VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
FACULTY OF INFORMATION TECHNOLOGY

ÚSTAV INTELIGENTNÍCH SYSTÉMŮ
DEPARTMENT OF INTELLIGENT SYSTEMS

ORCHESTRACE MODULŮ MULTITENANTNÍCH SYSTÉMŮ

MODULE ORCHESTRATION OF MULTITENANT SYSTEMS

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE **Bc. FILIP JEŘÁBEK**
AUTHOR

VEDOUCÍ PRÁCE **Ing. ALEŠ SMRČKA, Ph.D.**
SUPERVISOR

BRNO 2022

Vysoké učení technické v Brně
Fakulta informačních technologií

Ústav inteligentních systémů (UITS) Akademický rok 2021/2022

Zadání diplomové práce

Student: **Jeřábek Filip, Bc.**
Program: Informační technologie a umělá inteligence
Specializace: Vývoj aplikací
Název: **Orchestrace modulů multitenantních systémů**
Module Orchestration of Multitenant Systems
Kategorie: Informační systémy
Zadání:

1. Nastudujte multitenantní systémy. Nastudujte možnosti použití kontejnerů při orchestraci informačních systémů.
2. Analyzujte požadavky na tvorbu multitenantních systémů z jednotlivých instancí informačních systémů. Navrhněte architekturu pro orchestraci informačního systému s vyčleněným sdíleným aplikačním modulem za účelem multitenance daného systému.
3. Implementujte navržené řešení pomocí systému Kubernetes.
4. Demonstrujte navržené orchestrační řešení.

Literatura:

- IEEE standard 1471-2000. IEEE Recommended Practice for Architectural Description of Software-Intensive Systems. doi: https://doi.org/10.1109/IEEESTD.2000.91944
- Oracle. "Introduction to the Multitenant Architecture." Dostupné na URL: https://docs.oracle.com/en/database/oracle/oracle-database/19/multi/introduction-to-the-multitenant-architecture.html

Při obhajobě semestrální části projektu je požadováno:

- První dva body zadání.

Podrobné závazné pokyny pro vypracování práce viz https://www.fit.vut.cz/study/theses/
Vedoucí práce: **Smrčka Aleš, Ing., Ph.D.**
Vedoucí ústavu: Hanáček Petr, doc. Dr. Ing.
Datum zadání: 1. listopadu 2021
Datum odevzdání: 18. května 2022
Datum schválení: 3. listopadu 2021

Abstrakt

Cílem práce je navrhnout a demonstrovat řešení alternativního převodu aplikace do multitenantní podoby spolu s jejím přesunem ze zákaznického serveru a nasazením na sdílený server firmy, která tento systém vyvíjí. Součástí převodu není klasická modifikace původní aplikace a implementace multitenance přímo do jejího kódu. Pomocí systémů jako je Docker a Kubernetes budou vyčleňovány, kontejnerizovány a orchestrovány moduly původní aplikace, díky čemuž vznikne iluze multitenance. Práce necílí na předložení jednoho řešení, ale apeluje na poskytnutí potřebných znalostí, více variant návrhu a implementace univerzálního demonstračního řešení, kdy je před použitím nějakého z návrhů předpoklad jeho upravení dle specifických potřeb vlastního řešení a jeho následná implementace. Navrhovaná řešení mají za cíl zjednodušení procesu tohoto převodu, což souvisí s ušetřením prostředků, a také poskytnutí možnosti vytvoření iluze multitenance u systémů, kde je klasický postup příliš náročný nebo nemožný.

Abstract

The aim of this thesis is to propose and demonstrate a solution for alternative conversion of an application into multitenant form together with its transfer from a customer server and deployment on a shared server of the company that develops this system. The conversion does not include a classical modification of the original application and implementation of multitenancy directly into its code. Using systems such as Docker and Kubernetes, modules of the original application will be extracted, containerized and orchestrated, creating the illusion of multitenancy. This thesis does not aim at presenting a single solution, but appeals to provide the necessary knowledge, multiple design variants and the implementation of a universal demonstration solution, where before using any of the designs, it is expected to modify it according to the specific needs of the actual solution and its subsequent implementation. The proposed solutions aim to simplify the process of this conversion, which is related to saving resources, and also to provide the possibility of creating the illusion of multitenancy for systems where the classical approach is too difficult or impossible.

Klíčová slova

Orchestrace, Multitenantní, Docker, Kontejnery, Kubernetes, Zjednodušení

Keywords

Orchestration, Multitenant, Docker, Containers, Kubernetes, Simplification

Citace

JEŘÁBEK, Filip. *Orchestrace modulů multitenantních systémů*. Brno, 2022. Diplomová práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedoucí práce Ing. Aleš Smrčka, Ph.D.

Orchestrace modulů multitenantních systémů

Prohlášení

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně pod vedením pana Ing. Aleše Smrčky Ph.D. Uvedl jsem všechny literární prameny, publikace a další zdroje, ze kterých jsem čerpal.

.....
Filip Jeřábek
16. května 2022

Poděkování

Mé poděkování patří panu Ing. Alešovi Smrčkovi, Ph.D za cenné rady, trpělivost, ochotu a odborný dohled při tvorbě této diplomové práce. Dále si poděkování zaslouží také má rodina a nejbližší přátelé, kteří mi při studiu pomáhali, motivovali mě a podrželi v těžkých chvílích.

Obsah

Kapitola 1

Úvod

V práci je popsán návrh úpravy aplikace na její multitenantní verzi, zjednodušeně takovou verzi, jejíž jedna instance bude schopna obsloužit více oddělených uživatelských skupin (tenantů) a nebude nutné pro každého tenanta nasazovat zvláštní instanci této aplikace. Je toho docíleno použitím kontejnerů, což jsou spustitelné balíky, které obsahují část software i vše potřebné k jeho spuštění. Následně je použit nástroj Kubernetes, který slouží na jejich automatickou konfiguraci, řízení a koordinaci, tedy orchestraci. Práce obsahuje podrobnou teorii týkající se dané problematiky, několik různých návrhů na převod a orchestraci, z nichž je každý návrh vysvětlen, ilustrován, jsou zhodnoceny jeho výhody, případně nevýhody, a také jsou popsány modelové situace, kdy je tento návrh vhodné použít. Součástí je i jednoduchá demonstrační aplikace.

Přínosem práce je představení možnosti zjednodušeného přidání funkcionality multitenance aplikaci, která je původně jednotenantní. Důležitým aspektem této úpravy je, že se nebude implementovat multitenance jako taková do jádra původní aplikace, ale pomocí vyčlenění modulů původní aplikace, jejich následné kontejnerizace a orchestrace aplikacemi Docker a Kubernetes se dosáhne iluze multitenance. Díky tomuto řešení je možné výrazně omezit množství kódu, které je nutné v původní aplikaci změnit a tím docílit ušetření zdrojů potřebných pro tento převod.

Dodatečná implementace kompletní multitenance do jakékoli již vytvořené komplexnější aplikace je náročný proces, který stojí spoustu času a peněz. U již existujících aplikací není z důvodu nereálných časových či finančních nároků vždy možné použít klasický přístup úpravy jejího kódu. S řešením, které práce popisuje, je možné převod aplikace realizovat ve fázích a postupně se k multitenantní verzi aplikace blížit. V úvodní fázi se systém bude podobat spíše vytvoření více instancí původní aplikace, kdy bude dosaženo přesunutí aplikace na vlastní servery (Což může být také cílem požadovaného převodu.), ale postupem času bude výsledek obdobou multitenantní verze.

Kontejnerizace je oblast, která se už několik dekád rozvíjí. Jejím největším dnešním přínosem je izolování obsahu kontejnerů od parametrů vnějšího systému a s tím spojené jednodušší instalace a přesuny aplikace na další stroje. Používá se například pro implementaci mikroslužeb, kdy je aplikace rozdělena po minimálních funkčních částech právě do kontejnerů a zjednodušuje se tak vývoj těchto oddělených částí.

Téma vytvoření iluze multitenance u již existující aplikace je zajímavou problematikou. Multitenantní aplikace dosahují významného rozvoje, ale náklady na převod klasických jednotenantních aplikací do multitenantní verze jsou velké. Proto je cílem práce ulehčení právě onoho převodu aplikace do multitenantní verze.

V první kapitole práce s názvem Teorie a zhodnocení současného stavu je shrnuta teorie, která obsahuje veškeré potřebné znalosti pro provedení převodu aplikace podle poskytnutých návrhů. Na začátku se nachází obecný úvod do problematiky multitenantních systémů, a způsobů poskytování aplikací, dále následuje popis technik virtualizace a kontejnerizace. Následující kapitoly jsou podrobným popisem principů a komponent nástrojů Docker a Kubernetes. Na závěr jsou v kapitole zhodnocena současná řešení týkající se problematiky práce a tato řešení jsou zasazena do jejího kontextu.

Na soubor teorie byl kladen důraz proto, aby práce tvořila komplexní oporu a nebylo nutné dohledávat teorii z externích zdrojů. Témata týkající se použitých nástrojů jsou docela rozšířená a na internetu je k dispozici poměrně velké množství článků a seriálů obsahujících informace k dané tématice. Zdroje se ale soustředí vždy jen na část problematiky a je proto těžké spojit informace z různých zdrojů tak, aby tvořily uspořádaný kontext. Specifické použití těchto nástrojů, které práce řeší, je jen minimálně probírané a většina nalezených informací pomocí klíčových slov se týká podobné, avšak principiálně jiné tématiky.

V druhé kapitole práce nesoucí název Návrh řešení jsou předvedena různá řešení návrhu tohoto převodu. Každý návrh řešení je popsán, jsou zhodnoceny jeho výhody a nevýhody, na ilustraci je zobrazena jeho architektura a jsou zmíněny vhodné případy jeho použití. Z důvodu snahy o univerzálnost není cílem vytvoření jednotného návrhu, který by bylo možné použít na převod jakéhokoliv systému. To ani kvůli různorodosti převáděných systémů není možné. Předpokládá se nastudování kontextu problematiky, například z poskytnuté teorie v jedné z kapitol práce, následná analýza vlastního převáděného systému, výběr vhodného návrhu pro převod a jeho úprava dle specifických potřeb systému a implementace. Kapitola obsahuje také popis jednoduchého modelového systému, na kterém byly demonstrovány základní principy orchestrace týkající se tohoto převodu.

Třetí kapitolou práce je Implementace a vyhodnocení, která obsahuje přiblížení implementace navrhnutých řešení a také popisuje detaily demonstračního řešení. Problematika, která byla třeba nastudovat k vytvoření této práce je velice komplexní, a proto v rozsahu této práce je předvedeno pouze tohle zjednodušené, ale funkční řešení. K částem implementace, které nejsou součástí demonstračního řešení, kapitola poskytuje alespoň základní informace a vhodné zdroje. Právě rozšíření demonstračních řešení a jejich rozbor je vhodným možným navázáním na tuto práci.

Kapitola 2

Teorie a zhodnocení současného stavu

Kapitola týkající se teorie tvoří velkou a důležitou část této práce. Převod aplikace do požadované simulované multitenantní podoby je velice komplexní proces a rozhodně nepřipadá v úvahu pouze jeden přístup k dosažení této podoby a dokonce ani finální podoba nebude vždy stejná. Jak postup tak i výsledek bude záležet na spoustě aspektů, které se budou odvíjet od povahy aplikace, ale i případu použití, či vnitřních politik správců systémů nebo aplikací. Právě z důvodů velké variability a pravděpodobné nutnosti přizpůsobení řešení byl při tvorbě práce na teorii kladen velký důraz, jehož výsledkem je relativně stručný výtažek všech informací potřebných k převodu systému, který neslouží pouze jako podklad pro demonstrované řešení, ale obsahuje i informace navíc, které nejsou přímo použité v popsaném návrhu a implementaci, ale v případě hledání alternativního řešení je možné je využít.

2.1 Cloud computing a multitenance

Multitenance [1, 6, 7] je důležitým aspektem modelu nazývaného Cloud Computing[1], který je bez pochyby jeden z největších trendů dnešního vývoje aplikací. Cloud computing je možné charakterizovat jako provádění pracovní zátěže v cloudovém prostředí. Cloud computing se dělí na tři základní typy – IaaS, PaaS a SaaS. Z těchto tří typů je pro účel multitenantní aplikace nejdůležitější SaaS.

Tenant z pohledu aplikací označuje jednu entitu, které je SaaS aplikace poskytována. Typicky se jedná o uživatele jedné organizace.

2.1.1 SaaS

Software as a service [9] je způsob nasazení aplikace, kdy poskytovateli na serveru běží z pravidla jedna instance dané aplikace a zákazníci k ní přistupují vzdáleně, například pomocí internetového prohlížeče. Uživatelé neplatí za službu(aplikaci) jako takovou, ale za její využívání. Výhodou tohoto způsobu je absence nutnosti zákazníků spravovat aplikaci. O veškerou opravu chyb a nasazování se stará poskytovatel. Do aplikace se přistupuje hromadně pomocí prohlížeče a není nutné aplikaci na každý počítač instalovat zvlášť. Příkladem aplikace poskytované tímto způsobem je webová emailová služba Gmail[2].

[1] https://www.redhat.com/en/topics/cloud
[2] https://www.google.com/gmail/

2.1.2 Víceuživatelské versus multitenantní systémy

V těchto dvou pojmech je zásadní rozdíl [4]. Termín víceuživatelský (V anglickém jazyce se používá „multiuser“, proto jsou tyto termíny snadněji zaměnitelné.) vypovídá o tom, že systém může používat více uživatelů, ale neříká nic o architektuře systému. Na druhou stranu termín multitenantní o systému říká, že je víceuživatelský a zároveň popisuje i architekturu systému. Multitenantní systém je tedy nutně vždy víceuživatelský, ale víceuživatelský systém nemusí být vždy multitenantní.

2.1.3 Multitenantní aplikace

Multitenantní aplikace umožňují více uživatelům (tenantům) užívat jednu a tu samou instanci aplikace bez toho, aniž by se vzájemně mohli jakkoliv ovlivnit. Základním aspektem je právě oddělení prostorů jednotlivých tenantů. Multitenance by měla působit dojmem, že uživatelé aplikaci používají jako jediní.

Výhodou multitenance je možnost nasazení jedné instance aplikace a její užití velkým množstvím tenantů. Tenanti mezi sebou sdílí prostředky v rámci aplikace. Což znamená, že jedna služba nemusí běžet zvlášť pro každou potenciální instanci jednotenantní aplikace, ale stačí jeden její běh a tenanti ji mohou paralelně využívat. Kromě výhod ale multitenantní aplikace přináší také nevýhody. Je důležité se soustředit na izolaci jednotlivých prostředí. Při pochybení by mohlo dojít k napadení prostředí tenanta a ztrátě, poškození, či odcizení dat. Proto je dobré dodržovat některé bezpečnostní požadavky pro multitenantní aplikace:

1. Jeden tenant by měl být v rámci multitenantního systému „uzavřen“ ve vlastním odděleném prostoru. Veškerá data, která mu náleží, by neměla být žádným způsobem přístupná jakýmkoli dalším tenantům.

2. Pokud je umožněno nějakým způsobem v rámci tenantu modifikovat nebo konfigurovat aplikaci, pak se tyto změny musí projevit pouze v rámci prostředí daného tenantu a v žádném jiném.

3. Výměna dat mezi tenanty musí probíhat pouze po předem definovaných a zabezpečených rozhraních.

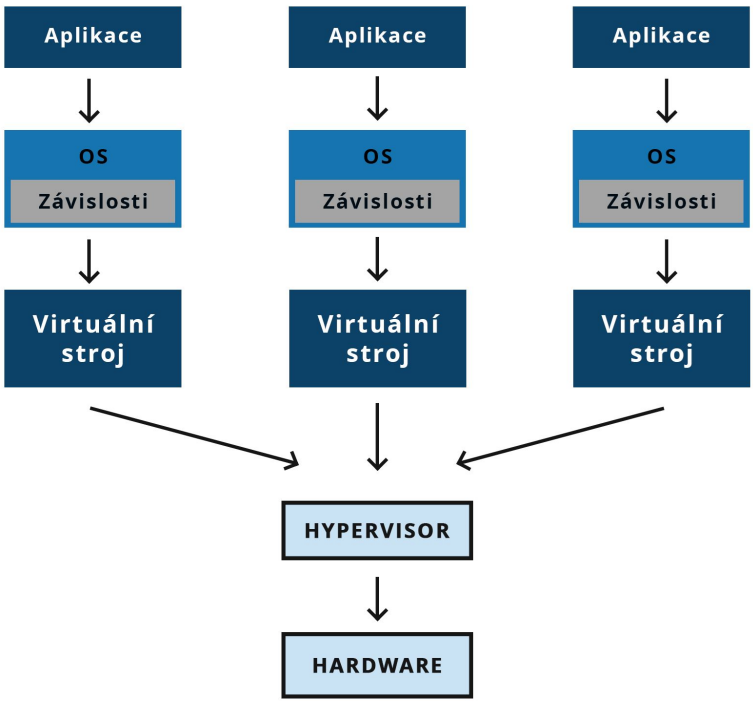
2.2 Virtualizace versus kontejnerizace

Kontejnerizace a virtualizace [8, 13] jsou na jednu stranu velice podobné věci, plní podobný účel, ale na druhou stranu jsou při konkrétním praktickém využití dost odlišné. Kontejnery odstraňují nedostatky plné virtualizace a jsou tak jejím vylepšením.

U plné virtualizace se na fyzický server nainstaluje softwarová komponenta jménem hypervisor, která umožňuje vytvářet virtuální stroje. Každý virtuální stroj, který vznikne, se chová jako samostatný server s vlastním operačním systémem. Tím ale vznikají velké režijní náklady, protože provoz hypervisoru a jednotlivých operačních systémů může spotřebovat až 20 procent výkonu serveru. Na obrázku 2.1 se nachází vizualizace plné virtualizace.

Na druhou stranu při kontejnerizaci dochází k virtualizaci jádra operačního systému. Všechny kontejnery tedy běží v rámci jednoho operačního systému a sdílejí paměť, knihovny a další zdroje, což oproti plné virtualizaci dopomáhá snižovat režijní náklady a dosáhnout lepšího využívání zdrojů. Vizualizace kontejnerizace se nachází na obrázku 2.2. Spuštění kontejneru je také mnohem rychlejší, než spuštění virtuálního stroje s instalací operačního

systému. Mezi další výhody kontejnerů patří například jejich možnost izolace od okolního prostředí a následné nasazení v různých prostředích.



Obrázek 2.1: **Plná virtualizace.** Každá virtualizace potřebuje svůj vlastní operační systém a jeho závislosti. Převzato z článku KONTEJNERY A VIRTUALIZACE [13].

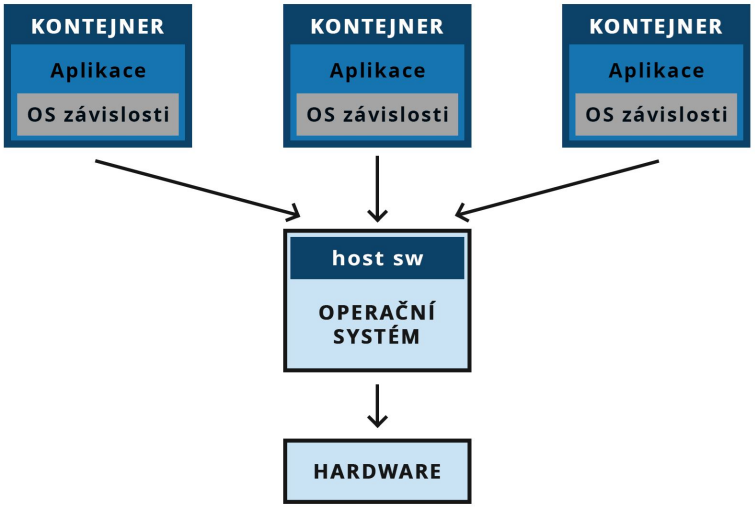
2.2.1 Kontejnery

Kontejnery [13] mají s virtualizací mnoho společného, ale jak již bylo zmíněno v kapitole 2.2; nejedná se o stejný typ technologie. Z bezpečnostního hlediska se kontejnery mohou zdát zranitelnější než plná virtualizace, ale díky úsilí firem vyvíjejících nástroje pro správu kontejnerů tomu tak není. Více o bezpečnosti kontejnerů se nachází v kapitole 2.2.1.1.

Kontejner je standardní jednotka softwaru, která izoluje software od prostředí, na kterém běží, díky čemuž docílí stejného chování na více různých strojích. Když tedy chceme aplikaci spustit na jiném stroji, není nutné instalovat všechny konkrétní verze potřebných nástrojů.

Při vytváření kontejneru se uloží běhové prostředí dané aplikace společně se všemi potřebnými binárními soubory, závislostmi a konfiguracemi. Ty poté dohromady tvoří jeden ucelený balík, který ke spuštění potřebuje pouze aplikaci, která s daným typem kontejneru umí pracovat.

Nejčastěji se kontejnerizace používá při vývoji mikroslužeb, což je způsob, kdy se celá aplikace skládá z minimálních oddělených modulů uzavřených v kontejnerech, které spolu komunikují pomocí API. To zjednodušuje jak vývoj, tak údržbu, protože moduly jsou malé, jednoduché a ucelené části a lze je modifikovat bez nutnosti zásahu do celé aplikace. Zde hrají důležitou roli také orchestrátory, které se starají o správu velkého množství kontejnerů, díky kterým je možné například při nasazení nových verzí aplikací nechat spuštěné repliky staré i nové verze a při problémech plynule přejít zpět na původní verzi – rolling update.



Obrázek 2.2: **Kontejnerizace.** Kontejnery sdílí jeden operační systém hosta a jeho software. Převzato z článku KONTEJNERY A VIRTUALIZACE [13].

Dále pak mezi obrovskou výhodu kontejnerizace patří velké možnosti škálovatelnosti, kdy je při změně náporu velice jednoduché zduplikovat nebo zredukovat množství aplikačních kopií. Navíc jsou jednotlivé části v kontejnerech od sebe relativně oddělené, čímž přispívají k bezpečnosti z pohledu modularity.

Pro vysvětlení je možné přirovnat Docker kontejner (Docker je nástroj pro práci s kontejnery popsaný v kapitole 2.3.) ke kontejneru používanému pro přepravu zboží lodní dopravou. Jen Docker kontejnery místo zboží obsahují software. Každý kontejner tedy obsahuje softwarový image (kapitola 2.3.1), což je jeho „náklad“ a jako s jeho fyzickým protějškem je možné s kontejnerem dělat nějaké operace. Může být například vytvořen, zapnut, zastaven, restartován a zničen. Stejně jako u reálného kontejneru, Docker při práci s Docker kontejnerem nehledí na obsah tohoto kontejneru, ať kontejner obsahuje databázi, aplikaci, nebo něco jiného, bude s ním nakládáno vždy stejně.

2.2.1.1 Bezpečnost kontejnerů

Jak již bylo zmíněno dříve, kontejnery využívají jádro hostitelského systému (na rozdíl od virtualizace, kde je využíván hypervisor), což může vyvolávat otázku, zda je jejich spuštění bezpečné. Společnosti poskytující nástroje pro práci s kontejnery naštěstí tyto problémy usilovně řeší. Například aplikace Docker (kapitola 2.3) nabízí podpisovou infrastrukturu, kde každý kontejner lze podepsat a zvýšit tím jeho důvěryhodnost. Pokud jsou nalezeny nějaké nedůvěryhodné/nepodepsané kontejnery, je uživatel upozorněn.

Další možností zabezpečení, kterou poskytují aplikace spravující kontejnery, je možnost skenování řešení pro hledání zranitelnosti jednotlivých kontejnerů. Příklad služby soustředící se na tento typ zabezpečení je Twistlock[3].

[3] https://www.esecurityplanet.com/products/twistlock/

2.3 Docker

V této kapitole jsou použita a skloňována původní anglická slova jako „Docker“ a „cluster“, v odborných textech pojednávajících o dané tématice jsou tato slova běžně používána a stejně tak i jejich vyskloňované tvary a bylo by matoucí zde používat jejich překlady.

Tato kapitola je přeloženým výtažkem knihy The docker book[4] [17] doplněným o informace z blogu TechWorld with Nana [12], který obsahuje spoustu článků a výukových videí týkajících se této problematiky.

Docker je open-source engine, který slouží k automatizaci nasazování aplikací do kontejnerů popsaných v kapitole 2.2.1. Je navržený tak, aby poskytoval „lightweight“[5] a rychlé prostředí, ve kterém je možno jak spouštět kód, tak i efektivně převádět kód z počítače do testovacího a produkčního prostředí. Mezi výhody Dockeru patří:

- **Jednoduchý způsob modelování reality, který není náročný na zdroje.** Docker je rychlý, „dockerizovat“ aplikaci zabere pouze pár minut. Využívá modelu „copy-on-write“, díky kterému se provedené změny projeví rychle – změní se pouze to, co chceme změnit. Většina kontejnerů spravovaných Dockerem se zvládne nastartovat za méně než sekundu.
- **Rychlý a efektivní životní cyklus vývoje aplikace.** Docker cílí na redukci času cyklu mezi psaním kódu a jeho testováním, nasazením a užíváním. Aplikace je jednoduše přenositelná a sestavitelná, díky čemuž je zjednodušená i spolupráce při vývoji aplikace.
- **Podporuje architekturu orientovanou na služby.** Doporučuje se, aby v každém procesu běžela pouze jedna aplikace nebo služba. Díky čemuž je vytvořen distribuovaný aplikační model, kde se aplikace nebo služba skládá z řady vnitřně propojených kontejnerů, díky čemuž je aplikace jednoduchá na distribuci, škálovatelnost, ladění a introspekci.
- **Izolace aplikace od okolí.** Díky izolaci je možné pohodlně provádět testy.

Docker engine se dříve používal v pracovních uzlech (kapitola 2.4.2.2), které jsou součástí architektury nástroje Kubernetes (kapitola 2.4). Kubernetes využíval některé prvky Docker Engine – například CLI, API, Server, Volumes, Container Runtime, nebo Network. Časem ale vznikly alternativy na tyto prvky přímo v rámci Kubernetes – například Kubernetes CLI, K8s Volumes, nebo K8s Network (K8s je zkratka používaná pro Kubernetes). Ze všech původních prvků použitých z Docker Engine zůstala nutnost použití pouze Container Runtime (který je zodpovědný za správu životního cyklu kontejnerů). Vývojáři Kubernetes se rozhodli označit „dockershim“, což je prvek Kubernetes zodpovědný za interakci s Docker engine, jako zastaralý (deprecated), čímž ukončili jeho vývoj a už ho není doporučené používat. Veškeré dříve vytvořené Docker images mají zaručené, že budou fungovat jako doposud [3, 5].

Aktuálně je tedy místo Docker prvků doporučeno používat alternativy v Kubernetes. Dříve zmíněný Container Runtime, který spadá pod Docker a je jeho součástí, je nyní vyvíjen jako samostatná komponenta pod názvem „containerd“[6]. Výhodou tohoto nového

[4] https://www.docker.com/

[5] **Lightweight** systém je navržený tak, aby měl sám o sobě malé nároky na RAM, CPU a celkově na systémové zdroje.

[6] https://containerd.io/

řešení je snížení nároků na zdroje (Z důvodu omezení množství použitých prvků, které se implementovaly spolu s Docker.) a také zvýšení bezpečnosti, protože se snížením množství komponent se snížilo i množství potenciálních rizik. Aktuálně se jedná o druhý nejpopulárnější container runtime.

Pro vývojáře, kteří používají Kubernetes společně s Dockerem v rámci nějakých cloudových služeb, kde se nestarají o přímou konfiguraci clusteru, nahrazení Docker engine samostatným containerd nic neznamená. Společnosti poskytující tyto služby nahradí a změní container runtime. Vrstva práce s kontejnery je abstraktní a vývojář s ní nepracuje. Nejsou třeba ani žádné změny. Stejně tak se změny neprojeví při lokálním spuštěni za pomoci Minikube (kapitola 2.4.2.10).

V případě, že vývojář používá vlastní server a vlastní konfiguraci Kubernetes clusteru společně s původním Dockerem nainstalovaným jako container runtime, pak je možnost nahrazení původního dockershim implementovaného v Kubernetes za dockershim. Ten je nyní vyvíjen jako samostatná komponenta firmou Mirantis[7] a je možné ho nainstalovat zvlášť do clusteru a dále tak používat komponenty Docker engine.

I přes tuto skutečnost je stále Docker užitečný při použití Kubernetes. Docker umí vytvářet images, které se poté nahrají do Kubernetes clusteru. Díky standardizaci „OCI“[8] (Open Container Initiative) je možné spustit Docker image v jakémkoli Container Runtime, který dodržuje tento standard.

2.3.1 Hlavní komponenty Dockeru

- **Klient a server** – Docker pracuje na architektuře klient-server. Docker klient komunikuje s Docker serverem nebo démonem[9] (anglicky deamon), který provádí veškeré operace. Docker obsahuje klientskou příkazovou řádku a také „full RESTful API“[10]. Je možné spouštět jak klienta, tak i démona, na stejném hostitelském stroji, nebo je možné lokálního klienta připojit na vzdálený server, na kterém běží démon. Obrázek 2.3 zobrazuje architekturu Dockeru.

- **Images** – Image se dá přirovnat k stavebnímu bloku Dockeru. Je to zdrojový kód pro kontejnery, ty na nich závisí a spouští se pomocí images. Jedná se o vrstvený formát využívající „Union file systems“[11], který vypadá například takto:

  – Přidat soubor.
  – Spustit příkaz.
  – Otevřít port.

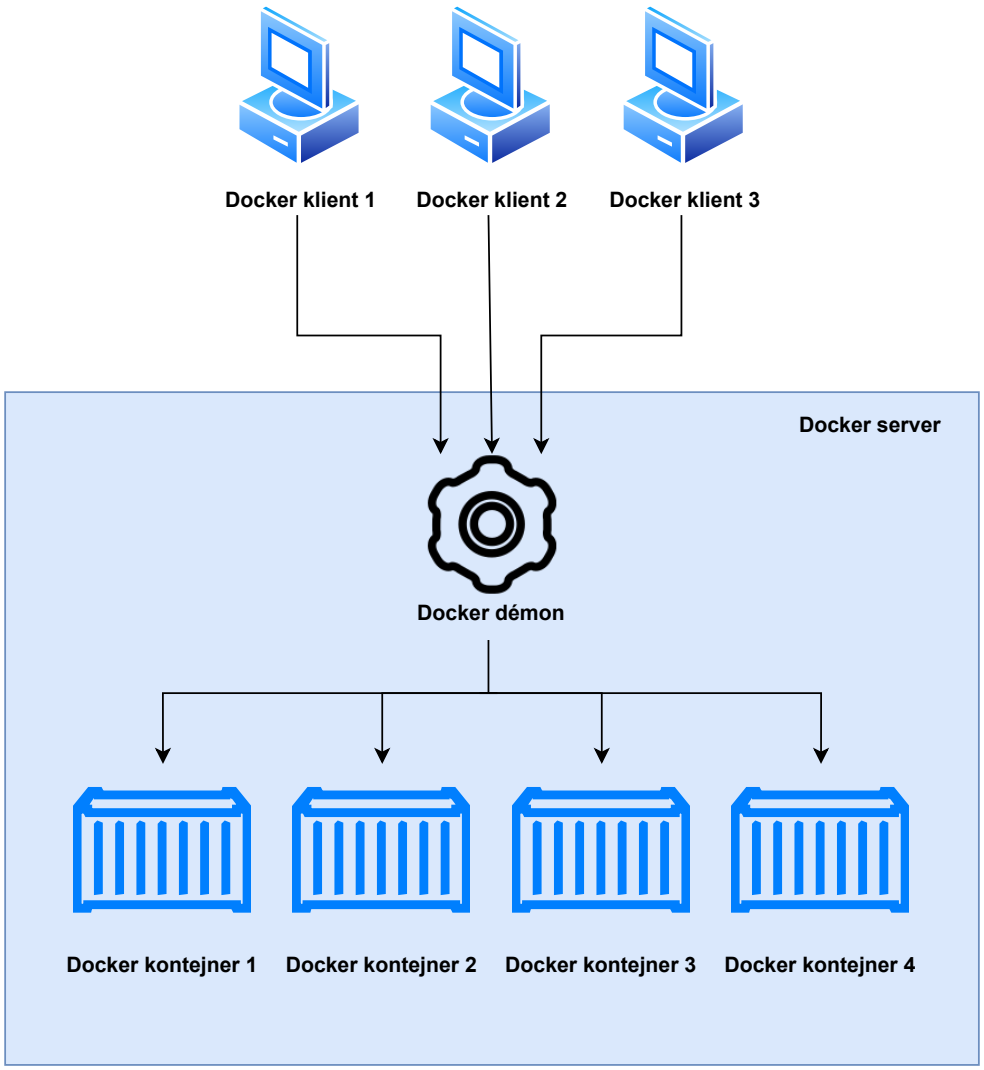
  Images jsou jednoduše přenositelné, mohou být sdíleny, ukládány a aktualizovány.

- **Registry** – Registr[12] je serverová aplikace, která slouží jako úložiště Docker images. Existují dva základní typy registru – veřejné a privátní. Příkladem veřejného registru je Docker Hub[13], který obsahuje přes 100 000 Images. Docker Hub ale umožňuje

[7] https://www.mirantis.com/blog/mirantis-to-take-over-support-of-kubernetes-dockershim-2/
[8] https://opencontainers.org/
[9] **Docker démon** poslouchá API server a spravuje objekty Dockeru (například images nebo kontejnery).
[10] https://docs.docker.com/engine/api/
[11] https://medium.com/@knoldus/unionfs-a-file-system-of-a-container-2136cd11a779
[12] https://docs.docker.com/registry/
[13] https://hub.docker.com/

ponechat images i jako privátní. Je také možné si vytvořit vlastní registr a ukládat images do něj.

- **Kontejnery** – Obecná teorie týkající se kontejnerů již byla zmíněna v kapitole 2.2.1.



Obrázek 2.3: **Architektura Dockeru.** Na obrázku je vidět architektura typu klient-server, tedy Docker deamon spravující Docker kontejnery a k němu přistupující Docker klienti.

2.3.2 Perzistence dat – Docker volume

Každý běžící kontejner má svůj vlastní virtuální souborový systém. Do toho si ukládá svá data. Problém tohoto virtuálního souborového systému je, že není perzistentní a při zániku kontejneru jsou data v něm uložena smazána. Řešením tohoto problému je Docker volume, které slouží k přiřazení části fyzického souborového systému hostitele ke kontejneru. Jedná se z pravidla o konkrétní složku. Existují tři základní druhy Docker volumes:

- **Host volumes** – při definici tohoto typu volume se určuje jak cesta v rámci kontejneru, odkud má být hostitelské úložiště přístupné, tak i cesta v rámci hostitelského

souborového systému. V tomto případě tedy definujeme konkrétní složku na hostitelském úložišti.

- **Anonymous volumes** – definice tohoto typu obsahuje pouze cestu v rámci kontejneru, ale už ne cestu v rámci hostitelského systému. O část na hostitelském úložišti se stará Docker, který například ve složce „/var/lib/docker/volumes/“ vytvoří složku nesoucí název náhodného řetězce a v něm složku „_data“.
- **Named volumes** – jedná se o nejpoužívanější typ volumes. Je to varianta, která je vylepšením předchozích anonymous volumes, kdy není specifikována konkrétní složka na hostitelském úložišti, ale je pro ni vytvořený pojmenovaný alias, pomocí kterého se pak můžeme odkazovat.

2.3.3 Nevhodné případy použití Dockeru

Ačkoliv umí kontejnerizace ve spojení s Dockerem být velice užitečná, existují jisté nevýhody a jsou případy, kdy je lepší se jejich použití vyhnout [15].

- **Při nutnosti zrychlení běhu aplikace** – Jen samotné použití Dockeru aplikaci nezrychlí. Docker funguje tak, že využije tolik systémových prostředků, kolik mu hostitelský plánovač jádra dovolí. To způsobuje jeden paradox. Pokud neomezíme velikost paměti nebo výkon CPU, které může docker použít, pak se může stát, že si Docker vezme moc výkonu a hostitelský kernel detekuje nedostatek prostředků pro běh systémových funkcí a začne procesy Dockeru „zabíjet“, což způsobí zpomalení nebo zhroucení.
- **Při velkém důrazu na bezpečnost** – jak již bylo řešeno v kapitole 2.2.1.1 – kontejnery jsou sice od sebe izolované, ale všechny mají přístup k hostitelskému jádru systému. Pokud tedy použijeme nějaký neověřený kód, který se bude snažit nabourat nebo poškodit hostitelský počítač, může se stát, že se k hostitelskému jádru dostane. Navíc, pokud používáme Docker, pak Docker deamon má práva root, pokud se pak nějaký proces dostane přes izolaci kontejneru, bude mít na hostitelském systému stejná práva jako měl v kontejneru. Prevence proti tomuto problému je nestahovat předpřipravené kontejnery z neověřených zdrojů a používat jen ty ověřené či vlastní. Například Docker Hub označené kontejnery, které splňují určitě požadavky na bezpečnost, efektivitu implementace apod. takto označuje.
- **Při tvorbě desktopové aplikace s grafickým rozhraním** – Docker je orientovaný spíše na izolaci aplikací s konzolovým rozhraním. Aplikace s grafickým rozhraním nejsou prioritou a podpora je jen pro specifické případy a aplikace. Kontejnery běžící na operačním systému Windows navíc nedovolují uživatelům spouštět grafické rozhraní vůbec.
- **Při nutnosti variability OS** – Při použití plné abstrakce a hypervisoru není problém užití různých hostitelských operačních systému pro jeden image. Pří kontejnerizaci a použití Dockeru v tomto případě nastává problém. Image spouštěný na jiném operačním systému, než byl vytvořený, může pracovat neefektivně, nebo vůbec.

Docker je velmi užitečný nástroj pro izolaci aplikací do samostatných kontejnerů. Takovou kontejnerizovanou aplikaci může být možné nainstalovat i jediným příkazem. Neméně užitečný je při automatickém testování, kde vytváří izolaci prostředí pro každý test. Ovšem

i přes jeho mnohé výhody jsou situace, kdy jeho použití nemusí přinést užitek, ba naopak, jeho použití někdy může být i na škodu. Jako u každého nástroje, tak i u Dockeru existují jeho alternativy, které mají různé výhody a nevýhody (například „rkt“[14] nebo „OpenVZ“[15]).

2.4 Kubernetes

V této kapitole jsou použita a skloňována původní anglická slova jako „cluster“ a „pod“, v odborných textech pojednávajících o dané tématice jsou tato slova běžně používána a stejně tak i jejich vyskloňované tvary a bylo by matoucí zde používat jejich překlady.

Tato kapitola je překladem výtahu z knihy „The kubernetes book“ [14] a wiki stránek o systému Kubernetes „cloud native wiki by aqua“ [2] doplněném o informace z blogu TechWorld with Nana [12], který obsahuje spoustu článků a výukových videí týkajících se této problematiky.

Kubernetes je orchestrátor pro kontejnerizované cloud-native aplikace využívající mikroslužby (Anglicky – Orchestrator for containerized cloud-native microservices apps.). **Orchestrátor** označuje systém, který spravuje a nasazuje aplikace. Po nasazení aplikace dynamicky reaguje na změny. Kubernetes například provádí:

- Nasazení aplikace,
- automatické škálování na základě poptávky,
- automatické obnovení nefunkčních součástí,
- nasazování nových verzí s nulovým časem, kdy je služba nedostupná. Stejně tak i vrácení do předchozí verze (anglicky zero-downtime rolling updates and rollbacks).

Kontejnerizace je popsaná v kapitole 2.2.1. Kromě kontejnerů dokáže Kubernetes orchestrovat například virtuální stroje, ale orchestrace kontejnerizovaných aplikací je nejčastější případem jeho použití.

Cloud-native aplikace jsou navrženy tak, aby splňovaly moderní obchodní požadavky, mezi které patří již dříve zmíněné automatické škálování, automatické obnovení nefunkčních částí a zero-downtime rolling updates a které mohou využívat systém Kubernetes.

Aplikace využívající mikroslužby jsou druh aplikací, které jsou seskládány z malých, samostatných a navzájem komunikujících částí, které dohromady tvoří výslednou aplikaci. Tyto oddělené části mohou být vyvíjeny samostatně v různých programátorských týmech. Výhodou tohoto přístupu je oddělení jednotlivých součástí, díky čemuž je možné jednoduše editovat části aplikace bez rizika nechtěného ovlivnění jiné části. Zvyšuje se tím také přehlednost a zjednodušuje se předání jinému programátorskému týmu. Vývoj aplikace tímto způsobem je velice důležitým aspektem cloud-native aplikací.

Kubernetes má jako každý nástroj i své alternativy. Vzhledem k obsáhlosti tohoto systému existuje mnoho alternativ, které se specializují pouze na část oblasti, kterou spravuje Kubernetes. Zde jsou příklady těchto alternativ spolu s jejich použitím, výhodami a nevýhodami:

- **Azure Container Instances** – Jedná se o službu, která slouží k nasazování kontejnerů na Microsoft Azure cloud bez nutnosti definovat nebo spravovat infrastrukturu. Podporuje jak windowsové kontejnery, tak i linuxové. Není zde nutné konfigurovat

[14] https://github.com/rkt/rkt
[15] https://openvz.org/

virtuální stroje, nebo implementovat orchestraci kontejnerů jako v Kubernetes. Nový kontejner lze jednoduše nasadit pomocí portálu Azure nebo Azure konzole.

- **Google Cloud Run** – Platforma umožňující spouštění kontejnerových image jako bezstavové (stateless) automaticky škálovatelné HTTP servisy. Výhodou této služby je jak již bylo řešeno automatické škálování na základě množství příchozích požadavků na dané běžící kontejnery a také možnosti řízení omezení maximálního množství požadavků na konkrétní kontejner.

- **Nomad** – Jedná se o systém vyvíjený firmou HashiCorp, který slouží jako kontejnerový orchestační nástroj. Umožňuje nasazovat jak kontejnery, tak i „legacy“[16] aplikace stejným postupem. Mezi výhody patří možnost pluginů zařízení, podpora GPU nebo možnost škálování až do 10 000 nodů na cluster.

2.4.1 Kombinace Kubernetes a Docker

Kubernetes a Docker jsou technologie, které se vzájemně doplňují. Často se aplikace vyvíjí v Dockeru a Kubernetes se použije na orchestraci její produkční verze. Docker Engine se také dříve používal jako „container runtime“, dnes už se používá vyčleněná část Dockeru – contained, která obsahuje, oproti kompletnímu Docker engine, pouze to, co Kubernetes potřebuje. Nahrazení Docker engine je více popsáno v kapitole 2.3.

2.4.2 Základní komponenty, principy a architektura systému Kubernetes

Kořen architektury Kubernetes je cluster. Jedná se o seskupení pracovních uzlů (anglicky worker nodes) a jednoho nebo více kontrolních prvků s názvem řídící uzly (anglicky master nodes). Kontrolní prvky obsahují API, plánovač pro přiřazovaní práce uzlům a uchovávají si stav v perzistentním úložišti. Pracovní uzly slouží k běhu procesů.

Základem je aplikace, která se zabalí a předá do clusteru. O chod se starají řídící uzly, které plánují operace, provádí monitorování, reagují na události a další věci. Pracovní uzly jsou místo, kde se spouští procesy a provádí cílové operace. Každý pracovní uzel je propojen s přiřazeným řídícím uzlem.

Spuštění aplikace v Kubernetes clusteru může probíhat například takto:

1. Vytvoření aplikace s pomocí nezávislých mikroslužeb.
2. Zabalení každé mikroslužby do vlastního kontejneru.
3. Zabalení každého vytvořeného kontejneru do vlastního podu.
4. Nasazení podů do clusteru pomocí kontrolérů vyšší úrovně jako například: Deployments, DeamonSets, StatefulSets, CronJobs apod.

Kubernetes spravuje aplikace deklarativně. To znamená, že uživatel pomocí YAML souborů vytvoří vzor toho, jak by měla aplikace vypadat a fungovat a tento popis předá do Kubernetes. Kubernetes obdrží popis systému a ví, jak má vypadat. Na základě tohoto popisu Kubernetes provádí operace a snaží se, aby systém odpovídal popisu.

[16] Legacy aplikace jsou takové, které jsou zastaralé a nemají už podporu. Přestože aplikace stále funguje, může způsobovat problémy s kompatibilitou.

2.4.2.1 Řídící uzly

Řídící uzel (anglicky master node) je kolekce systémových služeb. Nejjednodušší nastavení lze implementovat pomocí jednoho řídícího uzlu. To je ale vhodné pouze při vývoji nebo testování. V produkčním prostředí je většinou nutné využít více řídících uzlů. Tomuto se říká prostředí s vysokou dostupností (anglicky high aviability, zkratka HA). Většina cloudových poskytovatelů nabízí implementaci HA jako součást jejich Kubernetes platform. Jsou to například Azure Kubernetes Services (AKS), AWS Elastic Kubernetes Service (EKS) a Google Kubernetes Engine (GKE). Obecně se dá říci, že je dobře míti 3 až 5 replik řídícího uzlu.

Je také důrazně doporučeno nespouštět na řídících uzlech žádné uživatelské procesy. Řídící uzly se pak lépe mohou soustředit na svou práci. Na obrázku 2.4 je vidět architektura řídícího uzlu.

API server je hlavním základem systému Kubernetes. Veškerá komunikace mezi všemi komponenty prochází přes API server. Jedná se o RESTful API, přes které se nahrála konfigurace (požadovaný stav) v YAML souboru. Tato konfigurace obsahuje například určení Docker image, se kterým se bude pracovat, konfiguraci portů, počet replik podů a podobně.

Všechny požadavky, které přijdou na API Server, jsou autentizovány a autorizovány, poté je validován obsah YAML souborů, daná konfigurace je uložena do úložiště clusteru a nasazena na cluster.

Úložiště clusteru je jediná stavová část řídícího uzlu, která trvale ukládá kompletní konfiguraci a stav clusteru. Tvoří tedy nedílnou součást clusteru. Pokud nebude existovat cluster úložiště, nemůže existovat ani samotný cluster.

Úložiště clusteru je založeno na etcd[17], což je distribuovaná a silně konzistentní databáze (preferuje konzistenci oproti dostupnosti). Protože se jedná o jediné úložiště a zdroj informací pro cluster, je vhodné udržovat 3 až 5 replik a také zajistit způsob zotavení v případě chyb.

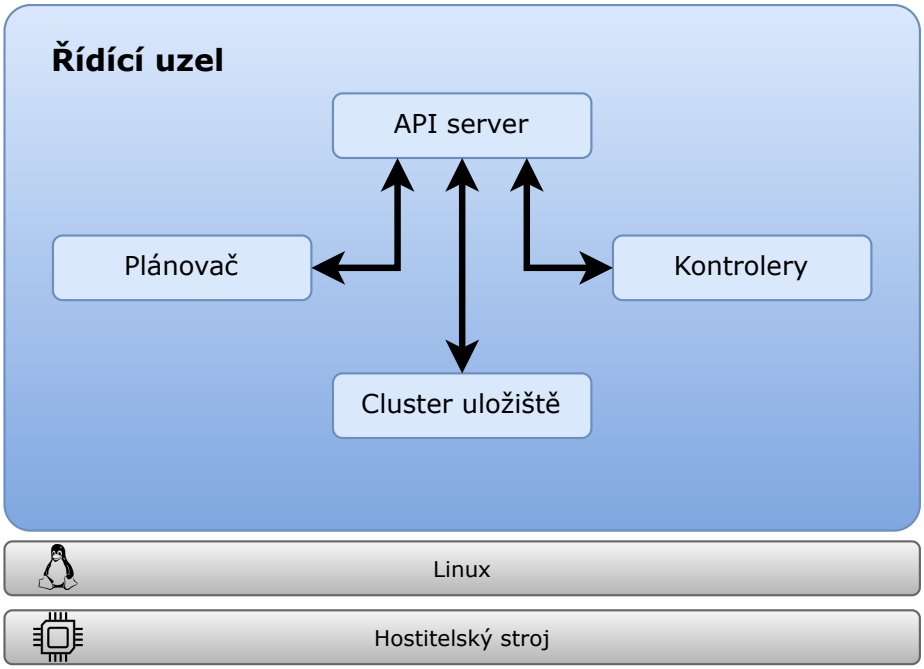
Správce kontroléru (anglicky controller manager) implementuje veškeré smyčky operací na pozadí, které monitorují cluster a reagují na události. Jedná se v podstatě o kontrolér kontrolérů. Mezi podřízené kontroléry patří: kontrolér uzlů, kontrolér replik a kontrolér koncových bodů. Tyto kontroléry běží ve smyčce, neustále sledují změny na API a provádí:

- Získání požadovaného stavu,
- zjištění aktuálního stavu,
- určení rozdílů,
- srovnání rozdílů.

Každý z těchto kontrolérů má jasně definovanou malou oblast, o kterou se stará a nic dalšího ho nezajímá. Díky tomu nevznikají žádné komplikace, každý kontrolér ví přesně, o co se má starat a je možné systém distribuovat.

Plánovač sleduje zadaní nové práce na API serveru a přiřazuje je pracovním uzlům. Jeho práce obsahuje komplexní logiku, která dokáže vyfiltrovat uzly, které nejsou schopny splnění úkolu a ohodnotit ty, které jsou pro úkol vhodné. Uzel s nejvyšším ohodnocením dostane přiřazený nový úkol. Hodnocení uzlů se skládá z mnoha věcí. Nejprve se při vyfiltrovaní nevhodných uzlů kontroluje, zda má daný uzel volný požadovaný síťový port, zda

[17] https://etcd.io/



Obrázek 2.4: **Řídící uzel.** Na obrázku je vidět architektura řídícího uzlu a komunikace jeho dílčích částí pomoci API serveru.

má dostatek zdrojů, zda existují nějaká pravidla vynucující konkrétní uzel apod. Poté následuje výběr nejvhodnějšího kandidáta ze všech vyhovujících. Hodnotí se, zda už uzel má požadovaný image, kolik má volných zdrojů, kolik úkolů je již uzlu přiřazeno apod.

Pokud plánovač nenalezne žádný uzel, který by mohl úkol vykonat, pak je úkol označen jako čekající. Plánovač také není zodpovědný za běh úkolů, pouze vybírá vhodné uzly.

Cloudový správce kontroléru – pokud cluster běží na nějaké veřejné cloudové platformě, pak je spuštěn i cloudový správce kontroléru. Jeho úkolem je spravovat integraci základních cloudových technologií a služeb, stejně tak jako instance, low-balancer a úložiště.

2.4.2.2 Pracovní uzly

V pracovních uzlech se provádí veškerá požadovaná práce. Na obrázku 2.5 je vidět jejich architektura. Jsou jednodušší než řídící uzly, skládají se z komponent – **kubelet, Container Runtime (CRI) a kube-proxy**. Tyto uzly provádí tři základní věci:

- Sledují API serveru, zda-li není nový požadavek na práci.
- Vykonávají nové požadavky na práci.
- Poskytují zpětnou vazbu řídícímu uzlu.

Kubelet je základem každého pracovního uzlu. Je hlavním Kubernetes agentem a je spuštěn na každém uzlu v clusteru. Po vytvoření nového pracovního uzlu instalační proces na uzel nainstaluje kubelet. Kubelet poté zodpovídá za registraci pracovního uzlu v clusteru. Registrace efektivně sdružuje CPU, paměť a úložiště uzlu do shromaždiště v clusteru.

Obrázek 2.5: **Pracovní uzel.** Na obrázku je vidět architektura pracovního uzlu a jeho jednotlivé části.

Jedním ze základních úkolů, které kubelet provádí, je sledovat API server a hlídat nové požadavky na práci. Vždy, když je dostupný nový požadavek, tak ho vykoná a postará se o poskytnutí zpětné vazby řídícímu uzlu.

Pokud nastane problém při vykonávání požadavku, pak to kubelet oznámí řídícímu uzlu. Už ale není zodpovědný za nalezení dalšího pracovního uzlu, který by tento požadavek mohl vykonat, o to se stará řídicí uzel.

Container runtime – kubelet potřebuje nějaký „container runtime“ k tomu, aby mohl provádět operace s kontejnery – například stahovat images, nebo spouštět a zastavovat kontejnery. Dříve měl Kubernetes integrovanou podporu container runtime, ale dnes už využívá externí pluginy formou Container Runtime Interface (CRI). Mezi nejpopulárnější CRI patří „containerd“, který byl vyňat z Dockeru a je vyvíjen jako samostatný produkt.

Kube-proxy je spuštěno na každém pracovním uzlu a stará se o lokální síť v rámci clusteru. Má například za úkol postarat se, aby každý uzel obdržel vlastní unikátní IP adresu. Také implementuje IP tabulky nebo IPVS pravidla pro správu směrování a vyvážení vytížení sítě mezi pody.

2.4.2.3 Pod

V kontextu systému Kubernetes není nejmenší jednotkou kontejner, ale pod. Základní stavební jednotkou je tedy právě pod, který obsahuje většinou právě jeden kontejner, nad kterým pod vytváří abstrakci. Abstrakce tvořená v rámci podu je užitečná například při přiřazování portů, na kterých aplikace běží. Při práci na jednom stroji není příliš složité zajistit, aby se dva kontejnery nenamapovaly na stejný port a nevznikl tak konflikt. Při distribuci práce mezi více strojů a použití stovek kontejnerů už tento úkol není tak jednoduchý. Proto pody tvoří abstrakci nad kontejnery, přičemž každý pod dostane automaticky

svou vlastní IP adresu. Jak bylo řečeno na začátku, tak v rámci jednoho podu většinou běží pouze jeden kontejner. V některých případech ale pod obsahuje více kontejnerů. Děje se tak v případech, kdy aplikace běžící v hlavním kontejneru potřebuje nějakou pomocnou aplikaci, kterou může být například při distribuci databáze synchronizátor, při použití autentizace a tak dále. Abstrakce v každém podu také vytváří síťový jmenný prostor. Pokud tedy dvě aplikace v rámci jednoho podu potřebují komunikovat, pak spolu komunikují v rámci lokální sítě pomocí příslušných portů (například localhost:8080).

2.4.2.4 Deployment

Deployment kontrolér je nástroj pro nasazení podů. Aplikační kód je možné zabalit do kontejneru a poté vytvořit a nasadit bez nutnosti vytvářet deployment, nicméně nasazení pomocí deployment přináší nezanedbatelné výhody, mezi které patří:

- **Self-healing** – tato funkcionalita je z hlediska životnosti podů velmi důležitá. Jak už bylo několikrát v této práci zmíněno, pody často zanikají a vznikají místo nich pody nové, tato vlastnost se označuje jako efemérní. A přesně z tohoto důvodu existuje deployment, který umí automaticky obnovit zaniklé pody. Deployment však nespravuje pody přímo, ale používá k tomu „replicaSet“, který vytvoří na pozadí. Uživatel s replicaSet přímo neinteraguje, ale deployment se stará o jeho konfiguraci dle konfigurace, která mu byla předána. Proces poté funguje pomocí klasického „desired state“, což je uživatelsky definovaný požadovaný stav a Kubernetes společně s jeho komponenty se stará o dosažení onoho požadovaného stavu.

- **Scalability** – škálovatelnost je další velice důležitá funkcionalita, kterou deployment společně s replicaSet poskytuje. Po každý samostatný pod je nutné vytvořit vlastní deployment s jeho konfigurací. To neplatí pouze v případě škálování, kdy se v konfiguraci deployment nastaví počet replik, které od daného podu chceme a Kubernetes se poté dynamicky stará o jejich vytvoření a zánik v případě zvýšení, či snížení pracovního nátlaku na daný pod.

- **Rolling updates a rollbacks** jsou další výhodou, kterou deployment poskytuje. Jedná se o plynulý přechod na novější verzi podu (aplikace) a případně i plynulý přechod na předchozí verzi. Pro dosažení plynulého přechodu není třeba implementovat žádnou další konfiguraci, deployment se o vše postará sám. Stačí nasadit novou verzi deployment a systém vytvoří nový deployment společně s replicaSet. Poté začne nahrazovat pod a jeho repliky jeden po druhém. V okamžiku, kdy nasadí v aktualizovaném deployment nový pod, postará se o zánik jeho starší verze. Analogicky se tento proces provádí pro přechod na původní verzi, pouze se místo novější verze nasadí starší a aktuální verze zanikne.

2.4.2.5 StatefulSet

Komponenta statefulSet je podobná deployment popsaném v kapitole 2.4.2.4. Obě tyto komponenty slouží k nasazování podů vytvořených z images (kapitola 2.3.1) a ke správě jejich škálování. Zásadní rozdíl mezi nimi je, že deployment spravuje pouze bezstavové (anglicky stateless) aplikace, tedy takové, které si neuchovávají žádný vnitřní stav. StatefulSet přináší možnost spravovat stavové (anglicky stateful) aplikace, tedy aplikace, které vnitřní stav uchovávají. Jako stavovou aplikací si je možné představit například databázi.

Při použití deployment pro správu a škálování stavových aplikací nastává problém s uložením těchto dat a jejich persistencí. Deployment totiž nijak nerozlišuje jednotlivé repliky podů (aplikace) a tedy k nim neposkytuje žádný systematický přístup. V případě zániku podu deployment automaticky vytvoří nový náhradní pod, který ale bude mít jiný identifikátor než jeho předchůdce. V tomto případě jsou pody označeny jejich názvem a pro unikátnost doplněny o náhodně vygenerovanou sekvenci znaků. Příklad označení dvojice podů tedy je „some-app-ryqbctqpi9johnk“ a „some-app-k0zweogvjo8qsmk“.

S použitím statefulSet přichází řešení nedostatků deployment při správě stavových aplikací. Repliky jednotlivých podů už nejsou unikátně označeny pomocí náhodné posloupnosti znaků, ale jejich unikátní označení se zaručuje přidáním pořadového čísla za název každé repliky, například „some-app-0“ a „some-app-1“, označení poté pokračuje analogicky čísly 2,3,4 a tak dále. Díky tomuto označení už ve vytváření replik existuje určitý systém. V případě zániku jedné repliky se místo ní vytvoří nová replika, která ponese stejné označení jako ta zaniklá.

Nejdůležitějším principem fungování u replikace stavových aplikací je správa konzistentního stavu. Pokud by systém pouze delegoval pracovní zátěž na repliky bez zajištění synchronizace dat, pak by v aplikaci vznikl chaos, protože každá replika by uchovávala a poskytovala jiná data. StatefulSet vždy obsahuje jeden hlavní pod označovaný jako master a jeho repliky označované jako slave. Jediný master má práva pro zapisování změn dat, ale číst data mohou všichni. Master při každé změně informuje všechny své repliky o změně dat a ty si uloží novou kopii. Tímto způsobem je zajištěná konzistence.

V případě vzniku a zániku replik statefulSet opět oproti deployment implementuje určitý systém. Každá nově vytvořená replika s unikátním číselným označením *n* požádá předcházející repliku s označením *n-1* o kopii jejích dat. Dalším pravidlem je, že další replika může vzniknout až je předchozí replika plně v provozu. Při zániku replik se vždy vybírá replika s největším číselným označením, tedy na konci pomyslného řetězce.

I při použití statefulSet stále platí, že veškerá data, která pody uchovávají, jsou po jejich zániku smazána, což je při orchestraci databáze problém. Z tohoto důvodu se statefulSet kombinuje s volumes, které jsou popsané v kapitole 2.3.2. Jedná se o konfiguraci podu tak, aby využíval perzistentní úložiště, které bude alokované například na pevném disku. Toto úložiště bude obsahovat duplikát dat, která pod obsahuje, ale také záznam o aktuálním stavu podu. Pokud tedy pod havaruje a bude nahrazen novým, novému podu bude přiděleno jeho perzistentní úložiště společně s jeho stavem.

Shrnutím je, že systém Kubernetes tedy poskytuje prostředky pro orchestraci stavových aplikací, jako jsou například databáze, ale konfigurace této orchestrace je velice komplexní věc. Kubernetes efektivně napomáhá zabezpečit synchronizaci dat jednotlivých replik, ale i přesto je nutné zajistit správu vzdáleného perzistentního úložiště, na kterém budou data vždy k dispozici a jeho zálohování. Způsob má jisté výhody, ale jsou případy, kdy bude dobré se tomuto raději vyhnout.

2.4.2.6 Service

Jak bylo zmíněno v kapitole 2.4.2.3, každému podu se po jeho vytvoření přidělí unikátní IP adresa. Pody často zanikají a vznikají (jsou efemérní) a při každém vzniku dostanou novou IP adresu. Z tohoto důvodu není vhodné na pod odkazovat přímo pomocí jeho IP adresy, protože bychom při každém novém vzniku podu museli upravovat konfiguraci.

Z tohoto důvodu existuje komponenta service, která tento problém řeší. Service má perzistentní IP adresu, pomocí které je možné komunikovat s podem. Service ovšem ale

neslouží pouze pro přidělení perzistentní adresy podu, ale je možné pomocí něj řešit i low-balancing, kdy je service vstupním bodem a deleguje požadavky na přidělené pody. Existuje více druhů service:

- **ClusterIP** je nejčastější a výchozí typ service. Pokud není specifikovaný typ, pak service bude právě typu clusterIP. ClusterIP neběží jako samostatný proces, ale je to abstrakce pro přístup k podu v rámci stejného clusteru. V konfiguračním YAML souboru jsou definována pravidla, která příchozí požadavky delegují na příslušné pody. V rámci service se na jednotlivé pody odkazuje pomocí tzv. „labels“, což jsou klíčové hodnoty definované při konfiguraci podu (například v rámci deployment).

- **Headless** service slouží pro komunikaci s konkrétním podem v případě, že je dynamicky vytvořeno více jeho replik. V případě service typu clusterIP je požadavek po obdržení v service náhodně přesměrován na jednu z replik příslušného podu. U stavových aplikací může být někdy třeba komunikovat s konkrétní replikou. Například pokud máme několik replik databáze, tak jedna replika bude tzv. hlavní, která bude jako jediná modifikovat hlavní zdroj dat a veškeré další repliky se k ní budou muset kvůli synchronizaci dat pravidelně připojovat. Pak je nutné spolu se service, která se bude starat o náhodnou delegaci mezi repliky použít i headless service, která bude sloužit pro komunikaci s konkrétním podem.

- **NodePort** je typ service, který je dostupný na statickém portu na každém pracovním uzlu v rámci clusteru. Což znamená, že je možné k service přistupovat i externě mimo cluster. Tento typ service ale není doporučené používat z důvodu bezpečnosti, jelikož při jeho konfiguraci umožňujeme přístup externím entitám přímo k pracovním uzlům. Service typu nodePort se používá spíše pro účely vývoje a testovaní.

- **LoadBalancer** je vylepšená alternativa typu nodePort. Také umožňuje externí přístup k service mimo cluster, ale používá k tomu loadBalancer poskytovatele cloudových služeb. Společně s loadBalancer typem jsou automaticky systémem Kubernetes vytvořeny i service typu nodePort a clusterIP, na které bude externí load balancer přesměrovávat požadavky.

2.4.2.7 Ingress

Ingress je API objekt, který slouží jako vstupní bod, dá se říct reverse-proxy server, do clusteru a je vhodným řešením pokud cluster obsahuje více services, které mají být dostupné zvenčí clusteru. Ingress také poskytuje možnosti předávání SSL certifikátu, což souvisí se zabezpečením komunikace a možností použití protokolu HTTPS. Pomocí ingress je také možno nastavit virtualizaci doménových jmen, díky čemuž není nutné přistupovat ke službě běžící v určitém podu pomocí adresy a portu, ale je možné vytvořit doménové jméno (například „example.com“). Pomocí těchto směrovacích pravidel je také možné v rámci virtualizace doménových jmen konfigurovat subdomény (například „sub.example.com“) a nebo cesty domény (například „example.com/path“). Pro použití ingress je nutné vytvořit konfigurační soubor, jako tomu bylo u service nebo deployment, ale navíc třeba definovat nějaký „ingress controller“ třetí strany, což je v podstatě implementace funkcionality ingress. Tento ingress controller poté funguje jako samostatná služba v rámci clusteru ve vlastním podu a zabezpečuje například zpracování směrovacích pravidel, správu přesměrování nebo vstupní

bod do clusteru. Oficiální ingress controller od Kubernetes se nazývá „K8s Nginx Ingress Controller“[18].

2.4.2.8 Config map

Config map je užitečným nástrojem při práci s konfigurovatelnými aplikacemi a systémy. Je obecně dobré mít konfiguraci (jako například adresu databázového serveru) zvlášť v externím souboru. Vzhledem k povaze aplikací běžících v systému Kubernetes je to ale spíše nutnost. Z aplikace je vytvořená image, na kterou je odkázáno v konfiguraci deployment a ze které se poté vytvoří kontejner a pod. Pokud bychom chtěli změnit nějakou konfiguraci, která je přímo v aplikačním kódu, tak bychom museli složitě upravovat kód aplikace, vytvářet z něj image, odkázat na novou image a tak dále. Při použití externí konfigurace, v tomto případě konkrétně config map, stačí změnit pouze jeden daný soubor. Mezi další případy, kde je vhodné použít config map, patří například konfigurace nodes, síťová a bezpečnosti politika a její pravidla, přístupové údaje, certifikáty a podobně.

2.4.2.9 Secret

Tato komponenta lehce souvisí s předchozí config map. Opět se jedná o externí konfiguraci aplikace nebo systému, kdy máme konfigurační data uložena v externím souboru a nemusíme při každé změně nahrávat novou verzi aplikace. U secret se ale cílí na něco trochu jiného. Hlavní podstatou je předávání citlivých informací, jako jsou například přístupové údaje, bezpečnější formou. Při nutnosti použití takto předaných údajů Kubernetes nečte tato data, ale pouze operuje s odkazy na tato data, která předá. Operátory v systému Kubernetes tedy nikdy přímo „nevidí“ tato citlivá data.

2.4.2.10 Minikube

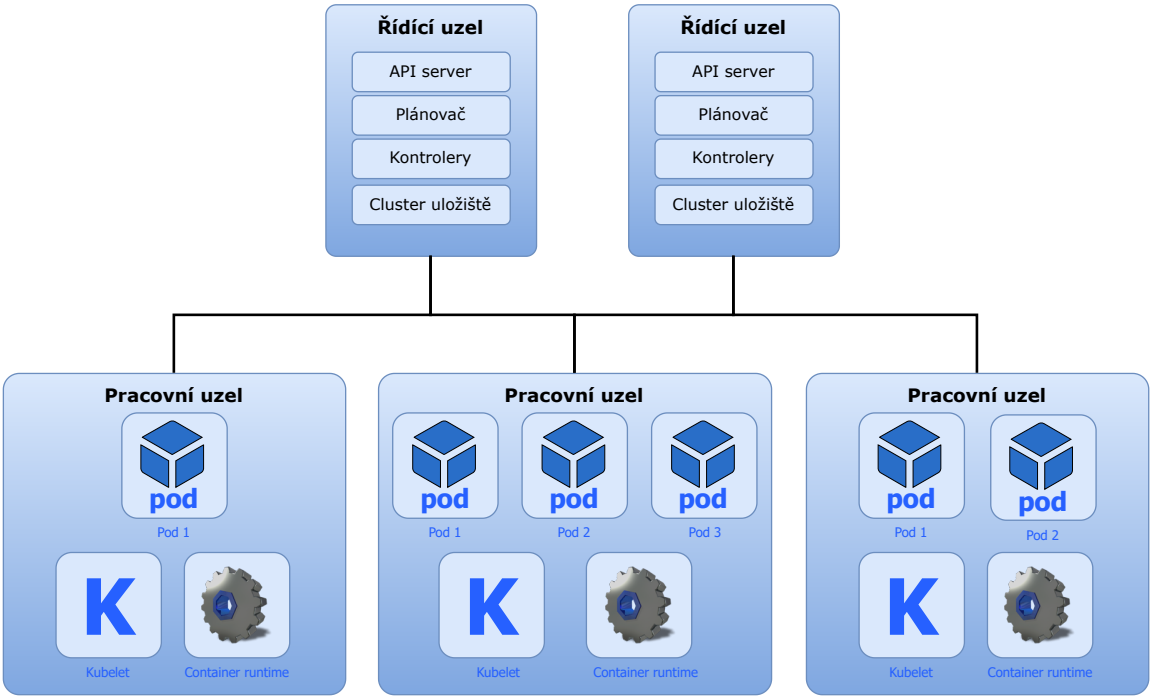
V praxi se v produkčním prostředí jeden cluster skládá z několika řídících uzlů (kapitola 2.4.2.1), které jsou v tomto clusteru nejméně dva a několika pracovních uzlů (kapitola 2.4.2.2). Tento případ je ilustrován obrázkem 2.6. Každý z těchto uzlů může běžet na samostatném virtuálním nebo fyzickém stroji. V případě vývoje, nebo testování, na lokálním prostředí při vytvoření clusteru tak, jak je výše popsáno, nastává problém s dostupnými prostředky (paměť, výkon procesoru, atd.) a od toho slouží Minikube[19]. V případě minikube veškeré řídící a pracovní procesy běží v jednom společném uzlu (anglicky node), tedy i na jednom společném zařízení. V tomto uzlu je již předinstalovaný Docker container runtime. Minikube ke spuštění uzlu využívá Virtual Box, ve kterým se onen uzel spustí v jednom clusteru.

2.4.2.11 Kubectl

Kubectl je nástroj příkazové řádky, který slouží pro komunikaci s Kubernetes clusterem a jeho konfiguraci. Jak již bylo popsáno v předchozí kapitole, minikube spouští všechny řídící procesy v jednom hlavním uzlu. Jeden z těchto řídících procesů se nazývá „API server“ a je hlavní komunikační branou uzlu. S API serverem můžeme komunikovat různými způsoby:

- **UI** – například kubernetes dashboard.

[18] https://www.nginx.com/resources/glossary/kubernetes-ingress-controller/
[19] https://minikube.sigs.k8s.io/docs/start/



Obrázek 2.6: **Architektura clusteru.** Na obrázku je architektura clusteru, který obsahuje dva řídící uzly a tři pracovní.

.

- **API** – například příkaz „curl“ nebo skript.
- **CLI** – například kubectl.

Právě použití konfigurace pomocí příkazové řádky s kombinací kubectl je nejvýkonnější varianta. Kubectl však neslouží pouze ke konfiguraci clusteru spravovaného v rámci minikube, ale je ho možné použít na jakýkoliv cluster.

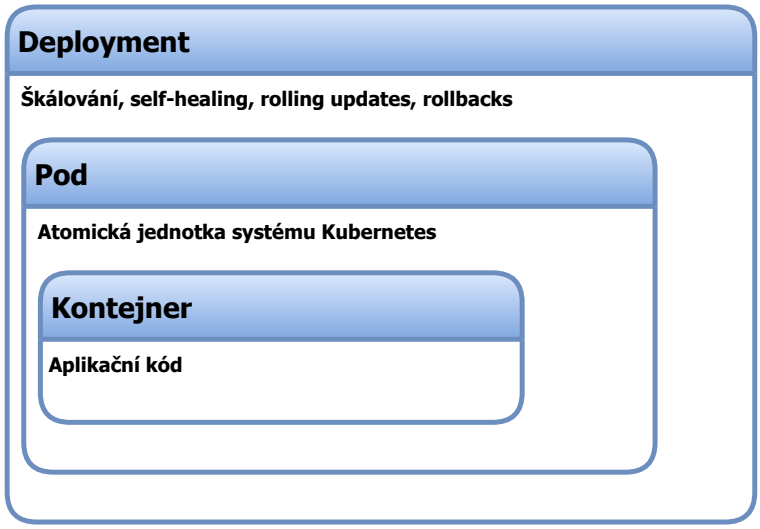
2.4.3 Příprava aplikace pro Kubernetes

Proto, aby mohly aplikace běžet v Kubernetes clusteru, je třeba provést (mimo jiné) některé základní věci. Ty obsahují:

1. Zabalení aplikace do kontejneru.
2. Zabalení výsledného kontejneru do podu.
3. Nasazení pomocí deklarativního manifest souboru.

Nejprve je samozřejmě nutné samotnou aplikaci vytvořit a naprogramovat. Poté je nutné aplikaci zabalit do kontejneru a ten nahrát do nějakého registru (2.3.1). V tuto chvíli lze aplikaci (případně aplikační službu) označit jako „kontejnerizovanou“.

Poté je třeba definovat Kubernetes pod, ve kterém poběží kontejnerizovaná aplikace. Jak bylo popsáno dříve, pod je v podstatě obal, který umožňuje kontejnerizované aplikaci běžet v Kubernetes clusteru. Nyní je třeba pod nasadit do clusteru.



Obrázek 2.7: **Struktura aplikace nahrávané do clusteru.** Na obrázku je vidět struktura aplikace dle popisu v kapitole 2.4.3.

Je možné v clusteru spustit přímo pod, ale většinou bude lepší řešení nasazení pomocí vysokoúrovňových ovladačů. Nejčastější takový ovladač je „deployment“. Rozšiřuje možnosti podu o škálovatelnost, samo-opravy a „rolling-updates“ 2.4. Deployment se definuje v YAML manifest souboru.

Na obrázku 2.7 je vidět výsledek zabalení aplikace dle popisu.

2.4.4 Deklarativní model a požadovaný stav

Deklarativní model a koncept požadovaného stavu je základním principem fungovaní Kubernetes. Deklarativní model funguje takto:

1. Deklaruje se požadovaný stav aplikace pomocí manifest souboru.
2. Pomocí POST a API serveru se deklarace nahraje.
3. Kubernetes deklaraci požadovaného stavu uloží do úložiště clusteru.
4. Kubernetes implementuje požadovaný stav clusteru.
5. Kubernetes implementuje smyčky pro kontrolu zda se aktuální stav rovná požadovanému stavu.

Výhody tohoto deklarativního způsobu spočívají v tom, že uživatel pouze „řekne“, jak má systém vypadat a veškerou další práci už odvede Kubernetes. Není třeba psát dlouhé skripty a řešit konkrétní případy, kdy havaruje část aplikace, postupy škálování a podobně. Pokud aktuální stav neodpovídá požadovanému, Kubernetes zařídí, aby odpovídal.

2.4.5 Multitenance v rámci systému Kubernetes

Multitenance uživatelských aplikací již byla popsána dříve v kapitole 2.1.3. V této práci ale je nutné rozlišit multitenance na dvou různých úrovních. První úrovní je multitenance

uživatelské aplikace, která je cílem této práce, a druhou úrovní je multitenance systému Kubernetes. Účel multitenance v rámci Kubernetes je zvýšení efektivity výsledného systému a jeho efektivnější využívání zdrojů. Při hledání odborných článků, které by se mohly zabývat stejným problémem jako tato práce (převedení aplikace do multitenantní podoby), za pomoci klíčových slov „kubernetes“ a „multitenance“ vyhledávače nabízí články pojednávající právě o druhé popsané úrovni multitenance. Druhá úroveň multitenance sice nebude v práci použita, ale právě z důvodu jejího výskytu ve vyhledávání je tu zmíněna.

2.4.5.1 Typ multitenance

Rozlišujeme dva základní typy multitenance – „Hard“ multitenanci a „Soft“ multitenanci [16].

Soft multitenance je typ multitenance, která nepožaduje implementování striktní izolace jednotlivých uživatelů, pracovní zátěže, nebo aplikací. Je to tedy vhodné řešení pro tenanty (uživatele), kteří si mohou v rámci mezí důvěřovat a nebudou si vědomě škodit. Může se tedy jednat například o spolupracovníky v jedné společnosti, nebo společné firmy. Izolace uživatelů je zde cílena spíše na prevenci neúmyslných nehod, ale nezabraňuje cíleným útokům mezi tenanty.

Hard multitenance na rozdíl od Soft multitenance vyžaduje striktnější izolaci, čímž předchází negativním dopadům nekalého jednání jednotlivých tenantů. Tento typ multitenance je vhodné použít do prostředí s citlivými daty, nebo do prostředí s možností ohrožení útokem. Pokud bychom brali v potaz pouze tyto dva způsoby multitenance, pak z důvodu bezpečnosti pro použití v praxi dává smysl použití právě tohoto způsobu.

2.4.5.2 Soft versus Hard multitenance

Ačkoli tyto dva pojmy mohou znít jako dva implementační protipóly v přístupu k multitenanci, pravdou je, že to jsou spíše takové dva krajní extrémy a výsledná implementace většinou tvoří kompromis mezi nimi. Teoreticky je hard multitenance lepší z pohledu variability použití – není nutné řešit důvěrnost tenantů, ale prakticky je jednodušší implementovat soft multitenanci. V této práci bude pro dosažení multitenance použit Kubernetes, který musí mít vždy alespoň nějaké části pro všechny tenanty společné, a proto s jeho pomocí teoreticky nelze implementovat stoprocentní Hard multitenanci, ale pouze se jí přiblížit.

2.5 Současná řešení

Problematika, kterou tato práce řeší, tedy modifikaci libovolné jednotenantní aplikace na její multitenantní verzi, není dosud zdokumentovaná, případně není dohledatelná ve veřejně přístupných zdrojích. V této kapitole nejsou tedy popsána řešení, která by vedla přímo k požadovanému výsledku této práce, ale postupy a nástroje, které s touto prací souvisí a jejich implementace by mohla tvořit část výsledného systému. Při hledání podkladů pro tuto práci, nebo podobných řešení, spadají výsledky hledání do dvou kategorií.

První kategorií je multitenance v rámci systému Kubernetes, což zprvu zní jako řešení problému této práce a teoreticky by mohlo být i alternativní metodou, která sice nebude splňovat podstatné aspekty multitenance, ale bude fungovat. Pokud bychom se pokusili dle dostupných návodů touto metodou vyřešit multitenanci původní aplikace, ve výsledku bude dosaženo spíše efektivního spuštění více instancí daného systému na jednom hostitelském stroji, ke kterému budou uživatelé přistupovat vzdáleně. Existují ale případy, kdy je po

úpravách tento způsob možné využít jako součást dosažení výsledku této práce. Více o této alternativě v následující kapitole 2.5.1.

Druhou kategorií výsledků hledání jsou články o rozdílech jednotenantních a multitenantních aplikací a s nimi související různé návody a příručky pro vytvoření multitenantní aplikace. Příručky by se opět daly považovat za krajní alternativu této práce, kdy se celá aplikace kompletně naprogramuje znovu s tím, že se zrecyklují použitelné části kódu a po případných úpravách se zasadí do nové multitenantní kostry této aplikace. Takovéto kompletní opětovné vytvoření celé aplikace nebo systému je „nejčistší“ variantou jak multitenance dosáhnout, ale zároveň také nejpracnější a z časových a finančních důvodů není vždy reálná. Čímž se dostáváme opět k základnímu smyslu práce, kterým je nabídnout postup pro usnadnění tohoto převodu, kdy nebude dosaženo přesné podoby multitenance, ale výsledek se k ní bude blížit za cenu ušetření potřebných zdrojů pro převod.

2.5.1 Multitenance implementovaná pomocí Kubernetes

Pro dosažení multitenance v rámci Kubernetes existuje spoustu návodů a dokonce i specializované nástroje. Použití multitenance v Kubernetes většinou slouží pro vývoj aplikací, kde více vývojářských týmů sdílí jeden cluster. Nástroje a postupy implementující multitenanci v Kubernetes, je možné použít bez úprav jako alternativní řešení cíle této práce, avšak dosažené řešení by se od multitenantního systému principiálně lišilo. Pokud bychom nebrali v potaz princip fungování aplikace na pozadí, tak by z uživatelského hlediska bylo možné dosáhnout stejného řešení.

Multitenance v rámci Kubernetes znamená rozdělení jednoho clusteru na více izolovaných částí, ve kterých mohou běžet aplikace odděleně. Z technického hlediska tato izolace ale není dokonalá a je možné ji použít pouze ve specifických případech. Více informací o výhodách a nevýhodách použití sdíleného clusteru se nachází v kapitole 2.5.2. Jak již bylo zmíněno dříve, použít sdílený cluster pro nasazení kompletní kopie původní aplikace není ideálním řešením, ale pokud by do něj byla nasazena pouze uživatelsky (tenantně) specifická část původního systému, pak jeho použití dává smysl i v řešení této práce.

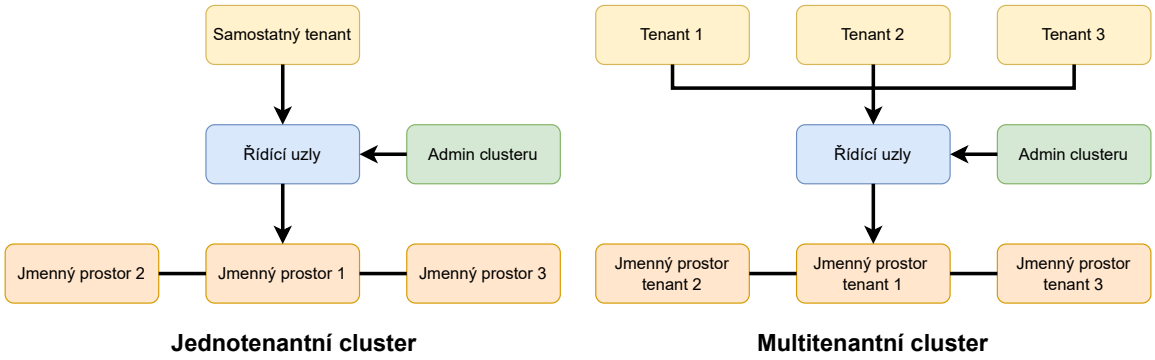
Pro vytvoření multitenance je možné využít jmenné prostory, které umožňují vytvoření oddělených prostředí pro více aplikací, či uživatelů v rámci jednoho clusteru. Jejich použití ale přináší určitá omezení. Spousta aplikací potřebuje ke svému běhu zdroje v rozsahu celého clusteru (anglicky „cluster scoper resources“, jako například uzly, role clusteru, perzistentní úložiště a podobně. Při použití těchto zdrojů aplikace vybočuje ze svého jmenného prostoru a nastává problém s izolací od ostatních aplikací.

2.5.2 Sdílený cluster vesrsus více menších clusterů

Alternativou implementace multitenantního clusteru je vytvoření více samostatných jednotenantních clusterů [19]. Tato řešení jsou si v principu fungování výsledného orchestrovaného systému velice podobná, ale každé z těchto řešení má své výhody a nevýhody [10, 11]. Z některých těchto nevýhod se mohou při určitých podmínkách stát omezení nebo i překážky bránící v použití tohoto přístupu. Na otázku, který přístup z těchto dvou je lepší a který použít, tedy neexistuje jednoznačná odpověď, ale použití záleží na okolnostech a prioritách.

Správná izolace a dostupnost jsou klíčové vlastnosti pro realizaci orchestrace multitenantního systému. Je nepřípustné, aby procesy jednoho tenantu, který by měl být izolovaný od toho druhého, mohly zasahovat do cizího prostoru. Toto chování by mohlo zapříčinit únik tajných dat nebo jiné napadení systému. Dalším důležitým aspektem je konzistence výkonu. V tomto případě nehrozí únik tajných dat, ale procesy jedné aplikace mohou zabrat výkon

potřebný k fungování druhé aplikace a ta se může stát nedostupnou. V systému Kubernetes je možné za použití správné konfigurace infrastruktury a cluster operátorů dosáhnout virtuální izolace, která tyto nežádoucí faktory eliminuje, ale existují určitá známá omezení, která limitují použití tohoto Kubernetes soft-tenancy (kapitola 2.4.5.1) modelu. Tato omezení jsou ve zbytku kapitoly rozebrána a po jejich analýze je možné určit, zda je lepší použít multitenantní cluster a nebo více jednotenantních. Na obrázku 2.8 je zobrazena jednoduchá ilustrace jak architektury jednotenantního clusteru, tak i popsaného sdílení řídících prostředků v multitenantním clusteru.



Obrázek 2.8: **Jednotenantní versus multitenantní cluster.** Na obrázku je zobrazena architektura dvou typů clusterů, kdy u multitenantní verze všechny tenanty sdílí stejně řídící prostředky clusteru.

2.5.2.1 Izolace

Nejvyšší úroveň izolace je dosažena systémovou izolací, kdy je každý cluster naprosto nezávislý na ostatních týmech, aplikacích a umístění. Jenže s tímto řešením přichází problém s režií všech samostatných clusterů, která při větším počtu clusterů již dosahuje mnohem větších, než zanedbatelných, hodnot. Na druhou stranu použití žádné nebo částečné izolace je jednodušší na správu, ale je to na úkor granularity izolace a kontroly nad systémy.

V rámci jednoho sdíleného clusteru je možné docílit logické izolace pomocí jmenných prostorů. Systém Kubernetes k tomuto účelu poskytuje několik typů nastavení pravidel jmenných prostorů (anglicky namespace policies):

- RBAC: izolace a oprávnění,
- LimitRange: omezení pro procesor a paměti,
- ResourceQuota: kvóty paměti a procesoru jednotlivých jmenných prostorů,
- NetworkPolicy: pravidla pro síťovou komunikaci,
- PodSecurityPolicy: podrobná autorizace vytváření a aktualizací podů.

Veškerá tato pravidla slouží k izolaci jmenných prostorů a jejich vzájemné ochraně, ale v nejlepším případě poskytují pouze soft multitenanci. Nejen, že veškeré výpočty a úkony všech tenantů sdílí stejnou infrastrukturu (výpočetní uzly, síťovou infrastrukturu a úložiště), ale i systémové komponenty Kubernetes, jako jsou například api-server, kube-proxy nebo kubelet (popsané v kapitole 2.4.2.2), jsou sdíleny napříč tenanty. Díky snaze společností

spravujících jak Kubernetes, tak i nástroje pro jeho multitenanci jsou zjištěné chyby týkající se této problematiky rychle opraveny, ale je více než pravděpodobné, že se vyskytnou nové. Pokud je tedy v požadovaném systému potřeba striktní izolace, pak je dobré použít spíše variantu více menších clusterů, kdy každému tenantu bude patřit jeden cluster.

2.5.2.2 Dostupnost

Kubernetes jakožto kontejnerový orchestrační nástroj velice dobře zvládá správu aplikací týkající se jejich škálování, vyvažování pracovní zátěže (low-balancing) a celkově životní cyklus aplikace, nicméně pokud dojde k chybě platformy, na které systém běží, ať už kvůli špatné konfiguraci, nepovedené aktualizaci nebo kritické chybě, veškeré operace vně clusteru budou pozastaveny. Přičemž takto způsobené chyby nemusí vždy znamenat pochybení správce clusteru, ale chyba může být i u poskytovatele cloudových služeb. Z tohoto důvodu je lepší použít více menších clusterů, případě při použití sdíleného multitenantního clusteru nepoužívat jen jeden. V ideálním případě by taková distribuce systému do více clusterů měla obsahovat i rozdělení do více fyzických oblastí, v extrémním případě i do více cloudových služeb. Služba pak při výpadku jednoho cloudu nebo fyzické lokace bude stále dostupná a dosáhne se tím lepší dostupnosti z hlediska přístupového času.

2.5.2.3 Složitost provozu

Dle zatím shrnutých informací jsou na tom malé jednotlivé clustery oproti velkým multitenantním mnohem lépe. Tento způsob ale nepřináší pouze výhody, ale nese s sebou i určité komplikace. Například s narůstajícím počtem clusterů také narůstá cena jejich údržby a provozu.

Dokonce i jeden samotný cluster může být složitý na údržbu. Tyto složitosti můžeme rozdělit do několika kategorií:

- Správa životního cyklu – aktualizace, záloha, obnova, poskytování systému.
- Integrování základních komponent jako je úložiště, nastavení sítě, bezpečnosti a podobně.
- Zabezpečení přístupu ke clusterům – uživatelská autentizace a autorizace nebo síťová politika.
- Přihlašování a pozorování – centralizované přihlašování, sledování chování aplikací a infrastruktury.

Čím více clusterů je v rámci tohoto systému spravováno, tím více náročnost na jejich údržbu roste. Nastává problém s konzistencí aktualizací politik, bezpečnostních pravidel a podobně pro podobné clustery. Z tohoto důvodu je nutné v rámci správy implementovat nějakou automatizaci, která tyto problémy bude řešit.

Nejsou to jen náklady na správu, co s rostoucím počtem clusterů rostou také. Žádný cluster pravděpodobně nebude zcela využívat alokovaný CPU a paměť. Z důvodu zvládnutí nečekané nárazové zátěže má cluster vždy alokováno o něco více CPU a paměti než opravdu potřebuje. Množství takto alokovaných prostředků je při jednotkách clusterů zanedbatelné, ale při větším množství už může způsobovat problémy. Zde přichází na řadu výhoda multitenantního clusteru, který také plýtvá nějakými alokovanými prostředky, ale je na tom mnohem lépe. Navíc je mnohem jednodušší nastavit správné škálování u několika sdílených clusterů než u velkého množství malých.

2.5.2.4 Shrnutí

Jak použití více menších clusterů, tak i použití multitenantního clusteru má své výhody a nevýhody. S množstvím clusterů rostou nároky na výkon a na udržovatelnost a při multitenantním prostředí v rámci clusteru není možné dosáhnout takové úrovně izolace jako při samostatných clusterech. Z těchto důvodů je nutné dobře analyzovat požadavky na výsledný systém a dle toho zvolit správné řešení.

V praxi se mnohem častěji používá způsob vytvoření více malých clusterů právě z důvodu lepší izolace jednotlivých prostředí, což bývá při poskytování aplikací priorita. Při vývoji, kde se snažíme oddělit například vývojové a produkční prostředí jednoho týmu nebo umožnit používat cluster týmům jedné firmy, nepředpokládáme komplikace týkající se bezpečnosti a vzájemné narušování prostorů tenantů.

2.5.3 Nástroje pro vytvoření multitenantního clusteru

Správné vytvoření multitenantního clusteru není triviální záležitost a nesprávnou konfigurací je možné způsobit kritické bezpečnostní chyby. Jak již bylo zmíněno v kapitole pojednávající o výhodách a nevýhodách multitenantních clusterů; jejich použití je vhodné spíše ve vývojovém prostředí, kde nemůže dojít ke ztrátě citlivých dat z důvodů napadení tenantem sdílejícím stejný cluster. Existují ale případy, kdy jsou bezpečnostní rizika menší a jeho použití je proto vhodné. Poslední dobou vznikají na správu multitenantních clusterů specializované nástroje. Použití takového nástroje vytváří novou úroveň abstrakce a efektivním způsobem ulehčuje vytvoření a správu multitenantních clusterů.

2.5.3.1 Loft vClusters

Společnost Loft[20] vyvíjí nástroj sloužící k virtualizaci clusterů, což umožňuje vytvoření oddělených prostorů v rámci jednoho Kubernetes clusteru. Na podzim roku 2021 došlo k vydání druhé verze tohoto nástroje [18]. Společnost Loft tvrdí, že jejich cílem je zjednodušit vývoj týmům použitím několika větších sdílených clusterů místo nutnosti spravovat velké množství malých individuálních Kubernetes clusterů.

Použití jmenných prostorů pro sdílení clusteru není ideálním řešením, a proto nástroj přináší novou úroveň abstrakce, díky které je možné v rámci jednoho clusteru vytvořit další virtuální cluster. To je možné jednoduše pomocí příkazové řádky, uživatelského rozhraní nebo pomocí Kubernetes API. Dle statistik společnosti byl image umístěný na serveru serveru Docker Hub[21], sloužící pro vytváření virtuálních clusterů, od jara minulého roku do loňského podzimu stažen 120 000 krát.

[20] https://loft.sh/
[21] https://hub.docker.com/

2.5.3.2 Capsule

Capsule[22] je dalším nástrojem soustředícím se na vytvoření a správu multitenantního clusteru bez nutnosti ruční konfigurace a použití jmenných prostorů. Tentokrát se jedná o open-source projekt. Hlavním přínosem Capsule je větší volnost tenantů například z hlediska možnosti vytváření a správy vlastních jmenných prostorů, které jsou izolované od ostatních tenantů v clusteru. Capsule je relativně nový nástroj, který vyšel roku 2021, takže o něm neexistuje moc uživatelských článků popisujících jeho chování a zkušenosti s používáním. Dle dostupných informací je nástroj založený na Kubernetes, díky čemuž nemění chování, a tenanti mohou využívat funkce clusteru téměř identicky, jako kdyby byl celý cluster jen pro ně.

[22] https://capsule.clastix.io/

Kapitola 3

Návrh řešení

Tato kapitola popisuje návrhy pro dosažení požadovaného výstupu práce – vytvoření iluze multitenance aplikace, která je původně jednotenantní a zajištění její orchestraci pomocí systému Kubernetes.

Problematika převodu aplikace do její multitenantní podoby je velice komplexní záležitost a jeho průběh a výsledek ovlivňuje velké množství faktorů. Při průzkumu současných řešení, dle kterých by bylo možné implementovat tento převod, byly nalezeny pouze návody související s cloudovými službami, které tyto převody implementují vlastními nástroji a aplikace je poté závislá na platformě.

Cílem práce není popsat převod vázaný na platformu, na které aplikace běží, ale vytvořit obecný návod pro převod aplikace bez ohledu na povahu zdrojového systému a jeho okolí. Výsledkem této práce tedy nebude jeden návrh, který bude možné pouze vzít a pomocí něj striktně provést převod, ale spíše soupis potřebných znalostí, okolností ovlivňujících převod a alternativ pro kombinaci různých faktorů, jejichž součástí je jednoduchá obecná demonstrace implementace, kterou vývojář s potřebnými znalostmi využije jako stavební kámen pro výsledný převedený systém.

Z těchto důvodů jak kapitola Návrh řešení, tak i kapitola Implementace a vyhodnocení obsahují několik alternativních řešení, ze kterých je nutné si vhodné řešení zvolit a přizpůsobit ho dle specifických potřeb.

3.1 Modelový systém

Pro navržení konkrétní vhodné architektury je nejprve nutné analyzovat současný systém, v případě této práce zvolit a definovat modelový systém, protože návrh architektury a její komponenty budou částečně závislé na podobě systému, případně modelu. Při návrhu základního konceptu postačí smyšlený systém bez větších detailů, návrh pak bude obecný a bude ho možné použít jako základ pro komplexnější systém. Vhodný modelový systém by měl pracovat s databází pro ukládání dat, obsahovat funkcionalitu, ze které se dá vytvořit bezstavová mikro-služba, která bude sdílená mezi tenanty, a mělo by mít smysl ho rozšířit o multitenantní verzi. Typicky by pak multitenantní systém měl obsahovat možnost specifické konfigurace systému dle požadavků tenanta.

Výhodou tohoto modelového systému je jednodušší demonstrace pro základní pochopení a uchopení problematiky. Reálné systémy, na které cílí tato práce, jsou však mnohem složitější a jejich převod je mnohem náročnější. Nemá příliš velký smysl se zabývat konkrétním převodem nějakého komplexního systému, protože by toto řešení, stejně jako každé jiné,

bylo specificky navržené a pro každý systém kromě základního principu odlišné. Jak již bylo zmíněno, práce tedy obsahuje návrh a demonstraci převodu jednoduššího systému, který bude možné použít jako základ a součástí práce je zohlednění různých možných aspektů převáděného systému a návrh příhodných řešení těchto situací, kterými je možné doplnit základní návrh.

Modelový systém tedy bude naprosto triviální aplikace pro výpočet průměrné hodnoty zadané výšky jednotlivých uživatelů. Instanci této aplikace je možné na základě požadavku externí firmy nasadit na její server, kam se budou jednotliví zaměstnanci připojovat, a zadáním pracovního emailu a výšky do formuláře odešlou požadavek s daty na zpracování. Takovýto požadavek přijme pomocí API rozhraní bezstavová mikro-služba (tato služba může, ale nemusí být na stejném serveru jako uživatelské rozhraní), která se připojí na databázi umístěnou na libovolném serveru. Mikro-služba validuje obdržený požadavek, v případě validních dat zapíše nové hodnoty do systému a vrátí vypočtenou průměrnou výšku, která bude zobrazena zaměstnanci. Další konkrétní detaily implementace není nutné specifikovat. Systém je modelový a slouží k demonstračním účelům, takže je možné zanedbat důvody, proč by firma tuto aplikaci požadovala a používala.

Modelová aplikace tedy obsahuje ukládání a práci s daty na vzdáleném serveru a bezstavovou mikro-službu, která bude sloužit jako vyčleněný sdílený aplikační modul mezi tenanty. Dále by demonstrační aplikace měla implementovat konfigurovatelnou část aplikace (bude obsahovat unikátní nastavení pro každého tenanta), kterou pro zjednodušení může být aplikační uživatelské rozhraní s formulářem pro zadávání dat.

3.2 Rozhodovací parametry převodu

Jak již bylo řešeno v kapitole 3.1, pro základní uchopení principu a demonstrace průběhu převodu je vhodné použít jednoduchou aplikaci. U takovéto aplikace není nutné analyzovat příliš dopodrobna vlastnosti popsané v této kapitole, u složitějších systému to ale může rozhodnout o úrovni a způsobu jeho převodu do multitenantní podoby.

Realita životního cyklu systému je taková, že vývoj často nekončí jeho vytvořením dle původních požadavků, ale dále se rozšiřuje i po jeho prvním nasazení či předání zákazníkovi. Systém se přizpůsobuje konkrétním případům užití, nebo se přidávají nové funkcionality, které v původním návrhu vůbec nebyly obsaženy. Do kódu aplikace zasahují lidé, kteří s jejím prvotním vývojem neměli nic společného, kvůli čemuž vznikají odlišnosti implementací v různých částech aplikace, konkrétně například způsob propojení jednotlivých modulů aplikace a jejich komunikace. Aplikační kód nabývá na objemu a ztrácí na přehlednosti, udržovatelnosti a jednoduchosti, čímž vznikají mnohé problémy týkající se převodu systému na multitenantní verzi.

U reálného systému tedy bude nutno zohlednit spoustu aspektů, které ovlivní výslednou podobu multitenantní verze a jejího způsobu implementace. Variabilita výsledku z velké části závisí také na nefunkcionálních faktorech, jako jsou například:

- **Časová dotace poskytnutá pro převod** – některé systémy se do multitenantní podoby začnou převádět, až když je to opravdu nutné. Při složitějších systémech převod zabere delší dobu a bude tak nutné zvolit postupný převod, kdy se aplikace po malých částech začne přesouvat do společného prostředí a buď bude zákazníky používána původní jednotenantní aplikace, nebo s každou novou převedenou částí nová verze základu multitenantní aplikace.

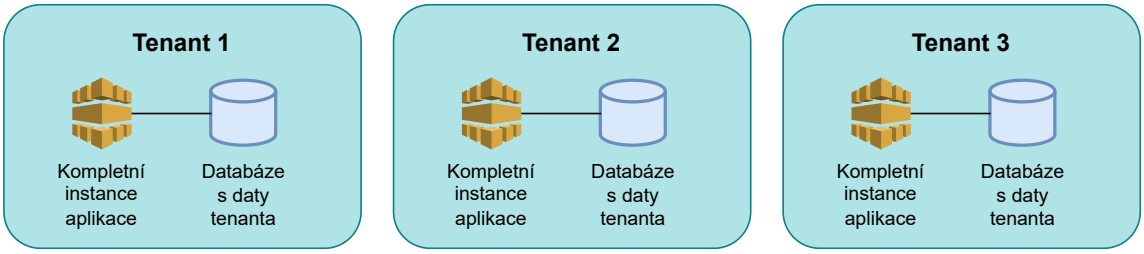
- **Množství pracovníků, kteří budou na převodu pracovat** – tento faktor se váže k poskytnuté časové dotaci. Opět určuje, jakým způsobem bude převod proveden.
- **Dovednosti pracovníků** – dovednosti pracovníků mohou ovlivnit jednak cenu a dobu trvání převodu, ale i kvalitu výsledného produktu. Pokud pracovníci nemají s touto problematikou moc zkušeností, tak by bylo dobré zvolit jednodušší variantu převodu, která ve výsledku může dosahovat větších kvalit než špatně udělaná složitější varianta, která by byla při správné implementaci efektivnější.
- **Velikost systému, jeho složitost a úroveň propojení a závislosti jednotlivých modulů** – při dobré modularitě systémových součásti nebude složité vyčlenit části systému, kontejnerizovat je a nasadit je na sdílený server. V extrémním případě, kdy systém bude tvořit obrovská nesmyslně propojená kupa modulů, pravděpodobně asi nebude mít smysl se aplikaci snažit rozdělit a možná bude lepším řešením původní aplikaci nasadit jako celou instanci do zvláštního clusteru a docílit tak simulace multitenance z uživatelského pohledu.
- **Provázanost aplikačního kódu s daty** je také důležitým aspektem při určování postupu převodu aplikace.
- **Průběh a nároky instalace** mohou ovlivnit výsledek z hlediska množství sdílených částí systému. Pokud pro každého nového zákazníka budeme muset složitě instalovat a nastavovat aplikaci, možná bude mít smysl strávit více času nad převodem aplikace a pokusit se co nejvíce částí aplikace převést na sdílenou formu, kterou bude třeba instalovat pouze jednou.
- **Způsob přístupu k systému a jeho zabezpečení včetně policy a privacy** – jsou případy, kdy interní politika zákazníka nedovoluje, aby určité součásti systémů fyzicky opustily prostor firmy. Bude se jednat především o citlivá data, ale může se jednat například i moduly, které jsou pro firmu důležité a jejich zveřejnění by firmu poškodilo. Při návrhu převodu je nutné myslet i na tyto aspekty a počítat s tím, že některé části systému musí zůstat u zákazníka.

Tyto faktory budou při popisu návrhu převodu zohledňovány a na jejich základě budou navržená možná alternativní řešení, která budou různě ovlivněna těmito faktory.

3.3 Volba vhodného řešení

Klasickým přístupem, jak z aplikace udělat její multitenantní verzi je kompletně modifikovat její kód a tuto funkcionalitu zavést do samotné aplikace. Pro rozsáhlejší aplikace je ale tento přístup příliš pracný a finančně náročný. Proto je cílem této práce vynechat standardní postup a navrhnout rychlejší a méně nákladné řešení, které minimalizuje nutné zásahy do aplikačního kódu a umožní převod aplikace v etapách.

Na obrázku 3.1 je zobrazena ilustrace původní architektury aplikace, kdy je systém poskytován „on-premise“, tedy tak, že každý každý zákazník vlastní kompletní instanci poskytovaného systému včetně vlastní databáze. Systémy jsou nasazeny přímo na zákaznických serverech.



Obrázek 3.1: **Původní architektura jednotenantní aplikace.** Na obrázku je zobrazena architektura jednotenantní aplikace, kdy má každý tenant (jediný klient aplikace) na svém serveru kompletní instanci aplikace a neexistují žádné sdílené moduly, o které by se aplikace dělily.

3.3.1 Jednotenantní SaaS

Nejjednodušší a také nejrychlejší řešení převodu je prosté přesunutí kompletních instancí včetně dat na servery vývojářské firmy. Jednou z možností, jak vytvořit izolované prostředí pro běh těchto systémů, je vytvoření stejného počtu clusteru jako je instancí systému a v každém takovém clusteru nasadit zvláštní instanci celé aplikace pro každý tenant.

Hlavní výhodou tohoto řešení je snadná implementace a absence nutnosti jakékoliv modifikace funkcionality systému. Odpadá také nutnost ošetřovat izolaci prostředí jednotlivých tenantů z důvodu ochrany dat před napadením jiným tenantem, protože nasazení každé aplikace do zvláštního clusteru je již dostatečnou izolací.

Nevýhodou tohoto řešení je obrovská neefektivita, co se týče využití zdrojů. Pro každý tenant musí být vytvořen zvláštní cluster s kompletní kopií aplikace, kvůli čemuž bude hodnota nároků na zdroje odpovídat minimálně nárokům původní aplikace vynásobená počtem tenantů. K těmto nárokům je nutné ještě připočítat nároky na správu každého z nově vytvořených clusterů.

Přesun tedy nepřináší žádnou úsporu z hlediska celkového součtu výkonu na provoz systémů, ani neimplementuje žádné sdílené části systému a není třeba téměř žádné orchestrace. Úspory je ale možné dosáhnout z hlediska serverů potřebných k běhu aplikace. V původním řešení bylo nutné, aby u každého zákazníka existoval zvláštní server pro každou instanci aplikace. V novém řešení se jednotlivé instance budou dělit o jeden server, případě o více serverů při distribuci aplikace.

Pokud bychom pracovali v řádech jednotek tenantů, je toto řešení v poměru cena/výkon uspokojivé. Při větším nárůstu počtu tenantů by ale toto řešení bylo neudržitelné (řádově stovky až tisíce). Prakticky toto řešení není ani multitenantní verzí aplikace, ale jen a pouze vytvořením zvláštní instance aplikace pro každý tenant na jednom stejném hostitelském serveru a umožnění distribuce aplikace jako SaaS (kapitola 2.1.1) řešení.

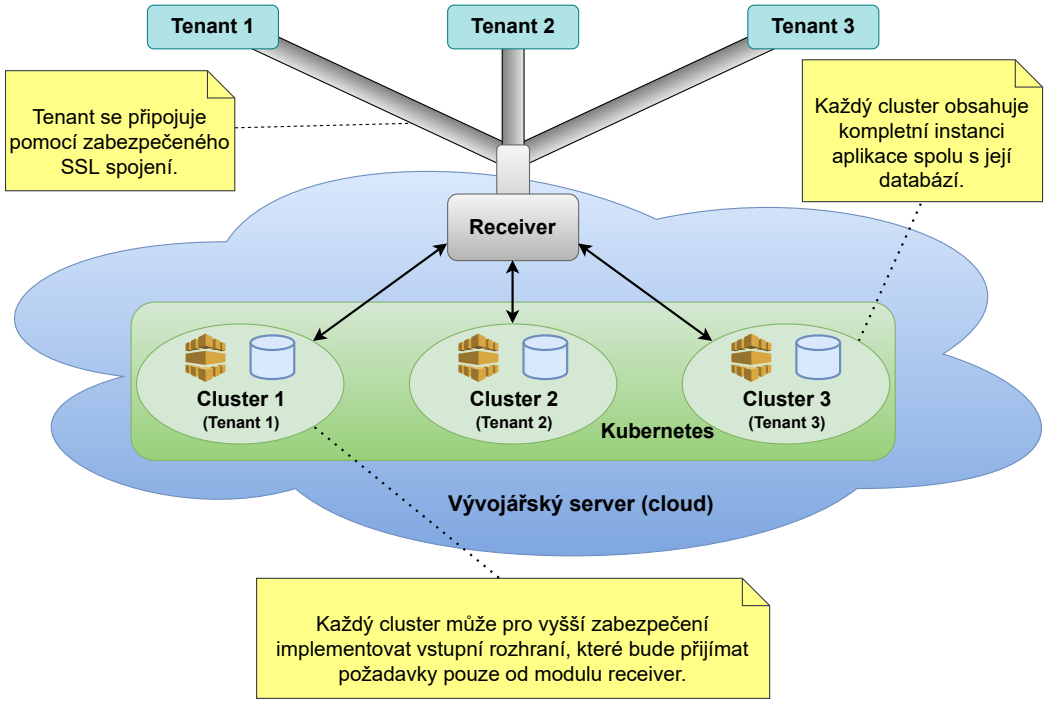
Řešení je vhodné v případě, pokud je nutné, či požadované, přesunout běžící instance na společné servery. Nasazením na své servery získá vývojář větší kontrolu nad aplikací a může libovolně nasazovat nové verze aplikace bez nutnosti kontaktu se zákazníkem. K provedení tohoto způsobu převodu není třeba velkého množství zodpovědných vývojářů a financí a ani časově toto řešení není náročné. Problémem tohoto řešení může být vnitřní politika zákaznické firmy, kdy může požadovat, aby systém běžel na HW firmy, případně aby data neopustila její prostory.

Jedinou novou součástí systému, kterou bude nutné vyvinout a nasadit, je vstupní rozhraní do celého systému, které bude přijímat požadavky od jednotlivých tenantů pomocí

zabezpečeného spojení, provádět autentizace a delegovat požadavky na příslušné clustery obsahující instance aplikací. Tato komponenta je v této práci nazvána jako „receiver“ a je blíže popsána v kapitole 4.3.2.

Předpokládá se, že v rámci cloudu neexistuje žádná jiná aplikace, která by mohla kromě modulu receiver odesílat požadavky na clustery. Pokud by tomu tak nebylo, případně pokud clustery neobsahují pouze důvěryhodné aplikace, je dobré pro vyšší zabezpečení implementovat vstupní rozhraní do každého clusteru, které bude přijímat požadavky pouze od modulu receiver.

Na obrázku 3.2 je architektura takto navrženého systému. Jednotliví tenanti se připojují vzdáleně na servery vývojářské firmy, kde jsou pomocí komponenty receiver přesměrováni na příslušné instance aplikace.



Obrázek 3.2: **Jednotenantní saas architektura.** Na obrázku je zobrazena architektura, kdy je na společném serveru pro každý tenant vytvořen cluster, ve kterém běží jeho vlastní instance systému včetně příslušné databáze. Žádné moduly systému nejsou sdíleny. Vstupní bod je receiver, který je jediným vstupem do tohoto cloudu a na kterém probíhá autentizace a následné přesměrování na příslušný cluster.

3.3.2 Přesun vyčleněných modulů

I když hlavním cílem celého převodu by mělo být přesunout ideálně celý systém, případně co možná největší část poskytovaného systému, na vlastní servery vývojářské firmy. Jak již bylo popsáno v kapitole 3.2, v některých případech nemusí být možné přesunout kompletně celé systémy a je nutné použít variantu přesunu pouze vybraných částí. Tento návrh je možné použít ve dvou hlavních případech:

- Prvním případem je **nutnost ponechat některé systémové moduly na serverech zákazníka**, případně zde ponechat databázi s citlivými daty a nebo kombinace

obou těchto možností – ponechání databáze společně s moduly, které pracují s těmito daty.

- Druhou možností využití tohoto návrhu je pouze jeho **dočasné použití jako mezikrok pro přesun kompletní aplikace**. Přesun celé aplikace je komplikovaný proces a v některých případech je lepší využít možného přesunu ve fázích. Sníží se komplexita jednotlivých kroků a přesun je možné vykonávat i při stálém provozu systému. Této možnosti je možné využít i v případě, že některé moduly bude možné v budoucnu přesunout, ale nyní jsou jakkoliv závislé na prostředí firmy a není je možné přesunout ihned.

Použití této varianty převodu je více náročné na zdroje (práce a finance) než předchozí varianta, která je popsána v kapitole 3.3.1. Zásadním krokem je totiž nejprve analyzovat převáděný systém a vybrat z něj části, které budou vyjmuty a přesunuty. Nejjednodušší přesun bude zpravidla malých mikroslužeb, které mezi sebou navzájem komunikují pomocí předem definovaného rozhraní a bude tedy jednoduché realizovat jejich vyjmutí a přesun.

Varianta je principiálně podobná převodu z kapitoly 3.3.1 s tím rozdílem, že se systém nebude přesouvat celý, pouze jednotlivé moduly, a zbytek systému zůstane u zákazníka. K nasazení takto vyjmutých aplikací je možné použít dva přístupy:

- Prvním základním přístupem je **nasazení každého samostatného modulu do separátního clusteru**. Co se týče nároků na zdroje, tak je tato varianta velice neefektivní, protože s každým clusterem navíc nároky rostou. Řešení se z počátku může zdát jako nejjednodušší na implementaci avšak zde hraje roli jeden důležitý aspekt, jímž je bezpečnost a s ním spojená důvěra mezi nasazenými aplikacemi a jednotlivými tenanty. Důvěra je v tomto případě důležitá věc, nicméně při větším množství tenantů je konfigurace složitější a problém nemusí nastat úmyslně, ale i chybou v konfiguraci. Každý cluster obsahující jeden modul bude muset implementovat metody pro přístup z vně tohoto clusteru. Logicky pak nastává problém, pokud dojde k nějaké komunikaci mezi komponenty nenáležícími různým tenantům.

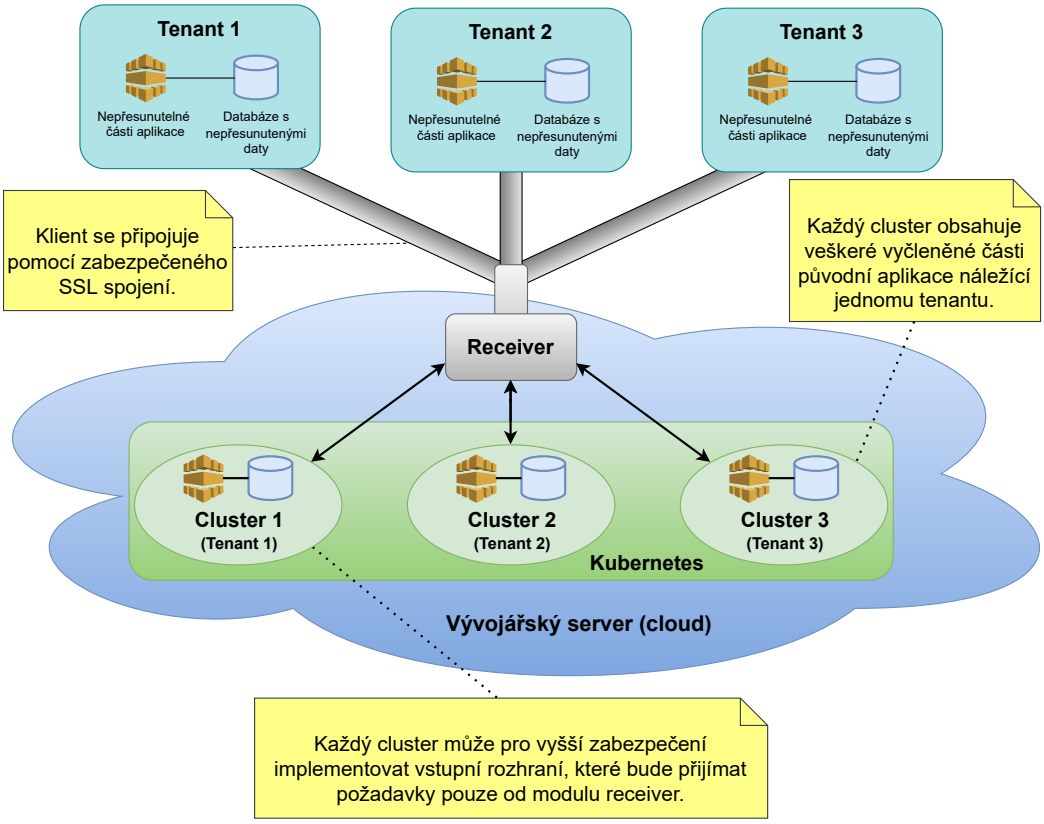
  Z tohoto důvodu bude nutné implementovat nějakou metodu autentizace.

  - Možností A je globální směrovač, který by obstarával veškerou komunikaci uvnitř systému Kubernetes. Zde opět může nastat problém s konfigurací, jako v podstatě všude, ale problematika směrování je alespoň centralizována do jednoho prvku. Tento způsob tedy zahrnuje vytvoření separátní aplikace a na základě implementace této aplikace je možné pro větší bezpečnost implementovat jednoduchou úpravu vyjmutých modulů – například implementaci vkládání identifikátoru do zprávy a kontrolu příchozího identifikátoru/podpisu.
  - Možností B je implementace autentizačního mechanizmu do každého clusteru. Opět je nutné vytvořit a nasadit autentizační modul. V tomto případě je ale nutné ho nasadit ke každé jedné vyjmuté části. Pokud už vyjmuté komponenty nějakou takovou funkcionalitu neobsahují, pak je tento přístup zbytečně náročný a je nevhodné ho použít.

- Druhým základním způsobem je **nasazení aplikací náležících jednomu tenantu do společného clusteru**. Každý takový cluster bude obsahovat centrální prvek pro příchozí komunikaci, který bude kontrolovat validitu a práva na přístup daného požadavku. Tento přístup opět vyžaduje přidání nějaké autentizace, ale výhodou je, že

komponenty v rámci clusteru už budou moci komunikovat bez dodatečného zabezpečení.

Z hlediska obtížnosti implementace i optimálnosti výsledného řešení je nejlepším způsobem nasazení aplikací patřících jednomu tenantu do stejného clusteru, tedy druhý výše popsaný způsob. Nasazení do separátního clusteru slouží především k izolaci, což je u modulů systému patřící jednomu tenantu zbytečné. S každým vytvořeným clusterem také rostou nároky na zdroje a vytvářením dalších clusterů se zbytečně zvýší. Implementace bude poté obdobná jako u varianty popsané v kapitole 3.3.1. Žádné moduly systému nebudou sdíleny, takže nároky na zdroje na provoz tohoto způsobu budou také obdobné. Jediným podstatným rozdílem bude, že část aplikace bude nasazena u zákazníka a část ve vývojářské firmě. Na obrázku 3.3 je zobrazena architektura tohoto způsobu převodu.



Obrázek 3.3: **Přesunutí vyčleněných modulů.** Na obrázku je zobrazena architektura, kdy na vývojářský server byly nasazeny vyčleněné moduly. Zbylé části systému jsou nasazeny na serverech tenantů. Tyto moduly nejsou sdílené a všechny moduly náležící jednomu tenantu jsou společně izolovány v rámci jednoho clusteru.

3.3.3 Přesun vyčleněných modulů a jejich sdílení

Tento způsob převodu je vylepšením alternativy popsané v kapitole 3.3.2. Základním rozdílem je sdílení vyčleněných modulů. Takovéto moduly budou bezstavové a budou přístupné pro všechny tenanty.

Hlavní výhodou tohoto přístupu je ušetření zdrojů na provoz systému z důvodu sdílení některých částí. V tomto přístupu se už začíná projevovat potenciál orchestrace z hlediska

možnosti škálování. Při použití správných komponent tato varianta získává mnohem větší potenciál než varianty předchozí. V kapitole 2.4 jsou popsány komponenty tohoto systému, které mohou sloužit k automatickému škálování a implementují low-balancing, self-healing a další funkce. Pro jednoduchost a přehlednost většina těchto komponent není v ilustraci architektury zahrnuta a bude zmíněna až v kapitole týkající se konkrétní implementace (kapitola 4).

Při použití tohoto přístupu je jako v předchozí variantě nutné analyzovat původní systém a vyčlenit některé moduly, které mohou být přesunuty do společného cloudu. K tomu je ale navíc nutné upravit moduly tak, aby byly bezstavové a mohly se sdílet napříč tenanty. Převod modulů do bezstavové verze je nad rámec této práce.

V tomto návrhu jsou přesunuty pouze bezstavové sdílené moduly. Je samozřejmě možné přesunout i moduly, které jsou tenantně specifické. Ty by byly nasazeny ve zvláštním clusteru pro každý tenant. Orchestrace specifických modulů společně se sdílenými moduly je popsána v kapitole 3.3.4.

Bezstavové moduly musí buď veškeré potřebné informace k výpočtu obdržet v rámci požadavku, nebo je nutné implementovat způsob přístupu tohoto sdíleného modulu k databázi, což navazuje na další problematiku, a sice na vytvoření sdílené databáze.

Sdílenou neboli multitenantní databázi je možné implementovat několika způsoby. Tyto způsoby jsou podrobněji rozebrány v rámci implementace v kapitole 4.2. V rámci návrhu je důležité pouze vědět, že bude třeba jedna ze dvou následujících věcí:

1. Modifikace sdíleného modulu tak, aby plnil jednu z následujících možností:
   - Na základě příchozího požadavku vytvořit připojení ke správné databázi patřící tenantu, který požadavek vytvořil.
   - Na základě požadavku vytvořit takový databázový dotaz, aby v případě multitenantní databáze pracoval se správnými daty.
2. Přítomnost aplikace, která bude přijímat požadavky od sdílených modulů a bude je modifikovat pro dotazy do multitenantní databáze, případně směrovat na správnou databázi.

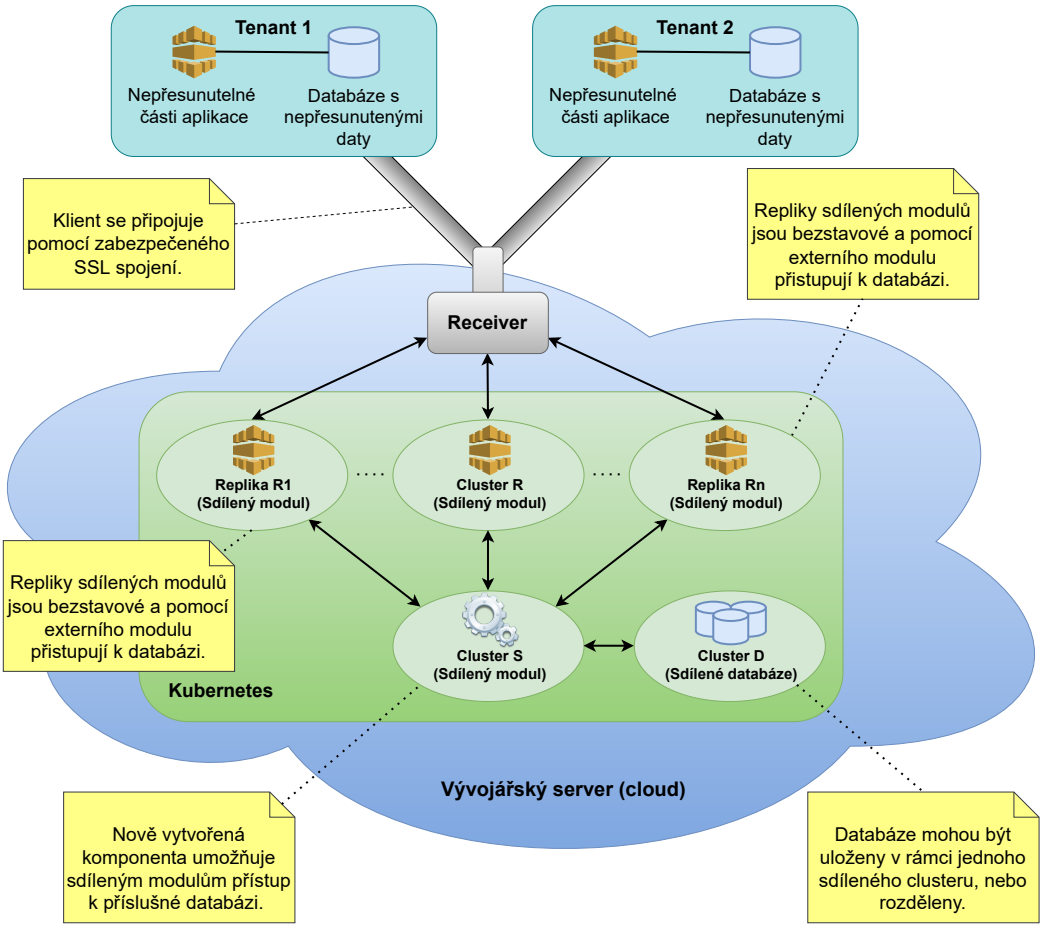
Opět nejde jednoznačně říct, která z variant je jednodušší či efektivnější na použití, vždy záleží na povaze sdílených modulů a odbornosti pověřených pracovníků. Pro obě varianty je možné použití vlastních databází pro každý tenant či jedné multitenantní databáze.

Na obrázku 3.4 je ilustrace návrhu této architektury. Zobrazená architektura opět cílí na popis základního principu fungování tohoto návrhu. Jednotlivé detaily, jako například použití komponent Kubernetes, jsou zmíněny až v kapitole týkající se implementace.

3.3.4 Přesunutí celého systému včetně sdílení modulů

Kompletní přesunutí systému tak, aby veškeré komponenty běžely na vývojářských serverech a zároveň nějaké přesunuté komponenty byly sdíleny, je cílem, kterým se zabývá tato práce. Předchozí varianty byly různými kombinacemi, kdy se přesunul celý systém, ale nebyl nijak orchestrovaný a sdílený, případně byly pouze přesunuty části systému a určité komponenty stále zůstaly nasazeny na serverech zákazníka.

I když je tato varianta co se týče výsledku z pohledu vývojářské firmy nejlepší, nemusí být vždy ideální variantou. Překážkou může být příliš náročný převod, nedostatečné množství pracovníků, případně jejich nedostatečná kvalifikace nebo jiné překážky komplikující přesun aplikace od zákazníka.



Obrázek 3.4: **Přesunutí vyčleněných modulů a jejich sdílení.** Na obrázku je zobrazena architektura, kdy na vývojářský server byly nasazeny vyčleněné moduly, které jsou bezstavové a jsou možné sdílet mezi tenanty. Každý sdílený modul je izolován v rámci jednoho clusteru a je pomocí komponentů kubernetes orchestrován. Orchestrace zabezpečuje například automatické vytváření replik v případě většího náporu na modul. Zbylé části systému jsou nasazeny na serverech tenantů.

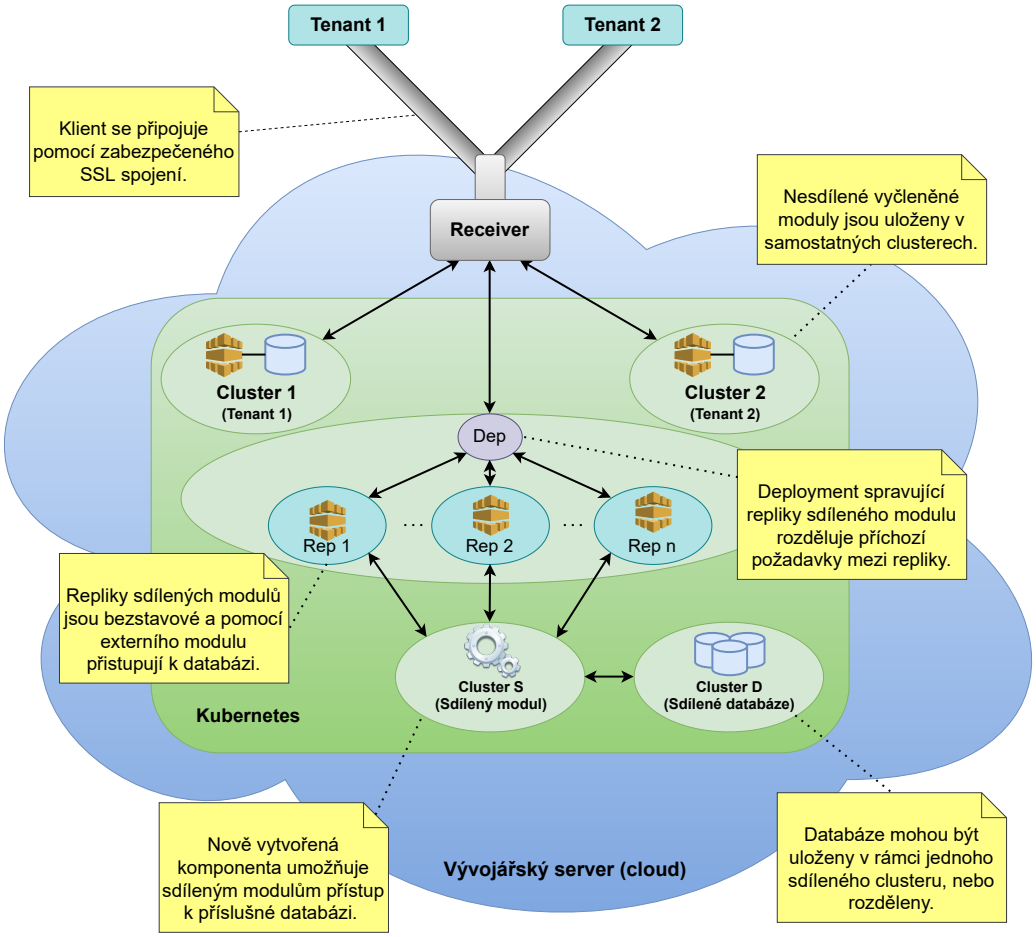
Předpokládaným výsledkem této varianty je kompletní přesunutí systému jako tomu bylo v případě návrhu v kapitole 3.3.1. Po přesunu systému ale nevznikne samostatný cluster s kompletní instancí aplikace, ale výsledkem budou vyčleněné moduly, které budou sdílené mezi tenanty a tenantně specifické části ve speciálním clusteru pro každý tenant.

Jako ve všech předchozích variantách bude implementovaný modul receiver, který bude vstupním bodem do běhového prostředí. Tento modul slouží hlavně pro zabránění neoprávněnému přístupu. Je možné, že aplikace už budou mít na jejich rozhraní implementovanou autentizaci a nebude nutné tento modul používat. Z hlediska obecné bezpečnosti je tento modul v práci uvažován.

Pomocí komponent Kubernetes bude sdílený modul v případě potřeby replikován. Tuto replikaci při správném nastavení řeší komponenty Kubernetes automaticky, stačí zadat požadovaný počet replik.

Implementace sdílených modulů vyžaduje dynamický přístup do databáze. Ten je vyřešen jako v předchozím návrhu pomocí nové komponenty, která bude řídit přístup do databáze. Alternativou je modifikace sdíleného modulu tak, aby byl schopný do příslušné databáze přistoupit sám. Možnosti návrhu přístupu tohoto řešení k databázi již byly probrány v předchozí kapitole a nejsou tu proto podrobně zmíněny.

Ilustrace architektury je zobrazena na obrázku 3.5, kde opět z důvodu jednoduchosti a přehlednosti návrh nepopisuje konkrétní implementační detaily tohoto řešení. Ty jsou popsané v kapitole 4.



Obrázek 3.5: **Přesunutí celého systému včetně sdílení modulů.** Na obrázku je zobrazena architektura, kdy na vývojářský server byly nasazeny jak vyčleněné moduly, které jsou bezstavové a jsou možné sdílet mezi tenanty, tak i části systému které jsou specifické pro každého tenanta. Každý sdílený modul je izolován v rámci jednoho clusteru a je pomocí komponentů Kubernetes orchestrován. Orchestrace zabezpečuje například automatické vytváření replik v případě většího náporu na modul.

3.3.5 Multitenantní cluster

Tato kapitola je rozšířením všech předchozích návrhů architektur. V rámci systému Kubernetes umístěného na sdíleném serveru je možné implementovat multitenantní cluster, který bude sloužit jako náhrada více menších clusterů.

Multitenantní cluster je možné použít jak při orchestraci sdílených modulů, které jsou vyčleněné v kapitole 3.3.3, tak i pro orchestraci tenantně specifických modulů, jejichž vyčlenění je popsáno v kapitole 3.3.4. Teorie týkající se multitenantních clusterů je popsána v kapitole 2.5.

Vzhledem k tomu, že rozdělení do clusterů sice vytváří silnou izolaci, ale rozhodně se nejedná o jednoduché řešení co se týče jeho vytvoření, tak v případě menších nároků na izolaci tenantů, například pokud jsou aplikace zabezpečeny již z jejich povahy, je možné jako alternativu využít právě toto řešení se sdíleným clusterem, kde by se v podstatě implementovalo navržené řešení, jen by místo opravdových clusterů existovaly virtuální clustery. Vývoj nástrojů vhodných k implementaci tohoto řešení jde neustále kupředu a je možné, že v budoucnu budou funkční alternativou vytváření opravdových clusterů i v oblasti izolace.

3.3.6 Kombinace návrhů

Předchozí návrhy popsané v kapitole 3.3 nejsou jedinými možnostmi, kterými je možné výsledný systém implementovat. Architektury je možné přizpůsobit nebo kombinovat. Výsledek záleží na mnoho faktorech a práce cílí na poskytnutí základních návrhů, myšlenek a implementací tohoto převodu. Vývojář implementující postup této práce by měl nastudovat poskytnutou teorii v kapitole 2 a poté se očekává využití této teorie k výběru některé z poskytnutých architektur, její modifikaci pro konkrétní požadované řešení a následné implementaci.

Kapitola 4

Implementace a vyhodnocení

Jak již bylo popsáno dříve, hlavním cílem této práce je uvést vývojáře, který bude implementovat převod systému, do dané problematiky, poskytnout mu soubor potřebné teorie spolu s návrhy řešení a části možných implementací včetně jednoduchého demonstračního řešení. Tato kapitola obsahuje v úvodu dodatečné informace k možným implementacím a dále je v kapitole rozebráno demonstrační řešení, které je k práci přiloženo. Demonstrační řešení je poté možné využít jako základ implementace vlastního řešení.

Vzhledem ke kombinaci objemu problematiky, kterou bylo nutné nastudovat a sepsat, a omezených vývojových prostředků nebyly všechny principy navrhovaných převodů implementovány. Právě tyto části jsou vhodnými kandidáty na navázání a rozšíření práce v rámci dalšího výzkumu.

4.1 Úvod k implementaci navržených řešení

V této kapitole jsou přiblíženy některé implementační detaily navržených řešení.

4.1.1 Clustery

Vytvoření a správa více clusterů není triviální problém. Pokud není třeba v multitenantním systému striktní izolace, pak je možné aplikace nasadit v rámci jednoho clusteru a k izolaci využít například jmenné prostory. Další alternativou by byly nástroje implementující multitenanci jednoho clusteru.

K vytvoření více clusterů se doporučuje využít více zařízení, pokud to z jakéhokoliv důvodu není vhodné řešení, pak existují i alternativy pro vytvoření více clusterů na jednom zařízení. Jedním z řešení je použití Linux containers[1], pomocí kterých se kontejnerizuje celý systém, jako je popsáno na webu Ubuntu v článku How to deploy one or more Kubernetes clusters to a single box[2].

Systém Kubernetes obsahující více clusterů se označuje jako „multi-cluster“. Obecná teorie týkající se této problematiky je dobře sepsaná ve článku Kubernetes Multi-Clusters: How & Why To Use Them[3]. Práce s multi-cluster systémem je popsána například ve článku What is Kubernetes multi-cluster?[4].

[1] https://linuxcontainers.org/

[2] https://ubuntu.com/blog/how-to-deploy-one-or-more-kubernetes-clusters-to-a-single-box

[3] https://www.bmc.com/blogs/kubernetes-multi-clusters/

[4] https://www.mirantis.com/cloud-native-concepts/getting-started-with-kubernetes/what-is-kubernetes-multi-cluster/

4.1.2 Řídící uzly

V rozsáhlejším produkčním systému Kubernetes je lepší implementovat větší počet řídících uzlů (v anglickém jazyce často označovaných jako „control plane“). Implementace těchto řídících uzlů je nad rámec práce. Clustery s více řídícími uzly jsou označované jako „High available clusters“ a jejich implementace je popsaná v oficiální dokumentaci Kubernetes[5].

4.1.3 Pomocné komponenty

Převod systému vyžaduje některé pomocné komponenty, jejichž implementace minimalizuje změny potřebné v původním aplikačním kódu. Na základě povahy původního systému může být vytvořeno a použito těchto komponent více či méně. Cílem práce je poskytnout potřebné komplexní znalosti, aby bylo možné s použitím informací v této práci navrhnout a implementovat převod dle specifických potřeb.

4.1.3.1 Databázový směrovač

Jedná se o možné řešení implementace sdílené databáze v poskytnutých návrzích převodu. V demonstračním řešení byl použit jiný přístup, takže práce se implementací této komponenty nezabývá, ale podobná tématika je zmíněna v kapitole 4.2.

4.1.3.2 Receiver

Komponenta receiver použitá v návrhu převodu slouží k přijímání a zpracování veškerých požadavků přicházejících do systému. Demonstrační příklad její možné implementace se nachází v kapitole 4.3.2.

4.1.4 Sdílené moduly

Převod vyčleněných modulů na sdílené moduly je velice subjektivní problematika, která závisí na konkrétní implementaci aplikace, a není tak součástí této práce. V ilustraci návrhu byly tyto komponenty pro jednoduchost umístěné přímo v clusterech. Ke správné funkci systému je nutné tyto moduly „zabalit“ do pomocných komponent jako jsou například deployment nebo service. Dodatečné komponenty, které budou potřeba, jsou teoreticky popsané v kapitole 2 a v kapitole 4.3 jsou příklady použití některých těchto komponent.

4.2 Implementace databáze multitenantního systému

Z důvodu použití sdílených bezstavových modulů bude nutné vyřešit implementaci databáze v multitenantní podobě. Tato multitenance bude muset být implementovaná přímo v systému, ale v rámci této práce jsou navrženy dva možné přístupy.

4.2.1 Samostatná databáze pro každý tenant

Z technického hlediska je možné vytvořit pro každý tenant zvláštní databázi a poté aplikačně vytvořit dynamický přístup k těmto databázím. Výhodou tohoto přístupu je silnější izolace a výhody spojené s případnou modifikací databáze pro jednotlivé tenanty, případně

[5] https://kubernetes.io/docs/setup/production-environment/tools/kubeadm/high-availability/

migrace či smazání databáze. Databáze mohou být dokonce i různých typů, což by se ale muselo poté zohlednit u modulu zprostředkujícího přístup k těmto databázím.

Tento přístup je použit v demonstračním řešení a je popsán v kapitole 4.3.5.1.

4.2.2 Multitenantní databáze

Řešením tohoto problému může být také vytvoření pouze jedné instance databáze a její sdílení mezi tenanty.

Takovým jednodušším přístupem k vytvoření multitenantní databáze je přidání jednoho atributu do každé tabulky relační databáze, kdy daný atribut bude určovat náležitost dat tenantu. Poté je nutné upravit implementaci sdílených modulů, aby při práci s daty dbaly na práci s daty s příslušnou identifikací v tomto sloupci, případně použít externí modul, který bude tuto funkcionalitu zabezpečovat.

K tvorbě multitenantní databáze existuje samozřejmě více přístupů. Další přístupy jsou popsané například v článku Strategies for Using PostgreSQL as a Database for Multi-Tenant Services[6] od Leonida Belkinda.

Pro použití multitenantních databází samozřejmě existují nástroje, které umí tento proces částečně automatizovat. Jedním takovým nástrojem je například FilteredQuery[7], který je možné použít při implementaci sdílených modulů (případně modulů spravujících databázi) pomocí frameworku Flask.

4.3 Demonstrační řešení

V rámci práce bylo vytvořeno jednoduché demonstrační řešení, které reflektuje základní principy převodu popsaného v práci a je možné toto řešení použít jako základ pro tvorbu vlastního systému.

Pro tvorbu aplikací v celém demonstračním řešení byl zvolen mikro webový framework pro jazyk python – Flask. Základem demonstračního řešení je modelový systém, který je teoreticky popsaný v kapitole 3.1. Tento systém se skládá z následujících částí:

- uživatelského rozhraní,
- modulu receiver,
- API modulu.

4.3.1 Uživatelské rozhraní

Tento modul je uživatelským rozhraním, kde se nachází formulář, který slouží k odeslání dat do další části aplikace na zpracování a po obdržení odpovědi k jejímu zobrazení.

Modul se skládá ze dvou HTML šablon, z nichž jedna obsahuje stránku s formulářem a druhá prostředí pro zobrazení odpovědi (průměrné výšky). Logika rozhraní se nachází v souboru *app.py*, který obsahuje metody pro směrování a následného vykreslení příslušné šablony.
První směrovací pravidlo je určené pomocí:

```
@app.route("/")
```

[6] https://dev.to/lbelkind/strategies-for-using-postgresql-as-a-database-for-multi-tenant-services-4abd

[7] https://github.com/sqlalchemy/sqlalchemy/wiki/FilteredQuery

Pomocí metody `render_template()` kterou obsahuje přímo flask je vyrendrována a vrácena stránka s formulářem.

```
return render_template("index.html")
```

Druhé směrovací pravidlo je určené pomocí:

```
@app.route("/success", methods=['POST'])
```

Kdy je specifikována nová cesta a povolena metoda POST. Odeslání formuláře uživatele přesměruje pomocí metody POST právě na tuto cestu a pokud by POST v konfiguraci specifikované nebylo, pak by server požadavek automaticky odmítl. V rámci metody zpracovávající tuto cestu jsou načtena data z formuláře:

```
email=request.form["email_name"]
height=request.form["height_name"]
tenant=request.form["tenant_name"]
```

A pomocí `requests` obsaženého opět přímo ve flasku je odeslán požadavek do dalšího modulu na zpracování:

```
response = requests.get('https://diplomka.fun/height/' + email + \
"/" + height + "/" + tenant)
```

Odpověď se poté zpracuje a pokud byla vstupní data validní, pak se zavolá metoda pro vyrendrování stránky s výsledkem a jako parametry jsou ji předány HTML šablona a data:

```
data = response.json()
if data["valid"] == True:
        return render_template("success.html", \
        average_height=data["average_height"])
    return render_template('index.html', \
    text="Od tohoto emailu jsme jiz data obdrzeli.")
```

Tato aplikace byla spouštěna na lokálním prostředí z důvodu zjednodušení vývoje. V rámci tohoto vývoje byl ke spouštění aplikace použit výchozí webový server, který flask poskytuje. Aplikace takto funguje v pořádku, ale v praxi je dobré použít nějaký externí webový server, například tak, jak je tomu u komponenty receiver v kapitole 4.3.2.

4.3.2 Receiver

Receiver slouží jako vstupní bod pro části orchestrovaných aplikací umístěných na serveru. Veškeré požadavky mířící na server jsou tímto modulem zpracovány. Modul je nově vytvořená část a je možné ho nakonfigurovat dle specifických potřeb.

4.3.2.1 Směrování příchozích požadavků

Základní funkci, kterou tento modul plní, je směrování příchozích požadavků dále na server obsahující orchestrovaný systém, což se dá označit jako „reverse proxy“. V demonstračním řešení je na serveru pouze jediný sdílený modul, a tedy pouze jedna možnost směrování. Receiver tedy zprávu přijme, zpracuje a přepošle na jediný možný další modul. Logika

zpracování a přeposlání požadavku byla převzata z článku Simple Reverse Proxy Server Using Flask[8].

Aplikace je opět vytvořena pomocí frameworku Flask. Každý požadavek je zpracováván na základě jeho metody a to z důvodu, že hlavičky požadavků jednotlivých metod jsou lehce odlišné. Požadavek je přijat následujícím způsobem, kdy se do proměnných uloží cesta požadavku, opět jsou specifikovány povolené metody:

```
@app.route('/<path:path>', methods=['GET', 'POST', 'DELETE'])
```

Poté proběhne dělení pomocí jednoduchých podmínek:

```
if request.method=='GET':
        .
        .
elif request.method=='POST':
        .
        .
elif request.method=='POST':
        .
        .
```

V tělech těchto podmínek jsou jednotlivé typy požadavků přeposlány a odpovědi zpracovány následovně:

- Get:

```
response_from_server = requests.get(f'{redirect_to}{path}')
excluded_headers = ['content-encoding', 'content-length', \
'transfer-encoding', 'connection']
headers = [(name, value) for (name, value) in \
response_from_server.raw.headers.items() if name.lower() \
not in excluded_headers]
```

- Post:

```
response_from_server = requests.post(f'{redirect_to}{path}', \
json=request.get_json())
excluded_headers = ['content-encoding', 'content-length', \
'transfer-encoding', 'connection']
headers = [(name, value) for (name, value) in \
response_from_server.raw.headers.items() if name.lower() \
not in excluded_headers]
```

- Delete:

```
response_from_server = \
requests.delete(f'{redirect_to}{path}').content
```

[8] https://medium.com/customorchestrator/simple-reverse-proxy-server-using-flask-936087ce0afb

Proměnná `redirect_to` obsahuje adresu serveru, na kterou bude požadavek přeposlán. Před přeposláním by tedy byla obsažena logika, která by na základě nějakého identifikátoru obsaženého v požadavku, případně dle identifikace pomocí autentizace, nastavila tuto proměnnou na příslušnou adresu. Poté je odpověď od vnitřního modulu seskládána a přeposlána na adresu původu dotazu:

```
response_to_client = Response(response_from_server.content, \
response_from_server.status_code, headers)
return response_to_client
```

4.3.2.2 Záznamy o příchozích požadavcích

Vzhledem k tomu, že veškeré vstupní požadavky prochází přes tento modul, je velice dobrým kandidátem na vytváření záznamů o těchto požadavcích. Nejprve je získán aktuální čas z knihovny `datetime`:

```
now = datetime.now()
formated_now = now.strftime("%d.%m.%Y %H:%M:%S:%f")
```

Poté je otevřen soubor se všemi záznamy a je do něj vepsán nový záznam:

```
with open("logs/all.txt", "a") as file:
    file.write(request.method + " " + request.remote_addr + " " + \
    path + "\n")
```

Dále je vytvořen, případně otevřen, soubor obsahující záznamy z jednoho zdroje:

```
with open("logs/client_" + request.remote_addr + "_" + \
path.split("/")[-1] + ".txt", "a") as file:
    file.write(formated_now + " " + request.method + " " + \
    request.remote_addr + " " + path + "\n")
```

Takovéto zpracovávání požadavků je vhodné pro případné omezování přístupu, kdy je možné kontrolovat množství požadavků na určité sdílené moduly a případně některé požadavky upřednostnit či zpozdit.

V případě potřeby je možné tento modul také kontejnerizovat pro jednodušší nasazování a poté je vhodné zajistit jeho orchestraci z pohledu automatického škálování pomocí komponenty deployment (popsané v kapitole 2.4.2.4). Při orchestraci tohoto modulu je také nutné zpřístupnit tento modul pomocí service (kapitola 2.4.2.6) či ingress (kapitola 2.4.2.7). Při orchestraci a uchovávání záznamů v souborovém systému bude také nutné použít statefulSet popsaný v kapitole 2.4.2.5.

4.3.3 Hosting a veřejná doména

Součástí práce bylo i nasazení tohoto modulu na cloudovou službu, konkrétně na Amazon ec2[9]. Tento cloud poskytuje možnosti bezplatných služeb. Ty jsou sice omezené, co se týče výkonu a úložiště, ale na testování demonstračního řešení to bylo dostatečné. Amazon pomocí intuitivního menu umožňuje vytvořit virtuální stroj obsahující jeden z nabízených systému. Pro účel této práce byla nejlepší volba poslední stabilní verze Ubuntu.

[9] https://aws.amazon.com/

Výchozí použití je pomocí SSH připojení a použití příkazové řádky. Pokusil jsem se na stroj doinstalovat přídavky pro grafické rozhraní z důvodu jednodušší obsluhy databáze, to ale virtuální stroj zpomalilo natolik, že jsem zůstal u konzolové verze.

Poté jsem pomocí gitu naklonoval projekt s vytvořeným demonstračním řešením, které jsem vyvinul na lokálním prostředí a nastavil vše potřebné pro spuštění nginx serveru, aby se aplikace stala veřejně dostupnou. Na server jsem umístil také databázi, jejíž konfigurace je popsaná v kapitole 4.3.6. Konfigurace nginx se zabezpečeným spojením byla složitější, ale veškeré důležité části této konfigurace jsou popsány v kapitole 4.3.4.

4.3.4 Nginx

Flask obsahuje vlastní server, na kterém aplikace může fungovat, ale není doporučeno ho používat v produkčním prostředí. Z tohoto důvodu se používá gunicorn[10], který slouží jako WSGI server komunikující s danou flask aplikací, v kombinaci s nginx[11], který slouží jako proxy server mezi gunicorn serverem a klientem.

Tento návod obsahuje kompletní rozjetí aplikace. Nejprve je nutné nainstalovat `pip`, pomocí kterého bude poté nainstalováno `virtualenv`, což je nástroj, který slouží pro vytvoření virtuálního python prostředí. Toto prostředí bude obsahovat veškeré závislosti pro běh aplikace.

```
sudo apt update
sudo apt-get install python3-pip
sudo pip3 install virtualenv
```

Poté, co je `virtualenv` nainstalováno, je nutné toto prostředí vytvořit, nejlépe ve složce aplikace:

```
python3 -m virtualenv virtual
```

Prostředí je nyní vytvořeno a pro přepnutí pro práci s tímto prostředím je nutné zadat příkaz:

```
source virtual/bin/activate
```

Nyní je nutné nainstalovat vše potřebné k běhu aplikace. Součástí demonstračního řešení je soubor `requirements.txt`, který slouží k jednoduché instalaci všech potřebných závislostí najednou a to následovně:

```
pip install -r requirements.txt
```

Nyní je možné pro ověření, že vše funguje zkusit spustit aplikaci pomocí gunicorn, zatím bez nginx:

```
gunicorn --bind 0.0.0.0:8000 receiver:app
```

Po ověření, že vše funguje tak, jak má, je možné přistoupit k instalaci a nastavení nginx. Nejprve je nutné opustit virtuální prostředí:

```
deactivate
```

[10] https://gunicorn.org/
[11] https://www.nginx.com/

Poté je nutné vytvořit konfigurační soubor, díky kterému gunicorn bude moci vytvořit `.sock` soubor, pomocí kterého se bude nginx odkazovat na aplikaci:

```
sudo nano /etc/systemd/system/flaskapp.service
```

Do tohoto souboru je třeba vložit tuto konfiguraci:

```
[Unit]
Description=Gunicorn daemon to serve my flaskapp
After=network.target
[Service]
User=ubuntu
Group=www-data
WorkingDirectory=/git/diplomka_demo_appka/Receiver
Environment="PATH=/git/diplomka_demo_appka/Receiver/virtual/bin"
ExecStart=/git/diplomka_demo_appka/Receiver/virtual/bin/gunicorn \
--workers 3 --bind unix:flaskapp.sock -m 007 wsgi:app
[Install]
WantedBy=multi-user.target
```

Po vytvoření tohoto souboru je nutné aktivovat tuto konfiguraci, zároveň nastavíme její provedení po spuštění systému:

```
sudo systemctl start flaskapp
sudo systemctl enable flaskapp
```

Ve složce s aplikací by nyní měl být vytvořen soubor `flaskapp.sock`. Následně je možné přistoupit k instalaci a konfiguraci samotného nginx:

```
sudo apt-get install nginx
sudo nano /etc/nginx/sites-available/app
```

Do tohoto souboru je nutno vložit následující konfiguraci, ve které je nutné změnit veřejnou IP daného serveru a cestu k `sock` souboru:

```
server {
 listen 80;
 server_name 172.31.36.86;

 location / {
 include proxy_params;
 proxy_pass http://unix:/git/diplomka_demo_appka/Receiver/flaskapp.sock;
 }
}
```

Dále je třeba vymazat výchozí konfiguraci nginx, jinak se bude místo aplikace na dané adrese zobrazovat domovská stránka tohoto nástroje, konfiguraci nahrát a nastavit spuštění po startu systému:

```
sudo rm -rf /etc/nginx/sites-available/default
sudo ln -s /etc/nginx/sites-available/app /etc/nginx/sites-enabled/
sudo systemctl start nginx
sudo systemctl enable nginx
```

Aplikace by měla být dostupná na serveru na portu 80. V případě potíží je možné, že má server tento port blokovaný v nastavení. Při potížích a aktualizaci konfigurace se hodí příkaz:

```
sudo systemctl reload nginx
```

Základní nastavení nginx je tímto hotové.

4.3.4.1 Zabezpečená komunikace

Pokud se předpokládá, že se bude k aplikaci přistupovat vzdáleně, je dobré, ale v dnešní době už spíše nutné, implementovat do aplikace možnost zabezpečeného spojení. Příkladem takového zabezpečení může být SSL a s ním spojený protokol HTTPS, který je vylepšením HTTP protokolu šifrovaného pomocí SSL nebo TLS. Implementace se samozřejmě bude lišit na základě zvoleného programovacího jazyka a nástrojů.

Při použití Kubernetes je možné k vytvoření zabezpečeného spojení použít komponentu ingress, která je popsaná teoreticky v kapitole 2.4.2.7.

V demonstračním řešení byl použit certbot[12], což je nástroj pro konfiguraci zabezpečeného připojení v rámci nginx s použitím certifikační autority Let's Encrypt[13]. Certbot implementuje průvodce konfigurací, díky čemuž je nastavení zabezpečeného spojení velice jednoduché. Jediný problém, který bylo nutné při tvorbě demonstračního řešení vyřešit, byla bezpečnostní zásada certifikační autority, která neposkytuje certifikáty pro doménová jména patřící pod aws z důvodu jejich časté změny a možných bezpečnostních rizik. Z tohoto důvodu byla zakoupena doména[14], na které byl nastaven DNS záznam tak, aby požadavek přesměroval na daný aws server. Před instalací nástroje certbot je lepší preventivně odstranit případnou jinou nainstalovanou verzi, aby byla použita ta nově nainstalovaná pomocí snap:

```
sudo apt-get remove certbot
sudo snap install --classic certbot
```

A posledním příkazem je spuštění průvodce nastavením:

```
sudo certbot --nginx
```

Po zadání požadovaných údajů by měla být aplikace dostupná i pomocí protokolu https.

4.3.5 API module

Api module je další část původní aplikace, která zpracuje příchozí požadavky. Na základě obdržených dat a dat z databáze provede výpočet, uloží nová data do databáze a vrátí výsledek.

Tento modul je z celého demonstračního řešení nejzajímavější. Je to totiž sdílený modul, který dynamicky přistupuje k příslušné databázi na základě příchozího požadavku. Zároveň je tento modul kontejnerizovaný a pomocí komponent systému Kubernetes je prováděno jeho automatické škálování.

Nejprve je popsán princip fungování tohoto modulu a poté bude demonstrováno vytvoření image a pomocných komponent jako deployment, service a secret. Implementace

[12] https://certbot.eff.org/
[13] https://letsencrypt.org/
[14] diplomka.fun

funkce původní aplikace není nijak zajímavá, a proto tu není zmíněna. Co je na aplikaci zajímavé je, že byly vytvořeny další moduly této aplikace, které slouží k dynamickému výběru databáze, ty jsou popsány v následující kapitole 4.3.5.1.

4.3.5.1 Výběr konkrétní databáze

Jak bylo zmíněno v návrhu převedeného systému v kapitole 3.5, je nutné zařídit, aby sdílený modul dynamicky přistupoval ke správné databázi. To je možné například pomocí speciálního modulu, který bude implementovat metody pro požadovanou práci s daty. Tyto metody jsou samozřejmě závislé na typu práce s daty. Alternativním řešením představeným v demonstračním řešení je vytvoření dodatečných modulů do původní aplikace a její minimální úpravy. Aplikace tak na základě příchozího požadavku a identifikace tenanta (Kde se v demonstračním řešení pouze přečte název tenanta z adresy dotazu, ale v reálném systému by mělo proběhnout nějaké bezpečnější ověření, například na základě podpisu, identifikátoru zakódovaného v požadavku a podobně.) vybere příslušnou databázi. Nejprve se tedy volá metoda:

```
get_tenant_session()
```

Tato metoda se nachází v nově implementovaném modulu, která pomocí metody `get_known_tenants()` ověří, zda tenant opravdu existuje. Metoda využívá k ověření existence tenanta databázi, která obsahuje seznam všech registrovaných tenantů. Poté se zavolá metoda:

```
prepare_bind(tenant_name)
```

Metoda se pokusí získat již existující připojení na danou databázi, které je v konfiguraci aplikace v položce `SQLALCHEMY_BINDS`. Pokud není připojení nalezeno, pak jsou načteny parametry pro připojení k databázi z proměnných prostředí a je vytvořený nový „bind“ na tuto databázi, který je pomocí `sessionmaker` uložen a tato session je předána do původní aplikace, kde bude použita pro přístup k datům.

Veškeré soubory obsahující konfiguraci tohoto typu připojení jsou přiloženy k této práci a je možné z nich implementaci převzít.

4.3.5.2 Vytvoření image

Základem pro vytvoření kontejnerizované aplikace (image tohoto kontejneru) je soubor `Dockerfile`, který se nachází ve složce s aplikací. Obsah tohoto souboru se liší podle povahy kontejnerizované aplikace. Každý kontejner je založen na nějakém již vytvořeném kontejneru, prvním řádkem tohoto souboru je tedy specifikace daného základního kontejneru:

```
FROM python:3.8.10 as compiler
```

Poté je vytvořena pracovní složka této aplikace:

```
RUN mkdir /app
WORKDIR /app/
```

Dále je vytvořeno virtuální prostředí, které aplikace používá pro svůj běh:

```
ENV VIRTUAL_ENV=/opt/virtual
RUN python3 -m venv $VIRTUAL_ENV
ENV PATH="$VIRTUAL_ENV/bin:$PATH"
```

Virtuální prostředí je nastaveno do ENV. Pokud by bylo použito `activate`, jako při klasickém spuštění aplikace, pak by nastal problém, protože v rámci Dockerfile by tento „bind“ na virtuální prostředí platil jen pro daný příkaz. Když už je virtuální prostředí vytvořeno, tak je možné do něj nainstalovat všechny potřebné závislosti:

```
COPY ./requirements.txt /app/requirements.txt
RUN pip3 install -r requirements.txt
```

Poté je možné pomocí dalšího python image nastavit průběh spouštění této aplikace:

```
FROM python:3.8.10 as runner
WORKDIR /app/
ENV VIRTUAL_ENV=/opt/virtual
COPY --from=compiler $VIRTUAL_ENV $VIRTUAL_ENV

ENV PATH="$VIRTUAL_ENV/bin:$PATH"

COPY . /app/
CMD [ "python3", "api_module.py" ]
```

Po nakonfigurování Dockerfile je možné vytvořit požadovaný image pomocí příkazu:

```
sudo docker build -f Dockerfile -t api-module:latest .
```

Takto vyrobený image je možné pro lepší dostupnost nahrát na nějaké veřejné úložiště, toto nahrání je popsané v kapitole 4.3.5.3.

4.3.5.3 Nahrání image do veřejné knihovny

Pro jednodušší nasazení kontejnerizované aplikace je možné nahrát image tohoto kontejneru na veřejný server, například Docker Hub[15]. Pro nahrání kontejneru je nutné se nejprve registrovat a následně vytvořit repozitář pro tento image. Poté se stačí přihlásit, nastavit vytvořenému image tag obsahující označení v knihovně (přidává se uživatelské jméno) a tento označený image nahrát:

```
sudo docker login
sudo docker tag api-module:latest rouskisroup/api-module:latest
sudo docker push rouskisroup/api-module:latest
```

4.3.5.4 Použití minikube

Vzhledem k povaze vývojového prostředí byl k jeho orchestraci použít nástroj Minikube (popsaný v kapitole 2.4.2.10), který zjednodušuje vývoj na lokálním zařízení s nižším výkonem. Dokumentace k nástroji Minikube je k dispozici na oficiálním webu Kubernetes[16], případně i na stránkách Minikube[17]. K interakci s takto vytvořeným clusterem je použit nástroj kubectl popsaný v kapitole 2.4.2.11.

Postup zprovoznění a instalace Minikube vypadá následovně. Nejprve je nutné nainstalovat nějaký hypervizor, v demonstračním řešení byl použit VirtualBox, který se instaluje příkazem:

[15] https://hub.docker.com/
[16] https://kubernetes.io/docs/tutorials/hello-minikube/
[17] https://minikube.sigs.k8s.io/docs/start/

```
sudo apt install virtualbox virtualbox-ext-pack
```

Po úspěšné instalaci VirtualBox je nutné nainstalovat samotný Minikube:

```
wget https://storage.googleapis.com/minikube/releases/latest\
/minikube-linux-amd64
chmod +x minikube-linux-amd64
sudo mv minikube-linux-amd64 /usr/local/bin/minikube
```

Jak již bylo řečeno, k interakci je použit kubectl, který se instaluje následovně:

```
curl -LO https://storage.googleapis.com/kubernetes-release/release\
/‘curl -s https://storage.googleapis.com/kubernetes-release/release\
/stable.txt‘/bin/linux/amd64/kubectl
chmod +x ./kubectl
sudo mv ./kubectl /usr/local/bin/kubectl
```

Poté je nutné spustit Minikube a při jeho spuštění specifikovat použitý hypervisor:

```
minikube start --driver=virtualbox
```

Pro nastavení VirtualBox jako výchozí hypervisor slouží příkaz:

```
minikube config set driver virtualbox
```

Nyní, když minikube běží je možné nasadit aplikaci spolu s jejimi pomocnými komponentami tak, jak je to popsané v následujících kapitolách 4.3.5.5, 4.3.5.6 a 4.3.5.7.

Po nasazení není ještě aplikace přístupná mimo cluster, i když má nakonfigurovaný external service, který běží. Je to z důvodu použití minikube. Nyní je tedy třeba ještě zadat příkaz:

```
minikube service api-module-service
```

A poté příkaz:

```
minikube service api-module-service --url
```

Díky kterému získáme veřejnou adresu.

4.3.5.5 Implementace secret

Secret (teorie popsaná v kapitole 2.4.2.9) je komponentou, pomocí které jsou v demonstračním řešení předávány přístupové údaje k databázi. Součástí údajů jsou i adresy serverů, které by mohly být předávány pomocí configMap popsaném v kapitole 2.4.2.9. V demonstračním řešení je definice této komponenty v souboru `api_module_secret.yaml`. V úvodu definice je opět specifikována verze a o jakou komponentu se jedná:

```
apiVersion: v1
kind: Secret
```

Dále je definováno označení komponenty:

```
metadata:
  name: api-module-secret
```

A na závěr je definován typ a jednotlivé předávané hodnoty v kódování base64:

```
type: Opaque
data:
  api-module-general-username: ujs6S5R6vrc5=
```

Nyní je možné komponentu nasadit pomocí příkazu:

```
kubectl apply -f api_module_secret.yaml
```

Komponentu je nutné nasadit před nasazením deployment, protože se na tuto komponentu odkazuje a nasazení by skončilo chybou.

4.3.5.6 Implementace deployment

Deployment je komponentou, která slouží k automatické replikaci aplikace (kapitola 2.4.2.4). Jeho konfigurace se provádí v `yaml` souboru. V demonstračním řešení konkrétně v souboru `deployment.yaml`. V konfiguračních souborech je třeba brát zřetel na dodržení odsazení a přesné notace souboru. Na prvních řádcích souboru je určena verze a typ komponenty:

```
apiVersion: apps/v1
kind: Deployment
```

Dále se nachází sekce `metadata` která obsahuje identifikátory:

```
metadata:
  name: api-module
  labels:
    app: api-module
```

V další sekci se specifikuje počet replik, identifikátory aplikace a image, ze kterého bude kontejner vytvořen:

```
spec:
  replicas: 3
  selector:
    matchLabels:
      app: api-module
  template:
    metadata:
      labels:
        app: api-module
    spec:
      containers:
      - name: api-module
        image: rouskisroup/api-module:latest
        ports:
        - containerPort: 5000
```

Poslední částí konfigurace je definice proměnných prostředí, která slouží k předávání konfiguračních dat do samotné aplikace. Hodnoty těchto proměnných jsou načítány ze `secret.yaml` souboru, který slouží k bezpečnějšímu předávání těchto hodnot. Proměnných v prostředí je definovaných více, zde je jen příklad jedné z nich:

```
env:
- name: API_MODULE_GENERAL_USERNAME
  valueFrom:
    secretKeyRef:
      name: api-module-secret
      key: api-module-general-username
```

Po vytvoření tohoto souboru je možné deployment nasadit a vytvořit tak pod s aplikací. V tomto souboru je ale ještě konfigurace service, a proto bude nasazení demonstrováno až v následující kapitole 4.3.5.7.

4.3.5.7 Implementace service

Pro přístup k aplikaci zvenku clusteru bylo nutné implementovat external service komponentu. Tato komponenta je stejně jako deployment definována v souboru `deployment.yaml`. Pojmenování souboru a obsah jak deployment, tak i service je lehce matoucí, ale je běžnou praxí definovat tyto dvě komponenty ve stejném souboru. V souboru jsou tyto dvě komponenty odděleny pomocí tří pomlček `---` na samostatném řádku. První řádek definice této komponenty opět obsahuje verzi a typ komponenty:

```
apiVersion: v1
kind: Service
```

Dále je v sekci metadata specifikován identifikátor této komponenty:

```
metadata:
  name: api-module-service
```

Poslední sekcí jsou `specs`, ve které je specifikována aplikace, na kterou se service váže, typ service (LoadBalancer označuje external service) a porty, pod kterými bude aplikace dostupná:

```
spec:
  selector:
    app: api-module
  type: LoadBalancer
  ports:
    - protocol: TCP
      port: 5000
      targetPort: 5000
      nodePort: 30000
```

Ve chvíli, kdy jsou obě komponenty definovány, je možné je nasadit pomocí:

```
kubectl apply -f deployment.yaml
```

Počet požadovaných replik je možné měnit i pomocí kubectl a není nutné nasazovat nový deployment:

```
kubectl scale --replicas=4 deployment/api-module-deployment
```

4.3.6 Postgresql databáze

Součástí řešení je i vytvoření databáze na veřejném aws serveru. Tato kapitola poskytuje stručný návod. Instalace posgresql:

```
sudo sh -c 'echo "deb http://apt.postgresql.org/pub/repos/apt \
$(lsb_release -cs)-pgdg main" > /etc/apt/sources.list.d/pgdg.list'
wget --quiet -O - https://www.postgresql.org/media/keys/ACCC4CF8.asc \
| sudo apt-key add -
sudo apt-get update
sudo apt-get -y install postgresql
sudo ufw allow http
sudo ufw allow https
sudo -u postgres psql
```

Po vytvoření databáze byl problém s přihlášením kvůli neznámému výchozímu heslu. Heslo lze změnit následovně:

```
ALTER USER postgres PASSWORD 'newpwd';
```

Poté je v případě aws nutné nastavit stav portu, na kterém bude tato databáze přístupná následovně: Security -> security groups -> rozkliknout skupinu -> edit inbound rules -> add rule -> type posgreSQL -> adresa 0.0.0.0/0 -> save rules.
Dále změnit konfiguraci na serveru v souboru `pg_hba.conf`:

```
sudo nano /etc/postgresql/14/main/pg_hba.conf
```

Přidáním řádku na konec tohoto souboru:

```
host all all all md5
```

A také v souboru `postgres.conf`:

```
sudo nano /etc/postgresql/14/main/postgresql.conf
```

Změnou řádku:

```
#listen_address='localhost'
```

Na řádek:

```
na listen_address='*'
```

Poslední nutností je restartování služby pomocí:

```
sudo service postgresql restart
```

Nyní by služba měla běžet v pořádku a je možné se k databázi připojit například z lokálního stroje pomocí nástroje pgadmin[18].

[18] https://www.pgadmin.org/

4.4 Vyhodnocení

Výstupem práce je obsáhlý soubor teorie převodu systému do multitenantní podoby a oblastí týkajících se této problematiky. Na tuto část práce byl kladen důraz z důvodu předpokládané nutnosti jejího použití při implementaci navržených řešení. Soubor informací se od těch nyní existujících a veřejně dostupných liší komplexností. Typické použití představených nástrojů se liší od použití v této práci, proto by bylo nutné pro získání veškeré potřebné teorie vyhledávat a spojovat informace z mnoha zdrojů.

V práci jsou poskytnuté návrhy na implementaci možného převodu, nicméně výsledná podoba převedeného systému je velice závislá na aspektech původního systému, a tak se výsledné řešení bude pravděpodobně vždy lišit od toho navrženého. Z tohoto důvodu aplikace poskytuje ono velké množství teorie, kterou vývojář při přizpůsobení návrhu dle specifických potřeb využije.

Součástí práce je i demonstrační řešení, které je zjednodušenou implementací daných návrhů. Problematika, která byla třeba nastudovat k vytvoření této práce je velice komplexní, a proto je zde předvedeno pouze tohle zjednodušené, ale funkční řešení. K částem implementace, které nejsou součástí demonstračního řešení, poskytuje práce alespoň základní informace a vhodné zdroje.

Právě rozšíření demonstračních řešení a jejich rozbor je vhodným možným navázáním na tuto práci.

Kapitola 5

Závěr

Cílem práce bylo analyzovat potřeby multitenantních systému a navrhnout vhodná řešení pro alternativní převod původního jednotenantního systému do nové multitenantní podoby. Předložený postup je zvláštní v tom, že funkcionalita multitenance nebyla zavedena přímou modifikací původního systému, ale bylo toho dosaženo za pomoci nástrojů jako Docker a Kubernetes, které slouží ke kontejnerizaci a následné orchestraci systému. Součástí jsou i další nově vytvořené pomocné moduly, které slouží opět k minimalizaci potřeby modifikace původního kódu.

V první části práce je shrnuta důležitá teorie týkající se multitenantních systémů spolu s principy a základními komponentami použitých nástrojů – Docker a Kubernetes. V závěru této části jsou sepsána a zhodnocena současná řešení týkající se této problematiky a jejich zasazení do kontextu.

Druhá část práce je zaměřena na předložení vhodných návrhů realizace tohoto převodu. Variabilita převáděných systému neumožňuje vytvoření jednoho univerzálního návrhu a jeho okamžité použití. Z tohoto důvodu kapitola obsahuje více možných návrhů, společně se zhodnocením jejich výhod či nevýhod a vhodnými případy použití. Příklady těchto návrhů jsou přesunutí kompletní instance aplikace do samostatných clusterů na sdílený server, vyčlenění sdílených modulů a jejich nasazení na společný server a nebo kompletní přesun aplikace včetně vyčlenění modulů, jejich sdílení a izolace tenantně specifických částí.

Třetí část práce obsahuje implementační detaily k navrženým řešením a demonstrační řešení. Z důvodu složitosti a velké komplexnosti této problematiky je demonstrační řešení pouze jednoduchým předvedením principu těchto převodů.

Při využití této práce, jako návodu k převodu systému, se předpokládá nejprve nastudování problematiky pomocí souboru teorie poskytnutého v první části práce. Poté, již s potřebnými znalostmi, bude analyzován původní převáděný systém a dle poskytnutých návrhů bude vytvořeno řešení, které bude odpovídat specifickým potřebám daného systému. Následně bude tento návrh s pomocí informací ze třetí části práce implementován.

Jak již bylo zmíněno, podoba a aplikace tohoto způsobu převodu bude velice závislá na aspektech původního systému. Obecně lze ale říci, že tento způsob představuje zjednodušení převodu, které souvisí s ušetřením potřebných zdrojů a v některých případech i jedinou možnost pro (částečný) převod systému do multitenantní podoby.

Práce tedy poskytuje teoretické znalosti potřebné k analýze původního systému a specifické úpravě návrhů a také obsahuje demonstrační řešení. Toto řešení je ale jen jednoduchým předvedením principů převodu bez implementace složitějších částí návrhu, či předvedení případných atypických použití. Dalším pokračováním práce by tedy mohlo být právě rozšíření demonstračních implementací.

Literatura

[1] AlJahdali, H., Albatli, A., Garraghan, P., Townend, P., Lau, L. et al. Multi-tenancy in Cloud Computing. In: *2014 IEEE 8th International Symposium on Service Oriented System Engineering*. 2014, s. 344–351. DOI: 10.1109/SOSE.2014.50.

[2] aqua. *Kubernetes: Why Use It, How It Works, Options and Alternatives*. 2020. Dostupné z: https://www.aquasec.com/cloud-native-academy/kubernetes-101.

[3] Authors, T. K. *Don't Panic: Kubernetes and Docker*. 2020. Dostupné z: https://kubernetes.io/blog/2020/12/02/dont-panic-kubernetes-and-docker/.

[4] Bezemer, C.-P. *MULTI-TENANCY*. 2009. Dostupné z: https://multi-tenancy.blogspot.com/.

[5] Boyd, W. *Kubernetes is deprecating Docker: what you need to know*. 2021. Dostupné z: https://acloudguru.com/blog/engineering/kubernetes-is-deprecating-docker-what-you-need-to-know.

[6] Fiaidhi, J., Bojanova, I., Zhang, J. a Zhang, L.-J. *Enforcing Multitenancy for Cloud Computing Environments*. 2012. DOI: 10.1109/MITP.2012.6.

[7] Furda, A., Fidge, C., Zimmermann, O., Kelly, W. a Barros, A. *Migrating Enterprise Legacy Source Code to Microservices: On Multitenancy, Statefulness, and Data Consistency*. 2018. DOI: 10.1109/MS.2017.440134612.

[8] Grygaříková, M. *Docker, Kubernetes a kontejnery. Jak fungují a proč je chtít.* 2019. Dostupné z: https://www.master.cz/blog/docker-kubernetes-kontejnery-jak-funguji-proc-je-chtit/.

[9] Hat, R. *IaaS vs PaaS vs SaaS*. 2020. Dostupné z: https://www.redhat.com/en/topics/cloud-computing/iaas-vs-paas-vs-saas.

[10] Jaffoo, Y. *Choosing Single-Tenancy or Multi-Tenancy Kubernetes Clusters*. 2020. Dostupné z: https://www.appvia.io/blog/single-tenancy-or-multi-tenancy-kubernetes-clusters.

[11] Malik, S. *Kubernetes: multi-tenant vs single tenant clusters?* 2020. Dostupné z: https://www.spectrocloud.com/blog/kubernetes-multi-tenant-vs-single-tenant-clusters/.

[12] Nicole Hiller, N. J. *TechWorld with Nana*. 2020. Dostupné z: https://www.techworld-with-nana.com/.

[13] Pekař, L. *KONTEJNERY A VIRTUALIZACE*. 2019. Dostupné z: https://bonsai-development.cz/clanek/kontejnery-a-virtualizace.

[14] Poulton, N. *The Kubernetes book*. SEPT 2020. United Kingdom: Nigel Poulton, 2019. ISBN 978-1521823637.

[15] Romanyuk, O. *7 Cases When You Should Not Use Docker*. 2019. Dostupné z: https://www.freecodecamp.org/news/7-cases-when-not-to-use-docker/.

[16] Thiry, D. *Kubernetes Multi-Tenancy – A Best Practices Guide*. 2020. Dostupné z: https://loft.sh/blog/kubernetes-multi-tenancy-a-best-practices-guide/.

[17] Turnbull, J. *The Docker Book: Containerization Is the New Virtualization*. V18.09. James Turnbull, 2014. ISBN 9780988820203. Dostupné z: https://books.google.cz/books?id=4xQKBAAAQBAJ.

[18] Vizard, M. *Loft Labs Makes Virtual K8s Clusters More Accessible*. 2021. Dostupné z: https://containerjournal.com/features/loft-labs-makes-virtual-k8s-clusters-more-accessible/.

[19] Weibel, D. *Architecting Kubernetes clusters — how many should you have?* 2020. Dostupné z: https://learnk8s.io/how-many-clusters.