FAKULTA INFORMAČNÍCH TECHNOLOGIÍ ČVUT V PRAZE

Zadání diplomové práce

Název:	Transformace systému z monolitické architektury do architektury mikroslužeb
Student:	Bc. Štěpán Severa
Vedoucí:	Ing. Filip Ravas
Studijní program:	Informatika
Obor / specializace:	Webové a softwarové inženýrství, zaměření Softwarové inženýrství
Katedra:	Katedra softwarového inženýrství
Platnost zadání:	do konce letního semestru 2021/2022

Pokyny pro vypracování

Navrhněte a proveďte transformaci části Uniqway backend systému z monolitické architektury do architektury mikroslužeb. Při návrhu spolupracujte se studentským týmem Uniqway.

• Proveďte rešerši vhodných způsobů transformace zadaného systému.

• Analyzujte stávající monolitické řešení backendové aplikace.

• Navrhněte proces transformace systému do architektury mikroslužeb a zrealizujte jej.

• Zaměřte se přitom na mikroslužby zabývající se správou textů a lokalizací, obchodem s odměnami a tvorbou a správou rezervací.

• Doplňte implementaci o unit testy a API testy.

• Vytvořte dokumentaci přepracovaného kódu.

Elektronicky schválil/a Ing. Michal Valenta, Ph.D. dne 28. prosince 2020 v Praze.

FAKULTA
INFORMAČNÍCH
TECHNOLOGIÍ
ČVUT V PRAZE

Diplomová práce

Transformace systému z monolitické architektury do architektury mikroslužeb

Bc. Štěpán Severa

Katedra softwarového inženýrství
Vedoucí práce: Ing. Filip Ravas

6. května 2021

Poděkování

Děkuji svému vedoucímu Ing. Filipu Ravasovi za jeho ochotu, připomínky a rady při tvorbě této diplomové práce. Také děkuji Ing. Václavu Jirovskému, PhD., mentorovi za ČVUT v projektu Uniqway, za jeho dlouhodobou podporu a pomoc při práci na projektech ať školních či pracovních.

Prohlášení

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

Nakládání s předloženou prací se řídí Smlouvou o spolupráci uzavřenou v návaznosti na spolupráci mezi Českým vysokým učením technickým v Praze a společností ŠKODA AUTO a.s. a ŠKODA AUTO DigiLab s.r.o. na výzkumném projektu „CarSharing pro vysokoškolské studenty“, uveřejněné v registru smluv na adrese https://smlouvy.gov.cz/smlouva/5973503.

Jsem vázán Smlouvou o zachování mlčenlivosti, že nezpřístupním třetí osobě důvěrné informace, které jsem při své práci na Projektu získal.

Tímto má předložená práce nemůže být zveřejněna po dobu platnosti závazku mlčenlivosti, tj. do 31. května 2024.

V Praze dne 6. května 2021 . . . . . .

Odkaz na tuto práci

Severa, Štěpán. *Transformace systému z monolitické architektury do architektury mikroslužeb*. Diplomová práce. Praha: České vysoké učení technické v Praze, Fakulta informačních technologií, 2021.

Abstrakt

Tato práce se zabývá analýzou, návrhem a implementací transformace serverové aplikace systému Uniqway z monolitické architektury do architektury mikroslužeb. Výsledné mikroslužby pokrývají část funkcionality systému Uniqway. K implementaci aplikace byl použit Java framework Play. Hlavním přínosem této práce je vytvoření analýzy a návrhu přechodu potřebných pro dokončení této transformace. Tato transformace umožní jednodušší správu serverové aplikace a rychlejší zaučování nových členů týmu.

Klíčová slova mikroslužby, server, monolitická architektura, Uniqway, Java Play, transformace

Abstract

This thesis engages analysis, draft and implementation of Uniqway server application transformation from monolithic architecture to microservices architecture.Resulting microservices provides a part of Uniqway system funcionality. Java framework Play was used for implementation of the microservices. The main merit of this thesis is creation of analysis and draft needed for completing the transformation. This transformation will enable simpler managing of server application and faster onboarding of new team members.

Keywords microservices, server, monolithic architecture, Uniqway, Java Play, transformation

Obsah

Seznam obrázků

Seznam tabulek

Úvod

Sdílená ekonomika se stává čím dál populárnější a každým dnem přibývají služby, které se sdílenou ekonomikou zabývají. Hlavním znakem sdílené ekonomiky je, že lidé produkty nevlastní, ale pouze si je půjčují, když je potřebují. Takto je možné si půjčit velkou řadu věcí a to od nářadí přes koloběžky až po auta. Půjčování aut (tzv. carsharing) je v posledních letech na vzestupu a ve velkých městech se začíná objevovat spousta projektů, které takovéto služby poskytují. Hlavním důvodem rozvoje carsharingu je dle názoru autora práce zahlcení měst auty, ve kterých již není místo na auta, která většinu času stojí jen zaparkovaná před domy. Carsharing umožňuje, aby auto, které není používáno majitelem, nestálo před domem, ale bylo užíváno někým jiným. Toto přinese menší množství aut ve městech a tím částečně řeší problém velkého množství aut. Další výhodou carsharingu je relativní bezstarostnost používání vozidla. Uživatel se nemusí starat o výměnu pneumatik, jízdy do servisu, opravy poškozených nebo opotřebovaných součástek. Uživatel si pouze zarezervuje auto, použije ho, vrátí a poté zaplatí poplatek za půjčení. To je vše. Jednorázový poplatek za danou jízdu může být vyšší, než při jízdě vlastním autem, ale z dlouhodobého hlediska s ohledem na ostatní náklady na vozidle může být carsharing levnější variantou. Pro obyvatele velkých měst je to velmi dobrá alternativa k vlastnění vozidla. Jedním z takových poskytovatelů carsharingu v Praze je Uniqway. Uniqway je prvním carsharingem, který je provozován studenty. Na jeho vývoji a provozu se podílí studenti ze tří pražských vysokých škol - ČVUT, VŠE a ČZU.

O celou logiku systému Uniqway se stará serverová aplikace. Základ této aplikace vznikl v rámci bakalářské práce „Databáze a rozhraní pro modul automobilu pro systém sdílení automobilů více uživateli“ od studenta FEL ČVUT Richarda Vachuly [1]. S postupem času se aplikace zvětšuje, zesložiťuje a je čím dál náročnější jí udržovat a vylepšovat. Přechod na architekturu mikroslužeb velmi usnadní celou správu serverové aplikace. Oprava chyb nebude vyžadovat znovunasazení celé aplikace, chyby nutně nezpůsobí pád celé aplikace, ale jen

dotčených mikroslužeb. Dále se také usnadní vývoj aplikace, jelikož každou mikroslužbu dostane na starosti malý tým, který ji bude spravovat a nasazovat nezávisle na ostatních. V neposlední řadě je nezanedbatelnou výhodou snadnější nábor nových členů na vývoj serverové aplikace. Pro nové členy je velmi náročné se zorientovat v tak velké aplikaci a přechod na mikroslužby toto velmi usnadní. Převést serverovou aplikaci z jedné architektury na jinou není snadný úkol a proto na tomto úkolu budu spolupracovat s ostatními členy týmu, kde jsme si rozdělili funkcionality, kterým se každý z nás bude věnovat. Ačkoliv máme úkoly rozdělené, neustále musíme spolupracovat při návrhu celkového pohledu na systém a při návrhu spolupráce různých mikroslužeb, kde každou dělá jiný člen týmu.

Toto téma diplomové práce jsem si zvolil, protože jsem členem Uniqway týmu již třetím rokem a téměř dva roky se podílím na vývoji zmíněné serverové aplikace. V týmu vznikl požadavek na přepsání serverové aplikace do architektury mikroslužeb a jelikož mě zajímá toto téma a obecně téma nových způsobů řešení, zadání diplomové práce jsem přijal.

Tato práce se bude zabývat problematikou architektury mikroslužeb a přechodu z monolitické architektury na architekturu mikroslužeb v rámci serverové aplikace služby Uniqway. V teoretické části budou probrány architektury serverových aplikací - monolitická a architektura mikroslužeb, jejich výhody a nevýhody. Dále bude provedena analýza existujícího řešení serverové aplikace Uniqway. Nakonec bude provedena analýza a návrh, ve kterém budou definovány požadavky na jednotlivé mikroslužby a bude vytvořen doménový model.

V praktické části bude navrženo API rozhraní každé mikroslužby a budou popsány jednotlivé komponenty, ze kterých se mikroslužby skládají. Dále budou popsány technologie, které byly k implementaci mikroslužeb použity. Nakonec bude aplikace otestována a způsob testování bude v práci popsán.

Kapitola 1

Cíl práce

Cílem této práce je analyzovat, navrhnout a částečně implementovat přechod z monolitické architektury na architekturu mikroslužeb serverové aplikace v projektu Uniqway, který provozuje systém sdílení automobilů více uživateli. Detailní popis cílů práce je následující:

- Rešerše vhodných způsobů transformace.
- Analýza současného řešení serverové aplikace.
- Návrh procesu transformace z monolitické architektury do architektury mikroslužeb. Návrh bude zaměřen na mikroslužby zabývající se správou textů a lokalizací, obchodem s odměnami a tvorbou a správou rezervací.
- Otestování a zdokumentování vzniklého řešení.
- Návrh budoucích kroků jak transformaci dokončit.

Kapitola 2

Architektura serverových aplikací

Většina dnešních systémů se neobejde bez serverové aplikace, která zajišťuje ukládání dat, obchodní logiku a další důležité funkcionality jako je např. validace dat. Tato architektura je nazývána client-server (klient-server) a skládá se většinou z jednoho serveru (serverové aplikace) a několika klientů. Velkou výhodou tohoto přístupu je to, že klienti jsou relativně jednoduché aplikace neobsahující obchodní logiku, která je ukrytá v serverové aplikaci.

V této kapitole budou popsány různé architektury, podle kterých se dají serverové aplikace realizovat. Nejprve budou popsány základní pojmy této problematiky, poté bude popsána monolitická architektura a její výhody a nevýhody. Dále se kapitola věnuje architektuře mikroslužeb, jejím výhodám a nevýhodám a zároveň tuto architekturu srovnává s monolitickou architekturou.

2.1 Základní pojmy

V této sekci budou vysvětleny základní pojmy věnující se problematice serverových aplikací.

API Application programming interface, je množina příkazů, funkcí, protokolů a objektů, která poskytuje možnost různým systémům komunikovat mezi sebou. [2]

HTTP HyperText Transfer Protocol je protokol používaný k přenosu dat přes internet. [3]

JSON JavaScript Object Notation je kompaktní formát na výměnu dat. JSON je založený na textové reprezentaci (není binární) a je jazykově nezávislý. [4]

Load Balancer Load balancer (vyvažovač zátěže) je zařízení, které distribuuje síťový nebo aplikační provoz napříč různými servery, nebo instancemi serverů. Zvyšuje celkový výkon aplikace, tím že rozprostírá zátěž mezi zmíněné servery / instance serverů. [5]

XML Extensible Markup Language je značkovací jazyk, který se používá pro definování dokumentů, které umí přečíst jakákoliv XML kompatibilní aplikace. Tento jazyk se také používá pro přenos dat přes internet. [6]

MVC Model View Controller je návrhový vzor, který aplikaci dělí na základní 3 části - Model (data), View (uživatelské rozhraní) a Controller (kontrolér, který zpracovává vstup od uživatele). Model obsahuje data, která aplikace používá a zajišťuje jejich ukládání např. do databáze. View obstarává zobrazování těchto dat uživatelům. Controller přijímá vstup od uživatelů a aktualizuje model a view podle daného vstupu. [7]

2.1.1 REST

REST, neboli REpresentational State Transfer je architektonický styl, který poskytuje standardy při komunikaci mezi systémy na webu a tím ji usnadňuje. Systémy, které jsou v souladu s REST pravidly (často se nazývají RESTful), jsou charakterizovány tím, že jsou bezestavové a rozdělují zodpovědnosti mezi klienta a server. [8]

Implementace klienta je nezávislá od implementace serveru. Klient i server mohou dělat změny ve své implementaci, aniž by to ovlivnilo toho druhého pokud dodrží komunikační rozhraní (API), tedy formát zasílaných zpráv a endpointy (URL cesta k danému zdroji), kam tyto zprávy zasílat. [8]

REST je bezestavový, to znamená, že server nepotřebuje vědět v jakém stavu je klient a obráceně. Díky tomu, že je bezestavový, obě komunikující strany dokážou porozumět jakékoliv obdržené zprávě aniž by potřebovali znát předchozí zprávy nebo další informace. [8]

V REST architektuře klienti zasílají požadavky na získání, nebo úpravu zdrojů a server jim na tyto požadavky odpovídá. REST používá ke komunikaci podmnožinu HTTP. Požadavek od klienta se většinou skládá z:

- HTTP slovesa, které definuje o jakou operaci se jedná,
- hlavičky, která specifikuje další informace o požadavku,
- cesty ke zdroji a
- nepovinného těla požadavku, které obsahuje data daného požadavku. [8]

Existují 4 základní HTTP slovesa:

- GET - Slouží k získání specifického zdroje podle id, nebo kolekci daných zdrojů.

- POST - Slouží k vytvoření nového zdroje.
- PUT - Slouží k modifikaci daného zdroje podle id.
- DELETE - Slouží ke smazání daného zdroje podle id. [8]

Tělo v POST a PUT požadavcích a odpovědích se vetšinou posílá ve formátu JSON [4], nebo XML [6]. Každá odpověď obsahuje i HTTP kód [9], který popisuje jestli se daná operace zdařila, nebo jaká chyba nastala. Kódy popisující úspěch jsou následující:

- 200 (OK) se používá u GET a PUT,
- 201 (CREATED) se používá u POST a
- 204 (NO CONTENT) se používá u DELETE. [8]

2.2 Monolitická architektura

Monolitická aplikace popisuje jednovrstvou aplikaci, kde jsou různé komponent spojeny do jednoho programu jedné platformy. Komponenty mohou dle [10] být:

Autorizace je zodpovědná za autorizaci uživatele.

Prezentace je zodpovědná za zpracování HTTP dotazů a odpovídá pomocí HTML nebo JSON/XML (pro rozhraní webových služeb).

Obchodní logika je zodpovědná za aplikační obchodní logiku.

Databázová vrstva obsahuje DAO (data access object, objekty obsluhující přístup k datům) objekty zodpovědné za přístup do databáze.

Aplikační integrace je integrace s dalšími službami (například přes zasílání zpráv nebo REST API 2.1.1), nebo integrace s dalšími zdroji dat.

Notifikační modul je zodpovědný za zasílání emailových notifikací.

Příkladem takové aplikace může být aplikace e-shopu. Tato aplikace autorizuje zákazníky, přijímá objednávky, kontroluje zásoby produktů, autorizuje platby a doručuje objednané produkty. Taková aplikace se skládá z několika komponent, kdy každá komponenta obsluhuje nějakou z výše uvedených činností. Přesto, že monolitická aplikace má vícero různých komponent, modulů nebo služeb, tak se vydává jako jedna aplikace. Monolitickou architekturu popisuje obrázek 2.1. Výhody této architektury jsou podle [10] následující:

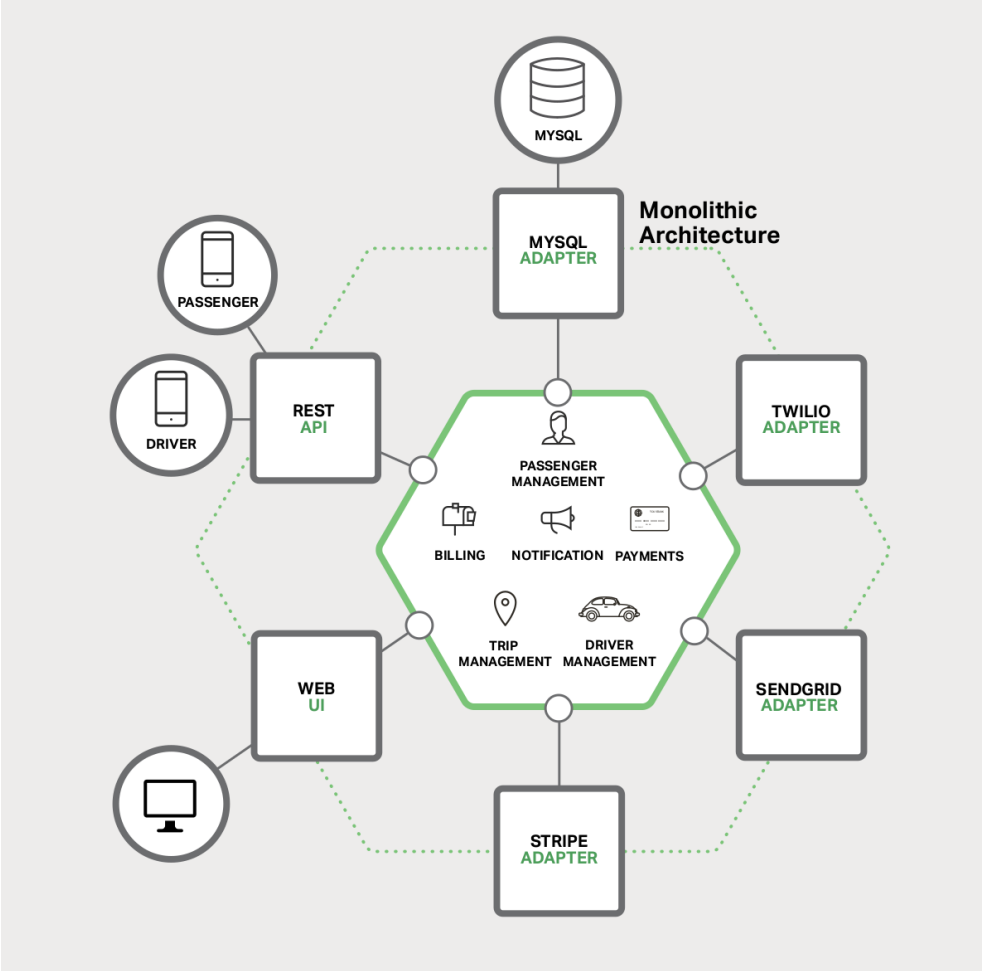
- Jednoduché na vývoj - Na začátku projektu je mnohem jednoduší začít monolitickou architekturou, kdy je vyvíjena a udržována pouze jedna aplikace.

- Jednoduché na testování - Například je možné jednoduše implementovat end-to-end (koncové, průchod celou aplikací) testování, pouze spuštěním aplikace a otestování UI (user interface, uživatelského rozhraní) například pomocí Selenia [11].
- Jednoduché na nasazení - Stačí zkopírovat zabalenou aplikaci na server.
- Jednoduché na škálování horizontálně pomocí spuštění více kopií běžícím za load balancerem (vyvažovač zátěže).

Jako u veškerých technologií, postupů nebo architektur se zde vyskytuje i několik nevýhod. Nevýhodami můžou podle [10] být:

- Údržba - pokud je aplikace příliš velká a komplexní na celkové pochopení, tak je velice náročné provádět nějaké změny rychle a správně.
- Velikost aplikace - velikost může zvyšovat čas potřebný k zapnutí aplikace.
- Znovunasazování - každé vylepšení (update) aplikace způsobí znovunasazení celé aplikace.
- Škálovatelnost - u monolitických aplikací může být problémem škálovatelnost, pokud mají různé moduly konfliktní nároky na zdroje. Další nevýhodou ohledně škálovatelnosti je, že pokud aplikace potřebuje obsluhovat více požadavků, tak za load balancerem udělá více instancí celé této monolitické aplikace místo toho, aby se zvýšil počet instancí jen té části, která je pod náporem.
- Spolehlivost - chyba v jakékoliv modulu (například únik paměti) může potenciálně shodit celý proces. A co víc, jelikož jsou všechny instance aplikace identické, tak tato chyba ovlivní dostupnost celé aplikace.
- Špatná adaptace na nové technologie - jelikož změna jazyka, nebo frameworku [12] ovlivní celou aplikaci a jejich změna vyžaduje velké úsilí, hodně práce a času.

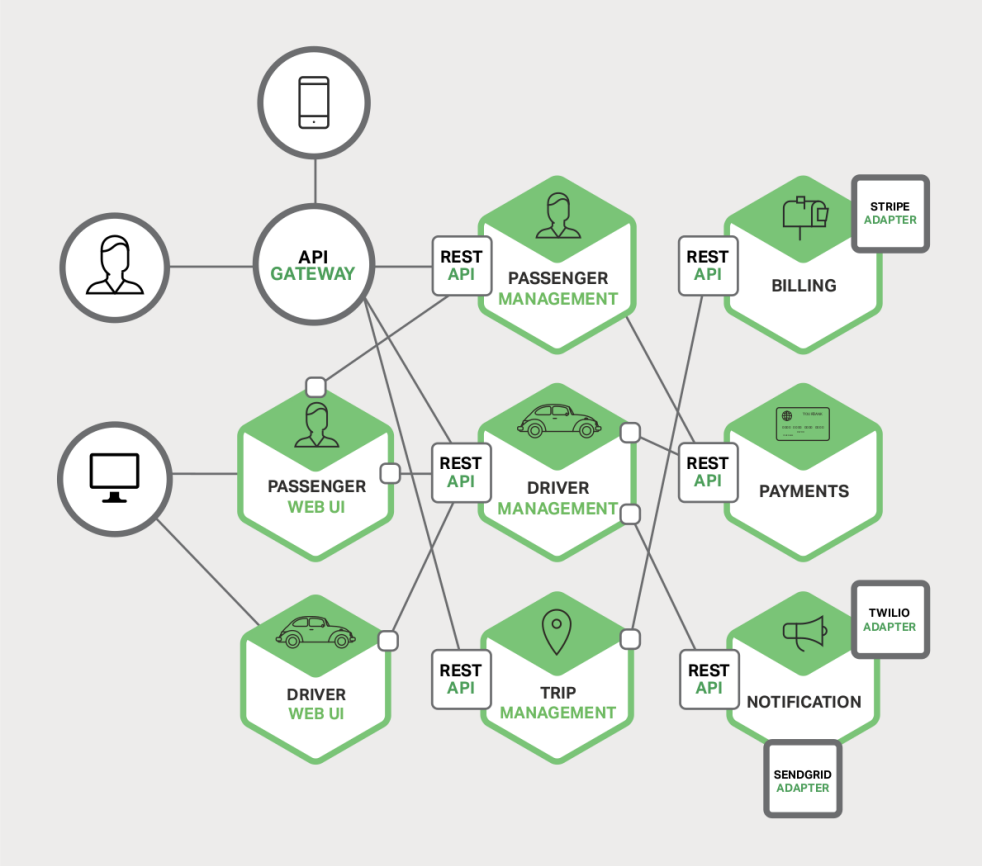
Hlavní nevýhodou je zvyšování komplexnosti takové aplikace. Postupem času se z monolitické aplikace stane aplikace s několika miliony řádky kódu, které jsou různě provázané a je extrémně náročné takovou aplikaci udržovat a ještě těžší ji nějak upravovat. Pro takovouto aplikace je téměř nemožné měnit používané technologie a proto postupem času zastarává. Toto je jedním z důvodů proč některé velké firmy používají 10 a více let staré technologie a verze jazyků, které již nejsou podporované. Zkrátka nejsou schopné jejich složitý kód předělat na novější. Jedním z řešení těchto problémů monolitických aplikací je přechod na architekturu mikroslužeb.



Obrázek 2.1: Architektura monolitické aplikace [13]

2.3 Architektura mikroslužeb

Idea, skrývající se pod tímto názvem je taková, že se místo tvorby jedné velké aplikace vytváří množina malých, navzájem propojených služeb (mikroslužeb). Tato služba běžně obstarává množinu souvisejících funkcionalit jako například správa objednávek, správa uživatelů apod. Každá mikroslužba je mini aplikace, která ma svou architekturu skládající se z obchodní logiky, perzistentní vrstvy a množství adaptérů, které zajišťují komunikaci s okolím. Některé mikroslužby vystavují API, které ostatní mikroslužby používají pro komunikaci s danou mikroslužbou, nebo je používáno klientskými aplikacemi. Jak by mohla vypadat architektura mikroslužeb je na obrázku 2.2. [13, 14]



Obrázek 2.2: Architektura mikroslužeb [13]

Jelikož každá mikroslužba obsluhuje relativně samostatnou a omezenou funkcionalitu, je mnohem snazší tuto funkcionalitu změnit a následně nasadit pouze tuto ovlivněnou mikroslužbu. V tento moment již nemusíme kvůli změně v kódu nasazovat celou monolitickou aplikace, ale pouze ty mikroslužby, kterých se daná změna týká. Architektura mikroslužeb má mnoho výhod. **Výhody** mikroslužeb podle [13] jsou:

- Snižuje problém komplexity. Dekomponuje velkou monolitickou aplikaci na množinu mikroslužeb. Zatímco funkcionalita zůstává stejná, aplikace byla rozebrána na malé udržitelné části a každá tato část má dobře definované rozhraní.

- Architektura mikroslužeb zajišťuje úroveň modularity, které je velice těžké dosáhnout v monolitické architektuře.

- Individuální mikroslužby je mnohem rychlejší vyvinout, porozumět jim a udržovat je.

- Každá mikroslužba může být vyvíjena nezávisle týmem (částí týmu), který se na tuto mikroslužbu soustředí. Vývojáři mají volnou ruku při výběru technologie pro danou mikroslužbu, pokud zachovají dohodnuté rozhraní. Toto napomáhá v používání současných technologií a ne zastaralých, které se vybrali na počátku projektu. A také je snazší přepsat mikroslužbu do nové technologie než celou monolitickou aplikaci.

- Mikroslužby mohou být vydávány nezávisle na sobě. Změny, které jsou pouze lokální a ovlivní jednu mikroslužbu mohou být vydané nezávisle na ostatních mikroslužbách.

- Mikroslužby mohou být škálovány nezávisle. Můžeme nasadit pouze takové množství instancí každé mikroslužby, které zvládne obsloužit nároky na danou funkcionalitu. Každou mikroslužbu můžeme nasadit na takový hardware, který nejlépe odpovídá její činnosti. Jaký požadavek bude obsloužen jakou instancí určuje load balancer.

Jako každá architektura má i architektura mikroslužeb své nevýhody. Tyto **nevýhody** mohou být:

- První nevýhoda se skrývá v názvu samotném - architektura mikroslužeb. Jak „mikro“ má být mikroslužba? Malé služby jsou prostředkem k dosažení a ne primárním cílem. Cílem mikroslužeb je dostatečně dekomponovat aplikaci s cílem usnadnit agilní vývoj aplikací.

- Další nevýhodou je komplexita způsobená tím, že aplikace poskládaná z mikroslužeb je distribuovaný systém. Vývojáři musí implementovat mezi-službovou komunikaci. Dále také musí implementovat kód, který řeší částečná selhání, protože cílová mikroslužba nemusí být dostupná.

- Rozdělené databázové schéma je další výzvou, se kterou se vývojáři musí při implementaci architektury mikroslužeb potýkat. Obchodní transakce, které upravují několik entit, jsou velmi běžné. Řešit takový problém v monolitické aplikaci je velmi jednoduché, protože používá pouze jednu databázi. V architektuře mikroslužeb musíme aktualizovat několik různých databází, kde každá připadá jiné mikroslužbě.

- Testování mikroslužeb je také mnohem komplexnější. Napsat testovou třídu, která spustí monolitickou aplikaci a otestuje její REST API je jednoduché. Avšak napsat takový test, který spustí mikroslužbu a všechny mikroslužby, na kterých závisí, nebo test, který odpovědi ostatních mikroslužeb jen simuluje, je mnohem náročnější.

- Implementování změny, která zasáhne vícero mikroslužeb, je taktéž mnohem složitější, než v monolitické aplikaci. V architektuře mikroslužeb musíme pečlivě naplánovat nejen dané změny, ale také v jakém pořadí budou upravené mikroslužby nasazeny. Naštěstí pro nás, změny, které zasahují více mikroslužeb jsou relativně vzácné.

- Nasazování aplikací založených na architektuře mikroslužeb je náročnější - musíme nasazovat několik (v mnoha případech desítky až stovky) různých mikroslužeb.

- Každá mikroslužba má vícero běžících instancí a mezi těmito instancemi musíme implementovat způsob jak se požadavky dostanou k dané instanci a jak se mikroslužby navzájem najdou (service discovery). [13]

Monolitická architektura dává smysl pouze pro malé, jednoduché aplikace. Poté co se aplikace rozroste její míra provázanosti a komplexnosti vystoupá nad únosnou mez, taková aplikace se stává neudržitelnou. Architektura mikroslužeb je lepší volbou pro komplexní a rychle se rozvíjející aplikace i přes všechny jejich nevýhody a výzvy při implementaci. [13]

2.3.1 Komunikace klientských aplikací se serverovou

Po rozhodnutí přejít na architekturu mikroslužeb nastává otázka, jakým způsobem budou klienti interagovat s mikroslužbami. V monolitické aplikaci je jen jedna množina API endpointů, které klienti mohou volat, ale v architektuře mikroslužeb vystavuje každá mikroslužba svoje API. Tento problém má dvě základní řešení: přímá komunikace klient-mikroslužba nebo pomocí API Gateway. V této sekci budou probrány oba přístupy.

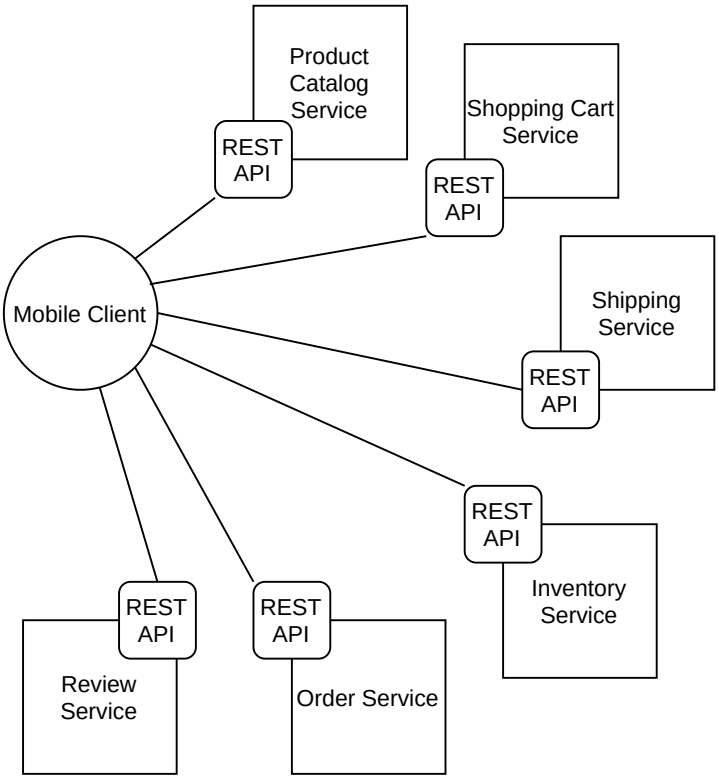
2.3.1.1 Přímá komunikace klient-mikroslužba

V tomto případě komunikuje klient přímo s danou mikroslužbou. Klient zašle požadavek přímo na endpoint mikroslužby. Každá mikroslužba má veřejné endpointy dostupný například na následující URL:

```
https://serviceName.api.company.name
```

Tato URL je namapovaná na load balancer příslušející dané mikroslužbě a ten distribuuje požadavky na instance této mikroslužby. Pokud chce klient komplexní informace, musí komunikovat s každou mikroslužbou, která poskytuje část této informace a celek si poskládat sám. Příklad jak by taková komunikace mohla vypadat je na obrázku 2.3

Bohužel v tomto řešení se skrývá několik problémů a limitací. Jedním z problémů je neshoda s potřebami klienta a rozdrobeností informací napříč mikroslužbami. V tomto případě musí klient udělat několik požadavků, aby získal



Obrázek 2.3: Přímá komunikace klient - mikroslužba [13]

informaci, kterou hledá, místo toho aby udělal jeden požadavek, který mu vrátí vše potřebné.

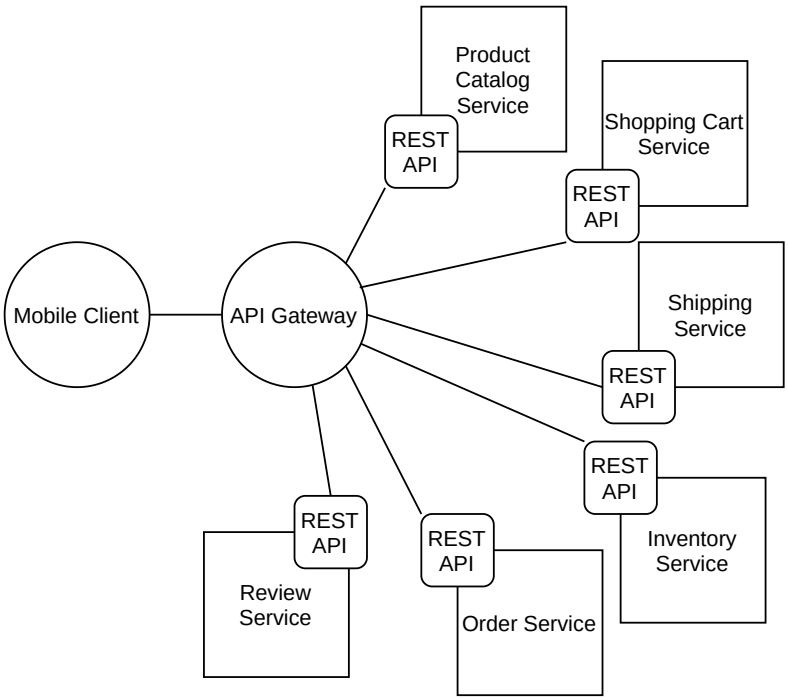
Další nevýhodou, kterou tento přístup přináší, je složitost refaktorování mikroslužeb. Časem se může stát, že je potřeba změnit rozdělení systému do mikroslužeb a některé mikroslužby sloučit do jedné, některé naopak rozdělit do více. Pokud klienti komunikují přímo s mikroslužbami, může tato změna být extrémně náročná a vyžaduje změny i na straně klienta.

Výhodou tohoto řešení je relativní snadnost implementace a není potřeba žádných dalších komponent. Tento přístup se v praxi ale příliš nepoužívá z důvodu nevýhod popsaných výše. [13]

2.3.1.2 API Gateway

Většinou mnohem vhodnější přístup je použít API Gateway (API brána). API Gateway je komponenta, která slouží jako jediný vstupní bod do systému. Tato brána zapouzdřuje interní systémovou architekturu a poskytuje API, které mohou klienti používat. API Gateway může mít i jiné zodpovědnosti

jako je autentizace, monitorování, load balancing apod. Všechny požadavky z klientů jdou nejprve do API Gateway, ta poté tyto požadavky přesměruje na mikroslužby, které mají daný požadavek na starosti. API Gateway často zpracovává požadavky tím, že volá několik mikroslužeb a poté poskládá jejich odpovědi v jeden výsledek, který je zpět zaslán klientovi. Příklad komunikace přes API Gateway je na obrázku 2.4.



Obrázek 2.4: API Gateway [13]

Použití API Gateway má své **výhody** a **nevýhody**. Hlavní výhodou používání API Gateway je, že zapouzdřuje vnitřní strukturu aplikace. Klient tedy může jednoduše volat bránu, místo zasílání dotazu na konkrétní službu. Toto snižuje počet požadavků jdoucích z klienta a také to zjednodušuje jeho kód.

API Gateway je další vysoce dostupná komponenta, která musí být vyvíjena a udržována a toto je hlavní nevýhodou. Snadno se může stát, že API Gateway bude úzkým hrdlem celé aplikace. Další nevýhodou je, že se brána musí upravit vždy, když je třeba vystavit novou funkcionalitu klientům. Je velmi důležité udržovat aktualizace API Gateway co nejjednodušší a nejmenší. Jinak se stane, že budou vývojáři nuceni čekat ve frontě na aktualizace brány, aby se jejich funkcionalita dostala ke klientům. Přes uvedené nevýhody dává použití API Gateway smysl pro většinu skutečných aplikací. [13]

2.3.2 Komunikace mezi mikroslužbami

V monolitické aplikaci spolu různé komponenty komunikují zavoláním metody na úrovni programovacího jazyka. V mikroslužbách to takto nejde, protože se jedná o distribuovaný systém běžící na více zařízení. Každá instance mikroslužby je většinou process a proto musíme použít mezi procesovou komunikaci (IPC, inter-process communication). [13]

Styly interakce mezi mikroslužbami mohou být kategorizovány podle dvou dimenzí. První dimenzí je jestli se jedná o komunikaci jedné mikroslužba s jednou (one-to-one), nebo jedné s více (one-to-many):

One-to-one Každý požadavek klientské mikroslužby je zpracován přesně jednou instancí jiné mikroslužby.

One-to-many Každý požadavek klientské mikroslužby je zpracován více mikroslužbami.

Druhou dimenzí je jestli je komunikaci synchronní, nebo asynchronní:

Synchronní Klient očekává odpověď včas a může být blokován v čase čekání.

Asynchronní Klient není blokován, zatímco čeká na odpověď a odpověď (pokud nějaká je) nemusí být zaslána ihned.

2.3.2.1 One-to-one komunikace

Detailní popis komunikačních stylů ve **one-to-one** komunikaci je následující:

Požadavek/odpověď Klient zašle požadavek na mikroslužbu a čeká na odpověď. Klient očekává, že odpověď dorazí včas a při čekání se může zablokovat, dokud nepřijde odpověď nebo nevyprší časový limit.

Notifikace Klient zašle požadavek na mikroslužbu, ale neočekává odpověď a odpověď se většinou ani nepošle. Tomuto způsobu komunikace se také říká one-way request (jednosměrný požadavek).

Požadavek / asynchronní odpověď Klient zašle požadavek na mikroslužbu, která odpovídá asynchronně. Klient není blokován zatímco čeká a je navržený tak, že odpověď nějakou dobu nemusí přijít.

2.3.2.2 One-to-many komunikace

One-to-many komunikace má následující styly:

Publikování/odebírání Klient zveřejní notifikační zprávu, která je zachycena 0 nebo více zaujatými službami. V angličtině se tento styl nazývá publish/subscribe.

Publikování / asynchonní odpovědi Klient zveřejní požadavek a čeká určitý časový úsek na odpovědi ze zaujatých služeb. [13]

Tabulka 2.1 popisuje tyto komunikační styly interakce v každé ze dvou dimenzí.

Tabulka 2.1: Styly IPC [13]

	ONE-TO-ONE	**ONE-TO-MANY**
Synch.	Požadavek/odpověď	-
Asynch.	Notifikace	Publikování/odebírání
	Požadavek / asynch. odpověď	Publikování / asynch. odpověď

2.3.3 Objevování mikroslužeb (service discovery)

Pokud je potřeba zavolat mikroslužbu, která vystavuje REST API, tak je potřeba znát pozici její instance na síti (IP adresu a port, URL). V případě mikroslužeb je toto problém, jelikož se instance mikroslužeb tvoří dynamicky podle škálovacích nároků a předem není známo jejich URL, které je jim přiřazeno až při jejich vzniku. Existují dva hlavní postupy při objevování mikroslužeb: objevování na straně klienta a objevování na straně serveru. [13]

2.3.3.1 Objevování na straně klienta

V tomto přístupu je klient zodpovědný za určování síťových pozic dostupných instancí mikroslužeb a rozkládání zátěže (load balancing) mezi ně. Klient se dotazuje do registru mikroslužeb, což je databáze dostupných instancí mikroslužeb. Klient poté použije algoritmus rozkládání zátěže k vybrání jedné instance a poté zašle požadavek. Síťová pozice instance mikroslužby je do tohoto registru přidána při startu této instance a je odebrána při ukončení dané instance. Registrace instance je většinou obnovována pomocí heartbeat mechanismu (periodické dotazování na danou mikroslužbu a když neodpoví několikrát za sebou, je z registru vyřazena).

Objevování na straně klienta má mnoho výhod a nevýhod. Tento vzor je relativně jednoduchý a kromě registru služeb nemá žádné další komponenty navíc. Další výhodou je schopnost klienta dělat inteligentní, specifické pro danou aplikaci vyvažování zátěže, jelikož má informace o instancích z registru. Hlavní nevýhodou tohoto vzoru je, že provazuje klienty a registr mikroslužeb a je tedy potřeba implementovat mechanismus na objevování mikroslužeb v každém klientovi a programovacím jazyce, ve kterém jsou klienti napsáni. [13]

2.3.3.2 Objevování na straně serveru

Dalším způsobem jak objevovat mikroslužby, je objevování na straně serveru. Klient posílá požadavek na mikroslužbu přes load balancer. Load balancer se dotazuje registru služeb a přesměrovává požadavky na dostupné instance mikroslužeb. Stejně jako při objevování na straně klienta jsou instance registrovány do registru a odebírány z něj při jejich vzniku a zániku. Příkladem takového load balanceru může být AWS Elastic Load Balancer (ELB) [15].

I tento způsob má několik výhod a nevýhod. Velkou výhodou je, že detaily objevování jsou abstrahovány z klientů. Klient jednoduše posílá požadavky na load balancer. Toto eliminuje potřebu implementovat vyvažující logiku a objevování na straně klientů. Další výhodou je, že některé prostředí poskytují tento load balancer zdarma (v rámci poskytovaných služeb) - například již zmíněný AWS ELB. Nevýhodou je, že je load balancer další systémovou komponentou, která musí být spravována. [13]

2.4 Způsoby přechodu mezi architekturami

V této práci bylo již vysvětleno co znamená monolitická architektura (2.2), co má za výhody a nevýhody. Také bylo probráno, co obsahuje architektura mikroslužeb (2.3), jaké má výhody a nevýhody a proč je vhodné velké aplikace psát v této architektuře. V této části práce budou probrány různé strategie přechodu mezi zmíněnými architekturami.

Prvním způsobem, jak monolitickou aplikaci přepsat do mikroslužeb, je přepsání celé aplikace najednou a od nuly. Tato strategie je nazývaná také „Velký třesk“. Přepisování monolitické aplikace pomocí této strategie má velmi malé naděje na úspěch a určitě není doporučeno takto postupovat. Lepším způsobem přepisu monolitické aplikace je inkrementálně přepisovat kusy monolitické aplikace do mikroslužeb. Strategie, jak toto dělat, jsou popsány v následujících částech. [13]

2.4.1 Strategie 1 - Nová funkcionalita v nové mikroslužbě

V této strategii při implementaci nové funkcionality nepřidáme tuto funkcionalitu do monolitické aplikace, ale vytvoříme novou mikroslužbu, která bude funkcionalitu zajišťovat. Vytvoříme směrovač požadavků (obdoba API gateway 2.3.1.2), který nové požadavky směruje na vzniklou mikroslužbu a staré na monolitickou aplikaci.

Tuto strategii je vhodné zvolit, pokud se monolit stal již neudržitelným a není žádoucí aby se nadále zvětšoval. Hlavními výhodami této strategie jsou:

- Monolitická aplikace se přestane zvětšovat a stávat se neudržitelnější.

- Nová mikroslužba může být vyvíjena, vydávána a škálována nezávisle na monolitické aplikaci.
- Vývojáři si vyzkouší, jaké to je tvořit mikroslužby, předtím, než začnou předělávat již existující funkcionalitu.

Ačkoliv má tato strategie několik již zmíněných výhod, tak nijak nezasahuje do existující monolitické aplikace a tudíž neřeší již existující problémy s komplexností a udržitelností této aplikace. Na tyto problémy slouží následující strategie. [13]

2.4.2 Strategie 2 - Rozdělit prezentační logiku od obchodní a databázové

Tato strategie již zmenšuje existující monolit a to na prezentační vrstvu a na vrstvu obchodní logiky a vrstvy přistupující k datům. Typická obchodní aplikace se většinou skládá ze tří typů odlišných komponent:

- Prezentační vrstva - zde se nachází komponenty, které zpracovávají HTTP požadavky a implementují buď REST API nebo webové rozhraní založené na HTML.
- Vrstva obchodní logiky - zde jsou komponenty, které jsou jádrem aplikace a implementují obchodní pravidla.
- Datová vrstva - datová vrstva se skládá z komponent, které přistupují k datům a to například do databází, nebo ke zprávám.

Většinou existuje jednoznačné oddělení prezentační vrstvy a obchodní a datové vrstvy. Toto rozdělení rozdělí monolitickou aplikaci na dvě menší aplikace. Jedna z nich obsahuje prezentační vrstvu a druhá obchodní vrstvu s datovou vrstvou.

Výhodou tohoto oddělení je, že se obě aplikace mohou vyvíjet nezávisle na sobě. Další výhodou je to, že to umožňuje vyvíjet prezentační vrstvu mnohem rychleji a usnadňuje to například A|B testování. Jinou výhodou je vystavení API, které mohou další mikroslužby volat. Tato strategie je také jen částečné řešení. Je pravděpodobné, že i po rozdělení monolitické aplikace na tyto dvě, budou obě stále neudržitelně velké. K dosažení lepší výsledků je potřeba použít i třetí strategii. [13]

2.4.3 Strategie 3 - Extrahování mikroslužeb

Třetí strategie zahrnuje postupné předělávání existujících modulů z monolitické aplikace do mikroslužeb. Pokaždé, když je modul extrahován z monolitu, tak se tato aplikace zmenší. Jakmile je takto odebráno dostatek modulů přestane být komplexita monolitické aplikace problémem - buď zmizí úplně, nebo se stane sama mikroslužbou.

Přijít na to, které moduly extrahovat jako první, je náročný úkol. Dobrým začátkem je extrahovat nejprve moduly, které nemají mnoho vazeb na zbytek aplikace a je jednoduché je z aplikace vyčlenit. Tímto postupem získají vývojáři zkušenosti s extrahování modulů a budou později moci se pustit do složitějších úprav. Dalším kritériem při volbě modulu k extrakci je, jak často se mění. Moduly, které se mění velmi často je lepší extrahovat dříve a využívat u nich výhod mikroslužeb co nejdříve. Další moduly, které se vyplatí extrahovat brzy jsou takové, které mají požadavky na zdroje velmi odlišné od zbytku aplikace. Díky vyjmutí jim můžeme poskytnout zdroje, které potřebují.

Prvním krokem v extrakci modulu je definování rozhraní mezi modulem a zbytek monolitu. Velmi často se jedná o oboustranné rozhraní, jelikož monolit potřebuje data z modulu a obráceně. Při implementaci tohoto rozhraní je často potřeba udělat významnou změnu kódu, aby se odstranila složitá provázanost mezi modulem a zbytek aplikace. Druhým krokem je předělání takto odděleného modulu do samostatné mikroslužby. [13]

Při extrahování mikroslužeb je dobré postupovat iteračně. Nejprve jsou vyčleněny velké moduly a aplikace je rozdělena na malé množství velkých „mikroslužeb“. V další iteraci se tyto mikroslužby opět rozdělí na další mikroslužby. Tato iterace je opakována dokud není granularita mikroslužeb dostatečně malá. [16]

2.4.4 Shrnutí

Při přechodu mezi monolitickou architekturou a architekturou mikroslužeb určitě není vhodné přepisovat celou aplikaci od začátku a najednou. Mnohem lepší přístup je postupné a inkrementální přepisování monolitu do mikroslužeb. K tomuto slouží 3 výše zmíněné strategie, kde ve většině případů je dobré nepoužít jen jednu z nich ale zkombinovat je všechny dohromady. Díky tomuto přístupu bude monolitická aplikace čím dál menší, většina funkcionality přejde do mikroslužeb a postupem času monolit buď úplně zanikne a nebo se stane jen další mikroslužbou. [13]

Kapitola 3

Analýza současného řešení

Tato kapitola se zabývá analýzou současného řešení Uniqway serverové aplikace. Kapitola se nejprve věnuje infrastruktuře v projektu Uniqway a poté se zaměřuje na samotnou serverovou aplikaci. Tuto aplikaci popisuje nejprve celkově a poté se zaměřuje na části, které se zaměřují na správu textů a lokalizací, obchodem s odměnami a tvorbou a správou rezervací.

3.1 Architektura Uniqway systému

Systém Uniqway se skládá z několika komponent, které spolu dohromady komunikují a vytváří tak všechny funkcionality, které jsou nezbytné k fungování celé služby. Hlavní komponentou v tomto systému je serverová aplikace, která je srdcem celého projektu a je jí věnována následující sekce.

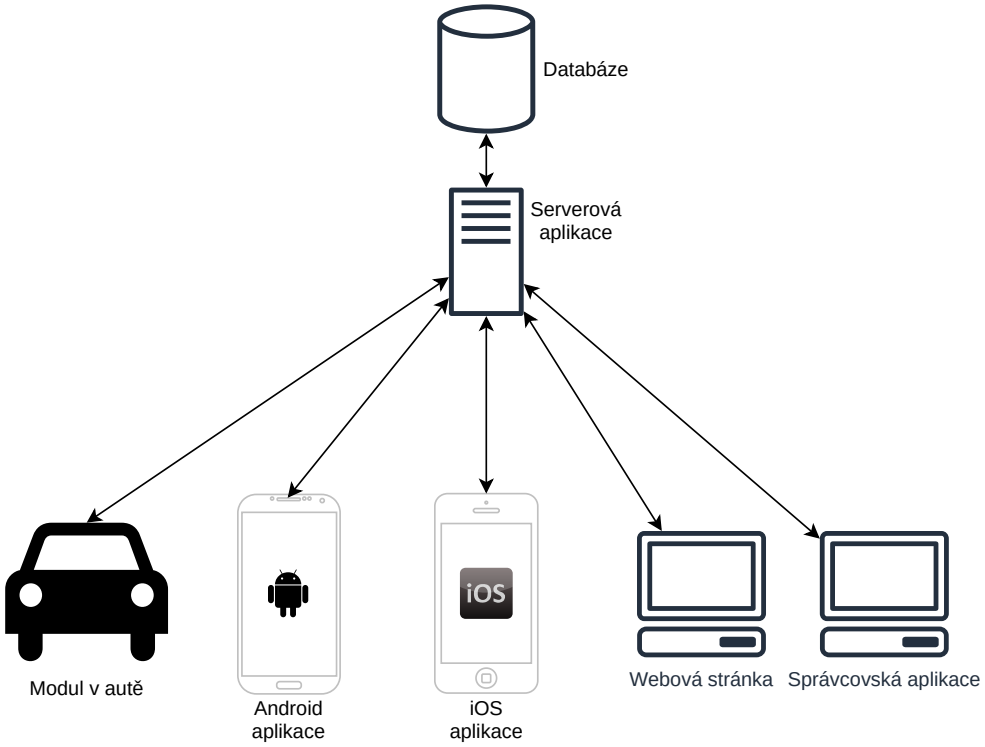
Dalšími komponentami Uniqway systému jsou Android a iOS klientské aplikace. Tyto aplikace zajišťují komunikaci uživatelů systému se serverovou aplikací. V těchto aplikacích si mohou uživatelé prohlédnout dostupná vozidla, zarezervovat si je, odemykat a zamykat, jelikož je použití vozů bezklíčové, prohlížet si parkovací zóny, ve kterých mohou auta vracet a v neposlední řadě zde mohou nakupovat Uniqway produkty v obchodu s odměnami.

Další komponentou bez které by systém Uniqway nemohl fungovat jsou moduly ve vozidlech. Tyto moduly komunikují se serverovou aplikací, hlásí jí svou polohu, stav zamknutí a jiné důležité informace. Server jim v odpovědích na jejich požadavky zasílá své instrukce. Tyto instrukce mohou být zamknutí, nebo odemknutí.

Správcovská webová aplikace je další komponenta Uniqway systému. Přes tuto komponentu správci Uniqway platformy spravují uživatele, vozový park, obchod s odměnami a další důležitá data, která jsou k fungování služby nezbytná.

Webová stránka Uniqway, je další komponentou, která komunikuje se serverovou aplikací a získává od ní užitečné informace jako polohy parkovacích

zón a právě dostupných vozidel. Přes tuto aplikaci se uživatelé mohou zaregistrovat a následně vyplnit údaje k jejich online ověření. Jak tato infrastruktura vypadá je na obrázku 3.1.



Obrázek 3.1: Architektura Uniqway systému

3.2 Serverová aplikace

Serverová aplikace (zkráceně server) systému Uniqway je nejdůležitější komponentou celého projektu. Obsahuje veškerou obchodní logiku, validační pravidla a propojuje všechny komponenty systému dohromady. V následující části bude tento server představen více podrobně.

3.2.1 Celkový pohled

Serverová aplikace Uniqway je psaná ve frameworku Play [17] a v programovacím jazyku Java [18]. Jako databáze je používána PostgreSQL [19]. Objektově-relační mapování (ORM) je realizováno pomocí Ebean ORM [20]. Aplikace komunikuje s ostatními komponentami v Uniqway systému pomocí REST API.

Databázový model a API budou popsány v následujících částech věnujících se funkcionalitě, která by měla být nahrazena mikroslužbami tvořenými v rámci této diplomové práce.

Celá aplikace je stavěna na architektuře MVC (model view controller). View část této architektury je trochu ochuzena, jelikož server komunikuje pouze přes API a pouze ve formátu JSON. Tudíž má každá entita nějaké své DTO (Data Transfer Object), které definuje jak bude daný objekt převeden do JSON formátu. Každý API endpoint má definovanou metodu v jednom z kontrolérů, který daný požadavek zpracovává. Tato metoda v kontroléru obsahuje validaci uživatelského vstupu a pokud je vše v pořádku, tak požadavek předá na nějakou službu (Service), která obsahuje obchodní logiku zpracování požadavku. Pro manipulaci s databází využívají služby tzv. DAO (Data Access Object) objekty, které obsahují SQL dotazy a výsledky dotazů mapují na Java objekty. Takto vrácený Java objekt, se z DAO vrátí do služby a služba tento objekt převede na DTO a ten vrátí kontroléru, který vrátí odpověď uživateli ve formátu JSON.

3.2.2 Databázový model

Databázový model serverové aplikace Uniqway je velice obsáhlý a proto se tato práce soustředí na popsání stávajícího řešení jen pro oblasti, kterými se dále zabývá v návrhu nových mikroslužeb. Nejprve bude popsán databázový model pro rezervace a poté pro obchod s odměnami. Databázový model pro správu textů nebude popsán, protože v současném řešení neexistuje.

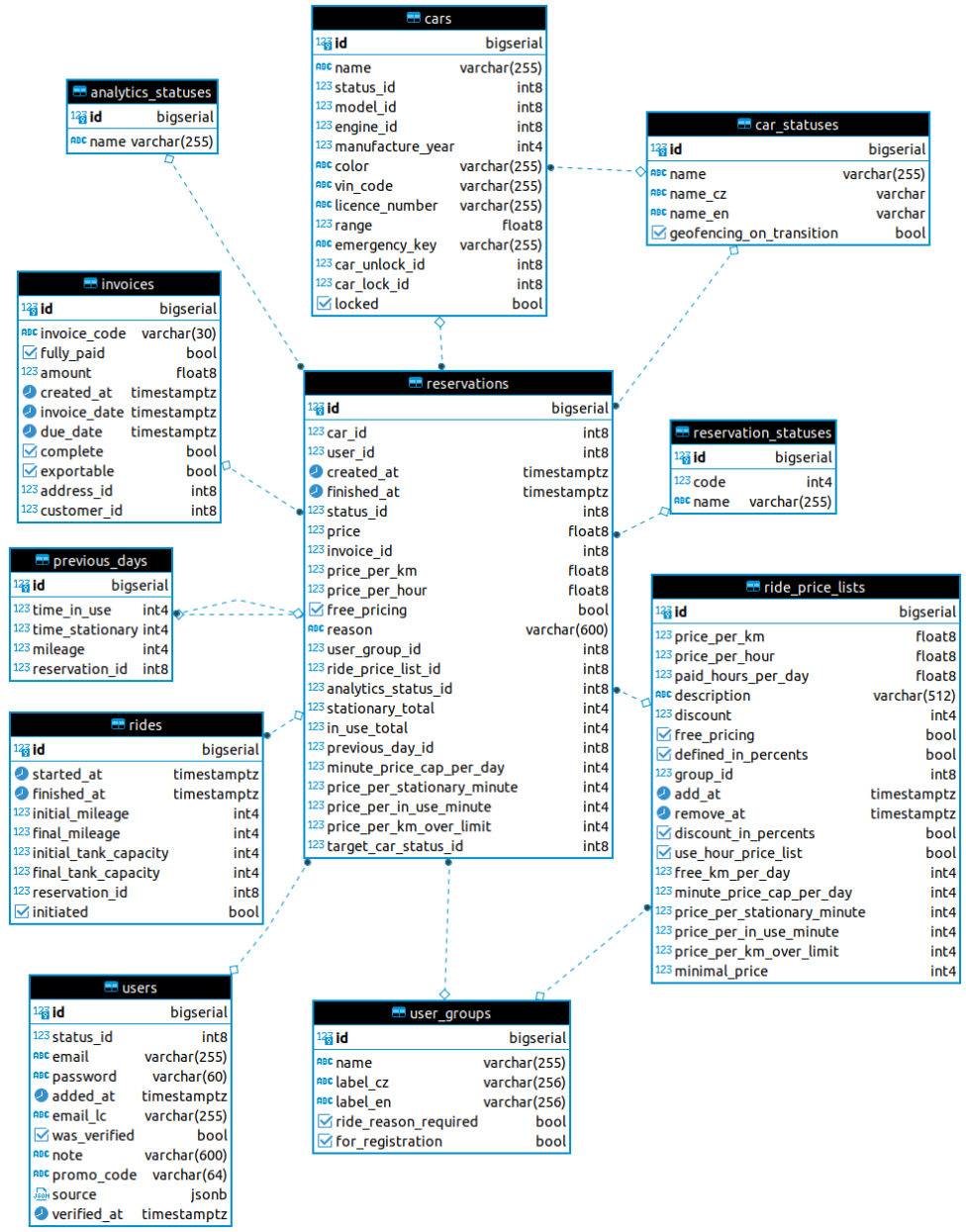
3.2.2.1 Rezervace

V projektu, který se zabývá carsharingem jsou rezervace ústřední entitou. Veškerá hlavní obchodní logika související s půjčováním aut používá entitu Rezervace. Databázový model části systému, který se rezervacemi zabývá je na obrázku 3.2. Výčet tabulek společně se stručným popisem je následující:

rezervace Tabulka (`reservations`) rezervací, která obsahuje důležité údaje o rezervaci. Tyto údaje jsou například uživatel, který rezervaci vytvořil, vozidlo, které je v rámci dané rezervace půjčeno, datum a čas vzniku a konce rezervace, celková cena rezervace, faktura, která bude za rezervaci vystavena, důvod rezervace (u rezervací zdarma), počet minut strávených stáním, počet minut strávených jízdou aj.

jízdy Tabulka (`rides`) jízd, které vznikají v rámci rezervací. Jedná se o 1:1 vazbu s tabulkou rezervací. Jízda obsahuje datum a čas začátku a konce jízdy, počáteční a koncový stav tachometru a nádrže.

uživatelé Tabulka (`users`) uživatelů, která obsahuje základní údaje o uživatelích jako email, heslo v hešované podobě, stav uživatele, poznámka, datum vytvoření aj.



Obrázek 3.2: Databázový model rezervací

uživatelské skupiny Uživatelské skupiny (`user_groups`), které určují které ceníky jsou kterým uživatelům dostupné. V rámci těchto uživatelských skupin si uživatelé rezervují vozidla (při zahájení rezervace volí pod kte-

rou uživ. skupinou chtějí jet) a na konci rezervace se automaticky vybere ten pro uživatele nejvýhodnější (nejlevnější) ceník, který je v rámci této uživ. skupiny dostupný.

ceníky Ceníky (`ride_price_lists`), které definují ceny rezervací a obsahují obdobné informace o ceně jako rezervace, které jsou cena za kilometr, cena za hodinu, pokud je to ceník hodinový, maximální cena, kterou rezervace za den může dosáhnout, cena za minutu stání a minutu jízdy pokud je to ceník minutový, cena za kilometr nad limit, tento limit aj.

stavy rezervací Stavy rezervací (`reservation_statuses`) určují v jakém se rezervace nachází stavu. V současné době existují pouze 2 stavy: vytvořena a ukončena.

stavy aut Stavy aut (`car_statuses`) určují ve kterých stavech se mohou jednotlivá auta nacházet. Každá rezervace určuje v jakém stavu auto po jejím ukončení skončí.

auta Auta (`cars`), která se rezervují v rámci rezervací. Obsahují o sobě základní údaje: jméno, stav, model, motor, rok výroby, SPZ, atd.

faktury Tabulka (`invoices`) slouží k evidenci faktur celého systému. Jedná se jak o faktury za jízdy, tak za faktury z obchodu z odměnami. Faktury, které jsou po skončení rezervace, nebo zaplacení objednávky vytvořeny a zaslány uživateli k proplacení, obsahují údaje jako cena, datum a čas vzniku, fakturační adresu, číslo zákazníka na kterého je fakturováno aj.

předchozí dny Tabulka (`previous_days`) obsahující základní data o předchozím dni rezervace, pokud rezervace trvá více dní. Slouží k výpočtu ceny u delších rezervací.

stavy pro analýzu Tabulka (`analytics_statuses`) obsahující stavy rezervace pro analytické účely. V současném stavu obsahuje pouze stav *technicalError*, kterým se označí rezervace, při které nastala technická chyba.

3.2.2.2 Obchod s odměnami

V systému Uniqway existuje obchod s odměnami, kde si uživatelé za nasbírané body (body se sbírají za každou placenou rezervaci a za utracené peníze) a malý peněžní doplatek mohou koupit různé drobné předměty s tématikou Uniqway. Tento obchod s odměnami je plně funkční eshop s evidencí skladu, objednávek, produktů k prodeji, plateb a sběru bodů. Uložení dat takto rozsáhlé funkcionality je realizováno prostřednictvím množství tabulek, jejichž diagram byl rozdělen na tři menší diagramy. Některé tabulky jsou pro diagramy společné. Diagram části databáze, která obstarává evidenci produktů a informací co s tím souvisí je vidět na obrázku 3.3. Tyto tabulky jsou:

produkty Tabulka (`products`) obsahuje všechny produkty, nabízené v rámci obchodu s odměnami. U každého produktu eviduje jeho kategorii, jméno v českém a anglickém jazyce, popis v českém a anglickém jazyce, cenu v bodech a cenu v korunách, příznak zda-li je produkt viditelný v nabídce, id obrázku, který má být zobrazen jako náhled aj.

kategorie Tabulka (`categories`) eviduje různé kategorie produktů. U každé kategorie je evidován její název v češtině a angličtině a datum a čas jejího vytvoření. V současné době existuje pouze kategorie „Doplňky".

typy položek na faktuře Tabulka (`invoice_item_types`) evidující typ položky na faktuře. Hodnota z této tabulky se projeví jako řádek na faktuře reprezentující daný produkt. Eviduje jednotku v češtině a angličtině a popis v češtině a angličtině.

atributy produktů Tabulka (`product_attributes`) atributů produktů eviduje hodnotu atributu daného typu (např. u objemu to je hodnota objemu, tedy 250 ml). Typ atributu (další tabulka) představuje vlastnost daného produktu (např. materiál), společnou pro všechny jeho varianty (jedna z dalších tabulek).

typy atributů produktů Tabulka (`product_attribute_types`) typy atributů produktů představuje typ atributu (např. materiál, objem apod.). O typech je evidováno jméno v češtině a angličtině a datum a čas vzniku.

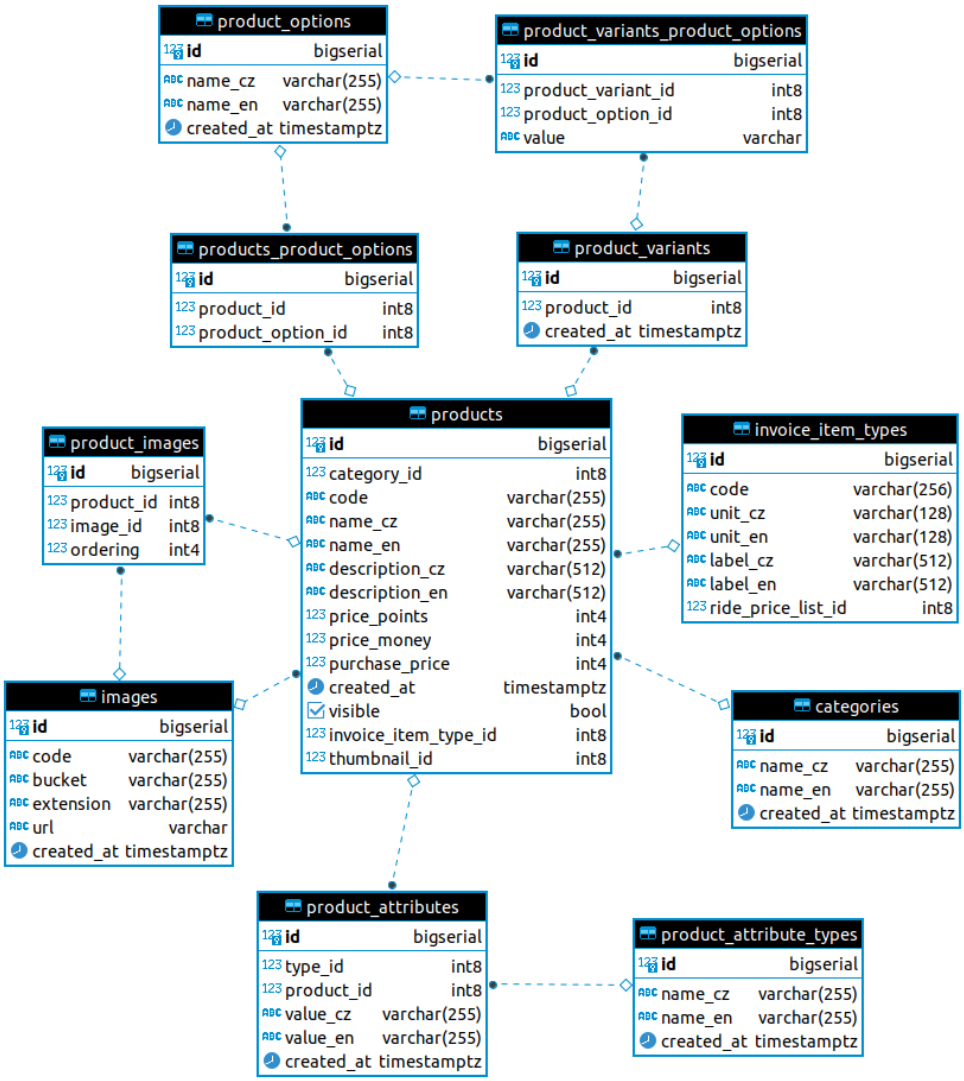
obrázky Tabulka (`images`) slouží na evidenci obrázků. Obrázky jsou uloženy v AWS S3 [21]. O obrázku eviduje kód (jednoznačný), název S3 bucketu, ve kterém jsou obrázky uloženy, příponu obrázku (např. jpg) a URL, ze které se dá obrázek stáhnout.

produktové obrázky Tabulka (`product_images`) je spojovací tabulka mezi produkty a obrázky a slouží k řazení obrázků v detailu produktu. Eviduje id produktu, id obrázku a pořadí v jakém se má daný obrázek zobrazit v detailu produktu.

produktové vlastnosti Tabulka (`product_options`) eviduje různé vlastnosti produktu, které odlišují různé varianty. Touto vlastností může být například barva, nebo velikost (u oblečení).

produktové varianty Tabulka (`product_variants`) eviduje varianty produktů. Tato tabulka reprezentuje hlavní entitu se kterou obchod s odměnami pracuje.

Diagram části databáze, která obstarává evidenci bodů, jejich získávání a informací co s tím souvisí je vidět na obrázku 3.4. Tyto tabulky jsou:



Obrázek 3.3: Databázový model produktů

bodové odměny Tabulka (`rewards`) eviduje bodové odměny, které uživatelé mohou získat za různé aktivity, které jsou popsány v tabulce odměnitelné aktivity. O odměnách je evidován mód expirace (absolutní, nebo relativní), počet dní do expirace (pouze u relativního módu), maximální datum do kdy odměna platí a počet bodů, které uživatel získá touto odměnou.

odměnitelné aktivity pro uživatelské skupiny Tato tabulka slouží k

evidenci, která odměna bude přiřazena za kterou odměnitelnou aktivitu v rámci dané uživatelské skupiny. Tabulka obsahuje id odměnitelné aktivity, id uživatelské skupiny, id odměny, počet, za který je odměna přidělena (např. pokud je odměna za počet ujetých kilometrů a toto je nastaveno na 5, tak uživatel dostane danou odměnu za každých 5 ujetých kilometrů), datum a čas vytvoření a do kdy tato uživatelská skupina získává dané odměny za dané aktivity. Tato tabulka má název `user_groups_rewardable_activities`.

odměnitelné aktivity Tabulka (`rewardable_activites`) popisuje za které aktivity může dostat uživatel odměnu. Tato tabulka obsahuje název aktivity v češtině a angličtině a typ aktivity (*jízda*, *objednávka*, *útrata*, *jiné*).

transakce odměn Tabulka (`reward_transactions`) obsahuje jednotlivé bodové transakce daného zákazníka. Každá transakce je evidována pomocí počtu bodů, kdy byla vytvořena, kdy expiruje (v případě transakcí, které body přidělují), id uživatele, který transakci vytvořil, id zákazníka kterého se body týkají, id odměnitelné aktivity, název transakce česky a anglicky a typ reference (název tabulky), za kterou byla odměna přiřazena a id této reference.

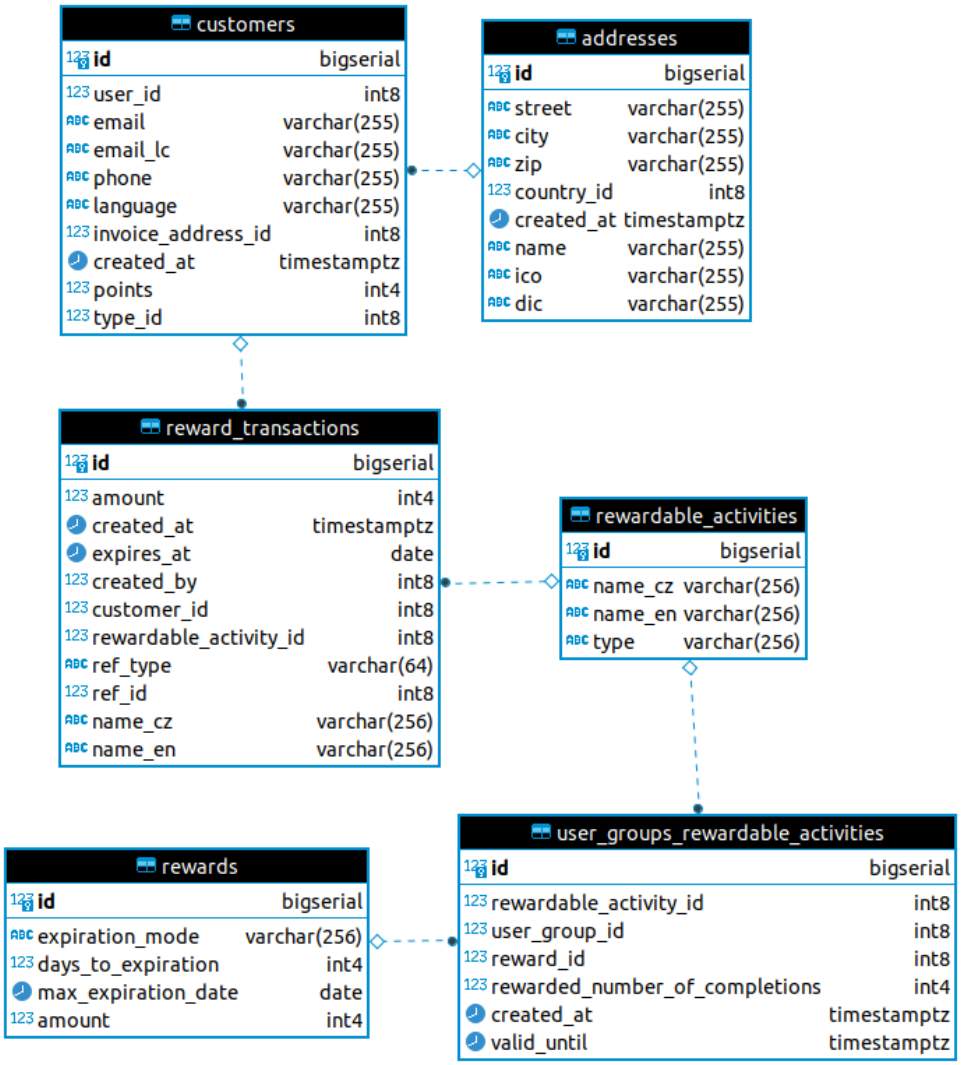
zákazníci Tabulka (`customers`) eviduje jednotlivé zákazníky, kteří mohou sbírat body a nakupovat v obchodu s odměnami. Tato tabulka byla vyčleněna z tabulky uživatelů, aby v obchodě mohli nakupovat i lidé, kteří nejsou uživatelé systému Uniqway.

adresy Tabulka (`addresses`) eviduje adresy v databázi. Tato tabulka je navázána na více tabulek. U návaznosti na tabulku zákazníků a faktur je to fakturační adresa. U návaznosti na výdejní místo je to adresa výdejního místa. U adresy se evidují klasické údaje jako ulice, město, psč, země a jméno.

Diagram části databáze, která obstarává evidenci objednávek, skladového hospodářství a fakturace je vidět na obrázku 3.5. Tabulky sloužící k těmto účelům jsou:

objednávky Tabulka (`orders`) eviduje jednotlivé objednávky v rámci obchodu s odměnami. Každá objednávka eviduje svůj stav, fakturu, zákazníka, který objednávku vytvořil, výdejní místo, cenu v bodech a cenu v korunách, kdy byla vytvořena, kdy byla označena jako připravena k vyzvednutí, kdy byla dokončena a číslo platby.

stavy objednávek Tabulka (`order_states`) evidující různé stavy objednávek. V současné době tyto stavy jsou *vytvořená*, *zaplacená*, *zpracovávaná*, *připravená k vyzvednutí*, *zrušená*, *dokončená* a *zrušená nezaplacená*.



Obrázek 3.4: Databázový model bodů

položky v objednávce Tabulka (`order_items`) je spojovací mezi objednávkou a variantou produktu. Kromě těchto dvou údajů ještě eviduje počet kusů daného produktu v objednávce.

výdejní místa Tabulka (`pickup_places`) eviduje výdejní místa, kam si může zákazník dojít předměty vyzvednout. O výdejním místě je evidováno jméno v češtině i angličtině, popis v češtině i angličtině, adresa, datum a čas vytvoření a příznak jestli je dostupné.

varianty produktů ve výdejních místech Spojovací tabulkou s názvem `pickup_places_product_variants` mezi variantami produktů a výdejními místy je tabulka, která eviduje množství produktových variant v daném výdejním místě.

naskladnění produktů Tabulka (`stock_receipts`) eviduje naskladnění produktových variant na nějaké výdejní místo. U naskladnění se eviduje výdejní místo, uživatel, který naskladnění vytvořil, datum a čas naskladnění a poznámka.

naskladnění variant produktů `stock_receipts_product_variants` je název spojovací tabulka spojující naskladnění produktů a produktové varianty. Eviduje kolik kusů produktových variant bylo v rámci daného naskladnění přidáno na výdejní místo.

vyskladnění produktů Tabulka (`stock_removals`) eviduje vyskladnění produktových variant z výdejních míst. Vyskladnění eviduje objednávku, v rámci které došlo k prodání produktů, výdejní místo, ze kterého byly produkty vyskladněny, uživatel, který vyskladnění udělal (nakoupil), typ vyskladnění (tabulka `stock_removal_types`, v současné době jen *objednávka*, datum a čas vyskladnění a poznámka.

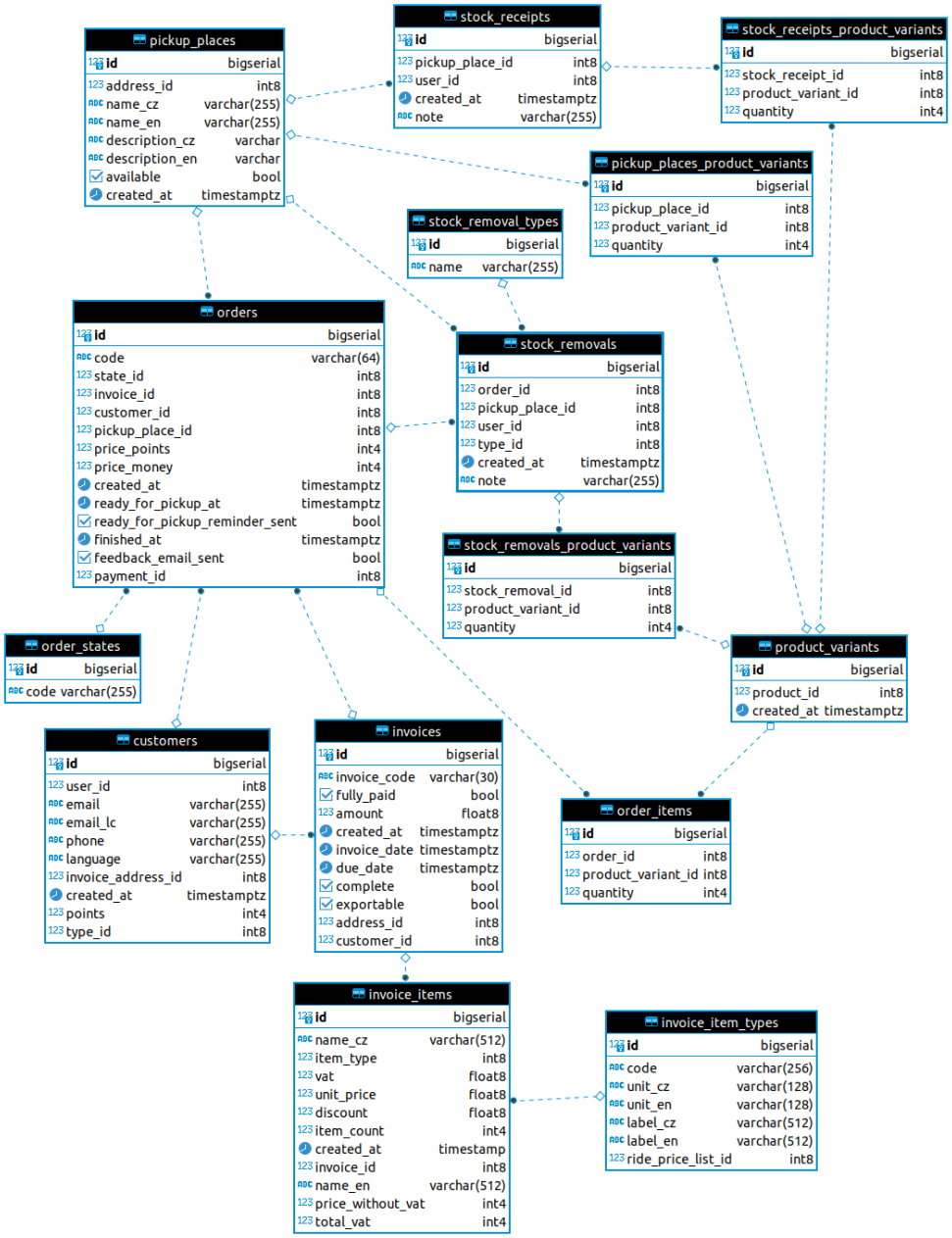
vykladnění variant produktů `stock_removals_product_variants` je spojovací tabulkou mezi vyskladněním a variantami produktů. Eviduje počet kusů vyskladnění dané varianty produktu.

faktury Tabulka (`invoices`) je blíže popsána v předchozí části 3.2.2.1.

položky na fakturách Tabulka (`invoice_items`) obsahuje jednotlivé položky na faktuře. O položce faktury se eviduje jméno v češtině i angličtině, typ položky, DPH, jednotkovou cenu, slevu, počet jednotek dané položky, datum vytvoření, cenu bez DPH a celkové DPH.

3.2.3 API

Serverová aplikace Uniqway komunikuje s ostatními komponentami v rámci Uniqway systému přes REST API. V této části bude toto API blíže popsáno a bude zaměřeno na funkcionalitu tvorby a správy rezervací a obchod s odměnami. API na správu textů v současném řešení neexistuje. API je rozděleno na klientskou část (`/client`), které může volat kterýkoliv uživatel Uniqway a část administrátorskou (`/admin`), kterou můžou volat pouze administrátoři systému Uniqway. Administrátorské API slouží ke správě systému, klientské slouží k běžnému používání služby. Data zasílaná pomocí tohoto API jsou ve formátu JSON.



Obrázek 3.5: Databázový model objednávek

3.2.3.1 Tvorba a správa rezervací

Klient může přes API vytvořit a ukončit rezervaci, zjistit aktuální cenu rezervace, zjistit informace o aktuální rezervaci a zjistit informace o všech svých rezervacích. Jednotlivé požadavky realizující tuto funkcionalitu jsou následující:

tvorba rezervace V těle odpovědi se vrací id právě vzniklé rezervace. V těle požadavku se zasílá id vozidla, které chce uživatel rezervovat, číslo uživatelské skupiny, pod kterou chce rezervaci vytvořit, roli uživatele a důvod rezervace (u rolí, které důvod vyžadují).
Metoda HTTP: POST
URL: `/reservations`
Očekávaná odpověď: 200 OK
Tělo požadavku:

```
{
    "carId": 1,
    "role": "admin",
    "reason": "testing",
    "userGroup": 1
}
```

všechny rezervace V těle odpovědi se vrací pole rezervací zabalené do stránkovacího mechanismu. Každá rezervace má své id, auto, cenu, datum a čas vzniku a ukončení. O autě v rezervaci se eviduje id, jméno a model auta. Model obsahuje mimo jiné ceník, podle kterého je rezervace naceněná.
Metoda HTTP: GET
URL: `/reservations`
Očekávaná odpověď: 200 OK
Tělo odpovědi (zjednodušeno na 1 rezervaci):

```
{
    "data": [
        {
        "id": 314,
        "car": {
            "id": 2,
            "name": "Prezident",
            "model": {
                "id": 1,
                "brand": "Škoda",
                "type": "Fabia",
                "name": "3",
                "seats": 5,
```

```
                    "tankCapacity": 45,
                    "trunkCapacity": 330,
                    "priceList": {
                        "id": 1,
                        "pricePerKm": 4.9,
                        "pricePerHour": 29,
                        "priceTextCz": "{0} Kč/km + {1}/h",
                        "priceTextEn": "{0} CZK/km + {1}/h"
                    }
                }
            },
            "price": 0,
            "createdAt": "2018-09-26 10:23:45",
            "finishedAt": "2018-09-26 10:23:45",
            "note": "Technical issue"
            }
        ],
        "meta": {
            "hasNextPage": true,
            "totalPageCount": 0,
            "totalCount": 0
        }
    }
```


ukončení rezervace V těle odpovědi se vrací id právě ukončené rezervace, celkový čas, celkový počet bodů získaný za rezervaci, celková ujetá vzdálenost, konečná cena, datum a čas vytvoření a ukončení rezervace. V těle požadavku se zasílá id cílového stavu vozidla, do kterého se má vozidlo po ukončení rezervace přepnout.
Metoda HTTP: POST
URL: `/reservations/current/finish`
Očekávaná odpověď: 200 OK
Tělo odpovědi:


```
{
    "reservationId": 316,
    "totalDistance": 32.87,
    "totalTime": 78,
    "totalPoints": 20,
    "finalPrice": 476.55,
    "createdAt": "2018-08-04 23:14:39",
    "finishedAt": "2018-08-05 23:14:39"
}
```


cena rezervace V těle odpovědi se vrací cena rezervace, informace o jízdě

(ujeté kilometry, počet minut parkování, počet minut jízdy) a popis.
Metoda HTTP: GET
URL: `/reservations/current/price`
Očekávaná odpověď: 200 OK
Tělo odpovědi:


```
{
    "description": "Přehled ceny rezervace v tuto chvíli",
    "totalPrice": "4982,32 Kč",
    "rideInformation": {
        "Vzdálenost": "0 km",
        "Parkování": "9 674 min",
        "Jízda": "0 min"
    }
}
```


aktuální rezervace V těle odpovědi se vrací informace o aktuální rezervaci. O rezervaci se vrací id rezervace, auto, které je rezervováno, stav jeho nádrže, jeho pozice, stav zamknutí auta, stav rezervace, cena, datum a čas vytvoření a palivová karta daného vozidla.
Metoda HTTP: GET
URL: `/reservations/current`
Očekávaná odpověď: 200 OK
Tělo odpovědi:


```
{
    "id": 314,
    "car": {
        "id": 12,
        "name": "Blogger",
        "manufactureYear": 2016,
        "color": null,
        "licenceNumber": "6AZ5482",
        "tank": {
        "capacity": 45,
        "litres": 36,
        "percentage": 87,
        "range": 256
        },
        "position": {
        "car": {
            "id": 12,
            "name": "Blogger"
        },
        "lat": 50.102757,
```

```
            "lng": 14.393184
            },
            "locked": true,
            "notResponding": false
        },
        "status": {
            "id": 1,
            "code": 1,
            "name": "created"
        },
        "price": 0,
        "createdAt": "2018-09-26 10:23:45",
        "fuelCard": {
            "id": 5,
            "number": "MW3mB",
            "pin": "created",
            "dailyLimit": 10000,
            "createdAt": "2018-08-05 23:14:39",
            "expiresAt": "2020-06-31",
            "cardVendor": {
            "id": 1,
            "code": "CCS",
            "name": "created"
            }
        }
        }
```


možnosti ukončení rezervace V těle odpovědi se vrací pole možných stavů aut, do kterých je možné auto po ukončení přepnout. Jeden z těchto stavů bude použit po ukončení rezervace (tento stav je označen příznakem `selected: true`. O každém stavu je zobrazeno jeho id, jeho jméno a zdali je zvolený pro přepnutí po ukončení rezervace. Bežný uživatel má na výběr pouze stav *dostupné*.
Metoda HTTP: GET
URL: `/reservations/current/finish-options`
Očekávaná odpověď: 200 OK
Tělo odpovědi (zjednodušeno na 1 stav):


```
[
    {
        "id": 1,
        "name": "Available",
        "selected": true
    }
```

```
]
```

Administrátorské API má za možnost zobrazit informaci o všech rezervacích, o konkrétní rezervaci, vytvořit rezervaci konkrétnímu uživateli, ukončit konkrétní rezervaci a zjistit možnosti ukončení pro konkrétní rezervaci. Tyto požadavky jsou více popsány v příloze.

3.2.3.2 Obchod s odměnami

V klientské části obchodu s odměnami může uživatel získat informace o všech kategoriích produktů, o produktech dané kategorie, o detailu zadaného produktu, o výdejních místech a může také objednávku vytvořit. Detailní popis těchto požadavků je následující:

kategorie produktů V těle odpovědi se vrací pole kategorií produktů. Každá kategorie je reprezentována pomocí id a jména.
Metoda HTTP: GET
URL: `/shop/categories`
Očekávaná odpověď: 200 OK
Tělo odpovědi (zjednodušeno na 1 kategorii):

```
[
    {
        "id": 314,
        "name": "Oblečení"
    }
]
```

produkty v kategorii V těle odpovědi se vrací pole produktů v dané kategorii. O každém produktu je evidováno jeho jméno, popis, cena v korunách a v bodech, atributy produktu, vlastnosti produktu a obrázky produktu.
Metoda HTTP: GET
URL: `/shop/categories/{id}/products`
Očekávaná odpověď: 200 OK
Tělo odpovědi (zjednodušeno na 1 produkt):

```
{
    "data": [
        {
            "id": 1,
            "code": "at8h63",
            "name": "Uniqway tričko",
            "description": "Nejlepší tričko pod sluncem",
            "price": {
                "points": 599,
```

```
                "money": 30
            },
            "attributes": [
                {
                    "type": "Materiál",
                    "value": "Bavlna"
                }
            ],
            "options": [
                {
                    "id": 2,
                    "name": "Velikost",
                    "values": [
                        "S",
                        "M",
                        "L"
                    ]
                }
            ],
            "images": {
                "thumbnail": {
                    "url": "https://i.picsum.photos/
                    id/325/400/400.jpg",
                    "ordering": 1
                },
                "gallery": [
                    {
                        "url":
                        "https://i.picsum.photos/
                        id/325/400/400.jpg",
                        "ordering": 1
                    }
                ]
            }
        }
    ],
    "meta": {
        "hasNextPage": true,
        "totalPageCount": 0,
        "totalCount": 0
    }
}
```


detail produktu V těle odpovědi se vrací detail daného produktu. Obsah je

stejný jako v případě všech produktů dané kategorie.
Metoda HTTP: GET
URL: `/shop/products/{id}`
Očekávaná odpověď: 200 OK

výdejní místa V těle odpovědi se vrací pole všech výdejních míst. O každém výdejním místě je evidována jeho adresa, název, jméno a popis.
Metoda HTTP: GET
URL: `/shop/pickup-places`
Očekávaná odpověď: 200 OK
Tělo odpovědi (zjednodušeno na 1 výdejní místo):


```
[
    {
        "id": 314,
        "address": {
            "id": 2,
            "type": {
                "id": 2,
                "name": "Correspondence"
            },
            "street": "Ulice",
            "city": "Mesto",
            "zip": "12345",
            "country": {
                "id": 1,
                "name": "Slovakia",
                "code": "SK"
            },
            "name": "CompanyName s.r.o",
            "createdAt": "2018-08-04 23:14:39",
            "ico": "002327546",
            "dic": "CZ00042354"
        },
        "name": "Kancelář ČVUT Dejvice",
        "description": "Kancelář na Fakultě strojní
                        1. patro místnost 123"
    }
]
```


vytvoření objednávky V těle požadavku se zasílá id výdejního místa a pole variant produktů. U variant produktů se uvádí jejich množství a různé vlastnosti.
Metoda HTTP: POST
URL: `/shop/orders`

Očekávaná odpověď: 200 OK
Tělo požadavku (zjednodušeno na 1 produkt):

```
[
    {
        "pickupPlaceId": 1,
        "productVariants": [
            {
                "productId": 10,
                "quantity": 3,
                "options": [
                    {
                        "id": 2,
                        "value": "red"
                    }
                ]
            }
        ]
    }
]
```

Administrátorské rozhraní obsahuje možnosti jak spravovat produktu, produktové kategorie, obrázky apod. Toto rozhraní je mnohem rozsáhlejší než klientské a obsahuje velké množství různých endpointů. Z tohoto důvodu jsou tyto endpointy popsány v příloze a celou API dokumentaci je možné najít v elektronické příloze.

3.2.4 Správa textů a lokalizací

Správa textů a jich jazykových verzí je v současné verzi roztroušena do několika míst. Jedním z nich je využití funkcionality na správu textů v Play Frameworku - Play Messages [22]. Toto řešení je jen částečně dostačující a pokud je potřeba text změnit, tak je nutné nasadit novou verzi systému. Toto je velká nevýhoda a bylo by dobré jí odstranit. Dalším způsobem jakým jsou v serverové aplikaci spravovány texty je pomocí záznamu v SQL databázi. Tam kde je text potřeba v obou jazycích (čeština i angličtina), tak jsou přidány dva sloupce do dané tabulky. Jeden pro české a druhý pro anglické znění. Toto opět není úplně ideální řešení, protože jsou texty roztroušeny po celé databázi a členové týmu co texty spravují k nim mají obtížný přístup.

3.2.5 Silná a slabá místa

Jako každý systém a aplikace, tak i serverová aplikace Uniqway má svá silná a slabá místa, která jsou blíže popsána v následující části práce.

3.2.5.1 Silná místa

Mezi hlavní silná místa se řadí členění aplikace do jednotlivých části podle MVC návrhového modelu. Kontroléry přijímají požadavky, zpracování předávají na služby, které pomocí DAO vrací data z databáze zpět k uživateli v JSON formátu. Dalším silným místem je ve většině případů správný databázový návrh, který zamezuje nekonzistencím a duplicitám v datech. API je také silné místo serverové aplikace, jelikož splňuje konvence HTTP a REST API.

3.2.5.2 Slabá místa

Hlavním slabým místem je komplexita monolitické aplikace, která je velice těžko upravitelná a těžko se předělávají existující funkcionality. Dalším slabým místem je správa textů, kdy ve většině případů je nutné pro změnu textů vydat novou verzi aplikace, jelikož se hojně využívají již zmíněné *Play Messages*. Nejednotnost umístění textů je také slabé místo, které velmi znesnadňuje práci členům týmu, kteří texty spravují.

Kapitola 4

Analýza a návrh přechodu architektur

V této kapitole bude provedena analýza a návrh přechodu z monolitické architektury do architektury mikroslužeb. Tato kapitola se bude zaměřovat na analýzu a návrh mikroslužeb obsluhujících:

- správu a lokalizaci textů,
- obchod s odměnami a
- tvorbu a správu rezervací.

Jako strategie přechodu mezi architekturami bylo, po dohodě s týmem, zvoleno využití kombinace všech tří strategií uvedených v kapitole 2.4. Nejvíce bude využita strategie 1 - nová funkcionalita v nové mikroslužbě a strategie 3 - postupné extrahování funkcionalit z monolitu do mikroslužeb. Tento proces bude iterativní, tedy v první fázi se z monolitické aplikace oddělí několik mikroslužeb a v dalších iteracích budou další mikroslužby postupně vyčleňovány z monolitu a podle potřeby dále děleny na menší a menší.

Komunikace klientských aplikací se serverovou aplikací byla navržena jako komunikace přes API Gateway (2.3.1.2). K vybrání API Gateway došlo z několika důvodů:

- V systému existuje několik různých klientských aplikací (více v sekci 3.1) a spravovat přímou komunikaci klient-mikroslužba by bylo zbytečně náročné.
- Při vytváření nových mikroslužeb není potřeba klientské aplikace měnit, pokud se zachová vnější API, tedy API které bude poskytovat API Gateway klientským aplikacím.

- Autentizaci nebude muset řešit každá mikroslužba zvlášť, ale bude se o to starat API Gateway a tím dojde ke zjednodušení samotných mikroslužeb.

Jako API Gateway bude sloužit existující monolitická aplikace, která bude provádět autentizaci požadavků a požadavky na funkcionality, které již byli implementovány v mikroslužbách, bude přesměrovávat na tyto mikroslužby.

Komunikace mezi mikroslužbami bude probíhat pomocí REST API a one-to-one požadavek/odpověď (více v sekci 2.3.2.1). Tento typ řešení byl vybrán kvůli následujícím důvodům:

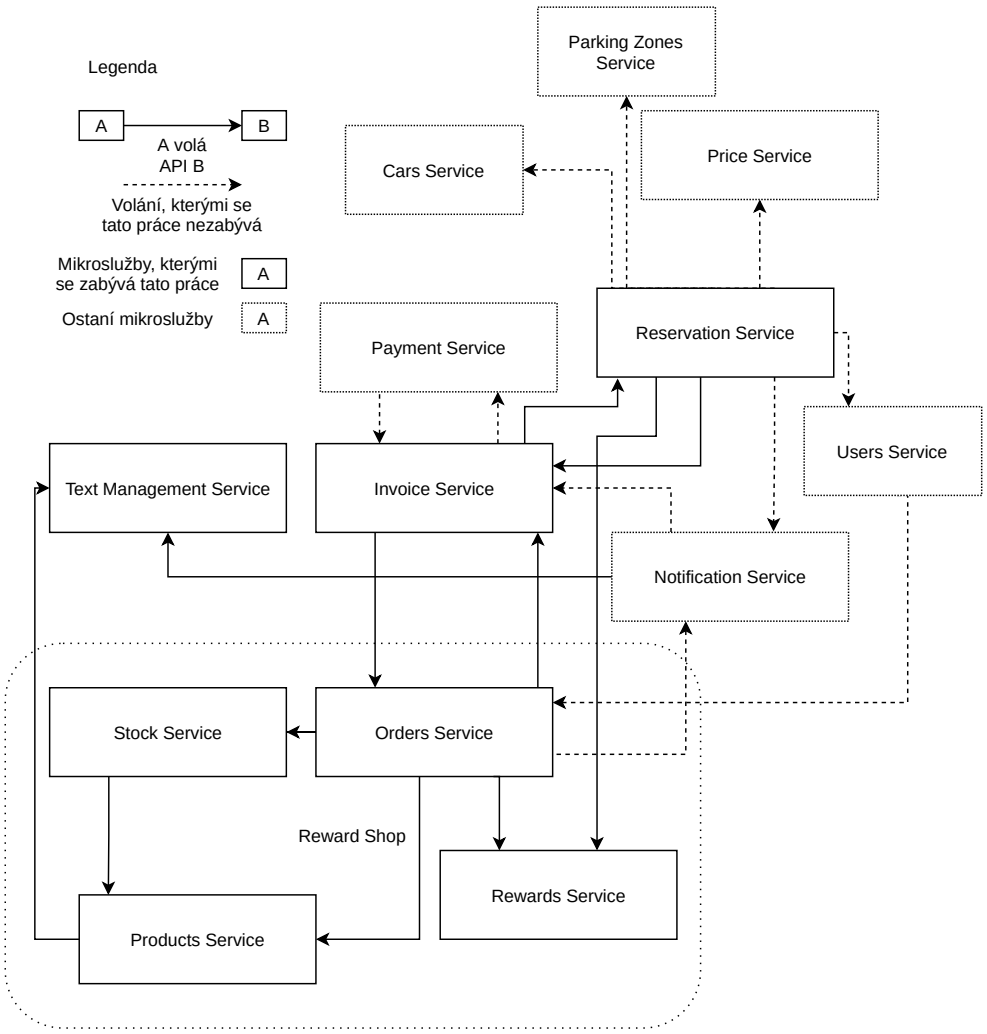
- Uniqway systém není tak velký, aby potřeboval složitější a komplexnější řešení typu publikování/odebírání.
- Není třeba spravovat systém, který spravuje zasílání zpráv mezi mikroslužbami v případě použití řešení publikování/odebírání.
- One-to-one komunikace přes REST API je jednoduché řešení, které plně vyhovuje nárokům Uniqway systému.

Pokud se Uniqway systém rozroste natolik, že nebude moci využívat toto navržené řešení, tedy rozroste se počet uživatelů natolik, že toto řešení bude velmi pomalé a přímá komunikace one-to-one bude systém zahlcovat, tak bude nutné komunikaci předělat na publikování/odebírání (více v sekci 2.3.2.2). Tato změna bude vyžadovat vytvoření systému na správu zasílání zpráv mezi mikroslužbami (tzv. message brokera), který bude mezi mikroslužbami zprávy předávat a u každé mikroslužby bude potřeba přidat komunikaci s tímto brokerem. Tato změna mikroslužeb by neměla být příliš náročná, ale v současném stavu Uniqway systému je zbytečná a neměla by být potřeba během několika následujících let s ohledem na velikost Uniqway systému (počet uživatelů a aut) a dosavadní vývoj této velikosti.

Objevování mikroslužeb bude realizováno pomocí objevování na straně serveru (viz sekce 2.3.3.2). Objevování na straně serveru bylo zvoleno z důvodu udržení jednotlivých mikroslužeb co nejjednodušších. Dalším důvodem k vybrání tohoto typu objevování mikroslužeb je skutečnost, že Uniqway systém běží na AWS (Amazon Web Services [23]) a využívá ECS (Elastic Container Service [24]) k nasazování současné aplikace. ECS poskytuje Service Discovery [25], které k tomuto účelu slouží. Tento Service Discovery přiřadí každé mikroslužbě DNS záznam (např. `text-management.uniqway.cz`), pod kterým se daná mikroslužba nachází. Tato komponenta automaticky dělá kontrolu funkčnosti (health checks), tím že volá nějaký zvolený endpoint mikroslužby a když mikroslužba neodpoví úspěchem několikrát v řadě, tak danou instanci vyřadí.

Návrh mikroslužeb a komunikace mezi nimi popisuje diagram z obrázku 4.1. Tečkovaně jsou v diagramu znázorněny mikroslužby a čárkovaně jejich

volání, které nejsou součástí zaměření této práce. V následujících sekcích bude vždy ukázán detail tohoto diagramu a bude více vysvětlen.



Obrázek 4.1: Návrh komunikace mezi mikroslužbami

Funkční požadavky definují požadavky na funkcionalitu dané komponenty systému [26]. U každé mikroslužby budou uvedeny tyto funkční požadavky, protože se pro každou mikroslužbu liší. Nefunkční požadavky definují požadavky na ostatní vlastnosti dané komponenty systému - jak se má systém chovat, jaké jsou jeho omezení atd. [26]. Nefunkční požadavky na každou mikroslužbu jsou následující:

N1 REST API Mikroslužba bude poskytovat svá data a funkcionalitu skrze

REST API rozhraní.

4.1 Šablona mikroslužeb

Všechny mikroslužby využívající stejnou technologii sdílí určitou část funkcionalit. Tato funkcionalita je sdílena skrze šablonu mikroslužeb. Toto je v oblasti mikroslužeb častá praxe a nazývá se *Microservice chassis* (šablona mikroslužeb) [27]. Takováto šablona slouží k implementaci funkcionalit, které jsou potřeba ve všech mikroslužbách a může se jednat o následující:

- Konfigurace - většina mikroslužeb vyžaduje obdobnou konfiguraci a tato konfigurace také může obsahovat přihlašovací údaje do externích služeb jako jsou databáze.
- Logování (zaznamenávání výstupů a chybových hlášek) - konfigurace logovacího frameworku jako je např. log4j nebo logback.
- Health checks (kontroly funkčnosti) - endpoint, který každá mikroslužba poskytuje a na kterém lze zjistit základní funkčnost (je služba dostupná, odpovídá na dotazy atd.). [27]

Tato šablona byla navržena společně s ostatními členy týmu pro vznikající mikroslužby a slouží k rychlejšímu vytvoření nové mikroslužby a sdílení společných funkcionalit napříč systémem mikroslužeb. Součástí této šablony je:

- základní konfigurace potřebná ve všech mikroslužbách,
- endpoint pro health check,
- systém pro logování pomocí logback,
- zpracování výjimek a jejich chybových hlášek,
- způsob připojení k databázi,
- základní testy sdílených funkcionalit,
- nastavení generátoru z OpenAPI a
- nastavení generátoru modelů z databáze.

4.2 Správa textů a lokalizace

Správa textů a lokalizace je nově vznikající funkcionalita, kterou současná verze serverové aplikace podporuje jen částečně (viz kapitola 3.2.4). Mikroslužba spravující texty a jejich lokalizace bude jednotné místo, kde budou všechny texty, které by se mohly zobrazit uživateli na jednom místě. Tato

mikroslužba bude vystavovat REST API, přes které bude možné texty upravovat, prohlížet, vytvářet a mazat. Jelikož se jedná o novou funkcionalitu, tak návrh této mikroslužby obsahuje i seznam případů užití a jejich realizaci funkčních požadavků. U ostatních mikroslužeb jsou tyto případy užití vynechány.

4.2.1 Analýza funkčních požadavků

V této části budou popsány funkční požadavky na mikroslužbu spravující texty a jejich lokalizace do více jazyků. Tyto požadavky jsou následující:

F1 Zobrazení všech textů Mikroslužba bude poskytovat zobrazení všech uložených textů. Takto získané texty budou obsahovat následující informace:

1. kód textu - jednoznačný identifikátor každého textu,
2. textovou hodnotu v českém jazyce,
3. textovou hodnotu v anglickém jazyce a
4. popis textu, který specifikuje co text obsahuje a čím budou nahrazeny zástupné symboly, pokud v textu nějaké jsou.

F2 Zobrazení konkrétního textu v konkrétním jazyce Podle zadaného kódu vrátí text, který tomuto kódu přísluší. V požadavku bude přijímat jazyk textu, který má vrátit a vrátí pouze textovou hodnotu daného textu podle zadaného jazyka. Texty získané pouze v konkrétním jazyce budou obsahovat následující informace:

1. kód textu a
2. textovou hodnotu v daném jazyce.

F3 Zobrazení konkrétního textu v obou jazycích Mikroslužba podle zadaného kódu vrátí text v obou jazycích.

F4 Upravení konkrétního textu Mikroslužba bude poskytovat rozhraní pro úpravu textu zadaného podle jeho kódu. Tímto způsobem půjde měnit jak česká, tak anglická verze daného textu a popis textu.

F5 Vytvoření konkrétního textu Mikroslužba bude umožňovat přidání nového textu. Nový text bude mít vždy unikátní kód a textovou hodnotu v češtině a v angličtině.

F6 Smazání konkrétního textu Texty bude možné podle kódu smazat.

F7 Zobrazení více textů podle kódů Bude možné zadat více kódů různých textů a mikroslužba vrátí jejich textové hodnoty ve zvoleném jazyce.

4.2.2 Popis případů užití

Případy užití (use case) je slovní popis kroků nějakých událostí, které vedou k realizaci daného úkolu (splnění funkčního požadavku) [28]. U každého případu užití musí být vyspecifikováni aktéři, které se na realizaci úkolu podílí. Pro mikroslužbu obsluhující správu textů a lokalizací to jsou dva aktéři:

- Klient - mikroslužba, která zasílá požadavky na tuto mikroslužbu
- Systém - tato mikroslužba obsluhující správu textů odpovídající na požadavky od klientů

Případy užití pro realizaci funkční požadavků ze sekce 4.2.1 jsou následující:

UC1 Vytvoření nového textu V tomto případu užití přidá klient nový text do databáze systému a ten mu vrátí identifikátor daného textu. Tento případ užití pokrývá funkční požadavek F5 (4.2.1).

Postup:

1. Klient zašle požadavek na vytvoření nového textu a předá systému všechny požadované informace. Očekává se, že u textu uvede jeho jednoznačný identifikátor, znění v českém jazyce, znění v anglickém jazyce a popis textu, který specifikuje co text obsahuje a čím budou nahrazeny zástupné symboly, pokud v textu nějaké jsou.
2. Systém zpracuje požadavek, vytvoří text v databázi a klientovi vrátí identifikátor vytvořeného textu.

UC2 Editace existujícího textu Klient chce změnit znění nějakého existujícího textu. Tento případ užití pokrývá funkční požadavek F4.

Postup:

1. Klient zašle požadavek systému na úpravu stávajícího textu. V požadavku uvede jednoznačný identifikátor textu, jeho nové znění v obou jazycích a popis textu.
2. Systém zpracuje požadavek, upraví text a vrátí potvrzení, že se operace zdařila.

UC3 Získání všech textů Klient chce získat informace o všech uložených textech. Tento případ užití pokrývá funkční požadavek F1.

Postup:

1. Klient zašle požadavek na získání všech textů.
2. Systém na požadavek odpoví polem všech existujících textů.

UC4 Získání všech textů podle zadaných indentifikátorů Klient chce získat všechny texty podle zadaných identifikátorů. Tento případ užití pokrývá funkční požadavek F7.

Postup:

1. Klient zašle požadavek na získání textů podle poslaných identifikátorů. Těchto identifikátorů může být 1 a více.
2. Systém odpoví na požadavek zasláním pole textů, které odpovídají zaslaných identifikátorům.

UC5 Získání zadaného textu v daném jazyce Klient chce získat pouze jeden text podle zadaného identifikátoru a jeho hodnotu pouze v zadaném jazyce. Tento případ užití pokrývá funkční požadavek F2.

Postup:

1. Klient zašle požadavek na systém. V tomto požadavku zadá jednoznačný identifikátor textu, který chce získat a jazyk ve kterém chce reprezentaci textu zaslat.
2. Systém na požadavek odpoví textovou hodnotou zadaného textu v zadaném jazyce.

UC6 Získání zadaného textu Klient chce získat detail daného textu v českém i anglickém jazyce. Tento případ užití pokrývá funkční požadavek F3.

Postup:

1. Klient zašle požadavek na získání detailu zadaného textu v obou jazycích.
2. Systém odpoví na požadavek zasláním detailu textu v obou jazycích. Tato reprezentace by měla obsahovat: jednoznačný identifikátor, text v českém jazyce, text v anglickém jazyce a popis textu.

UC7 Smazání zadaného textu Klient chce smazat zadaný text podle jednoznačného identifikátoru. Tento případ užití pokrývá funkční požadavek F6.

Postup:

1. Klient zašle požadavek na smazání textu podle jednoznačného identifikátoru.
2. Systém smaže text a potvrdí klientovi smazání.

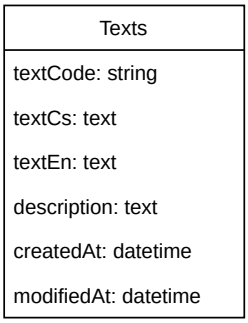
Tabulka 4.1 ukazuje realizace funkčních požadavků případy užití.

Tabulka 4.1: Realizace funkčních požadavků případy užití u správy textů

	UC1	UC2	UC3	UC4	UC5	UC6	UC7
F1			X				
F2					X		
F3						X	
F4		X					
F5	X						
F6							X
F7				X			

4.2.3 Doménový model

V této části budou popsány všechny entity se kterými bude tato mikroslužba pracovat. U každé entity budou popsány všechny parametry včetně jejich datového typu, které entita bude mít. Grafický návrh tohoto doménového modelu se nachází na obrázku 4.2.



Obrázek 4.2: Doménový model mikroslužby spravující texty a lokalizace

4.2.3.1 Text (Texts)

Tato entita je nejdůležitější a zároveň jedinou entitou v tomto doménovém modelu. Texty obsahují následující atributy:

textCode jednoznačný identifikátor daného textu, datový typ: string

textCs hodnota daného textu v českém jazyce, datový typ: text

textEn hodnota daného textu v anglickém jazyce, datový typ: text

description popis daného textu, datový typ: text

createdAt datum a čas vytvoření daného textu, datový typ: datetime

modifiedAt datum a čas poslední modifikace daného textu, datový typ: datetime

4.2.4 Návrh API rozhraní

V této části práce bude popsán návrh API rozhraní přes které bude mikroslužba komunikovat se zbytkem systému. Data přes toto API se zasílají ve formátu JSON. Kompletní popis API rozhraní pro mikroslužbu spravující texty je v příloze.

4.2.4.1 Získání všech textů

V těle odpovědi se vrací pole textů, a to jejich kód (jednoznačný identifikátor), hodnota v českém jazyce, hodnota v anglickém jazyce, popis, datum a čas vzniku a poslední modifikace. Tento požadavek na server splňuje funkční požadavek F1 (4.2.1).
Metoda HTTP: GET
URL: `/texts`
Očekávaná odpověď: 200 OK
Tělo odpovědi (zjednodušeno na 1 text):

```
[
    {
        "textCS": "Entita {0} s nazvem {1} nebyla nalezena.",
        "textEN": "Entity {0} with name {1} was not found.",
        "description": "Text, ktery se zobrazi pri vyjimce
                EntityNotFoundException. {0} bude nahrazen
                jmenem entity (napr. User) a {1} bude nahrazen
                hodnotou (napr. Adam)",
        "textCode": "exceptions.entityNotFoundException",
        "createdAt": "2021-03-19 11:00:00",
        "modifiedAt": "2021-03-19 11:00:00"
    }
]
```

4.2.4.2 Vytvoření nového textu

Tento požadavek slouží na vytvoření nového textu. V těle požadavku se zasílá text, a to jeho kód (jednoznačný identifikátor), hodnotu v českém jazyce, hodnotu v anglickém jazyce a popis. Tento požadavek na server splňuje funkční požadavek F5.
Metoda HTTP: POST
URL: `/texts`
Očekávaná odpověď: 200 OK
Tělo požadavku:

```
{
    "textCS": "Entita {0} s nazvem {1} nebyla nalezena.",
    "textEN": "Entity {0} with name {1} was not found.",
    "description": "Text, ktery se zobrazi pri vyjimce
            EntityNotFoundException. {0} bude nahrazen
            jmenem entity (napr. User) a {1} bude nahrazen
            hodnotou (napr. Adam)",
    "textCode": "exceptions.entityNotFoundException"
}
```

4.2.4.3 Smazání textu

Tento požadavek slouží ke smazání daného textu. V cestě se uvádí kód textu, který má být smazán. Tento požadavek na server splňuje funkční požadavek F6.
Metoda HTTP: DELETE
URL: `/texts/by-code/{textCode}`
Očekávaná odpověď: 200 OK

4.2.4.4 Upravení daného textu

V těle požadavku se zasílá kód textu, který má být změněn a jeho nová hodnota v českém a anglickém jazyce. Tento požadavek na server splňuje funkční požadavek F4.
Metoda HTTP: PUT
URL: `/texts/by-code/{textCode}`
Očekávaná odpověď: `200 OK`
Tělo odpovědi:

```
{
    "textCS": "Entita {0} s nazvem {1} nebyla nalezena.",
    "textEN": "Entity {0} with name {1} was not found.",
    "description": "Text, ktery se zobrazi pri vyjimce
            EntityNotFoundException. {0} bude nahrazen jmenem
            entity (napr. User) a {1} bude nahrazen
            hodnotou (napr. Adam)"
}
```

4.2.4.5 Získání textu v daném jazyce

V těle odpovědi se vrací text podle zvoleného jazyka, a to jeho kód (jednoznačný identifikátor) a hodnota ve zvoleném jazyce. Jazyk se volí pomocí hlavičky `Accept-Language`. Tento požadavek na server splňuje funkční požadavek F2.

Metoda HTTP: GET
URL: `/texts/by-code/{textCode}`
Očekávaná odpověď: `200 OK`
Hlavička: `Accept-Language`
Tělo odpovědi:

```
{
    "text": "Entita {0} s nazvem {1} nebyla nalezena.",
    "textCode": "exceptions.entityNotFoundException"
}
```

4.2.4.6 Získání detailu daného textu

V těle odpovědi se vrací detail textu podle zvoleného identifikátoru. Tento detail obsahuje kód textu, jeho hodnotu v českém a anglickém jazyce, jeho popis a datum a čas vzniku a poslední modifikace. Tento požadavek na server splňuje funkční požadavek F3.
Metoda HTTP: GET
URL: `/texts/by-code/{textCode}/detailed`
Očekávaná odpověď: `200 OK`
Tělo odpovědi:

```
{
    "textCS": "Entita {0} s nazvem {1} nebyla nalezena.",
    "textEN": "Entity {0} with name {1} was not found.",
    "description": "Text, ktery se zobrazi pri vyjimce
            EntityNotFoundException. {0} bude nahrazen
            jmenem entity (napr. User) a {1} bude nahrazen
            hodnotou (napr. Adam)",
    "textCode": "exceptions.entityNotFoundException",
    "createdAt": "2021-03-19 11:00:00",
    "modifiedAt": "2021-03-19 11:00:00"
}
```

4.2.4.7 Získání textů podle zadaných kódů v daném jazyce

V těle odpovědi se vrací pole textů podle zvoleného jazyka a zadaných kódů. Tyto kódy se specifikují jako query parametry. Každý text se vrací ve formě jeho kódu (jednoznačný identifikátor) a hodnoty ve zvoleném jazyce. Jazyk se volí pomocí hlavičky `Accept-Language`. Tento požadavek na server splňuje funkční požadavek F7.
Metoda HTTP: GET
URL: `/texts/by-codes?textCodes=code1&textCodes=code2`
Očekávaná odpověď: `200 OK`

Hlavička: `Accept-Language`
Tělo odpovědi:

```
[
    {
        "textCode": "code1",
        "text": "Entity {0} with name {1} was not found."
    },
    {
        "textCode": "code2",
        "text": "Entity {0} with name {1} already exists."
    }
]
```

4.3 Fakturace

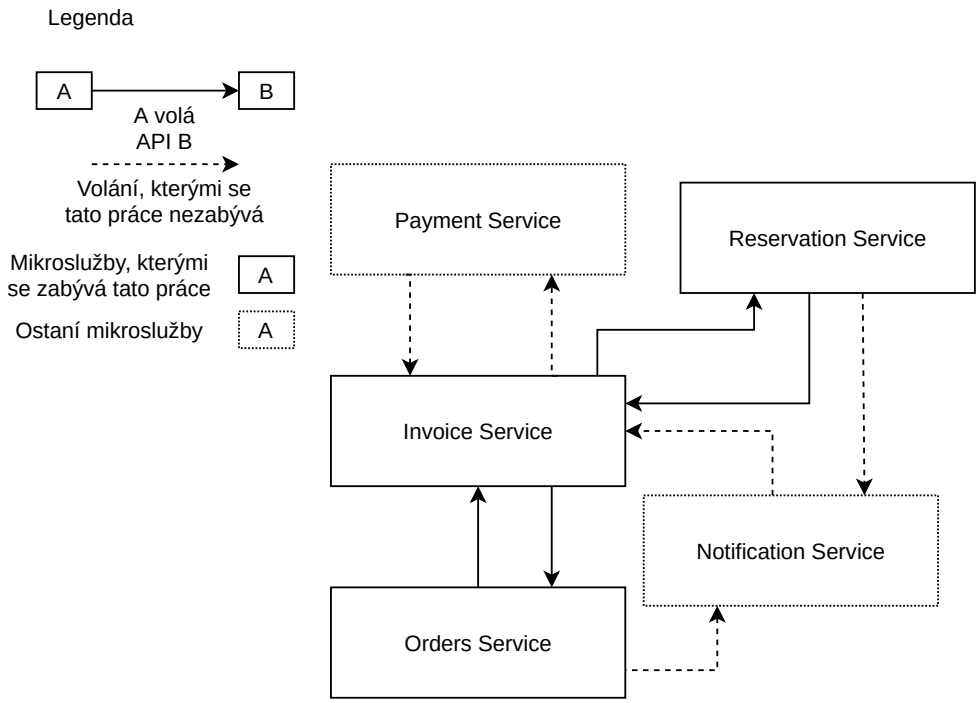
Součástí práce bude mikroslužba zabývající se fakturací. Implementace této mikroslužby je nezbytná jak pro obchod s odměnami (viz sekce 4.4), tak pro tvorbu a správu rezervací (viz sekce 4.5) a proto byla mikroslužba spravující fakturace vyhodnocena jako velmi důležitá a je zařazena před realizaci těchto dvou zmíněných funkcionalit.

Fakturační mikroslužba bude přijímat požadavky na vytvoření faktury. Tento požadavek v současném stavu může přijít buď po ukončení rezervace a nebo po ukončení objednávky v obchodu s odměnami. V rámci požadavku na vytvoření přijme id entity (rezervace nebo objednávky) a poté se dané mikroslužby doptá na detaily. Po vytvoření faktury vrátí odpověď mikroslužbě, která fakturu žádala s tím, že jí předá informace o vzniklé faktuře (její ID, aby si fakturu mohla jiná mikroslužba najít, pokud jí bude potřebovat). Mikroslužba na fakturace zašle po vytvoření faktury požadavek k zaplacení faktury na mikroslužbu spravující platby. Po zaplacení mikroslužba spravující platby zašle požadavek na tuto mikroslužbu, aby danou fakturu označila jako zaplacenou. Když chce mikroslužba (rezervační, nebo spravující objednávky) poslat email s fakturou ve formátu PDF v příloze, tak kontaktuje notifikační mikroslužbu s tím, že jí předá ID faktury. Tato notifikační mikroslužba se poté zeptá mikroslužby na správu faktur, aby jí zaslala danou fakturu ve formátu PDF (viz obrázek 4.3).

4.3.1 Analýza funkčních požadavků

V této části budou popsány funkční požadavky na mikroslužbu spravující faktury. Tyto požadavky jsou následující:

F1 Zobrazení všech faktur Mikroslužba bude poskytovat rozhraní na získání všech existujících faktur. Takto získané informace o každé faktuře budou následující:



Obrázek 4.3: Návrh komunikace s fakturační mikroslužbou

- id faktury,
- kód faktury, jednoznačný pro každou fakturu, slouží pro účetní účely,
- příznak, jestli byla faktura plně zaplacena,
- příznak, jestli je faktura kompletní (získala všechny potřebné informace),
- částka v českých korunách,
- datum a čas vytvoření,
- datum vystavení,
- datum splatnosti,
- příznak, jestli je faktura exportovatelná (jestli slouží k účetním účelům),
- id entity, ze které bude faktura vytvořená,
- typ této entity (objednávka nebo rezervace) a
- údaje o uživateli - id, jméno, příjmení, email, adresa atd.

F2 Zobrazení konkrétní faktury Mikroslužba bude poskytovat zobrazení konkrétní faktury a to stejné informace jako v požadavku F1.

F3 Zobrazení položek faktury Mikroslužba bude poskytovat rozhraní pro zobrazení položek dané faktury. U každé položky je evidováno jméno, typ, DPH, cena za jednotku, počet jednotek, cena za jednotku bez DPH, celková cena bez DPH, celková cena, sleva a datum vytvoření.

F4 Získání faktury ve formátu PDF Služba bude poskytovat rozhraní pro získání faktury ve formátu PDF.

F5 Vytvoření faktury Mikroslužba bude poskytovat rozhraní pro vytvoření nové faktury. Toto vytvoření bude obsahovat údaj jestli se jedná o faktury za rezervaci a nebo fakturu za objednávku v obchodě s odměnami a kde lze najít dodatečné informace potřebné k fakturaci.

F6 Smazání faktury Mikroslužba bude poskytovat rozhraní pro smazání zadané faktury.

F7 Úprava faktury Mikroslužba bude poskytovat rozhraní pro úpravu zadané faktury.

F8 Zobrazení všech typů položek Mikroslužba bude poskytovat rozhraní pro zobrazení všech typů položek na fakturách.

F9 Přidání typu položky Mikroslužba bude poskytovat rozhraní pro přidání nového typu položek na fakturách.

F10 Úprava typu položky Mikroslužba bude poskytovat rozhraní pro úpravu typu položek na fakturách.

F11 Odstranění typu položky Mikroslužba bude poskytovat rozhraní pro smazání typu položek na fakturách.

F12 Přidání zákazníka Mikroslužba bude poskytovat rozhraní pro přidání nového zákazníka a to včetně jeho adresy.

F13 Odebrání zákazníka Mikroslužba bude poskytovat rozhraní pro odebrání zákazníka.

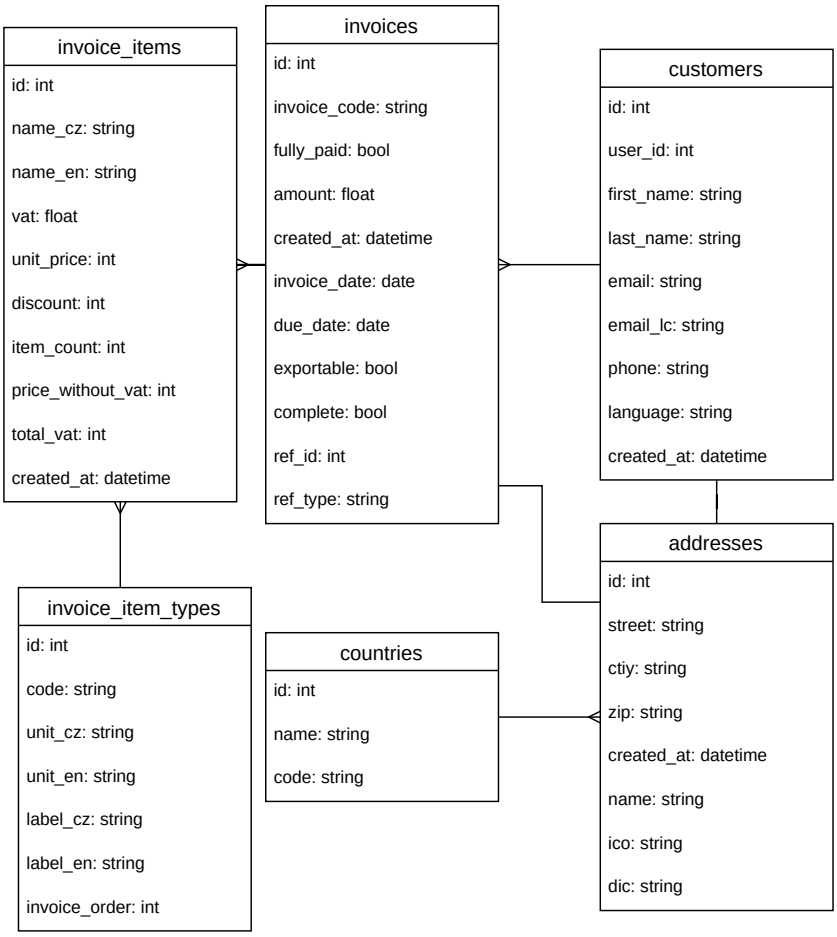
F14 Úprava zákazníka Mikroslužba bude poskytovat rozhraní pro úpravu zákazníka a to včetně jeho adresy.

F15 Zobrazení všech zákazníků Mikroslužba bude poskytovat rozhraní pro zobrazení všech zákazníků.

F16 Zobrazení zákazníka Mikroslužba bude poskytovat rozhraní pro zobrazení konkrétního zákazníka.

4.3.2 Doménový model

V této části budou popsány všechny entity se kterými bude tato mikroslužba pracovat. U každé entity budou popsány všechny parametry včetně jejich datového typu, které entita bude mít. Grafický návrh tohoto doménového modelu se nachází na obrázku 4.4.



Obrázek 4.4: Doménový model mikroslužby spravující fakturace

4.3.2.1 Faktury

Tato entita uchovává údaje o fakturách. Faktury obsahují následující atributy:

invoice_code jednoznačný identifikátor dané faktury, slouží pro evidenci v externím systému na správu účetnictví, datový typ: string

fully_paid příznak, jestli je faktura plně zaplacená, datový typ: bool

amount obnos na faktuře v Kč, datový typ: float

created_at datum a čas vytvoření faktury v databázi, datový typ: datetime

invoice_date datum vytvoření faktury, datový typ: date

due_date datum splatnosti faktury, datový typ: date

exportable příznak jestli je faktura exportovatelná do externího systému na správu faktur, datový typ: bool

complete příznak jestli je faktura kompletní, tedy jestli získala všechny potřebné informace k vytvoření faktury, datový typ: bool

ref_id id entity, ze které se tato faktura vytváří, datový typ: int

ref_type typ entity, ze které se tato faktura vytváří, aktuálně pouze „objednávka“ nebo „rezervace“, datový typ: bool

4.3.2.2 Položky faktur

Tato entita uchovává údaje o položkách faktury. Jedná se o jednotlivé řádky, které se na faktuře objeví. Položky faktur obsahují následující atributy:

name_cz jméno položky faktury v českém jazyce, název se bere z typu položky a doplní se k němu další údaje, zobrazí se na faktuře, datový typ: string

name_en jméno položky faktury v anglickém jazyce, název se bere z typu položky, zobrazí se na faktuře, datový typ: string

vat hodnota DPH této položky v procentech, datový typ: float

unit_price jednotková cena této položky v haléřích, datový typ: int

discount sleva na této položce v haléřích, datový typ: int

item_count počet položek, datový typ: int

created_at datum a čas vytvoření této položky, datový typ: datetime

price_without_vat cena této položky bez DPH v haléřích, datový typ: int

total_vat celková hodnota DPH této položky v haléřích, datový typ: string

4.3.2.3 Typy položek

Tato entita uchovává údaje o typech položek faktur. Jedná se o informace jednotlivých položek faktur, které mají společnou jednotku a název. Typy položek faktur obsahují následující atributy:

code jedinečný kód typu, podle kterého jsou přiřazovány názvy a jednotky k jednotlivým položkám, datový typ: string

unit_cz jméno jednotky v českém jazyce, ze kterého se tvoří jméno jednotky položky, datový typ: string

unit_en jméno jednotky v anglickém jazyce, ze kterého se tvoří jméno jednotky položky, datový typ: string

label_cz název typu položky v českém jazyce, ze kterého se tvoří jméno položky, datový typ: string

label_en název typu položky v anglickém jazyce, ze kterého se tvoří jméno položky, datový typ: string

invoice_order pořadí v jakém se položka tohoto typu objeví na faktuře, datový typ: int

4.3.2.4 Adresy

Entita uchovává adresy. Obsahuje základní údaje o adrese jako je ulice, město, PSČ, jméno zákazníka, který má danou adresu jako fakturační a pokud je to právnická osoba, tak i IČO a DIČ.

4.3.2.5 Země

Entita obsahuje jednotlivé země, které se nachází na adresách. Každá země má název a kód. Kód je dvoupísmenná zkratka země, pro ČR je to například CZ.

4.3.2.6 Zákazníci

Entita sloužící k evidenci zákazníků. Každý zákazník má své id uživatele (pokud je i uživatel), jméno, příjmení, email, email malými písmeny, telefonní číslo, jazyk, fakturační adresu a datum a čas kdy byl vytvořený.

4.3.3 Návrh API rozhraní

V této části práce bude popsán návrh API rozhraní přes které bude mikroslužba spravující fakturace komunikovat se zbytkem systému. Data přes toto API se zasílají ve formátu JSON. V této části bude uvedeno pouze několik základních endpointů. Kompletní popis API rozhraní pro tuto mikroslužbu je v příloze.

4.3.3.1 Získání všech faktur

V těle odpovědi se vrací pole faktur a to jejich id, kód (jednoznačný identifikátor), id entity, ze které se tvoří faktura, typ entity ze které se tvoří faktura, fakturační adresa s jejími atributy, celková cena faktury, datum splatnosti, datum vystavení, příznak jestli je faktura zaplacená, jestli je exportovatelná, jestli je kompletní, datum a čas vytvoření a zákazník na koho je faktura vytvořená včetně jeho atributů. Zobrazení jedné faktury je totožné, jen se v URL uvádí i její ID. Tento požadavek na server splňuje funkční požadavek F1 (4.3.1).
Metoda HTTP: GET
URL: `/invoices`
Očekávaná odpověď: 200 OK
Tělo odpovědi (zjednodušeno na 1 fakturu):


```
[
    {
        "id": 1,
        "refId": 25,
        "refType": "reservation",
        "address": {
            "street": "Ulice 1",
            "city": "Praha",
            "zip": "18000",
            "country": "CZ",
            "name": "Josef Novak",
            "ico": "12345678",
            "dic": "CZ12345678"
        },
        "amount": 456.5,
        "dueDate": "2021-04-15",
        "exportable": true,
        "complete": true,
        "fullyPaid": false,
        "invoiceCode": "4136305049",
        "invoiceDate": "2021-04-15",
        "customer": {
            "id": 1,
            "lastName": "Novak",
            "firstName": "Josef",
            "userId": 11,
            "email": "josef.novak@email.cz",
            "phone": "123456789",
            "language": "cs",
            "address": {
                "street": "Ulice 1",
```

```
                "city": "Praha",
                "zip": "18000",
                "country": "CZ",
                "name": "Josef Novak",
                "ico": "12345678",
                "dic": "CZ12345678"
            }
        },
        "createdAt": "2021-04-15 00:00:00"
    }
]
```

4.3.3.2 Vytvoření faktury

V těle požadavku se zásílá ID entity, ze které se má faktura vytvořit, a typ této entity. Typ entity může v současné době nabývat pouze hodnot *order*, pro faktury tvořené z objednávky v obchodě s odměnami a *reservation*, pro faktury tvořené z rezervací vozidel. Tyto údaje slouží mikroslužbě na správu faktur, aby se dané mikroslužby doptala na konkrétní informace, které k vytvoření faktury potřebuje. Požadavek na server na vytvoření faktury splňuje funkční požadavek F5.
Metoda HTTP: POST
URL: `/invoices`
Očekávaná odpověď: 200 OK
Tělo požadavku:

```
{
    "refId": 25,
    "refType": "reservation"
}
```

4.3.3.3 Získání faktury ve formátu PDF

Systém zasílá faktury uživatelům jako přílohu v emailu ve formátu PDF. Pro tyto účely potřebuje Notifikační mikroslužba (která se mimo jiné stará o posílání emailů uživatelům) získat fakturu ve formátu PDF. Odpověď se vrátí ve formátu `application/pdf`. Tento požadavek na server splňuje funkční požadavek F4.
Metoda HTTP: GET
URL: `/invoices/{id}/pdf`
Očekávaná odpověď: 200 OK

4.3.3.4 Vytvoření typu položky faktury

V těle požadavku se zasílají informace o novém typu položky na faktuře a to jeho kód, jednotka v českém a anglickém jazyce, popis v českém a anglickém jazyce a ID produktů, nebo ceníků u kterých se mají tyto typy používat. Stejné informace o typu vrací endpoint na získání kódu a stejné informace se poskytují endpointu na úpravu typu položky faktury. Tento požadavek na server splňuje funkční požadavek F9.
Metoda HTTP: POST
URL: `//invoice-item-types`
Očekávaná odpověď: 200 OK
Tělo požadavku:

```
{
    "code": "distance",
    "unitCz": "km",
    "unitEn": "km",
    "labelCz": "Ujetá vzdálenost",
    "labelEn": "Driven distance",
    "invoiceOrder": 2
}
```

4.4 Obchod s odměnami

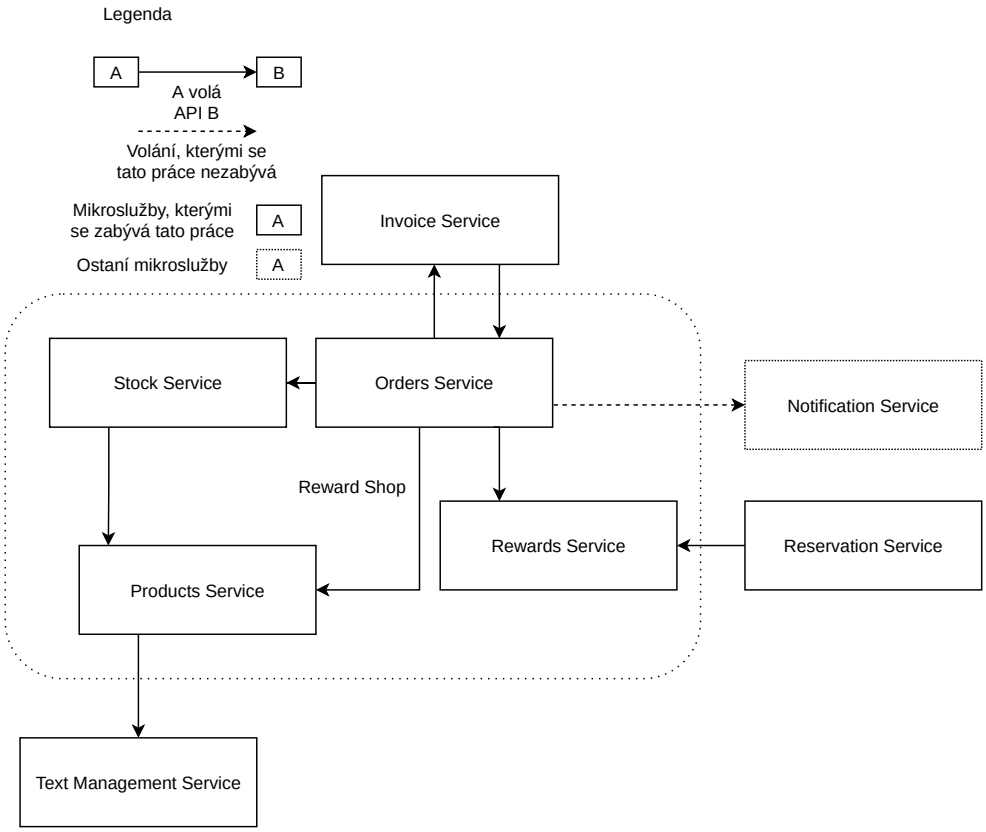
Funkcionalita obchodu s odměnami je velmi rozsáhlá a obsahuje v sobě tvorbu a správu objednávek, správu produktů, správu skladového hospodářství a správu bodových odměn uživatelů. Z důvodu této rozsáhlosti nebude obchod s odměnami realizován jednou, ale několika mikroslužbami. V návrhu vzniklém v rámci této práce se počítá se čtyřmi mikroslužbami jak je vidět na obrázku 4.5. Tyto mikroslužby jsou:

produkty Mikroslužba spravující všechny produkty, jejich ceny, popisy atd.

objednávky Mikroslužba spravující objednávky v rámci obchodu s odměnami. Eviduje který zákazník vytvořil jakou objednávku, posílá data o ní fakturační mikroslužbě, ptá se mikroslužby spravující sklad, jestli jsou produkty skladem, odměnové mikroslužby jestli má zákazník dost bodů atd.

skladové hospodářství Mikroslužba spravující skladové hospodářství, která eviduje kolik kusů které produktové varianty je na kterém výdejním místě a jednotlivá naskladnění a vyskladnění.

odměny Mikroslužba spravující odměny zákazníkům, která eviduje vytváření odměn po ukončené rezervaci a eviduje odečítání odměn po vytvoření objednávky apod.



Obrázek 4.5: Návrh komunikace mikroslužeb v rámci obchodu s odměnami

V následujících částech bude popsán detailněji návrh každé ze zmíněných mikroslužeb.

4.4.1 Mikroslužba spravující produkty

Mikroslužba spravující produkty slouží k evidenci produktů, jejich jednotlivých variant, možností, kategorií, obrázků a atributů. V této sekci bude blíže popsán návrh, jak by tato mikroslužba měla vypadat.

Mikroslužba spravující produkty komunikuje s mikroslužbou spravující texty, od které získává aktuální znění textů. Pokud textová mikroslužba neodpovídá, tak použije texty, které má u sebe uložené. Této mikroslužby se doptává mikroslužba na správu objednávek, kde získá více informací o produktech a mikroslužba spravující sklad, která získá informace o všech existujících produktech (viz obrázek 4.5).

4.4.1.1 Analýza funkčních požadavků

V této části budou popsány funkční požadavky na mikroslužbu spravující produkty. Funkční požadavky jsou následující:

F1 Správa produktů Mikroslužba bude poskytovat možnosti správy produktů, a to jejich vytvoření, editaci, smazání a zobrazování. Zobrazit půjde všechny produkty, všechny produkty dané kategorie, nebo jen jeden konkrétní. U vytvoření produktu půjde definovat všechny jeho parametry a to včetně atributů, variant a možností. U editace půjde měnit do které kategorie patří, jaké má obrázky, jejich pořadí, který obrázek je náhledový, jaké jsou jeho varianty, vlastnosti a jaké jsou jeho atributy.

F2 Správa kategorií produktů Mikroslužba bude poskytovat rozhraní pro správu kategorií produktů, a to jejich vytvoření, editaci, smazání a zobrazování. Kategorie musí být nejprve vytvořená, než půjde přiřadit nějakému produktu (nejde vytvořit při tvorbě produktu). Úprava kategorie spočívá ve změně jejího jména.

F3 Správa obrázků Mikroslužba bude poskytovat rozhraní na úpravu produktových obrázků. Takto půjde nahrát nový obrázek, či smazat existující. Produktům lze přiřadit pouze obrázky, které jsou takto do systému nahrané.

4.4.1.2 Doménový model

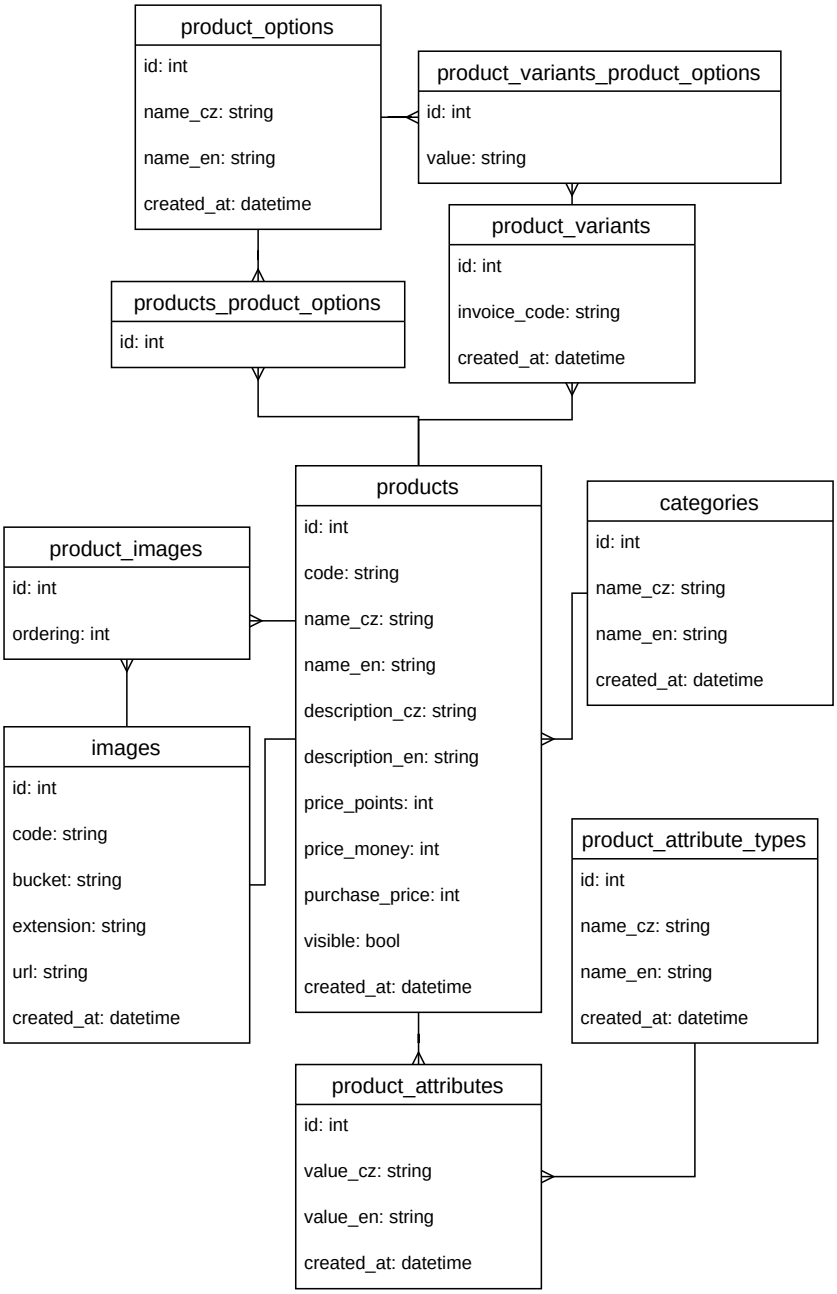
Doménový model produktů bude téměř stejný jako v současném stavu, jelikož je v současném stavu plně vyhovující. Jediné změny, které jsou v doménovém modelu udělány jsou následující:

- Do tabulky produktových variant přibude jednoznačný kód, který slouží ke spárování s typem položky na faktuře, aby se k dané produktové variantě přiřadila správná položka faktury.
- Z původního diagramu je odebrána tabulka `invoice_item_types`, která bude součástí mikroslužby spravující faktury. Návrh doménového modelu je vidět na obrázku 4.6.

Více informací o těchto tabulkách a jejich atributech je v kapitole o současném řešení 3.2.2.2.

4.4.1.3 Návrh API rozhraní

V této části bude popsán návrh API rozhraní pro mikroslužbu spravující produkty v obchodu s odměnami. Návrh tohoto rozhraní je následující:



Obrázek 4.6: Návrh doménového modelu pro mikroslužbu spravující produkty

Získání všech produktů V těle odpovědi se vrací pole produktů. O každém produktu jsou zaslány tyto informace: název produktu česky a anglicky, popis česky a anglicky, kategorie, cena (jak v bodech tak i v korunách),

jeho varianty, atributy, vlastnosti a jeho obrázky. Jako query parametr půjde určit i kategorie produktů, která slouží jako filtr a vrátí pouze produkty z dané kategorie. Tento požadavek na server částečně splňuje funkční požadavek F1 (4.4.1.1).
Metoda HTTP: GET
URL: `/products`
nebo: `/products?category={categoryId}`

Získání konkrétního produktu V těle odpovědi se vrátí informace o konkrétním produktu. Tento požadavek na server částečně splňuje funkční požadavek F1.
Metoda HTTP: GET
URL: `/products/{id}`

Vytvoření nového produktu V těle požadavku se zašlou informace o produktu, který se má vytvořit. Takto zasílané informace jsou stejné jako v případě získání informací o produktu, ale nebude umožňovat rovnou přidat obrázky. Obrázky se budou přidávat přes jiný endpoint, který slouží k nahrání a přidání obrázku konkrétnímu produktu. Tento požadavek na server částečně splňuje funkční požadavek F1.
Metoda HTTP: POST
URL: `/products`

Upravení produktu V těle požadavku se zašlou změněné informace o produktu. Takto umožní měnit všechny parametry produktů, kromě obrázků. Obrázky se budou měnit a přiřazovat v jiném endpointu. Tento požadavek na server částečně splňuje funkční požadavek F1.
Metoda HTTP: PUT
URL: `/products/{id}`

Smazání produktu V URL se zašle id produktu, který má být smazán. Tento požadavek na server částečně splňuje funkční požadavek F1.
Metoda HTTP: DELETE
URL: `/products/{id}`

Vytvoření kategorie produktu V těle požadavku se zašlou údaje o kategorii, která má být vytvořena. Tento požadavek na server částečně splňuje funkční požadavek F1.
Metoda HTTP: POST
URL: `/products/categories`

Smazání kategorie V URL se uvede id kategorie, která má být smazána. Tento požadavek na server částečně splňuje funkční požadavek F2.
Metoda HTTP: DELETE
URL: `/products/categories/{id}`

Úpravení kategorie V těle požadavku se zašlou informace o změněné kategorii. Tento požadavek na server částečně splňuje funkční požadavek F2.
Metoda HTTP: PUT
URL: `/products/categories/{id}`

Zobrazení všech kategorií produktů V těle odpovědi se vrátí pole všech kategorií. Tento požadavek na server částečně splňuje funkční požadavek F2.
Metoda HTTP: GET
URL: `/products/categories`

Nahrání obrázku V těle požadavku se zašle obrázek, který se má uložit k danému produktu a také informace v jakém pořadí se má zobrazovat a zda-li je to náhledový obrázek. Tento požadavek na server částečně splňuje funkční požadavek F2.
Metoda HTTP: POST
URL: `/products/{id}/images`

Úprava obrázku V těle požadavku se zašlou změněné informace o obrázku, tedy změna pořadí nebo zda-li je náhledový. Tento požadavek na server částečně splňuje funkční požadavek F3.
Metoda HTTP: PUT
URL: `/products/{id}/images/{id}`

Smazání obrázku V URL se zašle id obrázku, který se má smazat. Tento požadavek na server částečně splňuje funkční požadavek F3.
Metoda HTTP: DELETE
URL: `/products/{id}/images/{id}`

4.4.2 Mikroslužba spravující objednávky

Mikroslužba spravující objednávky (`Orders Service`) se bude zabývat evidencí objednávek zákazníků v rámci obchodu s odměnami.

Mikroslužba spravující objednávky komunikuje s velkým počtem dalších mikroslužeb. Ptá se produktové mikroslužby jaké existují produkty, poté se ptá mikroslužby spravující sklad jestli je daný produkt skladem na zvoleném výdejním místě. Při tvorbě objednávky si u mikroslužby spravující bodové odměny ověří, jestli má daný uživatel dostatek bodů. Po vytvoření objednávky zašle požadavek na vytvoření faktury k fakturační mikroslužbě. Ta se jí doptá na další informace potřebné k vytvoření faktury. Po získání faktury pošle požadavek na notifikační mikroslužbu, která zašle uživateli informaci o vytvořené objednávce a faktuře. Tato komunikace je vidět na obrázku 4.5.

4.4.2.1 Analýza funkčních požadavků

V této části budou popsány funkční požadavky na mikroslužbu spravující objednávky. Tyto požadavky jsou následující:

F1 Zobrazení všech objednávek Mikroslužba bude poskytovat možnost zobrazit si všechny objednávky.

F2 Zobrazení objednávky Mikroslužba bude poskytovat možnost zobrazit si konkrétní objednávku a to včetně jejích položek a stavu.

F3 Zobrazení objednávky daného zákazníka Mikroslužba bude poskytovat možnost zobrazit si všechny objednávky daného zákazníka.

F4 Vytvoření objednávky Mikroslužba bude poskytovat možnost vytvořit novou objednávku.

F5 Úprava objednávky Mikroslužba bude poskytovat možnost upravit objednávku a to konkrétně změnu jejího stavu. Pokud chce uživatel změnit celou objednávku musí jí zrušit a vytvořit novou.

F6 Zrušení objednávky Mikroslužba bude poskytovat možnost zrušit nedokončenou objednávku.

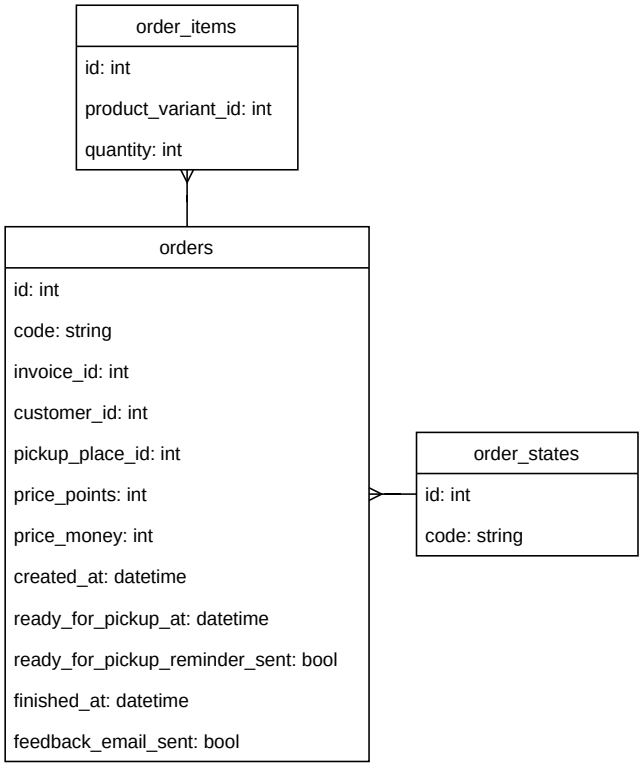
4.4.2.2 Doménový model

Doménový model bude velmi podobný tomu, co je v současném řešení. Tento model bude relativně jednoduchý, jelikož ostatní tabulky budou součástí jiných mikroslužeb. Návrh tohoto modelu je na obrázku 4.7. Bližší popis jednotlivých tabulek je v sekci 3.2.2.2, která se věnuje současnému stavu serverové aplikace.

4.4.2.3 Návrh API rozhraní

Návrh API rozhraní pro mikroslužbu spravující objednávky je následující:

Zobrazení všech objednávek Metoda HTTP: GET
URL: `/orders`



Obrázek 4.7: Návrh doménového modelu pro mikroslužbu spravující objednávky

Zobrazení objednávky Metoda HTTP: GET
URL: `/orders/{id}`

Zobrazení objednávky daného zákazníka Metoda HTTP: GET
URL: `/orders?customer={customerId}`

Vytvoření objednávky Metoda HTTP: POST
URL: `/orders`

Úprava objednávky Metoda HTTP: PUT
URL: `/orders/{id}`

4.4.3 Mikroslužba spravující skladové hospodářství

Mikroslužba spravující skladové hospodářství (`Stock Service`) se bude starat o evidenci počtu kusů jednotlivých produktových variant na jednotlivých výdejních místech.

Tato mikroslužba komunikuje s produktovou mikroslužbou, které se ptá na to, které všechny produkty a následně produktové varianty existují. U těchto variant poté udržuje záznamy o tom, kolik jich je skladem v jednotlivých výdejních místech. Mikroslužba spravující objednávky si poté u této mikroslužby ověřuje zda-li je daná produktová varianta skladem ve vybraném výdejním místě. Tato komunikace je zachycena na obrázku 4.5.

4.4.3.1 Analýza funkčních požadavků

V této části budou popsány funkční požadavky na mikroslužbu spravující skladové hospodářství. Tyto požadavky jsou následující:

F1 Správa výdejních míst Mikroslužba bude poskytovat rozhraní ke správě výdejních míst a to jejich vytvoření, editaci, smazání a zobrazení. Zobrazení výdejního místa bude umožňovat zobrazit seznam výdejních míst, detail výdejního místa včetně jeho skladových zásob a skladové zásoby všech výdejních míst dohromady.

F2 Správa naskladnění Mikroslužba bude poskytovat možnost správy naskladnění. Umožní vytvořit naskladnění, smazání, editaci a zobrazování. Zobrazit půjde seznam všech naskladnění, všech naskladnění konkrétního produktu a konkrétního naskladnění.

F3 Správa vyskladnění Mikroslužba bude poskytovat možnost správy vyskladnění. Umožní vytvořit vyskladnění, smazání, editaci a zobrazování. Zobrazit půjde seznam všech vyskladnění, všech vyskladnění konkrétního produktu a konkrétního vyskladnění.

4.4.3.2 Doménový model

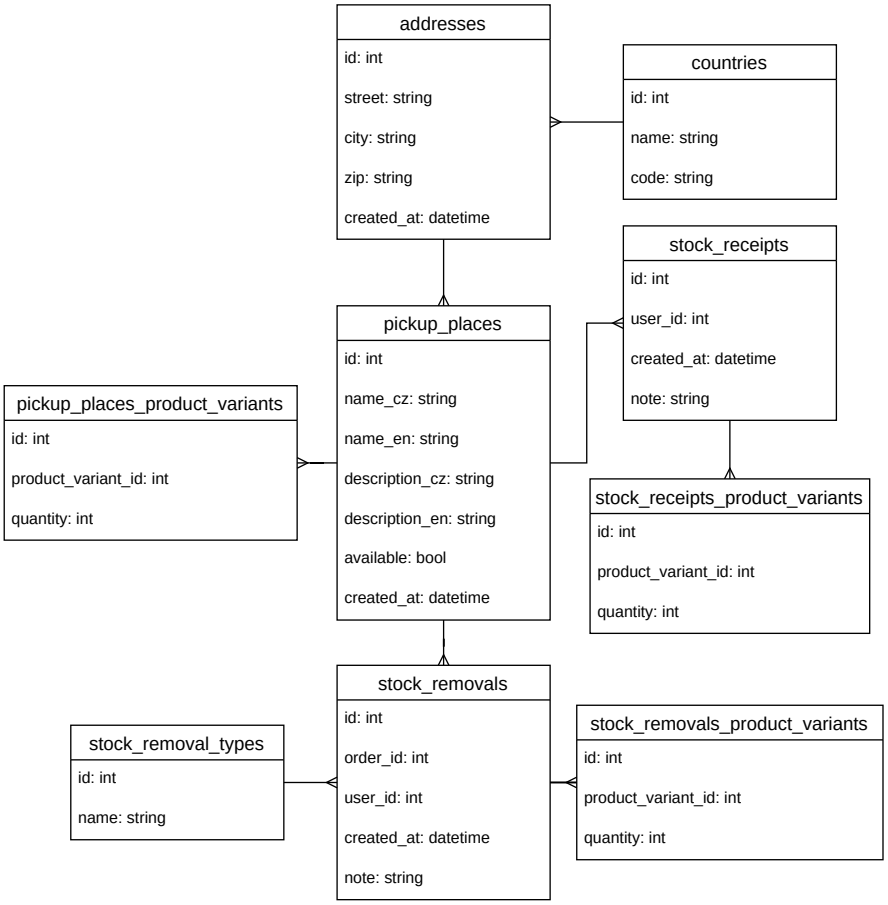
Doménový model pro skladové hospodářství vychází ze současného řešení a zachovává z něj víceméně vše co se skladového hospodářství týká. Více informací o jednotlivých tabulkách a jejich atributech je v sekci 3.2.2.2. Diagram doménového modelu je na obrázku 4.8.

4.4.3.3 Návrh API rozhraní

Návrh API rozhraní pro mikroslužbu spravující skladové hospodářství je následující:

Zobrazení všech výdejních míst Metoda HTTP: GET
URL: `/pickup-places`

Zobrazení výdejního místa Metoda HTTP: GET
URL: `/pickup-places/{id}`



Obrázek 4.8: Návrh doménového modelu pro mikroslužbu spravující sklad

Vytvoření výdejního místa Metoda HTTP: POST
URL: `/pickup-places`

Úprava výdejního místa Metoda HTTP: PUT
URL: `/pickup-places/{id}`

Smazání výdejního místa Metoda HTTP: DELETE
URL: `/pickup-places/{id}`

Zobrazení všech naskladnění Metoda HTTP: GET
URL: `/stock-receipts`

Zobrazení všech naskladnění produktu Metoda HTTP: GET
URL: `/stock-receipts?product-id={productId}`
nebo: `/stock-receipts?product-variant-id={productVariantId}`

Zobrazení konkrétního naskladnění Metoda HTTP: GET
URL: `/stock-receipts/{id}`

Vytvoření naskladnění Metoda HTTP: POST
URL: `/stock-receipts`

Úprava naskladnění Metoda HTTP: PUT
URL: `/stock-receipts/{id}`

Smazání naskladnění Metoda HTTP: DELETE
URL: `/stock-receipts/{id}`

Zobrazení všech vyskladnění Metoda HTTP: GET
URL: `/stock-removals`

Zobrazení všech vyskladnění produktu Metoda HTTP: GET
URL: `/stock-removals?product-id={productId}`
nebo: `/stock-removals?product-variant-id={productVariantId}`

Zobrazení konkrétního vyskladnění Metoda HTTP: GET
URL: `/stock-removals/{id}`

Vytvoření vyskladnění Metoda HTTP: POST
URL: `/stock-removals`

Úprava vyskladnění Metoda HTTP: PUT
URL: `/stock-removals/{id}`

Smazání vyskladnění Metoda HTTP: DELETE
URL: `/stock-removals/{id}`

Vytvoření typu vyskladnění Metoda HTTP: POST
URL: `/stock-removal-types`

Smazání typu vyskladnění Metoda HTTP: DELETE
URL: `/stock-removal-types/{id}`

Zobrazení stavu skladových zásob Metoda HTTP: GET
URL: `/stock`

4.4.4 Mikroslužba spravující odměny

Mikroslužba spravující odměny (`Rewards Service`) se bude zabývat evidencí, přiřazováním, odebíráním bodů a odměnitelných aktivit, za které zákazníci tyto body mohou získat.

Mikroslužba spravující odměny poskytuje své rozhraní dvěma mikroslužbám:

- mikroslužbě na správu objednávek, která u ní zjišťuje jestli uživatel má dost bodů a také jí posílá informace o vzniklých objednávkách, aby body odečetla a

- mikroslužbě spravující rezervace, která jí zašle informace o ukončené rezervaci a podle ní tato mikroslužba přiřadí zákazníkovi body.

Tato komunikace je zachycena na obrázku 4.5.

4.4.4.1 Analýza funkčních požadavků

V této části budou popsány funkční požadavky na mikroslužbu spravující odměny. Tyto požadavky jsou následující:

F1 Správa bodových transakcí Mikroslužba bude umožňovat spravovat bodové transakce a to jejich vytváření, editaci, mazání a zobrazování. Vytvořit bodovou transakci půjde buď předáním konkrétních údajů, nebo z právě ukončené rezervace. Zobrazovat bodové transakce půjde jako seznam všech transakcí, všech transakcí daného uživatele a aktuálního stavu bodového konta daného uživatele/uživatelů.

F2 Správa odměnitelných aktivit Mikroslužba bude umožňovat spravovat odměnitelné aktivity. Odměnitelná aktivita půjde vytvořit, upravit, smazat a zobrazit. Zobrazit půjde jak jedna konkrétní, tak všechny odměnitelné aktivity.

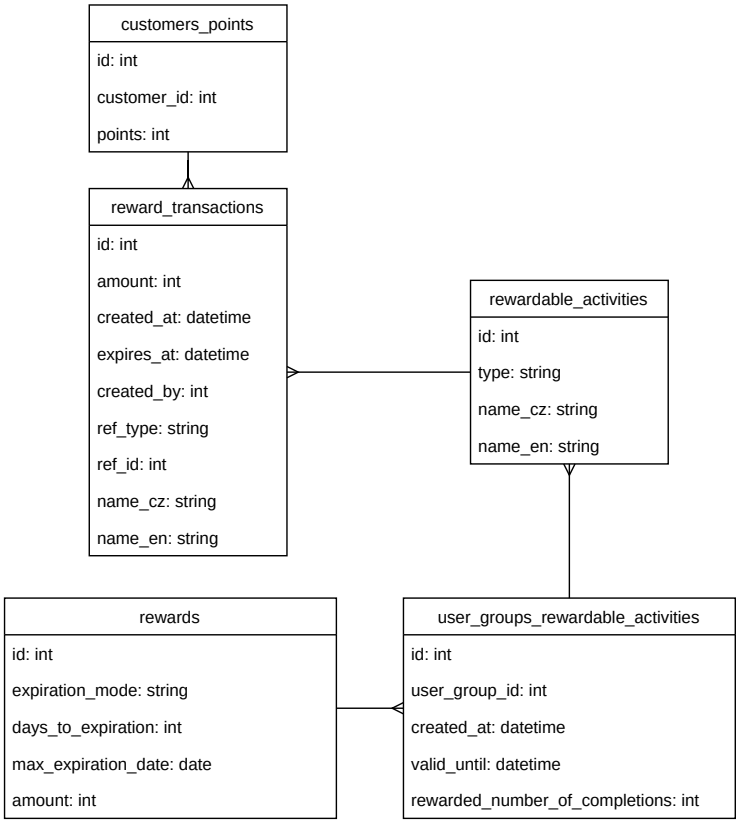
F3 Správa odměn Mikroslužba bude umožňovat spravovat odměny, které mohou být uživatelům přiřazeny. Odměna půjde vytvořit, upravit, smazat a zobrazit. Zobrazit půjde jak jedna konkrétní, tak všechny odměny. Odměny půjdou přiřadit k odměnitelné aktivitě v rámci konkrétní uživatelské skupiny a také z ní půjdou odebrat.

4.4.4.2 Doménový model

Doménový model u mikroslužby spravující odměny bude velmi podobný jako v současném řešení. Více informací o jednotlivých entitách a jejich informacích je uvedeno v části 3.2.2.2. Rozdíl je v tom, že u zákazníků nebude evidovat žádné údaje kromě toho, kolik který zákazník má bodů. Diagram je na obrázku 4.9.

4.4.4.3 Návrh API rozhraní

Každá mikroslužba komunikuje skrze API rozhraní. V této části je popsán návrh API rozhraní mikroslužby na správu bodových transakcí. Návrh pro toto API je následující:



Obrázek 4.9: Návrh doménového modelu pro mikroslužbu spravující odměny

Zobrazení bodů všech uživatelů Metoda HTTP: GET
URL: `/points`

Zobrazení bodů uživatele Metoda HTTP: GET
URL: `/points/{customerId}`

Zobrazení bodových transakcí všech uživatelů Metoda HTTP: GET
URL: `/reward-transactions`

Zobrazení bodových transakcí uživatele Metoda HTTP: GET
URL: `/reward-transactions?customer={customerId}`

Vytvoření bodové transakce Metoda HTTP: POST
URL: `/reward-transactions`

Upravení bodové transakce Metoda HTTP: PUT
URL: `/reward-transactions/{id}`

Smazání bodové transakce Metoda HTTP: DELETE
URL: `/reward-transactions/{id}`

Zobrazení všech odměnitelných aktivit Metoda HTTP: GET
URL: `/rewardable-activities`

Zobrazení odměnitelné aktivity Metoda HTTP: GET
URL: `/rewardable-activities/{id}`

Vytvoření odměnitelné aktivity Metoda HTTP: POST
URL: `/rewardable-activities`

Upravení odměnitelné aktivity Metoda HTTP: PUT
URL: `/rewardable-activities/{id}`

Smazání odměnitelné aktivity Metoda HTTP: DELETE
URL: `/rewardable-activities/{id}`

Zobrazení všech odměn Metoda HTTP: GET
URL: `/rewards`

Zobrazení odměny Metoda HTTP: GET
URL: `/rewards/{id}`

Vytvoření odměny Metoda HTTP: POST
URL: `/rewards`

Úprava odměny Metoda HTTP: PUT
URL: `/rewards/{id}`

Smazání odměny Metoda HTTP: DELETE
URL: `/rewards/{id}`

Přiřazení odměny k odměnitelné aktivitě Metoda HTTP: POST
URL: `/user-groups-rewards-reward-activity`

Odebrání odměny z odměnitelné aktivity Metoda HTTP: DELETE
URL: `/user-groups-rewards-reward-activity`

Přiřazení bodů za rezervaci Metoda HTTP: POST
URL: `/reward-transactions/reservation`

4.5 Tvorba a správa rezervací

Mikroslužba spravující tvorbu a správu rezervací bude ústřední mikroslužbou systému Uniqway. Tato mikroslužba se bude, jak již název napovídá, starat o rezervace vozidel. Návrh komunikace této mikroslužby s ostatními mikroslužbami je na obrázku 4.10. Jak je z návrhu patrné, musí tato mikroslužba

komunikovat s velkým množstvím ostatních mikroslužeb, které se v systému nachází. Při vytvoření rezervace musí tato mikroslužba kontaktovat mikroslužby spravující:

- uživatele (`Users Service`), ve které zjistí údaje o uživateli, v jakém je uživatel stavu (jestli může rezervovat) a jaká vozidla si může rezervovat,
- auta (`Cars Service`), od které zjistí informace o daném autě.

Při ukončení rezervace musí kontaktovat mikroslužby spravující:

- auta (`Cars Service`), od které zjistí informace o daném autě, tedy jestli je zamknuté, a jeho polohu,
- parkovací zóny (`Parking Zones Service`), kde zjistí jestli dané auto se nachází v jedné z parkovacích zón (mimo tyto zóny nelze auto vrátit),
- ceny (`Price Service`), od které zjistí kolik daná rezervace bude stát,
- faktury (`Invoice Service`), které zašle informace o ukončené rezervaci a požadavek na vytvoření faktury,
- notifikace (`Notification Service`), které pošle informace o ukončené rezervaci a faktuře k proplacení, která bude zaslána uživateli a
- odměny (`Rewards Service`), které pošle informace o ukončené rezervaci a ta přiřadí danému zákazníkovi body za jízdu.

4.5.1 Analýza funkčních požadavků

V této části budou popsány funkční požadavky na mikroslužbu spravující rezervace. Tyto požadavky jsou následující:

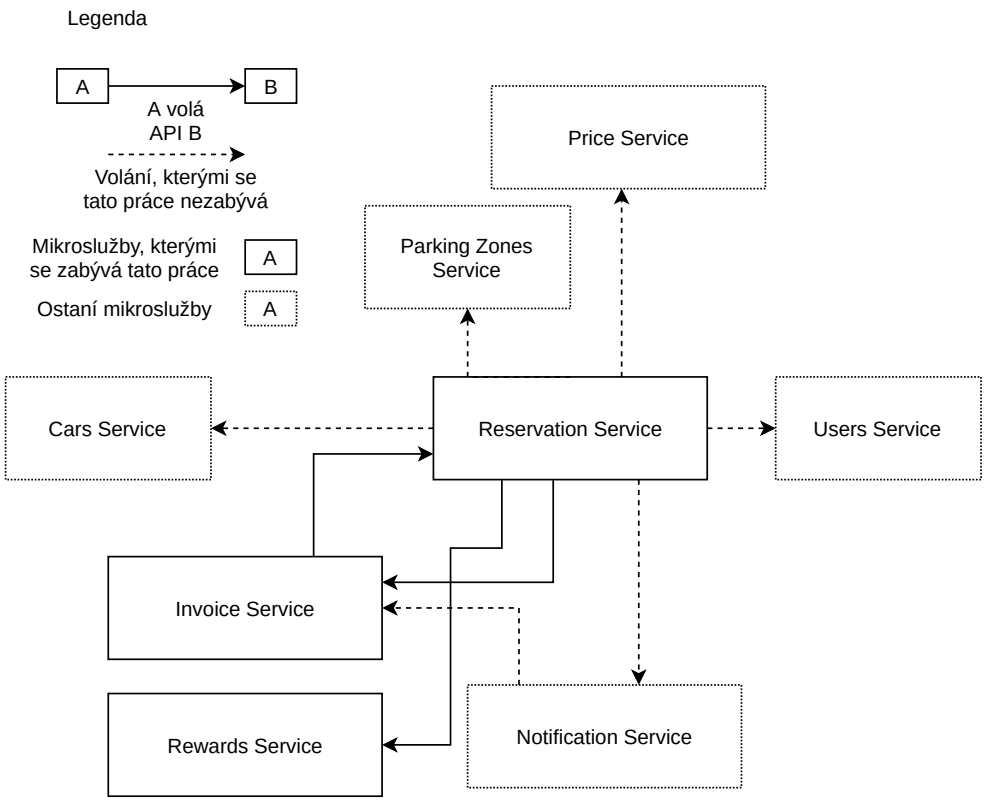
F1 Zobrazení všech rezervací Mikroslužba umožní zobrazit seznam všech existujících rezervací. O každé rezervaci bude evidovat stejné parametry jako v současném stavu uvedené v sekci 3.2.2.1.

F2 Zobrazení konkrétní rezervace Mikroslužba umožní zobrazit jednu konkrétní rezervaci.

F3 Vytvoření rezervace Mikroslužba bude umožňovat vytvoření rezervace. Jedná se o vytvoření daným uživatelem, nebo administrátorem, který rezervaci vytvoří na někoho jiného.

F4 Ukončení rezervace Mikroslužba bude umožňovat ukončení rezervace a to buďto uživatel sám sobě, nebo administrátor konkrétní rezervaci.

F4.1 Ukončení rezervace uživatelem Uživatel bude moci ukončit rezervaci, pokud auto zaparkuje v jedné ze zón k tomu určených.



Obrázek 4.10: Návrh komunikace s rezervační mikroslužbou

F4.2 Ukončení rezervace administrátorem Každou rezervaci bude moci ukončit administrátor a auto nebude muset být v tu chvíli v zóně.

F5 Úprava rezervace Rezervace bude možné upravovat pomocí rozhraní které bude mikroslužba poskytovat.

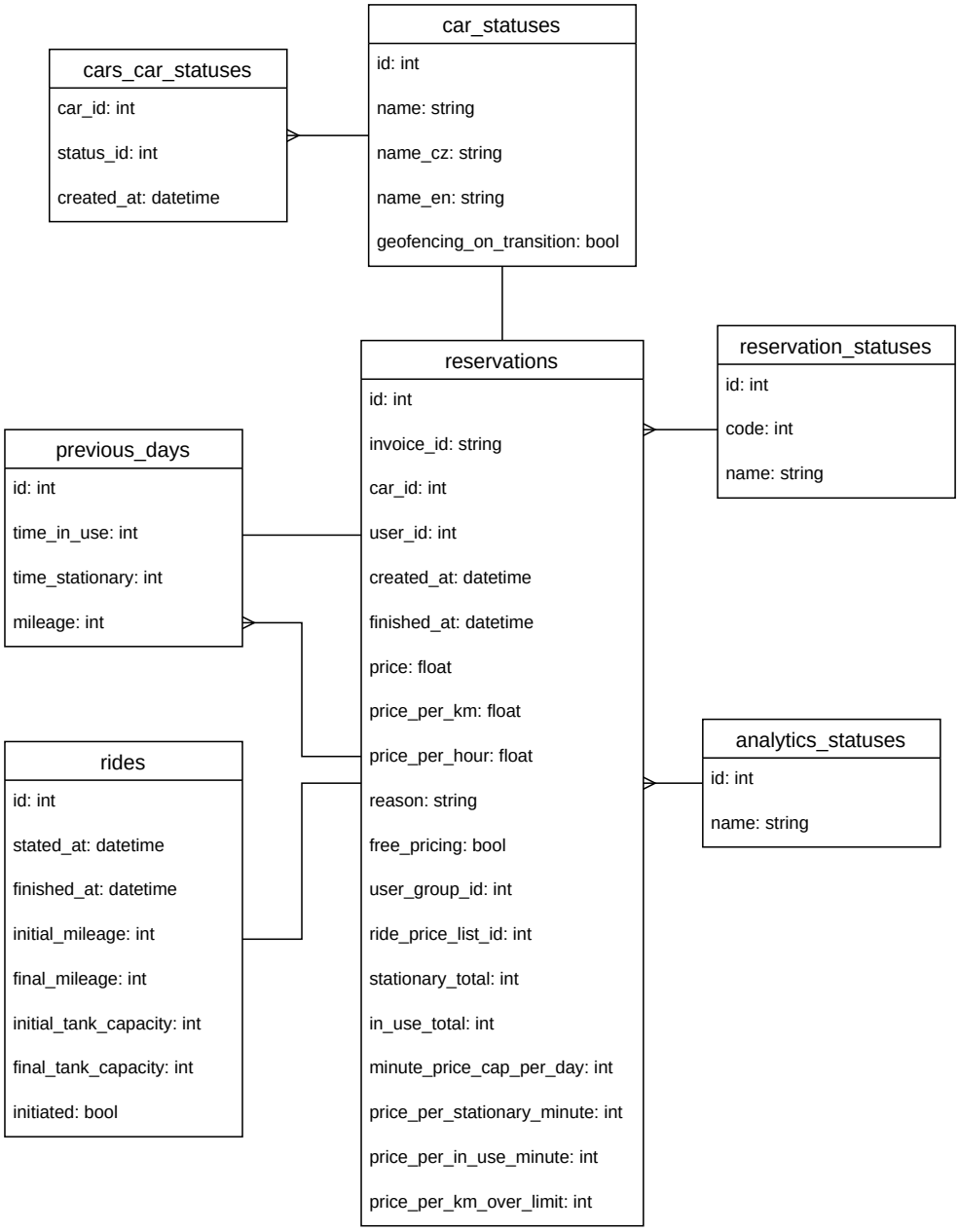
F6 Smazání rezervace Mikroslužba umožní smazání rezervace.

F7 Získání ceny rezervace Mikroslužba umožní získání ceny aktuální neukončené rezervace.

F8 Získání možností ukončení rezervace Mikroslužba umožní získat informace o ukončení dané rezervace. Tyto informace zahrnují možné stavy do kterých se má auto po ukončení rezervace přepnout.

4.5.2 Doménový model

Doménový model pro mikroslužbu spravující rezervace bude popsán v této části práce. Diagram doménového modelu je na obrázku 4.11.



Obrázek 4.11: Doménový model mikroslužby spravující rezervace

4.5.2.1 Rezervace

Entita rezervace (`reservations`) eviduje všechny důležité údaje o dané rezervaci. Rezervace obsahují následující atributy:

invoice_id jednoznačný identifikátor faktury, která přísluší dané rezervaci, datový typ: int

car_id jednoznačný identifikátor auta, které je v rámci dané rezervace rezervováno, datový typ: int

user_id identifikátor uživatele, na kterého je rezervace vytvořená, datový typ: int

created_at datum a čas vytvoření rezervace, datový typ: datetime

finished_at datum a čas ukončení rezervace, datový typ: datetime

price cena rezervace v haléřích, datový typ: int

price_per_km cena v haléřích za ujetý kilometr, datový typ: int

price_per_hour cena v haléřích za započatou hodinu, datový typ: int

reason důvod jízdy zdarma (musí mít *free_pricing* na *true*), datový typ: text

free_pricing příznak jestli je rezervace zdarma - nezáleží na ceně, datový typ: bool

user_group_id identifikátor uživatelské skupiny, ve které byla rezervace vytvořena, datový typ: int

ride_price_list_id identifikátor ceníku, ve kterém byla rezervace vytvořena, datový typ: int

stationary_total počet minut rezervace kdy auto bylo zamknuté a s vypnutým motorem, datový typ: int

in_use_total počet minut rezervace kdy auto bylo používáno, datový typ: int

minute_price_cap_per_day strop ceny za minuty v haléřích na den, datový typ: int

price_per_stationary_minute cena v haléřích za minutu stání, datový typ: int

price_per_in_use_minute cena v haléřích za minutu jízdy, datový typ: int

price_per_km_over_limit cena v haléřích za kilometr nad limit, datový typ: int

4.5.2.2 Stavy rezervací

Stavy rezervací (`reservation_statuses`) evidují různé stavy ve kterých se může rezervace nacházet. V současném stavu existují dva stavy: *vytvořená* a *ukončená*. Stavy obsahují následující atributy:

code jednoznačný identifikátor stavu, datový typ: int

name jméno stavu, datový typ: string

4.5.2.3 Analytické stavy rezervací

Analytické stavy rezervací (`analytics_statuses`) evidují různé stavy rezervací, které slouží k analýzám. V současném stavu existuje pouze jeden a to *technická chyba*. Analytické stavy obsahují následující atribut:

name jméno stavu, datový typ: string

4.5.2.4 Stavy aut

Stavy aut (`car_statuses`) evidují různé stavy aut, ve kterých se auta mohou nacházet. Stav auta obsahuje následující atributy:

name jméno stavu, datový typ: string

name_cz jméno stavu v českém jazyce, datový typ: string

name_en jméno stavu v anglickém jazyce, datový typ: string

geofencing_on_transition příznak, jestli se kontroluje zda-li je auto v zóně, když se auto má přepnout do tohoto stavu, datový typ: bool

4.5.2.5 Stavy aut přiřazené k autům

Entita (`cars_car_statuses`) eviduje v jakém stavu se jednotlivá auta nacházejí. Eviduje pouze které auto je ve kterém stavu a datum a čas kdy k přepnutí do tohoto stavu došlo.

4.5.2.6 Předchozí dny

Předchozí dny (`previous_days`) evidují předchozí dny rezervace. Slouží k vypočítávání ceny, pokud je rezervace delší než jeden den. Předchozí den obsahuje následující atributy:

time_in_use počet minut jízdy v minulém dni, datový typ: int

time_stationary počet minut stání v minulém dni, datový typ: int

mileage počet ujetých kilometrů za minulý den, datový typ: int

4.5.2.7 Jízdy

Jízdy (`rides`) evidují samotnou jízdu uskutečněnou v rámci rezervace. Slouží k vypočítávání ceny a evidence zneužívání tankování. Jízda obsahuje následující atributy:

started_at datum a čas začátku jízdy, datový typ: datetime

finished_at datum a čas konce jízdy, datový typ: datetime

initial_mileage počáteční stav tachometru, datový typ: int

final_mileage koncový stav tachometru, datový typ: int

initial_tank_capacity počáteční stav nádrže, datový typ: int

final_tank_capacity koncový stav nádrže, datový typ: int

initiated příznak jestli byla jízda zahájená, datový typ: bool

4.5.3 Návrh API rozhraní

V této části bude popsán návrh API rozhraní pro mikroslužbu spravující rezervace. Návrh tohoto rozhraní je následující:

Získání všech rezervací V těle odpovědi se vrací pole rezervací. Tento požadavek na server splňuje funkční požadavek F1 (4.5.1).
Metoda HTTP: GET
URL: `/reservations`

Získání konkrétní rezervace V těle odpovědi se vrátí informace o konkrétní rezervaci. Tento požadavek na server splňuje funkční požadavek F2.
Metoda HTTP: GET
URL: `/reservations/{id}`

Vytvoření nové rezervace V těle požadavku se zašlou informace o autě, které se má rezervovat, uživateli, na kterého se rezervace vytvoří, uživatelské skupině, roli a důvodu. V těle odpovědi se vrátí id vzniklé rezervace. Tento požadavek na server splňuje funkční požadavek F3.
Metoda HTTP: POST
URL: `/reservations`

Ukončení rezervace administrátorem V těle požadavku se zašlou informace o stavu, do kterého se má vozidlo přepnout po ukončení rezervace. V těle odpovědi se vrátí cena ukončené rezervace, datum a čas začátku a konce rezervace a počet ujetých kilometrů. Tento požadavek na server splňuje funkční požadavek F4.2.
Metoda HTTP: POST
URL: `/reservations/{id}/finish`

Ukončení rezervace uživatelem V těle odpovědi se vrátí cena ukončené rezervace, datum a čas začátku a konce rezervace a počet ujetých kilometrů. Tento požadavek na server splňuje funkční požadavek F4.1.
Metoda HTTP: POST
URL: `/reservations/current/finish`

Úprava rezervace V těle požadavku se zašlou údaje rezervace včetně těch změněných. Tento požadavek na server splňuje funkční požadavek F5.
Metoda HTTP: PUT
URL: `/reservations/{id}`

Smazání rezervace V URL se uvede id rezervace, která má být smazána. Tento požadavek na server splňuje funkční požadavek F6.
Metoda HTTP: DELETE
URL: `/reservations/{id}`

Získání ceny aktuální rezervace V těle odpovědi se vrátí informace o ceně aktuální rezervace. Tento požadavek na server splňuje funkční požadavek F7.
Metoda HTTP: GET
URL: `/reservations/current/price`

Získání možností ukončení rezervace V těle odpovědi se vrátí informace o možnostech ukončení: možné stavy aut, do kterých lze auto po ukončení přepnout. Tento požadavek na server splňuje funkční požadavek F8.
Metoda HTTP: GET
URL: `/reservations/current/finish-options`

Kapitola 5

Implementace

V této kapitole bude popsána implementace vzniklých mikroslužeb a ukázky použití některých funkcionalit. K realizaci jednotlivých mikroslužeb je používán Play Frameworku [17]. Tento framework byl zvolen z důvodu širokého rozšíření mezi programátory, dobrou dokumentací a tým Uniqway s ním má již zkušenosti, protože je v něm psaná současná serverová aplikace. Mikroslužby spolu komunikují přes REST API (2.1.1).

5.1 Implementace mikroslužby

Tvorba nových mikroslužeb je realizována pomocí naklonování šablony mikroslužeb a následných úprav. Tato šablona mikroslužeb je blíže popsaná v sekci 5.1.1.

Při tvorbě této práce a vytváření nových mikroslužeb bylo velké úsilí věnováno generování kódu. Generování kódu je technika, kdy je zdrojový kód automaticky generovaný. Jako zdroj informací pro generování se používá např. dokumentace. Generování kódu má mnohé výhody a ty jsou například:

- snadná změna funkcionality, u které se většina aplikace jednoduše přegeneruje,
- konzistentní kód,
- šetření času.

Nevýhodou generovaného kódu může například být větší komplexita, než u kódu psaného programátorem, či složitější změny funkcionality, pokud vygenerovaný kód neodpovídá požadavkům vývojářů. [29] Ke generování kódu mikroslužeb jsou používané dvě technologie: OpenAPI Generátor [30] a JOOQ generátor [31]. Obě tyto technologie budou v následujících sekcích blíže popsány.

Všechny generované třídy mají v názvu balíku, ve kterém se nacházejí, slovo `generated`, pouze kontroléry toto nesplňují, protože jsou v jednom balíku jak generované kontroléry, tak i ty psané programátorem. Tato konvence byla zavedena, aby se snadněji poznalo, které třídy jsou generované a neměli by se upravovat ručně, ale přegenerováním, nebo zděděním a upravením zděděné třídy. U kontrolérů se toto rozlišuje pomocí koncovky, kde programátorem upravované třídy mají koncovku `*Imp.java`.

Framework Play používá ke správě databáze tzv. `Play Evolutions` [32], což jsou soubory, které manipulují se strukturou databáze. Každá evolution obsahuje dvě sekce a to *Up* a *Down*. V *Up* sekci je SQL skript, který se má použít při nasazení dané evoluce a v sekci *Down* je skript, který se má použít při rušení dané evoluce. Tyto sekce se musí navzájem vyrušit např. pokud je v sekci *Up* vytvořena nová tabulka, v sekci *Down* musí být smazána. Tyto evoluce jsou používány pro vytvoření databázového schématu a jeho změnu. Díky nim mají všichni programátoři stejnou databázi. Tyto evoluce také slouží k vytvoření schématu databáze při testování.

5.1.1 Šablona mikroslužeb

Šablona pro mikroslužby vznikla ve spolupráci se členy týmu a slouží k usnadnění tvorby nových mikroslužeb a sdílení společných funkcionalit mezi mikroslužbami. Jak již bylo zmíněno v kapitole o návrhu 4.1, tato šablona obsahuje základní konfiguraci, health check endpoint, logování, zpracování výjimek, připojení k databázi, nastavení OpenAPI generátoru a generátoru modelů z databáze (JOOQ). Jak byly tyto funkcionality implementovány bude nyní blíže popsáno.

5.1.1.1 Výjimky

V každé aplikace může dojít k nějaké chybě a tyto chyby se většinou reprezentují pomocí výjimek. Výjimky nemusí reprezentovat jen chyby, ale i nepovolené operace apod.

V rámci implementace vznikla tzv. `BaseException`, což je rodičovská abstraktní třída výjimek, které jsou specifické pro vzniklé mikroslužby a pro systém Uniqway. Každá výjimka, které je specifická pro Uniqway systém dědí z třídy `BaseException`. Každá tato výjimka má svůj celočíselný kód, který jí jednoznačně reprezentuje. Dále má také popis chyby v českém a anglickém jazyce, kde se uživateli zobrazí chyba pouze v jazyce, který požaduje.

Texty výjimek používají funkcionalitu Play Messages [22], která podporuje vícejazyčné texty. K textům výjimek nebude využívána mikroslužba spravující texty, jelikož se texty výjimek málokdy mění a při vyhození výjimky dochází při nastalém problému a nebylo by dobré toto ještě zatěžovat dotazováním se na text nějaké chyby. Pokud v aplikaci dojde k vyhození výjimky, tak je zachycena v `ErrorHandler`u, což je třída, která spravuje zachytávání výjimek

a vracení rozumné odpovědi uživateli. Pokud dojde k vyhození výjimky, která dědí od `BaseException`, tak `ErrorHandler` poskládá odpověď z této výjimky a najde text v daném jazyce a ten vrátí uživateli. Ukázka takové odpovědi je v ukázce kódu 1. Pokud vhozená výjimka není podtřídou `BaseException`, tak se uživateli vrátí `BaseException` s kódem -1 a informací, že nastala interní chyba. Více informací lze dohledat v logu aplikace. Všechny vyhozené výjimky zapisovány do logu aplikace a lze je později dohledat.

```
{
    "error": {
        "code": 1,
        "message": "Text se zadaným kódem již existuje."
    }
}
```

Zdrojový kód 1: Ukázka odpovědi po vyhození výjimky

5.1.1.2 Health check

Každá mikroslužba poskytuje endpoint pro tzv. health check, neboli kontrolu funkčnosti. Pokud dojde k požadavku na tento endpoint, tak aplikace vrátí v odpovědi 200 OK a dotazující ví, že daná mikroslužba je funkční. Tělo odpovědi je v ukázce kódu 2. Pokud je vráceno cokoliv jiného, je s mikroslužbou

```
{
    "message": "Up and running"
}
```

Zdrojový kód 2: Ukázka odpovědi pro základní health check

něco v nepořádku. Tento endpoint bude využíván *Service discovery* mechanismem, který je součástí AWS a díky němu si load balancer eviduje seznam funkčních mikroslužeb a pokud nějaká mikroslužba na tento požadavek několikrát za sebou odpoví chybou nebo neodpoví je vyřazena z provozu.

5.1.1.3 Konfigurace

Šablona mikroslužeb také obsahuje základní společnou konfiguraci všech mikroslužeb. Tato konfigurace je klasická konfigurace aplikací psaných v Play frameworku. Konfigurace se nachází v `/conf/application.conf` a obsahuje nastavení jazykových variant, třídy, která zpracovává výjimky, spouštění Play Evolutions a nastavení ovladače k databázi. Citlivé údaje jako jsou přístupové údaje do databáze jsou do tohoto souboru pouze importovány z jiného kon-

figuračního souboru, který je pro každou mikroslužbu jiný a není součástí verzovacího systému.

5.1.2 OpenAPI Generátor

OpenAPI Generátor je generátor klientského a serverového kódu z API dokumentace [30]. OpenAPI Generator obsahuje přes 50 klientských generátorů a přes 40 serverových. Tento generátor byl zvolen kvůli jeho rozšířenosti mezi komunitou a schopnosti vygenerovat velké množství funkcionalit. Generování kódu z dokumentace má několik výhod a tyto výhody jsou: rychlá tvorba komponent, které zpracovávají požadavky, snadné změny a nutnost mít dokumentaci aktuální, protože se z ní kód generuje. Po vyspecifikování API dokumentace ve formátu OpenAPI [33], lze pomocí tohoto generátoru vygenerovat serverovou část pro všechny endpointy vyspecifikované v dokumentaci. Toto vygeneruje všechny kontroléry, které budou příchozí požadavky zpracovávat. Při generování vždy vzniknou 3 soubory od jednoho kontroléru:

- Abstraktní třídu kontroléru, která obsahuje všechny metody, které jsou v API vyžadovány. Tyto metody jsou abstraktní a je třeba jim doplnit implementaci. Jména těchto tříd vždy končí příponou `ImpInterface`.
- Implementaci abstraktní třídy, která dědí z výše uvedené abstraktní třídy neimplementované metody a ty implementuje. Jen tyto třídy obsahují zásahy od programátorů ve formě implementací daných metod. Tato třída se generuje jen v prvním generování, protože obsahuje programátorské změny a přegenerování by je smazalo. Ukázka takovéto implementace je v ukázce kódu 3. Tyto třídy končí vždy příponou `ControllerImp`.
- Třídu kontroléru, která dědí z Play Controlleru [34] a nastavuje výše uvedenou implementaci jako toho, kdo má příchozí požadavky zpracovávat. Tyto třídy končí vždy příponou `Controller`.

```
public class CheckApiControllerImp extends
                            CheckApiControllerImpInterface {
    @Override
    public CheckDto getCheck(Http.Request request)
                                    throws Exception {
        CheckDto dto = new CheckDto();
        dto.setMessage("Up and running.");
        return dto;
    }
}
```

Zdrojový kód 3: Implementace kontroléru a metody na základní health check

Kromě kontrolérů tak OpenAPI Generátor generuje i DTO objekty používané při přenosu dat přes REST API. Tyto DTO objekty vygeneruje v balíku `models.dto.generated`. Podle specifikované konvence jsou u názvů DTO souborů používány předpony charakterizující druh operace, na který se daný DTO používá: pro vytvoření nové entity je používána předpona *Store*, pro úpravy *Update* a pro získání objektu předpona *Get*. Příkladem může být DTO pro vytvoření nového Textu s názvem `StoreTextDto`.

5.1.2.1 Klientská část mikroslužby

Pomocí OpenAPI Generátoru se negeneruje jen serverová část aplikace, ale dá se vygenerovat i klientská část aplikace. Generátoru je poskytnuta API dokumentace serveru, který bude volán z daného klienta a generátor vygeneruje klientskou část, která volá dané endpointy serveru a zpracovává odpovědi do daných proměnných. Tyto vygenerovaní klienti se používání pro komunikaci mezi mikroslužbami, kde mikroslužba, která chce volat jinou si vygeneruje u sebe klientskou část mikroslužby, kterou chce volat. K samotnému volání je používaný framework Retrofit [35], který se také používá v Uniqway Android aplikaci. URL volané mikroslužby je v API definované jako název proměnné, který se hledá v konfiguraci, ve které je poté napsán skutečný název URL. Toto umožňuje měnit URL v závislosti na prostředí a umožňuje generovat klienty bez závislosti na výsledném URL.

Při vygenerování klienta se vytvoří rozhraní, které má nadefinované všechny metody, které je potřeba volat pro získání dat, které rozhraní poskytuje. Každá metoda tohoto rozhraní je oanotovaná HTTP metodou, kterou se má daný endpoint volat a příponou URL endpointu, která se přidá za základní URL definovanou pro daný server. Ukázka takové metody je v ukázce kódu 4. Při použití tohoto rozhraní je potřeba ve službě, kde chceme klientské API vo-

```
@GET("api/v2/demo/cars/detailed")
CompletionStage<Response<List<CarDetailed>>> carsDetailed();
```

Zdrojový kód 4: Ukázka metody v klientském rozhraní

lat, vytvořit implementaci tohoto rozhraní. Tato implementace se tvoří pomocí metody `createService()`, která se nachází ve vygenerované třídě `ApiClient`. Ukázka jak se vytvoří tato implementace a jak se volá metoda z klientského API je ukázce kódu 5.

5.1.3 JOOQ Generátor

JOOQ [31] je framework, který poskytuje jednoduchý způsob jak integrovat SQL do Java. JOOQ Generátor [31] je generátor modelových tříd, POJO (plain old java object), DAO a jiných tříd potřebných pro přístup do databáze. Tento generátor a následně používání JOOQ pro přístup k databázi bylo

```
public class CarsService {

    protected final CarApi carApi;

    @Inject
    public CarsService(ApiClient client, CarsDao carsDao) {
        // vytvori implementaci rozhrani CarApi
        this.carApi = client.createService(CarApi.class);
    }


    public List<CarDetailedDto> carsDetailed()
                                              throws Exception {
        // vola carsDetailed metodu z carApi rozhrani a odpoved
        // vrati ve formatu List<CarDetailedDto>
        return carApi.carsDetailed().toCompletableFuture()
            .get().body().stream().map(
                c -> Json.fromJson(Json.toJson(c),
                                   CarDetailedDto.class)
            ).collect(Collectors.toList());
    }
}
```

Zdrojový kód 5: Ukázka služby využívající volání klienta

zvoleno kvůli velkým generačním schopnostem, datově bezpečným SQL dotazům a možnosti kontroly nad množstvím dat, která se z databáze při jednom dotazu získají (nestane se, že při jednom dotazu se postupně z databáze získají data o téměř všech tabulkách, které jsou nějak napojeny na dotazovanou entitu).

V šabloně mikroslužeb se nachází základní nastavení tohoto generátoru, které se nachází v `/conf/jooq-config.xml.demo`, který je následně v každé mikroslužbě při spuštění generování vyplněn relevantními údaji jako jsou cesty k souborům a přihlašovací údaje do databáze, které se nachází již ve zmíněné konfiguraci 5.1.1.3. Tento konfigurační soubor je vyplněn relevantními údaji pomocí python skriptu, který byl v rámci této práce napsán a nachází se v `/src/utils/jooq/setJOOQConfigFile.py`. JOOQ Generátor se připojí na databázi, která je vyspecifikovaná v této konfiguraci a pomocí reverzního inženýrství k ní vygeneruje třídy pro práci s ní. Vygenerované třídy pomocí JOOQ generátoru se nachází v balíku `models.database.generated`.

5.2 Komponenty

V této sekci budou popsány komponenty, které má každá mikroslužba jiné a bude se soustředit na ty části, které nejsou generované pomocí výše uvedených generátorů, ale jsou psané ručně programátorem.

5.2.1 Kontroléry (controllers)

Jak již bylo zmíněno v sekci 5.1.2, tak kontroléry jsou generovány z API dokumentace a každy kontrolér se skládá ze tří tříd. Třída s příponou `Imp` implementuje abstraktní metody, které obsahují dané požadavky a tyto metody musí být implementovány programátorem.

Kontroléry by neměly obsahovat žádnou obchodní logiku a požadavky by měli jen přesměrovávat na služby (services) a vracet jejich odpovědi zpět klientům. Ukázka implementace metody na vrácení všech textů v mikroslužbě na správu textů je v ukázce kódu 6.

```
@Override
public List<GetTextDto> getTexts(Http.Request request) throws
                                                        Exception {
    return textsService.getTexts();
}
```

Zdrojový kód 6: Implementace metody kontroléru na získání všech textů

5.2.2 Služby (services)

Služby jsou srdcem celé aplikace a obsahují veškerou obchodní logiku. Služba přijme požadavek od kontroléru, vykoná příslušné operace a vrátí výsledek ve formě nějakého DTO objektu, nebo nic, pokud výsledek není vyžadován.

Služby k přístupu k databázi využívají DAO objekty, které byly vygenerované pomocí JOOQ generátoru. Pokud funkcionalita DAO objektu není dostatečná, je tento soubor zděděn a využívá se jeho rozšířená varianta, kde programátor dopsal pokročilejší funkcionalitu. Ukázka metody na získání všech textů ve službě je v ukázce kódu 7.

5.2.3 Evolutions

Evolutions jsou speciální SQL skripty, které manipulují s databázovým schématem. Tyto evoluce se nazývají celými čísly (např. `1.sql`) a jdou takto popořadě. Jak již bylo zmíněno v sekci 5.1, tak každá evoluce obsahuje dvě části: *Up* a *Down*. Při každém spuštění aplikace se spustí všechny neaplikované, nebo všechny evoluce následující první změněnou včetně (tedy pokud cokoliv je změněno v evoluci `3.sql`, tak je znovu aplikována evoluce 3,4,5 až

```
public List<GetTextDto> getTexts() {
    // textsDao najde vsechny texty v databazi
    // a vrati je ve forme POJO objektu Texts
    List<Texts> texts = textsDao.findAll();
    // seznam POJO textu prevedeme na seznam DTO textu
    List<GetTextDto> dtos = new ArrayList<>();
    for (Texts text : texts) {
        dtos.add(getGetTextDtoFromText(text));
    }
    return dtos;
}
```

Zdrojový kód 7: Implementace metody služby na získání všech textů

```
# --- !Ups

create table texts
(
    text_code   varchar(255) primary key,
    text_cs     text         not null,
    text_en     text         not null,
    description text,
    created_at  timestamptz not null default now(),
    modified_at timestamptz not null default now()
);

# --- !Downs

drop table if exists texts;
```

Zdrojový kód 8: Ukázka evolution u mikroslužby na správu textů

poslední). Při spuštění nových dojde pouze ke spuštění jejich *Up* části, při změně nějaké dojde nejprve ke spuštění *Down* části všech následujících včetně té změněné a poté až se spustí *Up* části. Kvůli tomuto lze přijít o data, pokud se omylem změní nějaká starší evoluce a zavolá se *Down* část všech jí následujících. Jinak je to skvělý pomocník při udržování databáze ve stejném stavu pro všechny programátory. V rámci evoluce jde do databáze i přidat data, která by měla obsahovat každá její instance (např. hodnoty číselníků). Ukázka evoluce v mikroslužbě na správu textů je v ukázce kódu 8.

Kapitola 6

Testování a dokumentace

V této kapitole bude popsáno proč testovat aplikace, jaké druhy testů existují, jaké jsou v rámci této práce realizovány a jaké technologie jsou na testování používány. Dále také bude popsáno jakým způsobem jsou mikroslužby dokumentovány.

6.1 Testování

Testování aplikací je velice důležité, avšak velmi zanedbávané téma v oblasti softwarového vývoje. Dobře otestovaná aplikace může ušetřit spousty peněz a času programátorů při hledání chyb. Správně napsané testy mohou odhalit kritické chyby v aplikaci, které by mohli přinést obrovské škody např. na ušlém zisku za dobu, kdy je systém kvůli takové chybě nefunkční. [36]

6.1.1 Typy testů

Testy se dají dělit do několika skupin podle různých kritérií. Tyto skupiny mohou dle [36] být:

- **Dle znalosti vnitřní struktury** - Dělení testů z hlediska znalosti vnitřní struktury může být následující:
  - **White box** - Whitebox testy jsou takové testy u kterých má tester znalosti vnitřní struktury aplikace a zná zdrojový kód. Toto mu umožňuje lépe identifikovat problémy. Nevýhodou je citlivost na změnu implementace.
  - **Black box** - Blackbox testy jsou takové testy u kterých tester neví nic o jejich vnitřní struktuře a testuje pouze chování na rozhraní. Tyto testy se nemusí měnit, pokud se změní implementace a chování na rozhraní zůstane stejné.

  - **Gray box** - Gray box testy jsou mezistupněm mezi whitebox a blackbox testy. U těchto testů zná tester použité algoritmy a technologie, ale nezná implementaci.

- **Dle rozsahu** - Členění testů podle rozsahu testované části od nejmenších testů až po testy celých aplikací je následující:
  - **Unit testy** - Unit testy, neboli jednotkové testy testují jednotlivé třídy. Jsou nejmenším druhem testů a většinou běží na počítači vývojáře. Jedná se o whitebox testy.
  - **Testy komponent** - Obdobné jako unit testy, ale testují více komponent najednou. Netestují jen jednu třídu ale i komunikaci mezi třídami, které představují nějakou komponentu systému. Většinou se jedná o whitebox testy.
  - **Integrační testy** - Integrační testy se snaží simulovat reálné prostředí a testují spolupráci komponent. Často používají i připojení k databázi. Bývají typu blackbox, pokud se jedná o test komponent jiných dodavatelů a whitebox pokud se jedná o vlastní komponenty.
  - **Systémové testy** - Systémové testy slouží k otestování aplikace jako celku. Většinou jsou typu blackbox a testují podle scénářů z modelu případu užití.

- **Dle způsobu kontroly** - Testy se mohou dělit podle způsobu provádění na statické a dynamické.
  - **Statické** - Statické testy nevyžadují spuštění aplikace. Často se jedná o statickou analýzu kódu, která v něm hledá chyby např. zapomenutí inicializace proměnných.
  - **Dynamické** - U dynamických testů je nutné aplikaci spustit. Tyto testy kontrolují chování aplikace a jestli splňuje očekávání.

6.1.2 Testování mikroslužeb

Testování každé mikroslužby bylo provedeno několika způsoby. Prvním z těchto způsobů je napsání unit testů pro služby v mikroslužbách, které testují správné chování metod ve službách. Dalším způsobem testování jsou API testy (integrační testy), které volají rozhraní mikroslužby a kontrolují jestli se na daný požadavek vrátila požadovaná odpověď a jestli došlo z požadovaným změnám v datové struktuře.

6.1.2.1 Unit testy

Unit testy byly napsány ve frameworku JUnit [37]. JUnit je jednoduchý a velmi rozšířený framework na testování Java aplikací. K označení testovací metody

slouží anotace `@org.junit.Test`. Ukázka takového unit testu je v ukázce zdrojového kódu 9. Dalším důležitým frameworkem pro Unit testy mikroslužeb je Mockito [38]. Mockito umožňuje tzv. mockování (nahrazení komponenty za jinou s předem daným chováním). Mockování slouží u unit testů k simulaci chování ostatních tříd, kromě té kterou testujeme, předem definovaným chováním. Ukázka mockování v testu je ve zdrojovém kódu 10. Unit testy se v každé mikroslužbě nachází ve složce `/test/unit`.

```
// Tato anotace oznacuje nasledujici metodu jako test
@Test
public void getText_en_shouldReturnTextInEnglish() {
    GetLocalizedTextDto textDto = textsService.
                              getText("test.code", "en");
    GetLocalizedTextDto expectedTextDto =
                                  new GetLocalizedTextDto();
    expectedTextDto.setText("Text in english");
    expectedTextDto.setTextCode("test.code");
    // nasledujici metoda overi, jestli se dane dva objekty
    // shoduji a pokud ne, tak test selze
    assertEquals(expectedTextDto, textDto);
}
```

Zdrojový kód 9: Ukázka unit testu v mikroslužbě na správu textů

6.1.2.2 API testy

API testy jsou typem integračních testů, které volají REST API dané mikroslužby a kontrolují odpověď s očekávanou odpovědí a jestli došlo k požadované změně v datové struktuře. API testy používají při svém běhu testovací databázi, ve které pomocí Play Evolutions vytvoří požadovanou datovou strukturu a pomocí SQL skriptu tuto databázi naplní daty. Toto se děje před každým testem a po skončení každého testu tuto databázi opět vymažou. Tímto je zaručeno, že každý test běží nezávisle a není ovlivňovaný žádným dalším testem. Tyto API testy netestují chování více mikroslužeb dohromady a proto jsou ostatní mikroslužby v těchto testech jen simulovány.

K volání API v testech je využíváno funkcionality, kterou poskytuje Play Framework a to konkrétně `play.test.Helpers.fakeRequest()`. Toto je metoda, která umožňuje připravit požadavek na nějaké rozhraní. Tento požadavek umožňuje zavolat metoda `play.test.Helpers.route(app, request)`, která vrátí odpověď daného požadavku. Další užitečnou metodou z Helpers balíčku je metoda `play.test.Helpers.contentAsString()` která vrátí tělo odpovědi ve formátu string. Tento string je poté nutné naparsovat na Java objekt. Jelikož se jedná o JSON formát, tak toto parsování je možné udělat přes

```java
// tato anotace zajisti, ze se promenne onanotovane
// jako @Mock inicializuji jako mock
@RunWith(MockitoJUnitRunner.class)
public class TextsServiceTest {
    private TextsService textsService;
    @Mock
    private TextsDao textDaoMock;
    // metoda onanotovana @Before se spusti pred
    // kazdym testem a slouzi k nastavovani
    @Before
    public void setUp() {
        // vytvori TextService, ktera neobsahuje
        // skutecne DAO ale jeho mock
        textsService = new TextsService(textDaoMock);
    }

    @Test
    public void getTexts_noTextsInDb_shouldReturnEmptyList() {
        // metoda when nastavuje co ma mockovana
        // metoda vracet, pokud je zavolana
        when(textDaoMock.findAll()).
                        thenReturn(Collections.emptyList());
        List<GetTextDto> list = textsService.getTexts();
        assertEquals(Collections.emptyList(), list);
    }
}
```

Zdrojový kód 10: Ukázka mockování v testu v mikroslužbě na správu textů

`com.fasterxml.jackson.databind.ObjectMapper`. Ukázka API testu je ve zdrojovém kódu 11.

6.2 Dokumentace

Každá mikroslužba je zdokumentovaná pomocí JavaDoc [39] dokumentace. Tato dokumentace popisuje téměř každou metodu, k čemu slouží, jaké má parametry, co vrací a jaké výjimky mohou nastat. Tento styl dokumentace také podporuje většina IDE (vývojových prostředí) při našeptávání při psaní kódu.

Další způsobem dokumentace mikroslužeb je README soubor v každé mikroslužbě, který popisuje základní vlastnosti dané mikroslužby, jak jí spustit, jak jí nakonfigurovat, jak spustit testy a popis struktury složek. Tento soubor je ve formátu Markdown.

```
@Test
public void getTexts_ok_shouldReturnTexts()
                                throws JsonProcessingException {
    Http.RequestBuilder request = Helpers.fakeRequest().
                                      method(GET).uri("/texts");
    Result result = route(app, request);
    assertEquals(OK, result.status());
    String content = Helpers.contentAsString(result);
    GetTextDto[] list = new ObjectMapper().
                          readValue(content, GetTextDto[].class);
    assertEquals(3, list.length);
}
```

Zdrojový kód 11: Ukázka API testu v mikroslužbě na správu textů

Psaní dokumentace na Wiki, která v Uniqway projektu existuje, je dalším způsobem dokumentování. Do této dokumentace se píší obecnější věci, které se nehodí psát přímo do kódu nebo do jednotlivých repozitářů k mikroslužbám. Wiki dokumentaci je možné nalézt v příloze této práce. Tato dokumentace popisuje následující:

- konvence při psaní kódu,
- diagram komunikace mikroslužeb,
- generování kódu pomocí obou generátorů (OpenAPI generátor a JOOQ generátor),
- spouštění a testování,
- doporučenou adresářovou strukturu, ve které mikroslužby mít,
- tvorbu nové mikroslužby a
- návrh zpráv při komunikaci typu publikování/odebírání, pokud na tento přechod někdy dojde.

API dokumentace ve formátu OpenAPI [33] je vytvořena pro každou mikroslužbu a nachází se v repozitáři, který obsahuje API dokumentaci všech mikroslužeb. Z této dokumentace se generuje kód mikroslužeb pomocí OpenAPI generátoru [30]. Při generování také vzniká soubor se samotnou dokumentací v repozitáři každé mikroslužby v souboru `/public/openapi.json`. Tato dokumentace se také nachází v příloze této práce.

Každá tabulka je zdokumentovaná pomocí komentářů přímo v databázi. Tyto komentáře popisují k čemu daná tabulka je a k ukládaní jakých dat jsou její sloupce.

Závěr

V rámci této diplomové práce vznikla analýza, návrh a částečná implementace transformace serverové aplikace carsharingového systému Uniqway z monolitické architektury do architektury mikroslužeb.

Zadání diplomové práce bylo splněno a naplnění jednotlivých cílů bude nyní blíže popsáno. První cílem diplomové práce byla tvorba rešerše vhodných způsobů transformace. Tato rešerše se nachází v kapitole 2.4. Dalším cílem této práce byla analýza současného řešení serverové aplikace Uniqway. Ta se nachází v kapitole 3. Návrh přechodu mezi architekturami zaměřený na mikroslužbu spravující texty a lokalizace, obchod s odměnami a tvorbu a správu rezervací je obsahem kapitoly 4 a tímto je splněný další z dílčích cílů této práce. Cíl otestování a zdokumentování vzniklého řešení je splněný a popis, jak toho bylo docíleno, je v sekcích 6.1 a 6.2. Poslední cíl této práce, a to návrh budoucích kroků, jak transformaci dokončit, je v následující podkapitole „Budoucí práce“.

Transformace mezi různými architekturami není vůbec jednoduchá záležitost a proto není divu, že v rámci této práce nebyla transformace dokončená. V této práci vznikla implementace dvou mikroslužeb (mikroslužba na správu textů a lokalizací a mikroslužba na správu faktur), implementace šablony mikroslužeb, která slouží k rychlému vytvoření základu pro novou mikroslužbu, a co je hlavní, vznikla analýza a návrh jak v této transformaci pokračovat a jak jí dokončit.

Budoucí práce

Jak již bylo řečeno, provést takovouto transformaci je velmi náročná záležitost a k jejímu dokončení zbývá ještě spousta práce. V této části bude popsáno jak tuto transformaci dokončit a jaké kroky jsou k tomu potřeba.

1. Prvním krokem k dokončení transformace je nasazení vzniklých mikroslužeb na Uniqway infrastrukturu.

2. Druhým krokem je upravit existující monolitickou serverovou aplikaci, aby požadavky na funkcionality, které již v mikroslužbách vznikly, přeposílala na tyto mikroslužby.

3. Po tomto přeposílání je nutné tyto funkcionality z monolitické aplikace odstranit.

4. Dalším krokem bude implementace mikroslužby navržené v rámci této práce. Jedná se o mikroslužbu, která spravuje rezervace vozidel a mikroslužby starající se o chod obchodu s odměnami. Mikroslužby spravující obchod s odměnami jsou následující: produktová mikroslužba, mikroslužba na objednávky, mikroslužba spravující skladového hospodářství a mikroslužba spravující odměny uživatelů. Tyto mikroslužby nebyly implementovány v rámci této práce z důvodu komplexity celé transformace a časových možností v rámci projektu.

5. Po implementaci těchto mikroslužeb bude potřeba zopakovat kroky 1-3, pro nově vzniklé funkcionality.

6. Nakonec bude potřeba navrhnout a implementovat do mikroslužeb i zbývající funkcionalitu, která v monolitické aplikaci zbývá a pro ni opět opakovat kroky 1-3.

Proces transformace je zdlouhavý a iterativní proces, kdy se vždy část systému přepíše do mikroslužby, mikroslužba se nasadí, funkcionalita se přesměruje do nové mikroslužby a poté se odstraní z monolitu. Pokud Uniqway tým bude dodržovat tato pravidla, tak věřím, že se transformaci úspěšně podaří dokončit a přinese to výše zmiňované výhody, které architektura mikroslužeb nabízí.

Bibliografie

1. VACHULA, Richard. *Databáze a rozhraní pro modul automobilu pro systém sdílení automobilů více uživateli*. 2017. bakalářská práce. FEL ČVUT.
2. CHRISTENSSON, P. API Definition. *Tech Terms* [online]. 2016 [cit. 2021-03-09]. Dostupné z: `https://techterms.com/definition/api`. (Překlad autora).
3. FIELDING, Roy; GETTYS, Jim; MOGUL, Jeffrey; FRYSTYK, Henrik; MASINTER, Larry; LEACH, Paul; BERNERS-LEE, Tim. *Hypertext transfer protocol–HTTP/1.1*. 1999. Tech. zpr. (Překlad autora).
4. BRAY, T. Ed. The JavaScript Object Notation (JSON) Data Interchange Format. *JSON* [online]. 2017 [cit. 2021-03-09]. ISSN 2070-1721. Dostupné z: `https://www.rfc-editor.org/rfc/pdfrfc/rfc8259.txt.pdf`. (Překlad autora).
5. F5 INC. Load Balancer. *F5 Glossary* [online]. 2021 [cit. 2021-03-11]. Dostupné z: `https://www.codecademy.com/articles/what-is-rest`. (Překlad autora).
6. CHRISTENSSON, P. XML Definition. *Tech Terms* [online]. 2006 [cit. 2021-03-09]. Dostupné z: `https://techterms.com/definition/xml`. (Překlad autora).
7. CHRISTENSSON, P. MVC. *Tech Terms* [online]. 2018 [cit. 2021-03-29]. Dostupné z: `https://techterms.com/definition/mvc`. (Překlad autora).
8. CODECADEMY. What is REST? *Codecademy* [online]. 2021 [cit. 2021-03-09]. Dostupné z: `https://www.codecademy.com/articles/what-is-rest`. (Překlad autora).

9. MOZILLA CONTRIBUTORS. HTTP response status codes. *MDN Web Docs* [online]. 2021 [cit. 2021-03-09]. Dostupné z: `https://developer.mozilla.org/en-US/docs/Web/HTTP/Status`. (Překlad autora).

10. SIRAJ UL HAQ. Introduction to Monolithic Architecture and Micro-Services Architecture. *Medium KoderLabs* [online]. 2018 [cit. 2021-02-08]. Dostupné z: `https://medium.com/koderlabs/introduction-to-monolithic-architecture-and-microservices-architecture-b211a5955c63`. (Překlad autora).

11. CONSERVANCY, Software Freedom. *Selenium* [online]. 2021 [cit. 2021-02-19]. Dostupné z: `https://www.selenium.dev/`. (Překlad autora).

12. SCHMIDT, Douglas C. *Applying Patterns and Frameworks to Develop Object-Oriented Communication Software*. 1997. Tech. zpr. Department of Computer Science Washington University, St. Louis, MO 63130. (Překlad autora).

13. RICHARDSON Chris with Smith, Floyd. *Microservices - From Design to Deployment*. 1st. NGINX, Inc., 2016. Dostupné také z: `https://www.nginx.com/resources/library/designing-deploying-microservices/`. (Překlad autora).

14. FOWLER Martin with Lewis, James. Microservices. *Martin Fowler* [online]. 2014 [cit. 2021-02-08]. Dostupné z: `https://martinfowler.com/articles/microservices.html`. (Překlad autora).

15. AMAZON WEB SERVICES, Inc. *Elastic Load Balancing* [online]. 2021 [cit. 2021-03-07]. Dostupné z: `https://aws.amazon.com/elasticloadbalancing/?elb-whats-new.sort-by=item.additionalFields.postDateTime&elb-whats-new.sort-order=desc`. (Překlad autora).

16. MICROSOFT. *Decompose a monolithic application into a microservices architecture* [online]. 2021 [cit. 2021-03-11]. Dostupné z: `https://docs.microsoft.com/en-us/learn/modules/microservices-architecture/5-analyze-decompose`. (Překlad autora).

17. LIGHTBEND. *Play Framework* [online]. 2021 [cit. 2021-03-12]. Dostupné z: `https://www.playframework.com/`. (Překlad autora).

18. ORACLE. *Java* [online]. 2021 [cit. 2021-03-12]. Dostupné z: `https://www.java.com/en/`. (Překlad autora).

19. THE POSTGRESQL GLOBAL DEVELOPMENT GROUP. *PostgreSQL* [online]. 2021 [cit. 2021-03-12]. Dostupné z: `https://www.postgresql.org/`. (Překlad autora).

20. EBEAN ORM. *Ebean ORM* [online]. 2021 [cit. 2021-03-15]. Dostupné z: `https://ebean.io/`. (Překlad autora).

21. AMAZON WEB SERVICES, Inc. *Amazon S3* [online]. 2021 [cit. 2021-04-19]. Dostupné z: `https://aws.amazon.com/s3/`. (Překlad autora).

22. LIGHTBEND. *Internationalization with Messages* [online]. 2021 [cit. 2021-03-29]. Dostupné z: https://www.playframework.com/documentation/2.8.x/JavaI18N. (Překlad autora).

23. AMAZON WEB SERVICES, Inc. *AWS* [online]. 2021 [cit. 2021-04-07]. Dostupné z: https://aws.amazon.com/. (Překlad autora).

24. AMAZON WEB SERVICES, Inc. *Elastic Container Service* [online]. 2021 [cit. 2021-04-07]. Dostupné z: https://docs.aws.amazon.com/ecs/index.html. (Překlad autora).

25. AMAZON WEB SERVICES, Inc. *Service Discovery* [online]. 2021 [cit. 2021-04-07]. Dostupné z: https://docs.aws.amazon.com/AmazonECS/latest/developerguide/service-discovery.html. (Překlad autora).

26. ERICCSON, Ulf. Why is the difference between functional and Non-functional requirements important? *ReQtest* [online]. 2012 [cit. 2021-03-18]. Dostupné z: https://reqtest.com/requirements-blog/functional-vs-non-functional-requirements/. (Překlad autora).

27. RICHARDSON, Chris. Pattern: Microservice chassis. *Microservice Architecture* [online]. 2020 [cit. 2021-04-06]. Dostupné z: https://microservices.io/patterns/microservice-chassis.html. (Překlad autora).

28. BITTNER, K.; SPENCE, I.; JACOBSON, I. *Use Case Modeling*. Addison Wesley, 2003. The Addison-Wesley object technology series. ISBN 9780201709131. Dostupné také z: https://books.google.cz/books?id=zvxfXvEcQjUC.

29. GUGUERE, Kevin. Source-Code Generation. *Medium SSENSE TECH* [online]. 2020 [cit. 2021-05-01]. Dostupné z: https://medium.com/ssense-tech/source-code-generation-43d0cc5594cc. (Překlad autora).

30. OPENAPI-GENERATOR CONTRIBUTORS. *OpenAPI Generator* [online]. 2021 [cit. 2021-04-06]. Dostupné z: https://openapi-generator.tech/. (Překlad autora).

31. DATA GEEKERY™ GMBH. *JOOQ* [online]. 2021 [cit. 2021-04-08]. Dostupné z: https://www.jooq.org/. (Překlad autora).

32. LIGHTBEND. *Managing database evolutions* [online]. 2021 [cit. 2021-04-11]. Dostupné z: https://www.playframework.com/documentation/2.8.x/Evolutions#Managing-database-evolutions. (Překlad autora).

33. THE LINUX FOUNDATION®. *OpenAPI Specification* [online]. 2020 [cit. 2021-04-06]. Dostupné z: https://spec.openapis.org/oas/v3.1.0. (Překlad autora).

34. LIGHTBEND. *Controllers* [online]. 2021 [cit. 2021-04-06]. Dostupné z: https://www.playframework.com/documentation/2.8.x/JavaActions#Controllers. (Překlad autora).

35. SQUARE, INC. *Retrofit* [online]. 2021 [cit. 2021-04-11]. Dostupné z: `https://square.github.io/retrofit/`. (Překlad autora).

36. MLEJNEK, Ing. Jiří. *Testování aplikací, přednáška 9, Softwarové inženýrství BI-SI1.* Fakulta informačních technologií, katedra Softwarového inženýrství, ČVUT v Praze, 2019.

37. JUNIT. *JUnit 4* [online]. 2021 [cit. 2021-04-08]. Dostupné z: `https://junit.org/junit4/`. (Překlad autora).

38. FABER, Szczepan et al. *Mockito* [online]. 2021 [cit. 2021-04-08]. Dostupné z: `https://site.mockito.org/`. (Překlad autora).

39. ORACLE. *Javadoc* [online]. 2017 [cit. 2021-04-08]. Dostupné z: `https://docs.oracle.com/javase/9/javadoc/javadoc.htm#JSJAV-GUID-7A344353-3BBF-45C4-8B28-15025DDCC643`. (Překlad autora).

Příloha A

Seznam použitých zkratek

API Application programming interface

DAO Data Access Object

DTO Data Transfer Object

HTML HyperText Markup Language

HTTP HyperText Transfer Protocol

IDE Integrated Development Environment

IPC Inter-process Communication

JSON JavaScript Object Notation

MVC Model View Controller

ORM Object-relation Mapping (Objektově relační mapování)

POJO Plain Old Java Object

REST REpresentational State Transfer

UI User Interface

URL Uniform Resource Locator

XML Extensible Markup Language

Příloha B

API stávajícího řešení

V této části přílohy budou více popsány jednotlivé endpointy současného řešení serverové aplikace Uniqway, které nejsou v práci uvedeny.

B.1 Rezervace

Administrátorské API má za možnost zobrazit informaci o všech rezervacích, o konkrétní rezervaci, vytvořit rezervaci konkrétnímu uživateli, ukončit konkrétní rezervaci a zjistit možnosti ukončení pro konkrétní rezervaci. Tyto požadavky jsou následující:

všechny rezervace V těle odpovědi se vrací pole všech rezervací. Informace jsou obdobné jako u seznamu rezervací daného uživatele. Navíc je u každé rezervace informace o uživateli, který danou rezervaci vytvořil.
Metoda HTTP: GET
URL: `/reservations`
Očekávaná odpověď: 200 OK

konkrétní rezervace V těle odpovědi se vrací informace o dané rezervaci. Informace jsou obdobné jako u seznamu všech rezervací.
Metoda HTTP: GET
URL: `/reservations/{id}`
Očekávaná odpověď: 200 OK

tvorba rezervace daného uživatele Tělo požadavku je stejné jako u tvorby rezervace daným uživatelem. Uživatel, pro kterého se má rezervace vytvořit je uveden v URL. V těle odpovědi se vrací id vzniklé rezervace.
Metoda HTTP: POST
URL: `/reservations/{userId}`
Očekávaná odpověď: 200 OK

ukončení rezervace V těle požadavku se zasílá cílový stav, do kterého se má vozidlo po ukončení přepnout. V těle odpovědi se vrací informace o ukončené rezervaci stejné jako při ukončení uživatelem.
Metoda HTTP: POST
URL: `/reservations/{id}/finish`
Očekávaná odpověď: 200 OK

možnosti ukončení rezervace V těle odpovědi se vrací pole možných stavů aut, do kterých je možné auto po ukončení přepnout. Stejné jako u endpointu pro klienty.
Metoda HTTP: GET
URL: `/reservations/{id}/finish-options`
Očekávaná odpověď: 200 OK

B.2 Obchod s odměnami

Administrátorské rozhraní obchodu s odměnami je mnohem rozsáhlejší než klientské a obsahuje velké množství různých endpointů. Z tohoto důvodu nejsou tyto endpointy popsány tak detailně, jako předchozí. Celou API dokumentaci je možné najít v elektronické příloze. Stručný popis těchto požadavků je následující:

GET /rewards Získá všechny existující bodové odměny.

POST /rewards Vytvoří novou odměnu.

GET /rewards/{id} Získá detail dané odměny.

GET /reward-transactions Získá všechny odměnové transakce.

POST /reward-transactions Vytvoří novou odměnovou transakci.

GET /reward-transactions/{id} Získá odměnovou transakci podle zadaného id.

GET /rewardable-actvities Získá všechny existující odměnitelné aktivity.

POST /rewardable-actvities Vytvoří novou odměnitelnou aktivitu.

GET /rewardable-actvities/{id} Získá odměnitelnou aktivitu podle zadaného id.

GET /shop/pickup-places Získá všechny výdejní místa.

POST /shop/pickup-places Vytvoří nové výdejní místo.

GET /shop/pickup-places/{id} Získá výdejní místo podle zadaného id.

GET /shop/orders Získá všechny objednávky.

GET /shop/orders/{id} Získá objednávku podle zadaného id.

PUT /shop/orders/{id}/state Změní stav zadané objednávky.

GET /shop/orders/states Získá všechny stavy objednávek.

GET /shop/products Získá všechny produkty.

POST /shop/products Vytvoří nový produkt.

GET /shop/products/{id} Získá produkt podle zadaného id.

PUT /shop/products/{id} Změní produkt s daným id.

POST /shop/products/{id}/images Přidá produktu s daným id nový obrázek.

GET /shop/products/attribute-types Získá všechny typy atributů produktů.

GET /shop/products/options Získá všechny vlastnosti produktů.

GET /shop/products/{id}/variants Získá varianty zadaného produktu.

GET /shop/categories Získá všechny kategorie produktů.

GET /shop/stock-receipts Získá všechny naskladnění produktů.

GET /shop/stock-receipts/{id} Získá naskladnění podle jeho id.

POST /shop/stock-receipts Vytvoří nové naskladnění.

GET /shop/stock-removals Získá všechny vyskladnění produktů.

POST /shop/stock-removals Vytvoří nové vyskladnění.

GET /shop/stock-removals/{id} Získá vyskladnění s daným id.

GET /shop/stock-removals/types Získá všechny typy vyskladnění.

GET /shop/stock/products Získá stav skladu produktů. Jedná se o součet všech množství přes všechny výdejní místa.

GET /shop/stock/products/{id} Získá stav skladu produktu s daným id.

Příloha C

Obsah přiloženého CD

- `readme.txt` ..... stručný popis obsahu CD
- `src`
  - `impl` ..... zdrojové kódy implementace a testů
  - `thesis` ..... zdrojová forma práce ve formátu LaTeX
  - `documentation` ..... dokumentace vzniklého řešení
    - `microservices-documentation` ..... dokumentace vzniklého řešení
      - `API-documentation` ..... API dokumentace vzniklého řešení
      - `WIKI-documentation` ..... dokumentace vzniklého řešení
    - `current-documentation` ..... dokumentace existujícího řešení
      - `API-documentation` ..... API dokumentace existujícího řešení
- `text` ..... text práce
  - `DP-Severa-Štěpán-2021.pdf` ..... text práce ve formátu PDF
  - `zadani.pdf` ..... zadání práce ve formátu PDF