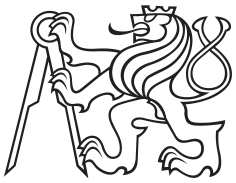
Diplomová práce



České
vysoké
učení technické
v Praze

F3

Fakulta elektrotechnická
Katedra počítačů

Provoz aplikací na platformě OpenShift

Bc. Pavel Krulec

Vedoucí: Ing. Jiří Šebek
Studijní program: Otevřená informatika
Studijní obor: Softwarové inženýrství
Leden 2021

ZADÁNÍ DIPLOMOVÉ PRÁCE

I. OSOBNÍ A STUDIJNÍ ÚDAJE

Příjmení: **Krulec** Jméno: **Pavel** Osobní číslo: **456920**

Fakulta/ústav: **Fakulta elektrotechnická**

Zadávající katedra/ústav: **Katedra počítačů**

Studijní program: **Otevřená informatika**

Specializace: **Softwarové inženýrství**

II. ÚDAJE K DIPLOMOVÉ PRÁCI

Název diplomové práce:

Provoz aplikací na platformě OpenShift

Název diplomové práce anglicky:

Applications operation on the OpenShift platform

Pokyny pro vypracování:

Trendem poslední doby je přesun aplikací do cloudu, často s využitím modelu Infrastructure-as-a-Service [1]. Tato transformace s sebou přináší řadu výhod včetně například snazšího horizontálního škálování či adopci DevOps [2]. Pro dosažení těchto výhod je však potřeba znát specifika vývoje aplikací určených pro provoz v cloudu [2][3]. Tato specifika pokrývají celý životní cyklus aplikace od návrhu jejího designu a architektury, přes implementaci až po konečné provozování na produkčním prostředí. V architektuře aplikací se setkáváme s pojmem tzv. microservices, které běží ve virtualizovaném či kontejnerizovaném prostředí. Vývoj probíhá agilní formou a platňuje se DevOps, tedy prolínání rolí vývojářů a provozu. To vše s vysokou mírou automatizace se CI/CD v hlavní roli. V každé části procesu vývoje se setkáváme s řadou nástrojů, které mají za cíl naši práci zefektivnit jak v v oblasti rychlosti, tak i kvality výsledných artefaktů. K tomu je však potřeba těmto nástrojům porozumět a vědět, jak je vhodně využít. Cílem práce je poskytnutí teoretických základů pro pochopení dané problematiky a především jejich aplikace v praktických příkladech na zvolených technologiích. Práce se bude věnovat tématům :
1/ Virtualizace a kontejnerizace, výhody a nevýhody; 2/ Orchestrace; 3/ Microservices; 4/ Přínosy automatizace k efektivitě práce; 5/ CI-CD; 6/ OpenShift a srovnání s Kubernetes; 7/ Logování a monitoring. Součástí práce je vytvoření ukázkové cloud-native aplikace, na níž budou zmíněné přístupy vhodně demonstrovány. Aplikace bude mít architekturu přizpůsobenou provozu v cloudu na platformě OpenShift, bude automaticky sestavována a testována pomocí CI/CD pipeline a běžící aplikace bude monitorována.

Seznam doporučené literatury:

[1] A. Balalaie, A. Heydarnoori and P. Jamshidi, "Microservices Architecture Enables DevOps: Migration to a Cloud-Native Architecture," in IEEE Software, vol. 33, no. 3, pp. 42-52, May-June 2016.
[2] Ferreira Leite, Leonardo & Aguiar, Carla & Kon, Fabio & Milojicic, Dejan & Meirelles, Paulo. (2019). A Survey of DevOps Concepts and Challenges.
[3] FOWLER, Martin; FOEMMEL, Matthew. Continuous integration. 2006.

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: **16.09.2020** Termín odevzdání diplomové práce: **05.01.2021**

Platnost zadání diplomové práce: **19.02.2022**

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

III. PŘEVZETÍ ZADÁNÍ

Diplomant bere na vědomí, že je povinen 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 studenta

Poděkování

Rád bych na tomto místě poděkoval své rodině za bezmeznou podporu v průběhu celého mého života. Můj dík patří i přátelům za jejich morální podporu v nelehkých časech studia. V neposlední řadě samozřejmě děkuji i svému skvělému vedoucímu panu Ing. Jiřímu Šebkovi za pomoc při vzniku této práce.

Prohlášení

Prohlašuji, že jsem předloženou práci vypracoval samostatně a že jsem uvedl veškeré použité informační zdroje v souladu s Metodickým pokynem o dodržování etických principů při přípravě vysokoškolských závěrečných prací.

V Praze, 1. ledna 2021

Abstrakt

Trendem poslední doby je přesun aplikací do cloudu, často s využitím modelu Platform-as-a-Service. Tato transformace s sebou přináší řadu výhod včetně například snazšího horizontálního škálování či adopci DevOps. Pro dosažení těchto výhod je však potřeba znát specifika vývoje aplikací určených pro provoz v cloudu.

Tato práce se v první části teoreticky i prakticky věnuje základům, na kterých cloud stojí – virtualizaci a kontejnerizaci. Dále čtenáře seznamuje s pojmem *cloud*, jeho rozdělením a navazujícími termíny. Následně představuje platformu *OpenShift* a s ní související technologie pro CI, CD, logování a monitoring. V závěrečné části je s využitím nabytých znalostí postupně vytvořena ukázková aplikace spolu se všemi patřičnými procesy, včetně CI/CD, zpracování logů a monitoringu.

Klíčová slova: openshift, kubernetes, devops, monitoring, microservices, ci, cd, virtualizace, kontejnerizace

Vedoucí: Ing. Jiří Šebek

Abstract

Modern days trend is moving applications into cloud, often using the Platform-as-a-Service model. This transformation comes with a number of advantages including easier horizontal scaling or DevOps adoption. To gain these advantages it is necessary to know the specifics of cloud-native application development.

In the first part, this thesis theoretically and practically discusses the basics on which the cloud stands – virtualization and containerization. It also introduces the reader to the term *cloud*, its division and related terms. Then it introduces the *OpenShift* platform and related technologies for CI, CD, logging and monitoring. In the final part, using the acquired knowledge, a sample application is incrementally created together with all related processes, including CI/CD, log processing and monitoring.

Keywords: openshift, kubernetes, devops, monitoring, microservices, ci, cd, virtualization, containerization

Title translation: Application operations on the OpenShift platform

Obsah

Obrázky

Tabulky

Kapitola 1
Úvod

Problematika vývoje a provozování software je velmi rozsáhlá a dynamická. Současný trend cloudových aplikací umožnil vznik celé řady specializací a profesí – odborník na cloudovou architekturu, specialista na DevOps, administrátor Kubernetes a podobně. Není možné, aby jeden člověk znal detailně všechny oblasti. Má-li však aplikace být udržovatelná a provozovatelná, je nezbytné, aby členové týmu kolem ní měli alespoň přehled o jednotlivých aspektech a vzájemně své znalosti doplňovali.

Tato práce se zabývá technologiemi a procesy, které souvisí s provozem aplikací na platformě OpenShift.

Nejprve se v kapitole 2 zaměříme na pojmy virtualizace, kontejnerizace a s nimi související termíny, tedy samotné základy, na kterých současné technologie stojí.

Jakmile budeme obeznámeni s teorií, podíváme se na tvorbu kontejnerů prakticky pomocí nástroje Podman v kapitole 3.

Když budeme umět vytvářet a spouštět jednotlivé kontejnery, podíváme se o několik úrovní výše, na cloud. Kapitola 4 přiblíží typy cloud computingu podle různých kritérií a pojmy, které s ním souvisí.

Po obecném úvodu do cloud computingu si v kapitole 5 představíme konkrétní *Platform-as-a-Service* technologii – OpenShift. Podíváme se na vztah OpenShiftu se současným trendem Kubernetes. Ukážeme si základní API objekty, které se využijí při tvorbě cloud-native aplikace. V téže kapitole si ukážeme, jak snadno vytvořit OpenShift cluster pro testování na vlastním počítači pomocí *CodeReady Containers*.

Kapitola 6 nás seznámí s technologiemi, které se uplatní při tvorbě a následném provozu aplikací na platformě OpenShift. Jmenovitě se budeme věnovat poměrně novému nástroji Tekton, který OpenShift nabízí (vedle Jenkinse) jako automatizační nástroj pro CI – prozatím jako tzv. Tech Preview. Dále si ukážeme, jak používat značně maturitnější projekt Helm pro CD a následně se seznámíme s nepsaným standardem na poli zpracování logů – Kibanou. Přehled technologií uzavřeme Grafanou, populární aplikací pro monitoring.

Závěrem si získané znalosti předvedeme prakticky při tvorbě vlastní aplikace. V kapitole 7 si ukážeme postup od samotného začátku, přes nasazení na OpenShift a nastavení monitoringu a sběr logů.

Kapitola 2

Virtualizace a kontejnerizace

Virtualizace a kontejnerizace jsou dva možné způsoby, jak spustit naráz více různých aplikací na jednom hardware, přičemž každá z těchto aplikací může existovat ve svém vlastním, odděleném prostředí.

Zatímco virtualizace je o izolaci celého operačního systému, kontejnerizace se soustředí na izolaci jednoho konkrétního procesu. Obě dvě metody mají své výhody i nevýhody a hodí se pro různé způsoby využití. Tabulka 2.1 [1] zobrazuje srovnání vlastností virtuálních strojů a kontejnerů.

Parametr	Virtuální stroje	Kontejnery
Hostovaný OS	Kernel načtený ve vlastní paměti	Všichni hosté sdílí jeden OS a kernel
Komunikace	Skrze Ethernetová rozhraní	Standardní IPC mechanismy (signály, pipes, sockety, ...)
Bezpečnost	Závisí na implementaci hypervisora	Lze vynutit MAC (Mandatory Access Control)
Výkon	Režijní náklady překladu instrukcí mezi hostem a hostitelem	Zanedbatelné režijní náklady
Izolace	Sdílení knihoven, souborů apod. mezi VM není možné	Adresáře je možné namountovat a sdílet mezi kontejnery
Rychlost startu	Řádově desítky vteřin a jednotky minut	Řádově jednotky vteřin
Úložiště	Řádově vyšší využití (nutné uložit kompletní OS)	Řádově nižší využití (základ OS je sdílen)

Tabulka 2.1: Porovnání vlastností VM a kontejnerů

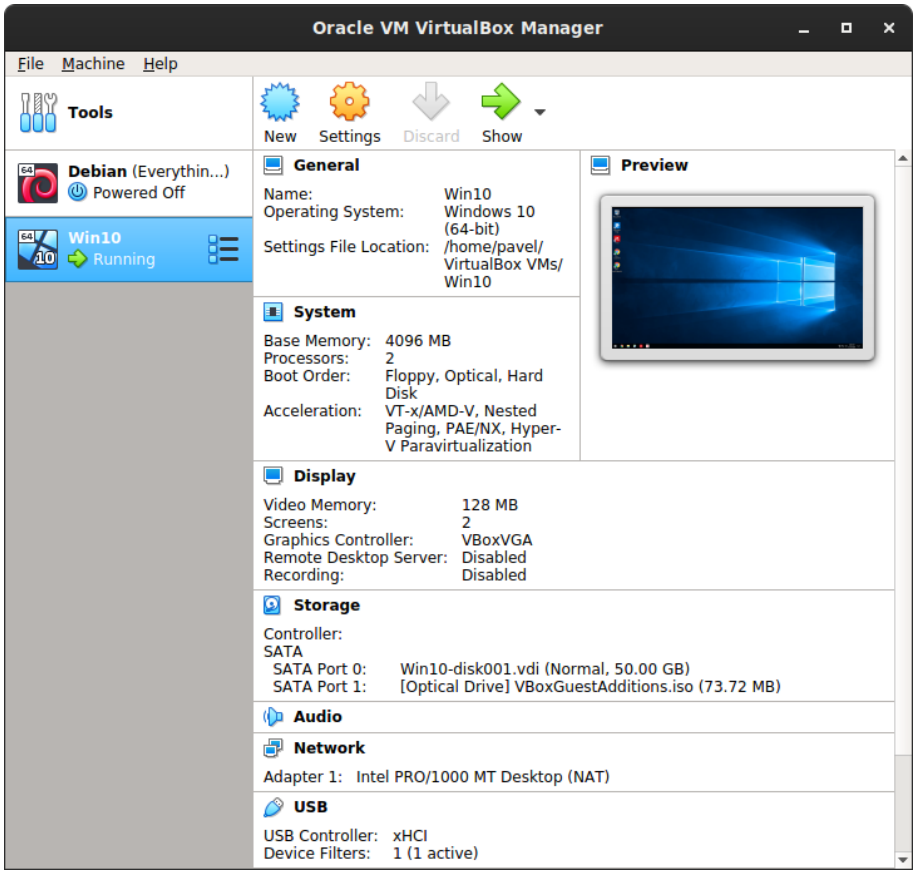
2.1 Virtualizace

Současný hardware je dostatečně výkonný, aby nám umožnil efektivně provozovat více operačních systémů na jednom fyzickém počítači. [2] Toho je dosaženo pomocí metody zvané virtualizace. Virtualizace nám umožňuje vytvářet několik menších počítačů, které sdílejí stejné zdroje, zatímco si myslí,

že mají k těmto zdrojům výhradní přístup.

Mezi výhody, které virtualizace nabízí, patří především zmíněná izolace jednotlivých virtualizovaných strojů. Díky tomu je zvýšena bezpečnost – při infikování jednoho virtuálního stroje je nižší šance na nakažení ostatních. Jednotlivá virtuální prostředí navzájem nekolidují, je tak možné provozovat různé nekompatibilní verze aplikací. Virtuální stroj je možné snadno pozastavit a opět spustit, vytvořit jeho zálohu či dokonce přesunout za běhu na jiného hostitele. Navíc lze díky virtualizaci snáze využít výpočetní kapacitu hardware a neplýtvat tak zbytečně dostupnými zdroji [3, 4].

Nevýhodou virtualizace je tzv. overhead, neboli určité režijní náklady, které během virtualizace vznikají. Ty lze však snížit např. použitím hypervisoru typu I (viz obrázek 2.2a) nebo využitím paravirtualizace.



Obrázek 2.1: VirtualBox se dvěma virtuálními stroji – neběžícím Debianem a spuštěným Windows 10

2.1.1 Historie

Myšlenka virtualizace vznikla v 60. letech 20. století, v době sálových počítačů [5]. Na počátku byla potřeba oddělit uživatelská prostředí a současně spouštět více než jednu aplikaci naráz.

Vzhledem k velmi vysokým cenám a absenci možnosti spouštět více programů současně bylo pro mainframy obtížné rozšířit se do podniků. V 60. letech 20. století přišla společnost IBM s řešením tohoto problému v podobě svého VM mainframe.

S příchodem operačních systémů jako Linux a Windows v 90. letech bylo možné spustit více programů najednou, přestože na jiném principu, než tomu bylo u VM mainframů.

V té době bylo ve společnostech běžnou praxí mít vyhrazený server pro každou aplikaci, aby se zabránilo konfliktům s jinými aplikacemi a pro snazší správu závislostí. Tato praxe však vedla k výraznému plýtvání zdroji, protože docházelo k využití pouze několik desítek procent kapacity serverů.

Podniky hledaly řešení ve formě virtualizace, tak jak ji známe dnes. Situace však byla poměrně obtížná, protože návrháři systémů x86, které v té době převažovaly, příliš nepočítali s podporou virtualizace. V roce 2000 bylo k dispozici několik řešení, která úspěšně dokázala virtualizovat systémy založené na x86.

2.1.2 Princip

Důležitou roli při virtualizaci hraje hypervisor, taktéž označovaný jako *Virtual Machine Monitor* (VMM) [4]. Hypervisor je program, který provádí samotnou virtualizaci – zprostředkovává přístup ke zdrojům, vynucuje bezpečnostní politiku, emuluje hardware a podobně.

Procesor vykonává instrukce ve dvou režimech:

- uživatelském a
- privilegovaném.

V uživatelském režimu běží většina aplikací – ty, které uživatel běžně používá. V privilegovaném režimu se vykonávají instrukce jádra operačního systému.

Při virtualizaci však není možné nechat běžet jádro hostovaného (virtualizovaného) systému běžet v privilegovaném režimu. Pokud by tomu tak bylo, bylo by například možné pomocí privilegovaných instrukcí ovlivňovat ostatní virtuální stroje běžící pod stejným hypervisorem. Je tedy nezbytné spouštět virtualizovaný stroj v uživatelském režimu hostitele. Pokud se virtualizovaný stroj, běžící v uživatelském režimu, pokusí vykonat privilegovanou instrukci, dojde k vyhození výjimky. Tuto výjimku musí obsloužit hypervisor. Této technice se říká trap-and-emulate a je jedním z hlavních příčin overheadu virtualizace.

Pokud to hostovaný systém umožňuje, je možné využít výhod paravirtualizace. V takovém případě virtualizovaný systém ví, že je virtualizován a s hypervisorem přímo spolupracuje.

Hypervisor

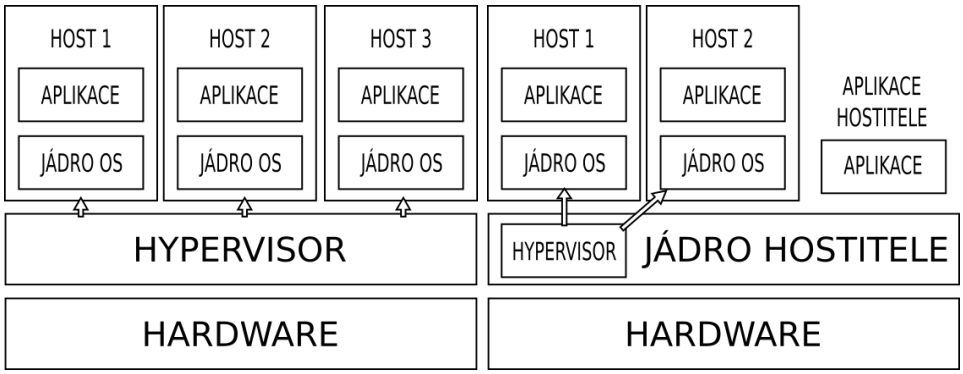
Jak již bylo řečeno, úkolem hypervisoru je vykonávat dohled nad virtualizovaným strojem, zprostředkovávat mu přístup ke zdrojům a další.

Rozlišujeme dva typy hypervisorů:

- Typ I (tzv. nativní, obr. 2.2a) a
- Typ II (tzv. hostovaný, obr. 2.2b)

Zatímco hypervisor typu II běží v rámci jádra hostitelského systému, v případě hypervisoru typu I žádná vrstva hostitelského jádra není – hypervisor komunikuje přímo s hardware.

Mezi nejznámější zástupce hypervisoru I. typu patří VirtualBox společnosti Oracle a KVM, jež je součástí linuxového jádra. Hypervisorem typu II je například vSphere Hypervisor společnosti VMware.



(a) : Hypervisor typu I **(b) :** Hypervisor typu II

Obrázek 2.2: Porovnání hypervisorů I. a II. typu

2.2 Kontejnerizace

Kontejnerizace odstraňuje řadu nevýhod virtualizace, avšak nejedná se o její náhradu. Obě dvě metody mají svá využití, pro která jsou vhodnější než druhá metoda [6].

Kontejnery z tohoto srovnání mohou na první pohled vycházet jako jasní vítězové. Virtuální stroje jsou obecně lepší v izolaci hostovaného prostředí. Zároveň kontejnerizace, na rozdíl od virtualizace, není přímo podporována ve všech rozšířených operačních systémech.

Například jedno z nejznámějších kontejnerizačních řešení *Docker* v operačním systém Windows absenci nativní podpory kontejnerizace obcházelo tak, že spustilo virtuální linuxový stroj v programu VirtualBox [7]. V nejnovějších verzích Windows již VirtualBox není potřeba. Na místo toho Docker dokáže využít integrovaný WSL (Windows Subsystem for Linux) [8].

2.2.1 Princip

Kontejnerizace využívá tří prostředků linuxového jádra k dosažení izolace [1, 9]:

- příkaz `chroot`,
- jmenné prostory *namespace* a
- řídící skupiny *cgroups*.

Příkaz **ch**ange **root** slouží ke spuštění procesu a jeho podprocesů s alternativním kořenovým adresářem („/“). Tímto způsobem je možné docílit izolace souborového systému mezi procesy. Nejedná se však se o bezpečnostní praktiku v pravém slova smyslu – z toho tak zvaného *chroot jail* je totiž možné uniknout.

Jmenné prostory fungují jako pomyslné bariéry mezi procesy v různých ohledech. V linuxovém jádře v současnosti existuje 8 různých jmenných prostorů [10], jejichž přehled je v tabulce 2.2. Pomocí příkazu `chroot` a jmenných prostorů lze docílit požadované izolace.

Namespace	Oblast izolace
Cgroup	Kořenový adresář Cgroup
IPC	System V IPC, fronta zpráv POSIX
Network	Síťová zařízení, porty a podobně
Mount	Mount pointy
PID	ID procesů
User	ID uživatelů a skupin
UTS	Hostname a doménové jméno NIS
Time	Časy (aktuální časová zóna)

Tabulka 2.2: Jmenné prostory v linuxovém jádře

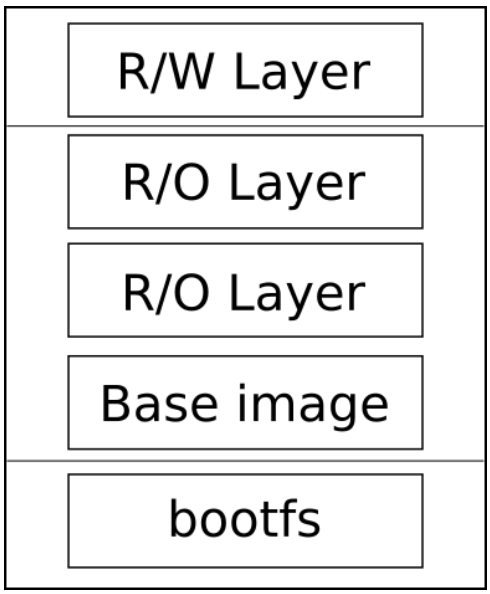
Řídící skupiny, neboli cgroups (**c**ontrol **groups**) dále rozšiřují možnosti kontroly nad procesy. Cgroups jsou zodpovědné za omezování fyzických zdrojů, které má daný proces k dispozici. Díky nim lze zajistit, že dané zdroje plně nevytíží pouze jeden proces, ale lze zajistit férové rozložení. To se hodí nejen při běžném využití, ale zároveň pomáhá předcházet nekontrolovaným následkům případných chyb v běžících procesech.

Obrazy a vrstvy

Řada kontejnerizačních nástrojů pracuje s konceptem takzvaných obrazů (*image*). Obraz je, zjednodušeně řečeno, šablona, podle které jsou následně vytvářeny kontejnery.

Image se skládá z jedné či více vrstev [11] speciálních filesystémů. Tyto vrstvy dohromady připomínají běžný linuxový systém. Nejníže je vrstva s bootovacím filesystémem, tzv. *bootfs*. Na této vrstvě se nachází kořenový filesystém, *rootfs*. Ten obsahuje operační systém samotný.

Při spouštění zde dochází k rozdílu mezi startem kontejneru a plnohodnotného systému. Zatímco v případě startu plnohodnotného systému se po bootu a kontrole integrity kořenový filesystém přepne z režimu read-only do read-write, kontejnery ponechávají *rootfs* v režimu pouze ke čtení.



Obrázek 2.3: Vrstvy běžícího kontejneru

Dalším důležitým konceptem je *union mount* [11]. Tento typ mountu umožňuje připojit několik filesystémů najednou tak, že se tváří jako jeden jediný filesystém. Díky tomu je možné na kořenový filesystém přidávat další read-only vrstvy, které se ve výsledku budou tvářit jako jeden filesystém obsahující soubory a složky ze všech nižších filesystémů.

Při spuštění kontejneru se nad všechny vrstvy přidá ještě jedna vrstva, do které lze zapisovat, viz obrázek 2.3. Pokud v kontejneru změníme nějaký soubor, dojde k jeho zkopírování do nejvyšší, zapisovatelné vrstvy a zde se upraví. Tomuto principu se říká *copy-on-write* [11]. Nezměněný soubor ve své původní vrstvě zůstává. Dokonce smazání souboru z nižší vrstvy nezpůsobí jeho skutečné odstranění, pouze se ve vyšší vrstvě označí jako odstraněný. Odstraněním souboru tak paradoxně velikost obrazu nepatrně vzroste [1].

Z výše uvedeného plyne jeden zásadní poznatek: Na pořadí vytváření vrstev velmi záleží.

2.2.2 Nástroje pro kontejnerizaci

Sada různých nástrojů, která umožňuje vytvářet takzvané kontejnery se nazývá Linux containers (LXC) [12]. Část těchto programů je zmíněna v kapitole 2.2.1. Aby bylo používání kontejnerů dostupné co nejširší uživatelské základně, vznikla řada projektů, které se snaží o zjednodušení a uživatelskou přívětivost.

[1] V Podmanu a experimentální verzi Dockeru lze při sestavování image použít přepínač `-squash`, který sloučí vrstvy do jedné. Odstraněné soubory pak budou skutečně odstraněné a velikost výsledného obrazu bude menší. Použitím tohoto přepínače však přijdeme o cache mezivrstev, což má zásadní negativní vliv na rychlost 2. a dalších sestavení image.

Docker

Nejznámější z těchto projektů je Docker [13]. Docker poskytuje vysokoúrovňové rozhraní pro práce s obrazy a kontejnery.

Obraz, *image*, si lze představit jako read-only šablonu, ze které se vytvářejí kontejnery. Kontejner je jedna běžící instance obrazu. Z jednoho obrazu lze spustit více kontejnerů, je tedy mezi nimi vztah 1:N.

Nevýhodou Dockeru je, že pro běh kontejnerů vyžaduje tzv. Docker démona, tedy systémovou službu běžící na pozadí v privilegovaném režimu. Pokud dojde k chybě a následnému pádu tohoto démona, ukončí se spolu s ním i všechny spuštěné kontejnery. Zároveň nutnost běhu v privilegovaném režimu s sebou nese i jisté bezpečnostní riziko.

Následující příkaz spustí kontejner s názvem *my-fdr* z image *fedora* s tagem *31*.

```
$ sudo docker run --rm -it --name my-fdr fedora:31
  bash
```

V tomto kontejneru spustí `bash` s interaktivním shellem a po jeho ukončení odstraní kontejner.

Podman

Dalším z projektů, které poskytují vysokoúrovňové rozhraní pro práci s kontejnery, je Podman [14]. Podman se snaží zachovat kompatibilitu s Docker API, avšak k běhu kontejnerů nevyžaduje žádného privilegovaného démona.

```
$ podman run --rm -it --name my-fdr fedora:31 bash
```

Jak vidno z příkladů výše, API Docker a Podman jsou téměř identická a je dokonce možné vytvořit alias `alias docker=podman` [14]. Hlavním rozdílem v uvedených příkladech je tak absence `sudo` před příkazem `podman`, což umožňuje běh v uživatelském režimu [2].

Podman dále obohacuje myšlenku kontejnerů o takzvané pody [15]. Pod sdružuje jeden a více běžících kontejnerů do společného izolovaného prostředí. Kontejnery v rámci podu tedy sdílí jmenné prostory a de facto mají společný *localhost*.

Můžeme například vytvořit nový pod *my-pod*. V tomto podu spustíme 2 kontejnery: první bude poslouchat na portu 1234 a druhý se k němu připojí a pošle zprávu `Hello`. Následně se pod smaže.

```
$ podman pod create --name my-pod
$ podman run --rm -it --pod my-pod fedora:31 dnf
  install -yq nc && nc -l 1234
$ podman run --rm -it --pod my-pod fedora:31 dnf
  install -yq nc && echo "Hello" | nc localhost
  1234
$ podman pod rm my-pod
```

[2] Docker též umožňuje volat příkaz `docker` bez `sudo`. Uživatel však musí být členem skupiny *docker*, která má oprávnění srovnatelná s *root*em.

Tento příklad dokazuje, že kontejnery v rámci jednoho podu skutečně sdílí *localhost*.

Detailněji se použití Podmanu věnuje kapitola 3.

2.2.3 CRI, OCI, CRI-O

V oblasti kontejnerizace se setkáváme s řadou termínů a zkratek. Mezi důležité pojmy, na které lze v kontextu Kubernetes narazit, patří *CRI*, *OCI* a *CRI-O*.

CRI – Container Runtime Interface

Existuje velké množství rozličných container enginů, programů, které se starají o běh kontejnerů (tzv. container runtime). Mezi zástupce patří například Docker, CRI-O, LXC, RunC, Podman a další. CRI je standard, který zahrnuje specifikaci kontejnerového runtime (API) pro Kubelet. Jinými slovy CRI definuje rozhraní, kterým Kubernetes umí komunikovat s runtime enginem.

Například runC je referenční implementace CRI. Takzvaně CRI compliant, tj. v souladu s CRI, je i Docker. Oproti tomu Podman není CRI a nelze ho proto použít jako běhové prostředí pro Kubernetes.

OCI – Open Container Initiative

Kromě velkého množství různých kontejner enginů existuje i řada image formátů. Mezi takové formáty patří kupříkladu Docker, AppC, LXD a další. OCI je specifikace formátu obrazů [16], která se snaží o standardizaci. Definuje jak formát image na disku (tzv. blob), tak i jeho metadata. Obrazy, které jsou OCI, lze spouštět v různých container runtime bez dalších úprav, pokud tyto container runtime OCI formát podporují.

Podman, resp. Buildah ve výchozím stavu vytváří OCI image. Je však možné použít přepínač pro vytvoření image ve formátu Docker.

CRI-O

CRI-O je container engine, který je CRI a OCI. To znamená, že jeho API je kompatibilní s Kubeletem a je tedy používán Kubernetes i OpenShiftem. Zároveň umožňuje běh OCI obrazů, avšak neomezuje se jen na ně – je možné spouštět i jiné formáty, například Docker.

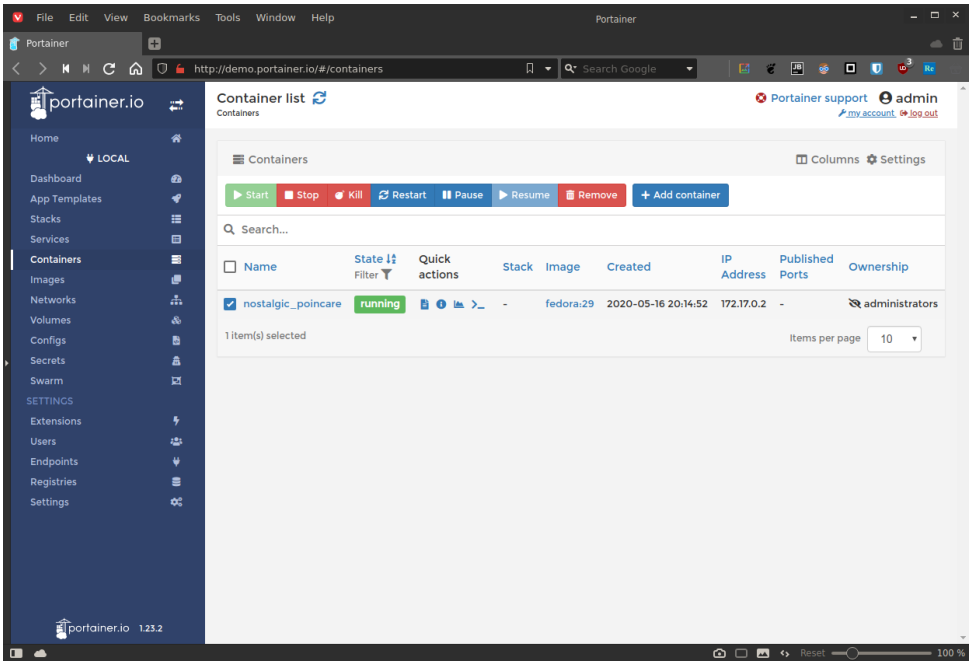
2.3 Správa kontejnerů

S kontejnery potřebujeme pracovat, především je spouštět, restartovat a vypínat, ale i sledovat jejich aktuální stav, zdraví, množství zdrojů, které vytěžují. K tomu lze primárně využít přímo terminálové aplikace typu Podman, kterému se věnuje kapitola 3, či Docker.

Pokud však uživatel pracuje s větším množstvím kontejnerů nebo není příliš technicky zdatný, lze k práci s kontejnery použít grafické uživatelské rozhraní.

2.3.1 GUI pro správu kontejnerů

Mezi uživateli Dockeru je populární aplikace Portainer [17].



Obrázek 2.4: Přehled kontejnerů a obrazů v aplikaci Portainer

Ta poskytuje detailní přehled nad aktuálním stavem kontejnerů a s nimi souvisejících věcí, např. virtuálních sítí, úložišť a podobně. Portainer je velmi propracovaný nástroj, který umožňuje provozovat kontejnerizované aplikace bez hlubších technologií na nižší úrovni. Bohužel však Portainer v současnosti podporuje pouze Docker jako správce kontejnerů.

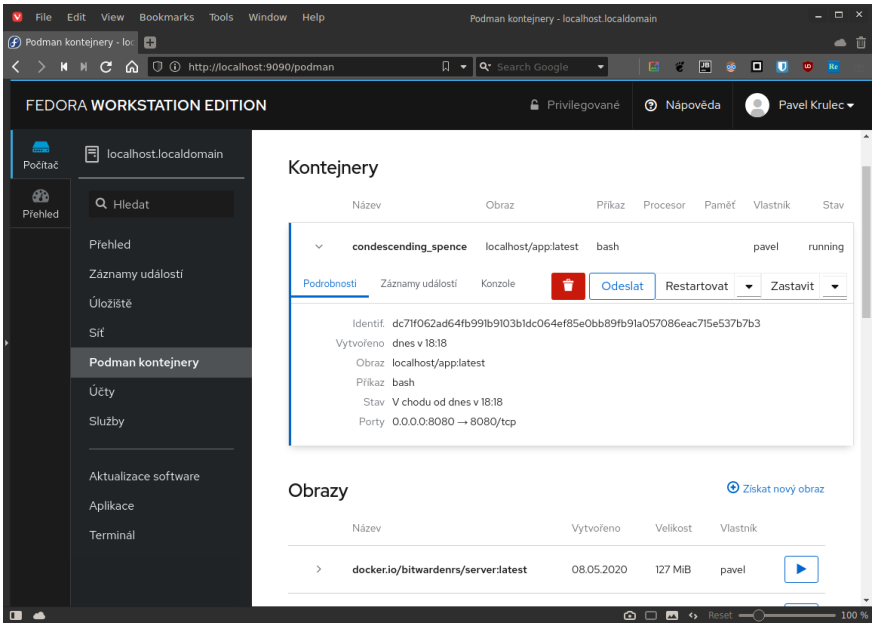
Docker démon běží jako systémová služba pod právy roota. S tímto démonem je možné komunikovat přes API vystavené skrze UNIX socket. Aplikace typu Portainer tento socket využívají pro získávání informací o různých objektech v rámci Dockeru a jejich správě.

Podman však žádného takového démona nemá – běží jako uživatelská aplikace (byť je samozřejmě možné ho spustit i pod rootem). Uživatelé používající Podman byli dlouhou dobu odkázáni pouze na CLI – příkazovou řádku. To se změnilo s příchodem projektu *varlink* [18] Varlink umožňuje vytvořit socket, podobně jako činí Docker, skrze který je možné s Podmanem komunikovat pomocí API.

Díky existenci tohoto API mohla být přidána podpora pro Podman do aplikace Cockpit [19]. Ta se primárně zaměřuje na správu serverů, avšak nyní je možné skrze ní spravovat i obrazy a kontejnery.

2.3.2 Orchestrace

Od jisté chvíle může počet kontejnerů dosáhnout takového množství, že není efektivní, aby se o ně staral člověk manuálně. Je potřeba nasadit nástroje,



Obrázek 2.5: Přehled kontejnerů a obrazů v aplikaci Cockpit

které kontejnery automaticky spravují a škálují podle aktuální zátěže. Takovému procesu se říká orchestrace. Uživatel pak například definuje, že chce vždy mít k dispozici minimálně 2 běžící instance (kontejnery) dané aplikace a maximálně 10. Orchestrátor se pak stará o naplnění těchto podmínek. V případě vysoké zátěže vytvoří nové kontejnery, po opadnutí vytížení některé ukončí. Pokud některý kontejner delší dobu neodpovídá, orchestrátor jej restartuje.

Uživatelé Dockeru mají k dispozici integrovaný orchestrátor Docker Swarm. Velké popularitě uživatelů se těší především projekt Kubernetes [20], který podporuje jako správce kontejnerů jak Docker, tak Podman.

Kubernetes oproti Docker Swarm vyniká především [21]:

- vyšší výkonností a škálovatelností (viz tabulka 2.3),
- integrovaným nástrojem na monitoring a logování,
- širší uživatelskou základnou,
- okamžitou náhradou nefunkčního hosta.

Nevýhodou je znatelně složitější proces instalace a konfigurace.

	Docker Swarm	Kubernetes	OpenShift
Počet hostitelů	1 000	5 000	2 000
Počet kontejnerů	30 000	300 000	-
Počet podů	-	150 000	150 000
Počet podů na hostitele	-	100	500

Tabulka 2.3: Srovnání maximálních testovaných clusterů [22, 23, 24]

Kapitola 3

Podman

Podman [14] je nástroj poskytující vysokoúrovňové API k vytváření a správě kontejnerů. Jeho API je z velké části kompatibilní s nástrojem Docker [13]. Jak popisuje kapitola 2.2.2, Podman zavádí koncept tzv. *pod*ů.

Pod je tvořen jedním či více kontejnery, které jsou společně izolovány. Každý pod po vytvoření obsahuje vždy minimálně tzv. *infra* kontejner [15].

Po vytvoření nového podu můžeme prozkoumat infra kontejner:

```
1 $ podman pod create --name my-pod
2 485fb0a708aa674add98257675834067274f85
3 $ podman pod inspect my-pod | jq .State.
    infraContainerID
4 "3104bed92bd251e7c7b01e20beff3efa963251"
5 $ podman container inspect "3104bed" | jq ".[0].
    ImageName"
6 "k8s.gcr.io/pause:3.1"
7 $ podman pod rm my-pod
8 485fb0a708aa674add98257675834067274f85
```

Zdrojový kód 3.1: Prozkoumání infra kontejneru nově vytvořeného podu

Na řádku 6 vidíme, že infra kontejner je založen na obrazu `k8s.gcr.io/pause:3.1`. Obdobným způsobem může zjišťovat mnoho dalších informací o obrazech, podech i samotných kontejnerech.

3.1 Vytvoření image

Image lze vytvořit dvěma způsoby [25]:

- souborem *Containerfile*
- skriptem pomocí `buildah`

Containerfile je textový soubor obsahující instrukce, podle kterých se vytvoří výsledný image. Formát souboru je totožný s Dockerfile [14, 26], je tedy možné názvy zaměňovat.

Druhou možností je sestavení image skriptem, typicky v bashi, za pomoci programu `buildah` [27]. Tento způsob nabízí větší flexibilitu a umožňuje využít

složitější logiku (např. podmínky a cykly), avšak na úkor vyšší složitosti nižší přehlednosti.

Důležitým rozdílem je také fakt, že při sestavování image z Containerfile (ať už pomocí programu Podman nebo Buildah), se po každé provedené instrukci vrstva uloží do cache. Následující sestavení jsou tak rychlejší. Při skriptovaném sestavení pomocí `buildah` se vrstvy automaticky nevytváří a neukládají do cache – o to se musí postarat autor skriptu.

3.2 Praktický příklad

Představme si tuto situaci: Máme zdrojové kódy staré aplikace napsané v jazyce C++. Jedná se o server, který umí přijmout TCP spojení. Aplikace při spuštění očekává dva parametry: port a protokol, kterým bude s klienty komunikovat. Podporované jsou dva protokoly: Avro a Protobuf.

Součástí zdrojových kódů je pouze definice protokolů Avro a Protobuf (soubory *.avro* a *.proto*) a je nutné z těchto souborů vygenerovat příslušné C++ třídy.

Dále víme, že aplikace má tyto závislosti:

- Avro,
- Protobuf,
- Boost a
- Jsoncpp.

Bohužel aplikace je příliš stará a na našem operačním systému nelze zkompilovat. Máme však informaci, že fungovala na operačním systému Fedora 29. V něm navíc byly všechny závislosti dostupné z repositářů, kromě první zmíněné. Avro bylo potřeba zkompilovat z přiložených zdrojových kódů a nainstalovat.

Struktura CMake projektu je zobrazena ve výpisu 3.2. Soubor *src.zip* obsahuje všechny soubory kromě *avro.zip*.

```
$ tree
.
|-- avro.zip              # Zdrojové kódy Avro
|-- CMakeLists.txt
|-- Containerfile
|-- dataset.cpp
|-- dataset.h
|-- datatype.h
|-- jsonserializable.h
|-- main.cpp
|-- measurementinfo.cpp
|-- measurementinfo.h
|-- measurements.avsc    # Definice pro Avro
```

```
|-- measurements.proto  # Definice pro Protobuf
|-- result.cpp
|-- result.h
+-- src.zip             # ZIP archiv se soubory výše
```

Zdrojový kód 3.2: Struktura staré C++ serverové aplikace

3.2.1 Containerfile

Kód 3.3 zobrazuje Containerfile, pomocí kterého můžeme sestavit aplikaci podle požadavků uvedených výše.

```
1  FROM fedora:29
2  LABEL maintainer="Pavel Krulec <krulepav@fel.cvut.cz
       >"
3  RUN dnf install -y protobuf-devel cmake gcc gcc-c++
       boost-devel make jsoncpp-devel unzip && \
4      dnf clean all
5  WORKDIR /tmp
6  COPY avro.zip avro.zip
7  RUN unzip avro.zip && cd avro && ./build.sh install
8  COPY src.zip src.zip
9  RUN unzip src.zip
10 RUN avrogencpp -o ./ measurements.avsc && \
11     protoc --cpp_out=./ measurements.proto
12 RUN mkdir build && \
13     cd build && \
14     cmake .. && \
15     make && \
16     cd .. && \
17     cp build/app app && \
18     rm -rf *.* avro/
19
20 EXPOSE 8080
21 ENV FORMAT=avro
22
23 CMD ./app 8080 $FORMAT
```

Zdrojový kód 3.3: Containerfile – předpis pro sestavení ukázkové aplikace

První řádek každého *Containerfile* začíná direktivou `FROM`, která je povinná. Ta určuje tzv. base image, neboli obraz, na kterém chceme stavět další vrstvy. V našem případě volíme obraz *fedora* s tagem *29*.

Na **2. řádku** přidáváme k našemu novému image label – informaci o správci obrazu. Každý obraz může mít nula nebo více labelů.

Direktiva `RUN` na **řádku 3** slouží k vykonání příkazu v sestavovaném obraze. Tato instrukce se vykoná během sestavování obrazu. Zde instalujeme potřebné závislosti pro kompilaci aplikace. Na závěr ještě odstraníme nepotřebné soubory pomocí `dnf clean all` (**řádek 4**).

Velmi důležité je provést vyčištění v rámci jednoho příkazu `RUN`. Jak bylo uvedeno v kapitole 2.2.1, vytvořené vrstvy jsou read-only. Pokud bychom příkaz na vyčištění volali v separátní `RUN` direktivě, žádné soubory by se ve skutečnosti nesmazaly. Zůstaly by v předchozí vrstvě a v té nové by se pouze skryly.

Příkaz `WORKDIR` na **řádku 5**, jak již název napovídá, přepíná do kontextu určitého adresáře. Všechny následující příkazy a relativní cesty budou vzhledem k tomu adresáři ve vznikajícím obrazu.

Následně příkazem `COPY` na **6. řádku** nakopírujeme soubor *avro.zip*. Kromě příkazu `COPY` existuje ještě příkaz `ADD`. Ten však není doporučené příliš používat kvůli jeho netransparentnosti (například implicitně rozbaluje TAR archivy).

Na **řádku 7** se rozbalí a nainstaluje Avro.

V tuto chvíli máme nainstalované všechny závislosti a můžeme vygenerovat C++ třídy z definic Avro a Protobuf. Nejprve však musíme do obrazu ZIP archiv se zdrojovými kódy nakopírovat a rozbalit (**řádky 8 a 9**).

Následně se na **řádkách 10–18** vygenerují potřebné soubory a aplikace se zkompiluje. Poslední 3 direktivy `RUN` z řádků 9, 10 a 12 by bylo možné sloučit do 1, abychom vytvořili pouze 1 vrstvu namísto 3. Vysoký počet vrstev má totiž negativní dopad na rychlost.

Jelikož máme serverovou aplikaci která bude komunikovat se svým okolím, musíme komunikaci explicitně povolit. Na **řádku 20** direktivou `EXPOSE` povolujeme komunikaci z a do kontejneru přes port 8080. Portů lze tímto způsobem povolit více.

Předposlední direktiva `ENV` na **řádku 21** nastavuje proměnnou prostředí a její výchozí hodnotu. Proměnné prostředí je možné přepisovat při startu kontejneru, což nám umožní elegantně zvolit protokol, kterým bude aplikace komunikovat.

Na posledním, **23. řádku** direktivou `CMD` specifikujeme, který výchozí příkaz se spustí při startu kontejneru, pokud nebude uvedený jiný.

Sestavení image

Podle takto napsaného Containerfile můžeme vytvořit image příkazem:

```
$ podman build --tag app:latest .
```

Parametrem `--tag` pojmenováváme nový image ve formátu `name:tag`. Pokud se neuvede část jména za dvojtečkou, tzv. *tag*, použije se výchozí tag *latest*. Tečka na konci značí kontext (adresář) ve kterém bude sestavení probíhat.

Nově vytvořený image si můžeme zobrazit:

```
$ podman images
REPOSITORY TAG IMAGE ID CREATED SIZE
localhost/app latest 6a8cc8444e 5 minutes ago 757 MB
```

3.2.2 Buildah skript

Containerfile 3.3 lze také přepsat do následujícího skriptu [28]:

```
#!/usr/bin/env bash
container=$(buildah from fedora:29)

buildah config --label \
    maintainer="Pavel Krulec <krulepav@fel.cvut.cz>"
    \
    $container

buildah run $container \
    dnf install -y protobuf-devel cmake gcc \
    gcc-c++ boost-devel make jsoncpp-devel unzip

buildah run $container dnf clean all
buildah config --workingdir /tmp $container
buildah copy $container avro.zip avro.zip

buildah run $container /bin/sh -c \
    'unzip avro.zip && \
     cd avro && \
     ./build.sh install'

buildah copy $container src.zip src.zip
buildah run $container unzip src.zip
buildah run $container /bin/sh -c \
    'avrogencpp -o ./ measurements.avsc && \
     protoc --cpp_out=./ measurements.proto'

buildah run $container /bin/sh -c \
    'mkdir build && \
     cd build && \
     cmake .. && \
     make && \
     cd .. && \
     cp build / app app && \
     rm - rf *.* avro /'

buildah config --port 8080 $container
buildah config --env FORMAT=avro $container
buildah config --cmd './app 8080 $FORMAT' $container
buildah commit $container app:latest
```

Zdrojový kód 3.4: Bash skript – sestavení ukázkové aplikace pomocí `buildah`

Po skončení skriptu získáme analogický image jako při sestavení pomocí Containerfile. Rozdíl je však v procesu, jakým se image vytváří. V tomto

případě se spustí kontejner z image `fedora:29`. S tímto kontejnerem jsou následně prováděny požadované operace. Nakonec se příkazem na posledním řádku z tohoto běžícího kontejneru vytvoří image.

Pomocí skriptu má programátor vyšší kontrolu nad procesem tvorby image. Skript může být napsaný v různých jazycích a může tak obsahovat podmínky, cykly a pokročilou logiku.

Na první pohled je však znát nižší přehlednost, přestože jak Containerfile 3.3, tak skript 3.4 vytvoří totožný image.

3.2.3 Spuštění kontejneru

Kontejner spustíme následujícím příkazem:

```
1 $ podman run --rm -itp 8080:8080 -e FORMAT=proto app
```

Přepínač `--rm` zajistí, že po ukončení kontejneru dojde k jeho odstranění. Pokud jej neuvedeme, kontejnery budou na disku zabírat místo i po ukončení a je nutné je smazat ručně.

Použitím `-it` získáme po startu kontejneru interaktivní (`-i`) TTY (`-t`).

Parametr `-p` specifikuje, jaké porty chceme přesměrovat. Zde přesměrováváme port 8080 z kontejneru na port 8080 na hostiteli.

Následně přepínačem `-e` nastavujeme proměnnou prostředí `FORMAT`.

Posledním parametrem je název obrazu, který chceme spustit.

Po spuštění kontejneru je server během okamžiku připraven obsluhovat klientská připojení ze svého izolované prostředí a to z počítače, kde původně nebylo možné server ani zkompilovat.

Kapitola 4

Cloud

V minulosti bylo běžné aplikace provozovat na vlastní infrastruktuře, takzvaně *on-premise*. Tato varianta má své výhody – především má společnost data pod svým dohledem. To může být v některých případech dokonce vyžadováno legislativou.

Na druhou stranu vlastnictví hardware s sebou přináší i jisté nevýhody, v prvé řadě nutnost starat se o zařízení, pravidelně jej kontrolovat, servisovat a obměňovat. Podobné úkony samozřejmě stojí nemalé finanční prostředky a ne vždy se vyplatí. Mezi další náklady patří elektrická energie nutná pro provoz této infrastruktury, její chlazení, zajištění záložního zdroje pro případ výpadku a to vše musí být někde umístěno. Dále je obvykle potřeba veškerá data zálohovat a spolu s faktem, že kritická infrastruktura by pro případ závady měla být zdvojená, se náklady na on-premise infrastrukturu mohou vyšplhat do velkých částek.

Výraz *cloud* se v dnešní době používá jako alternativa pro on-premise. V podstatě se jedná o sdílení prostředků poskytovatele skrze internet mezi vícero klienty. Díky tomu se fixní náklady rozpočítají mezi vícero uživatelů, čímž většinou dojde k úspoře nákladů.

Mezi největší přínosy cloudu však patří škálovatelnost. Pokud se aplikace dostane do velké, neočekávané zátěže, velmi snadno lze dodat další zdroje na odbavování požadavků.

Americký Národní institut pro standardy a technologie (NIST) v roce 2011 publikoval definici Cloud Computingu [30]. Podle NIST definice cloud-computing splňuje tyto základní charakteristiky:

- Samoobsluha na vyžádání – Zákazník si může sám nastavit využívané zdroje, aniž by k tomu potřeboval komunikaci s člověkem na straně poskytovatele.
- Široký přístup k síti – Zdroje jsou dostupné skrze internet prostřednictvím standardizovaných mechanismů, díky čemuž je možné využít libovolných klientů pro komunikaci s cloudovou službou.
- Sdružování zdrojů – Výpočetní zdroje poskytovatele jsou sdruženy tak, aby sloužili více zákazníkům naráz podle jejich poptávky. Zákazníci obvykle nemají kontrolu nad přesným umístěním zdrojů (konkrétní server),

Kategorie	Minulost	Současnost
Infrastruktura	Datová centra	Orchestrovaná prostředí
	Bare metal, virtuální stroje	Kontejnery
Síť	IP adresy, DNS	Service discovery
	TCP/IP	REST
Softwarový design	Monolitické aplikace	Mikroslužby
	Hardwarová redundance	Zohlednění možnosti chyb
	On premise	Cloud
Vývojová metodologie	Oddělený vývoj a provoz	DevOps přístup

Tabulka 4.1: Změna fundamentálních úrovní abstrakce ve vybraných kategoriích vývoje software [29]

ale mohou mít možnost specifikovat lokaci na vyšší úrovni, například konkrétní datacentrum.

- Vysoká pružnost – Zdroje lze alokovat a uvolňovat dle libosti, často automaticky, dle aktuální potřeby zákazníka. Z pohledu zákazníka se zdroje zdají být neomezené a lze si je kdykoliv alokovat v libovolném množství.
- Měřené služby – Cloudový systém automaticky řídí a optimalizuje využití zdrojů na takové úrovni abstrakce, která je vhodná pro daný typ služby (např. úložiště, paměť, počet uživatelských účtů a podobně). Díky tomu lze snadno monitorovat využití zdrojů, což poskytuje transparentnost jak pro zákazníka, tak pro poskytovatele služeb.

4.1 Modely cloud-computingu dle úrovně služeb

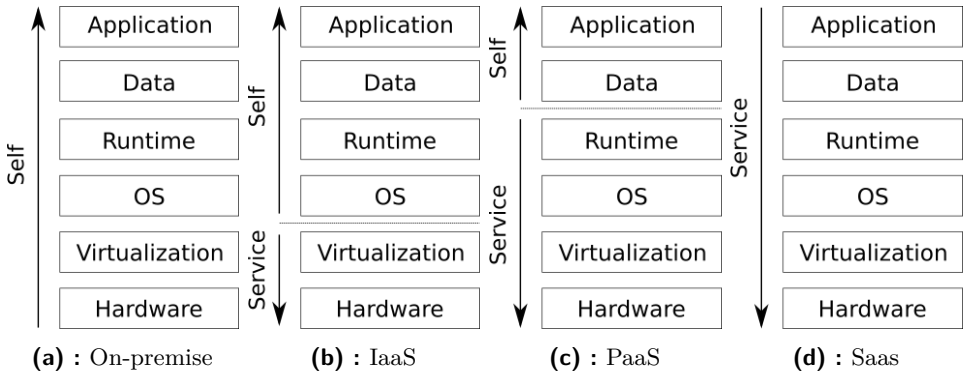
Cloud-computing je služba, kterou poskytovatel dodává svým zákazníkům. Jak je nastíněno na obrázku 4.1, rozlišujeme tři základní typy cloud-computingu podle úrovní, které zajišťuje poskytovatel, a které má ve správě zákazník [31, 32]:

- Infrastructure as a Service
- Platform as a Service
- Software as a Service

4.1.1 Infrastructure as a Service

Infrastructure-as-a-service je nejnižší formou cloud-computingu. Jak zobrazuje obrázek 4.1b, poskytovatel se stará o vrstvy virtualizaci a hardware, to jest na příklad disky, síťové spojení a samotné servery.

Zákazníkovi tato forma nabízí nejvyšší flexibilitu a snadnou škálovatelnost, vykoupeno je to však samozřejmě vyššími nároky na správu. Sám zákazník si volí, jaký operační systém instaluje, jakou platformu a jaké aplikace.



Obrázek 4.1: Porovnání on-premise a různých modelů cloud-computingu

Většina poskytovatelů IaaS také účtuje pouze skutečně využité zdroje. To může být na jednu stranu výhodné, pokud provozovanou aplikaci dobře známe a víme, co od ní očekávat. Na druhou stranu v případě, že se v aplikaci vyskytne nějaká chyba (memory-leak) nebo dojde k neočekávaně vysokému zatížení, může být výsledný účet za provoz takové aplikace velmi vysoký.

Vzhledem k rozličným modelům účtování u různých poskytovatelů je poměrně náročné odhadnout předem, jaké budou náklady na provoz aplikace a jaké poskytovatele zvolit.

Mezi největší světové poskytovatele IaaS patří na příklad společnosti Amazon (Amazon Web Services – AWS), Microsoft (Azure), Google (Google Cloud Platform – GCP), IBM (IBM Cloud), či Alibaba (Alibaba Cloud). Tyto společnosti nabízí cenové kalkulačky, které pomohou zákazníkovi provést prvotní odhad jeho nákladů. Přehled cenových kalkulaček je v tabulce 4.2.

Cloud	Cenová kalkulačka
AWS	`https://calculator.aws`
Azure	`https://azure.microsoft.com/pricing/calculator`
GCP	`https://cloud.google.com/products/calculator`
Alibaba Cloud	`https://alibabacloud.com/pricing-calculator`
IBM Cloud	`https://cloud.ibm.com/estimator/review`

Tabulka 4.2: Přehled největších IaaS poskytovatelů a cenových kalkulaček

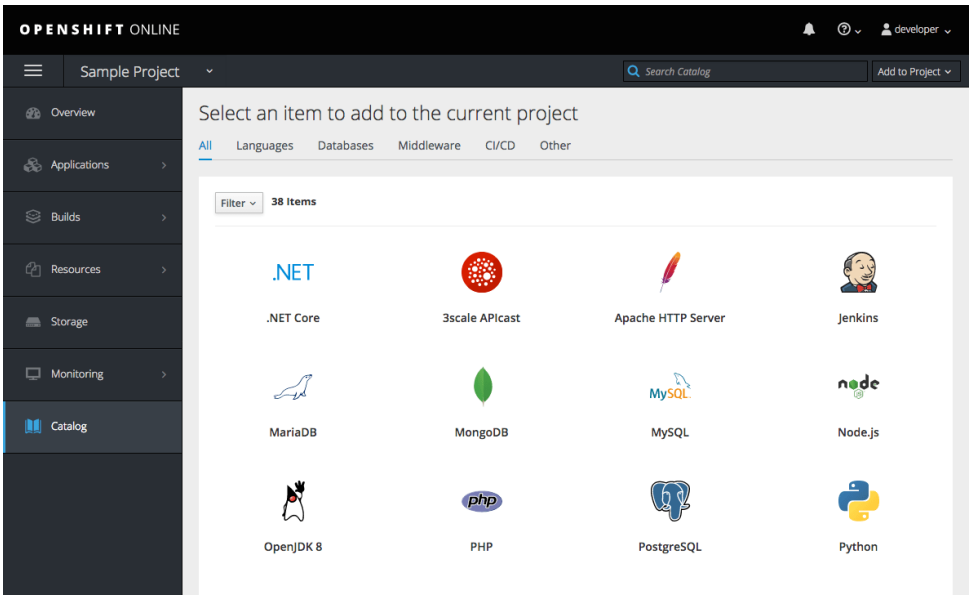
4.1.2 Platform as a Service

O úroveň nad IaaS je PaaS, tedy platforma jako služba. Obrázek 4.1c ukazuje, že při zvolení modelu Platform-as-a-Service se poskytovatel stará nejen o hardware, nýbrž i o základní softwarovou výbavu. Zákazník tedy dostává k dispozici připravené běhové prostředí a stará se pouze o svou aplikaci a jí patřící data.

Platforma, která je ve správě poskytovatele, může nabízet různé nástroje pro vývoj, testování a běh aplikací, komunikaci s middleware službami, správu databází a podobně.

Při zvolení modelu PaaS je potřeba dávat si pozor na tzv. *vendor lock-in*. Jedná se o případ, kdy je zákazník natolik závislý na konkrétních technologiích jednoho dodavatele, že by bylo velmi obtížné, ne-li nemožné, se od nich oprostit. Dodavatel tak svého zákazníka „uzamkne“, což na něj má zpravidla negativní finanční dopad.

Kromě různých speciálních služeb od poskytovatelů zmíněných v kapitole 4.1.2 může být příkladem PaaS i OpenShift od společnosti RedHat.



Obrázek 4.2: Webová konzole OpenShift, jednoho z PaaS produktů

4.1.3 Software as a Service

Na nejvyšší úrovni abstrakce se nachází Software jako služba.

Model SaaS odstiňuje zákazníka od všech nižších vrstev, který se nemusí o nic starat a má k dispozici jen samotnou aplikaci jakožto koncový uživatel (obr. 4.1d).

Software-as-a-Service je nejrozšířenější způsob využití cloud z byznys pohledu [32]. Většina SaaS je dostupná formou webových aplikací, které lze spustit přímo z prohlížeče. Uživateli tak odpadá nutnost cokoliv (kromě webového prohlížeče) instalovat do zařízení koncového uživatele. Zároveň díky tomu jsou poměrně nízké nároky na výkon koncových zařízení – minimální požadavky na spuštění SaaS jsou víceméně dané minimálními požadavky na spuštění webového prohlížeče.

Mezi další výhody webových aplikací provozovaných na modelu Software jako služba patří přenositelnost. Není potřeba řešit různé klienty pro různé operační systémy, rozlišovat mobilní a desktopová zařízení a podobně.

Velkým přínosem je i fakt, že webové aplikace jsou zpravidla bezpečnější díky centrálnímu způsobu aktualizací. Uživatel se nemusí starat o aktualizaci aplikace na svém zařízení, protože webová aplikace se mu vždy načte

v nejnovější verzi.

Model SaaS s sebou však přináší i jisté nevýhody a rizika [32]. Podobně jako v případě PaaS se jedná o tzv. *vendor lock-in*, tedy případ, kdy se zákazník natolik sváže s danou technologií, že přechod na jinou je velmi obtížný až nemožný.

Další problém může být s daty, které jsou de-facto v moci provozovatele a zákazník nad nimi nemá kontrolu. Například v roce 2019 byla odhalena chyba ve službě Google Takeout [33]. Tato služba umožňuje uživatelům produktů společnosti Google stáhnout si všechna data, která o nich uchovává, dle nařízení GDPR. Vyšlo najevo, že pokud si uživatel zažádal o svá videa nahraná v rámci služby Google Photos, v některých případech výsledný archiv obsahoval i videa z cizích účtů. Podle vyjádření společnosti touto chybou bylo zasaženo méně než 1 promile uživatelů, kteří v dané době o export svých dat požádali [33]. Tento případ jistě není ojedinělý, a proto je důležité si uvědomit, že podobné chyby se mohou objevit při využívání sdílené infrastruktury.

Absence zákazníkovy kontroly nad aplikací má za následek obtížnou integraci s dalšími nástroji či možnost přizpůsobení vlastním potřebám – zákazník je odkázán na rozhodnutí poskytovatele SaaS. To se týká i například odstávek za nejen účelem aktualizace, ačkoliv v dnešní době je na vzestupu snaha o bezodstávkové aktualizace.

Software-as-a-Service je obrovské množství, jako příklad lze uvést produkty společnosti Google: webový e-mailový klient GMail, tabulkový procesor SpreadSheet či textový editor Docs. Jedná se o plnohodnotné alternativy k desktopovým aplikacím společnosti Microsoft: Outlook, Excel a Word, které však už v dnešní době mají také svou SaaS podobu.

Na obrázku 4.3 je ukázka tabulkového procesoru Spreadsheet. Jak vidno, webová aplikace je velmi pokročilá, zde například obsahuje dokument tzv. Character sheet pro hru Dungeons & dragons.

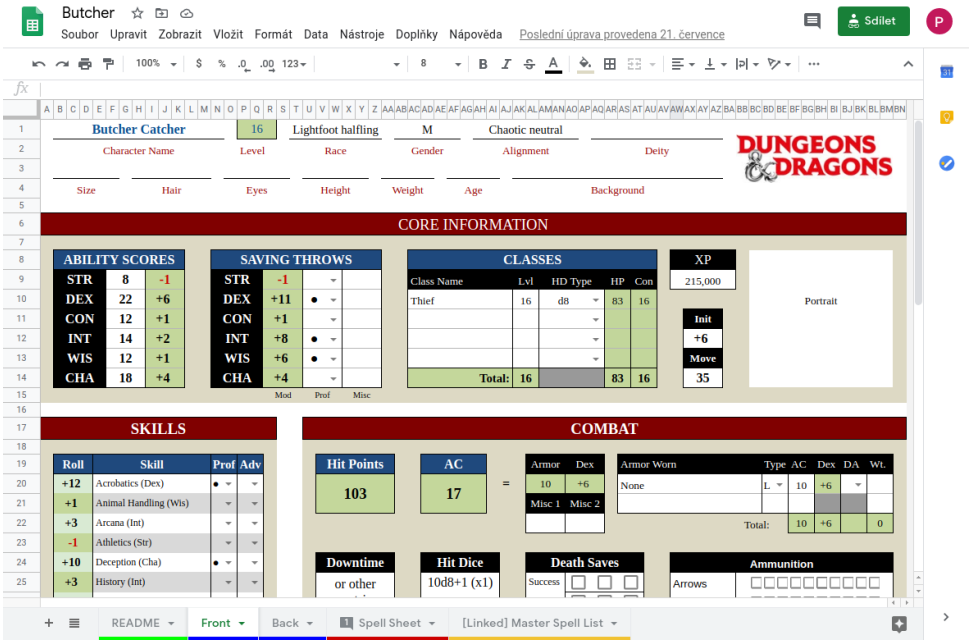
4.2 Modely cloud-computingu dle nasazení

Cloudy lze dále rozdělit podle způsobu nasazení do tří skupin [30, 31]:

- Privátní
- Veřejné
- Hybridní

Privátní cloudy mají infrastrukturu vyhrazenou pro potřeby jedné konkrétní organizace. Tato organizace může reprezentovat více zákazníků, například jednotlivá oddělení. Privátní cloud může provozovat jednak samotná organizace, jednak i nějaká třetí strana a to jak *on-premise*, tedy přímo v prostorách organizace, tak i v prostorech dodavatele (třetí strany).

Veřejný cloud je, jak název napovídá, určený pro sdílení velkým množstvím různých zákazníků, tedy veřejností. Takové typy cloudů provozují entity, například soukromé společnosti, státní organizace či univerzity, na své infrastruktuře.



Obrázek 4.3: Google Spreadsheet – SaaS tabulkový procesor

Hybridním cloudem nazýváme případ, kdy společnost využívá služeb jak privátního, tak veřejného cloudu za předpokladu, že tyto dvě infrastruktury si zachovávají svou samostatnost a jsou vzájemně propojené tak, aby bylo možné data a aplikace mezi nimi přesouvat (např. load-balancing mezi těmito dvěma cloudy).

Mezi výhody privátního i veřejného cloudu patří [31]:

- *Vysoká efektivita* založená na virtualizaci, sdílení zdrojů a rozdělení zátěže.
- *Vysoká dostupnost* vycházející z architektury cloudu a virtualizace.
- *Škálovatelnost* umožňující snadno přidat či odebrat výpočetní kapacitu podle aktuální potřeby.

Výhody veřejného cloudu, kterými privátní cloudy nedisponují, jsou:

- *Nízké vstupní náklady*, jelikož není potřeba pořizovat a konfigurovat hardware.
- *Rozprostření nákladů* díky optimalizaci spotřeby energie, prostoru a administrace.
- *Snazší správa*, jelikož o řadu věcí se stará poskytovatel.

Oproti tomu privátní cloudy mají take své výhody:

- *Vyšší bezpečnost a kontrola nad daty.*
- *Snazší integrace* s dalšími aplikacemi, které společnost využívá.

- *Nižší celkové náklady* z dlouhodobého hlediska, které plynou z výhody využití vlastních zdrojů oproti pronajímání cizích zdrojů.

4.3 Cloud-native aplikace

Cloud Native Computing Foundation (CNCF), člen Linux Foundation, definuje pojem *cloud-native* takto [34]:

> Cloud native technologie umožňují organizacím vytvářet a spouštět škálovatelné aplikace v moderních, dynamických prostředích, jako jsou veřejné, privátní a hybridní cloudy. Příkladem tohoto přístupu jsou kontejnery, tzv. *service mesh*, mikroslužby, neměnná infrastruktura a deklarativní rozhraní API.
>
> Tyto techniky umožňují vytvářet málo provázané systémy, které jsou odolné, snadno spravovatelné a dobře monitorovatelné. V kombinaci s robustní automatizací umožňují provádět zásadní změny často a předvídatelně s minimální námahou.

O kontejnerech pojednává kapitola 2.2.

Service mesh je vrstva infrastruktury, která je zodpovědná za zpracování komunikace mezi jednotlivými službami [29]. Jejím úkolem je spolehlivé doručování požadavků v komplexní topologii cloud-native aplikací. V praxi je service mesh typicky implementována formou tzv. *sidecar*. Jedná se o komponentu, která je „přilepená“ k existující službě tak, že původní služba o ni nemusí vůbec vědět (chová se jako proxy). Velmi důležitou součástí service mesh je funkce *service discovery*, která umožňuje volat jinou lokální službu, aniž bychom museli znát její přesnou adresu.

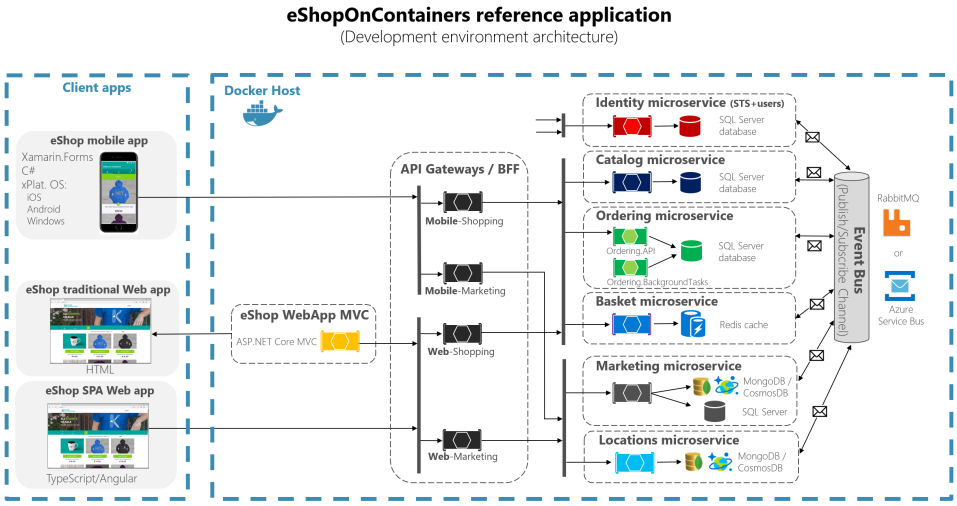
4.4 Mikroslužby

Mikroslužby jsou architektura software, jejíž podstatou je tvorba mnoha jednoduchých služeb, které dohromady tvoří výslednou aplikaci [35]. Na rozdíl od tradiční monolitické architektury mikroslužby jsou typicky snadno škálovatelné, lépe spravovatelné a podporující přístup *continuous integration / continuous delivery*.

Jednotlivé mikroslužby jsou nasazené nezávisle na sobě. Mohou být dokonce postavené na různých platformách a vytvořené v různých technologiích. Dílčí části spolu typicky komunikují skrze REST API ve formátu JSON. Díky tomu, že je každá mikroslužba poměrně malá, je typicky snadné pro nové vývojáře pochopit ji a zapojit se do procesu jejího vývoje. Zároveň je možné provádět průběžné a pravidelné sestavování a nasazování jednotlivých služeb bez ohledu na ostatní.

Další výhodou této architektury je možnost rozdělení práce na aplikaci mezi vícero týmů [36], které mohou být menší co do počtu členů.

Výhodou i nevýhodou mikroslužeb je jejich distribuovanost [36]. Na jednu stranu umožňuje snadnou škálovatelnost, na druhou stranu s sebou přináší



Obrázek 4.4: Ukázka mikroservisní architektury [36]

vyšší složitost z pohledu implementace. Komunikace mezi jednotlivými moduly samozřejmě zvyšuje celkovou latenci systému a může vést i k nedostupnosti některých služeb, se kterou je nezbytné počítat.

4.4.1 DevOps

DevOps (spojení slov Development a Operations) jsou postupy a doporučení, jejichž dodržování vede k minimalizaci času mezi změnou v systému a nasazením této změny na produkci [35]. DevOps je o propojení a spolupráci rolí vývojářů a provozních administrátorů, vzájemné komunikaci a úzké spolupráci. Cílem je zvýšit důvěru v aplikaci jednak z pohledu zákazníků při jejím používání, a jednak z pohledu jejích autorů, kteří ji vyvíjí a provozují.

Zásadní součástí DevOps je také vysoká míra automatizace. Často se tak setkáváme s pojmy CI/CD – Continuous Integration a Continuous Delivery.

Continuous Integration

CI je o pravidelné integraci práce na aplikaci [37]. Tento proces zahrnuje časté sestavování aplikace i několikrát denně. Každé jednotlivé sestavení je testováno na různých úrovních (unit testy, integrační testy, end-to-end testy), aby případné chyby byly odhaleny co nejdříve.

Aby bylo možné celý proces provádět i několikrát denně, je nezbytné, aby vše bylo automatizované. K tomu slouží řada nástrojů, mezi něž patří na příklad Jenkins CI nebo TeamCity. Často také bývá nějaká forma nástroje pro podporu CI součástí webových aplikací pro správu VCS repositářů, ku příkladu GitLab CI, GitHub Actions a podobně.

V roce 2016 provedli J. Bernardo, D. da Costa a U. Kulesza empirickou studii na GitHubu, ve které měřili vliv adopce CI nástroje na rychlost doručení nových pull-requestů u více než 80 vybraných projektů na GitHub.com [38]. Ze závěrů této studie vychází, že více než 70% sledovaných projektů

mergovalo pull requesty po adopci CI až dvakrát rychleji. Zároveň po zavedení automatizace se zvýšil počet vytvořených PR více než dvojnásobně a více než trojnásobně se navýšil počet PR dodaných v 1 release. Automatizace má tedy zásadní vliv na rychlost doručování nové funkcionality zákazníkům.

Jazyk	Projektů	PR před CI	PR po CI
JavaScript	33	39,548	17,556
Python	23	45,896	9,107
Java	11	4,267	3,433
Ruby	10	19,667	3,197
PHP	10	14,165	5,817
Celkem	87	123,543	39,110

Tabulka 4.3: Vliv adopce CI na počet Pull requestů [38]

Continuous Delivery

CD navazuje na CI a je to proces, kterým tým zajišťuje pravidelnou a rychlou dodávku software do produkce [39]. Nenasazuje se však jen na produkční prostředí, nýbrž i na všechna předchozí: vývojové, testovací, integrační, předprodukční a další. Proces nasazení musí být pro tým rutinní, aby se eliminovalo riziko chyby při nasazení na produkci.

Podobně jako CI i CD přispívá ke zlepšení stability produktu, zrychlení v dodávání nové funkcionality a celkové kvalitě.

4.5 Metodika 12 faktorů

Metodika 12 faktorů [40] je dokument popisující 12 různých faktorů, kterých je vhodné se držet při vývoji SaaS aplikací. Cílem těchto *best-practises* je vést vývojáře k tvorbě dobře spravovatelných aplikací, které:

- používají deklarativní formáty pro automatizaci s cílem minimalizovat čas a náklady při začleňování nových přispěvatelů;
- jsou snadno přenositelné mezi různými běhovými prostředími;
- jsou vhodné pro provoz na moderních cloudových platformách;
- minimalizují rozdíl mezi vývojovou a produkční verzí a
- jsou dobře škálovatelné bez nutnosti zásadních změn v architektuře, použitých nástrojích nebo vývojářských praktikách.

Autorem dokumentu je Adam Wiggins (a přispěvatelé), spoluzakladatel společnosti Heroku. Tato společnost je jedním z prvních poskytovatelů PaaS, založena byla v roce 2007 [41], díky čemuž nabízí mnohaleté zkušenosti z praxe.

Metodika definuje těchto 12 faktorů [40]:

1. **Zdrojový kód** – Zdrojový kód je pohromadě ve verzovacím systému, vznikne z něj více deploymentů
2. **Závislosti** – Explicitně deklarované a izolované závislosti
3. **Konfigurace** – Konfigurace uložená v prostředí, kde aplikace běží
4. **Podpůrné služby** – Podpůrné služby jako připojené zdroje
5. **Sestavení, vydání a spuštění** – Striktně oddělené fáze sestavení a spuštění
6. **Procesy** – Aplikace spuštěná jako jeden či více bezestavových procesů
7. **Port binding** – Služby exportované z kontejnerů skrze port binding
8. **Souběžnost** – Škálování do šířky formou tzv. proces modelu inspirovaného Unixem [42]
9. **Zahoditelnost** – Maximalizace robustnosti rychlým startem a korektním ukončením
10. **Rovnost vývojové a produkční verze** – Vývojová a produkční verze by si měly být co nejvíce podobné díky continuous delivery (CD)
11. **Logy** – Logy považované za proud událostí v čase
12. **Administrativní procesy** – Administrativní úkony spouštěné jako jednorázové procesy

Detailní znění jednotlivých doporučení je k dipozici přímo na `https://12factor.net`.

Kapitola 5
OpenShift

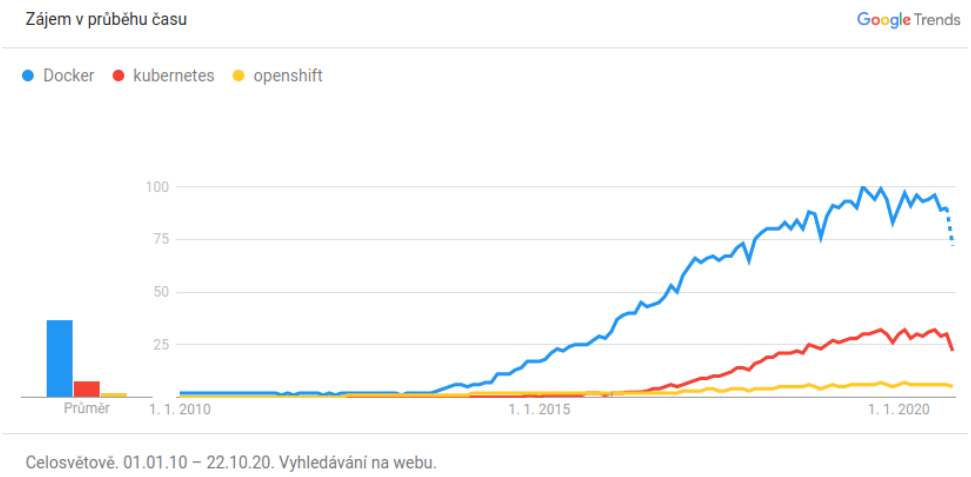
OpenShift je nadstavba nad Kubernetes od společnosti Red Hat. Jedná se o opensource produkt, který je zaměřený na využití v korporátním prostředí. Nic však nebrání použití kdekoliv jinde, kde je potřebný nástroj pro orchestraci kontejnerizovaných aplikací.

OpenShift zachovává kompatibilitu API s Kubernetes. Utilita `oc` používá příkazy stejné jako `kubectl` a přidává další vylepšení navíc, aby práce byla pro uživatele komfortnější a rychlejší.

Zde budeme pracovat s aktuálně nejnovější verzí OpenShiftu 4.6.

5.1 Historie

Historie OpenShiftu začíná v roce 2011 [43, 44], tedy ještě několik let před vznikem Kubernetes. V té době OpenShift závisel na technologii Linux containers. Verze 1 a 2 využívaly vlastní proprietární řešení pro orchestraci kontejnerů. V roce 2013 se Red Hat rozhodl použít pro běhové prostředí kontejnerů Docker, díky čemž se jeho plná podpora dostala na Red Hat distribuce Fedora, CentOS a RHEL.



Obrázek 5.1: Porovnání relativní četnosti vyhledávání pojmů „docker“, „kubernetes“ a „openshift“

Docker se tak stal základem pro OpenShift verze 3, která vyšla v roce 2015. Spolu se změnou container engine na Docker přišla i další zásadní vylepšení verze 3 – Kubernetes jakožto nástroj pro orchestraci kontejnerů. OpenShift verze 3 byl do jisté míry převratný, jelikož adoptoval právě tyto technologie v poměrně brzké fázi jejich vývoje (Red Hat je po Google druhým největším přispěvatelem do Kubernetes projektu [43]). Oproti „holému“ Kubernetes Openshift v3 nabízí například uživatelsky přívětivé webové rozhraní, které umožňuje správu aplikací i méně technickým uživatelům, kteří díky tomu nemusí být odkázáni na příkazovou řádku.

Aktuální verze OpenShift 4 byla vypuštěna v roce 2019 a přináší další velké změny. Zásadní novinkou je přechod z Dockeru na CRI-O a s tím související podpora standardizovaných OCI obrazů. Druhým zásadním vylepšením je podpora tzv. Operators, což jsou komponenty, které se starají o zabalení, nasazení a správu aplikací.

5.2 OpenShift vs Kubernetes

Ne vždy je zcela jasné, jaký je vlastně rozdíl mezi OpenShiftem a Kubernetes. Konkurují si? Je OpenShift jen placené Kubernetes?

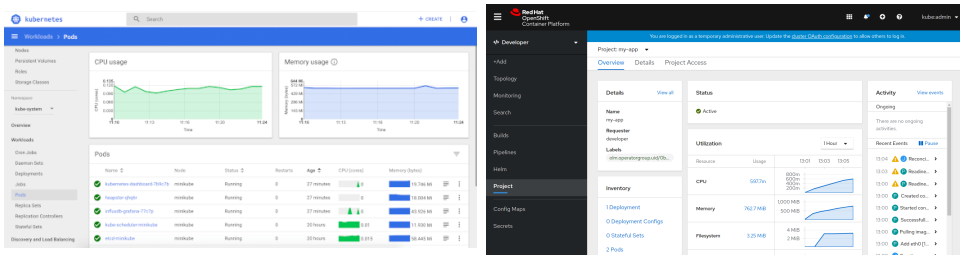
Mark Qu ve své prezentaci shrnul fundamentální rozdíly mezi OpenShiftem a Kubernetes [45] do přehledné tabulky:

Oblast	OpenShift	Kubernetes
Typ	Produkt s placenou podporou	Open-source projekt
Podporované OS	Fedora, RHEL, CentOS	Libovolný Linux
Instalace	Speciální Operátor	Libovolný nástroj
Bezpečnost	Velmi striktní, RBAC	Poměrně volná
Poskytování služeb	Operátory, šablony	Helm
Způsob nasazování	DeploymentConfig, Deployment	Objekt Deployment
Routování	Router	Ingress
Správa obrazů	ImageStream	-
Integrované CI/CD	Jenkins, Tekton, S2I	-
Separace workspace	Projekty (vylepšená namespace)	Namespace
CLI utilita	oc (vylepšená kubectl)	kubectl
Webové rozhraní	Pokročilá konzole, podpora SSO	Základní dashboard
Síťování	Nativní SDN	Ne vše je nativní

Tabulka 5.1: Rozdíly mezi OpenShiftem a Kubernetes [45]

OpenShift je open-source *produkt*, který staví na Kubernetes *projektu*. Je poskytován zdarma pro kohokoliv, placená je podpora od společnosti RedHat.

Velkou předností OpenShiftu z pohledu uživatele je propracované webové rozhraní, které umožňuje si téměř vše „naklikat“ bez nutnosti používat CLI. Oproti tomu Kubernetes nabízí pouze základní dashboard, v němž je navíc většina informací pouze ke čtení a nelze je přímo upravovat.

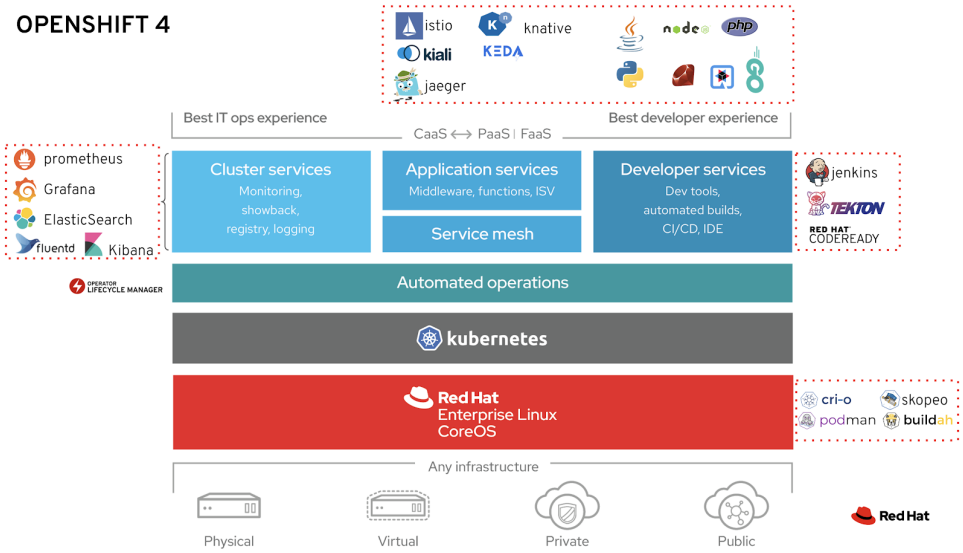


(a) : K8s dashboard **(b)** : OCPv4 Dashboard

Obrázek 5.2: Porovnání webových konsolí K8s a OCP

Z pohledu bezpečnosti OpenShift přidává další vylepšení v podobě Role-Based Access Control. Uživatelé mají určité role, na základě kterých mohou provádět určité akce nad konkrétními objekty. Dalším podstatným bezpečnostním prvkem je zakázání spouštění kontejnerů s uživatelem `root` ve výchozím stavu. Může se tak stát, že řada obrazů např. z DockerHubu nebude v OpenShiftu fungovat bez dodatečných úprav.

Velkou nevýhodou je velmi omezená množina operačních systémů, které OpenShift podporuje – jedná se pouze o ty z dílen Red Hat, čili Fedora, RHEL a CentOS. Oproti tomu Kubernetes je možné provozovat na libovolné linuxové distribuci.



Obrázek 5.3: Architektura OCPv4 [46]

OpenShift nativně podporuje Jenkins a Tekton pro CI/CD, což zrychluje proces on-boardingu a následného vývoje. Kromě toho je součástí OpenShiftu i sada nástrojů pro zpracování logů, tzv. *EFK* (ElasticSearch, Fluentd a Kibana) [47], a monitoring (Promethues a Grafana). Pro autentizaci a autorizaci do těchto nástrojů navíc OpenShift používá vlastní SSO. Vysokoúrovňový pohled na architekturu OpenShiftu zachycuje obrázek 5.3.

Zatímco pro zpracování příchozích požadavků do clusteru a jejich load

balancing OpenShift používá komponentu Router (což je ve skutečnosti program HAProxy), Kubernetes využívá tzv. *Ingress*, což je v podstatě pouze rozhraní, jež může implementovat různý software (např. Ngnix, Apache HTTPD, ale i zmíněná HAProxy). Od verze 3.10 OpenShift však podporuje Kubernetí objekty typu Ingress a překládá, resp. implementuje je pomocí Routeru [47].

OpenShift, na rozdíl od Kubernetes, obsahuje interní image registry. Pro práci s obrazy používá objekty typu ImageStream, které vytváří abstrakci a nabízí další přidanou hodnotu, ku příkladu v podobě automatizovaného nasazení nové verze při aktualizaci obrazu.

Důležitou součástí OpenShiftu jsou i tzv. Operátoři, kteří koncepčně vychází z Kubernetes Controllerů. Technicky se jedná o softwarovou smyčku, která neustále kontroluje stav spravovaných objektů a zajišťuje, že se objekty nachází v požadovaném stavu. V současnosti je na webu `https://operatorhub.io` k dispozici přes 160 Operátorů a je možné vyvinou vlastní pomocí Operator Framework SDK.

5.3 OpenShift API objekty

OpenShift definuje celou řadu API objektů, které definují různé komponenty [48]. Mezi ty nejdůležitější z pohledu vývojáře patří *Pod*, *DeploymentConfig*, *ImageStream*, *BuildConfig*, *ConfigMap*, *Secret*, *Service* a *Route*. API objekty, respektive jejich manifesty, jsou typicky textové soubory ve formátu YAML, které lze aplikovat v clusteru pomocí příkaz `oc apply -f manifest.yml`.

Objekty mohou obsahovat anotace a labely. Zatímco anotace slouží k uložení metadat (např. verze aplikace) a v některých případech k dodatečné konfiguraci objektů, pomocí labelů lze objekty filtrovat a sdružovat.

5.3.1 Pod

Pod je nejmenší jednotka, která je nasaditelná v rámci OpenShiftu [48]. Sestává se z jednoho či více kontejnerů, které jsou společně izolované – sdílí spolu linuxové namespace, CGroups a filesystem.

Typicky pod obsahuje pouze jeden kontejner. Více kontejnerů by pod měl obsahovat pouze tehdy, pokud k sobě neoddělitelně patří. Příkladem může být pod, který obsahuje 1 kontejner s aplikací a 1 proxy kontejner, který zajišťuje SSO přihlášení do aplikace (jako to používá Jenkins na OpenShiftu). V takové případě jsou kontejnery tak úzce provázané, že je správné je zabalit do 1 společného podu.

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: sample-pod
  namespace: sample-namespace
spec:
  containers:
  - image: sample-namespace/sample:latest
    name: sample
    env:
    - name: MY_CONFIG
      valueFrom:
        configMapKeyRef:
          name: my-config
          key: my-key
    volumeMounts:
    - mountPath: /var/run/secrets/my-secret
      name: my-secret-volume
      readOnly: true
  volumes:
  - name: my-secret-volume
    secret:
      defaultMode: 420
```

```
24          secretName: my-secret
```

Zdrojový kód 5.1: Manifest API objektu Pod

Kód 5.1 zobrazuje možnou definici podu. Na řádku 3 jsou definována metadata – název objektu (povinný) a namespace (nepovinný, pokud je neuveden, použije se aktuální namespace). Metadata mohou dále obsahovat již zmíněné anotace a labely, zde však žádné nejsou.

Nejpodstatnější je definice kontejnerů od řádku 7. Pod obsahuje jeden kontejner s názvem *sample*, který bude vytvořený z image *sample-namespace/sample:latest*. Kontejner bude mít k dispozici 1 dodatečnou proměnnou prostředí *MY_CONFIG*, jejíž hodnota bude získána z ConfigMapy *my-config* pod klíčem *my-key*. Proměnných prostředí je možné definovat libovolný počet a jejich hodnota může být buď vložená staticky manuálně, nebo může být načtena dynamicky z objektů ConfigMap a Secret.

Dále řádek 20 vyjmenovává *volumes*, které budou podu dostupné. V této ukázce se vytváří jeden *volume* a to z objektu *secret*. Stejně jako v případě proměnných prostředí kromě *Secrets* lze hodnoty načítat i z *ConfigMap*. Zároveň jsou to dvě různé metody, jak do aplikace dynamicky načítat konfiguraci na základě prostředí.

Dostupné *volumes* jsou pak na řádku 16 namountované na konkrétní umístění v kontejneru.

5.3.2 DeploymentConfig

DeploymentConfig specifikuje šablonu, podle které má být aplikace (pody) nasazená, a zajišťuje, aby skutečnost odpovídala této specifikaci [48]. Objekt typu DeploymentConfig vnitřně používá jiný API objekt – *ReplicationController*, který se stará o to, aby běžel specifikovaný počet podů dané aplikace. Tyto pody jsou identifikované pomocí společného labelu.

Kromě zajištění správného počtu běžících podů DeployementConfig umí v reakci na nějaké spouštěče nasadit novou verzi aplikace. Tímto spouštěčem může být na příklad změna konfigurace, změna referencovaného image v ImageStreamu a podobně.

Dále nabízí několik hooků životního cyklu, které ve spolupráci s různými technikami nasazení aplikace (např. rolling update, recreate nebo vlastní) umožňují flexibilně nastavit proces nasazení nové verze. To je užitečné především v případě, že nová verze vyžaduje změny v databázi a podobně.

DeploymentConfig udržuje verze a nabízí možnost rollbacku v případě neúspěšného pokusu a nasazení nové verze.

Skrze tento objekt je také možné za běhu škálovat aplikaci horizontálně (nastavit počet aktivních podů).

```
1 apiVersion: v1
2 kind: DeploymentConfig
3 metadata:
4   name: sample-dc
5 spec:
```

```
6     replicas: 3
7     selector:
8       name: sample
9     template:
10      metadata:
11      labels:
12        name: sample
13      spec:
14        containers:
15        - image: sample-namespace/sample
16          name: sample-container
17          ports:
18          - containerPort: 8080
19            protocol: TCP
20        restartPolicy: Always
21    triggers:
22    - type: ConfigChange
23    - imageChangeParams:
24        automatic: true
25        containerNames:
26        - sample-container
27        from:
28          kind: ImageStreamTag
29          name: hsample:latest
30      type: ImageChange
31    strategy:
32      type: Rolling
```

Zdrojový kód 5.2: Manifest API objektu DeploymentConfig

Kód 5.2 prezentuje ukázku DeploymentConfigu. Na řádku 6 specifikuje požadovaný počet běžících podů na 3. O řádek níže je definován label, který bude pody sdružovat.

Od 9. řádku je uvedená šablona, která bude použita pro tvorbu podů.

Dále od řádu 21 jsou vyjmenované 2 spouštěče, při jejichž spuštění dojde k postupnému (ř. 32) přenasazení aplikace.

5.3.3 ImageStream

OpenShift pracuje s konceptem *ImageStream* [48]. V tomto objektu je uloženo mapování tagů na konkrétní image (tzv. *ImageStreamTag*).

```
1 apiVersion: image.openshift.io/v1
2 kind: ImageStream
3 metadata:
4   name: sample-is
5   namespace: sample-namespace
6 spec:
```

```
7     tags:
8     - name: stable
9       from:
10        kind: DockerImage
11        name: docker.io/harakiwi/sample:1.0
```

Zdrojový kód 5.3: Manifest API objektu ImageStream

Na příkladu 5.3 je *ImageStream*, který obsahuje jeden tag s názvem *sample-is* (ř. 7). Tento tag ukazuje na image *sample:1.0* v externí Docker registry.

Pokud naše aplikace používá tento ImageStream *sample-is:stable*, je možné nastavit, aby došlo k automatickému rebuildu a nasazení ve chvíli, kdy změníme cílový image z *docker.io/harakiwi/sample:1.0* na něco jiného.

Do ImageStreamu lze dát odkaz na již existující image nebo je do něj možné pushnout nový image např. z BuildConfigu.

5.3.4 BuildConfig

API objekt BuildConfig specifikuje proces sestavení nového image [48].

OpenShift nabízí 3 různé způsoby, jak image sestavit:

- Source-to-Image (S2I)
- Dockerfile/Containerfile
- vlastní strategie

Source-to-Image je nejjednodušší a nejrychlejší možnost, kterou OpenShift nabízí. V principu tato strategie vezme zdrojový kód aplikace a vloží ho do již existujícího image, který obsahuje běhové prostředí. Umožňuje také spustit uživatelem definované skripty např. pro stažení závislostí a podobně.

Dockerfile strategie umožňuje provést build z libovolného base image podle předpisu v Dockerfile.

Speciálním případem vlastní strategie je Jenkins pipeline. Díky integraci OpenShiftu s Jenkinsem tato možnost poskytuje nejvyšší flexibilitu.

```
1  apiVersion: build.openshift.io/v1
2  kind: BuildConfig
3  metadata:
4    labels:
5      name: docker-build
6    name: docker-build
7    namespace: my-app
8  spec:
9    output:
10     to:
11       kind: ImageStreamTag
12       name: sample:latest
13   source:
14     git:
```

```
15          uri: git@bitbucket.org:krulepav/sample.git
16        sourceSecret:
17          name: bitbucket-ssh
18        type: Git
19      strategy:
20        dockerStrategy:
21          dockerfilePath: Containerfile
22        type: Docker
```

Zdrojový kód 5.4: Manifest API objektu BuildConfig

V ukázce 5.4 je použita Docker strategie. K naklonování git repository (ř. 15) se použije privátní SSH klíč uvedený v objektu Secret s názvem *bitbucket-ssh* (ř. 17). V repositáři se použije soubor `./Containerfile` (ř. 21) k sestavení aplikačního image, který se následně publikuje do lokální registry v ImageStreamu *sample*.

5.3.5 ConfigMap

ConfigMap je objekt, který udržuje dvojice informací klíč-hodnota [48]. Tyto informace je možné vložit do kontejneru formou proměnných prostředí nebo jako soubor na filesystému.

```
1 apiVersion: v1
2 kind: ConfigMap
3 metadata:
4   name: sample-config
5   namespace: sample-namespace
6 data:
7   my-config1: "sample value 1"
8   my-config2: "sample value 2"
```

Zdrojový kód 5.5: Manifest API objektu ConfigMap

Definice 5.5 zobrazuje ConfigMapu se 2 hodnotami v textové podobě. Je také možné vložit binární data zakódované pomocí base64 a celé soubory, např. konfigurace aplikace či veřejné klíče.

5.3.6 Secret

Podobně jako ConfigMapa Secret je úložiště dvojic klíč-hodnota [48]. Na rozdíl od ConfigMapy objekt typu Secret je designovaný pro uložení citlivých dat, např. hesel či privátních klíčů.

```
1 apiVersion: v1
2 type: Opaque
3 kind: Secret
4 metadata:
5   name: sample-secret
6   namespace: sample-namespace
```

```
7 data:
8   my-secret: UGF2ZWwgS3J1bGVj
```

Zdrojový kód 5.6: Manifest API objektu Secret

Kód 5.6 zobrazuje definici 1 Secret typu *Opaque* (na ř. 2), což je obecný typ, na který se neaplikují žádné validace. Kromě něj lze použít tajemství typu:

- `kubernetes.io/service-account-token` pro token servisního účtu
- `kubernetes.io/dockercfg` pro soubor `.dockercfg` s přihlašovacími údaji k různým image registry
- `kubernetes.io/dockerconfigjson` pro soubor `.docker/config.json` s přihlašovacími údaji k různým image registry
- `kubernetes.io/basic-auth` pro jméno a heslo Basic Auth
- `kubernetes.io/ssh-auth` pro privátní a veřejný SSH klíč
- `kubernetes.io/tls` pro TLS certifikáty důvěryhodných CA (např. pro `git clone`)

Tento konkrétní objekt obsahuje 1 tajemství, které je zakódované base64. Pozor, base64 v žádném případě není šifra. Objekty typu Secret by neměly být verzované ve VCS, ale měly by být uložené pouze v OpenShiftu.

5.3.7 Service

Objekt typu Service reprezentuje službu jako takovou [48]. Jelikož danou službu vzhledem ke škálování může tvořit více podů, Service tyto pody zastřešuje, přiřazuje jim jednu společnou IP adresu a provádí nad nimi load balancing.

```
1  apiVersion: v1
2  kind: Service
3  metadata:
4    name: sample
5    namespace: sample-namespace
6  spec:
7    ports:
8    - name: http
9      port: 8080
10     protocol: TCP
11     targetPort: 8080
12   selector:
13     name: sample
14   type: ClusterIP
```

Zdrojový kód 5.7: Manifest API objektu Service

Ukázka 5.7 zobrazuje objekt Service, jemuž bude automaticky přiřazena interní IP dostupná v rámci clusteru. Na řádku 7 definuje mapování portů z této Service na porty v podech. Předposlední řádek je jméno a hodnota labelu, podle kterého bude vybírat, na jaké pody směrovat požadavky.

Service umí dále na příklad definovat, jakým způsobem má probíhat load balancing (round-robin, client IP).

5.3.8 Route

Objekt Route umožňuje vystavit služby mimo cluster pomocí HTTP(S) a veřejného DNS záznamu [48]. Tento API objekt umožňuje konfigurovat chování HTTPS (vlastní terminace, pass-through), případné přesměrování nešifrovaných požadavků na zabezpečenou verzi protokolu a podobně.

```
1  apiVersion: route.openshift.io/v1
2  kind: Route
3  metadata:
4    name: sample
5    namespace: sample-namespace
6  spec:
7    host: sample-app-sample-namespace.apps-crc.
     testing
8    port:
9      targetPort: 8080
10   tls:
11     insecureEdgeTerminationPolicy: Redirect
12     termination: edge
13   to:
14     kind: Service
15     name: sample
16     weight: 100
17   wildcardPolicy: None
```

Zdrojový kód 5.8: Manifest API objektu Route

V ukázce 5.8 je vidět objekt Route, který požadavky směruje na službu s názvem *sample* a port 8080. Je možné také specifikovat vícero služeb a nastavit jim určitou váhu pro tzv. A/B testování.

Na řádku 7 je specifikováno doménové jméno, pod kterým chceme mít cestu dostupnou. Pokud ji neuvedeme, OpenShift vygeneruje vlastní.

5.3.9 PersistentVolumeClaim

Kromě správy vypočetních zdrojů (CPU a paměti) se OpenShift stará i o persistentní úložitě [48]. K tomu používá tzv. *PersistentVolume* (PV). Administrátor OpenShiftu muže vytvořit řadu PersistentVolumes, které se mapují např. na síťový file systém. Aplikace si pak může vyžádat určité množství úložiště pomocí objektu PersistentVolumeClaim (PVC). Následně

je možné PVC moutnovat do jednotlivých kontejnerů, kde slouží jako trvalé úložiště, které jé též možné sdílet mezi vícero pody. Adminstrátor nastavuje PV, co se má stát s daty poté, co kontejnery uvolní používané PVC – zda se data mají zachovat či smazat.

Dále je v PVC možné specifikovat tzv. *storage class*, tedy jakousi třídu úložiště, které chceme použít. Administrátor může nastavit různé třídy PV, které mohou reprezentovat např. SSD, HDD, cloudové úložiště a podobně.

```
kind: PersistentVolumeClaim
apiVersion: v1
metadata:
  name: sample-pvc
spec:
  accessModes:
    - ReadWriteMany
  resources:
    requests:
      storage: 1Gi
  storageClassName: gold
```

Zdrojový kód 5.9: Manifest API objektu PersistentVolumeClaim

PVC v ukázce 5.9 žádá 1 GiB úložiště z PV třídy *gold* a specifikuje, že tento PVC může být namountován vícero pody současně v režimu read-write.

Do podu se tento PVC namountuje takto:

```
kind: Pod
apiVersion: v1
metadata:
  name: sample-pod
spec:
  containers:
    - image: sample-namespace/sample
      name: sample
      volumeMounts:
      - mountPath: "/opt/data"
        name: sample-volume
  volumes:
    - name: sample-volume
      persistentVolumeClaim:
        claimName: sample-pvc
```

Zdrojový kód 5.10: Použití PVC v podu

5.3.10 StatefulSet

Objekt StatefulSet je alternativou k DeploymentConfigu, který najde své využití u stavových aplikací [48]. Stavové aplikace obecně nejsou vhodné pro provoz v cloudu vzhledem k jejich obtížnějšímu horizontálnímu škálování.

Nicméně s pomocí tohoto API objektu je možné nasadit v HA konfiguraci i aplikaci, která si udržuje nějaký vnitřní stav.

Zatímco hostname (názvy) podů, které jsou spravovány DeploymentConfigem (skrze ReplicaSet), mají tvar `<name>-<hash>`, kde `<hash>` je náhodný alfanumerický řetězec, hostname podů spravovaných objektem StatefulSet jsou ve tvaru `<name>-<index>`, kde `<index>` je pořadí podu. Díky tomu je hostname předvídatelný a aplikace s ním může snadno pracovat.

Kromě toho, pokud je aplikace vyžaduje, má každý z podů k dispozici svůj vlastní VolumeClaim, takže jednotlivé pody mohou mít svá vlastní data, která nesdílí s ostatními.

```yaml
apiVersion: apps/v1
kind: StatefulSet
metadata:
  name: sample
spec:
  selector:
    matchLabels:
      app: sample
  serviceName: "sample-service"
  replicas: 3
  template:
    metadata:
      labels:
        app: sample
    spec:
      containers:
      - image: sample-namespace/sample
        name: sample-container
        volumeMounts:
        - name: data
          mountPath: /opt/data
  volumeClaimTemplates:
  - metadata:
      name: data
    spec:
      accessModes: [ "ReadWriteOnce" ]
      resources:
        requests:
          storage: 1Gi
```

Zdrojový kód 5.11: Manifest API objektu StatefulSet

Manifest 5.11 definuje StatefulSet, který je svázaný se Service *sample-service*. OpenShift zajistí, aby běžely 3 pody:

- `sample-0`
- `sample-1`

- `sample-2`

Manifest dále specifikuje šablonu pro VolumeClaim o velikost 1 GiB. Jelikož požadujeme 3 repliky, celkem se vytvoří i 3 VolumeClaim a celkové velikosti 3 GiB, přičemž je zaručeno, že pod daným hostname vždy dostane stejný Volume.

5.4 CodeReady Containers

Pokud nemáme k dispozici plnohodnotný OpenShift cluster, můžeme pro testování použít CodeReady Containers [49]. CRC je projekt z dílny RedHat, který si klade za cíl přinést minimalistický OpenShift cluster na vývojářské počítače.

Hardwarové nároky jsou poměrně vysoké, což však není překvapující vzhledem k tomu, že se jedná o téměř plnohodnotný cluster. Minimální konfigurace vyžaduje 4 CPU, 9 GB RAM a 35 GB úložiště.

Z Linuxových distribucí CRC podporuje pouze Fedoru, CentOS a RHEL, tedy distribuce přímo od RedHatu.

Po základní instalaci dle `https://code-ready.github.io/crc/` je vhodné před spuštěním clusteru zvýšit dostupné zdroje. Pokud máme stroj s 12 vCPU a 32 GB RAM nastavíme prostředky např. takto:

```
crc config set cpus 8        # 8 vCPUs
crc config set memory 20408 # 20 GB RAM
```

CodeReady Containers má ve výchozím stavu vypnuté operátory na monitoring, alerting a telemetrii. Ty jsou sice poměrně náročné na zdroje, ale webová konzole je bez nich velmi pomalá, jelikož dochází k vyčerpávání časových limitů. Tyto operátory lze zapnout příkazem [49]:

```
oc patch clusterversion/version \
  --type='json' \
  -p '[{"op":"remove", "path":"/spec/overrides/0"}]'
```

Pro zlepšení efektivity práce je vhodné si nastavit terminál (zsh, bash) – přidat příkaz `oc` na `PATH` a zapnout automatické doplňování argumentů. Toho docílíme přidáním následujících řádků do souboru `/.zshrc` (analogicky i pro bash):

```
eval $(crc oc-env)
source <(oc completion zsh)
```

5.4.1 Příprava prostředí

Jelikož jsou CodeReady Container určené pro běh na vývojářských stanicích a ne na výkonných serverech, je nutné dodat některé komponenty manuálně.

Pro naše účely je potřeba doinstalovat z Operator Hub dodatečné operátory, které nejsou ve výchozím stavu v CRC dostupné. Jedná se o tyto operátory:

1. *Elasticsearch Operator* pro prohledávání a vizualizaci logů (Elasticsearch a Kibana)
2. *Cluster Logging* pro agregaci logů (Fluentd).
3. *Grafana Operator* pro vizuální monitoring (Grafana a Prometheus).

Kromě těchto operátorů je nutné do našeho namespace nainstalovat Jenkins pro CI. To lze provést z *Developer Catalogu* vyhledáním a aplikováním šablony *Jenkins*.

Elasticsearch a Cluster Logging operátoři

Po instalaci Elasticsearch a Cluster Logging operátoru [50] je potřeba vytvořit konfiguraci pro tyto operátory. Možná konfigurace vhodná pro běh na CRC je v příloze C. Tato konfigurace spouští komponenty pouze v 1 instanci.

Jelikož výchozí nastavení OpenShiftu má vypnuté parsování JSON logů, je potřeba to explicitně povolit. Nejprve musíme přepnout ClusterLogging operátora do tzv. "Unamanaged"stavu:

```
oc patch clusterlogging instance \
    -n openshift-logging \
    --type=json \
    -p '[{"op":"add","path":"/spec/managementState
   ","value":"Unmanaged"}]'
```

Následně upravíme konfiguraci Fluentd, aby korektně parsoval JSON logy před posláním do Elasticsearch:

```
oc set env ds/fluentd MERGE_JSON_LOG=true -n
   openshift-logging
```

V případě, že pod s Elasticsearch nestartuje z důvodu této chyby:

```
[1]: max virtual memory areas vm.max_map_count
   [65530] is too low, increase to at least [262144]
```

znamená to, že aktuální verze CRC má chybně vypnutého dalšího operátora (Node Tuning Operator), jehož úkolem je upravovat parametry kernelu jednotlivých uzlů clusteru dle specifikovaných profilů. Je tedy potřeba ručně upravit hodnotu dostupné virtuální paměti pro Elasticsearch těmito příkazy [51]:

```
oc get nodes
oc debug node/<node>
chroot /host
echo "vm.max_map_count=262144" >> /etc/sysctl.conf
sysctl -w vm.max_map_count=262144
```

Dalším krokem je nastavení OpenShiftu tak, aby monitoroval i naši aplikaci. To provedeme vytvořením následující ConfigMapy [52]:

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: cluster-monitoring-config
  namespace: openshift-monitoring
data:
  config.yaml: |
    enableUserWorkload: true
```

Grafana Operator

Zatímco Prometheus, který se stará o agregaci metrik z různých zdrojů, je již součástí OpenShiftu a lze jej přímo využít, integrovanou Grafanu není možné použít pro monitorování vlastních aplikací. Po nainstalování operátoru do separátního namespace je ještě nutné z nastavení operátoru vytvořit novou instanci Grafana s výchozím nastavením.

Následně musíme naši nově vytvořenou Grafanu napojit na existujícího OpenShift Promethea přidáním datového zdroje [53].

Nejprve přidáme servisnímu účtu Grafany roli ke čtení monitorovacích dat:

```
oc adm policy add-cluster-role-to-user \
    cluster-monitoring-view \
    -z grafana-serviceaccount
```

Dále si poznamenáme přihlašovací token k tomuto servisnímu účtu:

```
oc serviceaccounts get-token grafana-serviceaccount
```

Tento token se doplní do objektu DataSource (místo `<TOKEN>`), který definuje, jak má Grafana získávat data z Promethea:

```
apiVersion: integreatly.org/v1alpha1
kind: GrafanaDataSource
metadata:
  name: prometheus-grafanadatasource
spec:
  datasources:
    - access: proxy
      editable: true
      isDefault: true
      jsonData:
        httpHeaderName1: 'Authorization'
        timeInterval: 5s
        tlsSkipVerify: true
      name: Prometheus
      secureJsonData:
        httpHeaderValue1: 'Bearer <TOKEN>'
      type: prometheus
      url: 'https://thanos-querier.openshift-
   monitoring.svc.cluster.local:9091'
  name: prometheus-grafanadatasource.yaml
```

Nyní je možné se do Grafany přihlásit výchozím uživatelským jménem `root` a heslem `secret`.

Kapitola 6
Provoz aplikací

Podle DevOps přístupu práce vývojáře nekončí odevzdáním svého kódu do VCS. Aplikaci je dále potřeba sestavit a otestovat. Je zodpovědností vývojáře, že sestavení bude možné a testy projdou bez chyb. Pokud ne, vývojář musí chyby odstranit. Sestavení a otestování probíhá v rámci CI pipeline, kterou vykonává nějaký z celé řady dostupných nástrojů. Zde si přiblížíme nástroj *Tekton*.

Jakmile je aplikace sestavená a otestovaná, nasadí se na nějaké (typicky testovací) prostředí. Současné aplikace jsou však často velmi komplexní, sestávají se z mnoha komponent a proces nasazení tak může být komplikovaný. Pro zrychlení a eliminaci chyb se používá CD pipeline. Zde představíme nástroj *Helm*, s jehož pomocí lze snadno nasazovat i velmi komplexní aplikace do prostředí OpenShift.

Pokud uživatelé (ať už se jedná o testery nebo reálné zákazníky) narazí na nějakou chybu, opět je potřeba aby ji vývojář opravil. K tomu obvykle potřebuje porozumět logům aplikace a umět v nich vyhledávat. Předpokladem samozřejmě je, že aplikace správně loguje vhodné události a chyby. Ukážeme si, jak prohledávat logy pomocí nástroje *Kibana*.

Není však dobrý nápad jen pasivně čekat, až se vyskytne nějaká chyba s dopadem na uživatele. Aplikaci je potřeba průběžně monitorovat a sledovat její chování. Díky tomu je možné některým typům problémů předcházet. Pro vývojáře je jedině výhodou, pokud rozumí monitorovacím nástrojům a pozná chování své aplikace v zátěži mimo svoji vývojovou stanici. Zde si projdeme možnosti nástroje *Grafana* pro aplikační monitoring.

6.1 Tekton pro CI

Nástrojů pro CI existuje celá řada. OpenShift nativně podporuje Jenkins pipelines a Tekton. Zatímco Jenkins je velmi rozšířený nástroj i mimo svět OpenShiftu a cloudu obecně, Tekton je kubernetes-native řešení.

Tekton, podobně jako další nástroje tohoto typu, není striktně zaměřený na proces CI. Jeho úkolem je automatizace procesů a CI je jen jedním z nich.

Pro využití Tektonu v OpenShiftu je potřeba nainstalovat operátor *OpenShift Pipelines*. Tato funkcionalita se v současnosti nachází ve stavu *Tech Preview* a není tedy zatím vhodná pro produkční nasazení. Po instalaci toho

operátoru přibude v menu webové konzole nová položka *Pipelines*, která zobrazuje aktuálně instalované pipeliny.

Tekton využívá vlastní API objekty pro definici pipeline [54].

Task

Základním objektem je *Task*. Ve své nejstručnější podobě vypadá následovně:

```
apiVersion: tekton.dev/v1beta1
kind: Task
metadata:
  name: hello-world
spec:
  steps:
    - name: hello
      image: ubi8/ubi-minimal
      command: echo
      args: ["Hello world"]
```

Task obsahuje jeden či více kroků, které mohou být buď celý skript nebo 1 příkaz s argumenty.

Task aplikovaný na OpenShift cluster lze spustit příkazem:

```
tkn task start hello-world
```

Step je jedna operace v CI procesu, na příklad kompilace aplikace či spuštění unit testu. *Task* je uspořádaný seznam jednoho či více kroků a je spuštěn v podu. Tento pod obsahuje jeden či více kontejnerů – každý pro jeden krok v Tasku. Díky tomu je možné vytvořit sdílené prostředí pro jednotlivé kroky, na příklad formou mountování adresářů, které jsou dostupné všem krokům/kontejnerům.

Pipeline

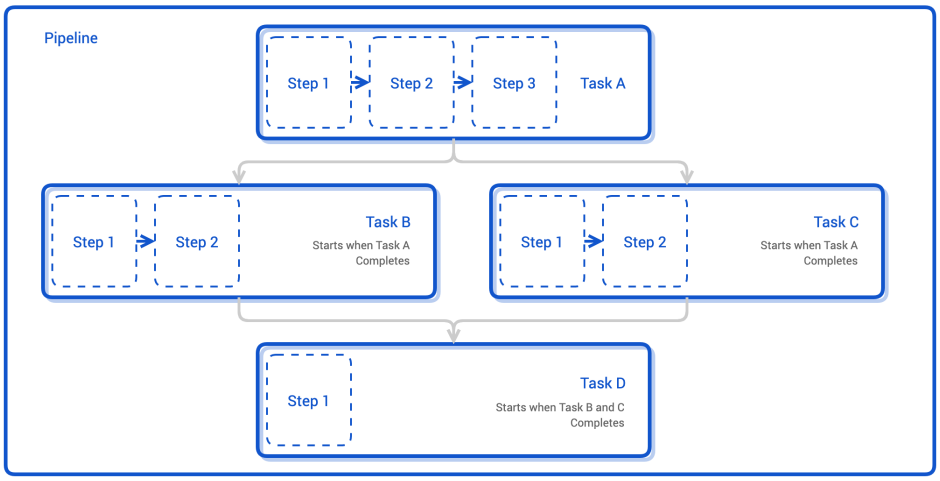
Úkoly se dále sdružují do *Pipeline*:

```
apiVersion: tekton.dev/v1beta1
kind: Pipeline
metadata:
  name: greeting
spec:
  tasks:
  - name: hello
    taskRef:
      name: hello-world
  - name: goodbye
    runAfter:
     - hello
    taskRef:
      name: goodbye-world
```

Pipeline se spustí příkazem:

```
tkn pipeline start greeting
```

Pipeline je v podstatě acyklický graf, který Tekton prochází a vykonává v prostředí OpenShiftu, jak znázorňuje obrázek 6.1.



Obrázek 6.1: Koncept Tekton pipeline [54]

Parametry a zdroje

Task a Pipeline mohou mít také vstupní parametry a vstupní a výstupní *resources*.

Parametry, dle očekávání, slouží ke zobecnění jednotlivých kroků a úkolů. Tekton rozlišuje 2 typy parametrů: string, který je výchozí, a pole. Rozlišení typů souží k validaci uživatelem poskytnutých hodnot.

Resources specifikují vstupní a výstupní zdroje objektu Task. Vstupní zdroje jsou dostupné na filesystemu na cestě `/workspace/<resource_name>`. Výstupní zdroje musí Task zapsat na cestu `/workspace/output/<resource_name>/`.

```
apiVersion: tekton.dev/v1beta1
kind: Task
metadata:
  name: hello
spec:
  params:
    - name: name
      type: string
  resources:
    outputs:
        - name: greeting
  steps:
    - name: greeting
```

```
14        image: ubi8/ubi-minimal
15        script: |
16          #!/bin/bash
17          echo "Hello $(params.name)" > /workspace
    /output/greeting
```

Resources je potřeba definovat ve vlastním API objektu typu *PipelineResource*. Typů Resources existuje několik [54], nejdůležitější jsou *Git* pro vstupní zdroje a *Image* typicky pro výstupní.

Workspace

Další důležitou součástí Tekton pipelines je *Workspace*. Pomocí nich mohou Tasky alokovat místo na filesystému.

Workspaces mohou být vytvořené pouze pro čtení na příklad z ConfigMaps či Secrets. Druhou možností jsou zapisovatelná Workspaces zprostředkovaná *PersistentVolumeClaim*, která umožňují sdílet data mezi Tasky, nebo *emptyDir*, které jsou smazány vždy po konci Tasku [54].

Workspace slouží jako úložiště vstupů a výstupů, místo pro sdílení dat mezi Tasky či cache pro zrychlení dílčích procesů.

Možnosti Tekton pipelines jsou ještě širší a celý ekosystém je stále předmětem intenzivního rozvoje. Aktuální informace tak lze vždy nalézt přímo v dokumentaci [54].

6.2 Helm pro CD

Helm je správce balíčku pro Kubernetes [55]. Pomáhá spravovat aplikace, které jsou definované pomocí *Helm Charts*. Umožňuje instalovat a aktualizovat i velmi komplexní služby.

O projekt Helm se stará Cloud Native Computing Foundation (CNCF), která stejným způsobem zastřešuje i Kubernetes.

Balíčky, se kterými Helm pracuje, se nazývají *Charts*. Jedná se o systém YAML definic, šablon a jejich hodnot a dalších informací. Charty mohou být zabalené do archivu a sdružují se v repositářích. Výrazem *release* se pak označuje jedna aktivní instance Chartu, která je nainstalovaná v clusteru.

Princip Helm je takovýto: Helm Chart obsahuje sadu šablon OpenShift API objektů. Dále má k dispozici výchozí hodnoty, které se do těchto šablon dosadí a je možné je uživatelsky přepsat při instalaci či upgrade Chartu. Helm provede 3-cestný merge, který bere v potaz specifikované objekty (šablony a hodnoty), aktuální release nasazený na clusteru a případné manuální změny, které byly provedeny v clusteru mimo Helm. Získané změny poté aplikuje na cluster.

6.2.1 Helm Chart

Nový Helm Chart vytvoříme příkazem:

```
helm create <chart-name>
```

Vznikne tak výchozí projekt, odkud lze čerpat inspiraci při psaní vlastního Chartu. Jeho adresářová struktura je následující:

```
sample/
+-- charts/
+-- Chart.yaml
+-- templates/
|   +-- _helpers.tpl
|   +-- serviceaccount.yaml
|   +-- service.yaml
|   +-- tests/
|       +-- test-connection.yaml
+-- values.yaml
```

Soubor `Chart.yaml` obsahuje metadata o daném Chartu, jako je například jeho název (který musí být totožný s názvem rodičovského adresáře), verze Chartu a verze aplikace, kterou instaluje.

Soubor `values.yaml` obsahuje hodnoty, které se doplní do šablon. Tento soubor si může uživatel přizpůsobit, hodnoty z něj je možné přepsat i jako parametr při instalaci či upgrade Chartu.

Ve složce `charts` mohou být definovány další Charty, na kterých je tento Chart závislý a potřebuje je jako svou prerekvizitu.

Složka `templates` pak obsahuje všechny šablony, které se budou nasazovat. Kromě nich mohou být součástí i testy ve složce `templates/tests`, které testují správné nasazení API objektů. V souboru `templates/_helpers.tpl` jsou definovány pomocné hodnoty (např. jméno Chartu nebo často používané labely).

Soubory ve složce `templates` Helm před instalací seřadí podle typu, pak podle jména a v tomto pořadí je nainstaluje nebo upgraduje. Pořadí některých vybraných API objektů Kubernetes je následující (nižší číslo odpovídá vyšší prioritě při instalaci):

1. `Namespace`
2. `Secret`
3. `ConfigMap`
4. `PersistentVolumeClaim`
5. `ServiceAccount`
6. `CustomResourceDefinition`
7. `Service`
8. `Pod`
9. `Deployment`

10. `Ingress`

Díky tomu je zajištěno, že nebude docházet ku příkladu k instalaci podu do neexistujícího namespace.

6.2.2 Šablony

Šablona je klasický API objekt, který však navíc může obsahovat speciální výrazy uzavřené do dvojitých složených závorek – `{{` a `}}`. Takovými výrazy mohou být nejčastěji proměnné, ale i řídící struktury pro podmínky a cykly, zahrnutí jiných šablon a tak dále.

Šablonovací systém, který Helm používá, je *Go template language* obohacený o některé dodatečné funkce [55, 56].

```
1 apiVersion: v1
2 kind: Secret
3 metadata:
4   name: {{ .Release.Name }}-secret
5 data:
6   greeting: "Hello world"
7   firstname: {{ .Values.name.first | default "John"
     | quote }}
8   lastname: {{ .Values.name.last | upper | quote }}
9   {{ if ge .Values.age 18 }}adult: true{{ end }}
```

Ukázka 6.2.2 zobrazuje šablonu API objektu Secret. Na řádku 4 se doplní název aktuální release.

Na 7. řádku se použije hodnota `name.first`, která se získá buď ze souboru `values.yaml` nebo jako parametr při instalaci či upgrade tohoto Helm Chartu. Pokud není hodnota specifikována ani v souboru `values.yaml`, ani jako parametr, použije se výchozí hodnota `John`. Celý výsledek se pak obalí do uvozovek.

8. řádek používá hodnotu `name.last`, kterou převede na velká písmena a výsledek opět zabalí do uvozovek.

Na posledním, 9. řádku, se objeví klíč `adult` s hodnotou `true` pouze v případě, že hodnota `age` je větší nebo rovna 18. V opačném případě zůstane řádek prázdný.

Helm Chart s touto šablonou je možné nainstalovat na příklad následujícím příkazem:

```
1 helm install helloworld ./helloworld/ \
2     --set name.last="Doe" \
3     --set age=20
```

Je také možné vytvořit vlastní soubor s hodnotami, což je užitečné, pokud máme různá prostředí, na kterých aplikace běží. Můžeme tedy vytvořit soubor `val-dev.yml`:

```
1 name:
2     first: George
```

```
    last: Doe
age: 20
```

a soubor `val-prod.yml`:

```
name:
    last: Smith
age: 14
```

Díky tomu můžeme snadno přepínat mezi aplikovanými parametry příkazem:

```
helm install helloworld ./helloworld -f val-dev.yml
```

respektive

```
helm install helloworld ./helloworld -f val-prod.yml
```

6.3 Kibana pro logování

Kibana je součástí tzv. EFK stack – systému 3 aplikací, Elasticsearch, Fluentd a Kibana, který tvoří nepsaný standard na poli zpracování logů v cloudových aplikacích.

Fluentd se stará o sběr a transformaci logů z kontejnerů a přeposílá je do Elasticsearch. Elasticsearch je aplikace, která se zaměřuje na full-textové vyhledávání v dokumentech. Kibana používá Elasticsearch jako zdroj dat, které následně vizualizuje.

Na záložce *Discover* je možné data procházet a filtrovat [57]. Kibana zobrazí každý jednotlivý dokument (záznam), který odpovídá zadaným kriteriím. V tomto pohledu lze také snadno pozorovat události, které předcházely událost vyhledávanou nebo následovaly těsně po ní (například různé chyby či jiné anomálie).

Vyhledávání probíhá na základě zvoleného indexu v Elasticsearch. Ku příkladu OpenShift operátor pro logování umožňuje nastavit 3 indexy [50]:

1. *app*, který obsahuje aplikační logy;
2. *infra*, který obsahuje infrastrukturní logy;
3. *audit*, který obsahuje auditní logy.

Pro uživatele je nejdůležitější nastavit první index, *app*, který je založený na čase podle hodnoty *@timestamp*.

Díky tomu, že zvolené indexy jsou založené na čase, je možné filtrovat události, které se udály ve zvoleném časovém intervalu. Toto nastavení se upravuje v pravém horním rohu webového rozhraní.

EFK a ELK stack

Kromě EFK se lze setkat i s výrazem ELF stack. V takovém případě se pro sběr a transformaci místo Fluentd používá program Logstash. Hlavním rozdílem mezi těmito dvěma technologiemi je, že Logstash je centralizovaný, zatímco Fluentd využívá decentralizovanou architekturu [58]. Z toho důvodu je řešení postavené nad Fluentd lépe škálovatelné a tedy vhodnější na rozsáhlejší infrastrukturu. LogStash má také o něco vyšší nároky na paměť. Nevýhodou Fluentd je jeho složitější konfigurace [58], nicméně s využitím OpenShift operátorů lze spolehlivě využívat i výchozí nastavení.

6.3.1 Lucene syntax

K vyhledávání Kibana používá tzv. Lucene syntax [59]. Hledání výrazů je case-insensitive, nezáleží tedy na velikosti písmen. Operátory však musí být psány velkými písmeny.

Prosté výrazy a zástupné znaky

V nejjednodušší podobě je možné vyhledávat jednotlivé výrazy. Výrazem může být jedno slovo:

```
hello
```

Případně více slov v uvozovkách:

```
"hello world"
```

Je také možné používat zástupné znaky `?` pro vynechání 1 znaku a `*` pro vynechání libovolného počtu znaků:

```
"S?mple exp*"
```

Vyhledávání na základě podobnosti

Lucene dále podporuje tzv. fuzzy vyhledávání – vyhledávání na základě Levenshteinovy vzdálenosti mezi dvěma výrazy. Používá k tomu operátor vlnovky, za kterým následuje desetinné číslo od 0 do 1, které vyjadřuje míru podobnosti (hodnota 1 znamená, že výrazy musí být totožné). Vyhledávání výrazu

```
that~0.75
```

tak najde slova jako *than*, *then*, *that*, *what* a podobně.

Kromě toho lze operátor vlnovky použít i pro tzv. vyhledávání v blízkosti. Pokud chceme získat pouze dokumenty, v nichž se slova „hello“ a „world“ nacházejí maximálně 5 slov od sebe, použije zápis:

```
"hello world"~5
```

Prohledávání položek

Jelikož většina záznamů, se kterými se setkáme v oblasti logování, obsahuje nějaké položky, lze vyhledávat i mezi nimi. Můžeme se dotazovat na všechny logy, jejichž úroveň je „error“:

```
level:error
```

Spojování výrazů pomocí operátorů

Podmínky je také možné řetězit pomocí operátorů `AND`, `OR`, `NOT` a závorek. Tyto operátory musí být zapsané velkými písmeny, případně je lze nahradit za `&&`, `||` a `!` respektive. Pokud nás budou zajímat všechny záznamy týkající se části *db* nebo *be*, ale ne *fe*, které skončily chybou, použijeme výraz:

```
(part:db OR part:be NOT part:fe) AND error:true
```

Zatímco operátor `+` před výrazem značí, že výraz musí být ve výsledku přítomný, operátor `-` naopak vynucuje jeho nepřítomnost. Můžeme použít následující dotaz pro vyhledání záznamů, které nutně obsahují *hello*, ale ne *sad*. Mohou, ale nemusí obsahovat výraz *world*:

```
+hello -sad world
```

Vážené výrazy

V některých případech může být užitečné přidat některým vyhledávaným výrazům vyšší váhu. To lze docílit operátorem `^` za výrazem následovaným nezáporným desetinným číslem. Výchozí hodnota je 1. V následujícím výrazu má *world* dvojnásobnou váhu oproti *hello*, zatímco *space* má váhu poloviční:

```
hello world^2 space^0.5
```

Intervaly

Lucene syntax umožňuje vyhledávat pomocí intervalů. Vyhledává se nad lexikograficky seřazenými záznamy.

Vyhledávání může být buď inkluzivní (uzavřený interval) pomocí hranatých závorek nebo exkluzivní (otevřený interval) pomocí složených závorek. Pokud nás budou zajímat stavové kódy od 400 do 599, použijeme zápis:

```
status:[400 TO 599]
```

6.4 Grafana pro monitoring

Grafana je multiplatformní webová aplikace na vizualizaci dat. Podporuje velké množství různých grafů a dalších vizualizačních metod, které lze organizovat do přehledných, jednoduchých i velmi komplexních dashboardů. Díky tomu je velmi oblíbeným nástrojem pro monitoring

Kromě samotného vykreslování poskytuje i funkcionalitu alertingu; zasílání upozornění na různé platformy, pokud jsou splněné určité podmínky. Díky tomu může provozní tým ještě rychleji reagovat na nastalou situaci. Monitoring by tedy měl být součástí každé aplikace, jejíž provoz je nějakým způsobem důležitý.

Grafana sama o sobě data pouze vizualizuje. Potřebuje je však někde získat. K tomu jí slouží tzv. *datasource*, datové zdroje. Grafana podporuje celou řadu datových zdrojů, namátkou na příklad [60]:

- AWS Cloudwatch,
- Azure Monitor,
- Elasticsearch,
- Google Cloud Monitoring,
- InfluxDB,
- MySQL,
- PostgreSQL,
- Prometheus,
- a další.

Pro nás je nejdůležitější Prometheus, který je nativně součástí OpenShiftu.

Prometheus

Prometheus aplikace, jejímž účelem je sběr, agregace a uchování metrik z různých zdrojů. V pravidelných, uživatelem definovaných intervalech provádí *scraping* dat. Typicky se jedná o HTTP GET požadavek na webovou službu, která implementuje tzv. Prometheus Exporter.

Prometheus Exporter je komponenta aplikace, která poskytuje data v určitém formátu tak, aby mu Prometheus rozuměl. Formát je následujícího tvaru:

```
# <Optional comment>
# HELP <metric> Optional description of metric
# TYPE <metric> Optional type of metric
<metric>{label1="optional",foo="bar"} <value>
```

kde `<metric>` je název metriky psaný malými písmeny a podtržítky a `<value>` je desetinné číslo. Můžeme mít vícero metrik se stejným názvem, které se navzájem liší labely (např. `request{status="ok"}` a `request{status="error"}`).

6.4.1 Typy metrik

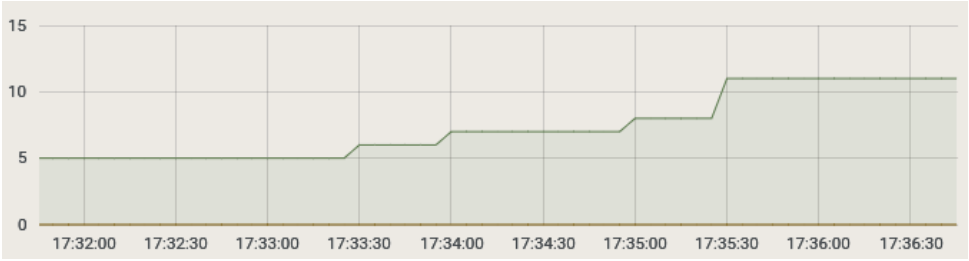
Metrika, jak již název napovídá, je nějaká měřitelná hodnota, číslo. Při provozu aplikací nás zajímá řada metrik, na příklad vytížení procesoru, využití paměti, ale i počet požadavků, kolik z nich skončilo chybou; doba odezvy a podobně.

Grafana, respektive Prometheus, rozlišuje 4 druhy metrik [61]:

- Counter
- Gauge
- Histogram
- Summary

Counter

Counter, neboli čítač, je kumulativní metrika, která zachycuje monotónní hodnoty [61]. Tyto hodnoty jsou neklesající, avšak mohou být resetovány na hodnotu 0.

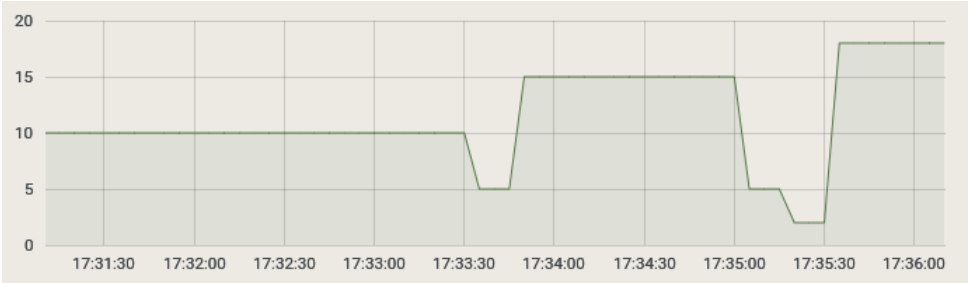


Obrázek 6.2: Vizualizace metriky typu Counter

Counter je vhodný na metriky typu počet zpracovaných úloh, počet chyb a další, neklesající hodnoty.

Gauge

Gauge, což je jednoduše „naměřená hodnota“, reprezentuje číslo, které může libovolně růst i klesat [61]. Využití najde při měření využití procesu, dostupné paměti, aktuálního počtu uživatelů a podobně.



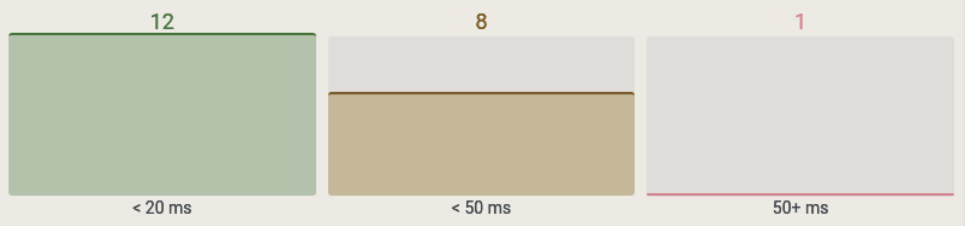
Obrázek 6.3: Vizualizace metriky typu Gauge

Histogram

Histogram zachycuje (sampluje) vzorky nějakého pozorování a zařazuje je do konfigurovatelných kumulativních segmentů (bucketů) [61]. Kromě počtu vzorků v každém segmentu histogram dále poskytuje celkový součet všech hodnot a jejich počet.

Formát histogramu, který Prometheus scrapuje, je tento:

```
<metric>_bucket{lq="<upper_inclusive_bound_1>"}
<metric>_bucket{lq="<upper_inclusive_bound_2>"}
<metric>_sum
<metric>_count
```



Obrázek 6.4: Vizualizace metriky typu Histogram

Příkladem metriky, kterou lze vyjádřit histogramem, je doba odezvy systému – kolik požadavků bylo vyřízeno do 10 ms, do 50 ms, do 200 ms a kolik nad 200 ms.

Summary

Summary neboli přehled, je podobný histogramu. Také zachycuje počet a součet pozorovaných dat. Na rozdíl od histogramu data nerozděluje do bucketů, ale na φ-kvantily ($0 \leq \varphi \leq 1$) v plovoucím časovém oknu [61].

Zatímco typ histogram je dobré zvolit, pokud víme, v jakém rozsahu se hodnoty budou pohybovat, typ Summary použijeme v případě, kdy předem neznáme ani přibližně rozsah metriky

Z histogramu je též možné vypočítat φ-kvantily, ale děje se tak až na straně serveru (Promethea), zatímco v případě Summary kvantily počítá client (tedy klientská aplikace, která poskytuje endpoint s metrikami).

6.4.2 Tvorba dashboardů pomocí PromQL

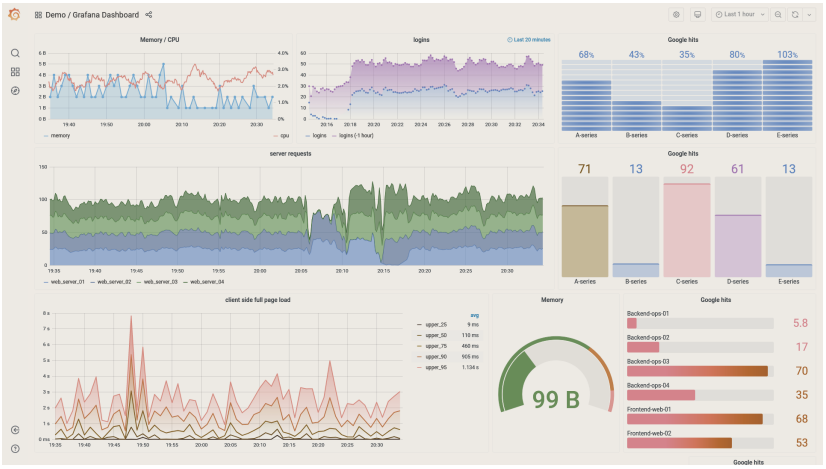
Grafana umožňuje tvorbu propracovaných dashboardů s řadou grafů a dalších vizualizací. Velké množství dashboardů vytvořených komunitou je dostupné přímo na webu Grafany `https://grafana.com/grafana/dashboards`. Odkud je možné je importovat rovnou do naší instance buď pomocí jejich ID nebo ve formátu JSON.

PromQL je dotazovací jazyk, kterým lze naměřená data zpracovávat. Jeho součástí je řada funkcí, které jsou svým chováním podobné například SQL.

Každý výraz se vyhodnotí do jednoho ze 4 možných typů [61]:

- Instant vector (Okamžitý vektor) – sada časových řad obsahující jeden vzorek pro každou časovou řadu, přičemž všechny sdílejí stejné časové razítko
- Range vector (Vektor rozsahu) – sada časových řad obsahující rozsah datových bodů v čase pro každou časovou řadu
- Scalar – desetinné číslo
- String – textový řetězec (aktuálně se nepoužívá)

Ne všechny uvedené typy je možné použít pro každý typ vizualizace. Například vykreslit graf lze pouze z hodnoty typu Instant vector.



Obrázek 6.5: Komplexní Grafana dashboard

Základním dotazem je přímo dotaz na metriku:

```
logback_events
```

Dále je možné filtrovat podle labelů na přesnou shodu, podle regulárního výrazu či negaci výrazu:

```
logback_events{lvl="info",env=~"dev-.*",member!="1"}
```

S hodnotami lze provádět aritmetické operace:

```
system_cpu_usage / (cpu_count{member="1"} + 1) * 100
```

K dispozici je i nepřeberné množství různých funkcí, například průměr hodnot v plovoucích 10 minutách:

```
avg(request_count{status="200"}[10m])
```

Užitečná je i funkce `rate`, která vrací přírůstek ze zadaného Counteru za zvolené časové období. Na příklad počet nově registrovaných uživatelů za plovoucích 30 dní:

```
rate(user_registrations[30d])
```

Tuto funkci lze použít pouze na hodnoty typu Counter, tedy neklesající řady.

Možnosti PromQL jsou mnohem širší a jejich výčet je dostupný v oficiální dokumentaci [61].

Kapitola 7

Ukázková aplikace

V této kapitole si na praktickém příkladu ukážeme, jak zužitkovat znalosti z předchozích kapitol. S použitím frameworku Spring Boot vytvoříme aplikaci v jazyce Kotlin, která bude běžet na platformě OpenShift.

Aplikace se bude jmenovat *randomservice* a bude uživatelům umožňovat vygenerovat náhodné celé číslo od 0 do zvoleného maxima a dále unikátní identifikátor, tzv. UUID.

7.1 Prvotní implementace

V první fázi začneme s nejjednodušší implementací a pokusíme co nejrychleji aplikaci dostat do prostředí OpenShiftu.

Na webu `https://start.spring.io` si vytvoříme základní kostru aplikace v jazyce Kotlin. Po stažení a otevření kostry projektu vytvoříme jednu komponentu typu *Controller* a jednu *Service*.

`RandomController` bude vystavovat 2 endpointy: `/random/integer` a `/random/uuid`:

```
@RestController
@RequestMapping("/random")
class RandomController(val rndSvc: RandomService) {

    @GetMapping("/integer")
    fun getRandomInt(
            @RequestParam bound: Int = Int.MAX_VALUE
    )
            : ResponseEntity<Any> {
        val resp = rndSvc.getInt(bound)
        return ResponseEntity(resp, HttpStatus.OK)
    }

    @GetMapping("/uuid")
    fun getRandomInt(): ResponseEntity<Any> {
        val resp = rndSvc.getUUID()
        return ResponseEntity(resp, HttpStatus.OK)
```

```
17      }
18  }
```

`RandomService` poskytuje implementaci požadovaných služeb:

```
1  @Service
2  class RandomService {
3      private val random = Random()
4
5      fun getInt(bound: Int = Int.MAX_VALUE) = random.
       nextInt(bound)
6
7      fun getUUID() = UUID.randomUUID()
8  }
```

Takovouto aplikaci můžeme sestavit, spustit a otestovat:

```
1  # Build
2  ./mvnw clean package
3
4  # Run
5  java -jar target/*.jar
6
7  # Test
8  curl http://localhost:8080/random/uuid
```

Aplikace vrátí náhodné UUID – Universální Unikátní Identifikátor a je tedy funkční.

7.2 Zabalení do kontejneru

V tuto chvíli máme funkční aplikaci, kterou je potřeba zabalit do kontejneru a připravit k nasazení na OpenShift. Vytvoříme tedy první verzi `Containerfile`:

```
1  # This is a very basic Containerfile which will be
       optimised
2  FROM openjdk:11-jdk-slim
3  LABEL maintainer="P. Krulec <krulepav@fel.cvut.cz>"
4
5  WORKDIR /tmp
6  # Unprofessional, do not copy whole source directory
7  COPY . .
8
9  RUN ["./mvnw", "clean", "package"]
10
11 EXPOSE 8080
12 CMD ["java", "-jar", "target/*.jar"]
```

Zdrojový kód 7.1: První, velmi neoptimální verze `Containerfile` ukázkové aplikace

Uvedený `Containerfile` je velmi neoptimální. Do image kopíruje celou aktuální složku, což znamená, že při změně libovolného souboru (dokonce i `Containerfile` samotného) dojde ke zneplatnění cache a celá akce bude muset probíhat vždy od řádku 6 (včetně) znovu. Navíc, ačkoliv je použit base image s tagem `slim`, který slibuje menší velikost, nakopírováním celého adresáře velikost velmi naroste.

Následujícími příkazy aplikaci vytvoříme image a nahrajeme jej do Docker-Hub registry:

```
podman build --tag randomservice .
podman login docker.io
podman push randomservice:latest \
    docker.io/harakiwi/randomservice:latest
```

7.3 Import externí image do OpenShiftu

Poté, co jsme nahráli image s aplikací do registry, můžeme jej importovat do OpenShiftu. Po spuštění clusteru můžeme v prohlížeči otevřít webovou konzoli na adrese `https://console-openshift-console.apps-crc.testing`. Přihlásíme se výchozím uživatelem `developer:developer` a vytvoříme si namespace *my-app*, kde aplikace poběží.

Tlačítkem *Add* se dostaneme k možnosti *Container Image*, která nám dovolí importovat image z externí registry.

Webová konzole umožňuje poměrně snadno a rychle vše potřebné nastavit pomocí formuláře, který je na obrázku 7.1. Tento formulář na pozadí vytvoří několik různých kubernetes objektů (*Deployment*, *Service*, *Route*, *ImageStream* a *Pod*), které budou rozebrané později.

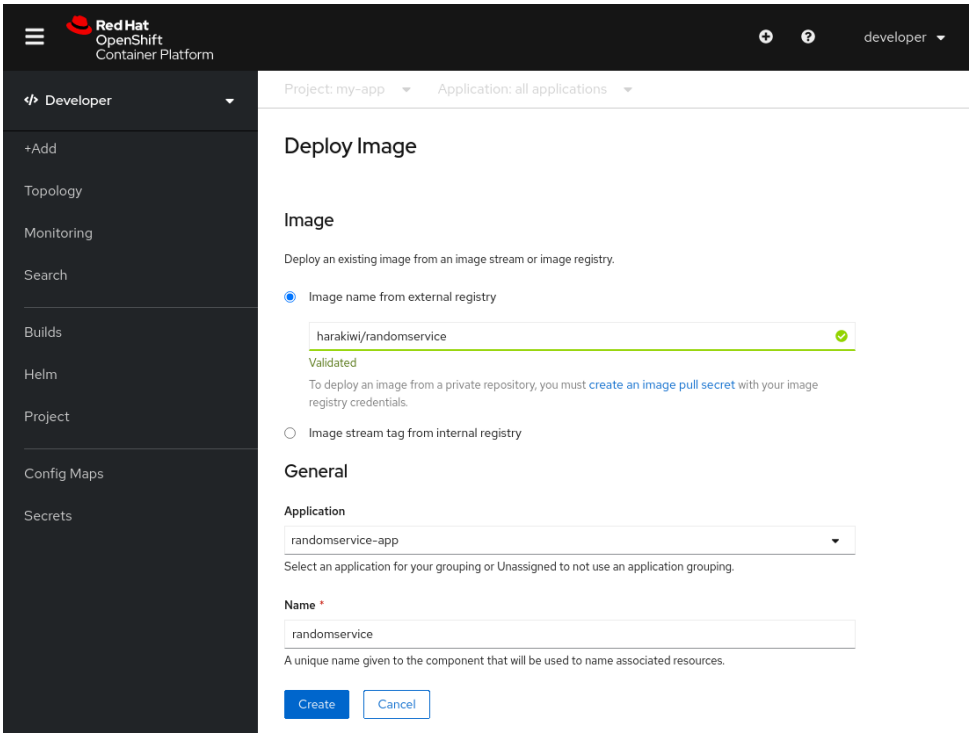
Funkčnost aplikace můžeme ověřit voláním:

```
curl -k "http://randomservice-my-app.apps-crc.
   testing/random/uuid"
```

7.4 Optimalizace Containerfile

První verze `Containerfile` 7.1 je velmi neoptimální, jak z pohledu rychlosti sestavení, tak z výsledné velikost. Image totiž zabírá téměř 600 MB a obsahuje celou řadu věcí, které jsou sice potřebné pro sestavení aplikace, ale ne pro její samotný běh. Pokusíme se tedy výsledný image zbavit těchto závislostí s použitím tzv. *multi-staged buildu* [62]:

```
# 2-staged build, still not optimal
FROM openjdk:11-jdk-slim as builder
WORKDIR /tmp/app
COPY . .
RUN ["./mvnw", "clean", "package"]

```



Obrázek 7.1: Nasazení aplikace na OpenShift z externí container registry

```
7  FROM openjdk:11-jre-slim
8  LABEL maintainer="P. Krulec <krulepav@fel.cvut.cz>"
9  WORKDIR /opt/app
10 COPY --from=builder /tmp/app/target/*.jar ./app.jar
11 EXPOSE 8080
12 CMD ["java", "-jar", "app.jar"]
```

Takto upravený `Containerfile` produkuje znatelně menší finální image – 230 MB. Úspory je dosaženo díky tomu, že sestavení aplikace probíhá v prvním, dočasném image. V tom jsou všechny závislosti potřebné k sestavení aplikace v čele s Java Development Kit (JDK). Po sestavení aplikace dojde ke zkopírování výsledného artefaktu `app.jar` do druhého image, který navíc obsahuje pouze Java Runtime Environment (JRE).

Stále však není příliš optimální kopírování celého adresáře do *builder* kontejneru na řádku 4. Zároveň si lze všimnou, že pokud upravíme nějaký soubor a necháme build proběhnout znovu, dojde ke zneplatnění cache a celý proces jede znovu načisto: Nakopírují se zdrojové kódy do *builder* image, postahují se závislosti, sestaví se aplikace a celý artefakt `app.jar` se nakopíruje do runtime kontejneru. V případě naší malé ukázkové aplikace se to nejeví jako problém, avšak v reálných rozsáhlých aplikacích může mít takový neoptimální proces velké dopady.

K odstranění tohoto neduhu využijeme vlastnost frameworku Spring Boot, tzv. layered JARs [63, 64]. Pokud přidáme do konfigurace pluginu *spring-boot-maven-plugin* v souboru `pom.xml` řádky 7.2, na první pohled se nic

nezmění. Ve skutečnosti však Spring Boot do výsledného JAR souboru přidá meta-informace popisující jednotlivé „vrstvy“.

```
<configuration>
    <layers>
        <enabled>true</enabled>
    </layers>
</configuration>
```

Zdrojový kód 7.2: Zapnutí *layered JAR* vlastnosti ve Spring Boot

Ve výchozím nastavení se vytvoří 4 vrstvy (seřazené vzestupně podle toho, jak pravděpodobně v nich bude docházet ke změnám):

1. `dependencies` – Všechny závislosti, které nejsou ve verzi *SNAPSHOT*
2. `spring-boot-loader` – Kód starající se o načtení tříd z JAR souboru
3. `snapshot-dependencies` – Všechny závislosti, které jsou ve verzi *SNAPSHOT*
4. `application` – Aplikační třídy a další zdroje

Tyto vrstvy nemají samy o sobě nic společného s kontejnery, avšak lze jich tam elegantně využít. Jedná se vlastně o 4 samostatné adresáře v rámci JAR, které si lze (po opětovném sestavené aplikace) vypsat příkazem:

```
./mvnw clean package
java -Djarmode=layertools -jar target/*.jar list
```

Zdrojový kód 7.3: Zobrazení vrstev v *layered JAR*

S takto upraveným JAR souborem můžeme vylepšit i `Containerfile` 7.4 tak, aby vzal již zkompilovaný JAR soubor, ten následně rozbalil na jednotlivé vrstvy. Do výsledného obrazu se pak tyto vrstvy nakopírují s tím, že při opakovaném buildu se využije cache.

```
FROM openjdk:11-jdk-slim as builder
WORKDIR /tmp/app
ARG JAR_FILE=target/*.jar
COPY target/*.jar application.jar
RUN java -Djarmode=layertools -jar application.jar
   extract

FROM openjdk:11-jre-slim
LABEL maintainer="P. Krulec <krulepav@fel.cvut.cz>"
WORKDIR /opt/app
COPY --from=builder /tmp/app/dependencies/ ./
COPY --from=builder /tmp/app/spring-boot-loader/ ./
COPY --from=builder /tmp/app/snapshot-dependencies/
   ./
COPY --from=builder /tmp/app/application/ ./
```

```
14 EXPOSE 8080
15 CMD ["java", "org.springframework.boot.loader.
       JarLauncher"]
```

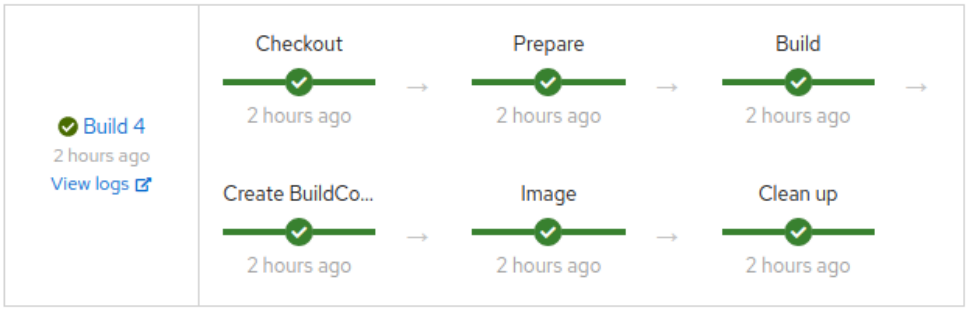
Zdrojový kód 7.4: Finální verze `Containerfile` ukázkové aplikace

7.5 CI pipeline

Nyní víme, jak aplikaci sestavit a zabalit do image. Tento postup automatizujeme pomocí CI pipeline.

Využijeme k tomu nástroj Jenkins, který je vedle Tektonu druhým nativně podporovaným řešením na OpenShiftu. Příklad Tekton pipeline pro tuto ukázkovou aplikaci je součástí zdrojového kódu aplikace.

Začneme vytvořením souboru `Jenkinsfile`, viz přílohu D.



Obrázek 7.2: CI pipeline se 6 fázemi

Tato takzvaná skriptovaná pipeline specifikuje, že se má vykonávat na agentovi s labelem `maven`. Tento agent je součástí výchozí instalace Jenkins na OpenShiftu. Pipeline obsahuje celkem 8 fází:

1. Checkout
2. Prepare
3. Compile application
4. Run tests
5. Package application
6. Create BuildConfig
7. Build image
8. Clean up

Výsledkem této pipeline je hotový image dostupný v interní registry jako ImageStreamTag *randomservice:<verze>*

Zdrojový kód je verzovaný v gitu. V první fázi si Jenkins Master repositář naklonuje, aby měl vůbec přístup k pipeline. Jakmile je pipeline dostupná, spustí master požadovaného agenta a deleguje na něj vykonávání pipeline.

Checkout

Jelikož zdrojový kód je naklonovaný na Jenkins master, je potřeba kód dostat i na agenta. Pokud chceme použít téže revizi, provedeme klonování prostým `checkout scm`. Objekt `scm` dědí hodnoty (URL, větev, autorizační klíč, ...) z mastera.

Prepare

Aby bylo možné image správně otagovat, potřebujeme načíst verzi aplikace ze souboru `pom.xml`. S tím pomůže přímo Maven jedním ze svých pluginů. Hodnotu si uložíme pro pozdější použití.

Compile application

Díky tomu, že build běží na agentovi s nainstalovaným programem Maven, mohli bychom použít ten přímo dostupný. Pro zajištění reprodukovatelnosti sestavení a kompatibility verzí však použijeme Maven wrapper stejně jako při lokálním buildu.

Výsledkem tohoto kroku je aplikace zkompilovaná do bytecode.

Run tests

Zkompilovanou aplikaci můžeme otestovat. Typicky se jedná o unit testy, ale je možné spustit libovolný typ testu, dokonce i end-to-end, dovolují-li to programy dostupné na Jenkins agentovi.

Package application

Jakmile je aplikace otestovaná, je možné z ní vytvořit spustitelný JAR balíček, což je výstupem tohoto kroku.

Create BuildConfig

Ve chvíli, kdy je aplikace sestavená a máme její JAR, necháme na OpenShiftu, aby se postaral o zabalení do image. K tomu potřebujeme vytvořit objekt BuildConfig. Jenkins agent dostupný v OpenShiftu má nainstalovanou utilitu `oc`, s jejíž pomocí BuildConfig snadno vytvoříme.

Za zmínku stojí argument `JAR_FILE=randomservice-${version}.jar`. Jelikož JAR soubor předáváme samotný, není již ve složce `target` a proto je potřeba adekvátně upravit Dockerfile.

Build image

Vytvořený BuildConfig spustíme a počkáme na jeho dokončení. Po dokončení image se OpenShift postará o jeho push do lokální registry, odkud bude možné jej nasadit.

Clean up

V posledním kroku smažeme dočasný BuildConfig. Pokud by během pipeline došlo k chybě, BuildConfig zůstane dostupný i s logy pro snazší dohledávání příčiny problému.

7.5.1 Import CI pipeline do OpenShiftu

Jakmile máme připravenou pipeline, můžeme vytvořit objekt typu BuildConfig pro import do OpenShiftu:

```
1  kind: "BuildConfig"
2  apiVersion: "build.openshift.io/v1"
3  metadata:
4    name: "randomservice-pipeline"
5    labels:
6      app: randomservice
7  spec:
8    source:
9      type: Git
10     git:
11       uri: "git@bitbucket.org:krulepav/
       randomservice.git"
12     sourceSecret:
13       name: bitbucket-ssh
14   strategy:
15     jenkinsPipelineStrategy:
16       jenkinsfilePath: Jenkinsfile
```

Na řádku 13 se odvoláváme na tajemství *bitbucket-ssh*, které obsahuje privátní SSH klíč pro přístup do git repositáře. Toto tajemství je nutné také vytvořit:

```
1  kind: Secret
2  apiVersion: v1
3  type: kubernetes.io/ssh-auth
4  metadata:
5    name: bitbucket-ssh
6  data:
7    ssh-privatekey: >-
8      [base64-encoded-private-key]
9    ssh-publickey: >-
10     [base64-encoded-public-key]
```

Jenkins OpenShift sync plugin se postará o vytvoření Jenkins Jobu a Credentials na základě uvedeného BuildConfigu a tajemství.

7.6 Objekty pro spuštění v OpenShitu

Jakmile máme image, je na čase jej spustit v OpenShiftu. K tomu budeme potřebovat tyto 3 objekty: DeploymentConfig, Service a Route.

DeploymentConfig

V objektu typu DeploymentConfig nastavíme šablonu pro vytváření podů. Především je podstatný image, ze kterého budou vznikat.

Image je potřeba specifikovat včetně adresy registry. Náš image je v interní registry OpenShiftu, jejíž interní adresa je `image-registry.openshift-image-registry.svc:5000`.

Dále uvádíme požadovaný počet replik (2) a label, který oba z podů budou mít.

```
apiVersion: apps.openshift.io/v1
kind: DeploymentConfig
metadata:
  name: randomservice
  labels:
    app: randomservice
spec:
  selector:
    app: randomservice
  replicas: 2
  template:
    metadata:
      labels:
        app: randomservice
    spec:
      containers:
        - name: randomservice
          image: image-registry.openshift-image-registry.svc:5000/my-app/randomservice:0.0.1-SNAPSHOT
          ports:
            - containerPort: 8080
```

Labely jsou důležité, protože pomocí nich mohou další objekty cílit na tyto konkrétní pody.

Aplikováním tohoto objektu po chvíli nastartují pody s naší aplikací. Každý z nich dostane svou interní IP adresu dostupnou v rámci clusteru. DeploymentConfig zajišťuje, že vždy poběží zvolený počet podů. Pokud některý z nich skončí chybou, OpenShift okamžitě spustí nový pod.

Aplikace však není dostupná mimo cluster a požadavky nejsou balancované mezi pody.

Service

Objekt Service vytvoří vrstvu abstrakce nad pody se zvoleným labelem. Tento objekt má vlastní interní IP adresu dostupnou v rámci clusteru a požadavky, které na ni směřují, balancuje nad vybranými pody.

```
apiVersion: v1
kind: Service
metadata:
  name: randomservice
  labels:
    app: randomservice
spec:
  selector:
    app: randomservice
  ports:
    - name: web
      protocol: TCP
      port: 8080
      targetPort: 8080
```

Load balancing je zajištěný, nicméně aplikace stále není dostupná vně clusteru.

Route

Posledním dílkem do skládanky je objekt Route. Ten zajišťuje dostupnost naší služby i z vně clusteru. Má přidělenou veřejnou IP adresu a doménové jméno a ví, na jaký objekt typu Service má požadavky směrovat.

Kromě toho tento objekt terminuje TLS – požadavky z venku na Route jsou šifrované, dále už nikoliv. Zároveň vyžadujeme, aby nešifrované požadavky byly přesměrované na šifrovanou podobu.

```
kind: Route
apiVersion: route.openshift.io/v1
metadata:
  name: randomservice
  labels:
    app: randomservice
spec:
  to:
    kind: Service
    name: randomservice
    weight: 100
  port:
    targetPort: 8080
  tls:
    termination: edge
    insecureEdgeTerminationPolicy: Redirect
```

```
17    wildcardPolicy: None
```

V našem případě delegujeme vygenerování veřejné IP adresy, DNS záznamu i TLS PKI OpenShiftu.

7.7 Nasazení

Aplikace nyní běží v OpenShiftu, je dostupná i mimo cluster, ale potřebujeme mít možnost spravovat verze a řídit jednotlivá nasazení. K tomu využijeme Helm, který OpenShift nativně podporuje.

Helm pracuje s API objekty, které obohacuje o možnost šablonování.

Nový Helm projekt vytvoříme příkazy:

```
mkdir helm && cd helm
helm create randomservice
```

Do vzniklé složky `templates` přesuneme všechny naše API objekty.

Aby nedošlo ke kolizi, musíme nejprve smazat již existující objekty v OpenShiftu, které jsme importovali v předchozích krocích manuálně:

```
oc delete all --selector app=randomservice
```

Následně můžeme aplikaci nasadit pomocí Helmu příkazem:

```
helm install helm/randomservice
```

7.7.1 Šablony

Jako další vylepšení využijeme šablonovacího systému, který Helm nabízí, a některé objekty parametrizujeme.

Konkrétně tajemství `bitbucket-ssh` upravíme tak, aby se SSH klíče načítaly z konfigurace:

```
data:
  ssh-privatekey: {{ .Values.ci.privateKey}}
  ssh-publickey: {{ .Values.ci.publicKey}}
```

Stejně tak upravíme BuildConfig, kterému budeme chtít předávat URL repositáře:

```
git:
  uri: {{ .Values.ci.gitUrl }}
```

Poslední úpravy budou v objektu DeploymentConfig, kde budeme načítat požadovaný počet replik a image, ze kterého se mají pody vytvářet:

```
replicas: {{ .Values.deployment.replicas }}
...
containers:
- name: randomservice
  image: {{ .Values.deployment.registry }}/{{ .
   Release.Namespace }}/randomservice:{{ .Values
   .deployment.version }}
```

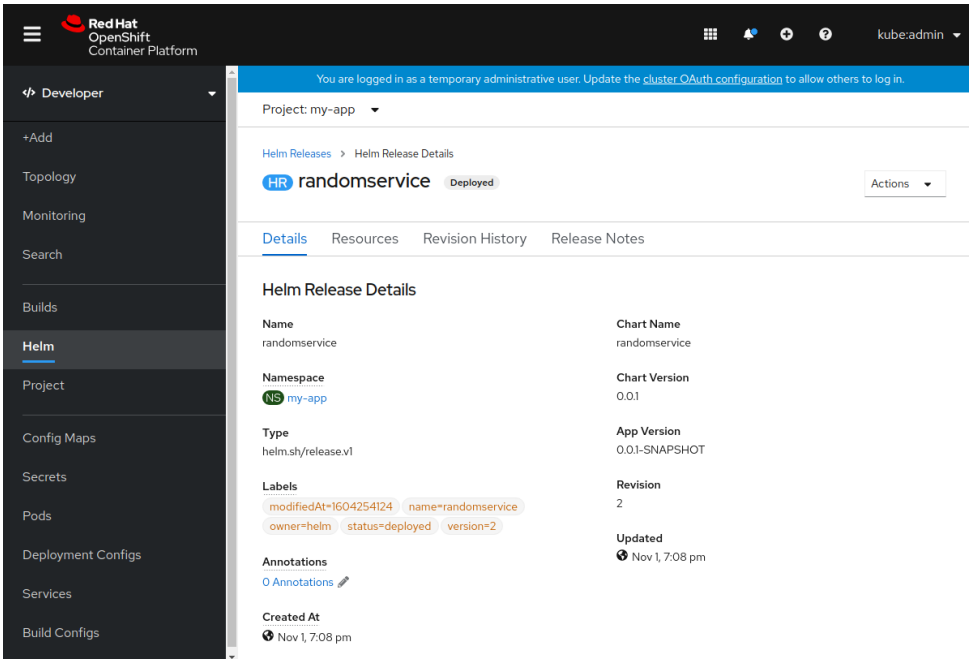
Hodnoty pro tyto parametry definujeme v souboru `values.yaml`:

```
deployment:
  replicas: 1
  version: 0.0.1-SNAPSHOT
  registry: >-
    image-registry.openshift-image-registry.svc
  :5000
ci:
  gitUrl: git@bitbucket.org:krulepav/
   randomservice.git
  privateKey: >-
    [base64-encoded-private-key]
  publicKey: >-
    [base64-encoded-public-key]
```

Takto upravené objekty můžeme znovu nasadit pomocí Helmu příkazem:

```
helm upgrade helm/randomservice
```

Webová konzole na obrázku 7.3 zobrazuje nainstalované Helm charty a jednotlivé releasy. Přimo z webu lze přepínat mezi různými dostupnými verzemi chartů, upravovat konfigurační hodnoty a podobně.



Obrázek 7.3: Detail Helm chartu v konzoli OCP

7.8 Logování

Logování je fundamentální součástí každé aplikace. Bez logů je téměř nemožné dohledat příčinu problémů, které se dříve či později v aplikaci objeví.

OpenShift používá tzv. EFK stack – Elasticsearch, Fluentd a Kibana. Fluentd se stará o sběr logů z podů, Elasticsearch ukládá logy jako JSON dokumenty a Kibana zprostředkovává vizualizaci těchto dat.

S využitím knihoven Slf4j Logback přidáme do aplikace na vhodná místa logování:

```
@Service
class RandomService {
  private val log = LoggerFactory
    .getLogger(RandomService::class.java)
  private val random = Random()

  fun getInt(bound: Int = Int.MAX_VALUE): Int {
    val result = random.nextInt(bound)
    log.debug("Random int between 0 and {} is {}",
        bound, result)
    return result
  }
}
```

Po spuštění aplikace a provolání endpointu v konzoli dostaneme patřičný záznam:

```
2020-10-30 12:47:05.747 DEBUG 1 --- [nio-8080-exec
   -9] c.k.o.r.service.RandomService: Random int
   between 0 and 100 is 42
```

Jedná se o jeden dlouhý řádek. Pokud jich bude více, bude se v nich dobře vyhledávat programem `grep` nebo jemu podobnými.

7.8.1 Logování v JSON formátu

Toto řešení však není vhodné pro cloudové využití. My používáme k agregaci logů Elasticsearch a Kibanu. Protože Elasticsearch pracuje s dokumenty ve formátu JSON, je vhodné nastavit výstup naší aplikace do tohoto formátu, aby bylo možné logy strojově číst a prohledávat.

Úpravu logování do formátu JSON provedeme přidáním konfiguračního souboru `logback-spring.xml` s obsahem:

```
<?xml version="1.0" encoding="UTF-8"?>
<configuration>
  <appender name="consoleAppender"
    class="ch.qos.logback.core.ConsoleAppender">
    <encoder class="net.logstash.logback.encoder.
   LogstashEncoder"/>
  </appender>
  <logger
    name="jsonLogger"
    additivity="false"
    level="DEBUG">
```

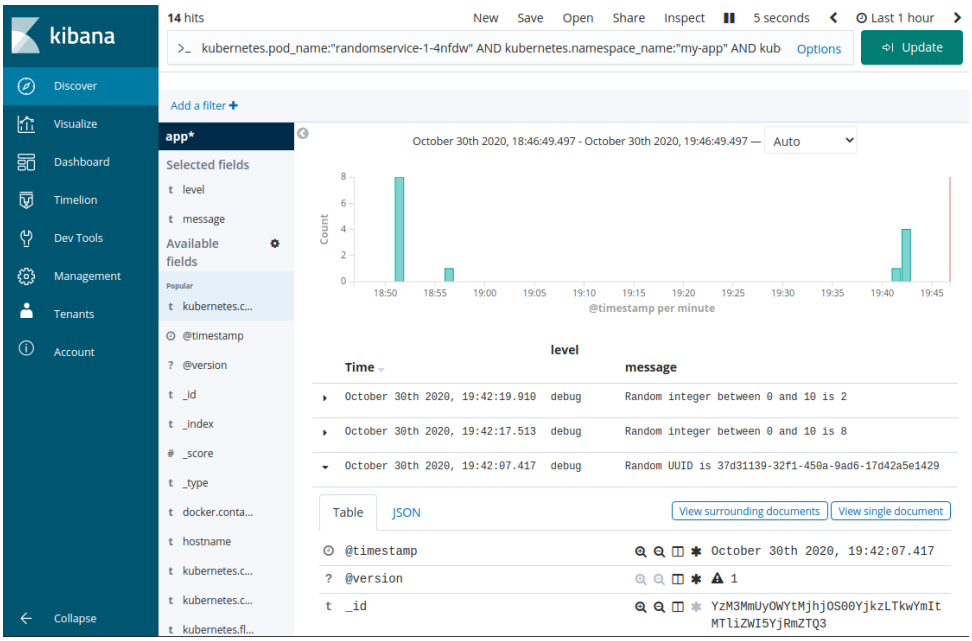
```
11          <appender-ref ref="consoleAppender"/>
12      </logger>
13      <root level="INFO">
14          <appender-ref ref="consoleAppender"/>
15      </root>
16  </configuration>
```

Jelikož knihovny, které Spring Boot používá, nemají podporu pro logování do formátu JSON (encoder na řádku 4), je ještě potřeba přidat dodatečnou závislost do `pom.xml`:

```
1  <dependency>
2      <groupId>net.logstash.logback</groupId>
3      <artifactId>logstash-logback-encoder</artifactId>
4      <version>6.4</version>
5  </dependency>
```

V tuto chvíli již aplikace loguje do formátu JSON, který je sice hůře čitelný lidským okem v porovnání s předchozím prostým formátem, na druhou stranu však nabízí mnohem širší možnosti strojového zpracování.

Logy se začnou postupně objevovat v Kibaně, jak zachycuje obrázek 7.4. Tam s nimi lze dále pracovat, filtrovat je či vizualizovat.



Obrázek 7.4: Zobrazení logů v Kibaně

Můžeme docílit toho, aby aplikace při lokálním vývoji logovala v prostém, neformátovaném textu a v prostředí OpenShiftu logovala ve formátu JSON.

Jednou z možností je, že soubor `logback-spring.xml` jednoduše převedeme do objektu ConfigMap příkazem:

```
1  oc create configmap logback-json \
2    --from-file=src/main/resources/logback-spring.xml
```

Tuto ConfigMapu pak namountujeme jako soubor do složky `config/` vedle aplikačního JAR pomocí záznamu v DeploymentConfigu, jak je popsáno v kapitole 5.3. Pro lokální vývoj můžeme soubor smazat a logování bude neformátované. Složku `config/` a její podsložky Spring Boot prohledává a soubory automaticky přidává na classpath, díky čemuž pak dokáže logování správně nastavit v prostředí OpenShiftu.

7.9 Monitoring

Monitorování stavu aplikace je vedle logování dalším pilířem provozování aplikací. Monitoring pomáhá včas odhalit hrozící problémy, ať už se jedná o využití zdrojů, nadměrné množství chybových odpovědí, zvýšenou síťovou latenci a podobně.

O sběr metrik se v OpenShiftu stará Prometheus. Ten slouží jako datový zdroj pro Grafanu, která na základě získaných metrik vizualizuje informace do uživatelsky vytvořených grafů.

Spring Boot sice umí exportovat metriky pro Prometheus, potřebuje k tomu však dodatečnou závislost:

```
<dependency>
  <groupId>io.micrometer</groupId>
  <artifactId>
    micrometer-registry-prometheus
  </artifactId>
</dependency>
```

Endpoint s metrikami je z bezpečnostních důvodů ve výchozím stavu zakázaný (na rozdíl od *health* a *info* endpointů) a je potřeba jej povolit explicitně přidáním řádku

```
management.endpoints.web.exposure.include=health,
   info,prometheus
```

do souboru `application.properties`.

V tuto chvíli aplikace exportuje základní metriky na cestě `/actuator/prometheus`. Sbíraná data jsou především z těchto oblastí:

- HTTP server – počty požadavků, jejich návratové hodnoty a podobně
- JVM – informace o paměti, garbage collectoru, vláknech, třídách, ...
- Logy – počet záznamů s danou severitou

Aby OpenShift, resp. Prometheus věděl, kde a jak má sbírat metriky z naší aplikace, je nutné vytvořit další API objekt. Jedná se o ServiceMonitor, kde specifikujeme label, podle kterého se budou pody vybírat, a kde se mají metriky číst.

```
apiVersion: monitoring.coreos.com/v1
kind: ServiceMonitor
```

```yaml
metadata:
  name: randomservice-monitor
  labels:
      app: randomservice
spec:
  endpoints:
    - interval: 15s
      port: web
      scheme: http
      path: /actuator/prometheus
  selector:
    matchLabels:
      app: randomservice
```

Po pár minutách se metriky začnou propisovat do Grafany, kde je možné je vizualizovat.

7.9.1 Vlastní metriky

Aplikace samozřejmě může vystavovat vlastní metriky, které pak budou vystavené na stejné adrese, jako ty systémové.

Jelikož všechny potřebné závislosti už máme k dipozici, můžeme do aplikace přidat metriku *random_bound*, která bude zachycovat poslední nastavený parametr `bound`, při získání náhodného celého čísla. Druhou metrikou pak bude počet vygenerovaných náhodných čísel a počet vygenerovaných UUID.

První metrika zachycuje aktuální hodnotu, bude tedy typu *Gauge*. Druhá metrika je kumulativní počet volání a použijeme tedy typ *Counter*.

Upravíme naši službu na generování čísel a UUID.

```
@Service
class RandomService(meterRegistry: MeterRegistry) {
  //...
  private var latestBound = Int.MAX_VALUE
  private lateinit var intCounter: Counter
  private lateinit var uuidCounter: Counter

  init {
      Gauge.builder("random_bound") { latestBound }
          .description("Latest used bound")
          .register(meterRegistry)

      intCounter = Counter.builder("random_count")
          .description("Count of number")
          .tags("type", "int")
          .register(meterRegistry)
      uuidCounter = Counter.builder("random_count")
          .description("Count of UUIDs")
          .tags("type", "uuid")
```

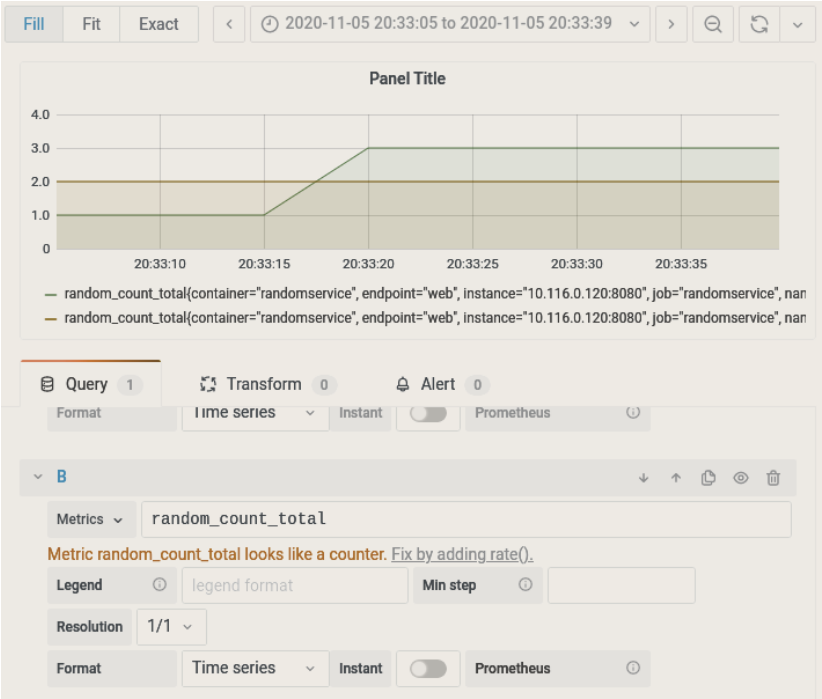
```
            .register(meterRegistry)
    }

    fun getInt(bound: Int = Int.MAX_VALUE): Int {
        intCounter.increment()
        latestBound = bound
        //...
    }

    fun getUUID(): UUID {
        uuidCounter.increment()
        //...
    }
}
```

V kódu přibyly 3 proměnné, ve kterých držíme uložená data. V konstruktoru registrujeme naše metriky do registru, který poskytuje knihovna Micrometer.

1. metrika registruje lambda funkci, která vždy vrátí aktuální hodnotu proměnně `latestBound`. Tu je samozřejmě nezbytné vždy správně nastavit ve funkci `getInt`. Tato metrika nemá žádné tagy.

Dále jsou metriky typu *Gauge*, které reprezentují počet volání každé z metod. Metriky mají stejný název, ale liší se svými tagy, podle kterých se též nechá filtrovat. Hodnoty těchto metrik se zvýší při každém zavolání metod.

Po nasazení nové verze aplikace si můžeme metriky nechat vizualizovat v Grafaně, jak zachycuje obrázek 7.5.



Obrázek 7.5: Vizualizace vlastních metrik v Grafaně

Kapitola 8

Závěr

Vývoj a provoz aplikací je komplexní oblast, která vyžaduje řadu odborností a znalostí a především nutnost být neustále ve střehu a zajímat se o novinky. Každým dnem vznikají nové technologie, některé úspěšnější, jiné nikoliv.

Cílem této práce bylo vysvětlit teorii podstatnou k pochopení problematiky provozování aplikací na cloudové platformě a tyto teoretické poznatky předvést na praktických příkladech a ukázkové aplikaci, kde je lze nejlépe pochopit. Představili jsme si současné technologie, které mají perspektivu na budoucí rozvoj a jsou využitelné v širším měřítku.

V první části této práce jsme si přiblížili cestu, která předcházela současnosti. Rozebrali jsme pojmy virtualizace a kontejnerizace, jaký je mezi nimi rozdíl a případy, pro které jsou vhodné.

Následně jsme si představili nástroj Podman, alternativu ke známému Dockeru. Ukázali jsme si dva různé způsoby, jak s jeho pomocí vytvořit image a spustit jej jako kontejner a na co při tvorbě dávat pozor.

Dále jsme se věnovali cloudu, jeho definici a rozdělení podle různých kritérií. Zmínili jsme praktiky, které se při vývoji a provozu aplikací v cloudu uplatňují, ku příkladu DevOps, CI a CD pipeline; a doporučení, kterých se při vývoji cloudové aplikace držet (tzv. metodika 12 faktorů).

Následně jsme se zaměřili na konkrétní PaaS technologii – OpenShift. Vysvětlili jsme si jeho souvislost s velmi populárním orchestračním nástrojem Kubernetes a představili základní API objekty, pomocí kterých můžeme na této platformě provozovat své aplikace. Abychom mohli vytvořit vlastní aplikaci a připravit ji na provoz v OpenShiftu, ukázali jsme aplikaci CodeReady Containers (CRC) a její nastavení. S pomocí CRC je možné na vlastním, dostatečně výkonném počítači spustit jednonodový OpenShift cluster, který dále využijeme pro vývoj a provoz naší aplikace.

V další kapitole jsme ukázali soubor cloud-native technologií, které nám pomohou s automatizací a dalšími procesy kolem vývoje a provozu software, a které mají v OpenShiftu přímou podporu.

Znalosti získané v předchozích kapitolách jsme nakonec využili při tvorbě ukázkové aplikace krok za krokem. Tato aplikace je sestavována, testována a běží na platformě OpenShift – stejně jako reálná cloudová aplikace.

Příloha A

Zdrojové kódy

Příloha B

Seznam zkratek

Zkratka	Význam
CLI	Command line interface
CD	Continuous delivery
CI	Continuous integration
CRC	CodeReady Containers
GUI	Graphical user interface
HA	High availability
HDD	Hard disk drive
IaaS	Infrastructure as a Service
IPC	Inter-process communication
K8s	Kubernetes
OCP	OpenShift Container Platform
PaaS	Platform as a Service
PR	Pull request
RBAC	Role-based access control
SaaS	Service as a Service
SSD	Solid state drive
SSO	Single sign on
SDN	Software-defined network
S2I	Source-to-Image
UI	User interface
VCS	Version control system

Příloha C

Elasticsearch konfigurační objekt

```yaml
apiVersion: logging.openshift.io/v1
kind: ClusterLogging
metadata:
  namespace: openshift-logging
  name: instance
spec:
  collection:
    logs:
      fluentd: {}
      type: fluentd
  curation:
    curator:
      schedule: 30 3 * * *
    type: curator
  logStore:
    elasticsearch:
      nodeCount: 1
      redundancyPolicy: ZeroRedundancy
      resources:
        limits:
          cpu: 500m
          memory: 4Gi
        requests:
          cpu: 500m
          memory: 4Gi
      storage:
        size: 250M
        storageClassName: ""
    retentionPolicy:
      application:
        maxAge: 1d
      audit:
        maxAge: 30m
      infra:
```

```
35            maxAge: 30m
36        type: elasticsearch
37    managementState: Managed
38    visualization:
39        kibana:
40            replicas: 1
41        type: kibana
42
```

Příloha D

CI pipeline ukázkové aplikace (Jenkinsfile)

```
node("maven") {

    String version
    String buildConfig

    stage("Checkout") {
        checkout scm
    }

    stage("Prepare") {
        version = sh returnStdout: true, script: "./mvnw help:evaluate -Dexpression=project.version -q -DforceStdout"
        buildConfig = "randomservice-$version".replace(".", "-").toLowerCase()
    }

    stage("Compile application") {
        sh "./mvnw clean compile"
    }

    stage("Run unit tests") {
        sh "./mvnw test"
    }

    stage("Package application") {
        sh "./mvnw package -DskipTests"
    }

    stage("Create BuildConfig") {
        sh """cat Containerfile | \
                oc new-build \
                --dockerfile=- \
                --name $buildConfig \
```

```
32                  --to randomservice:$version \
33                  --build-arg JAR_FILE=randomservice-$
    {version}.jar
34              """.trim()
35      }
36
37      stage("Build image") {
38          sh """oc start-build $buildConfig \
39                  --from-file=target/randomservice-${
    version}.jar \
40                  --follow \
41                  --wait
42              """.trim()
43      }
44
45      stage("Clean up") {
46          sh "oc delete buildconfig $buildConfig"
47      }
48  }
49
50
```

Příloha E

Bibliografie

1. DUA, R.; RAJA, A. R.; KAKADIA, D. Virtualization vs Containerization to Support PaaS. In: *2014 IEEE International Conference on Cloud Engineering* [online]. 2014, s. 610–614. Dostupné z DOI: 10.1109/IC2E.2014.41.
2. CROSBY, Simon; BROWN, David. The Virtualization Reality. *Queue* [online]. 2006, roč. 4, č. 10, s. 34–41. ISSN 1542-7730. Dostupné z DOI: 10.1145/1189276.1189289.
3. POMAZAL, Jiří. Virtualizace v kostce: K čemu, jak a proč? *IT Systems* [online]. 2010, roč. 2010. ISSN 802-615X. Dostupné také z: https://www.systemonline.cz/virtualizace/virtualizace-v-kostce.htm.
4. SOJKA, Michal. *Operační systémy: Virtualizace [online]* [online]. 2018 [cit. 2020-04-16]. Dostupné také z: https://cw.fel.cvut.cz/b181/_media/courses/b4b35osy/lekce12_virt.pdf.
5. YANOVSKYY, Vadym. *Forenzní analýza v cloudu [online]* [online]. 2015 [cit. 2020-04-15]. Dostupné také z: https://is.muni.cz/th/oe0kl/.
6. JENNINGS, Michael. Unprivileged Containers for High-Performance Computing [https://permalink.lanl.gov/object/tr?what=info:lanl-repo/lareport/LA-UR-19-23481]. 2019 [cit. 2020-04-16]. Dostupné také z: https://permalink.lanl.gov/object/tr?what=info:lanl-repo/lareport/LA-UR-19-23481.
7. *Docker Toolbox for Windows* [online]. Dostupné také z: https://docs.docker.com/toolbox/toolbox_install_windows/.
8. *Docker WSL for Windows* [online]. Dostupné také z: https://docs.docker.com/docker-for-windows/wsl-tech-preview/.
9. CARLE, Georg; RAUMER, Daniel; SCHWAIGHOFER, Lukas. *Proceedings of the Seminars Future Internet (FI) and Innovative Internet Technologies and Mobile Communications (IITM), Winter Semester 2015/2016*. Chair for Network Architectures a Services, Department of Computer Science, Technische Universität München, 2016. Dostupné z DOI: 10.2313/NET-2016-07-1.
10. *namespaces (7) manpage, release 5.02 of the Linux*. Dostupné také z: http://man7.org/linux/manpages/man7/namespaces.7.html.

11. TURNBULL, James. *The Docker Book: Containerization is the new virtualization*. James Turnbull, 2014. Dostupné také z: https://www.xarg.org/ref/a/B00LRROTI4/.
12. *Linux containers*. Dostupné také z: https://linuxcontainers.org/.
13. *Docker*. Dostupné také z: https://docker.com/.
14. *Podman*. Dostupné také z: https://podman.io/.
15. BAUDE, Brent. Podman: Managing pods and containers in a local container runtime. In: dostupné také z: https://developers.redhat.com/blog/2019/01/15/podman-managing-containers-pods/.
16. *OCI: Open Container Initiative specification*. Dostupné také z: https://github.com/opencontainers/image-spec/blob/master/spec.md.
17. *Portainer: Making Docker management easy*. Dostupné také z: https://www.portainer.io/overview/.
18. BAUDE, Brent. *Project Atomic*. 2018. Dostupné také z: http://www.projectatomic.io/blog/2018/05/podman-varlink/.
19. *Cockpit: Open web-based interface for your servers*. Dostupné také z: https://www.portainer.io/overview/.
20. *Kubernetes: Production-Grade Container Orchestration*. Dostupné také z: https://kubernetes.io/.
21. MODAK, A.; CHAUDHARY, S. D.; PAYGUDE, P. S.; LDATE, S. R. Techniques to Secure Data on Cloud: Docker Swarm or Kubernetes? In: *2018 Second International Conference on Inventive Communication and Computational Technologies (ICICCT)*. 2018, s. 7–12.
22. LUZZARDI, Andrea. *Docker blog*. 2015. Dostupné také z: https://www.docker.com/blog/scale-testing-docker-swarm-30000-containers/.
23. *Kubernetes: Building large clusters*. Dostupné také z: https://kubernetes.io/docs/setup/best-practices/cluster-large/.
24. *OpenShift Container Platform Tested cluster maximums for major releases*. Dostupné také z: https://docs.openshift.com/container-platform/4.4/scalability_and_performance/planning-your-environment-according-to-object-maximums.html.
25. *Building with Buildah: Dockerfiles, command line, or scripts*. Dostupné také z: https://www.redhat.com/sysadmin/building-buildah.
26. *Dockerfile reference*. Dostupné také z: https://docs.docker.com/engine/reference/builder/.
27. *Buildah: Building OCI container images*. Dostupné také z: https://github.com/containers/buildah/blob/master/docs/tutorials/01-intro.md.
28. *Getting started with Buildah*. Dostupné také z: https://www.redhat.com/sysadmin/getting-started-buildah.

29. LI, Wubin; LEMIEUX, Yves; GAO, Jing; ZHAO, Zhuofeng; HAN, Yanbo. Service Mesh: Challenges, State of the Art, and Future Research Opportunities. In: *2019 IEEE International Conference on Service-Oriented System Engineering (SOSE)*. IEEE, 2019. Dostupné z DOI: 10.1109/sose.2019.00026.

30. MELL, P M; GRANCE, T. *The NIST definition of cloud computing*. National Institute of Standards a Technology, 2011. Dostupné z DOI: 10.6028/nist.sp.800-145. Technická zpráva.

31. ARUTYUNOV, V. V. Cloud computing: Its history of development, modern state, and future considerations. *Scientific and Technical Information Processing*. 2012, roč. 39, č. 3, s. 173–178. Dostupné z DOI: 10.3103/s0147688212030082.

32. S. WATTS, M. Raza. *SaaS vs PaaS vs IaaS: What's The Difference and How To Choose*. 2019. Dostupné také z: https://www.bmc.com/blogs/saas-vs-paas-vs-iaas-whats-the-difference-and-how-to-choose/.

33. LI, Abner. *Some Google Photos videos in 'Takeout' backups were sent to strangers*. 2020. Dostupné také z: https://9to5google.com/2020/02/03/google-photos-video-strangers/.

34. *CNCF Cloud Native Definition v1.0*. Dostupné také z: https://github.com/cncf/toc/blob/master/DEFINITION.md.

35. BALALAIE, Armin; HEYDARNOORI, Abbas; JAMSHIDI, Pooyan. Microservices Architecture Enables DevOps : Migration to a Cloud-Native Architecture. *IEEE Software*. 2016, roč. 33, č. 3, s. 42–52. Dostupné z DOI: 10.1109/ms.2016.64.

36. *Design a microservice-oriented application*. Dostupné také z: https://docs.microsoft.com/en-us/dotnet/architecture/microservices/multi-container-microservice-net-applications/microservice-application-design.

37. MEYER, Mathias. Continuous Integration and Its Tools. *IEEE Software*. 2014, roč. 31, č. 3, s. 14–16. Dostupné z DOI: 10.1109/ms.2014.58.

38. BERNARDO, J. H.; ALENCAR DA COSTA, D.; KULESZA, U. Studying the Impact of Adopting Continuous Integration on the Delivery Time of Pull Requests. In: *2018 IEEE/ACM 15th International Conference on Mining Software Repositories (MSR)*. 2018, s. 131–141.

39. CHEN, Lianping. Continuous Delivery: Huge Benefits, but Challenges Too. *IEEE Software*. 2015, roč. 32, č. 2, s. 50–54. Dostupné z DOI: 10.1109/ms.2015.27.

40. WIGGINS, Adam. *The Twelve-Factor App*. Dostupné také z: https://12factor.net.

41. *Heroku*. Dostupné také z: https://www.heroku.com/about.

42. WIGGINS, Adam. *Applying the Unix Process Model to Web Apps*. Dostupné také z: https://adam.herokuapp.com/past/2011/5/9/applying_the_unix_process_model_to_web_apps/.

43. ANTON MCCONVILLE, Olaph Wagoner. *IBM Blog: A brief history of Kubernetes, OpenShift, and IBM*. 2019. Dostupné také z: https://developer.ibm.com/technologies/containers/blogs/a-brief-history-of-red-hat-openshift/.

44. FERNANDES, Joe. *Red Hat Blog: Why Red Hat Chose Kubernetes for OpenShift*. 2016. Dostupné také z: https://www.openshift.com/blog/red-hat-chose-kubernetes-openshift.

45. QU, Mark. *Introduction to Red Hat OpenShift 4: A hybrid cloud, enterprise Kubernetes application platform*. Dostupné také z: https://www.acsip.org/wp-content/uploads/2020/06/Red-Hat-OpenShift-Slides.pdf.

46. CHRAIBI, Jaafar. *Red Hat Blog: Enterprise Kubernetes with OpenShift (Part one)*. 2020. Dostupné také z: https://www.openshift.com/blog/enterprise-kubernetes-with-openshift-part-one.

47. CHOLEWA, Tomasz. *10 most important differences between OpenShift and Kubernetes*. 2019. Dostupné také z: https://cloudowski.com/articles/10-differences-between-openshift-and-kubernetes/.

48. *OpenShift Docs: API Index*. Dostupné také z: https://docs.openshift.com/container-platform/4.5/rest_api/index.html.

49. *Red Hat CodeReady Containers: OpenShift 4 on your Laptop*. Dostupné také z: https://github.com/code-ready/crc.

50. *OpenShift Docs: Installing cluster logging*. Dostupné také z: https://docs.openshift.com/container-platform/4.5/logging/cluster-logging-deploying.html.

51. *OpenShift/Elasticsearch-operator: Running eElasticsearch on CRC*. Dostupné také z: https://github.com/openshift/elasticsearch-operator/issues/527#issuecomment-706484497.

52. *OpenShift Docs: Monitoring your own services*. Dostupné také z: https://docs.openshift.com/container-platform/4.5/monitoring/monitoring-your-own-services.html.

53. CHUNG, Kevin. *Red Hat Blog: Custom Grafana dashboards for Red Hat OpenShift Container Platform 4*. 2020. Dostupné také z: https://www.redhat.com/en/blog/custom-grafana-dashboards-red-hat-openshift-container-platform-4.

54. *Tekton: Documentation*. Dostupné také z: https://tekton.dev/docs/.

55. *Helm: Documentation*. Dostupné také z: https://helm.sh/docs/.

56. *Go: Template Docs*. Dostupné také z: https://golang.org/pkg/text/template/.

57. *Kibana: Guide*. Dostupné také z: https://www.elastic.co/guide/en/kibana/7.10/index.html.

58. *Fluentd vs Logstash: A Comparison of Log Collectors*. Dostupné také z: https://logz.io/blog/fluentd-logstash/.

59. CARLSON, Peter. Apache Lucene-Query Parser Syntax. 2006. Dostupné také z: https://lucene.apache.org/core/2_9_4/queryparsersyntax.pdf.

60. *Grafana: Documentation*. Dostupné také z: https://grafana.com/docs/.

61. *Prometheus: Documentation*. Dostupné také z: https://prometheus.io/docs/.

62. *Docker Docs: Use multi-stage builds*. Dostupné také z: https://docs.docker.com/develop/develop-images/multistage-build/.

63. *Spring Blog: Creating Docker images with Spring*. Dostupné také z: https://spring.io/blog/2020/01/27/creating-docker-images-with-spring-boot-2-3-0-m1.

64. *Spring Docs: Layered Jars*. Dostupné také z: https://docs.spring.io/spring-boot/docs/current/maven-plugin/reference/html/#repackage-layers.