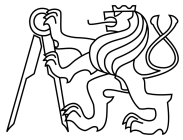
ČESKÉ VYSOKÉ UČENÍ TECHNICKÉ V PRAZE
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ



ZADÁNÍ DIPLOMOVÉ PRÁCE

Název:	Systém pro multikanálovou komunikaci se zákazníkem v reálném čase
Student:	Bc. Václav Dohnal
Vedoucí:	Ing. Adam Šenk
Studijní program:	Informatika
Studijní obor:	Webové a softwarové inženýrství
Katedra:	Katedra softwarového inženýrství
Platnost zadání:	Do konce letního semestru 2016/17

Pokyny pro vypracování

Projekt "The Story of Creation" se zaměřuje na podniky zabývající se zakázkovou výrobou a má za cíl stát se minimalistickým CRM systémem. Cílem této práce je vytvořit systém, který bude umožňovat podniku komunikovat se zákazníkem pomocí více kanálů, zejména pomocí webového rozhraní a pomocí e-mailů. Zákazníkům by pak měl umožňovat i zpětnou komunikaci s firmou. Ke splnění požadavků je nutná důkladná analýza podnikových procesů. Proveďte následující kroky:

1. Analyzujte a namodelujte podnikové procesy drobné zakázkové výroby.
2. Na základě analýzy navrhněte systém, který bude umožňovat komunikaci se zákazníkem pomocí různých kanálů.
3. Zaměřte se také na možnost komunikace v reálném čase a na škálovatelnost.
4. Zvolte vhodné technologie a platformu, na které bude systém nasazen.
5. Systém implementujte a integrujte s ostatními moduly.
6. Implementaci a integraci otestujte.
7. Zhodnoťte vhodnost použitých technologií a postupů.

Seznam odborné literatury

Dodá vedoucí práce.

L.S.

Ing. Michal Valenta, Ph.D.
vedoucí katedry

prof. Ing. Pavel Tvrdík, CSc.
děkan

V Praze dne 22. prosince 2015

České vysoké učení technické v Praze
Fakulta informačních technologií
Katedra softwarového inženýrství

Diplomová práce

Systém pro multikanálovou komunikaci se zákazníkem v reálném čase

Bc. Václav Dohnal

Vedoucí práce: Ing. Adam Šenk

29. června 2016

Poděkování

Děkuji Ing. Adamu Šenkovi za vedení celého projektu a podporu při realizaci této práce. Dále děkuji Ing. Miroslavu Hrstkovi za konzultace a pomoc při návrhu systému. Děkuji také rodině a blízkým za podporu.

Prohlášení

Prohlašuji, že jsem předloženou práci vypracoval(a) samostatně a že jsem uvedl(a) veškeré použité informační zdroje v souladu s Metodickým pokynem o etické přípravě vysokoškolských závěrečných prací.

Beru na vědomí, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., autorského zákona, ve znění pozdějších předpisů. V souladu s ust. § 46 odst. 6 tohoto zákona tímto uděluji nevýhradní oprávnění (licenci) k užití této mojí práce, a to včetně všech počítačových programů, jež jsou její součástí či přílohou, a veškeré jejich dokumentace (dále souhrnně jen „Dílo“), a to všem osobám, které si přejí Dílo užít. Tyto osoby jsou oprávněny Dílo užít jakýmkoli způsobem, který nesnižuje hodnotu Díla, a za jakýmkoli účelem (včetně užití k výdělečným účelům). Toto oprávnění je časově, teritoriálně i množstevně neomezené. Každá osoba, která využije výše uvedenou licenci, se však zavazuje udělit ke každému dílu, které vznikne (byť jen zčásti) na základě Díla, úpravou Díla, spojením Díla s jiným dílem, zařazením Díla do díla souborného či zpracováním Díla (včetně překladu), licenci alespoň ve výše uvedeném rozsahu a zároveň zpřístupnit zdrojový kód takového díla alespoň srovnatelným způsobem a ve srovnatelném rozsahu, jako je zpřístupněn zdrojový kód Díla.

V Praze dne 29. června 2016 .....

Odkaz na tuto práci

Dohnal, Václav. *Systém pro multikanálovou komunikaci se zákazníkem v reálném čase.* Diplomová práce. Praha: České vysoké učení technické v Praze, Fakulta informačních technologií, 2016.

Abstrakt

Práce se zabývá analýzou, návrhem a implementací webové aplikace s důrazem na modulární kontejnerovou architekturu.

Klíčová slova SaaS, Docker, Node.js, Angular, webová aplikace

Abstract

This thesis deals with analysis, design and implementation of web application with an emphasis on modular container architecture.

Keywords SaaS, Docker, Node.js, Angular, web application

Obsah

Seznam obrázků

Seznam tabulek

Úvod

V posledních dvou dekádách nastal masivní rozvoj internetu a s ním související vývoj webových technologií. Ty se dostaly do stavu, ve kterém reálně konkurují a v mnohém snad i předčí tradiční technologie pro vývoj software. Dalším odvětvím, které zaznamenalo ohromný růst, jsou mobilní zařízení – ta dnes slouží jako další plnohodnotný kanál k webovým službám.

Není tedy překvapením, že současným trendem je vyvíjet většinu nových aplikací jako webové služby modelem SaaS. Na takové aplikace jsou kladeny stále vyšší požadavky v oblasti škálovatelnosti, správné architektury, bezpečnosti, široké podpoře různých zařízení. Narůstající nároky zaznamenává i infrastruktura, na které jsou aplikace nasazeny. Vývojáři chtějí po infrastruktuře rychlé a automatizované nasazení, jednoduchou správu a údržbu a ideálně možnost vše otestovat s reálnými závislostmi na svém počítači.

Projekt „The Story of Creation“ umožňuje drobným podnikatelům komunikovat se svými zákazníky přes web a pomocí mobilní aplikace. Celý systém je minimalistické CRM pro správu menších zakázek na míru. Předmětem této práce je analyzovat, navrhnout a vhodně implementovat aplikaci, která bude spolupracovat s dvěma již zmíněnými součástmi projektu – webem a mobilní aplikací. Účelem této nové aplikace je přenést většinu funkcionality mobilní aplikace do webové aplikace, ze které mohou uživatelé spravovat své účty.

Při návrhu a implementaci je kladen důraz na modulární a škálovatelnou architekturu s možností provozu v systému PaaS.

Kapitola 1

Uvedení do problematiky

Cílem této kapitoly je uvést čtenáře do problematiky projektu, seznámit jej s řešeným problémem a definovat cíle pro tuto práci. V následujícím textu jsou použity dvě základní role v projektu: Uživatel a Zákazník. **Uživatelem** je myšlen malý podnik či podnikatel, který nabízí zakázkovou výrobu svým **Zákazníkům**. Podrobný popis těchto rolí se nachází v sekci Uživatelské role na straně 5.

1.1 Představení projektu

Projekt „The Story of Creation“ slouží menším podnikům a podnikatelům, kteří se zabývají zejména zakázkovou výrobou. Častým požadavek zákazníků, kteří si objednávají zakázkovou výrobu, je mít přehled o stavu práce na zakázce. Komunikace mezi firmou či podnikatelem a zákazníkem však nemusí být vždy efektivní a to zejména ze dvou důvodů:

- Uživatel může komunikovat se zákazníky více kanály najednou – typicky e-mailem, po sociálních sítích apod. Zpětná dohledatelnost informací je pak omezena a zákazník postrádá „globální“ pohled na stav své zakázky. Tento problém je umocněn s více lidmi ve zpracovatelském týmu.

- Po dokončení zakázky nejsou materiály, které vznikly při její výrobě, dále použity na příklad k propagaci uživatele.

Projekt „The Story of Creation“ řeší tyto problémy minimalistickou CRM aplikací, která umožňuje:

- komunikovat mezi uživatelem a zákazníkem více kanály;

- automaticky generovat z materiálů vytvořených během zpracování zakázky (texty, fotografie, ...) prezentaci uživatele.

1.2 Cíle projektu

Základem projektu „The Story of Creation“ je již existující mobilní aplikace, pomocí které uživatelé dokumentují průběh na zakázce. Mobilní aplikace je ideální pro pořizování a okamžité sdílení multimediálních dat, avšak některé úkony (zejména pak pro psaní delších textů) je vhodnější aplikace běžící i mimo mobilní zařízení.

Cílem této práce je vytvořit webovou aplikaci, která bude sloužit jako alternativa k mobilní aplikaci. Mobilní aplikace a vznikající webová aplikace pak budou používat společné API pro přístup a vytváření dat. Webovou aplikaci je důležité navrhnout s ohledem na již existující systém, tj. webová aplikace musí používat již definovaná data, měla by nabídnout podobný styl ovládání jako aplikace mobilní atp. Dalším důležitým faktorem při návrhu a implementaci je vybrat vhodný systém a platformu pro nasazení aplikace, aplikace by měla dále používat takovou architekturu, která umožní škálovatelnost.

Kapitola 2

Analýza a návrh

Cílem této je kapitoly je analyzovat požadavky na aplikaci a navrhnout možná řešení. Mimo jiné se zaměříme na metodologii The Twelve-Factor App, kterou se budeme řídit při návrhu a implementaci aplikace. Návrh struktury aplikace v podobě kontejnerové architektury se nachází v následující kapitole Kontejnerová architektura na straně 15.

Pro potřeby dalšího textu budou všechny entity a uživatelské role psány s Velkým písmenem.

2.1 Uživatelské role

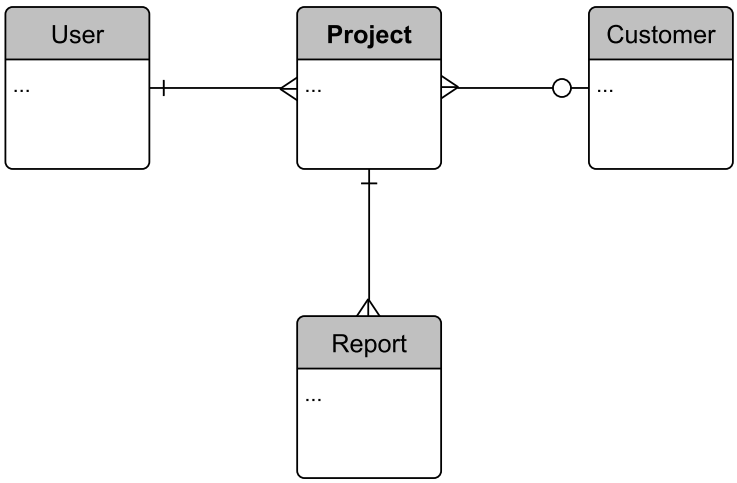
Pro potřeby dalšího textu je důležité rozlišit tři základní uživatelské role, které interagují s aplikací.

ID	Role	Význam
R1	**Uživatel**	Drobný podnikatel, který využívá aplikace pro komunikace se svými zákazníky o zakázkách pro ně.
R2	**Zákazník**	Zákazník Uživatele. Pro Zákazníka Uživatel zpracovává nějakou zakázku
R3	**Anonym**	Veřejný uživatel, který si např. prohlíží portfolio Uživatele a může se tak stát potencionálním Zákazníkem.

Tabulka 2.1: Uživatelské role v systému

2.2 Funkční požadavky

Funkcionalita aplikace částečně kopíruje funkcionalitu mobilní aplikace. Jádro aplikace tvoří vazba mezi Uživatelem `[User]` a jeho Projektem `[Project]` (nějaká zakázka, na které pracuje).



Obrázek 2.1: Zjednodušený ER diagram výběru nejdůležitějších entit v systému.

ID	Požadavek	Popis
F1	Přihlášení Uživatele	Uživatel se přihlásí svým účtem z mobilní aplikace.
F2	Správa Projektů	Uživatel spravuje své Projekty.
F3	Nastavení účtu	Uživatel spravuje nastavení svého účtu.
F4	Komunikace se Zákazníkem	Uživatel komunikuje v reálném čase se Zákazníkem.

Tabulka 2.2: Funkční požadavky

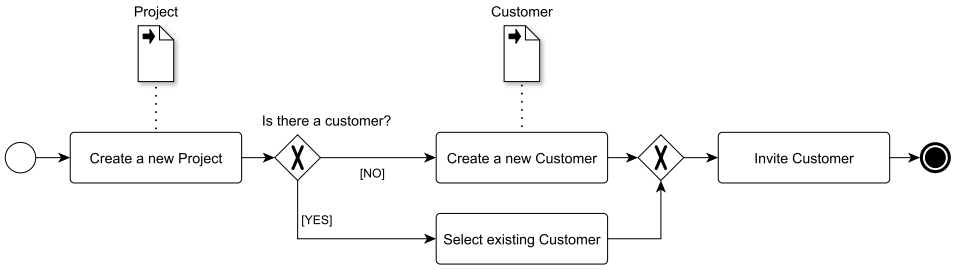
2.3 Nefunkční požadavky

ID	Požadavek	Popis
NF1	Škálovatelnost	Aplikace je škálovatelná.
NF2	Modularita	Aplikace není monolitická.
NF3	Testovatelnost	Aplikace je automaticky testovatelná.
NF4	Komunikace s API	Aplikace komunikuje s JSON REST API, a negeneruje žádná data pro vlastní potřebu.

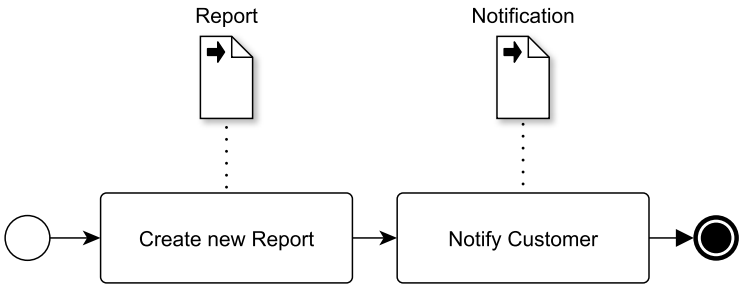
Tabulka 2.3: Nefunkční požadavky

2.4 Případy použití a podporované procesy

Celý životní cyklus použití aplikace začíná z pohledu Uživatele vytvořením nového Projektu a přidružení Zákazníka k tomuto projektu.



Obrázek 2.2: Základní podporovaný proces v projektu. Uživatel začíná pracovat na novém Projektu pro Zákazníka.



Obrázek 2.3: Proces generování Reportů k Projektu.

2.5 Metodologie The Twelve-Factor App

Metodologie The Twelve-Factor App[1] je určena především vývojářům, kteří vytvářejí aplikace běžící jako služba (SaaS), a tzv. „DevOps“ pracovníkům, jejichž role je nasazování (či spíše častěji užívaný anglický pojem *deployment*), provoz a monitoring aplikace.

Autorem metodologie je Adam Wiggins, mimo jiné spoluzakladatel populární PaaS Heroku [1].

Metodologie říká, jak vyvíjet, provozovat a udržovat moderní webovou aplikaci (dle metodologie je tato aplikace označovaná jako „**twelve-factor app**“), u které je kladen důraz zejména na: [2]

- Deklarativní přístup pro nastavení prostředí s cílem co nejrychleji začlenit nového vývojáře do týmu/projektu.
- Přenositelnost mezi operačními systémy.
- Schopnost běhu na moderních cloudových platformách a/nebo minimalizaci potřeby systémových administrátorů při správně vlastního serveru.
- Minimalizaci rozdílů mezi vývojovým a produkčním prostředí.
- Škálování bez potřeby měnit zásadním způsobem architekturu aplikace a/nebo použité nástroje.

[1] Volně přeložitelné z angličtiny jako „dvanáctifaktorová aplikace“, v této práci je však použit nepřeložený anglický originál.

Metodologie je rozdělena do dvanácti kapitol, z nichž se každá zabývá jinou částí při vývoji a provozu aplikace. Kompletní referenční příručka k této metodologii je dostupná ve formě otevřeného textu na http://12factor.net/.

1. Správa zdrojového kódu;
2. Závislosti aplikace;
3. Konfigurace;
4. Závislé služby;
5. Proces sestavení, nasazení a běhu aplikace;
6. Aplikace jako samostatný proces;
7. Vystavení síťových portů;
8. Škálovatelnost;
9. Spuštění a vypnutí aplikace;
10. Prostředí;
11. Logování;
12. Údržba.

Nejdůležitější relevantní části metodologie využité při návrhu a implementaci zde popsané aplikace jsou na následujících řádcích podrobně rozebrány. Mnohé doporučované principy a postupy jsou umožněny díky kontejnerové architektuře, která je rozebrána v samostatné kapitole Kontejnerová architektura na straně 15.

2.5.1 Externí závislosti aplikace

Aplikace by nikdy neměla spoléhat na existenci balíčku, software či knihovny nainstalované v systému, na kterém běží. A to převážně z toho důvodu, že nemůže být nikdy z podstaty věci garantováno, že takový balíček nebo knihovna budou skutečně přítomny. Nemluvě o rozdílech mezi různými verzemi takového balíčku.

Aplikace by měla vždy své externí závislosti specifikovat jasným manifestem – na příklad konfiguračním souborem pro správce externích závislostí. [3]

Aplikace, jež je předmětem této práce je izolována od okolního světa ve dvou úrovních:

1. **Systémové balíčky** a knihovny jsou nainstalovány v Docker kontejneru přesně tak, jak jsou specifikovány v konfiguračním souboru Dockerfile.

2. Další **externí závislosti** jsou specifikovány v souborech `bower.json` a `package.json`. První jmenovaný je využit balíčkovacím nástrojem Bower, který slouží pro správu závislostí na frontendu. Druhý jmenovaný využívá balíčkovací nástroj npm pro správu backendového kódu. Oba druhy závislostí jsou stejně jako v předchozím bodě instalovány do kontejneru.

2.5.2 Konfigurace aplikace

Metodologie říká, že v ideálním případě by se měla lokálně vyvíjená aplikace a aplikace nasazená v ostrém produkčním prostředí lišit pouze v konfiguraci. Veškerá konfigurace uložená napevno v kódu aplikace (typicky např. hesla) je tedy porušením tohoto principu, aplikace musí striktně oddělit kód a konfiguraci.

Metodologie dále říká, že je k tomuto účelu vhodné použít proměnné prostředí, ty jsou více známé pod anglickým termínem „environment variables“, „env vars“ nebo zkráceně „env“. [4]

Aplikace, jež je předmětem této práce, *env vars* používá:

- Direktiva `ENV` v Dockerfile nastavuje uvnitř kontejneru *env vars*, ty následně aplikace čte.
- Docker Compose při spojování kontejnerů (více v sekci Spojování kontejnerů na straně 39) neprivátní *env vars* automaticky sdílí mezi kontejnery.

2.5.3 Závislé služby

Závislou službou je myšlena nějaká služba, kterou aplikace vyžaduje a po síti s ní komunikuje – typicky se jedná na příklad o databázi.

Dle metodologie používá aplikace při vývoji i při nasazení identické závislé služby. [5]

Tohoto principu je dosaženo implicitně použitím kontejnerové architektury, více o tématu v sekci Spojování kontejnerů na straně 39.

2.5.4 Procesy a porty

Metodologie říká, že aplikace musí být spuštěna jako jeden nebo více procesů. Tyto procesy musejí být bezstavové a nesmějí mezi sebou přímo sdílet žádná data. Všechna data musejí být uložena v nějaké stavové závislé službě. Jako příklad uveďme uložení HTTP session:

- **Porušení principu:** Session je uložena v paměti procesu, který ji vytvořil. Aplikace tak předpokládá, že uživatel při dalším použití návštěvy použije stejný proces.

- **Správný přístup:** Session je uložena v nějaké závislé službě na příklad v *key-value cachovací* databázi Redis. Proces je tak bezstavový a uživatel může při dalším startu aplikace použít proces jiný (na příklad díky load balancingu).

Dále musí procesy aplikace správně reagovat na signály, které mohou od systému dostat (`SIGTERM`, `SIGKILL` ...). [6]

Druhým doporučení metodologie je, aby aplikace či závislá služba se svým okolím komunikovala výhradně přes síťové porty. [7]

Obou principů je rovněž dosaženo implicitně použitím kontejnerové architektury.

2.5.5 Prostředí

Dle metodologie:

- Je „twelve-factor“ aplikace připravena na continuous deployment tak, aby byla časová prodleva mezi vývojem a nasazením co nejmenší.
- Vývojář je součástí procesu nasazení.
- Rozdíly mezi vývojovým a reálným prostředím jsou nejmenší možné.

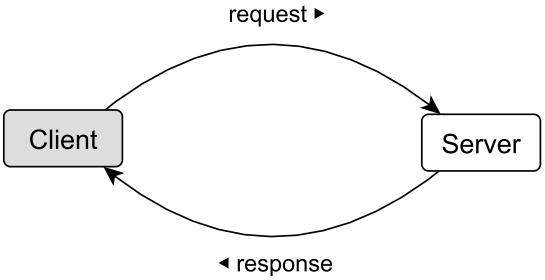
Poslední bod je v kontextu kontejnerové architektury důležitý. Vývojáři často používají při vývoji aplikace závislou službu, která je jednoduší na ovládání, správu a použití, zatímco v produkčním prostředí je použita jiná služba. Jako příklad uveďme databázi: vývojář při vývoji používá SQLite, zatímco v produkčním prostředí aplikace využívá PostgreSQL. Moderní frameworky často pracují na principu univerzálních adaptérů pro přístup k databázi/cache/..., a tedy po technické stránce je možné využívat různé služby v různých prostředích. Problém však může nastat v různých limitních prostředích (*edge case*). [8]

„Twelve-factor“ aplikace tedy vždy používá stejné závislé služby, při vývoji i reálném provozu. A právě na této myšlence staví kontejnerová architektura.

2.6 Protokol WebSocket a real-time komunikace

2.6.1 Úvod do protokolu Websocket

Hlavní myšlenka webu (HTTP protokolu) je od počátku postavena na principu, že klient (typicky webový prohlížeč) zašle *request* na vzdálený server. Server klientovi zašle data (*response*) a tím je celý proces komunikace ukončen.



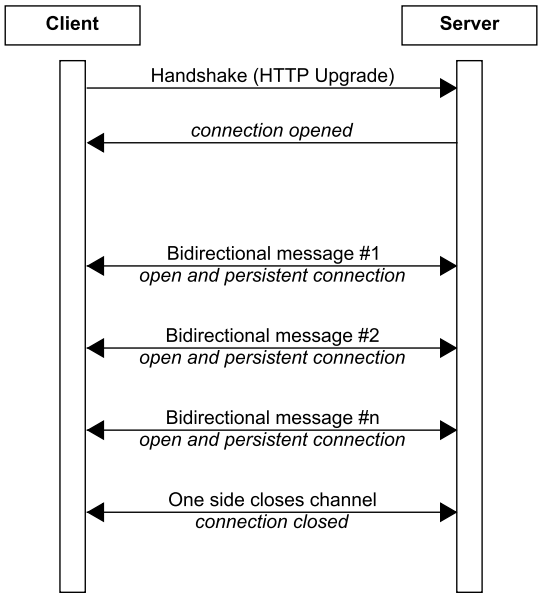
Obrázek 2.4: Cyklus HTTP požadavku. Klient zašle na server požadavek, server odpoví a celý cyklus je ukončen.

S rostoucími nároky na webové aplikace logicky vyvstal požadavek na real-time obousměrnou komunikaci mezi klientem a serverem. V roce 2005 se poprvé objevila technologie AJAX, která umožnila vývojářům iniciovat asynchronní požadavek od klienta k odeslání nebo získání jen části dat. Technologií AJAX bylo možné vytvořit aplikaci, která pracovala z pohledu uživatele „real-time“, avšak v jádru vše pracuje stále na principu HTTP request–respose a „real-time“ aplikace musela v konečném důsledku používat *polling* na server – tedy opakovaně zasílat *requesty* na server s dotazem, zda pro aplikaci nemá nová data. [9]

Definitivně byla všechna technická omezení překonána s představením protokolu WebSocket (http://websocket.org/) v roce 2011[2]. Tento protokol, stejně jako protokol HTTP, pracuje nad protokolem TCP. Umožňuje vytvořit mezi klientem a serverem jediné perzistentní full-duplexní spojení [10, 11] Podpora mezi webovými prohlížeči v červnu 2016 je velmi dobrá, všechny moderní prohlížeče protokol podporují – dle globálních statistik se jedná o přibližně 87,5 % všech zařízení. [12]

Kvalitně zpracovaná specifikace protokolu je dostupná na https://html.spec.whatwg.org/multipage/comms.html#network.

[2]Protokol Websocket je součástí standardu HTML5, který v sobě zahrnuje desítky dalších technologií.



Obrázek 2.5: Schéma komunikace pomocí protokolu Websocket. Mezi klientem a serverem probíhá komunikace na jediném full-duplexním perzistentním TCP spojení. [13]

2.6.2 WebSocket Handshake

Navázání WebSocket spojení mezi klientem a serverem probíhá pomocí tzv. WebSocket Handshake. Klient na server pošle standardní HTTP požadavek se speciální hlavičkou Upgrade, která serveru říká, že klient žádá o povýšení protokolu. [14, 13]

Ukázka kódu 2.1 První krok při WebSocket Handshake. HTTP požadavek od klienta k serveru informující, že další komunikace má probíhat protokolem WebSocket. [14]

```
1 GET /chat HTTP/1.1
2 Host: example.com:8000
3 Upgrade: websocket
4 Connection: Upgrade
```

Ukázka kódu 2.2 Druhý a poslední krok při WebSocket Handshake. HTTP odpověď od serveru ke klientovi. Server sděluje, že protokol podporuje a přepíná na něj. [14]

```
1 GET /chat HTTP/1.1
2 Host: example.com:8000
3 Upgrade: websocket
4 Connection: Upgrade
```

Po provedení Handshake probíhá komunikace přes URL s prefixem `ws://...` (ekvivalent `http://...`) resp. `wss://...` pro zabezpečenou verzi protokolu (ekvivalent `https://...`).

Kapitola 3

Kontejnerová architektura

Cílem této kapitoly je představení modulární kontejnerové architektury pro vývoj, testování, škálování a nasazení aplikací. Protože kontejnerová architektura v mnohém nabízí podobnou funkcionalitu jako tradičnější virtualizace, srovnáme tyto dva přístupy. Detailně se seznámíme s kontejnerovým nástrojem Docker, který se stal v posledních letech velmi populárním a diskutovaným tématem.

3.1 Úvod do kontejnerové architektury

Společnou vlastností moderních aplikací a zejména pak těch, které pracují na principu server–klient, je jejich narůstající komplexita. Nejedná se jen o složitost programového kódu, ale zvláště o komplexnost prostředí, ve kterém běží. Typická webová aplikace v dnešní době potřebuje aplikační server, webový server, (relační) databázi, *cachovací* databázi... Autor aplikace má dále na výběr z velkého množství dostupných jazyků a frameworků – NodeJS, Golang, Python, PHP, Ruby, Java atd., kdy každý z vyjmenovaných pracuje s různými správci externích závislostí – bower, npm, pip, Maven a další. Vzniká tak ohromné množství různých kombinací nástrojů, ze kterých může být aplikace sestavena. Do toho připočtěme požadavky na škálování anebo na časté změny požadavků a vývojářský tým se může dostat do stavu, kdy organizace a správa prostředí/infrastruktury vyžaduje nezanedbatelné množství času. Druhým, nikoliv však méně důležitým, častým požadavkem je rychlé spolehlivé opakovatelné a zejména automatizované vytvoření prostředí pro aplikaci. Schopnost jednoduše vytvořit (či vygenerovat) infrastrukturu oceníme v mnoha situacích, uveďme na příklad: automatizované testování aplikace (např. v Continuous Integration nástroji); havárii; nový člen týmu, který potřebuje vytvořit prostředí pro vývoj atd. Ve světě *enterprise* aplikací se tento problém často označuje termínem *spaghetti architecture*, v této oblasti je však kvalitně zdokumentován a existují řešení a metodologie, jak sestavit více aplikací do jednoho udržovatelného

systému. [15] Na následujících stránkách si ukážeme, jak vytvořit čistý návrh infrastruktury a prostředí i o úroveň níže při vývoji samotné aplikace, která konzumuje a vyžaduje řadu externích služeb, pomocí **kontejnerové architektury**. Hlavní myšlenkou aplikačních kontejnerů je rozdělit aplikaci na samostatné izolované procesy, které mezi sebou definují jednotným způsobem závislosti a komunikují spolu síťovými porty. Každý z procesů pak běží ve vlastním izolovaném prostředí, ve kterém není nijak ovlivněn okolním prostředím. Mnoho z těchto principů připomíná virtualizaci operačního systému, po technické stránce se však od kontejnerů v mnoha směrech liší: kontejnery využívají jádro operačního systému a hardware přímo, čímž umožňují běh řádově desítek kontejnerů na jediném stroji.

V březnu roku 2013 byl veřejnosti představen nový open-source kontejnerový nástroj Docker od stejnojmenné společnosti, mezi jehož základní funkce patří izolace procesů a jejich dat od svého okolí (tedy od samotného systému i ostatních kontejnerů). V době uvedení se nejednalo o nijak převratnou myšlenku, Docker v jádru staví a rozšiřuje myšlenku takzvaných Linux Containers (LXC), které byly začleněny do Linuxového jádra ve verzi `2.6.24` v lednu 2008 [16] (v červnu roku 2016 je aktuální *stable* Linuxové jádro ve verzi `4.6.2`). Samotné LXC má své předchůdce v technologii Linuxového jádra zvané `cgroups` (control groups), kterou představili Paul Menage a Rohit Seth ze společnosti Google v roce 2006. Byl to však právě Docker a posléze další technologičtí partneři projektu z řad IT společností, který vyvolal řadu diskuzí na téma budoucnosti virtualizace a způsobu, jakým se vyvíjí, testují, nasazují a provozují moderní internetové aplikace.

Na stránce projektu Docker https://www.docker.com/ se nachází kompletní dokumentace a soubory ke stažení. Technologie LXC včetně následnické technologie LXD jsou zastřešeny projektem Linux Containers (https://linuxcontainers.org/), který spravuje společnost Canonical[3].

V následujícím textu bude podrobně představena technologie stojící za projektem Docker, zaměříme se na reálné přínosy a dopady při vývoji aplikací.

3.2 Širší pohled na kontejnerovou architekturu

Docker je aktuálně považován za hlavního hráče v kontejnerové architektuře – ať už se jedná o množství vývojářů, kteří jej přijali, či o množství podporovaných platforem a operačních systémů. Nicméně Docker není jediný,

[3]Společnost Canonical stojí např. za populární linuxovou distribucí Ubuntu. Na svých stránkách pochopitelně propaguje svoji technologii LXD, avšak Canonical se zároveň podílí na vývoji Dockeru a tvrdí, že některé prvky LXD chtějí přenést i do nástroje Docker. Ostatně Docker base image „ubuntu“ patří mezi jeden z nejstahovanějších obrazů z Docker Hub. [17]

který se zabývá a prosazuje kontejnerovou architekturu. Představíme si několik projektů, které více či méně dělají podobné, co dělá Docker.

Velmi důležitý projekt ve světě kontejnerové architektury je Open Container Initiative (OCI, https://www.opencontainers.org/), který byl spuštěn 22. června 2015 [18] ve spolupráci s Linux Foundation[4] (http://www.linuxfoundation.org/). Součástí organizace OCI (původně označovaná jako OCP – Open Container Project) je mimo jiné i společnost Docker, mezi další technologické partnery patří velká jména ze světa IT jako je: Amazon Web Services, Cisco, Facebook, IBM, Oracle, Intel, Microsoft, Dell a další. [19] Jedním z hlavních cílů projektu OCI je vytvořit otevřený, univerzální a zejména standardizovaný formát pro popis bázových *images* kontejnerů. Ačkoliv je Docker otevřený projekt, který staví na svobodné licenci, stále se jedná o nestandardizovaný nástroj a mnoho firem se tak zcela oprávněně obává případného tzv. *vendor lock-in*[5]. [20] Aktuální znění specifikace „OCI Image Format Spec“ je dostupné na https://github.com/opencontainers/image-spec, specifikace vychází z Docker image formátu ve verzi 2.2. Významným projektem organizace OCI je linuxový operační systém CoreOS (https://coreos.com/). Ten se liší od tradičních distribucí jako je Debian, Ubuntu a další tím, že slouží výhradně ke spouštění aplikací v kontejnerech, proto neobsahuje např. ani *package manager* (apt-get, yum a další) k instalace software. CoreOS je tedy ideální systém pro instalaci na fyzický/virtuální server, pokud víme, že veškerý potřebný software poběží jen v kontejnerech. Systém podporuje kromě Docker kontejnerů i alternativní technologii k Dockeru – *Rocket* (také označovaná zkráceně *rkt*), která je zaměřena zejména na bezpečnost (kontejnery nejsou spouštěny s *root* oprávněním) a přenositelnost mezi budoucími kontejnerovými systémy.

3.3 Představení nástroje Docker

Autoři Dockeru jej na stránce projektu (https://www.docker.com/) popisují jako otevřenou[6] platformu pro vývoj, nasazení a běh aplikací. Hlavní předností tohoto nástroje je separace jednotlivých aplikací, které jsou zabaleny v kontejnerech, od zbytku systému – a to nejenom od dat a procesů ostatních aplikací, ale i od samotné infrastruktury (operační systém, síť atd.).

[4] Linux Foundation je velmi důležité a uznávané sdružení, které se stará o vývoj, financování a udržení nezávislosti Linuxu a mnoha dalších open-source projektů.

[5] „*Vendor lock-in*“ je stav, kdy je firma závislá na určité technologii/produktu/dodavateli a bez značných dodatečných nákladů nemůže tento stav opustit.

[6] Docker je vyvíjen pod licencí Apache 2.0 (APLv2), jejíž znění je dostupné na http://www.apache.org/licenses/LICENSE-2.0. Zvolená licence umožňuje komerční užití, modifikace a redistribuci software, její zjednodušený výklad je na https://tldrlegal.com/license/apache-license-2.0-(apache-2.0).

Docker a kontejnerová architektura obecně mají ambice stát se zavedeným průmyslovým standardem a alternativou ke klasické virtualizaci operačního systému, což dokládá i široká podpora napříč poskytovateli cloudových služeb, jmenujme některé významné zástupce:

- Amazon EC2 Container Service [21] od Amazon Web Services. Amazon je v současnosti největší poskytovatel cloudových služeb s tržním podílem 31 %. [22]
- Na platformě Microsoft Azure je možné přímo příkazem `docker-machine` vytvořit a komunikovat se vzdáleným hostitelským serverem a spouštět na něm kontejnery. [23]
- Platforma Heroku, která dlouho používala vlastní LXC technologii, podporuje od května 2015 spouštění Docker kontejnerů. [24]
- Google Cloud Platform umožňuje běh Docker kontejnerů [25] pomocí systému Kubernetes. Kubernetes je open-source platforma pro deployment a škálování kontejnerů napříč mnoha hostitelskými clustery. [26]
- DigitalOcean, který se na rozdíl od výše jmenovaných zabývá poskytováním VPS namísto PaaS, nabízí předkonfigurovaný „one-click install“ virtuální server pro rychlé nasazení Docker kontejneru.

Je nutné zdůraznit, že v současné době existují technická a architektonická omezení, která neumožňují kontejnerizaci libovolné aplikace. Software, který chceme zabalit do kontejneru, musí splňovat tyto minimální požadavky:

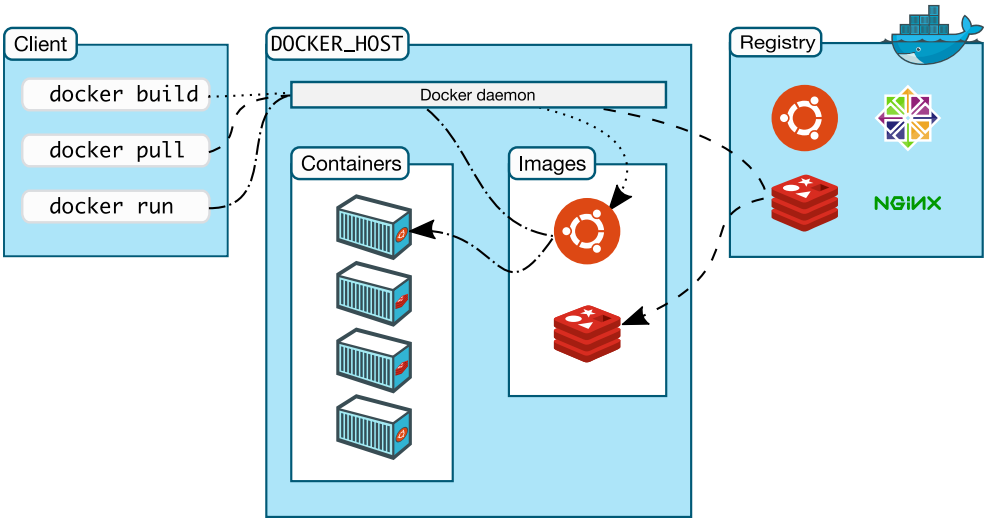
- Uvnitř kontejneru může v současné době běžet pouze linuxový operační systém. Operační systémy Windows, OS X a další nejsou podporované jako základ kontejneru (*base image*).
- Pokud se skládá aplikace z více procesů, musejí být tyto procesy oddělitelné tak, že každý běží v samostatném kontejneru. Až z těchto jednotlivých kontejnerů je posléze sestavena celá aplikace. Toto je architektonické omezení (či spíše doporučení), po technické stránce existují způsoby, jak spustit více procesů v jednom kontejneru – víceprocesové kontejnery však mohou značit nesprávné použití nástroje, a pak je ke zvážení, zda není vhodnější použít klasickou virtualizaci. [27, 28]
- Kontejnerizace se skvěle hodí k distribuci SaaS, v praxi tedy k aplikacím typu klient–server. Distribuce klasických desktop GUI aplikací pomocí Dockeru je technicky možná, avšak pro cílového uživatele aplikace není toto řešení vhodné.

3.4 Architektura Dockeru

V jádru využíval Docker nativní technologii linuxového jádra LXC, od verze `0.9` (aktuální verze v červnu roku 2016 je `1.12`) však přišel s vlastní implementací izolace procesů a knihovnou `libcontainer` [29], ta byla červnu roku 2015 přejmenována na knihovnu `runC` a předána do správy organizaci OCI[7]. [30]

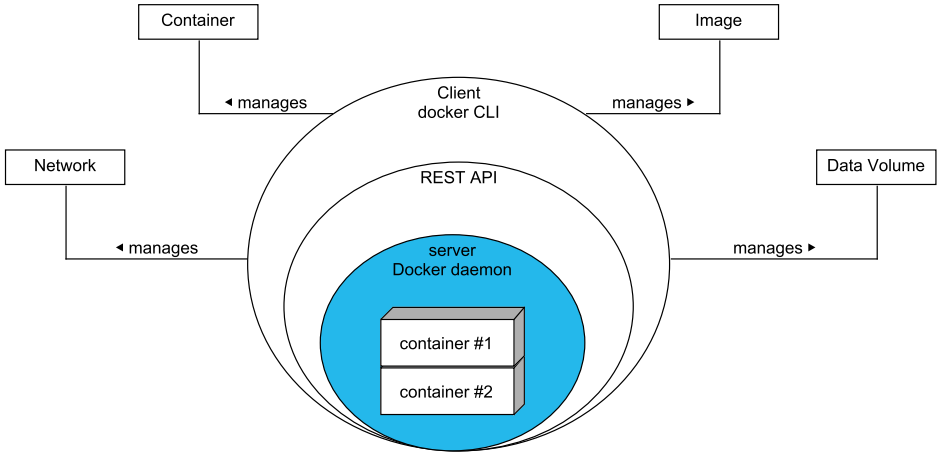
Docker se skládá ze třech hlavních součástí:

- **Docker Engine** – jádro pro vytváření a spouštění kontejnerů, Docker Engine je nainstalován na hostitelském počítači, kde s ním programátor komunikuje typicky přes CLI.
- **Docker Machine** – nástroj pro vytváření a správu tzv. Docker hosts (tj. fyzický, virtuální nebo cloudový stroj s nainstalovaným a nastaveným Docker Engine). Díky Docker Machine je možné aplikaci škálovat – tedy vytvářet a rušit hostitelské počítače a na nich spouštět kontejnery.
- **Docker Hub** (nebo také Registry Service nebo Docker Trusted Registry) – SaaS registr dostupný na https://hub.docker.com/, který umožňuje sdílení *base images*. V tomto repositáři jsou dostupné *base images* pro populární aplikace a nástroje (Mongo, Redis, MySQL atd.) stejně jako pro různé linuxové distribuce (Ubuntu, Debian, CentOS atd.).

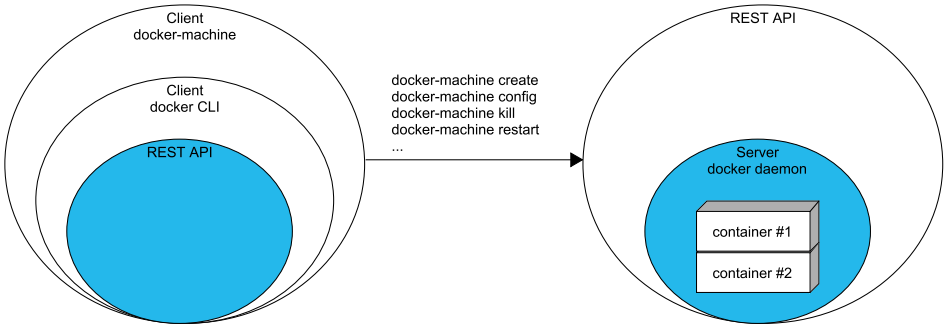


Obrázek 3.1: Celkový pohled na architekturu Dockeru. [31]

[7] Kód knihovny runC je dostupný jako open-source na https://github.com/opencontainers/runc.



Obrázek 3.2: Struktura Docker Engine. Komunikace mezi démonem a klientem (CLI) pracuje na principu klient–server pomocí REST API. [31]



Obrázek 3.3: Struktura Docker Machine, která je nadřazená Docker klientovi. Docker Machine určuje, s jakým strojem bude klient komunikovat – stroj může být lokální či vzdálený server. Díky této architektuře je možné komunikovat (tj. spouštět na něm kontejnery) se vzdáleným Docker hostem stejně jako s lokálním. [31]

Doposud bylo několikrát zmíněno, že Docker využívá pro spouštění kontejnerů linuxovou technologii LXC, která je součástí moderního linuxového jádra. Cílem Dockeru je mimo jiné usnadnit přenositelnost aplikace mezi různými operačními systémy, proto i samotný Docker Engine podporuje mnoho operačních systémů. Existují tři módy, ve kterých umí Docker Engine pracovat:

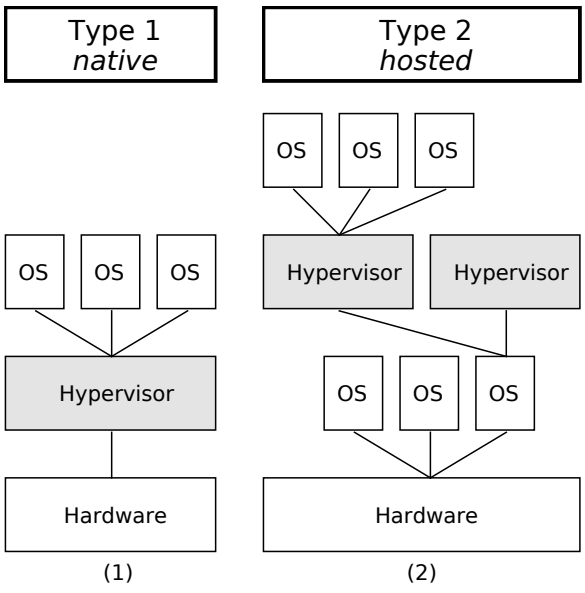
Nativně na Linuxu Toto je pro Docker nejpřirozenější cesta. Pokud je

linuxová distribuce postavena na moderním jádru, běží Docker nativně v systému, tedy přímo na hardware. Zmiňme některé populární linuxové distribuce, které tento mód podporují: Fedora 22+, Ubuntu 12.04+, Debian 8.0+, CentOS 7.X, Red Hat Enterprise Linux 7 a další.

Virtualizovaná Docker Machine OS X nepodporuje nativně technologii LXC a Microsoft Windows je postaven na zcela jiné architektuře. Pro OS X ve verzi 10.8 a vyšší a pro Microsoft Windows ve verzi x64 7 a vyšší je možné nainstalovat Docker pomocí Docker Toolbox. Docker Machine v tomto případě nepoužije současný stroj jako Docker host, ale pomocí Oracle VirtualBox vytvoří virtualizovaný stroj postavený na Tiny Linux (http://tinycorelinux.net/), na kterém pak běží Docker Engine. Jedná se de facto o stejnou situaci, jako kdybychom spouštěli kontejnery nativně z Linuxu na vzdáleném Docker hostu, který běží např. v cloudu. Oracle VirtualBox využívá Hypervisor typu 2, tedy hostovaný v operačním systému a běžící nad ním. Virtualizace je v tomto případě funkční řešení, díky kterému je Docker dostupný na široké škále systémů, z dlouhodobého hlediska však tento přístup vyžaduje mnoho kompromisů. Zásadním omezením je výkon, kontejner nemá přímý přístup k hardware a kontejnery jsou tak omezeny výkonem virtualizační aplikace. Druhým omezením je velmi komplikované sdílení dat, protože mezi kontejnerem a hostitelským počítačem leží mezivrstva navíc – v současnosti mají kontejnery přístup jen k datům umístěným v domovském adresáři uživatele („Home“ na Windows a „~/“ na OS X). Existují postupy, kterými je možné umožnit přístup kontejnerů k datům mimo domovské adresáře, to však vyžaduje netriviální úpravy Tiny Linux systému běžícím na Oracle VirtualBox. Instalace Dockeru pomocí Docker Toolbox je stabilní a podporovaná, z důvodu limitovaného výkonu a složitější konfiguraci na cílovém systému se však jedná pravděpodobně o slepou vývojovou cestu.

Nativně na Windows 10 a OS X Na konci března roku 2016 byl představen nativní Docker Engine (dostupný na https://beta.docker.com/) pro operační systémy Microsoft Windows a OS X. [32] Tato nová metoda přináší nativní aplikace pro obě platformy. Označení „nativní“ je v tomto kontextu zavádějící, protože stále platí, že Docker Engine využívá technologii LXC, avšak v oficiálních materiálech se o ní tak referuje. Zásadní novinkou oproti předchozí metodě je využití nativní virtualizace, kterou tyto systémy nabízí – v případě Microsoft Windows se jedná o technologii Hyper-V VM, na OS X pak o xhyve Virtual Machine. V principu se jedná o stejnou virtualizaci linuxového operačního systému jako u předchozí metody, s tím rozdílem, že Tiny Linux byl nahrazen distribucí Alpine Linux (http://www.alpinelinux.org/). Výhodou je, že obě virtualizační metody

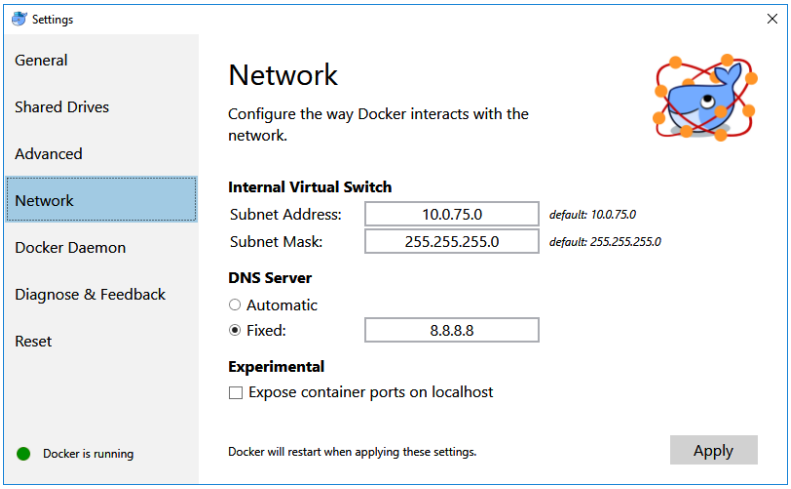
využívají hypervisor typu 1, který běží přímo na hardware – z tohoto důvodu se pravděpodobně označuje celá metoda jako „nativní“. Díky těsnější integraci se systémem byly také odstraněny omezení se sdílením souborů mezi hostem a kontejnery. Do budoucna je toto preferovaná metoda, která nahradí předchozí řešení s Oracle VirtualBox. Avšak i zde můžeme najít některé nevýhody – vše je ve fázi beta, tedy software může obsahovat a obsahuje chyby, není tedy připraven k produkčnímu nasazení. Druhou nevýhodou je v případě Microsoft Windows podpora pouze nejnovější verze x64 10 resp. OS X ve verzi 10.10.3+, pro starší systémy je nutné využít předchozí model. Mezi další vylepšení patří: automatická notifikace o změnách filesystémů mezi hostem a kontejnerem, podpora VPN (díky nativní podpoře sítě) a další.



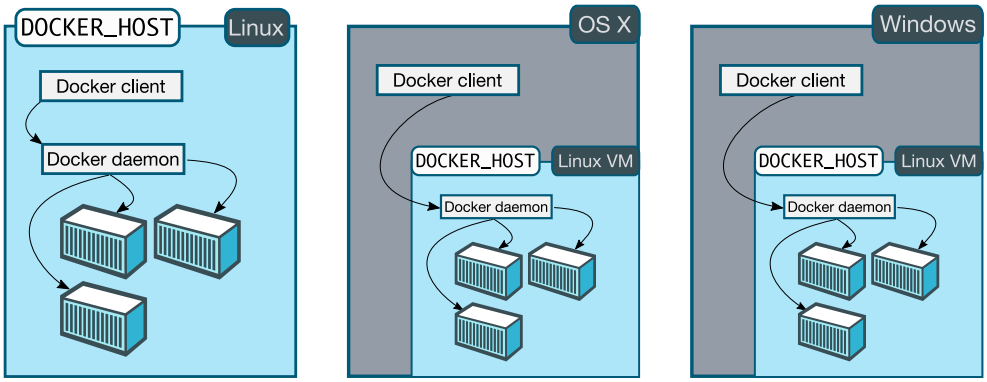
Obrázek 3.4: Dva typy virtualizačního nástroje hypervisor (VMM). **(1)** Hypervisor typu 1 je v systému nativní, běží přímo na hardware daného stroje (někdy se proto označuje pojmem hypervisor bare-metal). **(2)** Hypervisor typu 2 běží až na operačním systému, který běží na hardware na daném stroji.

Shrneme-li předchozí text do dvou bodů, Docker:

1. Staví na linuxové technologii LXC, která umožňuje izolaci procesů od jejich okolí (resp. kontejnerů).
2. Využitím UnionFS umožňuje verzovat jednotlivé vrstvy *base images*, které tvoří základ kontejnerů.



Obrázek 3.5: Hlavní rozhraní nativní aplikace Docker na Windows 10.

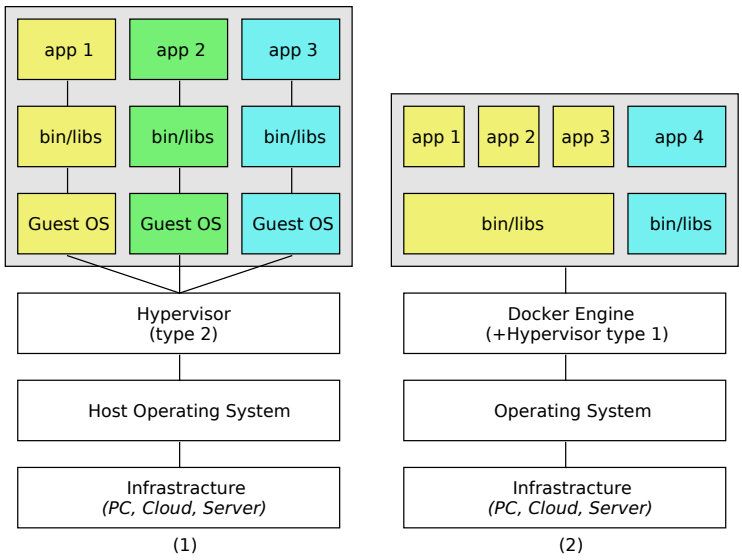


Obrázek 3.6: Rozdíly v architektuře Docker na třech hlavních podporovaných systémech. [31]

3.5 Docker vs virtualizace

Běžící Docker kontejner je možné přirovnat k Linuxovým procesům, oproti klasické virtualizaci nabíhají Docker kontejnery rychle (řádově milisekundy až

jednotky sekund), nevyžadují tolik systémových prostředků, a přesto zevnitř kontejneru vypadá okolní prostředí jako samostatná a izolovaná Linuxová distribuce. [33] Jedním z hlavních rozdílů je výkon – zatímco fyzický stroj zvládne spustit řádově jednotky virtuálních operačních systémů, kontejnerů mohou běžet na jediném stroji desítky až stovky. Na druhé straně je nutné zmínit, že plně virtualizovaný systém má alokované a garantované přiřazené prostředky a poskytuje nejvyšší možnou izolaci od svého okolí.



Obrázek 3.7: Srovnání virtualizovaného stroje a kontejnerové architektury.
(1) Každá aplikace běží na vlastním operačním systému, mezi kódem aplikace a hardware se nachází dvě mezivrstvy (Guest OS a virtualizace spravovaná hostem).
(2) Více aplikací může sdílet stejné knihovny, které tak nemusí být na stroji nainstalované několikrát. Docker Engine běží přímo na hardware (díky Hypervisoru typu 1).

3.6 Docker prakticky

V této kapitole si předvedeme Docker v praxi. Nejprve si ukážeme základní použití nástroje – získání *base image* třetí strany a jeho spuštění (tedy vytvoření kontejneru). Následně přejdeme k vytváření vlastních *images*, což je pravděpodobně hlavní úkol, ke kterému budou vývojáři Docker používat. Docker Engine se ovládá CLI klientem pomocí příkazu `docker`. Mezi klientem a Docker Enginem probíhá komunikace pomocí JSON REST API, je tedy relativně snadné postavit nad tímto API samostatnou aplikaci pro alternativní

ovládání Docker Engine. Mnohé alternativní i oficiální aplikace již existují, jednou z nich je např. GUI aplikace Kitematic přímo od autorů Docker (https://www.docker.com/products/docker-kitematic).

3.6.1 Spouštíme kontejner

Jako ukázku práce s Dockerem spustíme oficiální *base image* „ubuntu“. Desítky dalších oficiálních i neoficiálních *base images* je možné nalézt na Docker Hub (https://hub.docker.com/explore/), kam je možné bezplatně umisťovat vlastní veřejné *base images* anebo privátní za poplatek. Není však nutné spoléhat na cloudovou službu třetí strany a typicky firmy si mohou spustit vlastní privátní registr (zvaný Distribution nebo také Docker Registry 2.0[8]) pro distribuci vlastních *base images* např. na interní síti.

Základní *base image* „ubuntu“ stáhneme lokálně příkazem `docker pull ubuntu`. Každý *base image* může obsahovat několik tagů (verzí), jmenovitě ubuntu obsahuje tagy 12.04, 14.04, 15.10 atd. – jedná se o jednotlivé verze této distribuce, pokud tag vynecháme, použije se ve výchozí tag „latest“ (tedy to samé jako kdybychom napsali `docker pull ubuntu:latest`).

```
$ docker pull ubuntu
Using default tag: latest
latest: Pulling from library/ubuntu

5ba4f30e5bea: Pull complete
9d7d19c9dc56: Pull complete
...
a3ed95caeb02: Pull complete
Digest: sha256:46fb5d001...e35dbb6870585e4
Status: Downloaded newer image for ubuntu:latest
```

Docker právě stáhl z veřejného Docker Hub (pokud neřekneme jinak, jedná se o výchozí chování) všechny lokálně chybějící vrstvy obrazu „ubuntu“. Všechny další obrazy, které vycházejí z právě staženého kódu anebo využívají některou z jeho vrstev, použijí stažená data. Všechny lokálně dostupné obrazy je možné získat příkazem `docker images`.

```
$ docker images
REPOSITORY  TAG     IMAGE ID        CREATED     SIZE
...
ubuntu      latest  2fa927b5cdd3    2 weeks ago 122 MB
...
```

[8]Původní registr *base images* napsaný v jazyku Python se jmenoval Docker Registry, projekt byl však pozastaven. Aktuální verze registru je napsaná v jazyku Go (ve kterém je napsaný i Docker) a nachází se pod svobodnou licencí na https://github.com/docker/distribution. Zprovoznění vlastního registru je relativně jednoduché – aplikaci je možné spustit jako kontejner.

Nyní můžeme spustit první kontejner. Při spouštění nového kontejneru musíme znát I) název základního obrazu a II) proces, který se má v kontejneru spustit. Na příklad Bash v ubuntu spustíme následovně: `docker run -it ubuntu /bin/bash`, volba `-it` říká, že chceme vytvořit i iterativní terminál (tedy držet `STDIN`).

Ukázka kódu 3.1 Spuštění kontejneru. Příkazem `docker run` říkáme: spusť image „ubuntu“ a v něm pak příkaz (proces) „`/bin/bash`“.

```
$ docker run -it ubuntu /bin/bash
root@85f0d0e75474:/# pwd
/
root@85f0d0e75474:/# whoami
root
root@85f0d0e75474:/# uname -r
4.4.12-moby
```

Na předcházejícím textovém výstupu je vidět, že se nacházíme v právě spuštěném kontejneru a celé prostředí se chová jako běžná linuxová distribuce Ubuntu – je důležité poznamenat, že se jedná o skutečný Linux bez žádných omezení a kompromisů. Zajímavostí je, že výše uvedený výstup byl získán z počítače, na kterém běží Windows 10.

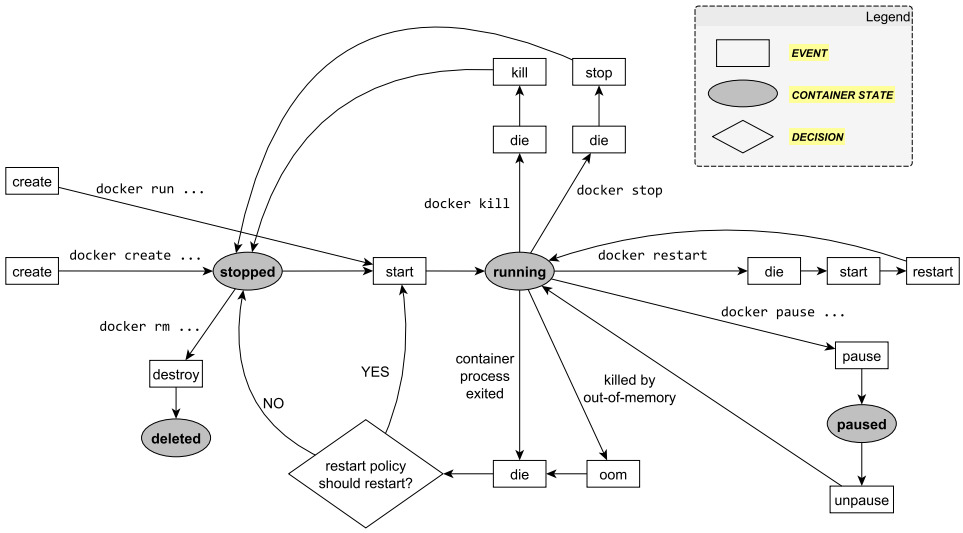
Připomeňme, že Docker Image je read-only šablona k vytvoření kontejneru. A dále připomeňme, že kontejner je instance *image*, jejíž životní cyklus končí s koncem procesu spuštěném v kontejneru. Tato tvrzení dokážeme smazáním kořene filesystému (`/`) v běžícím kontejneru.

```
$ docker run -it ubuntu /bin/bash
root@b92811fb7bfe:/# ls -l
total 64
drwxr-xr-x   2 root root 4096 May 25 23:11 bin
drwxr-xr-x   2 root root 4096 Apr 12 20:14 boot
drwxr-xr-x   5 root root  380 Jun 16 23:56 dev
...
drwxr-xr-x  11 root root 4096 May 27 14:14 usr
drwxr-xr-x  13 root root 4096 May 27 14:14 var
root@b92811fb7bfe:/# rm --no-preserve-root -rf /
root@b92811fb7bfe:/# ls bash:
ls: command not found
```

Takto jsme skutečně smazali data v kontejneru, nicméně základní image zůstal neporušen. Opustíme-li tento kontejner a spustíme-li nový kontejner ze stejného základního image, vše se nachází v původním stavu.

```
$ docker run -it ubuntu /bin/bash
```

```
root@9c4868202406:/# ls
bin  boot  dev  etc  home...
```



Obrázek 3.8: Diagram stavů, ve kterých se může nacházet Docker kontejner. [34] Jednotlivé stavy jsou: *stopped* – base image je připraven ke spuštění; *deleted*; *running* – uvnitř kontejneru je/jsou aktivní proces/procesy; *paused* – kontejner je dočasně pozastaven pomocí cgroups freezer (signálem `SIGSTOP`), proces uvnitř kontejneru nemůže odchytit událost, že má být uspán. Rozdíl mezi příkazy `kill` a `stop` je následující: `docker stop` vyšle signál `SIGTEM`, zatímco `docker kill` vyšle `SIGKILL`.

3.6.2 Ukládání dat v kontejneru

Data *base images* ukládá Docker pomocí UnionFS[9], který pracuje na principu uspořádaných vrstev změn filesystému – každá nová vrstva přidává informace o změnách k vrstvě předchozí. Princip je velmi podobný verzovacímu nástroji git, ostatně i *base image*s je možné verzovat příkazem `docker commit`. UnionFS pracuje nad adresářem (tedy nemá *mountovací* bod jako klasické filesystémy) a spojuje do jednoho logického pohledu na data fyzické soubory z různých míst

[9]Pro korektnost bychom měli tvrdit, že Docker používá filesystém typu UnionFS. Díky tzv. *storage drivers* dává Docker na výběr, jaký filesystém má být použit. [35] Na výběr jsou: OverlayFS, AuFS, Vrtfs, Device Mapper, VFS nebo ZFS. Při instalaci se Docker rozhodne, který z filesystémů má být použit jako výchozí, v případě AlpineLinux je výchozím filesystémem AuFS (zkratka pro „Another Union File System“). [36] Z pohledu „uživatele“ však pracují všechny systémy na podobných principech a liší se v technických detailech a případech použití, v textu tedy použijeme obecný termín „UnionFS“. Změnit filesystém je možné při startu Docker démona, aktuálně používaný filesystém je možné kdykoliv zjistit příkazem `docker info` v sekci „Storage Driver:“.

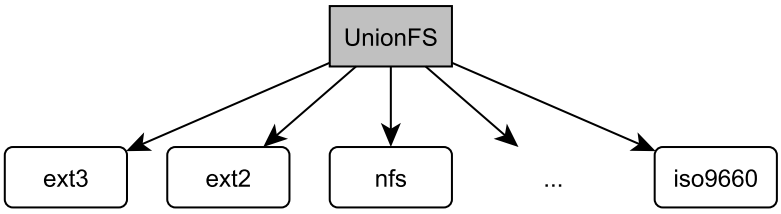
(i různých filesystémů). Každý takto logicky spojený adresář se nazývá *union* a jednotlivé fyzické adresáře se nazývají *branch*. [37] Docker využívá tohoto principu pro úsporu zabraného místa na disku[10], rychlé spouštění kontejnerů a ke snížení přenášených dat.

Na příkladu níže vidíme, že obraz nginx, který právě stahujeme, se skládá ze čtyř vrstev a při stahování obrazu z Docker Registry bylo detekováno, že vrstva se `sha-256` otiskem `51f5...` je již lokálně stažená a není potřeba ji stahovat znovu – značí to tedy, že obraz nginx sdílí tuto vrstvu s jedním nebo více obrazy, které se v systému nachází. Pro podrobný průzkum vrstev obrazu je možné použit příkaz `docker inspect [CONTAINER|IMAGE]`, dále existuje užitečná webová služba https://imagelayers.io/, která graficky znázorní jednotlivé vrstvy obrazu a příkazy (změny), které danou vrstvu vygenerovaly – to je může být užitečné i tehdy, pokud nemáme důvěru v obraz třetí strany a chceme se ujistit, že některá z vrstev neobsahuje závadný kód/příkaz.

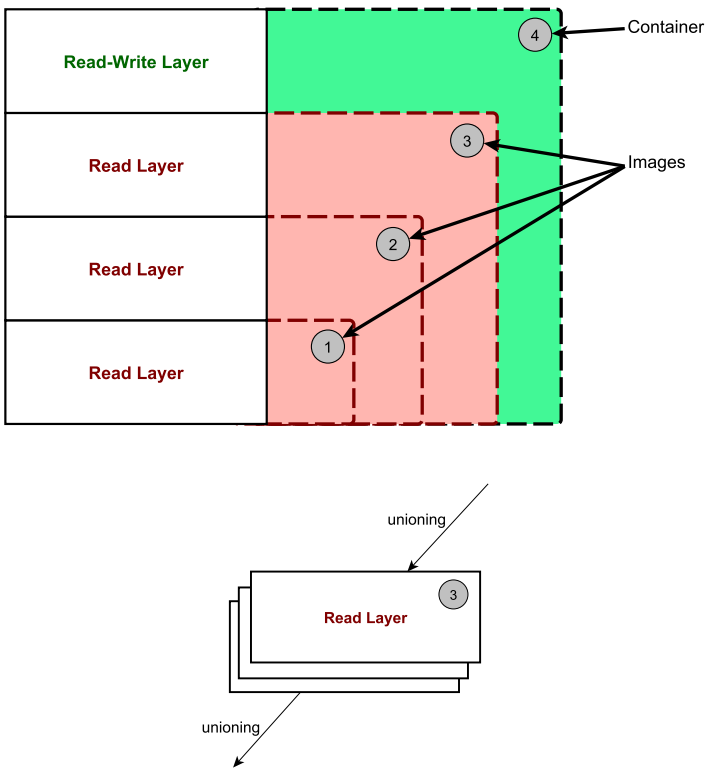
Ukázka kódu 3.2 Stažení image nginx z Docker Hub. Obraz je uložen pomocí UnionFS jako posloupnost vrstev. Pokud se nějaká z vrstev již nachází na lokálním stroji (protože ji předtím vyžadoval jiný obraz), Docker tuto vrstvu znovu duplicitně nestahuje. Více obrazů tak může sdílet jedna fyzická data.

```
1  $ docker pull nginx
2  Using default tag: latest
3  latest: Pulling from library/nginx
4
5  51f5c6a04d83: Already exists   # <-----
6  a3ed95caeb02: Pull complete    # Download
7  51d229e136d0: Pull complete    # Download
8  bcd41daec8cc: Pull complete    # Download
9  Digest: sha256:0fe6413...9e2f29d04
10 Status: Downloaded newer image for nginx:latest
```

[10] Uvažujme, že Docker nepoužívá UnionFS, poté by spuštění 100 MB obrazu jako 5 samostatných kontejnerů vyžadovalo přibližně 500 MB diskového prostoru.



Obrázek 3.9: Struktura UnionFS. Logický pohled na data (*union*) se skládá z několika fyzických zdrojů dat (*branches*), které se mohou nacházet v různých filesystémech. [37]



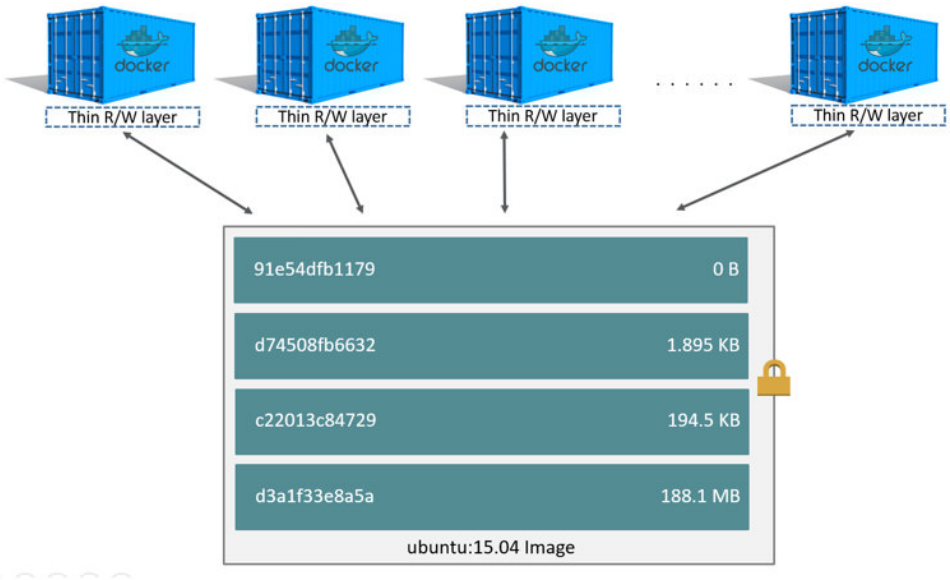
Obrázek 3.10: UnionFS z pohledu kontejneru. Kontejner namapovaný jako čtvrtá (poslední) vrstva vidí spojenou (union) třetí vrstvu (toto spojení vyobrazuje spodní obrázek). Kontejnerová vrstva umožňuje zápis do své vrstvy, ale již neumožňuje zápis do nižších vrstev. Data, která zapíše kontejner do své vrstvy můžeme buď perzistovat příkazem `docker commit` a vytvořit tak zcela nový image o čtyřech vrstvách anebo můžeme poslední vrstvu zahodit po ukončení kontejneru, další spuštěný kontejner získá vlastní novou vrstvu pro zápis. [38] Z tohoto plyne velmi důležitý poznatek: kontejner se od *image* liší pouze v poslední Read-Write vrstvě a to navíc tak, že nezáleží, zda kontejner běží či je zastaven.

Právě jsme si ukázali, že kontejner obsahuje Read-Write vrstvu, do které může proces(y) za svého běhu zapisovat. Tato data však existují jen do té doby, dokud existuje kontejner. Otázkou tedy je, jakým způsobem uchovat data perzistentně v systému, typicky to mohou být uživatelské soubory nahrané přes webové rozhraní, datové soubory databází atd. Za tímto účelem se aktuálně[11] používají dva koncepty: [40]

[11] Pohled na perzistentní data v kontejnerech se od verze Dockeru 1.9 relativně výrazně změnil. [39] Aktuální vývoj značí, že Data volumes a Data volume containers jsou tou správnou cestou.

1. Data volumes;

2. Data volume containers.



Obrázek 3.11: Princip sdílení vrstev mezi kontejnery. [35]

3.6.2.1 Data Volumes

Data volume je adresář, který obchází UnionFS a existuje mimo něj na Docker machine. Tento adresář je připojen ke kontejneru v momentě, kdy kontejner startuje. Stejný Data volume adresář může být připojen najednou k více kontejnerům, které tak mohou mezi sebou sdílet data. Změny v datech v tomto volume adresáři jsou propagovány okamžitě a data jsou v něm uchována perzistentně – Docker nikdy volume nesmaže a adresář existuje i poté, co jsou všechny kontejnery, které jej využívaly, odstraněny.

Data volume můžeme připojit (resp. vytvořit, pokud neexistuje) při spouštění kontejneru volbou `-v`, která jako parametr přijímá absolutní cestu v kontejneru, relativní cesty podporovány nejsou. Pokud mapovaná cesta v kontejneru již existuje, Data volume má přednost a původní cestu „překryje“ – originální data však zůstanou nezměněna a při dalším spuštění kontejneru bez Data volume jsou na svém místě. Každý kontejner může mít více než jeden volume adresář.

Ukázka kódu 3.3 Data volume připojené při spuštění kontejneru. Pomocí parametru `-v` jsme ke kontejneru připojili dva speciální data volume adresáře `/volume_1` a `/volume_2`. Obsah těchto adresářů i adresáře samotné jsou perzistentní a budou existovat i po ukončení/odstranění právě spuštěného kontejneru.

```
$ docker run -it --rm \
        -v /volume_1 -v /volume_2 ubuntu /bin/bash
root@569635fd126d:/# ll
total 80
...
drwxr-xr-x 11 root root 4096 May 27 14:14 usr/
drwxr-xr-x 13 root root 4096 May 27 14:14 var/
drwxr-xr-x  2 root root 4096 Jun 17 17:22 volume_1/
drwxr-xr-x  2 root root 4096 Jun 17 17:22 volume_2/
```

Informace o tom, jaké volumes jsou namapovány do kontejneru a jejich fyzické umístění na filesystému je možné získat příkazem `docker inspect`.

Ukázka kódu 3.4 Výstup `docker inspect` s namountovanými volumes. Vidíme, že data se reálně nacházejí na cestě `/var/lib/docker/volumes/`.

```
{
 "Id":"831d7283c6f34...462f103",
 "Mounts": [
  {
   "Name": "58e8bb1254ef15...86fe65e662",
   "Source": "/var/lib/docker/volumes/58e8b...65e662/_data",
   "Destination": "/volume_1",
   "Driver": "local",
   "Mode": "",
   "RW": true,
   "Propagation": ""
  },
  {
   "Name": "7b08dc59ea6c8168...b7316fd6b",
   "Source": "/var/lib/docker/volumes/7b08dc59...d6b/_data",
   "Destination": "/volume_2",
   "Driver": "local",
   "Mode": "",
   "RW": true,
   "Propagation": ""
  }
 ],
}
```

Užitečnou funkcí je schopnost namapovat adresáře na hostu jako Data

volume pomocí parametru `-v HOST_PATH:CONTAINER_PATH`. Toto je užitečné zejména při vývoji, kdy na lokálním počítači upravujeme kód, který je namapován dovnitř kontejneru, který jej spouští. Změny v kódu na lokálním počítači se pak propagují okamžitě do kontejneru. Mapování dat z hosta do kontejneru je z bezpečnostních důvodů určeno jen pro testování a vývoj, protože přesun podadresáře z namapovaného nadřazeného adresáře mimo může umožnit kontejneru přístup k datům hosta. [41]

3.6.2.2 Data Volumes Containers

Druhý způsob uchování perzistentních dat v kontejneru je Data volume container (DVC). Po technické stránce se jedná o stejná Data volume jako v předchozím způsobu. Liší se jen metoda, jak se adresáře připojují ke kontejneru a jak se definují. U Data volume containers se využívá principu, že kontejner může přijmou Data volume z jiného kontejneru. Demonstrujme DVC na příkladu. Mějme base image s databází postgres, která ukládá data do adresáře `/var/lib/postgresql/data`. Spustíme-li postgres image přímo (příkazem `docker run --rm -it postgres`), databáze nastartuje a čeká na připojení. Data se správně ukládají do `/var/lib/postgresql/data`, avšak po zastavení kontejneru o tato data přijdeme. Vytvoříme tedy kontejner, jehož jediným úkolem bude vystavit Data volume (uvnitř kontejneru tedy nebude žádný proces – aplikace). [42]

Ukázka kódu 3.5 Vytvoření Data volume kontejneru. Tento příkaz vytvoří volume s absolutní cestou `/var/lib/postgresql/data`, tedy místo, kam databáze postgres ve výchozím stavu ukládá data. Kontejner jsme pojmenovali jako `postgres-datastore`, tento krok není povinný, ale budoucí použití je pak přehlednější. Posledním parametrem je `busybox`, protože každý kontejner musí definovat nějaký image, který spouští. Busybox je miniaturní (~5 MB) linuxová distribuce jen se základními unix nástroji.

```
1 $ docker create -v /var/lib/postgresql/data \
2                 --name postgres-datastore busybox
3 6b3f23...3157
```

Jako poslední krok spustíme samotnou databázi, při spouštění kontejneru využijeme dříve vytvořený Data volume kontejner. Novému kontejneru řekneme pomocí parametru `--volumes-from`, že má převzít Data volume z jiného kontejneru.

Ukázka kódu 3.6 Spuštění databáze postgres jako kontejner se závislostí na Data volume kontejner. Parametr `--detach` říká, že má kontejner běžet na pozadí, jedná se tedy o přesný opak iterktivního kontejneru spuštěného s `-it`. Druhý příkaz `docker ps` jen dokládá, že kontejner skutečně běží.

```
$ docker run --detach \
             --volumes-from postgres-datastore postgres
b96a...212e

$ docker ps
CONTAINER ID   IMAGE    ... STATUS         PORTS     ...
b96a4069dfaf   postgres ... Up 10 seconds  5432/tcp  ...
```

Pro ověření se připojíme do kontejneru postgres-datastore a podíváme se na obsah volume adresáře.

```
$ docker run --rm -it \
             --volumes-from postgres-datastore busybox
/ #
/ # cd /var/lib/postgresql/
/ # ls -l
total 4
drwx------   19 999      root  ... data
/ #
/var/lib/postgresql # cd data/
/var/lib/postgresql/data # ls -l
total 120
-rw-------    1 999      ... PG_VERSION
drwx------    5 999      ... base
drwx------    2 999      ... global
drwx------    2 999      ... pg_clog
...
-rw-------    1 999      ... postgresql.auto.conf
-rw-------    1 999      ... postgresql.conf
-rw-------    1 999      ... postmaster.opts
-rw-------    1 999      ... postmaster.pid
```

Na výpisu vidíme, že se adresář skutečně plní z databáze postgres.

3.6.3 Vlastní images

Až doposud jsme používali obrazy třetích stran stažené z Docker registry. Tento postup dostačuje, chceme-li jen spouštět již připravené aplikace. V této kapitole si ukážeme, jak vytvořit vlastní Docker image, sestavit jej a spustit v podobě kontejneru. Existují dva způsoby, kterými je možné vytvořit obraz:

1. **Příkazem `docker commit`** je možné perzistovat poslední Read-Write vrstvu kontejneru. Docker commit zkopíruje tuto poslední vrstvu do nového image.
2. **Konfiguračním souborem Dockerfile**. Dockerfile je obyčejný textový soubor, který říká, jaké kroky jsou nutné pro sestavení image. Toto je obvyklejší a přehlednější cesta, zvláště pak při vývoji nové aplikace. Dockerfile je možné spravovat verzovacím nástrojem a zajistit tak, že např. všichni členové týmu mají stejnou konfiguraci. Drobnou nevýhodou při použití Dockerfile je fakt, že při každé změně je nutné znovu sestavit celý base image.

Nejprve si předvedeme, jak vytvořit nový base image perzistováním vrstvy kontejneru v UnionFS. Postup i názvy jednotlivých parametrů v mnohém kopírují verzovací nástroj git.

Ukázka kódu 3.7 Vytvoření vlastního image příkazem `commit`. Nejprve byl spuštěn základní image „ubuntu“. Do Read-Write vrstvy ve spuštěném kontejneru byl zapsán nový soubor `new_file.txt`. Po ukončení kontejneru byla poslední vrstva UnionFS na kontejneru perzistována příkazem `docker commit` – konvence pro pojmenovávání base images je `USERNAME/IMAGE-NAME[:/TAG_SEM_VER]`.

```
1  $ docker run -it ubuntu /bin/bash
2
3  root@812c5d311ea1:/# touch new_file.txt
4  root@812c5d311ea1:/# exit
5
6  $ docker commit \
7          --message "File was added to the container" \
8          --author "John Doe" \
9          812c5d311ea1 \
10         johndoe/my-ubuntu:v0.1.0
11 sha256:3383c53c7...6cde47dab851d
12
13 $ docker images
14 REPOSITORY          TAG      IMAGE ID       ...   SIZE
15 johndoe/my-ubuntu   v0.1.0   812c5d311ea1   ...   122 MB
```

Druhou cestou k vytvoření vlastního image je Dockerfile. Jedná se o textový soubor oddělený novými řádky. Na každém řádku se nachází právě jeden příkaz, při vytváření image se pak příkazy označené direktivou `RUN` spouštějí v defaultním *shellu* (typicky `/bin/bash` nebo `/bin/sh`). Technicky vše opět pracuje s vrstvami UnionFS, každá direktiva `RUN` vytvoří novou vrstvu v image, která je automaticky perzistována (`commit`).

Ukázka kódu 3.8 Formát konfiguračního souboru Dockerfile. Příkaz `INSTRUCTION` může být `FROM`, `MAINAINER`, `RUN`, `CMD`, `LABEL`, `EXPOSE`, `ENV`, `ADD`, `COPY`, `ENTRYPOINT`, `VOLUME`, `USER`, `WORKDIR`, `ARG`, `ONBUILD` nebo `STOPSIGNAL`. [43]

```
1 # Comment
2 INSTRUCTION arguments
```

Při sestavování image je výstup každého řádku *cachován*, tím je zajištěn rychlý opakovaný *build*. Pokud Docker zjistí, že se některý z řádků změnil, změněný řádek a všechny následující řádky jsou invalidovány[12] a vykonají se znovu. *Cachování* funguje i pro direktivy `ADD` a `COPY`[13], které kopírují adresáře nebo soubory z Docker host do image. U každého zkopírovaného souboru je spočítán *checksum*, pokud se při následujícím buildu *checksum* neliší, Docker považuje soubory za stejné a znovu je nekopíruje. Pokud se *checksum* liší, jsou aktuální řádek a následující řádky invalidovány. *Cachování* je možné zcela vypnout volbou `--no-cache=true` u příkazu `docker build`. [43, 44]

Cache je při vytváření image velmi důležitý prvek, který rozhoduje, zda se image sestaví prakticky ihned anebo po delší době. Uvažujme na příklad příkaz pro distribuce postavené na systému Debian `RUN apt-get update`, který ze vzdálených repozitářů stáhne informace o aktualizacích systémových balíčků a jejich závislostech. Tento krok může trvat řádově jednotky minut, navíc jej chceme typicky spouštět jen tehdy, pokud instalujeme do image nový balíček. Právě v těchto situacích pracuje Dockerfile cache výborně. Existují však i situace, při kterých může *cachování* způsobit obtížně nalezitelnou chybu. Pak má vývojář na výběr dvě možnosti, buď přepínačem `--no-cache=true` cache zcela vypne, problém je odstraněn, ale zároveň se připravil o rychlé *buildy* obrazů. Druhou citlivější možností je pochopení problému a jeho odstranění i s použitým *cachováním*. Uvažujme následující Dockerfile: [45]

```
1 RUN git clone https://github.com/.../some_repo.git
2 WORKDIR /some_repo
3 RUN git checkout v1.0.0
```

Spustíme-li build tohoto Dockerfile do *image* příkazem `docker build .-f Dockerfile`, nejprve se stáhne git repozitář a řádek je *cachován*, poté změníme pracovní adresář, řádek je opět *cachován*, jako poslední krok se přepneme v repozitáři na vybraný tag a výsledek je opět *cachován*. Spustíme-li znovu tento build, použijí se výsledky cache a žádná z výše popsaných operací se tedy reálně neprovede. Nyní uvažujme, že jsme do vzdáleného git repozitáře nahráli nový kód a označili jej novým tagem `v2.0.0`. Změníme tedy příslušný tag v Dockerfile a spustíme build znovu.

```
1 RUN git clone https://github.com/.../some_repo.git
2 WORKDIR /some_repo
3 RUN git checkout v2.0.0
```

[12] Cache je zneplatněna.

[13] Direktivy `COPY` a `ADD` slouží obě ke stejnému účelu – zkopírovat data do image. Direktiva `COPY` slouží pouze ke zkopírování lokálních dat nebo adresářů do image. Na druhé straně direktiva `ADD` podporuje více funkcí – dokáže zkopírovat data ze vzdálené URL nebo rozbalit TAR archiv. Dokumentace doporučuje preferovat direktivu `COPY`, její chování je transparentní.

Nový build však skončí chybou:

```
Step 5 : ...
Step 6 : ...
Step 7 : RUN git clone https://git.../some_repo.git
    ---> Using cache
    ---> 104e...220

Step 8 : WORKDIR /some_repo
    ---> Using cache
    ---> 7d1...1a5

Step 9 : RUN git checkout v2.0.0
    ---> Running in 86d...ac error: pathspec 'v2.0.0'
         did not match any file(s) known to git.

The command [/bin/sh -c git checkout v2.0.0 returned a non-zero
    code: 1]
```

Celý build havaroval, protože nástroj git nenalezl nový tag `v2.0.0`, my přitom s jistotou víme, že tento tag se ve vzdáleném repozitáři nachází. Problém spočívá v tom, že pro první dva řádky použil Docker cache, zatímco třetí řádek invalidoval a znovu jej spustil. Cache byla však vytvořena ve chvíli, kdy nový tag `v2.0.0` ještě neexistoval a Docker nemá způsob, jakým by zjistil, že se nějaká vzdálená *resource* změnila. Protože nechceme cache vypínat, upravíme Dockerfile, aby příkazy `git clone` a `git checkout` provedl zároveň pomocí operátoru `&&`[14]. Cachování bude stále fungovat a jestliže Docker invaliduje cache pro daný řádek, oba příkazy se vykonají znovu. Následuje upravený Dockerfile se správným použitím cache: [45]

```
WORKDIR /some_repo
RUN git clone https....git . && git checkout v2.0.0
```

[14]Operátor „`EXPRESSION_1 && EXPRESSION_2`“ spustí `EXPRESSION_2` právě tehdy, když `EXPRESSION_1` vrátí kód `0` (nenastala žádná chyba).

Stejný postup se používá při instalaci systémových balíčků pomocí např. `apt-get`, následuje příklad Dockerfile, který správně *cachuje* a umožňuje přidávat systémové závislosti:

```
1 # Latest LTS
2 FROM ubuntu:16.04
3
4 RUN apt-get update && apt-get install -y \
5     git \
6     python \
7     ...
8
9 RUN ...
```

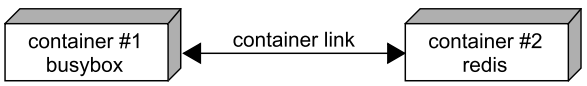
3.7 Spojování kontejnerů

3.7.1 Propojení dvou a více kontejnerů mezi sebou

Až doposud jsme se bavili o vytváření a spouštění jednoho jediného image resp. kontejneru. Reálná aplikace se však často neskládá z jediného procesu, nýbrž z mnoha procesů, které jsou na sobě závislé a které spolu komunikují. Uvažujme ten nejjednodušší příklad: aplikace využívá databázi. Filozofie kontejnerové architektury říká, že jednotlivé procesy by od sebe měly být izolovány a že mají komunikovat přes abstraktní jednotné rozhraní. V případě Dockeru je postup přímočarý: každý proces aplikace zabalíme do samostatného kontejneru.

V předchozím textu (viz sekce Ukládání dat v kontejneru na straně 27) jsme si ukázali, jak mohou dva a více kontejnerů společně komunikovat přes sdílený adresář. Docker poskytuje podobně jako v případě Data volume způsob, jak na síťové úrovni provázat kontejnery. [46]

Spojování kontejnerů demonstrujme na následujícím příkladu. Uvažujme kontejner `#1`, který ukládá a získává data z *cachovací* databáze Redis (`#2`).



Nejprve spustíme Redis kontejner:

```
1 $ docker run -d --name redis-container-1 redis
2 8e642bae6471...3716
```

Parametr `-d` říká, že kontejner běží na pozadí v tzv. *detached* módu. To jednoduše ověříme příkazem `docker ps`, který vypíše všechny právě spuštěné kontejnery.

```
$ docker ps
CONTAINER ID   IMAGE ... PORTS      NAMES
8e642bae6471   redis ... 6379/tcp   redis-container-1
...
```

Vidíme, že kontejner skutečně běží. Důležitou informací je, že proces **uvnitř kontejneru** poslouchá na portu `6379/tcp`. Při vytváření vlastního image slouží ke specifikaci portu direktiva `EXPOSE`[15].

Nyní spustíme druhý kontejner, při spouštění však Dockeru řekneme, že má nový kontejner propojit s již běžícím.

Ukázka kódu 3.9 Síťové propojení dvou kontejnerů. Propojení zajistí parametr `--link`, který přijímá formát `SOURCE_CONTAINER_NAME:CONTAINER_ALIAS_NAME`. [46]

```
$ docker run -it  \
             --link redis-container-1:redis \
             --name client-container-2 busybox
```

Prozkoumáním konfiguračního souboru `/etc/hosts`[16] vidíme, jakým způsobem je síťové propojení kontejnerů vytvořeno.

```
/ # cat /etc/hosts
127.0.0.1       localhost
::1             localhost ip6-localhost ip6-loopback
fe00::0         ip6-localnet
ff00::0         ip6-mcastprefix
ff02::1         ip6-allnodes
ff02::2         ip6-allrouters
172.17.0.2      redis 8e642bae6471 redis-container-1
172.17.0.3      3afff37659be
```

[15] Direktiva `EXPOSE` sama o sobě nezveřejní port kontejneru pro své okolí, jen říká, že existuje nějaký proces, který za běhu poslouchá na nějakém portu.

[16] Soubor `/etc/hosts` říká systému, jak namapovat *hostnames* na IP adresy, princip je podobný jako při získávání IP adresy z DNS serveru.

Dále jednoduchým testem zjistíme, že síťové propojení do jiného kontejneru skutečně funguje

```
/ #  ping redis
PING redis (172.17.0.2): 56 data bytes
64 bytes from 172.17.0.2: seq=0 ttl=64 time=0.121 ms
64 bytes from 172.17.0.2: seq=1 ttl=64 time=0.074 ms
64 bytes from 172.17.0.2: seq=2 ttl=64 time=0.083 ms
64 bytes from 172.17.0.2: seq=3 ttl=64 time=0.074 ms
--- redis ping statistics ---
4 packets transmitted, 4 packets received,
    0% packet loss
round-trip min/avg/max = 0.074/0.088/0.121 ms
```

3.7.2 Propojení kontejneru s Docker host

Port kontejneru je možné vystavit i do Docker host, tedy do systému, na kterém běží Docker engine. Díky této funkci můžeme namapovat např. port `80`, na kterém běží HTTP, dovnitř kontejneru a z něj pak odbavovat požadavky webové aplikace. Rovněž je možné propojit nekontejnerovou aplikaci s nějakým kontejnerem. Síťové propojování pracuje na jednoduchém principu, avšak je velmi flexibilní a nabízí tak mnoho možných způsobů použití.

Demonstrujme vystavení kontejnerového portu do okolního prostředí na komplexním příkladu webového serveru. Jednotlivé kroky jsou následující:

1. Vytvoříme jednoduchý HTTP server, který bude poslouchat na určitém portu a odpovídat na požadavky.

2. Tento server zabalíme do *image* a spustíme jej jako kontejner.

3. Interní port kontejneru namapujeme pod určitým portem do Docker host.

4. V internetovém prohlížeči se připojíme na lokání URL s namapovaným portem, uvidíme *respose* z kontejneru.

Následuje kód HTTP serveru. Jedná se o velmi jednoduchou implementaci, program pouze naslouchá na portu, který získá z ENV. Jakmile zpracuje příchozí *request*, jako *respose* vrátí prostý text.

Ukázka kódu 3.10 Implementace HTTP serveru v Node.js za použití HTTP frameworku Express. Soubor uložíme jako `server.js`.

```
1  // - - - - - - - -
2  //  server.js
3  // - - - - - - - -
4  var express = require('express');
5
6  require('dotenv').config({silent: true});
7  var app = express();
8
9  app.set('port', (process.env.APP_PORT || 3000));
10
11 app.get('/', function (req, res) {
12     console.log("Hello from container");
13
14     res.send('Hello World!');
15 });
16
17 app.listen(app.get('port'), function () {
18     console.log('Node app is running on port',
19                 app.get('port'));
20 });
```

Jako další vytvoříme Dockerfile, který obsahuje kroky potřebné pro sestavení *image* pro aplikaci.

Ukázka kódu 3.11 Dockerfile testovací aplikace. Nejprve je vytvořen adresář pro kód aplikace a kontext je přesunut do něj. Do adresáře aplikace je zkopírován soubor `package.json`, který obsahuje Node.js závislosti aplikace – těmi jsou v tomto případě `express` a `dotenv`. Závislosti jsou nainstalovány a celý kód aplikace je zkopírován – kopírování kódu aplikace a souboru se závislostmi je rozděleno na dva kroky úmyslně – díky tomuto způsobu bude *cachování* pracovat efektivněji a nebudeme muset při každé změně zdrojového kódu instalovat závislosti znovu.

```
1  # Set the base image
2  FROM node:6
3
4  # Set the file maintainer
5  MAINTAINER John Doe
6
7  # Create app directory
8  RUN mkdir -p /usr/src/app
9  WORKDIR /usr/src/app
10
11 # Install app dependencies
12 COPY package.json /usr/src/app/
13 RUN npm install
14
15 # Bundle app source
16 COPY . /usr/src/app
17
18 ENV APP_PORT 8080
19
20 # Port to expose
21 EXPOSE 8080
22 CMD [ "node", "server.js" ]
```

Nyní dle připraveného Dockerfile sestavíme image s názvem „`http-test-image`“.

Ukázka kódu 3.12 Build testovacího image. Spustíme-li příkaz `docker images` uvidíme nově vytvořený a připravený image s názvem „`http-test-image`“.

```
1  $ docker build -t http-test-image .
2  Sending build context to Docker daemon  5.12 kB
3  Step 1 : FROM node:6
4   ---> 72d4ec634f1f
5
6  ...
7
8  Step 7 : COPY . /usr/src/app
9   ---> c3a3ba839b46
10 Removing intermediate container 1da72ef33731
11 Step 8 : ENV APP_PORT 8080
12  ---> Running in 23719e68d346
13  ---> a787e1338a55
14 Removing intermediate container 23719e68d346
15
16 ...
17
18 Removing intermediate container dd286055a18b
19 Successfully built 2b3dfe5b4efb
```

Konečně se dostáváme do bodu, kdy nový image spustíme jako kontejner a namapujeme interní port kontejneru `8080` do Docker host. Port na Docker host nemusí být stejný jako uvnitř kontejneru, Docker engine se postará o přepojení.

Ukázka kódu 3.13 Zveřejnění vnitřního portu kontejneru na Docker host. Proces v kontejneru poslouchá na portu `8080` na Docker host je kontejner přístupný přes port `8000`.

```
1 $ docker run -p 8000:8080 -it --rm http-test-image
2 Node app is running on port 8080
```

Nyní můžeme příkazem docker ps vypsat seznam běžících kontejnerů, ověřit konfiguraci a zjistit IP adresu, kterou má Docker engine přiřazenou.

```
1 $ docker ps
2 ... IMAGE              ... PORTS
3 ... http-test-image    ... 10.0.75.2:8000->8080/tcp
```

Otevřeme-li adresu `http://10.0.75.2:8000`[17] v prohlížeči, vidíme text „*Hello world*“ – odpověď z kontejneru.

3.8 Docker Compose

3.8.1 Představení Docker Compose

Spojování kontejnerů do jednoho aplikačního celku manuálně vystavuje celý systém riziku lidské chyby. Při spouštění vícekontejnerové aplikace musí programátor:

- rozhodnout, zda jsou všechny *images* aktuální či zda potřebují *rebuild*;
- spustit kontejnery ve správném pořadí;
- vystavit porty kontejnerů do systému;
- připojit ke kontejnerům správné data volume (containers);
- případně nasdílet data mezi hostem a kontejnerem;
- propojit mezi sebou závislé kontejnery.

[17] Může být matoucí, že IP adresa kontejneru není `localhost` nebo-li `127.0.0.1`, i když spouštíme kontejner na „lokálním stroji“. Ve výše uvedeném příkladě byl kontejner spuštěn na platformě Windows 10, kontejner tedy reálně běží na Linuxu, který má přidělenou určitou lokální adresu (viz obrázek Hlavní rozhraní nativní aplikace Docker na Windows 10 na straně 23). Docker umožňuje experimentálně vystavit porty in na `localhost`, tato funkce je však zatím nestabilní.

Výše uvedené kroky mají jednu společnou vlastnost – dají se relativně jednoduše zautomatizovat. Představíme poslední nástroj z Docker ekosystému, který je podrobně rozebrán v této práci.

Docker Compose nepřidává k Dockeru po technické stránce žádnou novou funkcionalitu a využívá všechny principy, které byly v předešlém textu popsány. Docker Compose pouze orchestruje[18] všechny součásti Dockeru pro snadné a rychlé spuštění vícekontejnerové aplikace. Všechny úkony, které Compose provádí, je v konečném důsledku možné rozebrat na jednotlivé Docker příkazy.

Základem Docker Compose je soubor `docker-compose.yml` ve formátu YAML (oficiální stránka projektu a referenční příručka k jazyku je dostupná na http://yaml.org/). V tomto konfiguračním souboru jsou definovány následující informace:

- jednotlivé images, které tvoří aplikaci;
- závislosti mezi kontejnery;
- zveřejněné porty;
- data volumes.

Ideální umístění souboru `docker-compose.yml` v projektu je vedle souboru `Dockerfile`. Adresářová struktura projektu může tedy vypadat následovně:

Struktura souborů v Docker projektu

```
/project
├── .dockerignore
├── .gitignore
├── docker-compose.yml
├── docker-compose-dev.yml
├── Dockerfile
├── Dockerfile-dev
├── ⋮
└── src/
    └── ⋮
```

V kapitole Propojení kontejneru s Docker host na straně 41 jsme v souboru Dockerfile použili direktivu `COPY`. Ta přebírá dva parametry: odkud a kam. Ve výše uvedeném příkladu jsme použili jako zdroj tečku (`COPY . /usr/src/app`), ta říká Docker enginu, aby použil pro zdrojový adresář aktuální kontext – kontextem je adresář, ve kterém se nachází

[18] Automatické řízení nějakého počítačového systému nebo procesu.

Dockerfile. Dále jsme v kapitole Vlastní images na straně 34 uvedli, že si Docker ukládá u každého zkopírovaného souboru *checksum* pro potřeby *cachování*. Dále pak soubor `.dockerignore`[19] říká, které soubory se mají při použití direktiv `COPY` nebo `ADD` ignorovat. Do souboru `.dockerignore` však není možné umístit samotný soubor Dockerfile. Pokud je tedy soubor Dockerfile součástí kopírovaných dat a my jej upravíme, invalidujeme cache pro všechna ostatní data. U menších projektů to nemusí být na obtíž, avšak pokud je objem kopírovaných dat větší, bude celý *build* zbytečně zpomalován. To je také důvod, proč je vhodné umístit zdrojové kódy projektu do samostatného podadresáře.

3.8.2 Vývojové a produkční prostředí

Bylo již zmíněno, že použití nástroje Docker by mělo vést při vývoji aplikace k minimalizaci rozdílů mezi vývojovým prostředím a prostředím, kde je aplikace reálně nasazená a kde ji používají její uživatelé[20]. Tuto praktiku doporučuje i metodika 12Factor. V praxi se tedy jedná např. o to, že při vývoji použijeme stejný databázový kontejner jako v produkčním prostředí.

Určitý rozdíl mezi vývojovým prostředím a produkčním prostředím však z podstaty věci existuje a vždy existovat bude – jedná se o způsob, jakým vývojář pracuje s kódem aplikace. Při vývoji programátor kód upravuje a chce vidět změny okamžitě, zatímco v ostatních prostředích už by měl být kód aplikace součástí kontejneru a ponechán jen ke spouštění.

V tomto směru nám nabízí Docker Compose možnost mít pro jeden projekt více `docker-compose-*.yml` souborů, které přepínačem dle prostředí používáme – vývojář tak může použít např. `docker-compose-dev.yml` pro lokální vývoj a pokud chce otestovat aplikace v produkčním prostředí (ale stále na svém lokálním stroji), přepínačem `--file FILENAME` použije produkční `docker-compose.yml`.

[19] Název souboru `.dockerignore` skutečně začíná tečkou, jeho význam a struktura jsou stejné jako u souboru `.gitignore` pro verzovací nástroj git. Ve zkratce řečeno obsahuje `.dockerignore` na každém řádku nějaký *pattern* souboru nebo adresáře, který má být ignorován.

[20] Prostředí může být mnohem více než jen vývojové a produkční. Dále se může jednat např. o testovací prostředí, průběžnou integraci (CI), *staging* atd.

Ukázka kódu 3.14 Ukázka Docker Compose konfiguračního souboru (`docker-compose-dev.yml`) pro lokální vývoj. Kontejner `app` mapuje na Docker host port `8000` a zdrojový kód aplikace je namapován dovnitř kontejneru přes Data volume. Soubor `Dockerfile-dev` **neobsahuje** direktivu `COPY` nebo `ADD`.

```
version: '2'
services:
  ...
  app:
    build:
      context: ./src
      dockerfile: Dockerfile-dev
    command: ...
    volumes:
      - .:/usr/src/app
    ports:
      - "8000:8000"
    depends_on:
      - ...
```

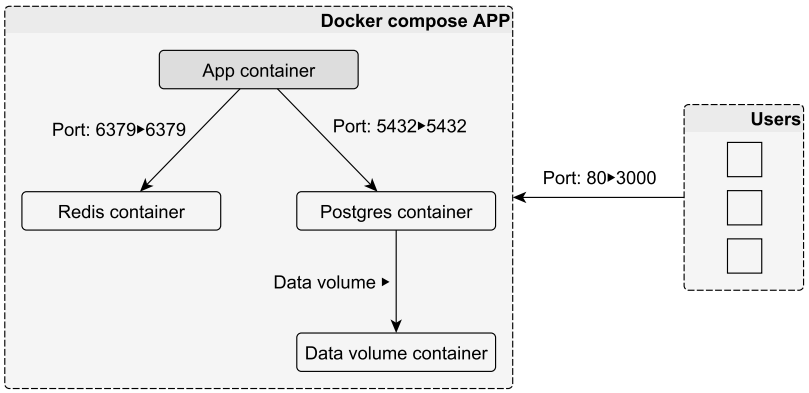
Ukázka kódu 3.15 Ukázka Docker Compose konfiguračního souboru (`docker-compose-production.yml`) pro produkční prostředí. Kontejner `app` mapuje na Docker host port `80`, na kterém typicky běží webové služby. Soubor `Dockerfile-production` **obsahuje** direktivu `COPY` ve tvaru: `COPY . /usr/src/app` – kód aplikace je tedy přímo součástí kontejneru.

```
version: '2'
services:
  ...
  app:
    build:
      context: ./src
      dockerfile: Dockerfile-production
    command: ...
    ports:
      - "80:8000"
    depends_on:
      - ...
```

3.8.3 Použití Docker Compose

Při práci s Docker Compose využíváme dvě součásti: I.) konfigurační soubor ve formátu YAML a II.) klient `docker-compose` ovládaný přes CLI.

Na obrázku Schéma vícekontejnerové aplikace na této straně je grafická reprezentace aplikace, kterou posléze vytvoříme v Docker Compose. Zdrojový kód aplikačního kontejneru je identický s kódem uvedeným v kapitole Propojení kontejneru s Docker host na straně 41.



Obrázek 3.12: Schéma vícekontejnerové aplikace. Aplikace se skládá ze čtyř kontejnerů, které jsou sestaveny do jednoho celku pomocí Docker compose.

Následuje stejné schéma definované v souboru `docker-compose.yml`. Tři kontejnery (`redis`, `data` a `postgres`) používají nezměněné base images, nepotřebujeme je nijak upravovat a tudíž není potřeba pro ně vytvářet vlastní Dockerfile. Čtvrtý kontejner `app` obsahuje naši aplikaci s již známým obsahem Dockerfile. Novou konfigurační volbou jsou tzv. *restart policies*, které Docker enginu říkají, za jakých okolností mají být jednotlivé kontejnery restartovány. [47]

Policy	**Význam**
`no`	Pokud se proces uvnitř kontejneru zastaví, kontejner znovu nenastartuje. Toto je výchozí chování.
`on-failure[:max-retries]`	Restartuje kontejner pokud se proces uvnitř kontejneru zastaví na nějakém nenulovém stavu. Volitelně je možné omezit počet pokusů na zotavení.
`always`	Docker v tomto módu ignoruje návratový kód procesu a kontejner spustí znovu. Kontejner také nastartuje při startu Docker démona – typicky při startu celého systému.
`unless-stopped`	Stejné jako volba `always`. Pokud však uživatel kontejner cíleně zastavil příkazem `docker stop`, Docker démon se jej nepokusí znovu spustit.

Tabulka 3.1: Přehled *restart policies* pro kontejnery. Volbu je možné předat parametrem `--restart POLICY` a to buď při spouštění kontejneru pomocí `docker run` anebo v souboru `docker-compose`. [47]

Ukázka kódu 3.16 Definice vícekontejnerové aplikace v `docker-compose.yml`.

```
version: '2'
services:
  redis:
    restart: "no"
    image: redis:latest
  data:
    restart: "no"
    image: busybox:latest
    volumes:
      - /var/lib/postgresql
    command: "true"
  postgres:
    restart: "yes"
    image: postgres:9.6
    volumes_from:
      - data
  app:
    build:
      context: .
      dockerfile: Dockerfile
    ports:
      - "80:3000"
    depends_on:
      - redis
      - postgres
```

Ve složce projektu nyní můžeme vytvořit všechny images příkazem `docker-compose build`:

```
$ docker-compose build
data uses an image, skipping
redis uses an image, skipping
postgres uses an image, skipping
Building app
Step 1 : FROM node:6
 ---> 72d4ec634f1f

...

Step 10 : CMD node server.js
 ---> Using cache
 ---> 44123bebac37
Successfully built 44123bebac37
```

Jako další krok spustíme celou vícekontejnerovou aplikaci příkazem `docker compose up`:

```
$ docker-compose up
Starting httptestcompose_redis_1
Starting httptestcompose_data_1
Starting httptestcompose_postgres_1
Starting httptestcompose_app_1
Attaching to httptestcompose_redis_1, \
             httptestcompose_data_1,\
             httptestcompose_postgres_1, \
             httptestcompose_app_1
...
```

Příkazem `docker-compose ps` jednoduše ověříme, že právě běží tři kontejnery[21], které jsou mezi sebou správně propojeny porty a data volumes jsou připojeny do kontejnerů.

[21] Skutečně běží jen tři kontejnery, i když je aplikace sestavená ze čtyř kontejnerů. Účelem Data volume kontejneru není spustit žádný proces, ale jen vytvořit Data volume pro uložení databázových dat. Kontejner se tedy po spuštění ihned správně vypne (návratový kód je `0`), ale jím vytvořené Data volume existuje i nadále. Nastavení `command: "true"` u kontejneru `data` není povinné, respektuje jen konvenci, že každý kontejner by měl mít definovaný nějaký příkaz, který Docker spustí.

Ukázka kódu 3.17 Výstup příkazu `docker-compose ps`.

```
$ docker-compose ps
Name                ... State    Ports
-------------------------------------------------------
ht..._app_1         ... Up      10.0.75.2:8080->3000/tcp
ht..._data_1        ... Exit 0
ht..._postgres_1    ... Up
ht..._redis_1       ... Up
```

Kromě úspory času při práci s kontejnery se Docker Compose postará i o spuštění jednotlivých kontejnerů ve správném pořadí dle závislostí mezi nimi. Aplikaci sestavenou z více kontejnerů pak ovládáme jako celek, při jejím spouštění pak Docker Compose určuje, které kontejnery jsou zastaralé a potřebují znovu sestavit.

3.9 Škálování

Okolo nástroje Docker vznikla celá řada open-source i uzavřených služeb a nástrojů, které se zaměřují na spouštění kontejnerů v produkčním prostředí, jejich orchestraci a škálování. Zmiňme zejména open-source nástroj Kubernetes (http://kubernetes.io/) od společnosti Google je v podstatě samostatná platforma pro provoz rozsáhlých kontejnerových clusterů. Kubernetes je vyspělý a robustní nástroj, uplatnění však nalezne ve středních a větších projektech.

V kapitole Širší pohled na kontejnerovou architekturu na straně 16 jsme zmiňovali CoreOS systém pro provoz kontejnerových aplikací. I tento systém má integrovanou podporu škálování pomocí nástroje Fleet (https://coreos.com/fleet/), mezi jehož hlavní přednosti patří tzv. *high availability*.

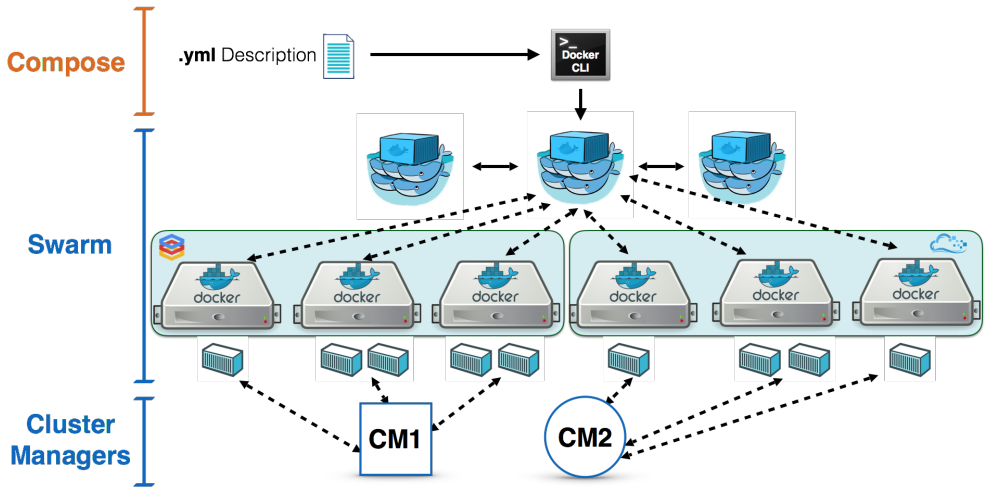
Pro potřeby škálování je však možné využít oficiální nástroje, které jsou do Dockeru integrovány. Aktuální doporučovaná cesta pro škálování je použít Docker Swarm – nativní nástroj pro clusterování Docker kotejnerů. [48]

Docker Swarm kombinuje dohromady již dříve představené nástroje.

- Docker Machine slouží k vytvoření Docker Engine na nějakém vzdáleném stroji nebo u poskytovatele cloudových služeb.
- Docker Compose slouží k definici a spuštění vícekontejnerové aplikace.
- A nakonec Docker Swarm umožňuje spojit více Docker Engine do jediného virtuálního hosta, na kterém je možné spouštět kontejnery.

Postup při provozování Docker Swarm je následující. [48]

Jako první krok vytvoříme pomocí `docker-machine` jeden až více vzdálených Docker Engine. Ty můžeme vytvořit buď na fyzických strojích nebo na virtualizovaných cloudových službách. Následuje propojení všech Docker Engine do jednoho jediného virtuálního clusteru pomocí Docker Swarm. Nyní můžeme příkazem `docker-compose scale` škálovat naše služby a distribuovat zátěž pomocí load balanceru. V případě internetových služeb je doporučována a oficiálně podporována HAProxy (base image a dokumentace jsou na https://hub.docker.com/_/haproxy/), která obstarává load balancing HTTP a TCP aplikací.



Obrázek 3.13: Princip škálování Docker kontejnerů pomocí Docker Swarm. [49]

Reálná ukázka škálování zde popsané aplikace je dostupná v sekci Kontejnerová struktura aplikace a škálovatelnost na straně 56.

Kapitola 4

Implementace

Cílem této kapitoly je popsat a zdokumentovat implementaci aplikace „Administrace uživatelských účtů The Story of Creation“, která je jednou ze čtyř součástí projektu „The Story of Creation“.

Aplikace je realizována na straně klienta (*frontend*) i na straně serveru (*backend*) v jazyce JavaScript. Aplikace implementuje modulární kontejnerovou architekturu využitím nástroje Docker popsanou v kapitole Kontejnerová architektura na straně 15 a řídí se metodologií 12Factor popsanou v sekci Metodologie The Twelve-Factor App na straně 8.

4.1 Struktura projektu

Celý projekt „The Story of Creation“ se skládá ze čtyř samostatných součástí:

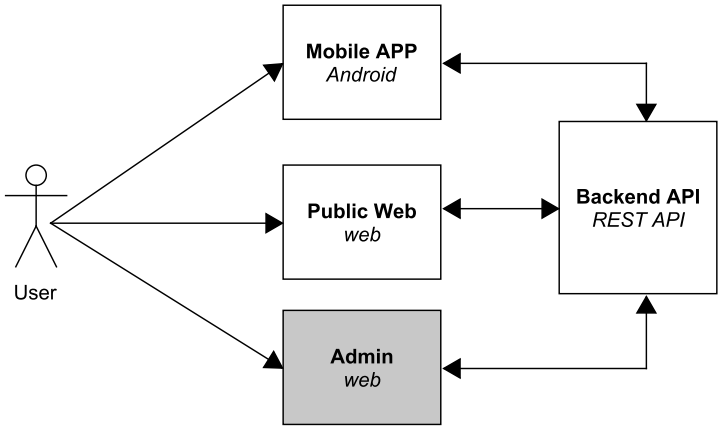
I.) mobilní aplikace pro systém Android;

II.) veřejného zákaznického webu pro prezentaci uživatelů;

III.) (předmět této práce)
webové aplikace pro administraci uživatelských účtů;

IV.) backend API pro správu všech dat.

Vztah mezi jednotlivými částmi jsou vyobrazeny na následujícím schématu. Významným prvkem je Backend API, které poskytuje všechna data v aplikaci.



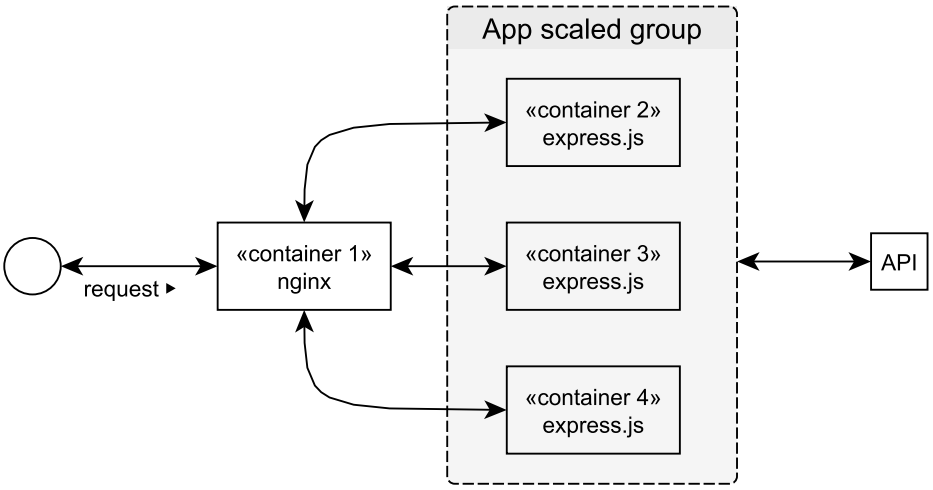
Obrázek 4.1: Čtyři hlavní části aplikace „The Story of Creation“. Předmětem této práce je webová administrační aplikace pro správu uživatelských účtů.

4.2 Kontejnerová struktura aplikace a škálovatelnost

Následuje kontejnerové schéma aplikace. Kontejner `<<container 1>> nginx` slouží k:

- odbavení requestu z internetu na portu `80`;
- v produčkním prostředí pak k odbavení TLS requestu z internetu na portu `443`;
- k load balancingu mezi aplikačními kontejnery.

Load balancing pracuje v režimu **least-connected** tedy request přesměrovává na ten kontejner, ke kterému je připojeno nejméně klientů. [50, 51] Celou aplikace pak ovládáme jako celek pomocí Docker Compose.



Obrázek 4.2: Kontejnerové schéma aplikace.

Ukázka kódu 4.1 Relevantní sekce konfigurace serveru nginx pro load balancing. [50, 51]

```
worker_processes 4;

events {
  worker_connections 1024;
}

http {

  upstream load-balanced-node-app {
    least_conn;

    server express-container-1:8080 weight=10 \
                                    max_fails=3 \
                                    fail_timeout=30s;

    server express-container-2:8080 weight=10 \
                                    max_fails=3 \
                                    fail_timeout=30s;

    server express-container-3:8080 weight=10 \
                                    max_fails=3 \
                                    fail_timeout=30s;

    # ...
  }

  server {
    listen 80;

    location / {
      proxy_pass http://load-balanced-node-app;
      proxy_http_version 1.1;
      proxy_set_header Upgrade $http_upgrade;

      # ...
    }
  }
}
```

4.3 Prostředí Node.js pro backend

4.3.1 Představení Node.js

Node.js je open-source[22] *runtime* prostředí pro jazyk JavaScript[23]. Jádro využívá V8 Engine od společnosti Google, které je použito na příklad ve webovém prohlížeči Google Chrome nebo v NoSQL databázi MongoDB. [52] Node.js využívá *event-driven* architekturu a neblokující (asynchronní) I/O, výborně se tak hodí pro vývoj webových aplikací a zejména pak API. [53]

Node.js je často mylně označováno jako framework či samostatný jazyk. Důvod těchto záměn je pravděpodobně způsoben chybějící standardní knihovnou v jazyce JavaScript, kterou však Node.js nabízí. [54, 55] V dalším textu bude o Node.js referováno jako o (běhovém/runtime) prostředí.

Důvodem pro výběr prostředí Node.js a jazyka JavaScript k implementaci aplikace byla již funkční implementace veřejného webu v témže prostředí. Ačkoliv v tuto chvíli spolu webová a administrační část nijak nekomunikují, tato situace se může v budoucnu změnit, a pak je výhodné mít vše v jednotném jazyce a prostředí.

4.3.2 Express framework

Jako open-source[24] framework pro práci s *request–response* byl zvolen Express.js (http://expressjs.com/) běžící v Node.js. Filozofií Express.js je poskytnout pouze robustní nástroj pro vytvoření HTTP serveru a vše ostatní ponechat na knihovnách třetích stran, z tohoto důvodu **nenajdeme** ve frameworku podporu databáze, autentizaci&autorizaci, šablonovací systém a mnohé další funkce, které jiné webové frameworky poskytují. [56] Faktem je, že tento minimalistický přístup frameworku ideálně sedí na požadavky aplikace – veškerý aplikační kód je vykonáván na straně klienta (frontend) a všechna data pocházejí ze vzdáleného REST API.

4.3.3 Šablonovací systém

Jediná funkcionalita, kterou potřebujeme navíc jako modul třetí strany do frameworku Express.js, je šablonovací systém. Ačkoliv by bylo možné distribuovat kód aplikace přímo z Express.js jako statický kód, díky šablonovacímu systému můžeme do kódu dodatečně dodat např. konfigurační proměnné ze serveru. Díky projektu Consolidate.js[25], který normalizuje

[22] Node.js je vyvíjeno pod licencí MIT (Massachusetts Institute of Technology), jejíž znění je dostupné na https://opensource.org/licenses/MIT. Zvolená licence umožňuje komerční užití, modifikace a redistribuci software, její zjednodušený výklad je na https://tldrlegal.com/license/mit-license.

[23] Korektně řečeno se jedná o implementaci standardizovaného jazyka ECMAScript 2015 (ES6). Tyto pojmy jsou však často zaměňovány.

[24] Express.js je vyvíjen stejně jako Node.js pod licencí MIT.

[25] Kód projektu Consolidate.js je dostupný na https://github.com/tj/consolidate.js.

interface desítek šablonovacích systému, je nabídka opravdu široká. Vzhledem k vyspělosti a stabilitě projektu byl vybrán open-source[26] šablonovací systém EJS (zkratka pro Effective JavaScript templating). [57]

Aktivace šablonovacího systému ve frameworku je velmi jednoduchá. Po instalaci správcem závislostí (`npm install ejs --save`) jej stačí jen aktivovat, viz následující kód.

Ukázka kódu 4.2 Použití EJS v Express.js.

```
1  var express = require('express');
2  var ejs = require('ejs');
3
4  // ...
5
6  var app = express();
7
8  // Setup template engine
9  app.engine('html', ejs.renderFile);
10
11 // ...
```

4.4 Frontend framework

Klientská část aplikace (frontend) je realizována modelem SPA – Single-page application. Tento model v praxi znamená, že první request od uživatele zpracuje aplikační server Express.js, který klientovi vrátí kompletní kód aplikace. Od tohoto okamžiku a dále neprobíhá kompletní request–response cyklus, avšak aplikace AJAXem už jen pracuje s daty, která uživateli vytváří/vyžaduje. Přístup má pozitivní vliv na chování celé aplikace, která má svižnější odezvu a v mnohém se s ní pracuje podobným způsobem jako s tradiční desktop aplikací.

4.4.1 Současný stav `MV*` frameworků

Aktuální stav ve světě frontend JavaScript nástrojů/knihoven/frameworků je možné popsat slovy chaotický a nepřehledný. [58, 59] Tato situace je dána do jisté míry ohromným rozvojem webových technologií v posledních letech. S rostoucími požadavky na webové aplikace logicky roste poptávka mezi vývojáři

[26] Šablonovací systém EJS je vyvíjen pod licencí Apache 2.0 (APLv2), jejíž znění je dostupné na http://www.apache.org/licenses/LICENSE-2.0. Zvolená licence umožňuje komerční užití, modifikace a redistribuci software, její zjednodušený výklad je na https://tldrlegal.com/license/apache-license-2.0-(apache-2.0).

po robustních a sofistikovaných nástrojích, které jim umožní vývoj moderních aplikací.

V posledních letech tedy vzniklo značné množství MVC (Model-View-Controller), MV (Model-View), MVP (Model-View-Presenter) a MVVM (Model-View ViewModel) frameworků – všechny tyto zkratky vycházejí z architektury MVC (**Model** definuje data aplikace, **View** typicky obstarává uživatelské rozhraní a **Controller** zpracovává vstup od uživatele a upravuje Model). Ne všechny frameworky obsahují všechny vrstvy z této architektury, a proto jsou souhrnně označovány jako `MV*`. [59]

Velké množství frameworků mapuje i zajímavý projekt http://todomvc.com/, na kterém je možné prozkoumat zdrojový kód jediné aplikace (jednoduchý nástroj pro zaznamenávání úkolů) implementované v desítkách různých `MV*` frameworků.

4.4.2 AngularJS

Pro implementaci zde popsané aplikace byl vybrán MVVM framework AngularJS od vývojářů společnosti Google, který byl poprvé představen v roce 2010. Hlavní stránka projektu včetně dokumentace se nachází na https://www.angularjs.org/. Ve srovnání s jinými frontend frameworky je AngularJS na trhu celých šest let (to je ve světě webových technologií relativně dlouhá doba) a za tu dobu se z něj stal uznávaný a široce používaný projekt i v *enterprise* světě. Okolo AngularJS vzniklo i několik zajímavých projektů, které jeho použití rozšiřují i nad rámec webových aplikací – zmiňme na příklad framework Ionic (hlavní stránka projektu je na http://ionicframework.com/), díky kterému je možné vytvářet hybridní HTML5 aplikace, které běží nativně na mobilních zařízení. Aktuálně existují dvě hlavní verze frameworku: `1.x` a `2.x`. Druhá verze frameworku je stále ve fázi vývoje, smyslem nové verze je mimo jiné připravit nástroj a styl vývoje aplikací pro budoucnost. Z tohoto důvodu je nová verze zpětně nekompatibilní s verzí předchozí – jedná se v podstatě o dva samostatné frameworky. Verze `1.x` je však nadále aktivně vyvíjena, oficiálně podporovaná a s jejím vývojem a podporou se počítá i v budoucnu. [60]

Zmiňme některé klíčové vlastnosti a funkce frameworku AngularJS.

- AngularJS používá tzv. „**two-way data binding**“, díky kterému se v aplikaci nachází jediná instance dat. Data mohou být následně aktualizována na různých místech aplikace a framework se postará o propagaci změn napříč kódem. Díky této funkci je nemusí programátor ručně kontrolovat, zda uživatel neupravil data v UI.
- AngularJS obsahuje vlastní **šablonovací systém**, který staví na již existujícím HTML standardu. Šablonovací systém tak jen přidává

do HTML nové konstrukce a klíčová slova.

- AngularJS má vestavěný systém pro tzv. ***dependecy injection***, tedy správu závislostí mezi jednotlivými moduly aplikačního kódu. Díky tomuto přístupu může programátor držet kód aplikace v mnoha drobných modulech, které je možné snáze testovat.

- Mezi další vlastnosti patří podpora routování (tj. mapování URL na kód aplikace), cachování, abstrakce nad REST API, *unit* a *e2e* (end-to-end) testování, ...

Poslední avšak neméně důležitou vlastností frameworku je ekosystém a komunita okolo něj. Do otevřeného kódu frameworku přispělo či přispívá více jak 1 400 vývojářů [61], díky velmi aktivní komunitě dále existuje mnoho modulů třetích stran – na příklad web http://ngmodules.org/ jich registruje bezmála dva tisíce.

4.5 Testování v AngularJS

Důležitou vlastností frameworku AngularJS je podpora vývojáře při psaní testovatelného kódu. AngularJS byl od počátku navržen tak, aby aplikace v něm byla jednoduše testovatelná – to je dáno zejména modulárním návrhem frameworku, kdy je možné při testování pomocí *dependency injection* předávat komponenty, která abstrahují např. API, prohlížeč atd.

Dependency injection je *software design pattern*, který umožňuje tvořit závislosti mezi komponentami tak, aby na sebe neměly referenci. Framework AngularJS se pak postará o to, aby každá komponenta dostala své závislost v momentě, kdy je skutečně potřebuje. [62]

Frontend frameworky a knihovny bývají často jednostranně zaměřeny na uživatelské rozhraní a tedy na to, co uživatel vidí v prohlížeči. Testování pak bývá omezeno na DOM (Document Object Model) a je tak redukováno na simulaci chování uživatele v aplikaci (k tomuto účelu je používán oblíbený nástroj Selenium – http://www.seleniumhq.org/). AngularJS se neomezuje jen na uživatelské rozhraní aplikace, ale i na data v aplikaci, komunikaci s API atd., při testování může být celý DOM abstrahován a vývojář se tak může zaměřit na testování logiky aplikace.

4.5.1 Unit Testování

Jednotkové (unit) testování je základní automatické testování pro ověření správné implementace nějaké samostatné komponenty aplikace, kterou je možné samostatně spustit a tedy i otestovat. V kontextu frameworku AngularJS je možné jednotkově otestovat direktivy (Directives), filtry (Filters), komponenty (Components), Controllers a další. [63]

AngularJS v jádru poskytuje tři hlavní nástroje pro jednotkové testování aplikace:

- **Karma** (http://karma-runner.github.io/) je tzv. „test runner“, který načte kód aplikace a spustí testy v jednom či několika definovaných webových prohlížečích.
- **Jasmine** (http://jasmine.github.io/) je samostatný BDD framework (Behavior-driven development), ve kterém jsou testy reálně definované. Jasmine abstrahuje testy od DOM a je primárně podporovaný nástrojem Karma. Alternativou k Jasmine pro testování aplikace jsou nástroje MochaJS nebo QUnit, oba plně podporované ze strany nástroje Karma.
- **ngMock** slouží k simulování nebo odstínění závislostí při testování.

Pro úplnost zmiňme, že nástroje Karma a Jasmine nejsou omezené jen pro použití ve spojení s AngularJS, ale je možné je použít pro testování i v jiných frameworcích a knihovnách.

4.5.2 E2E testování

Druhou samostatnou kapitolou při testování AngularJS aplikace je tzv. E2E testování (end-to end tests), ačkoliv z jiných programovacích jazyků a frameworků je pravděpodobně známější pojem „**integrační testování**“.

Zatímco jednotkové testování slouží k otestování samostatných komponent, E2E testování ověřuje, zda tyto komponenty dohromady tvoří požadovaný funkční celek – tj. zda spolu jednotlivé komponenty a moduly správně komunikují, zda je aplikaci jako celek možné spustit atd. [64]

Pro E2E testování vytvořil tým AngularJS open-source Node.js nástroj Protractor (http://www.protractortest.org/), ten interně využívá nástroj Selenium. Výhodnou při psaný a údržbě E2E i unit testů je, že kód obou typů testů je možné psát v nástroji Jasmine.

4.6 Uživatelské rozhraní (UI) aplikace

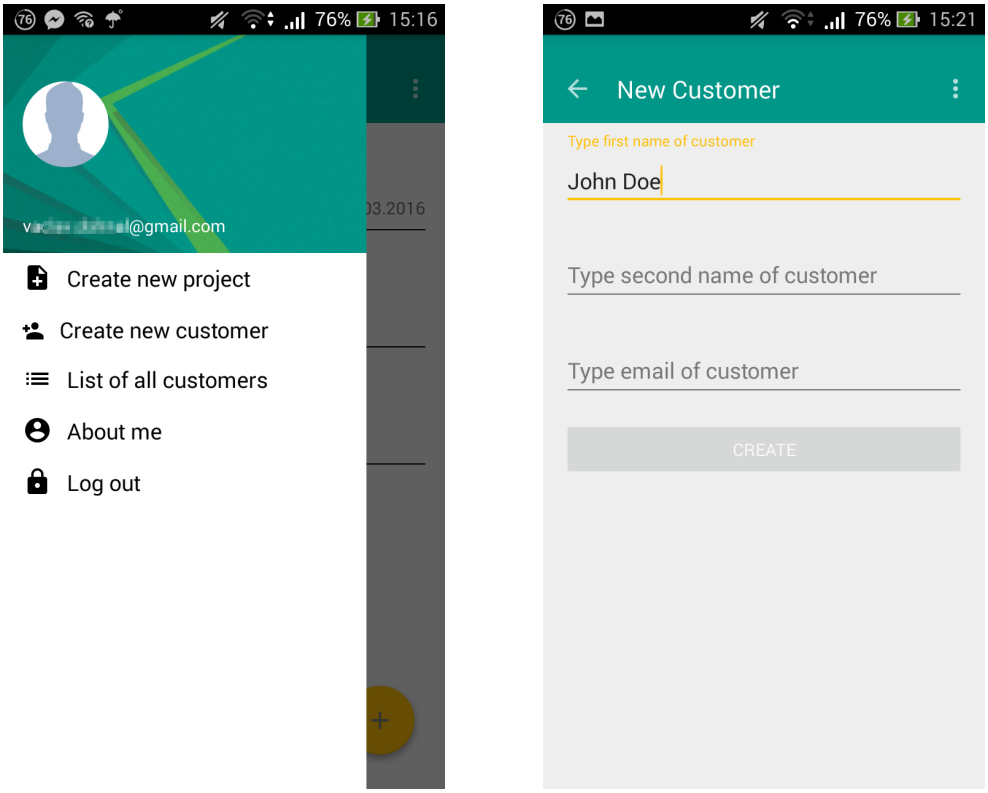
Jedním z důležitých faktorů, který rozhoduje o úspěchu aplikace mezi uživateli, je její uživatelské rozhraní (UI) a celkový uživatelský dojem a zážitek (UX, User experience). V této sekci objasníme kroky, které vedly k výběru grafického frameworku a návrhu uživatelského rozhraní.

4.6.1 Uživatelské rozhraní mobilní aplikace

Webová administrace uživatelských účtů, které se věnuje tato práce, navazuje na již existující mobilní aplikaci sloužící k podobnému účelu. Mobilní aplikace je implementována nativně na operačním systému Android, z kterého tedy

přebírá mnoho prvků typických pro daný systém – od ovládání (gesta, zkratky,...) po design (ikonky, animace, barvy,...)

Níže jsou předloženy snímky uživatelského rozhraní nativní mobilní aplikace.



Obrázek 4.3: Snímky obrazovky nativní mobilní aplikace na systému Android.

Při výběru grafického frameworku a návrhu uživatelského rozhraní je tedy nutné pro pohodlí uživatelů respektovat design a strukturu existující aplikace. Dále je však nutné pamatovat na rozdíly mezi ovládáním aplikace na mobilním zařízení a webu. Při ovládání webové aplikace má uživatel často (**ne však nutně vždy**) k dispozici myš, plnohodnotnou klávesnici, větší obrazovku oproti mobilnímu zařízení apod.

4.6.2 Google Material Design

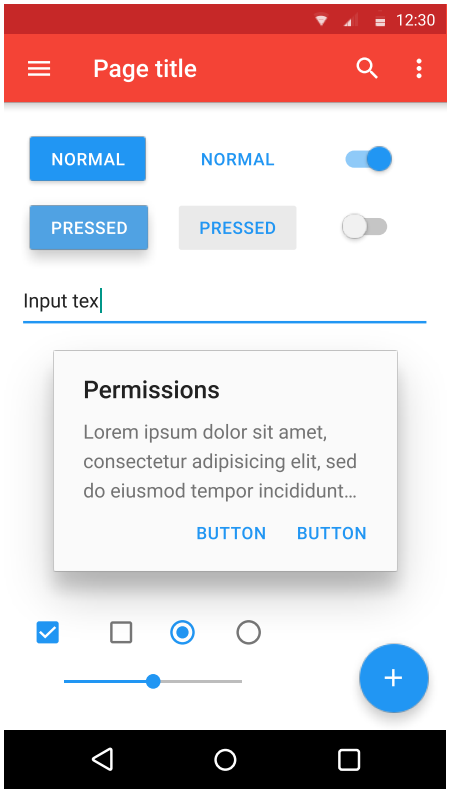
V roce 2014 na konferenci I/O představila společnost Google plán na sjednocení vizuální podoby systému Android, aplikací pro něj, webových aplikací a operačního systému Chrome OS. [65] Tímto sjednocujícím prvkem je „grafický jazyk“ Material Design. Jedná se o sadu doporučení,

pravidel a postupů, jak by mělo vypadat grafické rozhraní pro interakci s uživatelem.

Material Design komplexně řeší:

- principy *layoutu*, jeho jednotky a responsivitu (tj. přizpůsobení se velikost a rozlišení obrazovky);
- kombinace barev, ikonky, popisky ovládacích prvku, typografii;
- ovládací komponenty – tlačítka, navigační panely, dialogy atd.
- ...

Kompletní specifikace je dostupná na https://material.google.com/.



Obrázek 4.4: Některé z typických grafických komponent v Material Design. *Zdroj obrázku: [66]*

Mobilní aplikace „The Story of Creation“ tento grafický jazyk implementuje, je tedy žádoucí, aby stejná grafická doporučení respektovala i webová verze.

4.6.3 Výběr Google Material Design frameworku

Google Material Design je „pouze“ specifikace, která říká, jak má rozhraní vypadat, ale už neříká, jak specifikaci přenést do praxe. Existuje několik implementací specifikace Material Design, které se liší v kvalitě a cílovém použití. V kontextu této práce uvažujeme pouze o implementacích pro použití na webu.

Od společnosti Google existují tři oficiální implementace Material designu. [67]

- **Material Design Lite** nebo také MDL (https://getmdl.io/) je základní implementací bez externích závislostí. Nabízí strmou křivku učení, na druhé straně obsahuje jen malé množství komponent, které interagují s uživatelem. Material Design Lite je tak vhodný spíše pro menší projekty, které interagují s uživatelem málo či vůbec.
- Další implementací je **Polymer Project** dostupný na https://www.polymer-project.org. Polymer Project implementuje Material design jako „vedlejší efekt“. Hlavním cílem projektu je přenést do praxe relativně novou technologii WebComponents. Pro potřeby aplikace „The Story of Creation“ tedy není Polymer Project vhodný.
- Poslední oficiální implementací Material designu je **Angular Material** (https://material.angularjs.org/). Angular Material je plně závislý na frameworku AngularJS (pouze ve verzi `1.x`, podpora pro verzi `2.x` je ve vývoji). Nabízí velké množství komponent (v terminologii AngularJS se jedná o tzv. direktivy) a je plně připraven pro použití v produkčním prostředí.

Vzhledem k potřebě komplexního uživatelského rozhraní a využití frameworku AngularJS byla vybrána knihovna Angular Material.

Pro úplnost je nutné dodat, že existuje i celá řada neoficiálních implementací Material designu. Zmiňme některé zástupce:

- **Material-UI** (http://www.material-ui.com/) pro použití ve frameworku REACT.
- **Materialize** (http://materializecss.com/) pro obecné použití na webu.
- **Ionic Material** (http://ionicmaterial.com/) pro použití s frameworkem Ionic k tvorbě hybridních mobilních aplikací.

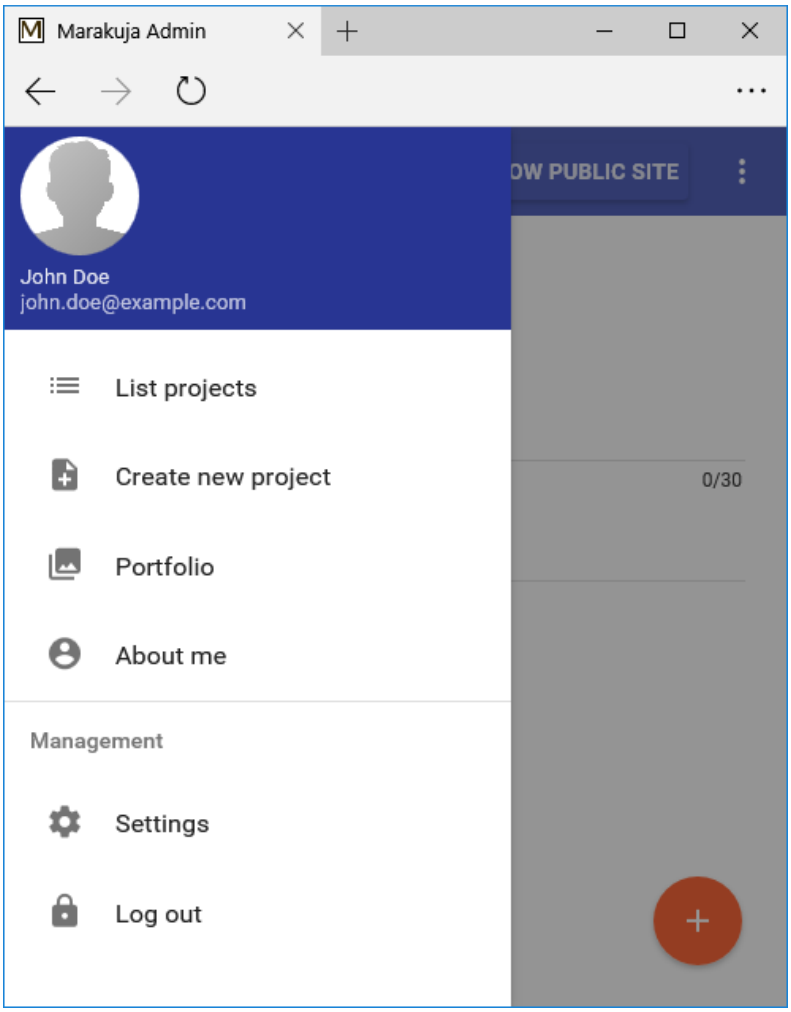
4.6.4 Angular Material

Hlavní síla Angular Material knihovny se nachází v tzv. direktivách, které jsou do existujícího projektu připojeny díky AngularJS dependency injection.

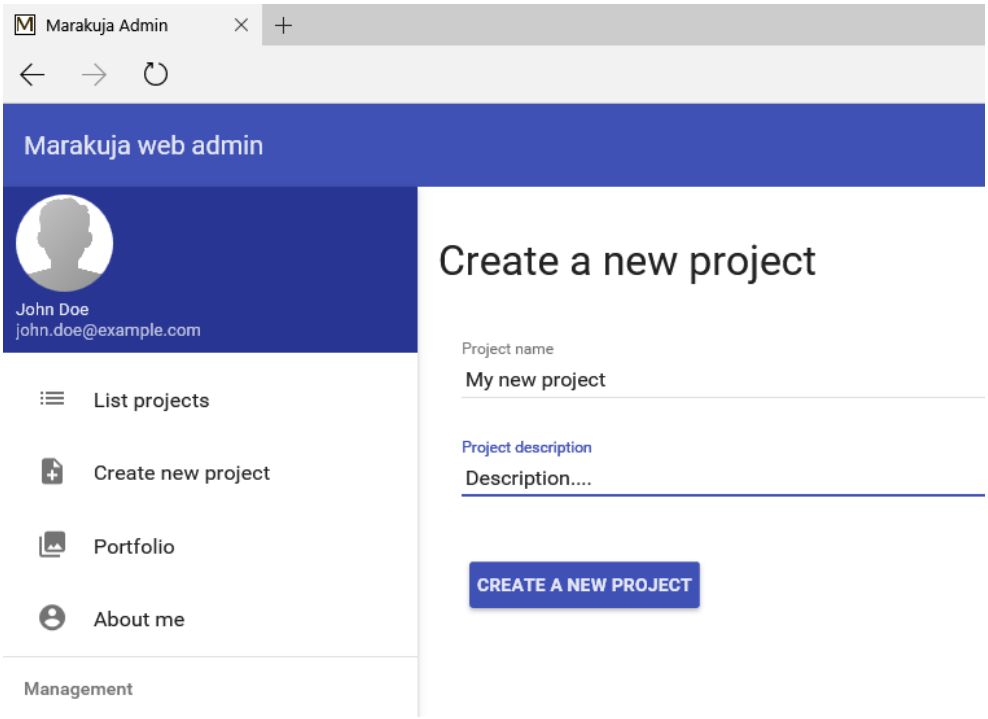
Celou knihovnu je možné rozdělit na:

- **vizuální direktivy**, které nějak interagují s uživatelem (tlačítka, formuláře, dialogy,...);
- **kontrolní direktivy**, které nejsou pro uživatele viditelné, ale zajišťují v aplikaci nějakou funkčnost (na příklad dotyková gesta)
- CSS **styly a témata** (*themes*) pro typografii, barvy, animace, ...
- **layout** – rozdělení aplikace do sloupců, řádků a oblastí a podpora různých rozlišení a velikostí obrazovky.

Následuje ukázka reálného rozhraní aplikace „The Story of Creation“ implementovaného v Angular Material knihovně.



Obrázek 4.5: Ukázka postranního menu aplikace „The Story of Creation“ v Material designu. Menu po grafické stránce respektuje mobilní aplikaci.



Obrázek 4.6: Ukázka formuláře v aplikaci „The Story of Creation“.

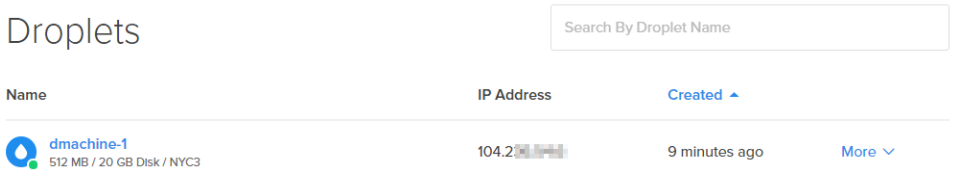
4.7 Zvolená PaaS DigitalOcean

Backend API používá pro svůj provoz v předchozím textu zmiňovanou PaaS Heroku (https://www.heroku.com/). Heroku interně používá kontejnerovou architekturu a umožňuje spouštění Docker kontejnerů. Docker podpora je však velmi omezená, Heroku na příklad neumožňuje orchestrovat aplikaci pomocí Docker Compose. Z těchto důvodů byl pro provozování aplikace vybrán poskytovatel cloudových služeb DigitalOcean (https://www.digitalocean.com/), který poskytuje plnou podporu Dockeru. Aplikaci budeme provozovat v režimu vzdálené Docker machine, více o tomto konceptu v sekci Škálování na straně 53.

Vzdálenou Docker machine vytvoříme příkazem `docker-machine create`.

Ukázka kódu 4.3 Vytvoření vzdálené Docker machine s názvem „dmachine-1“.

```
$ docker-machine create \
                --driver digitalocean \
                --digitalocean-access-token=cf...36 \
                dmachine-1
Running pre-create checks...
Creating machine...
...
Checking connection to Docker...
Docker is up and running!
```



Obrázek 4.7: Vzdáleně vytvořená Docker machine.

Po úspěšné instalaci je vhodné vzdálenou Docker machine upgradovat.

Ukázka kódu 4.4 Upgrade vzdálené Docker machine

```
docker-machine upgrade dmachine-1
```

Po úspěšném vytvoření vzdálené Docker machine musíme kontext všech Docker nástrojů přepnout na vzdálený stroj. Nové nastavení je možné vypsat příkazem `docker-machine env dmachine-1`.

Ukázka kódu 4.5 Vytvoření vzdálené Docker machine s názvem „dmachine-1“.

```
docker-machine create \
                --driver digitalocean \
                --digitalocean-access-token=cf...36 \
                dmachine-1
```

Nyní můžeme spustit aplikaci na vzdálené Docker machine stejně, jako kdyby se jednalo o lokální Docker machine, příkazem `docker-machine up`.

Závěr

Kontrola splnění zadání

Cílem této práce bylo analyzovat, navrhnout, implementovat a integrovat aplikaci pro komunikaci mezi drobnými podniky a jejich zákazníky s možností komunikace v reálném čase. Zvláštní důraz byl kladen na kvalitní a robustní návrh architektury a škálovatelnost.

Aplikace pro správu účtu a komunikaci se zákazníky je vytvořená jako webová aplikace v prostředí Node.js na backendu a v javascriptovém frameworku AngularJS na frontendu. O těchto dvou částech pojednává sekce Prostředí Node.js pro backend na straně 59 resp. Frontend framework na straně 60.

Real-time komunikace mezi Uživatelem a Zákazníkem (viz sekce Uživatelské role na straně 5) je realizována standardizovaným TCP protokolem Websockets, kterému je věnována sekce Protokol WebSocket a real-time komunikace na straně 11.

Stěžejní část této práce se věnuje modulární kontejnerové architektuře, které je podrobně rozebrána v samostatné kapitole Kontejnerová architektura na straně 15. Aplikace je reálně nasazená s použitím technologií Docker Engine, Docker Compose a Docker Machine na cloudové platformě DigitalOcean, o které pojednává sekce Zvolená PaaS DigitalOcean na straně 68. Kontejnery aplikace, které komunikují s Backend API, je možné pomocí Docker Compose horizontálně škálovat. Vyvažování zátěže (load balancing) zajišťuje nginx load balancer popsaný v sekci Kontejnerová struktura aplikace a škálovatelnost na straně 56. Škálování v prostředí kontejnerové architektury se pak věnuje sekce Škálování na straně 53.

Při návrhu a implementaci aplikace byla dodržována metodologie The Twelve-Factor App, jejíž doporučení relevantní pro tuto aplikaci jsou popsány v sekci Metodologie The Twelve-Factor App na straně 8.

Přínosy práce

Hlavním přínosem práce je zdokumentování a popsání procesu vývoje moderní webové aplikace od prvotního návrhu po výběr technologií až k samotné implementaci. Bylo potvrzeno, že webové aplikace na principu SaaS jsou funkční i technologickou alternativou k tradičním aplikacím. Můžeme očekávat, že komplexnost webových aplikací bude i nadále narůstat, je proto nezbytné, aby vývojáři vnímali návrh architektury a výběr platformy jako neoddělitelnou součást při procesu vývoje aplikace.

Další vývoj

Po funkční a obchodní stránce záleží budoucí vývoj aplikace na jejích uživatelích resp. na vlastnících tohoto projektu. Po stránce technické by bylo vhodné zautomatizovat celý vývojový proces od testování po nasazení aplikace např. vhodnou integrací s některou CI službou/platformou.

Literatura

[1] Wiggins, A.: About Adam Wiggins. červen 2016, [cit. 2016-06-10]. Dostupné z: http://about.adamwiggins.com/

[2] Wiggins, A.: The Twelve-Factor App – Introduction. červen 2016, [cit. 2016-06-10]. Dostupné z: https://www.heroku.com/what

[3] Wiggins, A.: II. Dependencies. *The Twelve-Factor App*, leden 2012, [cit. 2016-06-10]. Dostupné z: http://12factor.net/dependencies

[4] Wiggins, A.: III. Config. *The Twelve-Factor App*, leden 2012, [cit. 2016-06-10]. Dostupné z: http://12factor.net/config

[5] Wiggins, A.: IV. Backing services. *The Twelve-Factor App*, leden 2012, [cit. 2016-06-10]. Dostupné z: http://12factor.net/backing-services

[6] Wiggins, A.: VI. Processes. *The Twelve-Factor App*, leden 2012, [cit. 2016-06-10]. Dostupné z: http://12factor.net/processes

[7] Wiggins, A.: VII. Port binding. *The Twelve-Factor App*, leden 2012, [cit. 2016-06-10]. Dostupné z: http://12factor.net/port-binding

[8] Wiggins, A.: X. Dev/prod parity. *The Twelve-Factor App*, leden 2012, [cit. 2016-06-10]. Dostupné z: http://12factor.net/dev-prod-parity

[9] Garrett, J. J.: Ajax: A New Approach to Web Applications. *AdaptivePath.com*, únor 2005, [cit. 2016-06-17]. Dostupné z: https://web.archive.org/web/20080702075113/http://www.adaptivepath.com/ideas/essays/archives/000385.php

[10] Jaitla, J.: WebSockets vs REST: Understanding the Difference. *PubNub, Inc*, leden 2015, [cit. 2016-06-21]. Dostupné z: https://www.pubnub.com/blog/2015-01-05-websockets-vs-rest-api-understanding-the-difference/

[11] Fette, I.; Melnikov, A.: IETF RFC 6455: The WebSocket Protocol. Technická zpráva, IETF, Network Working Group, 2011, [cit. 2016-06-21]. Dostupné z: https://tools.ietf.org/html/rfc6455

[12] Deveria, A.; Schoors, L.: Can I Use Websockets. *Caniuse.com*, červen 2016, [cit. 2016-06-21]. Dostupné z: http://caniuse.com/#search=websockets

[13] West, M.: An Introduction to WebSockets. *Treehouse*, říjen 2013, [cit. 2016-06-21]. Dostupné z: http://blog.teamtreehouse.com/an-introduction-to-websockets

[14] Mozilla Developer Network and individual contributors.: Writing WebSocket servers. *Mozilla Developer Network*, červen 2016, [cit. 2016-06-21]. Dostupné z: https://developer.mozilla.org/en-US/docs/Web/API/WebSockets_API/Writing_WebSocket_servers

[15] Gorton, I.: *Essential Software Architecture*. Springer-Verlag GmbH, 2016, ISBN 3642191754. Dostupné z: http://www.ebook.de/de/product/14131572/ian_gorton_essential_software_architecture.html

[16] Corbet, J.: Notes from a container. *LWN.net*, říjen 2007, [cit. 2016-06-08]. Dostupné z: https://lwn.net/Articles/256389/

[17] Canonical Ltd.: LXD: the next-generation container hypervisor for Linux. *Ubuntu Cloud*, červen 2016, [cit. 2016-06-21]. Dostupné z: http://www.ubuntu.com/cloud/lxd

[18] OCI: About. *Open Container Initiative and Linux Foundation*, 2016, [cit. 2016-06-14]. Dostupné z: https://www.opencontainers.org/about

[19] OCI: Members. *Open Container Initiative and Linux Foundation*, 2016, [cit. 2016-06-14]. Dostupné z: https://www.opencontainers.org/about/members

[20] Bhartiya, S.: Open Container Initiative addresses Docker, CoreOS image problem. *CIO*, duben 2016, [cit. 2016-06-14]. Dostupné z: http://www.cio.com/article/3057579/open-source-tools/open-container-initiative-addresses-docker-coreos-image-problem.html

[21] AWS: Amazon EC2 Container Service – Docker Basics. *AWS Developer Guide*, 2016, [cit. 2016-06-08]. Dostupné z: https://docs.aws.amazon.com/AmazonECS/latest/developerguide/docker-basics.html

[22] Panettieri, J.: Cloud Market Share 2016: AWS, Microsoft, IBM, Google. *CHANNELe2e*, únor 2016, [cit. 2016-06-08].

Dostupné z: https://www.channele2e.com/2016/02/04/cloud-market-share-2016-aws-microsoft-ibm-google/

[23] Squillace, R.: Use Docker Machine with the Azure driver. *Microsoft Azure Documentation*, duben 2016, [cit. 2016-06-10]. Dostupné z: https://azure.microsoft.com/cs-cz/documentation/articles/virtual-machines-linux-docker-machine/

[24] Friis, M.: Introducing 'heroku docker:release': Build & Deploy Heroku Apps with Docker. *Heroku Blog*, červen 2016, [cit. 2016-06-10]. Dostupné z: https://blog.heroku.com/archives/2015/5/5/introducing_heroku_docker_release_build_deploy_heroku_apps_with_docker

[25] Google: Container Engine – Container Clusters. *Google Cloud Platform*, červen 2016, [cit. 2016-06-10]. Dostupné z: https://cloud.google.com/container-engine/docs/clusters/

[26] Kubernetes: What is Kubernetes? *Kubernetes Guides*, červen 2016, [cit. 2016-06-10]. Dostupné z: http://kubernetes.io/docs/whatisk8s/

[27] Docker: Using Supervisor with Docker. *Docker Docs*, 2016, [cit. 2016-06-12]. Dostupné z: https://docs.docker.com/engine/admin/using_supervisord/

[28] Czarkowski, P.: Multi Process Docker Images Done Right. prosinec 2014, [cit. 2016-06-12]. Dostupné z: http://tech.paulcz.net/2014/12/multi-process-docker-images-done-right/

[29] Swan, C.: Docker drops LXC as default execution environment. *InfoQ.com*, březen 2014, [cit. 2016-06-22]. Dostupné z: https://www.infoq.com/news/2014/03/docker_0_9

[30] Hykes, S.: Introducing runC: a lightweight universal container runtime. *The Docker Blog*, 2015, [cit. 2016-06-22]. Dostupné z: https://blog.docker.com/2015/06/runc/

[31] Docker: Docker Overview. *Docker Docs*, červen 2016, [cit. 2016-06-22]. Dostupné z: https://docs.docker.com/engine/understanding-docker/

[32] Chanezon, P.: Docker for Mac and Windows Beta: the simplest way to use Docker on your laptop. *The Docker Blog*, březen 2016, [cit. 2016-06-12]. Dostupné z: https://blog.docker.com/2016/03/docker-for-mac-windows-beta/

[33] Valoušek, M.: Lokální vývoj s Dockerem nebo Vagrantem? *Zdroják.cz*, červen 2016, [cit. 2016-06-10]. Dostupné z: https://www.zdrojak.cz/clanky/lokalni-vyvoj-s-dockerem-nebo-vagrantem/

[34] CHEF: docker Cookbook. *CHEF SUPERMARKET*, 2016, [cit. 2016-06-16]. Dostupné z: https://supermarket.chef.io/cookbooks/docker

[35] Docker: Select a storage driver. *Docker Docs*, červen 2016, [cit. 2016-06-22]. Dostupné z: https://docs.docker.com/engine/userguide/storagedriver/selectadriver/

[36] Natarajan, R.: Linux AuFS Examples: Another Union File System Tutorial (UnionFS Implementation). *The Geek Stuff*, květen 2013, [cit. 2016-06-22]. Dostupné z: http://www.thegeekstuff.com/2013/05/linux-aufs/

[37] Wright, C. P.; Zadok, E.: Kernel Korner - Unionfs: Bringing Filesystems Together. *Linux Journal*, prosinec 2004, [cit. 2016-06-17]. Dostupné z: https://www.linuxjournal.com/article/7714

[38] Eklund, D.: Visualizing Docker Containers and Images. říjen 2015, [cit. 2016-06-17]. Dostupné z: http://merrigrove.blogspot.com/2015/10/visualizing-docker-containers-and-images.html

[39] Gupta, R.: Why Docker Data Containers (Volumes!) are Good. listopad 2014, [cit. 2016-06-17]. Dostupné z: https://medium.com/@ramangupta/why-docker-data-containers-are-good-589b3c6c749e

[40] Docker: Manage data in containers. *Docker User Guide*, 2016, [cit. 2016-06-17]. Dostupné z: https://docs.docker.com/engine/userguide/containers/dockervolumes/

[41] Biederman, E. W.: PATCH review 19/19 vfs: Do not allow escaping from bind mounts. *The Linux Kernel Archives*, duben 2015, [cit. 2016-06-18]. Dostupné z: https://lists.linuxfoundation.org/pipermail/containers/2015-April/035788.html

[42] Eschinger, R.: Dockerized Postgresql Development Environment. březen 2015, [cit. 2016-06-18]. Dostupné z: http://ryaneschinger.com/blog/dockerized-postgresql-development-environment/

[43] Docker: Dockerfile reference. *Docker Docs*, červen 2016, [cit. 2016-06-19]. Dostupné z: https://docs.docker.com/engine/reference/builder/

[44] Minnihan, J.: Understanding the Docker Cache for Faster Builds. *The New Stack*, říjen 2014, [cit. 2016-06-19]. Dostupné z: http://thenewstack.io/understanding-the-docker-cache-for-faster-builds/

[45] DeHamer, B.: How to Maximize Your Docker Image Caching Techniques. *CenturyLink Cloud Developer Center*, srpen 2015, [cit. 2016-06-19]. Dostupné z: https://www.ctl.io/developers/blog/post/more-docker-image-cache-tips/

[46] Irani, R.: Docker Tutorial Series : Part 8 : Linking Containers. *Romin Irani's Blog*, červenec 2015, [cit. 2016-06-20]. Dostupné z: https://rominirani.com/docker-tutorial-series-part-8-linking-containers-69a4e5bf50fb

[47] Docker: Docker run reference – Restart policies (–restart). *Docker Docs*, 2016, [cit. 2016-06-21]. Dostupné z: https://docs.docker.com/engine/reference/run/

[48] Firschman, B.: Orchestrating Docker with Machine, Swarm and Compose. *The Docker Blog*, únor 2015, [cit. 2016-06-10]. Dostupné z: https://blog.docker.com/2015/02/orchestrating-docker-with-machine-swarm-and-compose/

[49] Alba, S.: Deploy and Manage Any Cluster Manager with Docker Swarm. *The Docker Blog*, listopad 2015, [cit. 2016-06-10]. Dostupné z: https://blog.docker.com/2015/11/deploy-manage-cluster-docker-swarm/

[50] Sankar, A. M.: A sample Docker workflow with Nginx, Node.js and Redis. březen 2015, [cit. 2016-06-25]. Dostupné z: http://anandmanisankar.com/posts/docker-container-nginx-node-redis-example/

[51] Nginx, Inc.: Using nginx as HTTP load balancer. 2016, [cit. 2016-06-25]. Dostupné z: http://nginx.org/en/docs/http/load_balancing.html

[52] Google: Chrome V8 – What is V8? *Google Developers*, 2016, [cit. 2016-06-21]. Dostupné z: https://developers.google.com/v8/

[53] Node.js Foundation: About Node.js. *Node.js Docs*, 2016, [cit. 2016-06-21]. Dostupné z: https://nodejs.org/en/about/

[54] Wen, B.: 6 things you should know about Node.js. *JavaWorld*, prosinec 2013, [cit. 2016-06-21]. Dostupné z: http://www.javaworld.com/article/2079190/scripting-jvm-languages/6-things-you-should-know-about-node-js.html

[55] Fuchs, T.: What if we had a great standard library in JavaScript? březen 2016, [cit. 2016-06-21]. Dostupné z: https://medium.com/@thomasfuchs/what-if-we-had-a-great-standard-library-in-javascript-52692342ee3f

[56] Express; Node.js Foundation: Express FAQ. 2016, [cit. 2016-06-22]. Dostupné z: http://expressjs.com/en/starter/faq.html

[57] Sevilleja, C.: Use EJS to Template Your Node Application. *Scotch*, červen 2014, [cit. 2016-06-22]. Dostupné z: https://scotch.io/tutorials/use-ejs-to-template-your-node-application

[58] Breck-McKye, J.: The State of JavaScript in 2015. prosinec 2014, [cit. 2016-06-22]. Dostupné z: http://www.breck-mckye.com/blog/2014/12/the-state-of-javascript-in-2015/

[59] Osmani, A.: Journey Through The JavaScript MVC Jungle. *Smashing Magazine*, červen 2012, [cit. 2016-06-22]. Dostupné z: https://www.smashingmagazine.com/2012/07/journey-through-the-javascript-mvc-jungle/

[60] Iffland, D.: Angular 2.0 Concerns Addressed at ng-conf 2015. *InfoQ.com*, 2015, [cit. 2016-06-22]. Dostupné z: https://www.infoq.com/news/2015/03/angular-2-concerns-addressed

[61] GitHub, Inc.: Contributors to angular/angular.js. *Github*, červen 2016, [cit. 2016-06-22]. Dostupné z: https://github.com/angular/angular.js/graphs/contributors

[62] Sankar, A. M.: AngularJS Dependency Injection - Demystified. září 2014, [cit. 2016-06-25]. Dostupné z: http://anandmanisankar.com/posts/angularjs-dependency-injection-demystified/

[63] Google: Unit Testing. *AngularJS: Developer Guide*, červen 2016, [cit. 2016-06-25]. Dostupné z: https://docs.angularjs.org/guide/unit-testing

[64] Google: E2E Testing. *AngularJS: Developer Guide*, červen 2016, [cit. 2016-06-25]. Dostupné z: https://docs.angularjs.org/guide/e2e-testing

[65] Brian, M.: Google's new 'Material Design' UI coming to Android, Chrome OS and the web. *Engadget*, červen 2014, [cit. 2016-06-22]. Dostupné z: https://www.engadget.com/2014/06/25/googles-new-design-language-is-called-material-design/

[66] Wikipedia: Material Design — Wikipedia, The Free Encyclopedia. 2016, [cit. 2016-06-24]. Dostupné z: https://en.wikipedia.org/w/index.php?title=Material_Design&oldid=724136285

[67] Joubran, J.: Angular Material vs. Material Design Lite. *Scotch*, září 2015, [cit. 2016-06-24]. Dostupné z: https://scotch.io/bar-talk/angular-material-vs-material-design-lite

Příloha **A**

Seznam použitých zkratek

API Application Programming Interface

AJAX Asynchronous JavaScript and XML

BDD Behaviour-driven development

CRM Customer relationship management

CLI Command Line Interface

DNS Domain Name System

DevOps Složenina anglických výrazů Development (vývoj) a IT Operations (provoz software jako byznys)

ER Entity-relationship

FS Filesystem

GUI Graphical user interface

I/O Input/Output

LXC Linux Containers

OCI Open Container Initiative

PaaS Platform as a Service

REST Representational State Transfer

SPA Single-page application

SaaS Software as a Service

TLS Transport Layer Security

TCP Transmission Control Protocol

UI User interface

UX User experience

URL Uniform Resource Locator

VPS Virtual Private Server

VMM Virtual Machine Manager

VPN Virtual Private Network

Příloha **B**

Obsah přiloženého CD

- `/`
  - `README.txt/` ..... Obsah přiloženého CD
  - `thesis/`
    - `images/` ..... Data obrázků pro LyX a LaTeX
      - ⋮
    - `thesis.LYX` ..... Text práce ve formátu LyX
    - `thesis.LATEX` ..... Text práce ve formátu LaTeX
    - `thesis.PDF` ..... Text práce ve formátu PDF
  - `docker/`
    - `http-test-simple/` ..... Demonstrace práce s Dockerfile
      - ⋮
    - `http-test-compose/` ..... Demonstrace práce s Docker Compose
      - ⋮
    - `scaling-example/` ..... Ukázka škálování kontejnerů.
      - ⋮