FAKULTA INFORMAČNÍCH TECHNOLOGIÍ ČVUT V PRAZE

Zadání bakalářské práce

Název:	Konfigurační management pro mikroslužby
Student:	Josef Vávra
Vedoucí:	Ing. Martin Komárek
Studijní program:	Informatika
Obor / specializace:	Webové a softwarové inženýrství, zaměření Softwarové inženýrství
Katedra:	Katedra softwarového inženýrství
Platnost zadání:	do konce letního semestru 2022/2023

Pokyny pro vypracování

1. Nastudujte a popište hlavní problémy konfiguračního managementu mikroslužeb v cloud native prostředí. Nastudujte existující řešení. Neopomeňte pokrýt i problematiku secrets managementu.
2. Navrhněte a iterativním způsobem implementujte řešení pokrývající hlavní problémy identifikované v bodu 1.
3. Na jednoduché ukázkové aplikaci průběžně testujte implementované řešení.
4. Porovnejte vámi implementované řešení s již existujícími.

Elektronicky schválil/a Ing. Michal Valenta, Ph.D. dne 16. února 2022 v Praze.

Bakalářská práce

KONFIGURAČNÍ MANAGEMENT PRO MIKROSLUŽBY

Josef Vávra

Fakulta informačních technologií
Katedra softwarového inženýrství
Vedoucí: Ing. Martin Komárek
11. května 2022

Obsah

Seznam obrázků

Seznam výpisů kódu

Děkuji Ing. Martinu Komárkovi za trpělivost a vedení bakalářské práce.

Prohlášení

Prohlašuji, že jsem předloženou práci vypracoval samostatně a že jsem uvedl veškeré použité informační zdroje v souladu s Metodickým pokynem o dodržování etických principů při přípravě vysokoškolských závěrečných prací. Beru na vědomí, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., autorského zákona, ve znění pozdějších předpisů, zejména skutečnost, že České vysoké učení technické v Praze má právo na uzavření licenční smlouvy o užití této práce jako školního díla podle § 60 odst. 1 citovaného zákona.

V Praze dne 11. května 2022 .....

Abstrakt

Tato práce zkoumá přístupy k nasazení aplikace na server. Předmětem tohoto zkoumání je hotová aplikace, která se skládá z několika komponent. Nedochází k žádné implementaci, pouze k popisu zacházení s aplikací po jejím vývoji. K tomu je v práci využita celá řada technologií a nástrojů. U každé z nich je kromě popisu i způsob, jak ji použít. Tento způsob je poté demonstrován. Výsledkem je plně funkční škálovatelná aplikace, která je spuštěna v cloud prostředí a uživatel ji může otevřít v prohlížeči.

Klíčová slova Konfigurace, kontejner, server, nasazení, vyvažnování zátěže, cluster

Abstract

This thesis is researching how to deploy application on server. Subject of this research is already created application, that is composed of multiple components. No implementation is done, there is only description of handling application after it has been developed. In this work was used a lot of technologies and tools. Besides from description of each technology or tool, there is shown a way how to use it. This usage is demonstrated. As a result there is fully functioning scalable application, that is running in cloud envrionment and user can open it in browser.

Keywords Configuration, container, server, deployment, load balancing, cluster

Seznam zkratek

CPU	Central processing unit
RAM	Random-access memory
REST	Representational state transfer
CORS	Cross-Origin Resource Sharing
CLI	Command-line interface
JSON	JavaScript Object Notation
URL	Uniform Resource Locator
GB	gigabyte
AWS	Amazon Web Services
EKS	Elastic Kubernetes Service
ECR	Elastic Container Registry
IAM	Identity and Access Managemen
OIDC	OpenID Connect
DNS	Domain Name System

Kapitola 1
Úvod

Podle The twelve-factor app[1] je konfigurace aplikace vše co se liší mezi prostředími (testovací, produkční, vývojové atd.). To mohou být přístupové údaje k externím službám, přihlašovací údaje k databázím, nebo třeba kanonické názvy hostitelů.

Konfiguraci je možné ukládat jako konstanty v kódu. Je to však vysoce nežádoucí. Jedná se o porušení twelve-factor metodiky. Tato metodika vyžaduje oddělené konfigurace od kódu. I když kód zůstává stejný, konfigurace se většinou mezi nasazeními výrazně odlišuje.

Každá aplikace má alespoň dvě prostředí: testovací a produkční. Každá taková aplikace má pak zhruba čtyři až pět konfiguračních parametrů. Pokud tyto dvě čísla vynásobíme, máme už deset parametrů. Toto číslo roste rychle, podle toho jak se rozšiřuje naše aplikace. Roste ještě rychleji podle toho, jak se zvyšuje počet prostředí (viz. článek o gitOps[2])

Konfiguracím je tedy nutné věnovat pozornost. Podle článku[3] od RedHat jsou nesprávné konfigurace hlavní příčinou bezpečnostních incidentů v kontejnerizovaném Kubernetes-prostředí. Nesprávné konfigurace vedou ke slabému výkonu, nekonzistenci, nefunkčnosti a mají negativní dopad na business logiku a bezpečnost. Pokud jsou nezdokumentované změny prováděny napříč mnoha systémy a aplikacemi, zvyšuje se riziko nestability a prostoje. Manuální řešení této problematiky vyžaduje identifikaci systémů, které potřebují pozornost. Poté rozhodnutí, které kroky je potřeba podniknout k napravení problému a rozhodnout které úkony mají prioritu. Validace správného výsledku je v komplexních systémech příliš složitá. Bez dokumentace kroků, které byly provedeny jsou navíc systémový administrátoři a softwarový vývojáři neschopni rozlišit, který server byl zaktualizován a jaké verze aplikace se kde nacházejí.

Tomu se dá však zabránit. Řešením tohoto problému je konfigurační management. Ten umožňuje definovat nastavení systému. Cílem je udržet aplikaci ve stavu, který je sledovatelný a funkční. Proč je konfigurační management důležitý, je rozebráno ve článku[4] od Atlassian. Konfigurační management podle textu pomáhá vývojářským týmům vytvořit robustní a stabilní systém, který automaticky řídí a monitoruje akutalizace konfiguračních položek. Jako příklad uvádí mikroservisní arhitekturu. Každá mikroslužba má vlastní konfigurační parametry, které jsou použity při inicializaci. Jde jak o hardwarové požadavky jako je CPU nebo RAM, tak softwarové specifikace jako jsou hesla a endpoints. Konfigurační management tedy umožňuje vytvořit, aktualizovat a monitorovat jediný „zdroj pravdy“ tím, že jsou veškeré informace na jednom místě.

Bohužel konfigurační management není žádná zázračná technika, která zadarmo a v krátkém čase přetvaruje libovolný systém na bezvýpadkový produkt, který má bezproblémový release a každý proces, který se odehraje je striktně kontrolován. Implementaci je nutné věnovat prostředky, jak na začátku vývoje, tak i v jejím průběhu. Jak je uvedeno v článku[5] od UpGuard je vyžadující zaškolit současně zaměstnance v oblasti konfiguračních procesů a vyčlenit někoho na stanovení zmíněných procesů a nastavení potřebných nástrojů.

V této práci budu demonstrovat důležitost správného řízení konfigurací. Budu iterativně demonstrovat technologie jako je například docker a kubernetes. Aplikaci vždy sestavím a zprovozním ji takovým způsobem, aby bylo možné ji používat. Budu používat různé přístupy, které na konci zhodnotím.

Na závěr vyberu cloud poskytovatele a celou aplikaci na něj nasadím. Popíšu nejjednodušší postup, jak se dostat s hotovým produktem do funkčního stavu.

V teoretické části práce popíšu best practices konfiguračního managementu v cloud native prostředí a jak přistupovat ke konfiguračním parametrům v aplikacích. Co jsou secrets a jaké jsou možnosti šifrování. Jedna z kapitol se bude věnovat problému několika prostředí, které vytvořím. Paralelně budu aplikovat vyzkoumané postupy na existující aplikaci, naimplementovanou vzoru mikroslužeb definovanou zde[6]. Cílem této práce je možnost demonstrovat migraci mezi vytvořenými prostředími na konkrétním příkladu. Celý proces bude zdokumentován. K práci přistupuji jako k Proof of concept[7]. Čtenáře provedu postupem od naimplementované aplikace k jejímu úspěšnému nasazení, provozu a údržbě.

Kapitola 2

Best practices konfiguračního managementu v cloud native prostředí

Jako v každé jiné softwarové disciplíně i v konfiguračním managementu je nutné dodržovat určité postupy, neboli best practices. Podle textu s názvem Best-practice recommendations configuration management od Sun Services[8] jsou jedněmi z hlavních doporučení tyto: udržování kontroly, sledování správných konfiguračních položek a vytvoření repozitáře.

Pod pojmem udržování kontroly je myšleno, že je nutné pečlivě hlídat konfigurační položky a udržovat systém aktuální. V momentě, kdy je systém zastaralý, přestává být užitečný.

Jak už bylo zmíněno v předchozím bodě, je další doporučení sledování správných konfiguračních položek. Pokud pracujeme s komplexním systémem, je velice pravděpodobné, že při potřebě mít kontrolu nad každým detailem, budeme zahlceni. Je proto důležité sledovat ty položky, které je potřeba a u kterých je to možné.

Vytvoření repozitáře je nezbytné pro správný chod celého systému. Pokud nemáme kam ukládat informace o konfiguračních položkách, nebude systém udržitlený.

Myšlenku udržování kontroly potvrzuje kniha Configuration management best practices[9]. Autoři v ní tvrdí, že cílem dobrého konfiguračního managementu je, mít jednoduše identifikovatelný veškerý kód, který se dostal do produkčního prostředí a nazývají to konfigurační audit.

Identifikaci stanovuje microfocus[10] ve druhém kroku v procesu při tvorbě produktu. Prvním krokem je podle této stránky plánování, které spočívá ve stanovení rozsahu a rozdělení rolí a delegaci povinností. Poté následuje zmíněná identifikace. Ta spočívá v určení a registraci konfiguračních položek a zaznamenání jejich vzájemných vztahů. Následující krok je řízení a definují se v něm stavy aplikace před modifikací a po modifikaci. Když se pak provede změna, je možné ověřit, zda je aplikace ve správném stavu. Čtvrtým krokem je reportování. Report by měl obsahovat identifikaci položky, její stav, verzi a příslušnou dokumentaci. Musí být přesný a aktuální. Posledním krokem je verifikace a audit. Na něj padá zodpovědnost za ověření, že každá informace v celém procesu je pravdivá a že veškeré potřebné položky jsou identifikovány a sledovány.

Kapitola 3
Seznámení s aplikací

Nejsem autorem naimplementované aplikace. Jak už název Seat Reservation Demo napovídá, jedná se o simulaci aplikace, která slouží k objednávání sedadel ve vlaku. Aplikace se skládá ze čtyř komponent. Názvy těchto komponent jsou demo-app-synchronous-fe (dále jen FE), demo-app-synchronous-api (dále jen API), demo-app-synchronous-bl (dále jen BL) a mail-sender-sync (dále jen MAIL). Komunikace mezi různými částmi kódu narozdíl od monolitické architektury neprobíhá provoláváním metod a funkcí, ale přes endpoints. Endpoint je adresa, na které je možné v daném kontextu dosáhnout na požadovanou komponentu.

Kromě FE jsou všechny komponenty napsané v Javě s použitím technologie Spring. To je framework, který usnadňuje práci s různým použitím Java aplikací. Hlavní podpora přichází při vývoji webových aplikací[11].

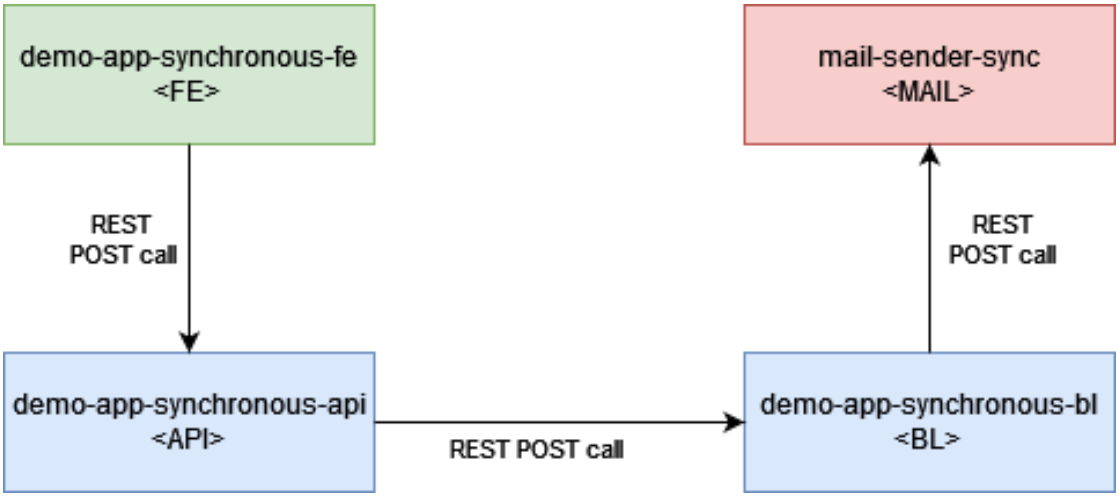
Každá jednotlivá komponenta je sestavena za pomoci nástroje maven[12], který na vstupu přijímá soubor s názvem pom ve formátu xml. Ten na základě příkazu vytvoří spustitelný jar soubor. Ten budu v průběhu práce několikrát používat. FE komponenta je React aplikace. To je javascript knihovna, která slouží k sestavení uživatelských rozhraní[13].

Sestavuje se za pomoci CLI nástroje npm[14], což je zkratka pro Node Package Manager. Tento nástroj vytvoří build adresář, který obsahuje sestavenou komponentu. Aby celá aplikace správně fungovala, budu s tímto adresářem pracovat.

Proces objednávky vypadá následovně. Uživatel vyplní formulář, který mu dodá komponenta FE. Po korektním (formulář kontroluje zda je mailová adresa validní) vyplnění všech údajů uživatel stiskne tlačítko 'Submit'. FE komponenta provede POST REST volání komponenty API. Tato komponeta zaloguje a provolá BL. BL zapíše do databáze a pak přes další POST REST volání zavolá MAIL. Tato poslední komponenta odešle mail obsahující shrnutí objednávky na uvedenou adresu. Nejlépe vztahy, které mezi s sebou v aplikaci jsou popisuje tento obrázek3.1.

3.1 Konfigurace pro lokální spuštění

Každá aplikace má vlastní konfigurační soubor, který je nutné zaktualizovat, pokud chci aby aplikace běžela alespoň v lokálním prostředí. Tento soubor se nachází ve složce codenow/config. Pro Java komponenty je to vždy soubor application.yaml. FE komponenta má soubor runtime-configuration.js. Každý konfigurační soubor (až na MAIL soubor, který se nachází na konci řetězu volání) obsahuje endpoint, který odkazuje na další komponenty. U FE je to dokonce jedinný údaj. Uvedený endpoint musí být vždy stejný jako adresa, na které je spuštěna odkazovaná komponenta. Konfigurace pro lokální spuštění a jeho užití tedy může u všech komponent vypadat následovně: V FE složce upravím codenow/config/runtime-configuration.js a nastavím hodnotu na: `BACKEND_URL: localhost:8081`



Obrázek 3.1 demo-app diagram

```
java -Dserver.port=8081 \
-Dspring.config.additional-location=file:/codenow/config \
-Dlogging.config=classpath:logback-codenow.xml \
-jar demo-app-synchronous-api-1.0.0-SNAPSHOT.jar
```

Výpis kódu 1 Spuštění jar souboru

Následně v terminálu spustím `npm run build` a pak `npm run start`. Pro API musím změnit v codenow/config/application.yaml hodnotu `port` na 8081 a `components.backend.url` na 8082. Pak spustím v terminálu `mvn clean install`. V nově vytvořené složce target spustím jar soubor s názvem demo-app-synchronous-api-1.0.0-SNAPSHOT.jar příkazem, který je uvedený v Dockerfile1. Postup pro BL bude prakticky stejný až na rozdílné hodnoty v portech. Port v konfiguračním souboru nastavím na 8082 a `components.mail-sender.url` nastavím na 8083 a sestavím za pomocí mavenu. Příkaz pro spuštění je stejný jako pro api s rozdílnými parametrem server.port, který je samozřejmě 8082 a název jar souboru. Nakonec sestavím MAIL s hodnotou `server.port` nastavenou na 8083. Opět sestavím a dotřetice použiji stejný příkaz s parametrem `server.port` nastaveným na 8083. V prohlížeči otevřu localhost:3000, vyplním údaje a stisknu tlačítko submit. Na uvedenou mailovou adresu dorazí shrnutí objednávky, což znamená, že aplikace je nastavena a spuštěna správně a požadavek postupně prošel přes FE, API, BL a nakonec MAIL, který je za zasílání mailů zodpovědný.

3.2 Opravy aplikace

API komponenta obsahuje drobnou chybu, která zůstává do jejího běhu nezjištěná. Třída `ReservationController` obsahující POST mapování pořadavku na adrese reservation, totiž nepřijímá v prohlížečí Mozilla Firefox CORS požadavek.[15]

Resource sharing je limitován ze strany prohlížeče pro požadavky, které směřují na více zdrojů. Mezi anotace třídy musím přidat tento kus kódu, abych službu mohl provolávat `@CrossOrigin(origins = "*", allowedHeaders = "*")`. Pro jednoduchost nastavuji hlavičky i zdroje na všechny možné. Následující příkaz mi ve složce target vytvoří jar soubor, který obsahuje opravenou třídu.
`mvn clean install`

Kapitola 4

Sestavení komponent

Aplikace je hotová a připravena k nasazení. Každou jednu komponentu je možné sestavit a výsledný soubor(jar v případě Java komponenty a celá složka build v případě React komponenty) lze nahrát na server. Tento proces je však zdlouhavý a vysoce náchylný k chybám. Kdyby se jednalo o košatější aplikaci, expertiza osoby, která je zodpovědná za nasazení aplikace, by musela minimálně pokrývat všechny technologie, které jsou v aplikaci použité.

Samozřejmě že tuto aktivitu by bylo možné delegovat na autory kódu. Pokud by se ale v den release kdokoliv nedostavil, nastává problém. Na ten existuje odpověď, kterou je kontejnerizace aplikace. Kontejnerizace spočívá v uzavření vytvořeného kusu kódu do izolovaného prostředí, které je nejmenší možné. Nejmenším možným prostředím je myšleno takové, jenž obsahuje pouze takový balík funkcí, který je nezbytně nutný k běhu aplikace. Dále musí být samostatné, což znamená že v něm není spuštěn paralelně žádný kód nesouvisející s danou funkcionalitou kontejneru.

V poslední řadě musí být kontejner samozřejmě spustitelný. To znamená, že se nejedná pouze o nic nedělající nahrané soubory. Je možné sestavit takovou image, která nic nedělá, nebo se hned po provedení nějakého příkazu ukončí. Takové image ale nejsou pro tuto práci zajímavé. Výhodou kontejnerizovaných aplikací je izolace kódu od okolních vlivů, což přináší další vrstvu bezpečnosti.[16]

4.1 Docker a jeho role v aplikaci

Nejrozšířejnějším nástrojem, jenž umožnuje kontejnerizaci je Docker. Jeho použití a jeho schopnosti řízení konfigurací se budu věnovat v této kapitole.

Kontejnerizace dosahuje tím, že vytvoří virtuální prostor, ve kterém proběhne sestavení a následné spuštění kódu. Nenastane tedy problém, se kterým se potýká mnoho programátorů. Tím je kód, který byl zkompilován na dvou různých strojích a výsledkem je kód, jenž se na každém stroji chová jinak. Vývojář si totiž už při vytváření aplikce může vyzkoušet, jak bude spuštěna v jakémkoliv prostředí do kterého bude následovně nasazena, a to včetně produkčního.

Docker spouští příkazy definované v souboru zvaném Dockerfile. Pokud všechny proběhnou bez problémů, Docker vytvoří tzv. image. Docker je schopný na základě podložené image vytvořit a spustit kontejner, což je už zmíněný virtuální stroj s aplikací.

Kontejnerizace aplikace se stává normou v každodenním životě DevOps inženýra. Alternativou k nástroji Docker je Podman. Ten má velice podobné CLI. Mnoho příkazů je hlavně ze začátku používání prakticky stejné. Zatímco Docker veškeré funkce poskytuje sám, Podman se spoléhá na jiné nástroje (například Buildah na sestavení kontejneru). Hlavní rozdíl je však v tom co se odehrává pod pokličkou. Docker potřebuje, aby na stroji byl spuštěn Docker daemon, zatímco architektura Podman je tzv. daemon-less. Podman je navíc považován za bezpečnější

nástroj, protože umozňuje rootless kontejnery. Docker daemon má root priviliegia a tak je zneužíván útočníky jako hlavní průchod do systému. V této práci budu využívat hlavně Docker, jelikož s ním mám více zkušeností.[17]

Aplikaci je možné sestavit pomocí Docker a spustit. K tomu je však zapotřebí několik úkonů. Je nutné ve složce každé komponenty spustit Docker build s příslušným tagem. Tag je proměnlivý odkaz na příslušnou Docker image. Funguje velmi podobně jako reference na větve v gitu. Jedna image může mít více tagů. Při stahování image například z Docker hub, je vždy dobrým zvykem specifikovat i verzi. Pokud verze není specifikována docker stáhne nejnovější možnou, tzv. latest. Tady může nastat problém, když je našim cílem konstatní funkcionalita kontejneru. Může se stát, že v nejnovější verzi existuje bug, který negativně ovlivní chod naší aplikace, nebo je naopak odstraněna nějaká funkcionalita, na kterou aplikace spoléhala. Proto je v případě produkčního kódu dobré u každé image specifikovat i její verzi. Teď už ale zpět k sestavení aplikace.[18]

4.1.1 Použití dockeru

Ponechám takové konfigurace, které byly použity při lokálním spuštění. Každá složka už obsahuje Dockerfile, ve kterém je nadefinován sestavovací postup, budu tedy pokračovat následovně: V FE složce spustím

`docker build . -t demo-fe-local`,

v API

`docker build . -t demo-api-local`,

v BL

`docker build . -t demo-bl-local`

a nakonec v MAIL

`docker build . -t demo-mail-local`.

Že jsou všechny komponenty správně sestaveny a otagovány otestuji příkazem

`docker images`.

Pak už každou z nich spustím. Spuštění však potřebuje další parametry. Jak už jsem zmínil, každý kontejner je izolovaný. To znamená, že pokud se komponenta, která v něm běží pokusí provolat jinou koponentu například na adrese localhost:8082, dostane zpět connection refused. Proto je nutné v každém kontejneru otevřít příslušný port, na kterém komponenta očekává provoz.

FE image spustím příkazem `docker run -it demo-fe-local -p 3000:80` přepínač p očekává na vstupu port na kterém poslouchá kontejner a port ze kterého má z lokálního stroje přesměrovat tok dat ve formátu

`{port kontejneru}:{vnější port}`

Z toho tedy plyne, že je možné ponechat výchozí konfigurace portu na kterém aplikace poslouchá (ne ten na kterém očekává následující komponentu). Pro jednoduchost ale ponechávám konfigurace tak, jak byly použity ve scénáři lokálního spuštění.

Pokračuji ve spouštění `docker run -it demo-api-local -p 8081:8081` pro API,

`docker run -it demo-bl-local -p 8082:8082` pro BL a

`docker run -it demo-mail-local -p 8083:8083` pro MAIL. Že jsou všechny kontejnery spuštěny otestuji příkazem `docker ps`, kde vidím čtyři spuštěné kontejnery, jejich názvy a stručné detaily ke každému z nich. U žádného jsem nespecifikoval jméno, tak jim docker dosadil nějaké náhodně vygenerované. Nyní spustím prohlížeč a dosadím adresu localhost. Na portu 3000 nic není, jelikož jsem přesměroval kontejner na port 80. Vyplním údaje, stisknu potvrzující tlačítko a opět obdržím mail na uvedené mailové adrese. Tím znovu ověřím, že celá aplikace funguje tak, jak má.

```
demo-fe-compose:
  image: demo-fe-compose
```

Výpis kódu 2 Definice služby v docker compose

```
components:
  backend:
    url: http://demo-bl-compose
```

Výpis kódu 3 Konfigurace api pro docker compose

4.1.2 Konfigurace pro lokální spuštění

Je možné každý kontejner samostatně spustit přes příkazovou řádku se všemi náležitými parametry. Tento postup je však manuální a tudíž vysoce náchylný k lidské chybě. Nehledě na to, že pokud si příkaz neuložíme do souboru, v budoucnu jednoduše zapomeneme co vše je ke spuštění zapotřebí a nás pak čeká dlouhé prohledávání souboru `bash_history`. Tuto problematiku řeší funkcionalita docker compose(nabízí ji i podman). Docker compose je nástroj, který slouží k definici a běhu multi-clusterových aplikací. Nabízí dokonce izolaci prostředí.

Na vstupu získá YAML soubor, který obsahuje instrukce s jedním příkazem. Pak nastartuje všechny nadefinované kontejnery s jejich parametry v jedné společné síti. Síť je důležitá, protože pokud se v ní nachází více kontejnerů, je možné se mezi nimi odkazovat za pomocí aliasů. To ulehčí definici endpoints mezi jednotlivými službami. Není nutné složitě vyhledávat na jaké IP adrese se bude služba nacházet, stačí použít název kontejneru a port a docker se o zbytek postará sám. Zmíněný port je samozřejmě nutné v odkazované službě zpřístupnit. Je důležité neplést si název image a název kontejneru. V předchozím případu jsem názvy kontejnerů nemusel specifikovat, jelikož pro běh aplikace neměly žádný význam, nyní už je ale jejich definování důležité.

Velkou výhodou docker-compose je právě možnost zachovat veškeré nastavení pro spuštění celé aplikace v jednom souboru, který je možné verzovat. Pokud se jedná o projekt, do kterého přispívá více lidí, účastníkovi stačí naklonovat respozitář a spustit docker compose. Není nutné složitě konfigurovat a spouštět kontejner po kontejneru, což ušetří mnoho práce.

4.1.3 Použití docker compose

Teď vytvořím samotný docker compose soubor. Podle dokumentace[19] specifikuji na prvním řádku verzi(jedná se o verzi docker compose, nikoliv verzi spouštěné služby). Na dalším pak ve formátu YAML začnu definovat services. Postupně uvedu všechny komponenty každou na vlastním řádku. Pro jednoduchost pojmenuji stejně název kontejneru a image kterou používá. Pro FE bude záznam vypadat následovně: 2

Každá komponenta bude mít vlastní záznam se svým vlastním jménem a vlastní image.

Nyní je nutné znovu sestavit images s novými konfiguracemi a s novým jménem. Do komponenty FE nedosadím zatím nic. Proč nedosazuji žádnou hodnotu vysvětlím později.

V API nastavím: 4 V BL nastavím

MAIL zůstává nezměněn.

V docker compose souboru takto nadefinuji porty, které bude používat FE: 5

Může se zdát, že docker compose je připravený ke spuštění, ale ještě nejsme zdaleka u konce. Jak už jsem zmínil docker compose vytvoří síť, ve které je možné dotazovat se na ostatní kontejnery. Tato síť však funguje pouze pro ně. I když je celá aplikace spuštěna na mém lokálním stroji,

```
components:
  mail-sender:
    url: http://demo-mail-compose
```

Výpis kódu 4 Konfigurace api pro docker compose

```
ports:
  - "8080:3000"
```

Výpis kódu 5 Porty FE komponenty v docker compose

nelze použít například curl k dotázání se na jakýkoliv spuštěný kontejner. Příkaz `curl http://demo-bl-compose` vrátí connection refused. Tento problém bohužel znemožňuje FE poslat POST request do API, jelikož i když je celá komponenta sestavena v docker image, vyhodnocení a dotaz probíhá na straně klienta, tedy v prohlížeči. Komponenta FE totiž nedělá nic jiného než že poskytne klientovi obsah svojí složky. Veškerý kód je spouštěn v prohlížeči.

Řešení tohoto problému lze vyčíst z blogu fragaria.cz [20]

Kromě portu FE tedy navíc musím otevřít port i pro API následovně: 6

Z tohoto důvodu jsem s FE konfigurací nic nedělal. V docker compose souboru takto nadefinuji hodnotu proměnné, kterou budu chtít do konfigurace dosadit. 7

Docker compose tedy vytvoří proměnnou prostředí. Pro dosazení do souboru runtime-configuration však ještě potřebuji upravit `package.json` a vytvořit skript. JSON soubor má přesně nadefinováno, jak spouštět danou komponentu. Při startu nahradí konfiguraci ve složce public za tu, jenž se nachází v `codenow/config`. Tomu musím zabránit, protože se nahradí název proměnné a za proměnnou se nic nedosadí. Prohlížeč by se pak pokoušel provolat POST request na adrese `${API_URL}`.

Úryvek z package.json tedy bude vypadat následovně: 8

Zmizí z něj příkaz, který kopíruje konfiguraci z codenow složky do public složky, kdyby tam zůstal, tak by přepsal připravenou konfiguraci. K tomu přibyde skript, který konfiguraci správně nahradí: 9

Skript v Dockerfile zkopíruji do kontejneru pod názvem copy-runtime-configuration.sh, abych s ním mohl pracovat. Tento skript se však sám od sebe nespustí. Docker má však zabudovanou funkcionalitu, která dovoluje spustit příkaz bezprostředně po nastartování kontejneru. Tuto funkcionalitu můžeme nadefinovat v Dockerfile klíčovým slovem CMD a specifikováním parametrů. FE image vychází z nginx image, která má na starost nastartování a poskytnutí html serveru. Pokud bych do Dockerfile jednoduše napsal `CMD ["./copy-runtime-configuration.sh"]`, přepsal bych tím nginx CMD příkaz, který startuje server. Vytvořím proto další skript run-all.sh, jenž spustí předchozí skript a zároveň spustí nginx server. 10

Ten pak stejně jako předchozí skript zkopíruji do kontejneru příkazem COPY a pak ho spustím ve fázi CMD. Celou image znovu sestavím. FE komponenta je připravena ke startu. Poté co bude sestavena a já otevřu localhost:8080, obdržím formulář, který opět vyplním a stisknutím Submit provolám API z prohlížeče. Tato komponenta, však zatím nemá otevřený port k vnější komunikaci. Musím ho tedy přidat do docker-compose stejně jako jsem ho přidal pro FE. Takto

```
ports:
  - "8090:80"
```

Výpis kódu 6 Porty API komponenty v docker compose

```
environment:
  - API_URL=http://localhost:8090
```

Výpis kódu 7 FE environment v docker compose

```
"scripts": {
  "start": "react-scripts start",
...
```

Výpis kódu 8 FE package.json úprava

```
#!/bin/bash
echo "window.env = {
    BACKEND_URL: \"$API_URL\"
};" > ./codenow/config/runtime-configuration.js
```

Výpis kódu 9 FE skript pro nahrazení runtime-configuration.js

```
echo "window.env = {
        BACKEND_URL: \"$API_URL\"
};" > /runtime-configuration.js

nginx -g 'daemon off;'
```

Výpis kódu 10 run-all skript

```
version: "3.9"
services:
  demo-fe-compose:
    image: demo-fe-compose
    ports:
    - "8080:3000"
    environment:
    - API_URL=http://localhost:8090
  demo-api-compose:
    image: demo-api-compose-cors
    ports:
    - "8090:80"
  demo-bl-compose:
    image: demo-bl-compose
  demo-mail-compose:
    image: demo-mail-compose
```

Výpis kódu 11 docker-compose.yaml

bude vypadat celkový finální docker-compose.yaml: 11

Celou aplikaci spustím příkazem:

`docker-compose -f docker-compose.yaml up`

Ověřím příkazem `docker ps`, zda jsou všechny komponenty spuštěny a pak otevřu v prohlížeči adresu http://localhost:8080. Opět se setkávám s formulářem, jenž vyplním a obdržím mail.

4.1.4 Docker compose pro více prostředí

Aplikace běžící na lokálním stroji je sice dobrá k ověření funkčnosti, ale jak postupovat, pokud ji chci rozdistribuovat na více prostředí? Přístupů je hned několik.[21]

Jak je zmíněno v článku , první z nich je nakopírování souboru docker-compose.yaml a instalace docker-compose ve všech prostředí. Hlavní nevýhoda tohoto řešení je, že pokud dojde ke změně souboru, je nutné znovu provést zkopírování do všech prostředí, což je jednak pracné a jednak náchylné k chybám. Dalším možným řešením je konfigurace `DOCKER_HOST`. Tento přístup se hodí, pokud uživatel nasazuje pouze na jedno prostředí. V tomto přístupu se totiž do systémové proměnné ukládá adresa vzdáleného hostitele. Při více prostředích se pak musí hodnota měnit, což je přinejmenším nepříjemné. Posledním přístupem a zdánlivě nejlepším je použití docker contextu.

Docker context[22] zjednodušuje používání docker CLI tím, že umožňuje přepínat mezi prostředími, na které chceme nasazovat aplikaci. Jednoduše lze vytvořit údaj, za použití příkazu `docker context create` s požadovanými údaji jako je endpoint, název a orchestrační nástroj, který se používá v daném kontextu. Uživatel mezi kontexy přepíná tímto příkazem

`docker context use <context-name>`

4.1.5 Různé proměnné pro různá prostředí

V případě více prostředí se dostávám do situace, kdy musím pro každé zvlášť upravovat konfigurační soubor. Jak už jsem zmínil v úvodu počet konfigurací bude prudce stoupat s počtem prostředí. Držet odděleně konfigurační soubor pro každé prostředí se sice nabízí jako řešení, ale pokud budu potřebovat zaktualizovat nasazení, budu muset zaktualizovat několik souborů stejnou změnou. Lepší způsob je použít takzvané env-file. To je přepínač docker-compose, který

```
API_URL_VAR=http://localhost:8090
API_PORT="8090"
```

Výpis kódu 12 .env.dev soubor pro docker compose

```
... // FE
environment:
- API_URL=${API_URL_VAR}
... // API
ports:
- "${API_PORT}:80"
...
```

Výpis kódu 13 Úprava docker-compose souboru s env proměnnými

umožňuje vložit na vstup soubor s proměnnými, které se pak propíší do samotného docker-compose.yaml. Stačí z yaml souboru odebrat pevně zakódované hodnoty a vložit je do připravených env souborů. Například pro dev prostředí může soubor vypadat takto: 12

V docker-compose.yaml stačí upravit řádky pro API a FE následovně: 13

Aplikaci pak spustím tímto příkazem: 14

Aplikace tedy běží na lokálním prostředí. Nastartování aplikace na jiném hostiteli, lze s použitím přepínače H.

```
docker-compose -f docker_compose.yaml --env-file .env.dev up
```

Výpis kódu 14 Příkaz pro spuštění zaktualizovaného docker-compose

Kapitola 5
Orchestrace komponent

I když je možné s komponentami manipulovat víceméně manuálně, automatizace některých částí je nezbytnou součástí každé správné aplikace. Distribuce, monitoring a udržení kontejnerů při životě jsou věci, které ani manuálně uhlídat nelze. Zde tedy přichází na řadu Kubernetes. Kubernetes(někdy nazývaný k8s kvůli 8 písmenům mezi k a s) je orchestrační nástroj, jenž umožňuje konfiguraci a automatizaci kontejnerizovaných aplikací. Když aplikace běží v libovolném prostředí, nezřídka se stává, že v některém z kontejnerů dojde k defektu. Pokud jeden z kontejnerů přestane odpovídat, dochází k nefunkčnosti celé aplikace. Kubernetes zajišťuje, že bude docházet k minimálnímu downtime. Nástroj sám totiž zajišťuje restartování spadnutých komponent. Důležitou složkou funkcí je automatizace, která ulehčuje nasazování na clustery. Stejně tak poskytuje možnost škálování aplikace. Pokud některá z komponent(například BL) bude v produkci fungovat jako takzvaný "bottle neck"celé aplikace, je možné provést škálování a nechat běžet paralelně více stejných kontejnerů. Mezi volání kontejnerů pak rozhoduje Loadbalancer, jenž lze v kubernetes nadefinovat.

K lokálnímu použití Kubernetes slouží nástroj minikube. Ten slouží tak, že vytvoří jeden node cluster ve virtuálním stroji, na který je možné nasazovat Pods. Pod[23] je nejmenší možný a nejzákladnejší objekt, se kterým dovede Kubernetes manipulovat. Obsahuje jeden nebo více kontejnerů. I když jich obsahujie více, jak bude demonstrováno později v této kapitole, je s nimi zacházeno jako s jednou entitou. Sdílí společně prostředky, jenž jim Pod poskytne. Každý Pod dostane v rámci clusteru přidělenou unikátni IP adresu. Kontejnery v Podu sdílí stejnou síť, takže mohou bez obtíží jeden s druhým komunikovat. K Podu je možné připojit i takzvané Volumes. To bude předmětem pozdější fáze této práce.

5.1 Minikube jako simulátor clusteru

Minikube nastartuji příkazem `minikube start`. Minikube umožňuje konfiguraci ovladače. Tento ovladač zajišťuje komunikaci s docker-machine. Výchozí nastavení používá ovladač přímo od dockeru. Uživatelé podmanu dají přednost podman ovladači. Pro tuto práci budu používat hyperv ovladač, jelikož pracuji ve Windows. Prostředí do něhož budu nasazovat je tedy připravené. Ještě spustím a nechám běžet příkaz `minikube dashboard`, jenž mi poskytne grafické rozhraní nad celým clusterem.

5.1.1 Kubernetes a definování deployment

Kubernetes má vlastní terminálový nástroj kubectl. Kubectl umí konzumovat soubory s definicemi služeb a nasazení. Jedno takové vytvořím, protože budu očekávat, že před spuštěním

```
D:\workspace\skola\dokumentace_semestralky\minikube usecase>kubectl get svc
NAME                     TYPE           CLUSTER-IP     EXTERNAL-IP    PORT(S)                       AGE
demo-app-single-service  LoadBalancer   10.98.97.196   10.98.97.196   3000:31000/TCP,8081:31081/TCP 178m
kubernetes               ClusterIP      10.96.0.1      <none>         443/TCP                       23h
```

Obrázek 5.1 Service detail z kubectl

Deployment budu upravovat docker image, použiji v definici jiné názvy. To dělám proto, že chci zachovat funkční image nedotčené a testování nových procesů udržovat separátně. Takto bude vypadat definice: 15

Z Deployment lze vyčíst že opět zpřístupňuji porty FE a API mimo cluster, aby je bylo možné provolat. V API a BL nově nastavuji proměnné prostředí, které budou sloužit ke komunikaci. Zpátky k jednotlivým komponentám. V FE není potřeba nic měnit, ale v rámci konzistence provolám následující příkaz, abych tím vytvořil novou image, jelikož chci oddělit jednotlivé obrazy pro různá použití.

`docker build . -t demo-fe-single-deploy`

V API, BL a MAIL se už objeví několik změn. Předně upravím konfigurační soubor application.yaml u API a BL a na místo původních endpointů vložím proměnné, které definuji v deployment.yaml. Jejich konfigrace tedy budou vypadat následovně: 16

Všechny java komponenty musí mít správně nastavený port v konfiguraci a v Dockerfile, tak aby korespondoval s nastavením uvedeným v deployment.yaml.

V API tedy přepíšu configurační soubor na `port: ${SERVER_PORT:8081}`

V BL `port: ${SERVER_PORT:8082}`

V MAIL `port: ${SERVER_PORT:8083}`

Docker file v API bude vypadat následovně: 18

Docker file v BL bude vypadat následovně: 19

Docker file v MAIL bude vypadat následovně: 20

Všechny komponenty sestavím v dockeru s příslušnými názvy uvedenými v deployment.yaml. Nyní nastává problém viditelnosti docker image v kubectl. Kubectl totiž nebere v potaz lokálně sestavené images, ale kontroluje rovnou vzdálenéregistry, které má nadefinované. Na nově sestavené komponenty samozřejmě vůbec nevidí a tak by nasazení na minikube skončilo s chybou. Mám dvě možnosti, jak tento problém vyřešit. První z nich je lokálně spustit vlastní registr, do kterého můžu svoji tvorbu poslat. Druhá možnost je zaslat vše do Docker hub. Docker hub je online registr, jenž slouží k uchování a verzování images, něco jako Github pro gitové repozitáře. Druhá možnost je rychlejší, takže akorát otaguji sestavené komponenty svým přihlašovacím jménem a nad každým tagem spustím docker push následovně:

`docker push josefvavra/demo-fe-single-deploy`

Poté můžu spustit nad deployment souborem:

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

Deployment je vytvořený, můžu ověřit přes:

`kubectl get services`

Ověření je možné i v dashboard, kde naleznu Deployment Pod záložkou Pods. Služba nyní běží, ale zatím není přístupná jinde než v samotném clusteru. Službu vystavím příkazem:

`kubectl expose deployment demo-app-single-deployment --type=LoadBalancer`

Použití LoadBalanceru vysvětlím v pozdější kapitole.

`kubectl get svc` ukáže následující detail

5.1

Porty jsou tedy přiřazeny jak FE tak API. EXTERNAL-IP však obashuje hodnotu `<pending>`. To je způsobené tím, že minikube nemá momentálně běžící load balancer. Ten budu muset manuálně nastartovat. V novém terminálu spustím `minikube tunnel`. Což kromě startu zobrazí i v terminálu url, na které je load balancer spuštěn.

5.2

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: demo-app-single-deployment
spec:
  selector:
    matchLabels:
      app: demo-app
  replicas: 1
  template:
    metadata:
      labels:
        app: demo-app
    spec:
      containers:
      - name: demo-fe-single-deploy
        image: josefvavra/demo-fe-single-deploy
        ports:
        - containerPort: 3000
        env:
        - name: API_URL
          value: "http://localhost:8081"
      - name: demo-api-single-deploy
        image: josefvavra/demo-api-single-deploy
        ports:
        - containerPort: 8081
        env:
        - name: DEMO_BL_ENDPOINT
          value: "http://localhost:8082"
      - name: demo-bl-single-deploy
        image: josefvavra/demo-bl-single-deploy
        env:
        - name: DEMO_MAIL_ENDPOINT
          value: "http://localhost:8083"
      - name: demo-mail-single-deploy
        image: josefvavra/demo-mail-single-deploy
```

Výpis kódu 15 demo-single-deployment.yaml

```
components:
  backend:
    url: ${DEMO_BL_ENDPOINT}
```

Výpis kódu 16 API konfigurace pro single deployment

```
components:
  mail-sender:
    url: ${DEMO_MAIL_ENDPOINT}
```

Výpis kódu 17 BL konfigurace pro single deployment

```
...
EXPOSE 8081
CMD java ${JAVA_OPTS} \
-Dserver.port=8081 \
-Dspring.config.additional-location=file:/codenow/config/ \
-Dlogging.config=classpath:logback-codenow.xml \
-jar app.jar
```

Výpis kódu 18 API Dockerfile pro single deployment

```
...
EXPOSE 8082
CMD java ${JAVA_OPTS} \
-Dserver.port=8082 \
-Dspring.config.additional-location=file:/codenow/config/ \
-Dlogging.config=classpath:logback-codenow.xml \
-jar app.jar
```

Výpis kódu 19 BL Dockerfile pro single deployment

```
...
EXPOSE 8083
CMD java ${JAVA_OPTS} \
-Dserver.port=8083 \
-Dspring.config.additional-location=file:/codenow/config/ \
-Dlogging.config=classpath:logback-codenow.xml \
-jar app.jar
```

Výpis kódu 20 MAIL Dockerfile pro single deployment

```
Status:
        machine: minikube
        pid: 14368
        route: 10.96.0.0/12 -> 172.24.8.150
        minikube: Running
        services: [demo-app-single-deployment]
    errors:
                minikube: no errors
                router: no errors
                loadbalancer emulator: no errors
```

Obrázek 5.2 Detail běžícího load balanceru

```
apiVersion: v1
kind: Service
metadata:
  name: demo-app-single-service
spec:
  type: LoadBalancer
  selector:
    app: demo-app-single
  ports:
  - name: demo-fe-single-deploy
    protocol: TCP
    port: 3000
    targetPort: 3000
    nodePort: 31000
  - name: demo-api-single-deploy
    protocol: TCP
    port: 8081
    targetPort: 8081
    nodePort: 31081
```

Výpis kódu 21 Service definice pro single deploy

Nyní vložím adresu do prohlížeče a použiji port, který byl uveden v detailu servis. http://172.24.8.150:31213/

FE mi správně naservírovalo formulář který vyplním a potvrdím. Obdržím však chybovou hlášku. Podle logu je to způsobeno tím, že jsem předal chybnou adresu API v parametru.

5.1.2 Řešení nastavení nesprávného endpointu v definici

Dostal se do stejné situace jako v předchozím příkladě. Komunikace v rámci Pod by mezi API a FE by s uvedenými endpointy fungovala, ale opět je požadavek odeslán z prohlížeče a nikoliv z clusteru. Abych vše opravil, smažu službu příkazem:

`kubectl delete service demo-app-single-deployment`

Nastává problém, jak předat správný endpoint FE. Url část adresy už znám, jednak mi ji sděluje spuštěný tunel, a pokud by tunel adresu neukazoval, mohu ji zjistit použitím příkazu:

`minikube ip`

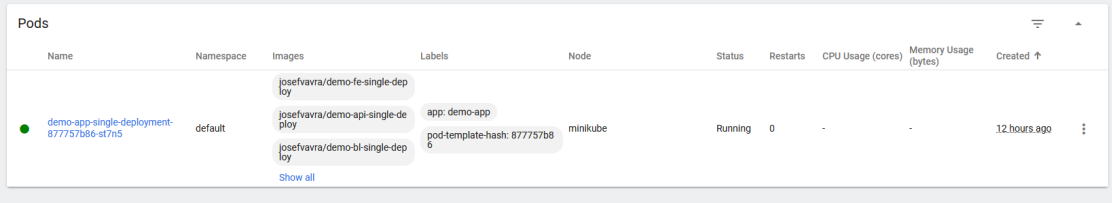
Problém nastává při volbě správného portu. Kubernetes volí port náhodně z předem nakonfigurovaného rozsahu(typicky je to 30000 až 32767). Tomu se však dá zabránit. Stačí nadefinovat Service. Service[24] je abstraktní způsob, jakým vystavit Pods mimo cluster. Kubernetes nastaví Pods jejich vlastní IP adresu a jediný DNS záznam a pak používá loadbalancing, aby mezi nimi přepínal. Službu nadefinuji takto: 21

V selector sekci jsem definoval stejný název, který se nachází v Deployment. Kubernetes pak ví, k čemu novou funkcionalitu přiřadit. Specifikoval jsem porty, na kterých běží aplikace uvnitř kontejneru a také nodePorts, což jsou porty, které budou vystaveny k vnějšímu přístupu. FE tedy uslyší na portu 31000 a API na portu 31081. Ještě než service zaregistruji, musím upravit Deployment a předat do proměnných správné parametry. Jediné co změním je hodnota `API_URL`, kterou nastavím na:

`http://172.24.8.150:31081`

Nechám kubernetes, aby se přepsal konfiguraci příkazem:

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



Obrázek 5.3 pod v dashboardu

Poté zaregistruji Service. `kubectl apply -f demo-single-service.yaml`

Že je vše proběhlo v pořádku otestuji příkazem `kubectl get svc`, kde vidím nastartovanou službu typu LoadBalancer, která má porty přiřazené tak, jak bylo uvedeno v definici. Navíc už nemá v EXTERNAL-IP hodnotu `<pending>`, ale stejnou jako je v CLUSTER-IP. Může se zdát, že by bylo jednodušší přiřadit do proměnné tento endpoint. Bohužel ale tato adresa je přiřazena službě až v moment jejího běhu. Ve fázi definování Deployment není známá.

V dashboard ověřím, že Pod zůstává zelený, což znamená že je vše v pořádku. V záložce services přibyla nová služba, která taky indikuje zelenou barvou úspěšný start. 5.3 Teď už na adrese http://172.24.8.150:31000 v prohlížeči získám formulář, který vyplním, potvrdím a opět dostávám potvrzení v mailu.

5.2 Škálování komponent

V případě velkého provozu na stránkách dojde k přetížení nekteré z komponent. Tato komponenta pak bude tzv. ”bottle neck”a bude brzdit hledké dodávání služby zákazníkovi. Pokud máme dostatek prostředků, můžeme danou komponentu škálovat. Replikovat existující Pod a spustit ho paralelně s existujícím, na kterém je příliš velká zátěž. Bohužel se právě celá aplikace nachází v jednom a tom samém Pod. To znamená, že pokud bych chtěl aplikaci škálovat, můžu maximálně zreplikovat momentální Pod vytvořený v Deployment. To je nežádoucí, protože zbytečne replikuji části služby které bez problémů vyšší zátěž snesou a mrhám tak prostředky. Řešením je rozpadnout existující Pod do více samostatných a škálovat jeden konkrétní, který má zrovna potíže.

Problém však nastává s okolními komponentami, které jsou momentálně vázané na jednu adresu. Na té se samozřejmě replikovaný Pod nenachází, už jen proto, že není ve stejné síti. Každý Pod také může být restartován, což způsobí že dostane přidělenou jinou adresu. Zadat tedy jeho pevnou adresu(ip v clusteru) při vytvoření není možné.

Selhání Pod může mít mnoho příčin a v každé aplikaci je třeba s tím počítat. Nic nám nezaručí, že Pod bude nastartován znovu na stejné adrese se stejným portem. Pokud k tomu dojde, aplikace nebude funkční. Tento problém není jediná komplikace kterou s sebou architektura mikroslužeb – narozdíl od architektury monolitické – přináší.

Jak první tak druhý problém je možné vyřešit load balancerem. Pro začátek pro každou komponentu vytvořím vlastní Pod. Pro FE bude Pod vypadat takto: 22

Veškeré hodnoty jsou stejné jako v případě definování za pomocí Deployment. Minikube byl však meztitím restartován, takže hodnota v `API_URL` je změněna, protože clusteru byla přiřazena jiná adresa. Jako label nastavím demo-fe-comp. To je nutné z toho důvodu, že momentálně je Pod uzavřený jak před okolním světem, tak před ostatními Pods. Zpřistupním ho za pomocí service, kterou nadefinuji do vlastního souboru takto: 23

Ve stejném duchu vytvořím i soubory obsahující definice pro ostatní komponenty. Zaregistrováním Service se v kubernetes prostředí vytvoří dns záznamy. Ty pak mohu použít při přístupu k Pod z jiného Pod. Změní se tedy i hodnoty předávané do proměnných prostředí. Pro API vyplním v definici Pod hodnotu http://demo-bl-service:8080 a pro BL http://demo-mail-service:8080. Všechny Pods i služby nastartuji a otevřu v prohlížeči formulář, který vyplním

```
apiVersion: v1
kind: Pod
metadata:
  name: fe-pod
  labels:
    app: demo-fe-comp
spec:
  containers:
  - name: demo-fe-pod
    image: josefvavra/demo-fe-single-deploy
    ports:
    - containerPort: 3000
    env:
    - name: API_URL
      value: http://172.30.58.46:31081
```

Výpis kódu 22 Definice FE Pod

```
apiVersion: v1
kind: Service
metadata:
  name: demo-fe-service
spec:
  type: LoadBalancer
  selector:
    app: demo-fe-comp
  ports:
  - protocol: TCP
    port: 3000
    targetPort: 3000
    nodePort: 31000
```

Výpis kódu 23 Definice FE Pod

```
apiVersion: apps/v1
kind: ReplicaSet
metadata:
  name: demo-bl-replicaset
spec:
  replicas: 2
  selector:
    matchLabels:
      app: demo-bl-comp
  template:
    metadata:
      labels:
        app: demo-bl-comp
    spec:
      containers:
      - name: bl-pod
        image: josefvavra/demo-bl-pod-build
        ports:
        - containerPort: 8080
        env:
        - name: DEMO_MAIL_ENDPOINT
          value: http://demo-mail-service:8080
```

Výpis kódu 24 Definice BL replicasetu

a obdržím potvrzení objednávky.

Nyní k samotnému škálování. Pod samostatně zreplikovat nelze. K tomu slouží ReplicaSet[25], což je funkce, která má na starost replikaci Pods. Budu replikovat bl-pod, takže jej napřed smažu příkazem `kubectl delete pod bl-pod` a konfiguraci přenesu do definice ReplicaSet takto: 24

Spustím příkazem `kubectl apply -f bl-replicaset.yaml` a otestuji, že aplikace funguje provoláním formuláře. Vše proběhlo v pořádku a průběh aplikace loguje do obou zreplikovaných Pods.

5.2.1 Service Discovery

Services, které jsem vytvořil v tomto scénáři poskytují ještě jednu funkcionalitu, která je velmi důležitá. A tou je service discovery. Pokud bych Services vůbec nepoužíval, musel bych se v komunikaci odkazovat na adresy Pods. Adresa je pro Pod vytvořena a přiřazena při startu. To je problém hned ze dvou důvodů. První je, že není možné předem zjistit na jaké adrese bude Pod v clusteru vystaven, takže mapování v konfiguračních souborech by se dělalo jen velmi obtížně. Musel bych napřed vytvořit Pod pro MAIL komponentu, zjistit jaká je adresa, tu vložit do konfiguračního souboru pro BL komponentu a poté vytvořit další Pod. Takto bych postupoval i pro zbytek komponent. To je zdlouhavý proces a není možné celou aplikaci spustit jedním příkazem. Druhý problém je, že pokud dojde k restartování Pod, bude mu přidělena nová adresa. V ten moment se celý řetěz volání rozpadá, protože jedna z komponent má nesprávnou konfiguraci. Z toho důvodu se vytváří Services.

Do konfiguračních souborů se jednoduše vloží název Service, jenž má na starost Pods ať už přímo nebo přes ReplicaSet nebo Deployment. Service není Pod, proto nepodléhá stejným problémům. Restartování nezmění její IP adresu. Service hlídá endpoints, na které má odkazovat. Získá je z Kubernetes API serveru pokaždé, když dojde k aktualizaci jejich stavů.

Cluster DNS pak zajišťuje, že je možné používat mezi Kubernetes objekty jen název služby a nikoliv její adresu.

Kapitola 6
Secrets

Do konfigurací je možné(a někdy nutné) zahrnout hesla, ssh klíče, přístupové údaje do databází, certifikáty a jiné šifrování. Tyto všechny údaje nazýváme secrets. Secrets je nutné konfigurovat a dost často konfigurovat pro každé prostředí různým způsobem. Pokud ale secret explicitně uvedeme v konfiguračním souboru, vzniká bezpečnostní riziko a s ním i velký problém. V jakémkoliv kolektivu je žádoucí, aby přístupové údaje znalo co nejmenší množství lidí.

6.1 Nastavení databáze v lokálním prostředí

Momentálně se v aplikaci používá inmemory databáze. Pokub by došlo k restartování Pod, tak budou všechna uložená data smazána. Zabránit se tomu dá tak, že nastavím samostatně běžící kontejner obsahující Postgres databázi.

PostgreSQL[26] je objektově orientovaný databázový systém. V Dockerhub lze najít obraz[27], který spustí Postgres v kontejneru. Proměnné prostředí které lze v kontejneru nastavit při startu nastaví databázi uživatelské jméno a heslo.

Jak nastavit postgres lokálně lze vyčíst z článku [28]. Postgres poslouchá na portu 5432. Pokud navíc při spuštení specifikuji uživatele a heslo a samozřejmě vystavím zmíněný port, bude možné se k databázi připojit na url `localhost:5432/nazev_uzivatele`. Navíc musím upravit konfigurační soubor v komponentě BL. Parametry databáze budu předávat jako proměnné prostředí, takže config bude vypadat následovně25.

Databázi pak spustím v kontejneru tímto příkazem 26

BL komponentu musím znovu sestavit a vzhledem k tomu, že se jedná o lokální spuštění budu opět přenastavovat porty, aby nedošlo ke kolizi. Nastevní zůstane stejné jako při lokálním běhu, takže přepoužiji všechny obrazy kromě BL. BL znovu sestavím a otaguji. Celou aplikaci spustím.

```
...
datasource:
  url: ${DB_RESERVATION_JDBC_URL}
  username: ${DB_RESERVATION_USERNAME}
  password: ${DB_RESERVATION_PASSWORD}
  driver-class-name: org.postgresql.Driver
...
```

Výpis kódu 25 Konfigurace připojení do databáze

```
docker run -p 5432:5432 \
-e POSTGRES_PASSWORD=password \
-e POSTGRES_USER=user -d postgres
```

Výpis kódu 26 Spuštění postgres databáze

Aplikace sice běží, ale pořád jsem v situaci, ve které můžu přijít o data. Pokud kontejner Postgres spadne, veškerá data budou pryč. Této nepříjemnosti zabraňuji techologie volume.

6.1.1 Docker compose a volume

Volume je mechanismus kontejnerů, který slouží k zachování dat navzdory restartovní Pod. Data jsou uložená přímo v souborovém systému clusteru[29]. Výhodou Volumes je, že nezvětšují prostor kontejneru, který volume používá a potřebuje ukládat data. Volume existuje mimo životní cyklus daného kontejneru.

Volume je možné vytvořit příkazem nebo přes zaregistrování definice v souboru. Pro lepší čitelnost a další použití budu registraci provádět v souboru. Volume vytvoří prostor na hostitelském stroji(tedy tam, kde je spuštěn Docker). Existuje možnost ukládat data na vzdáleném hostiteli. Toho je možné dosáhnout za použití takzvaných Volume drivers[30]. Tímto scénářem se však v práci zabývat nebudu.

Alternativou k Volumes jsou Bind Mounts. Ty narozdíl od Volumes fungují tak, že vytvoří mount[31] na hostitelském stroji a pak jej připojí ke kontejneru. To je velmi výkonné, ale spoléhá na souborový systém hostitele a jako vedlejší účinek může kontejner manipulovat s citlivými daty hostitele. V této práci budu pracovat výhradně s Volumes.

Aplikaci Volume ukážu na příkladu s docker-compose technologií. Aby vše fungovalo, upravím Docker compose soubor, který už jsem dříve vytvořil. Nově v něm použiji nový BL kontejner(budu ho muset sestavit ještě jednou, tentokrát už s výchozím portem), který už je možné nakonfigurovat aby se připojil k databázi. Dále přidám start Postgres databáze a samozřejmě vytvořím zmíněnou funkci Volume. Výsledný docker-compose soubor bude vypadat následovně:

27

Specifikoval jsem cestu, která je nutná k zachování dat v Postgres databázi[32]. Celou aplikaci nastartuji, vyplním formulář a odešlu. Poté co obdržím potvrzení, celou aplikaci vypnu. Při novém startu otestuji, zda se předchozí vypnění udrželo a nebylo smazáno. Zkontroluji, zda je Postgres kontejner nastartovaný a zároveň zjistím jeho identifikátor. Tento údaj předám nástroji Docker a tím získám připojení do kontejneru. `docker exec -it 725ffc0779ba bash`. V příkazové řádce, kterou nyní kontejner ovládá, spustím přihlášení do Postgres databáze tímto příkazem `psql -U user`. Přepínač `-U` očekává na vstupu název uživatele, pod nímž se hodlám přihlásit a už v definici yaml souboru jsem vytvořil uživatele user. Z třídy ReservationEntity v BL komponentě lze vyčíst, že ukládá data do tabulky reservation. Musím zjistit, zda ukládání dat probíhá bez problému. Proto v otevřeném kontejneru, kde jsem nyní spustil psql konzoli, spustím SQL příkaz, který z tabulky reservation získá data.

`SELECT * FROM reservation;`.

Obdržím záznam, který jsem vytvořil v předchozím běhu, což dokazuje že se data zachovávají i navzdory restartu.

6.1

6.1.2 Technologie docker secrets

Takto jsou však veškeré citlivé údaje odhalené a každý kdo otevře soubor, je může zneužít. Z docker-compose souboru je tedy odstraním a heslo dostanu do databáze jiným způsobem.

```yaml
version: "3.9"
volumes:
  db-data:
services:
  demo-fe-compose:
    image: demo-fe-compose
    ports:
      - "8080:3000"
    environment:
      - API_URL=http://localhost:8090
  demo-api-compose:
    image: demo-api-compose-cors
    ports:
      - "8090:80"
  demo-bl-compose:
    image: demo-bl-compose-secret
    depends_on:
      - db-postgres
    environment:
      - DB_RESERVATION_JDBC_URL=jdbc:postgresql://db-postgres:5432/user
      - DB_RESERVATION_USERNAME=user
      - DB_RESERVATION_PASSWORD=password
      - SPRING_JPA_HIBERNATE_DDL_AUTO=update
  demo-mail-compose:
    image: demo-mail-compose
  db-postgres:
    image: postgres:13.1-alpine
    environment:
      - POSTGRES_USER=user
      - POSTGRES_PASSWORD=password
    volumes:
      - db-data:/var/lib/postgresql/data
```

Výpis kódu 27 Docker compose s volume

```
PS C:\Users\vavra> docker ps
CONTAINER ID   IMAGE                  COMMAND                  CREATED          STATUS          PORTS
          NAMES
40f2581e6ea0   demo-bl-compose-secret   "/bin/sh -c 'java ${…"   9 minutes ago    Up 10 seconds   80/tcp
          dockercomposesecrets_demo-bl-compose_1
ad974fc36747   postgres:13.1-alpine     "docker-entrypoint.s…"   20 minutes ago   Up 12 seconds   5432/tcp
          dockercomposesecrets_db-postgres_1
4054e0d58ef5   demo-mail-compose        "/bin/sh -c 'java ${…"   22 minutes ago   Up 12 seconds   80/tcp
          dockercomposesecrets_demo-mail-compose_1
29e9e16d3508   demo-api-compose-cors    "/bin/sh -c 'java ${…"   22 minutes ago   Up 12 seconds   0.0.0.0:8090->80/tcp
          dockercomposesecrets_demo-api-compose_1
982150723f0a   demo-fe-compose          "/docker-entrypoint.…"   22 minutes ago   Up 12 seconds   80/tcp, 0.0.0.0:8080->
3000/tcp   dockercomposesecrets_demo-fe-compose_1
PS C:\Users\vavra> docker exec -it ad974fc36747 bash
bash-5.1# psql -U user
psql (13.1)
Type "help" for help.

user=# SELECT * FROM reservation;
 id |          date           |     email     | first_name | last_name | seat_id | train_id
----+-------------------------+---------------+------------+-----------+---------+----------
  1 | 2022-04-26 13:59:40.111 | josef@fake.cz | Josef      | Vavra     | 1-21    | ICE-575
(1 row)
```

Obrázek 6.1 Obsah tabulky reservation

```
secrets:
  postgres_password:
    file: postgres_pass.txt
```

Výpis kódu 28 Docker secret definice

```
demo-bl-compose:
  image: demo-bl-compose-secret
  depends_on:
    - db-postgres
  environment:
    - DB_RESERVATION_JDBC_URL=jdbc:postgresql://db-postgres:5432/user
    - DB_RESERVATION_USERNAME=user
    - DB_RESERVATION_PASSWORD=/run/secrets/postgres_password
    - SPRING_JPA_HIBERNATE_DDL_AUTO=update
  secrets:
    - postgres_password
```

Výpis kódu 29 BL odkaz na docker secret

Docker-compose je na takové scénáře připravený a umožňuje specifikovat secrets přímo v konfiguračním souboru. Slouží k tomu sekce ”secrets”, kterou takto v souboru nadefinuji:

28

Sekci secrets přidám i do definice BL části v Docker compose. V ní bude jeden záznam, který bude odkazovat na vytvořený secret. Proměnnou prostředí pro BL ponechám, ale změním její hodnotu tak, aby odkazovala na data uložená v souboru, který vytvoří Docker. Definici MAIL zatím ponechám tak jak je pro testovací účely.

Aktualizace BL části nyní vypadá takto: 29

Celou aplikaci spustím a bohužel získám chybu. Komponenta BL vůbec nenastartovala. Chyba je způsobená tím, že do hesla je uložená hodnota

"/run/secrets/postgres_password"

a ne *"password"*. Tento problém má poněkud složitější řešení. Intuice napovídá vložit do proměnné výsledek příkazu `cat /run/secrets/postgres_password`. To se však obalí do uvozovek a v proměnné je opět uložený příkaz nikoliv jeho vyhodnocení. Upravit konfigurační soubor tak, aby přijímal heslo ze souboru také není možné, jelikož rozhraní Spring properties to neumožňuje. Řešením je upravit CMD část Dockerfile. Stejně jako pro FE vytvořím skript, ve kterém uvedu původní příkaz, který startoval tuto komponentu a před něj vložím exportování proměnné s hodnotou, která je v uvedeném souboru. Skript vypadá následovně 30 a je uložený v codenow/config.

Dockerfile nyní v CMD části nespouští jar soubor, ale vytvořený skript run.sh, takto: `CMD ./codenow/config/run.sh`

Z docker-compose.yaml proměnnou pro BL úplně odstraním a naopak přidám secret do definice Postgres databáze. Proměnnou `POSTGRES_PASSWORD` úplně odstraním a nahradím jí `POSTGRES_PASSWORD_FILE`, do které vložím hodnotu `/run/secrets/postgres_password`. Celý Docker compose soubor spustím, otevřu v prohlížeči, vyplním a obdržím potvrzení o úspěšném založení rezervace.

Uchovávání hesel v plaintext souboru je v produkčním prostředí velmi nebezpečné. Je nutné nějakým způsobem soubory obsahující citlivé údaje schovat před případnými útočníky.

```
#!/bin/bash

export DB_RESERVATION_PASSWORD=`cat /run/secrets/postgres_password`

java ${JAVA_OPTS} \
-Dserver.port=80 \
-Dspring.config.additional-location=file:/codenow/config/ \
-Dlogging.config=classpath:logback-codenow.xml -jar app.jar
```

Výpis kódu 30 BL run script

6.2 Secrets v kubernetes

Celý proces nyní přenesu na minikube. Abych spustil postgres databázi na clusteru, použiji kubernetes deployment. Kubernetes má vlastní přístup k zachování hesel bezpečným způsobem, napřed ale musím celou aplikaci nastartovat včetně všech propojení a založit Volume. Volume potřebuji stejně jako v předchozích příkladech proto, že když dojde k restartování Pod, chci zachovat data. Proto použiji dvě funkcionality kubernetes, kterými jsou PersistentVolume a PersistentVolumeClaim. PersistentVolume vytvoří v clusteru zdroj, který ignoruje životní cykly všech Pods, takže nemůže dojít k jeho nechtěnému smazání. Někteří cloud poskytovatelé mají vlastní implementace této funkcionality. PersistentVolumeClaim je pak požadavek od uživatele, jenž definuje prostor, který chce alokovat a způsob jakým bude prostor využívat. Ten je podobný funkcionalitě Pod[33]. Začnu tím, že pro Postgres vytvořím Deployment. Stejně jako v Docker compose scénáři nadefinuji proměnné prostředí obsahující přístupové údaje. Zpřístupním port 5432, na kterém Postgres očekává připojení, definuji obraz, který chci použít a nakonec připojím volume. Volume se připojuje pouze specifikací názvu PersistentVolumeClaim, který vytvořím spolu se samotným Volume. Takto vypadá Deployment 31

V sekci volumeMounts jsem nadefinoval cestu, kterou má PersistentVolumeClaim použít pro zachování. Nyní vytvořím PersistentVolume32 a PersistentVolumeClaim33. Přístupový mód jsem nastavil na ReadWriteMany, což umožňuje libovolně velkému množství Pods se připojit a zapisovat do tabulky. V produkčním prostředí je toto nežádoucí praktika a je lepší poskytnout přístup jen nezbytným komponentám. Kapacitu úložiště jsem nastavil na 1GB, což je více než dostatek.

Po registraci těchto částí aplikace za pomoci kubectl můžu zaregistrovat i vytvořený Deployment. Na rozdíl od spuštění Docker compose se nevytvoří tabulka. Aby bylo možné vytvářet záznamy musím ji spolu se sekvencí pro primární klíč vytvořit. V Kubernetes dashboard rozkliknu pro Postgres Pod funkci Exec a v prohlížeči se mi spustí terminál. V něm se příhlásím příkazem `psql -U user`. A spustím příkaz pro vytvoření tabulky34 a pro vytvoření sekvence:

```
CREATE sequence reservation_id_sequence START 1;
```

Postgres je tedy nastartovaný a obsahuje veškerá potřebná data. Nyní musím propojit BL s databází, aby byla možnost zapisovat. K tomu je nutné vystavit Pod obsahující databázi ostatním Pods v clusteru. K tomu opět použiji Service. Připojím ji na Pod a tím ho zveřejním. Takto vypadá definice35

Upravím patřičnou proměnnou v ReplicaSet pro BL komponentu, aby odkazovala na novou službu a můžu celou aplikaci spustit. Po spuštění vyplním formulář a obdržím potvrzení, což dokazuje že celá apllikace funguje. Heslo však předávám aplikaci nezabezpečeným způsobem a navíc si ho může kdokoliv přečíst. Proto heslo zachovám kubernetes funkcí secrets. Nástroj příkazové řádky kubectl umí heslo zachovat několika způsoby. Nejrychlejší je rovnou v příkazové řádce spustit tento příkaz36, což vytvoří Secret s názvem postgres-pass a pak je možné ho použít podle tohoto článku[34] v yaml souborech.

Po registraci, Kubernetes s citlivým údajem zachází jako s každým jiným objektem. Ku-

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: postgres-deployment
spec:
  selector:
    matchLabels:
      app: postgres
  replicas: 1
  template:
    metadata:
      labels:
        app: postgres
    spec:
      containers:
      - name: postgres
        image: postgres:13.1-alpine
        volumeMounts:
            - mountPath: /var/lib/postgresql/data
          name: postgres-persistent-volume-claim
        ports:
        - containerPort: 5432
        env:
        - name: POSTGRES_USER
          value: user
        - name: POSTGRES_PASSWORD
          value: password
      volumes:
      - name: postgres-persistent-volume-claim
        persistentVolumeClaim:
          claimName: postgres-persistent-volume-claim
```

Výpis kódu 31 Postgres deployment definice

```yaml
apiVersion: v1
kind: PersistentVolume
metadata:
  name: postgres-volume-13
spec:
  storageClassName: postgres-storage
  capacity:
    storage: 1Gi
  accessModes:
    - ReadWriteMany
  hostPath:
    path: "/mnt/data/postgres-13"
```

Výpis kódu 32 Postgres volume definice

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: postgres-persistent-volume-claim
spec:
  storageClassName: postgres-storage
  accessModes:
  - ReadWriteMany
  resources:
    requests:
      storage: 1Gi
```

Výpis kódu 33 Postgres volume claim definice

```
CREATE TABLE reservation(
id INTEGER PRIMARY KEY,
first_name VARCHAR(100),
last_name VARCHAR(100),
email VARCHAR(100),
seat_id VARCHAR(100),
train_id VARCHAR(100),
date DATE
);
```

Výpis kódu 34 Vytvoření postgres tabulky

```
apiVersion: v1
kind: Service
metadata:
  name: postgres-service
spec:
  type: LoadBalancer
  selector:
    app: postgres
  ports:
  - protocol: TCP
    port: 5432
    targetPort: 5432
```

Výpis kódu 35 Postgres service

```
kubectl create secret generic postgres-pass \
--from-literal=postgres-password=password
```

Výpis kódu 36 Vytvoření Secret přes kubectl

```
valueFrom:
  secretKeyRef:
    name: postgres-pass
    key: postgres-password
```

Výpis kódu 37 Úprava pro zachování heslo pro BL a postgres

bernetes podle tohoto článku[35] aplikuje metodu nejmenší možné privilegovanosti. K uloženým citlivým údajům má přístup akorát Pod pokud je explicitně uveden jako součástí připojeného Volume, nebo pokud kubelet stahuje obraz, jenž je v Pod použit. Kubelet[36] je hlavní agent, který běží na každém Node. Jeho úkolem je registrace nových Nodes.

Upravím tedy BL a postgres definici tak, aby už heslo nebylo explicitně uvedeno, ale používal se místo něj Secret. Úprava bude vypadat takto.37

ReplicaSet a Deployment znovu zaregistruji. To způsobí jejich restartování s novými hodnotami. Celý proces odeslání formuláře zopakuji a vše projde bez problémů.

Kapitola 7

Cloudové prostředí a jeho poskytovatelé

V této kapitole aplikuji většinu vyzkoumaných řešení na reálné nasazení na cloud poskytovatele AWS. AWS jsem si vybral z toho důvodu, že je nejrozšířenější a nejvíce používaný poskytovatel cloud. Nabízí některé služby zdarma, a proto je vhodným kandidátem k nasazení aplikace. Na trhu se pohybuje více alternativ jako například Microsoft Azure nebo Google ClouFd. AWS je zkratka pro Amazon Web Services[37]. Nabízí obří škálu produktů od těch zdarma až po placené. Při zakládání je nutné specifikovat region, což je přístupová zóna pro danou službu. Regionů je po celém světě mnoho. Jen v Evropě je možnost vybrat hned z několika.

Pro práci je nejdůležitější služba s názvem EKS, což znamená Elastic Kubernetes Service[38]. EKS umožňuje spouštění Kubernetes aplikací bez nutnosti instalace a údržby. Spuštění a škálování probíhá přes mnoho AWS zón, což zaručuje vysokou přístupnost. Dovede automaticky kontrolovat řídící body a nahradit nefunkční části, případně je automaticky zaktualizovat. Funkcionalita kromě toho spočívá v integraci dalších služeb, jenž AWS poskytuje. Jednou z nich je ECR, což je Docker registr, kam je možné zasílat sestavené obrazy. Poté Elastic Load Balancing, což je služba, která jak název napovídá, poskytuje load balancing. Funkci budu muset při implementaci řešení na tomto poskytovateli využít. Kromě dalších služeb je pro tuto práci důležitá ještě IAM. Tato služba má na starost řízení přístupů. Rozhoduje jací uživatelé a za jakých podmínek mohou mít přístup k jakým službám a zdrojům.

Jak je popsáno v tomto videu[39], je nastavení clusteru do takového stavu aby byl schopný pracovat s aplikací je poměrně složitá záležitost. Naštěstí existuje komunitní nástroj eksctl[40]. Eksctl dovede vytvořit cluster s libovolnou konfigurací příkazem z terminálu. K autorizaci používá přihlašovací údaje uložené na harddisku. Vytvořil jsem si AWS účet a uložil přihlašovací údaje na správné místo podle tohoto návodu[41].

Vyberu jméno demo-cluster-micro. Pojmenuji soubor Nodes a jejich počet. Možností pro výběr typu node je ohromné množství[42]. Rozhodl jsem se pro t2.small, jelikož poskytuje minimální dostatečný počet Pods. Zvolím region `eu-west-2`. Vytvořím yaml soubor se všemi konfiguracemi, jenž eksctl zkonzumuje tímto příkazem:

```
eksctl create cluster -f cluster-definition.yaml
```

Soubor vypadá takto: 38

Až eksctl vykoná vše potřebné, uloží navíc do lokálního(myšleno na stroji z něhož byl příkaz spuštěn) konfiguračního Kubernetes souboru způsob, jak se připojit k danému clusteru. Tento konfigurační soubor využije kubectl, což hned ověřím příkazem `kubectl get nodes`. Získám záznamy s vytvořeným Nodes 7.1

Nyní můžu aplikaci spustit na AWS clusteru. Veškeré definice komponent přesunu z Pod do Deployment, aby se mi s nimi lépe pracovalo. Místo abych pracoval s velkým množstvím souborů,

```yaml
apiVersion: eksctl.io/v1alpha5
kind: ClusterConfig

metadata:
  name: demo-cluster-small
  region: eu-west-2

nodeGroups:
  - name: ng-1
    instanceType: t2.small
    desiredCapacity: 10
    volumeSize: 80
  - name: ng-2
    instanceType: t2.small
    desiredCapacity: 2
    volumeSize: 100
```

Výpis kódu 38 Definice clusteru

```
PS C:\Users\vavra> kubectl get nodes
NAME                                          STATUS   ROLES    AGE   VERSION
ip-192-168-0-67.eu-west-2.compute.internal    Ready    <none>   13h   v1.22.6-eks-7d68063
ip-192-168-10-17.eu-west-2.compute.internal   Ready    <none>   13h   v1.22.6-eks-7d68063
ip-192-168-14-227.eu-west-2.compute.internal  Ready    <none>   13h   v1.22.6-eks-7d68063
ip-192-168-3-25.eu-west-2.compute.internal    Ready    <none>   13h   v1.22.6-eks-7d68063
ip-192-168-54-71.eu-west-2.compute.internal   Ready    <none>   13h   v1.22.6-eks-7d68063
ip-192-168-62-203.eu-west-2.compute.internal  Ready    <none>   13h   v1.22.6-eks-7d68063
ip-192-168-63-249.eu-west-2.compute.internal  Ready    <none>   13h   v1.22.6-eks-7d68063
ip-192-168-64-151.eu-west-2.compute.internal  Ready    <none>   13h   v1.22.6-eks-7d68063
ip-192-168-67-52.eu-west-2.compute.internal   Ready    <none>   13h   v1.22.6-eks-7d68063
ip-192-168-70-252.eu-west-2.compute.internal  Ready    <none>   13h   v1.22.6-eks-7d68063
ip-192-168-75-199.eu-west-2.compute.internal  Ready    <none>   13h   v1.22.6-eks-7d68063
ip-192-168-94-209.eu-west-2.compute.internal  Ready    <none>   13h   v1.22.6-eks-7d68063
PS C:\Users\vavra>
```

Obrázek 7.1 Výsledek kubectl get nodes

```
eksctl utils associate-iam-oidc-provider \
--cluster demo-cluster-small \
--region eu-west-2 \
--approve
```

Výpis kódu 39 Vytvoření poskytovatele přes eksctl

```
PS C:\Users\vavra> aws iam list-open-id-connect-providers | grep  BE5A7DB7B93B1C77075A2BBDD7830D3E
            "Arn": "arn:aws:iam::126930530308:oidc-provider/oidc.eks.eu-west-2.amazonaws.com/id/BE5A7DB7B93B1C77075A2BBD
D7830D3E"
```

Obrázek 7.2 Zaregistrovaný OIDC poskytovatel

sloučím ty, které k sobě logicky patří do jednoho souboru a oddělím definice třemi pomlčkami, jak yaml formát umožňuje. Použiji soubory z předchozího příkladu. Už tedy nebudu používat separátně soubor pro definice Service a definici Pod pro BL a místo toho budu mít jeden soubor s definicí Deployment a Service. To samé zopakuji pro všechny komponenty. Přepnu počet replik zpět na jednu, protože víc v tomto příkladu není potřeba.

Veškeré potřebné služby, které musí být spuštěny aby fungoval Postgres, sloučím také do jednoho souboru. Přes kubectl veškeré služby nastartuji. Včetně vytvoření Secret pro heslo do Postgres. Kubectl se totiž změnil kontext a s ním zanikly veškerá předchozí nastavení. Zkontroluji, že se vše opravdu nahrálo na cluster příkazem `kubectl get pods`. Také provedu spuštění terminálu v Postgres Pod, kde založím sekvenci a tabulku.

Vše tedy běží, ale na jaký endpoint teď odkázat FE, aby se připojil k API? Spustím `kubectl get svc` a zjistím, že veškeré EXTERNAL-IP jsou ve stavu `<pending>`. S tímto už jsem se setkal při spouštění na lokálním minikube. Problém je v tom, že není spuštěný load balancer. Instalace load balancer na cluster je v AWS poněkud zdlouhavý proces. Z počátku musím zjistit, zda je na clusteru IAM OIDC poskytovatel[43]. OIDC je jednoduchá identifikační vrstva postavená na OAuth 2.0 protokolu. Zda už je poskytovatel nastavený zjistím podle tohoto návodu [44] tak, že spustím sérii příkazů, jenž odhalí zda existuje. Zjistil jsem že ne, a tak použiji nástroj eksctl abych poskytovatele vytvořil39.

Poskytovatel je založený. To potvrzuje výsledek příkazu7.2

Nyní můžu pokračovat v procesu instalace load balancer. Celý návod je popsaný na AWS[45], zde ho pouze shrnu. Podle první instrukce vytvořím IAM politiku. Pomocí eksctl založím iamserviceaccount. Poté zbývá zaregistrovat manažera certifikátů. Pak už stačí jen upravit specifikaci ovladače tak, aby byla shodná s mým clusterem. Když je vše hotovo můžu specifikaci zaregistrovat a tím spustit ovladač load balancer. Nyní už kuberenetes služby získávají EXTERNAL-IP. Použiji tu, která přísluší Service obsluhující API Pod a vložím ji do FE Deployment. Zobrazují se však i služby, jenž vůbec být vystaveny nemusí. To je tím, že veškeré Services jsou spuštěny jako typ LoadBalancer. Pokud je zmením na ClusterIP, budou jejich endpoints veřejné pouze v rámci clusteru a nebudou vystaveny ven. Ještě upravím Service pro FE, aby se nevystavovala na portu 3000, ale na 80, jak je u většiny webových stránek normální. Vše zaregistruji a znovu spustím za pomoci kubectl. Ve finále můžu otevřít url, jenž přísluší FE v prohlížeči. Obdržím formulář, ten vyplním a získám potvrzení o úspěšném založení rezervace.

Kapitola 8
Řešení problému více prostředí

V této práci jsem demonstroval, že spustit aplikaci tak, aby vše fungovalo není vůbec jednoduchý úkol. Navíc odstranit co největší množství lidského faktoru, jenž na rozdíl od stroje chybuje, je ještě složitejší. Aplikace vždy prošla určitým procesem, který nakonec vykrystalizoval v pokaždé více méně podobnou strukturu. Komponenty byly obaleny určitou abstrakcí. Stejně tak komunikace mezi nimi. Vše bylo spuštěno v jednom z prostředí a na konci tohoto procesu byla funkční aplikace. To vše jsem ale musel přetvářet pro každé prostředí zvlášť a pokaždé jsem začínal od téměř začátku. Takový proces je vysoce náchylný k chybám. Také je velmi časově náročný. Navíc předělávat celou strukturu spouštění při každé změně, se ve světě komerčních aplikací může velmi prodražit.

Nyní budu demonstrovat způsob, jak oddělit konfiguraci od definice struktur. Že je struktura funkční už jsem prokázal mnohokrát. Nemá smysl ji tedy měnit. Konfigurace se však bude pokaždé lišit. Předvedu dva koncepty, které je nutné zohlednit, pokud je cílem spuštění aplikace s minimálním úsilím.

8.1 Kubernetes ConfigMap

První co je nutné představit, abych mohl provést nasazení je Namespace[46]. To je funkce Kubernetes, která představuje mechanismus oddělení zdrojů v jednom clusteru. V tomto příkladu se opět vrátím k nástroji minikube, na kterém budu spuštění aplikace demonstrovat. Aplikaci spustím dvakrát na stejném clusteru. Pokaždé s jinou konfigurací a v jiném Namespace.

Kubernetes stejně jako u Service nebo Deployment dovede vytvořit Namespace na základě yaml souboru. Vytvořím dva. Jeden „vývojový“ s názvem `development` a druhý „produkční“ s názvem `production`. Specifikace Namespace není vůbec složitá. Není nutné nastavovat prakticky nic kromě názvu. Poté co Namespace zaregistruji, vytvořím pro každé prostředí vlastní konfigurační soubor. Tedy v kubernetes světě ConfigMap[47]. ConfigMap je způsob jak oddělit konfigurace od kódu a dokonce i struktury spouštění. Veškeré proměnné prostředí přesunu do ConfigMap a tu zaregistruji pro příslušný Namespace.

Jak správně nastavit Deployment, Service a Pods už jsem otestoval v předchozích případech. Proto veškerá nastavení shrnu do jednoho souboru, který budu registrovat. V tomto souboru už nebude žádná proměnná prostředí. Ty budou outsourcované ve zmíněných ConfigMaps. Takto bude vypadat ConfigMap40 pro development Namespace.

Nyní zaregistruji konfigurační soubor příkazem `kubectl apply -f dev-config.yaml`. Poté Namespace příkazem `kubectl apply -f development-namespace.yaml`. Nastavit nodePort z ConfigMap bohužel nejde. Proto budu muset vytvořit vlastní soubor, obsahující Service pro API a FE. Obě komponenty směřují ven a je pro ně nutné specifikovat na jakém portu budou přístupné.

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: config
  namespace: development
data:
  api.url: http://172.17.17.61:31181
  bl.endpoint: http://demo-bl-service:8080
  mail.endpoint: http://demo-mail-service:8080
```

Výpis kódu 40 Vývojový ConfigMap

Vytvořím tedy speciální soubor právě s těmito definicemi a zaregistruji ho. FE komponentu jsem nastavil, aby poslouchala na portu 31100 a API na 31181. Nyní můžu zaregistrovat celý namespace-deployment.yaml. Při spouštění příkazu dám pozor, abych specifikoval Namespace. `kubectl apply -f namespace-deployment.yaml -n development`. V prohlížeči už můžu otevřít příslušnou adresu s portem 31100, kde získám formulář, jenž můžu vyplnit.

8.2 Kubernetes Ingress

Zakládat pro každé prostředí dvě Service pro API a FE je však přesně situace, které se chci vyhnout. Potřebuji dostat tyto dvě definice do společného souboru. Při vytváření dalšího prostředí by bylo totiž ideální, aby stačilo zaregistovat ConfigMap s příslušnými konfiguracemi. Spouštění celé aplikace na novém prostředí by se značně zjednodušilo a bylo by určitě více uživateslky přívětivé.

To lze vyřešit přes Ingress[48]. Ingress je další způsob, kromě Loadbalancer a NodePort, jak zveřejnit na clusteru Service. Ingress Controller je pak load balancer, díky kterému není třeba používat minikube tunnel. Ten vypnu, protože už ho nebudu potřebovat. Ingress Controller se však musí nastavit. Většina cloud poskytovatelů poskytuje vlastní implementaci tohoto ovladače. Minikube není vyjímkou. Jestliže je nástroj spuštěný, stačí spustit tento příkaz `minikube addons enable ingress`. Mezi Namespaces se objeví ingress-nginx, ve kterém je spuštěný ovladač. Nyní můžu nadefinovat ConfigMap, Service a Ingress pro produkční Namespace. ConfigMap bude mít stejné parametry jako development verze, akorát s rozdílným namespace a touto proměnnou `api.url: http://api-demo-prod.com`. Service už není nutné vystavovat mimo cluster, takže je možné přepnout typ na ClusterIP z LoadBalancer. Tím pádem z definice zmizí i nodePort. Stroji ze kterého spouštím příkazy je vystaven pouze Ingress. V definici Ingress specifikuji dvě cesty. Jedna pro FE komponentu a druhou pro API. Jak už je vidět v ConfigMap, API url nastavím na `api-demo-prod.com`. FE komponentu bude možné navštívit na adrese `fe-demo-prod.com`. Pro obě adresy specifikuji na kterém portu mají s danými kontejnery komunikovat. Jsou to stejné porty, které jsou otevřeny pomocí příslušných Services. Když je vše zaregistrováno, můžu vyzkoušet aplikaci spustit. V prohlížeči otevřu `http://api-demo-prod.com`, ale žádný formulář se neobjeví. To je proto, že název domény je zaregistrovaný pouze v rámci clusteru. Prohlížeč se pokouší dohledat pod zadanou url stránku, která neexistuje. To lze vyřešit zadáním IP adresy nástroje Minikube do souboru pro mapování názvů hostitelů. V každém operačním systému se tento soubor nachází někde jinde. Pro Windows je na adrese `C:\Windows\System32\drivers\etc\hosts` a vložím do něj hodnotu `172.17.17.218 fe-demo-prod.com api-demo-prod.com`.
Nyní už prohlížeč otevře na adrese požadovaný formulář, který správně zavolá celý sled kroků a založí rezervaci.

Kapitola 9
Závěr

V této práci jsem demonstroval pozdější fáze životního cyklu aplikce. Ukázal jsem důležitost přístupu k aplikaci po její implementaci. Nasazování aplikace není jednoduchá disciplína a stejně jako k jejímu vývoji je nutné věnovat velkou pozornost a prostředky k její údržbě. Nastínil jsem, jak používat jednotlivé nástroje, jenž práci s hotovou aplikací ulehčují. Veškeré poznatky, které jsem nasbíral v průběhu práce na testovém clusteru jsem nakonec aplikoval na nasazení na cloud. Ukázal jsem hlavní a nejpoužívanější nástroje a neopomenul zmínit jejich alternativy. Také jsem ukázal důležitost užívání Secrets. Nakonec jsem celou aplikaci spustil paralelně v různých prostředích s odlišnými konfiguračními parametry.

Celou práci je možné rozšířit právě o popis alternativních nástrojů a poté jejich demonstraci na demo aplikaci. Kromě toho je možné ukázat na nástroje, které stávající užití rozšiřují, jako je například nástroj Helm[49]. Ten slouží k verzování Kubernetes aplikací a ještě rozšiřuje jejich použití. Dalším nástrojem, kterým jsem se mohl zabývat je Ansible[50] od firmy RedHat. Jedná se o open source nástroj, který má za úkol nasazení na mnoho cílů. Automatizuje celý proces rozeslání takzvaných ansible modules na jednotlivé Nodes podle definice, kterou obdrží v yaml souboru. I k němu existují alternativy jako je například Chef nebo Puppet. Ty ale narozdíl od Ansbile vyžadují znalost programovacího jazyka Ruby. Předchozí nástroj je jednodušší v tom smyslu, že pro užívání požaduje akorát znalost formátu yaml. Část práce zabývající se Secrets by mohla v budoucnu být rozšířena o užití takzvaných Vault. To je šifrovací způsob uchování citlivých údajů v externí službě.

Díky jednotlivým problémům na které jsem narazil, jsem rozšířil svoji znalost v oblasti webových aplikací. Ať už se jednalo o opravu chyby v aplikaci, díky které jsem se dozvěděl více o hlavičkách http requests, nebo o nemožnost provolání API komponenty, kde jsem zjistil nutnost vystavení správných endpoints.

Zprovoznění nefunkčních částí nasazené aplikace vyžadovalo velké úsilí. Naučil jsem se díky tomu, jak se vypořádat s častými chybami v nasazovacím procesu a v příští iteraci se jim já i čtenář budeme moci vyvarovat.

Bibliografie

1. The twelve-factor app [[online]]. 2017. Dostupné také z: https://12factor.net/cs/config.
2. Cloud-Native Configuration Management by GitOps [[online]]. 2020. Dostupné také z: https://medium.com/trendyol-tech/cloud-native-configuration-management-by-gitops-6-2401bf47418.
3. What is configuration management [[online]]. 2019. Dostupné také z: https://www.redhat.com/en/topics/automation/what-is-configuration-management.
4. Why is configuration management important? [[online]]. [B.r.]. Dostupné také z: https://www.atlassian.com/continuous-delivery/principles/configuration-management.
5. What Is Configuration Management and Why Is It Important? [[online]]. 2022. Dostupné také z: https://www.upguard.com/blog/5-configuration-management-boss.
6. Microservice Architecture [[online]]. [B.r.]. Dostupné také z: https://microservices.io/index.html.
7. Proof of concept [[online]]. [B.r.]. Dostupné také z: https://en.wikipedia.org/wiki/Proof_of_concept.
8. *BEST-PRACTICE RECOMMENDATIONSCONFIGURATION MANAGEMENT*. 2007.
9. AIELLO, Bob; SACHS, Leslie. *Configuration Management Best Practices*. Addison-Wesley Professional, [b.r.]. ISBN 978-0-321-68586-5.
10. Configuration Management Workflows [[online]]. [B.r.]. Dostupné také z: https://docs.microfocus.com/SM/9.61/Hybrid/Content/BestPracticesGuide_PD/ConfigurationManagementBestPractice/Configuration_Management_WorkFlows.htm.
11. *What is Spring*. [B.r.]. Dostupné také z: https://en.wikipedia.org/wiki/Spring_Framework.
12. *Maven*. [B.r.]. Dostupné také z: https://maven.apache.org/.
13. *What is React*. [B.r.]. Dostupné také z: https://en.wikipedia.org/wiki/React_(JavaScript_library).
14. *Npm dokumentace*. [B.r.]. Dostupné také z: https://docs.npmjs.com/about-npm.
15. *Cross-origin resource sharing (CORS) - http: MDN*. [B.r.]. Dostupné také z: https://developer.mozilla.org/en-US/docs/Web/HTTP/CORS.
16. *What is a container?* 2022. Dostupné také z: https://www.docker.com/resources/what-container/.
17. *Podman vs Docker*. 2021. Dostupné také z: https://www.imaginarycloud.com/blog/podman-vs-docker/.

18. CARNEGIE, James. *The Fundamentals of Docker Image Tags*. Content for Engineers by Engineers — Atomist Blog, 2021. Dostupné také z: https://blog.atomist.com/docker-image-tags/.
19. *Overview of docker compose*. 2022. Dostupné také z: https://docs.docker.com/compose/.
20. CHADIMA, Jirka. *Runtime configuration in web applications*. 2019. Dostupné také z: https://fragaria.cz/blog/2019/02/05/runtime-configuration-in-web-applications/.
21. 2022, Kevin Alvarez April 14; 2022, Tyler Charboneau April 13; 2022, Ajeet Singh Raina April 12. *How to deploy on Remote Docker hosts with Docker-compose*. 2021. Dostupné také z: https://www.docker.com/blog/how-to-deploy-on-remote-docker-hosts-with-docker-compose/.
22. *Docker Context*. [B.r.]. Dostupné také z: https://docs.docker.com/engine/context/working-with-contexts/.
23. *Mount*. [B.r.]. Dostupné také z: https://cloud.google.com/kubernetes-engine/docs/concepts/pod.
24. *Service*. 2022. Dostupné také z: https://kubernetes.io/docs/concepts/services-networking/service/.
25. *Replicaset*. 2022. Dostupné také z: https://kubernetes.io/docs/concepts/workloads/controllers/replicaset/.
26. *Mount*. [B.r.]. Dostupné také z: https://www.postgresql.org/.
27. *Postgres on dockerhub*. [B.r.]. Dostupné také z: https://hub.docker.com/_/postgres.
28. *How to easily create a postgres database*. 2021. Dostupné také z: https://dev.to/andre347/how-to-easily-create-a-postgres-database-in-docker-4moj.
29. *Use volumes*. [B.r.]. Dostupné také z: https://docs.docker.com/storage/volumes/.
30. *Docker Storage*. [B.r.]. Dostupné také z: https://docs.docker.com/storage/.
31. *Mount*. [B.r.]. Dostupné také z: https://en.wikipedia.org/wiki/Mount_(computing).
32. *Postgres data storage*. [B.r.]. Dostupné také z: https://www.postgresql.org/docs/8.1/storage.html.
33. *Persistent Volume*. 2022. Dostupné také z: https://kubernetes.io/docs/concepts/storage/persistent-volumes/.
34. *Distribute credentials secure*. [B.r.]. Dostupné také z: https://kubernetes.io/docs/tasks/inject-data-application/distribute-credentials-secure/.
35. *What are kubernetes secrets*. [B.r.]. Dostupné také z: https://www.mirantis.com/cloud-native-concepts/getting-started-with-kubernetes/what-are-kubernetes-secrets/.
36. *Kubelet*. [B.r.]. Dostupné také z: https://kubernetes.io/docs/reference/command-line-tools-reference/kubelet/.
37. *About AWS*. [B.r.]. Dostupné také z: https://aws.amazon.com/about-aws/.
38. *About AWS*. [B.r.]. Dostupné také z: https://docs.aws.amazon.com/eks/latest/userguide/what-is-eks.html.
39. *AWS EKS - Create Kubernetes cluster on Amazon EKS — the easy way*. [B.r.]. Dostupné také z: https://www.youtube.com/watch?v=p6xDCz00TxU.
40. *eksctl*. [B.r.]. Dostupné také z: https://eksctl.io/.
41. *Configuration and credential file settings*. [B.r.]. Dostupné také z: https://docs.aws.amazon.com/cli/latest/userguide/cli-configure-files.html.

42. *Pod count*. [B.r.]. Dostupné také z: https://github.com/awslabs/amazon-eks-ami/blob/master/files/eni-max-pods.txt.
43. *OpenID*. [B.r.]. Dostupné také z: https://openid.net/connect/.
44. *Enable IAM roles*. [B.r.]. Dostupné také z: https://docs.aws.amazon.com/eks/latest/userguide/enable-iam-roles-for-service-accounts.html.
45. *AWS load balancer controller*. [B.r.]. Dostupné také z: https://docs.aws.amazon.com/eks/latest/userguide/aws-load-balancer-controller.html.
46. *Namespaces*. [B.r.]. Dostupné také z: https://kubernetes.io/docs/concepts/overview/working-with-objects/namespaces/.
47. *ConfgiMaps*. [B.r.]. Dostupné také z: https://kubernetes.io/docs/concepts/configuration/configmap/.
48. *Ingress DNS*. [B.r.]. Dostupné také z: https://minikube.sigs.k8s.io/docs/handbook/addons/ingress-dns/.
49. *Helm documentation*. [B.r.]. Dostupné také z: https://helm.sh/docs/.
50. *Ansible documentation*. [B.r.]. Dostupné také z: https://docs.ansible.com/.

Obsah přiloženého média

- `README.txt` ..... stručný popis obsahu média
- `text` ..... zdrojová forma práce ve formátu LaTeX
- `zdrojový kód` ..... zdrojové kódy implementace
- `soubory`
  - `3 lokální spuštění` ..... instrukce pro spuštění na lokálním stroji
  - `4.1.3 docker compose úvod` ..... Vytvořené soubory pro kapitolu 4
  - `4.1.4 docker compose env` ..... Vytvořené soubory pro kapitolu 4
  - `5.1 minikube` ..... Vytvořené soubory pro kapitolu 5
  - `5.2 kubernetes škálování` ..... Vytvořené soubory pro kapitolu 5
  - `6.1.2 docker compose secrets` ..... Vytvořené soubory pro kapitolu 6
  - `6.2 kubernetes secrets` ..... Vytvořené soubory pro kapitolu 6
  - `7 aws` ..... Vytvořené soubory pro kapitolu 7
  - `8.1 kubernetes configmap` ..... Vytvořené soubory pro kapitolu 8
  - `8.2 kubernetes ingress` ..... Vytvořené soubory pro kapitolu 8
- `thesis.pdf` ..... text práce ve formátu PDF