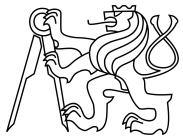
ČESKÉ VYSOKÉ UČENÍ TECHNICKÉ V PRAZE
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ



ZADÁNÍ DIPLOMOVÉ PRÁCE

Název:	Zajištění škálovatelnosti webových aplikací s využitím architektury mikroslužeb
Student:	Bc. Lukáš Hamrla
Vedoucí:	Ing. Jaroslav Kuchař, Ph.D.
Studijní program:	Informatika
Studijní obor:	Webové a softwarové inženýrství
Katedra:	Katedra softwarového inženýrství
Platnost zadání:	Do konce letního semestru 2017/18

Pokyny pro vypracování

Cílem práce je prozkoumat možnosti zajištění vysoké dostupnosti a škálovatelnosti webových aplikací s využitím konceptu mikroslužeb a souvisejících nástrojů. Zaměřte se zejména na situace, kdy je potřeba zajistit škálovatelnost pouze částí aplikací. Požadavky na práci:
- Popište základní možnosti pro řešení škálovatelnosti a vysoké dostupnosti webových aplikací.
- Seznamte se a detailně rozeberte architekturu založenou na mikroslužbách.
- Seznamte se a detailně rozeberte kontejnerový manažer Docker a jeho využití s architekturou mikroslužeb.
- Navrhněte a implementujte ukázkovou webovou aplikaci, která bude využívat kombinace těchto dvou technologií.
- Rozeberte a demonstrujte možnosti škálovatelnosti a zajištění vysoké dostupnosti na ukázkové aplikaci včetně výhod a nevýhod.
- Popište doporučení a důsledky pro praktická a reálná nasazení.

Seznam odborné literatury

Dodá vedoucí práce.

Ing. Michal Valenta, Ph.D.
vedoucí katedry

prof. Ing. Pavel Tvrdík, CSc.
děkan

V Praze dne 3. ledna 2017

Poděkování

Děkuji panu Ing. Jaroslavu Kuchařovi, Ph. D. za cenné rady a důležité připomínky při vedení této práce, stejně tak všem, kteří mě při tvorbě této práce podporovali.

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 4. května 2017 .....

Odkaz na tuto práci

Hamrla, Lukáš. *Zajištění škálovatelnosti webových aplikací s využitím architektury mikroslužeb.* Diplomová práce. Praha: České vysoké učení technické v Praze, Fakulta informačních technologií, 2017.

Abstrakt

Tato práce se věnuje zajištění škálovatelnosti a vysoké dostupnosti webových aplikací, tuto problematiku řeší zejména pro části aplikací. Pro dosažení tohoto cíle je použita architektura založená na mikroslužbách a kontejnerová platforma Docker. Obě tyto technologie jsou v textu detailně popsány a rozebrány. Zároveň práce obsahuje také ukázkovou aplikaci využívající tyto a jim příbuzné a podpůrné nástroje. Kromě samotné aplikace jsou zde popsány důsledky, doporučené postupy, možné problémy a jejich řešení vyplývající z použití výše zmíněných technologií.

Klíčová slova škálovatelnost webových aplikací, vysoká dostupnost webových aplikací, mikroslužby, Docker, Rancher, cluster

Abstract

This thesis is focused on web applications scalability and high availability, especially for applications parts. Microservices architecture and container platform Docker are used to achieve this goal. There is a detailed analysis of these two technologies also. The thesis contains demo application which uses Docker

and microservices and other supporting utilities. The text contains consequences, recommended procedures and possible problems and their solutions resulting from using these two technologies besides the application itself.

Keywords web application scalability, web application high availability, microservices, Docker, Rancher, cluster

Obsah

Seznam obrázků

Seznam tabulek

Seznam zdrojových kódů

Úvod

V dnešní době se poměrně dost dění a práce na počítačích přesouvá na internet. Kromě webových stránek sloužících primárně k zábavě či relaxaci (sociální sítě, multimediální sítě, atd.) je k dispozici mnoho nástrojů, které využívá velké množství uživatelů ve svém zaměstnání či v podnikání. Ať už se jedná o volnočasové či pracovní aplikace, s rostoucím počtem uživatelů je nutné zajistit jejich vysokou dostupnost.

Aby provozovatelé těchto nástrojů mohli tento požadavek zajistit, je potřeba sáhnout ke škálovatelnosti – horizontální či vertikální. Horizontální škálování není možné řešit do nekonečna, proto dříve či později přijde na řadu škálování vertikální. Pokud by se ovšem řešilo pouze v rámci celé aplikace, na každém serveru by musela být její celá kopie. To je však praktické pouze tehdy, pokud každá část této aplikace je využívána zhruba stejně. Pokud tomu tak není, klonování i těch částí, které jsou využívány málo, by se dalo označit za zbytečné plýtvání zdroji.

Proto jedinou možností, jak docílit škálovatelnosti pouze některých částí aplikace, je její dekompozice na menší části. K dosažení tohoto cíle existuje několik technik, ovšem jedna z nejpoužívanějších je využití architektury mikroslužeb. Tato architektura může být používána bez dalších nástrojů, nicméně pro lepší správu a jednodušší komunikaci mezi jednotlivými službami je výhodnější každou mikroslužbu uzavřít do izolovaného prostředí, tzv. kontejnerů, na jejichž správu cílí moderní platforma Docker.

Cílem této práce je tedy popis a detailní prozkoumání architektury založené na mikroslužbách společně s kontejnerovou platformou Docker a jejich využití v zajištění škálovatelnosti ve webových aplikacích. Dále popisuje výhody a nevýhody nasazení těchto technologií, stejně tak řešení možných problémů.

Na úvod jsou v textu rozebrány základní možnosti pro zajištění vysoké dostupnosti a škálovatelnosti webových aplikací, jsou zde popsány rozdíly mezi horizontální a vertikální škálovatelností, stejně tak se práce věnuje opomíjeným technikám, jako je škálování databáze či škálování pomocí cache.

V dalších kapitolách jsou detailně rozebrány kontejnerová platforma Docker

a architektura založená na mikroslužbách. Jsou popsány výhody a nevýhody používání a možnosti propojení těchto dvou technologií.

Toto propojení je poté demonstrováno na ukázkové aplikaci, která popisuje postup při analýze, návrhu, samotné realizaci, testování a nasazování. Problémy a důsledky, objevené při realizaci této ukázkové aplikace, jsou zde také rozebrány, včetně návrhu jejich řešení.

Kapitola 1

Základní škálovatelnost webových aplikací

Základní požadavek na veškerý webový software je jeho vysoká dostupnost, což je „schopnost aplikace při běžných selháních (nejčastěji hardwaru) s velkou pravděpodobností pokračovat v běhu“ [1] – jinými slovy je to požadavek, aby aplikace byla téměř vždy (v ideálním případě vždy) pro uživatele dostupná. U webových aplikací je to zejména z důvodu, aby byli návštěvníci spokojeni a také aby kvůli výpadkům nepřicházela firma o peníze.

V malém počtu návštěvníků je poměrně jednoduché tuto vysokou dostupnost zajistit, jelikož na server nejsou kladeny takové nároky. S přibývajícími návštěvníky, popř. se zvětšováním aplikace ovšem přirozeně roste zátěž na hardware. Aby se i s tímto nárůstem neporušila vysoká dostupnost aplikace, je nutné sáhnout ke škálovatelnosti.

Škálovatelnost je „žádoucí součást systému, která indikuje jeho schopnost buď zvládat vzrůstající práci elegantním způsobem nebo být snadno rozšířen tehdy, když požadavky na práci narůstají“ [2]. Ve webových aplikacích se využívá obou těchto přístupů.

1.1 Vertikální škálovatelnost

Jedním ze základních typů škálovatelnosti pomocí rozšíření systému je vertikální škálovatelnost. Jedná se o velmi jednoduché přidání nového hardwaru. Ať už to je pouhé doplnění nového modulu paměti RAM do počítače či zdvojení grafické karty, ve všech těchto případech se jedná o vertikální škálovatelnost. V kontextu webových aplikací je nejčastější případ dokoupení paměťového modulu nebo procesorového jádra do serveru. Tento typ škálování je nejjednodušší a zároveň nejrychlejší, nicméně jej nelze využívat donekonečna, ať už z limitů finančích či hardwarových.

1.2 Horizontální škálovatelnost

Z těchto důvodů se proto dříve či později využívá horizontální škálovatelnosti, která se řeší pomocí přidávání nových serverů. Na tyto servery se poté aplikace naklonuje. V případě výpočetních aplikací, které nemusí sdílet mezivýsledky, toto není problém – každý server dostane svou část pro výpočet. U webových aplikací je třeba další režie, protože je potřeba návštěvníky rovnoměrně rozprostřít mezi servery (tzv. load balancing). Pro dosažení tohoto cíle se používá komponenta zvaná load balancer, tu je možné implementovat několika způsoby.

1.2.1 DNS load balancer

Implementace load balanceru pomocí systému DNS[1] je základním a nejjednodušším přístupem k tomuto problému. DNS slouží k získání IP adresy na základě doménového jména, kdy provozovatelé aplikací ukládají do DNS záznamů tyto údaje. Obyčejně existuje k jednomu doménovému jménu jedna IP adresa, nicméně je umožněno mít k ní několik IP adres.

Právě této vlastnosti se využívá pro load balancing. V případě více IP adres většina DNS serverů funguje tak, že vrací jejich seznam v náhodném pořadí pomocí metody round robin [3], což je nejjednodušší metoda, která střídá jednotlivé záznamy v předem daném pořadí bez priorit. Tímto jednoduchým způsobem jsou jednotliví klienti rovnoměrně distribuováni mezi servery.

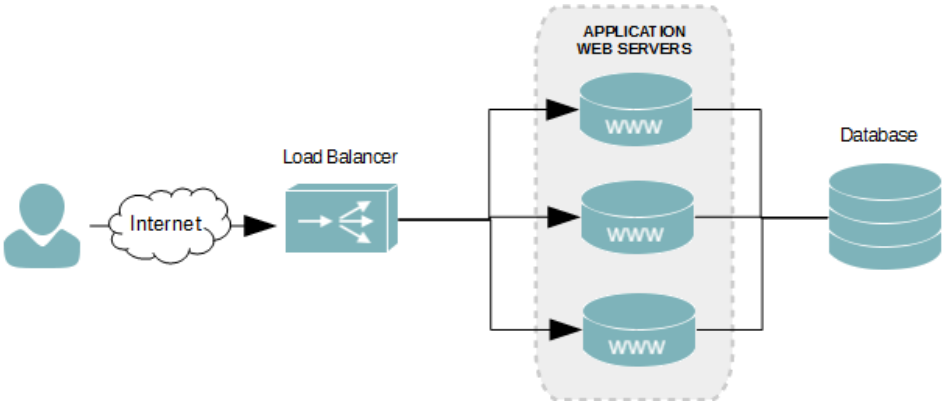
DNS load balancer má ovšem několik nevýhod, vycházejících primárně z jednoduchosti této implementace. Největším problémem je nekontrolování stavu daného serveru před odesláním IP adresy, takže v případě jeho poruchy či nedostupnosti klient i tak obdrží IP adresu směřující na tento nefunkční server. Druhým problémem je cachování samotných DNS záznamů, ať už klienty, či servery na trase, takže při změně záznamu může dlouho trvat aktualizace všech dotčených míst.

1.2.2 HTTP load balancer

Lepší, ale zároveň složitější implementací load balanceru je na základě samotného HTTP protokolu, kterého se dosáhne jeho vyčleněním do samostatné komponenty. Load balancer se v tomto případě předsune před servery s aplikací, DNS záznam vede na něj a tato samostatná součást poté řeší distribuci požadavků mezi jednotlivé servery. Ukázka komunikace je viditelná na obrázku 1.1.

Tuto komponentu je možné naprogramovat svépomocí, nicméně v současné době existuje mnoho již hotových řešení, které zároveň plní jiné funkce, jako je proxy, cachování, monitoring, atd. Jako příklad takového programu

[1] Domain Name System



Obrázek 1.1: Ukázka komunikace při použití HTTP load balanceru (zdroj: [4])

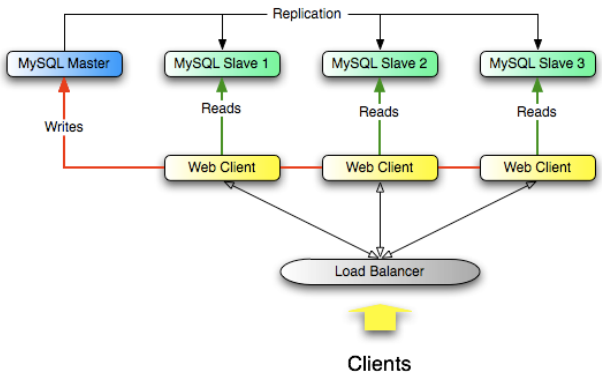
lze uvést Nginx[2], který kromě samotného load balancingu obsahuje mnoho dalších funkcí.

Co je u tohoto typu load balanceru rozhodující, jsou jeho algoritmy a metody vyvažování zátěže. V případě DNS load balancingu se jednalo o **round robin**, kromě něj existuje několik dalších, zejména:

- **least connection** – load balancer vybere ten server, který má nejméně aktivních spojení
- **least response time** – při použití této metody se kromě počtu aktivních spojení bere v potaz také nejkratší průměrný čas odpovědi serveru
- **least bandwidth** – tato metoda vybere server, který momentálně obsluhuje nejmenší objem dat, měřeno v přenesených MB za vteřinu
- **least packets** – tento algoritmus pracuje podobně jako předchozí, pouze měří počet přenesených paketů
- **hash metody** – metody založené na hashování nejprve vypočítají hash z předem daných parametrů (např. z IP adresy klienta), na základě které vyberou server

Všechny tyto metody si zároveň také udržují informaci o stavu jednotlivých serverů – v případě nedostupnosti jej vyřadí ze svého seznamu do té doby, než je opět funkční. Zároveň se můžou serverům přidávat váhy tehdy, když je potřeba jeden z nich při výběru zvýhodnit, např. v případě výkonnějšího hardwaru.

[2] `https://nginx.org/`



Obrázek 1.2: Zapojení databáze do režimu master-slave (zdroj: [7])

1.3 Škálování databáze

Nedílnou součástí většiny webových aplikací je databáze, která může obsluhovat také velký provoz, a tím pádem zpomalovat celý systém. Jako u samotného jádra webové aplikace, i zde se pomocí vertikálního škálování dosáhne zrychlení, stejně tak se využívá škálování horizontálního, pro které existuje několik přístupů.

1.3.1 Rozdělení na části

Pokud to databázový návrh umožňuje, je možné databázi rozdělit na několik disjunktních částí. Jelikož na sobě tyto díly nejsou závislé, je možné je provozovat na jakémkoliv serveru bez jakéhokoliv propojení. Nicméně málokdy je takový návrh k dispozici, proto se častěji využívá následujících metod.

1.3.2 Master-slave

Tento způsob je nejjednodušším zástupcem horizontálního škálování databází. Jedná se o rozdělení databáze na jeden hlavní (master) uzel, do kterého klienti pouze zapisují, a několik podřízených (slave) uzlů, sloužících pouze pro čtení. Jednotlivé uzly navíc musí být propojené a musí na nich být nastavena replikace, aby v každém byla stejná data. Toto rozdělení umožní jednoduché rozložení zátěže a tím zrychlení aplikace.

Sice je stále nejvíce zatěžován hlavní uzel, do kterého všichni klienti zapisují, nicméně je to nejčastější způsob zajištění škálovatelnost u SQL databází, jelikož většina těchto databází nativně umožňuje pouze tento způsob propojení [5]. Ukázka aplikace této metody je ilustrovaná na obrázku 1.2.

1.3.3 Multi-master

Pokud nastane situace, že aplikace mnohonásobně více zapisuje, než čte, je předchozí řešení nevyhovující. V takovém případě je nutné přejít na metodu multi-master, která umožňuje, aby všechny slave uzly z předchozího přístupu byly zároveň master, tedy dostupné i pro zápis.

V tomto případě je ovšem nutné, aby aplikace generovala univerzální unikátní identifikátory (UUID) pro jednotlivé záznamy. Jelikož je samozřejmě nutná replikace dat, v případě, že by se tyto identifikátory negenerovaly, mohlo by při replikaci dojít k přemazání některých řádků. Některé SQL databáze tento princip neumožňují přímo, je možné jej docílit pouze s velkým úsilím a množstvím konfigurace navíc [6].

1.3.4 Shardování

Dalším přístupem, jak škálovat databázi, je pomocí horizontálního dělení databáze nazývané jako shardování. Jedná se o distribuci dat mezi jednotlivé servery, na kterých je dostupný databázový stroj. Vzor rozdělení se určuje na základě předem definovaných parametrů, nejčastěji se využívá dělení dle primárních klíčů, lze využít také datum vytvoření, atd.

Celá množina dat je tedy rovnoměrně rozmístěna mezi všechny uzly a tím je dosáhnuto maximálního výkonu. Některé NoSQL databáze, jako je např. Elasticsearch[3], umožňují toto shardování společně s replikací nativně, dokonce s jejich použitím počítají. Pomocí těchto dvou metod je zajištěn maximální výkon, stejně tak bezpečná záloha dat [8].

1.4 Škálování pomocí cachování

Kromě klasických přístupů uvedených výše se u webových aplikací využívá také škálování pomocí cachování, což je často opomíjený přístup, pomocí kterého je ovšem škálování aplikace jednoduché, levné a zároveň při jeho správném použití většinou dosahuje výborných výsledků. Existuje několik způsobů cachování, které je možné využít, většina z nich využívá open-source nástroje, takže jsou bez poplatků.

1.4.1 Content Delivery Networks

Síť pro doručování obsahu, častěji zvané anglickým Content Delivery Networks (CDN), je „několik počítačů propojených mezi sebou, které u sebe mají duplikovaná uživatelská data (obrázky, webové stránky)“[9].

Tyto počítače většinou bývají strategicky rozmístěny po různých místech na světě. Díky tomu je možné, aby klient dostal odpověď od serveru, který je

[3] `https://www.elastic.co/products/elasticsearch`

pro něj nejvýhodnější – je mu nejblíže, je nejrychlejší, je dostupný, atd. Jeden z typů CDN sítí je CDN cache určená ke stahování dat.

Umístěním statického obsahu aplikace (CSS, JS, obrázky) na CDN se také šetří síť v okolí aplikačního serveru. Kombinací veškerých předchozích aspektů se docílí mnohem rychlejšího načtení stránky z pohledu uživatele.

1.4.2 HTTP cache

Samotný protokol HTTP obsahuje několik základních možností pro cachování, jejichž nastavení se definuje v hlavičkách HTTP odpovědí. Při této komunikaci klient-server mohou dle tohoto nastavení cachovat buď klienti (prohlížeče) anebo také mezilehlé servery.

Vývojáři a kodéři také často umisťují informace o cachování do meta tagů v HTML kódu. Tyto meta tagy ovšem akceptují a řídí se jimi pouze klienti, mezilehlé servery je ignorují [10]. Spolehlivější konfigurace se zadává do již zmíněných hlaviček HTTP odpovědi. Pro povolení/zakázání cachování obsahu je určen tag `Cache-control`, jehož hodnoty určují, kdo a co může cachovat. Kupříkladu hodnota `private` povoluje cachovat pouze klientům, kdežto při hodnotě `public` mohou cachovat všechny servery na trase mezi klientem a webovým serverem.

Další možností HTTP cachování je pomocí hlaviček `Last-Modified`, resp. `ETag`, do kterých se udává poslední datum změny dané stránky, resp. její podpis, který je reprezentován např. pomocí hashe obsahu. Další hlavičkou umožňující kontrolu cache je `Expires`, udávající datum expirace cachovaného obsahu.

Výše zmíněné hlavičky slouží také pro cachování AJAX požadavků. Tímto nastavením by se měly řídit všechny prohlížeče, stejně tak mezilehlé servery, nicméně stoprocentní jistota není. Uživatelé si např. mohou vypnout cachování ve svých prohlížečích, mezilehlé servery mohou toto nastavení ignorovat, atd.

Pro úplnou jistotu cachování se používají HTTP akcelerátory a cache servery, které jsou na straně spravovaného webového serveru, tím pádem má nad nimi administrátor veškerou moc.

1.4.3 HTTP akcelerátory

Programy s názvem HTTP akcelerátory, či také web akcelerátory jsou „software sloužící ke zrychlení přenosu obsahu mezi web serverem a klientem pomocí různých technik, jako je cachování a komprese“ [11]. Tyto programy jsou vložené před web server a fungují jako reverse proxy. Zároveň také mohou plnit funkci load balanceru.

K dispozici je mnoho HTTP akcelerátorů, mezi nejznámější patří Varnish[4] či Squid[5]. Většina těchto akcelerátorů cachuje statický obsah, jako jsou

[4] https://varnish-cache.org/
[5] http://www.squid-cache.org/

CSS a JS soubory, obrázky, HTML stránky, atd. Kromě toho také umožňují některé z těchto souborů komprimovat (např. minimalizováním počtu řádků, atd.). Kromě výše zmíněných samostatných programů existují pluginy do známých web serverů, jako je Apache (pomocí modulu Pagespeed[6] od společnosti Google), či Nginx.

1.4.4 Cache servery

V předchozích sekcích se jednalo o techniky, které se o cachování staraly víceméně samy, ať už použitím externích programů či využití HTTP protokolu. Poslední možností, jak zajistit škálování pomocí cache, je nainstalování cache serveru a jeho integrace přímo do webové aplikace.

Tyto cache servery jsou spuštěny na pozadí webového serveru, poslouchají na určitém portu a zpracovávají požadavky pro zápis a čtení. Většinou se jedná o tzv. key-value úložiště, kdy v paměti RAM tento server ukládá data pod daným klíčem. Mezi nejznámější cache servery patří Memcached[7] či Redis[8], pro které existují klientské knihovny pro širokou škálu programovacích jazyků, umožňující snadnou integraci do webové aplikace a jednoduché používání cache.

Často se využívají pro cachování opakujících se (a často neměnných) výsledků SQL dotazů, uchovávání mezivýpočtů, aj. díky čemuž se může rychlost aplikace mnohonásobně zrychlit.

[6] https://developers.google.com/speed/pagespeed/module/
[7] https://memcached.org/
[8] https://redis.io/

Kapitola 2

Architektura založená na mikroslužbách

Architektura založená na mikroslužbách (užívá se také zkrácený název **mikroslužby**), angl. microservices architecture, je poměrně moderní a rychle rostoucí architekturou používanou při vývoji softwaru. Samotná idea je známa již delší dobu, konkrétní název a rozšíření přichází na svět až v posledních pár letech. Konkrétně byl tento pojem poprvé použit v roce 2011 na workshopu poblíž italských Benátek [12]. Jelikož je tato architektura velmi mladá, neexistuje žádná konkrétní definice, pouze jakýsi souhrn společných rysů. Zjednodušeně se jedná o dekompozici celé aplikace na malé samostatné služby, které mezi sebou komunikují většinou pomocí protokolu HTTP [13]. Každá služba by měla obstarávat jednu konkrétní oblast aplikace, jako je např. správa objednávek, správa uživatelů, aj.

2.1 Motivace

I když se při vývoji softwaru snaží vývojáři co nejvíce oddělit jednotlivé kousky kódu do srozumitelných částí, k čemuž používají např. architekturu MVC[9], z pohledu celku se stále jedná o monolitickou aplikaci, která má hexagonální architekturu [14], což znamená, že v jádru aplikace obsahuje veškerou business logiku a na vrcholech šestiúhelníku jsou jednotlivé adaptéry komunikující s okolním světem, jako jsou databázové adaptéry, webové rozhraní, napojení na účetní programy, apod. Aplikace tohoto typu jsou velmi časté, jelikož jsou jednoduché pro vývoj, jsou pro ně uzpůsobené vývojové prostředí, nasazení probíhá pouhým zkopírováním na aplikační server, testování lze provést pomocí testovacích knihoven, jako je Selenium[10], atd.

[9] Model-View-Controller

[10] `http://www.seleniumhq.org/`

S rostoucím počtem nových funkcionalit ovšem zdrojový kód roste a objevuje se celá řada problémů, jako je údržba aplikace, kdy není možné, aby si vývojáři pamatovali všechny možné části kódu – většinu oprav je schopen vyřešit většinou ten programátor, který na projektu působí nejdéle, zároveň je také pro nového vývojáře velmi složité zorientovat se v kódu a začít brzy samostatně pracovat. Aplikaci také přibývá fyzická velikost, takže nasazování na produkční server stále zabírá více a více času, kromě nasazování trvá značnou dobu také samotný start aplikace, což je například při nějakém neočekávaném restartu problém. Adaptace nové technologie či migrace na novější verzi stávající je jak časově, tak finančně náročné s nejistým výsledkem v podobě možné nefunkčnosti některých částí. Pokud je některá část kódu paměťově náročná a jiná náročná na procesorový výkon, nemůže vyhrát ani jedna, výběr hardwaru vždy skončí nějakým kompromisem. Velkým problémem je také škálovatelnost, kdy jedinou možností je duplikace celé aplikace a její nasazení na jiný server, což je ale zbytečné, pokud je potřeba škálovat pouze některou část aplikace (např. výpočetně náročnou sekvenci kódu).

2.2 Základní myšlenky a otázky

2.2.1 Společné rysy

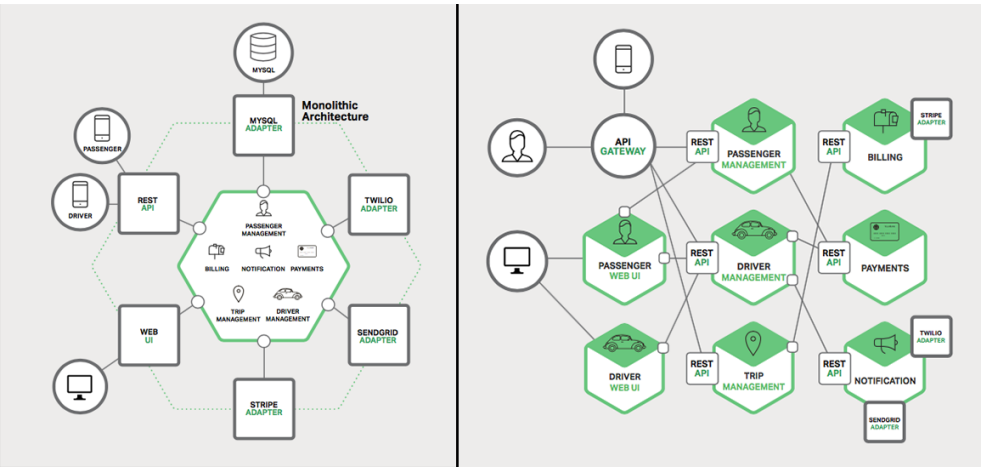
Jak bylo zmíněno výše, mikroslužba samotná není explicitně definována, nicméně většina definic je velmi podobná té od Sama Newmana [13]: „Mikroslužby jsou malé, autonomní služby, které vzájemně spolupracují“. Základním cílem architektury je tedy rozdělit monolitickou aplikaci na mnoho malých služeb, jež má každá uvnitř svou vlastní hexagonální architekturu a z této dekompozice těžit. Každá služba by měla mít tyto společné rysy:

- obstarávat nějakou logickou část aplikace
- mít vlastní repozitář
- obsahovat své vlastní testy
- nasazování by mělo být nezávislé

2.2.2 Podobnost se SOA

Již ze samotného názvu vyvstává pár otázek, jako například, zdali tato architektura není jen jiným názvem pro SOA[11], která je definována jako „množina komponent, které mohou být volány a popis jejich rozhraní může být uveřejněn a zjištěn“ [15]. Zároveň ale také SOA přesně definuje technologie a další parametry, kterými je třeba se řídit, což mikroslužby nespecifikují. Jedná se

[11] Service-oriented architecture



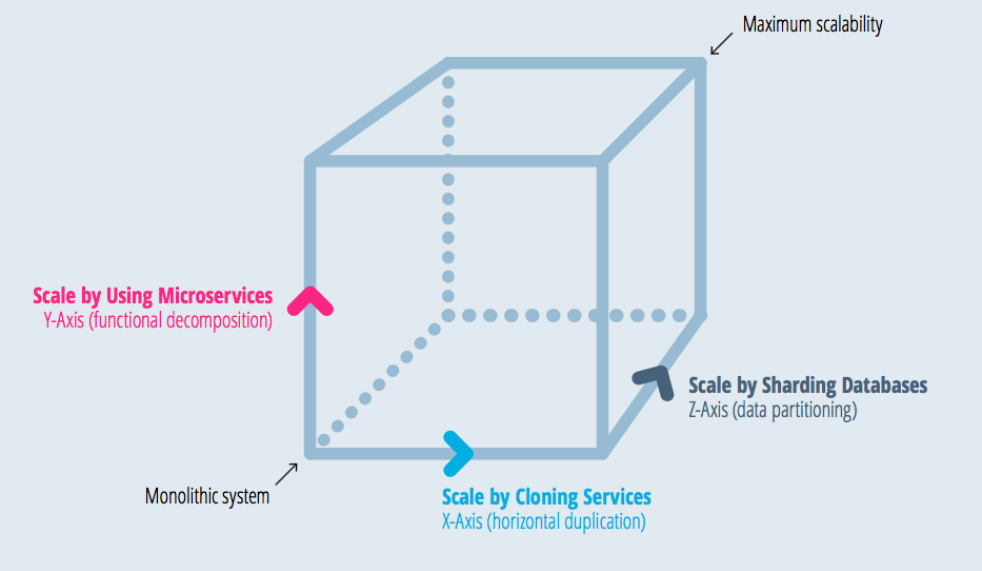
Obrázek 2.1: Ukázka rozdílu monolitické aplikace (vlevo) a její možné dekompozice na mikroslužby (vpravo) (zdroj: [17])

například o WSDL[12], SOAP[13], ESB[14], aj. Architektura mikroslužeb stejně tak nepoužívá některé koncepty, jako je canonical schema pattern[15].

2.2.3 Velikost mikroslužeb

Další otázkou je slovo **mikro**. Nikde totiž není jasně určeno, jak moc „mikro“ mají služby být a toto je jedna ze zásadních otázek při návrhu aplikace pomocí této architektury. Zároveň je to také jedno z hojně diskutovaných témat. Pokud budou služby příliš velké, stanou se postupně z nich také monolity, pokud budou naopak moc malé, vznikají problémy s komunikací a přílišnou roztříštěností. Čím menší služby budou, tím se více bude profitovat z jejich výhod, ale zároveň budou více pociťovány jejich nevýhody. Je proto tedy potřeba velmi dobře navrhnout doménový model, který je v této architektuře velmi důležitý a bez kterého není možné používat mikroslužby správně [13] [16]. V budoucnu je také vždy možné některou službu rozdělit na více menších, či naopak jich několik spojit dohromady. Na problém velikosti je možné také nahlížet představou, kolik času zabere přepsání služby do jiného programovacího jazyka – pokud by tato doba byla příliš dlouhá, znamená to, že je služba zbytečně velká.

[12] Web Services Description Language
[13] Simple Object Access Protocol
[14] Enterprise Service Bus
[15] `http://soapatterns.org/design_patterns/canonical_schema`



Obrázek 2.2: Krychle škálovatelnosti (zdroj: [16])

2.2.4 Krychle škálovatelnosti

Architektura mikroslužeb pokrývá celou krychli škálovatelnosti, popsanou Martinem Abbottem a Michaelem Fisherem, která je brána jako modelový standard pro popis škálování [19]. Každá dimenze krychle popisuje jiný typ škálovatelnosti. Samotná idea mikroslužeb, což je dekompozice rozdílných věcí, odpovídá ose Y. Horizontální osa X odpovídá klasickému klonování, které se s jednotlivými službami také často provádí. Poslední osa Z popisuje škálování pomocí shardování (jiný název je data partitioning, více informací obsahuje také kapitola Shardování), což je rozdělení například databáze, kdy každá datová část je na jiném serveru. Pro nalezení správné datové části se používá mapování, nejčastěji pomocí primárního klíče hledaného řádku. Toto rozdělení může být v mikroslužbách také aplikováno, kdy load balancer přeposílá request na vybranou instanci služby dle jeho typu, tím je zátěž rovnoměrně rozložena. Pohyb v krychli po jednotlivých osách odpovídá přechodu z monolitické aplikace do maximální možné škálovatelnosti, jak je naznačeno na obrázku 2.2.

2.3 Výhody a nevýhody

Z předchozích částí o motivaci a základních myšlenkách vychází mnoho výhod při použití architektury mikroslužeb:

1. **Škálovatelnost** – v monolotické aplikaci je nutné škálovat vše dohromady. Oproti tomu, s touto architekturou je škálovatelnost mnohem

jednodušší a dynamičtější. Stačí se zaměřit pouze na ty služby, které škálování potřebují a zároveň je klonování těchto služeb na sobě absolutně nezávislé.

2. **Komplexita** – použití mikroslužeb řeší problém složitosti aplikace. Velká monolitická aplikace je dekomponována na množinu malých služeb, které jsou mnohem lépe udržitelné.

3. **Rychlé a nezávislé nasazování** – v monolitické aplikaci je potřeba i při malé úpravě jednoho řádku kódu, aby celá aplikace prošla celým procesem nasazení. V této architektuře se opraví pouze ta mikroslužba, ve které je chyba a díky její malé velikosti je nasazení mnohem rychlejší. Zároveň se také jednotlivé služby mohou nasazovat nezávisle na sobě dle potřeby jednotlivých týmů. Současně je také snažší objevení problému a provedení rollbacku[16].

4. **Technologická dynamika** – každá mikroslužba může používat jinou technologii, která je pro ni z hlediska výkonu ideální, může být napsána v jiném jazyku či využívat jinou databázi. Zároveň také může být spuštěna na tom hardwaru, který je pro ni nejvýhodnější, tzn. paměťově náročná služba bude na serveru s vyšší pamětí a naopak nižším procesorovým výkonem, což má za následek šetření peněz za hardware. Co je také mnohem jednodušší, je adaptace nových verzí jednotlivých technologií, použití nového frameworku, atd., jelikož přepsání celé služby je mnohem rychlejší, méně náročnější a obnáší mnohem méně rizika, než je tomu u monolitické aplikace.

5. **Nezdolnost** – pokud se objeví chyba v některé z mikroslužeb, neznamená to automaticky kolaps celého systému. Ostatní služby by se totiž o tuto chybu měly postarat a použít například nacachovaná data. Je ovšem potřeba zajistit reakci služeb na tyto chyby.

6. **Organizační struktura** – v neposlední řadě je potřeba připomenout lepší organizační strukturu ve firmě, kdy se každý tým vývojářů může starat o jednu či více mikroslužeb. Jelikož pro správu těchto služeb není třeba velkých týmů, vzniká více menších týmů, ve kterých je lepší komunikace a produktivita.

7. **Znovupoužitelnost u jiných projektů** – jednotlivé mikroslužby mohou v budoucnu být využívány jinými klienty, jako jsou například mobilní aplikace či aplikace do nositelností (chytré hodinky, náramky), aj.

Jelikož použitím této architektury vzniká ze své podstaty distribuovaná aplikace, je potřeba konfigurace a správa dalších prvků, které u monolitických netřeba řešit:

[16] akce, která vrátí aplikaci do předchozí verze

1. **Komunikace** – protože každá služba je izolovaná od jiných, je potřeba mezi nimi zařídit komunikaci. V klasické aplikaci je volání metod zajištěno na úrovni programového volání metod, oproti tomu u mikroslužeb se komunikuje nejčastěji pomocí API[17], či různých front, aj.

2. **Service discovery a service registry** – aby služby mezi sebou mohly komunikovat, musí vědět, na jakém místě která služba je. To je problém zejména při dynamickém vznikání a zanikání jednotlivých instancí při běhu, proto je potřeba zajistit registraci služeb a následné získávání jejich umístění.

3. **Databázové transakce** – jelikož každá služba používá své vlastní úložiště, uskutečnění a zajištění transakce, která zasahuje do více databází, je poměrně složitá.

4. **Testování celé aplikace** – testování jednotlivých mikroslužeb je jednoduché a je to také jednou z výhod této architektury. Testování kompletní aplikace je nicméně již složitější, je potřeba si vytvořit stuby[18] ostatních služeb, či je mít spuštěné.

5. **Závislosti při nasazování** – ideálně by se každá služba měla nasazovat nezávislé na jiných, nicméně někdy se může stát, že změna provedená v jedné službě se dotkne několika dalších, a proto je potřeba správně zkoordinovat pořadí nasazování.

Výše popsané nevýhody sice nejsou žádným opravdu složitým problémem, nicméně je potřeba s nimi počítat. Je tedy nutné vždy rozmyslet, zdali použití této architektury má smysl. Obecně platí, že má smysl rozhodně pro větší projekty, případně pro ty aplikace, které počítají s budoucím růstem. Monolitická aplikace je totiž vhodná a udržitelná pouze pro menší projekty.

Podrobnější popis některých těchto problémů včetně možností řešení je poté popsáno v dalších částech textu.

2.4 Komunikace mezi klientem a mikroslužbami

Kromě komunikace mezi samotnými službami je důležitou součástí komunikace mezi klientem a aplikací. Klient, kupříkladu webové rozhraní (přesněji návštěvník na něm), totiž při zobrazení jedné webové stránky typicky komunikuje s více službami. Po přejití na stránku s detailem produktu v e-shopu se bude dotazovat katalogové služby pro název a detail produktu, cenové služby pro aktuální cenu produktu, služby s recenzemi pro zjištění zákaznického hodnocení, atd. Je tedy otázka, jak tuto komunikaci s jednotlivými službami vyřešit.

[17] Application Programming Interface

[18] část kódu, která pouze simuluje chování jiné třídy, funkce, aj.

2.4.1 Přímá komunikace klient-služba

Jednou z možností je každé službě přiřadit vlastní URL, která bude veřejně dostupná. Stránka s detailem by tedy provedla několik HTTP požadavků, každý z nich by se dotázal požadované služby, s jejímž výsledkem by se poté naložilo dle potřeby. Nejspíše jedinou výhodou tohoto řešení je jeho jednoduchost, avšak v tomto případě vysoce převažují nevýhody.

Prvním problémem je samotný počet HTTP requestů, pokud by se musel provádět každý požadavek pro každou službu zvlášť, zbytečně by se zatěžovala síť a s velkým počtem služeb by bylo velmi neefektivní nakládat s takovýmto počtem požadavků.

Další překážkou je nemožnost rozumného a jednoduchého zabezpečení přístupu k jednotlivým mikroslužbám. Jelikož se tyto požadavky mohou dít až na straně klienta ve skriptu, port mikroslužby by musel být veřejně přístupný všem IP adresám.

Pokud by se v budoucnu prováděly jakékoliv změny na straně veřejného rozhraní mikroslužeb, bylo by potřeba upravit všechna místa, kde se toto volání provádí.

Dále je potřeba mít na paměti, že ne každá služba musí komunikovat nutně přes HTTP, ale může používat například fronty či komunikaci s využitím binárního formátu či protokolu, který není vhodný pro web. Proto by volání této služby z webového rozhraní bylo dost složité.

2.4.2 API gateway

Mnohem rozumnějším řešením je mezi klienty a samotné mikroslužby vložit další prvek, nazývaný jako API Gateway [20]. Tento prvek odstiňuje veškeré mikroslužby s jejich rozhraními a je jediným vstupním bodem pro všechny klienty, kterým poskytuje své veřejné rozhraní, podobá se návrhovému vzoru Facade[19].

API Gateway tedy řeší veškeré problémy popisované v předchozí části – klientovi stačí poslat pouze jeden HTTP požadavek, tato brána zavolá všechny nutné mikroslužby, zároveň provede potřebné překlady na „exotické“ protokoly. Pokud se cokoliv změní na straně mikroslužby, volání stačí upravit na jednom centrálním místě. Mikroslužbám pak stačí na firewallu povolit pouze přístupy z IP adresy serveru, na které je spuštěna tato brána. Kromě jiného může řešit i cachování, load balancing, monitorování, autentifikaci a další.

Nevýhodou je ovšem konfigurace a údržba další funkční části, která se může navíc pokazit. Zároveň se toto místo může stát úzkým hrdlem, jelikož přes něj tečou veškeré požadavky klientů – je proto potřeba zajistit vysokou dostupnost a spolehlivost této brány.

[19] `https://sourcemaking.com/design_patterns/facade`

2.5 Komunikace mezi mikroslužbami

V monolitické aplikaci je komunikace mezi jednotlivými kusy kódu zajišťována velmi jednoduše, a to na úrovni programového volání metod. V této architektuře je zajištění komunikace mezi službami mnohem náročnější, je to také jedna z nevýhod mikroslužeb.

V rámci zprovoznění komunikace mezi službami (IPC[20]) je potřeba vyřešit tyto základní body:

1. jaký použít styl komunikace a jaký pro to zvolit protokol
2. jak správně navrhnout metody
3. co dělat, když některá služba není dostupná

2.5.1 Styly komunikace

V komunikaci klient → služba existuje několik možných stylů, v zásadě se mohou rozdělit na dvě dimenze pohledů [21]: první určuje, jestli jedna služba komunikuje s jednou (1:1) či více (1:N) ostatními službami. Druhá dimenze zahrnuje synchronní (blokující) a asynchronní (neblokující) komunikaci.

Konkrétně se jedná o tyto styly:

- **Požadavek/odpověď (1:1, synchronní)** – jedná se o základní typ komunikace známý z protokolu HTTP, kdy klient pošle požadavek serveru (v tomto případě službě) a čeká na odpověď. Po dobu čekání na odpověd klient blokuje své vlákno.
- **Notifikace (1:1, asynchronní)** – klient opět pošle požadavek, na rozdíl od předchozího typu ovšem nečeká na odpověď, takže nic neblokuje.
- **Asynchronní požadavek/odpověď (1:1, asynchronní)** – služba po přijetí požadavku pošle odpověď s informací, že požadavek se bude zpracovávat déle, dle návrhu na to klient rozumně zareaguje (např. obdrží v odpovědi URL, na které se poté bude dotazovat na stav požadavku).
- **Publish-subscribe (1:N, asynchronní)** – klient publikuje zprávu, která je poté zpracována žádnou či více službami.
- **Publish-asynchronní odpověď (1:N, asynchronní)** – klient publikuje zprávu, na kterou po nějakou dobu očekává odpověď od zodpovědných služeb.

Styl komunikace je potřeba zvolit při návrhu aplikace dle požadovaného chování konkrétní služby. Dle zvoleného stylu je poté nutno použít vhodnou technologii.

[20] Inter-Process Communication

2.5.1.1 Synchronní komunikace

Pokud návrh vyžaduje synchronní komunikaci, nejspíše se pro to bude používat protokol HTTP, který je na stylu požadavek-odpověď založen. Samotný protokol ovšem nestačí, je potřeba použít nějakou architekturu, která bude jasně definovat způsob komunikace.

Jedním z takových architektur je REST[21], který se v současné době stal standardem pro webové API po boku XML-RPC či SOAP. Jeho autorem je Roy Fielding, který je zároveň spoluautorem samotného HTTP. V rámci této architektury popisuje několik základních principů, které by měly aplikace používající tento standard dodržovat [22]. Nejvíce patrným prvkem je používání HTTP metod, jejichž chování je jasně definované.

Další možností je použití nějakých méně rozšířených nástrojů, jako je např. Apache Thrift[22], což je framework umožňující vzdálené volání metod mezi aplikacemi napsanými v různých jazycích. Podobnou knihovnou je také gRPC[23], jejíž použitím se dosáhne stejného cíle, jako v předchozím případě, tento nástroj ovšem používá novou verzi protokolu HTTP 2 a využívá jeho hlavní výhody, takže by měl být velmi rychlý [18].

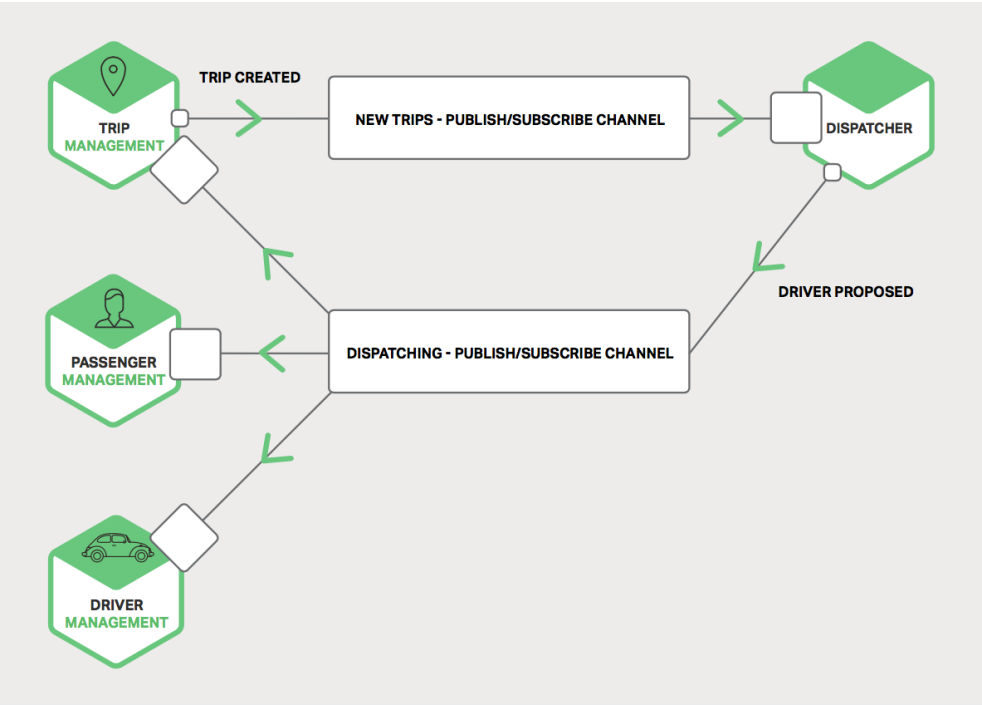
Mezi výhody používání HTTP protokolu zajisté patří jeho rozšířenost a jednoduchost. Nevýhody vyplývají z jeho podstaty – například obě strany musí být v době komunikace funkční a zapnuté, zároveň klient musí vždy vědět URL serveru, atd.

2.5.1.2 Asynchronní komunikace

Hlavním prostředkem k zajištění asynchronní komunikace jsou systémy založené na zprávách. Klient místo blokujícího volání při zaslání požadavku a následného čekání na odpověď pošle zprávu serveru, která má hlavičku (obsahuje metadata, jako jsou např. identifikace odesílatele) a poté samotný obsah. Klient po odeslání dále neblokuje žádná vlákna a očekává zprávu s odpovědí od serveru. Tyto zprávy se vyměňují v tzv. kanálech, které fungují na principu front. Existuje velké množství volně dostupných knihoven pro implementaci těchto kanálů, jako jsou např. RabbitMQ[24], ZeroMQ[25], Apache Kafka[26] a mnohé další.

Dle počtu příjemců (tzv. konzumentů) každé fronty se rozlišují dva typy kanálů: point-to-point (zpráva je doručena právě jedné službě) nebo publish-subscribe (zpráva je doručena všem konzumentům, kteří jsou „přihlášení k odběru“). Příklad použití takovéto komunikace je viditelný na obrázku 2.3.

[21] Representational state transfer
[22] `https://thrift.apache.org/`
[23] `http://www.grpc.io/`
[24] `https://www.rabbitmq.com/`
[25] `http://zeromq.org/`
[26] `https://kafka.apache.org/`



Obrázek 2.3: Příklad komunikace pomocí publish-subscribe kanálů (zdroj: [17])

Tento způsob komunikace přináší několik významných výhod. Požadavky se mohou skládat za sebe, tvoří se tedy tzv. buffering. Klient nemusí znát adresu služby, se kterou chce komunikovat, odpadá tedy nutnost řešení service discovery, rozebrané v další části textu. Dále také samozřejmě nevzniká žádné blokování vláken.

Hlavní nevýhoda je nutnost instalace a údržby další funkční části, oproti použití pouze HTTP protokolu je tato komunikace složitější co do zprovoznění a údržby. Zároveň se všechny zprávy musí správně štítkovat, aby vždy dorazily v pořádku zpět ke klientovi.

2.5.2 API

API dané služby je „smlouvou“ mezi službou samotnou a jejími klienty. Definice metod API vzniká při návrhu a řídí se jím všichni klienti, při jakékoliv neohlášené změně se jedná o porušení smlouvy a nemožnost klientů kontaktovat službu.

Je jasné, že do budoucna je potřeba počítat s vývojem API. V monolitické aplikaci se při takové změně projdou všechny místa v kódu, které staré API používají a jednoduše se přepíšou. Změny jsou poté nasazeny v jeden moment. V architektuře mikroslužeb je tato změna složitější, protože existuje několik

služeb, které danou metodu využívá. Je proto potřeba zajistit zpětnou kompatibilitu. Pokud jsou změny tak velké, že ji není možné zařídit, po nějakou dobu je nutné, aby obě verze běžely současně, při využití REST architektury se poté číslo používané verze může předávat např. v parametru URL.

2.5.3 Zajištění funkčnosti při chybě

Občas se může stát, že volaná služba nebude dostupná, ať už z důvodu chyby sítě, jiného hardwarového problému, atd. Aby návštěvník na webu nepocítil tuto nefunkčnost, je potřeba s touto možnou situací počítat a pokud možno se jí nějak vyvarovat.

Společnost Netflix[27], která je jedním z významných hráčů na poli mikroslužeb, poskytuje své zásady, jak se s touto situací vypořádat:

- vždy používat timeouty, po jejichž uplynutí se uzavře spojení
- definovat horní hranici nevyřízených požadavků od dané služby, po překročení již další požadavek neprovádět a rovnou jej ukončit
- počítat poměr neúspěšných/úspěšných požadavků, pokud poměr bude vyšší, než stanovená mez, požadavek opět neprovádět a rovnou ukončit. Po určité době tento poměr vynulovat a počítat jej znova
- mít záložní plán, pokud požadavek selže, jako například vrácení zacachovaných dat

2.6 Škálování mikroslužeb

Jednou z největších výhod mikroslužeb a zároveň hlavním tématem této práce je řešení škálovatelnosti pouze částí aplikací. Pro dosažení tohoto cíle stačí použít tuto architekturu a škálovat pouze ty služby, které to vyžadují. Na první pohled vypadá vše jednoduše, prostě se jen naklonují ty služby, u kterých je to potřeba. Tím je škálování, které se může navíc řešit na mnohem jemnější úrovni než u monolitických aplikací, vyřešeno.

Problém nastává tehdy, pokud se škálují služby v době běhu celé aplikace. Každá služba totiž musí znát přesné síťové umístění (IP a port) té služby, se kterou chce komunikovat. Pokud se škálování řeší staticky před spuštěním aplikace, a poté se již nemění, je řešením například použití konfiguračního souboru či proměnných prostředí, do kterých se přiřadí umístění jednotlivých mikroslužeb. Ve většině případů ovšem klientská služba musí řešit load balancing[28], což může být vyřešeno velmi jednoduše, třeba náhodným výběrem.

Pokud je ovšem třeba reagovat v době běhu aplikace a dynamicky jednotlivé klony služeb přidávat či naopak odebírat, předání umístění jednotlivých

[27] `https://www.netflix.com/`
[28] rozložení zátěže

služeb již není tak triviální. Tento problém řeší mechanismy, které spolu úzce souvisejí – service registry (též registr služeb), což je databáze běžících služeb a jejich umístění, a service discovery, aneb jak získat lokaci dané služby.

2.6.1 Service discovery

V aplikacích založených na mikroslužbách, které navíc běží v prostředí cloudu, je dynamické škálování častým jevem. Nově vytvořené služby navíc můžou mít dynamicky přidělenou IP adresu i port. Existují tři základní typy získávání umístění služeb – DNS service discovery, service discovery na straně klienta a service discovery na straně serveru.

2.6.1.1 DNS service discovery

Samostatnou skupinou implementací tohoto mechanismu je pomocí DNS, která je zároveň nejjednodušší a nejsnažší na pochopení. Tato metoda je podobná klasickému DNS load balanceru. Každá služba bude mít URL ve tvaru např. `nazev-sluzby.mojedomena.cz`. IP adresy jednotlivých instancí služeb se poté zadávají do DNS záznamu ke konkrétní doméně. DNS server by poté měl vrátit náhodnou IP adresu ze seznamu zadaných.

Nevýhody tohoto řešení také vychází z DNS load balanceru, největším problémem je zejména nemožnost kontrolovat stav serveru na dané IP adrese. Další nevýhoda nastává při aktualizaci DNS záznamů – než klienti zaznamenají tuto změnu, může to trvat hodiny, dále si také mohou jednotlivé servery na trase vést svou vlastní cache, takže aktualizace může trvat mnohem déle. Výhodou je jednoduchost a rozšířenost samotného DNS.

2.6.1.2 Service discovery na straně klienta

U tohoto vzoru je klient zodpovědný za získání správného umístění požadované služby. Před tím, než uskuteční požadavek, se nejdříve zeptá registru služeb, ten mu vrátí seznam IP adres. Klient poté aplikuje daný load balance algoritmus a vybere umístění, na které pošle požadavek.

Toto je velmi přímočarý způsob, jak implementovat service discovery, kromě registru služeb se zde nevyskytuje žádná další část, kterou je potřeba udržovat. Nevýhodou je ovšem nutnost psaní logiky pro service discovery pro každého klienta zvlášť.

2.6.1.3 Service discovery na straně serveru

Pokud je za service discovery zodpovědný server, řeší se to pomocí další funkční části, která se bude dotazovat registru služeb namísto klienta, stejně tak bude mít na starosti load balancing.

Tím se detaily a logika service discovery abstrahuje od klienta a ten pouze provádí požadavek na tento load balancer. Nevýhodou je ovšem opět správa další funkční části.

2.6.2 Service registry

Registr služeb je sám o sobě databáze všech běžících služeb a jejich umístění. Je ovšem potřeba tuto databázi aktualizovat o nové či zaniklé služby. Jsou zde dva vzory, jak tento mechanismus implementovat, podobné způsobům u service discovery.

2.6.2.1 Vlastní registrace

V tomto vzoru je za registraci a odhlášení zodpovědná sama služba. Zároveň také může posílat po uplynutí definovaného intervalu testovací paket, čímž dává najevo, že je stále aktivní, takže její přítomnost v databázi neexpiruje (tzv. heartbeat).

2.6.2.2 Registrace pomocí třetí strany

U této možnosti se pro registraci používá jiná systémová komponenta a nejsou za ní zodpovědné služby samotné. Tato komponenta pozoruje změny nad nějakou množinou instancí služeb, přidává nově vzniklé a odhlašuje zaniklé služby.

Dovětek

Pro zajištění a správného používání service registry a service discovery není třeba implementovat všechno na vlastní pěst. Existuje mnoho nástrojů, které jsou odladěné a velmi spolehlivé, jako např. Netflix Eureka[29], AWS Elastic Load Balancer[30] nebo Apache ZooKeeper[31]

2.7 Nasazování

Důležitým aspektem při používání mikroslužeb je jejich strategie nasazování, která je oproti nasazování monolitu dost odlišná. V případě jednolité aplikace běží na jednom či více serveru jedna či více stejných klonů dané aplikace. U mikroslužeb je možností nasazování více. Obecně jsou požadavky pro nasazování takové, aby bylo rychlé, jednoduché, aby jednotlivé služby byly izolované a mohly se jim jednoduše omezovat hardwarové zdroje.

[29] `https://github.com/Netflix/eureka`
[30] `https://aws.amazon.com/elasticloadbalancing/`
[31] `http://zookeeper.apache.org/`

2.7.1 Více služeb na jednom serveru

Nejjednodušší a nejpřímočařejší je taková strategie, kdy se na jeden server (ať už fyzický či virtuální) nasazuje skupina služeb, což pro jednoduché aplikace může být zajímavou možností.

Výhodou je jeho jednoduchost, a rychlost nasazování – v případě služby psané v Javě se na produkční server pouze nakopíruje war balíček, v jiných jazycích, jako je např. PHP či NodeJS, stačí pouze zkopírovat zdrojový kód.

Problémem je téměř neexistující izolace mezi službami, což by měla být jednou z hlavních výhod této architektury, také se velice špatně omezují hardwarové zdroje jednotlivým službám. Dále lidé, kteří mají nasazování na starosti, musí znát do detailu postup při kopírování tohoto kódu, jako například spuštění migrací, nutnost instalace podpůrné knihovny, atd.

2.7.2 Jedna služba na jednom serveru

Z důvodů uvedených v předchozích odstavcích se proto více přistupuje k modelu, kdy na jednom hostujícím zařízení běží pouze jedna služba. Pokud by ovšem na každém serveru běžela jedna služba, byla by vyřešena pouze izolace služeb, další nevýhody by zůstaly. Proto se v dnešní době využívají možnosti virtualizace, které tyto nevýhody eliminují.

2.7.2.1 Jedna služba na jednom virtuálním stroji

První způsob využití virtualizace jsou virtuální stroje (anglicky virtual machine – zkratka VM). Znamená to, že každá mikroslužba je zabalena do virtuálního obrazu, který je možné spustit na virtualizačním nástroji typu VMWare, VirtualBox, aj. jako virtuální stroj.

Tímto způsobem se perfektně zajistí izolace, jednoduše se můžou omezit přístupy k procesoru a paměti, s jednotlivými obrazy se mnohem jednodušeji manipuluje při kopírování, atd. Zároveň je také s jejich pomocí jednodušší vytvoření cloudové infrastruktury.

Problém nastává v celkové velikosti virtuálního stroje, která pramení z přítomnosti vlastního operačního systému v každém obrazu. Kvůli tomu také každý virtuální stroj potřebuje poměrně dost hardwarových zdrojů z hostujícího operačního systému, proto není možné, aby na jednom serveru běželo mnoho služeb. Další nevýhodou pramenící z robustnosti těchto obrazů, je pomalé nasazování nové služby, protože sestavení obrazu trvá dlouhou dobu, stejně tak spuštění služby, kdy zabere poměrně dost času samotné nastartování systému.

2.7.2.2 Jedna služba v jednom kontejneru

Pro vyřešení problému robustnosti je proto možné využití posledního virtualizačního přístupu, a to pomocí kontejnerů. Podobně jako v předchozím případě

je každá služba zabalena do virtuálního obrazu, který je poté použit ke spuštění kontejneru. Kontejner má poté veškeré výhody jako virtuální stroj, ale oproti němu je tato technologie mnohem jednodušší a méně robustní. Kontejnery totiž nemají svůj vlastní operační systém, ale sdílejí kernel s hostujícím operačním systémem.

Z toho vychází mnoho dalších výhod, jako je rychlé nasazování, sestavení obrazu, spouštění celého kontejneru a mnoho dalšího.

Oproti virtuálním strojům ovšem nejsou tak bezpečné, jelikož nejsou striktně odděleny od hostujícího OS, jako v předchozím případě.

Nicméně v současné době je toto preferovaným přístupem [13], právě kvůli své jednoduchosti a rychlosti. Velký rozmach využívání kontejnerů přišel s nástrojem Docker, který slouží pro jejich správu. Pro jeho výhody jsou tento přístup a Docker použity i v této práci. Tento nástroj, včetně detailnějšího rozboru kontejnerů a jejich rozdílům oproti VM jsou popsány v následující kapitole Docker a kontejnery. Kromě výhod uvedených výše Docker velmi zjednodušuje a zpříjemňuje práci s mikroslužbami, což je viditelné i na ukázkové aplikaci, která je součástí této práce.

Kapitola 3

Docker a kontejnery

Docker je open-source platforma, která umožňuje jednoduše spravovat softwarové kontejnery. Využívá se pro pohodlnější vývoj aplikací, nasazování na produkční server, architekturu mikroslužeb a mnoho dalšího. Tato kapitola rozebírá Docker do detailů, popisuje důvody pro jeho použití, včetně jeho výhod a nevýhod.

3.1 Kontejnery

Pro lepší pochopení, k čemu Docker slouží, je zapotřebí vysvětlit, co přesně kontejnery jsou. „Kontejner je technologie umožňující zabalit a izolovat aplikace včetně konfigurace prostředí – včetně všech aplikačních závislostí“ [23]. Na první pohled tato definice připomíná virtuální stroje, jelikož kontejner také obsahuje instanci nějakého operačního systému.

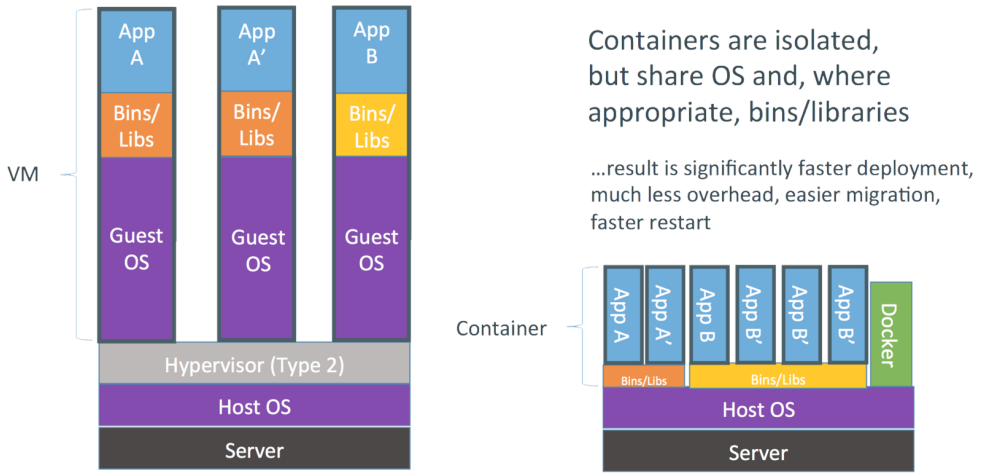
3.1.1 Rozdíly mezi VM a kontejnery

Největší rozdíl spočívá v tom, že kontejnery sdílí jádro operačního systému z hostitelského OS, kdežto v případě virtuálních strojů má každý svůj vlastní kernel.

Pro jasnější pochopení rozdílu poslouží nejlépe obrázek 3.1, ze kterého lze vyčíst, že v případě VM se zde vyskytuje tzv. hypervisor, který kompletně odděluje jednotlivé virtuální stroje od hostitelského operačního systému, kdežto kontejnery jsou napojeny na hostitelský OS pomocí kontejnerového softwaru (na obrázku toto zastupuje Docker Engine), který umožňuje kontejnerům sdílení jádra hostitelského systému. Kontejnerový software je z těchto důvodů oproti hypervisoru také mnohem méně robustní [25].

Z těchto rozdílů poté plyne řada výhod pro používání kontejnerů:

- jednotné prostředí, ve kterém aplikace běží – prostředí bude stejné jak na vývojářských počítačích, tak na produkčních serverech, či kdekoli v cloudu



Obrázek 3.1: Rozdíl VM a kontejnerů (zdroj: [24])

- odlehčenost kontejnerového enginu oproti hypervisoru umožňuje mít spuštěno desítky kontejnerů najednou
- kontejnery mohou být spuštěny či zastaveny v rámci několika vteřin
- zabalení do kontejnerů umožňuje jejich jednodušší distribuci, kterou využijí koncoví uživatelé či vývojáři
- jednoduché omezení hardwarových zdrojů

Nevýhodou poté může být pro někoho striktní neoddělení hostitelského OS a tím pádem nižší zabezpečení – toto může být ovšem vyřešeno spuštěním vícero kontejnerů pod jedním virtuálním strojem.

3.2 Historie a idea Dockeru

Samotný koncept kontejnerů je znám poměrně dlouhou dobu. Již po několik desetiletí mají unixové systémy příkaz `chroot`, který umožňuje jednoduchou formu izolace systému souborů [26]. Z tohoto konceptu poté vycházelo mnoho společností a postupně jej zdokonalovalo. Namátkou se jedná o Solaris se svými Solaris Zones [27], či Google, který přišel s utilitou `cgroups` [28], až nakonec přišel projekt Linux Containers (zkráceně LXC)[32], který některé z těchto technologií seskupuje dohromady a poskytuje kompletní řešení pro kontejnerizaci.

K používání LXC je ovšem zapotřebí velkého množství znalostí a manuální konfigurace, což je jeho největším problémem. Proto v roce 2013 přišel na scénu Docker, který zastřešuje linuxové kontejnery (obsahuje tedy veškeré

[32]https://linuxcontainers.org/

jejich výhody), rozšiřuje je o některé funkčnosti (jako například o přenositelné image kontejnerů) a poskytuje jednoduché rozhraní pro jejich správu. Tímto se začala psát nová kapitola kontejnerů v počítačové historii [25].

Jelikož je Docker i přes svou velkou rozšířenost stále velmi mladým produktem, jeho vývoj probíhá velmi dynamicky a progresivně. V průběhu tvorby této práce se událo mnoho změn, jednou z nejvýraznějších bylo představení nových edicí v březnu 2017: Docker Community Edition, což je verze dostupná zdarma pouze s komunitní podporou a Docker Enterprise Edition – placená verze určená pro firmy, v rámci které je přístupná i oficiální podpora techniků, certifikované image, atd. [33] V současné době z těchto důvodu také není dostupné větší množství tištěné literatury a ta, která dostupná je, rychle ztrácí na aktuálnosti.

Slovo Docker znamená v češtině přístavní dělník. Samotná společnost používá metaforu námořní dopravy pro ilustraci jejich cíle – tzn. zabalení aplikací do jednotlivých kontejnerů, následně jejich jednoduché distribuce a doručení kdekoliv po celém světě [24].

3.3 Architektura Dockeru

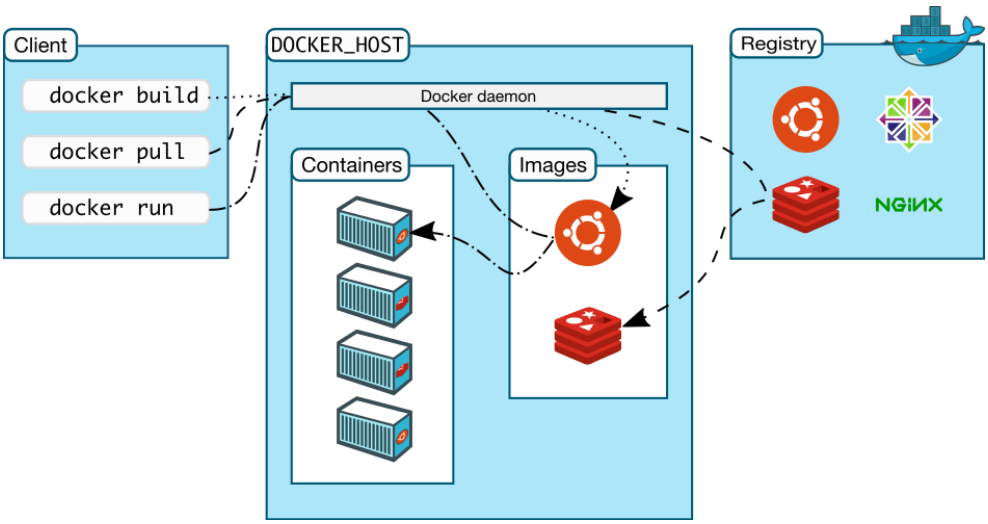
Občas se mylně objevuje informace, že Docker slouží pouze jako nástroj pro vytváření a běh kontejnerů. Není tomu tak, jelikož Docker má za cíl poskytnout celou platformu pro kontejnery [24], jejíž součástí jsou zejména Docker Engine[33] (software umožňující běh kontejnerů a sdílení kernelu hostujícího OS) a dále Docker Hub[34] (cloudová služba pro distribuci obrazů kontejnerů – o nich více v dalších kapitolách).

Jelikož je Docker založený na LXC, je dostupný pro nejrozšířenější platformy serverů (CentOS, Fedora, Debian, Ubuntu,...), stejně tak je možné jej nainstalovat na desktopové linuxové distribuce. Nativní aplikace existují i pro macOS a Windows, po jejich instalaci se vytvoří na pozadí virtuální stroj s minimalistickou distribucí Linuxu, ve které běží Docker daemon [34]. Amazon, Google, Digital Ocean a jiní poskytovatelé cloudových služeb Docker také podporují [35].

3.3.1 Docker Engine

Docker Engine je základní součástí Dockeru, která umožňuje tvorbu, běh a správu kontejnerů. Konkrétně má tyto úkoly na starosti tzv. **Docker daemon**, který běží na pozadí hostujícího systému. Ovládání Docker daemona je umožněno pomocí **Docker klienta**, který s ním komunikuje přes protokol HTTP (pomocí REST API, Unix socketů či síťového rozhraní). Jelikož je

[33] https://docs.docker.com/engine/
[34] https://hub.docker.com/



Obrázek 3.2: Ukázka komunikace Docker klienta, Docker daemona a Docker registry (zdroj: [30])

pro komunikaci použit protokol HTTP, je jednoduché pomocí Docker klienta ovládat vzdáleného Docker daemona, který běží např. na jiném serveru.

3.3.2 Docker registry

Docker registry je služba (registr) pro ukládání a sdílení Docker obrazů (využívá se více anglický výraz Docker image – zjednodušeně je to šablona pro kontejnery, více o těchto obrazech v další sekci). Výchozím a oficiálním registrem je Docker Hub[35], kde má každý uživatel zdarma neomezené množství veřejných repozitářů (kdokoliv si může image z repozitáře stáhnout) a také má možnost využít zpoplatněných privátních repozitářů. Princip je velmi podobný službám pro sdílení zdrojových kódů, jako je např. GitHub či GitLab. Stejně tak si každý může na svém serveru zprovoznit svůj vlastní registr, kromě toho je k dispozici několik registrů třetích stran, jeden z nich provozuje již zmiňovaný GitLab[36].

Pro spuštění kontejneru, který obsahuje operační systém Ubuntu potom probíhá celý proces následovně: Docker klient kontaktuje Docker daemona s žádostí o spuštění kontejneru, který je vytvořen z image Ubuntu. Docker daemon zkontroluje, jestli je tato image dostupná na lokálním počítači. Pokud ne, stáhne ji ze služby Docker Hub. Pokud je tato image dostupná, vytvoří z ní nový kontejner a spustí v něm definovaný proces (např. `bash`). Ukázka takové komunikace je vyobrazena na obrázku 3.2.

[35] https://hub.docker.com/
[36] https://about.gitlab.com/2016/05/23/gitlab-container-registry/

3.3.3 Docker Swarm

Docker Swarm je řešení pro vytvoření clusteru z více serverů, na kterých je nainstalován Docker. Swarm poté seskupí tyto servery tak, že se tváří jako jeden a je poté možné na tomto clusteru spouštět kontejnery [31]. Kromě tohoto oficiálního nástroje existuje mnoho programů třetích stran, které dosahují stejného cíle, mezi nimi jsou např. Kubernetes od Googlu[37], Marathon[38] či Rancher[39]. Docker Swarm byl dlouhou dobu samostatným nástrojem, který se instaloval zvlášť, nicméně byl v rámci verze 1.12 integrován přímo do Docker Engine [36].

3.3.4 Docker Compose

Compose je nástroj pro definování a spuštění více-kontejnerové aplikace. Jako příklad lze uvést klasickou PHP aplikaci, kde v jednom kontejneru běží PHP web server se zdrojovým kódem (Apache), v dalším je dostupná databáze (MySQL) a v posledním např. cache (Redis). Stejně tak je možné jednoduše spustit celou aplikaci, která využívá architektury mikroslužeb. V konfiguračním souboru se definují jednotlivé kontejnery a poté pomocí jednoho příkazu (`docker-compose up`) je celá aplikace připravena a spuštěna. Bez tohoto nástroje by se musel provést pro každý kontejner zvlášť příkaz `docker run` se všemi požadovanými parametry, tímto je spouštění výrazně usnadněno. Využívá se zejména pro vývoj, kdy mají vývojáři stejné prostředí, jako bude na produkčním serveru. Pro běh na produkčním serveru se Docker Compose využívá zejména u malých aplikací, u rozsáhlejších je lepší použít komplexnější nástroje typu Kubernetes, viz. výše.

3.4 Image

V předchozích odstavcích se často objevovalo spojení Docker image bez nějaké konkrétní definice. Je to proto, že image je jednou z fundamentálních součástí Dockeru. Zjednodušeně se jedná o základní jednotku (či také šablonu) pro tvorbu kontejnerů.

Každý, kdo má nainstalován Docker, si může vytvořit image dle své libosti a poté ji sdílet na Docker Hubu. Kromě toho Docker Hub poskytuje mnoho oficiálních obrazů, ať už to jsou jednotlivé operační systémy (Ubuntu, Debian), či konkrétní programy, aplikace a technologie. Namátkou to jsou databázové servery (MySQL[40], MariaDB[41]), programovací jazyky či frameworky (PHP[42],

[37] https://kubernetes.io/
[38] https://github.com/mesosphere/marathon
[39] http://rancher.com/
[40] `https://hub.docker.com/_/mysql/`
[41] `https://hub.docker.com/_/mariadb/`
[42] `https://hub.docker.com/_/php/`

```
1 FROM ubuntu:latest
2 RUN apt-get update && apt-get install -y cowsay fortune
3 CMD /usr/games/fortune | /usr/games/cowsay
```

Zdrojový kód 3.1: Ukázka Dockerfile

Ruby on Rails[43]) a mnoho dalšího. Tyto oficiální image jsou spravovány společnostmi stojícími za jednotlivými technologiemi a certifikovány Dockerem. Při jejich použití je tedy zaručena maximální funkčnost používané technologie.

3.4.1 Vytvoření vlastní image

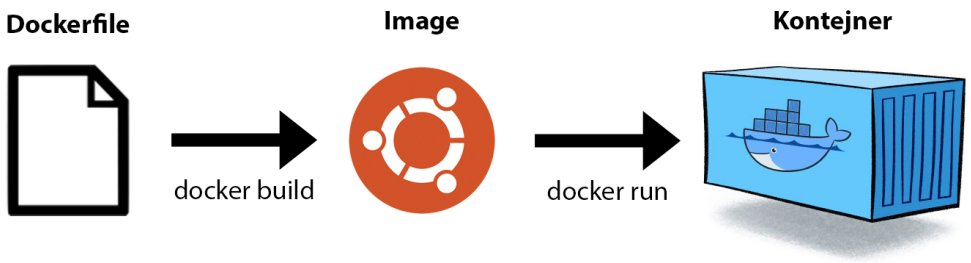
K vytvoření vlastní image slouží Dockerfile. Je to soubor, ve kterém se pomocí textových instrukcí podobných shellu definuje, jak má image vypadat. Jednoduchý Dockerfile je viditelný v ukázce zdrojového kódu 3.1.

Aby se z Dockerfilu stala image, je potřeba provést sestavení (anglicky též build), což je proces, který provede požadované instrukce z Dockerfile. V této konkrétní ukázce je na prvním řádku za klíčovým slovem `FROM` definovaná základní image, ze které bude nově vytvořená vycházet, což je povinná položka každého Dockerfile. Výběrem základní image se vlastně určuje rodičovská image nově vytvářené, která rozšiřuje její možnosti dle instrukcí z Dockerfile. V tomto případě je vybrána oficiální image linuxové distribuce Ubuntu. Poté je použita instrukce `RUN`, která provede spuštění příkazu, který se nachází za ní – zde se nainstaluje program `cowsay` a `fortune`. Nakonec se pomocí klíčového slova `CMD` určí, který proces se spustí po vytvoření a spuštění kontejneru. Zde lze samozřejmě definovat i spuštění vlastních skriptů.

Samotný build se poté provede spuštěním příkazu `docker build -t hamrla/example-image .`, kde parametr `-t` určuje název image. Znak tečky poté předává Docker daemonovi kontext, ke kterému má přístup a se kterým může operovat, v tomto případě tedy aktuální složku. Docker daemon nejdříve zjistí, zdali je dostupná základní image, v případě nepřítomnosti ji stáhne. Poté každou instrukci spustí ve svém vlastním kontejneru, který vychází z kontejneru z předchozí instrukce. Každá tato instrukce vytvoří novou vrsvtu v image. Jedna vrstva je read-only[44] systémem souborů, který je typu UFS[45], což je systém souborů, který umožňuje, aby se překrývalo více systémů souborů, čehož Docker využívá právě pro zmíněné vrstvy. Vrstva vytvořená z konkrétní instrukce je poté položena na předcházející.

Celá image je tedy množina několika vrstev ležících na sobě. Jelikož je každá vrstva současně systémem souborů, je snahou administrátorů mít jich co nejméně, aby se docílila co nejmenší velikost výsledné image, proto se často

[43] `https://hub.docker.com/_/rails/`
[44] pouze pro čtení
[45] Union File System (`https://en.wikipedia.org/wiki/UnionFS`)



Obrázek 3.3: Proces vytvoření vlastní image a spuštění kontejneru

několik příkazů spouští v jedné instrukci RUN. Zároveň je třeba mít na paměti, že pokud se v jedné instrukci vytvoří soubor a v další instrukci se smaže, v nižší vrstvě ten soubor stále je a jeho smazáním se nedocílilo snížení velikosti, proto se často také používají nástroje, které umožňují tzv. squash, což je „zdrcnutí“ vrstev do jedné. Docker zároveň jednotlivé vrstvy cachuje, takže pokud se spustí build znovu a v Dockerfilu nebo souborech kopírovaných do image nebylo nic změněno, měl by běžet zlomek vteřiny. Pro porovnání změn v souborech používá Docker svou vypočítanou hash.

Po dokončení buildu je k dispozici image, která je připravená ke spuštění. Příkazem `docker run hamrla/example-image` se spustí kontejner, který vychází z image `hamrla/example-image`. Z detailnějšího pohledu toto spuštění provede vytvoření poslední read-write[46] vrstvy, kterou položí na nejvyšší vrstvu image. Jelikož v ukázce byl v instrukci `CMD` spuštěn příkaz `/usr/games/fortune | /usr/games/cowsay`, v konzoli by měla být viditelná kresba krávy s komiksovou bublinou obsahující náhodně generovaný citát z programu `fortune`. Po tomto vytisknutí se kontejner ukončí – ten je totiž spuštěný pouze po dobu toho, co běží jeho hlavní proces.

Filozofie Dockeru je taková, že v každém kontejneru by měl běžet pouze jeden hlavní proces (jako je např. web server), nicméně často je nutné spuštění dalších procesů, jako třeba periodické běhy některých skriptů pomocí cronu. Pro tento požadavek se vžil způsob vytvoření skriptu, který se jmenuje `entrypoint` a který obsahuje spuštění více procesů, popř. se také využívá správců procesů, jedním ze zástupců je Supervisor[47], který pomocí definice v konfiguračním souboru udržuje spuštěné požadované procesy s danými parametry [37].

Celý proces vytvoření vlastní image je ještě shrnut na obrázku 3.3.

[46] možnost čtení i zápisu
[47] http://supervisord.org/

3.4.1.1 Název image

Image se identifikuje podle jména – to se skládá z několika částí a má takovýto formát: `[registr/][uživatel/]název-image[:tag]`. Komponenty uzavřené v hranatých závorkách nejsou povinné. Pokud není zvolen registr, použije se jako výchozí Docker Hub. Tag slouží pro rozlišení rozdílů, jako jsou například verze, výchozím tagem je `latest`. Definice uživatele je důležitá pro úspěšné sdílení image na Docker Hubu ke konkrétnímu účtu. Samotné nahrání do registru se poté provede příkazem `docker push hamrla/example-image`, pokud je uživatel v pořádku přihlášen, push by měl proběhnout bez problémů.

3.4.2 Základní image

Základní image je vždy první instrukcí v každém Dockerfile. Jedná se o rodiče vytvářené image, ze které vychází a dědí všechny jeho vlastnosti a z tohoto důvodu je její výběr poměrně důležitý, který ovlivňuje následné chování nového obrazu.

Jako základní image je možné zvolit jakoukoliv, která je dostupná ať už lokálně, z Docker Hubu, či jiného registru. Jak bylo nastíněno dříve, Docker Hub obsahuje mnoho oficiálních obrazů, které jsou spravovány společnostmi, starající se o vývoj dané technologie. Tyto oficální image by měly mít prioritu před těmi, které jsou vytvořené jinými uživateli, protože mají podporu a jistotu budoucích aktualizací. Zároveň je také třeba vybírat základní image dle požadavků. Pro vytvoření webové aplikace je tedy lepší použít oficiální image Nginx či Apache s PHP, než vycházet z operačního systému Ubuntu či Debian a instalovat všechny možné programy a nástroje manuálně pomocí instrukcí v Dockerfile.

Co se týče operačních systémů, na Docker Hubu jsou dostupné veškeré nejrozšířenější linuxové distribuce (Ubuntu, Debian, CentOS,...), které administrátoři a vývojáři velmi dobře znají, což je jejich největší výhodou, stejně tak dostupnost všech možných nástrojů. Jelikož ale tyto distribuce byly vyvíjeny v době, kdy Docker byl vzdálená budoucnost, nejsou vyvíjeny primárně pro něj. Dostupnost všemožných nástrojů může být výhoda, nicméně se s největší pravděpodobností v kontejnerech všechny nevyužijí. Tím pádem mohou být tyto image zbytečně velké, což se při distribuci přes internet značně pocítí, instalace nástrojů a provádění instrukcí z Dockerfile při buildu trvá také dlouhou dobu, využití zdrojů může být také vysoké. Nicméně mnoho administrátorů tyto image používá – z celkového počtu 540 000 souborů Dockerfile na GitHubu[48] jich 110 000 vychází z Ubuntu[49].

[48] `https://github.com/search?utf8=%E2%9C%93&q=language%3ADockerfile+filename%3ADockerfile&type=Code&ref=searchresults`

[49] `https://github.com/search?utf8=%E2%9C%93&q=from+ubuntu+language%3ADockerfile+filename%3ADockerfile&type=Code&ref=searchresults`

Zákl. image	Velikost zákl. image	Čas sestavení	Velikost výsl. image
Ubuntu	130 MB	58.039 s	197 MB
Alpine	3.99 MB	10.270 s	36.5 MB

Tabulka 3.1: Porovnání časů a velikostí při použití distribucí Ubuntu a Alpine jako základní image. Dockerfile obsahoval pouze instalaci MySQL klienta (zdroj: vlastní měření)

Druhou možností je využití minimalistické distribuce Alpine Linux[50], která je vyvíjená primárně pro využití v Dockeru a jejíž velikost je pouhé 4 MB. Díky této malé velikosti je build mnohokrát rychlejší, stejně tak jsou malé výsledné image, jak je viditelné z tabulky 3.1. Pokud je potřeba využití nějakého nástroje, stačí jej pomocí balíčkovacího systému `apk`[51] doinstalovat.

Je tedy potřeba důsledně zhodnotit výhody a nevýhody těchto dvou přístupů a vybrat si ten správný. Mnoho oficiálních obrazů to řeší dvěma různými tagy (např. `:latest` a `:alpine`), kde jedna image vychází z Ubuntu, Debianu či jim podobným a druhá právě z distribuce Alpine.

3.5 Komunikace a sdílení souborů kontejnerů

Častým případem je nutnost komunikace kontejnerů, ať už mezi sebou, či s veřejnou sítí. Interní komunikace probíhá na úrovni interní sítě Dockeru, která není viditelná hostujícímu stroji, externí komunikace se poté řeší publikováním portů z kontejneru na hostující počítač. Stejně tak je potřeba, aby kontejnery mezi sebou sdílely některá data, popř. aby tato data byla persistentní a dostupné i po odstranění kontejneru.

3.5.1 Interní komunikace

V architektuře mikroslužeb i v jiných případech je zapotřebí, aby spolu mohly dva kontejnery komunikovat. Kupříkladu webová aplikace potřebuje komunikovat s databázovým serverem a cache serverem, jedna mikroslužba s druhou, atd. Je tedy potřeba, aby kontejner s aplikací měl přístup k těmto ostatním kontejnerům. Dřívější přístup byl pomocí tzv. linků, které pouze propojily 2 různé kontejnery mezi sebou. Problém byl v nutnosti mít nejdříve spuštěný kontejner, na který se vytváří link. V tomto příkladě se tedy nejdříve musel spustit kontejner s databází a teprve poté ten s aplikací, a pomocí parametru `--link` přidat požadované propojení. Nebylo tedy možné vytvořit obousměrné spojení. Toto propojení způsobilo pouze přidání síťového umístění s názvem

[50] `https://hub.docker.com/_/alpine/`

[51] Alpine Linux Package management (`https://wiki.alpinelinux.org/wiki/Alpine_Linux_package_management`)

požadovaného kontejneru do souboru `/etc/hosts`, zajišťuje tak zároveň service discovery. Tento způsob je stále funkční, nicméně Docker jej považuje za zastaralý a místo něj doporučuje používat nové síťování [38], jež je jednou z velkých změn v průběhu vývoje Dockeru.

Tento nový systém síťování bere kontejnery a sítě jako dvě samostatně definovatelné komponenty. Nejdříve se nadefinují jednotlivé sítě a poté kontejnery (nazývané též jako služby), kterým se pouze předají názvy sítí, do kterých tyto služby patří. Kontejnery v jedné síti jsou poté vzájemně viditelné. Definice sítě obsahuje kromě názvu také její typ – existují dva druhy sítí.

První z nich je síť typu **bridge**, která je výchozím typem a využívá se pro kontejnery, které běží na jednom stroji. Tento typ vytvoří interní virtuální síť, na které jsou připojeny jednotlivé služby. Zároveň tento způsob nijak neřeší service discovery, takže je potřeba si pamatovat IP adresy jednotlivých služeb. Pro zajištění service discovery se využívá v základu již dříve popisovaného nástroje Docker Compose, či komplexnějších orchestrátorů typu Kubernetes, Rancher, atd.

Druhým typem sítě je **overlay**. Tento způsob se používá při Docker clusterech, tedy v případě, že je více Docker hostů propojených do sebe. Aby bylo možné využít tento způsob, je potřeba mít zajištěn Docker, který běží ve swarm módu. Poté je definice sítě a její chování podobné, jako v předchozím případě.

Nutno podotknout, že při použití rozsáhlejších nástrojů pro orchestraci si síťování mezi kontejnery řeší svým způsobem, stejně tak zajišťují service discovery.

3.5.2 Externí komunikace

Neméně častým požadavkem je otevření kontejneru do veřejné sítě, ať už z důvodu přístupu návštěvníku na webový server, připojení pomocí SSH protokolu, či mnoho dalšího. Toto je jednoduše vyřešeno pomocí parametru `-p`, kterému se předá informace ve tvaru `port-hostujícího-stroje:port-kontejneru`. Toto způsobí přesměrování daného portu z hostujícího stroje do kontejneru a zároveň také jeho povolení pro přístup na firewallu.

3.5.3 Sdílení souborů

Pokud se při běhu kontejneru jakkoliv změní adresářová struktura uvnitř něj, po jeho restartu jsou všechny změny vráceny zpět a opět se začíná se stejným stavem, který je definován v image. Je to dáno tím, že po instrukci `docker run` se přidá vždy nová vrstva systému souborů. Toto chování je ovšem nežádoucí například u databázových kontejnerů, kdy změny musí být persistentní. Pro docílení tohoto požadavku se používá parametr `-v`, kterým se určuje, která složka z kontejneru bude sdílena s hostujícím strojem. Toto sdílení se nazývá „volume“.

Po nadefinování této „volume“ je možné sdílení souborů mezi kontejnery a také sdílení s počítačem, na kterém běží Docker, čehož se využívá právě u zmíněných databázových serverů, kdy po restartu či úplném odstranění kontejneru jsou tyto soubory stále dostupné.

3.5.4 Omezení hardwarových zdrojů

Jedním z požadavků na jednotlivé mikroslužby je omezení jejich přístupu k hardwarovým zdrojům. Toto Docker umožnuje jednoduše pomocí různých parametrů při spuštění `docker run`: `-m` pro paměť RAM, `--cpus` pro omezení jader procesoru, stejně tak je možné omezit i některé IO zařízení, které kontejner může používat. Kromě těchto základních parametrů existuje také mnoho dalších, pomocí kterých je možné dosáhnout požadovaného omezení [39].

3.5.5 Nevýhody

Obecné nevýhody Dockeru pramení zejména z nutnosti znalosti této platformy, především při prvotní konfiguraci vyvíjené aplikace. Tato konfigurace se snaží být snadno pochopitelná, nicméně pro správné chování celého programu je nutné mít vše správně nastaveno. Další problém nastává při jeho použití mimo linuxové prostředí – Docker je sice dostupný pro všechny nejpoužívanější platformy, nicméně již delší dobu se při určitých situacích (zejména při použití volumes) potýká s problémy s rychlostí na Windows a macOS [40], takže při vývoji na těchto operačních systémech může být zhoršen uživatelský komfort. Je také potřeba dát pozor na možné zahlcení pevného disku počítače, Docker na něm totiž zanechává všechny původní a nepoužívané image, stejně tak zastavené kontejnery, dokud se manuálně nevymažou.

3.6 Využití Dockeru pro architekturu mikroslužeb

V předchozích částech a kapitolách bylo nastíněno, že přístup 1 kontejner – 1 mikroslužba je ideální pro provoz aplikace založené na této architektuře. Při rozboru Dockeru jsou zároveň popsány techniky a situace, které jsou při používání mikroslužeb v Dockeru typické.

Z praktičtějšího pohledu to poté vypadá následovně: každá služba má ve svém repozitáři Dockerfile, ze kterého se vytváří image. Samotný build image se velmi často řeší pomocí CI[52] nástrojů, které po změně v kódu nejdříve sestaví testovací image, nad kterou spustí definovanou sadu testů. Teprve poté se sestaví produkční image, která neobsahuje testovací a vývojářské nástroje či knihovny. Tato image je poté nasazena na produkční server pouhým publikováním do registru a následným restartem kontejneru. Některé nástroje,

[52] Continuous Integration

včetně Docker Compose, umožní provést tento restart bez výpadku – nejdříve spustí kontejner s novou verzí image, teprve poté starý kontejner vypnou.

Mikroslužby v kontejnerech spolu komunikují pomocí sítě vytvořené mezi nimi. Tuto síť je možné vytvořit dle řádků výše, nicméně pro jednodušší správu a přehled služeb jsou používány orchestrátory, které tuto komunikaci zaštitují a starají se o ni. Použitím těchto nástrojů tedy odpadá nutnost detailní znalosti jednotlivých odvětví Dockeru, zároveň jsou odladěné a tedy vhodné pro produkční server.

Kromě toho se umožňují orchestrátory jednoduše vypořádat se **škálovatelností**. Při nepoužívání těchto podpůrných služeb by se musela řešit service discovery, service registry a veškeré problémy spojené se škálovatelností, popisované v kapitole o mikroslužbách, manuálně. Po adaptaci orchestrátorů jsou tyto starosti pryč.

Docker tedy řeší veškeré požadavky dané architekturou mikroslužeb, jako je izolace jednotlivých služeb, omezení HW zdrojů, komunikace mezi nimi a škálovatelnost. Stejně tak přináší výhody a nástroje pro softwarový cyklus – jednotné prostředí pro vývojáře a produkční server, automatické testování a CI.

Kapitola 4

Ukázková aplikace

Pro demonstraci rozdílů monolitického přístupu a architektury mikroslužeb byla vybrána aplikace sloužící k nákupu přes internet – e-shop – a to zejména z důvodu rozšířenosti a všeobecné informovanosti o chování tohoto typu softwaru. Ukázková aplikace, která je součástí této práce, má za primární cíl ukázat propojení architektury mikroslužeb a Dockeru, postup při tvorbě takového softwaru a jejich využití pro škálování jednotlivých částí aplikace. Z tohoto důvodu e-shop obsahuje pouze základní funkčnost.

Při vývoji aplikace založené na architektuře mikroslužeb se využívá stejných postupů, jako u klasických přístupů vývoje softwaru, což jsou po sobě jdoucí vypracování **analýzy**, **návrhu**, **implementace** a **testování**. Rozdílem je pouze rozdrobení a opakování některých těchto kroků u jednotlivých mikroslužeb. Samozřejmostí je možnost využívat různých agilních metodik, které ovšem také obsahují tyto základní procesy.

4.1 Analýza

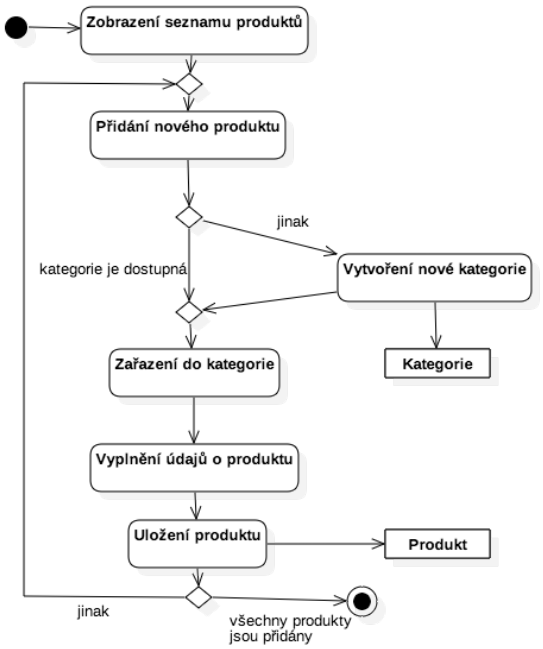
Analytická fáze probíhá nad celou aplikací, jelikož zejména analýza procesů se dotýká téměř vždy několika mikroslužeb najednou.

4.1.1 Analýza procesů

V rámci této analýzy bylo prozkoumáno několik základních procesů týkající se správy a nákupu v e-shopu.

4.1.1.1 Plnění e-shopu

Před samotným spuštěním obchodu, stejně tak v průběhu jeho provozu, je nutné jej plnit zbožím. Administrátor začíná přístupem na seznam produktů, kde po kliknutí na přidání nového produktu jej buď přiřadí do již existující kategorie, v opačném případě vytvoří kategorii novou, do které produkt za-



Obrázek 4.1: Diagram procesu přidávání produktů

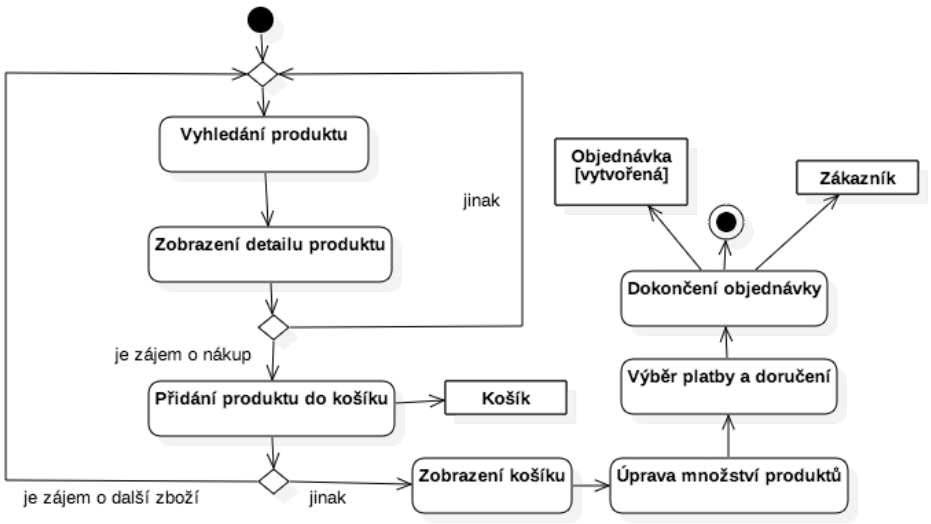
řadí. Po vyplnění údajů o zboží jej uloží a tak pokračuje dále, dokud všechny produkty nejsou v pořádku uloženy.

4.1.1.2 Nákup zákazníka

Základním a hlavním procesem z pohledu návštěvníka je nákup v e-shopu. Začíná při jeho přístupu na e-shop. Nejdříve si vyhledá požadovaný produkt, ať už pomocí fulltextového vyhledávání, či procházením kategorií. Po pročtení detailních informací si jej buď přidá do košíku nebo hledá jiný produkt. Pokud má zájem o další zboží, tato část se opakuje do doby, kdy chce objednávku dokončit. V tom případě přejde do košíku, kde si ještě může upravit množství jednotlivých produktů a po zvolení platby a doručení objednávku dokončí, čímž končí i tento proces, jehož diagram je viditelný na obrázku 4.2.

4.1.1.3 Odbavení objednávky

Po provedení nákupu od zákazníka je potřeba objednávku odbavit a odeslat. Nejdříve se objednávka zkontroluje, zdali je v pořádku. Poté se čeká na platbu od zákazníka. Po přijetí platby se kompletují jednotlivé položky a po



Obrázek 4.2: Diagram procesu nákupu zákazníka

zkompletování je objednávka předána dopravci. Až zákazník převezme balík, objednávka se vede jako vyřízená.

4.1.2 Analýza funkčních a nefunkčních požadavků

Funkční a nefunkční požadavky vycházejí z typických požadavků na provoz e-shopu a z dostupných technologií.

4.1.2.1 Funkční požadavky

- **F1. Databáze produktů** – systém bude evidovat produkty v databázi, bude možné jim přiřadit podrobné informace a zároveň je bude umět rozdělit do kategorií. Produkty bude umět filtrovat dle značky a řadit dle ceny či abecedně.
- **F2. Databáze zákazníků** – systém bude evidovat zákazníky a jejich e-mailové adresy pro jednoduché dohledání zákazníkových objednávek.
- **F3. Fulltextové vyhledávání** – návštěvníkům bude umožněno fulltextové vyhledávání v produktové databázi.
- **F4. Evidence objednávek** – systém bude evidovat objednávky, kterým bude administrátor moci měnit stavy, stejně tak bude možné mezi nimi filtrovat.

4.1.2.2 Nefunkční požadavky

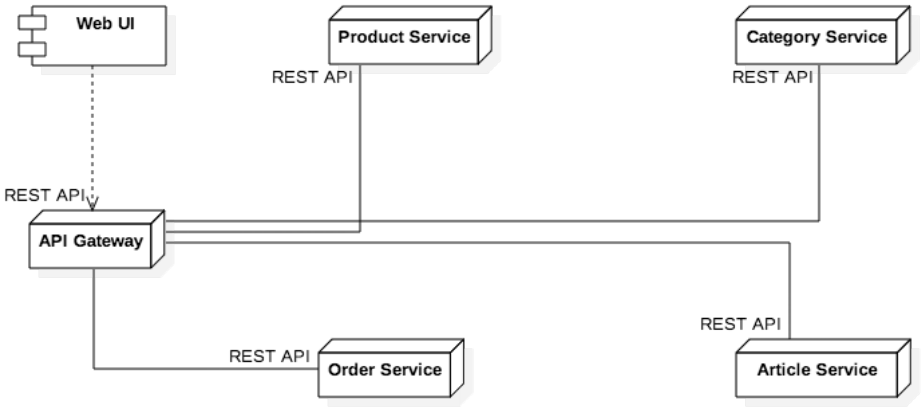
- **N1. Dostupnost přes webové rozhraní** – e-shop bude dostupný přes webové rozhraní pro snadný nákup ze zařízeních s připojením na internet.
- **N2. Responsivní design** – e-shop bude umět reagovat a upravovat své rozhraní dle používaného rozlišení displeje pro pohodlné prohlížení a nákup.
- **N3. Vysoká dostupnost** – systém bude připraven k nasazení na více serverů, aby byl funkční i při výpadku jednoho serveru.
- **N4. Vysoká rychlost** – systém bude rychle reagovat na uživatelské vstupy a umožní jim rychlé procházení. Zároveň bude také umožněno reagování na přetížené části aplikace jejich snadným škálováním.
- **N5. Připravenost na vývoj mobilní aplikace** – systém bude připraven na možný budoucí vývoj mobilní aplikace a jeho jednoduché napojení.
- **N6. Jednoduchá přenositelnost aplikace** – samotnou aplikaci bude možné jednoduše přenášet mezi jednotlivými počítači, toho se bude využívat pro automatické nasazování na více serverů a jednodušší vývoj.

Některé z těchto nefunkčních požadavků, zejména N3 – N6, jsou typickými požadavky mířící na použití architektury mikroslužeb, která umožní uspokojení všech těchto požadavků. Kromě toho je vyvíjená aplikace e-shop a tyto typy aplikací většinou počítají s dalším budoucím vývojem. Co se týče celkové analytické fáze, dá se snadno vyčíst, že není nijak odlišná od vývoje klasického softwaru.

4.2 Návrh

V návrhové části vývoje už ovšem rozdíly existují, a není jich málo. Předně by se tato fáze dala rozdělit na 2 podfáze – **globální**, kdy se provádí návrh pro celou aplikaci, a **lokální**, ve které se řeší návrh pouze pro konkrétní mikroslužbu. Těchto lokálních podfází je tedy tolik, kolik je mikroslužeb a mohou probíhat nezávisle na sobě.

V globální fázi se řeší zejména jak rozdělit aplikaci na mikroslužby, jak mezi sebou budou komunikovat a definují si rozhraní, které bude dostupné pro ostatní mikroslužby. Konkrétní technologie, včetně programovacích jazyků a databázových serverů, se poté řeší v rámci lokálních fází, kdy se na základě funkčních a nefunkčních požadavků vybere ta správná technologie.



Obrázek 4.3: Dekompozice aplikace na mikroslužby

4.2.1 Globální návrhová fáze

4.2.1.1 Dekompozice na jednotlivé mikroslužby

Nejdůležitějším úkolem v globální fázi je správná dekompozice monolitické aplikace na jednotlivé mikroslužby. Výsledný návrh je viditelný na obrázku 4.3. Vyskytují se zde služby pro správu **produktů**, **objednávek**, **článků** a **kategorií**. Naznačena je zde také komponenta API Gateway, stejně tak webové rozhraní, které bude také v samostatném kontejneru a bude nezávisle škálovatelné a nasaditelné.

Kromě toho také návrh ukazuje, že mezi jednotlivými službami se využívá REST API, které je jednoduché a díky protokolu HTTP velmi rozšířené. Pokud by v budoucnu komunikace pomocí REST API nevyhovovala, či zpomalovala chod aplikace, je možné využít jiných protokolů, jako např. gRPC. V této jednoduché aplikaci není nutné řešit komunikaci pomocí protokolů založených na zprávách, proto se zde nevyskytují, nicméně je také možné kdykoliv v budoucnu tuto komunikaci přidat.

4.2.1.2 Definice rozhraní

Jelikož služby spolu budou komunikovat pomocí protokolu HTTP, je nutná definice rozhraní u jednotlivých služeb. Návrh tohoto rozhraní je velmi důležitý pro samotnou implementaci a budoucí vývoj aplikace, jelikož tento návrh výrazně ovlivňuje celkové chování aplikace. Jednotlivá rozhraní, řídící se pravidly pro REST API, poté budou dostupná pro ostatní služby, které budou moci na jednotlivé zdroje přistoupit. Samotná dokumentace a návrh byly vytvořeny

```
## Categories Collection [/{?onlyTop,url}]


+ Parameters
    + onlyTop (optional) - If TRUE, this resource
    returns only top categories (without parent)
    + url (optional) - Finds category by its URL

### List All Categories [GET]

+ Response 200 (application/json; charset=utf-8)

        [
            {
                "id": 1,
                "title": "Mobilni telefony",
                "url": "mobilni-telefony",
                "parentCategory": null,
                "order": 1
            }
        ]
```

Zdrojový kód 4.1: Část definice API `CategoryService`

s pomocí nástroje Apiary[53], výsledek je dostupný ve formátu API Blueprint[54] na přiložené SD kartě. Pro ukázku je viditelná definice rozhraní mikroslužby `CategoryService` popisující zdroj pro získání všech kategorií, k nahlédnutí v ukázce 4.1.

4.2.1.3 Orchestrátory

Z důvodů uvedených v předchozích kapitolách bude použit pro nasazování jednotlivých mikroslužeb vzor, kdy je jedna služba v jednom kontejneru a pro jejich správu bude použit Docker. Aby bylo jednoduché celou aplikaci ovládat včetně spouštění, restartů a přenositelnosti, je možné používat standardní příkazy Dockeru (`docker run`, `docker stop`, atd.). S přibývajícím množstvím kontejnerů je ovšem toto manuální spouštění velmi nepohodlné, je proto výhodné použít podpůrné nástroje, tzv. orchestrátory, které umožňují konfiguraci a spuštění více-kontejnerové aplikace jednoduše. Zároveň řeší problémové oblasti, jako propojení služeb, komunikaci mezi nimi, service discovery, service registry, load balancing, aj.

[53] https://apiary.io/
[54] https://github.com/apiaryio/api-blueprint

Docker Compose Pro lokální vývoj skvěle poslouží nástroj Docker Compose, který byl již stručně představen v kapitole Docker Compose. Jedná se o oficiální nástroj Dockeru, který je nutné nainstalovat zvlášť. Pomocí konfiguračního souboru `docker-compose.yml` se definují jednotlivé služby včetně všech jejich vlastností, jako je image, publikované porty, atd. Zároveň se definují sítě a služby, které do nich patří. Bez definice sítě se vytvoří výchozí síť s názvem `docker_default`, která propojuje všechny služby z konfiguračního souboru. Pomocí příkazu `docker-compose up` poté vývojář jednoduše nastartuje celou aplikaci, včetně všech závislostí, sestavení potřebných obrazů, atd. Tento nástroj je velmi jednoduchý a ideální pro lokální vývoj, pro nasazení na produkci již není vhodný v případě větších aplikací, a to zejména z důvodu nepřítomností grafického rozhraní, takže horšího monitoringu a také nemožnosti vytváření Docker clusterů, stejně tak neřeší load balancing.

Docker Cloud Docker cloud[55] je oficiálním nástrojem Dockeru pro orchestraci kontejnerů. Jak už název napovídá, jedná se o nástroj běžící v cloudu, není tedy nutná jeho instalace na vlastní server. Původně se jednalo o nástroj s názvem Tutum, který Docker odkoupil [41]. Další výhodou je jeho napojení na Docker Hub a GitHub, umožňující mimo jiné automatické buildy, stejně tak jeho jednoduché a čisté prostředí. Toto uživatelské rozhraní je ovšem poměrně pomalé, s větším počtem kontejnerů má Docker Cloud výrazné problémy, kdy nestíhá zpracovávat všechny požadavky a celkový pohyb po aplikaci je velmi pomalý. Kromě toho také používá DNS service discovery, které není tak flexibilní jako jiné typy.

Kubernetes Dalším orchestrátorem pro Docker kontejnery je open-source software od firmy Google[56] s názvem Kubernetes. Tento software je možné nainstalovat na vlastní server nebo využít některé z hostovaných řešení. Umožňuje jednoduché škálování, load balancing, service discovery a zajišťuje celkovou komunikaci mezi službami. Momentálně se jedná o jeden z nejpoužívanějších nástrojů pro orchestraci. Nevýhodou je jeho složitější konfigurace, která je navíc nekompatibilní s konfiguračním souborem pro Docker Compose, takže je potřeba mít rozdílnou konfiguraci pro produkci a lokální vývoj.

Rancher Posledním nástrojem pro orchestraci je Rancher[57], který je také open-source softwarem, nicméně neexistuje hostovaná verze, je nutné si tento nástroj nainstalovat na vlastní server. Aplikace je distribuována jako Docker image, takže instalace je jednoduchá. Rancher je samostatná platforma umožňující správu kontejnerů nad zvoleným cluster frameworkem, jako je Kubernetes, Docker Swarm či Mesos. Má také svůj vlastní framework pro spuštění

[55] https://cloud.docker.com/
[56] https://kubernetes.io/
[57] http://rancher.com/rancher/

více-kontejnerové aplikace, podobný Docker Compose, s názvem Cattle. Soubor `docker-compose.yml` je navíc s ním kompatibilní, takže je možné využít stejnou konfiguraci, jako je pro lokální vývoj. Z důvodu jeho jednoduchosti byl proto pro tuto práci vybrán právě Rancher.

4.2.2 Lokální návrhová fáze

Tato fáze by měla probíhat pro každou mikroslužbu zvlášť. Jelikož v případě ukázkové aplikace je služeb málo a většina má podobné nároky, níže jsou popsány zejména rozdíly mezi jednotlivými službami.

4.2.2.1 Programovací jazyky

Z analýzy nefunkčních požadavků se návrh zaměřoval zejména na jazyky vhodné pro webovou platformu a mikroslužby.

ASP.NET ASP.NET je open-source webový framework firmy Microsoft postavený nad frameworkem .NET. Jedná se o druhou nejrozšířenější technologii z pohledu webových programovacích jazyků na straně serveru [42]. Nicméně je primárně vyvíjena pro platformu Windows, proto můžou nastat problémy při spouštění v Dockeru, který je založen na linuxovém jádru.

PHP Oproti tomu je podpora PHP pro Linux bezproblémová, stejně tak jeho rozšířenost je mnohonásobně vyšší, než je tomu tak u ASP.NET [42]. Jedná se o skriptovací jazyk určený pro tvorbu dynamických webových stránek. Existuje pro něj nepřeberné množství frameworků, mezi kterými si může vybrat každý dle svých požadavků. I přes výrazný nárůst výkonu u verze 7 [43] nedosahuje tak dobrých výsledků jako jiné jazyky, zároveň také není určen primárně pro tvorbu API. Díky dostupným frameworkům je nicméně vhodné jej použít na webové rozhraní pro klienty, jelikož umožňuje např. jednoduchou správu a validaci formulářů, přihlašování, atd. proto jej využívá ukázková aplikace v této práci. V prostředí mikroslužeb se také často objevuje webové rozhraní implementované pouze v Javascriptu a jeho frameworcích (AngularJS, ReactJS).

Node.js Node.js je Javascriptový framework, který je oproti klasickému Javascriptu spouštěn na straně serveru. Je postaven na enginu z prohlížeče Google Chrome a používá neblokující model, díky kterému je Node.js velmi rychlý [44]. Oproti PHP je mnohonásobně výkonnější, jak z pohledu rychlosti zpracování, tak z pohledu HW nároků [45]. Tento rozdíl ve výkonnosti vychází převážně ze dvou různých přístupů těchto jazyků – zatímco PHP používá interpretovanou kompilaci (způsob, kdy se kód předem nekompiluje), kdežto Node.js sází na dynamickou just–in–time kompilaci (kompilace při spuštění programu). Tento framework umožňuje psát kompletní webové aplikace, stejně

tak je velmi vhodný pro tvorbu API, zejména díky frameworkům, jako je např. Express[58], který vývoj výrazně zjednodušuje. Node.js společně s knihovnou Express je z těchto důvodů častou volbou při psaní mikroslužeb, stejně tak je použit v této práci.

Go Kromě výše zmíněného jazyku Node.js se v kontextu s mikroslužbami často objevuje jazyk Go[59], vyvíjený firmou Google. První stabilní verze tohoto mladého jazyka vyšla v roce 2011 [46]. Jelikož je open-source, kromě samotného Googlu se na jeho vývoji podílí mnoho přispěvovatelů. Vychází z jazyka C, ale není tak striktní a je více dynamický. Jeho největší výhodou je jeho rychlost a rychlost kompilace [47].

Všechny mikroslužby z ukázkové aplikace tedy využívají Node.js a pro webové rozhraní byl vybrán jazyk PHP společně s frameworkem Nette.

4.2.2.2 Databáze

Každá mikroslužba z této aplikace potřebuje ukládat data. Pro to slouží databázové servery, které se dají rozdělit do dvou typů – SQL a NoSQL databáze. Rozdíl je jednoduchý, SQL databáze používají dotazovací jazyk SQL, kdežto NoSQL databáze jej nepoužívají. Existuje mnoho druhů databází obou typů a jejich výběr je pro každou mikroslužbu velmi důležitý z hlediska rychlosti a spolehlivosti.

MySQL MySQL je jednou z nejznámějších SQL databází, využívaná na mnoha webech. Zároveň se jedná o nejrozšířenější open-source databázový server [48], pro který existuje mnoho klientských knihoven do nepřeberného množství jazyků.

MariaDB Tato databáze je odnoží předchozího serveru MySQL, kterou vytvořili nespokojení zaměstnanci stojící za původní verzí MySQL [49] a snaží se vývoj směřovat původním směrem, zároveň zachovávají zpětnou kompatibilitu s MySQL. Momentálně je MariaDB co do výkonu rychlejší, než MySQL [50], proto je v práci použita tato její odnož.

MongoDB MongoDB patří mezi nejpopulárnější NoSQL databáze – ukládá svá data ve formátu JSON. Oproti klasickým relačním databázím nemá definované schéma, takže každý dokument může mít jinou strukturu. I přes neexistenci schématu ovšem přebírá některé přístupy z relačních databází, takže

[58] https://expressjs.com/
[59] https://golang.org/

např. tvorba dotazů velmi připomíná MySQL. Jeho největší výhodou je dostupnost mnoha analytických nástrojů, jako např. analýza geografických dat.

Elasticsearch Elasticsearch[60] je velmi populární NoSQL databází vycházející z Apache Lucene. Jedná se o databázi napsanou v Javě, dosahující vysoké rychlosti při vyhledávání a analýze i mezi desetitisíci záznamy [51]. Podobně jako MongoDB má k dispozici mnoho analytických a agregačních příkazů, které fungují velice rychle, kromě toho má také velmi rozsáhlé možnosti pro fulltextové vyhledávání.

Redis Další často používanou NoSQL databází je Redis. Tato databáze je vlastně cache úložiště v paměti RAM, která funguje na principu klíč-hodnota a neumožňuje hlubší analýzu a prohledávání, lze pouze uložit, číst či smazat data na základě klíče. Nicméně díky tomu, že jsou veškerá data uložena v paměti RAM, je přístup k datům extrémně rychlý.

Je tedy ideální pro každou mikroslužbu vybrat tu databázi, která je pro ni nejvýhodnější, často se některé databáze také kombinují (jako např. relační a cache databáze). Z důvodu rychlosti je Elasticsearch použit u `ProductService`, jelikož tato služba bude využívaná často zákazníky pro prohlížení kategorií a produktů a ti vyžadují co nejrychlejší reakci aplikace při prohlížení, filtrování a řazení. Zároveň Elasticsearch umožňuje implementaci jednoduchého fulltextového vyhledvání. U ostatních služeb, které nepotřebují analytické nástroje, jako je `OrderService`, `ArticleService` a `CategoryService`, je použita databáze MariaDB, jako cache server je využit Redis.

4.3 Realizace

4.3.1 Backendové mikroslužby

Většina mikroslužeb zajišťující backendovou část aplikace (jedná se o `OrderService`, `ArticleService`, `CategoryService` a `ProductService`) jsou si podobné, jelikož je každá implementována v jazyku Node.js a pro komunikaci s ostatními službami publikuje REST API zdroje popsané v dokumentaci.

Jak bylo nastíněno v návrhu, používá se framework Express, který poskytuje jednoduché možnosti pro tvorbu API. Kromě něj se v jednotlivých mikroslužbách vyskytují další podpůrné knihovny, jako jsou `body-parser` (sloužící pro práci s různými formáty, zejména JSON), `cookie-parser` (práce s cookie), `config` (umožňující konfigurační soubory pro různé prostředí) a poté klientské knihovny pro jednotlivé databáze (MySQL a Elasticsearch). Všechny tyto nástroje byly instalovány a spravovány pomocí programu `npm`, což je správce závislostí pro programy napsané v Node.js.

[60] https://www.elastic.co/products/elasticsearch

Zajímavé situace jsou k nalezení u `OrderService`, která zajišťuje správu nejen objednávek, ale i košíků. Z pohledu logiky se totiž objednávka od košíku nijak neliší, proto se v této službě košíkem rozumí „nedokončená objednávka“. Po dokončení objednávky se nastaví příznak `finished` na hodnotu `TRUE` a pomocí tohoto příznaku se rozlišují košíky od objednávek. Dále se u `ProductService` využívá kombinací několika vlastností databáze Elasticsearch k docílení řazení, filtrování dle značek a fulltextového vyhledávání.

Aby se využilo co nejvíce předností Dockeru, je nutné, aby již samotný vývoj probíhal s využitím kontejnerů. Nejdříve je tedy nutné definovat, jak bude image vypadat, a to vytvořením Dockerfile. V něm je potřeba nainstalovat vše pro bezproblémový chod programu a zároveň zvolit správnou posloupnost příkazů, aby image nebyla zbytečně velká a aby se využilo co nejvíce cachovací schopnosti Dockeru při sestavení. Dockerfile, který se s mírnými změnami vyskytuje u každé z mikroslužeb, je dostupný na ukázce 4.2. Na něm je viditelná právě správná posloupnost příkazů, kdy se nejdříve zkopíruje soubor `package.json` (řádek 5), ve kterém je definice závislostí. Tento soubor se bude měnit mnohem méně častěji, než samotný kód, proto probíhá jeho zkopírování a samotná instalace závislostí dříve, než se zkopíruje aplikační kód (řádek 18). Takže při změně v kódu se nemusí znovu instalovat všechny závislosti, ty jsou již nacachované Dockerem. Zároveň je zde viditelná také snaha o co nejmenší výslednou velikost, kdy se ve skupině příkazů (začíná na řádku 11) nejdříve nainstaluje `git`, který je potřeba pro spuštění programu `npm`, ale ještě ve stejné instrukci (ihned po instalaci závislostí) se vymaže, takže žádná z vrstev výsledné image `git` neobsahuje. Pokud by se toto sesupení příkazů rozdělilo do jednotlivých instrukcí Dockerfile, image by poté byla zbytečně velká.

Na řádku 7 je poté viditelná instrukce `ARG`, která umožňuje předávat argumenty z příkazové řádky při spuštění buildu. Toho se využívá při sestavení produkční a testovací image, kdy je třeba, aby produkční image neobsahovala vývojářské a testovací knihovny. Nastavením tohoto argumentu na hodnotu `--production` se poté tyto knihovny nenainstalují (řádek 13).

Nakonec je definován hlavní proces kontejneru – pomocí instrukce `CMD` – skript s názvem `entrypoint.sh`, který vždy obsahuje kontrolu přítomnosti databáze a jejího schématu. V případě, že toto schéma není dostupné, vytvoří se a nahrají se také testovací data, takže při prvním spuštění je databáze ihned naplněna. Poté skript spustí samotný samotný Node.js server.

Po úspěšném sestavení image je potřeba propojit kontejner konkrétní služby s databázovým kontejnerem. Tohoto je možné docílit pomocí základních příkazů Dockeru `docker run`, atd. ale jednodušší je využít nástroje Docker Compose, kde se pomocí konfiguračního souboru `docker-compose.yml` nadefinují tyto dva kontejnery. Tento konfigurační soubor je viditelný na ukázce 4.3, ve kterém je definována služba `articles` – zde je určena image, proměnné prostředí, publikované porty a volumes, pomocí kterých se sdílí zdrojové kódy aplikace z hostitelského počítače do kontejneru. Také je zde definována služba pro databázi. Poté je po spuštění příkazu `docker-compose up` vše připraveno

```
1  FROM node:7.8.0-alpine
2
3  RUN mkdir /var/app
4
5  COPY ./package.json /var/app
6
7  ARG NPM_ARG=
8
9  WORKDIR /var/app
10
11 RUN apk update \
12   && apk add --no-cache git \
13   && npm install $NPM_ARG \
14   && apk del git \
15   && rm -rf /var/cache/apk/* \
16   && rm -rf /tmp/*
17
18 COPY . ./
19
20 RUN chmod a+x docker/entrypoint.sh
21
22 CMD ["docker/entrypoint.sh"]
```

Zdrojový kód 4.2: Dockerfile používaný pro sestavení image mikroslužeb

pro vývoj.

Samotné API je implementováno s pomocí již zmiňovaného frameworku Express, kdy se jednoduše definují jednotlivé zdroje a jejich metody za pomocí routeru z tohoto frameworku – `express.Router().get('/articles')`, `express.Router().post('/articles')`, atd. Tyto zdroje vychází z definice API, které bylo definováno v rámci návrhové fáze.

API Gateway API Gateway, které bylo rozebíráno v předchozích částech textu, je také implementováno pomocí frameworku Express. Většina zdrojů této části funguje pouze jako proxy, kdy přeposílá jednotlivé HTTP požadavky na danou mikroslužbu, nicméně některé zdroje, jako např. `/carts/:cartId`, komunikují s více službami a tím šetří síť mezi klienty a backendem aplikace. Stejně tak tato gateway může do budoucna sloužit pro komunikaci s mobilní aplikací a také zajišťovat další aspekty, jako autorizaci, aj.

4.3.2 Databáze

V architektuře mikroslužeb by každá mikroslužba měla mít svou vlastní databázi, která může běžet na jiném serveru. V ukázkové aplikaci se tento požadavek také dodržuje s menším rozdílem, kdy z důvodu jednoduchosti existuje pouze jeden kontejner s databázovým serverem, ve kterém má každá mik-

```
1  version: '2.1'
2
3  services:
4    articles:
5      image: dp-articles:latest
6      environment:
7        - NODE_ENV=development
8      build:
9        context: ..
10       dockerfile: docker/Dockerfile
11     ports:
12       - "8082:3000"
13     volumes:
14       - '../:/var/app'
15     depends_on:
16       - db
17   db:
18     image: mariadb:10.1.22
19     ports:
20       - "3307:3306"
21     environment:
22       - MYSQL_ROOT_PASSWORD=R2%5(z*uR77Zadpb2e
23     volumes:
24       - '../data/db:/var/lib/mysql'
```

Zdrojový kód 4.3: Konfigurační soubor `docker-compose.yml`

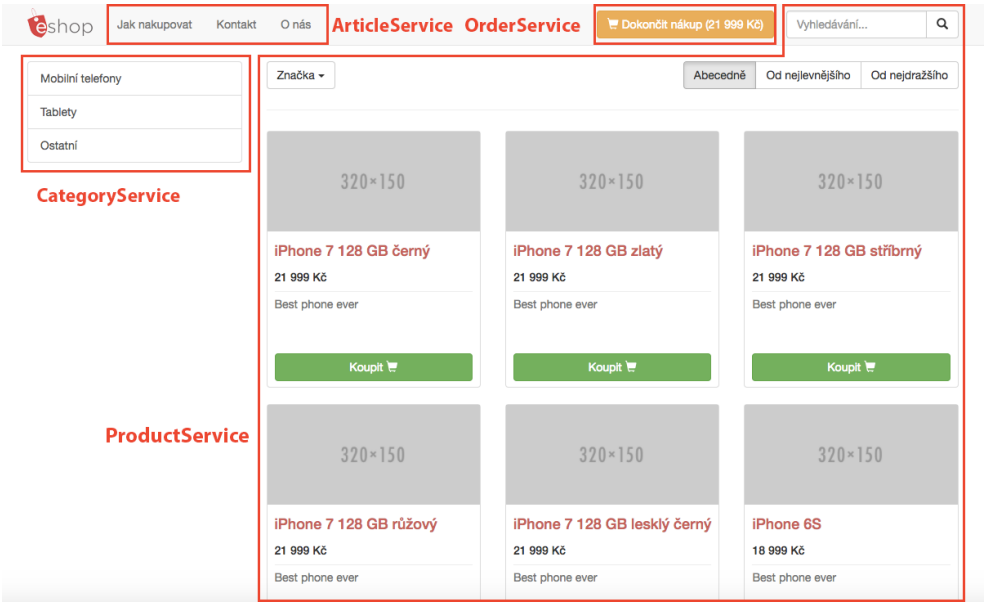
roslužba svou vlastní databázi a také svůj vlastní uživatelský účet. Tím je zajištěna nemožnost jednotlivých služeb zasahovat do jiných dat, než svých vlastních.

4.3.3 Webové rozhraní

Webové rozhraní je implementováno v jazyce PHP společně s využitím frameworku Nette. Jednotlivé mikroslužby se volají skrze HTTP požadavky mířící na API Gateway, tyto požadavky jsou volány pomocí knihovny Guzzle[61], která tuto komunikaci usnadňuje. Pro zajištění responzivity byl využit CSS framework Bootstrap[62].

Dockerfile je mírně odlišný od backendových mikroslužeb, jelikož PHP vyžaduje více nástrojů, nicméně logika je stále stejná – tzn. nejdříve se nainstalují potřebné aplikace, zejména web server Nginx, poté se nakopíruje soubor se závislostmi, pomocí jejich správce (zde Composer) se jednotlivé knihovny nainstalují a teprve poté se kopíruje celý aplikační kód.

[61] https://github.com/guzzle/guzzle
[62] http://getbootstrap.com/



Obrázek 4.4: Webové rozhraní aplikace s viditelnou dekompozicí na jednotlivé mikroslužby

Co se týče spuštění celé aplikace, webové rozhraní samozřejmě vyžaduje veškeré další mikroslužby, které jsou pro běh potřeba. Soubor `docker-compose.yml` proto tedy obsahuje definice všech nutných mikroslužeb, včetně jejich propojení, databází, aj.

Webové rozhraní obsahuje standardní možnosti e-shopových aplikací. Umožňuje návštěvníkům prohlížet zboží v **kategoriích** a **podkategoriích**, v seznamu produktů **filtrovat dle značky a řadit**, procházet **články**, vyhledávat zboží pomocí **fulltextu** a samozřejmě možnost **nakoupit**.

Kromě části, která je viditelná pro zákazníky, je zde také dostupné administrační rozhraní pro správu objednávek, produktů, článků a kategorií, které je dostupné pouze po přihlášení – tato autentizace je řešena na úrovni frameworku Nette, nevyužívá se pro to žádná externí služba.

Zajímavě je vyřešena identifikace zákazníků, kteří nemají možnost se přihlásit, ale při první návštěvě je jim vygenerován unikátní identifikátor, který se poté používá pro vyhledávání aktivních košíků, aj.

Výsledné webové rozhraní aplikace s naznačenou dekompozicí na jednotlivé mikroslužby je viditelné na obrázku 4.4.

4.4 Zabezpečení

Jelikož je dostupná API Gateway, žádný klient nebude přistupovat k jednotlivým službám napřímo, to znamená, že veškeré mikroslužby mohou být skryty

před veřejnou sítí a být dostupné pouze v interní síti Dockeru. Kromě těchto služeb může být skryta i samotná API Gateway, jelikož veškeré požadavky budou vždy mířeny od serverové části webového rozhraní, klient v podobě webového prohlížeče se tedy také nikdy nebude dotazovat Gatewaye napřímo. To znamená, že nikdo mimo síť Dockeru se na tyto komponenty nemůže dostat, tímto je zajištěna mnohem vyšší bezpečnost, jelikož jediná potenciální hrozba tak může pocházet od uživatelských vstupů a tím pádem zneužití např. SQL injection. Toto je nicméně zajištěno tzv. „prepared statements“[63], které tuto hrozbu zažehnávají. V případě NoSQL databáze Elasticsearch tato hrozba nehrozí.

4.5 Testování

Při použití architektury mikroslužeb má každá služba své vlastní testy nezávislé na ostatních. Jelikož všechny služby fungují správně a mají správně definované rozhraní, celková aplikace musí fungovat také. Problém nastává tehdy, pokud nějaká služba vyžaduje data z jiné. V této práci je využito aplikace Apiary, která byla použita pro dokumentaci rozhraní jednotlivých služeb. Tato služba umožňuje u každého zdroje definovat ukázkovou odpověď a poskytuje tzv. mock server, na který může kdokoliv poslat požadavek a dle zvoleného zdroje a metody se mu vrátí definovaný testovací výsledek. Tohoto chování se využívá například u testování API Gatewaye, která vyžaduje přítomnost všech služeb.

Samotné testování je zajištěno s využitím testovacích knihoven Chai[64] a Mocha[65], které umožňují jednoduché testování REST rozhraní. Za pomocí Dockeru a nástroje Docker Compose se definuje konfigurační soubor pro testovací prostředí (`docker-compose.test.yml`), které využívá stejné image jako produkční aplikace (pouze jsou zde dostupné tyto vývojářské a testovací nástroje) a navíc zde databáze obsahuje testovací data a nemá sdílené žádné soubory s hostitelským počítačem, takže při každém spuštění kontejneru databáze obsahuje původní testovací data.

Každý repozitář s mikroslužbou tedy obsahuje svá testovací data, testy a konfigurační soubor pro testovací prostředí. Jedná se zejména o unit testy, které pokrývají většinu zdrojů API, spouští se příkazem `npm test`.

4.6 Nasazování

Celý proces nasazování bývá v dnešní době zjednodušen pomocí CI nástrojů, jeden z nich byl použit i v této práci, konkrétně se jedná o GitLab CI. Po úspěšném dokončení tohoto procesu je potřeba celou aplikaci zprovoznit na

[63] https://dev.mysql.com/doc/refman/5.7/en/sql-syntax-prepared-statements.html
[64] http://chaijs.com/
[65] https://mochajs.org/

produkčním serveru, a to nejlépe za pomoci pokročilého orchestrátoru – v této práci byl použit Rancher společně s cluster frameworkem Cattle.

4.6.1 CI

Konfigurace nástroje GitLab CI je velmi jednoduchá, definují se jednotlivé fáze a v nich skripty, které se v nich spouští. Definice všech fází se uvádí do souboru `.gitlab-ci.yml`. Teprve poté, co jsou všechny skripty z jedné fáze spuštěny a úspěšně dokončeny, přichází na řadu fáze následující.

Každá mikroslužba tedy ve svém repozitáři obsahuje svůj soubor definující proces nasazení. Nejdříve se sestaví produkční a testovací image, pokud vše proběhne v pořádku, nastává testovací fáze, kdy se nejdříve pomocí programu Docker Compose rozběhne testovací prostředí a uvnitř něj se spustí sady testů. Po úspěšném otestování se produkční image poté nahraje do Docker registru, v tomto případě se opět využívá služby GitLab, která poskytuje zdarma privátní registr. Nyní je image připravena k použití na produkčním stroji.

Pokud již je k dispozici spuštěná aplikace, může se nakonfigurovat tak, aby se při změně image v registru automaticky aktualizovala na tuto nejnovější verzi nebo se také tyto změny mohou provádět ručně.

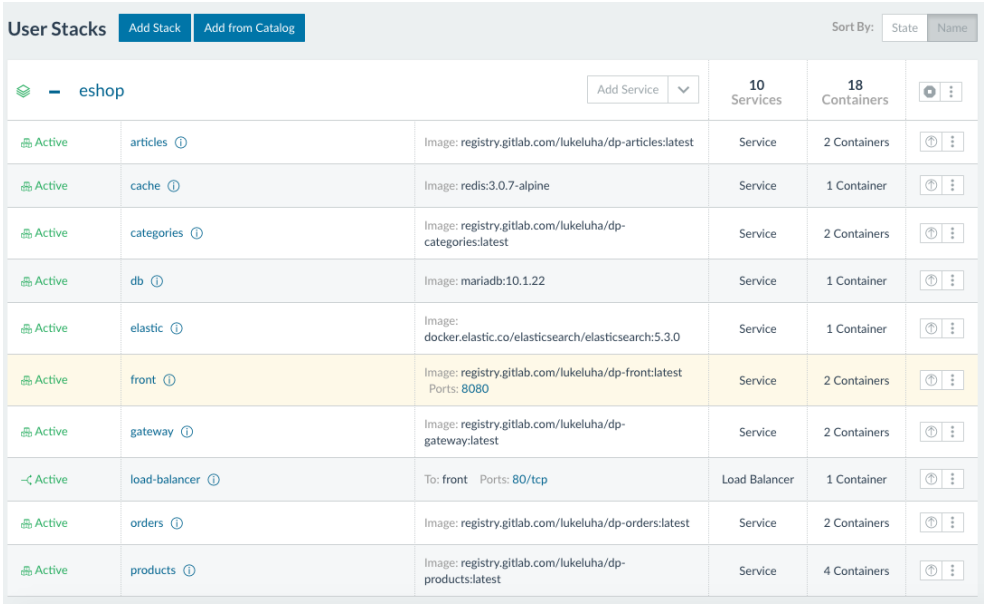
4.6.2 Spuštění a provoz aplikace

Existují dvě možnosti spuštění celé aplikace – první za pomoci jednoduchého a již zmiňovaného nástroje Docker Compose, kterému se předá konfigurační soubor vhodný pro produkční prostředí, viditelný v příloze D. Tento soubor je vhodný pro ostrý provoz, jelikož jednotlivé služby nesdílí žádná data s hostitelským počítačem (samozřejmě kromě databázových služeb) a nepublikují do veřejné sítě žádné porty. Tato možnost je nejvýhodnější pro jednoduché aplikace či spuštění na lokálním počítači.

Druhou možností je využití pokročilého orchestrátoru, který umožňuje mnohem rozmanitější způsoby nastavení, jednodušší monitoring za pomoci grafického rozhraní a tím pádem lepší přehled nad jednotlivými službami, atd.

Open-source platforma Rancher s frameworkem Cattle definuje následující strukturu: **services** (služby), v rámci kterých se nastavují image, proměnné prostředí, atd. a v rámci těchto služeb jsou spouštěny jednotlivé kontejnery. Logicky uspořádaná množina služeb se nazývá **stack**. Ukázková aplikace tedy obsahuje stack s názvem „eshop“, který obsahuje všechny mikroslužby potřebné pro běh celé aplikace.

Jelikož je Cattle kompatibilní s konfiguračními soubory pro Docker Compose, je možné využít tuto konfiguraci, pomocí které se poté spustí celá aplikace. Kromě tohoto souboru obsahující základní možnosti nastavení doplňuje Cattle svůj soubor s názvem `cattle-compose.yml`, který přidává další možnosti (jako například kontroly stavu služeb, škálování, atd.) a rozšiřuje chování



Obrázek 4.5: Rozhraní orchestrátoru Rancher s běžící ukázkovou aplikací

klasického Docker Compose souboru, specifické pro využití s orchestrátorem Rancher. Ukázka prostředí Rancheru z již běžící ukázkové aplikace je viditelná na obrázku 4.5. Tato aplikace byla úspěšně spuštěna na Docker clusteru čítající tři různé cloudové stroje od společnosti Digital Ocean[66].

Kompletní konfigurační soubory pro Rancher jsou k nalezení v příloze E.

4.7 Škálování

Aplikace je tedy úspěšně spuštěna a další fází je její škálování. Díky použitým technologiím je jeho docílení velmi jednoduché, jelikož Docker s pomocí orchestrátorů se postará téměř o vše. Stačí si vybrat zvolenou službu a tu za pomoci konfiguračního souboru či grafického rozhraní škálovat pouhým zvolením čísla reprezentující počet spuštěných kontejnerů dané služby – v konfiguračním soubory je u každé služby řádek `scale: x`, jehož hodnota udává počet spuštěných kontejnerů. Zároveň jsou tyto kontejnery spuštěny na různých serverech, takže je zajištěna i vysoká dostupnost při výpadku jednoho serveru. Vysoká dostupnost může být zajištěna i s přítomností jednoho serveru, na kterém může běžet více kontejnerů dané služby, jelikož jeden kontejner může spadnout kvůli neočekávané chybě, zatímco druhý kontejner poběží dále. V tomto případě se samozřejmě nevyřeší situace při výpadku celého serveru.

Rancher společně s frameworkem Cattle zajišťuje nejjednodušší service discovery, což je DNS service discovery rozebíraná v kapitole 2.6.1.1, v níž je zmi-

[66] https://www.digitalocean.com/

ňováno, že největší nevýhoda této metody tkví v nemožnosti kontrolovat stav daného serveru. To ovšem Rancher řeší tak, že na každém serveru v clusteru nejdříve spustí kontejner s názvem healthcheck. Slovo healthcheck znamená v češtině „kontrola zdraví“ a přesně to má tento kontejner na starosti – dle definice v souboru `rancher-compose.yml` kontroluje stav daného kontejneru a udržuje si u každého z nich informaci, jestli je zdravý, či nikoliv. Definice healthchecku obsahuje typ kontroly (otevřený port, vrácený HTTP kód), timeouty, intervaly kontroly a co dělat v případě, že kontejner je označen jako nezdravý (může se například automaticky restartovat). DNS service discovery poté vrací pouze IP adresy zdravých kontejnerů, čímž je eliminována největší nevýhoda tohoto typu service discovery.

V případě potřeby rozmanitějšího nastavení load balancingu je jedinou možností přidat další kontejner, který bude řešit jen load balancing a bude předsunutý před danou naškálovanou službou (Rancher na to má uzpůsobené webové rozhraní) anebo přejít na jiný typ cluster frameworku.

V rámci Docker Compose je také možné škálování vyřešit podobně (pomocí příkazu `docker-compose scale service=x`), zde se ovšem neřeší kontrola stavu daného kontejneru, je tedy nutné vždy před naškálovanou službu vložit samostatný kontejner s load balancerem, který bude stavy kontrolovat.

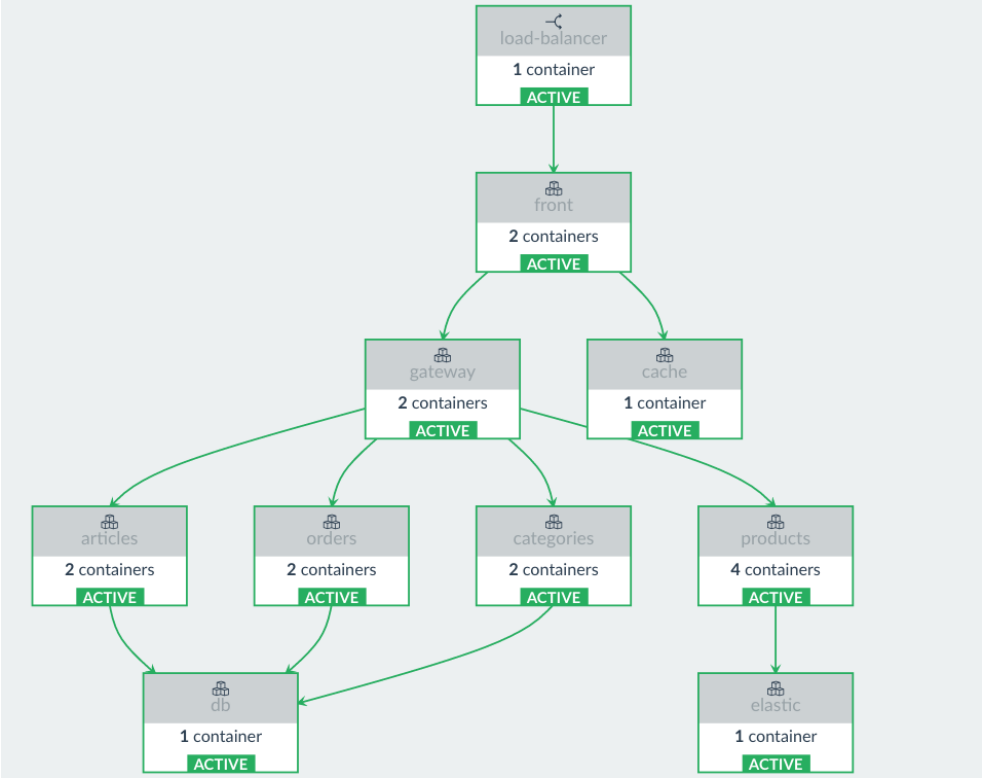
Výsledná architektura celé aplikace včetně finálního škálování je viditelná na obrázku 4.6, ze kterého lze vyčíst, že vstupním bodem aplikace je kontejner obsahující load balancer, který distribuuje požadavky mezi další dva kontejnery webového rozhraní aplikace s názvem `front`. Webové rozhraní poté komunikuje pouze s cache serverem Redis a s ostatními službami skrz API Gateway. Jednotlivé služby jsou poté propojeny s databázovým serverem dle požadavku – tedy `ProductService` s Elasticsearch a ostatní s MariaDB.

4.8 Zhodnocení

Aplikace tedy byla úspěšně implementována a spuštěna s využitím Dockeru a architektury mikroslužeb. Z použití těchto technologií vychází několik výhod a nevýhod, stejně tak praktických důsledků a situací. Některé z nich byly rozebírány v předchozích kapitolách Architektura založená na mikroslužbách a Docker a kontejnery, nicméně ukázka na konkrétním příkladu je vždy reprezentativnější.

4.8.1 Škálování a vysoká dostupnost

Nejspíše největší výhodou architektury mikroslužeb je možnost škálování pouze té služby, u které je to třeba, na rozdíl od monolitu, kdy se vždy musí škálovat celá aplikace, včetně webového rozhraní. U ukázkového e-shopu je nutné, aby celá aplikace byla vždy vysoce dostupná, aby provozovatel nepřicházel o zákazníky a peníze. Toto je zajištěno škálováním všech služeb na minimální počet 2 kontejnerů, které jsou navíc vždy na různých serverech v clusteru.



Obrázek 4.6: Výsledná architektura ukázkové aplikace

Dalším aspektem je škálování těch částí, u kterých je pravděpodobná vysoká zátěž či nárok na výkon. Kupříkladu `ProductService`, která se stará o výpis, řazení a filtrování produktů, bude hojně používána návštěvníky stránek, počítá se s její vysokou návštěvností a tím pádem i vyššími nároky na hardware. Proto je tato služba v příkladu naškálována na 4 kontejnery, které jsou opět rovnoměně rozmístěny mezi servery v clusteru. Zároveň se tyto náročnější služby mohou spouštět pouze na výkonnějších serverech, oproti tomu méně využívané, jako např. `ArticleService`, mohou být spuštěny pouze na méně výkonných strojích. V monolitické aplikaci není možné takovýchto výhod využívat.

Nejvýraznější nevýhodou škálování u samotné architektury mikroslužeb je nutnost zajištění service discovery a service registry, nicméně s využitím Dockeru a orchestrátorů je i o tuto problematiku postaráno.

4.8.1.1 Škálování webového klienta

Samostatnou kapitolou by se dalo nazvat škálování webového klienta, a to z několika důvodů. Zaprvé je potřeba před něj vždy vložit plnohodnotný load balancer, na který povedou DNS záznamy domény a který bude rovnoměrně

```
load-balancer:
    scale: 1
    start_on_create: true
    lb_config:
      certs: []
      port_rules:
      - path: ''
        priority: 1
        protocol: http
        service: front
        source_port: 80
        target_port: 80
```

Zdrojový kód 4.4: Konfigurace load balanceru

distribuovat požadavky návštěvníků. S pomocí Rancheru je toto velmi jednoduše vyřešeno přidáním dalšího kontejneru s definicí load balanceru, ukázka je dostupná v konfiguračním souboru 4.4, na kterém je viditelné nastavení protokolu, zdrojového portu a cílového portu společně s cílovou službou (zde služba `front`). Stejný problém se musí řešit i u monolitické aplikace, jelikož při škálování celé aplikace se klonuje i webové rozhraní, takže i zde se musí vložit load balancer. Nicméně Rancher či jiné orchestrátory umožňují velmi jednoduchou správu této komponenty, jakékoliv změny, konfigurace, přesměrování, aj. vyřeší za administrátora automaticky.

Dalším problémem při škálování webového klienta jak u architektury mikroslužeb, tak u monolitu, je úložiště tzv. sessions. Jelikož je HTTP bezestavový protokol, webové servery si ve výchozím stavu ukládají identifikátory návštěvníků a jejich sessions na předem určené místo na disku. Pokud ovšem zde existuje více webových rozhraní, každý server bude mít místo na svém disku a návštěvník poté nebude správně identifikován, protože při každém načtení stránky jej bude obsluhovat jiný web server. Vyřešení tohoto problému je poměrně jednoduché, stačí přidat jakékoliv centrální úložiště těchto sessions, ať už se jedná o sdílené síťové úložiště, či cache server. V ukázkové aplikaci bylo využito cache úložiště Redis, do kterého jednotlivé kontejnery s webových rozhraním ukládaly uživatelská data – tím se informace o návštěvnících sdílely a nenastávaly problémy.

4.8.2 Komunikace

Komunikace mezi službami je klíčovou součástí aplikace implementovanou v architektuře mikroslužeb. Je potřeba zajistit, aby se služby mezi sebou mohly jednoduše a rychle dorozumívat. S použitím Dockeru je zajištění tohoto požadavku mnohem jednodušší, než bez jeho použití. Umožňuje totiž nadefinovat interní sítě, do kterých jednotlivé služby spadají a samotnou komunikaci včetně service discovery si již řeší sám. Na administrátorovi je poté

pouze napsat správný konfigurační soubor, který toto umožní. Jak je viditelné na konfiguraci e-shopové aplikace, viz příloha D, zde jsou definované jednotlivé služby a ve výchozím stavu jsou všechny tyto služby mezi sebou viditelné, není tedy třeba dalšího nastavování.

V monolitické aplikaci není třeba žádnou komunikaci řešit, jelikož veškeré volání jiných komponent je zajištěno na úrovni jazykového volání kódu.

4.8.3 Autentizace a autorizace

Řešení autentizace a autorizace velmi závisí na tom, na kterou službu se tato problematika deleguje. V ukázkové aplikaci se vše řeší na úrovni webového rozhraní v jazyku PHP, jelikož žádná z dalších služeb není veřejné dostupná a vstupním bodem je tedy toto rozhraní.

Pokud ovšem už bude veřejně přístupná např. API Gateway z důvodu přítomnosti dalších klientů, jako je mobilní aplikace, je potřeba tento problém řešit na této komponentě. Jelikož je k ní ovšem přihlášeno více klientů, samotné řešení není snadné. Je možné využít generování přihlašovacích tokenů pro jednotlivé klienty anebo také použití autorizačních protokolů třetích stran, jako je např. OAuth[67], který je navržen právě pro tento problém.

Monolitická aplikace ze své podstaty nemá potřebu řešit distribuovanou autentizaci a autorizaci, stačí využít funkcionalit jednotlivých web serverů a programovacích jazyků.

4.8.4 Logování

Sdružování, ukládání a procházení záznamů činnosti, tzv. logů, jednotlivých komponent aplikace je stěžejní součástí při hledání chyb a monitoringu. V každé aplikaci existuje několik částí, které tyto logy tvoří (web server, databázový server, cache server, ...). U monolitu se většinou výstupy těchto logů nasměrují do nějakého souboru, který se poté dá jednoduše číst. U aplikace založené na mikroslužbách zde ovšem vzniká mnoho samostatných komponent, které logy tvoří a je potřeba je nějak agregovat. S využitím Dockeru je zajištění tohoto problému opět o něco snažší.

Nejjednodušší možností, jak vyřešit agregaci logů všech kontejnerů, je využití již připravené image s názvem logspout[68], jejíž jediným cílem je sesbírání logů všech kontejnerů a jejich přeposílání jinam – může se jednat o tcp či udp port, standardní výstup, atd. Logspout se ke všem kontejnerům dostane zpřístupněním socketu Docker daemona, takže vždy má přehled o všech spuštěných kontejnerech. Zároveň je možné do těchto logů přidat i identifikátory kontejnerů, takže výsledné logy jsou přehledné a jejich zdroje jsou také viditelné.

[67] https://oauth.net/

[68] https://hub.docker.com/r/gliderlabs/logspout/

```
1 logspout:
2   image: gliderlabs/logspout
3   command: 'udp://logstash:5000'
4   links:
5     - logstash
6   volumes:
7     - '/var/run/docker.sock:/tmp/docker.sock'
```

Zdrojový kód 4.5: Konfigurace nástroje logspout

Samotné nastavení nástroje logspout je velmi jednoduché a viditelné na ukázce 4.5.

Kromě nevýhod popsaných výše se zde vyskytuje ještě jeden hlavní problém, na který je třeba při použití architektury mikroslužeb a Dockeru brát zřetel. Dekompozicí na jednotlivé mikroslužby se sice snižuje komplexita aplikace, nicméně propojení a konfigurace všech funkčních částí je mnohem složitější, než při použití monolitického způsobu vývoje. Je tedy proto potřeba veškeré komponenty správně nastavit, udržovat a zejména monitorovat, aby se při případném problému mohla co nejdříve nalézt chyba a sjednat náprava. Před každým návrhem takovéto aplikace je tedy třeba dobře promyslet, zdali je nutné tuto architekturu použít.

Závěr

V rámci této diplomové práce byla provedena analýza základních možností škálovatelnosti webových aplikací, v rámci které bylo zjištěno, že tyto varianty nejsou vhodné pro situace, kdy je třeba škálovat pouze některé části webových aplikací.

Proto text dále detailně rozebírá moderní architekturu založenou na mikroslužbách (známá také pod anglickým pojmem microservices architecture), která je pro tuto jemnější granularitu celé aplikace výhodná.

Další kapitola této práce se podrobně věnuje kontejnerové platformě Docker, která je pro správu jednotlivých mikroslužeb ideální, zároveň jsou zde popsány techniky propojení těchto dvou technologií.

Pro demonstraci tohoto propojení byla v rámci práce provedena analýza, návrh a implementace webové aplikace pro online nakupování, která se dle funkčních a nefunkčních požadavků blížila reálným specifikacím pro provoz takovéhoto portálu, stejně tak tyto požadavky mířily na použití architektury mikroslužeb. Jednotlivé mikroslužby také obsahují sady testů a konfiguraci pro CI nástroj, který zajišťuje automatické sestavení, testování a nasazení na produkční server. Tato aplikace byla realizována pomocí jazyků Node.js, PHP a databází MariaDB, Elasticsearch a Redis. Využitím předností technologie Docker společně s orchestrátorem Rancher byla aplikace úspěšně nasazena na cluster skládající se z několika cloudových serverů. Na tomto clusteru se poté jednotlivé služby škálovaly dle jejich předpokládaného využití a nároků tak, aby byla zajištěna vysoká dostupnost aplikace společně s její vysokou rychlostí.

Díky jednoduché distribuci Docker obrazů je také možné celou aplikaci spustit pomocí jednoho příkazu na jakémkoliv počítači, který má tento nástroj nainstalován.

Zároveň jsou zde popsány také praktické důsledky při použití těchto dvou technologií, včetně výhod, nevýhod, problémů a návrhů na jejich řešení.

Použité zdroje

[1] *Techniky vysoké dostupnosti I.* [online]. prof. Ing. Pavel Tvrdík, CSc., Ing. Jiří Kašpar, Fakulta informačních technologií ČVUT. [vid. 24. 3. 2017] Dostupné z: https://edux.fit.cvut.cz/courses/MI-POA/_media/lectures/mi-poa06.pdf

[2] ENGATES, John. *7 Stages of Scaling Web Applications* [online]. [vid. 24. 3. 2017] Dostupné z: https://www.slideshare.net/davemitz/7-stages-of-scaling-web-applications/5-Scalability_Defined_ulliWhat_scalability_is

[3] NGINX Inc. *What is DNS load balancing?* [online]. [vid. 24. 3. 2017] Dostupné z: https://www.nginx.com/resources/glossary/dns-load-balancing/

[4] RAINVILLE, Shane. *Layer 4 Load Balancing with HAProxy* [online]. [vid. 28. 3. 2017] Dostupné z: http://www.serverlab.ca/tutorials/linux/network-services/layer-4-load-balancing-with-haproxy/

[5] BELL Charles, Mats KINDAHL a Lars THALMANN. *MySQL High Availability.* Sebastopol: O'Reilly Media, Inc., 2010. ISBN 978-0-596-80730-6

[6] DigitalOcean Inc. *How To Set Up MySQL Master-Master Replication* [online]. [vid. 28. 3. 2017] Dostupné z: https://www.digitalocean.com/community/tutorials/how-to-set-up-mysql-master-master-replication

[7] Oracle Corporation. *Using Replication for Scale-Out* [online]. [vid. 28. 3. 2017] Dostupné z: https://dev.mysql.com/doc/refman/5.7/en/replication-solutions-scaleout.html

[8] Elasticsearch BV. *Basic Concepts* [online]. [vid. 28. 3. 2017] Dostupné z: https://www.elastic.co/guide/en/elasticsearch/reference/current/_basic_concepts.html

[9] ABBOT Martin a FISHER Michael. *Scalability Rules: 50 Principles for Scaling Web Sites*. Crawfordsville: Addison-Wesley Professional, 2017. ISBN 978-0-13-443160-4

[10] VITVAR, Tomáš. *Web 2.0: HATEOAS, Scalability and Description* (přednáška). Praha: Fakulta informačních technologií ČVUT, 14. 3. 2016

[11] NGINX Inc. *WHAT IS WEB ACCELERATION?* [online]. [vid. 30. 3. 2017] Dostupné z: `https://www.nginx.com/resources/glossary/web-acceleration/`

[12] FOWLER, Martin. *Microservices* [online]. [vid. 13. 2. 2017] Dostupné z: `https://martinfowler.com/articles/microservices.html`

[13] NEWMAN, Sam. *Building Microservices: designing fine-grained systems*. Sebastopol: O'Reilly Media, Inc., 2015. ISBN 978-1-491-95035-7

[14] COCKBURN, Alistair. *Hexagonal architecture* [online]. [vid. 10. 2. 2017] Dostupné z: `http://alistair.cockburn.us/Hexagonal+architecture`

[15] W3C. *Web Services Glossary* [online]. [vid. 14. 2. 2017] Dostupné z: `https://www.w3.org/TR/ws-gloss/`

[16] WILLIAMS, Alex. Applications and Microservices with Docker and containers. *The Docker and Container Ecosystem eBook Series* [online]. [vid. 14. 2. 2017] Dostupné z: `https://thenewstack.io/ebookseries/`

[17] RICHARDSON, Chris a SMITH, Floyd . *Microservices: From Design to Deployment*. NGINX, Inc. [online]. [vid. 30. 9. 2016] Dostupné z: `https://www.nginx.com/resources/library/designing-deploying-microservices/`

[18] gRPC. *About gRPC* [online]. [vid. 2. 5. 2017] Dostupné z: `http://www.grpc.io/about/`

[19] ABBOTT, Martin a FISHER, Michael. *The Art of Scalability: Scalable Web Architecture, Processes, and Organizations for the Modern Enterprise*. Addison Wesley, 2015. ISBN 978-0134032801

[20] RICHARDSON, Chris. *API Gateway Pattern* [online]. [vid. 15. 2. 2017] Dostupné z: `http://microservices.io/patterns/apigateway.html`

[21] NADAREISHVILI Irakli, Ronnie MITRA, Matt MCLARTY a Mike AMUNDSEN. *Microservices Architecture: Aligning principles, practices and culture*. Sebastopol: O'Reilly Media, Inc., 2016. ISBN 978-1-491-95625-0

[22] VITVAR, Tomáš. *Web 2.0: Representational State Transfer* (přednáška). Praha: Fakulta informačních technologií ČVUT, 29. 2. 2016

[23] Red Hat, Inc. *Understanding Linux containers* [online]. [vid. 14. 3. 2017] Dostupné z: https://www.redhat.com/en/containers

[24] Docker Inc. *What is Docker* [online]. [vid. 10. 10. 2016] Dostupné z: https://www.docker.com/what-docker

[25] MOUAT, Adrian. *Using Docker: Developing and deploying software with containers*. Sebastopol: O'Reilly Media, Inc., 2016. ISBN 978-1-491-91576-9

[26] Computer Hope. *Chroot* [online]. [vid. 10. 10. 2016] Dostupné z: http://www.computerhope.com/jargon/c/chroot.htm

[27] Oracle Corporation. *Oracle Solaris Zones* [online]. [vid. 14. 3. 2017] Dostupné z: https://docs.oracle.com/cd/E18440_01/doc.111/e18415/chapter_zones.htm

[28] Linux Kernel Organization, Inc. *CGROUPS* [online]. [vid. 14. 3. 2017] Dostupné z: https://www.kernel.org/doc/Documentation/cgroup-v1/cgroups.txt

[29] Docker Inc. *Company* [online]. [vid. 14. 3. 2017] Dostupné z: https://www.docker.com/company

[30] Docker Inc. *Docker Overview* [online]. [vid. 14. 3. 2017] Dostupné z: https://docs.docker.com/engine/understanding-docker/

[31] Docker Inc. *Docker Swarm* [online]. [vid. 14. 3. 2017] Dostupné z: https://docs.docker.com/engine/swarm/

[32] Docker Inc. *Docker Compose* [online]. [vid. 14. 3. 2017] Dostupné z: https://docs.docker.com/compose/

[33] Docker Inc. *Announcing Docker Enterprise Edition* [online]. [vid. 15. 3. 2017] Dostupné z: https://blog.docker.com/2017/03/docker-enterprise-edition/

[34] CHANEZON, Patrick a Docker Inc. *DOCKER FOR MAC AND WINDOWS BETA: THE SIMPLEST WAY TO USE DOCKER ON YOUR LAPTOP* [online]. [vid. 15. 3. 2017] Dostupné z: https://blog.docker.com/2016/03/docker-for-mac-windows-beta/

[35] Docker Inc. *Use Docker Machine to provision hosts on cloud providers* [online]. [vid. 15. 3. 2017] Dostupné z: https://docs.docker.com/machine/get-started-cloud/

[36] Docker Inc. *Docker 1.12: Now with built-in orchestration!* [online]. [vid. 15. 3. 2017] Dostupné z: https://blog.docker.com/2016/06/docker-1-12-built-in-orchestration/

[37] Docker Inc. *Best practices for writing Dockerfiles* [online]. [vid. 21. 3. 2017] Dostupné z: https://docs.docker.com/engine/userguide/eng-image/dockerfile_best-practices/

[38] Docker Inc. *Legacy container links* [online]. [vid. 21. 3. 2017] Dostupné z: https://docs.docker.com/engine/userguide/networking/default_network/dockerlinks/

[39] Docker Inc. *Limit a container's resources* [online]. [vid. 22. 3. 2017] Dostupné z: https://docs.docker.com/engine/admin/resource_constraints/

[40] MUNK, Tobias. *Bug Report: File access in mounted volumes extremely slow* [online]. [vid. 3. 5. 2017] Dostupné z: https://github.com/docker/for-mac/issues/77

[41] JOHNSTON, Scott a Docker Inc. *DOCKER ACQUIRES TUTUM TO HELP IT TEAMS DEPLOY AND MANAGE PRODUCTION APPS* [online]. [vid. 6. 4. 2017] Dostupné z: https://blog.docker.com/2015/10/docker-acquires-tutum/

[42] Q-Success. *Usage of server-side programming languages for websites* [online]. [vid. 6. 4. 2017] Dostupné z: https://w3techs.com/technologies/overview/programming_language/all

[43] Zend Technologies Ltd. *Turbocharging the Web with PHP 7* [online]. [vid. 6. 4. 2017] Dostupné z: http://www.zend.com/en/resources/php7_infographic

[44] Node.js Foundation. *Node.js* [online]. [vid. 6. 4. 2017] Dostupné z: https://nodejs.org/en/

[45] HostingAdvice.com. *Comparing Node.js vs PHP Performance* [online]. [vid. 6. 4. 2017] Dostupné z: http://www.hostingadvice.com/blog/comparing-node-js-vs-php-performance/

[46] Google, Inc. *Pre-Go 1 Release History* [online]. [vid. 6. 4. 2017] Dostupné z: https://golang.org/doc/devel/pre_go1.html

[47] Software in the Public Interest, Inc. *Go programs versus Node.js* [online]. [vid. 6. 4. 2017] Dostupné z: https://benchmarksgame.alioth.debian.org/u64q/compare.php?lang=go&lang2=node

[48] solid IT gmbh. *DB-Engines Ranking* [online]. [vid. 6. 4. 2017] Dostupné z: http://db-engines.com/en/ranking

[49] MariaDB Foundation. *About MariaDB* [online]. [vid. 6. 4. 2017] Dostupné z: https://mariadb.org/about/

[50] MariaDB Foundation. *Performance evaluation of MariaDB 10.1 and MySQL 5.7.4-labs-tplc* [online]. [vid. 6. 4. 2017] Dostupné z: https://mariadb.org/performance-evaluation-of-mariadb-10-1-and-mysql-5-7-4-labs-tplc/

[51] Elasticsearch. *The Heart of the Elastic Stack* [online]. [vid. 6. 4. 2017] Dostupné z: https://www.elastic.co/products/elasticsearch

[52] Node.js Foundation. *Node.js*. [software]. [vid. 18. 3. 2017]. Dostupné z: https://nodejs.org/en/

[53] npm, Inc. *npm*. [software]. [vid. 18. 3. 2017]. Dostupné z: https://www.npmjs.com/

[54] Node.js Foundation. *Express*. [software]. [vid. 18. 3. 2017]. Dostupné z: https://expressjs.com/en/starter/installing.html

[55] GEURSEN, Piet. *express-generator-api*. [software]. [vid. 18. 3. 2017]. Dostupné z: https://www.npmjs.com/package/express-generator-api

[56] WILSON, Doug. *body-parser*. [software]. [vid. 18. 3. 2017]. Dostupné z: https://www.npmjs.com/package/body-parser

[57] WILSON, Doug. *cookie-parser*. [software]. [vid. 18. 3. 2017]. Dostupné z: https://www.npmjs.com/package/cookie-parser

[58] VELICHKOV, Simeon. *Request*. [software]. [vid. 18. 3. 2017]. Dostupné z: https://www.npmjs.com/package/request

[59] WEST, Loren. *config*. [software]. [vid. 18. 3. 2017]. Dostupné z: https://www.npmjs.com/package/config

[60] LUER, Jake. *Chai Assertion Library*. [software]. [vid. 18. 3. 2017]. Dostupné z: http://chaijs.com/

[61] LUER, Jake. *chai-http*. [software]. [vid. 18. 3. 2017]. Dostupné z: https://www.npmjs.com/package/chai-http

[62] DA SILVA CONTIN, David. *Mocha*. [software]. [vid. 18. 3. 2017]. Dostupné z: https://mochajs.org/

[63] WILSON, Doug. *morgan*. [software]. [vid. 18. 3. 2017]. Dostupné z: https://www.npmjs.com/package/morgan

[64] Elasticsearch. *Elasticsearch node.js client*. [software]. [vid. 18. 3. 2017]. Dostupné z: https://www.npmjs.com/package/elasticsearch

[65] WILSON, Doug. *Mysql node.js client*. [software]. [vid. 18. 3. 2017]. Dostupné z: https://www.npmjs.com/package/mysql

[66] The PHP Group. *PHP*. [software]. [vid. 18. 3. 2017]. Dostupné z: http://php.net/

[67] ADERMANN, Nils a Jordi BOGGIANO. *Composer*. [software]. [vid. 18. 3. 2017]. Dostupné z: https://getcomposer.org/download/

[68] Twitter, inc. *Bower*. [software]. [vid. 18. 3. 2017]. Dostupné z: http://bower.io/

[69] Nette Foundation. *Nette Framework*. [software]. [vid. 18. 3. 2017]. Dostupné z: https://nette.org/

[70] PROCHÁZKA, Filip. *Kdyby/Redis*. [software]. [vid. 18. 3. 2017]. Dostupné z: https://github.com/Kdyby/Redis

[71] DOWLING, Michael. *Guzzle*. [software]. [vid. 18. 3. 2017]. Dostupné z: https://github.com/guzzle/guzzle

[72] Twitter, Inc. *Bootstrap*. [software]. [vid. 1. 7. 2016]. Dostupné z: http://getbootstrap.com/

[73] Docker Inc. *Docker*. [software]. [vid. 18. 3. 2017]. Dostupné z: https://www.docker.com/

[74] Docker Inc. *Docker Compose*. [software]. [vid. 18. 3. 2017]. Dostupné z: https://docs.docker.com/compose/

[75] Rancher Labs, Inc. *Rancher*. [software]. [vid. 18. 3. 2017]. Dostupné z: http://rancher.com/

[76] GitLab Inc. *GitLab*. [software]. [vid. 18. 3. 2017]. Dostupné z: https://gitlab.com/

[77] Google, Inc. *Google Chrome*. [software]. [vid. 1. 7. 2016]. Dostupné z: https://www.google.com/chrome/browser/desktop/

[78] Jetbrains s.r.o. *PHP Storm*. [software]. [vid. 1. 7. 2016]. Dostupné z: https://www.jetbrains.com/phpstorm/

[79] Jetbrains s.r.o. *PHP Storm*. [software]. [vid. 1. 7. 2016]. Dostupné z: https://www.jetbrains.com/webstorm/

[80] OLŠÁK, Petr. *vlna*. [software]. [vid. 1. 5. 2017]. Dostupné z: http://ftp.linux.cz/pub/tex/local/cstug/olsak/vlna/

[81] Sequel Pro, CocoaMySQL. *Sequel Pro*. [software]. [vid. 1. 7. 2016]. Dostupné z: https://sequelpro.com/download

[82] MariaDB Foundation. *MariaDB*. [software]. [vid. 18. 3. 2017]. Dostupné z: https://mariadb.org/

[83] Elasticsearch. *Elasticsearch*. [software]. [vid. 18. 3. 2017]. Dostupné z: `https://www.elastic.co/guide/index.html`

[84] SANFILIPPO, Salvatore a Pieter NOORDHUIS. *Redis*. [software]. [vid. 1. 5. 2017]. Dostupné z: `https://redis.io/`

Příloha A

Seznam použitých zkratek

AJAX Asynchronous JavaScript and XML

API Application Programming Interface

CDN Content Delivery Network

CI Continuous Integration

CPU Central Processing Unit

CSS Cascading Style Sheets

DNS Domain Name System

ESB Enterprise Service Bus

HTML HyperText Markup Language

HTTP HyperText Transfer Protocol

IPC Inter–Process Communication

JS JavaScript

LXC Linux Containers

MVC Model-View-Controller

OS Operační systém

PHP PHP: Hypertext Preprocessor

RAM Random Access Memory

REST Representational State Transfer

SOA Service Oriented Architecture

SQL Structured Query Language

SSH Secure Shell

URL Uniform Resource Locator

VM Virtual Machine

XML eXtensible Markup Language

Příloha **B**

Obsah přiložené SD karty

- `analysis` ..... výstupy z analytické části ukázkové aplikace
- `api-design` ..... API dokumentace ukázkové aplikace
- `readme.txt` ..... stručný popis obsahu SD karty
- `src`
  - `app` ..... zdrojové kódy ukázkové aplikace
  - `thesis` ..... zdrojová forma práce ve formátu LaTeX
- `text` ..... text práce
  - `thesis.pdf` ..... text práce ve formátu PDF
  - `zadani.pdf` ..... schválené zadání práce

Příloha C

Instalační příručka

Pro instalaci a spuštění ukázkové aplikace i jednotlivých mikroslužeb je třeba mít nainstalovány tyto programy:

- Docker 1.10 a novější
- Docker Compose 1.9 a novější
- moderní webový prohlížeč

Tyto programy jsou dostupné pro všechny nejrozšířenější platformy. Pro spuštění samotné aplikace je třeba přejít do složky `src/app`, ve které je soubor `docker-compose.yml`. Pro spuštění aplikace stačí zadat příkaz `docker-compose up`, po několika desítkách vteřin by aplikace společně s testovacími daty měla být dostupná na adrese `localhost:8080`.

Administrační sekce je poté dostupná na adrese `localhost:8080/admin` – s uživatelským jménem `admin` a heslem `123` je možné se do této sekce přihlásit.

Pro spuštění pouze konkrétní mikroslužby stačí přejít do složky požadované služby, ve které se nachází složka docker. Zde se opět po spuštění příkazu `docker-compose up` spustí konkrétní mikroslužba ve vývojářském módu – jakákoliv změna ve zdrojovém kódu se ihned promítne do kontejneru, není třeba spouštět znovu build.

Příloha D

Konfigurační soubory aplikace pro Docker Compose

```
1  version: '2.1'
2
3  services:
4    front:
5      image: registry.gitlab.com/lukeluha/dp-front:latest
6      container_name: front
7      ports:
8        - "8080:80"
9      depends_on:
10       - gateway
11       - cache
12   gateway:
13     image: registry.gitlab.com/lukeluha/dp-gateway:latest
14     container_name: gateway
15     restart: always
16     environment:
17       - NODE_ENV=production
18     depends_on:
19       - products
20       - articles
21       - categories
22   articles:
23     image: registry.gitlab.com/lukeluha/dp-articles:latest
24     container_name: articles
25     restart: always
26     environment:
27       - NODE_ENV=production
28     depends_on:
29       db:
30         condition: service_healthy
31   categories:
```

```
32      image: registry.gitlab.com/lukeluha/dp-categories:latest
33      container_name: categories
34      restart: always
35      environment:
36        - NODE_ENV=production
37      depends_on:
38        db:
39          condition: service_healthy
40    orders:
41      image: registry.gitlab.com/lukeluha/dp-orders:latest
42      container_name: orders
43      restart: always
44      environment:
45        - NODE_ENV=production
46      depends_on:
47        db:
48          condition: service_healthy
49    products:
50      image: registry.gitlab.com/lukeluha/dp-products:latest
51      container_name: products
52      restart: always
53      environment:
54        - NODE_ENV=production
55      depends_on:
56        elastic:
57          condition: service_healthy
58    elastic:
59      image: docker.elastic.co/elasticsearch/elasticsearch:5.3.0
60      container_name: elastic
61      environment:
62        - xpack.security.enabled=false
63        - "ES_JAVA_OPTS=-Xms512m -Xmx512m"
64      volumes:
65        - "./data/elastic:/usr/share/elasticsearch/data"
66      healthcheck:
67        test: curl -s localhost:9200 > /dev/null
68        interval: 10s
69        timeout: 10s
70        retries: 12
71    db:
72      image: mariadb:10.1.22
73      container_name: db
74      environment:
75        - MYSQL_ROOT_PASSWORD=R2%5(z*uR77Zadpb2e
76      volumes:
77        - './data/db:/var/lib/mysql'
78      healthcheck:
79        test: bash -c "(echo > /dev/tcp/db/3306) >/dev/null 2>&1"
80        interval: 10s
```

```
81        timeout: 10s
82        retries: 12
83    cache:
84      image: redis:3.0.7-alpine
85      container_name: cache
```

Příloha E

Konfigurační soubory aplikace pro Rancher

E.1 docker-compose.yml

```
version: '2'

services:
  cache:
    image: redis:3.0.7-alpine
    stdin_open: true
    tty: true
  elastic:
    mem_limit: 1000000000
    image: docker.elastic.co/elasticsearch/elasticsearch:5.3.0
    environment:
      ES_JAVA_OPTS: -Xms512m -Xmx512m
      bootstrap.memory_lock: 'true'
      cluster.name: docker-cluster
      xpack.security.enabled: 'false'
    ulimits:
      memlock:
        hard: -1
        soft: -1
      nofile:
        hard: 65536
        soft: 65536
    volumes:
      - "/usr/share/elasticsearch/data:/usr/share/elasticsearch/data"
  orders:
    image: registry.gitlab.com/lukeluha/dp-orders:latest
    environment:
      NODE_ENV: production
    labels:
```

```
30        io.rancher.container.pull_image: always
31    front:
32      image: registry.gitlab.com/lukeluha/dp-front:latest
33      ports:
34      - 8080:80/tcp
35      labels:
36        io.rancher.container.pull_image: always
37    categories:
38      image: registry.gitlab.com/lukeluha/dp-categories:latest
39      environment:
40        NODE_ENV: production
41      labels:
42        io.rancher.container.pull_image: always
43    articles:
44      image: registry.gitlab.com/lukeluha/dp-articles:latest
45      environment:
46        NODE_ENV: production
47      labels:
48        io.rancher.container.pull_image: always
49    db:
50      image: mariadb:10.1.22
51      environment:
52        MYSQL_ROOT_PASSWORD: R2%5(z*uR77Zadpb2e
53    volumes:
54        - '/var/lib/mysql:/var/lib/mysql'
55    gateway:
56      image: registry.gitlab.com/lukeluha/dp-gateway:latest
57      environment:
58        NODE_ENV: production
59      labels:
60        io.rancher.container.pull_image: always
61    products:
62      image: registry.gitlab.com/lukeluha/dp-products:latest
63      environment:
64        NODE_ENV: production
65      labels:
66        io.rancher.container.pull_image: always
67    load-balancer:
68      image: rancher/lb-service-haproxy:v0.6.4
69      ports:
70      - 80:80/tcp
71      labels:
72        io.rancher.container.agent.role: environmentAdmin
73        io.rancher.container.create_agent: 'true'
```

E.2 rancher-compose.yml

```
version: '2'
services:
  cache:
    scale: 1
    start_on_create: true
  elastic:
    scale: 1
    start_on_create: true
  orders:
    scale: 2
    start_on_create: true
    health_check:
      healthy_threshold: 2
      response_timeout: 2000
      port: 3000
      unhealthy_threshold: 3
      initializing_timeout: 60000
      interval: 2000
      strategy: recreate
      request_line: GET "/" "HTTP/1.0"
      reinitializing_timeout: 60000
  front:
    scale: 2
    start_on_create: true
  categories:
    scale: 2
    start_on_create: true
    health_check:
      healthy_threshold: 2
      response_timeout: 2000
      port: 3000
      unhealthy_threshold: 3
      initializing_timeout: 60000
      interval: 2000
      strategy: recreate
      request_line: GET "/" "HTTP/1.0"
      reinitializing_timeout: 60000
  articles:
    scale: 2
    start_on_create: true
    health_check:
      healthy_threshold: 2
      response_timeout: 2000
      port: 3000
      unhealthy_threshold: 3
      initializing_timeout: 60000
      interval: 2000
```

```
48        strategy: recreate
49        request_line: GET "/" "HTTP/1.0"
50        reinitializing_timeout: 60000
51    db:
52      scale: 2
53      start_on_create: true
54    gateway:
55      scale: 2
56      start_on_create: true
57      health_check:
58        healthy_threshold: 2
59        response_timeout: 2000
60        port: 3000
61        unhealthy_threshold: 3
62        initializing_timeout: 60000
63        interval: 2000
64        strategy: recreate
65        request_line: GET "/articles" "HTTP/1.0"
66        reinitializing_timeout: 60000
67    products:
68      scale: 4
69      start_on_create: true
70      health_check:
71        healthy_threshold: 2
72        response_timeout: 2000
73        port: 3000
74        unhealthy_threshold: 3
75        initializing_timeout: 60000
76        interval: 2000
77        strategy: recreate
78        request_line: GET "/" "HTTP/1.0"
79        reinitializing_timeout: 60000
80    load-balancer:
81      scale: 1
82      start_on_create: true
83      lb_config:
84        certs: []
85        port_rules:
86        - path: ''
87          priority: 1
88          protocol: http
89          service: front
90          source_port: 80
91          target_port: 80
92      health_check:
93        response_timeout: 2000
94        healthy_threshold: 2
95        port: 42
96        unhealthy_threshold: 3
```

```
97          initializing_timeout: 60000
98          interval: 2000
99          reinitializing_timeout: 60000
```