Univerzita Hradec Králové

Fakulta informatiky a managementu

Katedra informatiky a kvantitativních metod

Docker kontejnerizace v procesech vývoje SW aplikací

Diplomová práce

Autor: Bc. Michal Šípek

Studijní obor: N1802 Aplikovaná informatika

Vedoucí práce: Ing. Pavel Kříž, Ph.D.

Hradec Králové srpen 2018

Prohlášení:

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

V Hradci Králové dne 13. srpna 2018 Bc. Michal Šípek

Poděkování:

Rád bych tímto poděkoval vedoucímu diplomové práce panu **Ing. Pavlovi Křížovi, Ph.D.** za příkladné vedení. Dále děkuji společnosti MoroSystems s.r.o. za možnost získání zkušeností nutných pro realizaci této práce. V neposlední řadě děkuji svým blízkým za podporu po celou dobu mého studia.

Anotace

Práce se především zabývá představením kontejnerové platformy Docker jako nástroje pro podporu vývoje, kontinuální integrace a nasazení softwarových aplikací na různá prostředí. První část této práce popisuje paradigma moderních cloudových řešení postavených na kontejnerech, kdy je klasická forma virtualizace postupně nahrazována a motivuje čtenáře k použití Dockeru v procesu vývoje aplikací. Následně tato pasáž volně navazuje na vysvětlení základních konceptů Dockeru a posléze přechází k detailnímu popisu a použití jednotlivých komponent. Toto téma přechází k ukázce dalších nástrojů souvisejících s Dockerem, které podporují nasazení a management. Třetí část a závěr se prakticky zaměřuje na příklady s cílem ukázat nasazení Dockeru při vývoji aplikací, správné přístupy a upozornit na typické problémy. Po přečtení práce se čtenáři dostane teoretického přehledu a znalostí jednotlivých komponent potřebných pro úspěšnou integraci Dockeru do budoucího vývoje aplikací, je mu také umožněno vyhnout se typickým chybám, se kterými se začátečníci běžně potýkají a zároveň pochytit osvědčené postupy.

Annotation

Title: Docker containers and their usage in software development

This thesis concentrates on the introduction of the Docker container platform as a tool to support the development, continuous integration and the deployment of the software applications in different environments. The first part of this thesis describes a paradigm of modern container-based cloud solutions, where the classic form of virtualization is gradually replaced, and also motivates readers to use Docker in the application development process. Furthermore, this part explains Docker´s basic concepts and subsequently describes in detail the use of individual components. The third part includes a practical example to demonstrate the deployment of the Docker when developing applications, the right approaches, and highlighting typical issues. After reading the work, the reader will have a theoretical overview and knowledge of each component needed to successfully integrate the Docker into future application development, which enables him to avoid common mistakes that beginners normally encounter.

Obsah

Seznam obrázků

Úvod

V posledních letech je možné se všude v řadě odborných článků dočíst o Dockeru jako kontejnerové platformě, která zažívá obrovský nárůst nejen mezi uživateli, ale i společnostmi, které na něj postupně přecházejí. Docker podporuje myšlenku cloud computingu, efektivně využívá prostředků, usnadňuje vývoj, distribuci a nasazení aplikací.

Důvody, proč Docker zavádět již během procesu vývoje aplikací, jsou čím dál tím vyšší nároky na úsporu času a financí, konzistentní prostředí, jednoduchá distribuce, úspora HW prostředků a mnoho dalších.

Použitím Dockeru je ve velké míře dosaženo úspory času a zjednodušení distribuce díky rychlosti instalace vývojového prostředí, následného testování a nasazení. Vývojáři se prostřednictvím něho mohou soustředit pouze na samotný vývoj a nemusí tak řešit instalaci lokálního prostředí, které může být zdlouhavé. Zároveň lidem zodpovědným za nasazení do produkčního prostředí (Operations) usnadňuje práci snížením míry rizika vzniku chyb lidským faktorem prostřednictvím zaručení identického prostředí na různých lokalitách. Popsat koncept Dockeru a způsob nasazení při vývoji aplikací, upozornit na začátečnické chyby a ukázat správné postupy k zajištění úspory času, prostředků a efektivní distribuce aplikací je cílem této diplomové práce.

Téma diplomové práce jsem si vybral především kvůli načerpaným zkušenostem během několika let ve společnosti MoroSystems s.r.o. získaných při migraci komplexních aplikací do cloudu a jejich kontejnerizací prostřednictvím platformy Docker. S Dockerem pracuji téměř denně, a proto bych se rád s mými zkušenostmi podělil i s ostatními čtenáři.

Cílem teoretické části je představení této platformy, vysvětlení důvodů, proč ji do portfolia používaných technologií a procesu vývoje ve svém projektu či organizaci zařadit a v neposlední řadě rozebrat důležité koncepty s ukázkami včetně zmínek o výhodách a nevýhodách použití.

V praktické části diplomové práce je cílem nejprve nastínit typické problémy, které mohou nastat při prvotní implementaci Dockeru do procesu vývoje, a poté zmínit návrhové vzory, osvědčené postupy apod. na praktických příkladech.

Dalším účelem je poskytnutí cenných rad, jak připravit vývojové prostředí takovým způsobem, aby podporovalo vývoj aplikací pro běh v Docker kontejnerech a na závěr zmínění dalších souvisejících nástrojů pro podporu nasazení a správu těchto kontejnerů.

Ze seznamu literatury, která je uvedena v závěru samotné práce, jsem hlavně čerpal z knihy The Docker Book od autora Turnbull J., kde jsou podrobně popsány jednotlivé komponenty a způsoby použití. Dalším důležitým zdrojem při zpracování mi byla oficiální dokumentace od společnosti Docker, Inc. Oba zdroje mi vyhovovaly stylem, jakým byly napsány a informacemi, které obsahovaly.

1 Paradigma moderních cloudových řešení

Cloud computing[1] je v posledních letech zodpovědný za poměrně velký převrat ve spojení s fungováním společností, které ke své činnosti využívají informační technologie. Výjimečnost cloudu nepochybně spočívá ve způsobu využití robustní infrastruktury. Jako jedna z možností, jak cloudovým technologiím rozdělovat fyzickou infrastrukturu do logicky oddělených virtuálních strojů, byla donedávna virtualizace.

Mezi hlavní výhody cloudu by měly patřit především vysoká škálovatelnost a spolehlivost. K tomu je ale zapotřebí, aby tyto vlastnosti podporovaly i samotné aplikace provozované v cloudu, např. pokud se aplikace zhroutí, musí být možné uživatele přesměrovat na její jinou instanci.

Moderní trendy cloud computingu se ale vydávají jiným směrem, než je klasická forma virtualizace, a oživují se tak pojmy, jako jsou kontejnery a kontejnerizace. Důvod je takový, že s postupem času se moderní vývoj životního cyklu softwaru stal obrovsky komplikovaným, rozmanitost dnešních softwarových sestav a hardwarových infrastruktur se výrazně zvýšila. Dokonce i jednoduchá webová aplikace může obsahovat databázový server, aplikační/webový server, každý potenciálně hostovaný na různých infrastrukturách se svým operačním systémem a na vývojových, zkušebních a produkčních prostředích. Když se k tomu přidá více složitosti v podobě škálování, load balanceru, analytické databáze a dalších závislostí, nastává u virtualizace problém ve složité údržbě, správě a dodávce softwaru. Dalšími známými nevýhodami virtualizace jsou především [1]:

- potřeba přenesení stávajících fyzických prostředí do virtuálních strojů,
- zvýšené licenční nároky a složitost,
- nízká dostupnost služeb na serverech,
- komplikované navyšování výkonu a kapacity,
- výpadky při zvyšování výkonu.

Cloud computing se rychle vyvinul do nového a výrazně rozšířeného IT ekosystému, a konkurence mezi poskytovateli je na mnoha frontách intenzivní. Kontejnerová technologie se jeví jako životaschopný další krok na cestě k otevřenému cloudu.

[1] Cloud computing je jednoduše dodávka výpočetní služeb, jako jsou servery, databáze, sítě, software a další, přes internet.

Přední promotér kontejnerových aplikací je v dnešní době bezpochyby společnost Docker, Inc. a její nástroj Docker, na jehož výhody a použití se tato práce zaměřuje. Jak již bylo několikrát zmíněno v různých článcích, kontejnery fungují jinak než VM[2] – je nesmysl snažit se je k takovému chování donutit. Cíl je opačný – kontejnery jsou nástroje pro nový styl v IT, správě i vývoji.

1.1 Cloud computing a virtualizace

V současné době je virtualizace založená na hypervizoru[3] nejčastěji používanou technologií pro virtuální prostředí. Je flexibilní a funguje pro libovolný hostitelský operační systém (OS), ale vede k přetížení paměti a CPU[4]. Nicméně virtualizační technologie přistupuje k fyzickému hardwaru pouze zprostředkovaně za pomoci hypervizoru. Na rozdíl od stávajícího virtualizačního přístupu, kontejnerizace nepoužívá tuto zprostředkující vrstvu, ale využívá služeb jádra operačního systému, což snižuje přetížení CPU, paměti, usnaďnuje škálování a přenositelnost díky malým kontejnerům [2]. Tímto je dosaženo určité izolace vůči ostatním procesům bez nutnosti provozovat celý operační systém pouze kvůli jedinému procesu [3].

1.2 Cloud computing a Docker

Docker je nepochybně novým trendem v oblasti virtualizace a Cloud computingu. Je důležité získávat a vysvětlovat výhody plynoucí z kontejnerizační techniky inspirované Dockerem nad široce využívanou a plně zralou virtualizační paradigmou [4]. Docker napomáhá tam, kde se virtualizace potýká s řadou problémů. Je potřeba si uvědomit, že zatímco u virtuálních strojů, které jsou nepřetržitě aktualizovány a „opečovávány“ do stavu, kdy na nich aplikace běží a fungují, kontejnery vznikají a zanikají. Za použití orchestračního nástroje je lze navíc automaticky řídit tak, že se během okamžiku automaticky vytvoří nový kontejner v případě, že některý zanikne. Docker kontejnery se považují za krátkodobé a jednorázové. Nejčastěji uváděným příkladem krátkodobého použití může být vývoj a testování. Kontejnery se v takovém případě stávají rysem rychlých, agilních a jednorázových prostředí, které jsou zapojeny do

[2] VM je zkratkou pro Virtual Machine, virtuální počítač vytvářející virtualizované prostředí mezi platformou počítače a operačním systémem.
[3] Hypervizor je označení pro jednu z technik virtualizace hardwaru počítače, která umožňuje na jednom PC spustit zároveň více operačních systémů.
[4] CPU je zkratkou centrální procesorové jednotky (procesoru).

procesu průběžné integrace (Continuous Integration[5]). V těchto případech použití je vytvořen a konfigurován Docker kontejner například pro spuštění požadovaných testů a poté je zahozen [5].

Hypoteticky si lze představit situaci, kdy zanikne takový virtuální stroj. Ve většině případů (pokud není vytvořena záloha) musí Operations[6] nebo vývojář (v závislosti na prostředí a kompetencích) provést kompletní reinstalaci celého prostředí, které trvá často dlouhou dobu. Proto se virtuální stroje udržují v tzv. aktualizovaném stavu, což má neblahý vliv na režii. Kromě toho nadměrně oceňovaný mechanismus virtualizace nedokáže splnit ani očekávání při řešení otázky přenositelnosti. To v praxi znamená, že VM vytvořené ve vývojových prostředích nefungují v testovacích, vývojových nebo produkčních prostředích bez podstatných úprav. Oficiální stránky Dockeru říkají, že vývojáři sestaví svou aplikaci jednou a pak si mohou být jistí, že poběží nezávisle kdekoliv. Operations mohou podobně konfigurovat servery a pak vědí, že na nich spustí libovolnou aplikaci.

[5] Continuous Integration je sada různých nástrojů a postupů, které urychlují vývoj, testují software a zároveň provádí nasazení aplikace a to automaticky několikrát denně.
[6] Operations je označení lidí, kteří se starají o provoz IT.

2 Kontejnerizace pomocí Docker

Vzhledem k tomu, že kontejnerová platforma Docker do velké míry využívá principů kontejnerů, o kterých bude později řeč, je první část této kapitoly věnována upřesnění několika skutečností, kterými se kontejnery odlišují oproti klasickým virtuálním strojům. V dalším průběhu čtení je navázáno na vysvětlení základních konceptů Dockeru, které jsou posléze detailně popsány. Konec kapitoly shrnuje důvody, proč Docker používat a v jakých případech se hodí přemýšlet o jeho zavedení a naopak, kdy je zcela nevhodné o této technologii v rámci inovace vývojového procesu uvažovat.

2.1 Kontejnery a virtuální stroje

I když je tato práce především zaměřena na kontejnerizaci pomocí platformy Docker a není cílem zabývat se do hloubky všemi možnými formami virtualizace, je žádoucí si oživit několik principů a pojmů, které je dobré znát ještě předtím, než s ní přijde kdokoliv do styku. Dalšími možnými virtualizačními platformami se například zabývá David Brych ve své diplomové práci [6] na str. 21.

V oblasti virtualizace mají kontejnery dlouhou historii. Na rozdíl od tradičních technologií virtualizace, kde jeden nebo více nezávislých strojů pracuje prakticky na fyzickém hardwaru prostřednictvím hypervizoru, jakožto zprostředkovací vrstvy, kontejnery místo toho běží v uživatelském prostoru nad jádrem operačního systému (dále jen OS), a tak tuto emulační vrstvu nepotřebují. Místo toho používají rozhraní pro běžná systémová volání a sdílí zdroje s hostitelským systémem, čímž se stávají více efektivními. Mohou být nastartovány a zastaveny během několika sekund, což u virtuálních strojů (dále jen VM) trvá mnohem déle. Obecně kontejnery zapouzdřují aplikaci a její závislosti, tj. konfigurační soubory, knihovny apod. Na první pohled se může zdát, že jsou jen jakousi jednoduchou formou VM, cosi kontejner obsahující samostatnou instanci OS, ve kterém jsou spouštěny aplikace, nicméně kontejnery mají v některých případech, které jsou u tradičních VM obtížné nebo zcela nemožné, řadu výhod.

Kontejnery zásadně rozvíjejí otázku, jakým způsobem vyvíjet, spouštět a distribuovat software. Distribuce kontejnerů má potenciál eliminovat velké množství chyb způsobených i nepatrnými změnami v prostředí. V celém procesu vývoje umožňují vývojářům nejen snadným způsobem vyvíjet a spouštět aplikace lokálně, ale dává jim i

větší jistotu identického prostředí, což v praxi znamená, že aplikace, která je spuštěna v kontejneru lokálně u vývojáře, bude s velkou pravděpodobností běžet stejným způsobem na produkčním prostředí [4].

Další nespornou výhodou je možnost spuštění několika izolovaných instancí na jediném hostiteli během pár sekund, čímž lze například dosáhnout vytvoření testovacího prostředí pro emulaci produkčního prostředí.

Nelze jednoznačně říci, zda používat kontejnery nebo VM, protože jejich použití a cíle jsou rozdílné. Účelem VM je plně napodobit prostředí cizím prostředím, zatímco účel kontejnerů tkví v přenositelnosti a samostatnosti aplikace. I přes dlouhou historii nedosáhly kontejnery velkého přijetí a velká část tomu je přisuzována právě kvůli jejich komplexnosti. Na první pohled se zdají být složité, je obtížné je nastavit, řídit a automatizovat. Cílem Dockeru je změnit tuto domněnku.

2.2 Základní koncepty Dockeru

Docker je nástroj, jehož zkladním konceptem je automatizace nasazení aplikací do kontejnerů vyvinutý týmem společnosti Docker, Inc. (dříve dotCloud Inc.) pod licencí Apache 2.0. Je určen k poskytování následujících služeb [4].

Jednoduchý způsob modelování reality

Kontejnerizaci aplikace s použitím Dockeru je možné provést během pár minut. Docker je rychlý a spoléhá se zde na model copy-on-write, takže změny v aplikaci jsou taktéž neuvěřitelně rychlé. Model copy-on-write je strategií sdílení a kopírování souborů pro maximální efektivitu. Pokud v nižší vrstvě obrazu existuje soubor nebo adresář a další vrstva potřebuje k jednomu z nich přístup ke čtení, použije pouze daný soubor. Poprvé, když další vrstva potřebuje soubor upravit (při vytváření obrazu nebo při spuštění kontejneru), se soubor do této vrstvy zkopíruje a upraví. Ve zkratce pouze to, co je třeba změnit, se změní. Tím se minimalizuje velikost každé z následujících vrstev. Spuštění většiny Docker kontejnerů trvá méně než jednu sekundu. Odstranění režie hypervizoru má za následek vysoký výkon kontejnerů, které lze snadno přenášet mezi hostitelskými počítači a co nejlépe tak využít zdroje.

Logické oddělení služeb

S nástrojem Docker dochází bezpochyby i k logickému oddělení služeb. Vývojáři se starají pouze o aplikaci běžící uvnitř kontejneru a Operations o správu kontejnerů. Docker je navržen tak, aby zajistil konzistenci tím, že zajistí vývojové prostředí, které pak odpovídá prostředí, do kterého jsou aplikace nasazovány. Tím se snižuje riziko chyb při nasazení do produkčního prostředí, protože je téměř vyloučeno, že aplikace, která běží na vývojovém prostředí, nebude běžet na jiném.

Podpora architektury orientované na služby

Docker podporuje architekturu orientovanou na služby a microservices[7]. Je velmi doporučováno, aby každý kontejner, který je spuštěn, obsahoval jednu běžící aplikaci nebo proces. To slouží k podpoře distribuovaného aplikačního modelu, kde aplikace nebo služby představují řadu několik vzájemně propojených kontejnerů. Výsledkem je snadná distribuce, ladění a zjišťování stavu aplikací. Mnoho uživatelů, kteří Docker používají, však toto doporučení opomíjejí a z jejich kontejnerů se pak postupem času stávají špatně spravovatelné celky, které jsou hůře udržovatelné.

Rychlý a spolehlivý vývojový životní cyklus

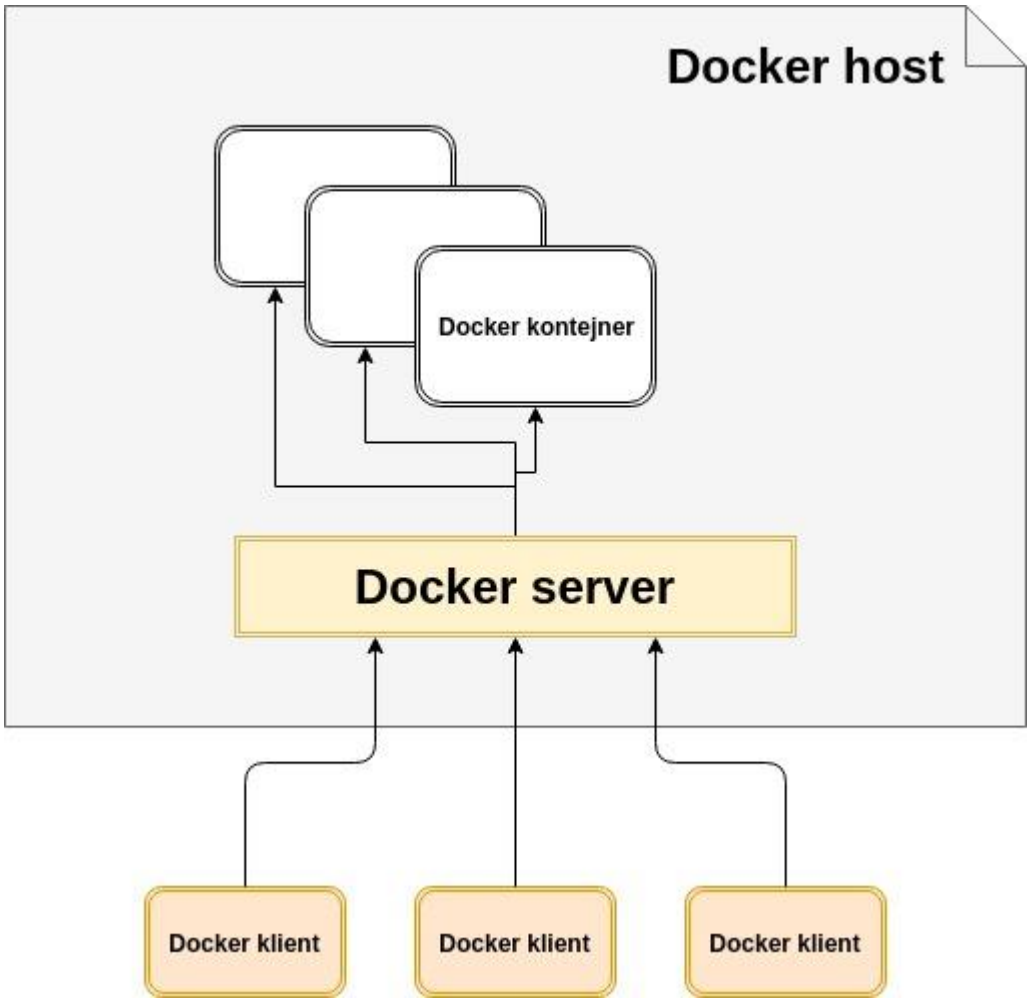
Úkolem Dockeru je, aby aplikace byly přenosné, snadno sestavitelné a snáze se s nimi pracovalo. Díky tomu se snižuje doba cyklu mezi vývojem a testováním, nasazením a používáním. Při vytváření kontejnerů, ve kterých běží aplikace, je velmi důležité porozumět stavebním blokům a principům této kontejnerové platformy. Záměrem kapitoly je vysvětlení základních komponent Dockeru především pro ty, kteří se s touto platformou setkávají poprvé, ale i jeho připomenutí pro pokročilejší uživatele. Další části se pak zaměřují na efektivní způsob využívání těchto komponent. Podrobnostmi se zabývá kapitola č. 2.4.

[7] Microservices je architektonický styl, který strukturuje aplikaci jako kolekci volně provázaných služeb.

2.2.1 Server a klient

V samém jádře celé architektury se tyčí server v komunitě též zvaný jako démon. V některých publikacích je o démonovi psáno jako o tzv. Docker Engine, který je zodpovědný za vytváření, běh a monitorování kontejnerů, sestavování a uchovávání obrazů.

Docker klient slouží pouze ke komunikaci s démonem přes vystavené rozhraní, které uživatel standardně používá prostřednictvím dodávané knihovny docker s příkazovou řádkou. Klienta a démona je možné spustit na stejném stroji a rovněž je i možné spustit klienta lokálně a připojit se k démonovi na vzdálenému serveru. Jak taková architektura může vypadat, je znázorněno na následujícím obrázku č. 1.



Obr. 1 - Architektura Docker
Zdroj: vlastní zpracování

2.2.2 Dockerfile

Dockerfile je jednoduchý textový soubor obsahující jednotlivé kroky v podobě instrukcí. Je to také jeden ze způsobů, jak vytvořit obraz. I když vytvoření obrazu není použitím této metody podmíněno, je velmi doporučováno, neboť díky němu se obrazy stávají více flexibilními a silnými. Jinou metodu, kterou lze obraz vytvořit, popisuje kapitola č. 2.4. V nadcházejících kapitolách je však vytváření obrazu realizováno výhradně pomocí Dockerfile.

2.2.3 Obraz

Obrazy jsou základními stavebními bloky životního cyklu Dockeru, z kterých jsou dále vytvářeny kontejnery. Skládají se z jednotlivých vrstev, které se tvoří z instrukcí. Tyto vrstvy slouží pouze ke čtení, ale ve chvíli, kdy je spuštěn kontejner, Docker připojuje na vrchol těchto vrstev prázdnou read-write vrstvu, kde jsou spouštěny jednotlivé procesy. Pokud dojde k provedení nějaké změny, například zkopírování existujícího souboru do jiného umístění, provede se to takovým způsobem, že se z read-only vrstvy vytvoří kopie souboru. Verze souboru z read-only vrstvy stále existuje, ale je schována pod kopií.

Na počátku procesu vytváření obrazů dochází k rozhodnutí, jakou vhodnou metodu zvolit, čímž se utváří možnost vytvořit obrazy, které jsou lehce přenosné, sdílitelné, jednoduše uchovávatelné a aktualizovatelné.

2.2.4 Registr

Registry poskytují možnost uchovávat a sdílet obrazy. Existují dva typy registrů: veřejné a soukromé. Veřejný registr zvaný Docker Hub spravuje společnost Docker, Inc. a kdokoliv si zde může vytvořit účet, používat ho ke sdílení a uchovávání svých vlastních obrazů. Docker Hub obsahuje desítky tisíc obrazů od různých uživatelů a společností. Pokud je potřeba vyhledat obraz pro Apache Tomcat[8], Redis[9] databázi apod., jsou zde tyto obrazy dostupné. Kromě těchto obrazů lze zde samozřejmě nalézt i mnohem více.

Většinou je však v úmyslu uchovávat obrazy obsahující citlivé informace a zdrojové kódy, u kterých je požadováno sdílení pouze v rámci týmu nebo organizace.

[8] Apache Tomcat je open-source aplikační server založený na jazyce Java.
[9] Redis je open-source NoSQL key-value databáze.

K tomu na Docker Hub existují vedle veřejných registrů také soukromé registry. Mimo oficiálního registru je možné založit a využít vlastní soukromé registry, o kterých je zmínka v kapitole 2.4.

2.2.5 Kontejner

Kontejner je instancí obrazu, kdy obraz definuje, co kontejner obsahuje, jaký proces v něm má běžet při jeho spuštění a řadu dalších konfiguračních souborů. Každý obraz a jeho vrstvy jsou pouze ke čtení, ale v okamžiku, kdy je z obrazu vytvořen kontejner, je přidána tzv. horní vrstva pro čtení a zápis, v němž je aplikace spuštěna. Z jednoho obrazu je tak možné spustit několik (teoreticky neomezené množství) stejných kontejnerů.

2.3 Instalace a příprava prostředí

V kapitole věnované instalaci a přípravě prostředí jsou stručně zmíněné důležité předpoklady potřebné k instalaci aplikace Docker. Bohužel instalace Dockeru není přímočarou záležitostí. Především závisí na tom, na kterém operačním systému poběží. V další fázi jsou vylíčeny informace o nabízených edicích, pro koho jsou určené, a v neposlední řadě informace o rozdílnosti instalace na různých platformách. Cílem není popsat postup instalace krok za krokem pro každou platformu hlavně z důvodu toho, že se Docker velmi rychle vyvíjí a při čtení této práce nemusí už být postupy aktuální.

Přesné postupy instalací pro různé platformy jsou k nalezení na adrese oficiální dokumentace https://docs.docker.com/engine/installation/.

V době, kdy je psána tato práce, je používána aktuální verze Dockeru 18.03.1-ce, na kterou je v rámci dalších kapitol odkazováno. Veškeré příkazy jsou testovány oproti této verzi a je velmi důležité podotknout, že jsou prováděny na linuxovém operačním systému založeném z větší části na Ubuntu. Uživatelé, kteří již mají nainstalovaný Docker v této verzi a vyšší, mohou bezpečně tento krok přeskočit a přejít na další kapitolu.

2.3.1 Community edice

Dříve známý Docker Engine byl v průběhu vývoje přejmenován na Docker Community Edition (dále jen CE) a jak název napovídá, jedná se o komunitou podporovanou verzi, která je k dispozici zcela bezplatně. Dodává se ve dvou verzích: Edge a Stable. Verze

Edge je povětšinou vydávána každý měsíc s nejnovějšími funkcemi a Stable se vydává čtvrtletně. Zatímco Edge pravidelně získává aktualizace zabezpečení a opravy chyb pro aktuální verzi, Stable verze dostává podobné aktualizace čtyři měsíce po prvním vydání. Tento cyklus aktualizací poskytuje uživatelům dostatečné okno pro plánování upgradů ze starších verzí [8]. Docker CE je ideální pro vývojáře a malé týmy, které s Docker platformou teprve začínají a experimentují s aplikacemi založenými na kontejnerech. Pro mnoho populárních infrastrukturních platforem, jako jsou desktopové, cloudové a open source operační systémy, Docker CE poskytuje instalátor pro jednoduchou a rychlou instalaci, takže je možné začít okamžitě vyvíjet [9].

2.3.2 Enterprise edice

Docker Enterprise edice (dále jen EE) se dodává ve třech verzích: základní (basic), standardní (standard) a pokročilá (advanced). Základní edice je dodávána s platformou Docker pro elementární podporu a certifikaci, zatímco standardní a pokročilá verze přidávají další funkce, jako jsou správa kontejnerů (Docker Datacenter) a Docker Security Scanning[10]. Docker EE je podporován společnostmi Alibaba, Canonical, HPE, IBM, Microsoft a sítí regionálních partnerů. Pro ty, kteří chtějí tuto verzi vyzkoušet, je možnost bezplatného stažení zkušební verze z oficiálních stránek. Docker nabízí také certifikační program, který pomáhá výrobcům třetích stran zajistit, aby jejich produkty spolupracovaly s Docker EE [8].

Docker EE je především určen pro podnikové vývojové týmy a IT týmy, kteří vytvářejí, dodávají a provozují kritické produkční aplikace. Docker je v této verzi integrován, certifikován a podporován tak, aby poskytoval podnikům zabezpečenou kontejnerovou platformu. Další informace o platformě Docker EE jsou k dispozici v dokumentu https://www.docker.com/enterprise-edition.

2.3.3 Podporované platformy

Docker je v současné době podporován nejrůznějšími linuxovými distribucemi, jako např. Ubuntu, Debian, CentOS, Fedora, Oracle Linux a je také podporován na několika cloudových platformách včetně Amazon EC2, Rackspace Cloud, Google Compute Engine atd. Pomocí virtuálního prostředí je možná instalace a provoz Dockeru

[10] Docker Security Scanning je služba, která provádí skenování obrazů na binární úrovni ještě před jejich nasazením, průběžně kontroluje chyby zabezpečení a poskytuje upozornění.

v systémech macOS a Microsoft Windows pomocí kolekce komponent, která instaluje vše, co je potřeba. Tyto systémy zpravidla obsahují malý virtuální stroj, který je dodáván jako součást instalace.

2.3.4 Požadavky na instalaci

Docker nemá mnoho požadavků na instalaci, ale přece jen jsou některé předpoklady, které je nutné před instalací ověřit, případně splnit. Instalace je podmíněna během na poměrně moderním linuxovém jádru (verze 3.10 nebo vyšší v době psaní této práce). Je velmi doporučováno udržovat jádro aktualizované. Dalším požadavkem je vlastnit systém se 64bitovou architekturou. Postup instalace na různých linuxových distribucích je velmi podobný a přesný postup lze zjistit na stránkách https://docs.docker.com/install/. Jiným způsobem však probíhá instalace na operačních systémech Windows a macOS. V následujících odstavcích je pro představu nastíněn postup instalace, který se může s odlišnými verzemi Dockeru lišit.

Instalace Docker na operačním systému Ubuntu

Jak již bylo předesláno, instalace má několik prerekvizit, které je potřeba splnit. Pro instalaci Dockeru na OS Ubuntu je vyžadována 64bitová verze systému a aktuálně jsou podporované následující verze [10]:

- Bionic 18.04 (LTS)
- Artful 17.10
- Xenial 16.04 (LTS)
- Trusty 14.04 (LTS)

Instalace nové verze probíhá ve dvou krocích – přípravy repozitáře a samotné instalace pomocí balíčkovacího systému APT. Nejprve je nutné nastavit správný repozitář, odkud se aplikace později stáhne.

Tento proces je rozdělen do celkem čtyř kroků.

1. Následující příkazy v ukázce č. 1 provádí aktualizaci dostupných a instalaci potřebných balíčků pro přístup k Docker repozitáři přes HTTPS pomocí balíčkovacího systému APT.

```
$ sudo apt-get update
$ sudo apt-get install \
    apt-transport-https \
    ca-certificates \
    curl \
    software-properties-common
```

Ukázka 1 - Instalace balíčků pro přístup k Docker repozitáři

2. Dalším krokem v ukázce č. 2 je přidání oficiálního GPG[11] klíče pro Docker a ověření, zda tento klíč existuje s patřičným otiskem. Další příkaz vyhledá klíč na základě otisku – to je provedeno vyhledáním posledních 8 znaků z otisku.

```
$ curl -fsSL https://download.docker.com/linux/ubuntu/gpg | sudo
apt-key add -
$ sudo apt-key fingerprint 0EBFCD88
```

Ukázka 2 - Přidání GPG klíče

3. Posledním příkazem se nastaví stabilní repozitář a provede instalaci nejnovější komunitní verze nástroje Docker. Díky tomu, že se používá balíčkovací systém APT, je syntaxe stejná jako při instalaci jakéhokoliv jiného balíčku. Nedříve je nutné aktualizovat seznam balíčků a pak je možné bezpečně nainstalovat aplikaci, jak je demonstrováno v ukázce č. 3 [11].

```
$ sudo add-apt-repository \
   "deb [arch=amd64] https://download.docker.com/linux/ubuntu \
   $(lsb_release -cs) \
   stable"
$ sudo apt-get update
$ sudo apt-get install docker-ce
```

Ukázka 3 - Nastavení repozitáře a instalace Docker CE

[11] GPG umožňuje šifrovat a podepisovat data a datovou komunikaci, obsahuje sofistikovanou správu kryptografických klíčů a systém veřejných adresářů tyto klíče shromažďující a distribuující.

Instalace Docker na systémech Windows a macOS

Stejně tak, jako pro Linuxové systémy, existují i pro instalaci Dockeru na systémech Windows a macOS určité minimální požadavky, které je vhodné vzít v potaz. Mezi těmito požadavky stojí za zmínku minimální podporovaná verze systému, kterou je v aktuální verzi Dockeru 64bitový systém Windows 10 (Pro, Enterprise nebo Education) a macOS 10.11 a výše. Pokud hostitelský počítač nesplňuje některý z požadavků, lze nainstalovat tzv. Docker Toolbox, desktopovou aplikaci, která zajišťuje rychlou a intuitivní instalaci. Obsahuje sadu komponent (Docker Engine, Docker Compose, Docker Machine, Kitematic, VirtualBox atd.) včetně malého virtuálního stroje s podporou příkazové řádky, který je nainstalován na hostitelském počítači a poskytuje prostředí pro Docker. Docker Toolbox spouští „boot2docker“ linuxovou distribuci, ve které je umístěn Docker Engine. Linuxová distribuce je v balíčku obsažena z důvodu doplnění potřebných funkcí linuxového jádra [11].

Druhým způsobem instalace, když jsou splněny instalační požadavky (a není tím pádem nutné instalovat Docker Toolbox), je stažení patřičné verze Dockeru z oficiálních stránek, která je pro Windows ve formátu „exe“ a macOs „dmg“. Samotná instalace pak spočívá pouze ve spuštění staženého instalátoru a následování instrukcí.

2.4 Obrazy a registry

V předchozí kapitole došlo k vysvětlení toho, co znamená pojem obraz a registr. Dalším účelem je zaměřit se na tyto komponenty z hlediska jejich použití. Následuje ukázka, jakými metodami lze obrazy vytvářet, jak má vypadat správně napsaný Dockerfile a seznámení se s důležitými a patřičnými příkazy pro vhodné užití. V neposlední řadě bude vytvořen vzorový obraz, který se pro další uchování nahraje do registru.

Už na začátku bylo zmíněno, že obrazy jsou jakési šablony pouze pro čtení, ze kterých jsou vytvářeny kontejnery. Každý obraz se typicky skládá z několika vrstev, které jsou skládány pomocí tzv. „union file systems“ neboli souborových systémů umožňujících transparentní překrývání souborů a adresářů. Výsledkem toho je pak jednotný souborový systém. Příkladem tohoto principu může být aktualizace (kopírování souboru) v obrazu. V této situaci se vytváří vrstva nová a neobnovuje se celý obraz. Celá distribuce je tím výrazně rychlejší a snazší oproti VM.

Každý obraz začíná od základního obrazu, např. pro operační systém Debian, který je volně dostupný v registru Docker Hub. Lze si však také zvolit vlastní obraz jako nový základ pro obraz jiný. Představit si lze obraz Java jako základ všech našich aplikací. Obrazy jsou pak tvořeny z těchto základních obrazů pomocí jednoduché sady kroků, které se nazývají instrukce. Každá instrukce tvoří jednu vrstvu v obrazu. Jak efektivně pracovat s instrukcemi, aby bylo dosaženo co nejmenšího počtu vrstev a z toho vyplývající eliminace výsledné velikosti obrazu, zmiňuje kapitola 2.4.4.

2.4.1 Vytvoření obrazu

V mnoha aspektech je vytvoření obrazu velmi důležitou součástí celého procesu dockerizace. Definuje totiž, jakým způsobem se bude chovat kontejner, který je z něho následně vytvářen. Špatně navržený obraz může později způsobit velké problémy, které mají za následek přetvoření obrazu nebo v nejhorším případě vytvoření obrazu nového, což v dockerizaci komplexních aplikací může vést k prodražení a zpoždění dodávky celého řešení. Těmito možnými problémy se zabývá a upozorňuje na ně kapitola Typické problémy implementace, se kterými se lze v praxi setkat, a kterým je dobré se vyvarovat.

Metody vytvoření obrazu

Pro případ, kde je žádoucí volně dostupné obrazy z registru nějakým způsobem modifikovat, nebo vytvořit zcela nové, které budou založené na některém z existujících obrazů, existují celkem dvě metody, jak tyto obrazy vytvořit:

- prostřednictvím příkazu *commit*,
- prostřednictvím příkazu build a souboru *Dockerfile* [4].

Vytvoření obrazu prostřednictvím příkazu commit

Metoda vytvoření obrazu prostřednictvím příkazu commit je způsob, kde je již z existujícího kontejneru vytvořen obraz, ve kterém oproti původnímu obrazu došlo k nějaké změně. Analogií může být přemýšlení o tomto způsobu jako o provádění commitu v některém z verzovacích systémů [4]. Existují situace, kdy je použití příkazu *docker container commit* k vytváření obrazů v některých případech užitečné z hlediska krátkého času nasazení, možnosti ladění a experimentování. Ladění obrazů je vhodným případem, kde je použití příkazu commit k vytvoření obrazu přijatelné. Výhodou je, že v této situaci vždy existuje možnost vrátit se zpět k původnímu obrazu a z něho vytvořit kontejner nový.

Obecně používání příkazu *docker container commit* není vhodné především z důvodu špatné udržovatelnosti. Další nevýhodou je skutečnost, že změny prováděné touto metodou se špatně dokumentují a nelze je reprodukovat [10]. Sofistikovaným a doporučovaným řešením je vytvářet obraz metodou příkazu *docker image build* s využitím souboru Dockerfile.

Vytvoření obrazu prostřednictvím souboru Dockerfile

Metoda vytvoření obrazu prostřednictvím souboru Dockerfile je velmi doporučovanou metodou vzhledem k její opakované použitelnosti, udržovatelnosti a transparentnosti. Dockerfile používá jazyk DSL[12] (Domain Specific Language) spolu s instrukcemi k vytvoření výsledného obrazu. Jakmile existuje jednou vytvořený Dockerfile, je možné příkazem *docker image build* vytvořit jeden a více totožných obrazů z instrukcí v něm obsažených. O principu a možnosti použití tohoto způsobu vytváření obrazů informuje následující samostatná kapitola.

[12] DSL je zkratkou Domain Specific Language, která definuje jazyk specifický pro konkrétní aplikační doménu.

2.4.2 Dockerfile

O vhodnosti použití metody vytváření obrazů pomocí souboru Dockerfile bylo již napsáno v předešlé části. Pro lepší pochopení principu, jak vytvořit obraz ze souboru Dockerfile, je vyobrazen příklad, který zachycuje vytvoření obrazu pro spuštění jednoduché JAVA aplikace.

Prvním krokem je vytvoření pracovního adresáře a souboru Dockerfile, jehož obsah je zobrazen v ukázce č. 4 níže. Ve stejném adresáři se vyskytuje (došlo k jejímu zkopírování) jednoduchá aplikace sloužící k demonstračním účelům spuštění libovolné aplikace. Její zdrojový kód a návod pro zkompilování a vytvoření souboru *HelloWorld.jar* je možný stáhnout z repozitáře:
https://github.com/sipekmichal/custom.java.app.helloworld.

```
$ mkdir custom-java-app
$ touch custom-java-app/Dockerfile
$ cp HelloWorld.jar custom-java-app
```

Ukázka 4 - Příprava kontextu sestavení

Adresář *custom-java-app* se stal prostředím pro vytvoření obrazu, kterému se v terminologii Dockeru říká build kontext (dále jen kontext nebo kontext sestavení). Soubor Dockerfile (ukázka č. 5) obsahuje následující instrukce.

```
FROM openjdk:8-jdk
LABEL maintainer="mail@sipekmichal.cz"
ENV REFRESHED_AT 2017-10-08
RUN apt-get update && apt-get upgrade -y && \
    mkdir /opt/custom-java-app/
COPY HelloWorld.jar /opt/custom-java-app/
CMD ["java","-jar","/opt/custom-java-app/HelloWorld.jar"]
```

Ukázka 5 - Dockerfile pro spuštění kontejneru s JAVA aplikací

Je důležité, v jakém pořadí jsou instrukce napsané. Docker totiž zpracovává a spouští jednotlivé instrukce v pořadí odshora dolů dle obecně daného postupu v následujícím pořadí:

- Docker spouští kontejner z obrazu.
- Instrukce změní obsah kontejneru.
- Docker spouští ekvivalent příkazu docker commit k vytvoření nové vrstvy.
- Docker spouští nový kontejner z tohoto nového obrazu.
- Zpracovává se další instrukce v souboru Dockerfile a celý proces se opakuje do doby, než jsou všechny instrukce zpracovány.

Pokud tedy provádění překladu Dockerfile z nějakého důvodu selže (například selže instrukce kvůli nedostatečnému oprávnění), je možné spustit kontejner z obrazu, který se vytvořil před tímto selháním. Zde je vidět, že metoda vytváření obrazu pomocí souboru Dockerfile dává taktéž možnost pro ladění kontejneru. Ladění se pak nejčastěji provádí takovým způsobem, že se spustí kontejner, který se vytvořil před selháním oné instrukce. V něm je tato instrukce explicitně provedena, a tím je snadným způsobem dosaženo zjištění důvodu selhání [4].

Rozbor Dockerfile souboru

Prvním rozborem a instrukcí v souboru Dockerfile je instrukce *FROM*, která specifikuje základní obraz, kde je za dvojtečkou specifikována verze. Všechny Dockerfile soubory musí obsahovat instrukci *FROM* jako první neokomentovanou instrukci, která definuje výchozí rodičovský obraz. Nad tímto obrazem se poté vykonávají specifikované instrukce. První zmíněnou instrukcí je *LABEL* vytvářející metadata obrazu. V tomto příkladu *LABEL* nahrazuje dříve používanou instrukci *MAINTAINER*, jejíž použití je velmi vhodné pro zveřejnění kontaktu na autora obrazu. Instrukce *ENV* je užívána k definování proměnné a je dostupná pouze v rámci procesu sestavování obrazu (docker image build). Zde je instrukce *ENV* vhodně použitá pro tzv. kešování, jehož princip je popsán na konci kapitoly. Instrukce *RUN*, jak již z názvu vypovídá, spouští příkaz uvnitř kontejneru. Posledním příkazem v souboru Dockerfile je *CMD*, který zajišťuje spuštění definovaného obsahu v okamžiku spuštění kontejneru. Ačkoliv se na první pohled může zdát, že totéž provádí instrukce *RUN*, ve skutečnosti se liší ve způsobu použití. O dalších instrukcích dostupných nejenom v tomto příkladu pojednává kapitola 2.4.5. Ve chvíli, kdy je Dockerfile hotový, je připraven k sestavení obrazu příkazem z ukázky č. 6.

```
$ sudo docker image build -t sipekmichal/custom-java-app .
Sending build context to Docker daemon 2.048 kB
Step 1/6 : FROM openjdk:8-jdk
8-jdk: Pulling from library/openjdk
...
Status: Downloaded newer image for openjdk:8-jdk
 ---> a30a1e547e6d
Step 2/6 : LABEL maintainer="mail@sipekmichal.cz"
 ---> Running in f268a1269e87
Removing intermediate container f268a1269e87
 ---> ab9d9f0a2bb1
Step 3/6 : ENV REFRESHED_AT 2017-10-08
 ---> Running in 1c76e3a968d9
Removing intermediate container 1c76e3a968d9
 ---> cba25645f839
...
Step 5/6 : COPY HelloWorld.jar /opt/custom-java-app/
 ---> f382cc4d7b94
Step 6/6 : CMD ["java","-jar","/opt/custom-java-app/HelloWorld.jar"]
 ---> Running in 222df966f00f
Removing intermediate container 222df966f00f
 ---> 9e90dd137c6f
Successfully built 9e90dd137c6f
Successfully tagged custom-java-app:latest
```

Ukázka 6 - Proces sestavení obrazu

Jak je z předchozí ukázky patrné, pro sestavení obrazů ze souboru Dockerfile se používá příkaz *docker image build*, za nímž mohou existovat další volby. Zde byla použita volba *-t* , která slouží pro pojmenování obrazu, resp. vytvoření tagu. Vytvořil se obraz s názvem *custom-java-app:latest*. Tím, že nebylo specifikováno konkrétní označení verze, automaticky se použilo označení latest. Je však velmi doporučováno vždy verzi specifikovat, neboť napomáhá k jasné identifikaci obrazu obzvláště v případech, kdy na prostředí běží aplikace v kontejneru vytvořeném z takového obrazu. V obvyklých situacích, kdy se objevují chyby v aplikaci, může QA[13] oddělení pohotově předat vývoji informaci o verzi, ve které dochází k chybám. Kompletní seznam možností, které lze v příkazu zadat, je dostupný na oficiálních stránkách dokumentace.

Od verze Dockeru 1.5.0 lze specifikovat relativní cestu k souboru Dockerfile prostřednictvím volby *-f*. Platí zde stále pravidlo, že je nutné spouštět příkaz z kontextu sestavení. Dockerfile také podporuje komentáře, které se hodí především při vývoji. Všechny řádky, které začínají znakem #, jsou považovány za komentáře a při sestavování jsou ignorovány.

[13] QA je zkratkou pro Quality assurance a způsob, jak předcházet chybám, vadám a zajišťovat kvalitu služeb nebo produktů.

Z předchozí ukázky *custom-java-app:latest* je možné si povšimnout procedury vytváření obrazu. Zde je vidět, že se kontext odesílá Docker démonovi. Další informací, kterou lze z výpisu sestavení obrazu vyčíst, je vytvoření nového obrazu s každou zpracovanou instrukcí, kterému je přiřazen jedinečný identifikátor, který je vrácen jako finální výsledek celého procesu.

Selhání instrukce a možnost ladění

Na začátku kapitoly 2.4.2 bylo lehce zmíněno, co dělat v situaci, kdy některá z instrukcí selže. Následující postup ladění se především hodí v procesu kontejnerizace aplikací v rámci sestavování souboru Dockerfile a následně obrazu, protože zde existuje možný výskyt chyb při jeho ladění. Aby byl princip dobře vysvětlen, je v ukázce č. 7 předchozí Dockerfile záměrně modifikován, aby jedna z jeho instrukcí byla chybná.

```
FROM openjdk:8-jdk
LABEL maintainer="mail@sipekmichal.cz"
ENV REFRESHED_AT 2017-10-08
RUN apt-get ~~u~~pdate && apt-get upgrade -y && \
    mkdir /opt/custom-java-app/
COPY HelloWorld.jar /opt/custom-java-app/
CMD ["java","-jar","/opt/custom-java-app/HelloWorld.jar"]
```

Ukázka 7 - Záměrně chybný Dockerfile

V souboru došlo k pozměnění příkazu *update* odstraněním písmene u. Následně je v ukázce č. 8 opakován pokus o sestavení obrazu, kdy dochází k selhání instrukce *RUN*.

```
$ sudo docker image build -t sipekmichal/custom-java-app .
Sending build context to Docker daemon   5.12kB
Step 1/6 : FROM openjdk:8-jdk
 ---> a30a1e547e6d
Step 2/6 : LABEL maintainer="mail@sipekmichal.cz"
 ---> Using cache
 ---> ab9d9f0a2bb1
Step 3/6 : ENV REFRESHED_AT 2017-10-08
 ---> Using cache
 ---> cba25645f839
Step 4/6 : RUN apt-get pdate && apt-get upgrade -y &&     mkdir
/opt/custom-java-app/
 ---> Running in 8d82cccbe5b4
The command '/bin/sh -c apt-get pdate && apt-get upgrade -y &&
mkdir /opt/custom-java-app/' returned a non-zero code: 100
```

Ukázka 8 - Demonstrace selhání instrukce RUN

V takové situaci lze tento vyskytnutý problém odladit díky poslednímu úspěšně vytvořenému obrazu s identifikátorem cba25645f839.

Ladění se provádí vytvořením kontejneru z tohoto obrazu, kde se v příkazové řádce spustí incidenční příkaz ke zjištění, co se uvnitř doopravdy děje, viz. ukázka č. 9.

```
sudo docker container run -i -t cba25645f839 /bin/bash
root@d07a262fa3a3:/# apt-get pdate && apt-get upgrade -y &&     mkdir
/opt/custom-java-app/
E: Invalid operation pdate
root@d07a262fa3a3:/#
```

Ukázka 9 - Ladění kontejneru

Z tohoto kroku jasně vyplývá, že je příkaz *pdate* chybný. Zkušeným pohledem a testováním se lze dostat pomocí drobné úpravy ke správnému výsledku. Následně se aktualizuje soubor Dockerfile a je vykonán další pokus o sestavení obrazu.

Kešování

Docker ve výchozím nastavení využívá při sestavování obrazu v každém kroku prováděného příkazu systém kešování, kdy jsou tyto obrazy ukládány do mezipaměti. V předešlé ukázce si lze povšimnout informace *Using cache*, která říká, že byl tento obraz v minulosti sestaven, tudíž je přeskočen a nedochází k jeho opětovnému sestavení. Zde je patrné, že výchozím bodem je krok 4, protože kroky 1 až 3 jsou v mezipaměti sestavené. Tím, že se předchozí kroky nezměnily, má za následek ušetření spoustu času v procesu sestavování finálního obrazu. Jakmile by ale došlo k drobné změně v krocích 1 až 3, pak se obraz znovu sestavuje od první instrukce v souboru Dockerfile.

Pro některé žádoucí případy existuje možnost vypnutí kešování pomocí *--no-cache* příznaku předaného v rámci *docker image build* příkazu. Sestavování obrazu se tedy vždy provádí od úplného začátku, tj. prvního příkazu. Podle Dockerfile souboru vytvořeného výše se dá při sestavování obrazu uvažovat o vypnutí keše vzhledem ke zde vyskytujícímu se příkazu *apt-get update* a *apt-get upgrade*. V rámci vývoje a laborování může být totiž vždy chtěné provést aktualizaci databáze dostupných balíčků a udržovat aplikace v současných verzích.

To není samozřejmě vhodné na produkčním prostředí, protože se aktualizace provede při každém sestavení obrazu. Existuje elegantní řešení na úrovni vytvořeného Dockerfile. Předchozí vytvořený Dockerfile pro demonstraci používá proměnnou (instrukce ENV) s názvem *REFRESHED_AT*, jejíž hodnotou je datum poslední aktualizace.

Úpravou hodnoty a následným spuštěním příkazu *docker image build* dochází k tomu, že je obraz od kroku 3 znovu sestavován a aktualizován, jak je možné zaregistrovat v ukázce č. 10.

```
Step 1/6 : FROM openjdk:8-jdk
 ---> a30a1e547e6d
Step 2/6 : LABEL maintainer="mail@sipekmichal.cz"
 ---> Using cache
 ---> ab9d9f0a2bb1
Step 3/6 : ENV REFRESHED_AT 2018-01-08
 ---> Running in b8b820c782da
Removing intermediate container b8b820c782da
 ---> d7f42c84c8d2
Step 4/6 : RUN apt-get update && apt-get upgrade -y &&     mkdir
/opt/custom-java-app/
 ---> Running in 8e91bebab7c0
```

Ukázka 10 - Využití kešování při sestavování obrazu

Tímto prostřednictvím lze vytvářet tzv. šablony základních obrazů a v případě, kdy je potřebné aktualizovat aplikace, stačí pouze změnit datum a provést nové sestavení. Další obrazy, které jsou potomky této šablony, se při sestavení taktéž aktualizují. Není třeba se spoléhat na to, že došlo k vypnutí kešování.

Build kontext

Adresáři, ve kterém se nachází Dockerfile s ostatními soubory se v terminologii Dockeru říká build kontext, jak již bylo zmíněno dříve. Kontextem se rozumí celý obsah adresáře, který se odesílá Docker démonovi, aby ho mohl zpracovat a sestavit z něho výsledný obraz. Docker démon má tedy přímý přístup ke všem kódům, souborům nebo jiným údajům, které je nutné zahrnout do výsledného obrazu. V takovém adresáři mohou být i soubory (například návody, zdrojové kódy aplikace apod.), které nemají žádný vliv na vytváření obrazu, a je tedy nežádoucí všechen obsah odesílat démonovi ke zpracování. Pokud se v kořeni kontextu nachází soubor *.dockerignore*, který je interpretovaný jako seznam souborů, adresářů atd., zabraňuje odesílání definovaného obsahu démonovi. Analogií takového použití je soubor *.gitignore* v distribuovaném verzovacím systému Git[14].

[14] Git je název systému správy verzí při vývoji software. Pomocí Gitu lze revidovat jednotlivé změny v kódu, navrátit k určité verzi jednotlivých souborů, integrovat změny souborů při spolupráci více lidí na stejném kódu apod.

2.4.3 Další operace s obrazy

Stahování obrazů

Jednou ze základních operací s obrazy je stahování obrazů. Vytvořením kontejneru prostřednictvím příkazu *docker container run* z obrazu, který na hostitelském počítači neexistuje, dochází k tomu, že Docker tento obraz explicitně hledá v jeho registru Docker Hub. Pokud v registru takový obraz existuje, dochází k jeho stahování. Navíc pokud není v příkazu *docker container run* uveden tag, který by definoval verzi obrazu, Docker stáhne nejnovější verzi označovanou jako *latest* a spustí z něho kontejner. Tento obraz je pak k dispozici na hostitelském počítači a lze ho zobrazit příkazem *docker images*.

Stažení obrazu se obvykle provádí pomocí příkazu *docker image pull*, který stáhne vybraný obraz předaný v rámci tohoto příkazu a je předem připravený k dalším operacím. Princip je stejný jako v předchozím případě s tím rozdílem, že je tu výhoda, která spočívá v rychlosti vytvoření kontejneru. Kontejner se z takového obrazu vytvoří během několika sekund, protože obraz se již nachází v hostitelském počítači a nedochází k jeho stahování během toho, co byl zadán příkaz *docker container run*. Jak takový proces vypadá, znázorňuje příklad v ukázce č. 11.

```
$ sudo docker image pull mysql
Using default tag: latest
latest: Pulling from library/mysql
2a72cbf407d6: Pull complete
8181cde51c65: Pull complete
Digest:
sha256:691c55aabb3c4e3b89b953dd2f022f7ea845e5443954767d321d5f5fa394e28c
Status: Downloaded newer image for mysql:latest
```

Ukázka 11 - Stažení obrazu MySQL

V této ukázce není obraz více specifikován např. jeho verzí, proto dochází ke stažení nejnovější verze *latest*. Pro stažení obrazu mysql verze 8.0.4 se použije příkaz *docker image pull mysql:8.0.4*. Že došlo ke stažení obrazů, lze ověřit prostřednictvím příkazu *docker images*, který je více popsán v následujícím odstavci.

Procházení obrazů

Zobrazení a procházení mezi všemi dostupnými obrazy lze provést příkazem *docker images*, jak demonstruje následující ukázka č. 12, která předpokládá, že existují následující obrazy v hostitelském počítači.

```
$ sudo docker images
REPOSITORY   TAG       IMAGEID           CREATED         VIRTUALSIZE
mysql        latest    890tde23643p      13 days ago     374 MB
mysql        8.0.4     19c775f2c310      13 days ago     291 MB
```

Ukázka 12 - Výpis všech obrazů

Tento výpis obsahuje mnoho užitečných informací o obrazu. Jedna z prvních informací obsahuje název repozitáře *mysql*, odkud obraz pochází. Informace taktéž říká, že z tohoto repozitáře byl obraz stažen. Je důležité si uvědomit, že jsou obrazy ukládány v repozitářích a repozitáře uchováváme v takzvaných registrech, jak bylo zmíněno v kapitole 2.2.4. Nelze si nepovšimnout, že jsou tu dva na první pohled stejné obrazy. Liší se od sebe kromě velikostí především názvem tagu, který identifikuje každý obraz, který se nachází v repozitáři. Většinou se tímto způsobem verzují obrazy, kde tag latest v tomto případě říká, že jde o nejnovější verzi obrazu MySQL[15] databáze. V ostatních případech tak lze ukládat a stahovat obrazy různých verzí. To umožňuje uložení více než jednoho obrazu do repozitáře. Tagování se především hodí při spouštění kontejnerů, kdy je na první pohled zřejmé, jaký je zdrojový obraz, ze kterého kontejner vychází. Jak se z takových obrazů různých verzí vytváří kontejner, popisuje kapitola o spuštění kontejneru.

Existují dva typy repozitářů: uživatelské repozitáře, které obsahují obrazy vytvořené klasickými uživateli Dockeru a repositáře vyšší úrovně, tzv. top-level, které jsou především kontrolovány lidmi z Docker týmu. Uživatelské repositáře se poznají na první pohled podle řetězce pod hlavičkou repository, který se skládá ze dvou částí: uživatelského jména a názvu repositáře ve tvaru *uživatelské_jméno/název_repozitáře*, jak znázorňuje ukázka č. 13.

```
$ sudo docker images
REPOSITORY                    TAG       ...
sipekmichal/custom-java-app   latest    ...
```

Ukázka 13 - Uživatelský repozitář

Případem repozitáře vyšší úrovně je předešlý příklad s mysql.

[15] MySQL je open-source relační databáze vycházející z deklarativního programovacího jazyka SQL.

Tyto repozitáře jsou spravovány společností Docker, Inc. a vybranými dodavateli, které poskytují základní obrazy. Výhoda těchto repozitářů spočívá v uzavření závazku mezi společností a dodavateli, kdy je záruka, že obsažené obrazy jsou kvalitně sestavené, bezpečné a aktuální. Je třeba mít na paměti, že obrazy z uživatelských repozitářů jsou vytvářeny samotnými uživateli komunity, nejsou nijak kontrolovány a měly by být používány na vlastní riziko. Docker navíc s vydáním verze 1.8 přidal podporu pro volitelné zabezpečení prostřednictvím podepisování obrazů.

Vyhledávání obrazů

Docker umožňuje prostřednictvím příkazu *docker search* (příkladem je ukázka č. 14) vyhledávat obrazy, které se nacházejí ve veřejném registru Docker Hub. To platí pro veřejně přístupné obrazy, nikoliv pro soukromé repozitáře.

```
$ sudo docker search mysql
NAME       DESCRIPTION              STARS   OFFICIAL   AUTOMATED
Mysql      MySQL is a widely…       5757    [OK]
Mariadb    MariaDB is a comm…       1824    [OK]
...
```

Ukázka 14 - Vyhledávání obrazu MySQL v registru Docker Hub

Příkaz vrací názvy všech dostupných obrazů souvisejících s řetězcem mysql spolu s dalšími informacemi:

- název repozitáře,
- popis obrazu,
- počet hvězdiček – označuje popularitu obrazu,
- oficiálnost, která podává informaci o tom, zda obraz pochází z repozitáře vyšší úrovně, resp. je spravován společností Docker, Inc. a vybranými dodavateli,
- automatizace informující o tom, zda byl obraz sestaven pomocí automatizovaného sestavovacího procesu z Docker Hub.

Tento způsob vyhledávání se hodí především pro ty, kteří rádi pracují s příkazovou řádkou a vědí, co hledají. V takovém případě je vyhledávání rychlejší. Tyto obrazy lze také vyhledat přímo na stránkách www.hub.docker.com.

Podepisování obrazů

Verze Docker Engine 1.8 přináší novou funkci podepisování obrazů zvanou Docker Content Trust, která umožňuje ověření vydavatele obrazu. Než vydavatel uloží obraz do vzdáleného registru, Docker Engine podepíše obraz soukromým klíčem vydavatele a uživatel, který později obraz stáhne, použije veřejný klíč vydavatele, aby ověřil jeho identitu. Tím je zabezpečeno, že obraz nebyl nikým narušen, podvržen a je aktuální. Podepisování obrazů se hodí zavádět zejména do produkčního prostředí, kde se používají citlivé údaje. Pro vývoj aplikací za podpory Dockeru není žádoucí tuto funkci používat [13].

2.4.4 Návrhové vzory sestavení obrazu

Jedna z nejnáročnějších věcí u sestavování obrazů je zachování jeho velikosti. Tím se zabývají různé návrhové vzory. Jak je již známo, každá instrukce v souboru Dockerfile přidává vrstvu v obrazu. Zde je třeba si uvědomit, že před tím, než dojde k přesunu na další vrstvu, je žádoucí se zbavit všech nepotřebných artefaktů. Pro vytvoření skutečně efektivního souboru Dockerfile je tradičně potřeba použít pár triků a jinou logiku, aby vrstvy byly co nejmenší a aby každá vrstva měla artefakty, které doopravdy potřebuje a nic jiného. Dnes již běžnou záležitostí je mít jeden Dockerfile pro vývoj (který obsahuje vše potřebné k sestavení aplikace) a optimalizovaný Dockerfile pro produkci obsahující pouze aplikaci s tím, co je potřeba k jejímu spuštění. Tento návrhový vzor je označován jako Docker Builder (vzor, který např. v jednom kontejneru sestaví aplikaci a výsledek odešle do druhého kontejneru) [10][14].

Návrhový vzor Docker Builder

Návrhový vzor sestavení obrazu Docker Builder popisuje nastavení, které mnoho lidí používá k vytvoření obrazu a redukci výsledné velikosti. Zahrnuje použití dvou obrazů - jednoho pro provedení sestavení označovaného jako „build“ a druhého pro odeslání výsledků prvního sestavení. Tento proces znázorňují další ukázky č. 15 a č. 16. Soubor *Dockerfile.build* obsahuje kód, který provede vše potřebné k sestavení aplikace. Z vytvořeného obrazu se aplikace přenese do druhého obrazu vytvořeného ze souboru Dockerfile.

```
FROM openjdk:8-jdk
RUN apt-get update && apt-get upgrade -y && \
    apt-get install git-all -y
RUN git clone
https://github.com/sipekmichal/custom.java.app.helloworld.git
RUN cp -R custom.java.app.helloworld/* .
RUN javac HelloWorld/Main.java && \
jar cfme HelloWorld.jar Manifest.txt HelloWorld.Main
HelloWorld/Main.class
```

Ukázka 15 - Soubor Dockerfile.build

Samotný obraz vytvořený ze souboru Dockerfile vychází pouze z miniaturní linuxové distribuce Alpine, kopíruje aplikaci, kterou následně spouští.

```
FROM alpine:latest
RUN apk --update add openjdk8-jre
COPY HelloWorld.jar .
CMD ["/usr/bin/java", "-jar", "HelloWorld.jar"]
```

Ukázka 16 - Soubor Dockerfile

Celý proces řídí skript *./build.sh*, jehož obsah je zobrazen v ukázce č. 17. Skript vyvolává sestavení prvního „build" obrazu, následně z něho připraví kontejner, odkud je aplikace kopírována do aktuálního adresáře na hostitelský souborový systém pro potřebu druhého obrazu, který je nakonec sestaven.

```
echo Building sipekmichal/custom-java-app:build
docker image build -t sipekmichal/custom-java-app:build . -f
Dockerfile.build
docker container create --name extract sipekmichal/custom-java-
app:build
docker container cp extract:HelloWorld.jar .
docker container rm -f extract
echo Building sipekmichal/custom-java-app:latest
docker image build --no-cache -t sipekmichal/custom-java-app:latest .
```

Ukázka 17 - Skript build.sh

Tím je dosaženo redukce obrazu. Pro představu v tomto příkladu dochází ke snížení velikosti obrazu *sipekmichal/custom-java-app:build* z 1.02 GB na obraz *sipekmichal/custom-java-app:latest*, který má výslednou velikost 83.2 MB. I přesto udržování dvou souborů není úplně ideální, proto vznikl další vylepšený návrhový vzor.

Návrhový vzor Multi-stage build

Funkce vícestupňového sestavení tzv. Multi-stage build, kterou nedávno Docker představil ve verzi 17.05, využívá možnosti použít více příkazů *FROM* v jednom Dockerfile souboru. Lze tak selektivně kopírovat artefakty z jedné fáze do druhé a nakonec vznikne pouze jeden výsledný obraz definovaný za posledním příkazem *FROM* [10]. Ukázka č. 18 znázorňuje, jak je možné přechozí příklad jednoduše převést do tohoto vzoru.

```
FROM openjdk:8-jdk AS build
RUN apt-get update && apt-get upgrade -y && \
    apt-get install git-all -y
RUN git clone
https://github.com/sipekmichal/custom.java.app.helloworld.git
RUN cp -R custom.java.app.helloworld/* .
RUN javac HelloWorld/Main.java && \
jar cfme HelloWorld.jar Manifest.txt HelloWorld.Main
HelloWorld/Main.class

FROM alpine:latest
RUN apk --no-cache --update add openjdk8-jre
COPY --from=build HelloWorld.jar .
CMD ["/usr/bin/java", "-jar", "HelloWorld.jar"]
```

Ukázka 18 - Dockerfile s využitím Multi-stage

2.4.5 Souhrn instrukcí

V předchozích kapitolách a ukázkách bylo možné vidět některé instrukce používané v souborech Dockerfile. Patří jich sem samozřejmě mnohem více, ale není vůbec od věci mít zde souhrn těch nejvíce používaných, aby bylo možné se k nim v případě potřeby vracet. Úplný seznam s detaily a vysvětlením je možné nalézt v oficiální dokumentaci Dockeru na stránce https://docs.docker.com/engine/reference/builder/.

FROM

Instrukce *FROM* inicializuje novou fázi sestavení a nastavuje základní obraz pro vykonání navazujících instrukcí. Každý platný Dockerfile soubor začíná touto instrukcí, která je povinná. Následující příklad v ukázce č. 19 znázorňuje její použití [15].

```
FROM ubuntu:14.04
```

Ukázka 19 - Použití instrukce FROM

LABEL

Instrukce *LABEL* přidává informace o obrazu ve formě metadat, které mají formát klíč-hodnota. V případě zadání mezer v rámci hodnoty se používají uvozovky a zpětné lomítko. Dockerfile může obsahovat jeden i více instrukcí *LABEL* [4].

Je velmi doporučováno nepoužívat tuto instrukci pro komentáře, protože každá tato instukce vytváří novou vrstvu obrazu. Často se používá pro uveřejnění autora souboru Dockerfile. Ukázka č. 12 nastavuje popisek *maintainer*, ve kterém se typicky uvádí kontakt na autora obrazu.

```
LABEL maintainer="mail@sipekmichal.cz"
```

Ukázka 20 - Použití instrukce LABEL

RUN

Během procesu vytváření obrazu se instrukce *RUN* používá ke spuštění příkazu. To je realizováno nad nejvýše položenou vrstvou obrazu, výsledkem je nová vrstva (prostřednictvím provedení commit nového obrazu), která vstupuje do dalšího kroku v souboru Dockerfile[4].

```
FROM ubuntu:14.04
RUN apt-get update && apt-get upgrade -y
```

Ukázka 21 - Použití instrukce RUN

EXPOSE

Instrukce *EXPOSE* informuje Docker o tom, že kontejner za běhu naslouchá na určených síťových portech. Lze určit, zda port naslouchá na protokolech TCP (výchozí) nebo UDP. Instrukce tyto porty ještě nevystavuje veřejně [15]. Je určená především jako typ dokumentace mezi osobou, která sestavuje obraz a osobou, která provozuje kontejner.

```
EXPOSE 8080
```

Ukázka 22 - Použití instrukce EXPOSE

ADD

Instrukce *ADD* je podobná instrukci *COPY*. Rozdíl mezi těmito příkazy je popsán níže. Kopíruje soubor nebo adresář z hostitelského počítače do cílového adresáře obrazu. Je také možné použít URL adresu odkazující na některý soubor, prostřednictvím které ho Docker stáhne. Lze ji zadat následujícím předpisem [15].

- ADD [--chown=<uživatel>:<skupina>] <zdroj>... <cíl>

Všechny nové soubory a adresáře jsou vytvořeny s identifikátory UID[16] a GID[17] 0, pokud volitelný příznak *--chown* neurčuje zadané uživatelské jméno, název skupiny nebo UID/GID.

Je ještě důležité zmínit, že funkce *--chown* je podporována pouze v souborech Dockerfile použitých pro vytváření kontejnerů v prostředí Linux a nebude tak fungovat v prostředí systému Windows. Následující ukázka č. 23 demonstruje možné použití této instrukce, kde dochází ke zkopírování souboru *schema.sql* z adresy veřejného repozitáře GitHub a dále všechny soubory začínající *src* do cílového adresáře */usr/src/*, přičemž u druhého příkazu navíc nastavuje vlastníka a skupinu těchto souborů.

```
ADD https://github.com/sipekmichal/bikeo/blob/master/schema.sql
/usr/src
ADD --chown=myuser:mygroup files* /somedir/
```

Ukázka 23 - Použití instrukce ADD

Pokud souborový systém kontejneru neobsahuje soubory */etc/passwd* nebo */etc/group* a přesto jsou v příznaku *--chown* zadány údaje jako jméno uživatele nebo skupiny, sestavení obrazu při zpracování této instrukce selže. Používání číselných identifikátorů však nevyžaduje žádné vyhledávání a nebude záviset na obsahu souborového systému kořenového kontejneru. To však neplatí pro číselné identifikátory [4].

COPY

Instrukce *COPY* se velmi podobá té předchozí. Používá se ke kopírování souborů a adresářů ze zadaného zdroje do cíle (v souborovém systému kontejneru). Lze ji zadat následujícím předpisem.

- COPY [--chown=<uživatel>:<skupina>] <zdroj>... <cíl>

Ukázka č. 24 kopíruje soubor *HelloWorld.jar* z aktuálního umístění (kontextu sestavení) do absolutní cesty */opt/custom-java-app/*.

```
COPY HelloWorld.jar /opt/custom-java-app/
```

Ukázka 24 - Použití instrukce COPY

[16] UID – User Id je číslo uživatele, které Linux využívá k identifikaci namísto uživatelského jména.
[17] GID – Group Id je číslo skupiny, podobné jako UID.

Rozdíl mezi ADD a COPY

Ačkoliv jsou instrukce *ADD* a *COPY* funkčně podobné, je mezi nimi určitý rozdíl. Obecně se preferuje *COPY*, protože je pro uživatele čitelnější. *COPY* podporuje pouze základní kopírování lokálních souborů do kontejneru, zatímco *ADD* obsahuje některé další funkce, jako například extrahování a stažení souborů z předané URL, které nemusí být na první pohled zřejmé. V důsledku toho je vhodné instrukci *ADD* použít pouze pro případy extrahování souboru.

Je velmi doporučováno a s postupem času se osvědčilo používat v Dockerfile kopírování souborů individuálně a nikoliv najednou v rámci jedné instrukce. Ačkoliv se obecně usiluje o snížení počtu vrstev výsledného obrazu, zde v ukázce č. 25 platí výjimka. Výsledkem je menší počet vrstev, které se musí znovu sestavit, pokud dojde k změně některého ze souborů v kontextu sestavení, než kdyby poslední instrukce byla před instrukcí *RUN* [10].

```
COPY requirements.txt /tmp/
RUN pip install --requirement /tmp/requirements.txt
COPY . /tmp/
```

Ukázka 25 - Vhodné použití instrukce COPY ve více řádcích

ENV

Instrukce *ENV* slouží k nastavení proměnné prostředí ve tvaru klíč-hodnota. Ta je potom dostupná v daném prostředí a v případě potřeby ji lze v následujících krocích sestavení obrazu nahradit. Má dvě formy:

- ENV <klíč> <hodnota>
- ENV <klíč>=<hodnota> ...

První forma nastavuje jednu proměnnou na hodnotu definovanou místo *<value>*, přičemž druhá umožňuje nastavit více proměnných v rámci této jedné instrukce [15]. Ukázka č. 16 popisuje konkrétní zápis instrukce *ENV*.

```
ENV REFRESHED_AT 2018-01-08
```

Ukázka 26 - Použití instrukce ENV

USER

Instrukce *USER* slouží pro nastavení uživatelského jména (případně UID) a volitelně skupiny (případně GID), pod kterou se vykonávají všechny následné instrukce *CMD*, *RUN* a *ENTRYPOINT*.

Lze ji v souboru Dockerfile přetěžovat.

```
USER root
```

Ukázka 27 - Použití instrukce USER

CMD

Instrukce *CMD* určuje příkaz, který se vykoná při spuštění kontejneru. Je obdobný příkazu *ENTRYPOINT* a rozdíl mezi těmito dvěma je vysvětlen později níže. Instrukce *CMD* nevykonává nic v době sestavení obrazu.
Lze ji definovat ve třech následujících formách [4]:

- CMD ["spustitelný-soubor","parametr1","parametr2"] (často používaná forma),
- CMD ["parametr1","parametr2"] (jako výchozí parametry pro ENTRYPOINT),
- CMD příkaz parametr1 parametr2 (ve formě shellu).

V celém Dockerfile souboru lze použít maximálně jednu instrukci typu *CMD*. Pokud je jich i přesto více, použije se ta poslední. V ukázce č. 28 je použita instukce pro spuštění příkazu *java* s dalšími předanými parametry, které zajistí spuštění aplikace *HelloWorld.jar*.

```
FROM openjdk:8-jdk
CMD ["java","-jar","/opt/custom-java-app/HelloWorld.jar"]
```

Ukázka 28 - Použití instrukce CMD

ENTRYPOINT

Instrukce *ENTRYPOINT* definuje výchozí příkaz, který je spuštěn při běhu kontejneru. Příkaz lze zapsat dvěma různými formami:

- ENTRYPOINT ["spustitelný-soubor", "parametr1", "parametr2"] (často používaná forma)
- ENTRYPOINT příkaz parametr1 parametr2 (ve formě shellu)

Následující ukázka č. 29 odkrývá a dále vysvětluje, jak instrukce funguje.

```
FROM ubuntu:14.04
ENTRYPOINT ["/bin/ping"]
CMD ["localhost", "-c", "2"]

$ sudo docker container run -it test_image
$ sudo docker container run -it test_image google.com
```

Ukázka 29 - Použití instrukce ENTRYPOINT

Předpokladem je sestavený obraz z tohoto Dockerfile, který se nazývá *test_image* (sudo docker image build -t test_image). Jsou-li spuštěny dva kontejnery příkazy níže, tak v prvním případě je výsledkem provedení *ping localhost* a druhém *ping google.com*. Pro ještě lepší pochopení rozdílu mezi instrukcí *CMD* a *ENTRYPOINT* je dobré si přečíst následující odstavec.

Rozdíl mezi CMD a ENTRYPOINT

Pro pochopení instrukcí *CMD* a *ENTRYPOINT* je vhodné si ukázat jejich rozdílné použití. Obě instrukce určují, jaký příkaz se spustí při spuštění kontejneru. Existuje několik pravidel, které popisují jejich spolupráci:

- *ENTRYPOINT* - by měl být specifikován vždy při spuštění kontejneru v produkčním prostředí.
- *CMD* - specifikuje výchozí argumenty, které se použijí pro instrukci ENTRYPOINT.

Pro potřeby vývoje aplikací a testování se doporučuje použít místo *ENTRYPOINT* instrukci *CMD*, protože ji lze v momentu spouštění kontejneru přetížit, a vykonat tak v kontejneru jiný příkaz [15].

WORKDIR

Instrukce *WORKDIR* nastavuje pracovní adresář pro všechny *RUN*, *CMD*, *COPY*, *ENTRYPOINT* a *ADD* instrukce, které po něm následují v souboru Dockerfile. Pokud pracovní adresář neexistuje, je vytvořen i přesto, že není například použit v další instrukci [17]. V následující ukázce č. 30 si lze povšimnout, jak je možné pracovat s relativní cestou.

```
WORKDIR /user
WORKDIR home
RUN pwd
```

Ukázka 30 - Použití instrukce WORKDIR

Výstupem je cesta */user/home*.

VOLUME

Instrukce *VOLUME* povoluje propojení adresáře mezi hostitelským počítačem a kontejnerem. Hodnota může být pole typu JSON nebo prostý řetězec [4]. S vytvořením nového kontejneru prostř. příkazu *RUN* se inicializuje svazek se všemi daty, které existují na určeném místě v základním obrazu. Drobná ukázka č. 31 demonstuje vytvoření adresáře */data* prostřednictvím příkazu *mkdir*, přesměrování výstupu řetězce *14.04* do souboru *ubuntu-version.txt* a nakonec tento soubor zpřístupní hostitelskému počítači z adresáře */data*.

```
FROM ubuntu:14.04
RUN mkdir /data
RUN echo "14.04" > /data/ubuntu-version.txt
VOLUME /data
```

Ukázka 31 - Použití instrukce VOLUME

Pro více informací ohledně tohoto mechanismu existuje dokumentace dostupná na https://docs.docker.com/storage/volumes/.

2.4.6 Registry

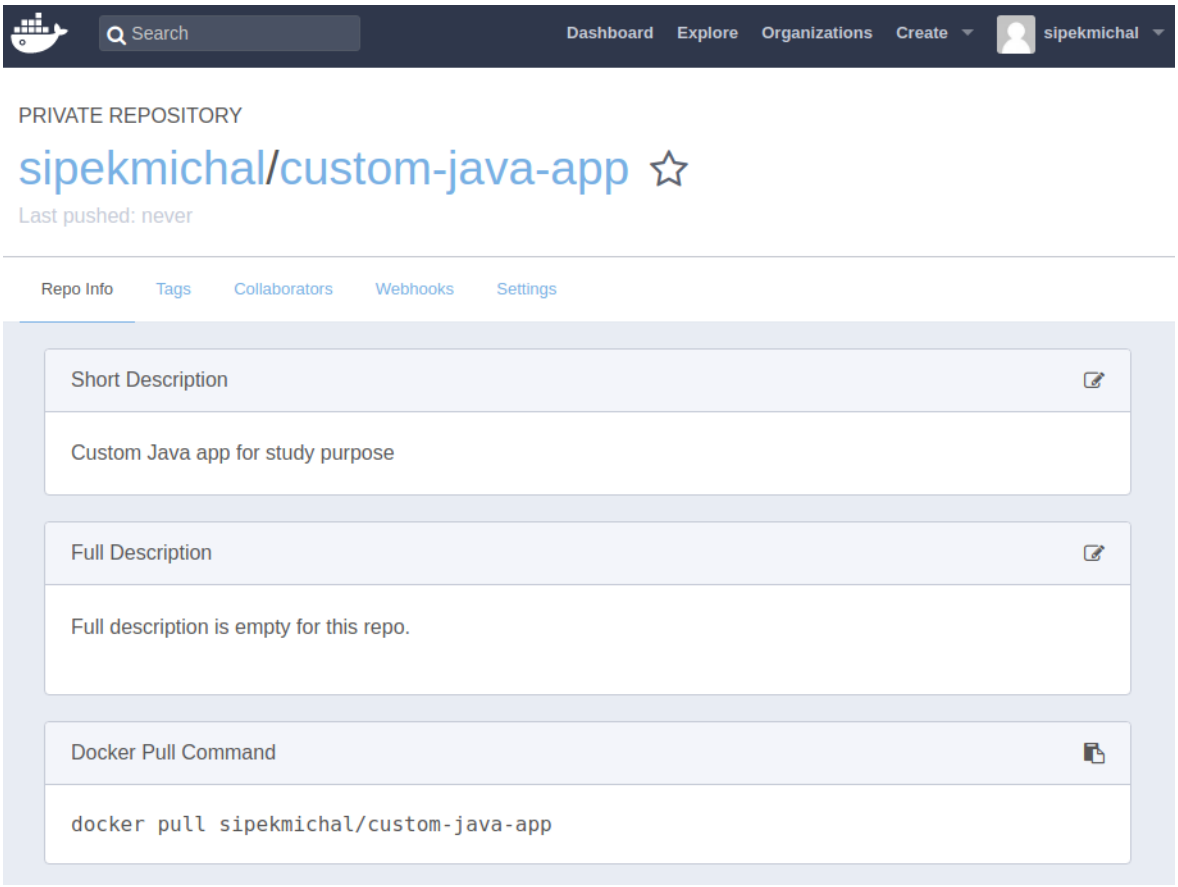
Ve chvíli sestavení obrazu nastává situace, kdy je žádoucí uložit sestavený obraz do registru k budoucímu uchování, či sdílení mezi ostatní členy týmu. Jak již bylo zmíněno, existují dva typy registrů: veřejné a soukromé. O rozdílech mezi nimi bylo taktéž napsáno v úvodní části základních konceptů, proto je zde přímo zobrazen postup instalace a použití.

Veřejný registr Docker Hub

Docker Hub je cloudová služba, která poskytuje centralizovaný prostředek pro vyhledávání, správu uživatelů a týmů, distribuci obrazů a automatizační pipeline [10]. Pro vývojáře aplikací je vhodné volit tento registr především z důvodu jeho rychlého a snadného použití. Vývojář se s velkou pravděpodobností nebude chtít zabývat instalací registru na svém lokálním prostředí, či vzdáleném serveru, nastavováním parametrů pro zabezpečené spojení apod. Docker Hub nabízí mimo veřejně dostupných repozitářů také možnost využít tzv. soukromých repozitářů. Jedná se však o placenou funkci, která umožňuje uchovávat citlivé obrazy. Pro účely vývoje je k dispozici jeden soukromý repozitář zdarma, kde lze uchovávat obrazy dostupné pouze jeho vlastníkovi a lidem, se kterými vlastník repozitář sdílí.
Docker Hub poskytuje následující hlavní funkce:

- Repozitáře pro obrazy – vyhledávání, stahování obrazů a ukládání obrazů z veřejných a soukromých repozitářů.
- Automatické sestavení – automatické sestavení nových obrazů na základě změny ve zdrojovém kódu.
- Vytvoření pracovních skupin pro správu přístupů k repozitářům.
- Integrace řešení systému Git prostřednictvím GitHub a Bitbucket [10].

Prvním krokem pro využívání registru je založení účtu na stránkách https://hub.docker.com a následné přihlášení. Uživatel si na začátku registrace mimo jiné volí tzv. Docker ID, dle něhož se uživatel identifikuje. Dalším krokem může být vytvoření repozitáře pro uložení obrazu sestaveného v předešlé kapitole věnované sestavení obrazu pomocí Dockerfile, jak znázorňuje obrázek č. 2.



Obr. 2 - Přehled soukromého repozitáře
Zdroj: vlastní zpracování

Pro uložení obrazu se uživatel ze stroje, kde se sestavený obraz nachází, přihlašuje do registru Docker Hub prostřednictvím příkazu *docker login*. Následně zadává Docker ID a heslo zvolené při registraci, jak je možné vidět v ukázce č. 32.

```
$ sudo docker login
Login with your Docker ID to push and pull images from Docker Hub. If
you don't have a Docker ID, head over to https://hub.docker.com to
create one.
Username: sipekmichal
Password: *****
Login Succeeded
```

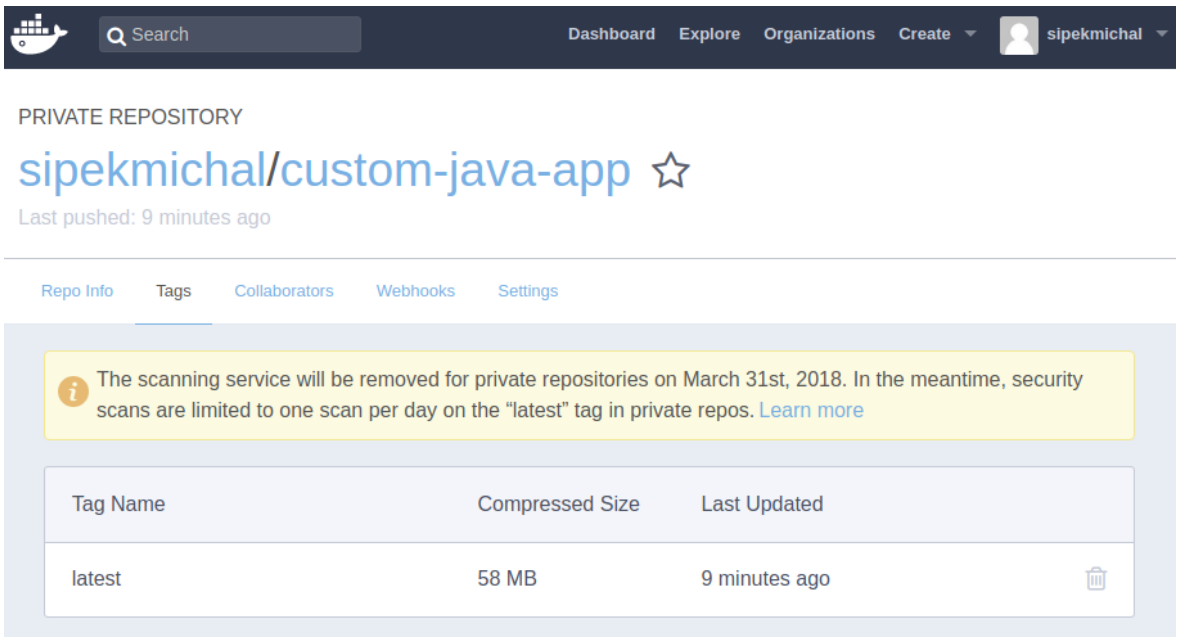
Ukázka 32 - Docker login

Po úspěšném přihlášení lze použít příkaz *docker image push* pro zaslání obrazu do již předtím vytvořeného repozitáře (proces zobrazen v ukázce č. 33).

```
$ sudo docker image push sipekmichal/custom-java-app:latest
The push refers to repository [docker.io/sipekmichal/custom-java-app]
9b968062fab0: Pushed
808d5c5befea: Pushed
cd7100a72410: Mounted from library/alpine
latest: digest:
sha256:53015ab9bb526d57545f6e0ea0e80bc2a06c7dc8d1355b38b504d1f72757c093
size: 947
```

Ukázka 33 - Uložení obrazu do Docker Hub

Ve většině případů je nutné před provedením příkazu *docker image push* použít příkaz *docker image tag*, aby obraz odkazoval na příslušný repozitář v Docker Hub registru. Vzhledem k tomu, že má již obraz referenci na daný repozitář, není nutné referenci obrazu prostřednictvím příkazu *docker tag* měnit. Nakonec se výsledný obraz uloží do registru a je dostupný k budoucímu použití. Obrázek č. 3 ukazuje výsledek úspěšného uložení na stránkách Docker Hub.



Obr. 3 - Uložený obraz v Docker Hub repozitáři
Zdroj: vlastní zpracování

Všechny postupy týkající se práce a použití Docker Hub jsou dopodrobna vysvětleny na stránkách https://docs.docker.com/docker-hub.

Instalace vlastního soukromého registru

Pro bezpečné uložení obrazů v rámci organizace na vlastních serverech pro potřeby Operations k nasazení do produkčního prostředí obvykle dochází k instalaci vlastního soukromého registru. Důvody často bývají hlavně bezpečnostní, kde sestavené obrazy typicky obsahují konfigurační soubory s citlivými údaji do produkce apod.

V následujícím kroku je popsán postup instalace takového registru na server s operačním systémem Ubuntu.

Instalace prerekvizit

Prerekvizitou instalace registru je nastavení zabezpečení. Pro tento účel je použit Docker Compose, který je vysvětlen v dalších kapitolách a předpokládá se, že je již nainstalován. K zabezpečení přístupu do registru je v tomto příkladu použit Nginx[18] spolu s utilitou *htpasswd* generující šifrovaná hesla. Protože je tato utilita součástí balíku apache2-utils, je nutné ji nainstalovat prostřednictvím následujícího příkazu.

```
$ sudo apt-get -y install apache2-utils
```

Ukázka 34 - Instalace apache2-utils

Instalace a konfigurace registru

Pro instalaci základního registru je nejprve zapotřebí provést konfiguraci, tj. definovat umístění, ve kterém bude registr ukládat data a dále vytvořit soubor YAML pro Docker Compose pro spuštění instance registru, jak dokládá ukázka č. 35.

```
$ mkdir ~/docker-registry && cd $_
$ mkdir data
$ nano docker-compose.yml
```

Ukázka 35 - Příprava kontextu sestavení

Do souboru *docker-compose.yml* je nutné přidat obsah (ukázka č. 36) pro paralelní spuštění kontejneru s registrem a kontejneru s Nginx pro zaručení zabezpečeného spojení. Tyto dva kontejnery jsou mezi sebou prolinkované, aby měl Nginx přístup k registru.

```
nginx:
  image: "nginx:1.9"
  ports:
    - 443:443
  links:
    - registry:registry
  volumes:
    - ./nginx/:/etc/nginx/conf.d:ro
registry:
  image: registry:2
  ports:
    - 127.0.0.1:5000:5000
  environment:
    REGISTRY_STORAGE_FILESYSTEM_ROOTDIRECTORY: /data
  volumes:
    - ./data:/data
```

Ukázka 36 - docker-compose.yml

[18] Nginx je open-source webový server s load managementem a reverzní proxy.

Nastavení Nginx kontejneru

Předchozím souborem dojde k vytvoření kontejneru založeného na oficiálním obrazu Nginx. Zajímavým povšimnutím pro někoho může být link, který slouží k vytvoření odkazu na jiný kontejner. Tím je docíleno přístupu do kontejneru s registrem bez ohledu na to, jaká je skutečná IP adresa tohoto kontejneru (Ve skutečnosti Docker odvádí práci za vývojáře/uživatele tím, že vkládá záznam s IP adresou do souboru */etc/hosts* – v tomto případě v kontejneru nginx). Sekce volumes je podobná jako u kontejneru s registrem, dává možnost uložit konfigurační soubory, které jsou pro Nginx použity, na hostitelském počítači. Řetězec *:ro* na konci řádku říká, že kontejner má přístup k souborům na hostitelském počítači pouze pro čtení [18]. Pro uložení konfigurace je tedy v první fázi vytvořen adresář a v něm soubor *registry.conf* pro konfiguraci Nginx s obsahem z ukázky č. 38.

```
$ mkdir ~/docker-registry/nginx
```

Ukázka 37 - Vytvoření adresáře pro konfiguraci Nginx

```
upstream docker-registry {
  server registry:5000;
}
server {
  listen 443;

  ssl on;
  ssl_certificate /etc/nginx/conf.d/domain.crt;
  ssl_certificate_key /etc/nginx/conf.d/domain.key;

  client_max_body_size 0;
  chunked_transfer_encoding on;

  location /v2/ {
    if ($http_user_agent ~ "^(docker\/1\.(3|4|5(?!\.[0-9]-dev))|Go
).*$" ) {
      return 404;
    }
    auth_basic "registry.localhost";
    auth_basic_user_file /etc/nginx/conf.d/registry.password;
    add_header 'Docker-Distribution-Api-Version' 'registry/2.0' always;

    proxy_pass                          http://docker-registry;
    proxy_set_header  Host              $http_host;
    proxy_set_header  X-Real-IP         $remote_addr;
    proxy_set_header  X-Forwarded-For   $proxy_add_x_forwarded_for;
    proxy_set_header  X-Forwarded-Proto $scheme;
    proxy_read_timeout                  900;
  }
}
```

Ukázka 38 - Konfigurační soubor registry.conf

Nastavení autentizace

Pro kontrolu, kdo má do registru přístup, je nutné nastavit autentizaci pomocí http. Za tímto účelem je vytvořen ověřovací soubor [18].

```
$ cd ~/docker-registry/nginx
$ htpasswd -c registry.password USERNAME
```

Ukázka 39 - Vytvoření uživatele

Nastavení SSL

V tomto okamžiku je připravený registr, který poběží za Nginx se základní http autentizací. Toto nastavení není stále dostatečně bezpečné, protože připojení jsou nešifrovaná. Na řadu tedy přichází nastavení SSL. V předešlém souboru registry.conf je připravená konfigurace pro použití SSL. Pokud již disponujeme SSL certifikátem, stačí pouze zkopírovat soubory certifikátu a klíče do cest uvedených v souboru *registry.conf* (ssl_certificate and ssl_certificate_key). V jiném případě je nutné vytvořit tzv. self-signed certifikát. Existují spousty návodů dostupných na internetu, jak takový certifikát vytvořit, proto tento postup není součástí této kapitoly. Předpokládá se, že jsou certifikáty vytvořeny.

Pokud certifikáty nejsou ověřeny žádnou známou certifikační autoritou, je zapotřebí informovat všechny klienty, které budou komunikovat s registrem včetně serveru, kde je registr nainstalovaný, že jde o oprávněný certifikát a následně restartovat Docker démona, aby změny zaregistroval, jak je patrné z ukázky č. 40 [18].

```
$ sudo mkdir /usr/local/share/ca-certificates/docker-cert
$ sudo cp docker.crt /usr/local/share/ca-certificates/docker-cert
$ sudo update-ca-certificates
$ sudo service docker restart
```

Ukázka 40 - Důvěryhodnost certifikátu

Tyto kroky je nutné opakovat pro každý server, který bude komunikovat s registrem. Nakonec lze registr spustit pomocí Docker Compose a otestovat příkazem z ukázky č. 41.

```
$ sudo docker-compose up
$ curl https://<YOURUSERNAME>:<YOURPASSWORD>@YOUR-DOMAIN/v2/
```

Ukázka 41 - Spuštění registru

Příkaz curl[19] by měl v úspěšném případě vrátit prázdný json objekt {}.

[19] Curl je nástroj pro přenos dat z nebo na server pomocí jednoho z podporovaných protokolů (DICT, FILE, FTP, FTPS, GOPHER, HTTP, HTTPS, IMAP, IMAPS, LDAP atd.).

2.5 Kontejnery

Ke správě kontejnerů se používá příkaz *docker container*, kterému lze předat další příkazy, které definují příslušnou operaci (celkový výčet operací lze najít v dokumentaci). V této pasáži nejsou rozebrány všechny tyto příkazy, ale jen ty nejdůležitější. První část této kapitoly je věnována práci s kontejnery a druhá složitějším situacím, jako je kooperace několika kontejnerů. V neposlední řadě je podstatné pochopení mentality a praktik využití kontejnerů, které přijde na řadu ke konci kapitoly.

2.5.1 Spuštění kontejneru

Spuštění kontejneru lze nejlépe demonstrovat na některém obrazu z předchozích kapitol. Pro tyto účely je možné použít obraz *sipekmichal/custom-java-app:latest*. Spuštění kontejneru je pak realizováno prostřednictvím příkazu docker container run. Existuje několik možností, jak takový kontejner spustit a argumentů, které mu lze předat.

```
$ sudo docker container run -i -t sipekmichal/custom-java-app:latest sh
Unable to find image 'sipekmichal/custom-java-app:latest' locally
latest: Pulling from sipekmichal/custom-java-app
Digest:
sha256:53015ab9bb526d57545f6e0ea0e80bc2a06c7dc8d1355b38b504d1f72757c093
Status: Downloaded newer image for sipekmichal/custom-java-app:latest
/ #
```

Ukázka 42 - Jednoduché spuštění kontejneru

Spuštění příkazu z předchozí ukázky č. 42 vyvolá několik akcí, které se dějí na pozadí. Docker nejprve kontroluje, zda se obraz *sipekmichal/custom-java-app:latest* nachází v hostitelském počítači, a pokud takový obraz nenajde, hledá ho ve veřejném registru Docker Hub. V případě úspěchu si obraz odtamtud stáhne na hostitelský počítač a použije ho pro vytvoření a spuštění kontejneru [4]. Po zadání předchozího příkazu dojde k interakci s kontejnerem a je k dispozici jeho příkazová řádka. Kontejner je možné opustit příkazem exit. Důležité je podotknout, že v tuto chvíli dojde k terminování kontejneru (kontejner je ve stavu EXITED), protože byl přerušen hlavní proces. Že je kontejner opravdu v tomto stavu, lze ověřit příkazem (ukázka č. 43), který zajistí zobrazení všech kontejnerů.

```
$ sudo docker container ps -a
CONTAINER ID        IMAGE                    COMMAND
CREATED             STATUS                   PORTS               NAMES
675f054d2096        sipekmichal/custom-java-app:latest   "sh"
7 minutes ago       Exited (0) 2 seconds ago
```

Ukázka 43 - Výpis běžících kontejnerů

Většinou se ale pro opouštění kontejneru používá klávesová zkratka *Ctrl-p, Ctrl-q*, která z interaktivního módu vytvoří démona.

Detašovaný kontejner

Pro to, aby kontejner běžel na pozadí, tj. jako démon, slouží volba *-d*, viz. ukázka č. 44 která se zadává při spouštění kontejneru.

```
$ sudo docker container run –i -t –d sipekmichal/custom-java-app:latest
25e6ccf98d62d2291c51707b61d6546c93b6f0a39c86aa4952f41cfddf8441a9
$
$ sudo docker container ps -a
CONTAINER ID        IMAGE                    COMMAND
CREATED             STATUS                   PORTS               NAMES
25e6ccf98d62        sipekmichal/custom-java-app:latest   "/usr/bin/java
-jar …"   2 seconds ago       Up 1 second
silly_feynman
```

Ukázka 44 - Spuštění detašovaného kontejneru

Při spuštění kontejneru s příznakem *-d* dochází k vytvoření kontejneru běžícího na pozadí. Uvnitř kontejneru je spuštěna aplikace, která na výstup vypíše hlášku *Hello World!*. To lze ověřit např. příkazem *docker container logs název_kontejneru*. Pro demonstraci celého procesu je tato konkrétní aplikace ukončena v momentě, kdy uživatel stiskne klávesu enter. Pro to, aby bylo možné připojit se na standardní vstup kontejneru, stisknout klávesu enter a přerušit tak aplikaci, slouží příkaz *docker container attach název_kontejneru* [4]. Za povšimnutí stojí stav kontejneru, ve kterém se nyní nachází. S přerušením hlavního procesu kontejneru, jímž byla tato aplikace, dochází také k přerušení kontejneru.

Pojmenování kontejneru

Docker řeší pojmenování kontejneru tím způsobem, že automaticky generuje náhodný název pro každý vytvořený kontejner. Je vidět, že právě vytvořený kontejner se nazývá *silly_feynman*. Pojmenovat kontejner konkrétním názvem je možné provést pomocí příznaku *–name*, jak je znázorněno v ukázce č. 45.

```
$ sudo docker container run –i –t –d --name=java_app
sipekmichal/custom-java-app:latest
```

Ukázka 45 - Identifikace kontejneru

Název může obsahovat znaky a až z, A až Z, číslice 0 až 9, podtržítko a pomlčku (nebo vyjádřeno jako regulární výraz: [a-zA-Z0-9 _.-]). Ve většině příkazů se pak lze odkazovat právě prostřednictvím tohoto jména namísto ID kontejneru.
Názvy kontejnerů pomáhají identifikovat a utvářet logická spojení mezi kontejnery a aplikacemi. Je mnohem jednodušší zapamatovat si určitý název kontejneru (např. web nebo db) než ID kontejneru nebo dokonce náhodný vygenerovaný název. Z toho vyplývá používat vždy správné pojmenování pro správu kontejnerů [4].

Práce s porty

Co se týká práce s porty, ve výchozím nastavení při vytváření kontejneru nejsou publikovány žádné z jeho portů do vnějšího světa. Pro zpřístupnění portu službám mimo Docker nebo kontejnery Dockeru, které nejsou připojeny k síti kontejneru, se používá příznak *-p* nebo *--publish*. Tím je vytvořeno pravidlo brány firewall, které vystavuje port kontejneru do hostitelského počítače [4].

```
-p 8080:80
-p 192.168.1.100:8080:80
```

Ukázka 46 - Mapování portů a IP adres

Předchozí ukázka č. 46 demonstruje způsob publikování portů, kde první případ mapuje port 80 (uvnitř kontejneru) na port 8080, který je dostupný na hostitelském počítači a druhý navíc specifikuje IP adresu 192.168.1.100.

2.5.2 Další operace s kontejnery

Následující odstavce nabízejí přehled dalších zajímavých operací s kontejnery, které by mohly být ve fázi vývoje a testování užitečné. Nepokrývá kompletní seznam, který je možné nalézt na:
https://docs.docker.com/engine/reference/commandline/container.

Procházení a investigace kontejnerů

Kromě informací o kontejneru, které je možné získat pomocí příkazu *docker container ps*, lze prostřednictvím *docker container inspect* provádět detailní investigaci kontejnerů.

```
$ sudo docker container inspect java_app
[
    {
        "Id":
"babd90440e0e935449947f6df1ff019075758fc8ad734c8533ede4b199999bed",
        "Created": "2018-06-19T18:54:36.401961473Z",
        "Path": "/usr/bin/java",
        "Args": [
            "-jar",
            "HelloWorld.jar"
        ],
        "State": {
            "Status": "running",
"Running": true,
            "Pid": 11050,
            "ExitCode": 0,
            "Error": "",
            "StartedAt": "2018-06-19T18:54:36.822695302Z",
...
```

Ukázka 47 - Investigace kontejneru prostř. docker inspect

Příkaz *docker container inspect* se dotazuje na předaný kontejner a vrací informace o jeho konfiguraci včetně jmen, příkazů, konfigurace sítě a řady dalších užitečných dat, jak dokládá předešlá ukázka č. 47.

Další užitečnou operací je zjištění statistiky daného kontejneru. Kromě příkazu *docker container top* exsituje také příkaz *docker container stats* pro zobrazení statistik pro jeden nebo více běžících kontejnerů [4].

```
$ sudo docker container stats java_app
CONTAINER ID   NAME      CPU %     MEM USAGE / LIMIT     MEM %
NET I/O          BLOCK I/O    PIDS
babd90440e0e   java_app  0.10%     13.52MiB / 15.35GiB   0.09%
2.72MB / 0B      0B / 0B      15
```

Ukázka 48 - Zobrazení statistik kontejneru

Ukázka č. 48 zobrazuje seznam, který poskytuje informace o kontejneru a jeho CPU, paměti, sítě a dalších metrik užitečných pro sledování. Zobrazení toho, co se děje v kontejneru a získání logů (hodí se hlavně pro detašované kontejnery) zajišťuje příkaz *docker container logs*. Jedná se o příkaz, po kterém by měl každý uživatel sáhnout, když se kontejner chová neočekávaně.

```
$ sudo docker container logs java_app
Hello world!
```

Ukázka 49 - Získání logu z kontejneru

Propojení kontejnerů

Mimo komunikace pomocí vystavení služby (portu) na stejném hostiteli existuje pro propojení kontejnerů jednodušší cesta. Tou je vytvoření linky. Nejen, že Docker zajistí příslušné lokálně-síťařské úkony (překlady adres apod.), ale cílovému kontejneru sdělí informace o službě toho prvního, tedy IP adresu, port, typ protokolu atd. [19].
Linky umožňují, aby se kontejnery navzájem objevovaly a bezpečně přenášely informace z jednoho kontejneru do druhého. Pro vytvoření linky se Docker spoléhá na jména kontejnerů. Pro vytvoření se používá příznak *--link* ve formě:

- --link <název nebo id kontejneru>:alias

V případě, že by se předchozí kontejner *java_app* teoreticky připojoval k databázi, vytvořila by se linka následujícím způsobem (ukázka č. 50).

```
$ sudo docker container run –i –t -d --name database postgres
$ sudo docker container run –i –t –d --name=java_app --link database:db
sipekmichal/custom-java-app:latest
```

Ukázka 50 - Propojení kontejnerů pomocí linky

V takovém případě se služby uvnitř kotejneru *java_app* mohou připojovat k databázi prostřednictvím aliasu *db* uvedeném za dvojtečkou. Výhodou linkování kontejnerů oproti propojování skrze IP adresy a porty, je velmi volné svázání, možnost kontejnery různě prohazovat, měnit a restartovat – není potřebné se starat o čísla portů běžících služeb ani IP adresy [19].

Automatický restart kontejneru

Pokud z nějakého důvodu přestal kontejner pracovat, je možné ho nakonfigurovat tak, aby se sám v takové situaci automaticky restartoval. Tato konfigurace se díky příznaku *--restart* provádí již při spouštění kontejneru. V principu to funguje tím způsobem, že se kontroluje návratový kód kontejneru a na základě toho se rozhoduje, zda kontejner restartovat, či nikoliv [10]. Ve výchozím stavu je tato funkce vypnutá, a proto ji je potřeba explicitně povolit, jak demonstruje následující ukázka č. 51.

```
$ sudo docker container run –i –t –d --name=java_app --restart=always
sipekmichal/custom-java-app:latest
```

Ukázka 51 - Automatický restart kontejneru

V tomto příkladu je aplikován příznak *--restart=always*. V takovém případě Docker restartuje kontejner vždy bez ohledu na jeho návratový kód. Alternativně je možné definovat politiku, jak kontejner restartovat.

Například *--restart=on-failure:3* zajistí restart kontejneru pouze v případě, když kontejner skončí s nenulovým návratovým kódem a navíc se o to pokusí maximálně třikrát. Další možnosti této politiky lze v případě potřeby snadno dohledat na stránkách dokumentace Dockeru.

3 Nástroje podporující nasazení a management

Existuje mnoho nástrojů, které poskytují kompletní rámec pro orchestraci kontejnerů, API[20], kontinuální integraci a nasazení (CI/CD), loadbalancing atd. Následující odstavce stručně seznamují s některými důležitými nástroji pro správu kontejnerů, které stojí určitě za to prošetřit, pokud je uživatel připraven s nimi pracovat.

3.1 Docker Compose

Pokud se aplikace skládá z více než jednoho kontejneru (například z webového serveru a databáze), je v takovém případě sestavení a spuštění kontejnerů těžkopádné a časově náročné. Nástroj Docker Compose tento problém řeší použitím souboru YAML k definování jednotlivých kontejnerů (označovaných služby), které se mají spustit. Lze nakonfigurovat libovolné množství kontejnerů, jak mají být sestavené obrazy, kde mají být uložena data atd. Jakmile je takový soubor sestaven, je možné pomocí jednoho příkazu provést zároveň sestavení, spuštění a konfiguraci všech definovaných kontejnerů. V této práci se pro názorné ukázky používá verze *docker compose 1.21.2*.

Použití Compose je v podstatě třístupňový proces:

1. Vytvoření souborů Dockerfile pro sestavení jednotl. obrazů
2. Definování služeb, které tvoří aplikace v souboru *docker-compose.yml*
3. Spuštění příkazu *docker-compose up*

Příklad takového *docker-compose.yml* souboru je zobrazen v ukázce č. 52.

```
version: "3.6"
services:
  database:
    image: mysql
    ports:
      - "3306:3306"
    environment:
      - MYSQL_ROOT_PASSWORD=password
      - MYSQL_DATABASE=demodb
      - MYSQL_USER=admin
      - MYSQL_PASSWORD=password
  web:
    image: nginx
```

Ukázka 52 - docker-compose.yml

[20] API je zkratkou využívanou v informatice pro označení rozhraní k programování aplikací.

Rozborem jednotlivých klíčových slov, které se i v tomto příkladu vyskytují, se zaobírá kapitola 4.3 zabývající se kontejnerizací existující aplikace, kde implementace Docker Compose je jedna z částí celého procesu.

3.1.1 Funkce Docker Compose

I když hlavní funkcí aplikace Docker Compose je vytvoření kontejnerů a vazeb mezi nimi, nástroj je schopný mnohem více. Disponuje příkazy pro správu celého životního cyklu aplikace pro spouštění. Více o těchto funkcích se lze dozvědět na stránkách oficiální dokumentace https://docs.docker.com/compose/. Následující ukázka č. 53 demonstruje výčet jednotlivých funkcí, kterými Compose disponuje.

```
build    Sestavení obrazů
help     Zobrazení nápovědy
kill     Přerušení kontejnerů
logs     Zobrazení výstupu kontejnerů
port     Zobrazení veřejného portu
ps       Výpis seznamu kontejnerů
pull     Stažení obrazů
rm       Odstranění zastavených kontejnerů
start    Spuštění služeb
stop     Zastavení služeb
restart  Restart služeb
up       Vytvoření a spuštění kontejnerů
```

Ukázka 53 - Přehled funkcí Docker Compose

Některé funkce jsou níže více rozebrány především díky tomu, že jejich použití je mezi vývojáři velmi časté.

Sestavení obrazů

Pro sestavení všech obrazů definovaných v rámci služeb jednoho docker-compose.yml souboru se používá příkaz *docker-compose build*. Za příkaz lze zadat několik voleb, mezi nimiž stojí za zmínku volba *docker-compose build --pull*, která se nejprve vždy pokusí o stažení nejnovější verze obrazu. Je ještě nutné upozornit na fakt, že pokud dojde ke změně souboru Dockerfile nebo nějakého obsahu v kontextu sestavení, je nutné zavolat tento příkaz znovu.

Škálování kontejnerů

V některých případech je vhodné řešit škálování kontejnerů, obzvláště když na vývojovém procesu, či testování spolupracuje více lidí. V tom případě je potřeba připravit několik identických kontejnerů s danou aplikací.

Docker Compose umožňuje definovat počet služeb, které se vytvoří při zavolání příkazu *docker-compose up*. Počet instancí lze definovat a předat v rámci příkazu *docker-compose up --scale NÁZEV_SLUŽBY=POČET*.

Pokud by ale došlo ke spuštění tohoto příkazu, celý proces nasazení by zřejmě havaroval důsledkem chyby při pokusu několikrát alokovat ten samý port. V souboru Compose pak musí být definován rozsah portů, ve kterém jsou jednotlivé služby spouštěny [20][21].

3.1.2 Obvyklé případy použití

Docker Compose lze použít mnoha různými způsoby. Některé běžné případy použití jsou uvedeny níže.

Vývojové prostředí

Při vývoji softwaru je rozhodující schopnost spustit aplikaci v izolovaném prostředí a komunikovat s ní. Nástroj příkazového řádku Compose lze použít k vytvoření prostředí a interakci s ním. Soubor *docker-compose.yml* navíc poskytuje způsob, jak dokumentovat a konfigurovat všechny závislé služby (databáze, fronty, mezipaměti, API webových služeb atd.). Pomocí příkazového řádku je možné vytvořit a spustit jeden nebo více kontejnerů pouze jediným příkazem (již zmíněný docker-compose up). Společně tyto funkce poskytují vývojářům pohodlný způsob, jak začít s projektem.

Prostředí automatizovaných testů

Důležitou součástí procesu průběžného nasazení nebo průběžné integrace je automatizovaná testovací sada. Automatizované testování typu end-to-end vyžaduje prostředí, ve kterém je možné testy provádět. Docker Compose poskytuje pohodlný způsob, jak vytvořit a zničit takové prostředí pro testovací sadu pomocí několika následujících příkazů z ukázky č. 54 [22].

```
$ sudo docker-compose up -d
$ ./run_tests
$ sudo docker-compose down
```

Ukázka 54 - Spuštění sady testů s využitím Docker Compose

Produkční prostředí

Pokud je použit Compose v rámci vývojového a testovacího prostředí, lze ho taktéž využít v produkci. Nejjednodušší způsob nasazení je spuštění na jednom serveru, podobně jako by bylo spuštěno pomocí Compose vývojové prostředí. Pro použití v produkčním prostředí je nutné provést několik konfiguračních změn, které mohou zahrnovat [23]:

- Odstranění datových svazků, které jsou dostupné z hostitelského počítače tak, aby kód aplikace nemohl být změněn zvenčí.
- Zpřístupnění odlišných portů oproti jinému prostředí.
- Nastavení odlišných proměnných například pro povolení odesílání mailů apod.
- Určení zásad pro restart kontejneru, aby se zabránilo prodlevám opětovného spuštění při manuálním zásahu.

Z tohoto důvodu je velmi vhodné zvážit vytvoření dalšího souboru Compose, např. *production.yml*, který specifikuje konfiguraci odpovídající produkčnímu prostředí. Je důležité, aby tento konfigurační soubor obsahoval pouze změny, které chtějí být zahrnuty oproti původnímu souboru Compose. V praxi to funguje tím způsobem, že soubor production.yml je aplikován přes *docker-compose.yml*, dojde tedy k automatickému sloučení těchto dvou souborů. Spuštění takového prostředí pak musí být realizováno prostřednictvím následujícího příkazu z ukázky č. 55.

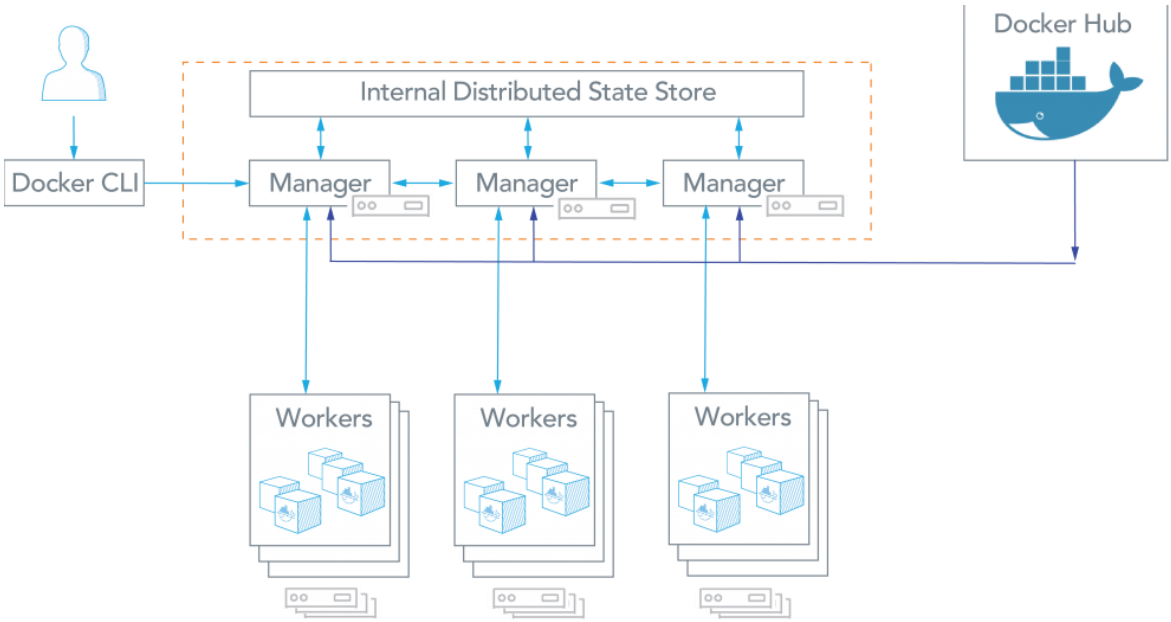
```
$ sudo docker-compose -f docker-compose.yml -f production.yml up -d
```

Ukázka 55 - Použití Docker Compose v produkčním prostředí

3.2 Docker Swarm

Swarm je cluster[21] jednotlivých modulů Docker Engine, na který lze nasazovat služby (services). Stručně řečeno, provádět orchestraci kontejnerů. Od verze v1.12.0 je součástí Docker Engine jako takzvaný swarm režim a lze ho tedy volat přímo z příkazové řádky Dockeru. Jednotlivé moduly Docker Engine jsou nasazené na několika ulech (nodes) [10].

Swarm se skládá z manažerských (Manager Nodes) a pracovních uzlů (Worker Nodes), jak lze vidět na obrázku č. 4 níže. Manažeři provádějí orchestraci, správu clusteru a udržují swarm aktivní. Pracovní uzly tzv. Worker Nodes přijímají a provádějí úlohy. Jednotlivé úkoly jsou definovány službami, které lze deklarativním způsobem specifikovat [24]. Uživatel tak může určit požadovaný stav různých služeb pomocí souborů YAML. Každá taková služba se skládá z obrazu a sérií příkazů, které se mají uvnitř kontejneru(ů) vykonat.



Obr. 4 - Architetura Docker Swarm
Zdroj: převzato z [25]

Následují některé běžně používané pojmy.

Node

Node neboli uzel, je instancí Docker Engine participujícím ve swarmu. Lze spustit jeden nebo více uzlů na jednom fyzickém počítači nebo cloudovém serveru.

[21] Cluster je zkráceně seskupení počítačů, které spolu úzce spolupracují a navenek se tváří jako celek.

Manager Nodes

Pro nasazení aplikace do swarmu se posílá definice služby uzlu manažera tzv. Manager Node, který dále odesílá „pracovní jednotky“ nazývané úkoly pracovním uzlům.

Worker Nodes

Pracovní uzly tzv. Worker Nodes přijímají a provádějí úlohy, které jsou jim předány prostřednictvím manažera. Pracovní uzel oznamuje manažerovi aktuální stav přidělených úkolů, aby manažer mohl udržovat požadovaný stav.

Service

Služba je definicí úkolů, které je třeba provést na uzlech manažera nebo pracovníků. Jedná se o centrální strukturu swarmu, se kterou uživatel přijde k interakci. V modelu *Replicated services* distribuuje manažer konkrétní počet replik na základě měřítka, které uživatel nastavil. V modelu *Global services* spouští swarm jednu úlohu v rámci jedné služby na každém uzlu v clusteru [10].

Task

Úloha je atomickou jednotkou swarmu, která obstarává Docker kontejner a provádí v něm definované příkazy.¨

3.2.1 Vytvoření Docker Swarm clusteru

První částí je inicializace clusteru, která vytvoří Manager Node prostřednictvím příkazu *docker swarm init*.

```
$ sudo docker swarm init
Swarm initialized: current node (tip8cecle9ofu36isckh422zc) is now a
manager.

To add a worker to this swarm, run the following command:

docker swarm join --token SWMTKN-1-
608w1e4xgts5b6rm1az6xtufjw0hy07i14lphd7alzu5wg0yzy-
c5ll9mr8z7o94j63clkp9x66f 192.168.1.104:2377
```

Ukázka 56 - Vytvoření Docker Swarm

Po zadání předchozího příkazu je zobrazen výstup viz. ukázka č. 56. Token uvedený výše se použije pro následné přidání pracovních uzlů (Worker Node) do clusteru. Stav vytvořeného manažera lze zobrazit příkazem *docker node ls*, jehož výstup včetně dalších jeho informací je znázorněn v ukázce č. 57 [29].

Přidání pracovního uzlu do swarm clusteru

V okamžiku vytvoření uzlu manažera je na čase přidat pracovní uzel do clusteru. Přidání je realizováno příkazem *docker swarm join* z ukázky č. 56. V tuto chvíli je uzel viditelný na uzlu manažera. Pro zobrazení všech dostupných uzlů slouží příkaz *docker node ls*. Celý proces přidání pracovního uzlu do clusteru a zobrazení uzlů je možné shlédnout v ukázce č. 57.

```
$ sudo docker swarm join --token SWMTKN-1-
608w1e4xgts5b6rm1az6xtufjw0hy07i14lphd7alzu5wg0yzy-
c5ll9mr8z7o94j63clkp9x66f 192.168.1.104:2377
This node joined a swarm as a worker.

$ sudo docker node ls
ID        HOSTNAME  STATUS  AVAILABILITY  MANAGER STATUS
tip8ce *  Manager   Ready   Active        Leader
gnrfyhi   Worker    Ready   Active
```

Ukázka 57 - Přidání pracovního uzlu v Docker Swarm

Spuštění služby

Na běžícím clusteru lze nyní spustit službu. Na uzlu manažera je možné spustit příkaz *docker service create* z ukázky č. 58, který vytvoří již známý kontejner *bikeodb* s MySQL databází a namapuje port 3306 na hostitelský systém. Kontejner s databází vychází z obrazu *sipekmichal/bikeomysqldb:5.7*. Dále je v ukázce znázorněno zobrazení všech běžících služeb příkazem *docker service ls*.

```
$ sudo docker service create --name bikeodb \
   -e MYSQL_ROOT_PASSWORD=passw0rd \
   -e MYSQL_DATABASE=bikeo \
   -e MYSQL_USER=admin \
   -e MYSQL_PASSWORD=admin \
   --name bikeodb \
   -p 3306:3306 \
  sipekmichal/bikeomysqldb:5.7

$ sudo docker service ls
ID       NAME     MODE        REPLICAS  IMAGE    PORTS
alw8a..  bikeodb  replicated  1/1       sipekmichal/bikeomysqldb:5.7
*:3306->3306/tcp
```

Ukázka 58 - Vytvoření služby v Docker Swarm

Vytvořenou službu lze nyní například škálovat prostřednictvím příkazu *docker service scale bikeodb=2*. Příkaz vytvoří druhou repliku služby, výsledek si lze prohlédnout příkazem *docker service ps bikeodb* (ukázka č. 59).

```
$ sudo docker service scale bikeodb=2
bikeodb scaled to 2
overall progress: 2 out of 2 tasks
1/2: running   [==================================================>]
2/2: running   [==================================================>]
verify: Service converged

$ sudo docker service ps bikeodb
ID      NAME       IMAGE                           NODE     DESIRED STATE
CURRENT STATE           ERROR  PORTS
p1uu..  bikeodb.1  sipekmichal/bikeomysqldb:5.7  Manager  Running
Running 32 seconds ago
uv5v..  bikeodb.2  sipekmichal/bikeomysqldb:5.7  Worker   Running
Running 10 seconds ago
```

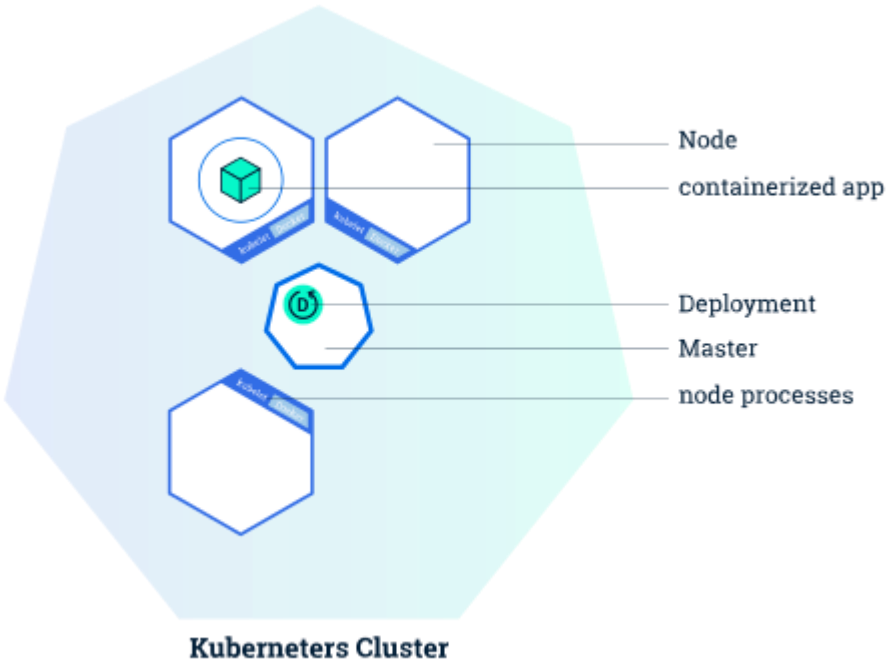
Ukázka 59 - Škálování a zobrazení služby v Docker Swarm

Pokud by došlo k pádu jednoho z kontejnerů, uzel manažera se postará o to, aby spustil službu novou a udržoval tak požadovaný počet replik.

3.3 Kubernetes

Kubernetes, ve zkratce označován jako K8s, je open-source systém pro automatizaci, zavádění, škálování a správu kontejnerových aplikací, který byl původně navržen v roce 2014 společností Google a který byl poskytnut Cloud Native Computing Foundation. Jeho cílem je provádět orchestraci kontejnerů automatizovaně, tj. spravovat, zavádět, škálovat a monitorovat aplikační kontejnery napříč clusterem hostů (virtuálních nebo fyzických). Podporuje celou řadu kontejnerových nástrojů včetně nástroje Docker [26][27]. Jeho použití se zejména hodí v prostředí, kde je vyžadováno spravovat desítky až stovky kontejnerů. Bez použití takového nástroje je správa velice obtížná. Pro vývojáře, kteří pracují na migraci aplikací do Dockeru, tento nástroj není vhodný, protože pravděpodobně nevyužijí jeho potenciál. Implementace Kubernetes je velmi náročnou operací a její rozbor by vydal na samostatnou publikaci. V této části dochází alespoň k vysvětlení několika základním pojmům, aby čtenář získal povědomí o principu a existenci tohoto systému.

Pro pochopení základní architektury, jak Kubernetes funguje, jsou v následující části definovány jednotlivé komponenty. Následující obrázek č. 5 zobrazuje pohled na Kubernetes cluster a jeho komponenty, které jsou rozebrány náže.



Obr. 5 - Kubernetes Cluster
Zdroj: převzato z [28]

3.3.1 Kubernetes Master Node

Master Node (hlavní uzel) je první komponentou, která je zodpovědná za správu clusteru. Slouží ke koordinaci, plánování, udržování požadovaných stavů aplikací, škálování a zavádění nových aktualizací. Níže je rozpis jednotlivých klíčových součástí masteru [26].

API Server

Jedná se o jedinou součást ovládacího panelu Kubernetes s uživatelsky přístupným API a jedinou hlavní komponentou, s níž uživatel interaguje. API server vystavuje rozhraní Kubernetes API, pomocí kterého uzly komunikují s Master Node.

Cluster Data Store

Kubernetes používá "etcd" pro trvalé ukládání všech objektů API.

Controller Manager

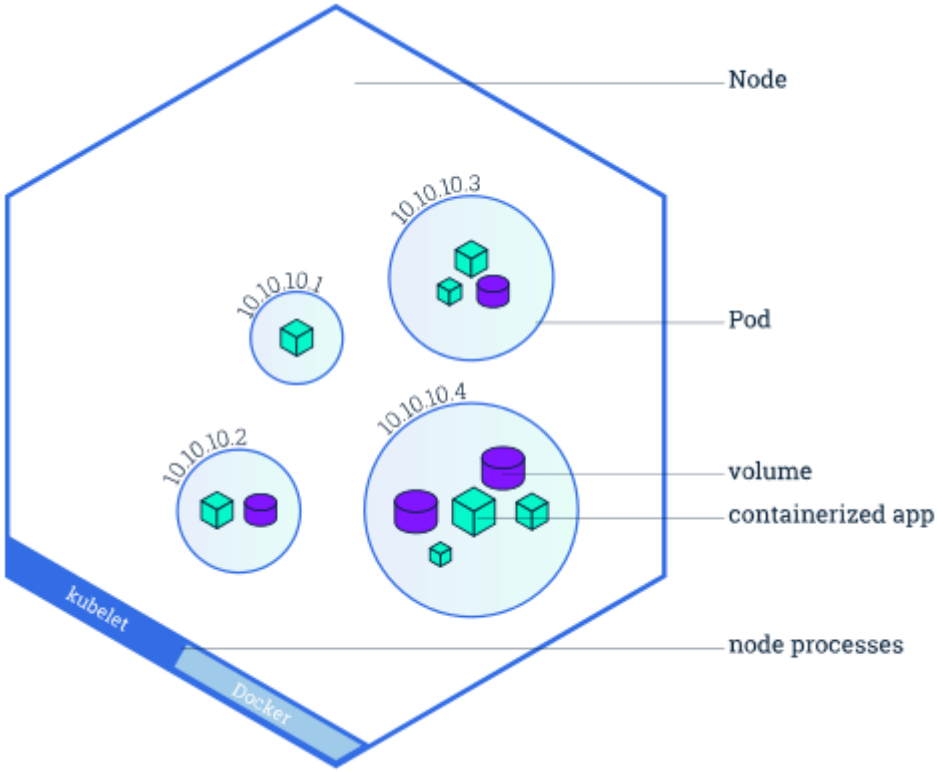
Controller Manager známý jako "kube-controller manager", spouští všechny řadiče, které zpracovávají rutinní úlohy v clusteru. Patří sem například řadič uzlu (Node Controller), řadič replikace (Replication Controller), řadič koncových bodů (Endpoint Controller) a Service Account a Token Controller. Každý z těchto řadičů pracuje samostatně, aby udržel požadovaný stav [27].

Scheduler

Scheduler sleduje nově vytvořené Pody a přiřazuje je uzlům (nodes).

3.3.2 Kubernetes Worker Node

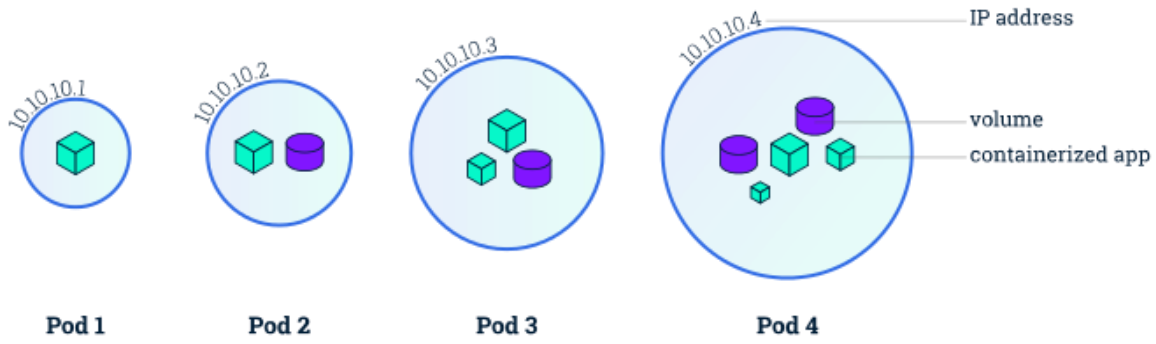
Druhou důležitou součástí Kubernetes jsou uzly (nodes). Zatímco Master Node řídí cluster a komunikuje s Kubelety, tyto pracovní uzly spouštějí kontejnery a poskytují pro Kubernetes runtime prostředí [27]. Každý uzel obsahuje Kubelet, což je agent pro správu uzlu a komunikaci s master Kubernetes. Uzel by měl mít také nástroje pro manipulaci s kontejnerovými operacemi, jako je například Docker nebo rkt.



Obr. 6 - Kubernetes Node
Zdroj: převzato z [28]

3.3.3 Kubernetes Pod

Základní jednotkou deploymentu, se kterým Kubernetes operuje, je pod. Typicky obsahuje jeden kontejner, ale může jich být více, pokud patří k sobě (např. aplikace s proxy serverem). Kromě kontejneru(ů) může obsahovat i datové svazky (volumes). Pod je bezstavový a není určen pro to, aby žil dlouho, může být kdykoliv přesunut, zničen nebo nahrazen [27]. Obrázek č. 7 níže znázorňuje, jak takový Pod může vypadat.



Obr. 7 - Kubernetes Pod
Zdroj: převzato z [28]

3.3.4 Stavební bloky

Cluster

Kubernetes cluster je soubor fyzických nebo virtuálních strojů a dalších infrastrukturních prostředků, které Kubernetes používá ke spuštění aplikací.

Node a Pod

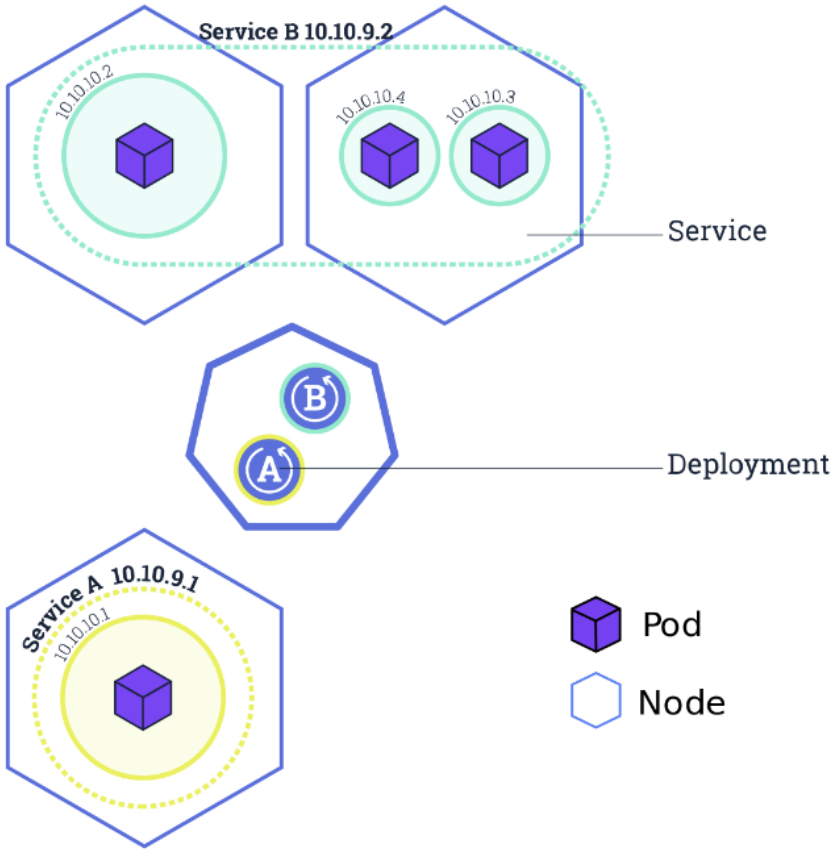
Node je fyzický nebo virtuální stroj, na kterém běží Kubernetes, na který lze naplánovat Pody. Pod je popisem souboru kontejnerů, které je třeba společně spouštět, viz. výše.

Label

Label je dvojice klíč/hodnota, která je připojena k prostředku, jako je pod, pro přenos identifikátoru definovaného uživatelem. Labely lze použít k uspořádání a výběru podmnožin zdrojů [28].

Service a Deployment

K tomu, aby na sebe repliky podu na sebe navzájem síťově „viděli“, slouží koncept Service, ve kterém lze deklarovat službu, která získá nějaké interní DNS jméno, interní IP adresu, která balancuje provoz nebo externí adresu přicházející z IaaS vrstvy (tedy externě přístupná služba dosažitelná z vně clusteru).



Obr. 8 - Kubernetes Service
Zdroj: převzato z [28]

Předchozí obrázek č. 8 slouží k demonstraci následujícího výkladu. Pokud je žádoucí volat z microservice A microservice B, je potřeba microservice B vystavit jako Service. Tři repliky Podu služby B představují Deployment. Deployment je popis toho, co má výše zmíněný pod obsahovat, kolik podů se má spustit, jak má škálovat a jaký Docker obraz se má použít. Z deploymentu se pak vytváří ReplicaSet, což je skupina reálně běžících kontejnerů [27].

ReplicaSet

Sadou kopií/replik nějakého Podu je ReplicaSet. Pokud je žádoucí dockerizovanou aplikaci spustit například ve třech instancích, dá se do Podu a spustí se tři repliky.

Ingress

Ingress je nástroj pro zpřístupnění Service clusteru zvenku. Jde o operaci, kde výsledkem je IP adresa. Tu je potřeba zanést do DNS záznamů, řešit nějaký URL routing (pokud je vyžadováno), řešit šifrování apod. Ingress koncept je definice vstupních pravidel včetně rozhazování na jednotlivé Service podle cesty v URL, TLS akcelerace a řízení certifikátů, pokročilejší pravidla či automatické vytváření subdomény. Typickou implementací Ingress je automatizovaný NGINX [27].

3.4 Grafické nástroje

Kromě pluginů, které lze nainstalovat do vývojového prostředí, existuje mnoho řešení (grafických nástrojů) usnadňujících práci s Dockerem. Na internetu se objevují spousty článků, které uživateli pomůžou vybrat správný nástroj, nicméně málo takových, které stručně shrnou a pomůžou rozhodnout, který z nich je opravdu vhodný pro konkrétní potřeby uživatele [25]. První dva zde popsané nástroje jsou určené především pro vývojáře. Operations tým by měl dát přednost spíše komplexnímu nástroji, který je vhodný pro správu, nasazení i škálování. Takovým vhodným nástrojem může být například Rancher, který je jako třetí zmíněný. Právě ten podporuje potřeby této skupiny lidí [31].

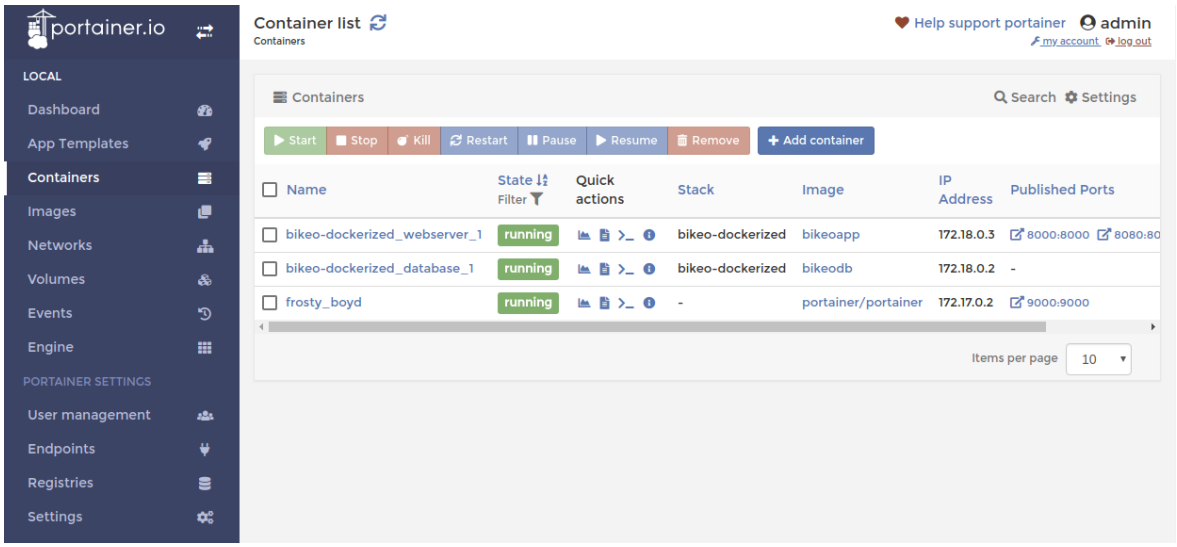
3.4.1 Portainer

Portainer je bezplatná open-source webová aplikace, která běží v samostatném kontejneru. Lze ho jednoduše nainstalovat a spustit pomocí příkazu uvedeného v ukázce č. 60.

```
$ sudo docker volume create portainer_data
$ sudo docker container run -d -p 9000:9000 \
-v /var/run/docker.sock:/var/run/docker.sock \
-v portainer_data:/data \
portainer/portainer
```

Ukázka 60 - Instalace aplikace Portainer

Pro zvládnutí interakcí využívá rozhraní Docer API a pokrývá většinu hlavních funkcí pro vytváření, úpravu, správu, sledování a mazání kontejnerů, stejně jako schopnost přidávat, odstraňovat a prohlížet obrazy, sítě a svazky [32]. Bohužel je ale nemůže upravovat. Oproti nástroji Kitematic má výhodu v množství funkcionalit a konfigurovatelnosti. Obrázek č. 9 znázorňuje grafické prostředí pro správu kontejnerů.



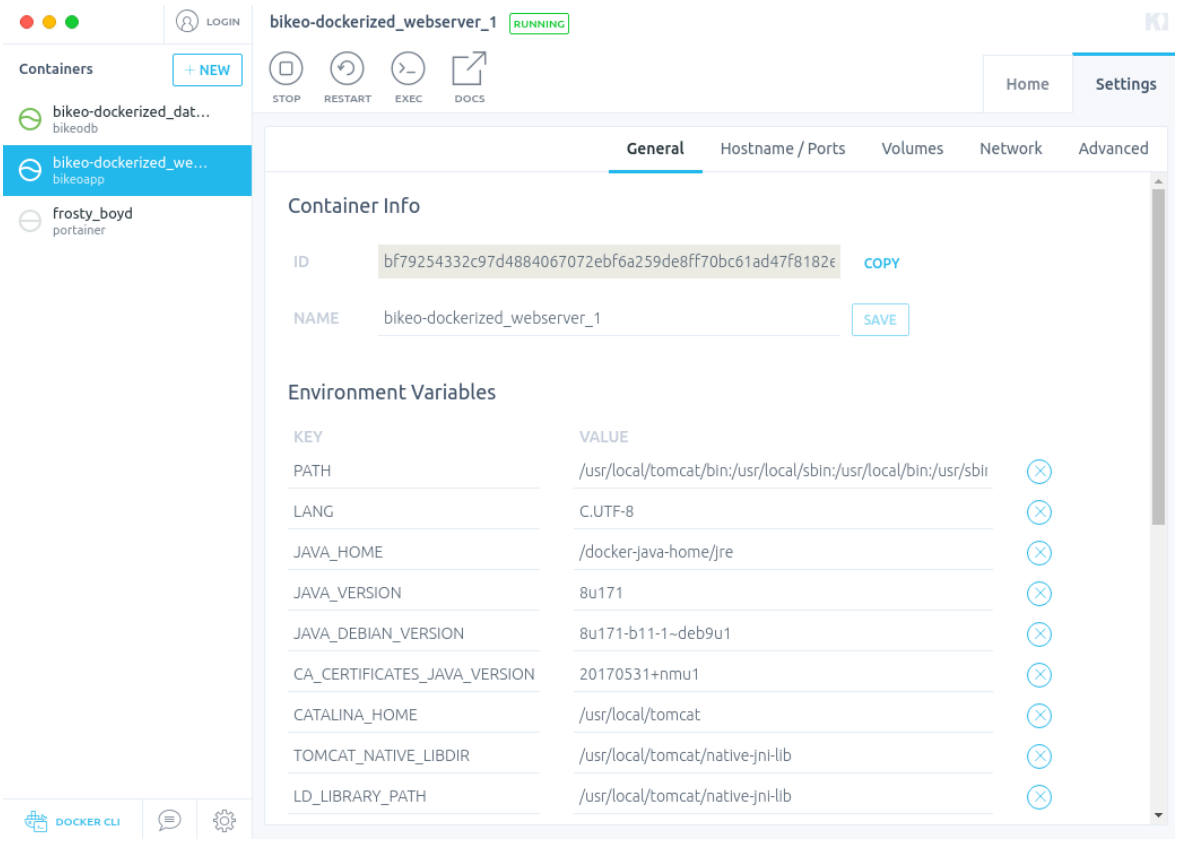
Obr. 9 - Přehled kontejnerů v aplikaci Portainer

Portainer se postupem času stal skvělým nástrojem, který obsahuje spoustu funkcí a obrazovek:

- podrobný přehled,
- seznam, detail a statistika kontejneru,
- logy a operování s kontejnerem,
- seznam a detail obrazu,
- seznam sítí a datových svazků,
- správa uživatelů a řízení přístupů apod.

3.4.2 Kitematic

Kitematic je open-source projekt, který počínaje vydáním verze Docker 1.8. je mimo jiné dodáván pomocí nástroje Docker for Mac a Docker for Windows, který automatizuje proces instalace, nastavení Dockeru a poskytuje intuitivní grafické uživatelské rozhraní pro běh kontejnerů Docker. Kitematic umožňuje přepínat mezi Docker CLI a grafickým uživatelským rozhraním. Také automatizuje pokročilé funkce, jako jsou správa portů a konfigurace svazků. Kitematic je také možné stáhnout přímo z oficiálního repozitáře https://github.com/docker/kitematic/releases, ve kterém je možné najít i verzi pro operační systém Ubuntu. Jak vypadá prostředí tohoto GUI nástroje, ukazuje obrázek č. 10.



Obr. 10 - Zobrazení informací o kontejneru v aplikaci Kitematic

Protože je dodáván spolu s nástrojem Docker for Mac nebo Docker for Windows, je skvělou alternativou, jak začít s Dockerem. Oproti ostatním zde zmíněným nástrojům je ale nevýhodou jeho nízké pokrytí různých funkcí Dockeru a je téměř nepoužitelný jako řešení pro více než jeden projekt. Obsahuje pouze následující funkcionality [31] [33]:

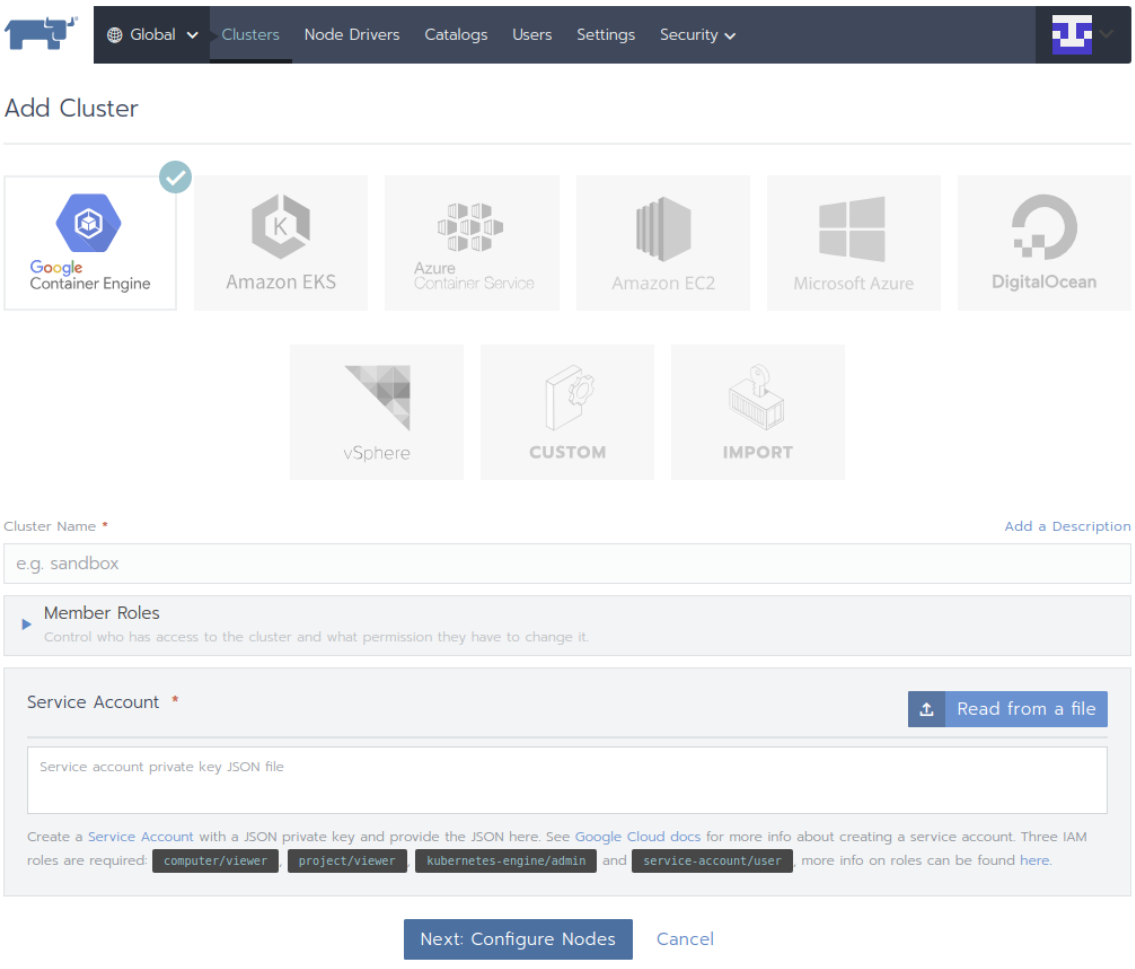
- seznam kontejnerů,
- základní operace s kontejnery,
- seznam sítí a datových svazků,
- příkazová řádka apod.

3.4.3 Rancher

Rancher je open-source nástroj, který kombinuje vše, co organizace potřebuje k tomu, aby provozovala kontejnery v produkčním prostředí. Zatímco předchozí nástroje jsou navrženy pro vývojáře, nebo ty, kteří s Dockerem začínají, Rancher je nástroj, který toho pomocí GUI[22] zvládne mnohem více. Založený na Kubernetes, usnadňuje týmům Operations testovat, nasazovat a spravovat své aplikace [34]. Lze ho jednoduše nainstalovat a spustit pomocí příkaz, který je v ukázce č. 61.

```
$ sudo docker container run -d \
--restart=unless-stopped \
-p 80:80 \
-p 443:443 \
rancher/rancher
```

Ukázka 61 - Instalace aplikace Rancher



Obr. 11 - Náhled na GUI aplikace Rancher

[22] GUI je zkratkou grafického uživatelského rozhraní, pomocí kterého lze aplikaci ovládat interaktivními grafickými prvky.

3.5 Logování a monitorování

Logování v kontejnerové infrastruktuře, přesněji v Dockeru, je horkým tématem. Kontejnery mění povahu logování a postupem času se poukázalo na několik osvědčených postupů.

Stejně jako u jakékoli služby je logování hlavní součástí Dockeru. Analýza logů poskytuje přehled o výkonu, stabilitě a spolehlivosti kontejnerů a samotné službě Docker. Nicméně vzhledem k flexibilní a dynamické povaze Dockeru neexistuje jediný přístup ke shromažďování a ukládání. Místo toho je k dispozici řada řešení, z nichž každé má své opodstatněné výhody a nevýhody [35].

3.5.1 Logování na standardní výstup

Prvním nejjednodušším přístupem, kterým Docker disponuje, je logování kontejnerů na standardní výstup (STDOUT) a standardní chybový výstupho (STDERR). Zobrazení logovaných informací pak lze provést příkazem *docker container logs*.

Jakým způsobem jsou logy předávány k cíli, definují ovladače tzv. logging drivers. Každý Docker démon má výchozí ovladač *json-file*, který používá každý kontejner, pokud není v konfiguraci řečeno jinak. Seznam všech dostupných ovladačů je přehledně vysvětlen v dokumentaci [26].

V některých případech však nemusí tento přístup zobrazovat všechny užitečné informace, obzvláště pokud některá z aplikací loguje svůj výstup do souboru namísto STDOUT a STDERR. Ovšem i tak je možné tento problém vyřešit a příkladem může být oficiální distribuce *nginx* obrazu, který vytváří symbolický link z */var/log/nginx/access.log* do */dev/stdout* a další z */var/log/nginx/error.log* do */dev/stderr* [10].

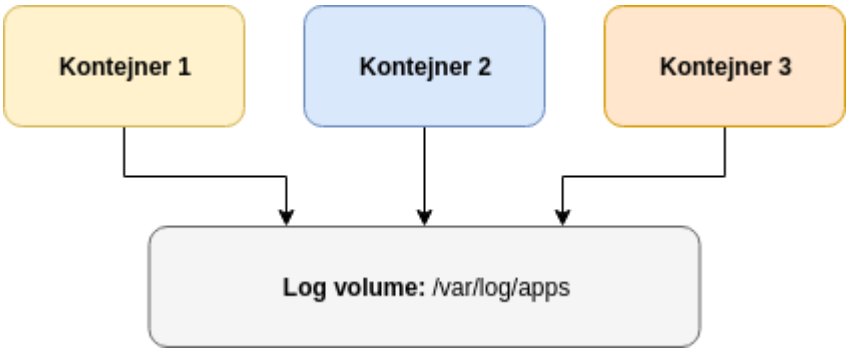
3.5.2 Logování aplikace do kontejneru

Proces logování aplikace do souboru v kontejneru pravděpodobně většina vývojářů zná. Pro případ je tento proces vyobrazen na obrázku č. 12. Aplikace spuštěná uvnitř kontejneru provádí vlastní logování a odesílá logy do určeného místa. Logování je omezeno pouze na samotný kontejner, takže všechny logy uložené v souborovém systému kontejneru jsou ztraceny, pokud dojde k zániku kontejneru. Je proto vhodné logované informace přesměrovat na STDOUT nebo v lepším případě logy sdílet prostř. datového svazku dostupném v hostitelském systému.

Obr. 12 - Logování do kontejneru
Zdroj: vlastní zpracování

3.5.3 Logování do datového svazku

Vzhledem k tomu, že jsou všechny logovací soubory v případě logování aplikace do kontejneru s jeho zánikem ztracené, musí se logy předávat buď do nějaké centralizované logovací služby, nebo do datového svazku na hostitelském systému. S datovým svazkem je možné ukládat dlouhodobá data do kontejnerů mapováním adresáře v kontejneru do adresáře hostitelského počítače [35]. Lze také sdílet jeden svazek v několika kontejnerech kvůli centralizaci logování mezi více službami. Datové svazky jsou účinné pro centralizaci a ukládání logů po delší dobu. Protože odkazují na adresář v hostitelském počítači, významně snižují pravděpodobnost ztráty dat v důsledku selhání kontejneru. Vzhledem k tomu, že jsou nyní data k dispozici hostitelskému počítači, lze provádět kopie, provádět zálohování nebo k nim dokonce přistupovat z jiných kontejnerů. Jedná se o velmi vhodnou metodu obzvláště pro vývojový proces. Princip logování do datového svazku naznačuje obrázek č. 13.



Obr. 13 - Logování do datového svazku
Zdroj: vlastní zpracování

3.5.4 Logování do dedikovaného kontejneru

Zatímco dvě předchozí metody mají několik výhod, sdílejí společnou nevýhodu: závisí na hostitelském počítači. Na druhé straně dedikované kontejnery umožňují spravovat logování přímo z prostředí Dockeru a načítat logy z jiných kontejnerů, agregovat, uchovávat je nebo předávat jiným službám třetích stran. Tento přístup eliminuje závislost kontejnerů na hostitelském počítači. V praxi to funguje tím způsobem, že všechny kontejnery logují do jednoho společného dedikovaného kontejneru [36].

Tato metoda není vhodná pro vývojový proces vzhledem k její komplexnosti a časové náročnosti instalace. V praxi je používána převážně v produkčním prostředí týmem Operations použitím tzv. ELK Stacku, což je skupina nástrojů (Elasticsearch, Logstash a Kubana), která slouží ke sběru dat z kontejnerů, indexaci, analýze a následné vizualizaci.

3.5.5 Monitorovací nástroje

Existuje mnoho způsobů, jak monitorovat Docker kontejnery. Tento odstavec představuje několik velmi jednoduchých a snadno použitelných možností.

Docker stats

Příkaz docker container stats podává aktuální informace následujících statistik jednotlivých kontejnerů:

- CPU % využití,
- využití paměti, limit, % využití,
- využití síťě i/o,
- využití disku i/o.

Statistiky jsou aktualizovány každou vteřinu, jak znázorňuje kapitola 2.5.2.

Docker Remote API

Docker démon poskytuje REST API rozhraní pro komunikaci klienta s Docker Engine. Většinou je nutné tuto možnost povolit následujícími kroky [37]:

1. Vytvořit soubor */etc/systemd/system/docker.service.d/startup_options.conf* s následujícím obsahem:

```
$ /etc/systemd/system/docker.service.d/override.conf
[Service]
ExecStart=
ExecStart=/usr/bin/dockerd -H fd:// -H tcp://0.0.0.0:2376
```

2. Načíst soubory

```
$ sudo systemctl daemon-reload
```

3. Restartovat docker démona

```
$ sudo systemctl restart docker.service
```

4. Zobrazení statistiky je potom možné přes URL v tomto tvaru:

```
$ curl http://localhost:2376/containers/5cc93a52ad0d/stats
```

Výsledkem je zobrazení výstupu ve formátu JSON, jak je možné vidět v ukázce č. 62.

```
{
   "read":"2018-07-07T05:22:00.988697731Z",
   "preread":"0001-01-01T00:00:00Z",
   "pids_stats":{
      "current":47
   },
...
```

Ukázka 62 - Docker Stats JSON výstup

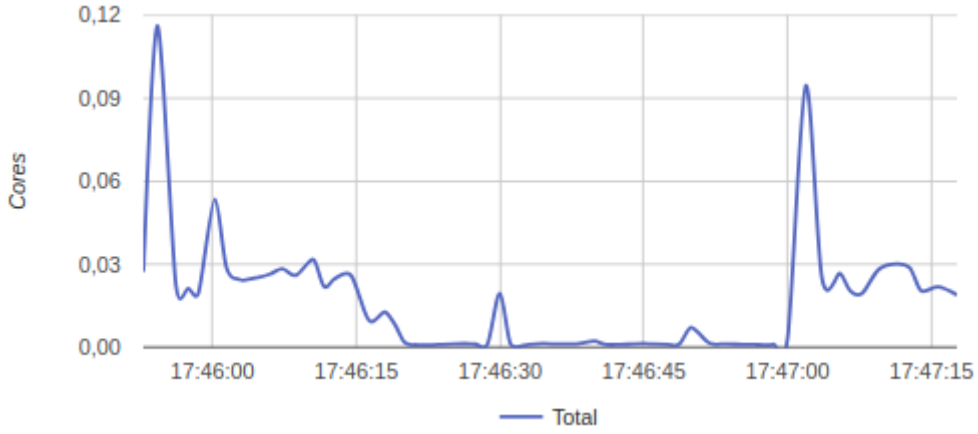
cAdvisor

Container Advisor je jednoduchý nástroj, běžící démon, který poskytuje metriky kontejnerů. Shromažďuje, agreguje, zpracovává a exportuje informace o běžících kontejnerech. Lze ho spustit jako samostaný kontejner [38]:

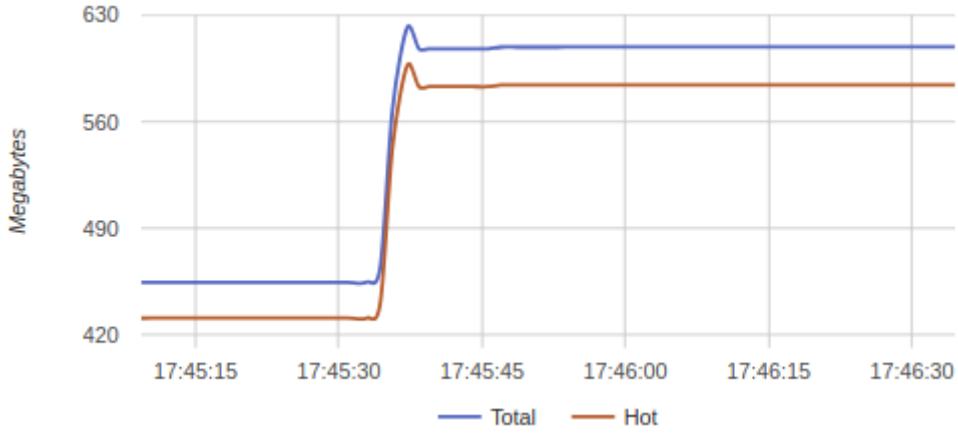
```
$ sudo docker container run \
 -d --name=cadvisor \
 -p 8088:8080 \
 --volume=/var/run:/var/run:rw \
 --volume=/sys:/sys:ro \
 --volume=/var/lib/docker/:/var/lib/docker:ro \
 google/cadvisor:latest
```

Služba cAdvisor je nyní spuštěna (na pozadí) na adrese *http://localhost:8088.* Obrázky č. 14 a č. 15 ukazují, jak vypadají různé grafy tohoto nástroje.

Informační panely zobrazují data pouze za posledních 60 vteřin.



Obr. 14 - Celkové využití CPU kontejneru bikeoapp



Obr. 15 - Celkové využití paměti kontejneru bikeoapp

Existuje spousta nástrojů, které mohou využívat data generovaná službou cAdvisor a zobrazit je tak na přehledném panelu webové stránky. Více informací lze najít na *github.com/google/cadvisor/tree/master/docs*.

Prometheus

Dalším nástrojem je Prometheus, sada nástrojů, která společně zajišťuje metriky pro ukládání, agregaci, vizualizaci a upozorňování. Pro Prometheus je dostupných několik exportérů, kteří zachycují metriky a poté je vystavují přes webové rozhraní. Kromě toho existují knihovny, které lze použít k vytvoření vlastních exportérů. Pro sledování kontejnerů je možné použít připravený tzv. *container_exporter* dostupný z https://github.com/docker-infra/container_exporter, který poskytuje seznam kontejnerů běžících na hostitelském počítači a shromažďuje různé metriky, které dále poskytuje [39].

Pomocí příkazu uvedeného v ukázce č. 63 je vyvoláno spuštění kontejneru *container-exporter*.

```
$ sudo docker container run -d --name exporter \
-p 9104:9104 \
-v /sys/fs/cgroup:/cgroup \
-v /var/run/docker.sock:/var/run/docker.sock \
prom/container-exporter
```

Ukázka 63 - Spuštění kontejneru container-exporter pro získání metrik

Jednotlivé metriky, které exportér zajistil, lze zhlédnout na adrese *http://localhost:9104/metrics*. Jakmile je připraven běžící exportér, lze spustit Prometheus server. Předtím je ale nutné vytvořit konfigurační soubor *prometheus.yml*, který serveru slouží pro získání metrik, dle kterých je možné provádět monitoring. Pro demonstraci tohoto nástroje je v aktuálním adresáři vytvořen soubor YAML s následujícím obsahem.

```
global:
  scrape_interval: 15s
scrape_configs:
  - job_name: 'prometheus'
    scrape_interval: 5s
    static_configs:
      - targets: ['localhost:9090']
  - job_name: "node"
    scrape_interval: "15s"
    static_configs:
      - targets: ['exporter:9104']
```

Ukázka 64 - prometheus.yml

V tomto souboru existují dvě sekce – global a sekce pro konfiguraci jednotlivých úloh (jobs). V global části je nastaven interval sběru dat (scrape_interval) na 15 vteřin.

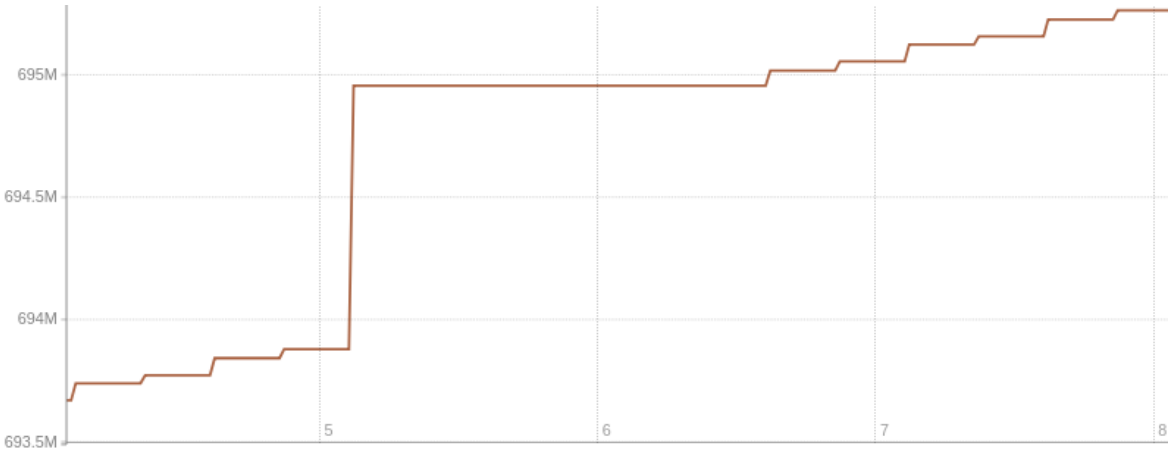
V sekci jobs lze definovat jednu nebo více úloh, z nichž každá má svůj název, volitelný interval sběru dat (přetěžuje interval z globálního nastavení), cíle, ze kterých lze měřit metriky apod. V ukázce jsou celkem dva cíle, jeden je samotný server Prometheus a druhý je kontejner s exportérem vytvořeným v přechozím kroku [39].

Nyní, když je soubor vytvořený, je možné přejít k ukázce č. 65, ve které dochází ke spuštění Prometheus kontejneru z aktuálního adresáře díky volně dostupnému obrazu *prom/prometheus*.

```
$ sudo docker container run -d --name prometheus-server \
-p 9090:9090 \
--link=exporter \
-v $PWD/prometheus.yml:/etc/prometheus/prometheus.yml \
prom/prometheus
```

Ukázka 65 - Spuštění kontejneru s Prometheus serverem

Po spuštění kontejneru je webové rozhraní serveru k dispozici v prohlížeči na adrese *http://localhost:9090*. Poté je možné provést analýzu dat pomocí dotazů k odfiltrování dat. Následující obrázek č. 16 znázorňuje graf maximálního využití paměti kontejneru *bikeoapp* dle dotazu:
container_memory_max_usage_bytes {image="sipekmichal/bikeoapp:45"}.



Obr. 16 - Graf maximálního využití paměti kontejneru bikeoapp
Zdroj: vlastní zpracování

Všechny metriky z kontejnerů jsou označeny názvem obrazu, názvem kontejneru a hostitelem, na kterém je kontejner spuštěn. Tyto dočasné grafy jsou skvělé pro občasnou investigaci, ale pro perzistentní ukládání dat a grafů je použit nástroj Prometheus Dashboard Builder.

Prometheus je komplexním nástrojem vhodným především pro Operations týmy, které ho často integrují s nástrojem Grafana, grafickým řešením pro vizualizaci různých metrik.

4 Implementace Docker do procesu vývoje aplikací

Studium typických problémů, se kterými se již většina lidí z Docker komunity setkala, by mělo být prvotním krokem. Spousta uživatelů z praxe totiž opakuje ty samé chyby pramenící zpravidla z nepochopení principu nebo neznalosti nástroje Docker. V této kapitole je snahou hned zpočátku na tyto problémy upozornit a poté zmínit některé postupy vytváření Dockerfile souborů, které se v průběhu kontejnerizace osvědčily. V neposlední fázi se přechází k migraci existující Java aplikace do jednotlivých kontejnerů, na níž je předvedena demonstrace toho, jak se v takovém procesu postupuje a nakonec zavedení průběžné integrace pro automatizované přenasazení aplikace.

4.1 Typické problémy implementace

Postupnou kontejnerizací aplikací a rozšiřující se komunitou uživatelů, kteří Docker nově implementují do svých řešení, se na internetu v různých zdrojích objevují tzv. anti-patterns (špatné vzory). Většina těchto příkladů vychází z neporozumění kontejnerizace. Mým cílem je definovat tyto problémy a poskytnout jejich pevné pochopení.

Uchovávání dat v kontejnerech

Kontejnery nejsou stavěné pro uchovávání dat, naopak jsou ideální pro aplikace, u kterých nezáleží na jejich stavu. To znamená, že v kontejneru by neměly být uloženy žádné údaje ani logy, jinak se ztratí, až kontejner skončí. Namísto toho se používá mapování sdíleného adresáře, aby bylo možné udržet tato data mimo kontejnery [37].

Používání IP adres kontejnerů

Každému kontejneru je přiřazena IP adresa a to uživatele svádí použít tuto adresu ke komunikaci s jinými kontejnery. Aplikace, která se skládá z vícera mezi sebou komunikujících kontejnerů, se může například připojovat k databázi. Problém nastává opět v případě, kdy kontejner zanikne a vznikne nový. Spoléhání se na adresu IP kontejneru bude vyžadovat neustálou aktualizaci konfigurace aplikace. To způsobí, že aplikace bude náchylná na chyby.

Namísto toho je vhodné mezi takovými kontejnery použít linky s logickými jmény, na které lze odkazovat nezávisle na rostoucím nebo klesajícím počtu kontejnerů. Docker navíc od verze 1.10 přišel s podporou DNS[23], takže pokud jsou kontejnery připojeny ke stejné síti, mohou se navzájem odkazovat pomocí názvu kontejneru.

Spouštění více procesů

Při vytváření obrazu pomocí souboru Dockerfile je na konci vždy uvedena instrukce *CMD* nebo *ENTRYPOINT*, která provede určitou konfiguraci obrazu a poté spustí kontejner. Cílem není spustit více procesů pomocí tohoto skriptu [37]. Je důležité, aby se při vytváření obrazů identifikovaly a separovaly dílčí procesy, které budou odpovídat jednotlivým kontejnerům.

Nevhodné používání příkazu docker container exec

Dalším nevhodným příkladem je používání příkazu *docker container exec*. Příkaz spouští nový příkaz v běžícím kontejneru. To je užitečné pro připojení shellu pomocí *docker container exec -it {cid} bash*. Ale v jiném případě má v kontejneru běžet pouze ten proces, který je definován v Dockerfile. Záměrem je nesnažit se spouštět jiné příkazy, které by porušily zapouzdření.

Redundantní data v obrazu

V první fázi Docker kontejnerizace je třeba se snažit o zbavení se redundantních dat v obrazu. Je tedy vhodné vždy vytvořit nový adresář, ve kterém je umístěn soubor Dockerfile a další relevantní soubory. Je také na zvážení použití *.dockerignore*, aby se před vytvořením obrazu odstranily všechny logy, zdrojový kód atd. Tím je zaručeno, že obraz bude obsahovat ta data, která potřebuje.

Vytváření obrazu z běžícího kontejneru

Ačkoliv lze nový obraz vytvořit z běžícího kontejneru pomocí příkazu *docker container commit*, není tato varianta ve většině případů vhodná. O možnosti, kdy se použití *docker container commit* naopak stává užitečné, je zmíněno v předchozích kapitolách. Celý problém spočívá v tom, že obrazy vytvořené tímto způsobem nejsou reprodukovatelné [37]. Namísto toho se provádí změny v souboru Dockerfile, ukončí se stávající kontejner a z aktualizovaného obrazu spustí nový kontejner.

[23] DNS umožňuje přiřadit k číselné IP adrese určité symbolické jméno, tzv. jméno domény, které si uživatelé snadno zapamatují a dokážou jej i intuitivně napsat např. do webového prohlížeče.

Ukládání tajných informací v obrazu

Je pravidlem, že jakékoliv ukládání tajných informací, hesel apod. do souboru Dockerfile není přípustné. Údaje uložené v takové formě (textovém souboru) jsou totiž lehce čitelné, a tím pádem zranitelné. Namísto toho by k předávání hesel měla sloužit proměnná prostředí, tzv. *ENV*. Zadané heslo se předá v rámci sestavení obrazu.

Nadměrné používání tagu latest

Dalším nešvarem při sestavování obrazů je nadměrné používání tagu *latest*. Pokud při sestavení obrazu nejsou zadány žádné tagy, pak je automaticky obraz označen jako latest. To ale nemusí znamenat, že se jedná skutečně o aktuální verzi tohoto obrazu. Nadměrné užívání tagu *latest* vede k nekonzistencím. Spuštění kontejneru například v produkčním prostředí vyžaduje plnou kontrolu a řízení nad jednotlivými kontejnery a jejich verzemi. V praxi se nejvíce osvědčuje označení obrazu použitím tagu s jednoznačným identifikátorem.

Latest neznamená nejlepší

Při sestavování obrazů je důležité pochopit a rozhodnout, kterou verzi základního obrazu zvolit. Je neuvěřitelně lákavé při psaní souboru Dockerfile, aby bylo dosaženo nejnovější verze každé závislosti. Většina veřejných obrazů, které jsou k dispozici na Docker Hubu, má jeden tag *latest* a dokonce, i když mají více deterministickou verzi, nezkušení uživatelé mají tendenci používat tu poslední. Problém nastává v situaci, kdy u veřejného obrazu dojde k vydání nové verze. Znovusestavením obrazu, který z něho vychází, dojde k aktualizaci [41]. Nezkušený uživatel tuto změnu nemusí vůbec zaregistrovat. Zlatým pravidlem je vytvoření kontejnerů se známými a stabilními verzemi systému a závislostí, na kterých bude požadovaný software fungovat.

Různé obrazy pro jednotlivá prostředí

V praxi se občas vyskytují případy, kdy se používají různé obrazy pro dev, test a produkční prostředí. Obraz by měl být vytvořen jednou, následně uložen do repozitáře a odtud použit pro různá prostředí [41]. Obzvláště integrační testování by mělo být provedeno na stejném obrazu, který bude nasazen do produkce.

Spouštění kontejnerů pod uživatelem root

Je nevhodné spouštět kontejnery jako root uživatel. Jde o to, že hostitelský systém a kontejner sdílejí stejné jádro a pokud by byl kontejner kompromitován, může uživatel root tento systém poškodit. Namísto toho lze v souboru Dockerfile použít instrukce *RUN* k vytvoření skupiny a uživatele v ní. Instrukce *USER* pak slouží k přepnutí na daného uživatele. Je dobré mít na paměti, že každá taková instrukce vytvoří novou vrstvu v obrazu. Cílem je vyhnout se nadměrnému přepínání uživatele tam a zpět, aby se snížil celkový počet vrstev [37].

Používání SSH v kontejnerech

Stejně nešťastnou praxí je instalace ssh démona do obrazu. Jedná se o normální impuls, protože zde existují návyky z oblasti virtualizace a virtuálních strojů. Z pohledu kontejnerizace a tzv. neměnné infrastruktury je však tento krok chybný. Změna obsahu kontejneru musí být zdokumentovaná prostřednictvím souboru Dockerfile [41].

4.2 Osvědčené postupy psaní Dockerfile

Tato kapitola pokrývá osvědčené postupy, doporučení a metody pro vytváření efektivních obrazů. Docker automaticky vytváří obrazy tak, že si přečte instrukce z Dockerfile - textového souboru, který obsahuje všechny příkazy v pořadí potřebné k vytvoření daného obrázku. Dockerfile dodržuje určitý formát a sadu pokynů.

Vytváření „dočasných“ kontejnerů

Obraz vymezený prostřednictvím souboru Dockerfile by měl vytvářet kontejnery, které jsou dočasné. Termínem ,,dočasné" se rozumí, že kontejner může být zastaven a zničen, poté znovu sestaven a nahrazen s minimálním zásahem do konfigurace. Je důležité Dockerfile koncipovat s ohledem na tuto skutečnost [42] .

Porozumění kontextu sestavení, tzv. build context

Při sestavování obrazu, resp. spouštění příkazu *docker image build* se aktuální pracovní adresář nazývá kontext sestavení. Ve výchozím nastavení se předpokládá, že soubor Dockerfile je umístěn zde, ale lze zadat i jiné umístění pomocí příznaku *-f*. Bez ohledu na to, kde je skutečně umístěn Dockerfile, je veškerý obsah adresářů včetně souborů v nich umístěných odesílán démonu [42]. Zpráva z ukázky č. 66 informuje o velikosti kontextu sestavení, která se pokaždé objevuje při sestavení obrazu a je důležité jí věnovat pozornost.

```
Sending build context to Docker daemon  247.9MB
```

Ukázka 66 - Hláška o velikosti kontextu

Neúmyslné zahrnutí souborů, které nejsou nezbytné pro vytvoření obrazu, má za následek větší kontext sestavení a větší výslednou velikost obrazu. Tím se může prodloužit doba vytváření obrazu, nahrání do registru apod.

Oddělení aplikací

Každý kontejner by měl mít jediný zájem. Oddělení aplikací do více kontejnerů usnadňuje opakované použití kontejnerů. Například webová aplikace se může skládat ze tří samostatných kontejnerů, z nichž každý má svůj vlastní jedinečný obraz pro správu webové aplikace, databázi a vyrovnávací paměť. Omezení každého kontejneru na jeden proces je dobrým pravidlem, aby nedocházelo k tomu, že některé programy samy o sobě způsobí další spuštění několika procesů. Proto je vhodné používat svého nejlepšího úsudku k tomu, aby kontejnery zůstaly modulární a co nejčistší [42].

Minimalizace počtu vrstev

Ve starších verzích Dockeru bylo velmi důležité snažit se o minimalizaci počtu vrstev výsledného obrazu k zajištění co největší výkonnosti. Postupným vývojem byly přidány další funkce, které tuto problematiku usnadňují. Nejprve přišel Docker ve verzi 1.10 s tím, že všechny ostatní instrukce kromě *RUN*, *COPY*, *ADD* vrstvy nevytváří, ale vytváří pouze jakési meziobrazy, které nezvyšují výslednou velikost. Instrukce, které vrstvy vytváří, je dobré převést v Dockerfile do takového tvaru (pokud jich existuje více), který sloučí instrukce do jedné a zajistí, že výsledný obraz je co nejmenší. Příklad je uveden v ukázce č. 67.

```
FROM openjdk:8-jdk AS build
RUN apt-get update
RUN apt-get upgrade -y
RUN apt-get install git-all –y

FROM openjdk:8-jdk AS build
RUN apt-get update && apt-get upgrade -y && \
    apt-get install git-all -y
```

Ukázka 67 - Minimalizace vrstev obrazu a počtu instrukcí v Dockerfile

Následně verze 17.05 a výše přinesla funkci vícestupňových sestav, tzv. návrhový vzor Multi-stage build (pojednává kapitola 2.4.4), která taktéž redukuje výsledný obraz.

4.3 Kontejnerizace existující Java aplikace

Tato kapitola je věnována demonstraci procesu migrace Java aplikace do architektury založené na kontejnerech pomocí nástroje Docker. Cílem je ukázat, jak se v takovém procesu postupuje a zároveň poskytnout nejlepší možnou šablonu pro další potenciální využití. V celém procesu se postupuje v několika krocích, kde v první fázi dochází k seznámení se s postupy samotné migrace včetně popisu a vysvětlení. V další fázi je kontejnerizace zapojena do procesu průběžné integrace (Continuous Integration), aby došlo k automatizaci a urychlení vývoje. Výsledkem je přenositelná aplikace běžící v několika spolu komunikujících kontejnerech, které se dle změn v kódu automaticky aktualizují.

4.3.1 Kontejnerizace aplikace

Prvním krokem je kontejnerizace aplikace, aniž by byl změněn existující kód. V tomto příkladě je znázorněna migrace webové Java aplikace, která pro sestavení používá Maven a je nasazena v aplikačním serveru Tomcat. Data jsou uložena v MySQL databázi. Výsledkem migrace jsou dva kontejnery, jeden pro server s webovou aplikací a druhý pro databázový server.

Stažení aplikace a příprava prostředí

Jak už bylo řečeno, webová aplikace používá mimo jiných technologií Maven[24] a Tomcat. Slouží jako ukázková aplikace, video portál, který je určen pro sdílení cyklistických videí ze zábavného serveru YouTube. Aplikace Bikeo byla použita jako závěrečná práce v předmětu PPRO a je dostupná na serveru GitHub prostřednictvím odkazu https://github.com/sipekmichal/bikeo, ale také v příloze. Předpokladem připraveného prostředí je nainstalovaný Docker CE 18.03.1 a Docker Compose.

Pro práci s Dockerem ve vývojovém prostředí a následnou kontejnerizací se velmi osvědčují různé pluginy, které jsou dostupné pro řadu prostředí. V tomto příkladu je použito vývojové prostředí IntelliJ IDEA s pluginem Docker integration, pomocí něhož lze z prostředí jednoduše sestavovat obrazy a stahovat je z registrů, spouštět z nich kontejnery a mnoho dalšího. Pro vývojáře je to velmi užitečný nástroj s kontrolou a zvýrazněním syntaxe.

[24] Maven je nástroj pro správu, řízení a automatizaci buildů a sestavování aplikací postavených nad platformou Java.

Vytvoření souboru Dockerfile

V rámci vytváření souboru Dockerfile, který je zobrazen v ukázce č. 68, je použit návrhový vzor Multi-stage build (vícestupňové sestavení), kdy v první části dochází ke kompilaci zdrojového kódu za použití Maven kontejneru. Všechny závislosti jsou staženy do tohoto kontejneru na základě souboru *pom.xml*. Nástroje pro sestavení aplikace jsou zcela v samostatném kontejneru a tím je zaručeno, že sestavení bude vždy fungovat nezávisle na prostředí.

```
# Build the app using Maven in a container
#
FROM maven:3.5.3 AS mavencontainer
LABEL maintainer="mail@sipekmichal.cz"
WORKDIR /usr/src/bikeo
COPY ./app .
RUN mvn clean package -DskipTests
```

Ukázka 68 - Dockerfile webové aplikace 1

Instrukce zahrnuté v souboru jsou celkem prosté. Při sestavení obrazu dochází ke stažení základního obrazu Maven verze 3.5.3 z oficiálního repozitáře Docker Hub. Z předchozích kapitol je známo, že definovat verzi namísto použití tagu latest, se ve většině případů osvědčuje. Následuje instrukce *LABEL*, která identifikuje vlastníka obrazu a *WORKDIR* nastavující pracovní adresář. Na konci se volá příkaz, který provádí sestavení aplikace ze zdrojového kódu na základě souboru *pom.xml*, který se předtím spolu s kódem v adresáři app zkopíroval do aktuálního umístění.

Druhá část vícestupňového sestavení z ukázky č. 69 nasazuje aplikaci a konfiguruje aplikační server Tomcat přidáním vlastních konfiguračních souborů. Nakonec dochází ke kopírování aplikace zabalené v souboru WAR a spouštění kontejneru.

```
# Deploy application to Tomcat
#
FROM tomcat:7-jre8
# tomcat-users.xml sets up user accounts for the Tomcat manager GUI
COPY tomcat/tomcat-users.xml $CATALINA_HOME/conf/
COPY tomcat/run.sh $CATALINA_HOME/bin/run.sh
RUN chmod +x $CATALINA_HOME/bin/run.sh

# Create mount point for volume with application
WORKDIR $CATALINA_HOME/webapps/
COPY --from=mavencontainer /usr/src/bikeo/target/bikeo*.war bikeo.war

# Start tomcat7
EXPOSE 8080
CMD ["run.sh"]
```

Ukázka 69 - Dockerfile webové aplikace 2

Díky němu je možné vytvořit obraz, který plní funkci aplikačního serveru a zbývá už jen vytvořit Dockerfile pro sestavení obrazu MySQL databáze, jehož obsah je v ukázce č. 70.

```
FROM mysql:5.7
LABEL maintainer=mail@sipekmichal.cz
# Copy the database initialize script:
# Contents of /docker-entrypoint-initdb.d are run on mysqld startup
ADD  docker-entrypoint-initdb.d/ /docker-entrypoint-initdb.d/
```

Ukázka 70 - Dockerfile MySQL databáze

Sestavení obrazů Dockerfile je vysvětleno v následující kapitole.

4.3.2 Sestavení a spuštění aplikace

Soubory Dockerfile jsou vytvořené a umístěné v adresářích *./appserver* a *./database*. Je tedy možné provést sestavení obrazu pomocí příkazu *docker image build* a následné spuštění aplikace. Nabízí se zde předem otázka, zda nepoužít soubor *.dockerignore*, aby výsledné obrazy byly, co se velikosti týká, minimální. V tomto případě se však použití tohoto souboru nehodí, protože veškeré soubory, umístěné v rámci kontextu sestavení, jsou pro sestavení nepostradatelné.

```
$ cd ./appserver
$ sudo docker image build -t sipekmichal/bikeoapp:1.0.0 .
$ cd ../database
$ sudo docker image build -t sipekmichal/bikeomysqldb:5.7 .
```

Ukázka 71 - Příprava adresářové struktury pro kontext sestavení

Předchozí ukázka č. 71 zobrazuje správné tagování při buildu obrazu. Kromě názvu kontejneru je dosazena i verze, která zřetelně identifikuje daný obraz. Následujícími příkazy z ukázky č. 72 lze aplikaci spustit, vytvoří se dva démonizované kontejnery s vystavenými porty.

```
$ sudo docker container run -d -p 3306:3306 \
-e MYSQL_ROOT_PASSWORD=passw0rd \
-e MYSQL_DATABASE=bikeo \
-e MYSQL_USER=admin \
-e MYSQL_PASSWORD=admin \
--name bikeodb \
sipekmichal/bikeomysqldb:5.7
$
$ sudo docker container run -d -p 8080:8080 \
--link bikeodb:database \
--name bikeoapp \
sipekmichal/bikeoapp:1.0.0
```

Ukázka 72 - Spuštění kontejnerů

Je dobré si všimnout, že se při spuštění kontejneru s databází definují proměnné nastavující přístupové údaje MySQL.

Spoustu lidí možná napadne tyto proměnné definovat instrukcí *ENV* přímo v souboru Dockerfile, což je samozřejmě možné maximálně pro vývojové prostředí, ale obecně to není osvědčený a vhodný způsob, jak tyto údaje předávat.

Výsledkem spuštění předchozích příkazů je běžící webová aplikace ve dvou kontejnerech na adrese *http://localhost:8080/bikeo*. Seznam těchto kontejnerů, který je v ukázce č. 73, lze zobrazit příkazem *docker container ps*. Náhled na webovou aplikaci pak zobrazují přílohy č. 3-6.

```
CONTAINER ID        IMAGE                  COMMAND
CREATED             STATUS              PORTS                    NAMES
6610142d2789      sipekmichal/bikeoapp:1.0.0      "run.sh"
2 seconds ago       Up 1 second         0.0.0.0:8080->8080/tcp
bikeoapp
17e39df97d4a      sipekmichal/bikeomysqldb:5.7    "docker-
entrypoint.s…"   12 minutes ago      Up 12 minutes       0.0.0.0:3306-
>3306/tcp   bikeodb
```

Ukázka 73 - Seznam běžících kontejnerů

Celý předešlý proces lze s nástroji Docker částečně i plně automatizovat. Některými těmito možnostmi a jejich implementacemi se zabývají následující kapitoly.

4.3.3 Automatizace sestavení a nasazení

Docker Compose umožňuje automatizovat proces sestavení a nasazení aplikace pomocí jediného souboru. Používá k tomu textový soubor strukturovaný podle jazyka YAML. Slouží ke konfiguraci služeb aplikace, kde poté prostřednictvím jediného příkazu lze vytvořit a spustit všechny tyto služby. Sestavení obrazů a spuštění kontejnerů z předchozího příkladu je nyní potřeba převést do strukturované podoby, která zajistí vše potřebné.

```
version: "3.6"
services:
  database:
    image: sipekmichal/bikeomysqldb:5.7
    build:
      context: ./database
    environment:
      - MYSQL_ROOT_PASSWORD=password
      - MYSQL_DATABASE=bikeo
      - MYSQL_USER=admin
      - MYSQL_PASSWORD=admin
    restart: always
  webserver:
    image: sipekmichal/bikeoapp:latest
    depends_on:
       - database
    links:
       - database
    build:
      context: ./appserver
    ports:
      - "8080:8080"
      - "8000:8000"
    volumes:
      - /tmp/logs/bikeoapp:/usr/local/tomcat/logs
    restart: always
```

Ukázka 74 - docker-compose.yml aplikace bikeo.cz

Ve výše uvedené ukázce č. 74 je zobrazen vyhotovený soubor *docker-compose.yml*, který je sestaven ze dvou základních služeb – *database* pro databázový kontejner a *webserver* určený pro spuštění kontejneru s aplikačním serverem. Služby jsou založené na souboru Dockerfile umístěném v adresáři *./database* a *./appserver* pomocí definice context určující kořenový adresář kontextu sestavení pro vytvoření obrazu. Direktiva *image* určují, z jakého obrazu služba vychází a je následně provedeno sestavení. Pro nastavení přístupů do MySQL databáze je zde použito proměnné prostředí *environment*. Za zmínku stojí část *volumes*, která perzistentně ukládá data z kontejneru. Tím je zaručeno, že v okamžiku zániku kontejneru nedojde ke ztrátě doposud uložených dat. V poslední řadě je nutné namapovat porty jednotlivých služeb.

Sestavení obrazů a spuštění démonizovaných kontejnerů se vykoná následujícím příkazem z ukázky č. 75.

```
$ sudo docker-compose up --build -d
```

Ukázka 75 - Sestavení obrazů a spuštění kontejnerů s Docker Compose

4.3.4 Monitoring, správa a logování kontejnerů

Pro monitoring kontejnerů v rámci vývoje je plně dostačující přístup pomocí příkazu *docker container stats*, který podává aktuální informace různých statistik. Tímto rychlým způsobem je dosaženo získání důležitých informací, které jsou vhodné především pro účely ladění nebo získání přehledu o celkovém využití zdrojů pro budoucí nasazení do produkčního prostředí

Spíše než monitorovací nástroj vývojáři ocení především možnost jednoduše spravovat kontejnery z jednoho místa pomocí grafického uživatelského rozhraní, vystavovat u nich různé porty, měnit cesty k logům, vytvářet kopie apod. K tomu slouží nástroje zmíněné v kapitole 3.4. Vybrat vhodný nástroj nemusí být jednoduché, ale obecně zde platí, že většina zmíněných nástrojů si je velice podobná a liší se opravdu pouze v několika detailech a funkcí. Záleží na preferencích samotného vývojáře.

Portainer

V části kapitoly týkající se kontejnerizace je zvolen nástroj Portainer především jako podpůrný nástroj z důvodu toho, že má např. oproti Kitematic propracovanější uživatelské rozhraní a více funkcí. Jako příklad lze zmínit vytváření nového obrazu, přidávání nových svazků, sledování historie, vytváření kopií kontejneru, zobrazení statistik, možnost psát do konzole a mnoho dalšího. I zobrazení statistik lze tímto nástrojem, který pro to využívá výhod *docker container stats*. Postup instalace je uveden v kapitole 3.4.2. Obrázek v příloze č. 2 ukazuje statistiky kontejneru *bikeodb* v nástroji Portainer.

Logování do datového svazku

Asi nejběžnější způsob používaný vývojáři je logování do datového svazku, pomocí kterého lze perzistentně ukládat data z kontejneru v adresáři na hostitelském počítači. Je velmi rychlý a podporuje ho většina grafických nástrojů včetně Portainer, díky kterému je možné přidávat a upravovat svazky přímo z webového rozhraní.

V přechozích kapitolách byly vytvářené kontejnery *bikeoapp* a *bikeodb* nejprve klasickým způsobem, tj. příkazem *docker container run* a poté pomocí nástrojem Docker Compose. Následující ukázka č. 76 popisuje, jak by se postupovalo v obou těchto případech.

```
$ sudo docker container run -d -p 3306:3306 \
-e MYSQL_ROOT_PASSWORD=passw0rd \
-e MYSQL_DATABASE=bikeo \
-e MYSQL_USER=admin \
-e MYSQL_PASSWORD=admin \
-p /tmp/logs/bikeodb:/var/log/mysql
--name bikeodb \
sipekmichal/bikeomysqldb:5.7
$ sudo docker container run -d -p 8080:8080 \
-p /tmp/logs/bikeoapp:/usr/local/tomcat/logs \
--link bikeodb:database \
--name bikeoapp \
sipekmichal/bikeoapp:1.0.0
```

Ukázka 76 - Logování do datového svazku - příkaz docker run

Jak je možné si povšimnout, přibyly zde dva řádky:

- -p /tmp/logs/bikeodb:/var/log/mysql,
- -p /tmp/logs/bikeoapp:/usr/local/tomcat/logs,

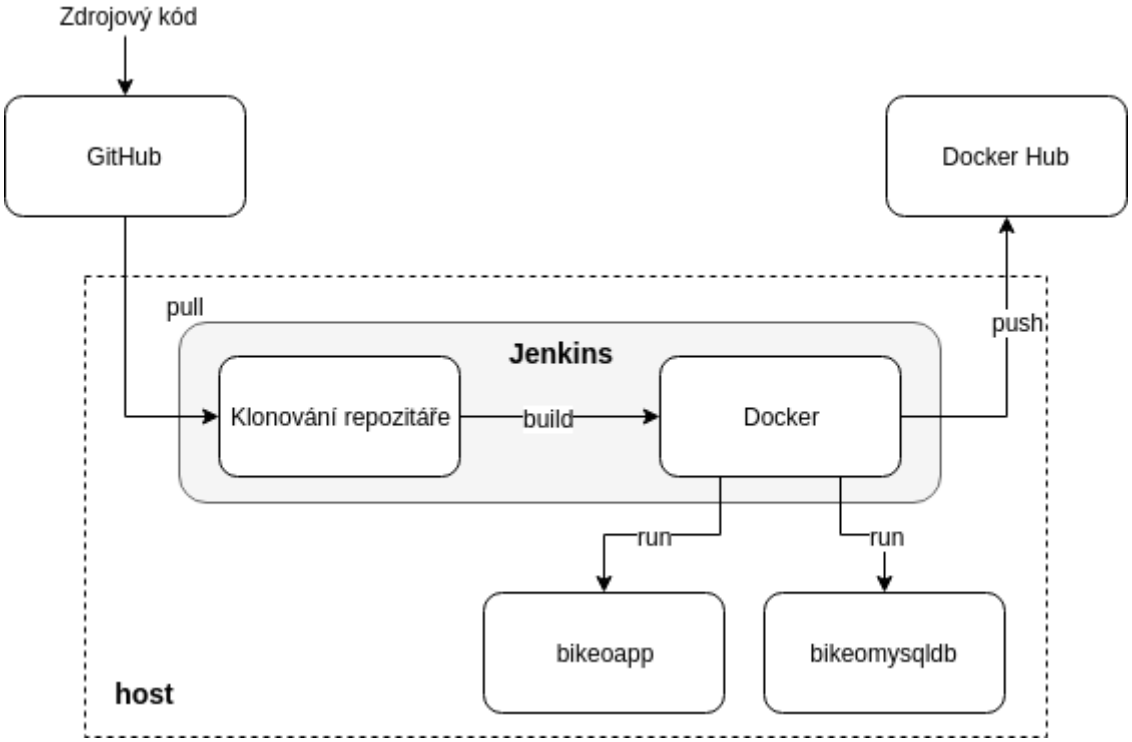
které v adresáři tmp vytvoří další adresáře obsahující logy pocházející z kontejneru databáze MySQL a aplikace Bikeo.cz. Po odstranění kontejnerů logy zůstanou perzistentně uložené. Vzhledem k použití Docker Compose pro sestavení dvojice těchto kontejnerů je potřeba přidat logování i do souboru *docker-compose.yml*. Zde je přidání datového svazku pro uchování a zobrazení logů realizováno definicí *volumes*.
V již vytvořeném *docker-compose.yml* souboru (ukázka č. 72) je pro službu webserver zrealizováno logování do sdíleného svazku. Pro službu database je ještě potřeba svazek přidat (ukázka č. 77).

```
version: "3.6"
services:
  database:
    image: sipekmichal/bikeomysqldb:5.7
    build:
      context: ./database
    environment:
      - MYSQL_ROOT_PASSWORD=password
      - MYSQL_DATABASE=bikeo
      - MYSQL_USER=admin
      - MYSQL_PASSWORD=admin
    volumes:
      - /tmp/logs/bikeodb:/var/log/mysql
    restart: always
```

Ukázka 77 - Logování do datového svazku – část docker-compose.yml

4.3.5 Průběžná integrace

V části průběžné integrace se používá Jenkins jako nástroj pro jednoduchý způsob, jak nastavit kontinuální integrační prostředí. Jako vývojář chci mít prostředek pro automatizované sestavení Docker obrazů a nasazení kontejnerů do prostředí v okamžiku odeslání kódu do repozitáře, abych se mohl podívat na výsledek a případně otestovat funkčnost nového kódu. Pro lepší představu následuje obrázek č. 17, který zobrazuje jednoduché schéma celého procesu.



Obr. 17 - Schéma procesu CI
Zdroj: vlastní zpracování

GitHub

GitHub je v tuto chvíli repozitářem pro zdrojový kód projektu *bikeo.cz*. Je integrován jako plugin v Jenkinsu stejně jako např. Maven, tudíž bude velice snadné pracovat s repozitářem, stahovat a sestavovat kód.

Docker Hub

S podporou Jenkins dochází k automatickému nahrání sestavených obrazů do veřejných repozitářů Docker Hub pro pozdější možné sdílení a využití. Těmito repozitáři jsou *sipekmichal/bikeoapp* a *sipekmichal/bikeomysqldb*.

Jenkins

Jenkins je open-source automatizovaný server napsaný v jazyce Java, který napomáhá automatizaci procesu vývoje softwaru s trvalou integrací a usnadnění technických aspektů nepřetržitého doručení. Podporuje nástroje pro správu verzí, včetně nástrojů CVS, Subversion, Git, Mercurial, Docker a mnoho dalších. Umožňuje spouštění projektů Apache Ant, Apache Maven a libovolných shell skriptů atd.

Spuštění Jenkins kontejneru

Pro rychlou instalaci nástroje Jenkins je vytvořen Dockerfile (ukázka č. 78) vycházející z existujícího obrazu a následně je provedeno sestavení obrazu.

```
FROM jenkins/jenkins:lts
USER root
RUN apt-get update \
      && apt-get upgrade -y \
      && apt-get install -y sudo libltdl-dev \
      && rm -rf /var/lib/apt/lists/*
RUN echo "jenkins ALL=NOPASSWD: ALL" >> /etc/sudoers
USER jenkins

# sudo docker image build -t jenkins:lst .
```

Ukázka 78 - Jenkins Dockerfile

Spuštění kontejneru je realizováno následujícím příkazem, který vystavuje port webové aplikace (port 8080) poslouchající v kontejneru na port, který je dosažitelný z hostitelského počítače (port 5003). Navíc veškerá data, kterými Jenkins disponuje (pluginy a konfigurace), jsou perzistentně uložena na disku v umístění */opt/jenkins*. Pro zpřístupnění Dockeru uvnitř kontejneru je nutné vytvořit volume pro */var/run/docker.sock* (socket, na kterém naslouchá Docker démon pro komunikaci z kontejneru)a */usr/bin/docker* (rozhraní dává možnost spouštět Docker příkazy).

```
sudo docker run -d –t -p 5003:8080 \
--name=jenkins \
-v /var/run/docker.sock:/var/run/docker.sock \
-v /usr/bin/docker:/usr/bin/docker \
-v /opt/jenkins:/var/jenkins_home:z \
jenkins:lst
```

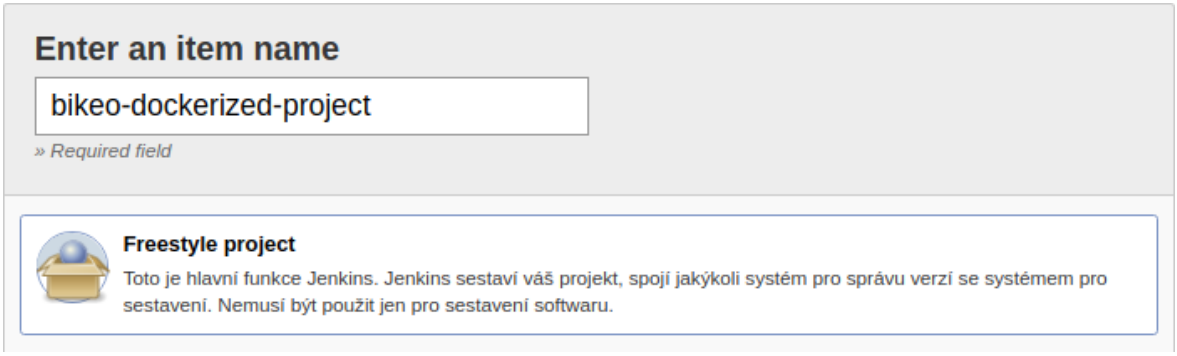
Ukázka 79 - Spuštění Jenkins v kontejneru

Po spuštění příkazu z předchozí ukázky č. 79 může trvat několik vteřin, než bude Jenkins k dispozici. K aplikaci je pak možné přistoupit prostřednictvím webového prohlížeče zadáním adresy *localhost:5003*. Zobrazí se průvodce nastavením a jako první se objeví obrazovka pro zadání administrátorského hesla, které je v tomto případě uloženo v souboru */opt/jenkins/secrets/initialAdminPassword.*

Na další obrazovce se vybírají zásuvné moduly, které budou nainstalovány. Lze ponechat výchozí nastavení *Install suggested plugins* a počkat několik minut na dokončení instalace. Další obrazovka vybízí k vytvoření nového administrátora a jeho přihlašovacím údajům. Nakonec je možné Jenkins plnohodnotně využívat.

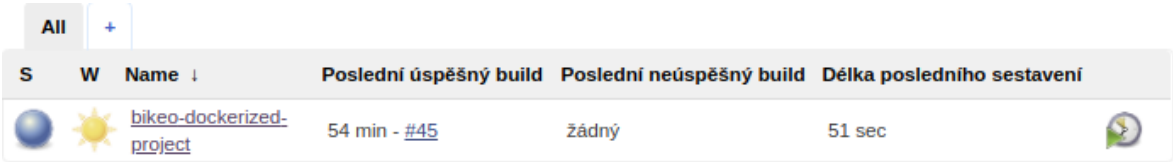
Vytvoření projektu

Poslední věc, kterou je potřeba udělat, je vytvoření projektu, který bude spouštěn v závislosti na odeslání kódu do GitHub repozitáře a bude provádět příslušné operace. Nejjednodušším možným způbem je vytvořit tzv. *Freestyle project*. Jak se takový projekt v grafickém rozhraní zakládá, znázorňuje obrázek č. 18.



Obr. 18 - Vytvoření projektu v Jenkins
Zdroj: vlastní zpracování

Po stisku tlačítka OK v dolní části obrazovky se vytvoří projekt a zobrazí se jeho konfigurační část, různá nastavení apod. Na úplném začátku je vhodné zaškrtnout volbu *Zahodit staré sestavení* a nastavit maximální počet sestavení, které se budou uchovávat na celková tři poslední. To zaručí, že se nebudou ukládat data všech sestavení, která by zbytečně zabírala místa na disku. Po vytvoření projektu je uživatel vrácen zpět na úvodní obrazovku, ve které je zobrazen celkový přehled projektů, viz. obrázek č. 19.

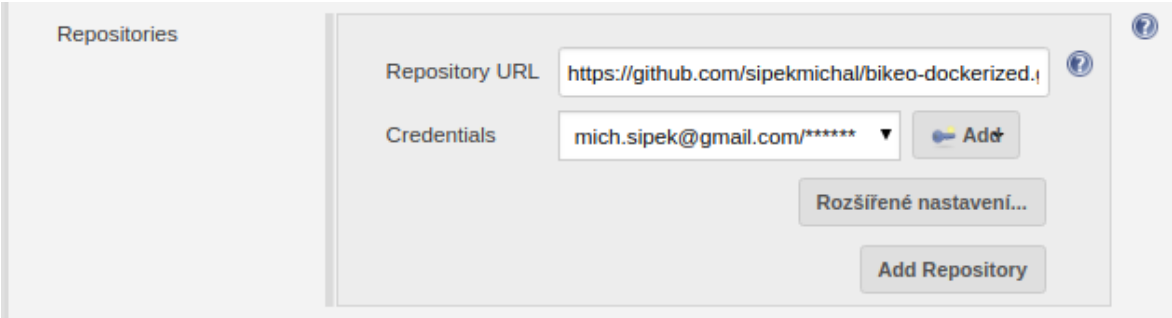


Obr. 19 - Přehled projektů
Zdroj: vlastní zpracování

Hned na to následuje napojení projektu na konkrétní GitHub repozitář projektu *bikeo-dockerized* vysvětleném v následujícím odstavci.

Napojení na GitHub repozitář

Napojení projektu na GitHub repozitář se provádí v záložce *Source Code Management*. Je zde na výběr několik verzovacích systémů. Jelikož projekt *bikeo-dockerized* používá Git a veřejný repozitář GitHub, je nutné tuto položku označit, nastavit URL k danému repozitáři tj. *https://github.com/sipekmichal/bikeo-dockerized.git* a přidat přístupové údaje k němu prostřednictvím tlačítka *Add*. Jediná vývojová větev je master, tudíž lze ponechat výchozí hodnotu **/master*. Obrázek č. 20 ukazuje, jak takové nastavení vypadá v uživatelském rozhraní nástroje Jenkins.

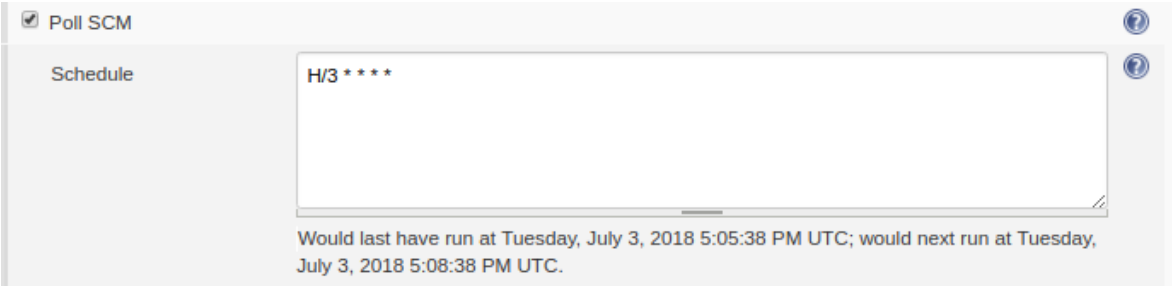


Obr. 20 - Napojení Jenkins na GitHub
Zdroj: vlastní zpracování

V tuto chvíli je Jenkins projekt napojen na GitHub repozitář a v případě jeho spuštění dochází ke stažení zdrojového kódu do pracovního prostoru, kde s ním lze pracovat.

Nastavení spouštěče sestavení

Celý proces průběžné integrace by neměl smysl, pokud by k němu nedocházelo automaticky. Je potřeba nastavit spouštěč, který následně spustí vytvořený Jenkins projekt. Vhodným typem spouštěče pro tento účel se jeví tzv. *Poll SCM,* který automaticky kontroluje, zda do repozitáře nastaveném v předchozí části nedošlo k nějaké změně zdrojového kódu. Perioda kontroly změn je nastavena na každé tři minuty díky speciální syntaxi cronu *H/3 * * * **, viz. obrázek č. 21.



Obr. 21 - Nastavení spouštěče sestavení v Jenkins
Zdroj: vlastní zpracování

Sestavení obrazů a nastartování kontejnerů s aplikacemi

Pro účely sestavení obrazů a nastartování kontejnerů s aplikacemi lze v záložce *Build* například použít *Build step* typu *Execute shell*, který slouží jako klasická příkazová řádka. Skript uvedený v tomto textovém poli je spuštěn nad aktuálním adresářem v pracovním prostoru. Je proto nutné vytvořit skript, který postupně provede:

1. sestavení obrazu *sipekmichal/bikeoapp* s tagem čísla (verze) buildu,
2. označení tohoto obrazu jako *sipekmichal/bikeoapp:latest*,
3. sestavení obrazu *sipekmichal/mysqldb:5.7* s tagem konkrétní verze databáze
4. zastavení a odstranění kontejnerů *bikeoapp* a *bikeodb*, pokud existují
5. nastartování nových kontejnerů *bikeoapp* a *bikeodb* sestavených v tomto kroku

Následující ukázka č. 80 demonstruje příklad takového skriptu, který je obsahem textového pole *Command* Jenkins projektu.

```
#BUILD IMAGE BIKEO APP
cd appserver
docker image build -t sipekmichal/bikeoapp:${BUILD_NUMBER} .
docker image tag sipekmichal/bikeoapp:${BUILD_NUMBER}
sipekmichal/bikeoapp:latest

#BUILD IMAGE MYSQL DATABASE
cd ../database
docker image build -t sipekmichal/mysqldb:5.7 .

#STOP AND REMOVE OLD CONTAINERS
docker container ps -q --filter "name=bikeoapp" | grep -q . && docker
container stop bikeoapp && docker container rm -fv bikeoapp
docker container ps -q --filter "name=bikeodb" | grep -q . && docker
container stop bikeodb && docker container rm -fv bikeodb

#RUN NEW CONTAINERS
docker container run -d -p 3306:3306 -e MYSQL_ROOT_PASSWORD=passw0rd -e
MYSQL_DATABASE=bikeo -e MYSQL_USER=admin -e MYSQL_PASSWORD=admin --name
bikeodb sipekmichal/mysqldb:5.7
docker container run -d -p 8080:8080 --link bikeodb:database --name
bikeoapp sipekmichal/bikeoapp:${BUILD_NUMBER}
```

Ukázka 80 - Build Execute shell Command 1

Odeslání do Docker Hub

Stejně tak jako v předešlém případě lze vytvořit další *Build step* typu *Execute shell* pro přihlášení a odeslání sestavených obrazů do úložiště Docker Hub. Ačkoliv by to bylo možné řešit v původním kroku, tj. textovém poli, je toto řešení jakýmsi oddělením pro dosažení lepší čitelnosti. Ukázka č. 81 zobrazuje shell skript, kde je v prvním kroku realizováno přihlášení příkazem *docker login* a dále odeslání jak obrazu *sipekmichal/bikeoapp* ve verzi odpovídající číslu sestavení, tak i obrazu: latest, který značí, že se jedná o poslední sestavenou verzi.

```
#DOCKER LOGIN AND PUSH TO DOCKER HUB REPOSITORY
docker login -u sipekmichal -p *****
docker image push sipekmichal/bikeoapp:${BUILD_NUMBER}
docker image push sipekmichal/bikeoapp:latest
docker image push sipekmichal/mysqldb:5.7
```

Ukázka 81 – Build Execute shell Command 2

Kontrola a shrnutí výsledku

Po nastavení všeho předchozího je možné projekt uložit a spustit. Spuštění je provedeno stisknutím tlačítka *Build Now* v levém panelu.

Pro rekapitulaci jsou zde shrnuté jednotlivé kroky, které jsou součástí průběžné integrace:

- vývojář upraví zdrojový kód projektu *bikeo-dockerized* a odešle ho do GitHub,
- každé tři minuty Jenkins kontroluje, zda nedošlo ke změně zdrojového kódu,
- pokud došlo ke změně, Jenkins spouští *bikeo-dockerized-project* a všechny jeho procesy,
- automaticky jsou sestaveny obrazy (včetně sestavení aplikace ze zdrojového kódu),
- pokud na lokálním počítači běží kontejnery s původní nasazenou aplikací a databází, dochází k jejich odstranění,
- jsou nastartovány nové kontejnery, které reflektují nové změny v kódu,
- sestavené obrazy jsou odeslány do veřejného repozitáře Docker Hub,
- aplikace je dostupná skrze webový prohlížeč a *http://localhost:8080/bikeo*,
- průběžně se dokola opakuje celý tento proces.

Výsledek procesu a jeho kompletní log lze zobrazit rozkliknutím položky *Console Output*, který je v ukázce č. 82.

```
Started by user Michal Šípek
Building in workspace /var/jenkins_home/workspace/bikeo-dockerized-
project
[WS-CLEANUP] Deleting project workspace...
[WS-CLEANUP] Done
Cloning the remote Git repository
Cloning repository https://github.com/sipekmichal/bikeo-dockerized.git
 > git init /var/jenkins_home/workspace/bikeo-dockerized-project #
using GIT_ASKPASS to set credentials
...
```

Ukázka 82 - Console Output

Dosaženým cílem je automatizovaný proces – vývojář má neustále k dispozici běžící aplikaci v Docker kontejnerech, které jsou automaticky přenasazované prostřednictvím nástroje Jenkins v závislosti na úpravě kódu aplikace. Během pár minut, co je kód odeslán do GitHub, má vývojář možnost otestovat aplikaci přímo ve webovém prohlížeči. Vždy pracuje s aktuální verzí, nemusí se o nic starat, pouze vyvíjet.

5 Shrnutí výsledků

Práce představila kontejnerovou platformu Docker z hlediska hlavních konceptů, umožnila čtenáři a začínajícímu uživateli dozvědět se o teoretických poznatcích. Praktické ukázky implementace Dockeru do vývojového procesu zachytily nejen osvědčené postupy, ale upozornily na obvyklé problémy, pomocí nichž je možné se vyhnout typickým chybám. Migrace webové aplikace v jazyce Java do architektury založené na kontejnerech pomocí nástroje Docker ukázala, jak se v takovém procesu postupuje a zachytila jednoduchost vývoje. Kromě toho bylo dosaženo rychlosti nasazení nové verze aplikace i samotného vývoje tím, že odpadla nutnost nastavení vývojového prostředí. Vzhledem k povaze Docker kontejnerů bylo docíleno schopnosti spuštění aplikace na libovolném prostředí a zajištění konzistence.

6 Závěry a doporučení

Práce rozebrala hlavní rozdíl mezi klasickou formou virtualizace a kontejnerizace pomocí nástroje Docker. Byl vytvořen srozumitelný a ucelený přehled těch nejdůležitějších konceptů a komponent vysvětlenými názornými ukázkami kódu prostupujícími celou prací. Dále kromě vysvětlení základních konceptů, používaných příkazů a operací s obrazy a kontejnery došlo k podrobnému popisu celého procesu kontejnerizace na vlastní Java aplikaci. Kromě toho byly představené další nástroje podporující management a nasazení, jako již zmíněný Docker Compose, Docker Swarm a Kubernetes pro orchestraci kontejnerů a další grafické nástroje pro práci s Dockerem.

Na základě poznatků a znalostí o Dockeru, byla provedena migrace existující webové aplikace do kontejnerů pro maximální možnou automatizaci vývojového procesu. Jednalo se o webovou aplikaci vyvinutou v jazyce Java za použití dalších technologií, jako např. Spring MVC, Maven, Tomcat, JSP apod., která komunikovala s databází MySQL. Aplikace zde použitá demonstrovala vytváření obrazů dle osvědčených postupů. Dosaženým cílem byl automatizovaný proces sestavení obrazů a přenasazení Docker kontejnerů prostřednictvím průběžné integrace v závislosti na úpravě kódu aplikace za pomoci nástrojů Docker, Docker Hub, Jenkins a GitHub.

Práce nepokrývá celou problematiku kontejnerové platformy Docker, ale vyzdvihuje pouze to nejpodstatnější, co by měl začínající uživatel vědět a co by při vývoji aplikací nejvíce využil. Díky Dockeru lze v oblasti vývoje, nasazení a distribuce aplikací spatřovat veliký pokrok. Dokazuje to i narůstající počet firem, které Docker přijímají jako obecný standard. Ačkoliv se jedná o poměrně mladou technologii, která se neustále vyvíjí kupředu, je otázkou, jak v dohledné době zareagují společnosti VMWare, Oracle apod. Pokud si uživatel pokládá otázku, jakým způsobem nastavit vývojové prostředí, jednoduše spouštět aplikace a nasazovat je na různá prostředí, aby mezi nimi nedocházelo k nekonzistenci a zároveň byl celý tento proces velmi rychlý, je seznámení se s nástrojem Docker vhodnou volbou.

Budoucí rozšíření by mělo pokračovat nasazením několika instancí kontejnerů do produkčního prostředí. Současná podoba je spíše zaměřena na práci s Dockerem v oblasti vývoje a nezahrnuje otázky, jakým způsobem řešit bezpečnost, logování, orchestraci, konfiguraci aplikace, monitoring apod. ve větším měřítku v produkci. Tento problém by byl řešitelný např. pomocí nástrojů Kubernetes, Elasticsearch, Kibana, Logstash, Chef atd.

7 Seznam použité literatury

[1] HRŮŠA, Ing. Petr. Virtualizace. In: SystemOnLine [online]. Praha: CCB, 2009 [cit. 2018-06-24]. Dostupné z: http://m.systemonline.cz/virtualizace/virtualizace.htm

[2] KUMAR, Krishan a Manish KURHEKAR. Economically Efficient Virtualization over Cloud Using Docker Containers. 2016 IEEE International Conference on Cloud Computing in Emerging Markets (CCEM) [online]. Bangalore, India: IEEE, 2016, 2016, , 95-100 [cit. 2018-06-26]. DOI: 10.1109/CCEM.2016.025. ISBN 978-1-5090-4573-0. Dostupné z: http://ieeexplore.ieee.org/document/7819678/

[3] POLEDNIK, Martin. Virtualizace a kontejnerizace. In: Fakulta informatiky Masarykovy univerzity [online]. Brno: Polednik, 2014 [cit. 2018-06-26]. Dostupné z: https://www.fi.muni.cz/~kas/pv090/referaty/2014-podzim/virt.html

[4] RAJ, Pethuru, Jeeva S. CHELLADHURAI a Vinod SINGH. Learning Docker. Birmingham: Packt Publishing, 2015. ISBN 9781784397937.

[5] RAJ, Pethuru. The key differentiators of Docker technology. In: IBM Cloud computing news [online]. New York: IBM, 2014, July 11, 2014 [cit. 2018-06-25]. Dostupné z: https://www.ibm.com/blogs/cloud-computing/2014/07/11/the-key-differentiators-of-docker-technology/

[6] BRYCH, David. Virtualizace serverů [online]. Plzeň, 2016 [cit. 2018-06-24]. Dostupné z: https://dspace5.zcu.cz/bitstream/11025/22931/1/Diplomova%20prace%20-%20David%20Brych.pdf. Diplomová práce. ZÁPADOČESKÁ UNIVERZITA V PLZNI. Vedoucí práce Ing. Petr Kašpar, Ph.D.

[7] TURNBULL, James. The Docker Book: Containerization is the new virtualization [online]. 2. New York: Lulu.com, 2017 [cit. 2018-06-24]. ISBN 9780988820234. Dostupné z: https://dockerbook.com

[8] BHARTIYA, Swapnil. Docker draws distinctions between enterprise and community editions. In: CIO [online]. London: IDG Communications, c2018, MAR 3, 2017 [cit. 2018-06-24]. Dostupné z: Docker draws distinctions between enterprise and community editions

[9] Docker Community Edition. Docker [online]. San Francisco: Docker, c2018 [cit. 2018-06-24]. Dostupné z: https://www.docker.com/community-edition

[10] Docker Documentation [online]. San Francisco: Docker, c2018 [cit. 2018-06-24]. Dostupné z: https://docs.docker.com/

[11] VÍŠEK, Dominik. Využití technologie Docker pro virtualizaci vývojového prostředí PHP / Python [online]. Pardubice, 2018 [cit. 2018-06-27]. Dostupné z: http://dspace.upce.cz/bitstream/handle/10195/70878/VisekD_TechnologieDocker_MB_2018.pdf. Bakalářská práce. Univerzita Pardubice. Vedoucí práce Ing. Monika Borkovcová, Ph.D.

[12] RICE. To commit or not to commit?. In: Hackernoon [online]. San Francisco: Medium Corporation, 2017, Mar 23, 2017 [cit. 2018-06-24]. Dostupné z: https://hackernoon.com/to-commit-or-not-to-commit-5ab72f9a466e

[13] MÓNICA, Diogo. Https://blog.docker.com/2015/08/content-trust-docker-1-8/. In: Docker Blog [online]. San Francisco: Docker, 2015, August 12, 2015 [cit. 2018-06-24]. Dostupné z: https://blog.docker.com/2015/08/content-trust-docker-1-8/

[14] NOBACK, Matthias. Docker build patterns. In: Matthias Noback - Blog: About software development [online]. Netherlands: Noback, 2017, Apr 4th 2017 [cit. 2018-06-24]. Dostupné z: https://matthiasnoback.nl/2017/04/docker-build-patterns/

[15] Dockerfile reference. In: Docker Documentation [online]. San Francisco: Docker, c2018 [cit. 2018-06-24]. Dostupné z: https://docs.docker.com/engine/reference/builder/

[16] CANE, Ben. Understanding Docker's CMD and ENTRYPOINT Instructions. In: Codeship [online]. Boston: Codeship, 2017, 2017-07-18 [cit. 2018-06-24]. Dostupné z: https://blog.codeship.com/understanding-dockers-cmd-and-entrypoint-instructions/

[17] Dockerfile Instructions and Syntax. In: Deis [online]. San Francisco: Deis, 2015, 17 Sep 2015 [cit. 2018-06-24]. Dostupné z: https://deis.com/blog/2015/dockerfile-instructions-syntax/

[18] VAN DER PLOEG, Nik. How To Set Up a Private Docker Registry on Ubuntu 14.04. In: DigitalOcean [online]. New York: DigitalOcean, c2018, October 15, 2014 [cit. 2018-06-24]. Dostupné z: https://www.digitalocean.com/community/tutorials/how-to-set-up-a-private-docker-registry-on-ubuntu-14-04

[19] KUBICA, Tomáš. Praktický úvod do Docker a kontejnerů (2) – propojování. In: Cloud Svět [online]. Praha: Kubica, 2015, 30.8.2015 [cit. 2018-06-24]. Dostupné z: http://www.cloudsvet.cz/?p=266

[20] How to Use Docker Compose. In: Linode: Cloud Hosting for Developers [online]. Philadelphia: Linode, 2017, December 28, 2017 [cit. 2018-06-24]. Dostupné z: https://www.linode.com/docs/applications/containers/how-to-use-docker-compose/

[21] GUMINSKI, Lukasz. Orchestrate Containers for Development with Docker Compose. In: Codeship [online]. Boston: Codeship, 2018, 2018-02-27 [cit. 2018-06-24]. Dostupné z: https://blog.codeship.com/orchestrate-containers-for-development-with-docker-compose/

[22] Overview of Docker Compose: Preserve volume data when containers are created. In: Docker Documentation [online]. San Francisco: Docker, c2018 [cit. 2018-06-24]. Dostupné z: https://docs.docker.com/compose/overview/#preserve-volume-data-when-containers-are-created

[23] Use Compose in production. In: Docker Documentation [online]. San Francisco: Docker, c2018 [cit. 2018-06-24]. Dostupné z: https://docs.docker.com/compose/production/

[24] PERY, Steve. What is Docker Swarm?: A discussion on container orchestration & potential solutions. In: The developerWorks Blog [online]. New York: IBM, 2017, September 14, 2017 [cit. 2018-06-24]. Dostupné z: https://developer.ibm.com/dwblog/2017/what-is-docker-swarm-cluster-engines/

[25] Kubernetes vs Docker Swarm. In: PLATFORM9 [online]. Sunnyvale: Sunnyvale, CA 94086, 2017, June 22, 2017 [cit. 2018-08-06]. Dostupné z: https://platform9.com/blog/kubernetes-docker-swarm-compared/

[26] HIGHTOWER, Kelsey, Brendan BURNS a Joe BEDA. Kubernetes: Up and Running: Dive Into the Future of Infrastructure. Kalifornie, USA: O'Reilly Media, 2017. ISBN 1491936029, 9781491936023.

[27] Under the Hood: An Introduction to Kubernetes Architecture: A 101 on Kubernetes components and how they all work together. In: Kublr: Kubernetes for the enterprise [online]. Washington: Kublr Team, 2017, September 11, 2017 [cit. 2018-08-03]. Dostupné z: https://kublr.com/blog/under-the-hood-an-introduction-to-kubernetes-architecture/

[28] Viewing Pods and Nodes. In: Kubernetes [online]. Kubernetes Authors, 2018 [cit. 2018-08-03]. Dostupné z: https://kubernetes.io/docs/tutorials/kubernetes-basics/explore/explore-intro/

[29] JETHVA, Hitesh. How to Install and Configure Docker Swarm on Ubuntu. In: Dataquest: Data science, data analysis, and data engineering tutorials and articles. [online]. San Francisco, 2018, 15 FEBRUARY 2018 [cit. 2018-08-11]. Dostupné z: https://www.dataquest.io/blog/install-and-configure-docker-swarm-on-ubuntu/

[30] A brief comparison of GUI Docker tools. In: Medium [online]. San Francisco: Medium Corporation, 2018, Mar 21 2018 [cit. 2018-06-24]. Dostupné z: https://medium.com/@dockstation/a-brief-comparison-of-gui-docker-tools-46bd6a24ae31

[31] WARD, Chris. A brief comparison of GUI Docker tools. In: DZone: DevOps Zone [online]. New York City: Ward, 2017, Jul, 19, 17 [cit. 2018-06-24]. Dostupné z: https://dzone.com/articles/a-comparison-of-docker-guis

[32] Portainer [online]. Auckland: Portainer.io, c2018 [cit. 2018-06-24]. Dostupné z: https://portainer.io

[33] Kitematic user guide. In: Docker Documentation [online]. San Francisco: Docker, c2018 [cit. 2018-06-24]. Dostupné z: https://docs.docker.com/kitematic/userguide/

[34] Rancher Makes It Easy. Your Enterprise Kubernetes Platform [online]. California: RANCHER LABS, c2014-2018 [cit. 2018-06-24]. Dostupné z: https://rancher.com/what-is-rancher/overview/

[35] NEWMAN, Andre. Top 5 Docker logging methods to fit your container deployment strategy. In: SolarWinds Loggly [online]. San Francisco: Loggly, 2015, 14 Dec 2015 [cit. 2018-06-24]. Dostupné z: https://www.loggly.com/blog/top-5-docker-logging-methods-to-fit-your-container-deployment-strategy/

[36] HROMIS, Andrew. Docker Reference Architecture: Docker Logging Design and Best Practices. In: SolarWinds Loggly [online]. San Francisco: Docker, c2018 [cit. 2018-06-24]. Dostupné z: https://www.loggly.com/blog/top-5-docker-logging-methods-to-fit-your-container-deployment-strategy/

[37] CASEY, Siobhan. How do I enable the remote API for dockerd. In: *Docker Success Center: The Docker Enterprise Customer Portal* [online]. San Francisco: Docker, c2018 [cit. 2018-07-13]. Dostupné z: https://success.docker.com/article/how-do-i-enable-the-remote-api-for-dockerd

[38] GUPTA, Arun. Monitoring Docker Containers – docker stats, cAdvisor, Universal Control Plane. In: *Codeship* [online]. Boston: Codeship, 2017, 2017-05-05 [cit. 2018-07-13]. Dostupné z: https://blog.codeship.com/monitoring-docker-containers/

[39] SISSONS, Gord. Comparing 7 Monitoring Options for Docker. In: *Rancher* [online]. California: RANCHER LABS, c2014-2018, APRIL 12, 2015 [cit. 2018-07-13]. Dostupné z: https://rancher.com/comparing-monitoring-options-for-docker-deployments/

[40] GUPTA, Arun. Docker Container Anti Patterns. In: Miles to go 4.0 [online]. Palo Alto: Arun, c2018, October 30, 2016 [cit. 2018-06-24]. Dostupné z: http://blog.arungupta.me/docker-container-anti-patterns/

[41] DASQUE, Erik. A Tale of Three Docker Anti-Patterns. In: Constant Contact Tech Blog [online]. Massachusetts: Constant Contact, c1996-2018, JANUARY 11, 201 [cit. 2018-06-24]. Dostupné z: http://techblog.constantcontact.com/devops/a-tale-of-three-docker-anti-patterns/

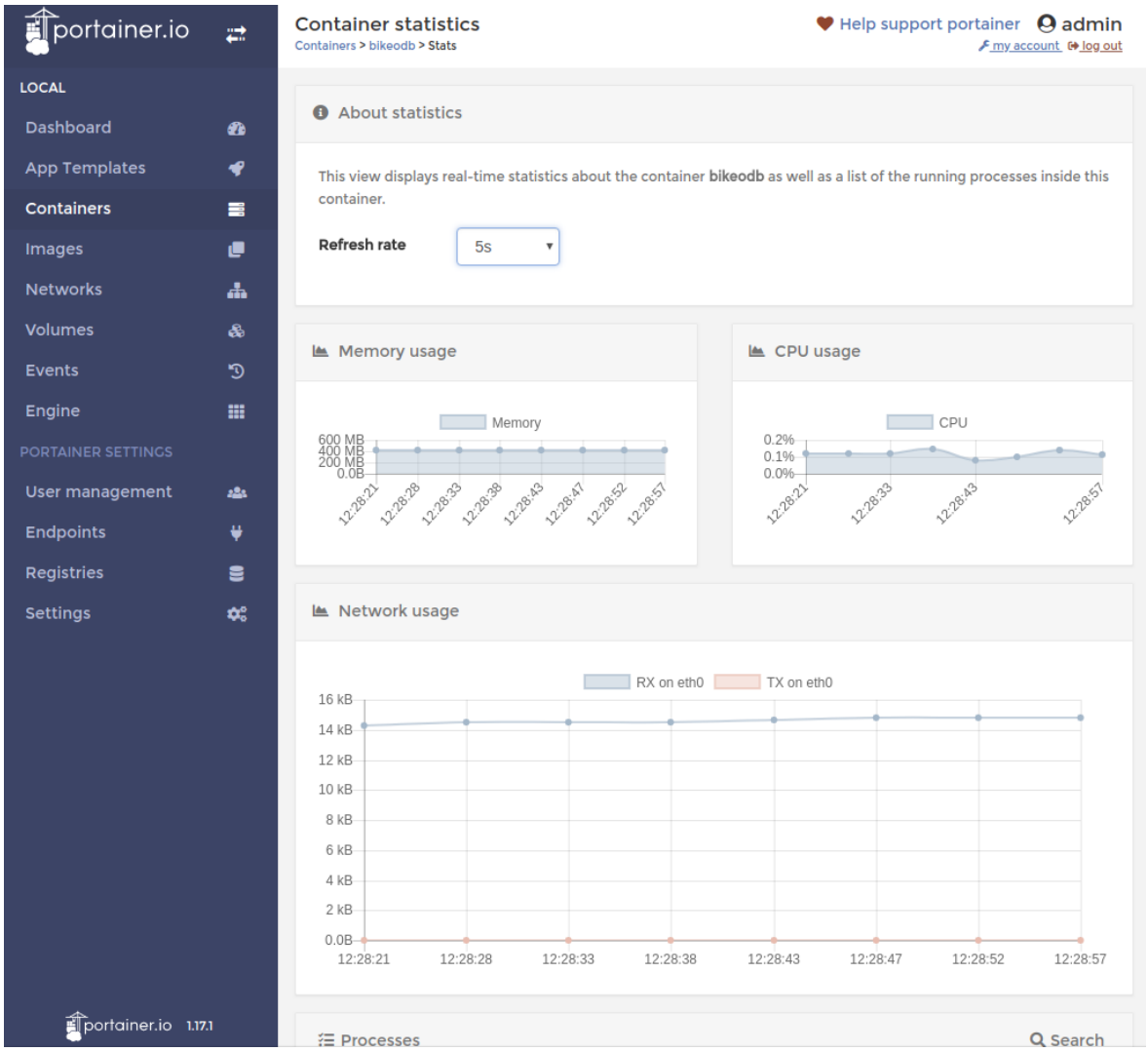
[42] Best practices for writing Dockerfiles. In: Docker Documentation [online]. San Francisco: Docker, c2018 [cit. 2018-06-24]. Dostupné z: https://docs.docker.com/develop/develop-images/dockerfile_best-practices/

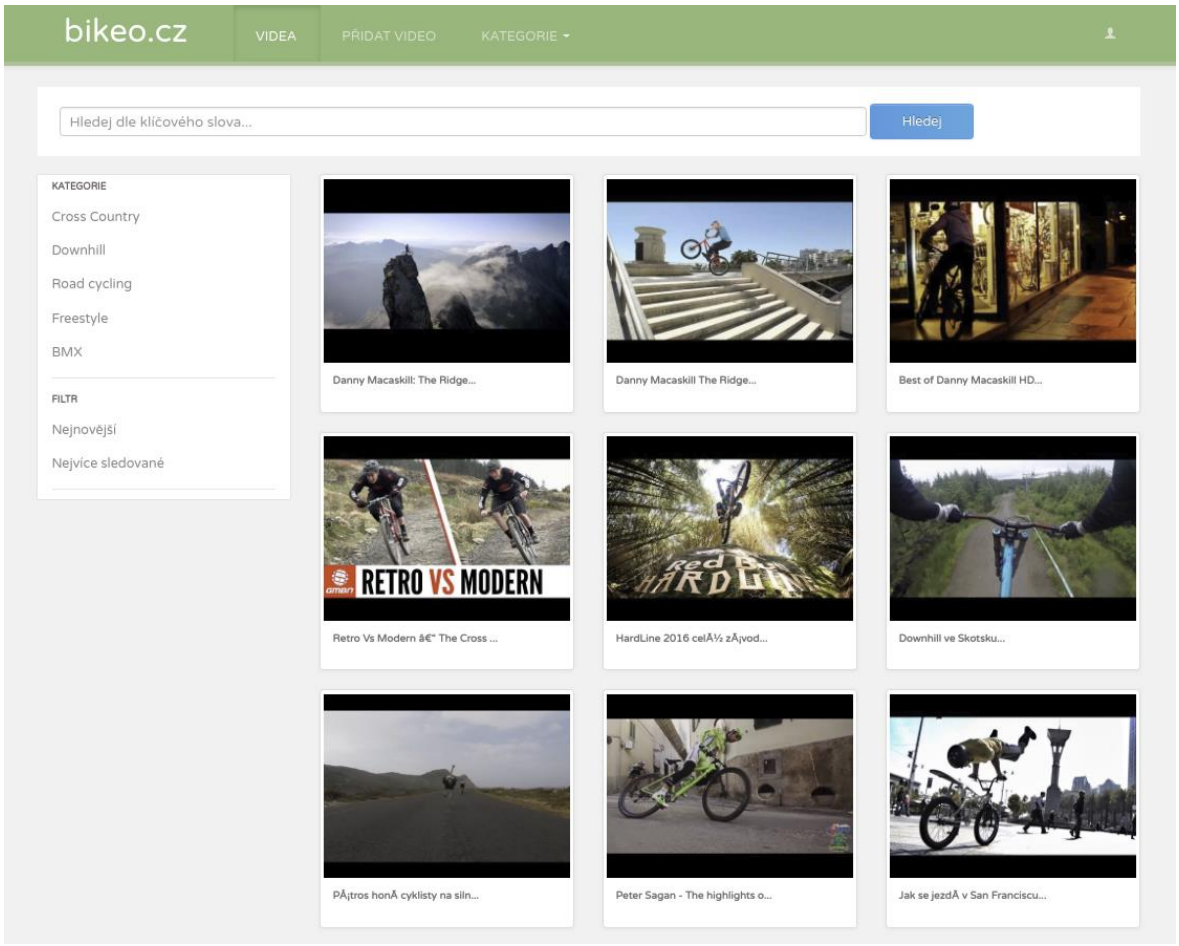
8 Přílohy

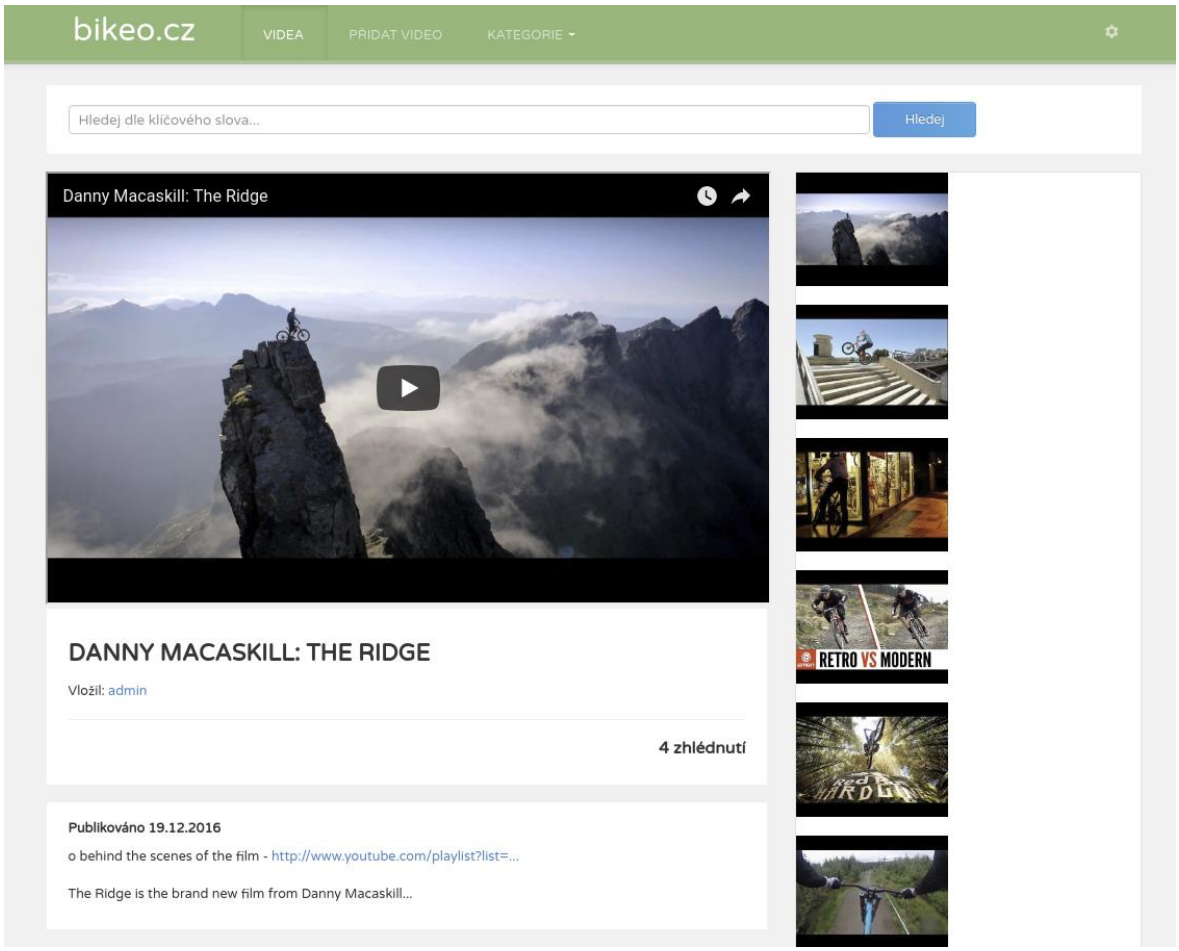
1) CD s ukázkami *custom-java-app* a s aplikací *bikeo* před a *bikeo-dockerized* po kontejnerizaci pomocí nástroje Docker včetně souborů Dockerfile apod.
2) Stránka se statistikami kontejneru *bikeodb* v nástroji Portainer
3) Hlavní stránka aplikace bikeo
4) Stránka aplikace bikeo zobrazující detail videa
5) Stránka aplikace bikeo zobrazující administrační část
6) Konfigurační obrazovka nástroje Jenkins projektu *bikeo-dockerized-project*
7) Stránka se zobrazeným výstupem (Console Output) sestavování Jenkins projektu *bikeo-dockerized-project*

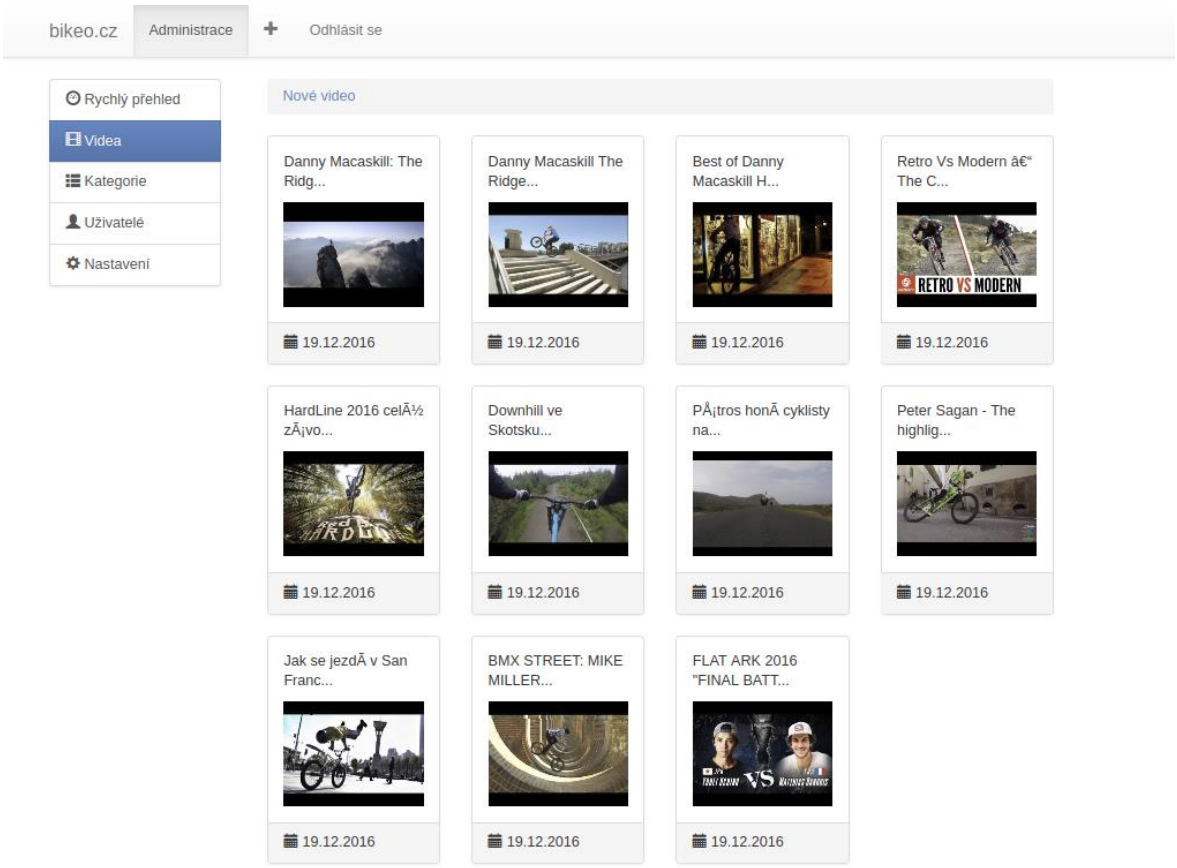
Adresářová struktura přiloženého CD

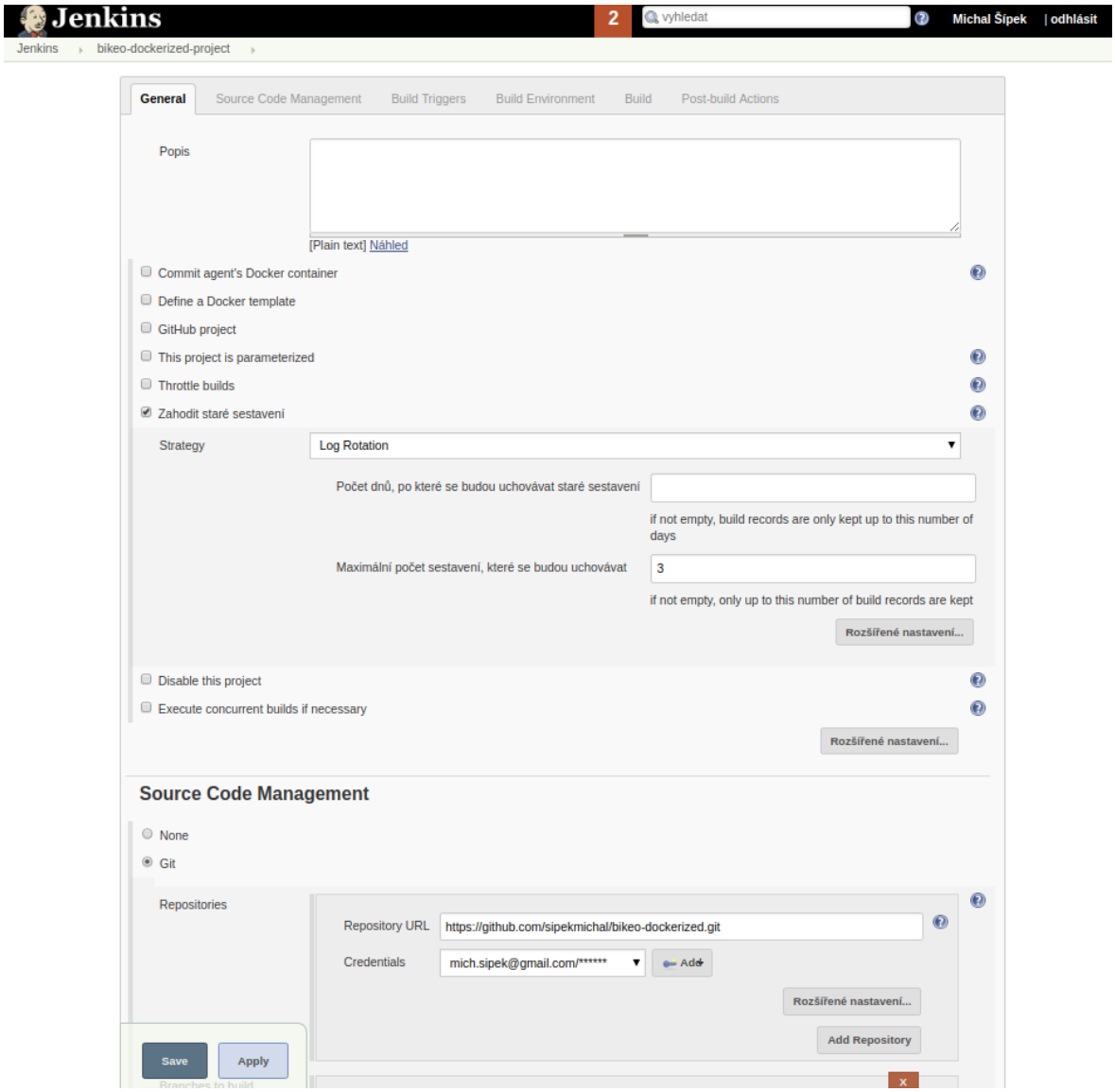
```
/
├── repozitare
│   ├── bikeo
│   │   ├── pom.xml
│   │   ├── README.md
│   │   ├── schema.sql
│   │   └── src
│   ├── bikeo-dockerized
│   │   ├── appserver
│   │   │   ├── Dockerfile
│   │   │   └── tomcat
│   │   │       ├── run.sh
│   │   │       └── tomcat-users.xml
│   │   ├── bikeo.iml
│   │   ├── database
│   │   │   ├── docker-entrypoint-initdb.d
│   │   │   │   └── initialize_db.sql
│   │   │   └── Dockerfile
│   │   ├── docker-compose.yml
│   │   └── README.md
│   └── custom.java.app.helloworld
│       ├── HelloWorld
│       │   └── Main.java
│       ├── Manifest.txt
│       └── README.rst
└── ukazky
    ├── custom-java-app
    │   ├── dockerfile
    │   │   └── Dockerfile
    │   ├── Dockerfile
    │   ├── HelloWorld.jar
    │   └── init.sh
    ├── custom-java-app-navrhove-vzory
    │   ├── app
    │   │   ├── HelloWorld.jar
    │   ├── build.sh
    │   ├── Dockerfile
    │   ├── Dockerfile.build
    │   ├── HelloWorld.jar
    │   ├── Manifest.txt
    │   ├── multistage
    │   │   └── Dockerfile
    │   └── README.rst
    └── jenkins-dockerfile
        └── Dockerfile
```

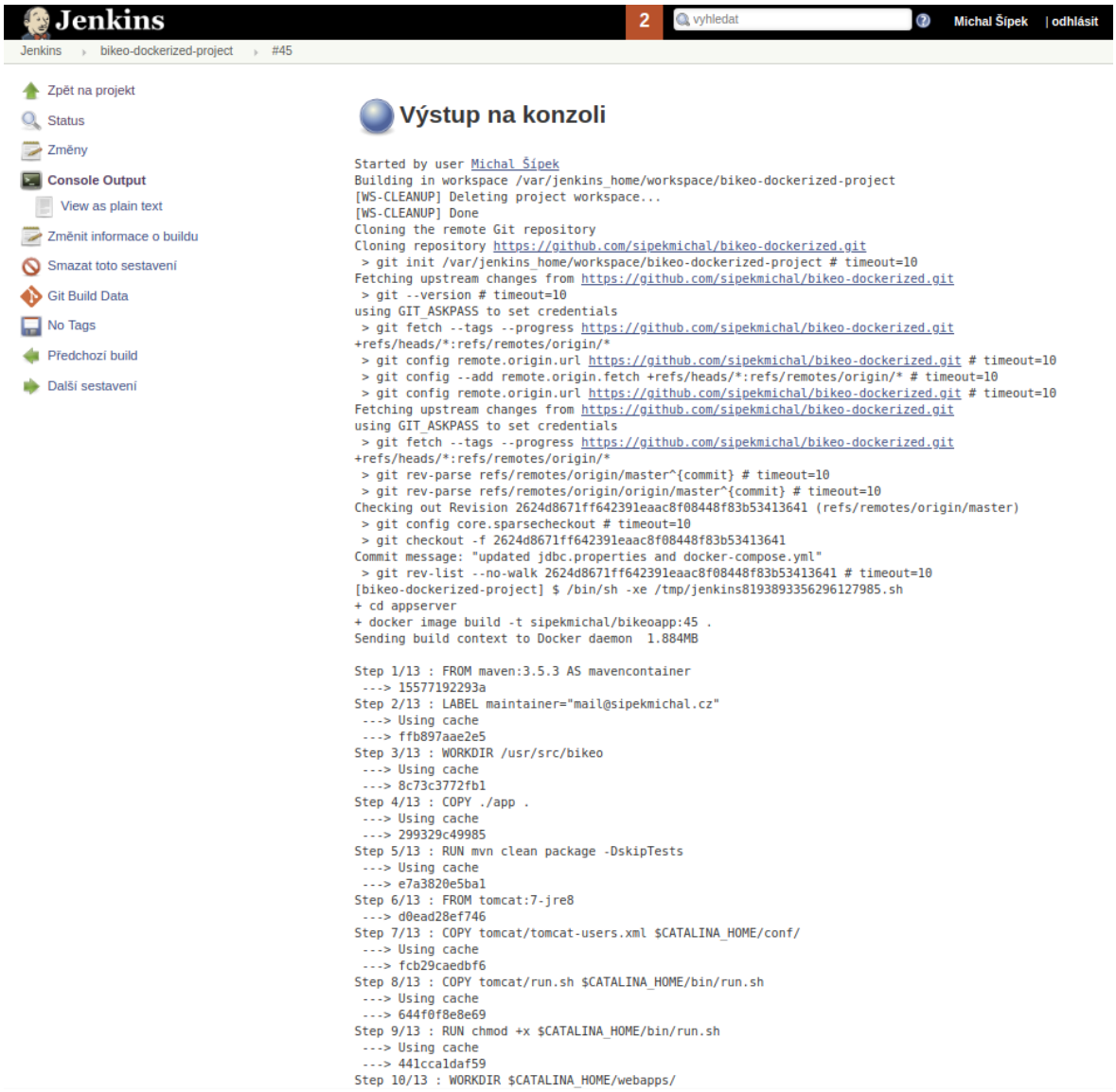

portainer.io
LOCAL
Dashboard
App Templates
Containers
Images
Networks
Volumes
Events
Engine
PORTAINER SETTINGS
User management
Endpoints
Registries
Settings
portainer.io 1.17.1
Container statistics
Containers > bikeodb > Stats
Help support portainer
admin
my account
log out
About statistics
This view displays real-time statistics about the container bikeodb as well as a list of the running processes inside this container.
Refresh rate
5s
Memory usage
Memory
600 MB
400 MB
200 MB
0.0B
12:28:21
12:28:28
12:28:33
12:28:38
12:28:43
12:28:47
12:28:52
12:28:57
CPU usage
CPU
0.2%
0.1%
0.0%
12:28:21
12:28:33
12:28:43
12:28:57
Network usage
RX on eth0
TX on eth0
16 kB
14 kB
12 kB
10 kB
8 kB
6 kB
4 kB
2 kB
0.0B
12:28:21
12:28:28
12:28:33
12:28:38
12:28:43
12:28:47
12:28:52
12:28:57
Processes
Search


bikeo.cz
VIDEA
PŘIDAT VIDEO
KATEGORIE
Hledej dle klíčového slova...
Hledej
KATEGORIE
Cross Country
Downhill
Road cycling
Freestyle
BMX
FILTR
Nejnovější
Nejvíce sledované
Danny Macaskill: The Ridge...
Danny Macaskill The Ridge...
Best of Danny Macaskill HD...
RETRO VS MODERN
Retro Vs Modern â€" The Cross ...
HardLine 2016 celÃ½ zÃ¡vod...
Downhill ve Skotsku...
PÃ¡tros honÃ cyklisty na siln...
Peter Sagan - The highlights o...
Jak se jezdÃ v San Franciscu...


bikeo.cz
VIDEA
PŘIDAT VIDEO
KATEGORIE
Hledej dle klíčového slova...
Hledej
Danny Macaskill: The Ridge
DANNY MACASKILL: THE RIDGE
Vložil: admin
4 zhlédnutí
Publikováno 19.12.2016
o behind the scenes of the film - http://www.youtube.com/playlist?list=...
The Ridge is the brand new film from Danny Macaskill...
RETRO VS MODERN


bikeo.cz
Administrace
Odhlásit se
Rychlý přehled
Videa
Kategorie
Uživatelé
Nastavení
Nové video
Danny Macaskill: The Ridg...
19.12.2016
Danny Macaskill The Ridge...
19.12.2016
Best of Danny Macaskill H...
19.12.2016
Retro Vs Modern â€“ The C...
RETRO VS MODERN
19.12.2016
HardLine 2016 celÃ½ zÃ¡vo...
19.12.2016
Downhill ve Skotsku...
19.12.2016
PÅ¡tros honÃ cyklisty na...
19.12.2016
Peter Sagan - The highlig...
19.12.2016
Jak se jezdÃ v San Franc...
19.12.2016
BMX STREET: MIKE MILLER...
19.12.2016
FLAT ARK 2016 "FINAL BATT...
VS
19.12.2016


Jenkins
2
vyhledat
Michal Šípek
| odhlásit
Jenkins
bikeo-dockerized-project
General
Source Code Management
Build Triggers
Build Environment
Build
Post-build Actions
Popis
[Plain text] Náhled
Commit agent's Docker container
Define a Docker template
GitHub project
This project is parameterized
Throttle builds
Zahodit staré sestavení
Strategy
Log Rotation
Počet dnů, po které se budou uchovávat staré sestavení
if not empty, build records are only kept up to this number of days
Maximální počet sestavení, které se budou uchovávat
3
if not empty, only up to this number of build records are kept
Rozšířené nastavení...
Disable this project
Execute concurrent builds if necessary
Rozšířené nastavení...
Source Code Management
None
Git
Repositories
Repository URL
https://github.com/sipekmichal/bikeo-dockerized.git
Credentials
mich.sipek@gmail.com/******
Add
Rozšířené nastavení...
Add Repository
Save
Apply


Jenkins
2
vyhledat
Michal Šípek
| odhlásit
Jenkins › bikeo-dockerized-project › #45
Zpět na projekt
Status
Změny
Console Output
View as plain text
Změnit informace o buildu
Smazat toto sestavení
Git Build Data
No Tags
Předchozí build
Další sestavení
Výstup na konzoli
Started by user Michal Šípek
Building in workspace /var/jenkins_home/workspace/bikeo-dockerized-project
[WS-CLEANUP] Deleting project workspace...
[WS-CLEANUP] Done
Cloning the remote Git repository
Cloning repository https://github.com/sipekmichal/bikeo-dockerized.git
 > git init /var/jenkins_home/workspace/bikeo-dockerized-project # timeout=10
Fetching upstream changes from https://github.com/sipekmichal/bikeo-dockerized.git
 > git --version # timeout=10
using GIT_ASKPASS to set credentials
 > git fetch --tags --progress https://github.com/sipekmichal/bikeo-dockerized.git
+refs/heads/*:refs/remotes/origin/*
 > git config remote.origin.url https://github.com/sipekmichal/bikeo-dockerized.git # timeout=10
 > git config --add remote.origin.fetch +refs/heads/*:refs/remotes/origin/* # timeout=10
 > git config remote.origin.url https://github.com/sipekmichal/bikeo-dockerized.git # timeout=10
Fetching upstream changes from https://github.com/sipekmichal/bikeo-dockerized.git
using GIT_ASKPASS to set credentials
 > git fetch --tags --progress https://github.com/sipekmichal/bikeo-dockerized.git
+refs/heads/*:refs/remotes/origin/*
 > git rev-parse refs/remotes/origin/master^{commit} # timeout=10
 > git rev-parse refs/remotes/origin/origin/master^{commit} # timeout=10
Checking out Revision 2624d8671ff642391eaac8f08448f83b53413641 (refs/remotes/origin/master)
 > git config core.sparsecheckout # timeout=10
 > git checkout -f 2624d8671ff642391eaac8f08448f83b53413641
Commit message: "updated jdbc.properties and docker-compose.yml"
 > git rev-list --no-walk 2624d8671ff642391eaac8f08448f83b53413641 # timeout=10
[bikeo-dockerized-project] $ /bin/sh -xe /tmp/jenkins8193893356296127985.sh
+ cd appserver
+ docker image build -t sipekmichal/bikeoapp:45 .
Sending build context to Docker daemon 1.884MB
Step 1/13 : FROM maven:3.5.3 AS mavencontainer
 ---> 15577192293a
Step 2/13 : LABEL maintainer="mail@sipekmichal.cz"
 ---> Using cache
 ---> ffb897aae2e5
Step 3/13 : WORKDIR /usr/src/bikeo
 ---> Using cache
 ---> 8c73c3772fb1
Step 4/13 : COPY ./app .
 ---> Using cache
 ---> 299329c49985
Step 5/13 : RUN mvn clean package -DskipTests
 ---> Using cache
 ---> e7a3820e5ba1
Step 6/13 : FROM tomcat:7-jre8
 ---> d0ead28ef746
Step 7/13 : COPY tomcat/tomcat-users.xml $CATALINA_HOME/conf/
 ---> Using cache
 ---> fcb29caedbf6
Step 8/13 : COPY tomcat/run.sh $CATALINA_HOME/bin/run.sh
 ---> Using cache
 ---> 644f0f8e8e69
Step 9/13 : RUN chmod +x $CATALINA_HOME/bin/run.sh
 ---> Using cache
 ---> 441cca1daf59
Step 10/13 : WORKDIR $CATALINA_HOME/webapps/

Univerzita Hradec Králové
Fakulta informatiky a managementu
Akademický rok: 2017/2018

Studijní program: Aplikovaná informatika
Forma: Kombinovaná
Obor/komb.: Aplikovaná informatika (ai2-k)

Podklad pro zadání DIPLOMOVÉ práce studenta

PŘEDKLÁDÁ:	ADRESA	OSOBNÍ ČÍSLO
Bc. Šípek Michal	U Hřiště 151, Cerekvice nad Loučnou	I1500720

TÉMA ČESKY:

Docker kontejnerizace v procesech vývoje SW aplikací

TÉMA ANGLICKY:

Docker containers and their usage in software development

VEDOUCÍ PRÁCE:

Ing. Pavel Kříž, Ph.D. - KIKM

ZÁSADY PRO VYPRACOVÁNÍ:

Cíl: Cílem je popsat problematiku Dockeru a paradigma moderních cloudových řešení postavených na kontejnerech atp. Na příkladech ukázat nasazení Dockeru při vývoji aplikací, upozornit na typické problémy, ukázat správné přístupy apod. Na příkladech ukázat další nástroje související s Dockerem, podporující nasazení a management ve větším měřítku (škálování).

1. Úvod
2. Paradigma moderních cloudových řešení
3. "Virtualizace" pomocí Docker
4. Implementace Docker do procesu vývoje aplikací
5. Nástroje podporující nasazení a management
6. Závěr
7. Literatura

SEZNAM DOPORUČENÉ LITERATURY:

1. JAMES TURNBULL: The Docker Book
2. https://docs.docker.com/

Podpis studenta: ..... **Datum:** 11.10.17

Podpis vedoucího práce: ..... **Datum:** 11.10.17