pro daného uživatele. Mapování Uživatel-Identita je 1 k N (pro jednoho uživatele může existovat více způsobů přihlášení, například OAuth token, certifikát apod.).
3. UserIdentityMaping – objekt, který mapuje uživatele na identitu. Tento objekt vždy mapuje jednu identitu na jednoho uživatele.

Při přihlášení uživatele odpovídající provider vrací systému identitu uživatele. Pokud v clusteru existuje taková identita a je mapovaná na existujícího uživatele, pak přihlášení proběhlo úspěšně. Pokud zatím neexistuje identita ani uživatel, pak řídicí jednotka vytvoří nového uživatele a objekty identity. V případě, že v systému existuje identita, ale není přiřazená k uživateli, pak přihlášení selhává [71].

Výše zmíněné objekty nejsou jediné nové zdroje, které poskytuje OpenShift vůči Kubernetes. V této práci jsem zmínila jen ty nejzajímavější a nejzákladnější objekty, s použitím kterých se setká skoro každý uživatel OKD. Další objekty a podrobný popis jejich použití jsou dostupné v oficiální dokumentaci OKD [72].

3.2.3 Ovládání OpenShift clusteru

Stejně jako Kubernetes, OKD poskytuje speciální program pro komunikaci s clusterem pomocí příkazové řádky – `oc`. Protože jádrem OpenShift je Kubernetes, komunikace pomocí `kubectl` je taky podporována. Nevýhodou jeho použití je ale to, že umožňuje přístup jen k omezenému počtu objektů (pouze ke K8s API) a neumožňuje uživateli používat nové objekty OKD. Pro jednodušší přechod z `kubectl` jsou příkazy podporované K8s namapované s minimálním rozdílem na příkazy `oc`. Například příkaz `kubectl get deployments` vrátí stejný výsledek jako příkaz `oc get deployments`. Znamená to, že probrané v kapitole 3.1.2 základní příkazy se dají pustit i pomocí tohoto klienta.

Klient `oc` přináší i nové možnosti pro interakci s objekty clusteru. V této kapitole vyjmenuji nejpoužitelnější příkazy, se kterými se setká většina vývojářů. Ostatní příkazy jsou dostupné v oficiální dokumentaci OKD [72].

Práce s clusterem se většinou začíná přihlášením uživatele. V případě `kubectl` nastavení přihlášeného uživatele jsou spravovány pomocí změny kontextu. Nově `oc` poskytuje příkazy pro přihlášení a odhlášení uživatele. Níže jsou uvedeny analogické příkazy pro `kubectl` a `oc`:

```
$ kubectl config set-context --current --user=my-
 user
$ oc login -u my-user
```

Klient OKD taky podporuje rychlejší přepínání jmenných prostorů. Jelikož je projekt rozšířením jmenného prostoru, následující příkazy můžeme také považovat za analogické:

```
$ kubectl config set-context --current --namespace
 =my-namespace
$ oc project my-namespace
```

Vytvoření projektu je pak možné zahájit příkazem `oc new-project <name>`.

Pro nové objekty, například buildy, lze použit jak známé příkazy typu `oc get build <name>` nebo `oc delete build <name>`, tak i speciální rozhraní pro jejich správu. Například příkaz `oc start-build <buildconfig>` spustí nový build pomocí jeho konfigurace. Existuje více zdrojů, ze kterých lze build spustit, například spuštění pomocí `Dockerfile`, kopírováním konfigurace jiného buildu, ze specifického commitu apod. Níže jsou uvedeny ukázkové příkazy ve výše zmíněném pořadí:

```
$ oc start-build --from-file=<Dockerfile name>
$ oc start-build --from-build=<build name>
$ oc start-build --commit=<commit-hash>
```

OKD klient také rozlišuje příkazy běžného uživatele a administrátora. Pro použití příkazů administrátora uživatel musí být autentizován a musí mít dostatečná oprávnění. Syntaxe takových příkazů je následující:

```
$ oc adm [ACTION] [PROPERTIES]
```

Ve výše uvedeném fragmentu kódu `ACTION` je akce, kterou chce administrátor provést, a `PROPERTIES` – vstupní parametry této akce. Například vytvoření projektu administrátorem by vypadalo jako `oc adm new-project <name>`, správa uživatelských skupin – `oc adm groups`.

Kapitola 4

Návrh aplikace pro migraci clusteru

Protože je OpenShift založen na Kubernetes, jeho API je kompatibilní s tímto orchestrátorem. To znamená, že OpenShift podporuje všechny objekty, které poskytuje Kubernetes. Z těchto důvodů základní migrace mezi jejich clustery nevyžaduje změny konfiguračních souborů, dodatečné úpravy apod. Například pro migraci podu stačí do clusteru OpenShift pomocí příkazu `oc apply -f pod-config.yaml` importovat konfiguraci tohoto objektu, a výsledek bude stejný jako kdyby byl objekt nahrán do Kubernetes clusteru. Z toho vyplývá jednoduchý postup manuální migrace clusterů. Nevýhodou takového přístupu je jednak to, že pro migraci uživatel potřebuje mít k dispozici všechny konfigurační soubory, ale taky i to, že ty soubory bude muset postupně nahrávat pro všechny objekty clusteru. Jedná se o jednoduchou, ale dlouhou práci, kterou bude chtít většina uživatelů nějak automatizovat. Z těchto důvodu v rámci této práce vytvořím webovou aplikaci, která uživateli pomůže takovou jednoduchou migraci provést pomocí několika kliknutí.

Důležitou poznámkou je to, že taková aplikace nebude v první verzi poskytovat uživateli více než přenesení Kubernetes objektů do nového clusteru. Pokud bude chtít uživatel použit specifické pro OpenShift objekty (buildy, routes apod.), bude je muset stále vytvářet ručně. Cílem výsledné aplikace není konfigurace nového OpenShift clusteru, ale přenesení Kubernetes jádra do OpenShift clusteru.

4.1 Rešerše existujících nástrojů

Před návrhem vlastní aplikace jsem rozhodla provést rešerši existujících řešení. Velké množství návodů pro migraci clusteru z jednoho prostředí do jiného se buď nezabývá otázkou přenesení objektů, nebo předpokládá manuální nahrání konfigurace. Na trhu ale existuje několik nástrojů, které umožňují provést automatickou migraci Kubernetes clusteru.

Prvním nástrojem je produkt společnosti Portworx[1]. Projekt umožňuje vytvořit párování mezi clustery a přenést objekty z jedné platformy do jiné. Služba migruje Kubernetes objekty, konfigurace a úložiště. Není podporována migrace systémových jmenných prostorů, objektů typu ServiceAccount a síťových konfigurací [73]. Pro fungování produktu je třeba jeho objekty nahrát do Kubernetes clusteru. Produkt podporuje jak migraci mezi clustery s vlastním hostingem, tak i migraci mezi známými platformami (například OpenShift, AWS EKS, Microsoft AKS apod.). Výhodou tohoto produktu je to, že kromě přemístění samotných objektů umožňuje přenést i datová úložiště.

Dalším nalezeným nástrojem je Velero[2]. Tento produkt je primárně určen k zálohování dat a stavu clusteru. Migraci pak uživatel udělá tak, že zálohuje stav prvního clusteru a následně zálohu aplikuje ve druhém clusteru. Podle dokumentace projektu taková migrace předpokládá, že oba clustery běží na stejné platformě [74].

Posledním projektem, který zmíním v této práci, je Kubemove[3]. Produkt je stále ve fázi vývoje. Bude podporovat migraci Kubernetes clusteru mezi různými platformami. Podle vize projektu jádro služby bude nahráno do Kubernetes clusteru a správa migrace bude přístupná pomocí příkazu `kubectl` [75].

Porovnání těchto třech projektů je dostupné v tabulce 4.1.

	Portworx	Velero	Kubemove
Stav projektu	Dokončen	Dokončen	Ve vývoji
Podpora migrace mezi různými platformami	Ano	Ne	Ano
Podpora zálohy dat	Ano	Ano	Ne
Způsob instalace	Nasazení do clusteru		

Tabulka 4.1: Porovnání existujících projektů pro migraci clusterů

Z provedené rešerše vyplývá, že existují projekty, které by mohly splnit požadavky uživatele při migraci Kubernetes clusteru do platformy OpenShift. Společnou nevýhodou všech projektů je to, že pro jejich fungování je třeba původní cluster upravit, tj. nahrát tam nové konfigurační soubory. Takový přístup je někdy nevhodný pro větší clustery nebo pro produkční prostředí, kde chceme původní cluster nechat beze změn a neohrožovat jeho funkčnost. Z těchto důvodů se v dalších kapitolách zaměřím na vývoj vlastní aplikace, která bude umožňovat migraci s read-only přístupem ke zdrojovému clusteru.

[1]Dostupné z: `https://portworx.com/products/px-backup/` (01.05.2021)
[2]Dostupné z: `https://velero.io/` (01.05.2021)
[3]Dostupné z: `https://github.com/kubemove/kubemove` (01.05.2021)

4.2 Návrh funkčností

Hlavním cílem je vytvořit takovou automatizaci migrace, která by byla efektivnější než ruční migrace. Jak už bylo zmíněno dříve, uživatel při existenci souborů konfigurací může jednoduše objekty z Kubernetes clusteru přemístit do OpenShift clusteru pomocí příkazu `apply -f <file>`. Níže jsou uvedeny požadavky, které tento postup vylepší a zjednoduší tak, aby uživatelé preferovali použití aplikace před ruční migrací:

- **Aplikace má umět migrovat cluster bez použití konfiguračních souborů.** Tento požadavek umožní migrovat cluster uživatelům, kteří nemají přístup ke konfiguračním souborům nebo z nějakých důvodů nemají možnost je použit.
- **Aplikace má umožnit migrovat nějakou část clusteru, tj. nějaký specifický namespace, deployment apod**. Výběr dat k migraci pomocí aplikace bude jednodušší, než ruční výběr vhodných konfiguračních souborů.
- **Aplikace umožní migrovat objekt bez jeho předka. Například v případě deploymentu umožní migrovat jen jeho pody**. Tenhle případ užití se hodí uživatelům, kteří chtějí při migraci změnit strukturu clusteru. Pro ruční implementaci takového přístupu je nutné vytvoření nových konfiguračních souborů.
- **Aplikace má poskytovat náhled na strukturu Kubernetes clusteru, aby uživatel věděl, co všechno v clusteru má a jaké prvky chce migrovat**. Při ruční migraci by měl uživatel používat jiné nástroje pro zobrazení takové struktury.
- **Aplikace má po migraci zobrazit uživateli výsledek, tj. mu sdělit, jaké migrace se nepovedli a proč.** Výhodou aplikace oproti ruční migraci bude to, že uživatel dostane zpětnou vazbu rovnou pro všechny prvky, které se snaží migrovat, pak může případně chyby opravit rovnou pro všechny objekty.
- **Aplikace po migraci umožní aplikované změny vrátit zpět.** V případě ruční migrace by měl uživatel manuálně mazat veškeré zmigrované objekty.

V případě, že se povede v aplikaci implementovat všechny výše zmíněné funkčnosti, bude určitě efektivnější pro uživatele než ruční migrace.

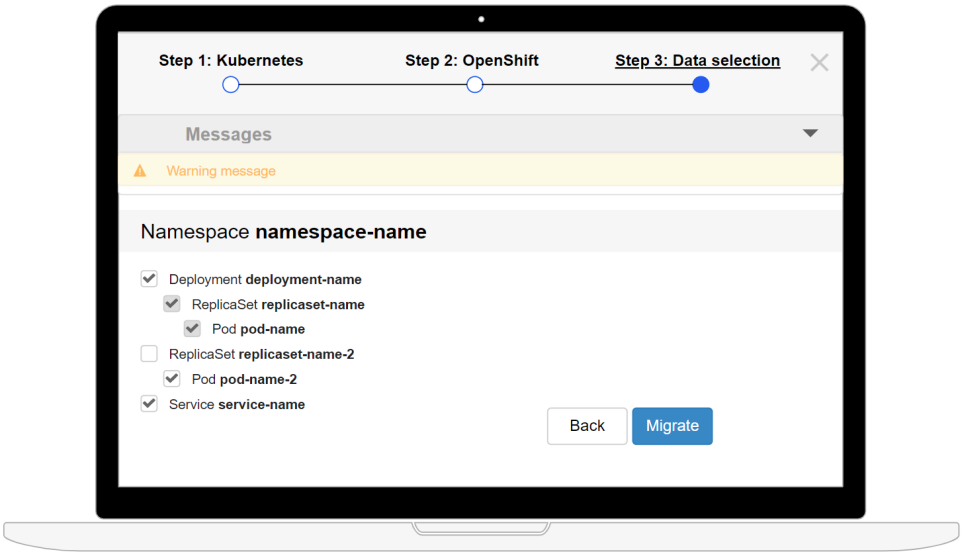
Po analýze požadavků jsem vytvořila návrh budoucí aplikace. V první verzi se bude systém skládat z pěti obrazovek.

Po spuštění aplikace uživatel uvidí úvodní obrazovku. Bude mu poskytnuta základní informace o fungování systému. Po stisknutí tlačítka „Start“ bude uživatel přesměrován na wizard pro zadání nutných dat k migraci. Prvním krokem je zadání adresy a přihlašovacích údajů ke Kubernetes clusteru. V první verzi bude aplikace podporovat přihlášení pomocí service account tokenu a Bootstrap tokenu. Po stisknutí tlačítka „Dále“ aplikace rovnou ověří, že přístup ke clusteru je funkční, případně uživateli zobrazí příslušnou chybovou hlášku. Dalším krokem je zadání stejných údajů pro OpenShift cluster. V tomto případě je podporováno základní přihlášení a autentizace pomocí tokenu. Po úspěšném ověření připojení ke clusteru bude uživatel přesměrován na poslední krok – výběr dat k migraci (obrázek 4.1).

Tato obrazovka se bude skládat ze dvou částí. V horní části aplikace zobrazí důležité informace a upozornění týkající se migrace, a to například podporované typy objektů nebo seznam systémových jmenných prostorů. Dále bude zobrazena stromová struktura celého clusteru. Pro všechny jmenné prostory budou vypsány všechny dostupné k migraci objekty (tj. ty, migraci kterých již aplikace podporuje). Zaškrtnutím checkboxu si uživatel vybere objekty, které chce migrovat, podle následující pravidel:

- Pokud je vybrán rodič, všechny jeho potomci jsou automaticky vybrány a uživatel je nemůže z výběru vyhodit. Namespace se nepovažuje za rodiče objektů. Například aplikace neumožní uživateli migrovat pouze deployment bez příslušných podů.
- Uživatel si může zvolit jenom potomka, předek potom nebude migrován. Například, pokud by ve výše zmíněném příkladu uživatel vybral jenom pod, příslušný deployment by nebyl migrován.
- Pokud je vybrán alespoň jeden objekt z daného namespace, aplikace se pokusí tento namespace vytvořit. Pokud již namespace existuje, aplikace zobrazí uživateli o tom upozornění, ale příslušné objekty stejně do tohoto jmenného prostoru zmigruje.
- Na stránce se zobrazí i systémové jmenné prostory (začínající se „kube-“). Ty jsou ale zobrazeny jenom pro úplnost struktury clusteru a uživatel je nemá možnost migrovat.

Po kliknutí na název příslušného objektu aplikace uživateli zobrazí o něm informaci, a to v podobě YAML konfigurace. Výsledek je stejný jako při zavolání příkazu `kubectl get [TYPE] [NAME] -o yaml`. Kliknutím na tlačítko „Migrovat“ uživatel začne migraci. Po její dokončení se mu zobrazí stránka s výsledky. Aplikace zobrazí seznam zmigrovaných objektů, a pokud během migrace některé objekty nemohli být přenesené (například z důvodu existence objektu se stejným názvem nebo kvůli chybějícím oprávněním), uživateli se zobrazí hláška s důvodem této chyby. Navíc systém zobrazí příkaz, pomocí kterého může uživatel manuálně zkontrolovat stav cílového clusteru po



Obrázek 4.1: Krok 3 wizardu – výběr dat k migraci

migraci. Pokud bude chtít uživatel stornovat provedené změny, musí stisknout odpovídající tlačítko, které bude dostupné při libovolném výsledku migrace.

Všechny návrhy obrazovek jsou dostupné v příloze A.

V rámci této práce budu implementovat jen první verzi aplikace, která bude podporovat omezený počet objektů k migraci. Návrh dalšího rozvoje aplikace je popsán v kapitole 10.

Kapitola 5

Implementace – vývoj testovací aplikace

Za účelem ukázky nasazení na Kubernetes a testování migrace byla vyvinutá jednoduchá testovací aplikace. Aplikace se skládá ze dvou microservice, napsaných v jazyce Java s použitím frameworku Spring Boot. Struktura aplikace a plán budoucího nasazení jsou zobrazeny na obrázku 5.1.

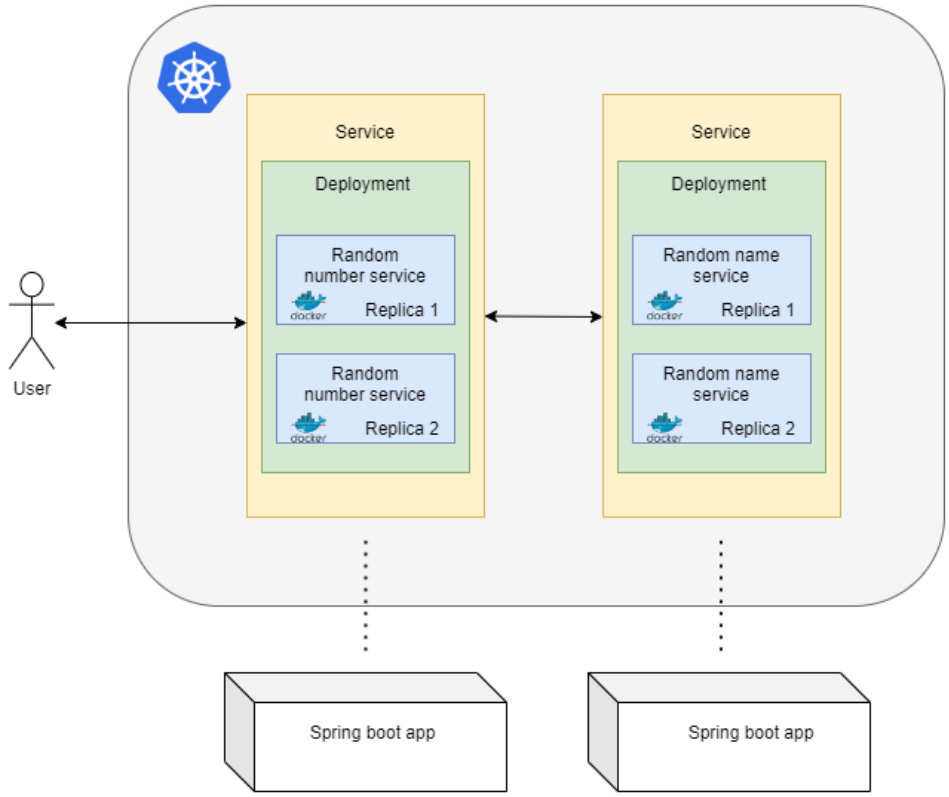
„Random name service“ při každém volání na url `/random-name/name` vrátí náhodné jméno z předdefinovaného seznamu. „Random number service“ má o trochu komplikovanější funkčnost. Při volání na url `/random-number/number` služba udělá následující:

1. Provolá „Random name service“. V případě selhání volání vrátí výchozí jméno „Dummy Name“.
2. Vrátí JSON objekt se strukturou:
   a. `name` – výsledek volání „Random name service“
   b. `number` – náhodné číslo, které se určí při spuštění instance a bude stejné do konce běhu aplikace.

Na příkladě této aplikace budu moci zkontrolovat, že migrace proběhla úspěšně, pokud obě služby budou v novém clusteru komunikovat mezi sebou a bude služba „Random number service“ dostupná přes REST API zvenku. Navíc pomocí testovací aplikace ukážu použití Kubernetes a OpenShift orchestrátorů.

5.1 Nasazení aplikace do platformy Kubernetes

Po dokončení vývoje aplikace máme k dispozici dvě služby, které mezi sebou komunikují. Jedná z nich navíc komunikuje se vnějším světem. Pro nasazení



Obrázek 5.1: Struktura testovací aplikace

těchto služeb do Kubernetes máme nejprve vytvořit Docker image, které pro jednoduchost nahrajeme do Docker Hubu (což je vzdálený repositář image). Pro vytvoření Docker image použijeme nejjednodušší způsob – manuální vytvoření Dockerfile a jeho spuštění pomocí `docker build` příkazu. Existuje určitě více způsobů, jak tento proces automatizovat, není to ale tématem této práce. Výsledný Dockerfile je uveden ve fragmentu kódu 5.1 (tady je zobrazen příklad pro „Random number service“).

```
FROM openjdk:11-jre-slim
COPY target/random-number-*.jar /app.jar
EXPOSE 8082
CMD ["java", "-jar", "/app.jar"]
```

Fragment kódu 5.1: Dockerfile pro vytvoření Docker image

Je důležité zmínit, že jelikož „Random number service“ komunikuje napřímo s „Random name service“, potřebuje nějak získat jeho adresu. Tohle nastavení se bude měnit v závislosti na prostředí, proto nebudeme adresu uchovávat přímo kódu. Místo toho použijeme proměnnou prostředí `randomservice.name.address`.

5.1.1 Iniciální nasazení

Po vytvoření Docker image můžeme začít služby nasazovat do clusteru Kubernetes. Plán nasazení je zobrazen na obrázku 5.1. Pro každou službu vytvoříme deployment, který bude spravovat dvě repliky stejného podu. Pro každou službu bude vytvořena jedna service. „Random name service“ bude mít typ ClusterIP, protože s ní komunikujeme jenom uvnitř stejného clusteru. Druhá služba, „Random number service“, bude typu LoadBalancer, což nám umožní s ní komunikovat i mimo cluster.

Pro nastavení komunikace mezi těmito službami potřebujeme zjistit IP adresu „Random name service“, a k tomu použijeme příkaz `kubectl get service`. Získanou hodnotu nastavíme jako proměnnou prostředí pomocí položky `env` ve specifikaci příslušného podu.

Výsledné YAML soubory pro nasazení se nachází v příloze B.

5.1.2 Vylepšení – best practices

Způsob, kterým byla aplikace nasazená do Kubernetes v kapitole 5.1.1, je plně funkční, ale je těžko udržovatelný. Hodí se pro případ, kdy máme malý cluster a jednu aplikaci, kterou neplánujeme nikdy přenasazovat. Ve většině případů bude ale takový způsob nasazení nepoužitelný. V následných podkapitolách se podíváme na možná vylepšení a best practices, která aplikujeme, aby nasazení maximálně odpovídalo reálným projektům. Výsledné YAML soubory spolu s kódem testovací aplikace se nachází v GitHub repositáři na adrese `https://github.com/kulyndar/openk8s-test`.

Použití jmenných prostorů

Jednou důležitou vlastností Kubernetes je podpora virtuálních clusterů, které sdílí společný fyzický cluster. Tato technologie se v terminologii K8s nazývá namespace[1]. Jmenné prostory usnadňují použití clusteru většímu množství uživatelů a nasazení více aplikací. Jednou z výhod je to, že použití namespace eliminuje riziko vzniku kolizí pojmenování zdrojů v rámci jednoho clusteru. Kubernetes vyžaduje unikátní názvy prostředků, ale jenom v rámci jednoho namespace. Názvy napříč jmenným prostorům být unikátní nemusí. Proto je vhodné zavádět namespace pro každou aplikaci, prostředí apod. Další výhodou namespace je možnost přidělení role uživatelům v konkrétním jmenném prostoru. Tím lze kompletně oddělit práci uživatelů z různých týmů a více zabezpečit aplikaci zvlášť v produkčním prostředí. Nakonec Kubernetes umožňuje nastavit limit použitých prostředků (CPU, paměť) pro

[1] česky – jmenné prostory

každý namespace zvlášť, což pomůže se vyhnout situaci, kdy jedna aplikace kvůli zvýšené spotřebě zdrojů bude zasahovat do běhu ostatních systémů.

Výchozím jmenným prostorem v Kubernetes je `default`. Právě do tohoto prostoru byly vkládány entity uvedené v podkapitole 5.1.1. Změnu namespace lze udělat dvěma způsoby: buď ji uživatel zadá v každém příkazu `kubectl` nebo ji nastaví natrvalo. Oba dva způsoby jsou uvedeny ve fragmentu kódu 5.2.

```
$ kubectl --namespace=my-ns [command] [TYPE]
$ kubectl config set-context --current --namespace
 =my-ns
```

Fragment kódu 5.2: Nastavení jmenného prostoru

Na závěr této části je důležité zmínit, že ve výchozím nastavení clusteru existuje několik systémových jmenných prostorů:

- `default` – výše zmíněný namespace, který se používá pro všechny objekty, u kterých namespace není definován.
- `kube-system` – systémový jmenný prostor, kde běží například `kube-proxy` nebo `kube-dns`.
- `kube-public` – jmenný prostor, který ve výchozím nastavení je přístupný pro všechny uživatele (dokonce i pro neautentizované). Je vyhrazen pro ukládání objektů, které by měly být přístupné pro celý cluster (není to ale nutnou podmínkou použití tohoto namespace).
- `kube-node-lease` – namespace pro vnitřní objekty Kubernetes, spojené se škálovatelností a dostupností pracovních jednotek.

Obecně platí doporučení, že uživatelsky vytvořený jmenný prostor by neměl začínat předponou `kube-`. Takové názvy jsou do budoucna vyhrazeny pro systémové objekty Kubernetes [76].

Testovací aplikaci vylepším tím, že ji přesunu do nového namespace `random`, který vytvořím příkazem `kubectl create namespace random`. Pak po nastavení tohoto jmenného prostoru jako kontextu aplikuji výše uvedené konfigurační soubory pro vytvoření všech objektů testovací aplikace.

Nastavení limitu prostředků

Motivace definice minimálních a maximálních zdrojů pro pody byla již uvedena v podkapitole 3.1.3. Jedná se o případy, kdy chceme zajistit, že buď pod dostane minimálně požadované zdroje, nebo nepřesáhne definovaný limit a nebude vytěžovat celý cluster. V předchozí kapitole bylo taky zmíněno, že

existuje způsob přiřazení limitů na zdroje pro celý namespace. Použijeme tyto přístupy pro vylepšení testovací aplikace.

Za prvé definujeme minimální a maximální požadavky pro konkrétní pody. Vzhledem k tomu, že se jedná o jednoduchou aplikaci, nastavíme jí následující limity (stejné pro každý pod):

- minimální: cpu – 250m, paměť – 128M.
- maximální: cpu – 500m, paměť – 256M.

Tím zaručíme, že celá aplikace včetně všech replik potřebuje minimálně 1 CPU a 512 Mb paměti. Maximálně však zatíží cluster na 2 CPU a využije 1024 Mb paměti.

Pro nastavení limitů pro namespace použijeme dříve nezmíněný objekt Kubernetes – ResourceQuota. Tento objekt je agregovaným požadavkem na zdroje pro celý jmenný prostor. Dovolí požádat o stejné prostředky, které se definují na podech, a to maximální (respektive minimální) paměť a cpu. Při tom platí, že pokud je nějaká z těchto hodnot definována v ResourceQuota, musí být pak definována na všech podech. Navíc ResourceQuota se použije pro zadání dalších omezení pro jmenný prostor, a to například maximální počet podů, service apod.

Pro celý namespace testovací aplikace vytvoříme ResourceQuota, kde specifikujeme maximální paměť a maximální počet service typu Load Balancer. Pro připomenutí jen uvedu, že service s typem Load Balancer je přístupná mimo cluster, takže slouží propojením vnější sítě Internet a vnitřní sítě clusteru. V tomto případě víme, že takový Load Balancer potřebujeme jen jeden. Maximální velikost paměti omezíme na 1300 Mb. Teď když zkusíme vytvořit ještě jednu service s typem Load Balancer nebo přidat další repliku v každém z deploymentů (suma maximální dostupné paměti pak bude 1536 Mb, protože každá replika požaduje 256 Mb paměti), Kubernetes zabrání vytvoření těchto objektů. Důvodem je to, že přesahují limity, nastavené pro namespace.

Dalším objektem pro nastavení limitů je LimitRange. Je podobný ResourceQuota s tím rozdílem, že ResourceQuota se nastaví pro celý namespace a definuje omezení pro všechny objekty namespace dohromady. LimitRange ale limity nastaví pro každý typ objektů zvlášť v rámci daného namespace. Například vytvoříme LimitRange pro pody s maximální pamětí 512 Mb a požadovanou paměti 128 Mb. Mohou pak nastat následující případy:

1. **Pod má definovány limity na zdroje a tyto limity jsou v souladu s LimitRange (požadované prostředky jsou větší, než je definováno na LimitRange, a maximální – nižší)**. V tomto případě se použijí limity, definované na konkrétním podu.

2. **U pod se definují limity na zdroje, které nejsou v souladu s LimitRange**. V tomto případě Kubernetes zabrání uživateli vytvořit pod.
3. **Pod nemá nastaveny limity**. Pro požadované a maximální zdroje se použije výchozí hodnota z LimitRange (nebo maximální hodnota, pokud výchozí není specifikována).
4. **Pod má specifikován jenom jeden limit (buď požadovaný nebo maximální)**. V tomto případě se vždy zvolí větší hodnota z maxima pro tento pod a výchozí hodnoty z LimitRange.

Například pokud u jednoho z naších podů odebereme požadavek na limity, nastaví se mu požadována a maximální paměť na 512 Mb. Musíme ale dávat pozor, aby upravené hodnoty byly v souladu s dříve nastavenou ResourceQuota.

Detekce připravenosti a živosti podů

Jednou z výhod orchestrace kontejnerů je schopnost automatické opravy. V případě, že orchestrátor detekuje, že nějaký pod neběží, zkusí ho znovu spustit. Tento proces probíhá následně:

1. Řídicí jednotka se periodicky ptá pod, zda všechny jeho kontejnery běží.
2. Pokud nějaký z kontejnerů na požadavek neodpověděl, označí se jako „nezdravý“.
3. Load balancer přestává posílat požadavky na „nezdravý“ kontejner, zatímco řídicí jednotka se ho snaží restartovat.
4. Řídicí jednotka periodicky posílá požadavek, aby zjistila, zda je kontejner připraven přijímat trafik.
5. Až bude kontejner připraven, load balancer ho znovu použije jako fungující backend.

Kubernetes umožňuje vývojářovi definovat vlastní metody pro detekci živosti a připravenosti kontejnerů. Tomu se říká **liveness probe**, respektive **readiness probe**. Uživatel má celkem tři možných způsoby implementace těchto metod. První z nich je HTTP probe, jedná z nejpoužitelnějších technologií. Kontejner se považuje za „zdravý“, případně připravený, pokud při dotazu na uživatelem definovanou adresu vrátí status v rozsahu 200 až 400. Dalším způsobem je Command probe, jehož princip je docela podobný. V tomto případě se kontejneru pošle nějaký příkaz. Pokud příkaz vrátí 0, kontejner je považován za zdravý a připravený. Posledním způsobem je TCP

probe, kdy řídicí jednotka se pokusí navázat TCP spojení na definovaném portu. Pokud bude připojení navázáno, kontejner je považován za „zdravý“.

Pro uplatnění těchto principů u testovací aplikace použijeme HTTP probe. Potřebujeme proto u obou microservice implementovat nový controller, který bude vždy vracet HTTP 200 OK. To, že při chybě kontejneru bude vracen kód větší nebo roven 400, zajistí používaná technologie, tj. Java a framework Spring Boot. Ve fragmentu kódu 5.3 vidíme, jak se změnil YAML soubor pro deployments. Taková konfigurace říká, že pro detekci živosti každých 10 sekund bude posílán dotaz na adresu `/random-number/health`. První dotaz se pošle po 3 sekundách po spuštění kontejneru. V případě, že bude více než třikrát vracen kód větší než 400, kontejner je považován za nezdravý. Pro detekci připravenosti kontejneru se každých 2 sekundy pošle požadavek na stejnou adresu. První požadavek se pošle rovnou po spuštění kontejneru.

```
...
spec:
  containers:
  ...
    livenessProbe:
      httpGet:
        path: /random-number/health
        port: 8082
      initialDelaySeconds: 3
      periodSeconds: 10
      failureThreshold: 3
    readinessProbe:
      httpGet:
        path: /random-number/health
        port: 8082
      initialDelaySeconds: 0
      periodSeconds: 2
```

Fragment kódu 5.3: Liveness a readiness probes pro Random number service

Řízení přístupů k objektům

Nedílnou součástí Kubernetes clusteru je řízení přístupu k jeho objektům. K8s poskytuje mnoho možností autentizace a autorizace. V této kapitole se ale budu zabývat jen autorizací, tj. vyhodnocením oprávnění pro konkrétního uživatele. Způsoby přihlášení se liší v závislosti na typu instalace Kubernetes (vlastní hosting, lokální instalace, cloud provider apod.), proto se tímto tématem v práci zabývat nebudu.

Kubernetes poskytuje 4 způsoby řízení autorizace:

1. Node – speciální způsob autorizace, používaný `kubelety` za účelem řízení

běhu podů.

2. ABAC (Attribute-based access control) – způsob autorizace, při kterém se definují zásady pro přístup[2]. Autorizace je založená na tom, že administrátor definuje množinu pravidel pro přístup ke zdrojům na základě jejich atributů (třeba typu zdroje). Jinými slovy, pro každý požadavek se vyhodnocují jeho atributy, a pokud tyto atributy odpovídají pravidlům, požadavek je autorizován.

3. RBAC (Role-based access control) – na rozdíl od ABAC je založen na principu existence rolí. Uživatelům je přiřazená množina rolí, každá definuje seznam povolených akcí. Kubernetes nedovolí definovat zákazy akcí.

4. Webhook – tento způsob autorizace dovoluje implementovat vlastní logiku řízení přístupů. Při pokusu o autorizaci Kubernetes posílá POST požadavek na uživatelem definovaný REST server. Nevýhodou tohoto přístupu je to, že konfigurace Webhook autorizace se definuje při spuštění apiserveru a nelze ji za běhu clusteru měnit (na rozdíl od RBAC autorizace, kde pro přidání role stačí nahrát nový konfigurační soubor pomocí `kubectl`).

V této práci se budu zabývat autorizaci RBAC. Kubernetes poskytuje 4 hlavních objekty pro řízení rolí. Objekt Role definuje seznam povolených akcí, které může učinit vlastník této role. Role platí jen v rámci jednoho jmenného prostoru. Pro přiřazení role ke konkrétnímu uživateli se používá RoleBinding. Analogicky existuje i dva zdroje pro nastavení přístupů k objektům mimo jmenný prostor nebo k objektům ve všech namespace zároveň. Takovým zdrojům se říká ClusterRole, respektive ClusterRoleBinding.

Uživatelem, který bude v budoucnu přistupovat ke clusteru, je aplikace pro migraci. Musí mít možnost přečíst objekty z libovolného namespace. Proto vytvoříme pro nového uživatele `cluster-reader` novou cluster roli (ClusterRole). Tato role bude mít přístup ke všem objektům clusteru, ale seznam povolených operací bude obsahovat jen read-only příkazy typu `get` nebo `describe`. Tím zaručíme, že migrační aplikace nezmění nic ve stávajícím clusteru, ale bude mít možnost stav tohoto clusteru přečíst.

Service discovery uvnitř clusteru

Ve většině microservisních aplikací jednotlivé služby komunikují mezi sebou. Problém objevování služeb jsem popisovala v kapitole 2.3. Kubernetes poskytuje několik možností, jak můžeme se službou komunikovat. První a nejjednodušší možnost je použití IP adresy služby, kterou může uživatel zjistit

[2] angl. Policies

pomocí příkazu `kubectl get service <service-name>`. Tento způsob jsme použili při iniciálním nastavení testovací aplikace. Nevýhodou je ale to, že není zaručeno, že IP adresa zůstane stejná. Při její změně bude komunikace narušena.

Další možností, kterou poskytuje Kubernetes, jsou proměnné prostředí [3]. Při startu kontejneru K8s propaguje proměnné s hostem a portem každé běžící service. Například pro službu `random` budou propagované dvě proměnné:

- `RANDOM-SERVICE-HOST` – host služby,
- `RANDOM-SERVICE-PORT` – port služby.

Nevýhodu tohoto přístupu je vidět ze samotného algoritmu propagace – pořadí spuštění služeb a kontejnerů je důležité. Pokud bude kontejner spuštěn před spuštěním služby, její host a port nebudou propagované jako proměnné prostředí.

Analogicky ke komunikaci mimo cloud existuje možnost použití DNS jména. Po nasazení bude služba dostupná pomocí následujících DNS jmen:

- V rámci stejného namespace může být služba dostupná pomocí hostu `<service-name>`.
- V rámci jiného namespace je služba dostupná pomocí hostu `<service-name>.<namespace-name>`.
- Služba může být dostupná pomocí celého DNS jména, a to ve formátu `<service-name>.<namespace-name>.svc.<cluster-domain>`

Pro naši testovací aplikaci požijeme druhý typ DNS jména, tj. `random-name-service` bude teď dostupná na adrese `random-name-service.random`.

[3] angl. environment variables

Kapitola 6

Implementace – přidání pluginů

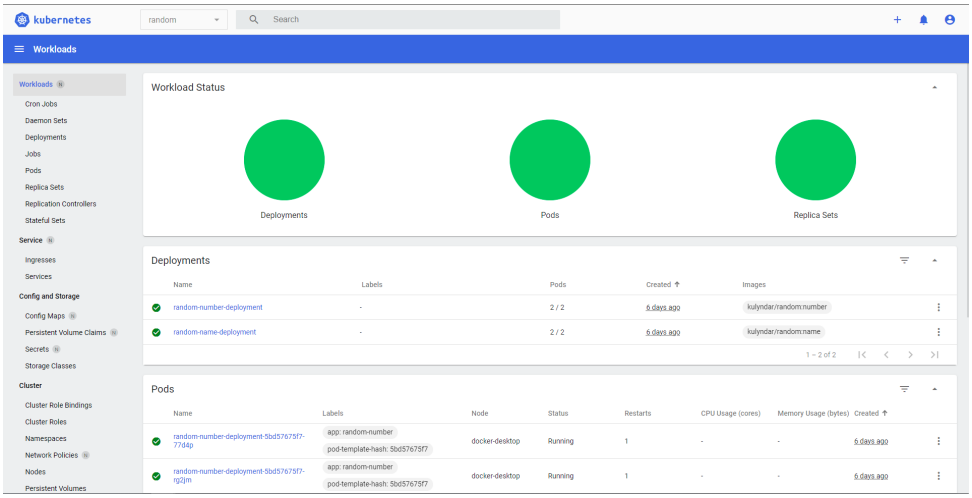
Jak jsme mohli vidět z kapitoly 3.1, Kubernetes je silný nástroj pro orchestraci kontejnerů. Neobsahuje ale některé věci, které by velké množství uživatelů potřebovalo. Takové nástroje mohou uživatele přidat pomocí pluginů. Jeden z pluginů, který jsme již měli předinstalován a který jsme již používali při vylepšení testovací aplikace, je `core-dns` plugin. Je to DNS server, který je rychlejší a flexibilnější než vestavěný `kube-dns` server [77].

Další plugin, který může být vhodný i pro malé projekty, je Kubernetes Dashboard. Jedná se o webovou stránku, pomocí které může uživatel komunikovat s clusterem bez použití `kubectl`. Tento plugin se instaluje jedním příkazem `kubectl apply -f <url>`. Adresa pro instalaci pluginu záleží na jeho verzi a lze ji najít na stránkách projektu na GitHubu[1]. Po instalaci tohoto pluginu bude dostupná webová stránka, pomocí které můžeme zobrazovat a upravovat různé objekty clusteru, například pody, deploymenty, role apod. Příklad rozhraní pluginu je vidět na obrázku 6.1.

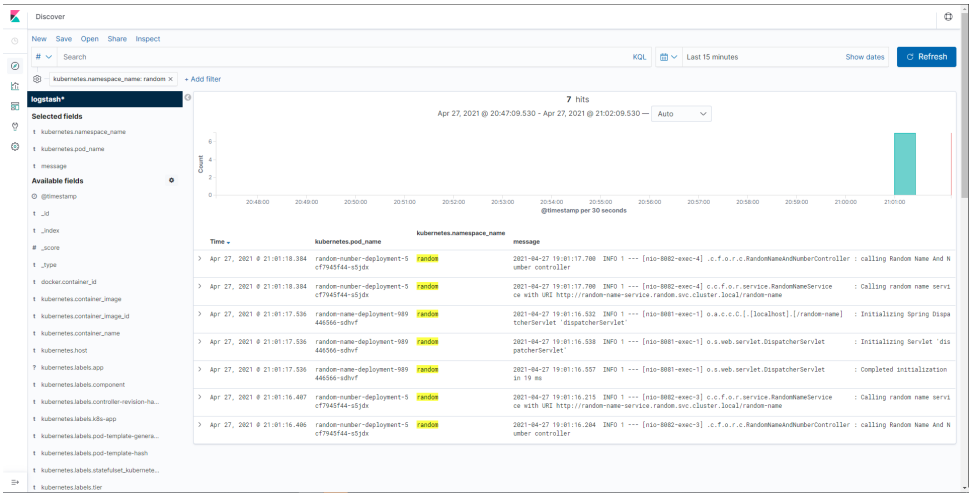
Důležitou otázkou zvlášť pro produkční prostředí je sledování logů aplikace. Kubernetes podporuje zobrazení logů pomocí příkazu `kubectl logs`, ale výstup tohoto programu není dost přehledný. Navíc pomocí tohoto příkazu je možné zobrazit logy jenom od momentu spuštění podu. Většinou bude ale uživatel chtít logy uchovávat delší dobu, umět je zpracovávat, filtrovat apod. V kapitole 3.2 jsem zmiňovala, že jedním z rozdílů mezi OpenShift a Kubernetes je vestavěná podpora EFK v clusteru OKD. Do Kubernetes takovou funkčnost lze přidat pomocí pluginu, zdrojové kódy kterého se nachází v repositáři pluginů projektu Kubernetes[2]. Na rozdíl od předchozího pluginu, EFK se neinstaluje jedním příkazem. Uživatel musí aplikovat všechny konfigurace z výše zmíněného repositáře. Po nahrání souborů do clusteru stačí jen přesměrovat tok dat na externí port pomocí

[1] Dostupné z: `https://github.com/kubernetes/dashboard` (23.04.2021)

[2] Dostupné z: `https://github.com/kubernetes/kubernetes/tree/master/cluster/addons` (27.04.2021)



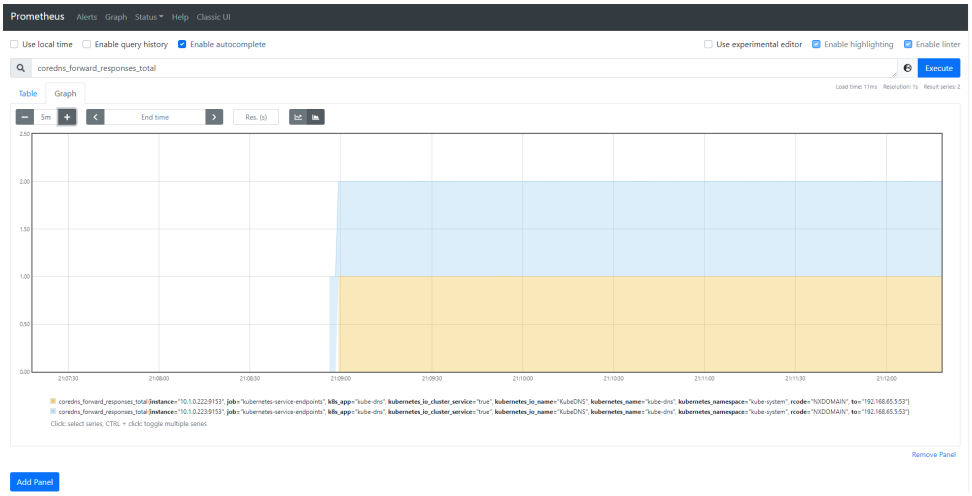
Obrázek 6.1: Kubernetes Dashboard – rozhraní



Obrázek 6.2: Kibana – rozhraní přehledu logů

příkazu `kubectl port-forward <kibana-pod> 5061 -n logging`. Potom bude aplikace dostupná přes webové rozhraní. U lokální instalace Kubernetes kromě výše zmíněného příkazu je třeba spustit proxy pomocí `kubectl proxy`, aby byla aplikace dostupná i bez přihlášení. Dále již bude EFK sbírat logy jednotlivých podů v clusteru (obrázek 6.2).

Stejně důležitý problém jako monitoring logů je sběr metrik. Za tímto účelem je pro Kubernetes dostupný plugin pro integraci Prometheus serveru. Oficiální postup instalace tohoto pluginu [78] vyžaduje použití programu Helm, kterým jsem se v rámci této práce nezabývala. Z těchto důvodu jsem vytvořila YAML soubor, po nahraní kterého bude v clusteru spuštěn oficiální Docker image Prometheus. Konfiguraci serveru může uživatel upravovat změnou odpovídajícího ConfigMap objektu. Soubor se nachází v repositáři `https://github.com/kulyndar/openk8s-test`. Spuštěním příkazu



Obrázek 6.3: Prometheus – ukázka sběru metrik

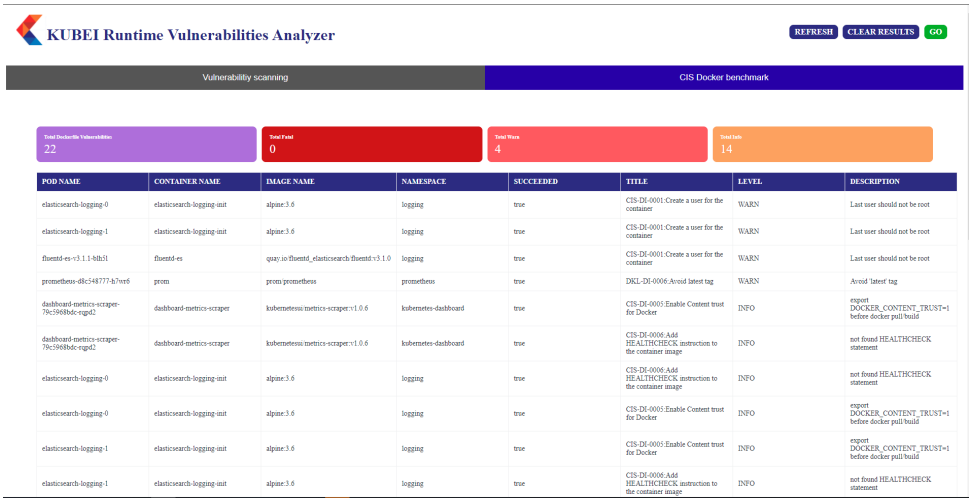
`kubectl apply -f <filename>` vytvoříme mimo jiné service typu NodePort, která bude přístupná z adresy `http://<node-ip>:30021`. Příklad výsledného rozhraní Prometheus je vidět na obrázku 6.3.

Posledním pluginem, na který se zaměřím v rámci této práce, je Kubei. Toto rozšíření má za účel sledovat stav zabezpečení clusteru a objevovat bezpečnostní rizika. Analyzuje všechny image použité v clusteru a provádí analýzu jejich bezpečnosti, například vyhledává známé bezpečnostní díry v CVE (databáze známých bezpečnostních ohrožení) [79]. Instalace tohoto pluginu je snadná – uživatel příkazem `kubectl apply -f <url>` nahraje veškeré konfigurace. Odkaz pro stažení konfigurací je možné najít na stránkách projektu[3]. Po instalaci veškerých objektů stejně jako v případě EFK má uživatel udělat přesměrování trafiku na port 8080. Pak v prohlížeči přes cestu `/view` se dostane na webovou aplikaci pluginu. Stisknutím tlačítka „Go“ uživatel začne kontrolu celého clusteru. Po obnovení stránky se mu zobrazí výsledky (obrázek 6.4).

Jak může čtenář vidět, i přestože v samotném Kubernetes některé důležité funkčnosti chybí, je možné je přidávat jako pluginy. Složitost instalace se liší v závislosti na konkrétním rozšíření: v jednom případě stačí jeden příkaz, v jiném – použití vedlejších programů typu Helm. Další pluginy může uživatel nalézt v oficiální dokumentaci Kubernetes[4] nebo přímo na stránkách jednotlivých produktů.

[3] Dostupné z: `https://github.com/Portshift/kubei` (28.04.2021)

[4] Dostupné z: `https://kubernetes.io/docs/concepts/cluster-administration/addons/` (28.04.2021)



Obrázek 6.4: Kubei – sledování zabezpečení clusteru

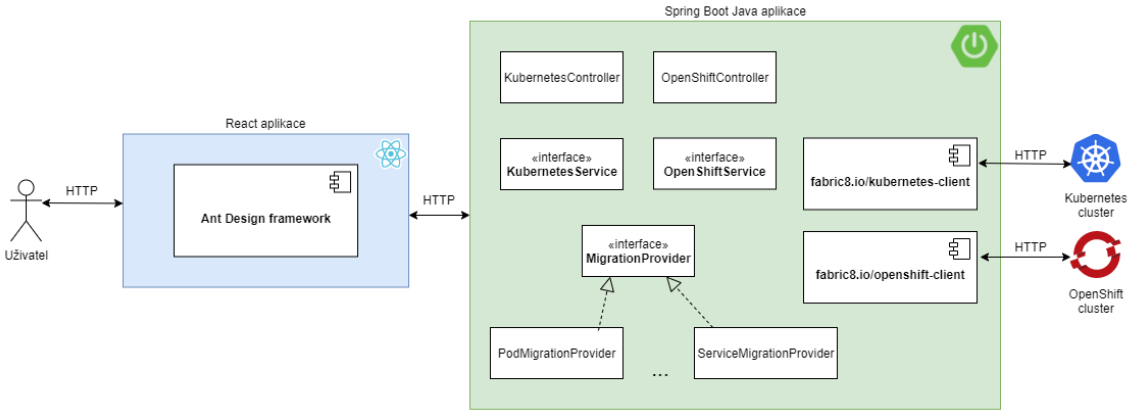
Kapitola 7

Implementace – migrace aplikace do OpenShift

Pro vývoj aplikace jsem si rozhodla zvolit technologie, se kterými nejčastěji pracuji a které jsou pro tento typ aplikací vhodné. Pro implementaci frontendu jsem zvolila React framework[1], který pomůže s vykreslením objektů a zvládne komplexní logiku výběru dat k migraci. Ke stylování jsem použila Ant Design framework[2], který poskytuje velké množství různých komponent. Výhodou tohoto balíčku je to, že komponenty v sobě již zapouzdří základní logiku (zobrazení modálního okna, udržování dat formuláře apod.), proto se vývojář může soustředit na složitější úkoly. Jako backend jsem použila Spring Boot[3] aplikaci, se kterým frontend komunikuje přes REST API. Pro komunikaci s clustery Kubernetes a OpenShift používám Java klienta od fabric8.io[4]. Výhodou tohoto klienta je to, že poskytuje kompatibilní rozhraní pro oba orhestrátory. Výsledná architektura aplikace je zobrazená na obrázku 7.1.

Backend aplikace má klasickou vrstevnatou architekturu. Kontroller, který příjme HTTP požadavek, přesměruje ho na zodpovědnou service, která požadavek zpracuje a vrátí výsledek. Aplikace v první verzi má dva controllery a dvě příslušné service – pro Kubernetes (řeší připojení ke clusteru a získání jeho struktury) a pro OpenShift (je zodpovědná za připojení ke clusteru a za samotnou migraci). Jelikož Kubernetes poskytuje velké množství objektů a každý z nich má různé rozhraní, pravidla pro migraci apod., pro práci s objekty bylo vytvořeno rozhraní `MigrationProvider`. Každá implementace tohoto rozhraní je zodpovědná za jeden typ objektů. Pomocí klientu od fabric8.io získává údaje o objektu a provádí migraci. Taková architektura splňuje Open-closed princip ze známé množiny principů SOLID, který říká, že objekt má být otevřen pro rozšíření, ale uzavřen pro modifikaci. Jinými slovy, v případě přidání nového objektu bude nutné vytvoření nové implementace

[1] Dostupné z: `https://reactjs.org/` (27.04.2021)
[2] Dostupné z: `https://ant.design/` (27.04.2021)
[3] Dostupné z: `https://spring.io/projects/spring-boot` (27.04.2021)
[4] Dostupné z: `https://github.com/fabric8io/kubernetes-client` (27.04.2021)



Obrázek 7.1: Architektura migrovací aplikace

provideru, nikoliv změna existujícího kódu. Provider má za účel v první verzi implementovat následující metody:

- `getKind` – metoda, která vrátí podporovaný typ objektu.
- `isResponsibleFor` – metoda, která rozhodne, zda je daná implementace provideru zodpovědná za nějaký Kubernetes objekt, tj. zda její implementace umožní tento objekt zmigrovat.
- `getInfo` – metoda, která vrátí informaci o objektu ve formátu jednoho řetězce. Informace se následně zobrazí v modalu při kliknutí na název zdroje.
- `migrateResource` – metoda, která zmigruje objekt do nového clusteru. Implementace nemusí řešit případně chyby při migraci (existence objektu se stejným názvem, chyba připojení, nedostatečná oprávnění apod.). Chyby jsou řešeny v rámci default metody `migrate`. Pokud bude nutné odchycovat specifické chyby, potomek je schopen tuto metodu přepsat.
- `getLogger` – pomocná metoda, která vrátí objekt pro zalogování případných chyb.
- `rollbackResource` – metoda, která smaže zmigrovaný objekt z cílového clusteru.

Kromě samotné migrace aplikace podporuje i návrat provedených změn, a to při libovolném výsledku migrace. Vzhledem k tomu, že komunikace probíhá přes REST API, které je podle RESTful principů bezestavové, požitý klient neposkytuje možnost transakčního zpracování. Pro návrat změn aplikace udržuje seznam provedených migrací v paměti a příslušné objekty pomocí stejného REST API smaže, jak by to udělal uživatel ručně.

Zdrojový kód výsledné aplikace je umístěn v repositáři `https://github.com/kulyndar/openk8s`.

Kapitola 8

Testování migrace do OpenShift

Posledním krokem vývoje migrační aplikace je otestování správnosti funkčnosti. Cílem aplikace je umožnit uživateli migrovat existující objekty do jiného clusteru, ideálně tak, aby byly rovnou funkční a nepotřebovaly další konfigurace ani úpravy. Proto jako způsob testování jsem zvolila provedení integračních testů uvnitř clusteru. Výhodou je to, že rovnou můžu vyzkoušet, že migrované objekty mezi sebou stále komunikují. Dalším důvodem zvolení takového typu testování je to, že většina projektů, udržovaných v Kubernetes clusteru, již mají definovány vlastní testy, pomocí kterých se dá po migraci funkčnost clusteru zkontrolovat. Pokud by uživatel takové testy neměl, může se inspirovat popsaným níže způsobem testování a pomocí pluginu Cicada definovat vlastní testy.

Záměrem testování je tedy zkontrolovat, že každá migrovaná service je stále dostupná a že funguje komunikace mezi jednotlivými službami uvnitř clusteru. Komunikaci služeb mimo cluster v rámci této práce netestuji automaticky, jelikož nastavení dostupnosti služeb mimo cluster se mění v závislosti na způsobu hostingu. Pro testování služeb typu ClusterIP, které mají přiřazenou IP adresu jenom uvnitř clusteru, musím použit nějaký objekt, který do toho clusteru nahraji a který uvnitř clusteru provede testování. Takové rozhraní poskytuje framework Cicada[1]. Existuje několik důvodů, proč byl tento projekt zvolen:

1. Testovací jádro běží přímo v clusteru, což mu umožňuje komunikovat se všemi pody.
2. Testy lze spustit paralelně a opakovaně.
3. Lze definovat závislosti testů, a to jak pořadí spuštění, tak i závislost na stavu nějakého podu nebo služby.

[1]Dostupné z: `https://cicadatesting.github.io/cicada-2/` (29.04.2021)

4. Framework poskytuje několik způsobů testování (například testování pomocí REST, SQL apod).
5. Po dokončení testování se generuje report.

Způsob instalace frameworku je podobný instalaci pluginů. Instalační příručka pro Kubernetes a platformy založené na Kubernetes lze nalézt v dokumentaci projektu [80].

Testování se začíná definicí testů, které mají proběhnout. Testy se definují pomocí YAML konfigurace. Každý test musí mít unikátní jméno, které slouží identifikátorem testu v systému. Další důležitou položkou je runner, který definuje způsob testování. Jak už bylo zmíněno dříve, mohou to být REST testy, SQL apod. Na typu runneru záleží, jaké akce a kontroly[2] můžeme definovat. Uživatel si může implementovat vlastní runner a uvést ho jako image při definici testu. Po dokončení kostry testu se vrhneme na definici akcí a kontrol. Pro pochopení k čemu tyto objekty slouží, koukneme na životní cyklus testu [80] :

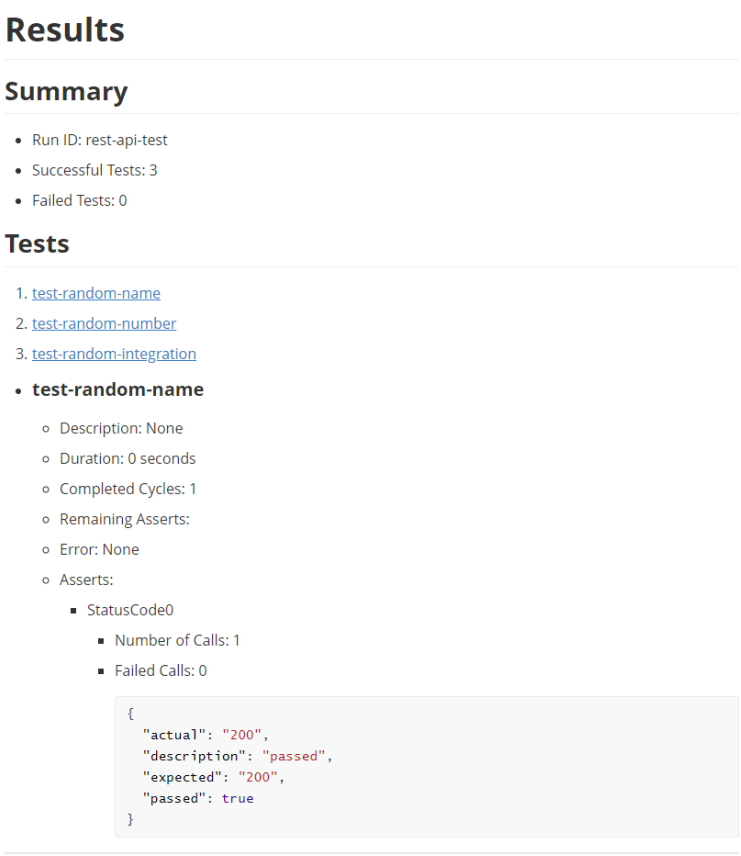
1. Systém spustí každou definovanou akci.
2. Následně se spustí všechny asserty.
3. Pokud ještě zbyly nějaké asserty (v případě, že test neprošel a máme nastaveno, že chceme neúspěšné testy několikrát opakovat), zase spustíme všechny akce a potom jen zbylé kontroly.

Akce je něco, co musíme v systému provést, abychom připravili data pro testování. Na výsledku akcí nezáleží, ty se nekontrolují. Například před kontrolou načítání dat z databáze musíme nejprve data vložit, ale v rámci tohoto testu nekontrolujeme, zda se správně ukládají. Výsledek akce se ale ukládá jako stav testu, a ten lze pak získat při spuštění assertu. Assert je akce, výsledek které chceme zkontrolovat. Parametry těchto objektů záleží na konkrétním runneru.

```
tests:
  - name: test-random-name
    runner: rest-runner
    asserts:
      - type: StatusCode
        params:
          method: GET
          actionParams:
            url: http://random-name-service.random
.svc.cluster.local/random-name/name
          expected: 200
```

Fragment kódu 8.1: Cicada – příklad testu

[2] angl. asserts



Results

Summary

- Run ID: rest-api-test
- Successful Tests: 3
- Failed Tests: 0

Tests

1. test-random-name
2. test-random-number
3. test-random-integration

- **test-random-name**
  - Description: None
  - Duration: 0 seconds
  - Completed Cycles: 1
  - Remaining Asserts:
  - Error: None
  - Asserts:
    - StatusCode0
      - Number of Calls: 1
      - Failed Calls: 0

```
{
  "actual": "200",
  "description": "passed",
  "expected": "200",
  "passed": true
}
```

Obrázek 8.1: Cicada – report provedených testů

Ve fragmentu kódu 8.1 je uveden příklad testu, který kontroluje, že služba „Random name service“ z testovací aplikace komunikuje uvnitř clusteru. Pro testování požíváme REST runner, který udělá HTTP dotaz na zadanou adresu, kterou definujeme v položce `actionParams.url`. Dále si nastavíme typ kontroly. REST runner podporuje celkem tři typy: StatusCode (kontrola vraceného HTTP statusu), Headers (ověření hlaviček odpovědi) a JSON (porovnání konkrétních položek v odpovědi). Jelikož nám služba vrací vždy náhodná jména, nastavíme si způsob kontroly StatusCode a očekávaný výsledek 200.

Takto definované testy máme nahrát do perzistentního úložiště v systému, ke kterému bude mít testovací jádro přístup. Tímto testovacím jádrem slouží TestEngine, což je vlastní objekt v Kubernetes. Při jeho konfiguraci stačí jen definovat umístění testů. Po nahrání definice TestEngine vytvoří pod, který bude řídit celý běh testů a generování reportů, a také další pody pro každý test. V momentě kdy testy skončí, řídicí pod uloží výsledky do předem definovaného umístění v clusteru. Výsledek testu je uložen ve formátu Markdown a jeho příklad je vidět na obrázku 8.1.

Sada testů, definice kterých se nachází v příloze C, umí po nahrání do

OpenShift clusteru ověřit, že po migraci testovací aplikace stále funguje, všechny její služby jsou dostupné a komunikují mezi sebou. Tím můžeme považovat migraci za úspěšnou.

Kapitola 9

Závěr

Všechny definované cíle projektu byly splněny. V rámci práce byla prozkoumaná problematika kontejnerizace a orchestrace kontejnerů. Byla zpracovaná rešerše populárních platforem pro orchestraci kontejnerů, popsány jejich vlastnosti a výhody. Dále se práce věnovala prozkoumání vybraných platforem – Kubernetes a OpenShift, popisu jejich vlastností a architektury. Bylo ukázáno použití zmíněných orchestrátorů pomocí klientů příkazové řádky a definice konfiguračních souborů. V rámci praktické části byla vyvinutá testovací microservisní aplikace pro následnou ukázku nasazení do platformy Kubernetes. Po nasazení byl cluster Kubernetes rozšířen o pluginy pro logování, sběr metrik, analýzu bezpečnosti clusteru a vizualizaci práce s objekty Kubernetes. Nedílnou součástí práce byl vývoj aplikace pro migraci clusteru Kubernetes do platformy OpenShift. Pro ukázku migrace a její otestování byla použitá dříve zmíněná testovací aplikace.

Výstupem diplomové práce je první verze aplikace pro migraci clusteru Kubernetes do prostředí OpenShift. Vzhledem k tomu, že na trhu je představen jen malý počet podobných produktů, bude aplikace přínosná, použitelná a pravděpodobně získá nějaké množství uživatelů. Důležitým výsledkem práce je i pochopení problematiky kontejnerizace a orchestrace kontejnerů. Po přečtení tohoto projektu i čtenář bez předchozích zkušeností s kontejnerizací bude moci začít pracovat s Kubernetes platformou a migrovat své aplikace do clusterů. Pro mne byla práce zajímavá z pohledu aplikace teoretických znalostí, získaných během studia, na reálné použití populárních orchestrátorů a nasazení vlastního systému do platformy Kubernetes.

Kapitola 10

Návrh dalšího rozvoje aplikace

V rámci této práce jsem vytvořila první verzi aplikace pro migraci clusteru z Kubernetes do OpenShift. Cílem implementace bylo ukázat, že taková aplikace může být efektivnější než ruční migrace a že stojí za to tento projekt dále rozvíjet.

Jedním z nejdůležitějších kroků pro rozvoj aplikace je rozšíření seznamu objektů pro migraci. V první verzi byly implementovány jenom základní typy objektů jako jsou pody, deploymenty apod. Aplikace vždy uživatele informuje, co mu umožní migrovat a co bude muset uživatel migrovat ručně. Dalším možným krokem je přidání nových způsobů autentizace, například pomocí certifikátů.

Při rozvoji aplikace je nutné uživateli dávat co nejvíc možností si přizpůsobit migraci clusteru. Vzhledem k tomu, že se clustery nemusí být úplně totožné (například mají různý rozsah IP adres apod.), aplikace může některé objekty migrovat vícero způsoby. Například v případě service v první verzi nikdy nemigruje položku `clusterIp`, protože by to mohlo způsobit chybu migrace. V dalších verzích by bylo přidáno nastavení, pomocí kterého by uživatel mohl migraci této položky zapnout, kdyby věděl, že rozsah IP adres clusterů je stejný.

Literatura

[1] SECTION. *A Brief History of Container Technology | Section* [online]. 26.10.2020. [cit. 2020-10-26]. Dostupné z: https://www.section.io/engineering-education/history-of-container-technology/.

[2] *Chapter 14. Jails* [online]. 26.10.2020. [cit. 2020-10-26]. Dostupné z: https://www.freebsd.org/doc/handbook/jails.html.

[3] OSNAT, R. *A Brief History of Containers: From the 1970s Till Now* [online]. 21.10.2020. [cit. 2020-10-26]. Dostupné z: https://blog.aquasec.com/a-brief-history-of-containers-from-1970s-chroot-to-docker-2016.

[4] IBM Cloud Education. *Containerization* [online]. 2019. [cit. 2020-10-11]. Dostupné z: https://www.ibm.com/cloud/learn/containerization.

[5] FRAJTÁK, K. Multi-container applications, container orchestration, automatic deployment and scaling. Dostupné z: https://moodle.fel.cvut.cz/pluginfile.php/246928/mod_resource/content/0/10%20Multi-container%20applications%2C%20container%20orchestration%2C%20automatic%20deployment%20and%20scaling.pdf.

[6] API Blog: Everything You Need to Know. *4 Examples of Microservices in Action | DreamFactory Blog* [online]. 2020. [cit. 2020-10-27]. Dostupné z: https://blog.dreamfactory.com/microservices-examples/.

[7] JRebel by Perforce. *2020 Java Microservices Report | JRebel by Perforce* [online]. 18.12.2020. [cit. 2020-12-19]. Dostupné z: https://www.jrebel.com/blog/2020-java-microservices-report.

[8] The New Stack. *From Monoliths to Microservices: An Architectural Strategy – The New Stack* [online]. 2016. [cit. 2020-12-19]. Dostupné z: https://thenewstack.io/from-monolith-to-microservices/.

[9] Load Balancer. *Load Balancing Algorithms and Techniques* [online]. 19.12.2020. [cit. 2020-12-19]. Dostupné z: https://kemptechnologies.com/load-balancer/load-balancing-algorithms-techniques/.

[10] Learning Center. *Load Balancing Algorithms | Network and Application Layer | Imperva* [online]. 09.12.2020. [cit. 2020-12-19]. Dostupné z: https://www.imperva.com/learn/availability/load-balancing-algorithms/.

[11] *How Elastic Load Balancing works - Elastic Load Balancing* [online]. 19.12.2020. [cit. 2020-12-19]. Dostupné z: https://docs.aws.amazon.com/elasticloadbalancing/latest/userguide/how-elastic-load-balancing-works.html.

[12] Daniel Abadi. Data Partitioning. s. 599–600. Dostupné z: https://link.springer.com/referenceworkentry/10.1007%2F978-0-387-39940-9_688.

[13] *Návrhové vzory* [online]. 01.05.2021. [cit. 2021-05-01]. Dostupné z: http://www.algoritmy.net/article/51224/Navrhove-vzory.

[14] ALEXANDER, C. – ISHIKAWA, S. – SILVERSTEIN, M. *A pattern language*. vol.2 / *[Center for Environmental Structure series]*. Oxford University Press, 1977. ISBN 0195019199.

[15] MICROSERVICES.IO. *Microservices Pattern: A pattern language for microservices* [online]. 16.12.2020. [cit. 2020-12-21]. Dostupné z: https://microservices.io/patterns/.

[16] LINDBERG, F. Circuit breaker pattern — What and why? - Bonnier News Tech - Medium. Dostupné z: https://medium.com/bonniernewstech/circuit-breaker-pattern-what-and-why-a17f8babbec0.

[17] XIANG, K. Patterns for distributed transactions within a microservices architecture. Dostupné z: https://developers.redhat.com/blog/2018/10/01/patterns-for-distributed-transactions-within-a-microservices-architecture/.

[18] Denis Rosa, Developer Advocate, Couchbase. Saga Pattern | Application Transactions Using Microservices – Part I. Dostupné z: https://blog.couchbase.com/saga-pattern-implement-business-transactions-using-microservices-part/.

[19] TEAM, W. Service Discovery in a Microservices Architecture. Dostupné z: https://www.nginx.com/blog/service-discovery-in-a-microservices-architecture/.

[20] STATISTA. *Leading container orchestration tool use 2019*

/ *Statista* [online]. 03.11.2020. [cit. 2020-11-03]. Dostupné z: https://www.statista.com/statistics/588827/worldwide-container-technology-orchestration-tool-use/.

[21] KUBERNETES. *What is Kubernetes?* [online]. 22.10.2020. [cit. 2020-11-04]. Dostupné z: https://kubernetes.io/docs/concepts/overview/what-is-kubernetes/.

[22] INDIA, S. Kubernetes-As-A-Service (KaaS): Scale Seamlessly, Grow Rapidly. Dostupné z: https://medium.com/faun/kubernetes-as-a-service-kaas-scale-seamlessly-grow-rapidly-4219d0f5f666.

[23] RENSKI, B. PaaS vs. KaaS: A Primer. Dostupné z: https://containerjournal.com/topics/container-ecosystems/paas-vs-kaas-a-primer/.

[24] *Azure Kubernetes Service (AKS). Customer Stories | Microsoft Azure* [online]. 10.12.2020. [cit. 2020-12-10]. Dostupné z: https://azure.microsoft.com/cs-cz/services/kubernetes-service/#customer-stories.

[25] *Azure Kubernetes Service (AKS). Solution architectures | Microsoft Azure* [online]. 10.12.2020. [cit. 2020-12-10]. Dostupné z: https://azure.microsoft.com/cs-cz/services/kubernetes-service/#solution-architectures.

[26] *Mikroslužby se službou Azure Kubernetes Service | Microsoft Azure* [online]. 15.11.2020. [cit. 2020-11-15]. Dostupné z: https://azure.microsoft.com/cs-cz/solutions/architecture/microservices-with-aks/.

[27] *Getting started with IBM Cloud Kubernetes Service* [online]. 17.11.2020. [cit. 2020-11-17]. Dostupné z: https://cloud.ibm.com/docs/containers?topic=containers-iks-overview.

[28] *IBM Cloud Kubernetes Service - Overview* [online]. 2020. [cit. 2021-01-04]. Dostupné z: https://www.ibm.com/cz-en/cloud/kubernetes-service.

[29] *Kubernetes Service - Overview* [online]. 2020. [cit. 2020-11-16]. Dostupné z: https://www.ibm.com/cz-en/cloud/container-service.

[30] *Kubernetes Service - Pricing* [online]. 2020. [cit. 2020-11-18]. Dostupné z: https://www.ibm.com/cloud/container-service/pricing.

[31] TRUSTRADIUS. *IBM Cloud Kubernetes Service Reviews & Ratings 2020* [online]. 18.11.2020. [cit. 2020-11-18]. Dostupné z: https://www.trustradius.com/products/ibm-kubernetes-service/reviews?rk=ibmck20180&utm_campaign=tqw&utm_medium=widget&utm_source=www.ibm.com&trtid=70da7813-7f20-46a5-9987-0fae49a8c966.

[32] *Customers / Google Cloud* [online]. 14.11.2020. [cit. 2021-01-04]. Dostupné z: https://cloud.google.com/customers#/products=Containers.

[33] Google Cloud. *Using Google Cloud services from Google Kubernetes Engine* [online]. 16.11.2020. [cit. 2020-12-27]. Dostupné z: https://cloud.google.com/solutions/using-gcp-services-from-gke.

[34] *Using Google Cloud services from Google Kubernetes Engine* [online]. 22.10.2020. [cit. 2020-11-18]. Dostupné z: https://cloud.google.com/solutions/using-gcp-services-from-gke#using_external_services_through_a_nat_gateway.

[35] Google Cloud. *Regions and zones / Compute Engine Documentation / Google Cloud* [online]. 16.11.2020. [cit. 2020-11-18]. Dostupné z: https://cloud.google.com/compute/docs/regions-zones.

[36] Google Cloud. *Types of clusters / Kubernetes Engine Documentation / Google Cloud* [online]. 22.10.2020. [cit. 2020-11-18]. Dostupné z: https://cloud.google.com/kubernetes-engine/docs/concepts/types-of-clusters.

[37] *Pricing / Kubernetes Engine Documentation / Google Cloud* [online]. 10.11.2020. [cit. 2020-11-18]. Dostupné z: https://cloud.google.com/kubernetes-engine/pricing.

[38] TRUSTRADIUS. *Google Kubernetes Engine Reviews & Ratings 2020* [online]. 18.11.2020. [cit. 2020-11-18]. Dostupné z: https://www.trustradius.com/products/google-kubernetes-engine/reviews.

[39] Amazon Web Services, Inc. *Amazon ECS Customers | Container Orchestration Service | Amazon Web Services* [online]. 09.11.2020. [cit. 2020-12-08]. Dostupné z: https://aws.amazon.com/ecs/customers/.

[40] Amazon Web Services, Inc. *Amazon EKS | Managed Kubernetes Service | Amazon Web Services* [online]. 01.12.2020. [cit. 2020-12-08]. Dostupné z: https://aws.amazon.com/eks/customers.

[41] *What is Amazon EKS? - Amazon EKS* [online]. 07.12.2020. [cit. 2020-12-08]. Dostupné z: https://docs.aws.amazon.com/eks/latest/userguide/what-is-eks.html.

[42] *What is Amazon Elastic Container Service? - Amazon Elastic Container Service* [online]. 07.12.2020. [cit. 2020-12-08]. Dostupné z: https://docs.aws.amazon.com/AmazonECS/latest/developerguide/Welcome.html.

[43] OPSANI. *Amazon ECS vs EKS : What's the Difference? | Opsani* [online].

2020. [cit. 2020-12-07]. Dostupné z: https://opsani.com/blog/amazon-ecs-vs-eks-whats-the-difference/.

[44] JANBASKTRAINING. *What is AWS Fargate? Difference Between Amazon EC2 vs. Amazon Fargate* [online]. 2018. [cit. 2020-12-08]. Dostupné z: https://www.janbasktraining.com/blog/what-is-aws-fargate/.

[45] TRUSTRADIUS. *Amazon Elastic Container Service (Amazon ECS) Reviews & Ratings 2021* [online]. 03.01.2021. [cit. 2021-01-03]. Dostupné z: https://www.trustradius.com/products/amazon-elastic-container-service-ecs/reviews.

[46] GITHUB. *openshift/okd* [online]. 10.12.2020. [cit. 2020-12-10]. Dostupné z: https://github.com/openshift/okd.

[47] *Try Red Hat OpenShift Container Platform 4* [online]. 28.11.2020. [cit. 2020-12-10]. Dostupné z: https://www.openshift.com/try.

[48] *Red Hat OpenShift Dedicated* [online]. 07.12.2020. [cit. 2020-12-10]. Dostupné z: https://www.redhat.com/en/resources/openshift-dedicated-datasheet.

[49] PICOZZI, S. – HEPBURN, M. – O'CONNOR, N. *DevOps with OpenShift*. O'Reilly, first edition edition, July 2017. Picozzi, Stefano (Verfasser.) Hepburn, Mike (Verfasser.) O'Connor, Noel (Verfasser.). ISBN 9781491975930.

[50] *Welcome | About | OpenShift Online Pro* [online]. 10.12.2020. [cit. 2020-12-10]. Dostupné z: https://docs.openshift.com/online/pro/welcome/index.html.

[51] *OpenShift Product Pricing - Red Hat OpenShift* [online]. 10.12.2020. [cit. 2020-12-10]. Dostupné z: https://www.openshift.com/products/pricing/.

[52] *Customer Success Stories - Red Hat OpenShift* [online]. 10.12.2020. [cit. 2020-12-10]. Dostupné z: https://www.openshift.com/learn/success-stories/.

[53] KUBERNETES. *Production-Grade Container Orchestration* [online]. 25.12.2020. [cit. 2020-12-25]. Dostupné z: https://kubernetes.io/.

[54] KUBERNETES. *Kubernetes Components* [online]. 28.08.2020. [cit. 2020-12-25]. Dostupné z: https://kubernetes.io/docs/concepts/overview/components/.

[55] *etcd* [online]. 25.12.2020. [cit. 2020-12-25]. Dostupné z: https://etcd.io/.

[56] KUBERNETES. *Service* [online]. 25.11.2020. [cit. 2020-12-26]. Dostupné z: https://kubernetes.io/docs/concepts/services-networking/service/.

[57] KUBERNETES. *Service* [online]. 25.11.2020. [cit. 2020-12-26]. Dostupné z: https://kubernetes.io/docs/concepts/services-networking/service/#ips-and-vips.

[58] BURNS, B. – BEDA, J. – HIGHTOWER, K. *Kubernetes. Up and running : dive into the future of infrastructure.* O'Reilly, second edition edition, 2019. ISBN 978-1-492-04653-0.

[59] KUBERNETES. *Overview of kubectl* [online]. 20.11.2020. [cit. 2020-12-27]. Dostupné z: https://kubernetes.io/docs/reference/kubectl/overview/.

[60] KUBERNETES. *kubectl Usage Conventions* [online]. 08.12.2020. [cit. 2020-12-28]. Dostupné z: https://kubernetes.io/docs/reference/kubectl/conventions/#generators.

[61] KUBERNETES. *Pods* [online]. 28.12.2020. [cit. 2020-12-28]. Dostupné z: https://kubernetes.io/docs/concepts/workloads/pods/.

[62] KUBERNETES. *Managing Resources for Containers* [online]. 17.12.2020. [cit. 2020-12-29]. Dostupné z: https://kubernetes.io/docs/concepts/configuration/manage-resources-containers/#meaning-of-cpu.

[63] VOHRA, D. *Kubernetes microservices with Docker.* Apress, 2016. ISBN 9781484219072.

[64] *Aggregating Container Logs | Configuring Clusters | OpenShift Container Platform 3.11* [online]. 25.04.2021. [cit. 2021-04-25]. Dostupné z: https://docs.openshift.com/container-platform/3.11/install_config/aggregate_logging.html#deploying-the-efk-stack.

[65] Lakshmi Narasimhan. *Openshift vs Kubernetes: which one should you choose?* [online]. 2018. [cit. 2021-04-25]. Dostupné z: https://medium.com/@lakshminp/openshift-vs-kubernetes-which-one-should-you-choose-4c260daf0ad0.

[66] *Authentication - Additional Concepts | Architecture | OKD 3.11* [online]. 26.04.2021. [cit. 2021-04-26]. Dostupné z: https://docs.okd.io/3.11/architecture/additional_concepts/authentication.html#oauth-clients.

[67] *Prerequisites - Installing | Installation and Configuration | OpenShift Enterprise 3.0* [online]. 26.04.2021. [cit. 2021-04-26]. Dostupné z: https://docs.openshift.com/enterprise/3.0/install_config/install/prerequisites.html#prereq-selinux.

[68] CLOUDOWSKI.COM. *10 most important differences between OpenShift and Kubernetes* [online]. 2019. [cit. 2021-04-26]. Dostupné z: https://cloudowski.com/articles/10-differences-between-openshift-and-kubernetes/.

[69] *Projects and Users - Core Concepts | Architecture | OKD 3.11* [online]. 26.04.2021. [cit. 2021-04-26]. Dostupné z: https://docs.okd.io/3.11/architecture/core_concepts/projects_and_users.html.

[70] *Builds and Image Streams - Core Concepts | Architecture | OKD 3.11* [online]. 26.04.2021. [cit. 2021-04-26]. Dostupné z: https://docs.okd.io/3.11/architecture/core_concepts/builds_and_image_streams.html.

[71] *Other API Objects - Additional Concepts | Architecture | OKD 3.11* [online]. 26.04.2021. [cit. 2021-04-26]. Dostupné z: https://docs.okd.io/3.11/architecture/additional_concepts/other_api_objects.html.

[72] *Welcome | About | OKD 3.11* [online]. 26.04.2021. [cit. 2021-04-26]. Dostupné z: https://docs.okd.io/3.11/welcome/index.html.

[73] Portworx Documentation. *Migration* [online]. 30.04.2021. [cit. 2021-05-01]. Dostupné z: https://docs.portworx.com/concepts/migration/.

[74] *Velero Docs - Cluster migration* [online]. 01.05.2021. [cit. 2021-05-01]. Dostupné z: https://velero.io/docs/v1.6/migration-case/.

[75] GITHUB. *kubemove/kubemove* [online]. 01.05.2021. [cit. 2021-05-01]. Dostupné z: https://github.com/kubemove/kubemove.

[76] KUBERNETES. *Namespaces* [online]. 10.10.2020. [cit. 2021-03-15]. Dostupné z: https://kubernetes.io/docs/concepts/overview/working-with-objects/namespaces/.

[77] CHRIS. *Cluster DNS: CoreDNS vs Kube-DNS* [online]. 2018. [cit. 2021-04-23]. Dostupné z: https://coredns.io/2018/11/27/cluster-dns-coredns-vs-kube-dns/.

[78] GITHUB. *prometheus-community/helm-charts* [online]. 27.04.2021. [cit. 2021-04-27]. Dostupné z: https://github.com/prometheus-community/helm-charts/tree/main/charts/prometheus.

[79] GITHUB. *Portshift/kubei* [online]. 28.04.2021. [cit. 2021-04-28]. Dostupné z: https://github.com/Portshift/kubei.

[80] *Quickstart · Cicada* [online]. 26.10.2020. [cit. 2021-04-29]. Dostupné z: https://cicadatesting.github.io/cicada-2/docs/quickstart.

Seznam zkratek

AKS	Azure Kubernetes Service
AWS	Amazon Web Services
CaaS	Container as a Service
CI/CD	Continuos integration, Continuous delivery
CPU	Central processing unit, tady myšleno procesor počítače
DB	Databáze
ECS	Amazon Elastic Container Service
EKS	Amazon Elastic Kubernetes Service
GKE	Google Kubernetes Engine
HW	Hardware
IaaS	Infrastructure as a Service
IKS	IBM Cloud Kubernetes Service
JVM	Java Virtual Machine
K8s	Kubernetes
KaaS	Kubernetes as a Service
OCP	OpenShift Container Platform
OS	Operační systém
PaaS	Platform as a Service
RAM	Random Access Memory, tady myšleno operační paměť počítače

S2I Source-to-Image

SaaS Software as a Service

SW Software

VLAN Virtual Local Area Network

VM Virtual Machine, virtuální stroj

VPC Virtual private network

Slovník pojmů

ACID Vlastnosti relačních databází: atomicita, konzistence, izolovanost, trvalost. Tady se používá jako název pro tranzakci, která zajistí konzistenci dat.

API Application programming interface, rozhraní aplikace.

Cluster (Kubernetes) Neprázdná množina pracovních jednotek.

Control plane (Kubernetes) pracovní jednotka, na které běží software pro orchestraci clusteru.

CVE Databáze veřejně známých chyb zabezpečení a bezpečnostních hrozeb.

Docker Populární open-source platforma pro kontejnerizaci aplikací.

JSON Způsob zápisu, serializace a přenášení dat.

Kontejner Software, který obaluje aplikaci spolu s její závislostmi a knihovnami.

Load balancer Nástroj, který vyvážuje zátěž mezi několika jednotkami aplikace.

Node (Kubernetes) Pracovní jednotka, například virtuální stroj.

Pod (Kubernetes) Množina běžicích kontejnerů v clusteru.

REST Architektonický styl, návrhový vzor rozhraní pro HTTP komunikaci.

Rollout update Způsob aktualizace aplikace, který zajistí její nepřetržitou dostupnost.

SELinux Security-Enhanced Linux – rozšíření jádra systému Linux o MAC (mandatorní řízení přístupu), který slouží ke zvýšení bezpečnosti.

Service (Kubernetes) Způsob vystavení rozhraní aplikace.

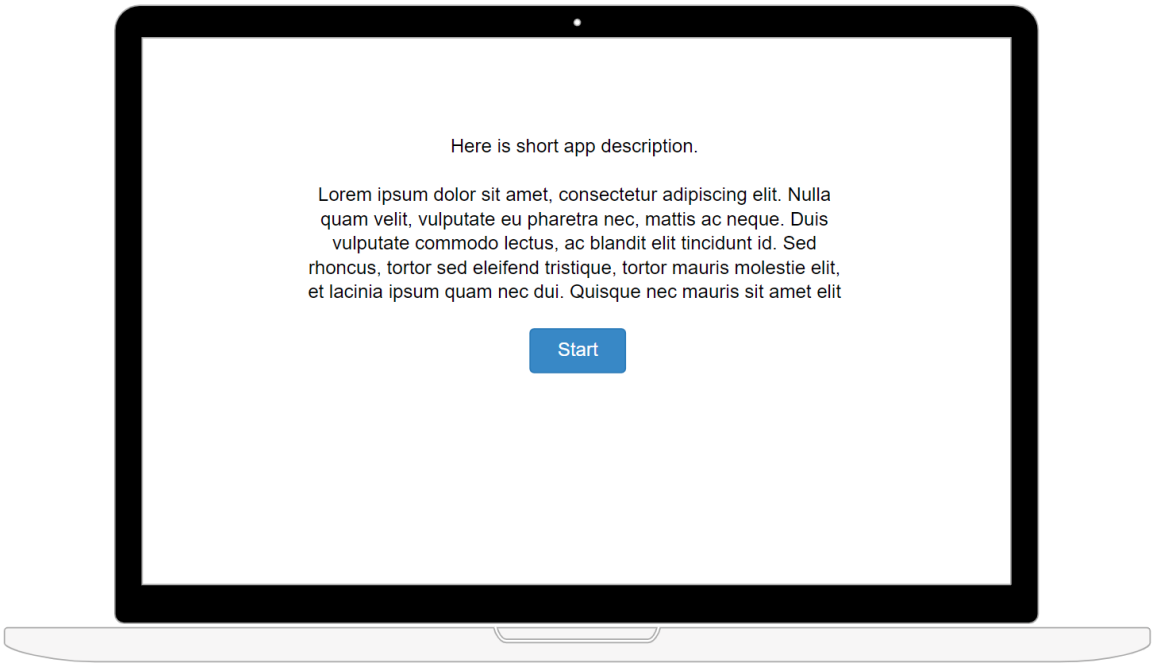
SOAP Simple Object Access Protocol, protokol pro komunikaci, založený na posílání XML zpráv pomocí HTTP.

SOLID Acronym pro pět principů návrhu objektově orientovaného software, účelém kerých je udělat kód srozumitelnějším, flexibilnějším a lépe udržovatelným.

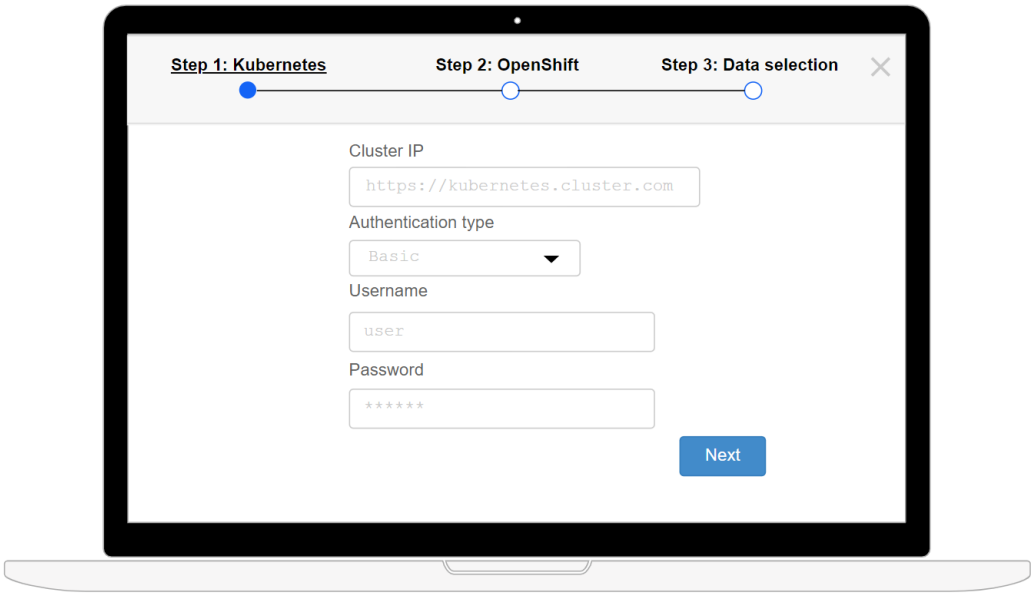
YAML Formát pro serializaci strukturovaných dat.

Příloha A

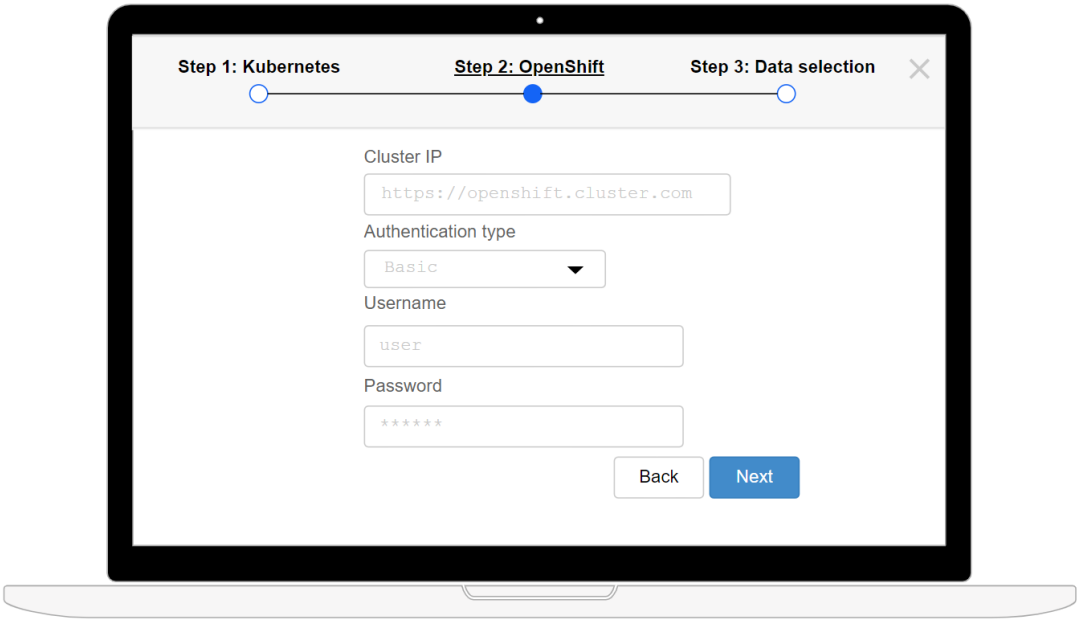
Návrhy obrazovek aplikace pro migraci



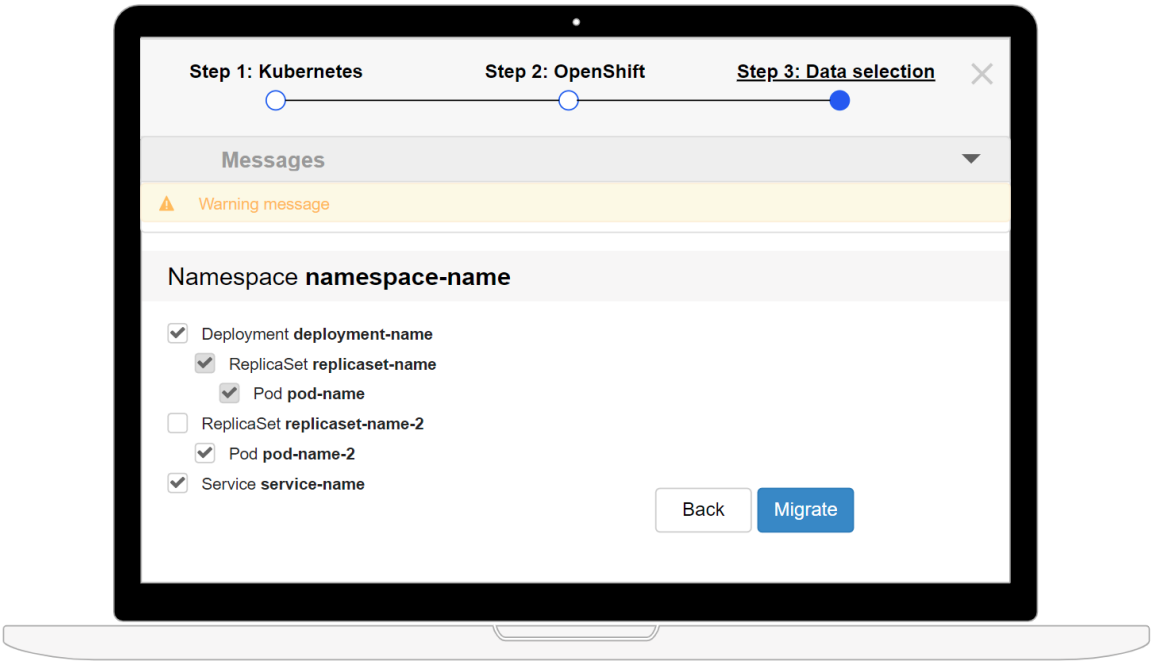
Obrázek A.1: Úvodní stránka



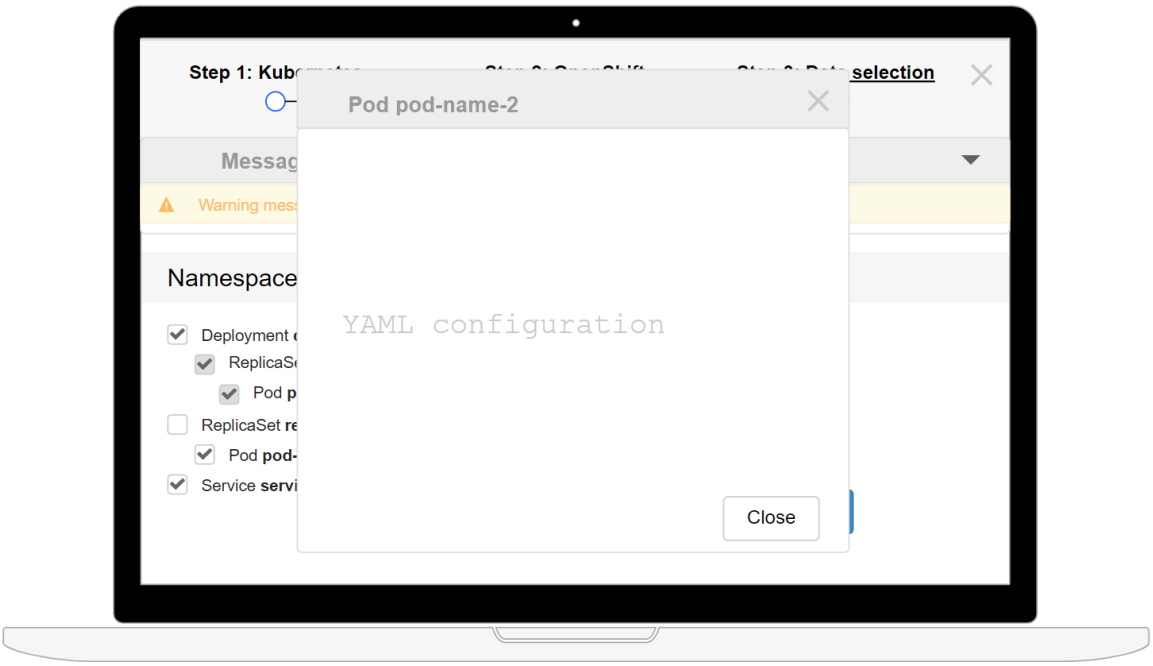
Obrázek A.2: Zadání adresy Kubernetes clusteru



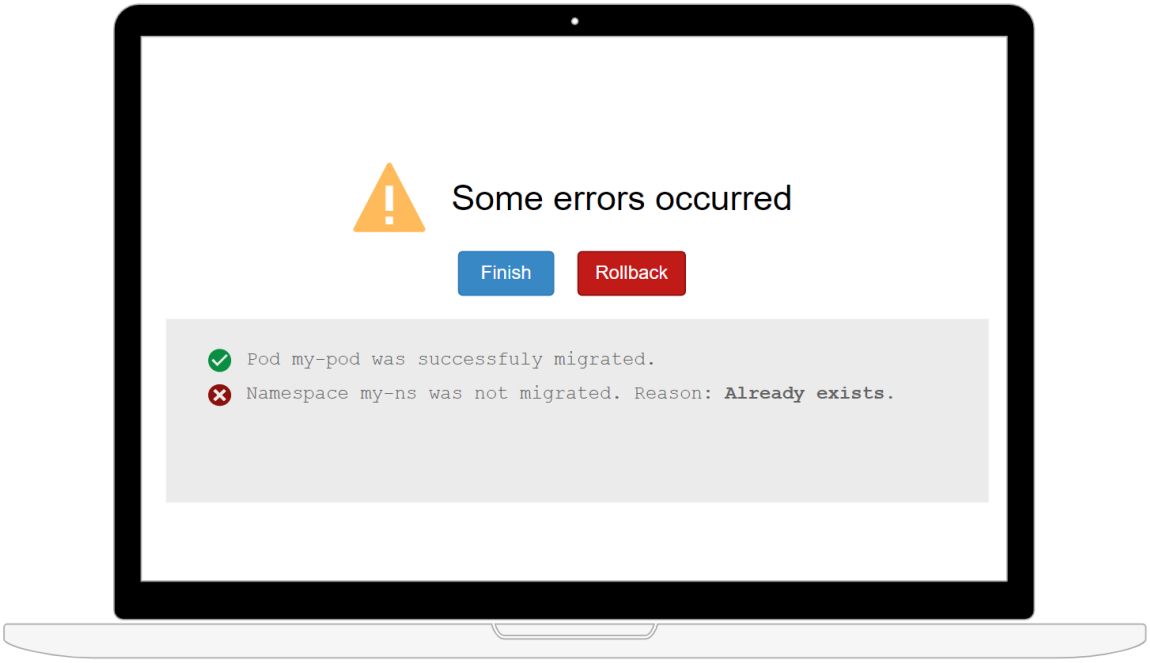
Obrázek A.3: Zadání adresy OpenShift clusteru



Obrázek A.4: Výběr dat k migraci



Obrázek A.5: Zobrazení informace o objektu Kubernetes



Obrázek A.6: Zobrazení výsledků migrace

Příloha B

Iniciální nasazení testovací aplikace

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: random-name-deployment
spec:
  replicas: 2
  selector:
    matchLabels:
      app: random-name
  template:
    metadata:
      name: random-name-pod
      labels:
        app: random-name
    spec:
      containers:
        - name: random-name-container
          image: kulyndar/random:name
          ports:
            - containerPort: 8081
```

Fragment kódu B.1: random-name-deployment.yml

```
apiVersion: v1
kind: Service
metadata:
  name: random-name-service
spec:
  selector:
    app: random-name
  ports:
    - protocol: TCP
      port: 80
      targetPort: 8081
```

Fragment kódu B.2: random-name-service.yml

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: random-number-deployment
spec:
  replicas: 2
  selector:
    matchLabels:
      app: random-number
  template:
    metadata:
      name: random-number-pod
      labels:
        app: random-number
    spec:
      containers:
        - name: random-number-container
          image: kulyndar/random:number
          env:
            - name: randomservice.name.address
              value: http://10.97.233.67:80/random-name
              ports:
                - containerPort: 8082
```

Fragment kódu B.3: random-number-deployment.yml

```
apiVersion: v1
kind: Service
metadata:
  name: random-number-service
spec:
  selector:
    app: random-number
  type: LoadBalancer
  ports:
    - protocol: TCP
      port: 80
      targetPort: 8082
```

Fragment kódu B.4: random-number-service.yml

Příloha C

Cicada – definice testů

```
tests:
  - name: test-random-name
    runner: rest-runner
    asserts:
      - type: StatusCode
        params:
          method: GET
          actionParams:
            url: http://random-name-service.random.svc.cluster
  .local/random-name/name
          expected: 200
  - name: test-random-number
    runner: rest-runner
    asserts:
      - type: StatusCode
        params:
          method: GET
          actionParams:
            url: http://random-number-service.random.svc.
  cluster.local/random-number/number
          expected: 200
  - name: test-random-integration
    runner: rest-runner
    asserts:
      - type: JSON
        params:
          method: GET
          actionParams:
            url: http://random-number-service.random.svc.
  cluster.local/random-number/number
          expected:
            name: Philip
```