FAKULTA INFORMAČNÍCH TECHNOLOGIÍ ČVUT V PRAZE

Zadání diplomové práce

Název:	Cloud native vývoj a jeho aplikování na projekt License Manager
Student:	Bc. Viktor Holý
Vedoucí:	Ing. Oldřich Malec
Studijní program:	Informatika
Obor / specializace:	Softwarové inženýrství
Katedra:	Katedra softwarového inženýrství
Platnost zadání:	do konce letního semestru 2023/2024

Pokyny pro vypracování

Cílem práce je nastudovat principy cloud native vývoje a aplikovat je na aplikaci License manager, která je aktuálně implementována monoliticky.

Postupujte v těchto krocích:
1) Nastudujte a popište principy cloud native vývoje, zaměřte se na použití pro webové aplikace.
2) Popište aktuální funkčnost aplikace License manager a její plánované úpravy.
3) Nastudované principy aplikujte při návrhu nové architektury backendové části aplikace.
4) Implementujte alespoň MVP (minimum viable product).
5) Výsledek otestujte vhodnými testy.
6) Zhodnoťte klady a zápory realizovaného řešení oproti původnímu.

Elektronicky schválil/a Ing. Michal Valenta, Ph.D. dne 22. prosince 2022 v Praze.

FAKULTA
INFORMAČNÍCH
TECHNOLOGIÍ
ČVUT V PRAZE

Diplomová práce

Cloud native vývoj a jeho aplikování na projekt License Manager

Bc. Viktor Holý

Katedra softwarového inženýrství
Vedoucí práce: Ing. Oldřich Malec

3. května 2023

Poděkování

Rád bych poděkoval vedoucímu práce Ing. Oldřichu Malcovi za jeho ochotu, čas a cenné rady, které mi poskytl.

Děkuji svým rodičům, bez kterých by mé studium nebylo možné a kteří mě po celou dobu podporovali. Za podporu děkuji také sestře Anet a babi Jarce.

Prohlášení

Prohlašuji, že jsem předloženou práci vypracoval samostatně a že jsem uvedl veškeré použité informační zdroje v souladu s Metodickým pokynem o dodržování etických principu 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 předpisu. V souladu s ust. § 2373 odst. 2 zákona č. 89/2012 Sb., občanský zákoník, ve znění pozdějších předpisu, tímto uděluji nevýhradní oprávnění (licenci) k užití této mojí práce, a to včetně všech počítačových programu, jež jsou její součástí či přílohou a veškeré jejich dokumentace (dále souhrnně jen „Dílo"), a to všem osobám, které si přejí Dílo užít. Tyto osoby jsou oprávněny Dílo užít jakýmkoli zpusobem, který nesnižuje hodnotu Díla a za jakýmkoli účelem (včetně užití k výdělečným účelum). Toto oprávnění je časově, teritoriálně i množstevně neomezené. Každá osoba, která využije výše uvedenou licenci, se však zavazuje udělit ke každému dílu, které vznikne (byť jen zčásti) na základě Díla, úpravou Díla, spojením Díla s jiným dílem, zařazením Díla do díla souborného či zpracováním Díla (včetně překladu) licenci alespoň ve výše uvedeném rozsahu a zároveň zpřístupnit zdrojový kód takového díla alespoň srovnatelným zpusobem a ve srovnatelném rozsahu, jako je zpřístupněn zdrojový kód Díla.

V Praze dne 3. května 2023 . . . . . .

Odkaz na tuto práci

Holý, Viktor. *Cloud native vývoj a jeho aplikování na projekt License Manager*. Diplomová práce. Praha: České vysoké učení technické v Praze, Fakulta informačních technologií, 2023.

Abstrakt

Tato diplomová práce se zaměřuje na nativní cloudový vývoj. V teoretické části popisuje jeho principy a principy architektury mikroslužeb. Teoretické poznatky jsou následně aplikovány na aplikaci pro prodej softwaru License Manager, která má v současné verzi monolitickou architekturu a je realizována nová nativní cloudová verze její backendové části.

Praktická část pokrývá všechny fáze softwarového cyklu. V analýze je popsána stávající verze a jsou revidovány požadavky na novou verzi. Návrh popisuje dělení funkcí mezi mikroslužby a jejich spolupráci. Výsledkem fáze návrhu je distribuovaný systém se šesti mikroslužbami, který je následně implementován s využitím nové sady technologií zahrnující TypeScript, gRPC, RabbitMQ, GraphQL a grafovou databázi. Výsledný kód je pokryt jednotkovými testy a funkčnost systému jako celku ověřena integračními testy. Proces testování, sestavení a vydání nových verzí je automatizován. Součástí implementace je vzorové nasazení aplikace do Kubernetes včetně nástrojů pro monitorování aplikace.

Výsledkem práce je nové řešení, které oproti původnímu umožňuje mimo jiné rychlejší změny, horizontální škálování a nabízí násobně vyšší výkon při stejném zatížení hardwaru.

Klíčová slova nativní cloudový vývoj, mikroslužby, kontejnery, Node.js, TypeScript, Kubernetes, gRPC, GraphQL, prodej softwaru

Abstract

This master thesis focuses on cloud native development. In the theoretical part it describes its principles and the principles of microservices architecture. The theoretical knowledge is then applied to a software sales application, License Manager, which has a monolithic architecture in its current version and a new native cloud version of its backend is implemented.

The practical part covers all phases of the software cycle. In the analysis, the current version is described and the requirements for the new version are revised. The design describes the division of functions between microservices and their collaboration. The design phase results in a distributed system with six microservices, which is then implemented using a new set of technologies including TypeScript, gRPC, RabbitMQ, GraphQL and a graph database. The resulting code is covered by unit tests and the functionality of the system as a whole is verified by integration tests. The process of testing, building and releasing new versions is automated. The implementation includes a sample deployment of the application to Kubernetes, including tools for monitoring the application.

The result of the work is a new solution that, compared to the original one, allows, among other things, faster changes, horizontal scaling, and offers multiple times better performance under the same hardware load.

Keywords cloud native development, microservices, containers, Node.js, TypeScript, Kubernetes, gRPC, GraphQL, software sales

Obsah

Seznam obrázků

Seznam tabulek

Úvod

Cloudové technologie hýbou světem informačních technologií. Zřízení nebo změna počítačové infrastruktury na žádost v řádu několika vteřin přináší nevídanou flexibilitu a tím i zcela nový pohled na provoz webových aplikací.

Spolu s příchodem flexibilní počítačové infrastruktury začala i transformace, která podobnou flexibilitu přináší i do architektury a způsobu vývoje aplikací, které budou na cloudových platformách provozovány. Aplikace, které plně využívají potenciálu cloudu, se označují jako nativní cloudové. Na jejich vývoj se zaměřuje tato diplomová práce.

První dvě kapitoly práce budou teoretickým úvodem do problematiky. V první definuji nativní cloudový vývoj, jeho vlastnosti, způsoby jeho dosažení a požadavky, které nativní cloudová aplikace musí splňovat. V druhé kapitole popíšu architekturu mikroslužeb, která je s nativním cloudovým vývojem spojována. Zaměřím se na dělení logiky a dat mezi mikroslužby, veřejné rozhraní a zabezpečení aplikace založené na této architektuře.

V praktické části se vrátím k projektu License manager, který jsem realizoval v rámci své bakalářské práce. License manager je webová aplikace pro prodej uživatelsky konfigurovatelných informačních systémů společnosti Jagu, s.r.o. Současná verze aplikace je implementovaná monoliticky. Hlavním cílem této práce je navrhnout a implementovat novou, nativně cloudovou verzi backendu aplikace License manager. Dílčími cíli jsou:

1. analýza funkčnosti stávající aplikace,
2. návrh nové verze založené na architektuře mikroslužeb,
3. implementace alespoň minimální, prakticky využitelné sady funkčních požadavků,
4. srovnání vlastností s původní verzí.

V kapitole Analýza představím funkcionalitu, technické řešení stávající verze a provedu revizi požadavků. Na analýzu navážu kapitolou Návrh, ve

které se budu věnovat tvorbě zcela nové architektury aplikace. V kapitole Implementace popíšu volbu nové sady technologií, samotnou realizaci, proces sestavení, vydání a nasazení aplikace. Ověření správné funkčnosti aplikace a jejich vlastností se budu věnovat v kapitole Testování. V poslední kapitole práce provedu srovnání nového a stávajícího řešení. Popíšu rozdíly v procesu vývoje, jeho efektivitě a připravenosti na další rozvoj. Na základě měření srovnám výkonnost obou verzí.

Výsledná nativně cloudová backendová část aplikace License manager bude využitelná ke kompletnímu zajištění procesu on-line prodeje softwaru. Poslouží jako kvalitní základ pro budoucí rozšiřování funkcionalit. Po zhotovení grafického uživatelského rozhraní bude možné aplikaci využít nejen ve společnosti Jagu.

Kapitola 1

Cloud native

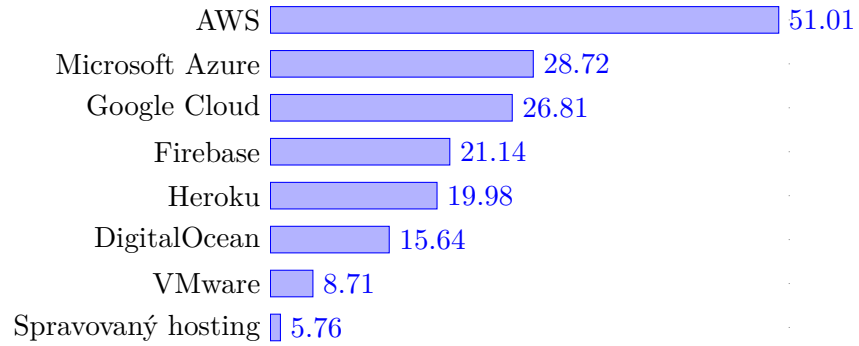
Tato práce se zaměřuje na cloud native vývoj. V první kapitole se proto zaměřím na definování tohoto přístupu, jeho pilíře, vlastnosti, výhody i nevýhody. Popíšu, proč se jedná o stále používanější přístup ve vývoji a provozu webových aplikací a co je nutné udělat, aby softwarový projekt byl „cloud native“.

Cloudové platformy a vývoj pro ně je bezesporu velkým tématem současného softwarového inženýrství. Dle společnosti Statista se v roce 2023 očekává růst segmentu cloudové infrastruktury o 30,5 %. [1] Vývoj aplikací pro cloudové platformy je stále populárnější i mezi vývojáři. Trendům v softwarovém inženýrství se věnuje například každoroční anketa Developer Survey, kterou organizuje vývojářské fórum StackOverflow. Výsledky z roku 2022 říkají, že minimálně nadpoloviční většina vývojářů, kteří se do průzkumu zapojili, pracuje s nějakou cloudovou platformou. Na grafu 1.1 je pro vybrané platformy znázorněn podíl vývojářů (z celkových 56 553), kteří s danou platformou intenzivně pracují. [2]

Popularita cloudových platforem je tedy zřejmá a mnoho aktivně vyvíjených aplikací bude provozováno právě na některé z nich. Kdybych všem vývojářům, kteří s cloudem pracují, položil otázku: „Co je to cloud native?“, dostal bych mnoho různých odpovědí. Za autoritu v oblasti nativního cloudového vývoje lze považovat organizaci Cloud Native Computing Foundation, která *cloud native* popisuje následovně:

„Cloudové technologie umožňují organizacím vytvářet a provozovat škálovatelné aplikace v moderních dynamických prostředích, jako jsou veřejné, soukromé a hybridní cloudy. Příkladem tohoto přístupu jsou kontejnery, sítě služeb, mikroslužby, neměnná infrastruktura a deklarativní rozhraní API. Tyto techniky umožňují přináší vázané systémy, které jsou odolné, spravovatelné a pozorovatelné. V kombinaci s robustní automatizací umožňují inženýrům provádět změny s velkým dopadem často, předvídatelně a s minimální námahou.“ [3]

Cloud native je často zaměňován se slangovým pojmem *cloud*. Oba pojmy označují ale něco jiného. Cloud označuje *cloud computing*, tedy dodávku



Obrázek 1.1: Podíl vývojářů zúčastněných ankety Developer Survey v procentech, kteří intenzivně pracují s danou platformou [2]

výpočetní infrastruktury na vyžádání – jako službu. *Cloud native* není o infrastruktuře, ale o architektuře a přístupu k vývoji aplikací tak, aby byly optimalizované pro prostředí cloudu. [4]

Cloud native je tedy souhrn technologií, procesů a návrhových vzorů, jejichž vhodná kombinace vede k vytvoření dobře škálovatelného systému složeného z malých nezávislých částí, které jsou jednoduše udržovatelné a snadno nasaditelné. Cílem je také snadnější rozdělení práce mezi týmy a kratší doba implementace a nasazení nových funkcí. [5]

1.1 Distribuovaný systém

Nativní cloudové aplikace se skládají z malých nezávislých komponent, které běží na jednom či více počítačích, a které mezi sebou komunikují po síti. [5] Jedná se tedy o distribuovaný systém. Tato vlastnost je zásadní a umožňuje dosáhnutí cílů nativního cloudového vývoje, ale přináší i několik negativ.

Při vývoji nativních cloudových aplikací je nutné zohlednit, že žádná síť není 100% spolehlivá – může docházet (a v praxi dochází) k výpadkům sítě, propustnost sítě není neomezená a síťová volání mají nezanedbatelnou latenci. [5] V neposlední řadě, komunikace po síti představuje bezpečnostní riziko a hrozí tak např. man-in-the-middle útok (Útočník naruší komunikaci a pro obě strany předstírá druhou stranu. Získá tak obsah komunikace. [6]).

Distribuovaný systém znamená vyšší celkovou komplexitu systému proti monolitické aplikaci. Jednotlivé služby musí zpřístupnit své funkce formou síťového API, zajistit autorizaci požadavků či při volání jiné služby definovat chování v případě její nedostupnosti. Síťová volání mají také vyšší režii oproti volání funkce v rámci jedné aplikace. Je nutné provést serializaci dat, vytvořit

spojení s volanou službou, přenést požadavek, případně vyčkat na odpověď. [5]

Síťová volání jsou zdrojem nestability v distribuovaném systému. Na druhou stranu, samotné rozdělení logiky do nízce provázaných a vysoce soudržných služeb implikuje dobrou izolaci chyb. Nastane-li chyba v jedné službě, která zapříčiní její ukončení, zbytek systému zůstává nadále funkční a dostupný pro uživatele, přestože některé funkce kvůli výpadku jedné služby nebudou použitelné. [5] Administrátoři distribuovaného systému mají lepší přehled o funkčnosti a zatížení jednotlivých částí a mohou rychleji reagovat na případné defekty.

1.1.1 Škálovatelnost aplikace

Jedním z hlavních cílů nativního cloudového vývoje je dosažení dobré škálovatelnosti aplikace. Škálovatelnost je měřítkem schopnosti systému zvyšovat nebo snižovat výkon a náklady v reakci na změny v požadavcích na zpracování. [7]

Existují dva způsoby, jak aplikaci škálovat. Vertikální škálování spočívá v navýšení výpočetního výkonu stroje, na kterém aplikace běží. [8] Vyšší výkon serveru znamená rychlejší zpracování požadavků. Výkon jednoho serveru není možné navyšovat do nekonečna a zároveň aplikace nemusí být schopná vyšší výkon využít.

Řešením vhodným pro nativní cloudové aplikace je využití horizontálního škálování. Při něm se, namísto navýšení výkonu serveru, spustí další instance dané komponenty (na stejném či jiném výpočetním uzlu). Aby horizontální škálování bylo možné, musí být aplikace bezestavová. Dalšími požadavky na nativní cloudové aplikace se budu zabývat v podkapitole 1.3. [5, 8]

1.1.2 CAP teorém

Horizontálním škálováním vzniká distribuovaný systém, ve kterém mohou existovat komponenty se sdílenými daty. CAP teorém říká, že distribuovaný systém se sdílenými daty může mít pouze dvě z následujících tří vlastností:

- Konzistence
- Vysoká dostupnost
- Odolnost k přerušení spojení [5]

Sítě jsou nespolehlivé a aplikace musí být odolná vůči přerušení spojení. V praxi je tedy na výběr pouze jedna z prvních dvou vlastností. [5]

Konzistence znamená, že bez ohledu na to, která fyzická komponenta poskytne odpověď na požadavek, data jsou vždy stejná. Vysoká dostupnost vyjadřuje odolnost systému vůči výpadkům. Dosažení konzistence v praxi implikuje mít jednu fyzickou kopii dat (například jednu databázi). Dojde-li

k výpadku databáze, přestanou fungovat všechny služby. Takový systém tedy nemá vysokou dostupnost. Naopak, pokud každá instance dané komponenty má vlastní kopii dat (vlastní databázi), v případě výpadku jedné komponenty vyřizují požadavky zbývající. Kvůli výpadku ale různé komponenty mají různá data, tedy systém není konzistentní. [5, 9]

1.2 Agilita vývoje

Při změnovém řízení (úpravě stávajících nebo přidání nových funkcionalit) je nutné provést všechny fáze životního cyklu softwaru (analýza požadavků, návrh, implementace, testování, nasazení a údržba [10]). V případě komplexních monolitických aplikací se jednotlivé fáze stávají čím dál časově náročnější – návrh a implementace musí dbát na správnou integraci do současného kódu, testování musí být provedeno na celé aplikaci. Kvůli časové náročnosti změnového řízení monolitických aplikací obvykle vydání nové verze obsahuje více změn a je prováděno v delších intervalech. Pokud je po nasazení objeven problém, je nutné přejít na starší verzi, tedy stáhnout veškeré změny, které byly v rámci dané verze uvolněny. [5, 9]

Protože jsou nativní cloudové aplikace složené z mnoha nezávislých aplikací (služeb), jsou stejné fáze vývoje aplikované na mnohem menší jednotky, a tedy i provedené rychleji. Díky nezávislosti služeb může na každé pracovat jiný tým. Tyto specializované týmy velmi dobře znají části, za které jsou zodpovědné. To opět zrychluje práci i adaptaci nových členů týmu. [5]

Důraz je kladen na co nejrychlejší nasazení nových změn do produkce. Aktualizace nejsou vypouštěny ve velkých balících, ale změny jsou nasazovány ihned, jakmile jsou dokončeny a otestovány. Aby tento přístup byl možný, je nutné automatizovat vše, co lze (sestavení, testování, nasazení). Tyto činnosti jsou souhrnně označovány jako DevOps a blíže je popíšu v praktické části práce (5.10).

Průběžné nasazování nových verzí jednotlivých služeb přináší problém se vzájemnou kompatibilitou. Vývojáři musí dbát na zpětnou kompatibilitu rozhraní, případně provozovat současně více verzí API či služeb. [5]

Technologie se neustále vyvíjejí a v průběhu času se mohou objevovat programovací jazyky, frameworky, knihovny a nástroje, které budou lépe splňovat naše požadavky. Převést monolitickou aplikaci na jinou sadu technologií je velmi náročný úkol, který může znamenat i kompletní přepsání aplikace. Naopak nativní cloudovou aplikaci je možné aktualizovat postupně, po jednotlivých službách, nebo dokonce pro každou službu použít jinou sadu technologií. V takovém případě je ovšem nutné hlídat míru diverzifikace, neboť systém jako celek by se stal hůře udržovatelným nebo by byla znemožněna vzájemná výpomoc mezi týmy. [5, 9]

1.3 Twelve-factor manifest

The twelve-factor app manifest je považovaný za základ moderního vývoje nativních cloudových aplikací. Byl poprvé představen inženýry z Heroku (poskytovatel cloudových služeb) a představuje 12 požadavků založených na osvědčených postupech vývoje nativních cloudových aplikací. Následuje stručné představení všech požadavků. [5]

Zdrojový kód

Zdrojový kód je vždy verzovaný pomocí verzovacího systému (například Git). Vztah mezi kódem a aplikací je vždy 1:1 (jedna aplikace má jeden kód a jeden kód odpovídá jedná aplikaci). Sdílený kód mezi aplikacemi musí být vložen prostřednictvím správce závislostí (maven, npm, atd.). Aplikace založená na jednom kódu může být nasazena ve více prostředích. [11]

Závislosti

Aplikace vždy explicitně deklarují všechny své závislosti. Závislosti jsou izolované, aby neprosakovaly z jiných částí systému. Aplikace se nespoléhají na existenci žádných systémových nástrojů – pro běh aplikace musí stačit deklarované závislosti. [11]

Konfigurace

Aplikace dodržuje přísné oddělení konfigurace od kódu. Konfigurace je vše, co se liší mezi nasazeními aplikace (údaje pro připojení k databázi, klíče, atd.), a jsou uložené v proměnných prostředích (*env vars*). [11]

Podpůrné služby

S podpůrnými službami (taková, služba, kterou aplikace konzumuje přes síť v rámci normálního běhu například databáze, messaging systémy, emailové servery, atd.) aplikace nakládá jako s připojeným zdrojem. Aplikace nedělá rozdíly mezi lokální službou a službou třetí strany – lokální službu je kdykoli možné nahradit službou třetí strany a naopak. Připojené zdroje mohou být dle libosti odpojovány či nahrazovány za jiné beze změn v kódu. [11]

Sestavení, vydání, spuštění

Aplikace striktně rozlišuje mezí fází sestavení (převedení zdrojového kódu na spustitelný balíček), vydání (kombinace sestavení a konfigurace pro dané nasazení) a spuštění (spuštění aplikace v daném prostředí). Spuštění by mělo být přímočaré, aby mohlo být automatizované bez dohledu administrátorů. [11]

Procesy

Procesy aplikace jsou bezestavové a nic nesdílejí. Veškerá data jsou uložena v některé podpůrné službě. Aplikace nikdy nespoléhá, že jsou potřebná data na disku. [11]

Vazba s portem

Aplikace nikdy nespoléhá na vložení webserveru do běhového prostředí, ale sama obsahuje webserver (obvykle formou knihovny) a naslouchá požadavkům na daném portu. [11]

Souběh

Aplikace je škálována do šířky. V případě potřeby je spuštěna v několika bezestavových instancích. [11]

Zahoditelnost

Procesy aplikace mohou být kdykoliv bezpečně zapnuty či vypnuty. Aplikace by měla být spuštěna v co nejkratším čase. Tato vlastnost umožňuje efektivní škálování a rychlé změny v konfiguracích. [11]

Podobnost vývojového a produkčního prostředí

Rozdíly mezi vývojovým a produkčním prostředím by měly být co nejmenší. Vývojáři v lokálním prostředí využívají stejné podpůrné služby jako v produkčním. [11]

Logy

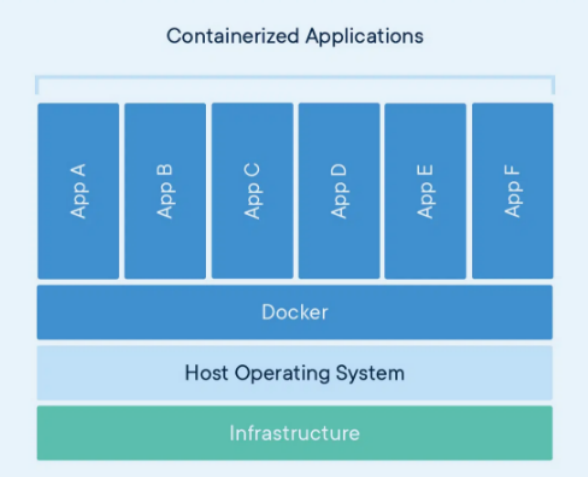
Aplikace pracuje s logy jako s proudem událostí. Aplikace se nestará o směrování logů či jejich zápis do specifikovaného souboru, ale vypisuje logy na standardní výstup. O sběr logů se stará běhové prostředí. [11]

Administrátorské procesy

Jednorázové administrátorské procesy (např. databázové migrace) jsou dodávány společně s aplikací a spouštěny ve stejném prostředí. [11]

1.4 Komponenty nativních cloudových aplikací

Z hlediska cloudové infrastruktury může být distribuovaný systém složen ze dvou základních typů komponent – kontejnerů a funkcí. [5] Ty se liší zejména granularitou rozdělení funkcionalit mezi dílčí části.



Obrázek 1.2: Kontejnerizace aplikací

1.4.1 Kontejnery

V oblasti nativních cloudových aplikací se kontejnery typicky pojí s architekturou mikroslužeb (té se budu detailně věnovat v kapitole 2). Nejznámější nástroj pro práci s kontejnery je Docker, který kontejnery definuje takto: *„Kontejner je standardní softwarová jednotka, která balí kód a všechny jeho závislosti tak, aby aplikace rychle a spolehlivě fungovala z jednoho výpočetního prostředí do druhého. Obraz kontejneru Docker je lehký, samostatně spustitelný balíček softwaru, který obsahuje vše potřebné ke spuštění aplikace: kód, běhové prostředí, systémové nástroje, systémové knihovny a nastavení.“* [12] Kontejnery využívají jádro hostitelského operačního systému a obsahují pouze aplikaci, její závislosti a potřebné systémové nástroje. Velikosti kontejnerů jsou v řádu desítek až stovek megabytů, zatímco klasické virtuální stroje, které mimo aplikaci obsahují kompletní operační systém, dosahují velikosti až desítek gigabytů. [12]

Schéma technologického řešení kontejnerizace je na obrázku 1.2. Všechny kontejnerizované aplikace spuštěné na jednom PC sdílí hardware (fyzický nebo virtuální), jádro operačního systému a vrstvu kontejnerové virtualizace. Funkce hostitelského operačního systému jsou dostupné všem kontejnerům. Výhodou tohoto řešení je, že kontejnery nemusí provozovat vlastní operační systém – šetří se tak výpočetní prostředky i místo na disku. Nevýhodou je, že kontejnery musí být založené na stejném operačním systému a platformě jako hostitelský stroj (například na operačním systému Linux mohou být spuštěny pouze Linuxové kontejnery; na operační systému pro procesory ARM nemůže být spuštěný kontejner založený na operačním systému pro platformu x86, atd.). [13]

Balíčku, který obsahuje aplikaci se všemi jejími závislostmi, se říká obraz

(*image*). Kontejnery jsou běžící instance obrazů. Předpis pro vytvoření obrazu je zapsaný v souboru zvaném *Dockerfile*. Obsahuje jednotlivé kroky sestavení, například kopírování souborů z lokálního adresáře, spuštění příkazů/skriptů, vystavení portů, definování příkazu, který se provede při startu kontejneru. Obrazy mohou být uloženy lokálně na počítači vývojáře nebo nahrány do veřejného či soukromého repozitáře.

Pro použití v cloudu jsou kontejnery vhodné zejména proto, že obsahují vše potřebné pro spuštění aplikace v relativně malém balíčku. Nasazení je tedy rychlé a snáze automatizovatelné.

1.4.2 Funkce

Druhým způsobem, jak dělit monolit a provozovat dílčí části distribuovaného systému v prostředí cloudu, jsou funkce. Funkce se obvykle pojí s pojmem *serverless* („bez serverů“) – jedná se způsob nasazení, při kterém se také používají servery, ale jejich správa je kompletně abstrahována od zákazníka. Zřizování služby a její škálování je plně v režii provozovatele cloudové platformy. [14]

Funkce jsou velmi malé jednotky distribuovaného systému, které dělají jednu jedinou věc (případně reagují na jediný druh události). Výhodou tohoto řešení je, že umožňuje velmi jemné škálování, které je v režii provozovatele cloudu. Obvykle je navíc zákazníkovy účtován pouze čas, kdy funkce provádí nějaký výpočet (pokud je v nečinnosti, poplatky účtovány nejsou). Nevýhodou je, že způsoby, jakým se funkce nasazují, se mezi poskytovateli cloudu liší. Vzniká tak silná vazba aplikace na vybraného poskytovatele, tzv. *vendor lock-in*, a to je v rozporu s požadavkem na přenositelnost nativních cloudových aplikací. [5, 14] Distribuovaným systémům založeným na funkcích se v této práci nebudu dále věnovat.

1.5 Komunikace služeb

V předchozích kapitolách jsem několikrát uvedl, že nativní cloudové aplikace se skládají z nezávisle nasaditelných částí – služeb, které mezi sebou komunikují po síti. V této kapitole se zaměřím na komunikaci samotnou. Prozatím se oprostím od konkrétních protokolů a technologií (těm se budu věnovat v praktické části, kapitole 5.4).

1.5.1 Synchronní a asynchronní komunikace

Komunikační vzory můžeme rozdělit podle mnoha kritérií. Prvním kritériem je prodlení zpracování. Komunikace se podle tohoto kritéria dělí na synchronní a asynchronní. Při synchronní komunikaci volající očekává, že volaný je k dispozici, požadavkem se začne zabývat okamžitě po jeho přijetí a odpověď odešle (stejným spojením) okamžitě, jakmile má k dispozici výsledek požadavku. Příkladem synchronní komunikace v běžném životě je telefonní hovor. [15]

Asynchronní komunikace naopak neklade požadavek na stálou dostupnost volaného a je obvykle realizována pomocí prostředníka, který zprávu volanému předá, jakmile je k dispozici. Volaný může začít požadavek zpracovávat ihned, nebo zpracování odložit na později. Obdrží-li volaný zpráv více, je na jeho rozhodnutí, v jakém pořadí je bude zpracovávat. Je-li očekávána odpověď na požadavek, je obvykle zaslána opět asynchronním způsobem. Volající pro provedení dalších kroků nepotřebuje okamžitou odpověď na požadavek a pokračuje v provádění dalších úkolů. Příkladem asynchronní komunikace z běžného života je e-mail. [15]

Oba způsoby se také liší v tom, na kom leží zodpovědnost za dokončení požadavku. V případě synchronní komunikace kontroluje dokončení volající, zatímco v asynchronní leží zodpovědnost na volaném. Pro názornost použiji opět analogii s telefonním hovorem a e-mailem. Pokud volaný nepřijímá hovor, vypadne spojení během hovoru nebo volaný nemůže v daný moment požadavek splnit, musí volající zkusit zavolat znovu později. Odešle-li ale volající požadavek e-mailem, předpokládá, že ho volaný dříve nebo později vyřídí a o požadavek se dále nezajímá (dokud nepřijde případná odpověď). Volaný nechává e-mail ve schránce, dokud ho nezpracuje. E-mail může zůstat ve schránce i několik pracovních dní (tedy i v době, kdy volaný není aktivní). Pokud je očekávána odpověď, volaný ji odešle opět e-mailem (přestože v určitých situacích může zavolat telefonem – tedy pro odpověď použít synchronní komunikaci).

Synchronní komunikace je rychlejším, jednodušší na implementaci a méně náročná na výpočetní zdroje. Vyžaduje ale vysokou dostupnost volaných služeb a nízkou latenci komunikace. Asynchronní komunikace je vhodná, pokud volané služby nejsou vysoce dostupné nebo pokud je zpracování požadavku vyžaduje hodně času. Oproti synchronní je ale obecně pomalejší a často vyžaduje třetí stranu, která komunikaci zprostředkuje. [15]

1.5.2 Vzory požadavek-odpověď a publikace-odběr

Dalším kritériem pro klasifikaci typy komunikace je, zda-li volající očekává odpověď. Základní vzorem je *požadavek-odpověď*. Volající pošle požadavek a očekává odpověď. Přímočaré je pro tento vzor využít synchronní komunikaci, možná je ale i asynchronní, potom požadavek obsahuje jednoznačný identifikátor, kterým je následně označena i odpověď. [5]

Druhým vzorem je publikace-odběr (známý pod označením *PubSub*). PubSub je typickým vzorem asynchronní komunikace. Do komunikace vstupují 3 typy účastníků: publikující (*publihser*), odebírající (*subscriber*) a zprostředkovatel (*message broker*). [5]

Publikující prostřednictvím *brokeru* publikuje zprávu o události (s potřebnými daty). Uvnitř *brokeru* existuje pro každý typ události takzvaný *topic*, do kterého jsou zprávy publikovány. Odebírají se mohou přihlásit k odběru *topiců*, které jsou pro jejich činnost relevantní. Jakmile je do *topicu* publikována

zpráva, je rozeslána všem odebírajícím. Existují různé implementace PubSub s rozdílným chováním v některých aspektech (např. perzistence zpráv). [5, 16]

Vzor požadavek-odpověď je vhodný například pro operace, kdy je požadovaná interaktivita mezi uživatelem a aplikací (uživatel provede akci v UI a očekává reakci v co nejkratším čase). Nevýhodou tohoto vzoru je, že vzniká těsné spojení mezi službami – volající musí vědět o volaném, znát jeho API a volaný musí být dostupný, jinak volající nemůže dokončit operaci, jíž je požadavek součástí. *PubSub* naopak vede na nízkou provázanost služeb. Publikující služba nemá o odebírajících žádné informace a napojená je pouze na zprostředkovatele. Nevýhody *PubSub* jsou shodné s výše zmíněnými nevýhodami asynchronní komunikace. [5]

Oba vzory mají své výhody, nevýhody i rozdílné využití. Nativní cloudové aplikace běžně používají kombinaci obou.

1.6 Automatizace procesu vývoje

Jedním z hlavních pilířů nativních cloudových aplikací je automatizace co největší části softwarového cyklu. Automatizace je nezbytná, chceme-li dosáhnout flexibility, rychlých a častých dodávek nových verzí. [5, 17]

Analýzu požadavků, návrh a implementaci je sice možné podpořit specializovanými nástroji, ale plně automatizovat nejdou. Zaměřme se proto na fáze testování, nasazení a monitorování.

1.6.1 Průběžná integrace

Průběžná integrace je v IT světě známá pod anglickým termínem *continuous integration (CI)*. Jak už název napovídá, jejím účelem je průběžná integrace nových změn do stávající aplikace. Při integraci je nutné aplikaci se změnami sestavit a otestovat, zda se změny chovají dle očekávání a nemají žádný negativní vliv na současnou aplikaci. Průběžná integrace je součástí procesu sloučení větve se změnami do hlavní vývojové větve. [5]

Ve fázi CI se spouští automatizované jednotkové (*unit*) testy a testy služeb. Jednotkové testy testují malé části kódu (např. funkce/metody). Jednotky jsou testovány odděleně, pokud jednotka závisí na jiné, je tato závislost simulována. Testy služeb jsou komplexnější a testují úspěšné provedení procesů, které služba má podporovat (např. provedení objednávky, registrace nového uživatele, atd.). Test služby netestuje integraci s dalšími službami. Testování se budu dále věnovat v kapitole 6). [5]

Je-li testování úspěšné, lze aplikaci se změnami (alespoň do určité míry) považovat za funkční a změny způsobilé k přijetí. Po sloučení s hlavní větví je provedeno sestavení kontejneru a jeho publikování do registru. Než se přistoupí k nasazení aktualizované aplikace do vývojářského prostředí, provádí se integrační testy. Ty jsou podobné testům služeb, ale testuje se funkcionalita včetně spolupráce mezi službami. [5]

1.6.2 Průběžné dodávky

Průběžné dodávky, nebo také nepřetržité doručování, známé pod anglickým termínem *Continuous delivery (CD)* je: *„schopnost doručit nové funkce, změny konfigurace, opravy chyb či experimentální funkce do produkčního prostředí a do rukou uživatelů bezpečně, rychle a udržitelným způsobem.“* [18] Jedná se o navazující fázi na CI (viz předchozí podkapitola).

Primárním cílem CD je hladké nasazení nových verzí bez přerušení provozu aplikace. Bezpečného nasazení může být dosaženo například využitím metody *blue-green*. To minimalizuje prostoje pomocí dvou identických produkčních prostředí. Jedno je využíváno pro obsluhu uživatelů a druhé je předprodukční/testovací. Změny jsou nasazeny do předprodukčního prostředí. Po jejich otestování se provoz přepne na toto prostředí (stává se produkčním) a změny se nasadí i do druhého. Tato technika umožňuje snadný návrat k původní verzi v případě problémů, ale vyžaduje migrace změn mezi databázemi. [19]

Komplexita CD se může zásadně lišit a odvíjí se od množství prostředí, do kterých je nutné změny nasadit, či složitosti schvalovacího procesu (například povolení nasazení změn do dalšího prostředí bližšího produkčnímu). Obecně můžeme CD rozdělit na 3 fáze – nasazení, vydání, a fáze po vydání.

Úkolem deploy fáze je nasadit sestavenou aplikaci (například Docker image vzniklou v rámci CI) do některého prostředí. Konkrétní podoba nasazení závisí na platformě, do které je prováděno. V kapitole 5.11 se budu prakticky věnovat nasazení do Kubernetes. [5]

Jakmile jsou k dispozici výsledky testování nasazené aplikace (např. integrační testování, testování uživatelského rozhraní, testování výkonu, atd.) začínám fáze vydání. To je realizováno již zmíněnou technikou *blue-green* a směrováním části provozu do nové verze s postupným navyšováním podílu v případě, že se neobjeví žádné problémy. Během této fáze je nutné aplikaci pečlivě monitorovat, aby případné incidenty byly okamžitě odhaleny a nedošlo ke škodám.

V poslední fázi, je aplikace nadále monitorována a analyzována zpětná vazba od uživatelů.

1.6.3 Monitorování distribuovaného systému

Jak jsem uvedl, aplikaci je nutné během vydání pečlivě monitorovat. Potřeba kontroly ale přetrvává po celou dobu běhu aplikace. U distribuovaných systémů je nutné mít přehled o funkci jednotlivých částí i stavu sítě, aby se odhalila úzká hrdla systému.

Pro dobrý přehled o aplikaci potřebujeme dva druhy dat: logy a metriky. Logy jsou záznamy o událostech v aplikaci a jsou jedním z faktorů *twelve-factor* manifestu (viz 1.3). Dávají nám povědomí o stavu aplikace, probíhajících operacích či chybách. Tyto informace jsou nezbytné při řešení problémů. Kvalitní logy mají jednotný formát, který je strojově čitelný a mimo samotné

zprávy obsahují také metadata pro klasifikaci záznamu (např. typ, čas, uživatele, IP adresu). Služby v rámci nativní cloudové aplikace by neměly obsahovat logiku pro distribuci logů – tento úkol by měl být přenesen na platformu, ve které je distribuovaný systém provozován. [20]

Druhým typem sbíraných údajů jsou metriky, tedy kvantifikující data. Metriky pomáhají při ladění a optimalizaci výkonu distribuovaného systému. Typickými metrikami jsou podíl požadavků končících chybou, frekvence požadavků, latence zpracování nebo vytížení služby.

Na trhu existuje celá řada nástrojů (s otevřeným zdrojovým kódem i komerčních) pro podporu či automatizaci monitoringu aplikací, například Grafana (vizualizace metrik), Prometheus (open source nástroj pro monitorování a upozornění na incidenty), Sentry (nástroj pro monitorování a správu chyb v kódu).

1.6.4 Orchestrace kontejnerů

Poslední oblastí automatizace, kterou se budu v této kapitole věnovat, je orchestrace kontejnerů. *„Orchestrace je automatizovaná konfigurace, správa a koordinace počítačových systémů, aplikací a služeb, která IT týmům umožňuje snadněji spravovat složité úkoly a pracovní postupy.“* [21]

Orchestrace kontejnerů se vztahuje k samotnému provozu nativních cloudových aplikací. Automatizuje poskytování, nasazení, síťování, škálování, dostupnost a správu životního cyklu kontejnerů. [22] Existuje celá řada nástrojů pro orchestraci kontejnerů, například Openshift od společnosti Red Hat, Nomad od Hashicorp. Jednoznačně nejpoužívanější je Kubernetes se 77% podílem na trhu v roce 2020. [23, 24]

Kubernetes (zkratka *k8s*) je projekt s otevřeným zdrojovým kódem spravovaný Cloud Native Computing Foundation. Mimo jiné umožňuje bezpečné nasazení nové verze bez přerušení provozu služby a případný návrat k původní verzi, dojede-li k problémům, manuální i automatické horizontální škálování, vyvažování zátěže mezi instancemi, detekci chyb a automatické obnovení vadného kontejneru. [25] Používání Kubernetes se budu více věnovat v praktické části práce (kapitole 5.11).

Kapitola 2

Mikroslužby

V předchozí kapitole jsem se věnoval nativním cloudovým aplikacím, uvedl jsem jejich vlastnosti a obecné principy, jak těchto vlastností dosáhnout. Jedním z hlavních rysů nativních cloudových aplikací je, že se jedná o distribuovaný systém. V kapitole 1.4 jsem uvedl, že nativní cludovou aplikaci lze provozovat jako soubor kontejnerů, funkcí nebo jejich kombinace. Toto rozdělení primárně zohledňuje způsob provozu, samotný návrh softwaru až v druhé řadě. V této diplomové práci se věnuji nativním cloudovým aplikacím založených na kontejnerech. Tento způsob provozu se obvykle pojí s architekturou mikroslužeb, které je věnovaná tato kapitola.

Než přistoupím k popisu vlastností, je nutné samotný pojem *mikroslužby* definovat. Univerzální a všemi akceptovaná definice mikroslužeb neexistuje, proto uvedu tři definice od třech uznávaných společností, působících v oblasti cloudu.

Společnost Microsoft definuje mikroslužby takto: *„Mikroslužby popisují architektonický proces vytváření distribuované aplikace ze samostatně nasaditelných služeb, které vykonávají specifické obchodní funkce a komunikují prostřednictvím webových rozhraní. Týmy DevOps uzavírají jednotlivé části funkcí do mikroslužeb a budují větší systémy kombinováním mikroslužeb jako stavebních bloků.“*[26]

Amazon: *„Mikroslužby představují architektonický a organizační přístup k vývoji softwaru, kdy se software skládá z malých nezávislých služeb, které komunikují prostřednictvím přesně definovaných rozhraní API. Tyto služby vlastní malé, samostatné týmy. Architektura mikroslužeb usnadňuje škálování a zrychluje vývoj aplikací, což umožňuje inovace a zrychluje uvádění nových funkcí na trh.“*[27]

IBM: *„Mikroslužby (nebo architektura mikroslužeb) je nativně cloudový architektonický přístup, v němž se jedna aplikace skládá z mnoha volně provázaných a nezávisle nasaditelných menších komponent nebo služeb. Tyto služby obvykle mají svůj vlastní technologický stack, včetně databáze a modelu správy dat; komunikují mezi sebou prostřednictvím kombinace rozhraní REST API,*

streamování událostí a zprostředkovatelů zpráv a jsou uspořádány podle obchodních schopností.“[28]

Mikroslužby jsou tedy malé komponenty distribuované aplikace, které spolu komunikují, jsou samostatně nasaditelné a společně vytváří funkční celek dostupný uživatelům. Každá mikroslužba má na starost určitou byznysovou funkcionalitu. Aby se maximalizovala efektivita vývoje, pojí se s architekturou mikroslužeb i způsob organizace práce, kdy každou mikroslužbu vyvíjí jeden tým a vzájemná kompatibilita je zaručena předem domluveným rozhraním.

Definice od IBM zmiňuje, že mikroslužby jsou *„nativně cloudový architektonický přístup“*. Naopak v definici nativního cloudového vývoje v kapitole 1 jsou mikroslužby uvedené jako jedna z technik, jak vytvářet cloudové aplikace. Souvislost mezi těmito dvěma pojmy je tedy zřejmá.

Architektura mikroslužeb přímo podporuje dosažení požadovaných vlastností cloudových aplikací. Mikroslužby umožňují rozdělení práce mezi úzce specializované týmy a paralelní práci na všech mikroslužbách. Díky nezávislosti mikroslužeb je možné často nasazovat nové změny, a to bez ohledu na stav nasazení nových verzí spolupracujících mikroslužeb. Mikroslužby ve formě kontejnerů je snadné nasadit do cloudového prostředí.

2.1 Architektura systému

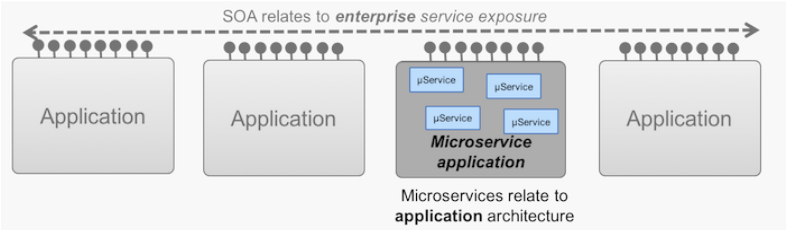
V této kapitole se zaměřím podrobněji na architekturu samotnou, jakým způsobem je rozdělena logika, jak spolu mikroslužby spolupracují, či jak je distribuovaný systém zpřístupněn uživateli.

2.1.1 Servisně orientovaná architektura vs. architektura mikroslužeb

V 90. letech 20. století vznikla servisně orientovaná architektura (zkr. SOA), která je do jisté míry architektuře mikroslužeb podobná a nezřídka jsou tyto pojmy zaměňovány. Vymezení rozdílu pomůže i s vysvětlením mikroslužeb samotných. [29]

Servisně orientovaná architektura je přístup k vývoji aplikací pomocí nízce provázaných znovupoužitelných softwarových komponent. Každá komponenta zajišťuje konkrétní obchodní funkci a předpokládá se její opakované použití napříč podnikem. Integrace služeb je zajištěna pomocí sběrnice podnikových služeb (zkr. ESB; nástroj pro centralizovanou integraci, zajišťuje například konektivitu, transformaci datových modelů, směrování zpráv, aj. [30]). Každá služba poskytuje rozhraní (definuje poskytované funkcionality), kontrakt (definuje způsob interakce mezi službou a konzumentem) a samotnou implementaci. [29]

Architektura mikroslužeb je, stejně jako SOA, složena ze znovupoužitelných komponent, které pracují nezávisle jedna na druhé. Na rozdíl od slu-



Obrázek 2.1: Vztah SOA a architektury mikroslužeb [29]

žeb v SOA, mikroslužby nejsou určeny k znovupoužití v rámci celého podniku (prostřednictvím ESB), ale skupina vzájemně komunikujících mikroslužeb (prostřednictvím API) tvoří aplikaci poskytující nějakou byznysovou funkci. [29]

Rozdíl mezi SOA a architekturou mikroslužeb je tedy v oblasti jejich působnosti. Služby jsou používané na úrovní podniku, zatímco mikroslužby jsou používané na úrovni aplikace. Služby v rámci SOA mohou být implementovány s využitím architektury mikroslužeb. [29] Tato situace je znázorněna na obrázku 2.1.

2.1.2 Dělení na mikroslužby

Dalším rozdílem služeb SOA a mikroslužeb je jejich granularita. Mikroslužby jsou vysoce specializované. Naopak služby SOA mohou být jak úzce specializované, tak velmi komplexní. V této podkapitole se blíže zaměřím na způsob, jak dělit logiku a data mezi jednotlivé mikroslužby.

Nízká provázanost a vysoká soudržnost

Při návrhu aplikace založené na mikroslužbách je žádoucí neustále dbát na splnění nízké provázanosti a vysoké soudržnosti. Zajištění těchto vlastností je nutnou podmínkou pro splnění cílů architektury mikroslužeb a nativních cloudových aplikací.

Pokud jsou dvě služby nízce provázané, změna jedná služby by neměla vyžadovat změnu druhé. Služba by měla o službách, se kterými spolupracuje, vědět jen to nejnutnější. Při návrhu se dbá na to, aby počet využívaných funkcí jiné služby byl co nejnižší. Nízká provázanost je předpokladem pro nezávislé nasazení mikroslužby, znovupoužitelnost a rozšiřitelnost. [9]

Vysoká soudržnost znamená, že veškeré související chování je přítomno v jedné službě a všechno nesouvisející je ve službách jiných. Výhoda vysoké soudržnosti spočívá v tom, že v případě požadavku na změnu chování je nutné upravit pouze jednu službu. V praxi tedy jeden změnový požadavek řeší pouze jeden tým a vyřízení je tak rychlejší. [9]

Ohraničený kontext

Požadavek na nízkou provázanost a vysokou soudržnost implikuje velký důraz na správné rozdělení problémové domény mezi mikroslužby. Jedním z návrhových přístupů je návrh řízený doménou (*Domain-driven design, zkr. DDD*). DDD prosazuje modelování založené na realitě podniku a o problémech hovoří jako o doménách. [31]

Nezávislé problémové oblasti jsou v DDD označovány jako ohraničené kontexty (*bounded contexts*). Ohraničený kontext klade důraz na jednotnou terminologii, která je v něm používaná. Naopak jeden termín může mít v různých ohraničených kontextech různé významy. Při integraci přes hranice ohraničených kontextů může být nutný překlad. [32]

Při identifikace termínů a ohraničených kontextů je vhodné uvažovat nikoli nad daty, která jsou v rámci kontextu spravována, ale nad byznysovými schopnostmi zajišťovanými v rámci kontextu. Nejprve je tedy odpovídáno na otázku: „Co kontext dělá?“, až potom: „Jaká data k tomu potřebuje?“

Z hlediska návrhu nativní cloudové aplikace hrají ohraničené kontexty významnou roli – jsou kandidáty na mikroslužby. Velikost ohraničeného kontextu závisí na úrovni detailu, který při klasifikaci používáme. Jeden ohraničený kontext se může při větším detailu rozpadnout na více menších. Určení míry detailu, a tím i umístění hranic musí vyvážit dva protichůdné cíle:

1. vytvořit co nejmenší mikroslužby (kvůli maximalizaci efektu architektury),
2. udržet vysokou soudržnost a vyhnout se příliš vysoké „upovídanosti“ mezi službami. [9, 31]

Doporučeným postupem je začít v většími ohraničenými kontexty a postupně je rozdělovat. Při každém pokusu o rozdělení se kontroluje, zda nedošlo k příliš vysokému nárůstu komunikace přes hranice. Pokud dvě mikroslužby musí hodně spolupracovat, pravděpodobně by měly být součástí jedné mikroslužby. [31]

Datový model mikroslužeb

Monolitická aplikace má typicky jeden datový model perzistovaný v jedné databázi. V této podkapitole popíšu, zda-li je tento přístup vhodný i pro architekturu mikroslužeb, případně jak datový model upravit.

Dle Sama Newmana, autora knihy Building Microservices, je velmi častým způsob perzistence dat v architektuře mikroslužeb sdílený datový model uložený v jedné databázi. Tedy existuje jediné databázové schéma, ke kterému přistupují všechny mikroslužby. Při tomto řešení nejen že vidí všechny mikroslužby veškerá dat, ale často manipulují s daty, která logicky přísluší jiným mikroslužbám. Tomuto přístupu se říká datová integrace a je hojně používaná, jelikož se jedná o nejjednodušší řešení. [9]

Prvním záporem datové integrace je, jak už jsem zmínil, viditelnost všech dat všem. Odhalí-li mikroslužba svůj datový model, odhaluje část implementačních detailů. To je v konfliktu s požadavkem na nezávislost mikroslužeb a nízkou provázanost. Čte-li jedna služba data jiné služby, musí znát její databázové schéma. Při změně modelu mikroslužby je nutné modifikovat všechny mikroslužby, které přistupují k jejímu datovému modelu. V budoucnu může vzniknout požadavek na kompletní výměnu databáze (například dokumentové za relační). To může vést k rozsáhlým změnám napříč celým distribuovaným systémem. [9]

Druhým záporem je, že jiné služby mohou manipulovat s datovým modelem, který není v jejich logickém vlastnictví. To může způsobit, z pohledu vlastnící mikroslužby, nokonzistentní stav. Obecně manipulace je svázána s určitou byznysovou logikou. Tato logika tedy musí být součástí všech mikroslužeb, které s danými daty manipulují. Tento fakt porušuje podmínku vysoké soudržnosti. [9]

Je zřejmé, že datová integrace není ideálním přístupem. Mikroslužby by měly vlastnit a spravovat data potřebná pro svůj běh a mít k nim výhradní právo. V praxi obvykle nelze data zcela izolovat. Vezměme příklad dvou mikroslužeb v internetovém obchodě: sklad a nákupní košík. Nákupní košík pro své fungování vyžaduje znalost počtu naskladněných položek. Nyní uvedu několik možných řešení tohoto problému.

Prvním řešením je ponechat entitu ve výhradním vlastnictví jedné mikroslužby a ostatním data poskytnout prostřednictvím API. Tím je zajištěna vysoká konzistence dat, neboť existuje jediný zdroj pravdy. Může však nastat situace, kdy si dvě mikroslužby vyměňují data příliš často Potom je nutné přistoupit k jejich sloučení. Z CAP teorému vyplývá, že existuje-li jediný zdroj pravdy, konzistence dat je sice vysoká, ale znamená to také ztrátu vysoké dostupnosti systému (výpadek služby A ochromí služby, které vyžadují data patřící A).

Druhým řešením je, že daný model (nebo jeho část) může existovat ve schématech více mikroslužeb. Je zřejmé, že tento přístup přináší problémy se zajištěním konzistence, naopak zvyšuje dostupnost systému. Jelikož jsou v systému datové duplicity, je nutné zajistit synchronizaci dat. Ta může být implementována například pomocí asynchronních událostí, kdy při každé změně jsou změny propagovány všem službám, které změny nad danou entitou sledují. Dalším způsobem implementace změn ovlivňujících více mikroslužeb je návrhový vzor *Plánovač-Agent-Supervizor*. Plánovač rozděluje úlohy mezi agenty, supervizor kontroluje aktuální stav transakce. Dojde-li k chybě, supervizor vrátí systém do stavu před začátkem transakce pomocí kompenzační transakce (inverze k prováděné transakci). [33]

2.1.3 Rozhraní aplikace založené na mikroslužbách

Dosud jsem řešil pouze komunikaci backendových komponent (části webové aplikace běžící na straně serveru). V této podkapitole se budu věnovat interakci uživatele s backendovou částí nativní cloudové aplikace (pro jednoduchost dále jen aplikace).

Uživatelem aplikace může být uživatel prostřednictvím uživatelského rozhraní (frontendu) či jiná aplikace prostřednictvím poskytnutého API. Aplikace může být také integrovaná v podniku pomocí ESB. Ve všech případech uživatel/konzument obvykle během jednoho sezení interaguje s mnoha mikroslužbami, které běží v různých kontejnerech na různých adresách. Uživatel s určitým požadavkem musí vědět, jakou službu volat a na jaké je adrese. Nutné je vyřešit situace, kdy je služba nasazena na jiné adrese než doposud nebo že byla změněna architektura aplikace (přidány nové služby, funkce přesunuty, apod.).

Brána

Nativní cloudové aplikace jsou obvykle nasazovány společně s API bránou (*API gateway*), která řeší výše zmíněné problémy. API brána je mikroslužba, která zprostředkovává veškerou komunikaci s klienty. Obvykle je API brána jediná mikroslužba, jejíž rozhraní je přístupné „zvenku“ (z internetu, firemní sítě, atd.). Brána může mít několik funkcí. Základní z nich je směrování požadavků. Klient odesílá veškeré požadavky bráně a ta, na základě vnitřní konfigurace rozhodne, kterou službu a její metodu zavolat. Následně brána volání realizuje a vrátí výsledek klientovi. [5]

Další možnou funkcí brány je agregace. Některé byznysové operace můžou vyžadovat provedení akce s více mikroslužbami. Klient volá bránu s takovým požadavkem, ta provede několik akcí s jednou či více mikroslužbami a klientovi vrátí jedinou odpověď sestavenou z dílčích výsledků. Rizikem agregace je, že brána přejímá část byznysové logiky. To je v rozporu s jednoúčelovostí mikroslužeb a brána se tím může stát monolitem. [5] Příkladem agregace může být odeslání objednávky v internetovém obchodě. Brána přijme požadavek a provede vytvoření objednávky v mikroslužbě A, vystaví fakturu v mikroslužbě B a vyprázdní košík v mikroslužbě C. Klientovi vrátí potvrzení objednávky s informacemi k platbě.

Brána může také převzít funkcionality, které jsou standardně součástí většiny služeb, jako autentizace, autorizace, kešování, limitace počtu požadavků, apod. [5]

Je nutné mít na paměti, že přes bránu jde veškerý provoz mezi klientem a distribuovaným systémem. Obzvláště pokud brána poskytuje některou z výše uvedených funkcionalit, může se snadno stát úzkým hrdlem aplikace. U brány je tedy mimořádně důležité, aby ji bylo možné horizontálně škálovat. [5]

V praxi může běžně může nastat situace, kdy s backendem komunikuje několik typů klientů, například mobilní aplikace, webový prohlížeč, aplikace pro počítače, přičemž každý typ klienta může mít jiné požadavky na poskytované API (zejména se mohou lišit agregované metody). V takovém případě může projekt aplikovat techniku známou pod termínem *backends for frontends*, kdy každý typ klienta má unikátní bránu, která poskytuje API přesně na míru danému klientovi. Zpravidla to ale znamená přidání byznys logiky do brány, což je, jak jsem uvedl, problematické. [9]

2.1.4 Zabezpečení

Na závěr teoretické části práce se budu věnovat jednomu z nejdůležitějších aspektů každé aplikace, a tím je její zabezpečení. Bezpečnost nativních cloudových aplikací je velmi komplexní téma, které je nutné řešit na mnoha úrovních od architektury, přes implementaci až po infrastrukturu. V této části se budu věnovat zejména architektuře ověřování uživatelů v distribuovaném systému. Bezpečností samotné implementace a provozu se budu zabývat v praktické části práce, protože je úzce svázána s použitými technologiemi.

Proces ověřování uživatelů, kteří interagují s aplikací, se skládá ze dvou činností: autentizace a autorizace. Autentizace je činnost, při které je zjištěna identita uživatele. Při autentizaci je vždy posouzena nějaká skutečnost, která je unikátní pro daného uživatele. Tyto skutečnosti mohou být ze třech kategorií:

1. něco, co víme (informace známá pouze uživateli, například heslo),
2. něco co máme (potvrzení identity předmětem ve výhradním vlastnictví uživatele, například telefon, přístupová karta),
3. něco, co jsme (ověření něčím, co je částí uživatele, například otisk prstu, rozpoznání tváře). [34]

Autentizace je nejčastěji prováděna na základě hesla. [34]

Autorizace je činnost, při které je ověřeno, že je uživatel oprávněný přistoupit k danému zdroji nebo provést danou operaci. Termín *oprávnění* označuje právo provést danou operaci. Je-li oprávnění přiděleno uživateli, jedná se o *privilegium*. [35]

Jednotné přihlášení

V monolitických aplikacích je autentizace a autorizace přímočará, neboť všechny zdroje jsou součástí jedné aplikace a uživateli stačí jedno přihlašovací jméno a heslo. V aplikaci postavené na mikroslužbách je situace složitější, protože každá mikroslužba je, dá se říci, samostatnou aplikací. V praxi je neproveditelné, aby se uživatel registroval do každé mikroslužby zvlášť. Řešením tohoto problému je jednotné přihlášení (zkr. SSO).

Jednotné přihlášení znamená, že se uživatel přihlásí do jedné aplikace a následně je automaticky přihlášen do dalších aplikací bez ohledu na platformu, technologii nebo doménu, kterou používá. [36] Z pohledu uživatele přináší SSO zjednodušení v množství přihlašovacích údajů, které si musí pamatovat. Tím se i zvyšuje pravděpodobnost, že uživatel zvolí unikátní a bezpečné heslo. Z pohledu administrátora systému jednotné přihlášení znamená úsporu času, neboť správa rolí a přístupů je řízena centrálně, nikoli pro každou službu zvlášť. [37] Zřejmou nevýhodou je, že dostupnost poskytovatele identity je nezbytná pro požití celé aplikace – selže-li poskytovatel identity, nebude možné vystavit nové tokeny, a tedy i používat jakékoli funkce aplikace chráněné ověřením uživatele. [38]

SSO využívá pro autentizaci službu zvanou poskytovatel identity (*identity provider*). Poskytovatel identity je centrální úložiště identifikačních údajů, které na žádost provede ověření uživatele. Implementací SSO existuje více, například SAML (*Security Assertion Markup Language)* a OpenID Connect. Obecně je ale proces velmi podobný. Uživatel v prohlížeči přejde do požadované aplikace a jelikož není přihlášený, prohlížeč ho přesměruje na přihlašovací stránku SSO, kde vyplní své přihlašovací údaje. Ty jsou následně ověřeny prostřednictvím poskytovatele identity. Proběhne-li ověření úspěšně, prohlížeč obdrží token (obvykle zakódované údaje o identitě) a uživatel je přesměrován do původně navštívené aplikace. Aplikace může token ověřit buď veřejným klíčem poskytovatele identity nebo prostřednictvím jeho API. [39]

Ověření tokenu a autorizace

Aplikace obdrží token s údaji o identitě uživatele od klienta. Zbývá vyřešit otázku, kdo je v distribuované aplikaci zodpovědný za ověření tokenu a autorizaci požadavku. Existují dva základní přístupy:

1. autorizace v rámci API brány,
2. autorizace v režii jednotlivých mikroslužeb. [9]

Jsou-li všechny požadavky směrovány prostřednictvím API brány, vybízí se provést ověření tokenu a autorizaci požadavku během zpracování v API bráně. Toto řešení je také vhodné, pokud brána provádí agregace (při autorizaci v mikroslužbách a provádění agregovaného požadavku by byly autorizovány jen jeho dílčí části). Podobně jako agregace může autorizace v rámci API brány představovat porušení zásady vysoké soudržnosti i nízké provázanosti (brána se může stát významným vazebným prvkem). [9, 38]

Druhým přístupem je ověření tokenu a autorizace jakou součást jednotlivých mikroslužeb. API brána, existuje-li, provádí jen směrování bez ověření. Použití tohoto přístupu implikuje, že každá mikroslužba musí obsahovat logiku pro ověření tokenu a autorizaci požadavku. Změní-li se například struktura tokenu, je nutné upravit všechny mikroslužby. Tento přístup tedy také

porušuje zásadu vysoké soudržnosti. Dalším negativním aspektem je, že volání agregačních funkcí brány budou ověřeny v každé mikroslužbě zvlášť – je-li token ověřován pomocí poskytovatele identity, znamená to několik volání navíc, oproti autorizaci v rámci API brány.[9]

Názory odborníků se na tento problém zásadně liší. Oba zmíněné přístupy mají své klady i zápory a volba se odvíjí od požadavků a architektury každé aplikace individuálně. [38]

Zabezpečení vzájemné komunikace mikroslužeb

Dosud jsem se věnoval ověření identity a oprávnění uživatele. V architektuře mikroslužeb ale existuje kromě komunikace klient-mikroslužba i komunikace mikroslužba-mikroslužba. Přístupům k zabezpečení tohoto typu komunikace se budu věnovat v této podkapitole.

Vezmeme-li distribuovaný systém, kde se o autorizaci starají samotné mikroslužby, musí se provést autentizace a autorizace i požadavků inicializovaných jinými mikroslužbami. Tokeny s identifikačními údaji jsou tedy vystavovány i mikroslužbám samotným. Při obdržení požadavku od jiné mikroslužby je provedeno ověření, jako by šlo o lidského uživatele.

Vezměme nyní případ, kdy autorizaci obstarává brána a mikroslužby zpracovávají bez ověření všechny požadavky, které obdrží. Jak v tomto případě zabezpečit komunikaci typu mikroslužba-mikroslužba? Kvůli své jednoduchosti je velmi častým, ale ne ideálním, přístupem je povolení veškeré komunikace přicházející z určité oblasti (typicky lokální sítě, ve které mikroslužby komunikují). Celé zabezpečení tedy spočívá v tom, že ve stejné síti s mikroslužbami nesmí být žádný neoprávněný subjekt. Spoléhat se na tento předpoklad je značně riskantní, protože došlo-li by k infiltraci útočníka do sítě, měl by prakticky neomezenou moc nad distribuovaným systémem. [9]

Logickým krokem ke zvýšení bezpečnosti je vzájemné ověření identity služeb před zahájením komunikace a šifrování přenášených dat. Tyto dva problémy řeší protokol mTLS (*Mutual Transport Layer Security*). mTLS je protokol pro zřízení šifrovacího klíče a šifrování přenosu pomocí něj, při kterém si strany vzájemně poskytnou certifikáty potvrzující jejich autentičnost podepsané certifikační autoritou. Po provedení mTLS inicializace komunikace ví obě strany, že druhá strana je opravdu ta, za kterou se vydává. Nehrozí tak, že by mikroslužby komunikovaly s mikroslužbami podvrhnutými útočníkem.[40]

Kapitola 3

Analýza

Předchozí dvě kapitoly byly teoretickým úvodem do problému vývoje nativních cloudových aplikací a architektury mikroslužeb. V praktické částí práce se budu věnovat aplikování teoretických poznatků na projektu License manager, který byl předmětem mé bakalářské práce.

Cílem praktické částí je navrhnout a implementovat zcela nové řešení webové aplikace License manager tak, aby splňovala kritéria nativní cloudové aplikace. V následujících podkapitolách představím současnou verzi aplikace a definuji požadavky na novou.

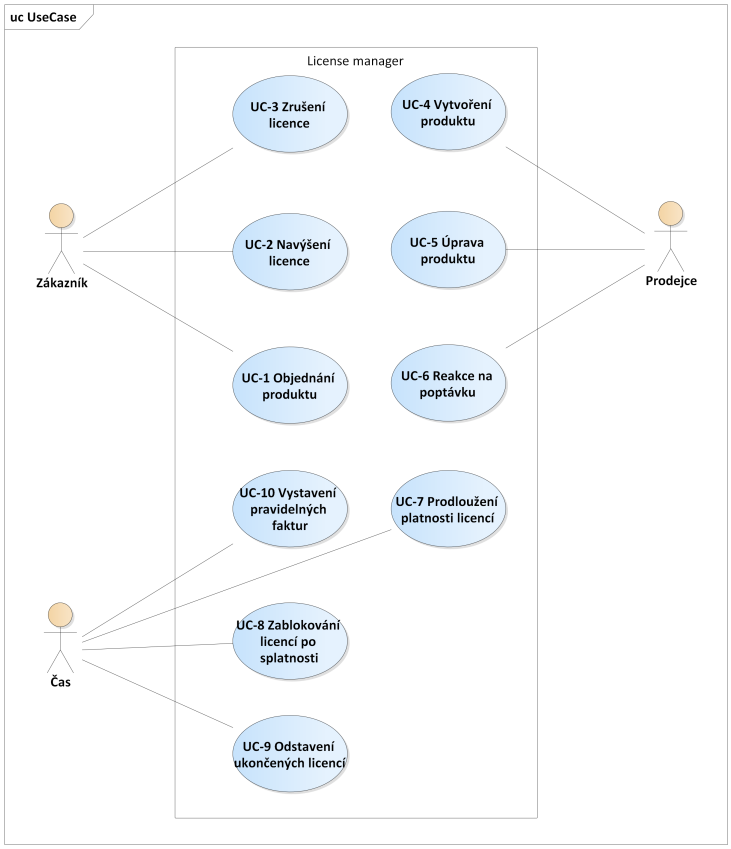
3.1 Představení současné verze aplikace

Jak jsem již zmínil, současnou verzi aplikace License manager jsem vytvořil v rámci své bakalářské práce, License manager – webová aplikace, v letech 2020-2021.

License manager (dále jen zkráceně LM) je webová aplikace vytvořená pro společnost Jagu, s.r.o., která se, mimo jiné, zabývá vývojem aplikací pro sklady či optiky. LM je určený pro prodej tohoto konfigurovatelného softwaru formou *software jako služba* (zkr. SaaS). Tedy formou prodeje softwaru na bázi předplatného, kdy aplikaci provozuje na svých serverech prodávající. [41] Konfigurovatelnost v případě aplikací prodávaných přes LM spočívá v předání určitých informací během nákupu a jejich využití při sestavení dodávané aplikace. Například při objednávce informačního systému pro optiky OptiLynx zákazník zadává údaje o společnosti, provozující síť optik a jednotlivých pobočkách. Při prvním spuštění aplikace ji má již plně nastavenou a připravenou k okamžitému provozu.

3.1.1 Funkce aplikace

V této podkapitole stručně popíšu nejdůležitější funkce současné verze aplikace License manager. Popis je členěn podle oblastí, které License manager



Obrázek 3.1: Případy užití současné verze aplikace License manager [42]

zajišťuje. Pro úplnost je na obrázku 3.1 formální diagram případů užití současné verze.

Správa nabídky

Aplikaci LM používají dva typy uživatelů: zákazníci a administrátoři. Administrátoři pracují v zákazníkům nepřístupné části aplikace. Ta umožňuje mimo jiné správu nabídky a zásahy do objednávek či licencí.

Hlavním případem užitím administrátorské sekce je ale definování nabízených produktů. Produkty jsou typicky dostupné v několika cenových plánech, které závisí na určité proměnné (například počet poboček, počet uživatelů, atd.). Proměnné a jejich hodnoty jsou taktéž nastavovány z administrátorského rozhraní.

Dále je možné specifikovat formulář pro konfiguraci produktu zákazníkem během objednávky. Konfigurační data mohou mít strukturu libovolného JSONu (datová struktura složená z dvojic klíč-hodnota, kdy hodnoty mohou být skaláry, objekty nebo pole [43]), v administraci lze tedy definovat formu-

lář, který umožňuje takto strukturovaná data získat. Pro formulář lze rovněž definovat závislost počtu odpovědí zákazníka určitého datového typu na výše zmíněné proměnné (například při konfiguraci systému pro optiky a jeho plánu „2 pobočky“ lze přidat maximálně dvakrát formulářovou položku typu „pobočka“).

Produkty je možné vytvářet, zveřejňovat (zpřístupnit ke koupi), skrývat (stáhnout z dostupné nabídky) a aktualizovat. Při aktualizaci produktu se vytvoří jeho kopie, aby nedošlo k ovlivnění probíhajících objednávek.

Objednávky

Zákazníci mohou v systému realizovat objednávky požadovaných produktů. Objednávka obsahuje vždy jen jednu položku, kterou je vybraný cenový plán požadovaného produktu. Během objednávky vyplní zákazník konfigurační formulář a zadá fakturační údaje. Platba je prováděna převodem na účet na základě zaslané faktury.

Nevybere-li si uživatel z dostupných plánů, může definovat vlastní plán nastavením hodnot proměnných, na kterých cenové plány závisí. Uživatelské plány jsou dostupné pouze u produktů, kde je tato možnost explicitně povolena. Prodejní oddělení po obdržení poptávky vytvoří cenovou nabídku, případně poptávku odmítne. Po přijetí cenové nabídky zákazníkem je vytvořena objednávka.

Licence

Výsledkem úspěšně dokončené a zaplacené objednávky je vystavení licence a vyžádání nasazení produktu pomocí aplikace Builder. Z podstaty SaaS jsou licence na dobu určitou. Platnost licence je prodloužena vždy po zaplacení předplatného. Nedojde-li k zaplacení faktury, licence není prodloužena a prodejní oddělení je e-mailem upozorněno na nezaplacené předplatné. Po individuálním posouzení případu může prodejní oddělení v administraci LM zákazníkovu instanci aplikace zablokovat (znemožnění přístupu) nebo zrušit (smazání instance včetně zákaznických dat). Zablokovaná licence je po obdržení platby automaticky prodloužena.

Zákazník může z existující licence přejít na vyšší cenový plán. V takovém případě je vytvořena nová objednávka, ve které zákazník doplní dodatečné konfigurační údaje pro vyšší cenový plán. Po zaplacení objednávky navýšení je vystavena nová licence a původní zrušena. Přejít lze pouze na vyšší cenový plán, nikoli na nižší.

Fakturace

Pro fakturaci se používá externí webová aplikace Fakturoid se kterou LM komunikuje pomocí REST (*Representational State Transfer*) API. Během objednávky je ve Fakturoidu vytvořen kontakt na základě poskytnutých faktu-

račních údajů. Tomuto kontaktu jsou potom přiřazovány veškeré faktury. LM provádí každý den kontrolu platnosti licencí a první den po konci zaplaceného období vystaví novou fakturu. Zákazník má poté 15 dní na zaplacení (doba splatnosti lze konfigurovat ve Fakturoidu). Faktury jsou uloženy kromě Fakturoidu také v databázi LM.

LM v pravidelných intervalech načítá stavy faktur (například „zaplaceno“, „po splatnosti“) z Fakturoidu a aktualizuje stav ve své databázi. Při změně stavu faktury provede jednu z těchto akcí:

1. vystaví licenci, pokud je faktura zaplacená a licence ještě neexistuje (po první platbě),
2. prodlouží platnost existující licence, pokud je faktura zaplacená,
3. označí licenci k blokaci a upozorní oddělení prodeje, pokud je faktura po splatnosti.

3.1.2 Technické řešení

Předchozí podkapitola byla změřena na funkce současné verze License manageru. Nyní popíšu její technické řešení – použité technologie, architekturu či integraci s dalšími systémy.

Sada technologií

Současná aplikace LM je napsaná skriptovacím jazykem PHP. Tento jazyk jsem zvolil, protože jsem s ním měl v době psaní bakalářské práce nejvíce zkušeností, a protože je ve velké míře používán na komerčních aplikacích zadavatele, firmy Jagu, s.r.o.

PHP (rekurzivní zkratka pro *PHP: Hypertext Preprocessor* je obecně zaměřený skriptovací jazyk s využitím zejména pro vývoj webových aplikací. PHP skripty mohou být vloženy do HTML. Spuštění skriptu, v případě webových aplikací, probíhá na serveru. [44]

Moderní PHP vývoj spoléhá na využití některého z frameworků (obecně podpůrná struktura, na které je dále stavěno [45]), například Laravel, Symphony nebo Nette. LM je postavený na prvním zmíněném. Laravel je framework, který si klade za cíl zjednodušit a zrychlit vývoj webových aplikací řešení typických problémů každé moderní webové aplikace jako je připojení k databázi, autentizace, autorizace a další. Mimo funkce frameworku samotného je k dispozici celá řada oficiálních i neoficiálních knihoven například pro podporu platebních bran, integraci se sociálními sítěmi či vyhledávání v obsahu. [46]

Architektura

Architektura současné verze LM je silně ovlivněna použitým frameworkem, který doporučuje (až předepisuje) členění kódu a tím i architekturu. LM využívá architekturu *Model-View-Controller* (zkr. MVC), která rozděluje aplikaci do třech komponent. První komponentou je *Model*, který obstarává obchodní logiku, druhou *Controller*, který přijímá požadavky uživatele, manipuluje s modelem a předává data třetí komponentě – *View*, které řeší zobrazení dat uživateli.

Přestože je kód členěn do třech komponent, prezentační vrstva i veškerá logika jsou součástí jediné aplikace, jedná se tedy o typickou monolitickou aplikaci (přístup vývoje softwaru, kdy výsledná aplikace tvoří jednotný, samostatný celek, nezávislý na ostatních aplikacích [47]).

Perzistence dat

Data jsou uložena v relační databázi. Aktuálně je využíván databázový systém PostgreSQL, ale díky přístupu k datům výhradně skrze knihovnu pro objektově relační mapování (objektově relační mapování, zkr. ORM je technika překladu dat mezi objektovým a relačním paradigmatem [48]) Eloquent, která je součástí frameworku Laravel, je možné využít také MySQL, SQLite nebo Microsoft SQL Server. [49]

Eloquent je založený na ORM typu *Active record*. Třídy objektového modelu jsou mapovaný 1:1 na tabulky databáze. Hlavní vlastností *Active record* je, že třídy kromě definice atributů (které se mapují na sloupce tabulky) obsahují také metody obchodní logiky. [50]

Integrace s dalšími systémy

V příchozí podkapitole jsem zmínil, že LM využívá pro fakturaci aplikaci Fakturoid. Dalším využívaným systémem je e-mailový server, se kterým LM komunikuje protokolem SMTP (*Simple Mail Transfer Protocol*), který je technickým standardem pro přenos elektronické pošty po síti. SMTP slouží pouze pro výměnu zpráv mezi e-mailovými servery, neslouží pro získání zpráv klientem ze serveru. [51]

Pro praktické nasazení LM je nutná aplikace s pracovním názvem *Builder*, která zajišťuje samotné sestavení zakoupených produktů dle konfiguračních údajů a nasazení na server. Po dokončení mé bakalářské práce na aplikaci Builder pracovali studenti Fakulty informačních technologií ČVUT v Praze v rámci předmětů Softwarový týmový projekt 1 a 2 (v letním semestru 2021 a zimním semestru 2022). Na Builderu dále pracuje Alena Ježková ve své bakalářské práci.

V mém návrhu komunikoval LM s Builderem prostřednictvím REST API a operacemi Builderu bylo sestavení, blokace licence, reaktivace licence a zrušení licence. Adam Staš ale ve své bakalářské práci pracuje na třetím mo-

dulu, který doplní LM a Builder, tzv. Deployment manager. Ten bude mít za úkol evidovat nasazené verze zákaznických aplikací a například provádět aktualizace. Dle zadání jeho práce bude Deployment manager figurovat jako prostředník mezi LM a Builderem.

Asynchronní operace

Některé operace prováděné LM jsou časově náročné a v případě jejich synchronního zpracování v rámci vyřízení požadavku by vedly ke zhoršení responzivity aplikace. Tento problém jsem vyřešil použitím front úloh, jež poskytuje samotný framework Laravel. Příkladem těchto úloh je odesílání e-mailů, zpracování faktur, řízení procesu sestavení apod. Ve chvíli, kdy je nutné provést některou z těchto úloh, je vytvořen záznam do fronty úloh obsahující identifikaci metody, která úlohu provede, a parametry pro spuštění.

Laravel podporuje několik úložišť pro frontu. Základním a aktuálně používaným je vložení záznamů do databáze (jedná se vzhledem k nízkému očekávanému zatížení systému o plně dostačující metodu). V případě potřeby je možné ukládání úloh do databáze nahradit samostatnými a vysoce výkonnými aplikacemi pro správu front jako je Amazon SQS nebo Redis. [52]

Úlohy z fronty odebírá a zpracovává samostatný proces, který je plně nezávislý na chodu aplikace samotné. Jeho činnost tedy neovlivňuje výkonnost aplikace.

3.2 Požadavky na nativní cloudovou verzi aplikace License Manager

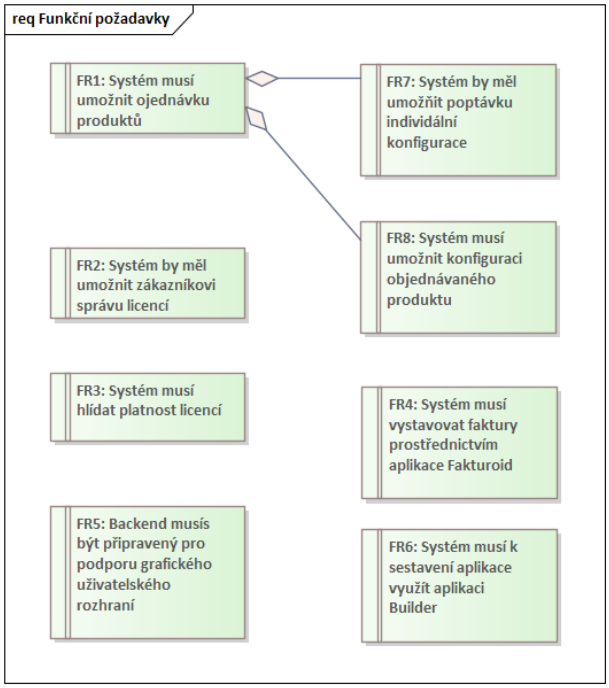
V předchozí podkapitole jsem provedl revizi výstupu mé bakalářské práce a představil její důležité aspekty. Kromě uvedení čtenáře do problematiky aplikace License manager slouží tento popis jako cenný podklad pro definování požadavků na novou verzi aplikace a návrh její architektury.

Jak jsem již uvedl, cílem praktické části diplomové práce je na základě zkušeností s vývojem současné verze aplikace navrhnout a implementovat backendovou část aplikace License manager, aby odpovídala požadavkům a vlastnostem nativní cloudové aplikace. Od nové aplikace se tedy očekává zcela nová architektura. Pro implementaci je možné využití odlišné sady technologií.

Na samém počátku práce jsem se setkal s vedoucím práce, Ing. Oldřichem Malcem a Ing. Jiřím Hunkou, oběma z firmy Jagu, s.r.o., pro kterou je aplikace určena. Účelem této schůzky bylo definovat požadavky na novou verzi.

3.2.1 Funkční požadavky

Zadavatel práce nespecifikoval žádné dodatečné funkční požadavky na aplikaci. Jinými slovy účelem práce není rozšířit funkčnost aplikace, ale navrhnout a implementovat aplikaci s odlišnou architekturou. Vzhledem k výrazně



Obrázek 3.2: Diagram funkčních požadavků

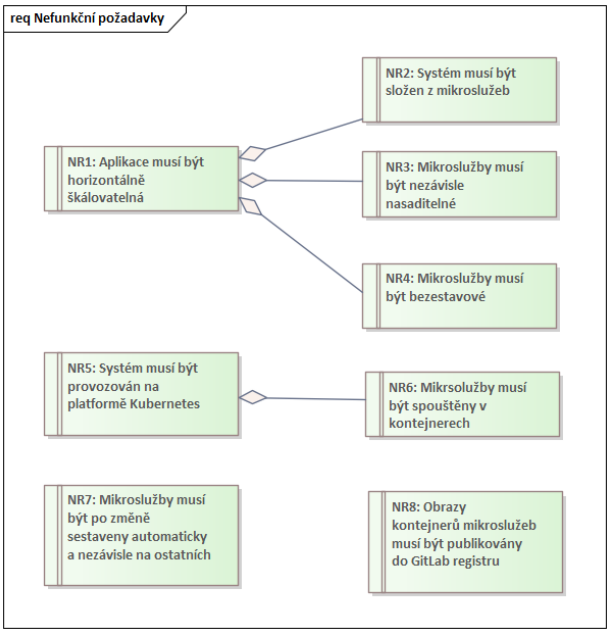
vyšší očekávané časové náročnosti realizace nativní cloudové aplikace založené na mikroslužbách, v porovnání s monolitickou architekturou, je v zadání specifikováno, že výsledkem práce má být *Minimum Viable Product* (minimální životaschopný produkt). Tedy je připuštěna možnost realizace pouze podmnožiny funkcionalit současné verze, ale s ohledem na její praktickou použitelnost.

Vhledem k tomuto faktu přejímá nová verze funkční požadavky téměř beze změny. Upravený diagram funkčních požadavků v modelovacím jazyce UML (Unified Modeling Language) je na obrázku 3.2. Změny spočívají ve funkčním požadavku FR5 (požadavek na implementaci grafického rozhraní je změněn na přípravu rozhraní pro grafické uživatelské rozhraní) a FR1 je dospecifikován dílčími funkčními požadavky FR7 (podpora individuálních konfigurací) a FR8 (konfigurace produktu během objednávky).

3.2.2 Nefunkční požadavky

Co se však od původní verze liší jsou nefunkční požadavky. Funkční požadavky definují *co* má aplikace dělat, zatímco nefunkční specifikují, *jak* má být cílů dosaženo a jaké má mít aplikace provozní vlastnosti [53]. Nefunkční požadavky se vztahují například k zabezpečení aplikace, kompatibilitě, výkonu, spolehlivosti, udržovatelnosti, škálovatelnosti či použitelnosti.

Pro novou verzi aplikace jsem na základě teoretické části a konzultací



Obrázek 3.3: Diagram nefunkčních požadavků

s vedoucím práce definoval osm nefunkčních požadavků, jejich diagram je na obrázku 3.3.

Prvním nefunkčním požadavkem (NR1) je horizontální škálovatelnost aplikace, tedy jeden z hlavních cílů vývoje nativních cloudových aplikací. Aby tohoto požadavku mohlo být dosaženo, definoval jsem dílčí požadavky NR2, NR3 a NR4, které říkají, že aplikace musí být členěna na nezávislé a bezestavové mikroslužby.

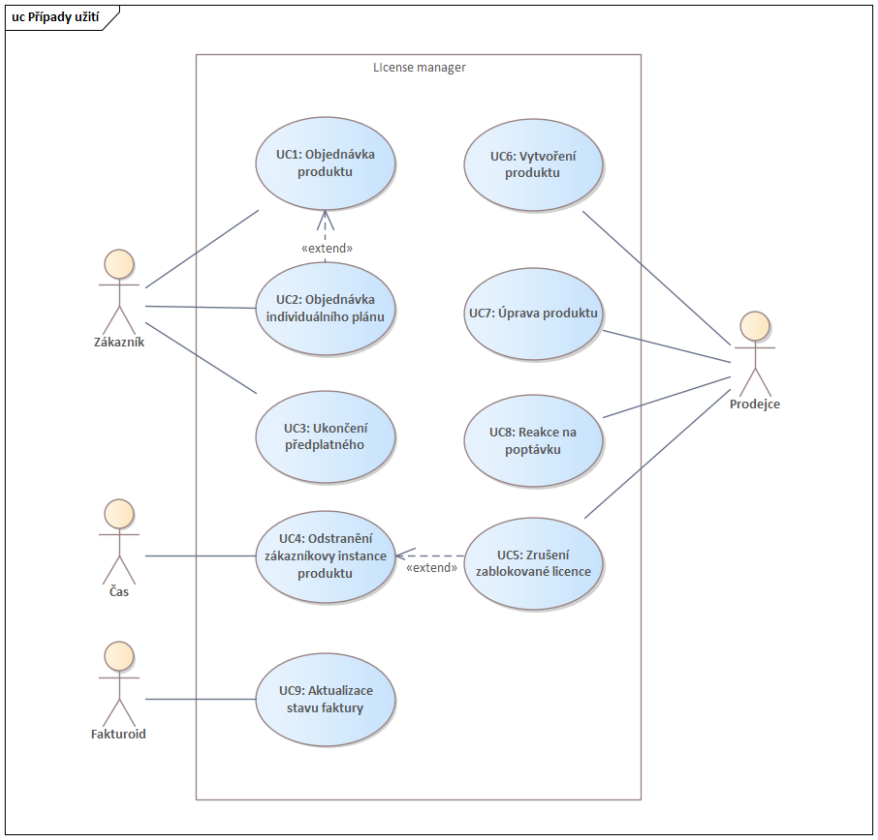
V kapitole 1.4 jsem uvedl dva základní způsoby provozu aplikací na cloudových platformách – kontejnery a funkce. Funkcím a přístupu serverless se v této práci nevěnuji z důvodu silné vazby na konkrétního poskytovatele cloudové platformy a tím porušení požadavku na přenositelnost nativních cloudových aplikací. Požadavek NR6 proto určuje, že mikroslužby budou provozovány v kontejnerech. Konečně NR5 požaduje, aby kontejnery byly orchestrovány nástrojem Kubernetes, který jsem již krátce představil v podkapitole 1.6.4.

Poslední dva nefunkční požadavky souvisí s distribucí mikroslužeb. NR7 určuje, aby mikroslužby byly po změně automaticky sestavovány, a to nezávisle na ostatních mikroslužbách. NR8 definuje, že výsledné obrazy kontejnerů mikroslužeb budou uchovávány v registru GitLab spravovaném zadavatelem.

3.2.3 Případy užití

Případy užití taktéž vychází z analýzy provedené pro původní verzi aplikace. Odebrány byly případy užití aktivované časovou událostí: vystavení pravidelných faktur, a prodloužení platnosti licencí. Nahrazeny byly novým případem užití aktéra Fakturoid: aktualizace stavu faktury. V současné verzi vystavování pravidelných faktur a prodlužování licencí bylo prováděno pomocí nástroje CRON (utilita operačního sytému, která v pravidelných intervalech spouští určitý příkaz [54]). Na základě revize API Fakturoidu jsem se rozhodl využít jeho podpory pro webhooky (webhook je funkce zpětného volání po síti [55]). V případě aktualizace stavů faktur se tedy nebude LM pravidelně dotazovat Fakturoidu na aktuální stav, ale Fakturoid v případě změny stavu faktury zavolá předem danou metodu LM, která je součástí jeho veřejného API. Na základě změny stavu faktury jsou pak provedeny úkony jako prodloužení licence, nebo její blokace.

Pro již zmíněnou časovou náročnost implementace aplikace využívající architekturu mikroslužeb jsem v průběhu analýzy a návrhu odstranil z případů užití navýšení licence. Její doplnění tedy bude předmětem případného dalšího rozvoje aplikace. Upravený diagram případů užití pro novou verzi aplikace je na obrázku 3.4. Součástí zadávací dokumentace a dokumentace architektury systému vytvořené v softwaru Enterprise Architect 15 (dostupné na přiloženém médiu) jsou formální a strukturované scénáře jednotlivých případů užití.



Obrázek 3.4: Případy užití nové verze aplikace License manager

Kapitola 4

Návrh nativně cloudové aplikace

Dalším krokem v softwarovém cyklu je návrh. Při realizaci nové verze aplikace jsem prováděl dva typy návrhů. První před počátkem implementace, který měl za cíl navrhnout distribuovaný systém jako celek, ale nezacházel do implementačních detailů. A druhý, který byl prováděn pravidelně v rámci agilních iterací při postupné implementaci funkcí mikroslužeb. V této kapitole se budu věnovat prvnímu typu návrhu architektury distribuované verze aplikace License manager.

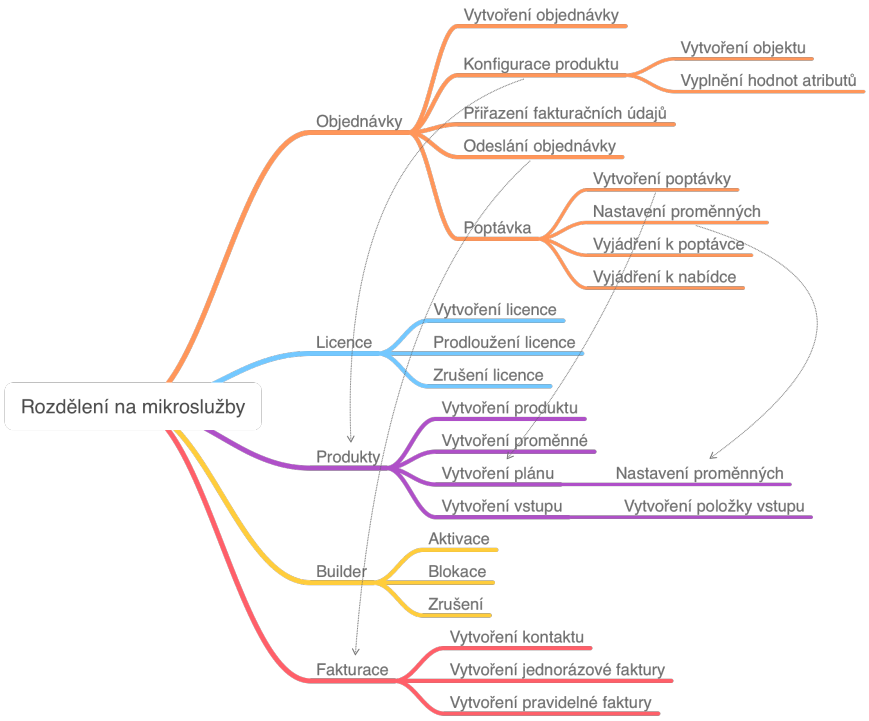
4.1 Rozdělení monolitu na mikroslužby

Jednou z největších výzev při přechodu z monolitické architektury na architekturu mikroslužeb je nalezení vhodných hranic tak, aby vzniklé mikroslužby byly vysoce soudržné a nebyly úzce provázané. Teorii dělení na mikroslužby jsem se věnoval v kapitole 2.1.2. Nyní tyto teoretické poznatky budu aplikovat při návrhu nové verze aplikace License manager.

V kapitole 2.1.2 jsem popsal hledání hranic pomocí ohraničených kontextů. Cílem je nalézt oblasti se souvisejícími byznysovými operacemi a jednotnou terminologií. Ohraničený kontext může být rozdělen na více dílčích ohraničených kontextů – doporučeným postupem je začít s většími oblastmi a pokoušet se je dále dělit.

V prvním kroku jsem identifikoval pět oblastí: objednávky, licence, produkty, fakturace a nasazení aplikací. Následně jsem s využitím myšlenkové mapy (obr. 4.1) oblastem přiřazoval operace, přičemž jsem vycházel z případů užití a jejich scénářů definovaných během analýzy. V dalším kroku jsem si do myšlenkové mapy zaznačil některé závislosti mezi operacemi (tenké šedé šipky).

Vysoká provázanost se projevila mezi objednávkami a produkty. Zejména konfigurace produktu v rámci objednávky byla problematická, protože definice konfiguračního vstupu byla součástí oblasti produktů, zatímco vyplnění konfi-



Obrázek 4.1: Myšlenková mapa prvotního rozdělení na ohraničené kontexty

gurace v oblasti objednávek. Po analýze kladů a záporů této provázanosti jsem se rozhodl objednávky zcela oddělit od nabídky (mikroslužbu Objednávky tak bude možné použít i pro zcela odlišné situace než je prodej softwaru). Uživatelskou konfiguraci produktů jsem tedy přesunul z kontextu objednávek do kontextu produktů.

Provázanost ale stále způsobovaly operace poptávky individuálních plánů. V rámci poptávky je vytvořen individuální plán (operace z kontextu produktů) a úspěšné dokončení procesu poptávky vede na vytvoření objednávky (kontext objednávek). Abych odstranil závislost objednávky ↔ produkty, vyčlenil jsem poptávky do samostatného kontextu. Vyčlenění navíc umožní nakládat s poptávkami jako volitelným modulem systému.

Identifikovanými kandidáty na mikroslužby jsou tedy: objednávky (Orders), licence (Licenses), produkty & konfigurace (Products), nasazení aplikací (Deployments), fakturace (Invoices) a poptávky (Inquiries).

Nejkomplexnější mikroslužbou v popsaném rozdělení jsou produkty a konfigurace. Již samotný název navádí k rozdělení na dvě samostatné mikroslužby

– definice produktů a uživatelské konfigurace. Dle postupu popsaném v teoretické části jsem provedl analýzu dopadů tohoto rozdělení. Definice produktů, cenových plánů a konfiguračních formulářů jsou v dané doméně silně související entity, které by bylo obtížné rozdělit (např. podoba konfiguračního formuláře se odvíjí od parametrů cenového plánu). Samostatná mikroslužba uživatelských konfigurací by obsahovala pouze vyplnění konfiguračního formuláře. Problematické je zde rozdělení definice formuláře a jeho vyplnění do dvou subjektů, protože znalost definice je nezbytná pro vyplnění formuláře a validaci výsledných dat. Rozdělení by mělo za následek, že většina operací mikroslužby uživatelských konfigurací by obsahovala minimálně jedno volání mikroslužby s definicí produktů. V teoretické části jsem uvedl, že v takovém případě není doporučeno provést rozdělení.

4.1.1 Data mikroslužeb

Dosud jsem pracoval se zobecněnými byznysovými funkcemi. Než se budu moci věnovat návrhu konkrétních rozhraní mikroslužeb, je nutné specifikovat, jaká data jsou potřeba pro vykonávání požadovaných byznysových funkcí, protože rozdělení dat mezi mikroslužby ovlivní komunikační schéma systému. Jak jsem uvedl v teoretické části, mikroslužby mají mít ve výhradním vlastnictví všechna data nutná pro svou činnost. Předávání dat mezi mikroslužbami zvyšuje provázanost a zatížení systému.

Na základě datového modelu současné verze aplikace jsem identifikoval entity, se kterými budou jednotlivé mikroslužby manipulovat, a vložil je do tabulky 4.1. V tabulce chybí mikroslužba Deployments, protože její realizace je mimo rozsah této práce (na řešení pracují ve svých bakalářských pracích Adam Staš a Alena Ježková).

Následuje popis dat jednotlivých mikroslužeb. V aktuální fázi návrhu systému jako celku pracuji stále s vysokou mírou abstrakce a navrhované modely neobsahují implementační detaily a nezávisí na způsobu perzistence dat. Konkrétní návrh a implementace datových modelů bude bude realizována až v agilních iteracích jednotlivých mikroslužeb a bude záviset na použitých technologiích.

Produkty a konfigurace

Na obrázku 4.2 je konceptuální datový model mikroslužby Produkty v syntaxi UML Entity Relationship Diagram, zkr. ERD (znázornění dat s nejvyšší mírou abstrakce, vynechávající veškeré implementační detaily a sloužící k předání informace o základní struktuře dat [56]).

Mikroslužba Produkty zajišťuje definici produktů a během objednávky sběr dat od uživatele nutných pro sestavení a nasazení zakoupené instance aplikace. Ústřední entitou mikroslužby je entita Produkt. Entita Produkt obsahuje název produktu, jeho popis a verzi. Produkt lze zakoupit pouze v některém

Mikroslužba	Manipulace s daty
Produkt	Produkt Cenový plán Definice konfiguračního objektu Definice atributu konfiguračního objektu Konfigurační proměnná Hodnota proměnné Konfigurační objekt Hodnota atributu konfiguračního objektu
Objednávky	Objednávka
Fakturace	Faktura Fakturační údaje
Licence	Licence
Poptávky	Poptávka

Tabulka 4.1: Tabulka mikroslužeb a dat, se kterými manipulují

z nabízených cenových plánů, které označuje entita Cenový plán. Úroveň cenového plánu je určena hodnotou produktové proměnné (např. počet poboček). Definice těchto proměnných přísluší produktu a všechny cenové plány musí specifikovat hodnotu všech proměnných.

Produkt také obsahuje definici konfiguračního objektu (např. adresa pobočky). Definice konfiguračního objektu obsahuje definice atributů (např. pro adresu pobočky jsou atributy ulice, město, atd.). Instance konfiguračních objektů a jejich atributů označují entity Konfigurační objekt a Hodnota atributu konfiguračního objektu, které vytváří uživatel během objednávky.

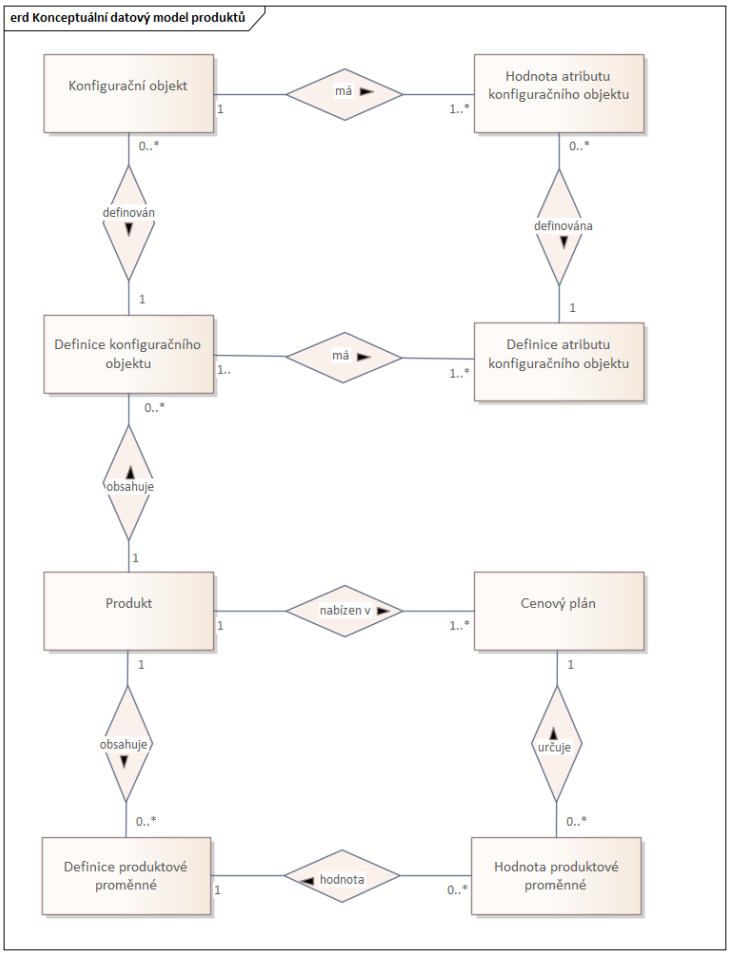
Konfigurační objekty mohou obsahovat také jiné konfigurační objekty (vazbu sama na sebe nelze v UML ERD diagramu zachytit). Stejně tak definice konfiguračního objektu obsahuje definici dceřiného konfiguračního objektu.

Na obrázku 4.3 je stavový diagram produktu. Produkt je nejprve ve stavu „nový“, ve kterém probíhá jeho konfigurace. Následně je publikován (zařazen do nabídky). Produkt může být také dočasně skryt. Produkty vyřazené z nabídky (například po duplikaci a úpravě) jsou ve stavu „ukončený“.

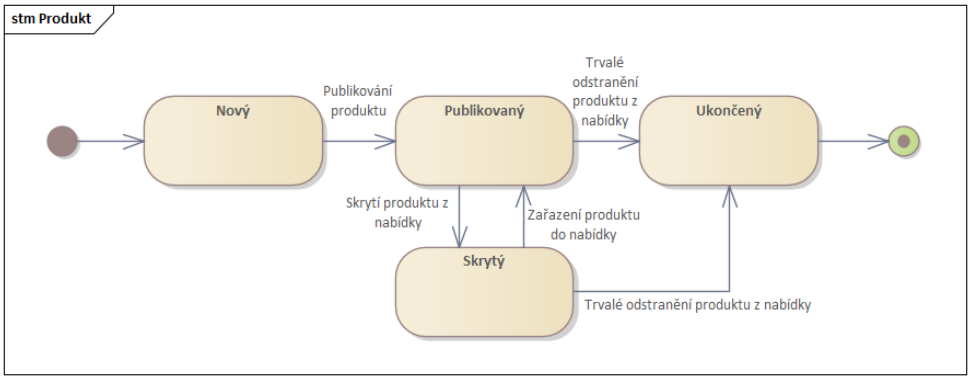
Objednávky

Mikroslužba Objednávky obsahuje jedinou entitu stejného jména. Jejím úkolem je zaznamenat identifikátor objednávaného artiklu (v případě License manageru se bude jednat o identifikátor cenového plánu), identifikátor uživatele, identifikátor fakturačních údajů poskytnutý mikroslužbou Fakturace a stav objednávky.

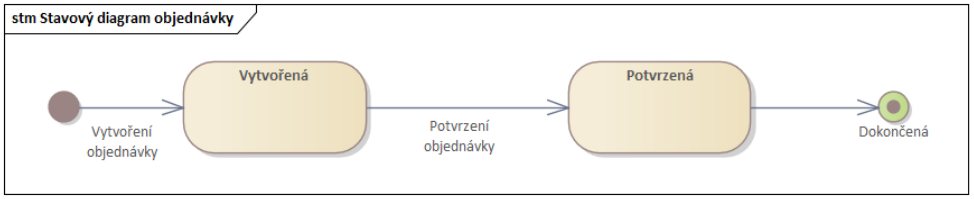
V původní verzi LM proces objednávky řídí všechny činnosti až po sestavení produktu. Díky rozdělení zodpovědnosti za proces objednávky mezi více



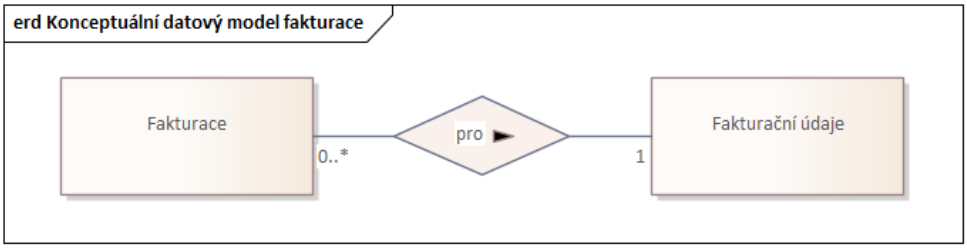
Obrázek 4.2: Konceptuální datový model mikroslužby Produkty



Obrázek 4.3: Stavový diagram produktu



Obrázek 4.4: Stavový diagram objednávky



Obrázek 4.5: Konceptuální datový model Fakturace

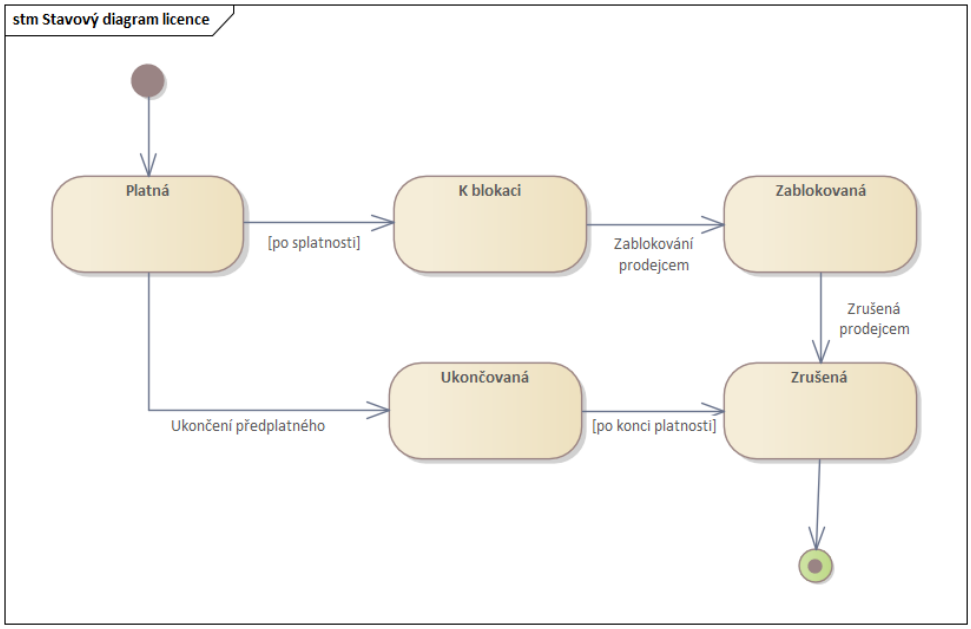
mikroslužeb končí aktivity Objednávek v nové verzi odesláním požadavku na vytvoření faktury. Stavový diagram objednávky (obrázek 4.4) je tedy oproti původní verzi výrazně zjednodušený.

Fakturace

Proces objednávky končí začátkem procesu fakturace. Mikroslužba Fakturace má roli prostředníka mezi aplikací LM a fakturačním softwarem Fakturoid. V kapitole 3.2.3 jsem popsal zjednodušený způsob fakturace a komunikace s Fakturoidem. Správa stavu faktur a jejich pravidelné vystavování je plně v režii Fakturoidu. Kromě zprostředkování komunikace s Fakturoidem zajišťuje mikroslužba Fakturace uložení fakturačních údajů uživatelů a jejich mapování na kontakty ve Fakturoidu a uložení informací o způsobu fakturace objednávek. Entita Fakturace obsahuje referenci na fakturační údaje, částku pro jednorázovou a pravidelnou platbu, délku zúčtovacího období, číslo objednávky a referenci na generátor pravidelných faktur ve Fakturoidu. Konceptuální datový model je na obrázku 4.5.

Licence

Mikroslužba Licence obsahuje jedinou entitu: Licenci. Účelem je udržovat informaci o počátku a konci platnosti licence, identifikátoru příslušné objed-



Obrázek 4.6: Stavový diagram licence

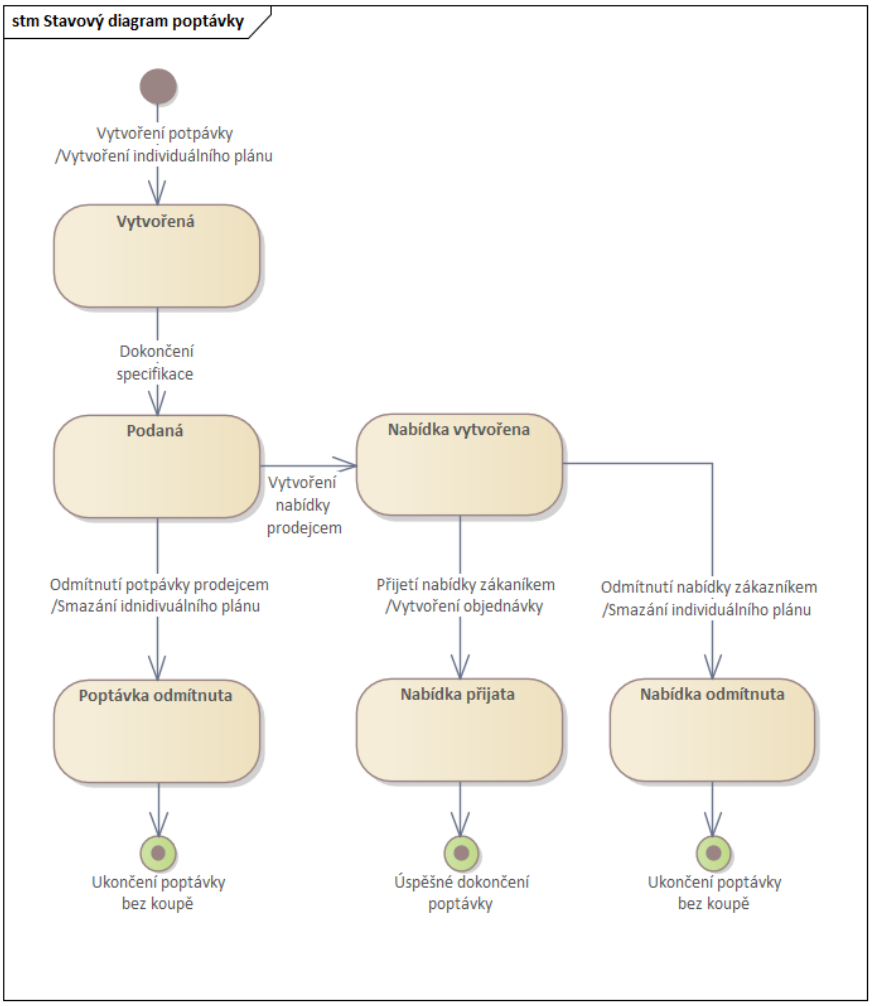
návky a stavu licence. Stavy faktury jsou znázorněny na stavovém diagramu 4.6.

Za standardních okolností je licence ve stavu „platná“. Není-li předplatné zaplaceno do určeného termínu, přejde licence do stavu „k zablokování“. Potom je na rozhodnutí prodejce, zda licenci (a příslušnou instanci aplikace) zablokuje. Dalším krokem po zablokování je stav „zrušená“, což je finální stav, ze kterého nelze licenci obnovit. Rozhodne-li se zákázník sám ukončit předplatné, přejde licence do stavu „ukončovaná“ a po konci její platnosti bude automaticky zrušena.

Poptávky

Mikroslužba má jedinou entitu: Poptávku. Při vytvoření poptávky je současně vytvořen individuální cenový plán mikroslužbou Produkty. Poptávka uchovává identifikátor tohoto plánu, identifikátor zákazníka (uživatele) a stav poptávky.

Stavový diagram poptávky je znázorněn na obrázku 4.7. Po vytvoření je poptávka ve stavu „vytvořená“. V tomto stavu zákazník specifikuje hodnoty produktových proměnných nově vytvořeného individuálního plánu dle svých potřeb. Po dokončení specifikace odešle poptávku, čímž přechází do stavu „podaná“. Prodejce může poptávku odmítnout (přechod do stavu „odmítnutá“) nebo doplnit individuální plán o nabízenou cenu (přechod do stavu „nabídka



Obrázek 4.7: Stavový diagram poptávky

vytvořena“). Zákazník může nabídku přijmout, čímž se vytvoří objednávka, nebo odmítnout.

Mikroslužba Poptávky pracuje i s entitami Cenový plán a Objednávka, které jsou ve vlastnictví mikroslužeb Produkty a Objednávky. Manipulace s těmito entitami je prováděna prostřednictvím API zmíněných mikroslužeb.

4.2 Rozhraní mikroslužeb a jejich komunikace

V předchozí podkapitole jsem rozdělil monolitickou aplikaci na mikroslužby, identifikoval jejich byznysové funkce a data potřebná pro jejich chod. V této podkapitole se zaměřím na návrh konkrétních komunikačních vzorů mezi mi-

krosložbami, jejich rozhraní a celkovou kompozici systému License manager z dílčích částí.

4.2.1 Model vzájemné komunikace

Každá mikroslužba bude nabízet sadu samostatně využitelných funkcí. Byznysovým cílem aplikace License manager jako celku je podpora procesu prodeje softwaru formou předplatného. Je zřejmé, že pro splnění tohoto cíle bude nutná spolupráce mikroslužeb.

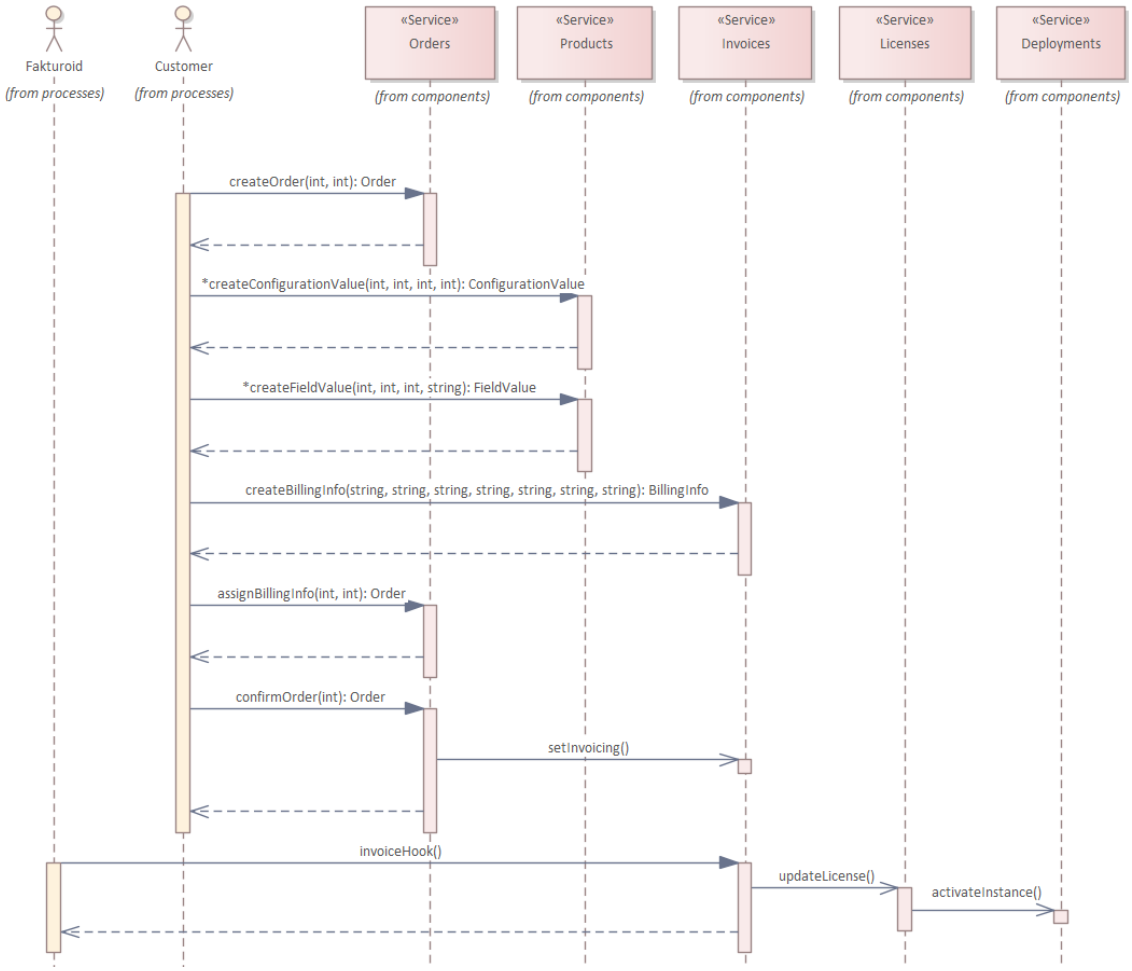
Návrh komunikace jsem prováděl v nástroji Enterprise Architect 15 pomocí sekvenčního diagramu notace UML. Sekvenční diagram podrobně popisuje provádění operací / zasílání zpráv mezi subjekty (služby či uživatelé) s důrazem na jejich pořadí. [57]

Začal jsem modelováním sekvenčního diagramu procesu prodeje (obrázek 4.8). Uživatel nejprve vytvoří objednávku, následně vyplní konfigurační formulář, zadá fakturační údaje (nebo vybere stávající), přiřadí je k objednávce a objednávku potvrdí. Všechny tyto operace jsou synchronní. Potvrzení objednávky vyvolá publikování události o dokončení objednávky, na kterou reaguje mikroslužba Invoices (Fakturace) a vystaví faktury prostřednictvím Fakturoidu. Jakmile Fakturoid registruje platbu, zavolá pomocí webhooku Invoices, která publikuje událost o platbě objednávky. Na ni reaguje mikrolusžba Licences (Licence), která vystaví licenci a publikuje event, který způsobí spuštění procesu nasazení uživatelské instance zakoupené licence. Operace realizující zpracování objednávky po jejím potvrzení jsou tedy již asynchronní.

Obecně operace, které budou inicializovány uživatelem z grafického rozhraní jsou v návrhu vždy synchronní, protože každá akce v uživatelském rozhraní by měla mít svou reakci, a ta je závislá na odpovědi backendu. Naopak u komunikace mezi mikroslužbami volím všude, kde je to možné, komunikaci asynchronní, protože snižuje provázanost systému a díky zprostředkovateli doručení zpráv zvyšuje odolnost vůči výpadkům mikroslužeb. Vlastnosti asynchronní komunikace jsem více popsal v kapitole 1.5.

Druhým procesem, který využívá více mikroslužeb, a který může předcházet procesu nákupu, je již zmíněná poptávka individuální konfigurace. Sekvenční diagram poptávky je na obrázku 4.9.

Proces poptávky jsem popsal již v předchozí podkapitole. V této části návrhu je důležitá zejména komunikace mezi mikroslužbou Inquiries (Poptávky) a mikroslužbami Products (Produkty) a Orders (Objednávky). Mikroslužba Products je volána při vytvoření poptávky, vytvoření nabídky a odmítnutí nabídky. Mikroslužba Orders je volána při přijetí nabídky. Ve všech případech se jedná o synchronní komunikaci, přestože jde o komunikaci mezi mikroslužbami. Synchronní komunikace je vyžadována, protože výsledky těchto dílčích operací ovlivňují výsledek synchronního požadavku uživatele na mikroslužbu Inquiries – před odesláním odpovědi z Inquiries je nutné vyčkat na odpověď z jiných mikroslužeb.



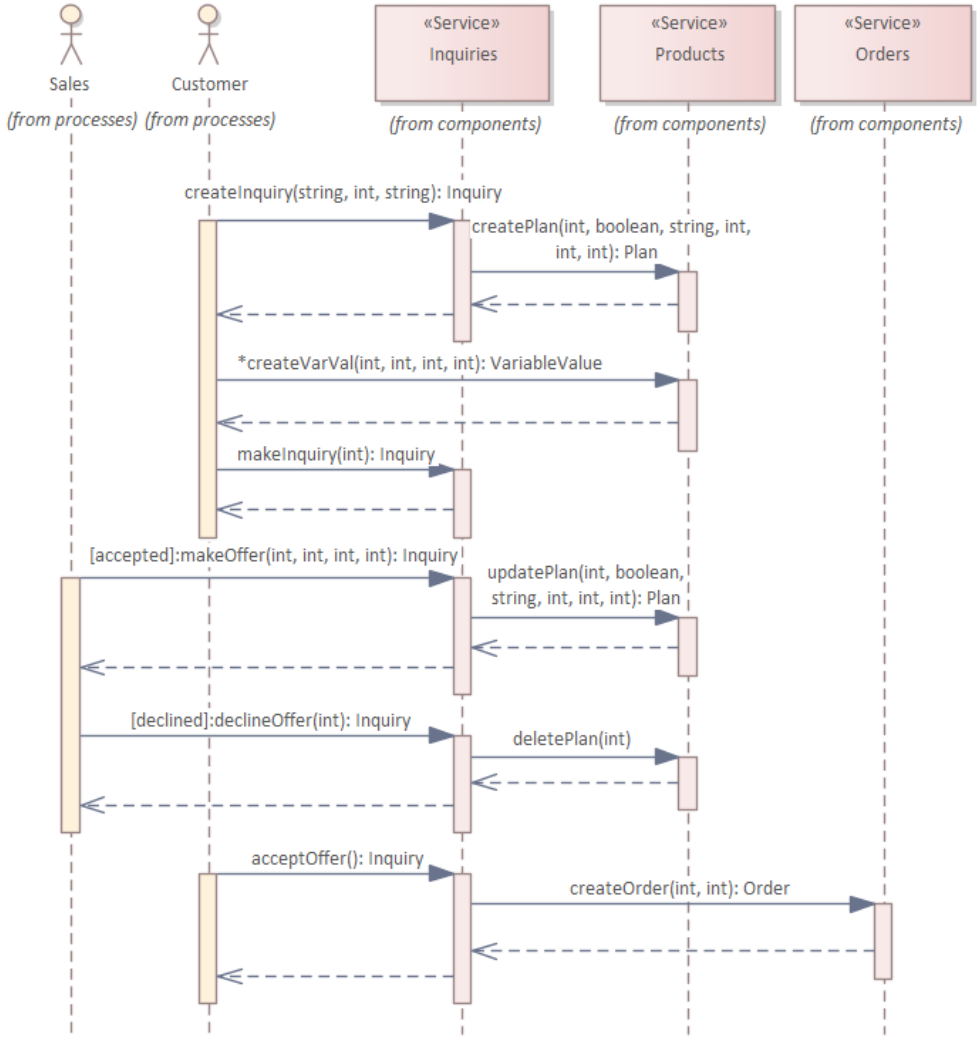
Obrázek 4.8: Sekvenční diagram procesu prodeje

Kvůli synchronní komunikaci je mikroslužba Inquiries silně závislá na dostupnosti mikroslužeb Products a Orders. V tomto konkrétním případě to nepovažuji za problém, protože závislost je jednostranná a poptávky jsou doplňkem/rozšířením uvedených mikroslužeb.

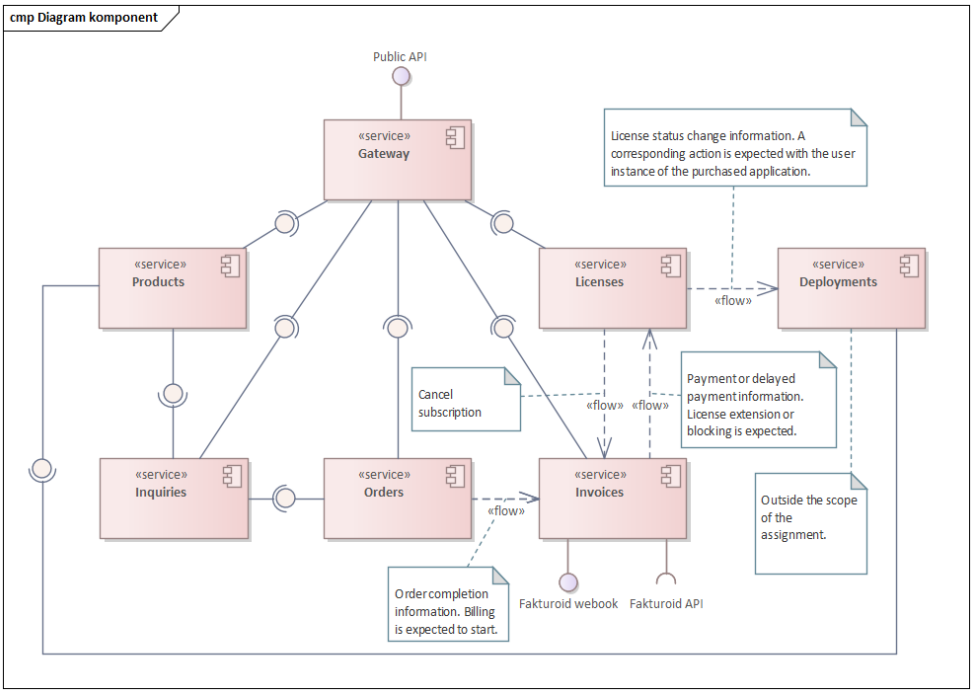
4.2.2 Kompozice systému

Na základě návrhu komunikace mezi službami jsem mohl sestavit finální návrh kompozice systému. Pro znázornění jsem využil diagram komponent notace UML, který slouží ke znázornění vztahů mezi komponentami v systému (obrázek 4.10). [58]

V diagramu přibyla jedna mikroslužba, které jsem se prozatím v návrhu nevěnoval – je jí API brána. Ta zprostředkovává komunikaci mezi uživatelem



Obrázek 4.9: Sekvenční diagram procesu poptávky



Obrázek 4.10: Diagram komponent systému License manager

a jednotlivými mikroslužbami. Má tedy vazbu na všechny mikroslužby a vystavuje veřejné rozhraní. O API bráně a podobě veřejného API budu psát v kapitole 4.2.4.

V diagramu je taktéž vidět závislost Inquiries na Products a Orders. Naopak cesta asynchronních zpráv mezi Orders, Invoices, Licenses a Deployments je jen naznačena, jelikož přímá komunikace mezi těmito mikroslužbami neexistuje – všechny komunikují pouze s prostředníkem (ten v diagramu není zaznačen).

Diagram na obrázku 4.10 je zjednodušenou verzí, kde rozhraní mikroslužeb jsou pouze naznačeny oranžovými kuličkami. V této fázi návrhu už jsem ale definoval konkrétní metody rozhraní, jejich parametry i návratové hodnoty. Diagram komponent včetně definovaných rozhraní je na přiloženém médiu.

4.2.3 Stanovení komunikačních protokolů

Součástí společného návrhu je stanovení využívaných komunikačních protokolů. Počet používaných protokolů by měl být co nejmenší, dle odborné literatury nejvýše dva v jednom distribuovaném systému. [9]

Dvěma základními přístupy pro API jsou Represenatational State Transfer (zkr. REST) a Remote procedure call (zkr. RPC). REST je architektonický

	REST	**gRPC**
Orientace na	Data	Akce
Technologicky nezávislý	Ano	Ano
Přenos pomocí	HTTP, HTTP/2	HTTP/2
Přenášená data	Obvykle textový formát	Protocol buffer (efektivně serializovaná data)
Schéma	Dokumentační charakter	Nutné

Tabulka 4.2: Srovnání základních charakteristik RESTu a gRPC [59, 60]

styl, který určuje sadu omezení a protokolů, pro vytváření webových služeb. Koncovým bodem REST API je URL adresou jednoznačně určený datový zdroj, nad kterým jsou prováděny operace vytvoření, čtení, aktualizace a smazání (zkr. CRUD). RPC je metodika pro komunikaci využívající volání vzdálených funkcí/procedur. Hlavním rozdílem tedy je, že REST je orientovaný na zdroje (data) a RPC na operace. [59]

Osobně mi je pro mikroslužby bližší využití RPC, protože je orientované na akce a mikroslužby jsem také dělil podle funkcí, nikoli podle dat. Aby ale rozhodnutí bylo zodpovědné, provedu bližší porovnání výhod a nevýhod využití obou přístupů v projektu License manager.

Zatímco REST je standard, RPC existuje mnoho implementací. Jako kandidáta z řad RPC technologií volím gRPC. Jedná se o projekt s otevřeným zdrojovým kódem, původně vyvinutý společností Google a nyní zastřešený organizací Cloud Native Computing Foundation.

Srovnání základních charakteristik obou přístupů je v tabulce 4.2. Po zhodnocení jsem se přiklonil k využití gRPC jako protokolu pro komunikaci mezi mikroslužbami, a to zejména z níže uvedených důvodů.

Schema-first přístup

Schema-first (schéma na prvním místě) je přístup, kdy vývoj začíná definováním API všech mikroslužeb a jejich realizace následuje až poté. Tento přístup je obzvláště vhodný při vývoji nativních cloudových aplikací, protože umožňuje paralelní práci více týmů – všichni dopředu vědí, jaké rozhraní musí implementovat, a zároveň, jaké rozhraní poskytují ostatní mikroslužby. [61]

Schema-first přístup lze samozřejmě aplikovat i na REST API. Výhodou gRPC je, že schéma je pro komunikaci striktně vyžadováno a má standardizovanou notaci. Výstupem návrhu jsou soubory se schématy, tzv. protofily, které následně využívají servery i klienti. Toto schéma je závazné a technicky nelze tento kontrakt porušit. Kromě umožnění paralelního vývoje tedy navíc gRPC eliminuje riziko chyby způsobené špatným použitím API.

gRPC poskytuje knihovny pro jedenáct široce používaných serverových programovacích jazyků. Tyto knihovny zajišťují generování kódu jak pro server, tak pro klienty.

Jedním z úspěšných osvojitelů gRPC je společnost Netflix, která zveřejnila případovou studii adopce gRPC. Z jejích zkušeností vyplývá, že použitím gRPC se snížilo množství konfiguračního kódu o 99 % a implementace nového klienta mikroslužby se zkrátila ze 2-3 týdnů na jednotky minut. [62]

Efektivita komunikace

Druhým rozhodujícím aspektem je efektivita komunikace a výkonost distribuovaného systému založeném na gRPC. Výsledkem volání REST API jsou obvykle strukturovaná data v textové formátu jako JSON nebo XML. Velký objem přenesených dat, při využití těchto formátů, slouží pouze jako anotace, nikoli užitečné informace. gRPC přenáší data v binární podobě a minimalizuje množství anotací, protože kódování a dekódování je provedeno na základě schématu, které mají k dispozici server i klient.

Experiment zjišťující vlivy použití RESTu a gRPC na celkovou výkonost systému založeném na mikroslužbách provedl Tim Bastin. Pokus provedl na systému složeném ze 3 mikroslužeb a uživatelském požadavku (provedení objednávky), který vede na celkem 7 volání mezi mikroslužbami. Mikroslužby byly implementované v jazyce Go s oficiální knihovnou gRPC. Mikroslužby nevyužívají databázi, aby nedošlo ke zkreslení výsledků. [63]

Z měření vyplývá, že při nízké zátěži (10 simultánních uživatelů) systém postavený na gRPC dokázal zpracovat 3.4x více požadavků oproti systému využívajícímu REST. Při vyšší zátěži (100 simultánních uživatelů) je počet zpracovaných požadavků 3.7x vyšší a při 300 simultánních uživatelích gRPC systém zpracoval dokonce 4.8x více požadavků. Zároveň délka zpracování požadavku byla v případě gRPC vždy nižší a konzistentnější oproti RESTu. Autor také uvedl, že množství dat přenesených během jednoho požadavku bylo v jeho konkrétním případě o více než 30 % nižší. [63]

4.2.4 Veřejné rozhraní

Dosud jsem se věnoval komunikaci mezi mikroslužbami. Nezbytnou součástí webové aplikace, a tedy i jejího návrhu, je veřejné API. Veřejným rozhraním aplikací založených na mikroslužbách jsem se věnoval v teoretické části, kapitole 2.1.3. Popsal jsem, že komunikaci mezi vnějším světem a distribuovanou aplikací zajišťuje API brána, případně více bran, kdy každá je specializovaná pro potřeby jednoho typu klienta (např. mobilní aplikace, desktop aplikace, atd.).

Problém s návrhem rozhraní je, že současně s realizací backendu neprobíhá i realizace frontendu. Požadavky na podobu API tedy neexistují a vycházet

můžu pouze ze zkušeností s implementací prototypu uživatelského rozhraní původní verze aplikace.

gRPC API

Elementárním řešením brány by bylo vystavení gRPC metod všech mikroslužeb v nezměněné podobě. Brána by pak fungovala pouze jako prostředník s eventuální autorizací požadavků. Problém tohoto řešení je, že klient by musel obsahovat gRPC knihovnu a znát schéma mikroslužeb, což by ztížilo například testování nebo konfiguraci aplikace pomocí skriptů.

REST API

Další variantou je zveřejnění REST API a překlad požadavků na gRPC uvnitř API brány. REST je široce používaný a problémy s kompatibilitou mizí. Problémem ale zůstává, že v tuto chvíli nevím, jaká data bude uživatelské rozhraní na jednotlivých obrazovkách potřebovat a nemůžu provést jakoukoliv optimalizaci. Při využití RESTu by tedy API muselo být maximálně obecné, což je žádoucí, ale klientská aplikace by musela načíst data v několika požadavcích z různých koncových bodů, což je nežádoucí, protože načítací doba uživatelského rozhraní může být dlouhá.

GraphQL API

Existuje ale elegantní řešení, které zpřístupní obecné API, ale zároveň budou volání uzpůsobena potřebám jednotlivých uživatelských rozhraní a jejich obrazovek. Je jím GraphQL – *„dotazovací jazyk a běhové prostředí na straně serveru pro provádění dotazů nad systémem typů, definovaným pro poskytovaná data“*. [64]

GraphQL gateway obsahuje jednotné schéma (graf) všech dat v distribuovaném systému, a pro každý typ a jeho atribut funkci, které získá příslušná data. Dále schéma obsahuje operace dvou typů: dotazy a mutace. Dotazy jsou operace čtení a definují „vstupy“ do grafu, tedy typy, nad kterými je možné se dotazovat. Mutace je operace, která mění data. Mutace i dotazy mohou obsahovat parametry, podobně jako funkce v programovacích jazycích, a mají návratový typ. Uživatel specifikuje, které atributy návratového typu chce obdržet. Jeden požadavek může obsahovat několik dotazů i mutací.

GraphQL server načte dotaz a pro každý typ či atribut provede předem definovanou funkci tak, aby získal všechna data, která uživatel požaduje. Výsledky všech dotazů a mutací z jednoho požadavku vrátí v jedné odpovědi.

Hlavní výhody GraphQL jsou:

- Klient specifikuje, která data potřebuje a dostane jen a pouze tato.

- Klient může poslat více dotazů a mutací v jednom požadavku, což má pozitivní vliv na odezvu aplikace, například na pomalém mobilním připojení.
- GraphQL server vystavuje jediný koncový bod, přes který jsou realizovány veškeré operace.
- Požadavky jsou zasílány pomocí HTTP (metoda POST). Nejsou tedy kladeny žádné netypické požadavky na kompatibilitu klienta.
- Usnadňuje inkrementální změny API bez nutnosti verzování rozhraní. [65]

Díky GraphQL bude veřejné API maximálně obecné, ale zároveň přizpůsobitelné potřebám uživatelského rozhraní.

Kapitola 5

Implementace

V této kapitole navážu na návrh nové architektury aplikace její implementací. Fáze implementace probíhala agilně, v malých iteracích přidávajících postupně dílčí funkce jednotlivých mikroslužeb. Při implementaci jsem pracoval na více mikroslužbách současně.

Vzhledem k rozsahu aplikace nebudu v této kapitole zacházet do příliš velkého detailu a popisovat implementaci každé mikroslužby jednotlivě, ale popíšu výběr technologií, způsob práce a implementaci funkcí, které jsou přítomné ve všech mikroslužbách nebo specifické pro nativní cloudové aplikace.

5.1 Výběr technologie

Vlastností aplikací založených na mikroslužbách je možná technologická heterogenita. Protože všechny navrhované mikroslužby budou principem totožné, rozhodl jsem se použít společnou sadu technologií. Benefitem bude snadnější nastavení vývojového prostředí, možnost vytváření společných knihoven či potencionální sdílení vývojářů mezi mikroslužbami bez nutnosti učení se dalším technologiím.

Na technologii jsem si určil tyto požadavky:

- musí podporovat protokol gRPC,
- musí mít nízké výpočetní a paměťové nároky,
- měla by být spíše změřena na rychlé I/O operace, než složité výpočty,
- měla by být z již používaných ve společnosti Jagu (PHP, JavaScript, C#).

Jazyk PHP jsem stručně představil v analytické části, protože je v něm implementovaná současná verze aplikace.

JavaScript (zkr. JS) je lehký, interpretovaný programovací jazyk. Ve společnosti Jagu je používán primárně pro tvorbu uživatelských prostředí webových aplikací. S použitím běhového prostředí (např. Node.js) je ale použitelný i pro vývoj serverových aplikací. JS je dynamicky typovaný jazyk založený na prototypech (styl objektově orientovaného programování, kdy třídy nejsou definovány explicitně, ale odvozeny přidáním metod a atributů do instance jiné třídy [66]), podporující více programovacích paradigmat (objektově orientované, funkcionální). [67]

C# je víceúčelový, multiplatformní, objektově orientovaný a staticky typovaný programovací jazyk vyvíjený společností Microsoft. Pro tvorbu webových aplikací se používá spolu s frameworkem ASP.NET. Vyznačuje se velmi dobrým výkonem. [68]

Po analýze vlastností zmíněných technologií a doporučení expertů jsem se rozhodl mikroslužby implementovat v technologii Node.js. Node.js se vyznačuje neblokujícím I/O (vstupně výstupní operace jako zápis na disk nebo síťové volání neblokují proces [69]). Může tedy zpracovávat více požadavků v jednom vlákně současně. Tato vlastnost přináší velmi dobrý výkon při nízkém výpočetním a paměťovém zatížení. [70]

Node.js pro implementaci mikroslužeb využívá například Netflix, který přechodem z Javy snížil startovací čas systému o 70 %. LinkedIn přechodem z Ruby on Rails snížil náklady na provoz o 90 %. [71]

Kromě efektivity přinese použití JS i obrovskou výhodu jednotné technologie pro mikroslužby i uživatelské rozhraní. Společnost Jagu je spíše menší a lze očekávat, že vývojáři budou pracovat na frontendu i backendu. Společná technologie zrychlí proces vývoje aplikace i adaptace nových vývojářů.

Problémem JS pro vývoj bezpečných aplikací je chybějící explicitní definování typů a statická typová kontrola. Tento problém vyřeším použitím jazyku TypeScript (zkr. TS). TS je dialekt JavaScriptu vyvíjený společností Microsoft, který přidává syntaxi pro datové typy a statickou kontrolu při překladu. TS je překládaný do JS a aplikaci lze po překladu provozovat v Node.js. Pro implementaci License manageru jsem zvolil TS verze 4.9 (v době realizace nejnovější verze). [72]

5.2 Organizace kódu

Požadavkem zadavatele je verzovat kód systémem Git a ukládat ho do soukromého vzdáleného repozitáře GitLab společnosti Jagu. Přechodem na architekturu mikroslužeb ale vyvstává otázka, jak kód organizovat. Existují dva přístupy pro organizaci kódu distribuovaného systému – monorepo a polyrepo. Monorepo znamená, že kód všech mikroslužeb je uložen a verzován v jednom repozitáři. Polyrepo je naopak přístup, kdy každá mikroslužba je ve vlastním repozitáři.

Polyrepo lépe odpovídá požadavku na nezávislost vývoje mikroslužeb a nasazení jejich změn. Umožňuje také definovat přístupová práva vývojářů k jednotlivým mikroslužbám.

Monorepo je vhodné, pokud vývojáři chtějí pracovat na více mikroslužbách. Vybízí také ke sdílení kódu mezi mikroslužbami. V takovém případě je ale nutné dávat pozor, aby nedošlo k nechtěné provázanosti. Monorepo je výhodné při provádění rozsáhlého refaktorování napříč mikroslužbami, ale také pro společné změny frontendových a backendových komponent.

Oba přístupy mají své klady i zápory. Pro projekt License manager jsem se rozhodl využít monorepo, a to zejména kvůli příjemnější práci pro vývojáře bez neustálého přecházení mezi adresáři a snadnějšímu zapojení nových členů do vývojového týmu, kteří získají vše potřebné v jednom repozitáři.

5.2.1 Knihovny

Jedním z benefitů monorepa je snadné sdílení kódu mezi mikroslužbami, které může ušetřit mnoho času a podporuje princip DRY (*Do not repeat yourself* – neopakuj se). Redundance v kódu jsou nežádoucí, neboť snižují čitelnost kódu a v případě nutnosti změny je potřeba provést aktualizaci na mnoha místech.

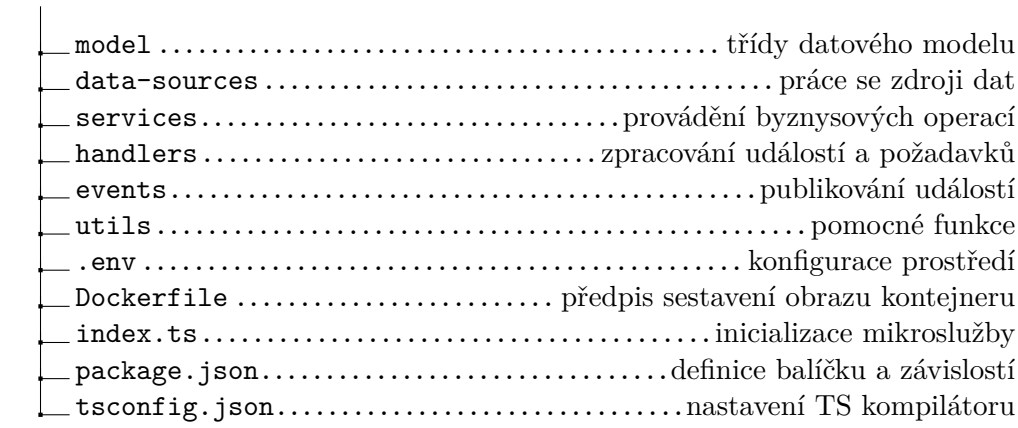
Jak jsem již ale uvedl, sdílení kódu může vést na silnou provázanost mikroslužeb. Abych tomuto jevu zabránil, určil jsem podmínky, pouze za kterých bude sdílení kódu možné. Podmínky zároveň odpovídají požadavku 2, „Závislosti“, z Twelve-factor mainifestu popsaného v teoretické části práce, podkapitole 1.3.

Sdílený kód bude dostupný pouze jako knihovna. Kód knihovny bude zcela izolovaný od kódu mikroslužeb. Knihovny budou vydávány ve verzích a ukládány do soukromého repozitáře knihoven ve službě GitLab. Mikroslužby, které budou chtít knihovnu použít, ji uvedou v seznamu svých závislostí a nainstalují pomocí správce balíčků (v případě JavaScriptu například npm, Yarn, PNPm).

5.2.2 Členění kódu typické mikroslužby

Monorepo projektu obsahuje dva stěžejní adresáře: `apps` a `packages`. První obsahuje podadresáře se samotnými mikroslužbami, druhý s knihovnami. Napříč mikroslužbami jsem v maximální možné míře udržoval jednotné členění kódu, které nyní krátce popíšu. Stromová struktura typické mikroslužby je na obrázku 5.1).

Adresář `model` obsahuje třídy datového modelu pro techniku ORM, které detailněji popíšu v podkapitole 5.8, v adresáři `data-sources` jsou potom funkce pro práci s modelem/databází. V adresáři `services` jsou umístěny třídy a funkce realizující byznysovou logiku. Funkce, které zpracovávají příchozí požadavky a události, jsou umístěny v adresáři `handlers`. Funkce realizující publikování událostí různých typů jsou v adresáři `events`. Sdílené



Obrázek 5.1: Členění kódu typické mikroslužby

pomocné funkce (například filtrování dat, zpracování textových řetězců) jsou v adresáři `utils`.

Soubor `index.ts` je vstupní bod do aplikace – soubor, který je předán běhovému prostředí Node.js. Obsahuje inicializaci aplikace, tedy načtení potřebných modulů, nastavení připojení k databázi, dalším mikroslužbám či inicializaci web serveru poskytujícího API. Zodpovědný je také za načtení proměnných prostředí, na základě kterých je inicializace provedena.

Proměnné prostředí jsou definované v souboru .env. Jedná se například o údaje nutné k připojení k databázi, adresy jiných mikroslužeb, port aplikace a podobně. Definování těchto konfigurací jako proměnné prostředí je zároveň jedním z požadavků Twelve-factor (viz1.3).

Soubor `package.json` obsahuje metadata modulu (mikroslužby, knihovny) jako název, verzi, jméno autora, ale také produkční a vývojové závislosti aplikace (seznam veřejných a privátních knihoven a jejich požadovaných verzí). Závislosti z tohoto seznamu jsou následně nainstalovány zvoleným správcem balíčků.

Každé mikroslužbě přísluší `Dockerfile`, na základě kterého je sestaven obraz produkčního kontejneru. O Docker kontejnerech jsem psal v kapitole 1.4.1.

Posledním souborem, který je součástí všech mikroslužeb a knihoven, je `tsconfig.json`, který slouží ke konfiguraci TypeScript kompilátoru. Mezi parametry jsou například zdrojové soubory, úroveň statické kontroly, cílový adresář kompilace nebo verze JavaScriptu, do které mám být TypeScript přeložen. Mikroslužby License manageru jsou překládány do standardu ES2022 (aktuální verze standardu v době realizace práce).

5.3 Vývojové prostředí

Zatímco v produkčním prostředí budou aplikace provozovány v kontejnerech při vývoji je nutné, aby aplikace běžela přímo na počítači vývojáře a změny v kódu se okamžitě projevovaly. V této podkapitole popíšu používané nástroje při vývoji aplikace.

Nejdůležitější částí vývojového prostředí je Node.js – běhové prostředí JavaScriptu. Zvolil jsem verzi 18, nejnovější verzi s dlouhou podporou. Verze 18 bude podporovaná ještě dva roky po dokončení práce a nebude nezbytně nutné po tuto dobu přejít na novější verzi.

Už jsem zmínil, že závislosti aplikace instaluje správce balíčků. Výchozím správcem Node.js je npm. Rozhodl jsem se ale použít správce balíčků Yarn, který nabízí paralelní instalaci závislostí, zatímco npm instaluje závislosti sekvenčně. [73]

V předchozí podkapitole jsem vybral monorepo pro organizaci kódu. Monorepo může být prostým společným ropozitářem. Abych maximalizoval výhody a zároveň minimalizoval problémy, využil jsem nástroje pro práci s monorepem. Těchto nástrojů je celá řada (například Lerna, nx nebo Turborepo), poskytované funkce jsou ale u všech velmi podobné. Pro projekt LM jsem zvolil Turborepo, které je snadné na nastavení i práci s ním.

Každý mikroslužba obsahuje v souboru `package.json` implementaci těchto třech příkazů:

- **build** – překlad aplikace do JavaScriptu
- **start** – spuštění přeložené aplikace
- **dev** – překlad a spuštění aplikace se sledováním změn

Příkaz `build` přeloží aplikaci do JavaScriptu. Tento příkaz se volá i během sestavení obrazu kontejneru. Takto přeložený kód se spustí příkazem `start`. Při vývoji je velmi užitečný příkaz `dev`, který spouští kód v TypeScriptu, na pozadí ho překládá a sleduje, zda-li byly provedeny změny v kódu. Pokud ano, automaticky přeloží novou verzi a aplikaci restartuje. Všechny tři operace se spouští příkazem `yarn <prikaz>` v adresáři mikroslužby.

Turborepo přináší možnost spouštět zmíněné příkazy najednou pro všechny mikroslužby v monorepu. Příkaz `dev` v kořenovém adresáři monorepa tedy spustí celý distribuovaný systém. Navíc, má-li příkaz nějaký výstup (například `build` zapisuje zkompilované aplikace do adresářů `dist`), Turborepo sleduje, zda-li od posledního spuštění příkazu došlo ke změně kódu mikroslužby. Pokud ne, tak výstup načte z cache. Nespouští tedy příkaz pro nezměněnou mikroslužbu. Díky cachování výsledků příkazů a jejich paralelnímu zpracování je práce v monorepou rychlá i přes větší množství mikroslužeb.

Příkaz `build` kromě mikroslužeb implementují i knihovny. Pokud mikroslužba používá knihovnu, která je součástí monorepa, tak při vývoji Turborepo

použije lokální kód, nikoli publikovanou verzi (pokud se schodují verze lokální knihovny a závislosti). Vývojář tak snadno může otestovat změny v knihovnách. Navíc při spuštění příkazu `build` Turborepo nejprve rekurzivně sestaví všechny lokální závislosti modulu, až potom modul samotný.

Mikroslužby využívají podpůrné služby – databáze a message broker, které jsou pro chod mikroslužeb v lokálním vývojovém prostředí nezbytné. Způsob jejich provozu je na rozhodnutí každého vývojáře. Konfiguraci připojení mikroslužbám předá prostřednictvím proměnných prostředí (soubor `.env`). Podpůrné služby jsem spouštěl v jejich kontejnerizované podobě nástrojem Docker. Pro zjednodušení začátku práce budoucích vývojářů jsem připravil soubor `docker-compose.yaml`, jehož aplikování příkazem `docker compose` spustí všechny potřebné podpůrné služby v Dockeru. Jaké konkrétní podpůrné služby License manager používá, popíšu v následujících podkapitolách.

5.4 Synchronní komunikace

V kapitole 4 jsem navrhl komunikaci mezi mikroslužbami pomocí gRPC. V této podkapitole popíšu její implementaci.

gRPC komunikace je realizována na základě schémat, které definují dostupné metody serveru, jejich parametry a návratové hodnoty. Schéma je zapsáno v souboru zvaném protofile. Ukázka kódu 1 obsahuje definice metod služby Invoices. Parametry a návratové hodnoty funkcí jsou vždy objekty určitého typu, který je taktéž definovaný ve schématu. Na ukázce kódu 2 jsou definice typů `Invoicing` a `Invoicings`. Za zdůraznění stojí čísla „přiřazená“ atributům. Nejedná se o přiřazení hodnoty, ale určení pořadí, v jakém budou atributy serializovány. Druhý typ na ukázce obsahuje pole, které je určeno klíčovým slovem `repeated`.

Protože dostupnost schématu je nezbytná jak pro server, tak klienta, musel jsem vymyslet způsob, jak distribuci schémat zajistit. Naivním řešením by bylo vložit kopii protofilu do každé mikroslužby, která se účastní dané komunikace. V okamžiku, kdy by se schéma změnilo, bylo by nutné nahradit všechny kopie. Tento způsob by byl neudržitelný a znamenal by vysoké potencionální riziko vzniku nekonzistencí mezi klienty a serverem.

Rozhodl jsem se se schémata pracovat jako s verzovanými závislostmi – knihovnami. Každá mikroslužba má vlastní schéma, které je umístěno v separátní knihovně. Knihovny jsou v monorepu v již zmíněném adresáři `packages`.

Nezůstal jsem ale u pouhého uložení protofilu v knihovně a využil funkcionality gRPC knihovny – generování TypeScript typů podle schématu. Při operaci `build` knihoven se schémata se tak provede generování typů a jejich překlad do JavaScriptu. Všechny tyto soubory pak tvoří knihovnu a jsou publikovány do registru knihoven na GitLabu.

Vygenerované typy v knihovně umožňují statickou typovou kontrolu implementace komunikace jak na straně serveru, tak klienta. Při sestavení serveru je

```
syntax = "proto3";
package InvoicesPackage;
service InvoicesService {
  rpc getInvoicings(Void) returns (Invoicings) {};
  rpc getInvoicing(IdRequest) returns (Invoicing) {};
  rpc getInvoicingsByUser(StringIdRequest) returns (Invoicings)
↪  {};
  rpc getBillingInfos(Void) returns (BillingInfos) {};
  rpc getBillingInfo(IdRequest) returns (BillingInfo) {};
  rpc getBillingInfosByUser(StringIdRequest) returns
↪  (BillingInfos) {};
  rpc createBillingInfo(CreateBillingInfoRequest) returns
↪  (BillingInfo) {};
}
```

Ukázka kódu 1: Ukázka definice funkcí gRPC

```
message Invoicing {
  int32 id = 1;
  int32 orderId = 2;
  int32 oneTimePayment = 3;
  int32 regularPayment = 4;
  int32 period = 5;
  BillingInfo billingInfo = 6;
}

message Invoicings {
  repeated Invoicing invoicings = 1;
}
```

Ukázka kódu 2: Ukázka definice typů gRPC

kontrolováno, že gRPC handler implementuje veškeré funkce ze schématu, a že odpověď na požadavek odpovídá návratovému typu ve schématu. Na ukázce kódu 3 je úryvek gRPC handleru požadavků mikroslužby Invoices. Na prvním řádku je importování definice rozhraní handleru z knihovny schématu. TypeScript následně odvodí typy parametrů a návratových hodnot funkcí zpracovávajících konkrétní požadavky. Na ukázce je také vidět funkce `handlerWrapper`, což je má pomocná funkce pro dodatečnou validaci vstupů a jednotné zpracování chyb. Objekt `handlers` obsahující zpracovávající funkce, je při inicializaci aplikace předán jako parametr gRPC serveru, který potom volá při obdržení požadavku příslušné funkce.

Implementace popsaného procesu distribuce schémat sice zabrala několik

```
import { InvoicesServiceHandlers } from
↪  "@lm/invoices-proto/InvoicesPackage/InvoicesService";


const handlers = (dataService: InvoicingDataService,
↪  fakturoidService: FakturoidService): InvoicesServiceHandlers
↪  => ({
  getInvoicings: handlerWrapper(
    (call) => ({}),
    async (_required, _call) => {
      const invoicings = await dataService.getInvoices();
      return { invoicings };
    }
  ),
  getInvoicing: handlerWrapper(
    (call) => ({ id: call.request.id }),
    async (required, _call) => {
      return await dataService.getInvoice(required.id!);
    }
  ),

  ...
  };
```

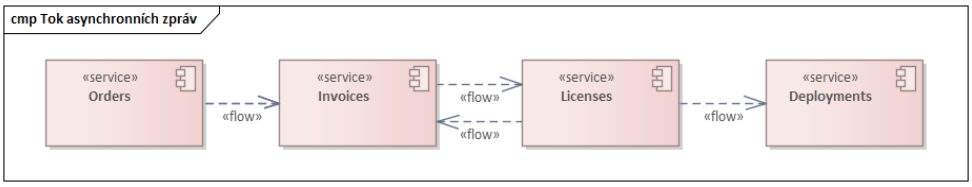
Ukázka kódu 3: Ukázka gRPC handleru (mikroslužba Invoicing)

dní práce, ale benefity v podobě statické typové kontroly (a našeptávání kódu při vývoji serverů i klientů) tento čas plně kompenzovaly. Díky statické typové kontrole jsem okamžitě objevil velké množství chyb, které by jinak byly odhaleny až při testování, nebo hůře, až v ostrém provozu.

5.5 Asynchronní komunikace

Jak jsem popsal v návrhu nového řešení, od okamžiku odeslání objednávky bude proces jejího zpracování prováděn pomocí asynchronních volání. Nyní popíšu, jak jsem asynchronní komunikaci implementoval.

Asynchronní komunikace je zajištěna pomocí prostředníka, kterému se říká *message broker*. Výběr technologie brokeru jsem konzultoval s vedoucím práce, abych využil technologii, kterou již společnost Jagu využívá na svých stávajících projektech. Zvoleným je RabbitMQ – široce používaný message broker s otevřeným zdrojovým kódem, možností horizontálního škálování a nízkými výpočetní nároky. [74]



Obrázek 5.2: Tok asynchronních zpráv

5.5.1 Typy zpráv

Nyní definuji typy zpráv, které si budou mikroslužby posílat. Také předepíšu formát zpráv asynchronní komunikace, tzv. data kontrakt. Obě strany komunikaci budou muset tento kontrakt dodržet. [16] Na obrázku 5.2 je výňatek z diagramu komponent znázorňující tok zpráv.

Orders → Invoices

Mikroslužba Orders předává mikroslužbě Invoices zprávy o potvrzených objednávkách. Na základě těchto zpráv jsou vystaveny faktury. Název zprávy je `order_created`. Pro obsah zprávy jsem zvolil formát JSON (JavaScript Object Notation), který je JavaScriptem, jak už název napovídá, nativně podporovaný. Formát je na ukázce kódu 4.

```
{
  "orderId": "<id_objednavky>",
  "oneTimePayment": "<jednorazova_platba>",
  "regularPayment": "<pravidelna?platba>",
  "period": "<interval_pravidelnych_plateb>",
  "billingId": "<identifikator_fakturacniho_kontaktu>",
  "name": "<nazev_objednavaneho_produktu>"
}
```

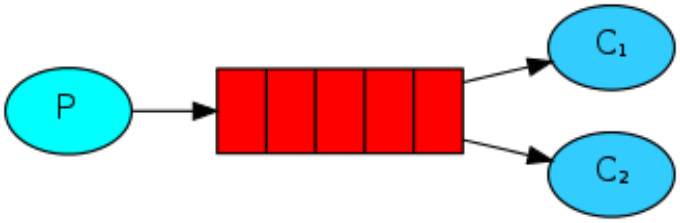
Ukázka kódu 4: Formát zprávy `order_created`

Invoices → Licenses

Mikroslužba Invoices předává mikroslužbě Licenses zprávy o přijaté platbě (`order_paid`) a platbě po splatnosti (`order_payment_overdue`). Pro oba typy zpráv je formát identický (ukázka kódu 5). Jak je z názvů a formátu zpráv patrné, platby se vztahují vždy k objednávce, proto je ve zprávě její identifikátor. Dále zpráva obsahuje identifikátor zákazníka a délku fakturačního období, o kterou je prodloužena platnost licence.

```
{
  "orderId": "<id_objednavky>",
  "period": "<interval_pravidelnych_plateb>",
  "userId": "<identifikator_uzivatele>"
}
```

Ukázka kódu 5: Formát zprávy `order_paid` a `order_payement_overdue`



Obrázek 5.3: Schéma message brokeru

Licenses → Deployments

Mikroslužba Deployments je mimo rozsah této práce. Přesto jsem navrhl a implementoval publikování zpráv na straně mikroslužby Licenses. Licenses publikuje tyto typy zpráv: `new_deployment` pro vytvoření nové zákaznické instance produktu , `block_deployment` pro zablokování instance, `unblock_deployment` pro reaktivaci zablokované instance a `delete_deployment` pro definitivní smazání instance. Všechny zprávy obsahují pouze identifikátor objednávky, na základě kterého bude mikroslužba Deployments načítat uživatelskou konfiguraci z mikroslužby Products.

Toto schéma je možné upravit podle budoucích potřeb. Předávat je možné například číslo licence, či identifikátor uživatele, navíc mohou být posílány zprávy o prodloužení platnosti. Integrace mikroslužeb pro nasazení a správu instancí bude předmětem případného dalšího vývoje aplikace.

Licenses → Invoices

V případě zrušení licence je nutné ukončit i vystavování pravidelných faktur předplatného. O zrušení licence je informována mikroslužba Invoices zprávou typu `license_cancelled`, která obsahuje identifikátor objednávky.

5.5.2 Vzor asynchronní komunikace

Pomocí RabbitMQ lze realizovat několik komunikačních vzorů. Pro potřeby LM jsem se rozhodoval mezi pracovními frontami a vzorem PubSub (publikování, odběr).

Schéma komunikace je na obrázku 5.3. Producent publikuje zprávy do úložiště brokeru a ten je rozesílá konzumentům. Rozdíl mezi pracovní frontou a PubSub je ten, že pracovní fronta pošle zprávu právě jednomu konzumentovi, zatímco PubSub ji pošle všem konzumentům, kteří přihlásili odběr.

V mém případě na událost bude reagovat vždy jeden typ konzumenta, který ale bude mít více instancí (kvůli horizontálním škálování). Protože po přijetí zprávy bude zpravidla následovat reakce (určitá byznysová funkce), je nutné, aby zprávu přijal pouze jeden příjemce. Použiji tedy vzor pracovních front.

5.5.3 Zpracování událostí

Každý typ zprávy je publikován do samostatné fronty, které jsou pojmenované podle typů zpráv. Mikroslužby při inicializaci vytvoří jedno spojení s RabbitMQ, po kterém bude probíhat veškerá komunikace (publikování i odběry). Následně se přihlásí k odběru příslušných front funkcí `consume`, které jako argument předám funkci volanou při přijetí zprávy.

Využití message brokeru zvyšuje dostupnost systému, protože není nutná 100% dostupnost mikroslužeb. Dalším cílem použití asynchronní komunikace je větší odolnost vůči chybám. Vezmu-li například situaci, kdy je zpráva o platbě doručena mikroslužbě Licenses, ale při jejím zpracování dojde k chybě, tedy není prodloužena licence, jedná se z obchodního hlediska o závažnou chybu. RabbitMQ tento problém řeší potvrzovacími zprávami. Mikroslužba po dokončení zpracování zprávy odešle RabbitMQ potvrzení o úspěšném zpracování. Pokud RabbitMQ neobdrží v předem daném časovém intervalu toto potvrzení, odešle zprávu znovu. Až při úspěšném potvrzení zprávu smaže ze svého úložiště.

Pro zvýšení odolnosti systému jsem všechny fronty nakonfiguroval pro ukládání zpráv na disk. Zprávy v brokeru tedy zůstanou i v případě jeho pádu.

5.6 API brána

Ve fázi návrhu jsem určil, že API brána bude vystavovat GraphQL API. Abych vytvoření GraphQL serveru zjednodušil, rozhodl jsem se použít rozšířenou implementaci s otevřeným zdrojovým kódem pro Node.js – Apollo Server. Výhodou Apollo Serveru, kromě početné komunity uživatelů a velkého množství informačních materiálů, je také přidružená knihovna Apollo Client. Ta zjednodušuje implementaci volání serveru z uživatelského rozhraní a zajištění například cachování výsledků požadavků na straně klienta. Dostupná je, mimo jiné, pro framework Vue.js, který společnost Jagu využívá pro implementaci uživatelských rozhraní. [75]

Při použití Apollo Serveru mi zbývalo definovat schéma a implementovat resolvery (funkce volané Apollem, které získávají nebo upravují určitá data a slouží k naplnění požadavku).

```
type Product {
    id: ID!
    builderId: String
    name: String!
    description: String
    customConf: Boolean!
    state: ProductState!
    version: String
    plans: [Plan]
    configurationInputs: [ConfigurationInput]
}


type Query {
    ...
    products: [Product]
    product(id: Int!): Product
    ...
}

type Mutation {
    ...
    createProduct(builderId: String!, name: String!,
    ↪  description: String!, customConf: Boolean!, state:
    ↪  ProductState!, version: String!): Product
    ...
}
```

Ukázka kódu 6: Ukázka GraphQL schéma

5.6.1 Schéma

Schéma GraphQL brány je definováno jazykem GraphQL. Fungování GraphQL jsem již teoreticky popsal v kapitole Návrh. Pro názornost je na ukázce kódu 6 úryvek definice schématu. Na ní je nejprve definovaný datový typ `Product` (mezi atributy je například `plans` obsahující pole cenových plánů produktů). V typu `Query` jsou definovány všechny dostupné dotazy a v typu `Mutation` mutace (úpravy dat). V případě dotazů i mutací lze na ukázce vidět argumenty těchto funkcí a návratové typy. Funkce definované v GraphQL

schématu kopírují funkce mikroslužeb dostupné přes gRPC. Dá se tedy říci, že GraphQL schéma vzniklo sjednocením datových modelů a gRPC funkcí všech mikroslužeb.

5.6.2 Agregace

Z povahy GraphQL, které skládá požadovanou odpověď z dílčích částí, provádí brána agregované operace. Tyto agregace však neobsahují byznysovou logiku (skládání požadavků slouží pouze k získání potřebných dat, nikoli k provedení vícekrokového obchodního procesu). Bráně nejsou přidávány funkce, které by porušily princip vysoké soudržnosti. Během implementace jsem ale narazil na problém, který jsem vyřešil dodatečnými agregacemi v bráně. Například při vytváření nové objednávky je mikroslužbě Orders předáván identifikátor produktu. Potenciálně by uživatel ale mohl vytvořit objednávku neexistujícího produktu. Řešení byly dvě: validovat existenci produktu voláním mikroslužby Products z mikroslužby Orders, nebo validovat požadavek už v API bráně.

Rozhodoval jsem se tedy mezi silným provázáním mikroslužeb, které jsem se snažil ve fázi návrhu maximálně eliminovat a přidáním další funkce bráně. Jak jsem již naznačil, přiklonil jsem se k druhé variantě. Brána musí mít napojení na všechny mikroslužby, validací tedy nevzniká žádná další vazba. V teoretické části jsem popsal, že názory na funkce brány se v odborné veřejnosti liší. Osobně ji ale vnímám jako bod systému, který zodpovídá za to, že všechny přístupy k datům a manipulace s nimi jsou oprávněné. Řešení popsaného problému validace požadavku v bráně jsem tedy vyhodnotil jako přijatelné.

5.6.3 Cache

GraphQL umožňuje vývojářům definovat agregovaný požadavek na míru jejich potřebám. Toto zároveň ale přináší riziko příliš komplexních dotazů pro jejichž vypořádání bude muset GraphQL server provést i desítky volání mikroslužeb. Takový dotaz zákonitě povede na dlouhou dobu jeho zpracování. Abych zajistil maximálně rychlou odezvu aplikace, využil jsem možnosti Apollo Serveru pro cachování odpovědí.

Při prvním volání určitého požadavku je odpověď uložena do mezipaměti. Pokud je požadavek volán znovu, odpověď je načtena z mezipaměti a okamžitě odeslána. Zároveň je požadavek standardně zpracován a jeho výsledkem aktualizována hodnota v mezipaměti.

Ve výchozím stavu Apollo Server automaticky cachuje odpovědi jako celek. Vzhledem k velkému množství variant požadavků to ale není nejefektivnější řešení. Proto jsem implementoval cachování dílčích odpovědí pomocí knihovny `apollo-datasource-grpc`. Neukládají se tedy celé odpovědi, ale data, ze kterých jsou odpovědi skládány.

Nyní jsou i velmi komplexní dotazy, jejichž vyřízení bez mezipaměti trvalo v řádu jednotek sekund, vyřízeny v intervalu nižších desítek milisekund.

Cachování se neaplikuje na mutace – vrácen je vždy skutečný výsledek operace.

5.6.4 Nástroje brány pro vývojáře

Další výhodou použití Apollo Serveru je integrované grafické rozhraní pro testování API - Apollo Studio. Rozhraní je dostupné z webového prohlížeče, je-li brána spuštěna ve vývojářském režimu.

Grafické rozhraní je určeno vývojářům frontendu aplikace License manager a nabízí dvě důležité funkce.

První je dokumentace API. Ve Studiu je možné zobrazit datové typy ve schématu, dostupné dotazy a operace. Navíc u všech operací a typů jsou zobrazeny dodatečné komentáře, které jsem do schématu zapsal. License manager tedy nevyžaduje dodatečnou API dokumentaci.

Druhou funkcí Studia je možnost spustit zdokumentované operace. Grafické rozhraní obsahuje nástroj pro sestavení dotazu – i vývojář méně znalý API je díky tomu schopný sestavit validní dotaz, který maximálně využije potenciál GraphQL brány. Po provedení dotazu Studio zobrazí výsledek v přehledné podobě. Sestavený dotaz je možné zkopírovat pro použití v GraphQL klientovi nebo vygenerovat příkaz pro nástroj cURL.

Schéma z Apollo Serveru je schopný načíst také například populární nástroj pro práci s API – Postman. Volba nástroje tedy zůstává plně na vývojáři.

5.7 Autentizace a autorizace

Autentizaci a autorizaci jsem po teoretické stránce věnoval v kapitole 2.1.4. Z teorie jsem při realizaci autentizace a autorizace věděl, že je nutné použití SSO. Nyní popíšu, jak jsem ověřování totožnosti a práva na přístup realizoval.

5.7.1 Poskytoval identity

Na trhu jsou k dispozici existující implementace poskytovatele identity. Nebyl důvod tato existující řešení nevyužít. Volbu poskytovatele jsem opět konzultoval s vedoucím práce, aby odpovídala požadavkům společnosti Jagu. Vznikl tak nový funkční požadavek na autentizaci prostřednictvím nástroje Keycloak, který Jagu postupně adoptuje na svých projektech.

Keycloak je nástroj pro správu identit a přístupů s otevřeným zdrojovým kódem, spravovaný společností Red Hat. Keycloak podporuje standardní autentizační a autorizační protokoly OpenID Connect, OAuth 2.0 a SAML. [76]

Autentizaci LM jsem implementoval pomocí protokolu OpenID Connect (zkr. OIDC). OIDC je autentizační protokol, který umožňuje klientům různých typů získávat informace o koncových uživatelích a jejich aktivních re-

lacích. [77] Proces ověření spočívá v přesměrování uživatele na přihlašovací stránku Keycloaku, kde klient na základě přihlašovacích údajů obdrží ID token a následně jej zasílá License manageru s každým požadavkem. Bližší popis protokolu je nad rámec této práce.

5.7.2 Autorizace

Autorizace přístupů bude prováděna na základě uživatelských rolí a vlastnictví. Přístup na základě vlastnictví znamená, že uživateli je umožněno manipulovat s daty, která mu patří (například vlastní objednávky, licence, faktury). Přístup na základě vlastnictví bude typický pro zákazníky.

Zaměstnanci, kteří se budou starat o prodej a chod aplikace, potřebují přístup i k datům zákazníků. U nich se tedy bude uplatňovat přístup na základě role. Role opravňuje k provádění určité operace nebo skupiny operací. Jsou vytvářeny a přidělovány v administrátorském rozhraní Keycloak. Prozatím jsem definoval tři role:

1. `inquiries_sales` – uživatel může vyřizovat poptávky
2. `orders_admin` – uživatel může provádět veškeré operace týkající se objednávek
3. `products_admin` – uživatel může spravovat nabídku produktů

Uživatelé jsou rozděleni do třech skupin, kterým jsou přiděleny různé role:

1. **administrátoři** – mají přiděleny všechny role
2. **prodejci** – mohou vyřizovat poptávky
3. **zákazníci** – nemají žádné z výše přidělených rolí

Přiřazení rolí skupinám lze měnit v Keycloak administraci za běhu aplikace a okamžitě se projeví. Změna definice rolí zpravidla vyžaduje úpravy autorizační logiky v kódu License manageru.

Pro autorizaci přístupu je postačující role. Až pokud uživatel nemá potřebnou roli, ověřuje se vlastnictví zdroje, ke kterému přistupuje.

5.7.3 Implementace v API bráně

V podkapitole 5.6.2 jsem popsal, že v bráně budou prováděny validace požadavků, které se týkají dat napříč mikroslužbami. Uvedl jsem, že o bráně přemýšlím jako o místě, které je zodpovědné za správnost požadavku. Při této filozofii dává smysl do brány umístit i autentizaci a autorizaci.

Při implementaci jsem se držel doporučeného postupu dokumentací Apollo Serveru. Při inicializaci brány se registruje funkce, která je volána při obdržení každého požadavku, a která v případě, že hlavička obsahuje token, provede

autentizaci uživatele. Informace o autentizovaném uživateli jsou potom vloženy do kontextu požadavku (objekt, který provází požadavek) a mají k nim přístup všechny resolvery.

Autorizační logiku jsem vložil do resolverů (funkce zpracovávající dílčí požadavky). Příklad je na ukázce kódu 7. Voláním pomocné funkce `auth` je kontrolováno, zda požadavek odeslal autentizovaný uživatel. První parametr obsahuje informace o uživateli z kontextu, druhý parametr specifikuje roli, která má k danému zdroji přístup a třetí je výraz ověřující vlastnictví zdroje v případě, že uživatel nemá potřebnou roli. Na příkladu je ověření přístupu k objednávce.

Selže-li autentizace nebo autorizace, je vyhozena výjimka a odeslána odpověď s HTTP kódem 401 (uživatel nepřihlášen) nebo 403 (neautorizovaný přístup).

```
{
    order: async (_: any, { id }: { id: number }, { dataSources,
    ↪  userSession }: Context) => {
          const order = await
          ↪  dataSources.orderAPI.getOrderById(id);
          auth(userSession, Roles.ORDERS_ADMIN, (requestUserId:
          ↪  string) => order?.userId == requestUserId);
          return order;
    }
}
```

Ukázka kódu 7: Příklad autorizace požadavku v resolveru

5.7.4 Validace tokenu

Uživatelské požadavky (při volání zabezpečeným koncových bodů) musí mít v hlavičce `Authorization` ID token ve formátu JWT (JSON Web Token). JWT je zakódovaný JSON obsahující informace o identitě uživatele. Skládá se ze třech částí: hlavičky (specifikace typu tokenu a jeho podpisu), obsah (samotné informace o identitě) a podpis (podpis tokenu vytvořený s použitím soukromého klíče poskytovatele identity). [78]

V případě LM obsahuje JWT mimo jiné unikátní identifikátor uživatele, jeho jméno, e-mail, seznam přiřazených rolí a čas vypršení platnosti tokenu.

Token obsahuje veškeré potřebné informace o uživateli, ale identifikační informace samy o sobě žádným způsobem nezaručují, že požadavek skutečně odeslal tento uživatel. Je nutné provést ověření pravosti tokenu. Na výběr jsem měl dva přístupy validace: online, kdy platnost tokenu je ověřena voláním poskytovatele identity, nebo offline, kdy je ověřen podpis tokenu veřejným klíčem poskytovatele identity.

Výhodou offline validace je, že není nutné při každém požadavku volat poskytovatele identity, a tím se zvyšuje dostupnost systému. Nevýhodou je, že již vystavený token nelze invalidovat a uživatel ho může používat až do konce jeho platnosti. I přes toto negativum jsem se rozhodl použít právě offline validaci. V případě zvýšených nároků na zabezpečení je možné v Keycloaku nastavit krátkou platnost tokenu.

Díky offline validaci tokenu jsem zcela odstranil vazbu LM na Keycloak, se kterým bude komunikovat pouze frontend. Jediný nutný krok pro nastavení autentizace a autorizace při nasazení aplikace je předání veřejného klíče poskytovatele identity API bráně.

5.8 Perzistence

V této podkapitole se budu věnovat perzistenci dat mikroslužeb v databázi. V kapitole 4.1.1 jsem navrhl konceptuální datový model jednotlivých mikroslužeb. Ve fázi implementace jsem iterativně konceptuální modely realizoval.

Preferovanou databázovou technologií ve společnosti Jagu je PostgreSQL – objektově-relační databázový systém s otevřeným zdrojovým kódem. Z konceptuálních modelů je vidět, že datové modely většiny mikroslužeb jsou velmi jednoduché (jedna, maximálně dvě entity). Pro všechny tyto mikroslužby jsem použil PostgreSQL. Mikroslužba `Products` se ale složitostí svého datového modelu vymyká ostatním.

Konfigurace produktů mohou nabývat podoby komplexního grafu, nad kterým je nutné efektivně provádět dotazy a validace. V původní verzi LM jsem tento graf ukládal do relační databáze – přístup to byl funkční, ale dotazy nad daty byly složité na vytvoření a náročné na zpracování.

V nové verzi jsem se rozhodl pro uložení produktů a jejich konfigurací použít databázi, která lépe odpovídá podobě ukládaných dat – databázi grafovou.

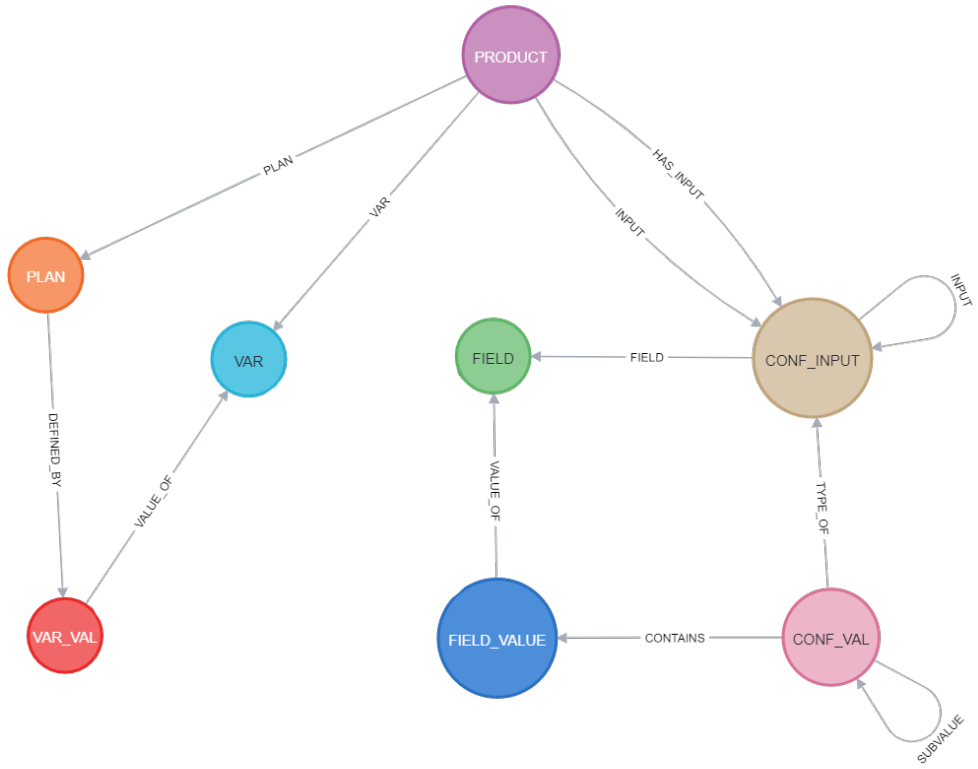
5.8.1 Použití grafové databáze

Mezi nejznámější grafové databáze patří Neo4j. Neo4j je nativní grafová databáze, to znamená, že data jsou fyzicky uložena jako graf (uzly obsahují ukazatel na adresu uložení svých sousedů). Neo4j uvádí, že pro data s mnoha vazbami a komplexní dotazy nad nimi je vyhodnocení až tisíckrát rychlejší oproti použití relační databáze.

Každý záznam v databázi odpovídá jednomu uzlu grafu. Neo4j nemá schéma a každý uzel může mít libovolné atributy. Atributy lze rovněž přiřazovat vazbám. Jistou analogií k názvům tabulek jsou tagy, kterými lze označit uzly, ale i vazby.

Z pohledu vývojáře je velkým benefitem dotazovací jazyk Cypher, který ve velmi kompaktním zápisu umožňuje psát velmi složité dotazy nad grafem.

Možnosti jazyka Cypher lze navíc rozšířit knihovnou APOC (Awesome Procedures On Cypher). Pro účely License manageru jsem vytvořil vlastní



Obrázek 5.4: Schéma grafu mikroslužby Products

obraz Docker kontejneru Neo4j s předinstalovanou knihovnou APOC. Tuto knihovnu využívám například pro duplikaci produktu při úpravě, což je operace duplikace podgrafu. Tato operace v původní verzi s relační databází byla složitá – kopírování záznamu a rekurzivně všech, co s ním mají vztah.

Vygenerované schéma grafu z grafického rozhraní Neo4j je na obrázku 5.4 a odpovídá téměř 1:1 konceptuálnímu modelu. Obsahuje navíc pouze vazbu Konfiguračních hodnot a Konfiguračních vstupů sama na sebe (která, jak jsem poznamenal při návrhu, nelze v UML ERD zachytit). Přidána je také vazba s názvem Input z Produktu do Konfiguračního vstupu, která značí nastavení kořenové otázky konfiguračního formuláře produktu. Popisky uzlů v grafu na obrázku značí tagy (obdoby názvů tabulek v relační databázi).

Nevýhodou použití Neo4j je, že neexistují pokročilejší JavaScript knihovny

pro práci s ní. Oficiální knihovna je pouze klient, který zajišťuje odeslání textového řetězce se Cypher dotazem. Použití Neo4j tedy znamenalo pro každou databázovou funkci vytvořit Cypher dotaz a ten vložit jako parametr do Neo4j klienta. Na ukázce kódu 8 je získání cenových plánů produktu z databáze. První parametrem metody `run` Neo4j klienta je Cypher dotaz, druhým je objekt s hodnotami, kterými lze Cypher dotaz parametrizovat (na ukázce je parametrem identifikátor produktu). Typickým problémem bylo aktualizování výčtu atributů projekce (část Cypher dotazu na klíčovým slovem `return`) ve všech dotazech vracejících data uzlů s daným tagem, pokud jsem provedl změnu struktury datového typu.

```
await session.run(
          "match (p:PRODUCT)-[:PLAN]->(plan:PLAN) where id(p) =
          ↪  $productId optional match
          ↪  (plan)-[:DEFINED_BY]->(val:VAR_VAL)-[:VALUE_OF]->(var:VAR)
          ↪  with plan, {id: id(var), name: var.name, val:
          ↪  val.value} as variable with plan,
          ↪  collect(variable) as variables return id(plan) as
          ↪  id, plan.name as name, plan.period as period,
          ↪  plan.oneTimePayment as oneTimePayment,
          ↪  plan.regularPayment as regularPayment, plan.custom
          ↪  as custom, variables",
          { productId }
        )
```

Ukázka kódu 8: Práce s Neo4j V TypeScriptu

5.8.2 Objektově relační mapování

Mikroslužby využívající relační databázi PostgreSQL (Orders, Inquiries, Invoices a Licenses) k ní přistupují pomocí ORM (objektově-relačního mapování). ORM je technika mapování mezi objektovým a relačním modelem. Při použití ORM vývojáři nemanipulují s databází pomocí jazyka SQL, ale objektového modelu, který reflektuje data v databázi. [48]

Populárním ORM nástrojem pro Node.js aplikace napsané v TypeScriptu je Prisma, která na základě schématu v proprietární syntaxi generuje jak objektový model, tak databázové migrace. Kvůli specifickému způsobu generování objektového modelu jako balíčku v adresáři závislostí `node_modules`, který je v monorepu sdílený pro všechny mikroslužby, nelze Prismu použít (v aktuální verzi v době realizace práce) za předpokladu, že každá mikroslužba má svůj vlastní nezávislý datový model.

Po analýze dostupných alternativ jsem se rozhodl použít knihovnu TypeORM. Důvody pro toto rozhodnutí jsou velmi dobrá podpora TypeScriptu

a přehledná definice modelu využívající anotace. Příklad definice třídy objektového modelu je na ukázce kódu 9.

```
@Entity()
export class Invoicing {
  @PrimaryGeneratedColumn()
  id!: number;

  @Column()
  orderId: number;

  @Column()
  oneTimePayment: number;

  @Column()
  regularPayment: number;

  @Column()
  period: number;

  @ManyToOne(() => BillingInfo, (billingInfo: BillingInfo) =>
  ↪  billingInfo.invoices, { eager: true })
  billingInfo: BillingInfo;
}
```

Ukázka kódu 9: Příklad definice třídy objektového modelu

Pro manipulaci s objektovým modelem podporuje TypeORM dva přístupy: *Active Record* a *Data Mapper*. První zmíněný je používaný v původní verzi aplikace a vyznačuje se umístěním metod pro dotazování a manipulaci s entitou (například hledání záznamu podle ID, aktualizace hodnot, smazání záznamu) do třídy definující entitu v objektovém modelu. [79]

Mikroslužby jsem implementoval druhým zmíněným přístupem. Ten ve třídách objektového modelu nechává pouze datové atributy a veškerou logiku pro získání a úpravu dat vyčleňuje do samostatných tříd (umístěných v adresářích `data-sources`). *Data Mapper* lépe odpovídá principu jedné zodpovědnosti třídy. [79]

5.9 Sestavení aplikace

Nefunkční požadavek č. 8 říká, že mikroslužby musí být nasazeny v kontejnerech. Součástí implementace tedy bylo sestavení obrazů kontejnerů pomocí nástroje Docker.

Jak jsem již popsal, v podadresáři každé mikroslužby v monorepu je soubor zvaný Dockerfile, který slouží jako předpis pro sestavení obrazu kontejneru. Příklad Dockerfilu pro mikroslužby Invoices je na ukázce kódu 10.

```
# Stage 1
FROM node:18-alpine
LABEL stage=build
WORKDIR /app

COPY ./ /app/

RUN --mount=type=secret,id=npmrc,target=/app/.npmrc yarn install
RUN yarn build

# Stage 2
FROM node:18-alpine
WORKDIR /app

COPY --from=0 /app/dist/ ./
COPY --from=0 /app/package.json ./

RUN --mount=type=secret,id=npmrc,target=/app/.npmrc  yarn
↪  install --prod

EXPOSE 3000

CMD ["node", "./index.js"]
```

Ukázka kódu 10: Dockerfile pro mikroslužbu Invoices

Kontejnery s mikroslužbami by měly být co nejmenší. Z toho důvodu jsem použil tzv. multi-stage build, tedy sestavení ve více fázích.

První fáze sestavení začíná načtení obrazu `node:18-alpine`, který bude sloužit jako základ kontejneru. Jedná se o oficiální obraz Node.js verze 18 založený na Alpine Linuxu, který se vyznačuje svou malou velikostí.

Následně jsou zkopírovány všechny zdrojové soubory z lokálního prostředí a pomocí speciálního mechanismu pro vkládání souborů s tajnými informaci vložen soubor `npmrc` obsahující údaje pro připojení k soukromému registru knihoven. V dalším kroku jsou nainstalovány závislosti aplikace (včetně těch pro vývojové prostředí) a aplikace je sestavena.

Vývojové závislosti (například TypeScript překladač) byly nezbytné pro sestavení aplikace, ale pro běh již potřeba nejsou a pouze zvětšují velikost kontejneru. Sestavení obrazu tedy přechází do druhé fáze, která opět začne načtením obrazu `node:18-alpine`, ale pokračuje kopírováním již sestavené

	Merge request	Hlavní větev
Sestavení aplikace	Ano	Ano
Unit testování	Ano	Ano
Sestavení obrazu kontejneru / balíčku knihovny	Ne	Ano
Publikování obrazu / knihovny	Ne	Ano
Nasazení	Ne	Prozatím ne

Tabulka 5.1: Plán CI/CD

aplikace z předešlé fáze. Následně jsou nainstalovány produkční verze závislostí a vystaven port kontejneru. Při spuštění je volán příkaz zapsaný na posledním řádku Dockerfilu, který spustí Node.js aplikaci.

Díky multi-stage sestavení je velikost produkčního obrazu kontejneru většiny mikroslužeb menší než 100 MB.

5.10 CI/CD

Jak jsem popsal v teoretické části práce, neodmyslitelnou částí realizace nativní cloudové aplikace je automatizace sestavení, testovaní, vydání a případně nasazení. Jedná se o činnosti souhrnně označené jako *průběžná integrace* a *průběžné dodávky* (CI/CD).

Společnost Jagu využívá pro CI/CD nástroje integrované v GitLabu a License manager nebude výjimkou. V rámci CI/CD se bude provádět sestavení aplikace, její testování, sestavení obrazu kontejneru (nebo balíčku knihovny) a jeho publikování.

Nejprve jsem vytvořil plán, za jakých podmínek se budou jaké operace provádět. Plán jsem definoval pro dvě situace: merge request (požadavek na přijetí nových změn) a změny v hlavní větvi (po přijetí merge requestu). Plány jsou zaznačené v tabulce 5.1.

V obou případech se provádí sestavení aplikace a unit testování. V hlavní větvi se navíc provede sestavení obrazu kontejneru (v případě mikroslužeb) nebo balíčku knihovny a jejich publikování do GitLab registrů. Průběžné dodávky mohou být povýšeny na průběžné nasazení přidáním kroku automatického nasazení aplikace. Konfigurační soubory pro nasazení nové verze jsem připravil (viz 5.11), ale než bude realizován frontend, nedává příliš smysl aplikaci nasazovat do produkčního prostředí.

Kroky CI/CD jsou zapsány v souboru `.gitlab-ci.yml`, který je načten a vykonán GitLabem při změnách v merge requestu nebo hlavní větvi. Výzvou CI/CD v kontextu monorepa bylo spouštění jen relevantních úkolů. V monorepu je 6 mikroslužeb a 6 knihoven. Pokud by při každém změně měly být pro-

vedeny všechny výše popsané úkoly pro každou mikroslužby, CI/CD proces by trval velmi dlouho a výsledkem by bylo publikování nových verzí mikroslužeb, které by se nelišily od předcházejících. Problém jsem vyřešil přidáním pravidel do `.gitlab-ci.yml`. Ty testují, zda se kód mikroslužby nebo knihovny změnil. V MR se změna kódu testuje proti aktuálnímu stavu hlavní větve, v hlavním větvi oproti předcházejícímu commitu.

Výhodou současného použití GitLab CI/CD a GitLab registrů pro kontejnery a knihovny je automatické vložení přístupových tokenů k registrům do proměnných prostředí CI/CD (tokeny jsou nutné při stažení knihoven z privátního registru pro sestavení aplikace a obrazů a následně pro publikování obrazů a knihoven). Nemusel jsem tak řešit problém jejich bezpečného uložení a použití v procesu CI/CD.

5.11 Nasazení aplikace

Nasazení aplikace může být automatizované v rámci CI/CD nebo manuální. Liší se však jen aktérem, který operaci provede a následující popis je platný pro obě varianty. V teoretické části jsem představil nástroj Kubernetes pro orchestraci kontejnerů a v analýze jeho použití specifikoval jako jeden z nefunkčních požadavků.

Pro použití aplikace bude nutné do Kubernetes clusteru nasadit všechny mikroslužby (Gateway, Inquiries, Invoices, Licenses, Orders, Products), databáze PostgreSQL, Neo4j a message broker RabbitMQ. Keycloak nebude součástí standardního nasazení, protože předpokládám použití existující instance v organizaci.

Nejprve stručně popíšu princip fungování Kubernetes clusteru. Schéma komponent je na obrázku 5.5.

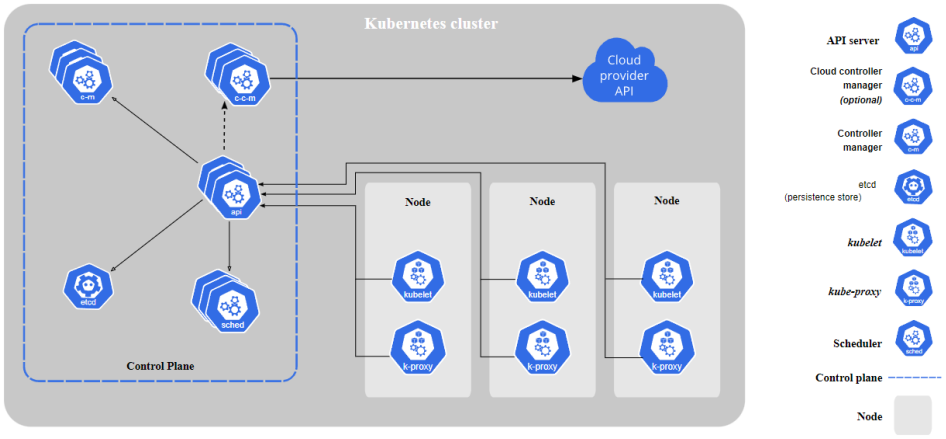
Control Plane je soubor komponent, které řídí provoz clusteru. Node je pracovní stroj (virtuální nebo fyzický počítač). Cluster musí obsahovat alespoň jeden node. Komponentě aplikace se říká Pod a je spuštěna na některém nodu.

Součástí nodu je kubelet – agent, který zajišťuje, že kontejnery v pořádku běží, a kube-proxy, které spravuje síťovou komunikaci nodu.

Control Plane obsahuje například komponentu vystavující API pro správu clusteru, plánovač, který přiděluje prostředky nebo key-value úložiště pro data a konfigurace clusteru.

Pro nasazení License manageru jsem zvolil deklarativní přístup, tedy v sadě konfiguračních souborů je popsáno, jaký má být výsledek nasazení a Kubernetes zajistí realizaci. Pro konfiguraci nasazení jsem vytvořil adresář `k8s` v kořenovém adresáři monorepa. Konfigurace jsou zapsány v syntaxi YAML. Pro nasazení aplikace License manager používám tyto typy konfigurací:

- `ConfigMap` – úložiště typu klíč-hodnota. Hodnotami mohou být i binární data. Používám ji například pro uložení veřejného certifikátu poskytovatele identity.



Obrázek 5.5: Schéma komponent Kubernetes clusteru

- `Secret` – úložiště pro malé množství citlivých dat. Využívám například pro uložení přístupových údajů k databázím a Fakturoidu.
- `Deployment` – specifikace nasazení kontejneru s mikroslužbou. Obsahuje například identifikátor obrazu kontejneru, nastavení proměnných prostředí, vystavení portů, omezení výpočetních zdrojů, počet replik nebo připojení ConfigMapy jako souboru dostupného z kontejneru.
- `Service` – slouží k vystavení mikroslužby (běžící v jednom či více podech) pro síťová volání v rámci clusteru, případně i pocházející mimo cluster (pro vývojové účely). [80]

Konfigurační soubory s citlivými údaji nejsou verzovány. V repozitáři jsou pouze jejich šablony (přípona `.example`), do kterých je nutné doplnit údaje. V prostředí s nakonfigurovaným přístupem ke Kuberneetes clusteru je nasazení provedeno příkazem `kubectl apply -f k8s --recursive -n lm`, který aplikuje všechny soubory v adresáři `k8S` do separátního jmenného prostoru clusteru `lm`.

5.12 Síť služeb

Po výše popsaném nasazení jsou mikroslužby aktivní, vzájemně komunikují a jsou připravené vyřizovat uživatelské požadavky. Pro produkční prostředí se ale aplikace stále nehodí. Důvodem je například, že komunikace mezi mikroslužbami není zabezpečená. K dispozici není ani žádný nástroj pro monitorování aplikace. Zdánlivě se jedná o nahodilé a nesouvisející problémy, spojuje je ale řešení, kterým je nástroj Istio.

Istio je síť služeb s otevřeným zdrojovým kódem, která přidává vrstvu do stávající distribuované aplikace a zjišťuje jednotný způsob zabezpečení, propojení a monitorování služeb. [81] Funkcí, které může Istio zajišťovat, je velké množství. Pro projekt License manager využiji pouze dvě: šifrování komunikace mikroslužeb a monitoring distribuované aplikace. Pro použití Istia jsem se rozhodl na samém počátku práce, proto jsem mu mikroslužby přizpůsobil – respektive neimplementoval jsem funkce, které jsem věděl, že zajistí Istio. Výhoda Istia spočívá právě v tom, že extrahuje funkce související s provozem distribuovaného systému z mikroslužeb samotných.

Velmi zjednodušeně, po instalaci Istia do clusteru a jeho povolení v daném jmenném prostoru (např. `lm`) vloží Istio ke každému zdroji typu `Service` Envoy proxy (proxy s otevřeným zdrojovým kóde vytvořená speciálně pro použití v nativních cloudových aplikacích [82]). Veškerá komunikace s danou službou následně probíhá skrze tuto proxy.

V clusteru je nasazen soubor komponent Istio control plane, který zajišťuje konfiguraci zmíněných proxy a zároveň od nich sbírá metriky o provozu. Součástí control plane je například certifikační autorita, která slouží k ověřování certifikátů služeb při vytváření mTLS spojení mezi mikroslužbami (viz 2.1.4).

Po instalaci Istia je díky toku veškerého provozu přes zmíněné proxy automaticky zabezpečena interní komunikace mezi mikroslužbami uvnitř clusteru pomocí mTLS.

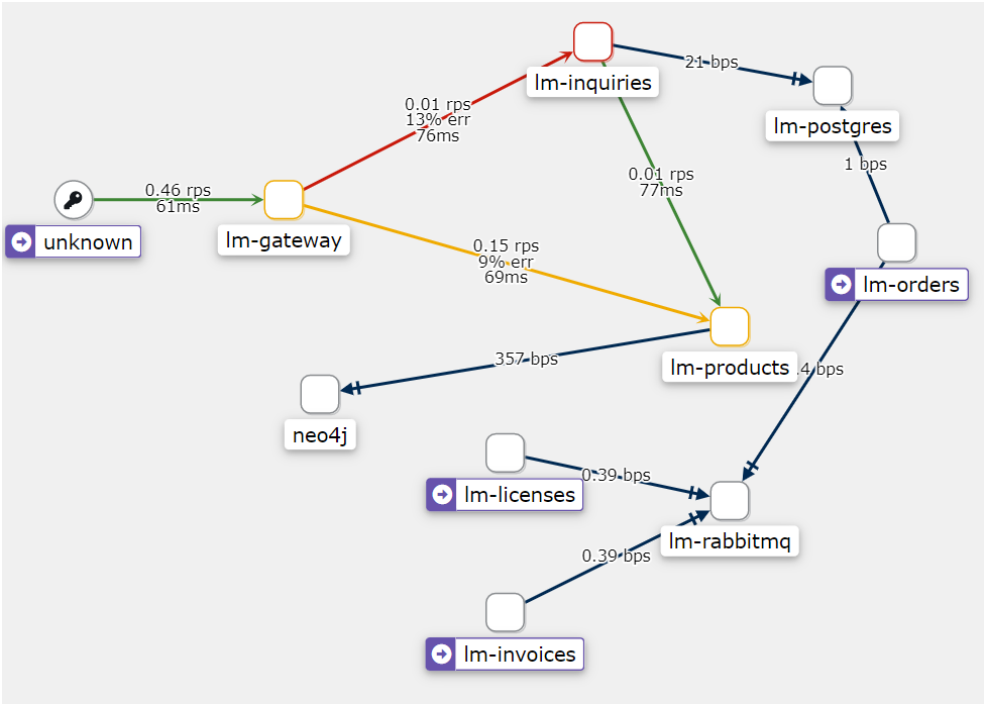
5.12.1 Monitorování aplikace

Monitorování nativní cloudové aplikace je nezbytné, protože věcí, které se mohou pokazit je mnoho (výpadek spojení mezi mikroslužbami, špatné použití API, nedostupnost mikroslužby, ztráta připojení k databázi, atd.). Jak jsem již zmínil, Envoy proxy zasílá řídící jednotce Istia metriky o provozu. Do Kubernetes clusteru je možné následně nasadit aplikace, které budou umět tato data číst a vhodně zobrazit.

Výčet dostupných rozšíření Istia je v jeho dokumentaci. Pro monitoring License manageru jsem zvolil aplikaci Kiali. Jedná se o konzoli pro monitorování a konfiguraci sítě služeb. Slouží k vizualizaci struktury sítě služeb a sledování toku provozu. [83]

Na obrázku 5.6 je snímek obrazovky z aplikace Kiali s grafem služeb. Na grafu je znázorněna komunikace mezi mikroslužbami a už na první pohled vidět stav systému – například oranžovou a červenou barvou jsou zvýrazněny cesty, na kterých je vyšší výskyt chyb. Nad spojeními je také vypsána frekvence požadavků a průměrné doba odpovědi. Modře jsou zvýrazněny datové streamy a množství dat, které jimi protéká.

Na grafu lze zobrazit celou řadu dalších metrik týkající se doby zpracování požadavků, jejich množství a chybovosti. Každou službu lze také zobrazit v detailu a zkoumat podrobněji příchozí a odchozí komunikaci (původ, cíl, typ, zabezpečení, atd.).



Obrázek 5.6: Graf sítě služeb v aplikaci Kiali

Kiali jsem využíval už při vývoji, kdy mi pomohlo odhalit nefunkční spojení mezi mikroslužbami. Také jsem ho použil při optimalizaci výkonu aplikace – detekovalo úzká hrdla systému při vysoké zátěži.

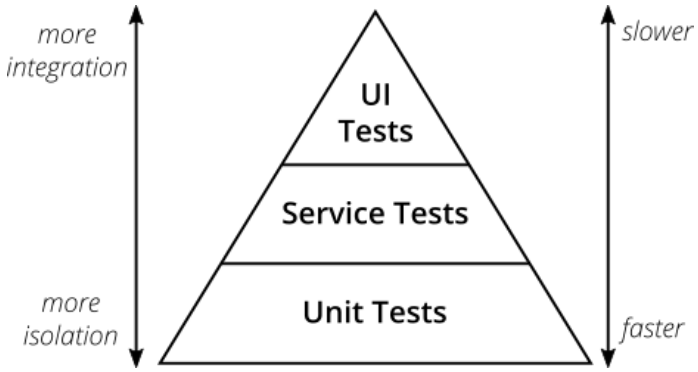
Kapitola 6

Testování

Testování aplikace je nezbytnou součástí cyklu vývoje softwaru. Pro nativní cloudové aplikace to platí dvojnásob, protože vývojový cyklus je oproti tradiční architektuře rychlejší. Nové verze nativních cloudových aplikací jsou malé, ale vydávané často. Manuální testování tedy nepřichází v úvahu. V této kapitole se budu věnovat automatizovaným testům ověřujících správnou funkčnost aplikace. V závěru kapitoly provedu testování výkonu.

Existuje celá řada typů testů s různým zaměřením, účelem a dobou provádění. Uznávanou základní klasifikací testů je Cohnova pyramida testů. Ta rozděluje testy podle četnosti, míry izolace testované části a rychlosti provedení do třech vrstev: jednotkové testy, testy služeb a testy uživatelského rozhraní (zkr. UI). [84] Pyramida je na obrázku 6.1.

S postupem do vyšších pater jsou testy zaměřené na komplexnější a více integrovanou část aplikace. S rostoucí komplexitou narůstá i čas potřebný k provedení testů. Počet testů by měl narůstat se sestupem od vrcholu pyramidy (UI testy) k základům (jednotkové testy). [84]



Obrázek 6.1: Cohnova pyramida testů [84]

6.1 Jednotkové testy

Jednotkové testy jsou v nejnižším patře pyramidy. Testují malé, izolované části kódu (obvykle funkce). Jejich počet by měl výrazně převyšovat počty ostatních testů. Jednotkové testy poskytují určitou míru jistoty, že při průběžném agilním vývoji nevznikají další chyby. Tato míra jistoty je přímo úměrná pokrytí kódu testy a jejich kvalitě (důsledné ověřování výsledků volání testovaných funkcí, testování v krajních případech).

Jednotkové testy mohou být *solitérní* (testovaná funkce je zcela izolované od ostatních) nebo *sociální* (testovaná funkce může komunikovat se spolupracujícími funkcemi). Při implementaci jednotkových testů jsem šel střední cestou – testované funkce mohly používat pomocné funkce (např. zpracování chyb, textových řetězců, apod.), ale závislosti obsahující obchodní logiku nebo integraci s jiným subjektem (databází, jinou mikroslužbou) jsem nahradil tzv. mockem. [85]

Mock je kód, který v testovacím prostředí nahrazuje část produkčního kódu, (například připojení k databázi, message brokeru či jiné mikroslužbě). [86] Mock funkce obsahuje implementaci simulující chování produkční funkce nebo vrací předem definovanou hodnotu. Zároveň umožňuje kontrolovat, že byla volána se správnými parametry. Při implementaci aplikace jsem zohlednil testovatelnost kódu a striktně dodržoval vzor *Dependency injection* (vkládání závislostí), kdy si subjekt nevytváří vlastní instance závislostí, ale přijímá je prostřednictvím konstruktoru při svém vzniku. [87] Při testování jsem mohl takovému konstruktoru předat takovému konstruktoru mock (jako na ukázce kódu 11).

```
// Mock
const channel = {
  sendToQueue: jest.fn()
};

// Testovaný objekt
const deploymentEventsPublisher = new
↪  DeploymentEventsPublisher(channel as unknown as Channel);
```

Ukázka kódu 11: Vložení mocku do testovaného objektu

V praxi je často kladen požadavek na minimální pokrytí kódu testy (například klesne-li pokrytí pod 90 %, nelze přijmout nové změny). Protože implementace automatizovaných testů je časově náročná, požadavek na vysoké pokrytí přináší vysoké náklady. Příliš vysoké pokrytí testy navíc může komplikovat provádění změn. Jednotkové testy by měly testovat pouze veřejné funkce, které navíc jsou netriviální (testovat například gettery a settery je zbytečné). [85]

Jednotkové testy jsem implementoval pro kód obsažený v podadresářích `handlers` (zpracování požadavků a událostí), `services` (komplexní byznysová logika) a `events` (publikování asynchronních požadavků). Naopak jsem vynechal podadresáře `data-sources` (manipulace s databází), kde by jednotkové testy měly minimální přidanou hodnotu (jednalo by se čistě o test volání mocků bez přidané logiky). Pro test práce s databází jsou vhodnější testy integrační.

Jednotkové testy jsem implementoval s využití testovacího frameworku Jest od Meta Open Source. Ten obsahuje všechny nástroje, které jsem pro testování potřeboval (paralelní spuštění testů, vytváření mocků, sledování pokrytí kódu testy, generování reportů, detekce místa vzniku chyby).

Soubory obsahující jednotkové testy jsou umístěny vždy ve stejném adresáři, jako testovaný soubor, mají stejný název, pouze jinou příponu (jednotkové testy k souboru `xyz.ts` jsou v souboru `xyz.spec.ts`). Kromě testování mají tyto soubory sekundární dokumentační funkci. Názvy testů popisují funkčnost v určité situaci a tělo testu potom definuje očekávaný výsledek pro daný vstup. Příklad testu je na ukázce kódu 12. Testy tedy slouží také jako vývojářská dokumentace popisující očekávané chování.

```
// Jednotkový test
it("should throw http code 401, when user info is missing", ()
↪  => {
    expect(() => {
      // Volání testované funkce
      auth(null, Roles.ORDERS_ADMIN);
    }).toThrow( // Kontrola vyhození výjimky
      new GraphQLError("Missing, expired or invalid token", {
        extensions: {
          code: "UNAUTHENTICATED",
          http: { status: 401 }
        }
      })
    );
  });
```

Ukázka kódu 12: Ukázka jednotkového testu

Jak jsem popsal, není nutné (ani žádoucí) dbát na 100% pokrytí všech řádků kódu testy. Testy jsem implementoval pro soubory ve vybraných adresářích. V těchto adresářích jsem už ale pokrytí kódu testy sledoval. Ve všech případech překračuje pokrytí 90 % řádků souborů obsažených ve zmíněných adresářích. Jednotkových testů je celkem 123.

Testy jsou spustitelné příkazem `yarn test`. Protože jednotkové testy jsou spouštěné taktéž v rámci CI, přidal jsem navíc příkaz `yarn test:ci`, který

Mikroslužba	Pokrytí řádků (%)
Products	93,02
Inquiries	97,56
Invoices	98,38
Licenses	92,12
Orders	97,5

Tabulka 6.1: Pokrytí řádků kódu jednotkovými testy

spustí Jest v režimu optimalizovaném pro CI a zároveň vygeneruje report. GitLab jsem nakonfiguroval, aby tento report načítal a výsledky testů přehledně zobrazoval v grafickém rozhraní. V případě chyby ji není nutné hledat v dlouhém logu, ale je viditelná v panelu CI/CD.

6.2 Integrační testy

Druhým patrem v pyramidě testů jsou testy služby, které Cohn popisuje jako test aplikace bez UI. [84] Za služby v kontextu architektury mikroslužeb je možné považovat celý distribuovaný systém. Do tohoto patra lze zařadit integrační testy, kterými navazuji na realizaci jednotkových testů.

Integrační test testuje aplikaci jako celek, včetně všech podpůrných služeb a v prostředí, které je shodné jako produkční. Obecně kroky integračního testu jsou následující: nasazení aplikace včetně podpůrných služeb, vložení testovacích dat, volání API aplikace a kontrola odpovědí. [85]

Integrační testování se provádí po přijetí změn do hlavní větve, před nasazením aplikace do produkčního / předprodukčního prostředí. Jak jsem uvedl v 5.10, v současné chvíli končí CI/CD publikováním knihoven a obrazů kontejnerů do registrů. Pro automatizované nasazení jsem připravil Kubernetes definice nasazení, ale než bude připraven frontend, není nutné vynakládat zdroje na provoz backendu.

Jelikož není aplikace nasazována v rámci CI/CD, nebude zde prováděno ani integrační testování. Automatizované integrační testování jsem ale přesto připravil, protože testuje i použití databáze, které je vynecháno v testech jednotkových. Integrační testy mohou být spuštěny na lokálním prostředí a jsou využitelné také k naplnění aplikace testovacími daty.

Integrační testování je realizováno oproti API bráně, tedy požadavky jsou v GraphQL. Pro implementaci integračního testování jsem se rozhodl použít aplikaci Postman (nástroj pro vytváření, testování, ladění a monitorování API). [88]

Postman jsem používal pro testování API po celou dobu vývoje kvůli vestavěné podpoře GraphQL a autorizace (Postman získá `id_token`, automaticky prodlužuje jeho platnost a vkládá ho do hlaviček požadavků). Rozšíření o kontrolu odpovědí bylo přirozeným krokem.

Výhoda Postmanu spočívá v grafickém uživatelském rozhraní zrychlujícím proces vytváření integračních testů, vedení statistik o integračních testech a sdílení pracovního prostoru v reálném čase s týmem. Postman lze integrovat do Gitlab CI/CD, vznikne-li v budoucnu potřeba spouštět integrační testy automaticky. I případná migrace integračních testů do jiného technologie (Jest, Cypress) by byla jen mechanickou činností.

Scénář integračního testu pokrývá proces konfigurace nabídky produktů (vytvoření produktu, jeho konfiguračního formuláře a plánů), poptávky (vytvoření poptávky, vytvoření nabídky, přijetí nabídky), objednávky (založení objednávky, zákaznická konfigurace produktu, zadání fakturačních údajů, potvrzení objednávky), fakturace a vystavení licence. Celkem je uskutečněno 44 požadavků.

Postman kolekce s testy je součástí monorepa. Na ukázce kódu 13 je kontrola správnosti odpovědi na požadavek vytvoření cenového plánu.

```
pm.test("Response", function () {
    var jsonData = pm.response.json();
    pm.expect(jsonData).to.eql({
    "data": {
        "createPlan": {
            "id": "1",
            "name": "2 pobocky (rocne)",
            "oneTimePayment": 1500,
            "period": 12,
            "regularPayment": 900,
            "custom": false
        }
    }
});
});
```

Ukázka kódu 13: Ověření správnosti odpovědi v integračním testu

Posledním patrem pyramidy testů je UI testování. Zástupcem je E2E testování (*End to End*), ve kterém je aplikace testována od uživatelského rozhraní, přes backend až po databázi. Testovací program ovládá aplikaci simulací činnosti reálného uživatele v grafickém uživatelském rozhraní. Uživatelské rozhraní je v případě E2E napojeno na backend běžící v prostředí, které je stejně jako v případě integračních testů totožné s produkčním prostředím. UI testování jsem nerealizoval, neboť uživatelské rozhraní nebylo součástí této práce. Bude-li ale vývoj nové verze License manageru pokračovat, E2E doplní současné jednotkové a integrační testy.

6.3 Testování výkonu

Jedním z cílů přechodu na architekturu mikroslužeb je zvýšení výkonu aplikace, tedy zkrácení doby zpracování požadavku a zvýšení počtu souběžných požadavků. Abych ověřil, že nová řešení dosahuje dobrých výsledků, provedl jsem testování výkonu.

Výsledky testování výkonu jsou zcela závislé na prostředí, ve kterém je test proveden. Aby informace získaná testováním byla přínosná, je nutné nejprve podrobně popsat hardware a software zařízení, na kterém byl zátěžový test spuštěn. Výsledky jsou platné pouze pro toto prostředí a přináší možnost administrátorům odhadnout, zda bude v produkčním prostředí nutné využít server více, či méně výkonný. Testy byly prováděny na aplikaci nasazené v Kubernetes. Využil jsem Kubernetes cluster dostupný v aplikaci Docker Desktop, tedy cluster s jedním nodem. Aplikace Docker Desktop běžela v operačním systému Windows 11 Pro s použitím Windows Subsystem for Linux (verze 2). Hardware se skládal z 16 jádrového procesoru AMD Ryzen 9 7950X (32 vláken, 4,5-5,7 GHz) a 64 GB RAM (typ DDR5, 5200 MHz).

Aplikace byla nasazená v Kubernetes včetně sítě služeb Istio. Mikroslužby v produkčních clusterech mají obvykle nastavené maximální využití hardwarových prostředků, aby nedošlo k negativnímu ovlivnění jiných aplikací běžících v clusteru. Pro simulaci produkčního prostředí jsem omezil využití prostředků i mikroslužbám při testu (Gateway: vytížení maximálně 1,5 jádra a 2000 MiB paměti, Products: 0,5 jádra a 1000 MiB paměti).

Pro provedení zátěžového testu jsem zvolil software k6 of Grafana Labs. Testovací scénáře pro k6 jsou zapsané v JavaScriptu a z důvodu lepší výkonosti následně kompilovány do nativního strojového kódu. Díky tomu se k6 vyznačuje schopností spustit velké množství virtuálních uživatelů. [89]

Princip zátěžového testu spočívá v spuštění definovaného počtu virtuálních uživatelů, kteří současně provádí stále dokola (do vypršení časového limitu testu) poskytnutý testový scénář.

V testu jsem se zaměřil na API poskytující data o nabídce produktů pro budoucí hlavní stránku. Lze očekávat, že právě hlavní stránka bude nejnavštěvovanější. Další funkce API souvisí již s realizací objednávek a pokud vezmu průměrný konverzní poměr (počet uskutečněných nákupů ku počtu návštěvníků) pro odvětví prodeje B2B služeb (prodej služeb firmám) 2,7 %, tak lze předpokládat, že výkon zbytku systému nebude představovat problém. [90]

6.3.1 Testovací scénář

Virtuální uživatelé budou stále dokola zasílat GraphQL dotaz, který je na ukázce kódu 14. Dotaz simuluje načtení dat pro domovskou stránku (seznam produktů a jejich cenových plánů). V databázi je uložen jeden produkt s jedním cenovým plánem. GraphQL server nejprve načte seznam všech produktů

a potom pro každý načte seznam plánů. Pro testovací data tedy jeden testovací GraphQL dotaz znamená dvě volání mikroslužby Products.

Test trval vždy deset sekund a spustil jsem ho postupně s 10, 100, 200 a 1000 virtuálními uživateli. Abych ověřil odolnost systému proti prudkému nárůstu uživatelů, virtuální uživatelé nebyli spuštěni pozvolně, ale všichni najednou (díky výkonému hardwaru i 1000 uživatelů bylo spuštěno prakticky okamžitě).

Přestože zátěžový test běžel na poměrně výkonném stroji, velkou část prostředků využila samotná k6. Během testu jsem monitoroval vytížení hardwaru, které ale v žádném testu nepřekročilo 60 % CPU a 75 % paměti, tedy výsledky nebyly negativně ovlivněny nedostatečným výkonem.

```
query Products {
  products {
    id
    name
    plans {
      name
      custom
      oneTimePayment
      period
      regularPayment
    }
  }
}
```

Ukázka kódu 14: GraphQL dotaz pro zátěžový test

6.3.2 Měření výkonu aplikace bez škálování

První měření jsem provedl na systému, kdy každá mikroslužba měla spuštěnou právě jednu instanci. Z naměřených hodnot jsem se zaměřil na průměrný počet zpracovaných požadavků za 1 sekundu, průměrnou a maximální dobu odpovědi. Výsledky jsou v tabulce 6.2.

Počet uživatelů	10	100	200	1000
Průměrný počet požadavků za 1 s	762	919	674	760
Průměrná doba odpovědi (s)	0,013	0,108	0,295	1,28
Maximální doba odpovědi (s)	0,075	0,2	0,513	3,41

Tabulka 6.2: Výsledky měření výkonu aplikace bez škálování

Výsledky měření říkají, že množství zpracovaných požadavků za 1 sekundu prakticky není ovlivněno množstvím uživatelů současně provádějících dotazy. Pozitivní odchylka je v případě 100 uživatelů, která je pravděpodobně způsobena vnějšími vlivy operačního systému.

Dle očekávání průměrná i maximální doba odpovědi roste s počtem uživatelů (pokud aplikace zpracovává konstantní množství požadavků za 1 sekundu, tak při vyšší zátěži trvá zpracování požadavku déle).

Stále zatížení 1000 současnými požadavky lze z obchodního hlediska považovat za extrémní a přesto je průměrná doba odpovědi 1,28 sekundy, což lze stále tolerovat (s přihlédnutím k faktu, že v testovacím požadavku jsou všechna data, která frontend bude načítat pro zobrazení hlavní stránky).

6.3.3 Měření výkonu s aplikovaným horizontálním škálováním

Aplikace dosáhla solidních výkonů už v základním stavu. Jednou z hlavních motivací přechodu na nativní cloudovou architekturu je ale možnost horizontálního škálování. Při měření neškálované aplikace jsem analyzoval zatížení distribuované aplikace pomocí monitorovacího nástroje Kiali (viz 5.12.1). Z grafu bylo vidět, že úzkým hrdlem byla mikroslužba Products. Proto jsem přidal její druhou instanci a provedl stejné měření znovu. Výsledky jsou v tabulce 6.3.

Počet uživatelů	10	100	200	1000
Průměrný počet požadavků za 1 s	1160	1292	1338	1310
Průměrná doba odpovědi (s)	0,0086	0,077	0,148	0,751
Maximální doba odpovědi (s)	0,069	0,212	1,08	2,72

Tabulka 6.3: Výsledky měření výkonu aplikace s horizontálním škálováním

Zde jsou opět vidět konzistentní výsledky napříč různou zátěží. Počet zpracovaných požadavků se zvýšil v případě 200 současných uživatelů téměř o 100 %. Stejnou měrou se zkrátila doba zpracování požadavků. Při trvalé zatížení 1000 současnými požadavky je průměrná doba odpovědi 0,7 vteřiny. Při stále vysokém zatížení 100 uživateli je pouze 77 ms.

Měření jsem provedl ještě se třemi instancemi mikroslužby Products. Zde již ale k významnému nárůstu nedošlo, protože jsem narazil na limity databáze. V případě, že by v budoucnu bylo nutné obsloužit ještě více požadavků, Neo4j databázi je možné také horizontálně škálovat (škálování databáze již je ale mimo zaměření této práce).

Kapitola 7

Srovnání nového a původního řešení

V předcházejících kapitolách jsem navrhl a implementoval nové řešení aplikace License manager odpovídající požadavkům nativních cloudových aplikací. V této kapitole srovnám nové a původní řešení, jejich vlastnosti výhody i nevýhody.

Architektuře a technickým vlastnostem obou řešení jsem se již dostatečně zabýval v kapitolách věnovaných analýze, návrhu a implementaci. V první části této kapitoly se zaměřím na srovnání mých zkušeností s procesem vývoje obou verzí. Druhá část srovnání bude více exaktní – provedu v ní srovnání výkonosti.

7.1 Srovnání procesu vývoje

Původní i nové řešení aplikace License manager je výsledkem mé práce. V této podkapitole provedu, na základě své zkušenosti, srovnání procesu vývoje obou aplikací z pohledu náročnosti, potřebné časové dotace apod. Protože po dokončení práce budou projekt dále rozvíjet jiní vývojáři, zaměřím se zde na připravenost řešení na adaptaci nových členů vývojového týmu a budoucí vývoj.

První, co musí nový vývojář na každém projektu udělat, je nastavení vývojového prostředí. Původní, monolitická verze je obsažena v jediném repozitáři, který musí vývojáři naklonovat do lokálního prostředí. Díky použití monorepa nová verze v tomto smyslu nepřidává žádnou složitost. Původní verze je implementovaná v PHP, tudíž vývojář musí nainstalovat PHP, jeho rozšíření (například ovladač pro připojení k databázi) a správce balíčků. Pro spuštění nové verze je nutné nainstalovat Node.js (instalační balíčky jsou dostupné pro všechny rozšířené operační systémy). Node.js přichází s vestavěným správcem

balíčků, pro License manager je ale doporučený správce Yarn. Celkově považuji nastavení vývojářského prostředí nové verze jako výrazně jednodušší.

Po dokončení mé bakalářské práce začali na původní verzi pracovat studenti předmětu Softwarový týmový projekt. Dle získané zpětné vazby pro ně bylo složité aplikaci zprovoznit (viz příprava vývojového prostředí) a zorientovat se v komplexní monolitické aplikaci i přes konzultace a dostupnou vývojářskou dokumentaci. Novou verzi považuji za výrazně jednodušší na prvotní seznámení a zahájení práce díky rozdělení na mikroslužby. Ty jsou menší, úzce zaměřené a množstvím kódu výrazně menší, než komplexní monolitická původní verze. Samostatnou vývojářskou dokumentaci nové verze jsem, po konzultaci s vedoucím práce, nepsal – kód je opatřen komentáři a mikroslužby jsou natolik malé, že pro programátora se znalostí TypeScriptu bude zorientování se v nich otázkou chvíle.

Původní verze je nejen monolitická z pohledu backendu, ale obsahuje také frontend. Její API je zcela orientováno na podporu konkrétních operací uživatelského rozhraní. Obsahuje velké množství často podobných koncových bodů, ve kterých je pro nového vývojáře těžké se zorientovat. API mikroslužeb nové verze je obecnější a pochopitelnější. Pro implementaci uživatelského rozhraní je k dispozici GraphQL, které nebude svazovat ruce při návrhu a implementaci nového uživatelského rozhraní.

Na časovou efektivitu vývoje se lze koukat ve dvou rovinách – počet odpracovaných hodin a délka vývoje. Strávený čas vývojem je výrazně vyšší v případě nové verze. Několik týdnů trvalo vytvořit vysokoúrovňový návrh. Ten potom sloužil jako podklad pro detailnější návrh a realizaci jednotlivých mikroslužeb. Implementace API a síťových volání je značně časově náročnější, než implementace funkce a její volání v rámci jedné aplikace. Přesné výkazy práce nemám k dispozici pro původní ani novou verzi aplikace. Realizace nové verze byla asi o 2 měsíce delší než původní, a to i přes to, že v původní verzi byl implementován i prototyp uživatelského rozhraní. Pokud by byl projekt realizován na komerční bázi, náklady na zhotovení nativní cloudové verze by byly vyšší, než na zhotovení monolitické.

Druhou rovinou je délka vývoje, tedy čas od zadání po dokončení. Jelikož jsem na projektu pracoval sám, odpovídá délka vývoje počtu odpracovaných hodin. Vezmu-li ale hypotetickou situaci, kdy by na projektu pracovalo více vývojářů/týmů, nová verze by byla zhotovena rychleji než původní. Na původní verzi také může pracovat více vývojářů současně, ale mnoho úkolů je nutné provádět sekvenčně. Mikroslužby nové verze mají závazné komunikační schéma, jejich vývojové cykly jsou již ale nezávislé. Pokud bude v budoucnu ve společnosti Jagu aktivně rozvíjena nová verze, může být zodpovědnost za jednotlivé mikroslužby rozdělena mezi více vývojářů a změny prováděny rychleji.

7.2 Porovnání výkonosti řešení

V předchozí podkapitole jsem řešení zhodnotil na základě praktické zkušenosti s vývojem. Popsal jsem zlepšení v procesu adaptace nových vývojářů a srovnal rychlost vývoje. V této podkapitole srovnám řešení na základě měřitelných dat. Nejlépe měřitelnou a zároveň důležitou metrikou pro provozovatele je výkon aplikace. Navážu tedy na testování výkonu (viz 6.3), změřím výkon původní aplikace a výsledky obou řešení srovnám.

Aby srovnání výkonu bylo relevantní, musel jsem vytvořit rovné podmínky pro obě řešení. Původní řešení, až na několik výjimek, neposkytuje REST API, ale odpovědí na většinou požadavků je HTML stránka obsahující všechna data. Nelze srovnávat výkon aplikací, kde jedna vrací pouze data a druhá navíc plní HTML šablonu. Implementoval jsem proto do původní verze nový koncový bod, který vrací data odpovídající GraphQL požadavku použitého v testování výkonu nové verze.

Zatímco mikroslužby nové verze mají vestavěné web servery, PHP aplikace musí být spuštěna pomocí externího. Společnost Jagu používá na svých projektech web server Nginx. Sestavil jsem obraz kontejneru původní aplikace založený na Nginx obrazu společnosti Jagu, který používá pro provoz svých komerčních aplikací.

Není objektivní srovnávat výkon monolitické aplikace a distribuovaného systému, který může využít mnohem více hardwarových prostředků. Při testování výkonu nové verze měly zapojené mikroslužby Gateway a Products omezený přístup k CPU a paměti. Původní verzi jsem nasadil také do Kubernetes a přidělil jí hardwarové zdroje odpovídají součtu zdrojů mikroslužeb Gateway a Products. Maximální vytížení hardwaru tedy bylo stejné pro původní i nové řešení.

Nyní jsou obě řešení ve stejné startovací pozici. Všechny rozdíly mezi nimi (implementační technologie, typ databáze, cachování, apod.) jsou už předmětem provedených vylepšení.

Testovací scénář původní verze byl totožný jako při testování nové verze, pouze místo GraphQL dotazu zasílal REST požadavek typu GET na koncový bod `/api/products`. Výsledky měření jsou v tabulce 7.1.

Počet uživatelů	10	100	200	1000
Průměrný počet požadavků za 1 s	85	83	78	54
Průměrná doba odpovědi (s)	0,116	1,14	2,34	13,76
Maximální doba odpovědi (s)	0,65	2,29	3,23	19,77

Tabulka 7.1: Výsledky měření výkonu původní verze aplikace

Na první pohled jsou výsledky původní verze řádově nižší oproti verzi nové. Relativní zlepšení jsou v tabulce 7.2. Při stejném využití hardwarových

prostředků zvládne nová verze aplikace zpracovat za 1 sekundu až 14x více požadavků a doba odpovědi je až 10,75x kratší. Při provedení horizontálního škálování mikroslužby Products, které znamená jen mírně zvýšenou zátěž, je počet zpracovaných požadavků vyšší až 26x (při 1000 uživatelích).

Počet uživatelů	10	100	200	1000
Zvýšení počtu zpracovaných požadavků (x-krát)	8,96	11,07	8,64	14,07
Zkrácení doby odpovědi (x-krát)	8,92	10,56	7,93	10,75

Tabulka 7.2: Zlepšení nové verze proti původní

Při stejném vytížení hardwaru, tedy stejných provozních nákladech, je výkon nové verze řádově lepší a zároveň je ho možné jemně ladit pomocí horizontálního škálování. Z hlediska maximálního výkonu je jednoznačným vítězem nová verze.

Nová verze má na druhou stranu vyšší hardwarové nároky při minimálním provozu. Důvodem je režie spojená s provozem Kubernetes clusteru a velkého množství kontejnerů (mikroslužby, řízení a proxy servery Istia).

Při nízkém využití License manageru je tedy z hlediska provozních nákladů vhodnější původní verze, při vysokém vytížení naopak nová.

Závěr

V této práci jsem se zabýval vývojem nativních cloudových aplikací. Teorii popsanou v prvních dvou kapitolách jsem aplikoval na projekt License manager, webovou aplikaci pro prodej konfigurovatelného softwaru formou předplatného.

První verzi aplikace License manager s tradiční monolitickou architekturou jsem realizoval v rámci své bakalářské práce. Cílem této práce bylo navrhnout a implementovat nativně cloudovou verzi backendu aplikace. Tedy vytvořit aplikaci s novou architekturou, využívající novou sadu moderních technologií a postupů vývoje tak, aby budoucí rozvoj aplikace byl maximálně agilní a aplikace držela krok s narůstajícím počtem zákazníků i jejich potřeb.

Práce pokrývá kompletní životní cyklus softwaru – analýzu, návrh, implementaci, testování, nasazení a monitoring. Ve fázi analýzy jsem popsal současné řešení a definoval požadavky na nové. Nové funkční požadavky nepřibyly, naopak z důvodu výrazně vyšší náročnosti implementace byl odebrán funkční požadavek na navýšení cenového plánu.

Ve fázi návrhu jsem zpracoval novou architekturu založenou na mikroslužbách. Při dělení funkcí a dat mezi mikroslužby jsem vycházel z teorie ohraničených kontextů. Při návrhu rozdělení jsem dbal na nízkou provázanost a vysokou soudržnost. Zvýšení nezávislosti mikroslužeb jsem dosáhl také výhradním použitím asynchronních požadavků od okamžiku potvrzení objednávky, přes fakturaci a vystavení licence až po instrukci k zahájení nasazení uživatelské instance zakoupeného produktu. Mimo maximální nezávislosti mikroslužeb jsem při návrhu dbal na zjednodušení všech procesů, které se projevilo zejména ve způsobu vystavování a aktualizace stavů faktur. Výsledkem fáze návrhu byl podklad pro implementaci šesti mikroslužeb.

Novou verzi jsem implementoval zcela znovu s využitím nové sady technologií. Nová verze je implementována v jazyce TypeScript a spouštěna pomocí běhového prostředí Node.js. Pro komunikace mezi mikroslužbami jsem místo obvyklého RESTu zvolil gRPC. Data většiny mikroslužeb jsou uložena v relační databázi, výjimkou je mikroslužba Products, kde data tvoři komplexní

graf, a proto jsem využil databázi grafovou. Všechny tyto technologické změny se projevily v testování výkonu – nová verze při stejném vytížení hardwaru dokázala obsloužit až 14x více požadavků než verze současná. gRPC navíc přineslo vyšší bezpečnost, protože správnost interní komunikace mezi mikroslužbami je ověřována statickou kontrolou při překladu kódu. Pro budoucí vývoj frontendové části jsem připravil GraphQL API. To umožňuje definovat podobu odpovědí a tím optimalizovat množství přenášených dat (a tedy i snížit náklady na provoz).

Na fázi implementace navazují fáze testování a vydání. Obě tyto činnosti jsem automatizoval pomocí nástroje GitLab CI/CD. Při každé změně kódu je automaticky provedeno jednotkové testování. Po přijetí změn do hlavní větve jsou sestaveny a publikovány obrazy Docker kontejnerů. Proces vydání každé mikroslužby je zcela nezávislý na jiných mikroslužbách. Mimo jednotkových testů spouštěných během CI jsem připravil integrační testy, které ověřují funkčnost systému jako celku.

Realizoval jsem také vzorové nasazení aplikace do Kubernetes clusteru. Součástí nasazení je i nástroj pro správu sítě služeb Istio, který využívám pro zabezpečení vzájemné komunikace mikroslužeb a monitorování systému. V budoucnu pomůže při postupném nasazování nových verzí mikroslužeb.

V poslední části práce jsem porovnal stávající a nové řešení jak po stránce výkonu, tak způsobu práce na nich. Jak už jsem zmínil, nová verze dosahuje výrazně lepších výsledků. Realizace nativně cloudové verze byla časově náročnější v porovnání se současnou monolitickou. Naopak nyní, kdy je již vytvořený základ, umožňuje nová verze lepší paralelizaci práce, a tím i rychlejší dodávky nových změn. Ve všech fázích vývoje jsem měl na paměti budoucí vývoj aplikace a dbal na rozšiřitelnost aplikace a snadnější zapojení případných dalších vývojářů.

Hlavní cíl práce, implementace nativně cloudové verze backendové části aplikace License manager, i jeho dílčí části, byly splněny.

Společnost Jagu má nyní k dispozici dvě technologicky zcela odlišné verze aplikace License manager. Zatím ani jedna není spuštěna komerčně. Původní verze má za sebou již delší vývoj, zatímco nová bude pro komerční nasazení vyžadovat doplnění některých chybějících funkcí (například navýšení cenového plánu, e-mailovou komunikaci) a implementaci frontendu. Pokračovat bude zřejmě vývoj pouze jedné. Nová verze nabízí solidní základ postavený na moderních technologiích, vyšší výkon, škálovatelnost a z mého pohledu autora obou verzí i lepší rozšiřitelnost a snadnější vývoj.

Bibliografie

1. VAILSHERY, Lionel Sujay. Public cloud services annual growth rate worldwide in 2022 and 2023. In: *Statista* [online]. Statista, 2022 [cit. 2023-03-06]. Dostupné z: `https://www.statista.com/statistics/258718/market-growth-forecast-of-public-it-cloud-services-worldwide/`.
2. STACK OVERFLOW. Technology. In: *Stack Overflow Developer Survey 2022* [online]. New York: Stack Overflow, 2022 [cit. 2023-03-06]. Dostupné z: `https://survey.stackoverflow.co/2022/#section-most-popular-technologies-cloud-platforms`.
3. CLOUD NATIVE COMPUTING FOUNDATION. Charter. In: *Cloud Native Computing Foundation Policy Repo* [online]. Cloud Native Computing Foundation, 2021 [cit. 2023-03-06]. Dostupné z: `https://github.com/cncf/foundation/blob/main/charter.md`.
4. REZNIK, Pini; DOBSON, Jamie; GIENOW, Michelle. *Cloud Native Transformation: Practical Patterns for Innovation*. Sebastopol: O'Reilly Media, 2020. ISBN 978-1-492-04890-9.
5. SCHOLL, Boris; SWANSON, Trent; JAUSOVEC, Peter. *Cloud Native: Using Containers, Functions, and Data to Build Next-Generation Applications*. Sebastopol: O'Reilly Media, 2019. ISBN 978-1-492-05382-8.
6. VERACODE. Man in the Middle (MITM) Attack. In: *Veracode* [online]. Burlington, 2023 [cit. 2023-03-15]. Dostupné z: `https://www.veracode.com/security`.
7. GARTNER. Scalability. In: *Gartner Glossary* [online]. 2023 [cit. 2023-03-15]. Dostupné z: `https://www.gartner.com/en/information-technology/glossary/scalability`.
8. CLOUD NATIVE GLOSSARY AUTHORS. Scalability. In: *Cloud Native Glossary* [online]. 2022 [cit. 2023-03-15]. Dostupné z: `https://glossary.cncf.io/scalability/`.

9. NEWMAN, Sam. *Building Microservices*. Sebastopol: O'Reilly Media, 2015. ISBN 978-1-491-95035-7.
10. AMAZON WEB SERVICES. What Is SDLC (Software Development Lifecycle)? In: *Cloud Computing Concepts Hub* [online]. 2023 [cit. 2023-05-02]. Dostupné z: https://aws.amazon.com/what-is/sdlc/.
11. WIGGINS, Adam. *The Twelve-Factor App*. 2017. Dostupné také z: https://12factor.net/12factor.epub.
12. DOCKER. Use containers to Build, Share and Run your applications. In: *Docker: Accelerated, Conteinerized Application Development* [online]. 2023 [cit. 2023-03-06]. Dostupné z: https://www.docker.com/resources/what-container/.
13. LONGBOTTOM, Clive. What are containers and how do they work? In: *TechTarget: IT operations news and how-tos* [online]. 2018 [cit. 2023-03-06]. Dostupné z: https://www.techtarget.com/searchitoperations/tip/What-are-containers-and-how-do-they-work.
14. What is serverless? In: *Red Hat: Understanding cloud-native applications* [online]. Red Hat, 2022 [cit. 2023-03-06]. Dostupné z: https://www.redhat.com/en/topics/cloud-native-apps.
15. KIRVAN, Paul. synchronous/asynchronous API. In: *WhatIs: The Tech Dictionary* [online]. 2022 [cit. 2023-03-06]. Dostupné z: https://www.techtarget.com/whatis/definition/synchronous-asynchronous-API.
16. BELLEMARE, Adam. *Building Event-Driven Microservices: Leveraging Organizational Data at Scale*. Sebastopol: O'Reilly Media, 2020. ISBN 978-1-492-05789-5.
17. VETTOR, Rob; SMITH, Steve. What is Cloud Native? In: *Architecting Cloud Native .NET Applications for Azure* [online]. Redmond: Microsoft Corporation, 2022 [cit. 2023-03-06]. Dostupné z: https://learn.microsoft.com/en-us/dotnet/architecture/cloud-native/definition.
18. HUMBLE, Jez. What is Continuous Delivery? In: *Continuous Delivery* [online]. 2017 [cit. 2023-03-06]. Dostupné z: https://www.continuousdelivery.com.
19. FOWLER, Martin. *BlueGreenDeployment* [online]. 2010. [cit. 2023-03-06]. Dostupné z: https://martinfowler.com/bliki/BlueGreenDeployment.html.
20. RAMATI, Ran. How to Log a Log: Application Logging Best Practices. In: *The Logz.io Blog* [online]. LogsHero, 2017 [cit. 2023-03-06]. Dostupné z: https://logz.io/blog/logging-best-practices.

21. What is orchestration? In: *Understanding automation* [online]. Red Hat, 2019 [cit. 2023-03-06]. Dostupné z: https://www.redhat.com/en/topics/automation/what-is-orchestration.

22. What is container orchestration? In: *IBM* [online]. 2023 [cit. 2023-03-06]. Dostupné z: https://www.ibm.com/topics/container-orchestration.

23. WILSON, Bibin. 16 Best Container Orchestration Tools and Services. In: *DevOpsCube* [online]. devopscube, 2022 [cit. 2023-03-06]. Dostupné z: https://devopscube.com/docker-container-clustering-tools.

24. CASEY, Kevin. Kubernetes by the numbers, in 2020: 12 stats to see. In: *The Enterprisers Project* [online]. 2020 [cit. 2023-03-06]. Dostupné z: https://enterprisersproject.com/article/2020/6/kubernetes-statistics-2020.

25. THE KUBERNETES AUTHORS. Kubernetes Features. In: *Kubernetes* [online]. 2023 [cit. 2023-03-06]. Dostupné z: https://kubernetes.io.

26. JACOBS, Michael. What are Microservices? In: *Azure DevOps documentation* [online]. Microsoft Corporation, 2022 [cit. 2023-03-12]. Dostupné z: https://learn.microsoft.com/en-us/devops/deliver/what-are-microservices.

27. AMAZON WEB SERVICES. Microservices: Build highly available microservices to power applications of any size and scale. In: [online]. 2022 [cit. 2023-03-12]. Dostupné z: https://aws.amazon.com/microservices/.

28. IBM. What are microservices? In: *IBM* [online]. 2023 [cit. 2023-03-12]. Dostupné z: https://www.ibm.com/topics/microservices.

29. IBM CLOUD TEAM. SOA vs. Microservices: What's the Difference? In: *IBM Blog* [online]. IBM, 2021 [cit. 2023-03-12]. Dostupné z: https://www.ibm.com/cloud/blog/soa-vs-microservices.

30. IBM. What is an enterprise service bus (ESB)? In: *IBM* [online]. 2023 [cit. 2023-03-22]. Dostupné z: https://www.ibm.com/topics/esb.

31. ANIL, Nish; JAIN, Tarun; PINE, David et al. Design a DDD-oriented microservice. In: *Microsoft Learn* [online]. Microsoft Corporation, 2022 [cit. 2023-03-22]. Dostupné z: https://learn.microsoft.com/en-us/dotnet/architecture/microservices/microservice-ddd-cqrs-patterns/ddd-oriented-microservice.

32. EVANS, Eric. *Domain-Driven Design: Tackling complexity in the heart of software*. Addison-Wesley, 2004. ISBN 9780321125217.

33. MICROSOFT. Scheduler Agent Supervisor pattern. In: *Microsoft Learn* [online]. 2023 [cit. 2023-03-22]. Dostupné z: https://learn.microsoft.com/en-us/azure/architecture/patterns/scheduler-agent-supervisor.

34. AUTH0. What is Authentication? In: *Intro to IAM* [online]. 2023 [cit. 2023-03-22]. Dostupné z: https://auth0.com/intro-to-iam/what-is-authentication.

35. AUTH0. What is Authorization? In: *Intro to IAM* [online]. 2023 [cit. 2023-03-22]. Dostupné z: https://auth0.com/intro-to-iam/what-is-authorization.

36. AUTH0. Single Sign-On. In: *Auth0 docs* [online]. 2023 [cit. 2023-03-20]. Dostupné z: https://auth0.com/docs/authenticate/single-sign-on.

37. PANDEY, Prashant; NISHA, T. N. Challenges in Single Sign-On. *Journal of Physics: Conference Series*. 2021, č. 4. Dostupné z DOI: 10.1088/1742-6596/1964/4/042016.

38. GÓES, Murilo; CANEDO, Edna Dias. Authentication and Authorization in Microservices Architecture: A Systematic Literature Review. *Applied Sciences* [online]. 2022, roč. 12, č. 6, s. 3023 [cit. 2023-03-20]. Dostupné z DOI: 10.3390/app12063023.

39. STENIUS, Petteri. The differences between OpenID Connect (OIDC), OAuth 2.0 & SAML. In: *Ubisecure* [online]. 2020 [cit. 2023-03-20]. Dostupné z: https://www.ubisecure.com/education/differences-between-saml-oauth-openid-connect/.

40. SIRIWARDENA, Prabath. Mutual Authentication with TLS. In: *Advanced API Security: Securing APIs with OAuth 2.0, OpenID Connect, JWS, and JWE*. Berkeley, CA: Apress, 2014, s. 47–58. ISBN 978-1-4302-6817-8. Dostupné z DOI: 10.1007/978-1-4302-6817-8_4.

41. GRANT, Mithchell. Software as a Service (SaaS): Definition and Examples. In: *Investopedia* [online]. 2022 [cit. 2023-04-17]. Dostupné z: https://www.investopedia.com/terms/s/software-as-a-service-saas.asp.

42. HOLÝ, Viktor. *License manager - webová aplikace*. 2021. Bakalářská práce. České vysoké učení technické v Praze, Fakulta informačních technologií. Vedoucí práce Ing. Jiří Hunka.

43. W3SCHOOLS. What is JSON? In: *Web Development* [online]. 2023 [cit. 2023-04-17]. Dostupné z: https://www.w3schools.com/whatis/whatis_json.asp.

44. THE PHP GROUP. What is PHP? In: *PHP Manual* [online]. 2023 [cit. 2023-04-17]. Dostupné z: https://www.php.net/manual/en/intro-whatis.php.

45. English Dictionary. In: *Cambridge Dictionary* [online]. Cambridge University Press, 2023 [cit. 2023-04-17]. Dostupné z: https://dictionary.cambridge.org/dictionary/english/.

46. LARAVEL. Laravel. In: *Laravel: The PHP Framework for Web Artisans* [online]. 2023 [cit. 2023-04-17]. Dostupné z: https://laravel.com/.

47. HARRIS, Chandler. Microservices vs. monolithic architecture. In: *Attlassian: Software development* [online]. 2023 [cit. 2023-04-17]. Dostupné z: https://www.atlassian.com/microservices/microservices-architecture/microservices-vs-monolith.

48. CHEHAB, Guilherme. What Is an ORM? In: *Baeldung* [online]. 2023 [cit. 2023-04-17]. Dostupné z: https://www.baeldung.com/cs/object-relational-mapping.

49. LARAVEL. Database. In: *Laravel: The PHP Framework for Web Artisans* [online]. 2023 [cit. 2023-04-17]. Dostupné z: https://laravel.com/docs/10.x/database.

50. FOWLER, Martin. Active Record. In: *martinfowler* [online]. [B.r.] [cit. 2023-04-17]. Dostupné z: https://martinfowler.com/eaaCatalog/activeRecord.html.

51. CLOUDFLARE. What is the Simple Mail Transfer Protocol (SMTP)? In: *Cloudflare: Learning Center* [online]. 2023 [cit. 2023-04-17]. Dostupné z: https://www.cloudflare.com/learning/email-security/what-is-smtp/.

52. LARAVEL. Queues. In: *Laravel: The PHP Framework for Web Artisans* [online]. 2023 [cit. 2023-04-17]. Dostupné z: https://laravel.com/docs/9.x/queues#introduction.

53. ROME, Paula. What are Non Functional Requirements. In: *Perforce: Blog* [online]. 2023 [cit. 2023-04-17]. Dostupné z: https://www.perforce.com/blog/alm/what-are-non-functional-requirements-examples.

54. HEDDINGS, Anthony. What is a Cron Job, and How Do You Use Them? In: *How-To Geek* [online]. 2020 [cit. 2023-04-17]. Dostupné z: https://www.howtogeek.com/devops/what-is-a-cron-job-and-how-do-you-use-them/.

55. RED HAT. What is a webhook? In: *Red Hat: Understanding automation* [online]. 2022 [cit. 2023-04-17]. Dostupné z: https://www.redhat.com/en/topics/automation/what-is-a-webhook.

56. SPARX SYSTEMS. Conceptual Data Model. In: *Enterprise Architect User Guide* [online]. 2023 [cit. 2023-04-17]. Dostupné z: https://sparxsystems.com/enterprise_architect_user_guide/15.2/model_domains/conceptual_data_model__.html.

57. VISUAL PARADIGM. What is Sequence Diagram? In: *Visual Paradigm* [online]. 2022 [cit. 2023-04-17]. Dostupné z: https://www.visual-paradigm.com/guide/uml-unified-modeling-language/what-is-sequence-diagram/.

58. LUCID SOFTWARE. Component Diagram Tutorial. In: *Lucidchart* [online]. 2023 [cit. 2023-04-17]. Dostupné z: https://www.lucidchart.com/pages/uml-component-diagram.

59. RATHORE, Hemant. Difference Between REST API and RPC API. In: *GeeksforGeeks* [online]. 2022 [cit. 2023-04-17]. Dostupné z: https://www.geeksforgeeks.org/difference-between-rest-api-and-rpc-api/.

60. GRPC AUTHORS. Introduction to gRPC: An introduction to gRPC and protocol buffers. In: *gRPC* [online]. 2023 [cit. 2023-04-17]. Dostupné z: https://grpc.io/docs/what-is-grpc/introduction/.

61. WAGNER, Janet. Understanding the API-First Approach to Building Products. In: *Swagger* [online]. 2023 [cit. 2023-04-17]. Dostupné z: https://swagger.io/resources/articles/adopting-an-api-first-approach/.

62. THE LINUX FOUNDATION. Netflix: Case study. In: *Cloud Native Computing Foundation* [online]. 2018 [cit. 2023-04-17]. Dostupné z: https://www.cncf.io/case-studies/netflix/.

63. BASTIN, Tim. Performance comparison: REST vs gRPC vs asynchronous communication. In: *Medium* [online]. 2022 [cit. 2023-04-17]. Dostupné z: https://medium.com/l3montree-techblog/performance-comparison-rest-vs-grpc-vs-asynchronous-communication-3ad27d144a13.

64. THE GRAPHQL FOUNDATION. Introduction to GraphQL. In: *GraphQL* [online]. 2023 [cit. 2023-04-17]. Dostupné z: https://graphql.org/learn.

65. THE GRAPHQL FOUNDATION. GraphQL: A query language for your API. In: *GraphQL* [online]. 2023 [cit. 2023-04-17]. Dostupné z: https://graphql.org.

66. MOZILLA FOUNDATION. Prototype-based programming. In: *MDN Web Docs* [online]. 2023 [cit. 2023-04-17]. Dostupné z: https://developer.mozilla.org/en-US/docs/Glossary/Prototype-based_programming.

67. MOZILLA FOUNDATION. JavaScript. In: *MDN Web Docs* [online]. 2023 [cit. 2023-04-17]. Dostupné z: https://developer.mozilla.org/en-US/docs/Web/javascriptg.

68. MICROSOFT. C#. In: *.NET: Build, Test, Deploy* [online]. 2023 [cit. 2023-04-17]. Dostupné z: https://dotnet.microsoft.com/en-us/languages/csharp.

69. ATHAPATHU, Rukshani. Blocking I/O and non-blocking I/O. In: *Medium* [online]. 2017 [cit. 2023-04-17]. Dostupné z: https://medium.com/coderscorner/tale-of-client-server-and-socket-a6ef54a74763.

70. MANANDHAR, Geshan. 5 important reasons to choose Node.js for your microservices. In: *Geshan Manandhar* [online]. 2020 [cit. 2023-04-17]. Dostupné z: https://geshan.com.np/blog/2020/11/nodejs-microservices/.

71. OVI, Muhammad. Why NodeJS for Microservices. In: *Muhammad Ovi* [online]. 2021 [cit. 2023-04-17]. Dostupné z: https://muhammadovi.com/why-nodejs-for-microservices.

72. MICROSOFT. TypeScript: JavaScript with syntax for types. In: *TypeScript* [online]. 2023 [cit. 2023-04-17]. Dostupné z: https://www.typescriptlang.org/.

73. GERCHEV, Ivaylo. Yarn vs npm: Everything You Need to Know. In: *SitePoint* [online]. 2021 [cit. 2023-04-24]. Dostupné z: https://www.sitepoint.com/yarn-vs-npm/.

74. VMWARE. RabbitMQ: Messaging that just works. In: *RabbitMQ* [online]. 2023 [cit. 2023-04-17]. Dostupné z: https://www.rabbitmq.com/.

75. APOLLO GRAPH. Apollo Docs. In: *Apollo GraphQL* [online]. 2023 [cit. 2023-04-24]. Dostupné z: https://www.apollographql.com/docs.

76. KEYCLOAK. Keycloak: Open Source Identity and Access Management. In: *Keycloak* [online]. 2023 [cit. 2023-04-17]. Dostupné z: https://www.keycloak.org/.

77. OPENID. Welcome to OpenID Connect. In: *OpenID Connect* [online]. 2023 [cit. 2023-04-17]. Dostupné z: https://openid.net/connect/.

78. OKTA. Introduction to JSON Web Tokens. In: *JSON Web Tokens* [online]. 2023 [cit. 2023-04-17]. Dostupné z: https://jwt.io/introduction.

79. OKTA. Active Record vs Data Mapper. In: *TypeORM* [online]. [B.r.] [cit. 2023-04-18]. Dostupné z: https://typeorm.io/active-record-data-mapper#what-is-the-data-mapper-pattern.

80. THE KUBERNETS AUTHORS. Kubernetes: Concpets. In: *Kubernetes Documentation* [online]. 2023 [cit. 2023-04-18]. Dostupné z: https://kubernetes.io/docs/concepts/.

81. ISTIO AUTHORS. Concepts. In: *Istio* [online]. 2023 [cit. 2023-04-18]. Dostupné z: https://istio.io/latest/about/service-mesh/#concepts.

82. ENVOY PROJECT AUTHORS. Envoy Proxy. In: *Envoy Proxy* [online]. 2023 [cit. 2023-04-18]. Dostupné z: https://www.envoyproxy.io/.

83. ISTIO AUTHORS. Kiali. In: *Istio* [online]. 2023 [cit. 2023-04-18]. Dostupné z: https://istio.io/latest/docs/ops/integrations/kiali/.

84. COHN, Mike. The Forgotten Layer of the Test Automation Pyramid. In: *The Mountain Goat Software Blog* [online]. 2009 [cit. 2023-04-25]. Dostupné z: https://www.mountaingoatsoftware.com/blog/the-forgotten-layer-of-the-test-automation-pyramid.

85. FOWLER, Martin. The Practical Test Pyramid. In: *martinfowler* [online]. 2018 [cit. 2023-04-25]. Dostupné z: https://martinfowler.com/articles/practical-test-pyramid.html.

86. FOWLER, Martin. TestDouble. In: *martinfowler* [online]. 2006 [cit. 2023-04-25]. Dostupné z: https://martinfowler.com/bliki/TestDouble.html.

87. BORSKÝ, Jiří; ZIMOLKA, Jan. *Non-GoF patterns* [online]. 2022. [cit. 2023-04-25]. Dostupné z: https://gitlab.fit.cvut.cz/zimoljan/adp-media/-/raw/master/media/materials/lectures/NI-ADP%5C%202022%5C%20-%5C%2007%5C%20-%5C%20Non-GoF%5C%20patterns.pdf.

88. POSTMAN. *Postman API platform* [online]. 2023. [cit. 2023-04-25]. Dostupné z: https://www.postman.com.

89. GRAFANA LABS. k6 Documentation. In: *k6* [online]. 2023 [cit. 2023-04-18]. Dostupné z: https://k6.io/docs/.

90. HOLMES, Katie. Updated 2023: Average Conversion Rate by Industry and Marketing Source. In: *Ruler Analytics* [online]. 2023 [cit. 2023-04-18]. Dostupné z: https://www.ruleranalytics.com/blog/insight/conversion-rate-by-industry/.

Příloha A

Seznam použitých zkratek

API Aplikační programové rozhraní

APOC Awesome Procedures On Cypher

CI Průběžná integrace

CRUD Vytvoření, čtení, aktualizace a smazání

DDD Návrh řízený doménou

DRY Neopakuj se

E2E End-to-End

ERD Entitně-relační diagram

ESB Sběrnice podnikových služeb

IT Informační technologie

JS JavaScript

JSON JavaScript Object Notation

JWT JSON Web Token

LM License manager

MR Žádost o sloužení

mTLS Mutual Transport Layer Security

MVC Model-View-Controller

OIDC OpenID Connect

ORM Objektově relační mapování

PubSub Publikování-odběr

REST Representational State Transfer

RPC Vzdálené volání procedury

SaaS Software jako služba

SAML Security Assertion Markup Language

SMTP Simple Mail Transfer Protocol

SOA Servisně orientovaná architektura

SSO Jednotné přihlášení

TS TypeScript

UI Uživatelské rozhraní

UML Unified Modeling Language

XML Extensible Markup Language

Příloha B

Obsah přiloženého média

- `readme.txt` ..... stručný popis obsahu média
- `src`
  - `thesis.zip` ..... zdrojová forma práce ve formátu LaTeX
- `app` ..... zdrojové soubory aplikace
- `docs` ..... dokumentace architektury
  - `dp.eapx` . projekt s dokumentací architektury pro Enterprise Architect
  - `component-diagram.png` ..... kompletní diagram komponent