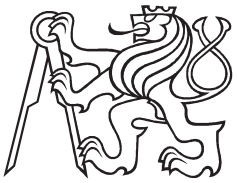
Bakalářská práce



České
vysoké
učení technické
v Praze

F3

Fakulta elektrotechnická
Katedra počítačů

Škálování aplikací založených na microservice architektuře

David Kiml

Vedoucí: Ing. Jiří Šebek
Studijní program: Otevřená informatika
Obor: Software
Květen 2023

ZADÁNÍ BAKALÁŘSKÉ PRÁCE

I. OSOBNÍ A STUDIJNÍ ÚDAJE

Příjmení: **Kiml** Jméno: **David** Osobní číslo: **499006**

Fakulta/ústav: **Fakulta elektrotechnická**

Zadávající katedra/ústav: **Katedra počítačů**

Studijní program: **Otevřená informatika**

Specializace: **Software**

II. ÚDAJE K BAKALÁŘSKÉ PRÁCI

Název bakalářské práce:

Škálování aplikací založených na microservice architektuře

Název bakalářské práce anglicky:

Scaling applications based on microservice architecture

Pokyny pro vypracování:

V současné době jsou mikroslužby skloňované jako nejlepší
přístup pro psaní komplexních aplikací a vůbec realizaci podnikových
architektur. Aplikace musí být pro uživatele dostupná, a proto je potřeba analyzovat její nasazení a celkové zatížení, které je cílem snížit.
Cíle této práce jsou:
1) Provést rešerši možností, jak škálovat aplikace a snížit tak jejich zátěž (hw...)
2) Vytvořit ukázkovou aplikaci založenou na microservice architektuře a demonstrovat možnosti škálování
3) Připravit scénáře pro škálování ukázkové aplikace a následně aplikaci otestovat

Seznam doporučené literatury:

ABBOTT, Martin L.; FISHER, Michael T. The art of scalability: Scalable web architecture, processes, and organizations for the modern enterprise. Addison-Wesley Professional, 2015.
BURNS, Brendan, et al. Kubernetes: up and running. " O'Reilly Media, Inc.", 2022.
ROSSI, Fabiana; CARDELLINI, Valeria; PRESTI, Francesco Lo. Hierarchical scaling of microservices in kubernetes. In: 2020 IEEE International Conference on Autonomic Computing and Self-Organizing Systems (ACSOS). IEEE, 2020. p. 28-37.

Jméno a pracoviště vedoucí(ho) bakalářské práce:

Ing. Jiří Šebek kabinet výuky informatiky FEL

Jméno a pracoviště druhé(ho) vedoucí(ho) nebo konzultanta(ky) bakalářské práce:

Datum zadání bakalářské práce: **10.02.2023** Termín odevzdání bakalářské práce: **26.05.2023**

Platnost zadání bakalářské práce: **22.09.2024**

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

III. PŘEVZETÍ ZADÁNÍ

Student bere na vědomí, že je povinen vypracovat bakalářskou práci samostatně, bez cizí pomoci, s výjimkou poskytnutých konzultací. Seznam použité literatury, jiných pramenů a jmen konzultantů je třeba uvést v bakalářské práci.

. Datum převzetí zadání

Podpis studenta

Poděkování

Chtěl bych poděkovat svému vedoucímu Ing. Jiřímu Šebkovi za jeho rady a poskytnuté materiály, které mi pomohly s prací.

Prohlášení

Prohlašuji, že jsem předloženou práci vypracoval samostatně, a že jsem uvedl veškerou použitou literaturu.

V Praze, 26. května 2023

Abstrakt

Tato bakalářská práce se zabývá možnostmi škálování aplikací. Škálování je potřeba kvůli tomu, aby aplikace zvládaly uspokojit větší nárůst uživatelů, kteří ji v daném momentě používají. Důraz je kladen především na mikroservisní architekturu, která je lépe postavena pro lepší způsoby škálování. Škálování je použito a otestováno ve vytvořené aplikaci.

Klíčová slova: Škálování, Mikroslužby, Docker, Kubernetes

Vedoucí: Ing. Jiří Šebek

Abstract

This bachelor's thesis deals with application scaling possibilities. Scaling is needed for applications to handle a larger number of users using it at a given time. The focus is primarily on microservices architecture, which is better built for better scaling methods. Scaling is used and tested on the created application.

Keywords: Scaling, Microservices, Docker, Kubernetes

Title translation: Scaling applications based on microservice architecture

Obsah

Zadání práce iii

1 Úvod 1

2 Rešerše 3
2.1 Škálování 3
2.1.1 Způsoby škálování 3
2.1.2 Škálovací kostka 4
2.1.3 Load Balancing 5
2.2 Mikroservisní architektura 7
2.2.1 Komunikace mezi mikroslužbami 8
2.2.2 Výhody mikroslužeb oproti monolitu 9
2.2.3 Nevýhody mikroslužeb 9
2.3 Kontejnerizace 9
2.3.1 Výhody kontejnerů oproti plné virtualizaci 10
2.3.2 Docker 11
2.3.3 Podman 12
2.4 Orchestrace kontejnerů 12
2.4.1 Docker compose 12
2.4.2 Kubernetes 13
2.4.3 OpenShift 15

3 Analýza 17
3.1 Požadavky 17

4 Návrh 19
4.1 Architektura aplikace 19
4.2 Komponenty 20
4.3 Třídy 21

5 Implementace 23
5.1 Zvolené technologie 23
5.2 Monolitická aplikace 25
5.3 Mikroservisní aplikace 25

6 Nasazení 27
6.1 Docker 27
6.2 Kubernetes 28
6.2.1 Aplikace 28
6.2.2 Databáze 29

7 Výkonnostní testy 31
7.1 Použité nastavení 31
7.2 Locust 31
7.3 Scénáře 32
7.4 Shrnutí 35

8 Závěr 37
8.1 Shrnutí 37
8.2 Možnosti budoucí práce 38

A Literatura 39

B Seznam použitých zkratek 41

C Elektronické přílohy 43

Obrázky

2.1 Škálovací kostka (zdroj: [19]) 4
2.2 Ukázka komunikace s použitím load balanceru 5
2.3 Porovnání monolitické a mikroservisní aplikace (zdroj: [6]) . . 8
2.4 Porovnání virtuálních strojů (nalevo) a kontejnerů (zdroj: [15]) . 10
2.5 Docker architektura (zdroj: [8]) . 11
2.6 Kubernetes architektura (zdroj: [12]) 14

3.1 Diagram případu užití 18

4.1 Diagram komponent 20
4.2 Diagram tříd mikroservisní aplikace 21

5.1 API dokumentace pro přidání nového jídla do menu 24
5.2 Ukázka více vrstevnaté aplikace (zdroj: [21]) 25
5.3 Vnitřní struktura aplikace 26

7.1 HTA diagram 33
7.2 Graf zachycující HW nároky podle počtu uživatelů 35

Tabulky

7.1 Výsledky testů při 100 uživatelích a rovnoměrném rozložení zátěže . . 32
7.2 Výsledky testů při 300 uživatelích a rovnoměrném rozložení zátěže . . 32
7.3 Přiřazené množství podů pro mikroslužby 34
7.4 Výsledky testů při 100 uživatelích a standardním rozložení zátěže . . . 34
7.5 Výsledky testů při 300 uživatelích a standardním rozložení zátěže . . . 34

Kapitola 1
Úvod

V poslední době stoupá trend přesouvání aplikací na internet a stále častěji jsou využívány webové aplikace. Také roste i zájem od uživatelů tyto služby používat. Aplikace proto musí tento velký narůst uživatelů zvládnout uspokojit v dostatečném čase a bez výpadků. Proto je potřeba aplikace některými způsoby škálovat. Možností je buď přidávat aplikaci dostupné zdroje, nebo vytvořit více instancí aplikace. Hardwarové prostředky na serveru, kde aplikace běží, nejsou ale neomezené a je proto potřeba využít je co nejlépe.

Mikroservisní aplikace, kdy je celkový systém složen z menších služeb, tento problém může řešit, protože se stačí soustředit pouze na zatíženou část a není potřeba přidávat zdroje celé aplikaci.

Cílem této práce je tedy prozkoumat možnosti škálování a použít tyto způsoby při vývoji aplikace založené na mikroservisní architektuře. Vytvořenou aplikaci je také vhodné otestovat výkonnostními testy, zda použité způsoby škálování pomohly nápor uživatelů zvládnout.

Kapitola 2
Rešerše

Tato kapitola se věnuje teoretické části práce, kde jsou popsány možnosti škálování, mikroservisní architektura, kontejnerizace aplikací a jejich orchestrace.

2.1 Škálování

Škálování je vlastnost aplikace, která popisuje schopnost růst a zvládat vysoké nároky na práci a její zdroje[4]. Díky škálování je možné zvýšit dostupnost aplikace a zvládat tak velký nárůst uživatelů, kteří ji v daném čase aktuálně používají.

2.1.1 Způsoby škálování

Škálování aplikace je možné docílit primárně dvěma hlavními způsoby. Jedná se o horizontální a vertikální škálování. Obě možnosti se liší svým přístupem k tomu, jak se dá aplikace škálovat[19].

Horizontální škálování

Technika horizontálního škálování vzniká vytvořením více instancí dané aplikace. Každá instance má přiřazeny své hardwarové prostředky a běží nezávisle na sobě. Díky tomu může být dokonce každá instance nasazena i na jiném serveru.

Při posílání požadavků aplikaci z jednoho přístupového bodu je ale potřeba vybrat určitou instanci, která požadavek obslouží. Ta se vybere pomocí specifikovaného algoritmu, které se požadavek následně odešle. Tuto práci má na starosti load balancer (viz kapitola 2.1.3). Při použití časově náročného algoritmu se může doba odezvy zvýšit.

Vertikální škálování

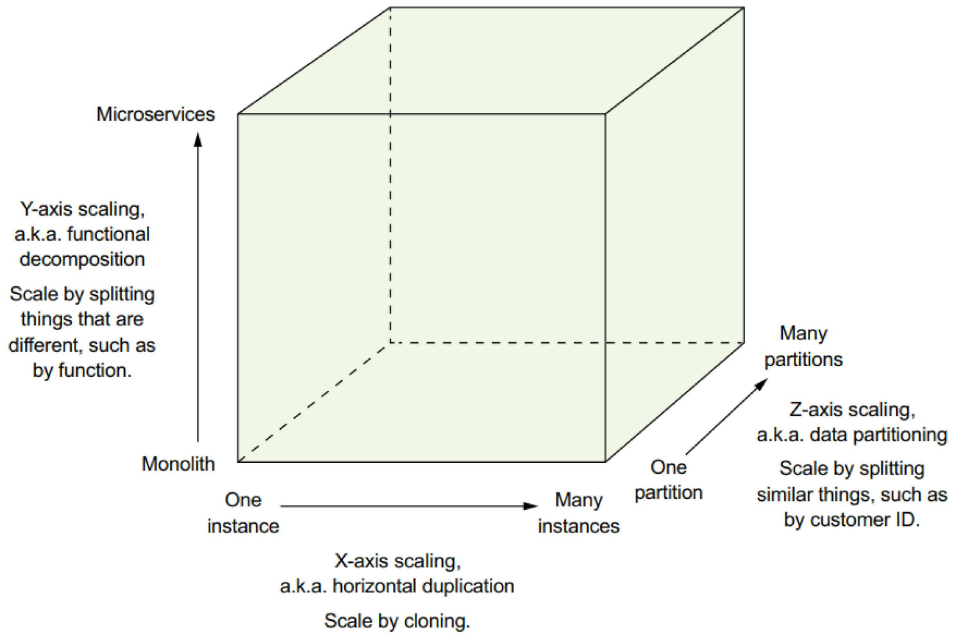
Vertikální škálování je forma škálování, kdy se aplikaci přidá více hardwarových prostředků. Jedná se primárně o paměť RAM a přiřazení dalšího výpočetního výkonu procesoru. Díky tomu může aplikace zvládat větší zatížení při zpracovávání požadavků. Vertikální škálování je však ale omezeno

fyzickými prostředky stroje, na kterém aplikace běží, a proto ji není možné škálovat neomezeně. Po dosažení limitu zdrojů je tedy potřeba přidat větší výkon stroje nebo změnit strategii škálování.

V případě pádu aplikace dojde oproti horizontálnímu škálování ke kompletní nedostupnosti, protože není jiná instance, která by požadavek zpracovala.

2.1.2 Škálovací kostka

Škálovací kostka o třech osách - x, y, z popisuje, jak je možné dosáhnout škálovatelnosti aplikace[1]. V souřadnicích (0,0,0) se jedná o monolitickou aplikaci, které lze navýšit maximálně hardwarové prostředky. Postupné změny souřadnic v osách x, y a z umožňují větší možnosti škálování. Některé aspekty je pak potřeba brát v potaz již při návrhu samotné aplikace.



Obrázek 2.1: Škálovací kostka (zdroj: [19])

X-Axis scaling

Škálování na ose x je založeno na vytvoření více instancí aplikace. Tento přístup je jednoduchý, protože není potřeba aplikaci modifikovat a stačí ji použít tak, jak je již naimplementovaná. Aby byl využit tento potenciál, je potřeba požadavky rozesílat mezi jednotlivé instance.

Při samotném použití škálování podle osy x se spouští nová instance celé aplikace i přes to, že nějaká část aplikace je jen zřídka využívaná. To může zbytečně plýtvat přiřazenými prostředky.

Y-Axis scaling

Škálování podle vertikální osy y je způsob rozdělení celé aplikace na jednotlivé služby. Tyto služby jsou rozděleny podle funkcionality, které budou umožňovat a dat, se kterými pracují. Tohoto způsobu je možné docílit tak, že se dekomponuje monolitická aplikace na části, ze kterých se například vytvoří jednotlivé mikroslužby. Při tomto přístupu je ale potřeba vyřešit komunikaci mezi službami, což u monolitické aplikace není potřeba. Službám je pak možné přiřadit podle potřeby různý výkon.

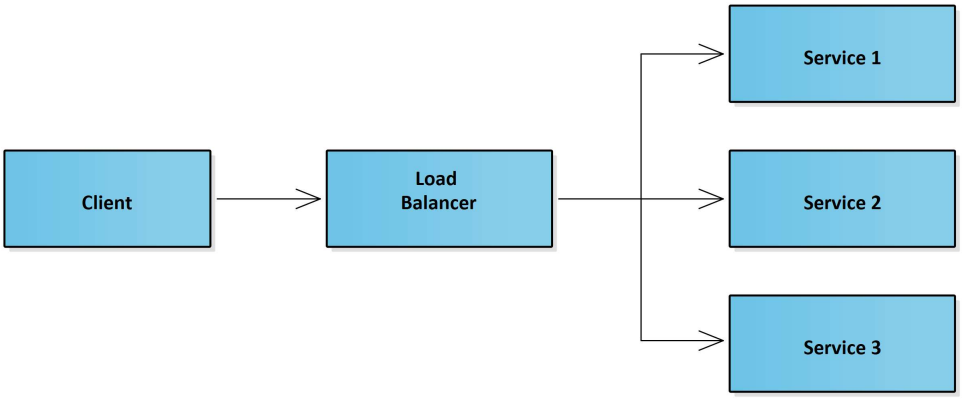
Z-Axis scaling

Při škálování na ose z je každá služba zodpovědná za nějakou podmnožinu dat. Tato data by měla být rozdělena podle unikátního identifikátoru pro správný výběr služby. Každá služba je však naimplementovaná stejně.

Tento přístup se používá hlavně v databázových systémech, kdy lze rozdělit instance podle primárních klíčů. Každá instance pak obsluhuje pouze část záznamů. Pro rozdělování požadavků mezi instance je potřeba služba, která je přeposílá podle vybrané vlastnosti.

2.1.3 Load Balancing

Load balancing je velmi důležitým přístupem k umožnění horizontálního škálování aplikací. Load balancer, také známý jako dispatcher, slouží k rozložení zátěže tím, že zasílá požadavky mezi instance služby. Ten se dá rozdělit na dva druhy[3]. Jedná se o load balancer na straně klienta a na straně serveru.



Obrázek 2.2: Ukázka komunikace s použitím load balanceru

Load balancer na straně serveru

Když přijde požadavek ze strany klienta, pošle se nejdříve load balanceru běžící na straně serveru. Load balancer jej přepošle podle strategie (viz kapitola 2.1.3) vybrané instanci. V případě, že bude nějaká instance potřebovat odeslat požadavek jiné, tak jej opět nejdříve pošle na centralizovaný load balancer. Ten se opět postará o přeposlání požadavku jiné službě.

Díky load balanceru na straně serveru stačí, aby klient znal pouze jeden přístupový bod k aplikaci. Nevýhodou tohoto přístupu je to, že se centralizovaný load balancer stane jediným bodem selhání[2] (tzn. část systému, která když se stane nefunkční, tak dojde k selhání a přestane fungovat celá aplikace), nebo může zpomalovat provoz celé aplikace při velkém počtu právě zpracovávaných požadavků. Tím může centralizovaný load balancer zvýšit čas odezvy aplikace. Lepším řešením může být implementace load balancerů pro každou množinu instancí zvlášť. Díky tomu se rozdělí zátěž mezi více load balancerů, jež mají na starosti jednotlivé služby. Stále se však ale jedná o jediný vstupní bod a při nefunkčnosti load balanceru dojde k selhání komunikace se službou.

Load balancer na straně klienta

Load balancer na straně klienta je koncipován tak, že každá instance klienta má svůj vlastní load balancer u sebe. Ten je přímo zodpovědný pro posílání požadavků právě od jednoho klienta a rozložení zátěže mezi cílové instance.

Oproti centrálnímu load balanceru se odstraní jediný bod selhání v systému. Klient ale potřebuje znát vstupní body všech instancí služby, kam může požadavek poslat.

Load balancing algoritmy

Aby bylo možné rozdělovat požadavky mezi instance služby, je potřeba použít nějaký algoritmus, podle kterého se budou cílové instance vybírat. K tomu slouží několik algoritmů, jež na základě metriky rozdělují jednotlivé požadavky mezi instance služby. Je potřeba brát ale v úvahu, že čím složitější algoritmus se použije, tím se může zbytečně zvýšit čas odezvy.

Následně jsou zde popsané vybrané load balancer algoritmy:

- **Round Robin** - Algoritmus round robin je ve své podstatě velmi jednoduchý. Požadavky jsou rozesílány mezi instance sekvenčně. To znamená, že jsou instance seřazeny do kruhové fronty a load balancer požadavky podle tohoto pořadí rozesílá. Když se load balancer dostane na konec fronty, začne stejným způsobem od znovu.

  Výhoda tohoto algoritmu je nenáročná implementace a jednoduchost. Instance jsou vybírány deterministicky, takže se dá jednoznačně určit, která instance se požadavku přiřadí. Nevýhodou je, že algoritmus není adaptivní a například může posílat požadavky přetížené instanci, která se dostane na řadu. Tento algoritmus je nejpoužívanější pro svou jednoduchost a rychlost[7].

  Existuje i modifikovaný algoritmus jménem vážený round robin, který instancím přidává i váhu. Podle této váhy jsou některé instance více preferovány.

- **Random** - Tento algoritmus vybírá instanci na základě náhodně zvoleného čísla v rozsahu celkového počtu instancí. Podle tohoto čísla se odešle

požadavek dané instanci. Při velkém počtu požadavků se statisticky zátěž mezi instance rovnoměrně rozloží. Díky tomu se může vyrovnat algoritmu jako je round robin[20]. Algoritmus random není výpočetně náročný a nedochází tak k velkému zpoždění.

- **IP hash** - Při tomto přístupu se požadavek odešle instanci podle IP adresy klienta. Tato adresa se pomocí hashovací funkce převede na číselnou hodnotu. Podle čísla vytvořeného hashovací funkcí je následně požadavek odeslán na instanci odpovídající vypočtenému číslu podle algoritmu. Požadavky od klienta se tak budou posílat na stejnou službu, dokud bude dostupná.

- **Least connection** - Při použití least connection se budou požadavky rozesílat instancím, které mají aktuálně nejméně přijatých požadavků a nejméně aktivních připojení. Tento způsob však ale potřebuje znát aktuální informace o instancích, což může být náročnější na implementaci oproti algoritmům jako je například round robin nebo random.

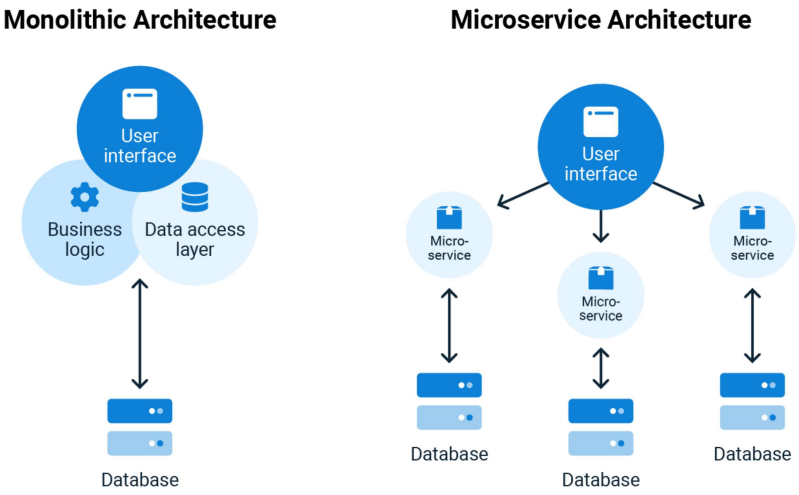
  Podobně jako vážený round robin existuje i upravená verze s názvem vážený least connection, který k instancím také přidává váhu jako metriku navíc.

- **Least response time** - Least response time, podobně jako least connection, vybírá instance podle aktivních spojení. Zároveň jako hlavní metriku používá čas odezvy, což je doba od odeslání požadavku a přijetí odpovědi. Po překročení nastavené hodnoty začne load balancer upřednostňovat jiné instance.

- **Sticky session** - Když potřebuje aplikace uchovávat spojení mezi klientem a serverem, je vhodné použít algoritmus sticky session. Při poslání prvního požadavku od klienta je vybrána některá instance a při dalších požadavcích je směrována na stejnou instanci, kde si drží relaci. Díky tomu si může aplikace uchovávat nějaký stav nebo cachovat specifičtější data.

2.2 Mikroservisní architektura

Mikroservisní architektura je softwarová architektura rozdělující monolitickou aplikaci, která zahrnuje všechny funkcionality a kód v jednom celku, na několik menších částí[5]. Jednotlivé mikroslužby jsou dekompozicí celkové aplikace, které jsou uzavřené na svoji funkcionalitu. Tyto služby fungují samostatně a díky tomu se služby dají nezávisle na sobě vyvíjet a nasazovat. To podporuje i lepší škálovatelnost jednotlivých mikroslužeb, kdy je možné škálovat jen určitou část místo celé aplikace.



Obrázek 2.3: Porovnání monolitické a mikroservisní aplikace (zdroj: [6])

Každá mikroslužba by měla mít svoji vlastní databázi a jiné by k ní neměly napřímo přistupovat. Místo toho by měly využít právě službu, která již data z databáze zpracuje a aplikuje nad nimi byznys logiku. Služby jsou mezi sebou tak méně vázány, a důsledkem toho je možné upravovat jednotlivá schémata služby bez potřeby upravení ostatních částí.

2.2.1 Komunikace mezi mikroslužbami

Ke komunikaci má každá služba vystavené přístupové body. Komunikace může probíhat buď synchronně nebo asynchronně pomocí RabbitMQ[1] nebo Apache Kafka[2]. U asynchronního posílání požadavků se nečeká na odpověď, ale požadavek je uložen do fronty, který pak volaná služba zpracuje až bude mít volné kapacity.

Komunikace probíhá nejčastěji pomocí HTTP API nebo gRPC[23]. Při komunikaci pomocí HTTP protokolu se používají pro posílání informace převážně dvě hlavní rozhraní: REST a SOAP.

- **REST** - U RESTu jsou data přenášena hlavně ve formátu JSON. Jako další možnost může být i formát XML. Služby jsou přístupné pomocí URI adres, kde každá adresa reprezentuje zdroj dat. Pro základní manipulaci s daty se používají HTTP metody GET, POST, PUT, DELETE.

- **SOAP** - SOAP je další možností, jak komunikovat mezi službami, která využívá XML zprávy a není přímo vázaná na HTTP protokol oproti RESTu. Zpráva má definovanou strukturu a je složena z kořenového elementu s názvem envelope, který pak obsahuje hlavičku a tělo. Hlavička

[1] https://www.rabbitmq.com/
[2] https://kafka.apache.org/

obsahuje pomocné informace pro přenos a tělo pak obsahuje informace ohledně požadavku, které mají být zpracovány.

- **gRPC** - gRPC je open source projekt vyvíjený společností Google pro vzdálené volání procedur. Umožňuje komunikovat mezi mikroslužbami pomocí binárního protokolu. Díky tomu je vhodným kandidátem pro rychlý a efektivní přenos dat[11].

2.2.2 Výhody mikroslužeb oproti monolitu

Rozdělení aplikace na více služeb přináší některé výhody. Tím může být jednodušší údržba a vývoj každé části nezávisle na sobě. Každá služba tak může být naimplementována pomocí jiného programovacího jazyka, pokud vystavuje definované rozhraní.

Mikroslužby jsou menší, mají méně řádek kódu a díky tomu bývají srozumitelnější a je možné vyvíjet službu bez větších znalostí celé aplikace. Také je díky tomu jednodušší hledat vyskytnuté chyby v kódu.

Jednotlivé mikroslužby jsou lépe přizpůsobeny pro horizontální škálování. Díky nezávislosti na ostatních je možné vytvářet více instancí jediné mikroslužby a není potřeba zbytečně škálovat celou aplikaci, což požaduje větší výkon stroje. Také je díky nezávislosti umožněno i lepší nasazení v produkčním prostředí, protože nedojde k nefunkčnosti celkové aplikace a všech funkcionalit při nasazení jiné verze jedné mikroslužby[9].

2.2.3 Nevýhody mikroslužeb

Mikroslužby jakožto distribuované systémy přináší větší komplexitu systému a je potřeba řešit meziprocesní komunikaci a režii mezi službami, což monolitická aplikace přímo nevyžaduje. Jelikož spolu mikroslužby komunikují po síti, tak může docházet k větším latencím a vytížení sítě.

Protože by měla každá mikroslužba mít vlastní databázi, je potřeba udržovat datovou konzistenci mezi službami. Bez celkového sdílení dat ale může docházet k nekonzistenci dat, která se musí nějakým způsobem vyřešit například pomocí návrhového vzoru Saga. Saga pattern je návrhový vzor, který řídí transakční procesy v distribuovaných aplikacích[22]. Pokud v nějaké službě dojde k selhání nebo nezdaru, vytvoří se protiakce, která navrátí provedené změny a transakci buď opakuje nebo ji ukončí.

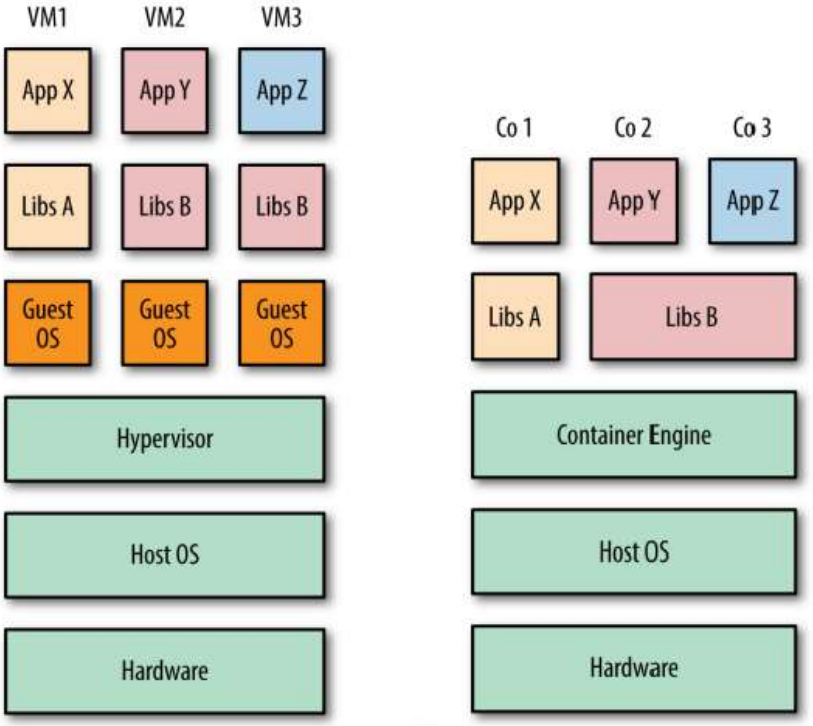
Pro mikroservisní architekturu existuje řada dalších návrhových vzorů, které popisují, jak řešit různé problémy tak, aby se dala řešení opakovaně použít.

2.3 Kontejnerizace

Kontejnerizace umožňuje zabalit aplikaci a její závislosti pro běh do takzvaného kontejneru. Díky tomu není pak potřeba řešit konfiguraci a instalaci závislostí při přenosu na jiný počítač. Tyto kontejnery jsou pak spouštěny izolovaně od operačního systému a od ostatních kontejnerů.

2.3.1 Výhody kontejnerů oproti plné virtualizaci

Každý virtuální stroj potřebuje svůj vlastní operační systém, ve kterém jsou pak spuštěny aplikace[24]. Kontejnery místo toho sdílí jádro operačního systému s hostitelským strojem a dalšími kontejnery. Na jednom místě jsou obsaženy i knihovny, které si mohou sdílet mezi sebou. Díky tomu jsou kontejnery odlehčené a je možné, aby jich na hostitelském počítači běželo mnohem více než v případě virtuálních strojů.



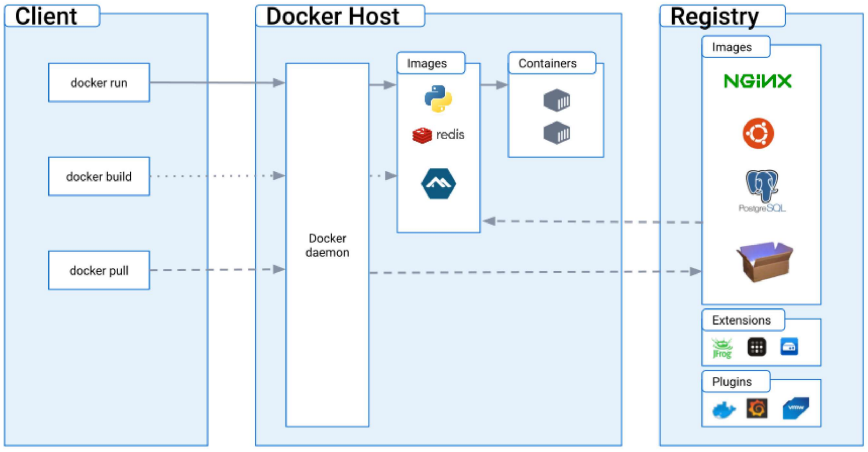
Obrázek 2.4: Porovnání virtuálních strojů (nalevo) a kontejnerů (zdroj: [15])

Kvůli menší režii je pak možné spouštět kontejnery mnohem rychleji a také je rychleji ukončovat. To umožňuje jednodušší distribuci a nasazení. Také je díky tomu poskytována podpora pro možnosti škálování aplikací, protože lze rychle spouštět více instancí aplikace nebo mikroslužby.

Mezi nevýhody se může řadit menší bezpečnost kvůli sdílení operačního systému. To může mít při napadení dopad jak na ostatní kontejnery, tak i na samotný operační systém hostitelského stroje. Tomu se dá zabránit například využitím obou přístupů a spouštět kontejnery až na vytvořeném virtuálním stroji[24].

2.3.2 Docker

Docker je open-source platforma, která slouží pro kontejnerizaci aplikací a jejich správu[8]. Je vytvořen jako aplikace fungující na principu klient-server, kde klient komunikuje se serverem - démonem[14]. Jeden klient může komunikovat i s více démony. Komunikace probíhá přes REST API démona, na které jsou směrovány HTTP požadavky od klienta. Démon může běžet buď na stejném stroji nebo na jiném, se kterým může vzdáleně přes síť komunikovat. Docker démon přijímá požadavky od klienta pro běh a správu kontejnerů a následně je zpracuje. Klient je hlavní cestou, jak uživatel interaguje s Dockerem.



Obrázek 2.5: Docker architektura (zdroj: [8])

Docker komponenty

- **Docker Image** - Docker image je šablona s instrukcemi pro následné vytváření kontejnerů. K vytvoření vlastního image je potřeba vytvořit soubor s názvem Dockerfile, kde jsou definovány postupné kroky, které jsou potřeba k jeho vytvoření a sestavení. Běžně se Docker image skládá z jiného, který již obsahuje potřebné závislosti. Do šablony je pak možné přidat další konfiguraci s dodatečným nastavením. Každá instrukce v Dockerfile představuje vrstvu. Pokud je již vrstva vytvořena a nebyly v ní provedeny změny, použije se v dalším sestavením znovu. To urychluje proces tvorby image.

- **Docker Registry** - Docker image jsou uloženy v Docker registry, což slouží jako repozitář, kde se dají image sdílet a následně odtud stahovat. Jedním z registrů je Docker Hub[3], který je veřejný a Docker je výchozím místem, kde Docker démon hledá image. Je však ale možné mít svůj

[3] https://hub.docker.com/

vlastní privátní registr nebo se dá použít již existující od třetích stran jako je například Container Registry od GitLabu[10].

- **Kontejnery** - Kontejnery jsou spustitelné instance Docker image, který předepisuje jejich podobu. Kontejnery jsou izolovány od ostatních běžících kontejnerů a od stroje, na kterém samy běží. Izolaci je ale možné částečně změnit vlastní konfigurací. Kontejner lze připojit k více sítím nebo k němu připojit datové úložiště. Když je kontejner ukončen, veškeré změny, které nebyly perzistentně uloženy, zmizí.

2.3.3 Podman

Alternativou k Dockeru pro správu kontejnerů může být open source nástroj jménem Podman[4], který byl vyvinut společností Red Hat. Oproti Dockeru, který potřebuje k práci běžící Docker démon a spouští kontejnery s přístupovými právy roota, se kontejnery mohou spouštět jako uživatelské procesy právě bez těchto práv (tzv. rootless mode). To zvyšuje jejich bezpečnost[17]. Pro sestavování image je potřeba navíc použít nástroj Buildah[5].

Podman dovoluje používat stejné příkazy jako v případě Dockeru, takže není problém na tento nástroj jednoduše zmigrovat. Dovoluje také spouštět kontejnery z image vytvořených pomocí jiných nástrojů jako je například zmíněný Docker.

2.4 Orchestrace kontejnerů

Předchozí sekce byla věnována kontejnerizaci aplikací. V případě, kdy je potřeba vytvářet více instancí aplikace a tím pádem i vytvoření více kontejnerů se zvedá obtížnost samostatné správy a údržby kontejnerů. Orchestrátory kontejnerů tento problém řeší a dovolují kontejnery monitorovat nebo přidávat a odebírat další instance aplikací. Při používání kontejnerizace se k aplikacím přistupuje pomocí IP adresy a portu. Místo IP adresy dovolují orchestrátory používat pro přístup nastavený název služby.

2.4.1 Docker compose

Docker compose slouží ke spouštění více kontejnerů jako jeden celek pomocí jednoho příkazu. Dále také povoluje pomocí příkazů další práci se službami[8]:

- Spouštění, zastavování nebo přestavění služeb
- Zobrazit stav běžících služeb
- Zobrazit stream logů běžících služeb
- Spustit příkaz v jedné ze služeb

[4] https://podman.io/
[5] https://buildah.io/

Aby bylo možné spustit služby je potřeba nejprve vytvořit soubor s formátem YAML, který obsahuje definice jednotlivých služeb a jejich konfiguraci.

```yaml
version: '3'

services:
  application:
    build: .
    ports:
      - 8080:8080
    depends_on:
      - db
    environment:
      DB_HOST: db
      DB_NAME: databasename
      DB_USERNAME: username
      DB_PASSWORD: password
  db:
    image: postgres:latest
    ports:
    - 5432:5432
    environment:
      POSTGRES_DB: databasename
      POSTGRES_USER: username
      POSTGRES_PASSWORD: password
```

Listing 2.1: Ukázka Docker compose

Ukázkový soubor (viz listing 2.1) je konfigurací pro Docker compose, kde jsou definovány dvě služby. První službou je aplikace, jejíž kontejner bude sestaven v aktuálním adresáři, kde se vyskytuje Dockerfile. Aplikace uvnitř kontejneru bude naslouchat na portu 8080, který bude na hostitelském stroji také namapován na port 8080. Následně je uvedeno, že aplikace má závislost na službě *db*. To znamená, že bude zajištěno spuštění nejdříve služby s databází. Nakonec jsou definovány proměnné pro přístup a připojení k databázi. Druhá služba je podobně definovaná jako první. Místo sestavení je zde ale obsažen image, který bude stažen z Docker registry.

2.4.2 Kubernetes

Kubernetes, také známá jako K8s, je open-source platforma, která slouží pro automatizaci nasazení, škálování a správu kontejnerizovaných aplikací[12].

Existuje i odlehčená Kubernetes distribuce s názvem K3s[6], vytvořená společností Rancher, která je distribuována v jediném binárním souboru s velikostí pod 100 megabajtů. Je navržena pro běh na zařízeních s menšími zdroji jako jsou například IoT zařízení.

[6] https://k3s.io/

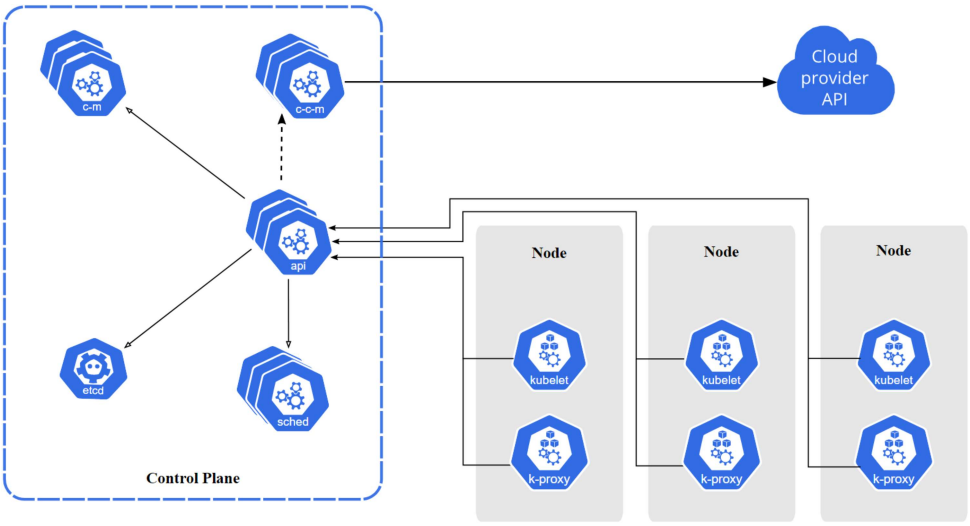
Kubernetes umožňuje spouštět jednotlivé kontejnery a zároveň také spustit větší počet instancí aplikace. Mezi tyto instance posílá požadavky pomocí load balancingu. Stará se o monitoring aplikací a při případném pádu aplikace jej dokáže automaticky znovu spustit. Aplikacím je možné nastavit, kolik prostředků budou moci využívat a při dosažení limitu využitých zdrojů dokáže vytvářet nové instance.

Konfiguraci Kubernetes komponent je možné spravovat v souborech ve formátu YAML. Tyto konfigurace lze následně v Kubernetes jednoduše nasadit pomocí příkazů. Podpora Kubernetes služeb je veliká a existuje mnoho dostupných nástrojů od třetích stran, které lze v Kubernetes integrovat.

Kubernetes architektura

Systém, který vznikne při nasazení Kubernetes se nazývá cluster[12]. Cluster obsahuje komponentu s názvem node. Každý cluster má minimálně jeden node, ale může jich mít i více. Node je buď fyzický nebo virtuální stroj.

Pro správu těchto komponent využívá Kubernetes cluster control plane, který komunikuje s node přes kubelet. Kubelet je agent, který již běží na jednotlivém node. Ten zajišťuje, že na node běží nakonfigurovaný pod, který následně monitoruje. Na node běží také kube-proxy, která udržuje pravidla pro komunikaci s pody a vnějším prostředím v síti.



Obrázek 2.6: Kubernetes architektura (zdroj: [12])

Hlavní Kubernetes objekty

Kubernetes objekty jsou entity, které slouží k popisu stavu kontejnerizovaných aplikací a běžících služeb[12]. Objekty jsou mezi sebou propojeny a vytváří závislosti ovlivňující celkové chování v Kubernetes.

- **Pod** - Jedná se o nejmenší nasaditelnou jednotku v Kubernetes, která slouží jako abstrakce nad kontejnery. Běžně je v podu právě jeden kontejner, nicméně je možné jich obsahovat více. Každý pod má přiřazenou

jednu IP adresu. Podu je možné nastavit jeho limity a minimální požadované zdroje jako je CPU a paměť RAM.

- **ReplicaSet** - ReplicaSet má za úkol udržovat nastavený počet instancí podů v daný okamžik. Když je v jednom momentu nedostatek spuštěných instancí, replicaSet vytvoří další pod pomocí dané šablony. V opačném případě může i běžící pod smazat, aby docílila požadovaného stavu.
- **Service** - Service je jednotka, která definuje množinu podů a možnosti, jak se k nim připojit. Množina podů je sloučena pomocí tzv. selectorů, které dovolují filtrovat zdroje podle labelů. Service automaticky skenuje síť a hledá Pody, jejichž labely se shodují se selectorem. Mezi tyto pody může posílat požadavky díky load balancingu. Výchozí strategií při load balancingu je round robin[16].
- **Deployment** - Deployment je hlavním způsobem toho, jak definovat vytváření instancí podů a nastavení replicaSetu. Konfigurace obsahuje požadovaný stav a podle ní se přizpůsobí například počet podů či změní Docker image. Právě proto je díky deployment objektu jednoduché vydat novou verzi nebo se vrátit k původní.
- **Secret** - Pro udržení bezpečnosti aplikace je vhodné mít uložené citlivé údaje jako jsou například hesla mimo samotný kód aplikace. K tomu slouží objekt s názvem secret. Objekty k údajům přistupují přes secret díky klíči, pro který bude vrácena nastavená hodnota v souboru pro secret. Funguje tedy jako úložiště klíč-hodnota.
- **Ingress** - Ingress je API objekt, který umožňuje vnější přístup z clusteru ke Kubernetes services. Běžně vystavuje své rozhraní vnějšímu prostředí pomocí protokolů HTTP a HTTPS. K funkčnosti ingressu je potřeba mít spuštěný ingress controller, který konfiguraci ingressu zpracovává a řídí.

2.4.3 OpenShift

OpenShift je open source platforma sloužící pro kontejnerovou orchestraci[18]. Orchestrátor vytvořený firmou RedHat je vrstvou nad Kubernetes a Dockerem, která přidává funkce a služby pro usnadnění nasazení a správu kontejnerů. Obsahuje tedy většinu Kubernetes objektů a přidává další. Některé objekty však upravuje a například místo ingresu nabízí objekt s názvem router.

Nabízí také i svůj kontejnerový registr pro ukládání image. RedHat poskytuje i placenou podporu.

Openshift má integrované služby pro CI/CD a monitorování aplikací. Také poskytuje webové rozhraní pro jednoduchou správu objektů. Openshift přidává větší zabezpečení co se týče autorizace a autentizace uživatelů. Oproti Kubernetes, který lze provozovat na libovolné linuxové distribuci, Openshift má podporu jen v limitovaných OS právě od firmy RedHat.

Kapitola 3
Analýza

Tato kapitola přechází z teoretické části práce na praktickou, a je zde představena zvolená ukázková aplikace spolu s vypsanými funkčními a nefunkčními požadavky.

3.1 Požadavky

Jako ukázková aplikace pro demonstraci škálování byl vybrán systém pro objednávání jídla. Základním cílem je umožnit zákazníkům objednávání jídla přes aplikaci a zaměstnancům ulehčit správu menu, objednávek a jejich doručení.

Pro aplikaci byly vytvořeny následující funkční a nefunkční požadavky tak, aby aplikace obsahovala základní aktivity z klasických webových objednávacích systémů a zároveň se věnovala problematice škálování:

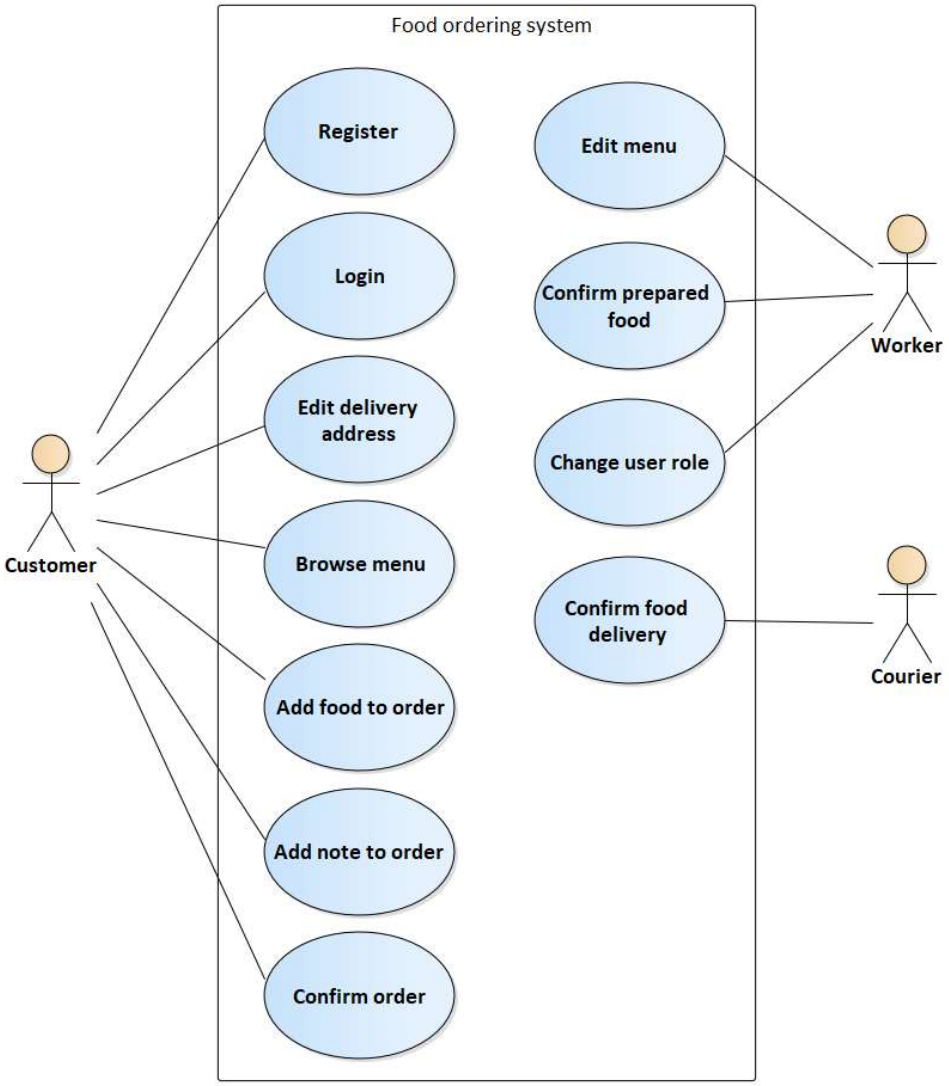
Funkční požadavky

1. Systém bude umožňovat uživateli registraci účtu.
2. Systém bude umožňovat uživateli přihlášení do účtu.
3. Systém bude umožňovat uživateli změnit adresu doručení jídla.
4. Systém bude umožňovat uživateli změnit roli uživatele.
5. Systém bude umožňovat uživateli prohlížet dostupná jídla.
6. Systém bude umožňovat uživateli přidat nové jídlo do menu.
7. Systém bude umožňovat uživateli upravit popis a cenu jídla.
8. Systém bude umožňovat uživateli přidat jídlo do objednávky.
9. Systém bude umožňovat uživateli přidat poznámku u objednávky.
10. Systém bude umožňovat uživateli měnit stav objednávky.
11. Systém bude umožňovat notifikaci o změnu objednávky.

Nefunkční požadavky

1. Systém by měl být schopný zpracovat velký počet požadavků současně.
2. Systém bude navržen tak, aby bylo umožněno snadné škálování aplikace.
3. Systém by měl být spolehlivý a bezporuchový. V případě výpadku by měl být schopen na tento problém reagovat.

Na obrázku 3.1 si je možné prohlédnout diagram případů užití zachycující funkční požadavky aplikace. Byli vytvořeni tři aktéři, kteří jsou rozděleným uživatele podle rolí. V tomto případě se jedná o zákazníka, pracovníka restaurace a kurýra.



Obrázek 3.1: Diagram případu užití

Kapitola 4

Návrh

V následujících kapitolách textu bude zaměřen důraz pouze na serverovou část aplikace kvůli demonstraci škálování. Klientská část tak nebude potřeba a bude postačovat jenom provolávání přístupových bodů aplikace.

4.1 Architektura aplikace

Objednávkový systém byl navrhnut pomocí dvou stylů. Prvním z nich je monolitická aplikace, která bude obsahovat všechny funkcionality systému v jednom celku. Jako další byla tato monolitická aplikace rozdělena na jednotlivé mikroslužby a tím se přetransformovala na mikroservisní architekturu. Tyto služby již obsluhují právě jednu část systému.

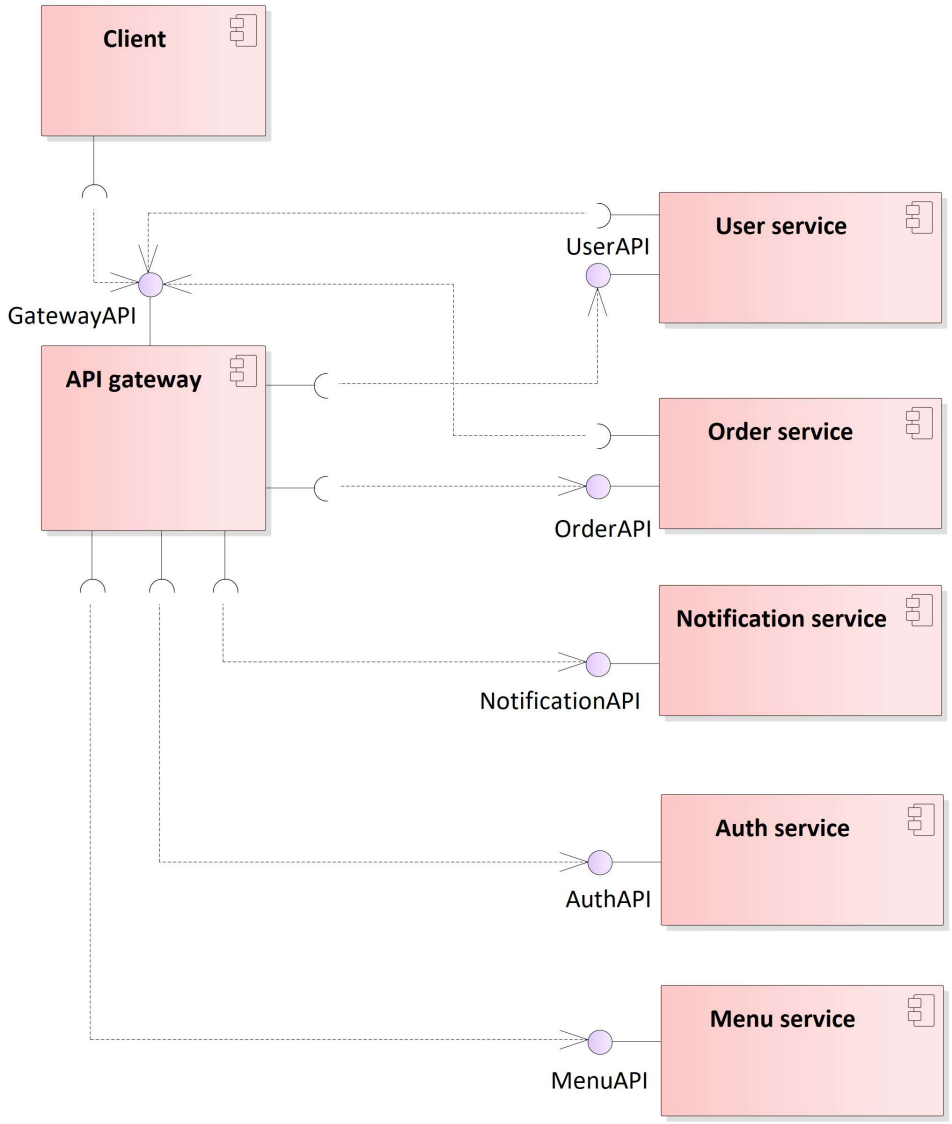
Při návrhu monolitické a mikroservisní aplikace jsou některé rozdíly. Během návrhu mikroslužeb je potřeba navrhnout, jak budou fungovat jako celek a zároveň i navrhnout vnitřní strukturu služeb. Návrh jednotlivých služeb už pak může být prováděn nezávisle na sobě. Při celkovém návrhu je potřeba nejprve rozdělit aplikaci na menší mikroslužby a navrhnout, jak mohou mezi sebou komunikovat.

Aplikace by měla primárně umožňovat správu a vytváření uživatelů, objednávek a položek v menu. Celkový systém byl tedy rozdělen podle domén na pět následujících mikroslužeb:

- **User service** - uživatelská služba spravuje údaje o uživateli jako je jméno, email nebo adresa, která slouží pro informaci o tom, kam objednané jídlo doručit.
- **Auth service** - autentizační služba slouží pro řízení dat ohledně přihlašovacích údajů a práv uživatele.
- **Menu service** - služba pro menu již ze samotného názvu spravuje informace o jídlu, které jsou k dispozici včetně ceny a popisu.
- **Order service** - služba pro objednávky obsahuje údaje o objednávkách uživatele. Ty se skládají z objednávky samotné a přidaných položek jídla.

- **Notification service** - notifikační služba slouží pro vytvoření a uložení upozornění, které se v tomto případě budou týkat ohledně změny stavu objednávky.

4.2 Komponenty



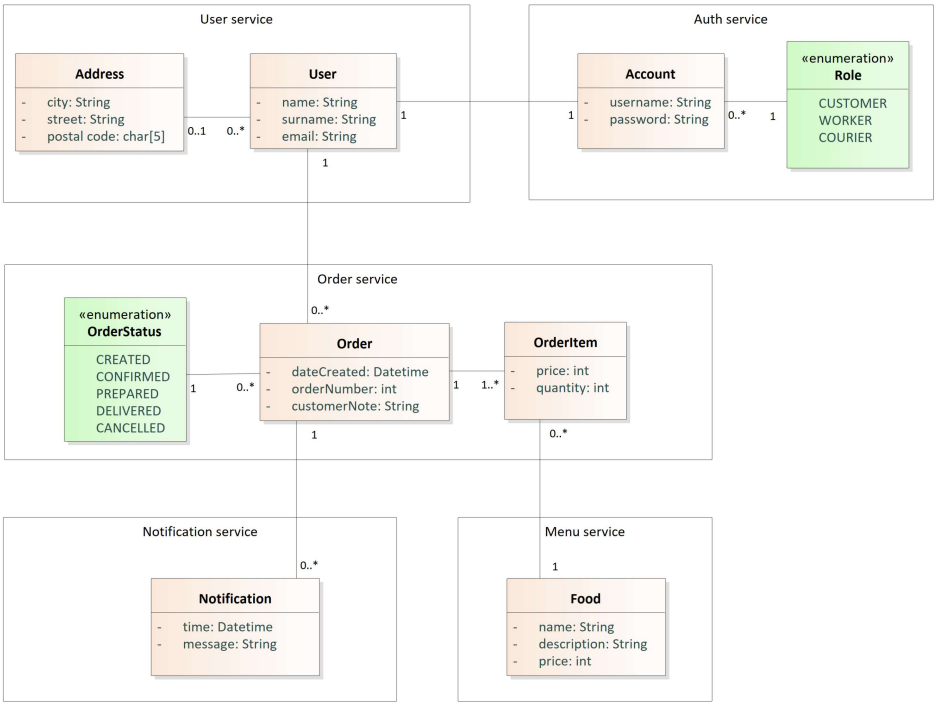
Obrázek 4.1: Diagram komponent

Propojení mezi službami a možnou komunikaci znázorňuje obrázek 5.3. Zde jde vidět, že všechny komponenty komunikují prostřednictvím komponenty s názvem API gateway. API gateway je zprostředkovatelem mezi klientem a mikroslužbami. Od klienta nejprve přijme požadavek, který následně odešle správné mikroslužbě. Při tom může poskytovat i další funkce jako je

autentizace a autorizace. Tím se může stát centrálním bodem bezpečnosti při komunikaci. Komponenta by měla být rychlá, aby nezpomalovala běh celého systému. Objednávková služba komunikující s notifikační službou a uživatelská s autentizační službou posílá požadavky také skrze API gateway.

4.3 Třídy

Podle požadavků z předchozí kapitoly byl vytvořen diagram tříd (viz obrázek 4.2), který obsahuje všechny potřebné třídy a atributy. V tomto případě se nejedná o složitou strukturu a cílem je navrhnout pouze ukázkovou aplikaci. Skupiny tříd jsou ohraničené rámečky pro indikaci, ke které službě patří.



Obrázek 4.2: Diagram tříd mikroservisní aplikace

Monolitická aplikace obsahuje téměř stejné třídy jako je to u mikroservisní aplikace. Jedinou změnou je akorát to, že třída user a account je sdružena do jedné. V mikroservisní aplikaci se mohou objevovat některé odkazy na třídy v rámci jiné služby, a proto je potřeba jej nahradit identifikátorem určující daný objekt. Je to navrhnuto kvůli tomu, že data jsou běžně ukládána v různých databázích, do které ostatní služby nemají přístup. V případě, že by služba potřebovala informace z jiné, musela by si je nejprve přes vystavěné aplikační rozhraní zavolat.

Kapitola 5
Implementace

Podle analýzy a návrhu z předchozích kapitol byla implementována aplikace tak, aby uspokojila její veškeré požadavky a řídila se návrhem tříd a komponent.

5.1 Zvolené technologie

Pro implementaci objednávkového systému jídla byly použity níže vybrané technologie. Tyto technologie byly použity kvůli dřívějšímu seznámení a používání již během předchozího studia a dají se použít jako vhodné řešení pro implementaci.

Java Spring Boot

Pro implementaci aplikace byl vybrán programovací jazyk Java. Konkrétněji byl zvolen framework Spring Boot[1]. Framework je určen pro vyvíjení webových a enterprise aplikací, který ulehčuje vývoj a umožňuje jednoduchou konfiguraci. Spring Boot obsahuje množství komponent, jako je třeba zabudovaný server Tomcat[2] ve kterém se aplikace spouští. Dále je možné přidat pro další funkcionality mnoho dalších dostupných knihoven od třetích stran.

K sestavení aplikace byl použit Maven[3], což je nástroj pro správu projektů. Tento nástroj spravuje závislosti projektu, dokáže kompilovat a zabalit kód do spustitelného formátu (typicky JAR) a mnoho dalšího.

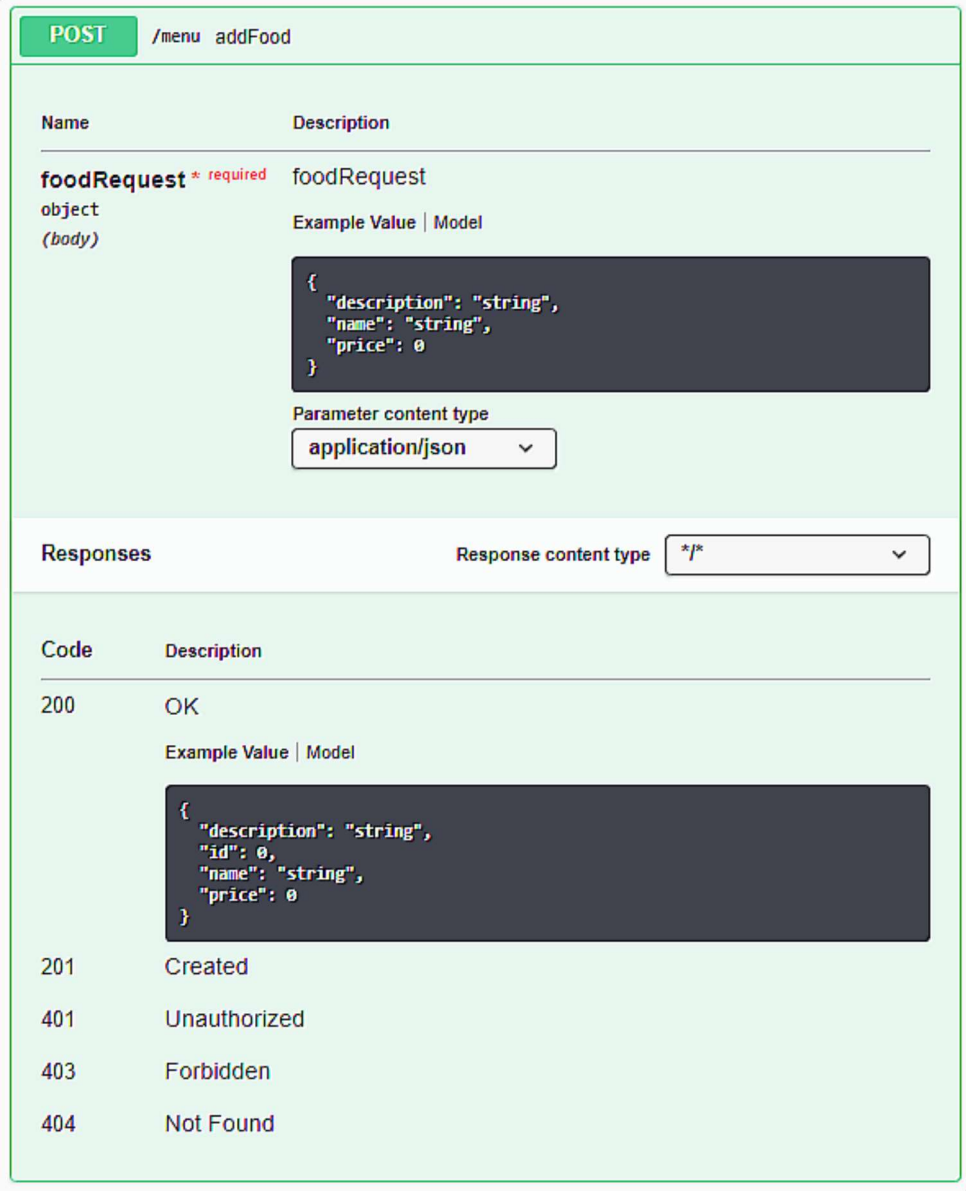
PostgreSQL

Aplikace potřebuje ke své funkčnosti databázi, ve které mohou být perzistentně uložena data, se kterými aplikace pracuje. Byl vybrán open source databázový systém s názvem PostgreSQL[4]. Jedná se o objektově-relační systém, který používá jazyk SQL. Data z databáze jsou pomocí objektově relačního mapování propojeny s entitami ve Spring Bootu.

[1] https://spring.io/
[2] https://tomcat.apache.org/
[3] https://maven.apache.org/
[4] https://www.postgresql.org/docs/

REST API

Pro komunikaci s aplikací bylo vybráno REST API. Přístupové body jsou reprezentací dat pomocí URI a pro manipulaci s nimi slouží HTTP metody. Aby bylo možné testovat monolitickou a mikroservisní aplikaci, je toto rozhraní pro oba přístupy stejně definované. Z aplikace byla vygenerovaná Swagger[5] dokumentace, která tyto API popisuje.



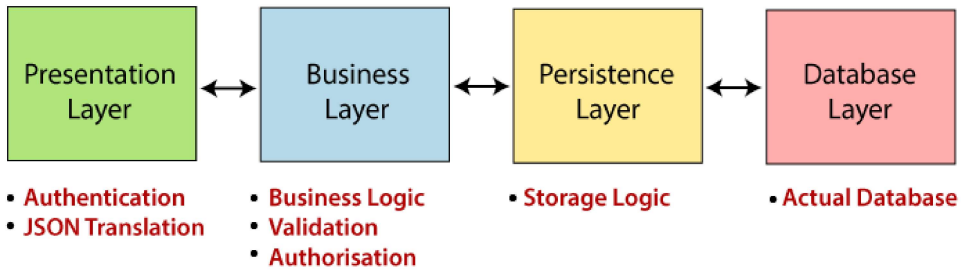
Obrázek 5.1: API dokumentace pro přidání nového jídla do menu

[5] https://swagger.io/

5.2 Monolitická aplikace

Jak již bylo zmíněno v předchozích kapitolách, monolitická aplikace bude obsahovat všechny funkcionality objednávkového systému jídla. Celková aplikace byla rozdělena do vrstev, kde každá vrstva plní specifickou funkci. Díky tomu je projekt přehlednější a při hledání chyby je možné rychleji najít zdroj problému. Struktura byla tedy rozdělena na tři vrstvy: prezentační, byznys a perzistentní vrstvu.

Prezentační vrstva zajišťuje komunikaci s uživatelem a vystavuje rozhraní aplikace. V tomto případě se jedná o zmiňované REST API. Byznys vrstva zajišťuje logiku aplikace a řeší zpracování a validaci dat. Poslední, perzistentní vrstva, se stará o trvalé uložení dat do databáze. Je tedy prostředníkem mezi aplikací a databází.



Obrázek 5.2: Ukázka více vrstevnaté aplikace (zdroj: [21])

Zabezpečení bylo zajištěno pomocí jednoduché HTTP autentizace. Když je potřeba uživatele identifikovat, server vyzve klienta, aby v požadavku zaslal i autentizační informace. Při posílání požadavku se přihlašovací údaje posílají v hlavičce "Authorization"se zakódovanými daty ve formátu base64. Tyto data nejsou ale při přenosu zabezpečena. Tato metoda byla implementována pouze pro demonstrační aplikaci a v reálném případě by se dalo implementovat zabezpečení například pomocí protokolu OAuth[6].

5.3 Mikroservisní aplikace

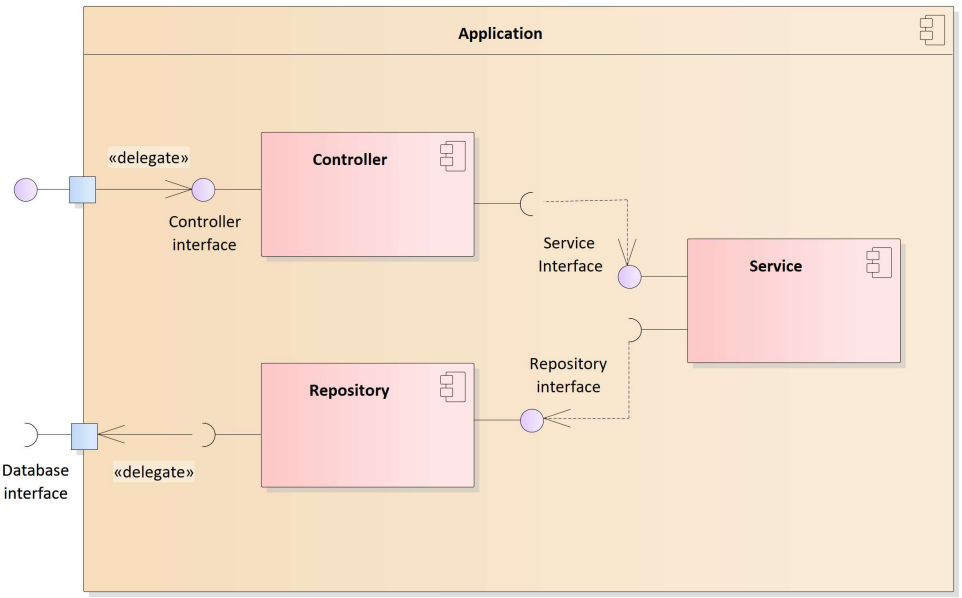
Každá mikroslužba využívá stejné více vrstevnaté rozdělení jako to je u monolitické aplikace. Bylo ale podle návrhu potřeba vytvořit každou službu jako samostatnou aplikaci se svým datovým modelem (viz obrázek 4.2).

Pro komunikaci mezi službami a klientem byla vytvořena ještě další služba jménem API gateway, která byla zmíněna již v předchozí kapitole. Pro implementaci byl vybrán Spring Cloud Gateway[7]. Ten směruje požadavky mikroslužbám podle nastavených cest v konfiguraci.

Když potřebuje služba komunikovat s jinou, pošle požadavek nejprve API gateway. To je docíleno třídou WebClient z modulu spring-webflux. Ta povoluje

[6] https://oauth.net/2/
[7] https://cloud.spring.io/spring-cloud-gateway/reference/html/

snadno posílat HTTP požadavky a je vhodným kandidátem pro komunikaci mezi službami. Toto bylo potřeba u uživatelské služby, která potřebuje poslat autentizační službě data o uchování informací přihlašovacích údajů. Stejně tak potřebuje objednávková služba poslat notifikační službě požadavek s informací, že došlo ke změně statusu objednávky a je potřeba vytvořit příslušnou notifikaci. V API gateway je použito stejné zabezpečení pomocí HTTP autentizace jako u monolitické aplikace. Údaje ohledně přihlášení a rolí jsou uloženy v autentizační službě.



Obrázek 5.3: Vnitřní struktura aplikace

Kapitola 6

Nasazení

Implementované aplikace byly potřeba nasadit pomocí kontejnerů do vybraného orchestrátoru kvůli jednodušší režii a konfiguraci škálování. V této kapitole je popis kontejnerizace aplikace a konfigurace pro nasazení v Kubernetes.

6.1 Docker

Aby byla aplikace a mikroslužby od sebe izolovány a bylo možné je jednodušeji škálovat a nasadit, bylo třeba tyto aplikace kontejnerizovat. Proto byl pro každou aplikaci vytvořený soubor Dockerfile, který předepisuje instrukce pro vytvoření Docker image.

```
1 FROM openjdk:17
2 WORKDIR /
3 COPY ./target/order-service-1.0.0.jar order-service-1.0.0.jar
4 EXPOSE 80
5 CMD ["java", "-jar", "order-service-1.0.0.jar"]
```

Listing 6.1: Ukázka souboru Dockerfile objednávkové služby

Ve vypsaném obsahu souboru (viz listing 6.1) je popsaný sled příkazů k sestrojení image. Nejprve se využije jako základ image pro spuštění aplikace image openJDK s verzí 17. Další řádek určuje aktuální pracovní adresář uvnitř kontejneru. Poté se zkopíruje již vytvořená aplikace do adresáře kontejneru a vystaví se port 80, který bude přístupný zvenčí. Poslední řádek popisuje příkaz, který se spustí při startu kontejneru. V tomto případě se jedná o spuštění zkopírované aplikace.

Pomocí souboru byl sestavený image a následně byl po autentizaci odeslán a uložen v osobním kontejnerovým repozitáři na platformě GitLab[1]. Tam byl pro uložení předem vytvořen privátní projekt.

[1] https://about.gitlab.com/

6.2 Kubernetes

Pro lehčí práci a správu kontejnerů s aplikací byloy nasazeny na platformu Kubernetes. V produkčním prostředí by aplikace běžela na Kubernetes u některého z poskytovatelů cloudových služeb jako je například Google Kubernetes Engine [2] nebo Azure Kubernetes Service[3]. Tyto služby od poskytovatele jsou ale běžně zpoplatněné, a proto byl pro ukázkové nasazení použit osobní počítač, na kterém byl zprovozněn Minikube[4]. Ten umožňuje provozovat Kubernetes lokálně na svém stroji a testovat aplikace v prostředí simulující produkční cluster.

6.2.1 Aplikace

Aby bylo možné se připojit k privátním repozitáři na GitLabu odkud je potřeba získat uložený image pro vytváření kontejnerů, byl pro uchování citlivých přihlašovacích údajů vytvořen Kubernetes objekt s názvem secret. Následně byly vytvořeny pro každou službu a monolitickou aplikaci dva potřebné objekty pro základní nasazení: deployment a service.

```
1  apiVersion: apps/v1
2  kind: Deployment
3  metadata:
4    name: order-service-deployment
5    labels:
6      app: order-service
7  spec:
8    replicas: 1
9    selector:
10     matchLabels:
11       app: order-service
12   template:
13     metadata:
14       labels:
15         app: order-service
16     spec:
17       containers:
18         - name: order-service
19           image: registry.gitlab.com/kimldavi/
20                 food-ordering-system/order-service:1.0.0
21           resources:
22             limits:
23               cpu: "400m"
24               memory: "500Mi"
25           ports:
```

[2] https://cloud.google.com/kubernetes-engine
[3] https://azure.microsoft.com/en-us/products/kubernetes-service/
[4] https://minikube.sigs.k8s.io/docs/start/

```
26              - containerPort: 80
27        imagePullSecrets:
28        - name: registry-credentials
29  ---
30  apiVersion: v1
31  kind: Service
32  metadata:
33    name: order-service
34  spec:
35    type: ClusterIP
36    selector:
37      app: order-service
38    ports:
39      - protocol: TCP
40        port: 80
41        targetPort: 80
```

Listing 6.2: Ukázka Kubernetes deployment a service souboru

Uvedený konfigurační soubor YAML (viz listing 6.2) je určen pro službu objednávek. Jak je možné vidět, soubor obsahuje oba potřebné objekty. To je možné díky třem pomlčkám, které označují konec jednoho dokumentu a začátek druhého. V případě konfigurace deploymentu a service je vhodné je udržovat na jednom místě, kdy je pak stačí aplikovat pomocí jednoho příkazu najednou pomocí příkazu "*kubectl apply -f FILE_NAME*".

Deployment především popisuje jaký kontejner má být spuštěn v rámci jednoho podu. Obsahuje tedy název image, vystavené porty a maximální zdroje, které může jeden pod využít. Dalším důležitým nastavením je pak počet replik. To znamená, kolik podů se má vytvořit a udržovat jejich definovaný počet spuštěný. Díky tomu je možné aplikaci horizontálně škálovat a je dovoleno po aplikování deploymentu počet dynamicky měnit.

Service je definovaná proto, aby směrovala požadavky mezi pody označené selektorem a sloužila jako jeden vstupní bod mezi tyto pody. Když je volán požadavek, je směrován na adresu s názvem definovaným v sekci metadata. Také je v service definováno, na jakém portu bude naslouchat a na jaký port podu bude požadavek přeposílat.

6.2.2 Databáze

V objednávkovém systému jídla je potřeba ukládat data perzistentně do databáze. Tu je možné taky nasadit v Kubernetes jako pod. Pro běh podu byl zvolen image s PostrgeSQL z výchozího repozitáře Docker Hub[5].

V tomto případě je však ale potřeba přiřadit databázi nějakou část datového úložiště, aby data mohla ukládat. Proto byl vytvořen konfigurační soubor s objektem PersistentVolumeClaim (PVC). Díky němu dojde k žádosti o prostor k ukládání dat, který zajistišťuje objekt PersistentVolume. Když

[5]https://hub.docker.com/_/postgres

PersistentVolume odpovídá požadavkům PVC, připojí se k němu a aplikace bude díky tomu moci data ukládat. PersistentVolumeClaim se pak musí přiřadit v konfiguraci deploymentu.

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

Listing 6.3: Konfigurace PersistentVolumeClaim pro nasazení databáze

Kapitola 7

Výkonnostní testy

V předchozích kapitolách byla vytvořena serverová část aplikace objednávkového systému jídla. Nyní je vše připravené aplikaci otestovat výkonnostními testy s použitím horizontálního a vertikálního škálování aplikace.

7.1 Použité nastavení

Pro testování byly vybrány 4 přístupy. V prvních dvou variantách se jedná o klasickou monolitickou a mikroservisní aplikaci vytvořeného systému. Jako další možnost byla vybrána monolitická aplikace, která byla vertikálně naškálována. Na závěr se jedná o mikroservisní aplikaci, kde jsou jednotlivé mikroslužby škálovány horizontálně.

Testování bylo prováděno na nasazených kontejnerech s aplikací a službami v Minikube. Podrobnější popis je k dispozici v přechozí kapitole s číslem 6. Pro rozdělování hardwarových prostředků bylo na osobním počítači k dispozici 3GB paměti RAM a 2,4 CPU. U cloudového poskytovatele je dostupnější mnohem větší výkon, nicméně pro základní porovnání bude osobní počítač stačit.

7.2 Locust

K posílání požadavků byl použit nástroj Locust. Locust je open-source nástroj pro zátěžové testování webových aplikací[13]. Tento nástroj umožňuje vytvářet velký počet uživatelů a simulovat tak uživatele, kteří v daný moment přistupují k aplikaci. Díky tomu Locust umožňuje jednoduché testování škálovatelnosti aplikace. Testovací scénáře je možné definovat pomocí kódu v programovacím jazyce Python. Locust také obsahuje i webové rozhraní, které znázorňuje data o posílaných požadavcích. Informace o průběhu a výsledcích je schopen zobrazit pomocí grafů.

7.3 Scénáře

Rovnoměrné rozložení zátěže

Při tomto scénáři se posílaly požadavky na všechny koncové body rovnoměrně. To znamená, že každý přístupový bod měl stejnou váhu a žádný nebyl preferován. Testy byly spuštěny nejdříve s 100 uživateli a následně s 300 uživateli. 300 uživatelů bylo pro osobní počítač limitem, kdy po navýšení docházelo k nefunkčnosti Minikube a požadavky byly pak odmítány.

Dostupný výkon byl rozdělen mezi služby rovnoměrným rozdělením zdrojů. Každé mikroslužbě bylo tedy přiřazeno 0,4 CPU a 500 MB paměti RAM. Tyto stejné parametry byly pro porovnání přiřazeny i monolitické aplikaci. Po otestování na 300 uživatelích vychází mikroservisní aplikace lépe, kdy se průměrná doba odpovědi liší téměř o jednu sekundu.

Následně byla monolitická aplikace vertikálně naškálována. Tudíž byly aplikaci zvětšeny dostupné hardwarové zdroje na 2 CPU a 2GB RAM. Tyto prostředky by měly dostatečně stačit, aby nedošlo ke 100% využití. Při 100 uživatelích byla propustnost 327 požadavků za sekundu a při 300 uživatelích 336 požadavků za sekundu. Oproti monolitické a mikroservisní aplikaci s menším výkonem každé služby je zde vidět velký rozdíl.

Jako další bylo vytvořeno více instancí jednotlivých mikroslužeb. Jelikož je zde rovnoměrné rozložení, byl výkon rozdělen tak, aby každá služba měla 3 pody. V tomto testu pomohlo horizontální škálování mikroservisní aplikace, kdy vzrostla propustnost u 300 uživatelů o 80 požadavků za sekundu oproti klasické mikroservisní aplikaci. Průměrný čas odpovědi se také snížil.

	Ø doba odpovědi	**Propustnost**
Monolitická aplikace	1164 ms	87 RPS
Mikroservisní aplikace	898 ms	110 RPS
Vertikální škálování	301 ms	327 RPS
Horizontální škálování	632 ms	157 RPS

Tabulka 7.1: Výsledky testů při 100 uživatelích a rovnoměrném rozložení zátěže

	Ø doba odpovědi	**Propustnost**
Monolitická aplikace	3168 ms	93 RPS
Mikroservisní aplikace	2222 ms	111 RPS
Vertikální škálování	885 ms	336 RPS
Horizontální škálování	1507 ms	192 RPS

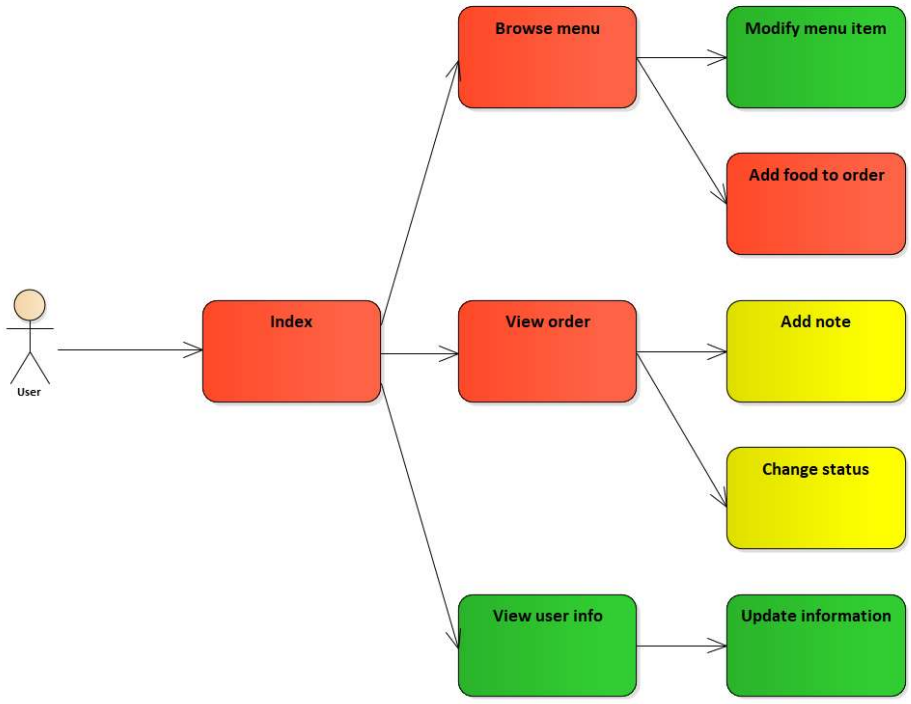
Tabulka 7.2: Výsledky testů při 300 uživatelích a rovnoměrném rozložení zátěže

Celkově aplikaci pomohlo škálování jak vertikální, tak především horizontální v případě mikroservisní architektury.

Standardní použití aplikace

V předchozím scénáři byly všechny koncové body volány se stejnou váhou. Při reálném použití aplikace se ale ne všechny koncové body volají rovnoměrně. Například přidávání nového jídla se nebude volat stejně často jako získání všech jídel. V praxi je potřeba zátěž monitorovat a podle získaných informací upravit počet instancí tak, aby bylo možné zpracovat predikované množství požadavků a systém tak dokázal zvládat nárůst uživatelů.

Proto byl k demonstraci využití aplikace vytvořen HTA[1] diagram. Ten popisuje stromovou strukturu aplikace, která znázorňuje, jakými kroky se uživatel dostane k požadované funkcionalitě. Červeně podbarvené části se budou pravděpodobně používat nejvíce. Žlutě vybarvené budou používány se střední mírou a zeleně ty, které nebudou zas tolik aktivně používány.



Obrázek 7.1: HTA diagram

V připravených testech definovaných pro Locust byla přidána váha červeným tak, aby se požadavky objevovaly 8x více a žlutým byla přiřazena 4x větší váha. To by mělo simulovat reálné použití objednávkového systému jídla. Testy byly opět spuštěny se 100 uživateli a následně s 300 uživateli.

Oproti předchozímu scénáři se již během testu při 300 uživatelích objevovaly u klasické mikroservisní a monolitické aplikace selhání. U mikroservisní aplikace došlo k odmítnutí 45 požadavků. Odmítnuté požadavky byly směrovány pouze mikroslužbě, která obsluhuje koncové body pro menu. Služba byla tedy zatížená a další požadavky v danou dobu přestala přijímat. V

[1] Hierarchical task analysis

případě monolitické aplikace došlo v průběhu testu k odmítnutí 87 požadavků. Tyto požadavky byly ale oproti mikroservisní aplikaci odmítány i z různých koncových bodů obsluhující například uživatele nebo objednávky.

Následně byl test spuštěn na monolitické aplikaci s navýšenými hardwarovými prostředky jako v předchozím scénáři. Aplikaci bylo tedy nastaveno 2 CPU a 2 GB RAM. Díky tomu dokázala aplikace zvládat větší zatížení a k odmítání požadavků kvůli přetížení již nedocházelo.

Při horizontálnímu škálování mikroservisní aplikace bylo množství podů pro službu rozděleno podle HTA diagramu následovně podle tabulky 7.3.

Název služby	**Počet podů**
Menu service	5
Order service	4
Api gateway	3
Authorization service	3
User service	2
Notification service	1

Tabulka 7.3: Přiřazené množství podů pro mikroslužby

Notifikační služba běžela pouze na jednom podu. Nicméně volání této služby dochází jen při změně statusu objednávky, takže nebyla tolik přetížená, aby jich potřebovala více.

V předchozím testu byla mikroslužba pro menu zatížena a odmítala některé požadavky. V tomto testu, kdy bylo přidáno mikroslužbě více podů se zátěž rozložila a již k žádnému selhání nedocházelo. Celkově byla průměrná doba odpovědi díky škálování dvakrát menší než u klasické mikroservisní aplikace. Škálování tedy aplikaci pomohlo výsledky testů zlepšit.

	Ø doba odpovědi	**Propustnost**
Monolitická aplikace	1060 ms	94 RPS
Mikroservisní aplikace	894 ms	110 RPS
Vertikální škálování	250 ms	391 RPS
Horizontální škálování	436 ms	228 RPS

Tabulka 7.4: Výsledky testů při 100 uživatelích a standardním rozložení zátěže

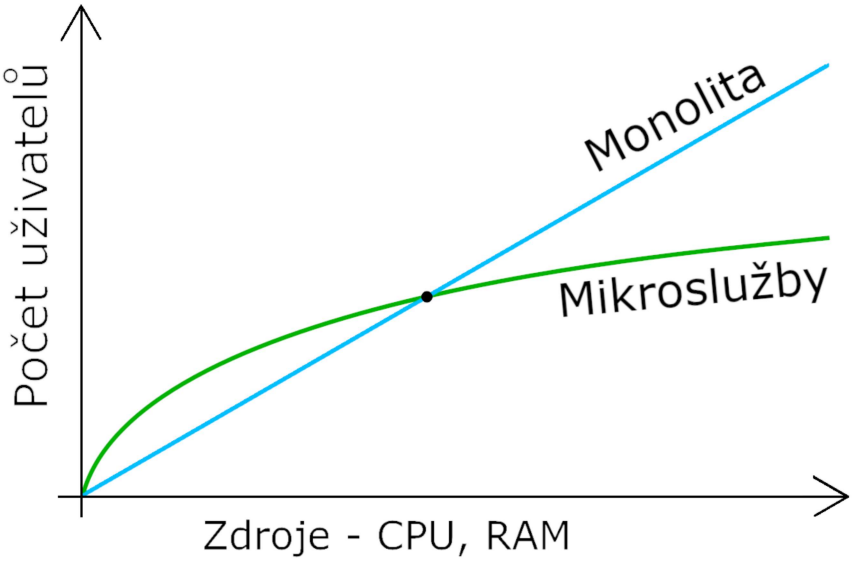
	Ø doba odpovědi	**Propustnost**	**Selhání**
Monolitická aplikace	3040 ms	97 RPS	87
Mikroservisní aplikace	2736 ms	89 RPS	45
Vertikální škálování	799 ms	368 RPS	0
Horizontální škálování	1323 ms	225 RPS	0

Tabulka 7.5: Výsledky testů při 300 uživatelích a standardním rozložení zátěže

7.4 Shrnutí

V předchozí sekci byl na zátěž uživatelů otestován objednávkový systém jídla. Testy ukázaly, že škálování aplikaci pomohlo zvýšit propustnost požadavků a snížit průměrnou dobu odpovědi. Nicméně vertikálně naškálovaná monolitická aplikace si oproti naškálované mikroservisní aplikaci vedla lépe. V případě, kdy bude narůstat počet uživatelů, by se měla mikroservisní aplikace od nějakého bodu dostat k menší potřebě hardwarových zdrojů, protože bude stačit škálovat pouze zatíženou službu (viz obrázek 7.2).

Při velkém počtu požadavků může pomoci upravit i počet vláken na serveru, aby bylo možné toto množství paralelně zpracovávat.



Obrázek 7.2: Graf zachycující HW nároky podle počtu uživatelů

Kapitola 8
Závěr

V poslední kapitole je shrnut text a praktická část v bakalářské práci. Následně jsou zde napsány možnosti, jak na tuto práci navázat v případných dalších budoucích pracích.

8.1 Shrnutí

Cílem této bakalářské práce bylo prozkoumat možnosti škálování aplikace, aby dokázala rychle reagovat na růst uživatelů v daném čase a snížit její zátěž. Důraz byl kladen na mikroservisní architekturu, kontejnerizaci pro jednodušší nasazení a škálování aplikace.

V teoretické části práce proběhla rešerše možností ohledně vertikálního a horizontálního škálování. Při horizontálním škálování byla popsána komponenta s názvem load balancer a jeho používané algoritmy pro přeposílání požadavků. Dále byla popsána mikroservisní architektura a její výhody a nevýhody. Nakonec byly prozkoumány možnosti kontejnerizace aplikací a automatizace jejich běhu pomocí různých kontejnerových orchestrátorů.

V dalších kapitolách byly pokryty vývojové procesy aplikace, kdy byl pro demonstraci škálování navrhnut a implementován systém objednávání jídla. Ta umožňuje základní funkcionality správy a objednávání položek z menu. Aplikace byla navrhnuta jako monolitická a následně byla rozdělena podle domén na mikroservisní architekturu. Naimplementovaná aplikace byla kontejnerizována a nasazena v prostředí Kubernetes na osobním počítači.

V poslední kapitole praktické části byla aplikace pomocí nástroje Locust otestována výkonnostními testy, kde bylo provedeno porovnání doby odezvy a propustnost vytvořené aplikace. Ve scénáři byly mezi sebou porovnány dvě vytvořené aplikace se stejnou funkcionalitou, ale různou architekturou, a jejich naškálované verze. V testech došlo u naškálovaných aplikací k výraznému zlepšení, co se týče propustnosti a doby odezvy.

Tato práce by mohla sloužit jako vhled do problematiky a zdroj informací pro čtenáře, kteří mají o možnosti škálování aplikací zájem.

8.2 Možnosti budoucí práce

V bakalářské práci bylo provedeno škálování jenom na osobním počítači. Pro lepší výsledky by se dala aplikace nasadit u některého z poskytovatelů Kubernetes na cloudu. Zde pak provést testy s daleko více uživateli a vyšším zatížením kvůli většímu množství dostupných hardwarových zdrojů pro jejich rozdělování mezi instance.

V ukázkové aplikaci v mikroservisní architektuře dochází při změně statusu objednávky k volání notifikační služby, kde komunikace probíhá synchronně. V tomto případě se čeká na odpověď od této služby. Tuto komunikaci by bylo možné implementovat i asynchronně. Díky tomu by nebyly služby mezi sebou tak vázány a snížila by se i doba odpovědi.

Další možností pro budoucí práci je prozkoumání škálovatelnosti databází. Ta se pak při zpracovávání většího množství požadavků může stát úzkým hrdlem aplikace. Zde je ale potřeba řešit konzistenci dat a zajišťovat, aby byla stejná data v každé instanci databáze.

Příloha A

Literatura

[1] Martin L Abbott a Michael T Fisher. *The art of scalability: Scalable web architecture, processes, and organizations for the modern enterprise*. Addison-Wesley Professional, 2015.

[2] Klaithem Al Nuaimi et al. "A survey of load balancing in cloud computing: Challenges and algorithms". In: *2012 second symposium on network cloud computing and applications*. IEEE. 2012, s. 137–142.

[3] Marco Autili, Alexander Perucci a Lorenzo De Lauretis. "A hybrid approach to microservices load balancing". In: *Microservices: Science and Engineering* (2020), s. 249–269.

[4] Trieu C Chieu et al. "Dynamic scaling of web applications in a virtualized cloud computing environment". In: *2009 IEEE International Conference on e-Business Engineering*. IEEE. 2009, s. 281–286.

[5] Shahir Daya et al. *Microservices from theory to practice: creating applications in IBM Bluemix using the microservices approach*. IBM Redbooks, 2016.

[6] Octopus Deploy. *Monoliths versus microservices*. URL: `https://octopus.com/blog/monoliths-vs-microservices` (cit. 14. 04. 2023).

[7] D Chitra Devi a V Rhymend Uthariaraj. "Load balancing in cloud computing environment using improved weighted round robin algorithm for nonpreemptive dependent tasks". In: *The scientific world journal* 2016 (2016).

[8] *Docker docs: How to build, share, and run applications*. 2023. URL: `https://docs.docker.com/` (cit. 18. 04. 2023).

[9] Moisés Macero García a Anglin. *Learn Microservices with Spring Boot*. Springer, 2020.

[10] *Gitlab Container Registry*. URL: `https://docs.gitlab.com/ee/user/packages/container_registry/` (cit. 14. 04. 2023).

[11] Kasun Indrasiri a Danesh Kuruppu. *gRPC: up and running: building cloud native applications with Go and Java for Docker and Kubernetes*. O'Reilly Media, 2020.

[12] *Kubernetes Documentation*. URL: https://kubernetes.io/docs (cit. 18. 04. 2023).

[13] *Locust.io*. URL: https://locust.io/ (cit. 30. 04. 2023).

[14] Ian Miell a Aidan Sayers. *Docker in practice*. Simon a Schuster, 2019.

[15] Adrian Mouat. *Using docker: developing and deploying software with containers*. " O'Reilly Media, Inc.", 2015.

[16] Nguyen Dinh Nguyen a Taehong Kim. "Balanced leader distribution algorithm in kubernetes clusters". In: *Sensors* 21.3 (2021), s. 869.

[17] Borislav Đorđević et al. "Performance comparison of Docker and Podman container-based virtualization". In: *2022 21st International Symposium INFOTEH-JAHORINA (INFOTEH)*. IEEE. 2022, s. 1–6.

[18] *Red Hat OpenShift Enterprise Kubernetes Container Platform*. URL: https://docs.openshift.com/ (cit. 30. 04. 2023).

[19] Chris Richardson. *Microservices patterns: with examples in Java*. Simon a Schuster, 2018.

[20] Sandeep Sharma, Sarabjit Singh a Meenakshi Sharma. "Performance analysis of load balancing algorithms". In: *International Journal of Civil and Environmental Engineering* 2.2 (2008), s. 367–370.

[21] *Spring Boot Architecture - TAE*. URL: https://www.tutorialandexample.com/spring-boot-architecture (cit. 30. 04. 2023).

[22] Martin Štefanko, Ondřej Chaloupka, Bruno Rossi et al. "The saga pattern in a reactive microservices environment". In: *Proc. 14th Int. Conf. Softw. Technologies (ICSOFT 2019)*. SciTePress Prague, Czech Republic. 2019, s. 483–490.

[23] Chellammal Surianarayanan, Gopinath Ganapathy a Raj Pethuru. "Essentials of microservices architecture: Paradigms, applications, and techniques". In: (2019).

[24] James Turnbull. *The Docker Book: Containerization is the new virtualization*. James Turnbull, 2014.

Příloha B

Seznam použitých zkratek

API - Application Programming Interface

CI/CD - Continuous Integration/Continuous Delivery

CPU - Central Processing Unit

gRPC - Google Remote Procedure Call

HTA - Hiearchical Task Analysis

HTTP - HyperText Transfer Protocol

HW - Hardware

IoT - Internet of Things

IP - Internet Protocol

JAR - Java ARchive

JSON - JavaScript Object Notation

OS - Operating System

RAM - Random Access Memory

REST - Representational State Transfer

SQL - Structured Query Language

SOAP - Simple Object Access Protocol

URI - Uniform Resource Identifier

XML - Extensible Markup Language

YAML - YAML Ain't Markup Language

Příloha C

Elektronické přílohy

```
/
├── code - zdrojové kódy aplikace
├── kubernetes - soubory YAML pro Kubernetes
├── performance_tests - výsledky testů z Locustu
└── swagger - API dokumentace systému
```