Univerzita Hradec Králové

Fakulta informatiky a managementu

Katedra informatiky a kvantitativních metod

Tvorba a využití on-premise Kubernetes clusteru

Diplomová práce

Autor: Hynek Proske

Studijní obor: Aplikovaná informatika

Vedoucí práce: doc. Mgr. Tomáš Kozel, Ph.D.

Hradec Králové 21. listopad 2019

Prohlášení

Prohlašuji, že jsem diplomovou práci zpracoval samostatně a s použitím uvedené literatury.

V Hradci Králové dne Hynek Proske

Poděkování

Děkuji svému vedoucímu práce doc. Mgr. Tomáši Kozlovi, Ph.D. za vedení práce a užitečné připomínky, které mi při psaní velice pomohly. Dále patří mé poděkování firmě Generali Česká pojišťovna a.s., a to za možnost tuto práci realizovat v jejich prostředí a s jejími aplikacemi.

Anotace

Cílem této práce je přiblížení problematiky kontejnerizace aplikací, jejich provozu v on-premise Kubernetes clusteru a také tvorba a správa takového clusteru. V práci jsou popsány základy technologie kontejnerizace, jednotlivé prvky Kubernetes clusteru a jejich role, možnosti tvorby Kubernetes clusteru, a také nástroje, které lze využít k automatickému a kontrolovanému nasazování kontejnerizovaných aplikací do Kubernetes. Zmíněny jsou navíc způsoby, kterými lze získat větší přehled o stavu provozovaného clusteru a o stavu aplikací v něm.

Annotation

The goal of this Major thesis is to more closely describe the area of working with containerized applications, their deployment to an on-premise Kubernetes cluster and the creation of such cluster. This thesis describes the basics of technology behind containerization, individual pieces of the Kubernetes cluster and their roles, the ways to create a Kubernetes cluster and also tools, which can be used for automatic and controlled deployment of containerized applications to Kubernetes. Additionally, tools that can provide better overview of the cluster and its applications are introduced.

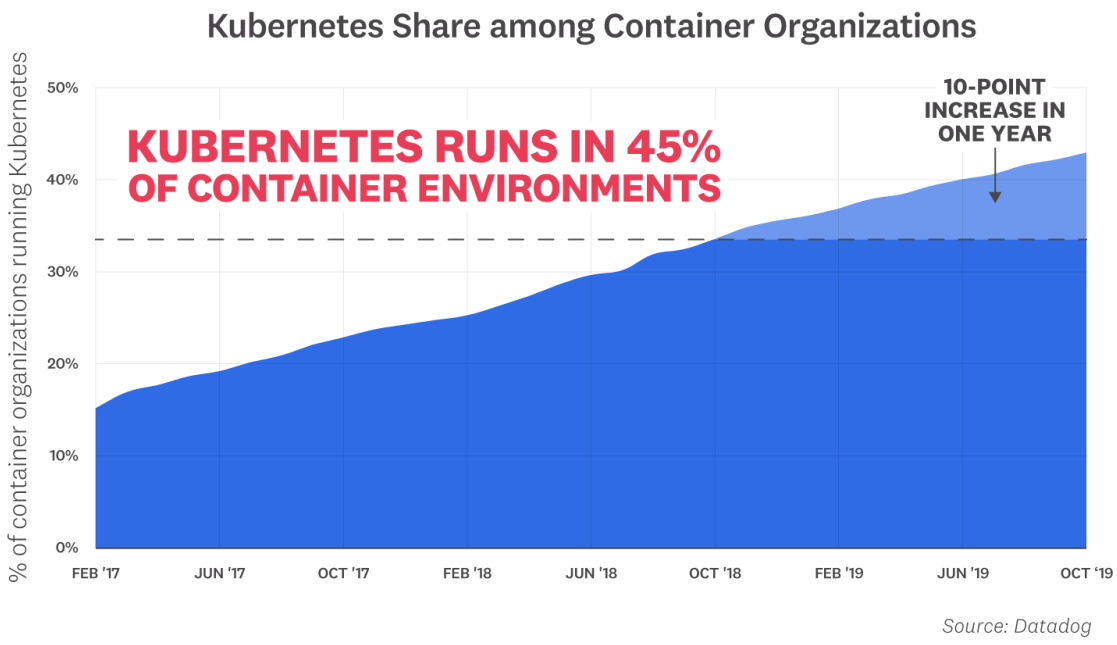
Obsah

1. Úvod

Využívání kontejnerů stabilně roste již několik let, což lze vidět například na datech publikovaných firmou Datadog, zabývající se monitoringem infrastruktury. Ta ukazují, že počet jejich zákazníků, kteří nějakým způsobem využívají Docker, roste o 3-5 procent ročně již od roku 2015 (1).

S růstem adopce kontejnerizace následně vyvstává potřeba takto provozované aplikace nějak spravovat. V tomto směru zaznamenává stabilní růst nástroj pro orchestraci kontejnerů Kubernetes. Ten je, podle společnosti Datadog (2), používán ve 45 % prostředí využívajících kontejnery.

Kubernetes je však komplexní nástroj s rozsáhlým ekosystémem a neexistuje jeden přímočarý způsob, jak jej zavést a používat. Navíc se velmi rychle vyvíjí, s novými minor verzemi [1] vycházejícími každé ~3 měsíce. Je tedy klíčové mít schopnost se v tomto ekosystému orientovat a zvolit vhodný způsob instalace a provozu.



Obr. 1 Vývoj počtu Datadog zákazníků, kteří k provozu kontejnerů používají Kubernetes. Zahrnuje jak vlastní řešení, tak cloudové nabídky spravovaných Kubernetes instalací jako GKE[2], AKS[3] a EKS[4] (2).

[1] Nová verze aplikace přidávající funkcionalitu se zachováním zpětné kompatibility. Přístup k verzování SemVer (3).
[2] Google Kubernetes Engine
[3] Azure Kubernetes Service
[4] Amazon Elastic Kubernetes Service

Motivací pro tuto práci je prozkoumat tyto způsoby pro firmu Generali Česká pojišťovna a.s. a poskytnout tak možnost adopce provozu kontejnerizovaných aplikací. K provozu Kubernetes clusteru je možné použít některé z cloudových nabídek, nebo jej provozovat vlastnoručně.

Tato práce bude pojednávat o vlastní instalaci Kubernetes na firmou vlastněných serverech, tedy o tzv. on-premise instalaci, a její správě. Dále se pokusí navrhnout vhodný přístup zavedení samotných aplikací do vytvořené Kubernetes instalace.

2. Cíle práce

Účelem této práce je seznámit čtenáře s problematikou kontejnerizace a se základy orchestrace kontejnerů pomocí nástroje Kubernetes. Mimo jiné si také klade za cíl uvést přehled možných přístupů k tvorbě on-premise clusteru a způsoby definice aplikací pro běh v Kubernetes clusteru.

Následným cílem je popis procesu tvorby on-premise clusteru pomocí nástroje Kubeadm a možné vylepšení či zjednodušení tohoto procesu. Zároveň budou zmíněny základní doporučené nástroje používané pro udržování přehledu nad clusterem a popsána jejich instalace a konfigurace.

Dále bude také popsán přístup ke konfiguraci a k nasazování aplikací do vytvořeného clusteru zvaný GitOps. Jeho hlavním představitelem je nástroj Weave Flux. Bude popsán princip jeho fungování, instalace a způsob využití k nasazení a konfiguraci aplikací.

Jako poslední bude uvedena doplňková aplikace pro nástroj Weave Flux, která přidává funkcionalitu přijímání webhooků o nových událostech potřebných k nasazování.

Výsledkem této práce tedy bude

- poskytnutí základních informací o kontejnerizaci,
- seznámení s fungováním nástroje Kubernetes a jím zaváděných prostředků,
- popis postupu tvorby on-premise Kubernetes clusteru pomocí Kubeadm,
- návrh automatizace a zjednodušení procesu pomocí Ansible,
- představení a popis nasazení nástrojů pomáhajících se správou clusteru,
- popis workflow pro nasazování a provoz aplikací pomocí Weave Flux.

3. Kontejnerizace

Kontejnerizace umožňuje spouštět aplikace v relativně izolovaném prostředí bez nutnosti použití hardwarově náročných technologií typu virtualizace. Je vhodné mít představu o tom, jak je této funkcionality dosaženo, a orientovat se v terminologii, která se při práci s kontejnery a Kubernetes používá.

3.1 Historie

Cesta ke kontejnerizaci, jak ji známe dnes, začala již v roce 1979 zavedením systémového voláni *chroot* v UNIX. Tento příkaz způsobil to, že změnil kořenový adresář [1] pro daný proces a jeho potomky, které tím pádem nemohly přistupovat k souborům ležícím mimo tuto nově vytyčenou strukturu. Proto se takovýmto prostředím přezdívalo *chroot jail* (vězení). (4)

V roce 2000 byl koncept volání *chroot* rozšířen v systému FreeBSD, kde bylo nyní možné rozdělit celý systém na několik nezávislých částí – *jails.* Co tato funkcionalita přinášela navíc oproti *chroot* bylo to, že izolování se nevztahovalo pouze na souborový systém, ale i na všechny ostatní aspekty operačního systému, včetně IP adresy. (4)

V roce 2005 vzniklo OpenVZ, jež umožňovalo systémovou virtualizaci, a tedy možnost spouštět několik izolovaných instancí operačního systému s vlastními soubory, uživateli, skupinami, procesy a virtuálními síťovými zařízeními. Takovým instancím se říkalo kontejnery. Této funkcionality bylo dosaženo použitím upraveného linuxového jádra, které bylo mezi kontejnery sdílené. (4)

Krátce po OpenVZ, v roce 2006, vznikly tzv. cgroups (control groups), původně pod názvem *process containers.* Jedná se o schopnost limitovat a izolovat využití zdrojů pro skupinu procesů přidanou přímo do linuxového jádra. Mezi tyto zdroje patří využití CPU, paměti, zápisu na disk nebo síťová aktivita. (4)

3.2 Moderní kontejnerizace v Linux

Kontejnerizaci tak, jak ji známe dnes umožňují dvě základní vlastnosti v linuxovém jádře. Konkrétně jde o tzv. namespaces a cgroups.

[1] V UNIX/UNIX-like operačních systémech je kořenový adresář nejvýš v adresářové hierarchii a je značen /.

Namespaces umožňují požadovanou izolaci, tedy mohou izolovat určité prvky systému tak, aby se procesům běžícím v tomto namespace jevily jako jejich vlastní izolované instance. (5)

Existuje několik typů namespace:

- Mount – izolace přípojných bodů souborového systému, umožňující upravovat to, jak proces vidí souborový systém nebo omezovat přístupová práva k nim.
- UTS – umožňuje, aby každý kontejner měl vlastní hostname[1] a doménové jméno.
- IPC – izolace IPC [2] umožňuje, aby několik kontejnerů vytvářelo sdílené paměťové bloky se stejnými názvy, ale nemohly k nim přistupovat mezi sebou.
- PID – izolace seznamu procesů – procesy v kontejneru vidí pouze zaváděcí proces a jeho potomky a žádné procesy z hostujícího operačního systému.
- Network – umožňuje kontejneru síťovou komunikaci, přidá vlastní loopback interface a udržuje vlastní informace o IP adrese a směrovací tabulce.
- User – izolace uživatelů a jejich oprávnění. (7)

Namespace tedy dokáží efektivně izolovat důležité části systému pro spouštěné procesy. Toho však dosahuje tím, že pro procesy vytvoří prostředí, ve kterém má proces zdánlivě k dispozici veškeré systémové zdroje. Aby se předešlo tomu, že proces v kontejneru zabere veškeré tyto zdroje, dalším klíčovým prvkem při práci s kontejnery jsou právě dříve zmíněné cgroups. (7)

Cgroups je vlastnost Linuxového jádra umožňující organizaci procesů do skupin a následnou kontrolu/omezování systémových zdrojů využívaných těmito procesy. Mezi tyto zdroje patří například využití paměti, CPU či zápisu na disk. (8)

3.3 Docker

Na základě těchto vlastností linuxového jádra vznikly kontejnery a nástroje pro práci s nimi. Jedním z takových nástrojů je Docker. Docker byl vytvořen jako monolitický systém řešící veškeré úkony spojené s prací s kontejnery.

[1] Název zařízení sloužící pro jeho identifikaci v síti.
[2] Mechanismus operačního systému umožňující správu sdílených dat mezi procesy (6).

Docker funguje na principu klient-server architektury. Skládá se tedy ze dvou částí; těmi jsou

- klient, který využívá docker CLI a
- daemon[1], který provádí veškeré operace nad kontejnery (9).

Jedním z hlavních prvků, se kterými Docker pracuje, je image. Jedná se o předpis umožňující tvorbu kontejnerů. Image definujeme konfiguračním souborem, obecně známým jako Dockerfile. V tom je popsáno, na jakém základu má kontejner běžet – většinou se jedná o nějaký linuxový operační systém nebo již předpřipravenou image s určitými nástroji – načež následuje libovolná konfigurace podle potřeb uživatele a příkaz pro běh samotné aplikace.

Spuštěním image vznikne kontejner. V kontejneru je spuštěn definovaný systém a aplikace. Jak bylo zmíněno výše, kontejner je od hostitelského systému izolovaný, avšak přesto je možné nakonfigurovat určitou interakci s ním, a to například vystavení portů nebo připojení úložiště.

Dalším z důležitých konceptů, které vznikly spolu s Dockerem je tzv. registry. Jedná se o repozitář pro kontejnerové image, který umožňuje jejich ukládání, nahrávání a stahování.

3.4 Alternativy a standardizace

Jelikož byl Docker vytvořen jako monolitický systém, příliš neumožňoval vznik nových nástrojů stavějících například jen na funkcionalitě pro běh kontejnerů Časem se ukázalo, že je výhodnější rozdělit jednotlivé funkcionality do samostatných nástrojů, které by mohly spolupracovat.

Tak vznikla Open Container Initiative (OCI) a OCI runtime specification (10), pro niž byla vytvořena knihovna runC, sloužící jako reference pro implementaci standardu pro běh kontejnerů (11). RunC používá převážné množství nástrojů pro práci s kontejnery.

S postupem času vznikaly nástroje specializující se na různé úkony spojené s prací s kontejnery. Docker momentálně pro implementaci veškerých operací nejen s kontejnery,

[1] Program běžící v pozadí systému na rozdíl od klasického interaktivního uživatelského módu.

ale i se systémem souborů, přenosem image apod. používá containerd (12). Pro samotný běh kontejnerů poté containerd používá právě runC.

V tuto dobu existuje velké množství nástrojů poskytujících různé úrovně funkcionality související s prací s kontejnery. Jedním z nástrojů, o němž je vhodné se zmínit, je CRI-O (13). Je nabízen jako lightweight alternativa Dockeru v roli container runtime pro Kubernetes. Implementuje Kubernetes Container Runtime Interface a pro běh kontejnerů umožňuje používat runC nebo Kata containers.

Další z nástrojů vzrůstajících na popularitě jsou Podman, Buildah a Skopeo jejichž použití doporučuje firma RedHat. Jedná se o sadu nástrojů s cílem nahradit monolitický přístup dockeru, který navíc vyžaduje daemon proces s root přístupem, což lze považovat za nevhodné řešení.

Buildah se soustředí na tvorbu kontejnerových image, Podman umožňuje nahradit Docker příkazy spojené s během kontejnerů a Skopeo slouží pro práci s kontejnerovými image v image registrech. Většina těchto nástrojů se snaží zachovat známe příkazy z Docker CLI. (14)

Na rozdíl od Dockeru jsou image a kontejnery specifické pro uživatele, který s nimi pracuje. Pokud tedy provedeme například build image jako root pomocí sudo, tato image bude náležet root uživateli a nacházet se v jeho domovském adresáři. (14)

4. Kubernetes cluster

Kubernetes cluster pracuje nad určitým počtem serverů, tzv. nodů, na kterých spouští a kontejnerizované aplikace a dohlíží na ně; tím se nad těmito servery vytvoří jedno logické prostředí. Nody pak mohou mít jednu ze dvou rolí, a to buď master, nebo worker.

Nody typu master slouží pouze pro obsluhu úkonů souvisejících s clusterem samotným. Starají se o správu worker nodů a o to, co v nich má běžet. Reagují na změny v konfiguraci a rozhodují o následných krocích potřebných k dosažení požadovaného stavu. Worker nody jsou využívány čistě pro běh požadovaných kontejnerů (15). V kontextu Kubernetes je nejmenší jednotkou nasazení tzv. Pod, jež je detailněji popsán v kapitole 4.4.1.

4.1 Komponenty clusteru

Kube-apiserver – vystavuje Kubernetes API. Je to prvek používaný pro jakoukoli manipulaci s definovaným stavem clusteru. Požadavky na změny zpracovává, validuje a synchronizuje s etcd (15). Je provozován na master nodu pomocí pevně dané konfigurace.

Etcd – úložiště typu klíč-hodnota obsahující celý stav Kubernetes clusteru (15). Tento prvek je provozován jako Pod na master nodu. Je možné mít několik instancí etcd pro větší odolnost v případě výpadku. Uložená data je vhodné zálohovat.

Kubelet – služba běžící na každém nodu. Jedná se o agenta, který má dva hlavní úkoly. Prvním z nich je registrovat node v apiserveru a informovat master node o stavu nodu, na kterém běží. Druhým je zajistit, že požadované kontejnery jsou na nodu spuštěny a běží. Kubelet je provozován jako linuxový daemon (16).

Kube-scheduler – prvek rozhodující o umístění nově vytvořeného Podu na některý z dostupných worker nodů. Může se rozhodovat na základě hardwarových požadavků spouštěné aplikace a zavedených politik nakonfigurovaných pro daný cluster (15). Běží jako Pod na master nodu.

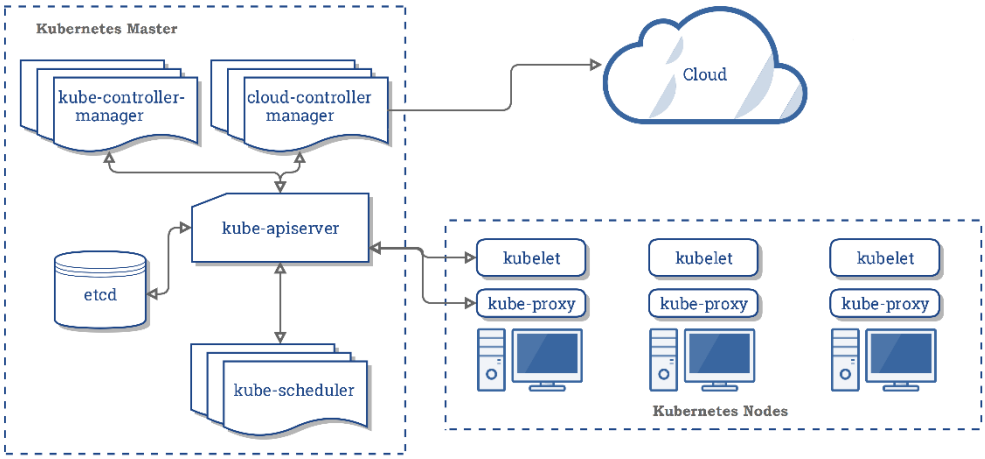
Kube-controller-manager – komponent provádějící hlavní control loop pro Kubernetes. Sleduje stav clusteru pomocí apiserveru a stará se o udržování požadovaného stavu. Mezi jeho úlohy patří například sledovat zdraví jednotlivých nodů, nebo kontrolovat, zda pro danou aplikaci běží správný počet Podů (17). Je vytvořen jako Pod na master nodu.

Kube-proxy – síťová proxy, která běží v Podu na každém nodu v clusteru. Spravuje síťová pravidla na jednotlivých nodech a umožňuje jednotné vystavování Podů jako Service[1].

CoreDNS – DNS server sloužící, jako DNS server pro Kubernetes cluster. Alternativou je kube-dns (18).

Container runtime – pro samotný běh kontejnerů je potřeba, aby na každém nodu byl runtime, který to umožňuje. Pro použití určitého container runtime v Kubernetes musí tento runtime implementovat tzv. Container runtime interface (CRI) (19). Tuto funkci může zastávat například dříve zmiňovaný Docker nebo CRI-O.

Networking – aby spolu mohly jednotlivé Pody v clusteru komunikovat, je potřeba zavést nějaké síťové řešení této komunikace. Problémům s mapováním portů jednotlivých aplikací lze předejít tím, že každý Pod v clusteru má přiřazenou vlastní IP adresu. Přiřazení těchto adres a požadovanou komunikaci mezi Pody zajišťuje network plugin (20).



Obr. 2 Komponenty Kubernetes clusteru a jejich spolupráce (15).

Jak je možné vidět na obr. 2, z výše popsaných komponent běží na všech nodech kubelet, kube-proxy a container runtime. Na master nodu navíc běží kube-controller-manager, kube-apiserver, a kube-scheduler. Cloud-controller-manager je pro tuto práci nepodstatný.

[1] Prostředek Kubernetes zaštiťující množinu Podů pod jedním logickým DNS záznamem. Více informací v kapitole 4.4.1.

4.2 Způsoby použití

Jak je zmíněno v úvodu této práce, Kubernetes je možné provozovat jako vlastní instalaci, nebo použít některé z řešení nabízených cloudovými providery. V této kapitole jsou popsány oba přístupy a základní rozdíly mezi nimi, které je potřeba zvážit při vybírání vhodného řešení.

Výhoda použití Kubernetes obecně je následující: ať zvolíme kterýkoli z možných přístupů, všechny následné konfigurace ohledně nasazování, toolingu apod. budeme schopni přenášet v rámci těchto řešení.

4.2.1 Cloudové řešení

Všichni tři hlavní poskytovatelé cloudových služeb – Google, Microsoft a Amazon – nabízejí spravovaný Kubernetes cluster, tedy Kubernetes as a Service. Uživatelé potřebují pouze specifikovat, na kolika serverech má cluster běžet, případně verzi Kubernetes, a poskytovatel se postará o tvorbu clusteru a dále o veškeré úkony spojené s jeho provozem. Výstupem jsou přístupové údaje ke clusteru a uživatel může rovnou začít cluster používat.

Při tomto řešení se zbavíme nejen povinnosti spravovat cluster, ale také povinnosti spravovat servery, na nichž cluster běží. Dalším plusem tohoto řešení je možnost využití Kubernetes prostředku Service typu LoadBalancer. Všichni uvedení poskytovatelé totiž nabízí vlastní implementace LoadBalanceru, který umožňuje směrovat komunikaci na vytvořený cluster. Uživatel se tedy nemusí zabývat loadbalancingem nebo alokací veřejné IP adresy.

Jednu ze zmíněných výhod lze však považovat i za nevýhodu, a sice ztrátu kontroly nad správou clusteru. Většina zmíněných poskytovatelů nabízí jen několik verzí Kubernetes, které mají otestovány. V případě, že potřebujeme novější/starší verzi, než která je nabízena, musíme volit jiné řešení. Nevýhodou mohou být také vyšší cenové nároky na provoz.

Další výhodou/nevýhodou může být fakt, že při problémech v clusteru je nutné spoléhat na poskytovatele; uživatel totiž nemá možnost do clusteru zasahovat. Z téhož důvodu ovšem uživateli odpadá povinnost provádět údržbu.

4.2.2 On-premise

Alternativou ke cloudovým řešením je provozovat Kubernetes jako vlastní instalaci na svých serverech. Podmínkou však je vlastnictví těchto serverů a následně schopnost vytvořit a spravovat cluster.

Za výhodu tohoto řešení se dá považovat mnohem větší kontrola nad používaným prostředím, a to jak nad servery, na kterých cluster běží, tak nad clusterem samotným. S touto výhodou souvisí nevýhoda cloudového řešení, tedy nemožnost volby libovolné verze Kubernetes. Na rozdíl od cloudového řešení lze při provozu on-premise použít jakoukoli verzi Kubernetes, jelikož instalaci provádí sám uživatel. Toto řešení je také vhodné zejména díky přínosu bezpečnosti, jelikož celé používané prostředí je pod kontrolou organizace.

Nevýhodou tohoto řešení je nemožnost použití LoadBalancer typu Service „out of the box“. Tuto funkcionalitu však lze používat i v on-premise clusterech pomocí nástrojů jako MetalLB (21) nebo je možné vytvořit Service typu NodePort a použít vlastní externí loadbalancer. Nutnost spravovat cluster může být také považována za nevýhodu, poněvadž v případě problémů je vlastník odkázán sám na sebe a musí se s problémy vypořádat bez pomoci expertů.

4.3 Nástroje pro tvorbu

On-premise cluster je možné vytvořit několika různými způsoby. Existuje množství nástrojů zabývajících se touto problematikou. Některé se orientují na větší škálu clusteru a nabízejí tedy možnost tvořit clustery o libovolné velikosti. Jiné se soustředí převážně na tvorbu hlavních prvků clusteru. V této kapitole budou popsány některé z těchto nástrojů, přičemž jeden z nich, Kubeadm, který je používán v této práci, je popsán detailněji.

Kops – kops se prezentuje jako kubectl[1] pro Kubernetes clustery. Jedná se o nástroj poskytující tvorbu Kubernetes clusterů na cloudových platformách AWS, OpenStack a GCE. Vznikl především jako odpověď na dlouho nedostupnou nabídku Kubernetes as a Service v AWS. Jako samotné Kubernetes používá deklarativní konfiguraci a umožňuje tedy tvořit a upravovat clustery a jejich stav pomocí konfiguračních souborů, které specifikují požadovaný stav. (22), (23)

[1] CLI nástroj pro interakci s Kubernetes clusterem.

Kubespray – tento nástroj se zabývá nejen nasazením Kubernetes na libovolné množství serverů, ale také přípravou těchto serverů pro úspěšný běh Kubernetes. Stará se tedy i o instalaci potřebných komponent, jakými jsou například container runtime, network plugin nebo ingress controller. K dosažení této funkcionality používá nástroj Ansible (24).

Ansible je nástroj pro správu a konfiguraci serverů. Jedná se o open-source software poskytovaný firmou RedHat. Jeho velkou předností je to, že ke správě konkrétního serveru potřebuje pouze SSH přístup na daný server a Ansible instalaci na zařízení, z něhož pracujeme. Umožňuje definovat stav, ve kterém má server být – například, že balíček python má být ve stavu nainstalován. Při spuštění vytvořené konfigurace Ansible zkontroluje, zda je tento stav na požadovaných serverech; pokud ne, pokusí se tento stav vytvořit (tedy balíček nainstalovat). Ansible má mnoho modulů umožňujících velkou škálu různých operací. (25)

Ansible představuje několik souborů/prvků potřebných pro konfiguraci:

- Hosts soubor – obsahuje seznam cílových serverů a jejich případné rozdělení.
- Role – seznam příkazů/definicí stavů zvaných Task, které mají být provedeny.
- Handler – úkon definován stejným způsobem jako Task, ale spouštěn jako reakce na události (například změna konfiguračního souboru).
- Playbook – konfigurační soubor definující jaké role mají být spuštěny nad jakými servery. Playbook je přímo spouštěn pomocí ansible příkazu.

Kubeadm – jako hlavní a oficiální alternativa tvorby Kubernetes ručně, tedy manuální instalace i spouštění veškerých prvků clusteru, je doporučován nástroj kubeadm. Kubeadm abstrahuje všechny základní akce související s tvorbou clusteru – jako inicializaci master nodu, připojování worker nodu apod. – do jednoduchých příkazů využívajících kubeadm CLI nástroje. Zároveň umožňuje specifikaci konfiguračního souboru určujícího vlastnosti tvořeného clusteru. Je možné jej použít na jakémkoli linuxovém serveru, který splňuje minimální hardwarové požadavky. Velké množství nástrojů pro práci s Kubernetes clustery interně používá právě Kubeadm. (26)

Hlavní příkazy Kubeadm, které budou relevantní pro tuto práci:

- kubeadm init,

- kubeadm token,
- kubeadm join,
- kubeadm upgrade a
- kubeadm reset.

První z těchto příkazů – kubeadm init – se používá k vytvoření Kubernetes na master nodu. Pod tímto příkazem jsou zahrnuty veškeré kroky vedoucí k vytvoření nutných prvků popsaných v kapitole 4.1. Ty zahrnují

- spuštění kubelet service;
- vytvoření potřebných certifikátů používaných v rámci clusteru a přístupových souborů pro cluster;
- vytvoření statických konfigurací Podů klíčových prvků clusteru jako apiserver, controller-manager, scheduler a etcd;
- instalace add-on komponent coredns a kube-proxy;
- vytvoření bootstrap tokenu pro připojování worker nodů. (27)

Příkaz kubeadm join slouží pro připojení worker nodu k vytvořenému master nodu. Jednotlivé prováděné kroky spočívají v získání potřebných informací od master nodu, dále ve spuštění kubelet service a v zaregistrování nodu do clusteru. Pro spuštění kubeadm join je potřeba mít token zmiňovaný v části o kubeadm init. Tento token je také možné získat pomocí příkazu *kubeadm token create*. (28)

Kubeadm upgrade umožňuje jednoduše aktualizovat verzi Kubernetes v existujícím clusteru. Ten obsahuje několik podpříkazů, přičemž dva důležité jsou *kubeadm upgrade plan* a *kubeadm upgrade apply*. Plan spustí kontrolu verzí, zjistí, zda je vůbec možné upgrade provést, a v případě problémů uživatele informuje. Apply spustí upgrade proces a průběžně uživatele informuje o prováděných akcích. (29)

Kubeadm reset zahrnuje pokus nástroje o rollback provedených kroků a o navrácení serveru do původního stavu před spuštěním kubeadm init nebo kubeadm join. Spočívá v odstranění nodu z clusteru, odstranění etcd a ve vyčištění souborového systému serveru. Některé později aplikované změny, jako úpravy iptables provedené network pluginem, však vráceny nejsou. (30)

Všechny kroky příkazů init, join, reset a upgrade je také možné spouštět jednotlivě pomocí *kubeadm <command> phase*.

4.4 Provoz aplikací

Všechno, co můžeme tvořit v Kubernetes, je definováno pomocí tzv. prostředků (resources). Tyto prostředky mohou být různého typu, přičemž všechny jsou definovány v Kubernetes API. Každý z prostředků má určitá povinná a volitelná pole, která lze použít k jeho konkrétní konfiguraci. Tato konfigurace je tvořena pomocí souborů ve formátu YAML.

Tyto prostředky umožňují definovat vše potřebné k provozu a správě stavu Kubernetes clusteru. Od definicí běžících aplikací, přes RBAC práva po konfigurační objekty a objekty pro uchování senzitivních informací.

Základní prvky, používané pro provoz aplikací, se kterými je vhodné se seznámit, jsou Pod, Deployment, Service, Ingress, ConfigMap, Secret, Namespace, ServiceAccount, Role/RoleBinding, ClusterRole/ClusterRoleBinding.

4.4.1 Prvky Kubernetes

Pod – nejmenší objekt, který je možné vytvořit, nasadit a spravovat. Pod obsahuje, kromě aplikačního kontejneru, také jeho lokální úložiště a konfiguraci stavu. Každému Podu je v Kubernetes přiřazena IP adresa, která je v clusteru jedinečná. Rozsah adres určuje využitý network plugin. Jeden Pod může obsahovat větší množství kontejnerů, které spolu většinou úzce souvisí, případně interagují. Důležité části konfigurace Podu jsou labels, ports a image.

Přímá tvorba Podů se však v praxi objevuje zřídka. A to proto, že Pod nemá v Kubernetes zaručený životní cyklus. To znamená, že pokud běh Podu selže nebo se projeví nedostatek místa na využívaném nodu, daný Pod zanikne a již nebude obnoven. Proto je preferováno použití prvku Deployment.

Deployment – popisuje požadovaný stav nasazované aplikace, přičemž Kubernetes zajišťuje, že je tento stav dodržován. Spolu s prvkem Deployment je vytvořen tzv. ReplicaSet, který se stará o to, aby vždy běžel požadovaný počet Podů. V případě, že Pod z nějakého důvodu zanikne, je díky prvku Deployment vytvořen znovu. V Deploymentu je tedy specifikována konfigurace podobná Podu, soustředící se na to, jaké kontejnery mají být spouštěny. Navíc však obsahuje informace nad rámec jednoho Podu, jako právě počet replik.

Po úspěšném nasazení aplikace pomocí Deploymentu, který vytvoří patřičné Pody budeme chtít s aplikací nějakým způsobem komunikovat. Může jít o komunikaci mezi jednotlivými aplikacemi v clusteru nebo také z vnějšku clusteru. Vzhledem k tomu, že Pod může kdykoliv zaniknout a být vytvořen znovu, není vhodné využít k této komunikaci jeho IP adresu, poněvadž ta se může změnit. Proto využijeme prostředek Service.

Service – je abstrakcí nad Pody a umožňuje k nim jednotný přístup. Service funguje tak, že zaštiťované Pody vystaví pod neměnnou unikátní IP adresou a DNS záznamem. Konkrétní Pody, které mají být vystaveny, najde Service podle tzv. selektoru. Ten nakonfigurujeme tak, aby odpovídal některému z labels v konfiguraci cílových Podů.

Prostředky typu Service se však, s výjimkou typu NodePort, používají převážně pro vystavování a komunikaci služeb uvnitř Kubernetes clusteru. Nenabízí tedy žádnou rozšířenou funkcionalitu, kterou bychom mohli potřebovat při směrování požadavků z vnějšku clusteru, jako specifikaci URL cesty nebo HTTP autentizaci. K těmto účelům se používá Ingress.

Ingress – kolekce pravidel pro zpracovávání příchozích požadavků na cluster. Prostředek Ingress je pouze tato kolekce pravidel a potřebuje mít v clusteru nasazen controller, který bude tato pravidla vykonávat. Populární implementací tohoto controlleru je NGINX Ingress.

Všechny z dosud zmíněných prostředků musí ve vytvářeném clusteru někam patřit. Například prostředky vytvářené pro určité prostředí by bylo vhodné logicky oddělit od prostředků jiného prostředí. K tomu jsou používány namespaces.

Namespace – umožňuje tvořit virtuální oddělené prostředí v rámci jednoho fyzického Kubernetes clusteru. Namespace poskytuje tzv. scope pro názvy vytvářených prostředků. V rámci jednoho Namespace musí být názvy prostředků unikátní. V rozdílných namespaces však mohou být prostředky se stejnými názvy. Prostředky v jednom Namespace o prostředcích jiného Namespace neví. (31)

Kubernetes také zavádí možnosti, jak přistupovat k zabezpečení prvků clusteru a jak kontrolovat oprávnění jednotlivých aplikací nebo uživatelů. Toho dosahuje pomocí modelu RBAC a tvorbou prostředků jako ServiceAccount, Role a RoleBinding (nebo jejich cluster ekvivalenty).

ServiceAccount – reprezentuje identitu přistupující entity a jsou na něj vázána práva přístupu/manipulace s prvky. Každý Namespace obsahuje ServiceAccount default, jež je přiřazován Podům v případě, že není specifikován jiný ServiceAccount. Každý ServiceAccount má přiřazen token, který je možný používat k autentizaci. (32)

Role / ClusterRole – prostředek Kubernetes obsahující seznam práv, která bude mít uživatel/ServiceAccount k dispozici, pokud tuto má tuto roli přidělenou. Pravidla jsou aditivní – neobsahují záznamy zakazující přístup k prvkům – a sestávají ze specifikace

- apiGroup – skupiny Kubernetes API, ve kterém se nachází požadovaný prvek,
- resources – seznam prvků,
- verbs – seznam akcí, které je možné nad prvky provádět.

Role umožnuje specifikovat pravidla v rámci Namespace ve kterém je tvořena. ClusterRole umožňuje navíc přidávat pravidla pro prostředky, které jsou koncipovány v rámci celého clusteru, jako node, Namespace apod. (33)

Vytvoření samotné Role nebo ClusterRole však nikomu tato práva nepřidá. K tomu je potřeba tuto roli někomu přiřadit, a to pomocí prostředku RoleBinding nebo ClusterRoleBinding.

RoleBinding / ClusterRoleBinding – tento prostředek obsahuje subjekty nebo seznam subjektů a referenci na Role / ClusterRole, jež má být těmto subjektům přiřazena. RoleBinding může odkazovat na Role ve svém Namespace. ClusterRoleBinding přidává přístupy v odkazované ClusterRole v rámci celého Kubernetes clusteru. (33)

Secret / ConfigMap – tyto prostředky slouží pro uchovávání různých dat. Secret je převážně používán pro ukládání senzitivních dat, například hesel. ConfigMap umožňuje uchovávání konfigurace, nejlépe textového typu, jako třeba properties. Oba tyto prvky mohou být následně připojeny k Podu pro čtení těchto informací.

4.4.2 Helm

Definovat veškerou konfiguraci manuálně, pomocí prvků popsaných v předchozí kapitole, se však při nasazování komplikovanějších aplikací stává neúnosným. Zároveň vyvstává problém, jak dynamicky měnit konfigurace nasazovaných aplikací, aniž bychom

vždy museli měnit tvořené konfigurační soubory. Nejen tento problém se snaží řešit nástroj pro nasazování a „packaging“ Kubernetes aplikací, Helm.

Helm by se dal považovat za něco jako yum[1] pro Kubernetes. S jeho pomocí lze vytvářet tzv. charts, které definují potřebné konfigurační YAML soubory používané pro nasazení v Kubernetes. Umožňují také mít v těchto souborech proměnné, jejichž hodnoty je – při vytváření chart v clusteru – možné doplnit do tzv. release (release se nazývá výsledná kolekce konfigurací, která vznikne použitím chart). Tyto proměnné je možné specifikovat buď přímo v helm příkazu nebo pomocí konfiguračního souboru (ve formátu YAML), který hodnoty obsahuje. Kromě proměnných je navíc možné používat a tvořit různé vestavěné funkce pomocí šablon programovacího jazyka Go. (34)

Vzhledem k tomu, že se většina aplikací skládá z prostředků Deployment, Service a Ingress, nasazování klasických aplikací je zredukováno na poskytnutí souboru s požadovanými hodnotami používané chart. Díky tomu je instalace konkrétních nástrojů ulehčena – pokud tvůrce poskytuje Helm chart, stačí pro nasazení najít tuto chart, zjistit jaké hodnoty můžeme nakonfigurovat a použít ji k nasazení. Oficiální GitHub repozitář (35) obsahuje Helm chart pro velké množství populárních aplikací.

V době, kdy vznikala tato práce, došlo v Helm projektu k významnému pokroku. Původní verze Helm, známá buď jako Helm nebo Helm 2 vznikla v době, kdy Kubernetes ještě neumožňovalo tvorbu CRD[2] a nebyl uveden princip tzv. operátorů. Tato verze Helmu zaváděla do Kubernetes komponent zvaný Tiller, který spravoval nasazované aplikace a staral se o práci s nimi.

S postupem času bylo zřejmé, že tento přístup není ideální – Tiller potřeboval mít kontrolu nad Kubernetes clusterem, což může být považováno za bezpečnostní riziko. Nasazované charty zároveň nedodržovaly Kubernetes princip namespaces, a nebylo tedy možné nasadit například stejně pojmenovanou aplikaci ve dvou různých namespaces. Tyto problémy a mnoho dalších vedly k návrhu a k následné implementaci Helm 3.

Helm 3 řeší většinu zmiňovaných problémů. Nová architektura kompletně odstranila Tiller komponent. Pracuje pouze pomocí uchovávání informací o nasazení v prvcích

[1] CLI nástroj pro správu balíčků (programů/aplikací) v linuxových distribucích používajících RPM balíčky.
[2] Custom Resource Definition. Kubernetes umožňuje, kromě klasických prostředků, z nichž některé jsou popsány v kapitole 4.4.1, vytvářet vlastnoručně definované prostředky, tzv. CRD.

clusteru, přičemž interakce přes CLI program helm probíhá prací s nimi. Vzhledem k tomu, že jsou tyto informace uchovávány v Kubernetes prostředcích, veškeré nasazené charty nyní patří do konkrétního Namespace, a je tedy možné nasazovat v různých namespaces naprosto nezávisle.

Jak s Helm 2, tak s Helm 3 pracujeme pomocí CLI nástroje helm. Některé příkazy byly při aktualizaci na Helm 3 odstraněny a chování některých příkazů bylo pozměněno. Hlavním rozdílem je však samozřejmě ten, že Helm 3 již nepracuje s Tillerem. Klíčovými příkazy pro práci jsou tyto:

- helm create <name> – vytvoří složku zdrojového kódu pro novou chart
- helm package <directory> – vytvoří chart archiv ze složky se zdrojovým kódem
- helm init – příkaz Helm 2 sloužící k vytvoření Tiller komponentu
- helm upgrade <releaseName> – aktualizace release s novými parametry
- helm uninstall <releaseName> – odstranění dané release
- helm rollback – navrácení release k předchozí verzi
- helm list – zobrazení existujících release (v Helm 3 v daném Namespace)
- helm repo – práce s chart repozitáři (přidání, odebrání, aktualizace…)

Ve výše zmiňovaném GitHub repozitáři jsou uchovávány pouze zdrojové kódy Helm chart, jejichž nasaditelné verze je možné vytvářet použitím příkazu helm package. Archiv tgz tvořený tímto příkazem je poté možné distribuovat do Helm chart repozitáře. Uživatel si může následně repozitář lokálně přidat pomocí helm repo add a používat v něm uchovávané Helm charty. (36)

4.4.3 Weave Flux

Při adaptaci kontejnerizace a Kubernetes však vyvstane citelný problém ve tvořené pipeline, a sice jak se dostat od vytvoření Docker image k nasazení do Kubernetes. Dosud zmiňované přístupy k provozu a k nasazování aplikací vždy zahrnovaly manuální interakci s Kubernetes clusterem pomocí příkazů kubectl nebo helm. Zároveň nebyla nijak řešena problematika uchovávání veškerých konfigurací, ať už základních Kubernetes manifestů, nebo souborů s hodnotami pro Helm charty.

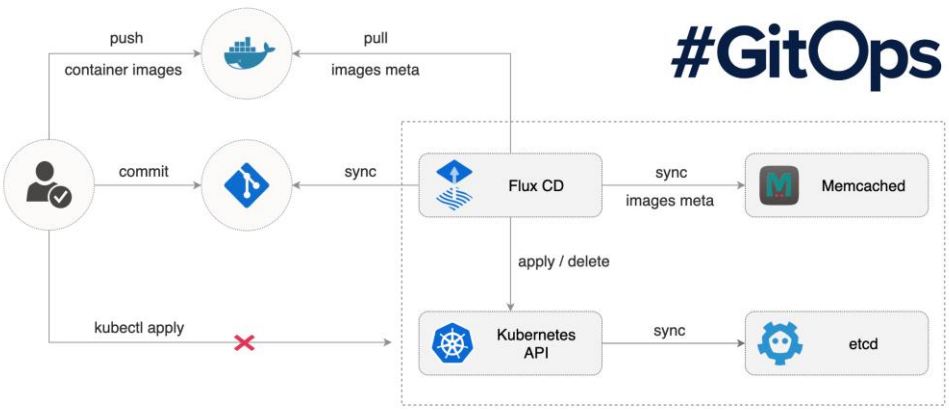
K tomuto problému je možné přistupovat různými způsoby. Ať už udržováním konfigurace v repozitáři aplikace a ručním nasazováním, což však není ideální kvůli nutnosti manuálního aplikování konfigurace. Nebo nasazením přímo z CI/CD software jako Jenkins, přičemž nevýhodou zde může být roztroušení konfigurace v jednotlivých projektech.

Touto problematikou se zabývá tzv. GitOps přístup. Ten spočívá v udržování konfigurace všeho, co má běžet v Kubernetes, v Git repozitáři, přičemž v Kubernetes je nasazen komponent sledující prováděné změny, které automaticky aplikuje. Hlavním představitelem tohoto přístupu je nástroj Weave Flux.

Weave Flux představuje několik komponentů, jejichž nasazením do Kubernetes clusteru získáme výše zmiňovanou funkcionalitu. Hlavní prvek – Flux – se stará o sledování změn v připojeném Git repozitáři a v případě zaznamenání změn aktualizuje svou verzi konfigurace a začne tyto změny aplikovat na cluster.

Konfigurace v Git repozitáři může obsahovat jak klasické Kubernetes manifesty, tak i konfigurace pro Helm charty. Nasazování Helm chart pomocí Flux je dosaženo vytvořením CRD zvaného HelmRelease. V tomto CRD je možné specifikovat zdroj chart, hodnoty, které do ní mají být dosazovány, a image, která má být použita. (37)

Tím se dostáváme k další z funkcí Weave Flux, a to k možnosti sledovat registry nasazených Docker image a – v případě, že je tato funkcionalita ve Fluxu nakonfigurována – stahovat nové verze image automaticky po jejich nahrání do registry. V konfiguraci HelmRelease lze nakonfigurovat pattern pro tagy nových Docker image a v případě zaregistrování novějšího tagu spadajícího do nastavené pattern tuto image použít a automaticky aktualizovat manifest v Git repozitáři s tímto tagem.



Obr. 3 Diagram Flux architektury (37).

5. Tvorba clusteru

Jak bylo zmíněno v předchozí kapitole, k tvorbě on-premise Kubernetes clusteru byl použitý nástroj pro bootstrap Kubernetes zvaný Kubeadm. Cluster byl vytvářen na třech virtuálních serverech běžících na fyzických serverech firmy Generali Česká pojišťovna a.s. Jeden z těchto serverů bude sloužit jako master node a zbylé dva jako worker node. Na serverech běžel operační systém CentOS 7, přičemž v průběhu tvorby této práce proběhl upgrade na verzi 8. V průběhu této práce byly používány verze Kubernetes 1.11 až 1.17.

Proces tvorby clusteru však, bohužel, nespočívá pouze v instalaci kubeadm CLI nástroje a ve spuštění příkazu kubeadm init. Servery je nutné připravit pro instalaci nainstalováním potřebných závislostí, container runtime, přidáním konfigurací apod. Zároveň do procesu zasahují určitá omezení či překážky vyvstávající z operování v interním prostředí firmy, jako nutnost pracovat s proxy či s interními repozitáři aplikací.

V této kapitole budou popsány kroky, které bylo nutné v tomto popisovaném prostředí provést pro úspěšné vytvoření clusteru o třech nodech. V případě instalace v méně omezeném prostředí mohou být kroky týkající se konfigurace proxy nebo privátních repozitářů vypuštěny.

Seznam úkonů, které je nutné provést na každém ze serverů je následující:

- Nastavit použití proxy serveru a nainstalovat potřebné CA certifikáty (specifické pro firemní prostředí).
- Nainstalovat container runtime implementující CRI – v této práci byl použitý Docker.
    - Nakonfigurovat Docker pro používání proxy a interního registry pro Docker image (specifické pro firemní prostředí).
- Přidat Kubernetes repozitář pro yum, nainstalovat nástroje pro tvorbu a práci s Kubernetes a provést konfiguraci nutnou pro běh Kubernetes.
- Vytvořit konfiguraci pro Kubeadm podle potřeb a provést inicializaci Master nodu a připojení Worker nodu.

5.1 Konfigurace proxy

Jelikož se používané servery nachází ve firemní síti, potřebují pro přístup k internetovým zdrojům správně nakonfigurovanou proxy. Proto je před prováděním příkazů

vhodné nastavit proměnné systému HTTP_PROXY, HTTPS_PROXY a NO_PROXY. Jako server je nastaven patřičný firemní proxy server a do proměnné NO_PROXY, která specifikuje adresy, pro které proxy nechceme používat, je nutné přidat IP adresy jednotlivých serverů, serverů v interní síti a rozsah IP adres používaného network pluginu.

Pro správné fungování proxy připojení je navíc potřeba na servery nainstalovat root CA certifikát. Ten je vložen do /etc/pki/ca-trust/source/anchors, načež je zaregistrován pomocí příkazu update-ca-trust.

5.2 Instalace a konfigurace Docker

Instalace nástroje Docker se liší v závislosti na používaném prostředí. Obecně instalace spočívá v přidání yum repozitáře umožňujícím ji provést. V případě existence interních firemních yum repozitářů je potřeba přidat některý z těchto repozitářů; pokud tyto repozitáře neexistují, je možné použít veřejný repozitář.

Do složky /etc/yum.repos.d/ přidáme soubor Docker.repo, obsahující konfiguraci pro potřebný repozitář. V případě veřejného repozitáře poskytovaného Dockerem je nutné nalézt požadovanou verzi, například https://download.docker.com/linux/centos/7/x86_64/stable/.

Ve firemním prostředí je potřeba navíc opět nakonfigurovat proxy. V tomto případě se jedná o proxy pro Docker daemon. Tu je možné nastavit za pomoci systemd drop-in[1] konfiguračního souboru. Ve složce /etc/systemd/system/docker.service.d/ vytvoříme soubor http-proxy.conf, ve kterém pod sekcí [Service] přidáme Environment proxy hodnoty nastavované výše.

Zároveň je vhodné nakonfigurovat daemon samotný. Ten v tomto případě konfigurujeme přidáním interních / insecure registry pro Docker image, úpravou runtime parametrů a různých „quality of life“ změn.

Daemon je konfigurován přidáním souboru daemon.json do /etc/docker/. Ve vytvořeném souboru poté pomocí klíče „insecure-registries“ specifikujeme seznam interních registry, které budeme chtít používat. Klíč „exec-opts“ umožňuje upravit runtime parametry,

[1] Systemd jednotky lze konfigurovat pomocí drop-in souborů nacházejících se ve složce name.service.d s příponou conf. Soubory musí být ve správném formátu. (38)

ten použijeme pro nastavení cgroup driveru na hodnotu systemd[1]. Jako poslední může být vhodné specifikovat parametr „graph“. Ten určuje umístění v systému, kam bude Docker ukládat data. Výchozí /var/lib/docker/ totiž nemusí být vhodné, například z důvodu nedostatku místa.

Po provedení výše zmíněných kroků je potřeba znovu načíst konfiguraci pro Docker service pomocí *systemctl daemon-reload,* restartovat Docker pomocí *systemctl restart docker* a, v případě, že jsme přidali nějaké registry, se do nich přihlásit pomocí *docker login <registry-url>*.

5.3 Instalace Kubernetes balíčků a potřebná konfigurace

Podobně jako při instalaci Dockeru je před instalací Kubernetes nástrojů nejdříve nutno přidat extra yum repozitář, ze kterého budou tyto nástroje staženy. V případě Kubernetes se opět do /etc/yum.repos.d/ přidá soubor Kubernetes.repo s odkazem na veřejný Kubernetes repozitář a gpgkey[2].

Následně je možné nainstalovat potřebné Kubernetes nástroje, tedy kubelet, kubeadm a kubectl pomocí příkazu *yum install kubelet kubeadm kubectl*.

Před samotnou tvorbou Kubernetes clusteru je ještě navíc potřeba provést určité konfigurační změny. Konkrétně jde o vypnutí SELinux a swapu, nastavení používání iptables na zpracovávanou síťovou komunikaci a o zapnutí IP forwardingu.

Zpracování síťové komunikace pomocí iptables opět nakonfigurujeme pomocí systemd konfiguračního souboru. Do /etc/sysctl.d/ přidáme soubor nastavující hodnotu net.bridge.bridge-nf-call-iptables a její IPv6 ekvivalent (ip6tables) na 1.

SELinux je možné vypnout buď dočasně pomocí příkazu *setenforce 0*, nebo trvale upravením hodnoty SELINUX v /etc/sysconfig/selinux na disabled. Swap je možné vypnout pomocí swapoff -a. IP forwarding zapneme nastavením hodnoty 1 v souboru /proc/sys/net/ipv4/ip_forward.

[1] Cgroups jsou jednou z klíčových funkcionalit pro běh kontejnerů. Na systémech se systemd je proto vhodné nakonfigurovat container runtime (v tomto případě Docker) pro jeho použití. (19)
[2] GPG klíče jsou používány pro ověření podpisů instalovaných balíčků ze vzdálených repozitářů (39).

Posledním krokem, který je nutné provést, je aktivace kubelet service. Toho je dosaženo příkazem *systemctl enable kubelet*.

5.4 Inicializace clusteru

Po provedení veškerých konfiguračních kroků je konečně možné vytvořit cluster jako takový. Jak bylo zmíněno v kapitole 4.3, s pomocí Kubeadm je cluster možné vytvořit příkazem kubeadm init. Tomu je možné navíc specifikovat konfigurační soubor upravující vlastnosti vytvářeného clusteru příznakem *--config*.

Tento soubor je vhodné vytvořit vzhledem k tomu, že minimálně potřebujeme specifikovat rozsah IP adres pro vytvářené Pody. Rozsah zvolíme podle network pluginu, který máme v plánu používat. V této práci byly vyzkoušeny pluginy Calico a Weave Net.

Konfigurační soubor pro Kubeadm se definuje, stejně jako klasické Kubernetes prostředky, pomocí YAML souboru. V tomto souboru tedy nastavíme požadované vlastnosti clusteru. Ačkoli konfigurační soubor poskytuje širokou škálu konfigurovatelných vlastností, v našem případě využijeme pouze možnosti konfigurace několika vlastností Kubelet service a clusteru samotného.

Konkrétně měníme výchozí umístění, ve kterém Kubelet uchovává data, a tzv. eviction hranici, při které vykazuje workload z nodu. Z vlastností clusteru poté nastavujeme podSubnet, tedy rozsah IP adres, které budou použité pro adresaci Podů. Dále také měníme rozsah portů, které lze vystavit pomocí NodePort service. Jelikož je cluster tvořen pro testovací účely a nebude vystaven za loadbalancerem, chceme mít možnost vystavovat aplikace i na klasických portech jako 80.

Jako poslední část konfigurace chceme zapnout tzv. admission plugin PodPreset. Pomocí tohoto pluginu budeme schopni upravovat konfiguraci Podů před jejich nasazením. Tato vlastnost se hodí pro automatické nastavování proměnných prostředí, specifikujících nastavení proxy.

```
apiVersion: kubeadm.k8s.io/v1beta2
kind: InitConfiguration
nodeRegistration:
  kubeletExtraArgs:
    root-dir: "/data/kubelet"
---
apiVersion: kubeadm.k8s.io/v1beta2
kind: ClusterConfiguration
networking:
  podSubnet: 192.168.0.0/16
apiServer:
  extraArgs:
    service-node-port-range: 0-31000
    enable-admission-plugins: PodPreset
    runtime-config: settings.k8s.io/v1alpha1=true
---
apiVersion: kubelet.config.k8s.io/v1beta1
kind: KubeletConfiguration
cgroupDriver: systemd
evictionHard:
  memory.available: "400M"
```

Obr. 4 Kubeadm konfigurace (Zdroj: autor).

Po vytvoření tohoto konfiguračního souboru a po provedení dříve zmíněných úkonů na všech severech, které plánujeme v clusteru mít, můžeme inicializovat master node pomocí *kubeadm init --config config.yaml*. Kubeadm provede potřebné kroky pro vytvoření master nodu a na konci výpisu zobrazí příkaz pro připojení dalších nodů do clusteru. Tento příkaz spustíme na zbylých worker node serverech a připojíme je tím do clusteru.

Úspěšné vytvoření clusteru lze následně otestovat pomocí CLI nástroje kubectl. Ze složky /etc/kubernetec/admin.conf na master node získáme kubectl config. Ten můžeme zkopírovat a na našem systému uložit do ~/.kube/config. V případě, že máme síťový přístup na master node, měli bychom pomocí kubectl get nodes vidět všechny naše servery zaregistrované jako nody clusteru.

Posledním krokem, souvisejícím se zprovozňováním samotného clusteru, je instalace network pluginu zmiňovaného výše. Jeho instalaci provedeme pomocí příkazu kubectl, odkazujícího na konfigurační soubory poskytnuté dodavatelem vybraného pluginu. V případě Weave Net lze plugin nainstalovat s pomocí příkazu uvedeného na stránce projektu (40), který odkazuje na potřebnou konfiguraci.

5.5 Automatizace úkonů s Ansible

Většinu kroků popsaných v této kapitole je nutné provést na všech serverech, což může být zdlouhavé. Může se stát, že cluster potřebujeme z nějakého důvodu zničit a vytvořit ho znovu, případně jej vytvořit na jiných serverech. Z těchto důvodů je vhodné tyto úlohy nějakým způsobem zautomatizovat. Ideálním nástrojem pro tuto automatizaci je Ansible, jež je blíže popsán v kapitole 4.3.

Pro provedení výše popsaných kroků tedy potřebujeme vytvořit Ansible konfiguraci, která za pomocí Ansible modulů zajistí, že stav systémů bude odpovídat našemu popisu. Pro přehlednost konfigurace logicky oddělíme vytvářené role podle toho, co mají na starosti. V podadresáři roles tedy vznikne osm dalších adresářů, a to

- Prerequisities
- Python
- Docker
- Kubernetes
- Kubeadm-master
- Kubeadm-master-jointoken
- Kubeadm-worker-join
- Reset

Pomocí těchto rolí budeme schopni vytvořit playbooky pro základní úkony, které budeme chtít nad servery provádět. Jeden playbook pro kompletní inicializaci clusteru, tedy instalaci závislostí, inicializaci master nodu a připojení worker nodů. Druhý playbook bude sloužit pro připojení nového worker nodu do clusteru. A třetí pro reset clusteru.

Kompletní konfigurace jednotlivých playbooků spolu se všemi rolemi, tasky a potřebnými soubory je v elektronické příloze této práce. V následujících odstavcích budou popsány pouze určité relevantní části.

Role Prerequisities a Reset patří mezi tasky s nejmenším množstvím příkazů. Prerequsities je potřeba pouze při instalaci na privátní prostředí a jejím úkolem je na server nainstalovat certifikát pro správné fungování proxy. Role reset naopak slouží k zastavení Kubernetes na cílovém serveru.

Adresář Python obsahuje tasky zajišťující existenci potřebných Python balíčků na cílových systémech. Tyto balíčky je potřeba nainstalovat, přestože nejsou zmiňovány výše, a to proto, že funkcionalita dále používaných modulů pracujících s Dockerem je na těchto Python knihovnách závislá. Pro jejich instalaci je navíc nutné přidat na server extras yum registry EPEL.

Další z rolí je Docker. Ta se bude starat o veškerou konfiguraci s Dockerem související. Konkrétně tedy obsahuje úlohy zahrnující instalaci potřebných závislostí, přidání yum registrů, instalaci samotného Dockeru a přidání jeho konfiguračních souborů apod. Obecně se tedy tato role stará o provedení veškerých kroků popsaných v kapitole 5.2 a také o některé extra konfigurovatelné kroky v závislosti na instalovaném prostředí (firemní / klasické). Tyto kroky jsou opatřeny podmínkou, jež podle proměnné specifikované na úrovni playbooku zkontroluje, zda má být task proveden.

```
- name: Add Docker repo
  yum_repository:
    name: Docker
    description: Docker repository
    baseurl: https://download.docker.com/linux/centos/7/x86_64/stable/
    gpgcheck: no
    enabled: yes
  when: target_env == "public"
```

Obr. 5: Využití yum Ansible modulu pro přidání Docker registry (Zdroj: autor).

V roli Docker je navíc definován handler, který se stará o načtení konfigurace a o restart service Docker. Handler je notifikován v případě, že se změní používané konfigurační soubory.

Kubernetes role se stará o instalaci nástrojů Kubernetes a o nastavení vyžadované konfigurace. Opět tedy přehledně popisuje a rozděluje jednotlivé kroky zmiňované v kapitole 5.3.

```
- name: Copy Docker proxy config
  copy:
    src: ../files/http-proxy.conf
    dest: /etc/systemd/system/docker.service.d/http-proxy.conf
  notify:
    - restart docker
  when: target_env == "private"
```

Obr. 6: Přidání proxy konfigurace a notifikace handleru (Zdroj: autor).

Role kubedam-master má na starosti samotné vytvoření clusteru, resp. inicializaci master nodu. Jak je popsáno v kapitole 5.4, to se děje pomocí příkazu kubeadm init. Před provedením tohoto příkazu však task zkopíruje konfigurační soubor pro kubeadm a zajistí, že potřebné porty jsou na serveru otevřené. Příkaz kubeadm init se aktivuje pouze tehdy, když cluster doposud neběží.

Po úspěšném vytvoření clusteru (nebo po kontrole, která potvrdí, že cluster běží) získáme pomocí kubeadm příkaz pro připojování nových nodů. Role kubeadm-worker-join příkaz převezme a připojí worker node servery ke clusteru.

Zároveň byla vytvořena role kubeadm-master-jointoken, která může být použita pouze pro získání tohoto join tokenu a následně uplatněna k připojení nového worker nodu. Tato role se používá v druhém z vytvořených playbooků pro připojování nových nodů.

V souboru hosts jsou následně definovány jednotlivé servery, na které chceme Ansible cílit. Ty rozdělíme na skupiny master a workers.

Jak již bylo zmíněno, průběh všech těchto rolí a definování, na kterých serverech mají být spuštěny, se konfiguruje v playbooku.

První playbook spouští společné role – Prerequisities, Python, Docker, Kubernetes – na všech serverech pomocí hosts: all. Role pro inicializaci clusteru je určena pouze pro master pomocí *hosts: master* a role připojení worker nodů pro workery pomocí *hosts: workers*.

```
- hosts: all
  become: true
  roles:
    - prerequisities
    - python
    - docker
    - kubernetes

- hosts: master
  become: true
  roles:
    - kubeadm-master

- hosts: workers
  become: true
  roles:
    - kubeadm-worker-join
```

Obr. 7 Playbook pro konfiguraci nodů, inicializaci master node a připojení worker nodů (Zdroj: autor).

Ve firemním prostředí je zároveň možné příhodně nakonfigurovat proxy pomocí proměnných prostředí, a to přímo v definici playbooku pomocí klíče environment. Tyto hodnoty budou poté použity pro všechny příkazy, které bude Ansible na cílovém serveru provádět. Konkrétní hodnoty těchto serverů však nebudou v této práci uváděny.

Na některých zobrazených částech konfigurací tasků můžeme vidět podmínky pracující s hodnotami, které jsou používané pro rozlišení různých typů nasazování (privátní/firemní prostředí nebo veřejné prostředí). Tyto hodnoty je možné nastavit na několika místech. My však chceme většinu těchto hodnot používat ve větším množství tasků spouštěných na více hostech. Proto je vhodné tyto proměnné definovat plošně pro celý playbook.

Toho dosáhneme vytvořením složky group_vars obsahující soubor all.yml. V tomto souboru definujeme potřebné hodnoty, které budou ovlivňovat chování Ansible, jako target_env nebo epel_repo.

Druhý playbook slouží k nainstalování potřebných závislostí na novém worker serveru a k jeho připojení do clusteru. Zde opět použijeme role Prerequisities, Python, Docker a Kubernetes pro instalaci závislostí a v případě privátního prostředí můžeme opět použít konfiguraci proxy z příkladu výše. Následně použijeme kubeadm-master-jointoken a kubeadm-worker-join.

```
- hosts: workers
  become: true
  roles:
    - python
    - docker
    - kubernetes

- hosts: master
  become: true
  roles:
    - kubeadm-master-jointoken

- hosts: workers
  become: true
  roles:
    - kubeadm-worker-join
```

Obr. 8: Playbook pro konfiguraci a připojení nové worker node (Zdroj: autor).

6. Podpůrné aplikace

Do clusteru vytvořeného v tomto stavu už bychom teoreticky mohli začít nasazovat kontejnerizované aplikace. Nebudeme však schopni aplikace vystavovat pomocí Ingress pravidel, a to vzhledem k potřebě Ingress controlleru zmiňovaném v kapitole 4.4.1, který by tato pravidla zpracovával.

Za nedostatek by se také dalo považovat to, že nemáme příliš velký přehled o stavu našeho clusteru. Jediné, co můžeme v tuto chvíli dělat je sledovat nasazované prostředky přímo pomocí kubectl. Nemáme však například žádný mechanismus, který by nás informoval o chybujících Podech.

Mít příkaz kubectl jako jedinou možnost interakce může být nevýhodné také v případě, že potřebujeme, aby s clusterem pracoval někdo, kdo kubectl neovládá. Je tedy vhodné nabídnout i možnost interakce pomocí grafického rozhraní.

V této kapitole bude popsána tvorba Ingress controlleru a různých nástrojů, jejichž instalace je vhodná pro úspěšnou správu Kubernetes clusteru, a dále také způsoby jejich nasazení.

6.1 NGINX Ingress controller

Jako první je vhodné provést instalaci Ingress controlleru, jelikož s jeho pomocí budeme následně vystavovat veškeré aplikace, které mají být dostupné i mimo Kubernetes cluster. Jednou z nejpopulárnějších implementací Ingress controlleru je NGINX Ingress (41), který pro zpracovávání příchozích požadavků na cluster používá NGINX web server. Princip tohoto prvku Kubernetes spočívá v jeho vystavení vně clusteru a směrování veškeré komunikace skrz něj.

K tomuto vystavení je potřeba nějaký typ Service umožňující příjem komunikace z vnějšku clusteru. Jelikož se tato práce zabývá nasazením on-premise, nemáme k dispozici Service typu LoadBalancer, jako by tomu bylo při nasazení u některého z cloud providerů. Musíme tedy použít typ Service NodePort. Ten vystaví zaštiťovanou aplikaci na všech nodech clusteru na daném portu.

Vzhledem k testovací povaze vytvářeného clusteru bude tento přístup postačující. Pokud by bylo potřeba zefektivnit přístup k nasazovaným službám nebo případně provádět různé

zásahy do směrování, je vhodné použít externí loadbalancer jako třeba NetScaler, který by teprve přeposílal požadavky na vystavený Ingress controller v Kubernetes.

S nasazením Ingress controlleru zároveň souvisí Kubeadm konfigurace použitá při tvorbě clusteru. V té je možné definovat rozsah portů dostupných pro vystavení pomocí NodePort typu Service. V případě, že jsme tento rozsah při tvorbě clusteru nespecifikovali, musíme tyto porty vybírat z rozsahu 30000- 32767 (42). Pokud jsme však rozsah zadali a zahrnuli do něj menší hodnoty, můžeme použít klasické porty 80 pro HTTP a 443 pro HTTPS.

K nasazení NGINX Ingress controlleru je dostupná Helm chart v komunitou spravovaném repozitáři, obsahujícím Helm charty pro velké množství populárních aplikací (43). S pomocí této Helm chart je možné controller jednoduše vytvořit i nakonfigurovat. Při spouštění nasazení pomocí Helm poté pouze specifikujeme soubor, obsahující požadovanou konfiguraci.

V této konfiguraci nastavíme zmiňovaný typ Service, který má být použitý pro vystavení – NodePort – a porty pro HTTP a HTTPS. Zároveň specifikujeme požadavek, aby controller běžel jako DaemonSet – jedna instance Podu na každém nodu – a přidáme vystavení metrik pro Prometheus, jenž bude popsán dále.

```
controller:
    stats:
      enabled: true
    metrics:
      enabled: true
      service:
        annotations:
          prometheus.io/scrape: "true"
          prometheus.io/port: "10254"
    kind: DaemonSet
    service:
      type: NodePort
      nodePorts:
        http: 80
        https: 443
```

Obr. 9 Konfigurace pro Nginx-ingress Helm chart (Zdroj: autor).

6.2 Kubernetes Dashboard

Jedním z nástrojů, které nám umožní získat přehled nad clusterem bez znalosti CLI nástrojů jako kubectl nebo docker, je Kubernetes Dashboard. Dashboard je oficiální webové

rozhraní pro Kubernetes cluster, v němž je možné vidět všechny prostředky, které v clusteru běží, a také jejich stavy. Zároveň je, v případě dostatečných práv, možné tyto prvky upravovat, mazat, přidávat apod. Pro nastavení těchto práv i samotného přístupu do Dashboard je potřeba další konfigurace, která bude popsána v této kapitole.

Kubernetes Dashboard je opět možné nasadit pomocí Helm chart, avšak vzhledem k on-premise způsobu nasazování a k omezeným možnostem konfigurace Ingress pomocí zmiňované Helm chart byla zvolena statická konfigurace, převzatá přímo z GitHub repozitáře projektu Kubernetes Dashboard (44). Nad ní byly následně provedeny určité úpravy a přidána konfigurace zajišťující tvorbu Ingress.

Dashboard je velmi cenným nástrojem pro vizualizaci stavu clusteru. V kombinaci s nástrojem pro sběr základních metrik využití zdrojů, jako např. zátěž CPU nebo využití paměti, je navíc možné, aby Kubernetes Dashboard poskytovala i základní grafické prvky nad těmito daty. O agregaci a poskytnutí těchto dat se nově stará nástroj metrics-server, dříve Heapster (45), přičemž Kubernetes Dashboard podporuje metrics-server až od své druhé verze.

Jak pro metrics-server, tak pro Heapster také existuje Helm chart. Vzhledem k tomu, že deployment konfigurace těchto služeb nevyžaduje přílišnou komplexitu, je možné danou chart jednoduše použít k instalaci tohoto nástroje do našeho clusteru.

Po jeho instalaci Heapsteru/metrics-serveru, podle verze naší Dashboard, budou v Kubernetes Dashboard dostupné grafy, zobrazující vytížení a využití paměti jednotlivými aplikacemi. Zároveň bude rozšířena i funkcionalita základního CLI nástroje kubectl, s nímž bude nyní možné volat kubectl top příkazy nad Pody/Nody k získání relevantních informací o využití CPU a paměti.

V případě, že všechno proběhlo úspěšně, máme nyní nasazený Kubernetes Dashboard spolu s metrics-serverem/Heapsterem pro sbírání základních metrik. Po navigaci na vystavenou URL však narazíme na přihlašovací obrazovku Kubernetes Dashboard, vyžadující kubeconfig nebo access token pro přihlášení. S tímto přihlášením také souvisí přístupová práva ke zdrojům zmiňovaná výše.

Nejpřímočařejším způsobem, jak zajistit přístup k Dashboard je pomocí access tokenu. Ten získáme vytvořením prostředku ServiceAccount, pro něhož následně specifikujeme

potřebné prostředky Role/RoleBinding, popř. jejich Cluster ekvivalenty (pro přístupy, které je potřeba definovat na úrovni celého clusteru). Vytvoření ServiceAccount je přímočaré.

```
apiVersion: v1
kind: ServiceAccount
metadata:
 name: dashboarduser
 namespace: default
```

Obr. 10: Konfigurace ServiceAccount vytvářející uživatele Dashboard (Zdroj: autor).

Při tvorbě těchto prostředků záleží především na tom, jaké pravomoci chceme poskytnout uživateli, který bude vytvořený access token používat. Nejběžněji budeme chtít vytvořit ServiceAccount s přístupem k jednomu konkrétnímu Namespace, ve kterém bude mít všechna práva. Toho dosáhneme tak, že jej (spolu s prostředkem Role) vytvoříme v tomto požadovaném Namespace. V prostředku Role poté v části konfigurace pravidel povolíme přístup ke všem prostředkům všemi možnými akcemi.

```
kind: Role
apiVersion: rbac.authorization.k8s.io/v1beta1
metadata:
  name: dashboarduser-role
  namespace: target-namespace
rules:
- apiGroups: ["*"]
  resources: ["*"]
  verbs: ["*"]
```

Obr. 11: Konfigurace Role pro uživatele Dashboard (Zdroj: autor).

Roli následně přiřadíme našemu ServiceAccount pomocí prostředku RoleBinding, který taktéž umístíme do cílového Namespace, a specifikujeme ServiceAccount jako subject a odkaz na vytvořenou Role.

```
kind: RoleBinding
apiVersion: rbac.authorization.k8s.io/v1beta1
metadata:
  name: dashboarduser-rolebinding
  namespace: target-namespace
subjects:
- kind: ServiceAccount
  name: dashboarduser
  namespace: default
roleRef:
  kind: Role
  name: dashboarduser-role
  apiGroup: rbac.authorization.k8s.io
```

Obr. 12: Konfigurace RoleBinding pro uživatele Dashboard (Zdroj: autor).

Zároveň je vhodné vytvořit ClusterRole povolující zobrazování prostředků Namespace, aby se uživatel mohl v Dashboard proklikat ke svému cílovému Namespace (nebo je možné uživatelům poskytnout odkaz se specifikovaným Namespace v URL). Takováto ClusterRole by v poli resources měla pouze namespaces a v poli verbs pouze list. Přiřazení role by poté bylo provedeno pomocí ClusterRoleBinding, přičemž postup by byl stejný jako při RoleBinding, až na ten rozdíl, že jako roleRef bychom odkazovali na vytvořenou ClusterRole.

Nyní je vše připraveno pro úspěšné přihlášení do Kubernetes Dashboard. Pomocí kubectl zobrazíme prostředky Secret vytvořené v Namespace, v němž se nachází náš ServiceAccount, a najdeme jemu náležící token. Obsah tohoto Secret následně získáme pomocí *kubectl get secret -n target-namespace <token-name> -o yaml* a zkopírujeme část výstupu z klíče token.

Tuto zkopírovanou část ještě nelze použít pro přístup, jelikož je zakódovaná pomocí base64. K jejímu dekódování lze použít CLI utilitu base64 s příznakem **-d** (decode). Buď zkopírovaný token uložíme do souboru a příkaz provedeme nad ním, nebo jej vypíšeme a následně zašleme do base64 příkazu.

6.3 Prometheus

Nasazením Kubernetes Dashboard získáme uživatelsky přívětivé zobrazení stavu clusteru. S kombinací metrics-serveru nebo Heapsteru navíc vidíme některé základní informace o tom, jaké je využití zdrojů na nodech a využití zdrojů konkrétními Pody. Kromě toho nám však, ve srovnání s kubectl CLI, Dashboard příliš možností nepřináší – jak bylo

zmíněno, jedná se především o grafické rozhraní pro kubectl. Je tedy vhodné do clusteru zavést i určité monitorovací nástroje, umožňující větší škálu funkcí a nabízející lepší přístup k získávání metrik. Tímto nástrojem je například Prometheus. V této kapitole bude popsán možný způsob jeho nasazení a konfigurace základních funkcionalit.

Prometheus umožňuje sbírat širokou škálu metrik, jak už o nodech, na kterých Kubernetes cluster běží, tak o Podech, síťové komunikaci nebo i samotných aplikacích, které v Podech běží. Například dříve nasazovaný NGINX Ingress controller jsme také nastavili pro vystavování metrik, což nám umožní sledovat síťový provoz směřující z/do clusteru.

Všechny tyto informace jsou sbírané do časové databáze, nad kterou je poté možné pomocí PromQL [1] provádět různé dotazy a sledovat tato data. Prometheus také dále umožňuje zavádět různé konfigurace pro sledování prostředků v clusteru a upozorňovat na nežádoucí stavy. K tomu je využíváno zmiňovaných PromQL dotazů.

Prometheus má taktéž Helm chart v komunitním Helm repozitáři. Instalace je tedy opět ulehčena a sestává pouze z nakonfigurování potřebných hodnot pro Helm chart a nasazení pomocí helm příkazu. V tomto případě však budou konfigurované hodnoty pro Helm chart hrát důležitou roli, jelikož zde budeme konfigurovat i požadovanou funkcionalitu Promethea.

Základní konfigurací prostředků se zde nebudeme příliš zabývat. V případě této práce se jednalo pouze o vypnutí persistentního úložiště vzhledem k jeho nedostupnosti v používaném prostředí (tento přístup není příliš vhodný, jelikož restart Podu aplikace znamená ztrátu zaznamenaných dat) a o konfiguraci Ingress pravidel pro vystavení Prometheus rozhraní. Zajímavá je však konfigurace umožňující další sbírání dat (klíč extraScrapeConfigs) a schopnost Helm chart přímo přijímat interní konfigurace Promethea.

V extraScrapeConfigs je možné vytvořit job, který bude na zadaném endpointu hledat a sbírat metriky, stejně jako je sbírá automaticky uvnitř Kubernetes clusteru. V tomto případě byl Prometheus nakonfigurován tak, aby sledoval metriky vystavené CICD nástrojem Jenkins. V Prometheu bylo následně možné například hledat počet buildů, které v daném časovém úseku selhaly.

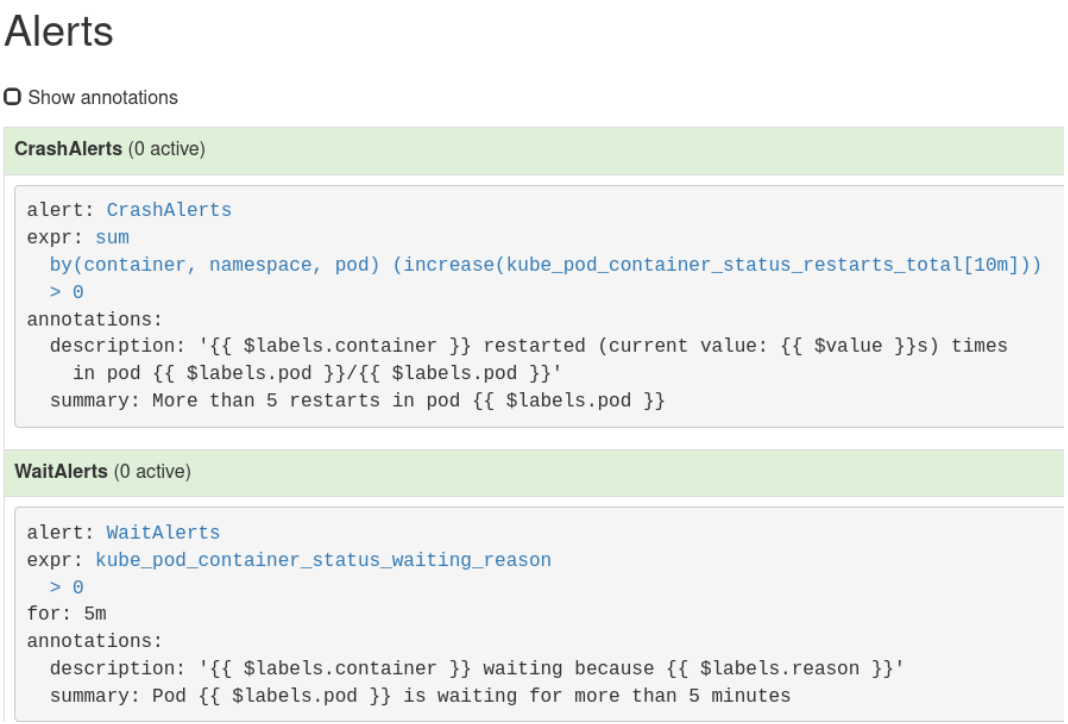
[1] Prometheus Query Language. Dotazovací jazyk umožňující vybírat a agregovat časově orientovaná data z Prometheus databáze. (46)

Další z užitečných funkcí, které Prometheus nabízí, je zasílání upozornění. Toho dosahuje pomocí komponentu AlertManager, který použitá Helm chart automaticky vytváří také. AlertManager lze nakonfigurovat na zasílání emailových upozornění. O tom, že má takové upozornění zaslat, a o jeho obsahu jej informuje samotný Prometheus. Emailová konfigurace je nastavována v Helm chart pomocí klíče, který umožňuje upravit přímo konfigurační soubor alertmanager.yml. V sekci pro úpravu Prometheus konfigurace prometheus.yml je poté nastavena interní adresa odkazující na AlertManager service v clusteru.

Konfigurace, která určuje, co bude spouštět samotná upozornění, je nastavována v serverFiles.alerts. V této sekci potom můžeme pod klíčem groups nastavovat libovolná pravidla pro upozorňování. Ta sestávají z dotazu na Prometheus databázi, který vyústí v true/false hodnotu rozhodující, zda má být upozornění zasláno. Zároveň je také možné definovat, jak dlouho musí podmínka být aktivní, aby se upozornění zaslalo.

Jako poslední se v konfiguraci definují pole summary a description, která při zasílání upozornění slouží jako pole předmět a samotné tělo zprávy. Pro základní kontrolu nad tím, zda Pody v clusteru fungují, jak mají, bylo nakonfigurováno upozornění na opakovaně crashující Pody. To je spuštěno, když se restart status Podu za posledních deset minut zvýšil. Zároveň bylo také vytvořeno upozornění na Pody, které se po dobu delší, než pět minut nacházejí v čekajícím stavu. Takový stav může indikovat například nemožnost stažení Docker image.

S takto nakonfigurovaným nasazením můžeme po vystavení nástroje Prometheus v jeho webovém grafickém rozhraní vidět vytvořená upozornění a pravidla pro jejich spouštění.



Obr. 13: Konfigurace upozornění v nástroji Prometheus (Zdroj: autor).

6.4 Grafana

Jako poslední z podpůrných aplikací zde bude zmíněn nástroj Grafana. Po instalaci a konfiguraci Promethea již máme způsob sbírání informací o dění v clusteru. Máme dokonce určité nastavení, které zajistí, abychom v případě problémů s clusterem byli včas informováni. Se všemi těmito metrikami je však možné ještě dále pracovat, a k tomu můžeme využít právě Grafanu.

Grafana umožňuje provádět dotazy nad různými typy datových úložišť včetně dat tvořených Prometheem. Tato data umí následně vizualizovat, či také zasílat oznámení v případě splnění určitých podmínek. Je možné vytvořit tzv. dashboardy s různými typy prvků, které vizualizují různá data získaná z Promethea.

Pro instalaci Grafany je taktéž možné použít komunitní Helm chart. Při její konfiguraci, podobně jako při konfigurace Promethea, vypneme persistentní úložiště a zároveň specifikujeme tzv. root_url serveru, jelikož budeme Grafanu vystavovat na URL /grafana a chceme předejít chybnému přepisování URL při přístupu.

Jako základní dashboard, který můžeme v případě potřeby dále modifikovat, lze využít některý ze široké nabídky existujících dashboardů, vytvořených komunitou. Vhodným kandidátem může být například Kubernetes All Nodes (47), poskytující základní přehled o stavu clusteru a jeho nodů.

Obr. 14: Kubernetes dashboard v nástroji Grafana (Zdroj: autor).

7. Weave Flux

Pro nasazování aplikací ve vytvořeném clusteru byl v této práci zvolen nástroj Weave Flux, který je představen v kapitole 4.4.3. Základní princip jeho použití spočívá v uložení veškeré konfigurace pro Kubernetes v Git repozitáři. Tento repozitář bude sledovat Flux prvek v clusteru a podle něj aplikovat stav do Kubernetes.

V této práci byla původně použita verze Weave Flux pracující s Helm 2, a tím pádem s prvkem Tiller. V době psaní této práce je nástroj zároveň rozšiřován o podporu Helm chart verze 3 a nasazování bez Tilleru. Zmíněny zde tedy budou oba přístupy. S přidáváním této podpory se zároveň rozdělily Helm chart pro nasazování na oddělené chart pro Flux a Helm-operator.

7.1 Helm 2

Flux je potřeba do Kubernetes nasadit stejně jako kterýkoliv jiný nástroj. Toho lze dosáhnout opět pomocí Helm chart. Jelikož však Weave Flux samotný používá Helm k nasazování (a v případě Helm 2 i Tiller), je potřeba mu popsat, jak s tímto prvkem v clusteru komunikovat. Proto bude v této části práce popsán i proces instalace zabezpečeného Tilleru do Kubernetes clusteru; je totiž nutné seznámit se s tímto postupem ještě před nasazením samotného Weave Flux.

Z bezpečnostních důvodů je vhodné vytvářet Tiller instance pouze v těch namespaces, ve kterých budou operovat, a následně k nim zabezpečit přístup a omezit jejich pravomoci pouze na daný Namespace. Jelikož však chceme používat Weave Flux pro nasazování aplikací v různých namespaces, bude muset stačit existence jednoho, mocnějšího Tilleru, k němuž bude přístup zabezpečen pomocí TLS.

Princip inicializace zabezpečeného Tilleru spočívá ve vytvoření prvku ServiceAccount pro Tiller s ClusterRoleBinding přidávající patřičná práva – v tomto případě možnost manipulace s veškerými prvků clusteru, tedy administrátorský přístup.

Pro inicializaci Tilleru s TLS musíme nejdříve vygenerovat tři soubory:

- CA
- klíč
- certifikát

K vygenerování potřebných souborů je možné použít CLI nástroj openssl nebo cfssl. Jako první musíme vygenerovat privátní klíč, který bude používán jako certifikační autorita neboli CA. Dále vygenerujeme hodnoty sloužící jako klíč a certifikát pro Tiller. Tyto klíče následně podepíšeme pomocí vygenerované CA.

```
openssl genrsa -out ./ca.key.pem 4096
openssl req -key ca.key.pem -new -x509 -days 7300 -sha256 -out ca.cert.pem -extensions v3_ca

openssl genrsa -out ./tiller.key.pem 4096
openssl req -key tiller.key.pem -new -sha256 -out tiller.csr.pem

 openssl x509 -req -CA ca.cert.pem -CAkey ca.key.pem -CAcreateserial -in tiller.csr.pem -out
tiller.cert.pem -days 7300
```

Obr. 15: Příkazy pro vygenerování potřebných klíčů/certifikátů pro Tiller (Zdroj: autor).

Nyní již můžeme inicializovat Helm se zabezpečeným Tillerem. K tomu použijeme příkaz helm init, kterému navíc zašleme příznaky specifikující, že přístup k prvku Tiller má být zabezpečený, a cesty k dříve vytvořeným kryptografickým souborům.

```
helm init \
--tiller-tls \
--tiller-tls-verify \
--tiller-tls-cert=tiller.cert.pem \
--tiller-tls-key=tiller.key.pem \
--tls-ca-cert=ca.cert.pem \
--service-account=tiller
```

Obr. 16: Příkaz pro inicializaci Helmu se zabezpečením Tilleru (Zdroj: autor).

S takto zabezpečeným Tillerem lze nyní komunikovat pomocí Helm příkazu odkazujícího na soubory pro CA, klíč a certifikát pod patřičnými příznaky. Pro tyto účely je kromě dříve vytvořeného klíče a certifikátu pro Tiller vhodné vytvořit také klíč a certifikát pro uživatele. Pro zjednodušení práce s helm příkazem je také možné tyto tři soubory (CA, klíč, certifikát) umístit do adresáře ~/.helm a při specifikování příznaku *--tls* budou použity automaticky.

```
helm list --tls --tls-ca-cert ca.cert.pem --tls-cert helm.cert.pem --tls-key helm.key.pem
```

Následné nasazení Weave Flux provedeme klasicky pomocí helm install s níže specifikovanými parametry. Oproti dříve popisovaným Helm nasazením je zde drobná změna, a to taková, že musíme nejdříve přidat Flux Helm repo.

```
helm repo add fluxcd https://fluxcd.github.io/flux
```

Zároveň také musíme vytvořit Flux CRD prvek HelmRelease, jehož definice je dostupná v GitHub repozitáři projektu (37). V době nasazování Flux s Helm 2 byla pro nasazení používána jen jedna chart nasazující jak Flux, tak Helm-operator.

Důležitým krokem při nasazování Weave Flux je nastavení parametrů pro tvořenou Helm chart. Jelikož byl Weave Flux nasazován ve firemním prostředí se zabezpečeným Git repozitářem, je potřeba zařídit ověření při komunikaci mezi Flux prvkem v Kubernetes a Git serverem. Zároveň je nutné poskytnout nástroji Flux informace potřebné k používání zabezpečené Tiller instance, jelikož s její pomocí bude nasazovat Helm charty.

Konfiguraci SSH spojení mezi Flux a Git serverem zajistíme nastavením ssh.known_hosts pro používanou Helm chart. Pod tento klíč přidáme hodnotu known_hosts z Git serveru. Tu je možné zjistit pomocí příkazu *ssh-keyscan -p<ssh-port> -H git.host.cz.* Příznak **-p** není nutné používat v případě, že SSH běží na standardním portu 22. V sekci git.url poté nastavíme URL pro SSH přístup k cílovému Git repozitáři.

Další částí konfigurace je nastavení Docker registru; jelikož v interním prostředí firmy je používáno tzv. insecure Docker registry, musí být přidáno do sekce registry.insecureHosts, aby k němu Flux úspěšně přistupoval. Zároveň zde můžeme nastavit výjimky pro ty registry, u nichž není žádoucí, aby je Flux skenoval pro nové verze image – zabráníme tím zbytečné síťové komunikaci a logování.

Kromě proxy konfigurace, která je v interním prostředí nutná, ještě zbývá konfigurace pro připojení k TLS zabezpečenému Tilleru. Ta spočívá v poskytnutí dříve vytvořeného klíče, certifikátu a certifikační autority, které byly generovány při tvorbě TLS zabezpečeného Tilleru. Tyto hodnoty jsou zadávány v sekci helmOperator.tls.

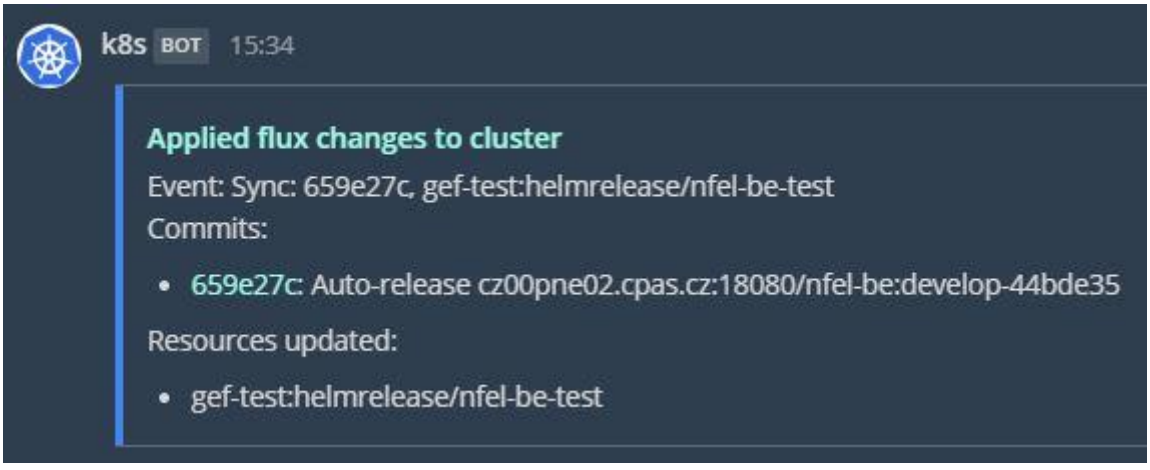
Dále je také možné specifikovat neveřejné Helm repozitáře, nacházející se například v interním Nexus registru. Informace o těchto repozitářích a přístupové údaje k nim definujeme v externím YAML souboru, popř. ve vytvořeném Secret prvku, na který odkážeme při tvorbě Helm chart v sekci configureRepositories.

Jako poslední přidáme konfiguraci pro velice užitečné rozšíření Weave Flux, a to tzv. fluxcloud (48). Weave poskytuje placené grafické uživatelské rozhraní umožňující správu Flux prvku a poskytující přehled nad probíhajícími operacemi a aktuálním stavem s názvem

Weave Cloud. Součástí této nabídky jsou také notifikace o nových událostech. Prvek fluxcloud umožňuje zasílání těchto notifikací i bez použití nástroje Weave Cloud. Konfigurace na straně Weave Flux je přímočará. Stačí pouze, když v poli additionalArgs uvedeme příznak --connect=ws://fluxcloud.

Samotný fluxcloud prvek nasadíme pomocí konfigurace v GitHub repozitáři, kterou si upravíme podle cílového systému, kam chceme zasílat notifikace. V případě této práce se jednalo o interně využívaný Mattermost, který kopíruje Slack rozhraní. Je tedy možné použít existující Slack exporter. Na straně Mattermostu je potřeba vytvořit uživatele a hoko pro proměnnou SLACK_URL.

Kromě samotného přístupu k Mattermostu bylo potřeba upravit proměnné COMMIT_TEMPLATE a BODY_TEMPLATE. Fluxcloud totiž předpokládá, že cílový repozitář se nachází na GitHubu, a podle toho formátuje zasílané zprávy. Cílový Git server byl však v tomto případě interní Bitbucket Server. Ten vystavuje informace o commitech na URL <adresa>/commits na rozdíl od <adresa>/commit, používané GitHubem. To znamená, že šablony zasílaných zpráv musely být upraveny, aby tyto změny reflektovaly.



Obr. 17: Mattermost notifikace o aktualizaci stavu Fluxem (Zdroj: autor).

7.2 Helm 3

První stabilní verze Helm 3, po předchozích beta verzích, byla vydána až v době, kdy tato práce již byla rozepsána. Zároveň byla také uskutečněna implementace přidávající podporu Helm 3 na straně Weave Flux. Proto zde bude popsán i proces práce s Weave Flux za použití právě Helm 3. Migrace na Helm 3 je rozhodně vhodným krokem; bez Tilleru se celý proces zjednoduší a zároveň bude bezpečnější.

V porovnání s nasazováním Weave Flux pro Helm 2, které bylo popsáno v kapitole 7.1, se změnil i tento proces. V průběhu implementace podpory pro Helm 3 byla totiž rozdělena původní jedna Helm chart na dvě samostatné – Flux chart a Helm operator chart. Musíme tedy nasadit obě a správně jim poskytnout potřebné parametry.

Kromě rozdělení jedné Helm chart používané k nasazení na dvě se však celý proces velice zjednodušil. Není potřeba řešit inicializaci ani zabezpečení Tilleru, jelikož Helm 3 pracuje s konfigurací používanou pro kubectl. Přístupy je tedy možné kontrolovat pomocí nativních Kubernetes RBAC konfigurací. Helm 3 také nově pracuje na základě Namespace a informace o nasazených release zároveň uchovává přímo v clusteru, a to v prostředcích Secret.

Můžeme se tedy úplně vyhnout tvorbě certifikátů a příkazům jako helm init, zmiňovaných v předchozí kapitole, a rovnou řešit konfiguraci, s jakou nasadíme Weave Flux. Jak bylo zmíněno, z původní konfigurace používané pro nasazení s Helm 2 zmizí nastavení pro Tiller a zůstane pouze nastavení pro používaný privátní Git repozitář.

Konfigurace pro tento repozitář bude muset být poskytnuta v konfiguračních hodnotách obou chart, jelikož obě budou potřebovat přístup ke Git repozitáři – Flux pro synchronizaci manifestů a helm-operator pro přečtení Helm chart, vytvořených pro nasazení vlastních aplikací a uchovávaných v Gitu. Konfigurace pro Flux tedy bude stejná s výjimkou chybějící helmOperator sekce.

Konfigurace pro helm-operator chart bude muset obsahovat sekci git pro přístup k repozitáři. Zde znovu specifikujeme known_hosts pod klíčem ssh.known_hosts a zároveň musíme nějak helm-operator prvku zajistit přístup k tomuto repozitáři. Toho nejsnáze dosáhneme tak, že nasadíme jako první Flux chart a následně se odkážeme na vytvořený klíč, který bude pro Flux přidáván do přístupových klíčů Git repozitáře.

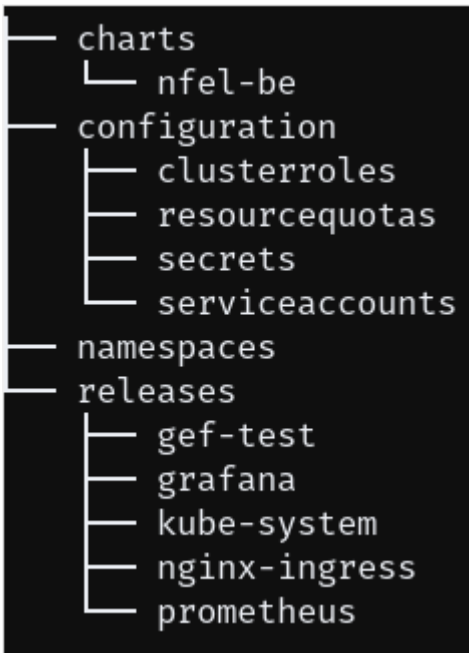
Jako poslední krok v konfiguraci helm-operator chart můžeme určit, jakou verzi chart bude operátor moci zpracovávat. Ve výchozím stavu toto nastavení povoluje jak Helm chart verze 2, tak verze 3. Jelikož se však chceme zbavit Tilleru, zvolíme zde pouze hodnotu v3.

7.3 Definice aplikace k nasazení

V Git repozitáři, na který ve Weave Flux odkazujeme, můžeme definovat jakýkoli validní Kubernetes manifest. Můžeme zde tedy přehledně spravovat jednotlivé základní

konfigurace, které chceme v Kubernetes mít, jako definice Namespace, ServiceAccount, Role, RoleBinding, ResourceQuota apod. Jelikož Flux pracuje nad soubory, můžeme jednotlivé konfigurace pro větší přehled dělit do podsložek.

Výsledná souborová struktura adresáře poté může vypadat například takto (obsahy složek a některé podsložky byly pro přehlednost odebrány):



Obr. 18: Struktura Git repozitáře pro Weave Flux (Zdroj: autor).

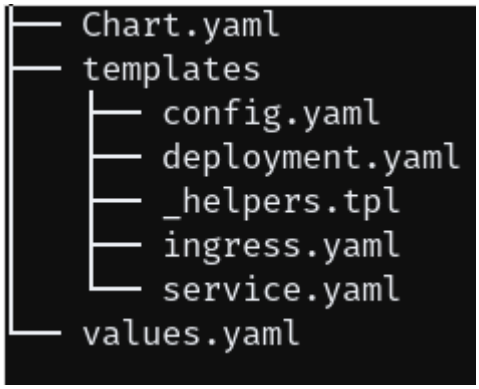
Kromě těchto prvků zde však budeme chtít uchovávat i konfigurace pro nasazení našich vlastních aplikací, a to pomocí Helm. Toho dosáhneme vytvořením konfigurace pro prvek HelmRelease, což je CRD vytvářený při instalaci Weave Flux. V HelmRelease můžeme odkazovat buď na existující chart ve vzdáleném Helm chart repozitáři, nebo na vlastní chart nacházející se v Gitu.

Vlastní Helm chart můžeme tvořit přímo v cílovém Gitu, protože Flux rozpozná, že se jedná o Helm chart, a nebude se snažit aplikovat konfigurovatelné manifesty, které se v ní nacházejí.

Většina aplikací potřebuje pro běh především Deployment, Service a Ingress. Tyto prvky můžeme v Helm chart vytvořit tak, aby byly následně konfigurovatelné bez nutného zásahu do samotné chart.

Základní strukturu Helm chart vytvoříme příkazem helm create, zmiňovaným v kapitole 4.4.2. Soubor Chart.yaml obsahuje základní informace o chart a values.yaml výchozí

hodnoty konfigurace. Do vytvořené podsložky templates budeme přidávat prostředky, které budeme chtít pomocí chart nasazovat.



Obr. 19: Struktura Helm chart (Zdroj: autor).

Prostředek, který z této možnosti konfigurace těží nejvíce, je Deployment. Ten můžeme vytvořit takovým způsobem, že do něj lze pouze pomocí předávání hodnot Helm příkazu například přidávat proměnné prostředí nebo připojovat úložiště.

```
spec:
    containers:
    - name: {{ template "nfel-be.fullname" . }}
      image: {{ required "Image is required" .Values.image }}
      imagePullPolicy: Always
      ports:
      - containerPort: 8080
      {{- if .Values.extraEnvs }}
      env:
{{ toYaml .Values.extraEnvs | trim | indent 8 }}
      {{- end }}
      {{- if .Values.extraVolumeMounts }}
      volumeMounts:
{{ toYaml .Values.extraVolumeMounts | trim | indent 8 }}
      {{- end }}
      resources:
        requests:
          memory: {{ .Values.memoryRequest }}
        limits:
          memory: {{ .Values.memoryLimit }}
    imagePullSecrets:
      - name: {{ default "nexus-docker" .Values.secretName }}
    {{- if .Values.extraVolumes }}
    volumes:
{{ toYaml .Values.extraVolumes | indent 6 }}
    {{- end }}
```

Obr. 20: Konfigurace Deployment v Helm chart využívající proměnných a template funkcí (Zdroj: autor).

Konkrétně se jedná o sekci spec v definici Deploymentu. Zde můžeme pomocí if/else bloků přidávat zmiňované bloky až při samotném nasazování. YAML soubor s těmito hodnotami může vypadat například takto (zde je přidáváno volume odkazující na prostředek ConfigMap a zároveň je definovaná extra proměnná prostředí):

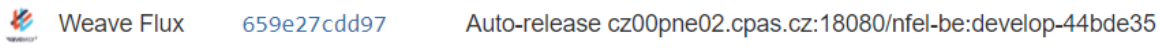
```
extraVolumeMounts:
- name: springconfig
  mountPath: /etc/springconfig/
extraVolumes:
- name: springconfig
  projected:
     sources:
     - configMap:
        name: &configmap_name springconfig-0.5
extraEnvs:
- name: SPRING_OPTS
  value: "--spring.config.additional-location=/etc/springconfig/application.yml\
      \ --server.servlet.context-path=/nfel-be"
```

Obr. 21: Část Helm chart konfigurace upravující Deployment pro použití ConfigMap (Zdroj: autor).

Po vytvoření Helm chart musíme Weave Flux nějak předat informaci, že ji chceme nasadit, čehož dosáhneme vytvořením konfigurace pro HelmRelease. HelmRelease obsahuje, stejně jako všechny Kubernetes prvky, pole pro specifikaci názvu a Namespace, ve kterém se má nacházet. Ve spec části konfigurace se poté uvádí název release, dále chart, která má být použita, a hodnoty (values) pro nasazovanou chart. Tyto hodnoty fungují stejným způsobem, jako když při ručním nasazení Helm chart poskytneme YAML soubor s hodnotami.

Navíc však HelmRelease umožňuje přidat tzv. anotace. Pomocí těchto anotací můžeme nastavit, zda má být release automatizovaná, a podle jakých pravidel. Automatizaci je možné zapnout pomocí *fluxcd.io/automated: "true"* v novější verzi CRD, nebo *flux.weave.works/automated: "true"* v té starší.

Anotace pravidla pro automatizaci následně určuje, jak má Flux hledat nové verze aplikace. Nastavením *filter.fluxcd.io/chart-image: glob:develop-**, popř. dřívějším *flux.weave.works/tag.chart-image: glob:develop-**, například zajistíme, aby se při výskytu nové Docker image s tagem začínajícím develop- aktualizovala označená HelmRelease. Flux v reakci na tuto změnu aktualizuje manifest v Gitu, konkrétně tedy tag používané image ve values sekci.



Obr. 22 Git manifest aktualizovaný nástrojem Weave Flux (Zdroj: autor).

Jednoduchá výsledná HelmRelease definice, umožňující automatické přenasazování při nalezení nové verze, může vypadat například takto:

```
apiVersion: flux.weave.works/v1beta1
kind: HelmRelease
metadata:
  name: nfel-be-test
  namespace: gef-test
  annotations:
    flux.weave.works/automated: "true"
    flux.weave.works/tag.chart-image: glob:develop-*
spec:
  releaseName: nfel-be-test
  chart:
    git: ssh://git@git.cpas.cz:7999/k8s/flux.git
    path: charts/nfel-be
    ref: master
  values:
    pathUri: nfel-be
    image: nexus.cpas.cz:18080/nfel-be:develop-7216180
```

Obr. 23: Konfigurace HelmRelease s automatickým nasazováním (Zdroj: autor).

Weave Flux tedy poskytuje jednoduchý, avšak efektivní přístup ke správě konfigurace pro Kubernetes a k nasazování aplikací. Zároveň slouží jako zálohovací řešení. Jelikož máme vše definováno v Gitu, v případě problémů s clusterem stačí do nového clusteru nainstalovat Flux a připojit jej ke Gitu, a veškerá konfigurace bude znovu vytvořena.

7.4 SealedSecrets

Vzhledem k tomu, že chceme v Git repozitáři pro Weave Flux uchovávat co největší množství naší nasazené Kubernetes konfigurace, brzy narazíme na problém uchovávání Secret prvků v tomto repozitáři. Uchovávat zde například hesla, která se mají v Secret prvcích nacházet, by znamenalo zanášet je do Gitu v plaintext podobě, což rozhodně není vhodné.

Jedním z možných řešení tohoto problému je použití nástroje Bitnami SealedSecrets (49). Spočívá v instalaci SealedSecrets controlleru spolu s CRD SealedSecret do Kubernetes clusteru. Tento prvek bude následně všechny vytvořené CRD přetvářet na Kubernetes Secret prvky.

Představa workflow je tedy tato: v případě potřeby uchovávání senzitivních informací v Git repozitáři vytvoříme místo klasického prostředku Secret prostředek SealedSecret, který bude následně v clusteru přetvořen na klasický Secret.

Instalace controlleru a CRD prvku je možná pomocí konfiguračních YAML souborů, které jsou publikovány na GitHub repozitáři projektu. Z téže stránky také můžeme stáhnout CLI nástroj pro práci s nasazeným controllerem – kubeseal.

Workflow po instalaci spočívá ve vytvoření klasického Secret, avšak s příznakem *--dry-run*, který způsobí, že se prvek ve skutečnosti v clusteru nevytvoří a bude pouze vypsán manifest, který by pro jeho tvorbu byl použit. Nad tímto výstupem následně zavoláme příkaz kubeseal, jehož výstupem bude manifest pro prvek SealedSecret. Tento prvek poté můžeme zanést do Git repozitáře a při aplikaci do clusteru z něj nasazený controller vytvoří klasický Secret prvek, který je možné konzumovat aplikacemi.

8. Flux webhook receiver

Jediným problematickým bodem Fluxu může být určité zpoždění mezi jednotlivými akcemi. Například po aktualizaci manifestů v Gitu může propsání do clusteru trvat až pět minut, a to vzhledem k tomu, že Flux funguje na principu pollování (opakovaného dotazování se v časových intervalech). Totéž platí pro nové verze Docker image v registry. S cílem odstranit tyto nedostatky byl vytvořen nástroj Flux webhook receiver. Nástroj byl vyvíjen a testován proti Flux verzi 1.15.0.

8.1 Návrh

Navrhovaným řešením výše zmíněného problému je rozšířit projekt Weave Flux o možnost konfigurace webhooků. Ty by Flux informovaly jak o změnách v Git repozitáři s konfiguracemi, tak o vytvoření Docker image s novou verzí aplikace.

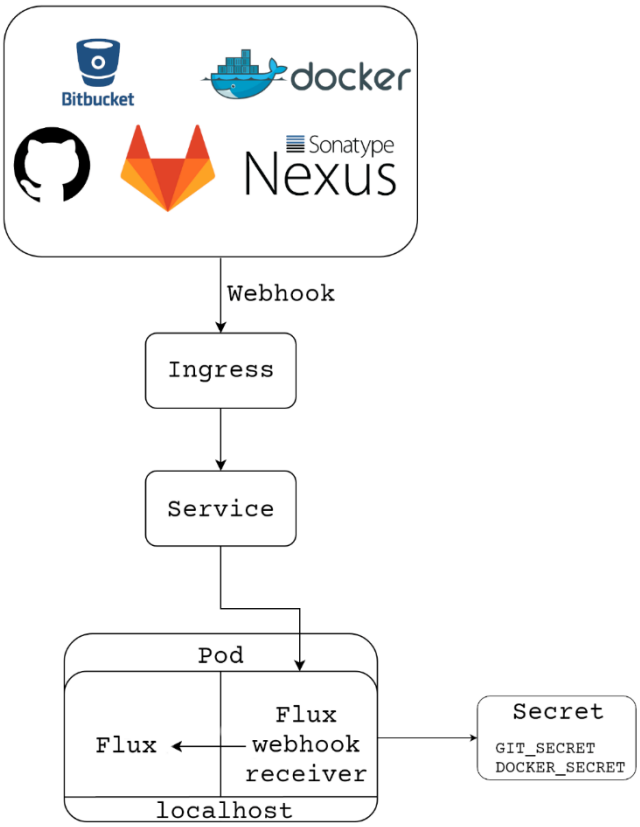
Toto rozšíření by mělo být konfigurovatelné pro přijímání webhooků hlavních poskytovatelů hostingů pro Git repozitáře a pro Docker image registry.

Zároveň je žádoucí umožnit konfiguraci tzv. secret pro vytvářené webhooky s cílem znemožnit falšování jejich volání. Dále by mělo být možné nastavit větve Git repozitáře, pro které mají být webhooky registrovány.

Weave Flux již obsahuje interní API, které umožňuje upozornit kontrolní prvek v Kubernetes na změny ve sledovaném Git repozitáři, popř. v image registry. Tuto funkcionalitu tedy není nutné implementovat.

Dalším požadavkem na řešení, kromě těch již zmíněných, je však ten, že toto notifikační Flux API nemá být přímo vystaveno. A to ani v Kubernetes, a už vůbec ne vně clusteru.

Pro splnění všech těchto požadavků je navržené řešení koncipováno jako samostatná aplikace, kterou lze připojit k běžícímu Flux kontejneru, resp. v terminologii Kubernetes k běžícímu Flux Podu.



Obr. 24 High-level overview po workflow s použitím webhooku (Zdroj: autor).

Pro splnění požadavku o nevystavování interního Flux API poběží vytvářená aplikace v jednom Podu spolu s Flux kontejnerem. Díky vlastnosti Podů, kdy kontejnery v Podu sdílejí localhost rozhraní, bude možné API volat, aniž by bylo kdekoli jinde vystaveno.

Vytvářená aplikace tedy bude zpracovávat příchozí webhooky podle zadané konfigurace a následně volat Flux s informacemi k notifikaci.

Většina poskytovatelů Git hostingu a Docker image hostingu podporuje konfiguraci secret pro vytvářené webhooky. Princip nastavení této hodnoty je takový, že zasílaná zpráva bude obsahovat MAC[1], umožňující validaci zprávy.

HMAC je typ MAC, založený na kryptografických hashovacích funkcích. Lze použít jakoukoli iterovanou[2] kryptografickou hashovací funkci a tajný klíč K (50). HMAC je poté zkonstruován ze zasílané zprávy a přidán do hlaviček volání.

[1] MAC – message authentication code. Kód používaný mezi dvěma komunikujícími stranami sdílejícími tajný klíč pro validaci zasílaných informací (50).
[2] Hashovací funkce H tvořena rozdělením vstupu na m-bitové bloky a následným množstvím volání kompresní funkce F (51), (52).

Pro vytvoření HMAC používají různí poskytovatelé různé kryptografické hashovací funkce. Nejčastěji však SHA-1, kterou používá například GitHub nebo SHA-256, používanou poskytovatelem BitBucket Server.

Navrhované řešení tedy předpokládá následující kroky:

- vytvoření kontejnerizovatelné aplikace Flux webhook receiver
- rozšíření Flux Deployment konfigurace o Flux webhook receiver kontejner
- vytvoření Secret prvku obsahujícího senzitivní informace o webhook secret
- vytvoření Service prvku, jež bude vystavovat pouze Flux webhook receiver
- vytvoření Ingress pravidel pro vystavení Flux webhook receiver Service
- webhook konfiguraci u vybraných poskytovatelů směřující na URI vystavenou pomocí Ingress.

8.2 Konfigurace

Konfigurace pro nasazení používá přístup tzv. sidecar pattern. Ten spočívá v přidání definice pro Flux webhook receiver do Deployment konfigurace pro Flux. Dále Flux webhook receiver používá prostředky Service a Ingress. Port 3033 Flux Podu je vystaven pomocí Service a na tento Service následně odkazuje Ingress.

Definice obsahuje informace o Docker image, která má být spouštěna, dále o používaném portu a také o nastavení konfiguračních proměnných, přičemž na hodnoty senzitivních informací je odkazováno do Secret prvku, který musí být vytvořen odděleně, například pomocí zmiňovaných SealedSecrets.

Aplikace přijímá konfiguraci upravující její chování, a to:

- zapnutí/vypnutí endpointů (výchozí hodnota zapnuto)
- secret pro Git webhook
- secret pro Docker image registry webhook
- typ Git hostingu
- typ Docker image registry hostingu
- Git větve, jejichž zprávy mají být přijímány

Podporovaní poskytovatelé Git repozitářů jsou GitHub, GitLab, BitBucket Server / BitBucket; podporovaní poskytovatelé Docker image registry jsou DockerHub a Sonatype Nexus.

8.3 Implementace

Aplikace je – vzhledem ke kompatibilitě s Flux codebase a k relativně malé velikosti výsledných spustitelných souborů – vytvořena v programovacím jazyce Golang.

Princip implementace spočívá ve vytvoření HTTP serveru, vystavujícího API pro příjem webhooku z Git repozitáře a z Docker image registry. Tento server je vystaven na TCP portu 3033 s konkrétními handlery webhooků, nacházejícími se na URL /gitSync a /imageSync.

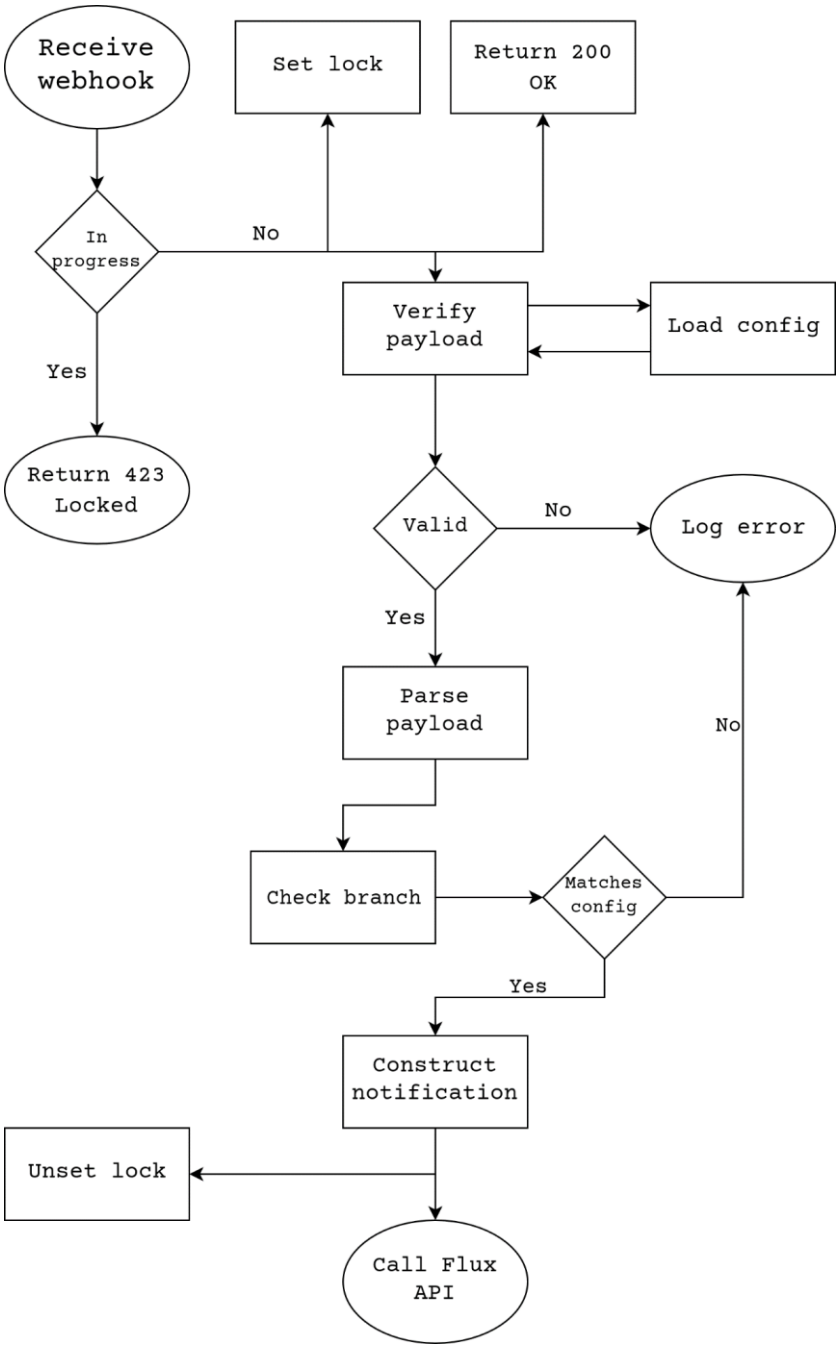
Na vystavené URL těchto handlerů poté budou nasměrovány webhooky konkrétního poskytovatele Git/Docker repozitáře s případně nastavenou hodnotou secret pro ověření správnosti.

Po obdržení webhooku je řízení předáno specifickému handleru. V případě Git webhooků se nejprve zkontroluje, zda již neprobíhá obsluha dříve přijatého webhooku. V případě, že ano, je vrácen HTTP status kód 423 Locked.

Cílem tohoto opatření je zabránit zahlcení Flux API množstvím žádostí o notifikaci v případě, že poskytovatel Git hostingu z nějakého důvodu zašle větší množství informací najednou. Z principu fungování Weave Flux plyne, že nevadí, pokud nejsou obslouženy všechny Git webhooky, protože Flux si po obdržení notifikace stáhne aktuální informace z Git repozitáře.

Pokud zrovna není Git webhook obsluhován, je nastavena kontrolní lock proměnná a vrácen HTTP status kód 200 informující, že webhook byl úspěšně přijat.

Obsah příchozí zprávy je následně validován. Validace probíhá pomocí ověření HMAC zaslaného v hlavičkách zprávy. Ověření je provedeno výpočtem vlastního HMAC pomocí secret, načteného z konfigurace, a jejich vzájemným porovnáním. Typ použité kryptografické funkce určuje konfigurace specifikující typ poskytovatele, od kterého webhook přichází.



Obr. 25 Flowchart zpracování Git webhooku (Zdroj: autor).

V případě, že zpráva není úspěšně ověřena, je chyba validace zapsána do logu aplikace. Pokud byla validace úspěšná, je obsah příchozí zprávy zpracován.

Ze získaného obsahu je přečtena informace o větvi, na kterou se přijatý webhook vztahuje. V případě, že větev neodpovídá žádné z větví nastavených v konfiguraci, aplikace dále nepokračuje a zapíše tuto informaci do logu.

Pokud větev odpovídá, je ze získaných informací zkonstruována zpráva notifikace pro Flux. Ta je tvořena použitím existující struktury pro notifikaci nacházející se ve Flux

codebase. Následně je zavoláno Flux API a zároveň je kontrolní lock proměnná nastavena na hodnotu indikující, že Git webhook není momentálně obsluhován.

8.4 Docker image

```
FROM instrumentisto/dep:0.5-alpine as builder

LABEL stage=builder

WORKDIR /go/src/github.com/proskehy/flux-webhook-receiver

COPY Gopkg.lock ./
COPY Gopkg.toml ./
RUN dep ensure --vendor-only

COPY cmd ./cmd
COPY pkg ./pkg
RUN cd cmd && go build

FROM alpine:latest
COPY --from=builder /go/src/github.com/proskehy/flux-webhook-receiver/cmd/cmd /bin/flux-webhook-receiver
EXPOSE 3033
ENTRYPOINT ["flux-webhook-receiver"]
```

Obr. 26: Dockerfile pro Flux webhook receiver (Zdroj: autor).

Aplikace je distribuována jako Docker image. Build image probíhá pomocí tzv. multi-stage buildu.

Ve výsledné Dockerfile se nachází dvě stage, které lze poznat tak, že každá začíná instrukcí FROM. V první části je použita image, obsahující Golang a nástroj pro správu dependencí dep.

Nejdříve jsou zkopírovány pouze soubory udávající závislosti a je provedena jejich instalace. Díky tomu se základní vrstva builder image se všemi závislostmi mění pouze v případě, že potřebujeme přidat novou závislost.

Následně je proveden build aplikace pomocí go build, jehož výstupem je spustitelný soubor.

V druhé části buildu, založené na lightweight distribuci Alpine, je zkopírován vytvořený spustitelný soubor, nastaven port 3033 k vystavení a ENTRYPOINT specifikující spuštění aplikace při startu kontejneru.

9. Shrnutí výsledků

V rámci této práce byl vytvořen on-premise Kubernetes cluster v prostředí firmy Generali Česká pojišťovna a.s. a konfigurace pro nástroj Ansible, automatizující jeho tvorbu či připojování nových nodů. Tvorba clusteru byla přizpůsobena firemnímu prostředí a s ním spojenými omezeními, jako například využití proxy.

Do vytvořeného clusteru byly přidány nástroje poskytující větší přehled nad děním v clusteru a umožňující manipulaci s ním. Konkrétně se jedná o nástroje Kubernetes Dashboard (ovládání clusteru přes GUI), Grafana (vizualizace metrik) a Prometheus (sběr metrik a zasílání upozornění).

Následně byl do clusteru nasazen nástroj Weave Flux, který umožnil nasazení aplikací pomocí GitOps přístupu (uchovávání konfigurace v Gitu). Weave Flux byl otestován jak s Helm verzí 2, tak 3. Nakonfigurována byla také možnost uchovávání senzitivních informací v Gitu pomocí SealedSecrets.

Na závěr bylo naprogramováno rozšíření nástroje Weave Flux přidávající webhook funkcionalitu. Díky tomuto rozšíření je propagování informací o novém commitu či nové Docker image instantní.

10. Závěr

Úspěšný provoz kontejnerizovaných aplikací vyžaduje robustní řešení, která umožňují efektivní správu množství microservice aplikací. V tomto ohledu je využití služby Kubernetes, nebo služeb na Kubernetes postavených, logickým krokem.

Již samotná tvorba a provoz clusteru však vyžadují určité znalosti a čas. Pro úspěšné zvládnutí této části musí firmy buď investovat do expertů, nebo vyhledat nějaké hotové nabízené řešení.

Kromě starosti o cluster je však dalším klíčovým prvkem k úspěchu návrh vhodné workflow pro nasazování a konfiguraci provozovaných aplikací. Vzhledem k velkému množství nabízených přístupů k této problematice je důležité vhodně zvolit a otestovat cílové řešení.

Informace poskytnuté v této práci vytváří základ pro provoz vlastního on-premise Kubernetes clusteru. Zároveň představují jednoduchou workflow pro automatizaci nasazování aplikací a jejich konfiguraci. S pomocí těchto informací lze sestavit prostředí například pro vývojáře či testery.

K úspěšnému použití tohoto řešení v produkčním prostředí by bylo nutné bližší zmapování aplikací, které mají být provozovány. Pro ně následně zdokonalit proces nasazování s ohledem na jejich konfiguraci, logování, dostupnost a zabezpečení.

11. Seznam zdrojů

1. Datadog. 8 surprising facts about real Docker adoption [Internet]. 8 surprising facts about real Docker adoption. [citován 25. listopad 2019]. Dostupné z: https://www.datadoghq.com/docker-adoption/

2. Datadog. 8 facts about the changing container landscape [Internet]. 8 facts about the changing container landscape. [citován 24. listopad 2019]. Dostupné z: https://www.datadoghq.com/container-report/

3. Preston-Werner T. Semantic Versioning 2.0.0 [Internet]. Semantic Versioning. [citován 30. leden 2020]. Dostupné z: https://semver.org/

4. Practical Docker with Python - Build, Release and Distribute your Python App with Docker | Sathyajith Bhat | Apress [Internet]. [citován 12. duben 2020]. Dostupné z: https://www.apress.com/gp/book/9781484237830

5. namespaces(7) - Linux manual page [Internet]. [citován 28. prosinec 2019]. Dostupné z: http://man7.org/linux/man-pages/man7/namespaces.7.html

6. comment 15 Apr 2019 Marty Kalin Feed 143up 1. Inter-process communication in Linux: Shared storage [Internet]. Opensource.com. [citován 12. duben 2020]. Dostupné z: https://opensource.com/article/19/4/interprocess-communication-linux-storage

7. Architecting and Operating OpenShift Clusters - OpenShift for Infrastructure and Operations Teams | William Caban | Apress [Internet]. [citován 12. duben 2020]. Dostupné z: https://www.apress.com/gp/book/9781484249840

8. cgroups(7) - Linux manual page [Internet]. [citován 28. prosinec 2019]. Dostupné z: http://man7.org/linux/man-pages/man7/cgroups.7.html

9. Introducing Azure Kubernetes Service - A Practical Guide to Container Orchestration | Steve Buchanan | Apress [Internet]. [citován 12. duben 2020]. Dostupné z: https://www.apress.com/gp/book/9781484255186

10. opencontainers/runtime-spec [Internet]. GitHub. [citován 12. duben 2020]. Dostupné z: https://github.com/opencontainers/runtime-spec

11. Container Runtimes Part 1: An Introduction to Container Runtimes - Ian Lewis [Internet]. [citován 28. prosinec 2019]. Dostupné z: https://www.ianlewis.org/en/container-runtimes-part-1-introduction-container-r

12. containerd [Internet]. [citován 28. prosinec 2019]. Dostupné z: https://containerd.io/

13. cri-o [Internet]. [citován 28. prosinec 2019]. Dostupné z: https://cri-o.io/

14. Say “Hello” to Buildah, Podman, and Skopeo [Internet]. Red Hat Services Speak. 2019 [citován 12. duben 2020]. Dostupné z: https://servicesblog.redhat.com/2019/10/09/say-hello-to-buildah-podman-and-skopeo/

15. Kubernetes Components [Internet]. [citován 25. listopad 2019]. Dostupné z: https://kubernetes.io/docs/concepts/overview/components/

16. kubelet [Internet]. [citován 25. listopad 2019]. Dostupné z: https://kubernetes.io/docs/reference/command-line-tools-reference/kubelet/

17. kube-controller-manager [Internet]. [citován 25. listopad 2019]. Dostupné z: https://kubernetes.io/docs/reference/command-line-tools-reference/kube-controller-manager/

18. Using CoreDNS for Service Discovery [Internet]. [citován 12. duben 2020]. Dostupné z: https://kubernetes.io/docs/tasks/administer-cluster/coredns/

19. Container runtimes [Internet]. [citován 28. prosinec 2019]. Dostupné z: https://kubernetes.io/docs/setup/production-environment/container-runtimes/

20. Cluster Networking [Internet]. [citován 25. listopad 2019]. Dostupné z: https://kubernetes.io/docs/concepts/cluster-administration/networking/

21. MetalLB, bare metal load-balancer for Kubernetes [Internet]. [citován 27. prosinec 2019]. Dostupné z: https://metallb.universe.tf/

22. kubernetes/kops [Internet]. Kubernetes; 2019 [citován 30. prosinec 2019]. Dostupné z: https://github.com/kubernetes/kops

23. What is Kops? Kubernetes Operations with Kops - Cloud Academy Blog [Internet]. Cloud Academy. 2017 [citován 30. prosinec 2019]. Dostupné z: https://cloudacademy.com/blog/kubernetes-operations-with-kops/

24. kubernetes-sigs/kubespray [Internet]. Kubernetes SIGs; 2019 [citován 30. prosinec 2019]. Dostupné z: https://github.com/kubernetes-sigs/kubespray

25. Ansible Documentation [Internet]. [citován 30. prosinec 2019]. Dostupné z: https://docs.ansible.com/

26. Overview of kubeadm [Internet]. [citován 30. prosinec 2019]. Dostupné z: https://kubernetes.io/docs/reference/setup-tools/kubeadm/kubeadm/

27. kubeadm init [Internet]. [citován 1. leden 2020]. Dostupné z: https://kubernetes.io/docs/reference/setup-tools/kubeadm/kubeadm-init/

28. kubeadm join [Internet]. [citován 1. leden 2020]. Dostupné z: https://kubernetes.io/docs/reference/setup-tools/kubeadm/kubeadm-join/

29. kubeadm upgrade [Internet]. [citován 1. leden 2020]. Dostupné z: https://kubernetes.io/docs/reference/setup-tools/kubeadm/kubeadm-upgrade/

30. kubeadm reset [Internet]. [citován 1. leden 2020]. Dostupné z: https://kubernetes.io/docs/reference/setup-tools/kubeadm/kubeadm-reset/

31. Namespaces [Internet]. [citován 31. prosinec 2019]. Dostupné z: https://kubernetes.io/docs/concepts/overview/working-with-objects/namespaces/

32. Configure Service Accounts for Pods [Internet]. [citován 31. prosinec 2019]. Dostupné z: https://kubernetes.io/docs/tasks/configure-pod-container/configure-service-account/

33. Using RBAC Authorization [Internet]. [citován 31. prosinec 2019]. Dostupné z: https://kubernetes.io/docs/reference/access-authn-authz/rbac/

34. Helm 2 docs [Internet]. [citován 3. leden 2020]. Dostupné z: https://v2.helm.sh/docs/

35. helm/charts [Internet]. The Helm Project; 2019 [citován 28. prosinec 2019]. Dostupné z: https://github.com/helm/charts

36. Helm 3 Docs [Internet]. [citován 3. leden 2020]. Dostupné z: https://helm.sh/docs/

37. fluxcd/flux [Internet]. Flux project; 2019 [citován 21. prosinec 2019]. Dostupné z: https://github.com/fluxcd/flux

38. systemd.unit [Internet]. [citován 30. leden 2020]. Dostupné z: https://www.freedesktop.org/software/systemd/man/systemd.unit.html

39. CentOS GPG Keys [Internet]. [citován 30. leden 2020]. Dostupné z: https://www.centos.org/keys/

40. Integrating Kubernetes via the Addon [Internet]. [citován 31. leden 2020]. Dostupné z: https://www.weave.works/docs/net/latest/kubernetes/kube-addon/

41. Welcome - NGINX Ingress Controller [Internet]. [citován 15. leden 2020]. Dostupné z: https://kubernetes.github.io/ingress-nginx/

42. Service [Internet]. [citován 15. leden 2020]. Dostupné z: https://kubernetes.io/docs/concepts/services-networking/service/

43. helm/charts [Internet]. The Helm Project; 2020 [citován 15. leden 2020]. Dostupné z: https://github.com/helm/charts

44. kubernetes/dashboard [Internet]. GitHub. [citován 15. leden 2020]. Dostupné z: https://github.com/kubernetes/dashboard

45. Resource metrics pipeline [Internet]. [citován 15. leden 2020]. Dostupné z: https://kubernetes.io/docs/tasks/debug-application-cluster/resource-metrics-pipeline/

46. Querying basics | Prometheus [Internet]. [citován 30. leden 2020]. Dostupné z: https://prometheus.io/docs/prometheus/latest/querying/basics/

47. Kubernetes All Nodes dashboardData for Grafana [Internet]. Grafana Labs. [citován 19. leden 2020]. Dostupné z: https://grafana.com/grafana/dashboards/3131

48. Justin. justinbarrick/fluxcloud [Internet]. 2020 [citován 21. leden 2020]. Dostupné z: https://github.com/justinbarrick/fluxcloud

49. bitnami-labs/sealed-secrets [Internet]. Bitnami Labs; 2019 [citován 28. prosinec 2019]. Dostupné z: https://github.com/bitnami-labs/sealed-secrets

50. Krawczyk H, Bellare M, Canetti R. HMAC: Keyed-Hashing for Message Authentication. 2. leden 1997 [citován 23. prosinec 2019]; Dostupné z: http://dl.acm.org/citation.cfm?id=RFC2104

51. Lucks S. Iterated Cryptographic Hash Functions [Internet]. Dostupné z: https://www.uni-weimar.de/fileadmin/user/fak/medien/professuren/Mediensicherheit/Teaching/SS17/Hash_Functions/hash02.pdf

52. Lucks S. Design Principles for Iterated Hash Functions [Internet]. 2004 [citován 23. prosinec 2019]. Report No.: 253. Dostupné z: https://eprint.iacr.org/2004/253

Univerzita Hradec Králové
Fakulta informatiky a managementu

Zadání diplomové práce

Autor:	**Bc. Hynek Proske**
Studium:	I1800119
Studijní program:	N1802 Aplikovaná informatika
Studijní obor:	Aplikovaná informatika
Název diplomové práce:	**Tvorba a využití on-premise Kubernetes clusteru**
Název diplomové práce AJ:	Creation and use of on-premise Kubernetes cluster

Cíl, metody, literatura, předpoklady:

Cílem práce je popsat proces tvorby on-premise Kubernetes clusteru a vybraný způsob nasazování a provozu aplikací v takovém clusteru. Práce také uvede teoretické základy ohledně jednotlivých prvků Kubernetes clusteru a rozdílech mezi on-premise a cloud řešeními.

Osnova:

1. Úvod
2. Cíl práce
3. Kubernetes cluster
4. Tvorba clusteru
5. Podpůrné aplikace
6. Shrnutí výsledků
7. Závěr

https://kubernetes.io/docs/home/

Garantující pracoviště:	Katedra informatiky a kvantitativních metod, Fakulta informatiky a managementu
Vedoucí práce:	doc. Mgr. Tomáš Kozel, Ph.D.
Datum zadání závěrečné práce:	14.1.2018