FAKULTA INFORMAČNÍCH TECHNOLOGIÍ ČVUT V PRAZE

Zadání diplomové práce

Název:	Transformace systému z monolitické architektury do architektury mikroslužeb
Student:	Bc. Petr Prouza
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 autorizací a autentizací uživatele, způsoby notifikace uživatele, komunikací s platební bránou a cenotvorbou.

• Doplňte implementaci o unit testy a API testy.

• Vytvořte dokumentaci přepracovaného kódu.

Elektronicky schválil/a Ing. Michal Valenta, Ph.D. dne 4. ledna 2021 v Praze.

FAKULTA
INFORMAČNÍCH
TECHNOLOGIÍ
ČVUT V PRAZE

Diplomová práce

Transformace systému z monolitické architektury do architektury mikroslužeb

Bc. Petr Prouza

Katedra softwarového inženýrství
Vedoucí práce: Ing. Filip Ravas

6. května 2021

Poděkování

Chci poděkovat svému vedoucímu Ing. Filipovi Ravasovi za cenné rady a připomínky při vedení této diplomové práce. Také chci poděkovat své rodině za podporu v průběhu studia.

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 . . . . . .

České vysoké učení technické v Praze
Fakulta informačních technologií

Odkaz na tuto práci

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

Abstrakt

Práce se zaměřuje na návrh mikroslužeb oddělujících funkcionality z monolitické aplikace. Praktická část se zabývá implementací navrženého postupu transformace do architektury mikroslužeb. Byla vytvořena finální podoba části mikroslužby pro funkcionalitu notifikování uživatele a proces pro transformaci mikroslužeb pro autentizaci a autorizaci požadavků, komunikaci s platební bránou a cenotvorbu.

Klíčová slova monolitická architektura, architektura mikroslužeb, transformace mezi architekturami, Uniqway, Java, Play Framework

Abstract

This thesis focuses on the design of microservices architecture functionality separation from a monolithic application. The practical part deals with implementation of the proposed transformation process. Final version of a part of a microservice for notifications and a transformation process for microservices dealing with authentication and authorization, communication with payment gateway and pricing was created.

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

Obsah

Seznam obrázků

Seznam tabulek

Seznam výpisů kódu

Úvod

Sdílená ekonomika je i přes probíhající pandemii stále oblíbená. Lidé už nepotřebují věci vlastnit, raději si zaplatí (v určitých případech) za využití věci sdílené. Služba Uniqway vyvíjená studenty poskytuje sdílené automobily pro studenty a zaměstnance českých vysokých škol.

Sdílení automobilů má pozitivní vliv i na životní prostředí. Půjčované automobily jsou novější než většina soukromých automobilů. Novější vozy musí splňovat přísnější ekologické normy a jejich provoz je šetrnější než používání starších soukromých automobilů.

Obvyklé využití soukromých automobilů je poměrně malé a většinu času stojí na parkovištích nebo v garážích. Takový obrázek je typický pro většinu větších měst. Pokud budou lidé používat sdílené automobily, počet automobilů ve městech by se mohl snížit a počet volných parkovích míst zvýšit.

Výsledek práce bude prospěšný pro službu Uniqway a její uživatele. Bude umožňovat rychlejší implementaci funkcionalit a změn. Výsledek práce rovněž usnadní obměny vývojového týmu, neboť se nový člen nebude muset zorientovat v celé aplikaci naráz. Aplikace by měla být lépe škálovatelná a tedy umožňovat spravovat velké množství uživatelů a půjčovaných automobilů. Po dokončení transformace by mělo být snazší přidávat nové funkcionality, které uživatelé jistě ocení.

Téma práce bylo vybráno, neboť stav Uniqway serverové aplikace je neuspokojivý. Implementace nových funkcí do aplikace trvá příliš dlouhou dobu a přináší určitou pravděpodobnost, že po změnách některé části přestanou fungovat. Navíc velikost aplikace je pro vývojáře začátečníka (studenta vysoké školy) poměrně zahlcující.

Tato práce dále pokračuje v následující struktuře: Nejdříve se v části 2 věnuje analýze architektur softwarových systémů, konkrétně monolitické architektuře a architektuře mikroslužeb, analýze vhodných způsobů transformace z monolitické architektury do architektury mikroslužeb a popisu a zhodnocení aktuálního stavu serverové aplikace Uniqway. Část 3 představuje návrh

funkcionalit pro použití v jednotlivých mikroslužbách s důrazem na aplikační rozhraní na databázové modely. Nakonec se v části 4 práce věnuje implementaci a testování.

Práce navrhuje transformaci systému, jehož vývoj započal s bakalářskou prací „Databáze a rozhraní pro modul automobilu pro systém sdílení automobil více uživateli“ [1] Ing. Richarda Vachuly. Systém od té doby prošel mnoha změnami a rozšířeními. Současně s touto diplomovou prací pracuje na své diplomové práci student Bc. Štěpán Severa stejného názvu zabývající se transformací jiných součástí monolitického systému. S autorem zmíněné práce se bude při tvorbě společných implementačních částí úzce spolupracovat.

Kapitola 1

Cíl práce

Cílem této diplomové práce je transformovat backendový systém služby sdílení automobilů z monolitické architektury do architektury mikroslužeb. Očekávaným benefitem je možnost snazšího zaučování nových členů vývojového týmu (nebudou muset znát celý systém) a rychlejšího doručování nových funkcí (díky nižší provázanosti bude snazší systém rozšiřovat).

- Prvním cílem je provést řešerši vhodných způsobů přístupu ke transformaci systému z monolitické architektury do architektury mikroslužeb.
- Druhým cílem je zanalyzovat aktuální verzi systému.
- Následně je nutné navrhnout proces, kterým se transformace provede.
- Dalším cílem je začít navrhnutý proces provádět a zaměřit se přitom na části systému věnující se autorizaci a autentizaci uživatelů, notifikacím uživatelů, komunikaci s platební bránou a cenotvorbě.
- Posledním cílem je aplikaci řádně otestovat a zdokumentovat.

Kapitola 2

Analýza

Následující podkapitoly představují službu Uniqway, analyzují webové služby a architektury softwarových systémů. Popisují možnosti škálování a monolitickou architekturu v porovnání s architekturou mikroslužeb. Po podrobné analýze architektury mikroslužeb následuje kapitola popisující možné způsoby transformace z monolitické architektury. V poslední podkapitole je analyzována stávající serverová aplikace Uniqway, která v rámci práce podstoupí částečnou transformaci architektury.

2.1 Představení služby Uniqway

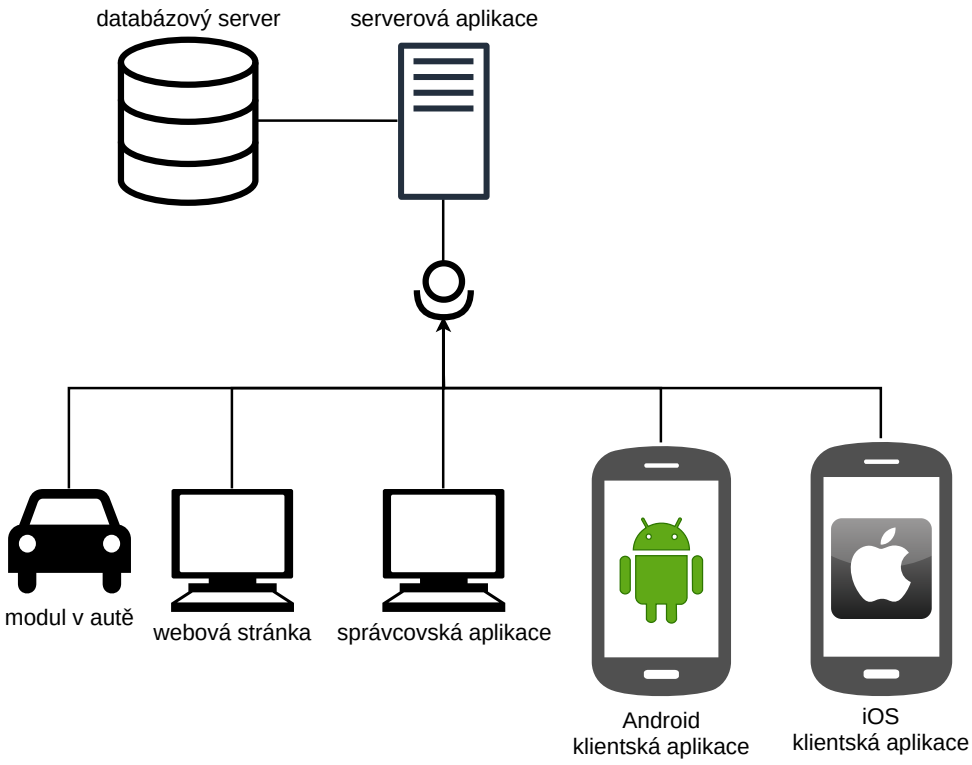
Službu Uniqway lze klasifikovat jako tzv. „free-floating carsharing“, tedy sdílení automobilů, které vlastní právnická osoba a jejich účel je právě sdílení mezi uživateli služby. Druhou variantou sdílení automobilů je tzv. „peer-to-peer“ sdílení. [2] Reprezentantem tohoto typu sdílení je služba HoppyGo.

Služba Uniqway poskytuje své služby studentům a zaměstnancům všech českých vysokých škol. Na vývoji, provozu a propagaci se podílí studenti z Českého vysokého učení technického, České zemědělské univerzity a Vysoké školy ekonomické v daném pořadí. Prvotní myšlenka a vývoj služby začal na jaře roku 2015, oficiální spuštění služby proběhlo po tříměsíčním testovacím provozu 17. října 2018. [3]

Služba kromě poskytování snadné a levné mobility nabízí studentům i možnost realizovat se ve studentském týmu. Mojí předchozí realizací je například navržení a implementace obchodu s odměnami pro Android platformu v rámci bakalářské práce [4]. Při vývoji projektu vzniklo i několik dalších závěrečných prací.

2.2 Popis celého Uniqway systému

Systém je logicky rozdělen do několika propojených částí. Jádrem systému je serverová aplikace společně s relační databází. Monolitická serverová aplikace vystavuje REST rozhraní, skrze které s ní komunikují ostatní části systému. Uživatelé se službou interagují skrze nativní klientské aplikace pro Android a iOS. Vybraná data ze serverové aplikace získává i webová stránka služby (uniqway.cz). Ta navíc umožňuje provedení obou kroků registrace (samotná registrace a verifikace). Pro ambasadory starající se o bezproblémový chod služby existuje správcovská webová aplikace, pro kterou je také vytvořené REST rozhraní na serveru. Posledním článkem systému je hardwarový modul v automobilu, který umožňuje např. jeho odemykání/zamykání s pomocí mobilních aplikací a předává data serveru, např. pro vyhodnocování cen rezervací.



Obrázek 2.1: Schéma systému Uniqway

Serverová aplikace se integruje s platební bránou, díky čemuž umožňuje automatické zaplacení za službu po ukončení rezervace. Rovněž vystavuje rozhraní pro služby třetích stran. Takovou službou je například Citymove. Ně-

která analytická data a dřívější data z automobilů (jeden z výsledků optimalizace výkonu systému, který vzešel z diplomové práce Ing. Filipa Ravase [5]) jsou ukládána do indexů v rámci služby *elastic search*.

2.3 Webové služby

Serverová aplikace Uniqway je instancí webové služby implementované s využitím monolitické architektury (více později v kapitole 2.6). Její REST rozhraní je určeno pro strojové zpracování ostatními částmi Uniqway platformy. Vhodnou definici webové služby lze nalézt v [6] „*Webová služba (anglicky Web Service) je zapouzdřená logika (funkcionalita) jedné aplikace, kterou využívá jiná aplikace. Jedná se tedy o strojovou interakci – webové služby NEJSOU určeny pro lidskou interakci (např. prohlížení ve webovém prohlížeči). Vystavená webová služba má definované veřejné rozhraní a funkce (prostřednictvím jazyka WSDL), které jsou druhé aplikaci předem známy. Webová služba tedy umožňuje interakci dvou aplikací, nejčastěji pomocí jazyka XML protokolem SOAP (prostřednictvím protokolu HTTP, SMTP...). Existují ale i jiné způsoby komunikace, například pomocí protokolu REST*“.

2.4 Architektury softwarových systémů

Jak je zmíněno v [7], architektura softwarového systému je nejasně definovaný pojem, nicméně popisuje jeden z nejdůležitějších aspektů vnitřního návrhu softwarových systémů. Dobrá architektura je nezbytná, jinak hrozí ztížení a zpomalení přidávání nových schopností a funkcionalit do daného systému.

V dalších částech [7] se lze dočíst, jaké běžné definice architektury softwarového systému se v oblasti vývoje software používají. Některými z nich jsou například *základní uspořádání systému*, nebo také *způsob, jakým spolu systémové komponenty, při pohledu s vysokou úrovní abstrakce, komunikují*. V těchto definicích je poté problém s uchopením pojmu *základní uspořádání* nebo ve druhé z nich požadovaná *úroveň abstrakce*. A tedy vhodnější definicí by mohlo být *společné chápání návrhu systému, kterým disponují zkušení vývojáři*.

Další používanou definicí architektury software je také *návrhová rozhodnutí, která je nutné provést v začátcích vývoje projektu*. Nicméně správná rozhodnutí se dají vyhodnotit pouze zpětně, a tedy je vhodnější definovat architekturu pomocí *rozhodnutí, která bychom rádi správně rozhodli v začátcích projektu*. Ale tato definice pro nás není moc užitečná.

Nakonec je tedy dle [7] nejvhodnější následující definice. *Architektura je to důležité, ať je to, co je to*. Ačkoliv je tato definice vágní, zdá se býti vhodnou. Vyplývá z ní, že pokud chceme mít dobrou architekturu software, musíme se rozhodnout, jaké součásti jsou důležité a musíme vynaložit nejvyšší úsilí, abychom právě tyto architektonické součásti udržovali v dobré kondici.

2.5 Škálování

Škálování aplikací je schopnost aplikace zvládnout zpracovat zvětšující se množství požadavků [8]. Dle [9] lze aplikace škálovat třemi různými způsoby, kde každý způsob je popisován jako jedna osa 3D krychle. V dalších kapitolách zmiňované architektury umožňují pouze některé formy škálování. Monolitická architektura umožňuje pouze škálování podle osy X. Architektura mikroslužeb umožňuje škálovat podle os X a Y.

2.5.1 Škálování podle osy X

Někdy také nazývané horizontální škálování. Principem je běh několika kopií aplikace současně, kde každá kopie (někdy také *instance*) obsluhuje pouze část příchozích požadavků. Jednotlivé požadavky přiděluje komponenta k tomu určená nazývaná *load balancer* (volně přeložitelné jako vyvažovač zátěže).

Jedna z nevýhod tohoty typu škálování je požadavek všech instancí na přístup ke všem datům (databázi). Tento požadavek lze částečně řešit zvětšením skryté paměti (*cache*). Problémy, které škálování podle osy X nevyřeší, jsou vzrůstající komplexita vývoje a samotné aplikace. [9]

2.5.2 Škálování podle osy Y

Škálování podle osy Y rozděluje aplikaci do několika menších celků – služeb. Každá služba je zodpovědná za jednu nebo více funkcí, které spolu souvisí. [9] Více o způsobech dekompozice v kapitole 2.7.1.

2.5.3 Škálování podle osy Z

Je podobné škálování podle osy X, neboť využívá běhu několika identických kopií aplikace. Hlavní rozdíl je nicméně v tom, že každá kopie se stará pouze o podmnožinu dat. Například jedna z kopií může zpracovávat požadavky jedné jednoznačně definované skupiny zákazníků a druhá kopie o požadavky jiné skupiny nebo zbývajících zákazníků.

Škálování podle osy Z se často používá u databází (v kontextu databází také nazývané *sharding*). V takovém případě jsou data rozdělena na více databázových serverů, například podle primárního klíče. Toto může být navíc kombinováno se škálováním podle osy X. Potom běží několik kopií, které se starají o danou podmnožinu dat současně. Typickým použitím škálování podle osy Z v kontextu aplikací je implementace vyhledávače. [9]

2.6 Monolitická architektura

Aplikace vyvíjené s využitím monolitické architektury jsou obvykle udržovány v rámci jednoho repozitáře systému pro správu verzí, přičemž se aplikace na-

sazuje jako jeden celek. Aplikace obvykle vystavuje rozhraní, přes které s ní komunikují klientské aplikace (např. aplikace ve webovém prohlížeči, nativní mobilní aplikace, či desktopové aplikace). V případě potřeby se aplikace integruje s nějakými dalšími třetími stranami (např. integrace s platební bránou) nebo vystavuje rozhraní pro třetí strany (např. integrace se Citymove). Typické rysy monolitických aplikací jsou následující [10]:

Komunikace skrze HTTP Obvykle ke komunikaci využívá protokol HTTP a data jsou obvykle přenášena ve formátech HTML, JSON nebo XML.

Členění do vrstev a komponent Zpravidla je monolitická aplikace dělena do vrstev a komponent, kde se jedna z vrstev může starat např. o komunikaci a transformace dat mezi aplikací a persistentním úložištěm, další vrstva o podnikovou logiku aplikace a poslední např. o transformaci dat a komunikaci skrze již zmiňovaný HTTP protokol.

Vydávána jako jediná aplikace Celý kód a všechny jeho součásti jsou nasazeny současně a změna i jen malé časti vyžaduje opětovné nasazení celé aplikace.

Snadný vývoj Vývoj monolitických aplikací je podporován současnými vývojovými prostředími (*IDE*).

Jednodimenzionální škálování Z principu monolitické architektury plyne, že je možné škálovat pouze podle osy X (viz 2.5.1).

Komplexita Velikost aplikace může zastrašit vývojáře, především ty nové. Nejen pro nové vývojáře může být složité pochopit, jak celá aplikace funguje a změny v ní mohou být problematické a jejich implementace může trvat dlouho.

Degradující modularita Vzhledem k neexistenci striktních hranic mezi moduly, přirozeně klesá modularita při postupném vývoji a údržbě.

Rozsáhlý kód aplikace Velikost projektu může zpomalovat vývojové prostředí a spouštění aplikace trvá delší dobu.

Složitá koordinace vývojových týmů Při určité velikosti aplikace může být vhodné rozdělit jednotlivé zodpovědnosti aplikace a jejich správu „odděleným“ vývojovým týmům. V takovém případě ale musí koordinovat své úsilí při vydávání nových verzí.

Závislost na zvolené technologii Zvolené technologie musí využívat každá součást aplikace, i když by pro některé mohly být jiné technologie vhodnější. Problémem také může být zastarání použité technologie a nutnost přepsání celého kódu do technologie novější.

2.7 Architektura mikroslužeb

Architektura mikroslužeb popisuje takovou aplikaci, kde se k plnění jednotlivých úkolů / případů užití využívá kooperace několika relativně malých málo provázaných služeb (aplikací). Architektura mikroslužeb je ukázková implementace škálování podle osy Y. O každé mikroslužbě, která je součástí systému by mělo platit následující [11]:

Snadno udržovatelná a testovatelná Dovoluje rychlý vývoj a časté vydávání.

Volně provázaná s ostatními mikroslužbami Umožňuje týmu spravovat mikroslužbu po většinu času nezávisle na ostatních mikroslužbách, neboť změny v mikroslužbě obvykle nemají na ostatní mikroslužby vliv.

Nezávisle vydávatelná Tým většinou může vydávat nové verze své mikroslužby, protože vydávané změny ostatní mikroslužby neovlivní.

Spravovatelná malým týmem Vzhledem k relativně malému rozsahu kódu je mikroslužbu schopen spravovat malý tým, u kterého (oproti správě rozsáhlé monolitické aplikace) rovněž klesnou režijní náklady vzniklé z nutné komunikace mezi členy.

Z použití architektury mikroslužeb vzniká několik problémů, které je nutné řešit. Jejich popis a možná řešení budou popsána v následujících kapitolách.

2.7.1 Dekomponování systému

Ať už vyrábíme novou aplikaci, nebo transformujeme monolitickou aplikaci do architektury mikroslužeb, musíme správně rozdělit odpovědnosti jednotlivých součástí tak, aby byly jednotlivé mikroslužby co nejméně závislé na ostatních a pokud už závislé být musí, je vhodné když dané rozhraní, přes které komunikují nebude mít potřebu se měnit. Potřeby vhodné dekompozice jsou následující [12, 13]:

- Architektura musí být stabilní. Nesmí se moc měnit. Chceme provádět co nejméně změn v rozhraních, přes které spolu služby komunikují.
- Rozdělení mikroslužeb dodržuje pravidlo vysoké soudržnosti. Každá mikroslužba implementuje relativně malou množinu souvisejících funkcí.
- Mikroslužby jsou děleny tak, aby změny, které mohou nastat ovlivnili pouze jednu mikroslužbu. Takový přístup způsobí, že vydávání změn není závislé na změnách v jiných mikroslužbách.
- Mikroslužby musí být volně provázané. Každá mikroslužba vystavuje rozhraní, kterým zapouzdřuje svou implementaci. Změna by neměla ovlivnit uživatele vystaveného rozhraní.

- Mikroslužby by měly být snadno testovatelné.
- Každá mikroslužba by měla být vyvíjena jedním týmem o velikosti maximálně 10 vývojářů [12].
- Každý tým by měl fungovat autonomně a vyvíjet a vydávat svou mikroslužbu bez nutnosti velké kooperace s týmy vyvíjejícími ostatní mikroslužby.

Dle [9, 11, 12, 13] lze dekomponovat do služeb různými způsoby. Můžeme dekomponovat dle :

sloves (*verb-based decomposition*) Každá mikroslužba implementuje jeden případ užití, například *zaplatit objednávku*

podstatných jmen (*noun-based decomposition*) Mikroslužba je zodpovědná za všechny operace související s danou entitou, například *správa zákazníků*.

podnikových schopností (*business capability*) Definuje mikroslužby na základě toho, co podnik dělá za účelem tvorby hodnoty. Například *vývoj* nebo *doručování produktů*.

subdomén doménou řízeného návrhu (*subdomains of domain-driven design*) Mikroslužby zde implementují funkce související s vybranou subdoménou. Existují tři různé typy subdomén [14]: základní (to, co podnik odlišuje od ostatních), pomocná (doplňuje základní doménu, bez této subdomény základní doména nebude fungovat) a obecná (nespecifická doména, kterou lze snadno koupit jako hotové řešení).

Jednotlivé způsoby dekompozice lze v určitých mezích kombinovat. Podstatné je správně identifikovat tzv. *bounded context* z kontextu doménou řízeného návrhu. Podle [15] tento *bounded context* popisuje část určité domény a jasně specifikuje, v jakém vztahu je s ostatními *bounded context* v dané doméně.

2.7.2 Distribuovaná data

Abychom docílili snadného paralelního vývoje, vydávání a nezávislého škálování, musíme data jednotlivých služeb distribuovat do různých databází. V ideálním případě bude mít každá mikroslužba vlastní databázi.

Z pohledu škálovatelnosti je nejlepším rešením přiřadit každé mikroslužbě vlastní databázový server. Takové řešení je poměrně nákladné na provoz, neboť daná mikroslužba většinou využije zdroje daného databázového serveru jen z malé části. Další možností je jeden databázový server sdílet, potom každá mikroslužba udržuje své vlastní schéma, do kterého má přístup právě tato

jedna mikroslužba. Poslední možností je sdílení databáze, kde přístup k jednotlivým tabulkám jednotlivými mikroslužbami je disjunktní a vyčerpávající. Nesmí dojít k možnosti manipulace s daty z jedné mikroslužby mikroslužbou jinou, jinak vzniknou větší problémy než v jedné velké databázi pro monolitickou aplikaci.

Použitím distribuovaných dat ztrácíme transakční zpracování v rámci aplikační domény na úrovni databáze. Řešením transakcí, které přesahují hranice mikroslužeb, může být implementace Saga návrhového vzoru. [16]

2.7.2.1 Návrhový vzor Saga

Popisuje možnou alternativní implementaci transakce přesahující hranici jedné mikroslužby. Dosahuje této funkcionality s pomocí sekvencí lokálních transakcí (transakce v rámci jedné mikroslužby). Každá lokální transakce aktualizuje persistentní úložiště (databázi) a zveřejní zprávu, nebo událost, na které reagují další mikroslužby svými lokálními transakcemi. Pokud dojde k chybě v lokální transakci, je spuštěn cyklus nápravných transakcí, které zruší změny provedené v předchozích lokálních transakcích v ostatních mikroslužbách.

Aby byla implementace spolehlivá, je nutné, aby služba atomicky aktualizovala databázi a publikovala zprávu/událost. Atomicity aktualizace databáze a publikování zprávy/události můžeme dosáhnout s využitím vzorů *event sourcing*, nebo *transactional outbox*). [17]

2.7.2.2 Event sourcing

Zdroj [18] popisuje *event sourcing* jako alternativní způsob ukládání entit. Entitu ukládá jako sekvenci změn stavů neboli událostí. Při každé změně entity se na konec její sekvence přidá nová událost popisující danou změnu. Toto chování je atomické, neboť vložení záznamu je atomické. Tyto entity jsou ukládány v databázi (tzv. *event store*) a tato databáze zároveň slouží jako zprostředkovatel zpráv / událostí.

Pokud chce mikroslužba získat aktuální stav entity, provede aplikaci všech změn v sekvenci. Aby každá mikroslužba nemusela aplikovat vždy všechny změny, můžeme proces optimalizovat ukládáním průběžných verzí (*snapshot*). Potom mikroslužba pro získání aktuálního stavu entity aplikuje pouze změny, které nastaly po vytvoření průběžné verze. Z použití *event sourcing* plyne několik benefitů:

- umožňuje spolehlivé publikování událostí při změnách,
- vzhledem k tomu, že se ukládají události, nikoliv doménové objekty, z velké části se vyhývá problému různých reprezentací v relačním a objektovém světě (viz *Object–relational impedance mismatch*),
- poskytuje spolehlivý log změn nad entitou,

- umožňuje dotazovat se do minulosti
- a podniková logika založená na *event sourcing* se skládá z volně provázaných entit, což usnadňuje jejich správu.

Aplikace tohoto vzoru představuje ale i několik nevýhod. Především se musíme vypořádávat s extra komplexitou:

- Jedná se o odlišný neobvyklý (ne moc známý) styl programování
- a zmiňovaná databáze událostí se těžko dotazuje, neboť rekonstrukce entit bývá komplexní a neefektivní.

Vhodným rozšířením je použití CQRS (více v 2.7.2.4) a tedy potřeba řešení eventuální konzistence. Alternativní možností je vzor *transactional outbox*.

2.7.2.3 Transactional outbox

Řeší problém atomicity operace uložení změny a publikování zprávy/události podobně jako *event sourcing*. Při použití tohoto vzoru mikroslužba uloží událost do tabulky ve své databázi, odkud jej další proces (tzv. *message relay*) přečte a předá zprostředkovateli zpráv (*message broker*). Zprostředkovatel poté informuje mikrslužby, které jsou k daným zprávám/událostem přihlášeny (*subscribed*). [19] Více o přihlašování ke zprávám/událostem v kapitole 2.7.4 o způsobech komunikace mezi mikroslužbami.

2.7.2.4 API kompozice a CQRS

Po zavedení distribuovaných dat do systému ztrácíme možnost jednoduchých spojování dat napříč doménami jednotlivých mikroslužeb na úrovni databáze – databázi má každá mikroslužba vlastní. Bohužel se bez funkce spojení většinou neobejdeme a musíme tedy využít kompozice API nebo CQRS (*Command Query Responsibility Segregation*). [16]

API kompozice Vybraná mikroslužba, nebo nějaká speciálně pro tento účel vytvořená, provede dotazy na mikroslužby, které pracují s požadovanými daty, a provede spojení ve své operační paměti a výsledek vrátí. Někdy tyto funkce může zastávat i API Gateway (viz kapitola 2.7.3). [20]

CQRS Odděluje (*segragation*) logiku pro změny (*command*) a dotazování (*query*). Jedná se v podstatě o materializovaný pohled nad více databázemi. Vytvořené pohledy jsou pro všechny klienty pouze ke čtení. Jejich úpravy se provádí s pomocí událostí, které jsou publikovány jednotlivými mikroslužbami, které s vybranými entitami pracují. Použití tohoto principu zvyšuje komplexitu systému. Může vynucovat duplikaci kódu a způsobovat zpoždění v aktualizaci dat pro dotazy (eventuální konzistence).

Použitím získáme možnost denormalizovaných pohledů a jednotlivé zodpovědnosti jsou lépe oddělené. CQRS je nutností pro implementaci vzoru *event sourcing*. [21]

2.7.3 Přístup k jednotlivým mikroslužbám

Při zavádění architektury mikroslužeb musíme také rozhodnout, jak se bude přistupovat k jednotlivým mikroslužbám. Při rozhodování musíme dle [22] zohlednit následující problémy.

- Různé části aplikace (mikroslužby) se starají o různé funkcionality.
- Každá mikroslužba se stará o něco jiného a z pohledu klienta někdy poskytuje až příliš podrobné rozhraní.
- Různí klienti mohou vyžadovat různá rozhraní.
- Provedení operace, která vyžaduje provedení několika požadavků, by mohlo být problematické z vysokolatenční sítě. Je tedy lepší pokud cesta od klienta na server a zpět bude absolvována jen jednou a o provedení souvisejících požadavků se postará zvolená komponenta serveru (tj. v rámci lokální sítě).
- Počet instancí jednotlivých mikroslužeb se může měnit, včetně jejich lokací. Každý klient tedy musí znát logiku pro hledání aktivních instancí. Tomu se můžeme vyhnout přidáním prostředníka. Více o objevování služeb v kapitole 2.7.3.3.
- Rozdělení do mikroslužeb se může měnit a nemělo by klienty ovlivnit.
- V určitých případech mohou některé mikroslužby používat ke komunikaci protokoly, které nejsou vhodné pro komunikaci po síti. V takovém případě musíme využít prostředníka, který data přetransformuje a k dalšímu přenosu využije protokol vhodný k přenášení dat po síti.

2.7.3.1 API Gateway

Aby klienti aplikace založené na mikroslužbách nemuseli řešit většinu problémů zmíněných v předchozí kapitole, je vhodné zavést prostředníka, přes kterého budou putovat zasílané požadavky. Zdroj [22] navrhuje použití tzv. *API Gateway*.

API Gateway může buď jen přesměrovat požadavek na správnou mikroslužbu a odpověď přeposlat klientovi, který se dotazoval, nebo může také zastávat funkce pro autentizaci a případně autorizaci. Někdy může dokonce agregovat informace z několika mikroslužeb (viz předchozí kapitola 2.7.2.4 o kompozici rozhraní).

Využití *API Gateway* můžeme posunout i o úroveň dále a rozhraní aplikace rozdělit pod několik různých *API Gateway*, kde jedna se může starat o požadavky z mobilních aplikací, jiná o požadavky z webové aplikace apod. Z použití prostředníka plyne několik výhod.

- Izoluje klienty od logiky rozdělení do mikroslužeb.
- Izoluje klienty od potřeby znát logiku pro objevování služeb.
- Optimalizuje řešení požadavků, které vyžaduje kooperaci více mikroslužeb. Síťová cesta od klienta na server je delší než cesta mezi mikroslužbami.

Zavedení *API Gateway* přináší i nevýhody jako zvýšení komplexity, či teoretické zvýšení latence. Tento prostředník je další součástí, kterou musíme provozovat, vyvíjet a udržovat. Zároveň každý požadavek musí přes tohoto prostředníka projít, což pro jednoduché požadavky může znamenat nepříliš znatelné opoždění.

2.7.3.2 Service registry

Registr služeb [23] je databáze mikroslužeb, jejich instancí a adres. Slouží klientům, či směrovačům při objevování služeb (viz následující kapitola 2.7.3.3). Jednotlivé instance jsou registrovány při spuštění a odregistrovány při vypnutí. Směrovač (v případě *server-side*) nebo klient (v případě *client-side*) se registru služeb dotáže, kam má přesměrovávat požadavky, pokud chce komunikovat s danou mikroslužbou.

Instance se můžou registrovat buď samy nebo jsou registrovány registrátorem, který detekuje spuštění (a vypnutí) nových instancí. V případě prvním zesložiťujeme kód mikroslužeb, v případě druhém je kód jednodušší, ale musíme se starat o další komponentu. Jednotlivé instance mohou přestat fungovat i nedobrovolně. Aby nedocházelo k přesměrovávání požadavků na nefungující instance, může registrátor, případně registr služeb využít *health check* rozhraní (více v kapitole 2.7.6.1) pro zjišťování, zda je daná instance schopná zpracovávat požadavky.

2.7.3.3 Objevování služeb

Jak uvádí [24, 25] v běžných distribuovaných systémech služby běží na fixní známé adrese a portu. V takovém případě je lze snadno volat s pomocí protokolů HTTP/REST, případně jiných RPC mechanizmů. V případě mikroslužeb může být existence instancí krátkodobá a jejich adresa a port se mění (obvykle jsou virtualizované a kontejnerizované). Musíme tedy vytvořit mechanizmus, který umožní komunikaci s neustále se měnící množinou instancí mikroslužeb na různých síťových adresách. Potřebujeme tedy vyřešit následující problémy:

- Každá instance mikroslužby vystavuje rozhraní, například HTTP/REST nebo Thrift (jazyk pro definici rozhraní využívající binární komunikační protokol) a my je potřebujeme umět zavolat.
- Počet a adresy instancí se mění.
- Virtuálním strojům a kontejnerům, na kterých mikroslužby běží, jsou dynamicky přiřazovány IP adresy.
- Počet instancí jednotlivých mikroslužeb se mění v závislosti na situaci, například aktuální zátěži.

Tento problém lze řešit dvěma způsoby. Buď využijeme další speciální komponentu, nebo budeme její logiku implementovat do klientů mikroslužeb.

objevování služeb na straně klienta Klient mikroslužby (klientská aplikace / API Gateway / jiná mikroslužba) se před dotazem zeptá registru služeb (viz předchozí kapitola) na adresu a port pokud chce komunikovat s vybranou mikroslužbou. Na základě získaných informací provede dotaz na běžící instanci mikroslužby. [24]

objevování služeb na straně serveru Požadavek klienta prochází směrovačem, který může současně zastávat funkce vyvažovače zátěže. Směrovač se při dotazu zeptá registru služeb, kam má dotaz přesměrovat a dotaz na základě získané informace přepošle na běžící instanci mikroslužby. Registr služeb může být součástí tohoto směrovače. [25]

Pokud neimplementujeme směrovač, vyhneme se dalšímu zachycování požadavku ve směrovači a tedy vyšší latenci, ale musíme do každého klienta implementovat logiku pro objevování služeb. Implementace směrovače vyžaduje vývoj a údržbu další komponenty systému a směrovač musí umět využívat všechny používané komunikační protokoly. [24, 25]

2.7.4 Komunikace mezi službami

V rámci monolitické aplikace jednotlivé komponenty volají ostatní komponenty za pomoci jednoduchých funkčních volání či posílání zpráv (v kontextu objektového návrhu). Naproti tomu v aplikaci s architekturou mikroslužeb jsou jednotlivé mikroslužby samostatnými procesy a tedy volání jiné mikroslužby je komunikace mezi procesy. [26]

Mikroslužby mohou komunikovat různými způsoby, jejichž dělení můžeme provést ve dvou dimenzích. V jedné dimenzi lze komunikaci dělit na [26]:

jeden-na-jednoho Každý požadavek zpracovává právě jedna instance mikroslužby.

	jeden-na-jednoho	jeden-na-více
synchronní	request/response	-
asynchronní	notification request/async response	publish/subscribe publish/async responses

Tabulka 2.1: Formy komunikace v architektuře mikroslužeb [26]

jeden-na-více Každý požadavek může být zpracován více různými mikroslužbami.

Ve druhé dimenzi řešíme, zda volající čeká na odpověď nebo nikoliv.

synchronní Klient očekává odpověď a může se zablokovat dokud odpověď nedostane.

asynchronní Klient se při čekání na odpověď neblokuje, nebo odpověď vůbec neočekává. Odpověď nemusí přijít hned.

Tabulka 2.1 pojmenovává jednotlivé formy komunikace a jejich zařazení do dimenzí. Následuje popis jednotlivých forem komunikace pojmenovaných ve zmíněné tabulce. [26]

request/response Klient zašle požadavek a čeká na odpověď. Odpověď očekává brzo a v případě vícevláknové aplikace se může vlákno zablokovat, dokud nedostane odpověď.

notification Klient zašle požadavek, ale neočekává odpověď, ani žádná nebude vrácena.

request/async response Klient zašle požadavek a ví, že odpověď nemusí dostat hned. Odpověď je zaslána asynchronně. Vlákno se nezablokuje.

publish/subscribe Klient zveřejní zprávu, kterou nemusí zpracovat žádná služba nebo ji zpracuje i více služeb.

publish/async responses Klient zveřejní zprávu a poté čeká určitou dobu na asynchronní odpovědi.

Varianta *request/response* je princip, který využívá HTTP/REST protokol (nebo i jiné RPC mechanizmy). Zbylé způsoby komunikace s pomocí HTTP/REST nelze implementovat. Použití REST nebo RPC [27]:

- je jednoduché a obecně známé,
- snadno se implementuje varianta request/response
- a nevyžaduje přítomnost další komponenty – zprostředkovatele zpráv.

Naopak z jejich použití plyne několik nevýhod. Mezi ně patří [27]

- nemožnost implementovat ostatní formy komunikace,
- nižší dostupnost aplikace, neboť během komunikace musí být obě strany dostupné
- a odesílatelé požadavků musejí umět najít cílovou adresu (například s použitím objevování služeb na straně klienta nebo serveru popsaného v předchozí kapitole 2.7.3.3).

Všechny zmíněné formy komunikace lze implementovat s pomocí zasílání zpráv (*messaging*) [28] skrze zprostředkovatele zpráv (*message broker*). Implementace *request/response* je komplikovanější v porovnání s její implementací v HTTP/REST a vyžaduje vysoce dostupnou komponentu v podobě zprostředkovatele zpráv. Použitím vzoru komunikace pomocí zpráv se systém stává

- méně provázaný,
- více dostupný (konzument zprávy nemusí být v době odeslání zprávy dostupný – zprostředkovatel zprávy dočasně ukládá (*buffer*))
- a podporuje všechny zmíněné varianty komunikace.

2.7.5 Strategie vydávání

Každou mikroslužbu je nutné vydat do produkčního prostředí. Pokud se omezíme na aplikace běžící v JVM prostředí (což bude případ mikroslužeb vytvořených v rámci této práce), jednotlivé mikroslužby můžou běžet buď samostatně v rámci jedné JVM, nebo může běžet více instancí v rámci jedné JVM. V případě *jedna instance - jedna JVM* [29]

- dosáhneme izolace jednotlivých instancí,
- nebude docházet ke konfliktům při přístupu ke zdrojům,
- instance nebude moci využít více zdrojů než má dané JVM přiřazené,
- vydávání, monitorování a spravování každé mikroslužby bude relativně přímočaré,
- ale volba této možnosti může způsobit menší využití fyzických zdrojů.

V případě *více instancí - jedna JVM* [30] získáváme lepší utilizaci zdrojů, ale riskujeme, že nastanou problémy, které u předchozí varianty nastat nemohou, včetně složitější správy a monitorování.

2.7.6 Zotavování z chyb

Pro snažší zotavování z chyb nebo jejich předcházení je nutné počítat s možností nedostupnosti některých mikroslužeb. S tím může pomoci implementace rozhraní pro kontrolu zdraví mikroslužby (*health check API*). Problémy mohou nastat při synchronní komunikaci mikroslužeb, které mají potenciál celý systém zahltit. Proto může být vhodné implementovat návhrový vzor *circuit breaker* popsaný v podkapitole 2.7.6.2.

S předcházením a řešením chyb může pomoci i důsledné a užitečné vypisování (*loggging*) a možnost trasování chyb napříč mikroslužbami. Je vhodné výpisy ze všech mikroslužeb agregovat v rámci služby k tomu určené. Zároveň je vhodné každý výpis opatřit společným identifikátorem externího požadavku. V agregující mikroslužbě je poté vhodné nastavit upozornění, pokud se objeví určité zprávy ve výpisu, které znamenají problém. Může být vhodné zaznamenávat jednotlivé operace všech uživatelů. Nicméně veškeré logování může být poměrně nákladné a vyžadovat extra infrastrukturu. [31, 32, 33, 34, 35]

2.7.6.1 Health check API

Rozhraní, přes které se lze dotázat na stav mikroslužby. Může zohledňovat stav aplikace, stav systému, na kterém běží nebo stav připojení na využívané služby. Už jen to, že mikroslužba na náš dotaz neodpoví je ukazatelem problému. [31]

2.7.6.2 Circuit breaker

Pokud mikroslužba vyžaduje synchronní spolupráci jiné mikroslužby, obvykle využije komunikace ve formě *request/response*. Někdy může dojít k výpadku dotazované mikroslužby nebo k nárůstu zpoždění v komunikaci, proto může být vhodné implementovat návrhový vzor *circuit breaker*. Tento vzor může zabránit v propagaci problémů do dalších mikroslužeb.

Principem je, že mikroslužby využívají určitou proxy pro dotazování se jiných mikroslužeb. Tato proxy analyzuje úspěšnost jednotlivých požadavků. Pokud nastane několik selhání po sobě a jejich počet přesáhne zvolený práh, všechny následující požadavky automaticky selžou. Požadavky selhávají po dobu určeného časového limitu a poté je opět několik požadavků zasláno na cílovou mikroslužbu pro otestování, zda problém stále trvá. V případě úspěchu obnoví normální fungování, v opačném případě se opakuje automatické selhávání požadavků po zvolený časový limit. [35]

2.7.7 Společný základ mikroslužeb

Ve všech, nebo téměř všech mikroslužbách můžeme potřebovat řešit stejné problémy (*cross-cutting concerns*) [36]. K tomu je vhodné vytvořit nějaký spo-

lečný kód, ze kterého bude každá mikroslužba vycházet. Mezi takové problémy lze počítat [37]

- konfiguraci (např. způsob připojování k ostatním službám),
- API komunikaci s klienty,
- komunikaci se zprostředkovatelem zpráv/událostí,
- komunikaci s databází, respektive persistentním úložištěm,
- vypisování
- a případně nějaká další.

2.7.8 Klady a zápory

Z použití architektury mikroslužeb plyne několik pozitiv, některá byla popsána v předchozích podkapitolách této sekce. Následuje souhrn pozitiv architektury mikroslužeb. [11]

- Umožňuje kontinuální vývoj a vydávání rozsáhlých komplexních aplikací.
  - Usnadňuje údržbu – Každá mikroslužba je relativně malá a je tedy jednodušší pochopit její fungování, případně ho upravit.
  - Lepší testovatelnost – Mikroslužby jsou menší a jsou rychleji testovatelné.
  - Snazší vydávání – Mikroslužby je možné vydávat nezávisle na sobě.
  - Snazší škálování vývoje – Jeden tým může vyvíjet jednu nebo více mikroslužeb nezávisle na ostatních mikroslužbách/týmech.
- Každá mikroslužba je relativně malá.
  - Pro vývojáře je snazší pochopit její fungování.
  - Vývojová prostředí jsou rychlejší a vývojáři jsou tak produktivnější.
  - Spouštění aplikace je rychlejší, z čehož plyne větší produktivita vývojářů a rychlejší vydávání.
- Lépe izoluje chyby. Problém v jedné mikroslužbě může způsobit její pád. Tento pád ale nemusí a neměl by způsobit pád celého systému. Oproti tomu v monolitické architektuře může malá chyba způsobit pád celé aplikace.
- Nevynucuje dlouhodobý výběr množiny technologií pro vývoj mikroslužeb, neboť každá mikroslužba může používat jinou technologii a zároveň přepsání jedné mikroslužby není takový problém, jako přepis kódu celé monolitické aplikace.

Používání architektury mikroslužeb představuje i několik nových problémů a negativ. Ty jsou shrnuty v následujícím seznamu.

- Vývojáři se musí vypořádat s dodatečnou komplexitou plynoucí z vývoje distribuovaného systému.
  - Vývojáři musí implementovat mechanizmy pro komunikaci mezi službami a musí se vypořádávat s možností částečného selhání.
  - Vývoj funkcí pro vypořádávání se s požadavky, které zahrnují kooperaci několika mikroslužeb je složitější a vyžaduje precizní kooperaci mezi týmy.
  - Testování interakcí mezi službami je složitější.
  - Vývojové nástroje jsou zaměřené na vývoj monolitických aplikací a podpora distribuovaných aplikací je problematičtější.
- Vydávání je složitější, neboť musíme obstarávat vydávání několika mikroslužeb.
- Větší náročnost na zdroje. Počet instancí mikroslužeb typicky přesahuje množství instancí ekvivalentní aplikace v architektuře monolitické. Vyžaduje tedy více virtuálních strojů a běhových prostředí (*runtime*), například JVM.

2.8 Vhodné způsoby transformace

Před zahájením transformace je nutné zhodnotit, zda je vhodné systém dané transformaci podrobit. Pokud máme relativně malý systém nemusí plynout z architektury mikroslužeb takový užitek jako u komplexnějšího systému. Jak již bylo zmíněno, se zavedením systému mikroslužeb přichází dodatečná komplexita, například v podobě nutnosti vývoje a údržby několika dalších komponent (např. *API Gateway*, *Service registry*, apod.). U malých systémů tedy může být výhodnější zůstat u monolitické architektury. [38]

Podle [38] lze k transformaci přistupovat různými způsoby. V různých situacích může být vhodný odlišný způsob.

- Aplikaci můžeme celou přepsat do nové architektury a starou verzi kompletně nahradit.
- V případě, že ještě neexistuje monolitická aplikace, můžeme začít s implementací rovnou v architektuře mikroslužeb.
- Postupně zbavovat monolit vybraných funkcionalit a implementovat je do nových mikroslužeb.

2.8.1 Celkový přepis

Celkový přepis aplikace nemusí být vždy vhodná volba. V době přepisování je složité doručovat nové funkcionality, neboť jejich zavedení přináší nutnost změny kódu, který se brzo zahodí a zároveň se s tímto kódem musí počítat při přepisu a můžou nám tyto změny ovlivnit plány. Tento způsob může být vhodný u aplikací, které jsou relativně stabilní a u nichž budeme benefitovat například z možnosti snazšího škálování v podobě aplikace založené na architektuře mikroslužeb.

2.8.2 Začít rovnou s mikroslužbami

Nejedná se přímo o způsob transformace systému, ale pokud bychom se na začátku vývoje aplikace rozhodli rovnou implementovat architekturu mikroslužeb, nemuseli bychom transformaci zvažovat. S touto volbou nicméně souvisí několik nevýhod. Při vytváření systému nemusíme vědět, jak moc úspěšný–využívaný náš systém bude, takže přímá implementace systému s použitím architektury mikroslužeb může být přehnaná a její benefity nemusíme nikdy využít. Zároveň může být těžší správně identifikovat zmiňované *bounded context* (viz kapitola 2.7.1) a refaktorování napříč mikroslužbami je mnohem náročnější než refaktorování v rámci jedné monolitické aplikace.

Pokud bychom se i po přihlédnutí ke zmíněným nevýhodám rozhodli implementovat aplikaci rovnou v architektuře mikroslužeb, v první řadě nebudeme muset transformovat monolitickou aplikaci, ale budeme rovnou implementovat modulární mikroslužbovou aplikaci a zvykneme si na vývoj v prostředí mikroslužeb. Bude se pracovat v malých týmech a bude relativně snadné vývoj škálovat. Dalším ukazatelem podporující přímou implementaci aplikace v architektuře mikroslužeb může být nutnost vysoké disciplíny pro implementaci dostatečně modulární monolitické aplikace, kterou je v případě úspěchu služby plánováno přepsat do architektury mikroslužeb.

2.8.3 Postupné oddělování funkcionalit

Zdroj [39] popisuje, jak můžeme postupovat, pokud již máme fungující monolitický systém. Jako dobrý způsob transformace z monolitické architektury do architektury mikroslužeb se jeví postupné oddělování funkcí z monolitu. Jak již z pojmenování vyplývá tento proces je iterativní, a tedy v každém kroku musíme rozdhodovat, která část systému bude oddělena a jak je tato část oddělena (viz dekompozice v kapitole 2.7.1). Vhodné je začít s relativně jednoduchou a málo provázanou schopností systému. Tým se při tom naučí, jak bude fungovat vydávání nových mikroslužeb, zlepší se dovednosti jednotlivých členů týmu a zároveň se zřídí minimální nutná infrastruktura pro vývoj a vydávání nových mikroslužeb.

Při rozhodování, které další schopnosti systému oddělit je vhodné vybrat takové, které nebudou mít závislost na existující monolit. Pokud bude mik-

roslužba záviset na monolitu, bude také provázána s jeho vývojovým cyklem a vývoj mikroslužby tedy může být pomalejší a nebude nezávislý. V případě, že není možné se vyhnout závislosti zpět do monolitu, je vhodné v monolitické aplikace vytvořit nové rozhraní, ke kterému bude nová mikroslužba přistupovat skrze tzv. *anti-corruption* vrstvu, čímž můžeme zamezit propagaci možných nevhodných konceptů z monolitu do nových mikroslužeb.

Častým způsobem rozdělení systému je oddělení vrstev vypořádávajících se s uživatelským rozhraním od podnikové logiky a persistentního úložiště. V takovém případě ale všechen provoz stále prochází skrze jeden databázový server a škálování je tak omezené. Abychom dosáhli výhod architektury mikroslužeb je nutné databázi rozdělit. Abychom z výhod architektury mikroslužeb začali těžit co nejdříve, je vhodné po iniciálním oddělení málo provázaných schopností začít s oddělováním schopností, které se často mění, a s funkcemi, které jsou pro podnik důležité.

Také může být vhodné začít rozdělením systému do několika větších mikroslužeb. Jejich správa bude jednodušší, protože jich nebude tak velké množství. Můžeme poté postupně jednotlivé mikroslužby dělit na menší, které mají přesněji definované hranice. Identifikace těchto hranic je obtížná a při dalších iteracích může být jednoduší tuto správnou hranici najít, tj. oproti její identifikaci přímo v monolitické aplikaci.

2.8.3.1 Oddělování komplexní funkcionality

Podle [40] je při oddělování komplexní funkcionality závisející na velkém množství persistentních dat nutné postupovat obezřetně, po malých krocích. Pokud se něco pokazí, můžeme se snadno o krok vrátit a snáze identifikujeme problém v daném malém kroku. Architektura systému během těchto kroků prochází transformací a teprve až po dokončení všech kroků máme systém v lepším stavu. Zmiňované kroky jsou následující:

1. **Identifikace logiky k oddělení** – Analyzujeme funkcionalitu a data související s oddělovanou funkcionalitou a určíme přesné hranice, co s funkcionalitou souvisí a co už ne. Může se jednat i jen o část tabulky nebo jen několik metod v rámci jedné třídy.

2. **Logické oddělení v rámci monolitu** – Izolujeme identifikovanou funkcionalitu a stritkně ji oddělíme v rámci monolitické aplikace. Později se tato funkcionalita zkopíruje do nové mikroslužby, s čímž při oddělování v monolitu musíme počítat. Například tedy nesmíme přistupovat k datům, která nebyla identifikována jako součást funkcionality napřímo, ale skrze existující služby (třídy v monolitu poskytující tyto informace). Stejně tak ostatní součásti systému by již neměli přistupovat napřímo k datům, která souvisí s oddělovanou funkcionalitou.

3. **Vytvoření tabulek pro podporu oddělovaných funkcí** – Rozdělíme existující tabulky, za předpokladu, že oddělujeme funkcionalitu, která využívá pouze část existující tabulky. Je vhodné nové tabulky nijak neměnit, tzn. mapovat sloupečky ze starých tabulek jedna-k-jedné ke sloupečkům v nových tabulkách. Musíme data ze starých tabulek migrovat do nových a chceme mít snadnou možnost vrátit se o krok zpět, pokud se něco nepovede. Po této fázi by k datům oddělované funkcionality měla přistupovat napřímo pouze oddělená funkcionalita a zároveň by oddělovaná funkcionalita měla napřímo přistupovat pouze ke svým datům.

4. **Vytvoření nové mikroslužby** – Nyní zduplikujeme oddělenou funkcionalitu do nové mikroslužby, která bude ke svému fungování využívat existující tabulky v databázi monolitu. Změna oproti předchozímu kroku je také v tom, že volání ostatních funkcionalit jíž není přímé zavolání metody ale požadavek posílaný skrze síť.

5. **Přesměrování klientů na rozhraní nové služby** – V tomto kroku přesměrujeme všechny klienty na nové rozhraní nové služby. Pokud používáme API Gateway přesměrování je relativně snadné a změna by klienty neměla vůbec zasáhnout. Nemusíme nutně přesměrovat všechny klienty najednou, ale zároveň není vhodné, aby tato fáze trvala příliš dlouho, protože bychom se museli starat o jednu identickou funkcionalitu implementovanou ve dvou místech systému.

6. **Vytvoření databáze nové mikroslužby** – Vytvoříme novou databázi a vytvoříme související tabulky pro novou mikroslužbu. V této fázi může být lákavé upravit schéma tak, aby zahrnulovalo budoucí změny, ale pokud schéma ponecháme stejné, budou další fáze, tj. migrace dat, snazší.

7. **Synchronizace dat z monolitu** – Nově vytvořenou databázi budeme plnit daty z existující monolitické databáze. Nová databáze se v podstatě stane replikou dat (pouze ke čtení) souvisejích s oddělovanou funkcionalitou.

8. **Změna databáze mikroslužby** – Před provedením tohoto kroku je již nutné mít všechny klienty přesměrované na novou mikroslužbu, jinak bychom spravovali dvě databáze, do kterých by se paralelně zapisovalo. Podstatou tohoto kroku je změna používané databáze v nové mikroslužbě. Mikroslužba bude využívat svou vlastní databázi, do které bude mít přímý přístup právě a jen tato mikroslužba. Pokud nastanou nějaké problémy je jednoduché mikroslužbu přesměrovat zpět na původní databázi.

9. **Odstranění staré logiky a dat** – Jeden z velmi důležitých kroků, někdy se zapomene. Jedná se o odstatnění staré logiky a starých dat a tabulek z monolitické aplikace.

2.9 Stávající Uniqway serverová aplikace

V následujících podkapitolách bude celkově popsána aktuální serverová aplikace a bude podrobně analyzován aktuální stav funkcionalit, které jsou předmětem transformace této práce. Architekturu aktuální verze Uniqway serverové aplikace lze popsat jako monolitickou, neboť

- veškerý její kód je udržován v rámci jednoho Git repozitáře,
- vydává se jako celek,
- vystavuje rozhraní pro všechny klientské aplikace a jejich implementace není striktně oddělena, včetně integrací s třetími stranami,
- umožňuje pouze jednodimenzionální škálování (škálování podle osy X – přidávání instancí se stejným aplikačním kódem)
- a projevuje známky degradující modularity. Například testování bývá náročné z důvodu potřeby využití zástupného objektu (*mock*) pro pouze část funkcí, což plyne z existence modulů/služeb, které mají příliš mnoho nesouvisejících zodpovědností.

K vývoji byl použit Play framework v Java verzi. Volba technologie Java usnadňuje nábor nových členů týmu, neboť Java je běžně vyučována na na software zaměřených oborech v rámci ČVUT.

2.9.1 Balíčkování

Aplikace je členěna do balíčků. Dodržuje předepsanou formu balíčkování pro Play frameworku a dále ji rozšiřuje.

actions Play framework poskytuje možnost před zpracováním vybraných požadavků nad nimi provádět určité akce definicí příslušné třídy a anotací metody v controlleru. Tato funkcionalita je využita například k parsování *user-agent* hlavičky, autentizaci uživatele a dalším aplikačně specifickým akcím.

controllers Obsahuje třídy starající se o zpracování HTTP požadavků z vystaveného rozhraní, parsování a validaci přicházejících dat a audit vypisování.

dao Obsahuje třídy definované dle návrhového vzoru DAO pro přístup k databázovým entitám.

mails Obsahuje třídy definující data a chování jednotlivých emailů posílaných s pomocí mailového klienta poskytovaného frameworkem Play.

models Obsahuje modelové třídy (převážně bez podnikové logiky) reprezentující databázové entity, dto přenášené od klientů / ke klientům, třídy obalující informace sloužící k logování a další pomocné třídy. Jednotlivé typy modelových tříd jsou rozděleny do dílčích balíčků.

modules Obsahuje moduly, které usnadňují konfiguraci pro vkládání závislostí s pomocí knihovny `com.google.inject` respektive `javax.inject`.

services Obsahuje třídy zprostředkovávající podnikovou logiku, logiku pro vytváření instancí DTO vracených v odpovědích apod.

utils Spousta dalších pomocných tříd a rozhraní.

views Soubory definující HTML dokumenty využívající knihovnu Twirl (používáno pro emaily, více později).

Balíčky *controllers* a *services* jsou dále děleny do balíčků podle toho, o které části systému / o které funkce se starají (např. *client* / *admin* nebo *car* / *user* / *shop*). Toto rozdělení bohužel není striktně dodržováno.

2.9.2 Ostatní funkcionality aplikace

U některých zdrojů je možné omezit výsledky požadavku s pomocí *query string*. Je možné určit počet objektů, které chceme voláním získat. Od toho se pak také odvíjí počet stránek, na které bude výsledek rozdělen. S tím souvísí, že můžeme volit, kolikátou stránku nám má aplikace vrátit. Další možností v *query string* je možnost definovat parametr a jeho hodnotu. Aplikace pak vrátí objekty, jejichž parametr má zadanou hodnotu. Poslední možností je určit řazení výsledků podle zvoleného parametru vraceného objektu.

Online verifikování uživatelů vyžaduje dočasné ukládání uživatelských dokladů (jejich fotografií). K tomu je využita služba AWS S3 (*Amazon Web Services Simple Storage Service*) – jednoduchá *key-value* databáze.

Vývoj klientských aplikací a integrace s poskytováným rozhraním je podporována ručně psanou API dokumentací v portálu Swagger. Tato dokumentace nebyla psána od počátku projektu, a tedy některé vystavené zdroje ani nejsou zdokumentovány a některé změny nejsou zanášeny, tím pádem dokumentace zastarává.

Aplikace využívá pro komunikaci s databází ORM technologii *Ebean*. DDL, respketive DML je psáno v rámci migrací na nové verze aplikace, v kontextu frameworku Play nazývané *evolutions*. Spolu s DDL je nutné ručně upravit i modelové třídy reprezentující jednotlivé entity. Existuje určitá možnost generování v rámci *Ebean*, ale nikdy nebyla využita.

Spolu s aplikací jsou rovněž psány automatické testy. Bohužel pokrývají malý zlomek funkcionality. Testy jsou psány na úrovní jednotkových testů. Tyto testy jsou psány v jazyku Java a jejich využití je podporováno samotným Play frameworkem. Spouští se snadno s pomocí příkazu `sbt test` v kořenovém adresáři projektu. Tyto testy nemají přístup do databáze ani nevyužívají vkládání závislostí (*dependency injection*). S určitými úpravami by bylo možné psát i testy s testovací databází a s vkládáním závislostí.

Dalším typem testů jsou testy vystaveného REST rozhraní. Tyto testy jsou psány v jazyce Python a s aplikací mohou interagovat jen skrze vystavené rozhraní. Z toho důvodu je omezená možnost kontroly změn, které na základě testovaných požadavků v aplikaci/databázi nastaly.

2.9.3 Vydávání aplikace

Aplikace běží v *docker* kontejneru ve službě Amazon ECS (*Elastic Container Service*). Nasazování nových verzí probíhá skrze nástroj TeamCity / AWS CodeDeploy. Pro vyvažování zátěže se využívá *load balancer* instance běžící v AWS EC2 (*Elastic Computing*). Pro ukládání výpisů a monitoring se využívá služba Amazon CloudWatch. Relační databáze, PostgreSQL, běží ve službě Amazon RDS (*Relational Database Service*). Pro automatizované spouštění funkcí s pomocí utility *cron* se využívá služba AWS Lambda.

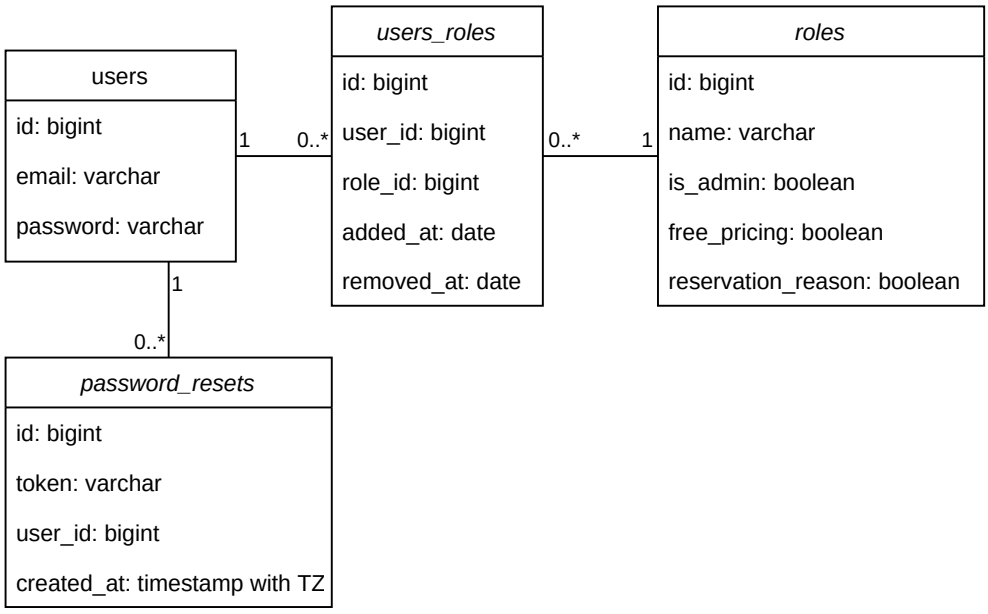
2.9.4 Autentizace a autorizace

Aplikace využívá `deadbolt` knihovnu s využitím JWT (*JSON Web Token*) pro autentizaci a uživatelské role pro autorizaci požadavků. Pro tvorbu a verifikaci JWT využívá knihovnu `com.auth0/java-jwt`.

Aplikace ukládá osolený hash uživatelského hesla. Při přihlášení je uživatelské heslo porovnáno s uloženým hash. Pro tuto funkcionalitu je využita knihovna `org.mindrot/jbcrypt`.

Pro vykonávání autentizace a autorizace uživatelů je implementováno několik tříd. Využívají nativní Play funkcionalitu tzv. *actions*, které umožňují provést určitý kód před zavoláním, respektive po vykonání, metody *controlleru*, která se má postarat o vyřešení požadavku. Konkrétní část kódu, který se má vykonat, se určí anotací nad metodou *controlleru*, případně nad *controller* třídou (poté se *action* zavolá před každou metodou *controlleru*). U veřejných zdrojů se anotace pro autentizaci ani autorizaci nepoužívají.

Kód vyvolaný anotací pro autentizaci ověří přítomnost a validitu JWT a určí, který uživatel daný požadavek vytvořil. Následuje autorizace uživatele podle přiřazených rolí (např. běžní uživatelé mají přístup pouze ke zdrojům určených pro klientské aplikace apod.). Pokud požadavek neobsahuje JWT nebo je token nevalidní, je vrácena odpověď s HTTP status kódem *401 Unauthorized* dle konvencí. Odpověď 401 aplikace vrací i v případě neúspěšné autorizace.



Obrázek 2.2: Databázové schéma související s autentizací a autorizací uživatelů

V obrázku 2.2 lze vidět tabulky a jejich parametry související s autentizací a autorizací uživatelů. Mimo primární klíče lze vidět tabulku *users* s přihlašovacím jménem (email) a heslem, tj. již zmíněný osolený hash uživatelova hesla. Dále existuje tabulka *roles*, která má vztah přes vazební tabulku s uživateli. Role mj. určuje, zda se může uživatel přihlásit do administrátorské aplikace, na jaké ceníky může jezdit apod.

Pro autentizaci modulů z automobilů se využívá zastaralá, nepříliš bezpečná metoda. Modul má uložený hash, kterým podepisuje své požadavky a výsledný hash přibaluje do požadavků. Stejný hash je uložen i v databázi aplikace. Aplikaci při přijetí požadavku použije uložený hash k ověření podpisu. Je v plánu tuto funkcionalitu změnit, aby fungovala podobně jako funkcionalita pro autentizaci uživatelů.

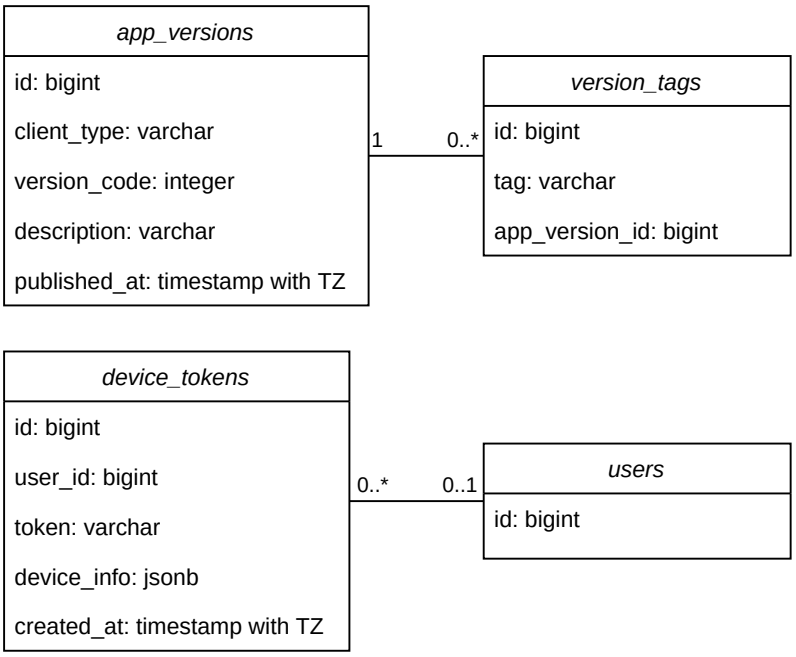
2.9.5 Notifikace uživatele

Aplikace se integruje se službou pro zasílání emailů, například emaily s fakturami za jízdy, nákupy v obchodě s odměnami apod. Využívá knihovnu `com.typesafe.play/play-mailer`, která je součástí Play frameworku.

Tato knihovna definuje rozhraní `MailerClient` s metodou `send`, která přijímá instanci třídy `Email`. Tato třída je rozšiřována – pro každý typ emailu existuje definice pro snažší instancování a zapouzdření logiky.

Pro vykreslování těla emailu je využívána knihovna `Twirl` pro správu parametrizovatelných šablon. Tato knihovna je podporována v rámci Play frameworku a umožňuje kombinovat Scala kód a samotné HTML.

Pro konfiguraci lze využít několik konfiguračních proměnných souvisejících s využitím protokolu SMTP. Lze využít konfiguraci *mock*, která způsobí vypisování obsahu emailu do konzole a fyzicky se žádné posílání emailů neprovede.



Obrázek 2.3: Databázové schéma související s push notifikacemi

Aplikace umí posílat push notifikace s pomocí FCM technologie (*Firebase Cloud Messaging*). Jak je vidět na obrázku 2.3 aplikace ukládá příslušné FCM tokeny ve své databázi s odkazem na uživatele. Spolu s FCM tokenem jsou také uloženy určité informace o použitém zařízení, jako je verze aplikace, operační systém a *bundle ID*, respektive *package name* v závislosti na platformě. Informace o zařízení slouží k rozhodnutí, jaké typy notifikace aplikace podporuje (v minulosti nastalo několik zpětně nekompatibilních změn) a zároveň slouží k zamezení odesílání notifikací z určitých prostředí (produkční vs. testovací) na zařízení, která jsou připojena do jiného prostředí. Současně je poznamenán čas uložení záznamu (parametr `created_at`).

Pro rozhodování, které FCM tokeny podporují jakou verzi notifikací, jsou vedeny tabulky `app_versions` a `version_tags`. První z nich popisuje určitou verzi aplikace pro vybranou platformu. Druhá přiřazuje k vybranému záznamu v `app_versions` textový tag, který identifikuje určitou funkcionalitu. Pro pochopení bude uveden příklad: Zařízení uživatele uložilo svůj FCM

token společně s informací, že verze používané aplikace je *10* na platformě *iOS*. V databázi je uložena `app_versions` s číslem verze *5*, která je odkazována cizím klíčem ze záznamu v tabulce `version_tags` s hodnotou tagu `push_notifications`. V moment, kdy posíláme push notifikaci je číslo verze používané aplikace (*10*) porovnáno vůči verzi aplikace, která implementovala push notifikace (*5*). Pokud je verze aplikace stejná nebo vyšší je push notifikace poslána.

Klientské aplikace své FCM tokeny posílají na server skrze vystavený zdroj při spuštění aplikace nebo při změně. Podle [41, 42] změna tokenu může nastat pokud uživatel aplikaci

- obnoví na novém zařízení,
- odinstaluje/přeinstaluje,
- nebo vymaže aplikační data.

Pro poslání notifikace uživateli stačí na serverem vystavený zdroj poslat obsah notifikace (nadpis a tělo), seznam uživatelů (jejich identifikátorů) a platnost notifikace. Systém notifikaci vytvoří a asynchronně ji pomocí FCM serverů doručí zvoleným uživatelům. Výsledek operace je zapsán do logu serveru.

Aplikace zároveň umožňuje s pomocí vystaveného zdroje vymazat neplatné tokeny. Tato funkcionalita se nepoužívá, neboť pročišťování se automaticky provádí při odeslání libovolné notifikace.

2.9.6 Komunikace s platební bránou

Aplikace se integruje s platební bránou GP webpay. Aplikace využívá specifikaci v jazyce WSDL poskytovanou platební bránou pro generování klientského kódu. Klientský kód je generován s pomocí utility *wsimport*. Každý požadavek poslaný na platební bránu musí být podepsaný. K podepisování se používá asymetrická šifra. Rovněž všechny odpovědi od platební brány jsou podepsány. Aplikace využívá funkce pro

- zpracování platby na základě poskytnuté služby,
- vygenerování odkazu na platbu
- a verifikaci platební karty.

Aplikace verifikuje platební kartu pokusem o vytvoření jednokorunové platby. Doporučený postup pro verifikaci platební karty popsaný v dokumentaci platební brány nebyl podporován mnoha bankami, proto byl zvolen postup s pokusem o platbu. Při registraci první platební karty nebo při změně se vygeneruje odkaz na jednokorunovou platbu, který se odešle uživateli emailem. Uživatel poté vyplní novou platební kartu na webu platební brány. Po

vyplnění je uživatel přesměrován na web `uniqway.cz` spolu s výsledkem. Odtud se uživatel dozví, zda vše proběhlo v pořádku a automaticky se výsledek pošle i na server. Po schválení bankou může uživatel využívat službu a platby se budou automaticky strhávat ze zvolené platební karty.

Platby se generují automaticky po ukončení rezervace (nebo po dokončení nákupu v obchodě s odměnami). Pokud platba z libovolného důvodu neprojde, vygeneruje se odkaz pro zaplacení, který se odešle na email uživatele.

Aplikace také podporuje možnost tvoření blokací. Tato funkcionalita se již dlouho nepoužívá a nebude se přenášet do mikroslužeb.

V obrázku 2.4 lze vidět tabulky související s komunikací s platební bránou. Tabulka `payment_actions` slouží k naplňování funkcionality blokací. Tabulka `payments` ukládá informace o platbách. Ke každé platbě existuje příslušná faktura, mimo neúspěšné platby za nákup v obchodě s odměnami. Aby mohla být platba vygenerována musí se identifikovat zákazník. Platební brána vyžaduje *celé jméno zákazníka*, *adresu bydliště včetně numerického kódu země* a *emailovou adresu*. Tabulka `payment_cards` ukládá vybrané informace o verifikované platební kartě. Tabulka `payment_card_tokens` ukládá *token*, pomocí kterého lze opakovat platby s danou platební kartou. Tabulka `card_verifications` ukládá informace o průběhu verifikace platební karty.

2.9.7 Cenotvorba

V obrázku 2.5 lze vidět databázové schéma popisující entity a jejich parametry podílející se na cenotvorbě rezervací. Automobily jsou na základě modelů rozděleny do skupin, kde každá taková skupina má přiřazený výchozí skupinový ceník (`car_model_pricelist`). Automobil je tedy přiřazen k právě jednomu aktivnímu skupinovému ceníku. Uživatelé jsou přiřazeni do uživatelských skupin. Toto přiřazení může být časově omezené, případně omezené počtem (tj. uživatel může využít danou uživatelskou skupinu jen na zadaný počet rezervací). Každá uživatelská skupina má odkaz na jízdní ceník (`ride_price_list`). Jízdní ceník je přiřazen ke skupinovému ceníku skrze model automobilu. Jízdní ceník může být přiřazen k několika různým skupinám automobilů. Jízdní ceník může být definován relativně vzhledem k výchozímu ceníku pro danou skupinu automobilů, nebo absolutně. Rezervace má odkaz na jízdu (ta existuje pouze pokud uživatel odemknul automobil), použitou uživatelskou skupinu, uživatele a po vypočtení ceny rezervace také na jízdní ceník.

Skupinový ceník definuje primární klíč, cizí klíče, parametry platnosti, lokalizované textové parametry popisující ceník a parametry definující samotný ceník. Primární klíč není popisován, podobně nejsou popisovaný cizí klíče, neboť odkazy byly popsány v předchozím odstavci. Následuje podrobnější popis jednotlivých parametrů skupinových ceníků.

platnost Parametry `added_at` a `removed_at` definují rozsah, během kterého

je možné daný ceník využít pro výpočet ceny rezervace. Rozhodné datum je začátek rezervace.

`price_text` Ať už česká nebo anglická varianta, tento text popisuje ceník a je zobrazován jakožto výchozí ceník v klientské aplikaci.

`pricing_parameters` Tento parametr je zde uveden pro zjednodušení schématu. Ve skutečnosti skupinový ceník definuje parametry pro hodinový ceník (cena za ujetý kilometr, cena za započatou hodinu rezervace a maximální počet placených hodin za 24 hodin rezervace) a pro minutový ceník (cena za minutu stání, cena za minutu jízdy – tj. odemčeného auta, počet kilometrů v ceně za den, cena za kilometr nad limit, maximální cena za minuty za 24 hodin rezervace a minimální cenu).

Podobné parametry definuje i jízdní ceník. O tom, zda je jízdní ceník relativní rozhoduje parametr `defined_in_percents`. Potom se v jednotlivých parametrech (jako je `price_per_km`) očekává procentuální hodnota. Nedefinuje parametry `price_text_cz` a `price_text_en`. Parametry platnosti definuje jako `add_at` a `remove_at`, jejich význam je stejný. Navíc definuje některé další parametry.

description Popisuje jízdní ceník, pro uživatele není nikde viditelné.

discount Definuje velikost slevy v korunách. V případě, že je sleva relativní (parametr `discount_in_percents` s hodnotou *true*) určuje procentuální hodnotu.

`free_pricing` Určuje, zda je ceník určen pro jízdy zdarma.

`use_hour_price_list` Určuje, zda je ceník hodinový, nebo minutový.

Pro přidávání nových nebo úpravu starých ceníků je vystaven zdroj. Změny ve starých cenících probíhají jejich duplikací, aby nebyla nechtěně změněna cena již probíhajících rezervací. Při vytváření nového jízdního ceníku je nutné specifikovat objekty definující typy položek na faktuře za jednotlivé složky ceny. Tyto položky jsou ukládány do tabulky `invoice_item_types`, která není v diagramu zanesena.

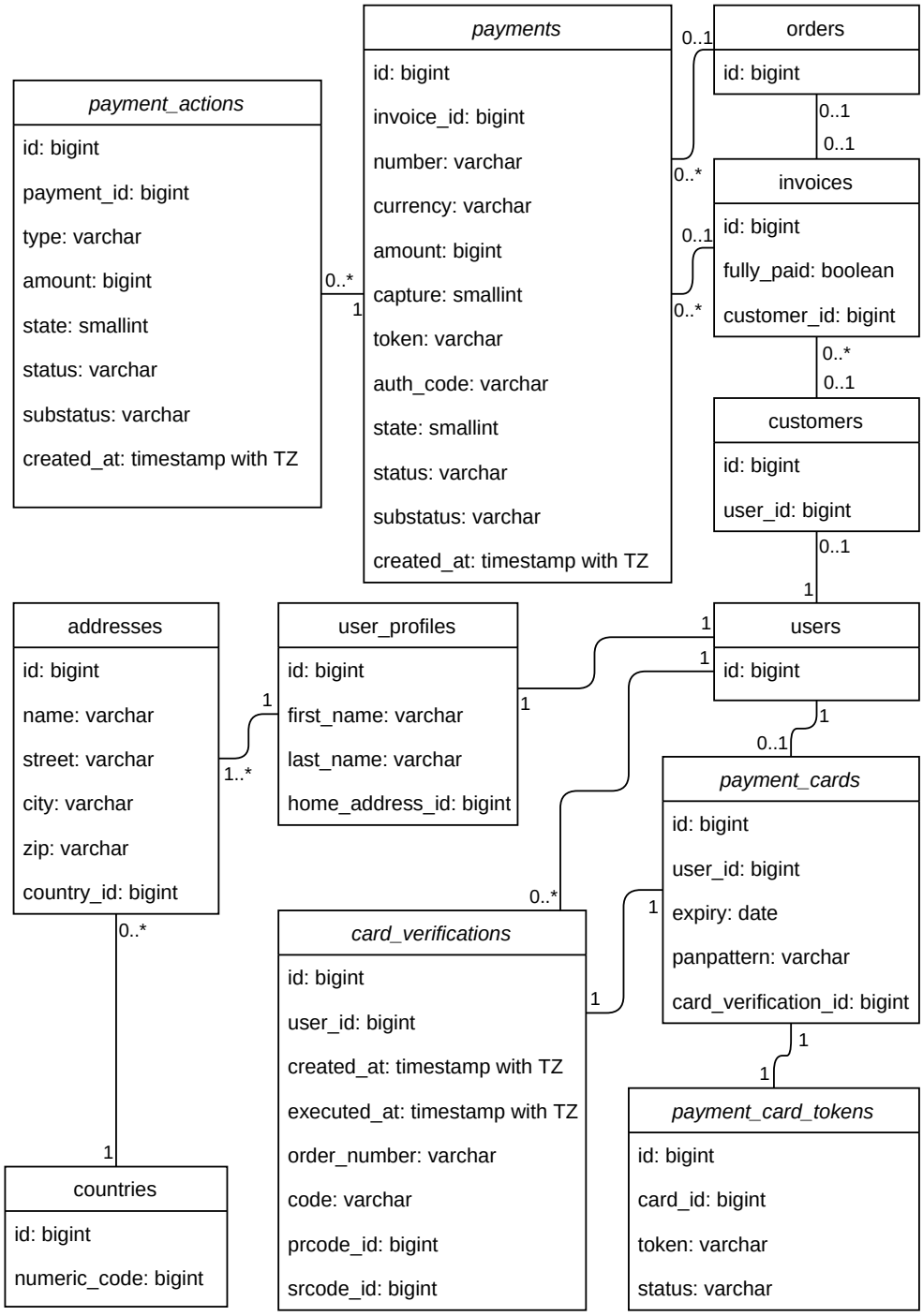
Jak bylo popsáno, jízdní ceníky mj. definují hodnotu sleva. Tento parametr je v plánu odstranit a vytvořit více robustní řešení slev. Aktuálně je pro vytvoření slevy nutné vždy vytvořit nový ceník. Spolu s touto změnou je v plánu také zavést možnost tzv. slevových voucherů, které umožní uživateli zakoupit kredit, který poté použije pro platby za jízdu a ve výsledku tak bude mít rezervace levnější.

2.9.7.1 Proces tvorby rezervace a výpočtu ceny

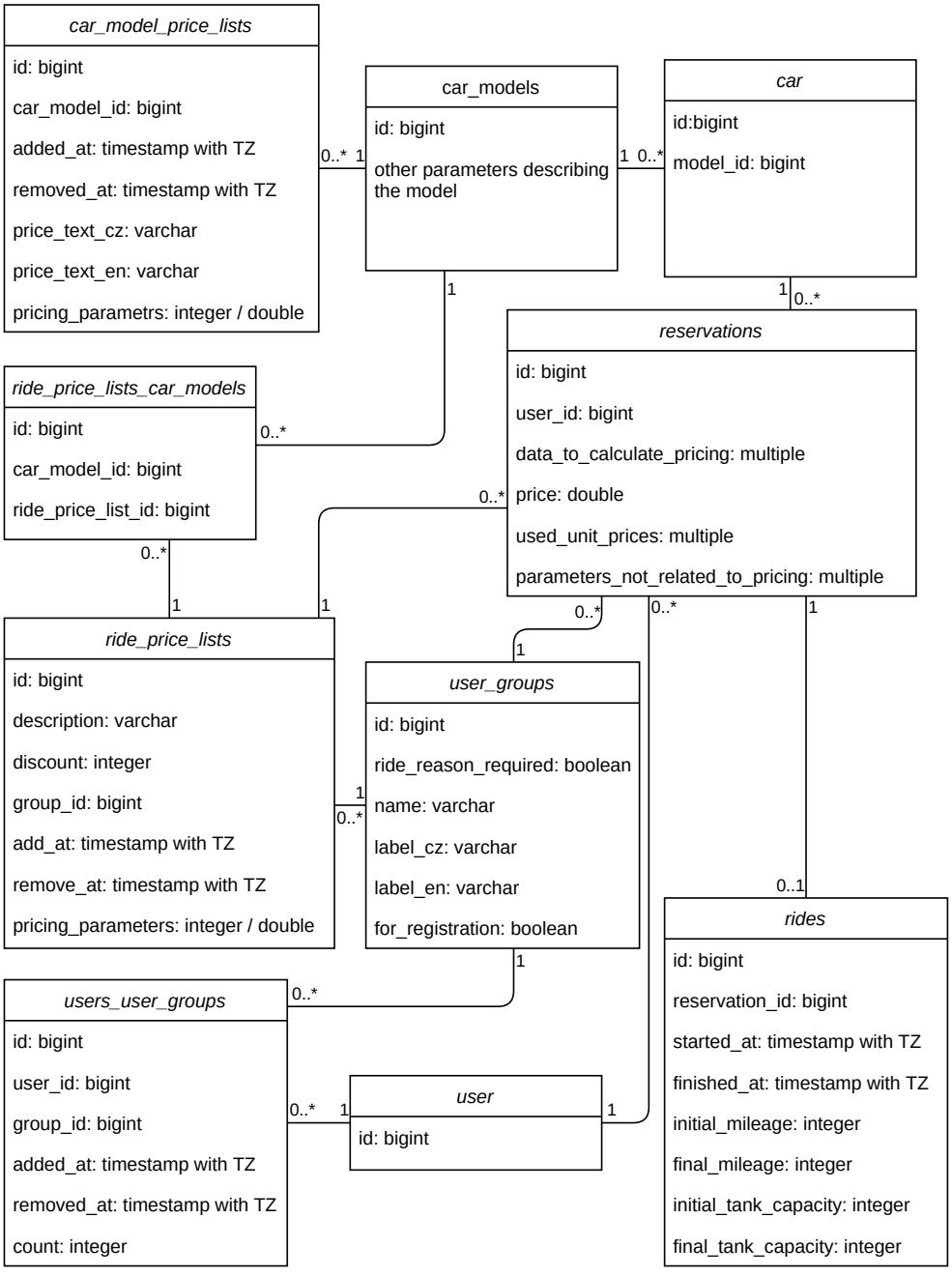
Uživatel vyhledá a vybere si auto, které chce rezervovat. Pokud má uživatel přiřazených více uživatelských skupin, vybere na začátku rezervace, kterou uživatelskou skupinu chce použít pro vznikající rezervaci. Při ukončování rezervace je vypočtena její cena pro každý přiřazený jízdní ceník pro daný automobil a uživatelskou skupinu. Pro fakturaci se vybere ten nejvýhodnější.

2.9.7.2 Kalkulačka na webu

Aplikace také umožňuje výpočet ceny potenciální jízdy na webové stránce. Pro tyto účely výpočtu je použit základní skupinový ceník. Vypočtená cena se vrací pro všechny podporované modely aut.



Obrázek 2.4: Databázové schéma související s komunikací s platební bránou



Obrázek 2.5: Databázové schéma související s cenotvorbou

Kapitola 3

Návrh

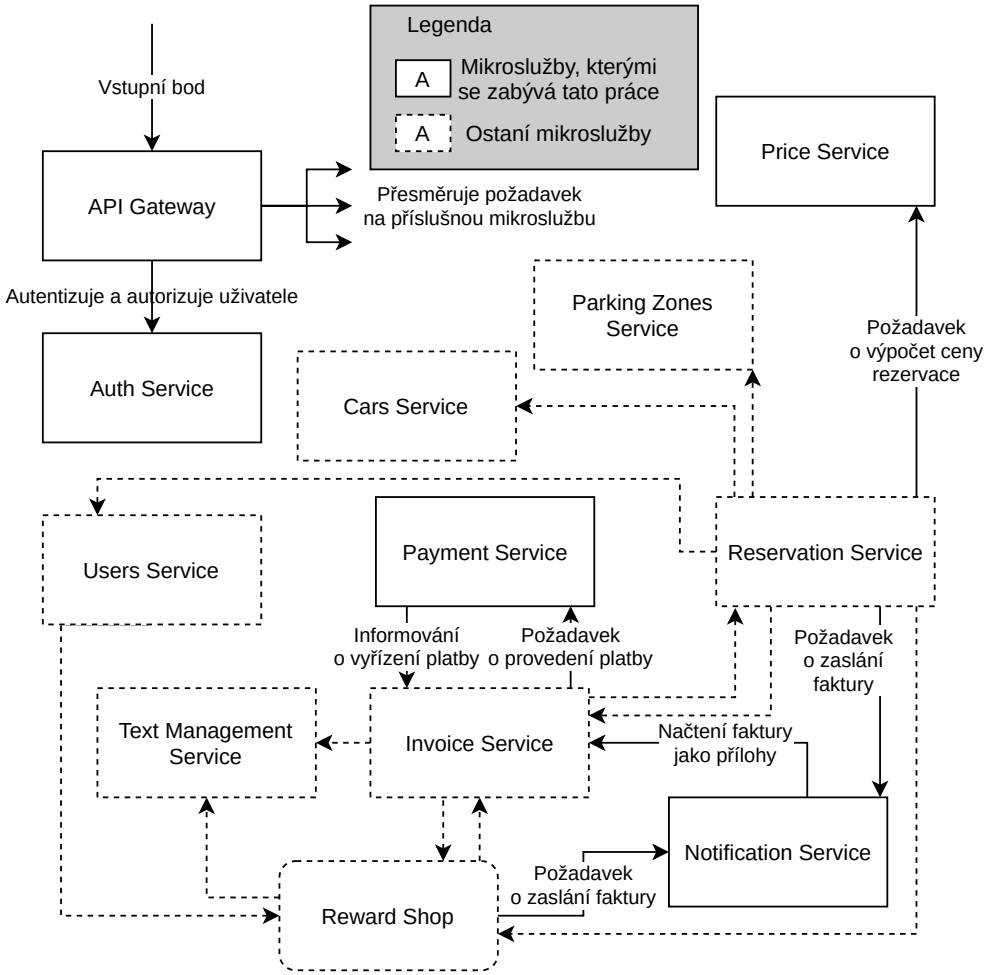
Spolu se zaváděním nové architektury bylo dohodnuto, že budeme chtít zůstat u Java Play Frameworku a zároveň využít generátorů kódu. Java Play Framework byl zvolen, neboť většina týmu je s ním obeznámena a tedy postupný přechod na mikroslužby nebude vyžadovat dovednosti s více různými frameworky.

Bylo rozhodnuto o použití generátoru z API dokumentace. Vyhneme se tak duplicitní práci a zbytečným chybám. API dokumentace se bude psát ve verzi OAS3. Existující dokumentace ve verzi Swagger 2.X bude aktualizována na novou verzi OAS3 a bude doplněna o nezdokumentované součásti. Pro generování kódu bude použit OpenAPI generátor. Generátor umožňuje generování kostry kódu pro Java Play Framework jak z hlediska server části, tak z hlediska klientské části [43]. Generátor mj. také generuje kód pro validaci vstupních (i výstupních) dat podle definice v API dokumentaci. Validace je možné vypnout s pomocí konfigurační proměnné.

Druhou možností, kdy je možné generovat kód je v případě databáze a entit reprezentující data. Spolu s rozhodnutím využívat technologii JOOQ se rozhodlo i o využití příslušného generátoru. JOOQ [44] umožňuje psaní SQL dotazů s pomocí typově bezpečného DSL. Navíc také umožňuje generování modelových tříd a tříd implementující DAO návrhový vzor. Touto technologií bude nahrazen ORM Ebean.

Pro lepší testovatelnost bude nutné zprovoznit testování s možností snadného ověření výsledků oproti testovací databázi. Při tomto bude nutné zajistit, aby se jednotlivé testy nemohly navzájem ovlivnit. Tyto testy se budou používat pouze pro komplexnější testování. Pro testování jednotlivých metod se bude nadále využívat jednotkové testování.

Na obrázku 3.1 lze vidět prvotní dekompozici systému do mikroslužeb. Součástí práce jsou mikroslužby, jejichž ohraničení je kreslené plnou čarou. Ostatní mikroslužby jsou kresleny čárkovanou čarou. V obrázku jsou rovněž zaneseny plánované interakce mezi mikroslužbami (více k tomuto tématu v pozdějších podkapitolách). Diagram není vyčerpávající a bude se s přibý-



Obrázek 3.1: Úvodní dekompozice systému do mikroslužeb

vajícími funkcemi měnit. Jednotlivé mikroslužby budou mít vlastní relační databáze (pokud ji budou potřebovat). Do databáze mikroslužby bude mít přístup pouze daná mikroslužba. Konzistence napříč databázemi se zpočátku bude pro jednoduchost řešit pomocí HTTP požadavků.

Pokud toto řešení nebude dostatečné, bude nutné implementovat jiné, spolehlivější způsoby pro zajištění konzistence dat napříč mikroslužbami analyzované v dřívějších kapitolách této práce (2.7.2). Navržené řešení sice vyžaduje vyšší dostupnost jednotlivých mikroslužeb (musejí být obě dostupné v době synchronizace), ale i v případě systému založeného na zasílání zpráv (nebo událostí) musíme mít minimálně zprostředkovatele zpráv vysoce dostupného. Problémy s dostupností mohou nastat převážně při velkém nárůstu požadavků,

což souvisí i s nárůstem uživatelů. Velký nárůst uživatelů se ovšem neočekává, neboť, jak již bylo zmíněno v 2.1, služba je určena pouze omezené skupině lidí. Využití jednoduchého způsobu s pomocí HTTP požadavků bude přívětivější pro nové členy pravdidelně se obměnujícího týmu vývojářů z řad studentů.

Jednotlivé služby budou nasazené v AWS ECS v privátní virtuální síti (Amazon VPC) podobně jako byla dosud nasazována monolitická aplikace. Jakožto jediný vstupní bod se zpočátku použije stávající monolitická aplikace. Později se přejde na službu Amazon API Gateway. Jednotlivé mikroslužby mezi sebou budou muset umět komunikovat, to usnadní služba pro objevování služeb, která je součástí Amazon ECS a funguje na principu DNS.

Mikroslužby mezi sebou budou komunikovat pomocí protokolu HTTP a to pouze metodou request/response. Komunikaci usnadní možnost generování klientského kódu z API dokumentace.

Každá mikroslužba bude poskytovat rozhraní pro otestování stavu instance (*health check API*, viz kapitola 2.7.6.1). Služby nebudou využívat vzor *circuit breaker*.

Pro vývoj mikroslužeb bude vytvořen společný základ, který bude poskytovat

- možnost snadné definice připojení do databáze,
- možnost snadno generovat modelové třídy, DAO třídy a ostatní potřebné třídy na základě DDL provedených v databázi s pomocí JOOQ generátoru,
- možnost snadno generovat kostru kódu pro serverovou část aplikace z API dokumentace,
- možnost snadno generovat klientský kód pro volání ostatních mikroslužeb,
- společný kód pro řešení výjimek, které nastanou při běhu programu a jejich logování,
- společný definiční soubor (`build.sbt`) pro sestavování aplikace (který bude možné rozšířit),
- a ostatní společný kód, který bude využíván ve většině mikroslužeb.

3.1 Návrh aplikačního rozhraní

Jak již bylo zmíněno, rozhraní aplikace se bude definovat a dokumentovat s pomocí specifikace OAS3. Bude se jednat o přístup *API-first*. Tento přístup umožní dřívější paralelizaci vývoje, například serverové aplikace a mobilních klientských aplikací. Jelikož se bude vše nejprve směrovat skrze existující monolitickou aplikaci, rozhraní se nebude měnit. Až bude nutné API měnit,

bude se implementovat API Gateway s pomocí služby Amazon API Gateway. Tato služba umožňuje mj. tvorbu poskytovaného rozhraní s pomocí definičního souboru v OAS3. Navíc definuje rozšíření této specifikace pro definice informací specifických pro tuto službu (např. autentizace zavoláním AWS Lambda funkce před propuštěním zvolených požadavků k řešení vybranou mikroslužbou). I když definiční soubor poskytovaného rozhraní zatím neexistuje, jeho vytvoření je i v případě, pokud by se Amazon API Gateway nevyužívala, plánováno. Jedná se o specifikaci „veřejného“ rozhraní pro snadnou integraci klientských aplikací. Implementace Amazon API Gateway tedy bude relativně snadnou záležitostí, tj. přidání několika málo specifických parametrů a nahrání souboru do AWS.

3.2 Návrhový model tříd

Z vygenerované serverové kostry získáme *controllery* a jejich implementace. Odtud budeme delegovat vyřešení požadavku na příslušné třídy starající se o podnikovou logiku. *Controllery* se postarají o serializaci a validaci vstupů a případné logování požadavků.

Pro přístup do relační databáze se použijí vygenerované DAO objekty, které budou případně rozšířeny o vhodnou funkcionalitu s pomocí podtřídy, abychom mohli relativně snadno přegenerovat třídy z databáze a zároveň nepřišli o existující vlastní implementaci.

3.3 Databázový model

Databázový model bude zpočátku v příslušných mikroslužbách stejný jako v monolitické aplikaci, přičemž bude omezený na relevantní tabulky. V jednotlivých kapitolách návrhu změn pro oddělované funkcionality bude uveden finální databázový model, ke kterému se budeme snažit přiblížit po dokončení oddělení.

3.4 Proces transformace

Transformace složitých funkcionalit bude probíhat s přihlédnutím k doporučenému postupu rozebíraného v kapitole 2.8.3.1. Nejprve se tedy jednoznačně oddělí funkcionalita v monolitické aplikace. Oddělovaná funkcionalita nebude napřímo přistupovat k datům, která bude v mikroslužbě přijímat z ostatních částí systému, stejně tak přístup do oddělovaných tabulek bude pouze skrze třídy, které se starájí o oddělovanou funkcionalitu. Potom bude následovat vytvoření mikroslužby pracující se starou formou dat s původní databází monolitické aplikace a poté se data přesunou do nové databáze, do privátní databáze mikroslužby. Nakonec se funkcionalita odstraní z původní monolitické aplikace. Následně se upraví funkcionalita a data v nové mikroslužbě, pokud

nějaká změna byla navržena. U některých funkcionalit budou některé kroky upraveny.

3.5 Autentizace a autorizace

Funkcionalita starající se o autentizaci uživatelů a autorizaci přístupu ke zdrojům bude zpočátku ponechána v monolitické aplikaci. Později bude oddělena do mikroslužby. Její finální podoba bude popsána v této kapitole. Mikroslužba bude muset umožňovat

- přihlašování do aplikací, tj. ověřování přihlašovacích údajů a tvorbu JWT,
- validaci příchozích JWT,
- resetování hesla
- a ověřování možnosti přístupu k vybranému zdroji podle uživatelských rolí uživatele identifikovaného přijatým JWT.

Pro ostatní mikroslužby bude nutné implementovat funkci pro získání uživatelských rolí. Například služba pro správu rezervací bude muset rozhodovat, zda může uživatel vytvořit rezervaci pro dané vozidlo (např. uživatelé s rolí *admin* mohou rezervovat vozidla i v jiných stavech než *Dostupné*) nebo zda může uživatel ukončit rezervaci a zároveň přepnout vozidlo do zvoleného stavu (volbu cílového stavu může provést pouze uživatel s rolí *admin* nebo *carAdmin*).

Centralizováním autorizace do této mikroslužby budou ostatní mikroslužby zjednodušeny. Nedostatkem je, že centralizovaná autorizace bude umět autorizovat pouze podle granularity cest ke zdroji. Pro mitigaci tohoto nedostatku bude vystaven již diskutovaný zdroj, který vráti role příslušného uživatele. Toto řešení je přijatelné, neboť takto podrobná granularita autorizace je řešena pouze výjimečně.

Mikroslužba bude také muset umožňovat autentizaci modulů z automobilů. Autentizace bude nejprve probíhat pomocí stávajícího řešení a později bude přepsána, aby využívala bezpečnější metodu s pomocí JWT.

Autentizace a autorizace přístupu k vybranému zdroji se bude provádět z Amazon API Gateway skrze službu AWS Lambda, která zavolá tuto mikroslužbu. Mikroslužba bude vracet informaci, zda je uživatel úspěšně autentizován a autorizován a AWS Lambda výsledek zpracuje a přeformátuje pro použití v Amazon API Gateway.

3.5.1 Návrh aplikačního rozhraní

Mikroslužba bude poskytovat následující rozhraní. Kompletní definici s využitím OAS3 lze nalézt v příloze. Rozhraní pro přidávání nebo úpravu rolí nebude implementováno. Historicky se role do systému v podstatě nepřidávali (systém obsahuje 5 rolí). V případě potřeby bude rozhraní doimplementováno.

URL: **`/api/client/login`**
HTTP metoda: POST
V těle požadavku očekává přihlašovací údaje uživatele. Pokud údaje odpovídají, vrátí uživatelský identifikátor a JWT. Abychom nemodifikovali rozhraní bude k tomuto zdroji přistupováno z mikroslužby spravující uživatele. Uživatelská mikroslužba poté odpověď doplní o uživatelská data, která jsou vracena i ze současné aplikace.

URL: **`/api/admin/login`**
HTTP metoda: POST
Funguje stejně jako přihlášení do klientské aplikace. Pouze omezuje skupinu autorizovaných uživatelů na uživatele s rolemi *admin*, *carAdmin*, *userAdmin* a *shopAdmin*.

URL: **`/api/modules/login`**
HTTP metoda: POST
Provede pokus o přihlášení modulu. Funguje podobně jako přihlašování uživatelů.

URL: **`/api/users/check`**
HTTP metoda: GET
Ověří, zda má uživatel přístup ke zvolenému zdroji.

URL: **`/api/client/check`**
HTTP metoda: GET
Ověří platnost použitého JWT.

URL: **`/api/modules/check`**
HTTP metoda: GET
Ověří, zda má modul přístup ke zvolenému zdroji.

URL: **`/api/(client|admin|modules)/logout`**
HTTP metoda: POST
Odhlásí uživatele a zneplatní používaný JWT.

URL: **`/api/client/password/reset/email`**
HTTP metoda: POST

Zašle uživateli email s odkazem, kde může nastavit nové heslo.

URL: **`/api/client/password/reset/:token`**
HTTP metoda: GET
Ověří, že uživatel v posledních 24 hodinách požádal o resetování hesla a ještě ho neprovedl. Proměnná *token* je unikátní identifikátor entity reprezentující požadavek o změnu hesla.

URL: **`/api/client/password/reset`**
HTTP metoda: POST
Nastaví nové heslo uživateli.

URL: **`/api/admin/resources`**
HTTP metoda: POST
Vytvoří nový zdroj a zvoleným uživatelským rolím nastaví přístupnost k přidanému zdroji.

URL: **`/api/admin/resources`**
HTTP metoda: GET
Získá detaily o všech zdrojích včetně přiřazených uživatelských rolí.

URL: **`/api/admin/resources/:id`**
HTTP metoda: GET
Získá detail vybraného zdroje včetně přiřazených uživatelských rolí.

URL: **`/api/admin/resources/:id`**
HTTP metoda: PUT
Upraví ukládané hodnoty u zvoleného zdroje, včetně možnosti upravit přiřazení rolí.

URL: **`/api/admin/resources/:id`**
HTTP metoda: DELETE
Odstraní vybraný zdroj.

URL: **`/api/users/:id/roles`**
HTTP metoda: PUT
Přiřadí vybranému uživateli seznam uživatelských rolí přiložených v těle požadavku.

URL: **`/api/users/:id`**
HTTP metoda: PUT
Upraví ukládané hodnoty u zvoleného uživatele, včetně možnosti upravit přiřazení rolí.

URL: `/api/users/:id`
HTTP metoda: DELETE
Odstraní vybraného uživatele.

URL: `/api/modules/:id`
HTTP metoda: PUT
Upraví hodnoty u zvoleného modulu.

URL: `/api/modules/:id`
HTTP metoda: DELETE
Odstraní vybraný modul.

Zdroje `/api/users/:id` `/api/modules/:id` jsou určené pro aktualizaci duplikovaných dat. Ostatní zdroje jsou určené pro běžné užití buď pro API Gateway nebo ostatní mikroslužby.

3.5.2 Databázový model

Jak je vidět v návrhu v obrázku 3.2, databáze mikroslužby bude ukládat informace pro podporu funkcí, které mikroslužba bude vykonávat. Autentizace a resetování hesla bude probíhat stejně jako v původní monolitické aplikace. Autorizace bude vyhodnocována na základě přiřazení jednotlivých zdrojů k uživatelským rolím. Pro starou autentizaci modulů bude součástí také tabulka *modules* se sloupečky *id* a *hash*. Pro novou autentizaci modulů se využijí parametry *code* a *password* podobně jako v případě autentizace uživatelů.

`resources` Každý záznam definuje cestu ke zdroji, regulární výraz definující cestu a HTTP metodu. Regulární výraz bude použit pro porovnávání, ke kterému zdroji se snaží uživatel přistoupit.

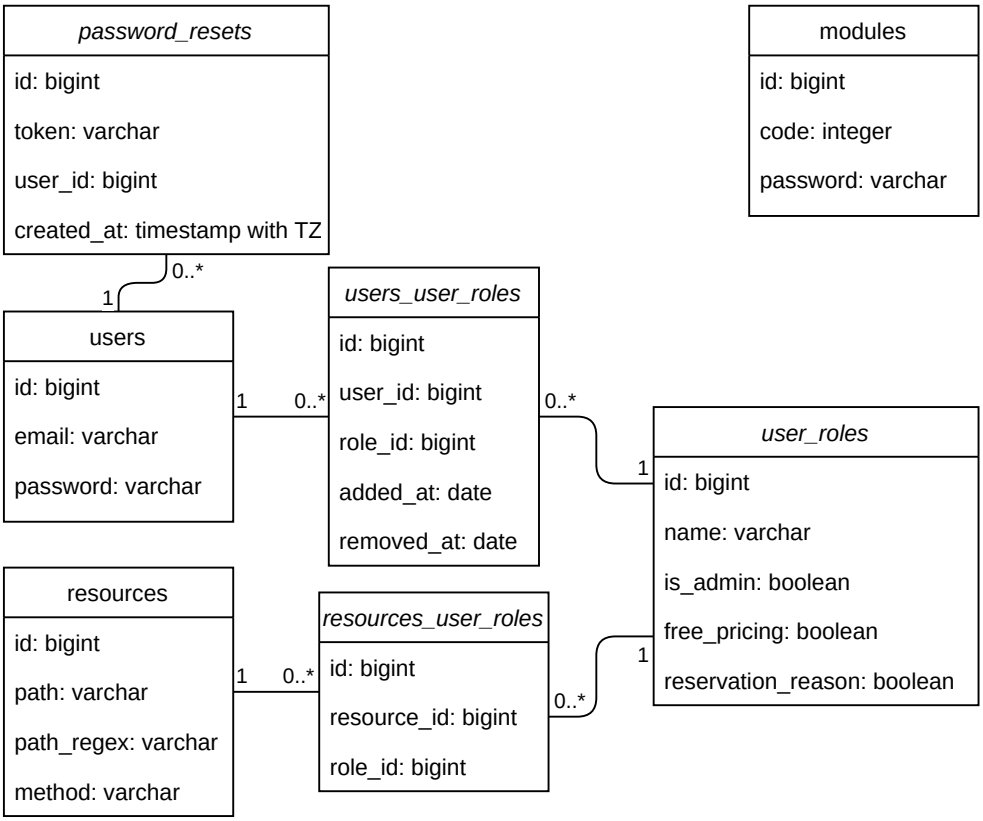
`resources_user_roles` Vazební tabulka určující, které zdroje jsou přístupné, kterými rolemi. Veřejné zdroje (např. ty, které nevyžadují přihlášeného uživatele) nebudou v databázi vedeny a autorizace přístupu k nim nebude prováděna.

`users` Jedná se o tabulku pouze ke čtení. Její úpravy jsou možné pouze skrze rozhraní `/api/users/:id`. Její hodnoty jsou kopiemi z mikroslužby spravující uživatele.

`modules` Jedná se o tabulku pouze ke čtení. Její úpravy jsou možné pouze skrze rozhraní `/api/modules/:id`. Její hodnoty jsou kopiemi z mikroslužby spravující moduly.

3.5.3 Proces oddělení

Proces oddělení bude víceméně dodržovat proces popsaný v 3.4. Transformace se nejprve provede pro funkce zajišťující autentizaci a autorizaci uživatelů. Po



Obrázek 3.2: Navržené databázové schéma související s autentizací a autorizací uživatelů

dokončení se bude stejným způsobem transformovat funkcionalita pro autentizaci modulů.

1. **Vytvoření repliky tabulky `users`** – V databázi monolitické aplikace vytvoříme novou tabulku pro udržování informací o uživateli pro potřeby nové mikroslužby (tj. bude udržovat pouze parametry `id`, `email` a `password`) Při libovolné změně některého z vybraných parametrů bude změna zanesena i do této tabulky. Rozhraní pro aktualizaci změn v této tabulce bude pečlivě odděleno, aby bylo později snadné tuto aktualizaci provádět s pomocí HTTP požadavku na novou mikroslužbu. Podobným způsobem budou replikována data původních tabulek `password_resets`, `users_roles` a `roles`.

2. **Vytvoření nové mikroslužby** – Následně vytvoříme novou mikroslužbu, která bude pracovat s vlastní databází a zatím může poskytovat

jen synchronizační rozhraní pro replikované tabulky.

3. **Přesměrování replikací** – Funkcionalitu pro replikaci tabulek přesměrujeme na rozhraní nové mikroslužby.

4. **Vytvoření tabulek `resources` a `resources_user_roles`** – V databázi nové mikroslužby vytvoříme tabulky pro jednotlivé zdroje, ke kterým bude přístup omezen na základě uživatelských rolí a naplníme je daty.

5. **Implementace funkcí** – Do nové mikroslužby implementujeme funkce, které má poskytovat a byly navrženy v předchozí kapitole.

6. **Přesměrovávání na novou mikroslužbu** – Přesměrujeme volání implementovaných funkcionalit do nové mikroslužby.

7. **Ukončení dočasných replikací** – Vypneme dočasné replikace tabulek, které se budou odstraňovat z původní aplikace (`password_resets`, `users_roles` a `roles`). Smažeme dočasná rozhraní, která sloužila k jejich replikaci.

8. **Odstranění staré funkcionality a dat** – Z monolitu odstraníme funkcionalitu podporující oddělenou funkcionalitu a z databáze odstraníme data, která v monolitu již nebudou potřeba.

9. **Vytvoření AWS Lambda** – Vytvoříme AWS Lambda funkci, která bude sloužit jako autentizátor/autorizátor požadavků využívající služby nově vytvořené mikroslužby.

10. **Autentizace a autorizace z API Gateway** – Vytvoříme rozhraní ve specifikaci OAS3 pro použití při tvorbě Amazon API Gateway. V definici využijeme v předchozím kroku vytvořenou AWS Lambda funkci.

11. **Odstranění zbývajících funkcí** – Odstraníme zbývající funkce, které se používali pro implementaci autentizace a autorizace voláním nové mikroslužby. V moment, kdy požadavek přijde do monolitické aplikace, je již autentizovaný a autorizovaný.

12. **Implementace pro autentizaci modulů** – Podobným procesem přidáme do nové mikroslužby funkcionalitu pro autentizaci modulů.

3.6 Notifikace uživatele

Funkcionalita pro notifikování uživatele bude nadále umožňovat posílat emaily a push notifikace. Nebude již dále podporovat možnost ověřit stav všech FCM tokenů / tokenů zvolených uživatelů. Naopak bude umožňovat ověřit stav jednoho konkrétního tokenu podle jeho identifikátoru. Mikroslužba bude nově umět zasílat SMS zvoleným uživatelům.

Spolu se zasíláním push notifikací souvisí funkcionalita pro správu verzí aplikací (`app_versions` a `version_tags` tabulky popisované v kapitole 2.9.5). Tato mikroslužba bude jejich primárním zdrojem. Pro ostatní mikroslužby prozatím nebude poskytovat rozhraní pro ověření, zda vybraná verze aplikace implementuje určitou funkcionalitu, neboť toto rozhodování je zatím využito pouze pro push notifikace.

3.6.1 Návrh aplikačního rozhraní

Kompletní návrh aplikačního rozhraní mikroslužby ve specifikaci OAS3 lze nalézt v příloze práce. Následující kapitola jej shrnuje.

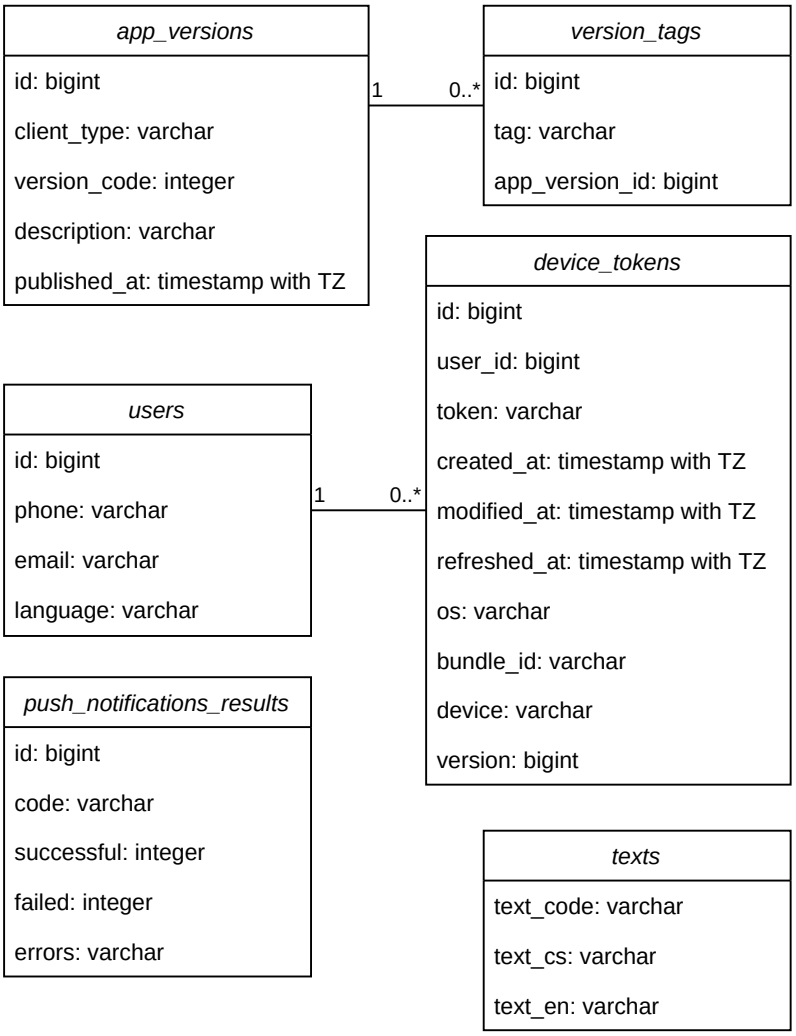
- Získání všech *app versions*.
- Přidání nové *app version* včetně vztahů k *version tags*. Tyto vztahy definuje seznam *features* a očekává se, že žádná *feature* nebude existovat pro platformu, pro kterou přidáváme *app version*.
- Úprava existující *app version*. Upraví parametry *app version* a pro *features*, pro které existuje ekvivalentní *version tag* upraví odkazovaný *app version* na upravovaný. Pro neexistující vytvoří nové *version tags*.
- Získání všech *version tags*.
- Přidání nového *version tag*. Nedovolí přidání, pokud již existuje *version tag* se stejným *tag* pro vybranou platformu, tj. podle *id* zvolené *app version*.
- Úprava existujícího *version tag*. Úprava se nepovede, pokud již existuje jiný *version tag* se stejnou hodnotou *tag* pro vybranou platformu.
- Uložení informací o zařízení a FCM tokenu. Informace o zařízení jsou očekávány v *user-agent* hlavičce, FCM token je v těle požadavku. Pokud v databázi existuje záznam se stejnou hodnotou FCM tokenu, ostatní informace se aktualizují. Pokud záznam s daným FCM tokenem neexistuje, pokusí se aplikace podle ostatních údajů najít záznam databáze, který odpovídá použitému zařízení a aktualizuje hodnotu FCM tokenu.
- Odstranění FCM tokenu. Po odhlášení z aplikace je uživatelův FCM token smazán.
- Odeslání mailu se zadanými parametry. Podle definovaného typu emailu se vybere šablona emailu. Z parametrů emailu se doplní dynamické části šablony a mail se odešle. Pro typ emailu *other* se pro tělo emailu použije vybraný parametr. V takovém případě se žádné doplňování parametrů neprovádí ani se nevybírá jazyk emailu.

- Poslání notifikace vybraným uživatelům. V těle požadavku se očekává seznam identifikátorů uživatelů, kteří mají push notifikaci obdržet. Dále se v těle požadavku očekává *nadpis* a *tělo* notifikace v obou podporovaných jazykových mutacích, volitelně je možné definovat *url*, které se má po kliknutí na notifikaci otevřít (lze specifikovat i *deeplink* do aplikace). Dalším volitelným parametrem je doba, po kterou se mají FCM servery pokoušet o zaslání push notifikace (odeslání se může opozdit např. z důvodu nedostupnosti internetového připojení v cílovém zařízení). Dle omezení Firebase [45] je možné specifikovat životnost maximálně 28 dní. Opačnou možností je hodnota nulová, potom se pokusí notifikaci doručit jedenkrát bez ukládání. Všechny hodnoty mezi 0 a 28 dny jsou možné. Návratovou hodnotou je unikátní kód, dle kterého lze získat výsledky odesílání, a počet zařízení, na které se bude notifikace posílat.
- Poslání notifikace všem uživatelům. Funguje stejně jako předchozí, jen se jako seznam cílových zařízení berou zařízení všech uživatelů.
- Získání výsledku odesílání push notifikací. Výsledkem bude agregace výsledků všech dosud vyřešených várek. Podrobněji v 3.6.2.
- Poslání SMS. Očekává text zprávy a seznam uživatelů. Aplikace neočekává různé jazykové mutace, jednoduše pošle získaný text zvoleným uživatelům. Výsledkem je informace o počtu úspěšných zaslání a ceně.
- Synchronizační rozhraní pro informace o uživateli. Aktualizuje email, preferovaný jazyk a telefonní číslo. Pokud se jedná o netradiční podezřelé číslo, je tato informace zanesena do výpisu.
- Synchronizační rozhraní pro smazání uživatele.
- Synchronizační rozhraní pro aktualizaci textů.
- Synchronizační rozhraní pro smazání textů.

Posílání SMS, emailů a push notifikací nevyužívá společnou logiku, ale všechny tyto funkce byly označeny jako způsoby notifikace, proto byly implementovány do jedné mikroslužby. Vytížení jednotlivých funkcí nebude vysoké, proto mohou být v jedné mikroslužbě s možností snadného oddělení v budoucnu.

3.6.2 Databázový model

Jak je vidět na obrázku 3.3, tabulky `app_versions` a `version_tags` zůstanou identické s původními (viz obrázek 2.3). Využívané parametry uživatelů si bude mikroslužba udržovat aktuální ve své databázi. Bude pracovat s telefonním číslem (pro posílání SMS), s emailem (pro posílání emailů) a preferovaným jazykem (pro výběr jazyka push notifikací a jazyku textů v emailech).



Obrázek 3.3: Navržené databázové schéma související s notifikováním uživatele

Tabulka `device_tokens` nebude používat datový typ jsonb, místo tohoto sloupečku definuje sloupečky `os`, `bundle_id`, `device` a `version`. Další změnou je rozšíření informací v `device_tokens` o dva datumy (přesněji *timestamp*). Datum `created_at` zůstavá a reprezentuje, kdy byl daný záznam vytvořen. Datum `modified_at` říká, kdy byl změněn token, nebo nějaké informace o zařízení a `refreshed_at` indikuje, kdy byl přijat požadavek, ale všechny informace byly aktuální.

Databáze bude dále rozšířena o tabulku shromažďující výsledky posílání push notifikací, tj. tabulka `push_notifications_results`. Posílání je rozděleno na várky podle použitého zařízení a do skupin po 100 tokenech (omezení

Firebase). Za každou takovou várku přibude do tabulky záznam, jak bylo odesílání úspěšné. Všechny várky pro vybraný požadavek budou mít stejnou hodnotu v parametru `code`.

Pro ukládání statických lokalizovaných textů pro šablony emailů se bude používat nová tabulka `texts`, která bude replikou tabulky z mikroslužby spravující texty.

3.6.3 Proces oddělení

Proces oddělení bude víceméně dodržovat proces popsaný v 3.4. Transformaci bude možné provádět postupně pro jednotlivé funkcionality (posílání push notifikací, mailů a SMS).

1. **Vytvoření repliky tabulky `users`** – V databázi monolitické aplikace vytvoříme novou tabulku pro udržování informací o uživateli pro potřeby nové mikroslužby (tj. bude udržovat pouze parametry `id`, `phone`, `email` a `language`) Při libovolné změně některého z vybraných parametrů bude změna zanesena i do této tabulky. Rozhraní pro aktualizaci změn v této tabulce bude pečlivě odděleno, aby bylo později snadné tuto aktualizaci provádět s pomocí HTTP požadavku na novou mikroslužbu. Podobným způsobem budou replikována data z původních tabulek `app_versions`, `version_tags` a `device_tokens`.

2. **Vytvoření nové mikroslužby** – Následně vytvoříme novou mikroslužbu, která bude pracovat s vlastní databází a zatím může poskytovat jen synchronizační rozhraní pro replikované tabulky.

3. **Přesměrování replikací** – Funkcionalitu pro replikaci tabulek přesměrujeme na rozhraní nové mikroslužby.

4. **Vytvoření nových tabulek** – V oddělené databázi nové mikroslužby vytvoříme tabulku pro texty (`texts`), které budou aktualizovány skrze synchronizační rozhraní z mikroslužby spravující texty a tabulku pro výsledky zasílání push notifikací (`push_notifications_results`).

5. **Implementace funkcí** – Do nové mikroslužby implementujeme funkce, které má poskytovat a byly navrženy v předchozí kapitole.

6. **Přesměrovávání na novou mikroslužbu** – Přesměrujeme volání implementovaných funkcionalit do nové mikroslužby.

7. **Ukončení dočasných replikací** – Vypneme dočasné replikace tabulek, které se budou odstraňovat z původní aplikace (`device_tokens`, `app_versions` a `version_tags`). Smažeme dočasná rozhraní, která sloužila k jejich replikaci.

8. **Odstranění staré funkcionality a dat** – Z monolitu odstraníme funkcionalitu podporující oddělenou funkcionalitu a z databáze odstraníme data, která v monolitu již nebudou potřeba.

3.7 Komunikace s platební bránou

Mikroslužba pro komunikaci s platební bránou bude muset umožňovat provádět verifikace platebních karet, provádět platby a generovat odkazy na provedení plateb, které nemohli být automaticky strhnuty. Mikroslužba nebude implementovat funkcionalitu pro tvorbu blokací. Bude se nadále využívat protokol SOAP pro komunikaci s platební bránou. Klientský kód se bude generovat s pomocí utility *wsimport* jako tomu bylo doposud.

3.7.1 Aplikační rozhraní

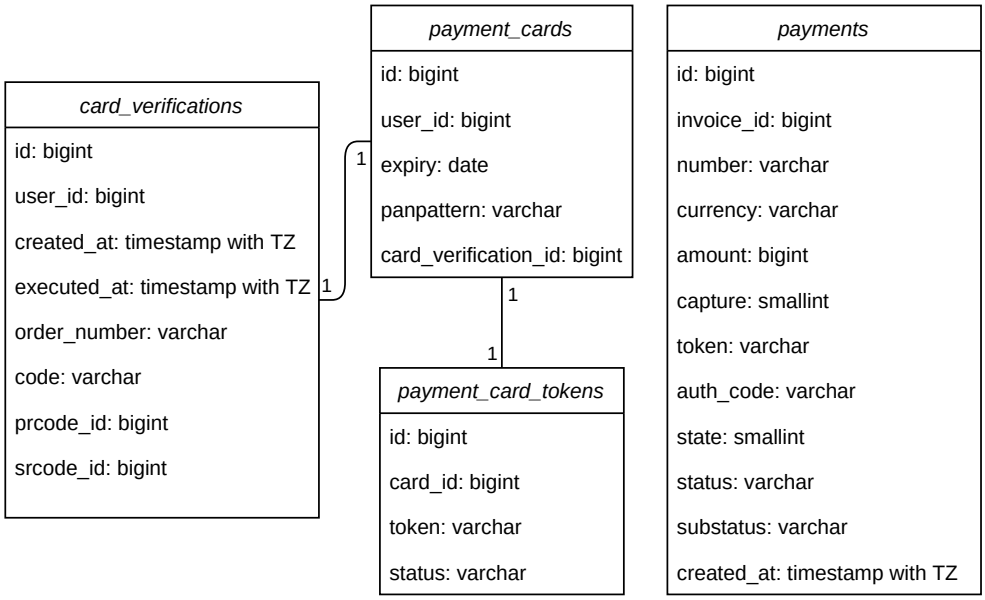
Kompletní návrh aplikačního rozhraní mikroslužby není součástí práce. Následující kapitola shrnuje, co bude muset mikroslužba umožňovat z pohledu rozhraní.

- Provedení platby na základě uživatelského identifikátoru, identifikátoru faktury, částky a měny.
- Ověření validity platební karty pro vybraného uživatele.
- Vygenerování odkazu pro verifikaci platební karty.
- Vygenerování odkazu pro manuální provedení platby.
- Zpracování výsledku platby od platební brány.
- Zpracování výsledku verifikace platební karty od platební brány.
- Synchronizační rozhraní pro odstranění uživatele. Provede anonymizaci informací o uživatelových platebních kartách.

Mikroslužba bude využívat rozhraní ostatních mikroslužeb. Bude informovat o zaplacení faktur službu spravující faktury. Bude informovat službu spravující uživatele o nedoplatcích, aby služba mohla uživatele přepnout do stavu *dlužník*. Bude informovat o neúspěšné platbě.

3.7.2 Databázový model

V obrázku 3.4 lze vidět schéma popisující tabulky udržované v databázi mikroslužby. Mikroslužba si bude udržovat informace o platbách (`payments`), platební karty (`payment_cards`), respektive jejich tokeny pro provádění plateb (`payment_card_tokens`) a tabulku reprezentující verifikace platebních karet (`card_verifications`). Tabulky `card_verifications`, `payment_cards`



Obrázek 3.4: Navržené databázové schéma související s komunikací s platební bránou

a `payments` budou udržovat uživatelský identifikátor. Tento identifikátor se nemůže a nebude měnit.

3.7.3 Proces oddělení

Proces oddělení bude víceméně dodržovat proces popsaný v 3.4. Tabulky se budou celé přenášet do nové mikroslužby bez plánovaných změn v jejich struktuře. Informace o zákazníkovi pro platební bránu budou přijaty v požadavku, který bude vyžadovat provedení platby, proto nebude nutné tyto informace ukládat (replikovat z jiné mikroslužby). Díky tomu by měl být proces transformace snazší než u ostatních mikroslužeb.

1. **Vytvoření dočasných replik** – V databázi monolitické aplikace vytvoříme nové tabulky, které budou udržovat replikovaná data o platbách, platebních kartách, jejich tokenech a verifikacích platebních karet.

2. **Vytvoření nové mikroslužby** – Následně vytvoříme novou mikroslužbu, která bude pracovat s vlastní databází a zatím může poskytovat jen synchronizační rozhraní pro replikované tabulky.

3. **Přesměrování replikací** – Funkcionalitu pro replikaci tabulek přesměrujeme na rozhraní nové mikroslužby.

4. **Implementace funkcí** – Do nové mikroslužby implementujeme funkce, které má poskytovat a byly navrženy v předchozí kapitole.

5. **Přesměrovávání na novou mikroslužbu** – Přesměrujeme volání implementovaných funkcionalit do nové mikroslužby.

6. **Ukončení replikací** – Ukončíme replikace tabulek. Všechny tabulky budou primárně spravovány mikroslužbou pro správu komunikace s platební bránou. Rovněž smažeme dočasná rozhraní, která sloužila k jejich replikaci.

7. **Odstranění staré funkcionality a dat** – Z monolitu odstraníme funkcionalitu podporující oddělenou funkcionalitu a z databáze odstraníme data, která v monolitu již nebudou potřeba.

3.8 Cenotvorba

Mikroslužba starající se o cenotvorbu bude podporovat jak minutový, tak hodinový ceník. Tedy bude umožňovat stanovit ceny jednotlivých součástí ceny rezervace včetně minimální ceny, kterou bude možné snížit slevou. Rovněž umožní definovat slevy, u kterých bude evidovat průběh jejich čerpání. Funkcionalita starající se o slevy je kandidátem pro oddělení do samostatné mikroslužby v dalších fázích vývoje.

Dále bude udržovat skupiny automobilů pro aplikování skupinových ceníků. Přidělení automobilů do skupin bude časově omezené. Pro každou skupinu automobilů bude definován ceník, od kterého se budou moci odvíjet jízdní ceníky. Mikroslužba bude rovněž spravovat přiřazení do uživatelských skupin, které reprezentují nárok na využití vybraných jízdních ceníků.

Služba bude umožňovat přiřazovat uživatelům slevy různých typů (sleva na minuty, n km nad limit zdarma, výhodnější tarif na 24 hodinový balíček apod.). Součástí implementace bude snadné zjišťování přidělování, čerpání a expirování slev.

3.8.1 Aplikační rozhraní

Kompletní návrh aplikačního rozhraní mikroslužby není součástí práce. Následující kapitola shrnuje, co bude muset mikroslužba spravující ceníky a výpočet cen rezervací umožňovat z pohledu rozhraní pro ostatní části systému – pro jiné mikroslužby.

- Správa skupinových ceníků
  - Získání všech skupinových ceníků.
  - Získání detailu konkrétního skupinového ceníku.

  - Vytvoření skupinového ceníku pro vybranou skupinu automobilů. Aplikace umožní existenci pouze jednoho skupinového ceníku pro vybraný interval platnosti ceníku. Interval platnosti ceníku musí začínat a končit (platnost může být neomezená) v budoucnosti.
  - Úprava platnosti vybraného skupinového ceníku. Platnost opět nebude moci kolidovat s ostatními ceníky pro vybranou skupinu automobilů. Pokud je upravován ceník, který začne platit v budoucnosti je možné upravit i začátek platnosti vybraného ceníku. Konec platnosti ceníku nelze nastavit do minulosti.

- Správa skupin automobilů
  - Získání všech skupin automobilů.
  - Získání detailu konkrétní skupiny automobilů. Součástí bude seznam automobilů, které do skupiny patří, včetně intervalu platnosti přiřazení do skupiny.
  - Vytvoření nové skupiny automobilů.
  - Úprava popisu skupiny automobilů.
  - Přiřazení automobilu do skupiny automobilů. Auto nemůže patřit do více skupin v jeden konkrétní čas. Bude možné přiřadit více automobilů najednou. Čas začátku přiřazení musí být v budoucnosti.
  - Ukončení přiřazení automobilu do skupiny automobilů. Bude umožněno ukončit přiřazení pro více automobilů současně. Čas ukončení přiřazení musí být v budoucnosti.

- Správa jízdních ceníků
  - Získání dat pro formulář pro tvorbu jízdního ceníku. Na tomto požadavku bude spolupracovat s mikroslužbou pro správu faktur. V odpovědi bude seznam volitelných uživatelských skupin, seznam skupin automobilů a seznam kódů identifikujících složky ceny, které s použitím ceníku mohou vzniknout (použije se pro mapování na položky faktur).
  - Získání všech jízdních ceníků.
  - Získání detailu konkrétního jízdního ceníku.
  - Vytvoření nového jízdního ceníku. Při tvorbě jízdního ceníku je nutné, aby existoval skupinový ceník pro vybranou skupinu v definovaný interval platnosti jízdního ceníku.
  - Úprava existujícího jízdního ceníku. U jízdního ceníku bude možné upravit pouze jeho interval platnosti, podobně jako v případě skupinového ceníku.

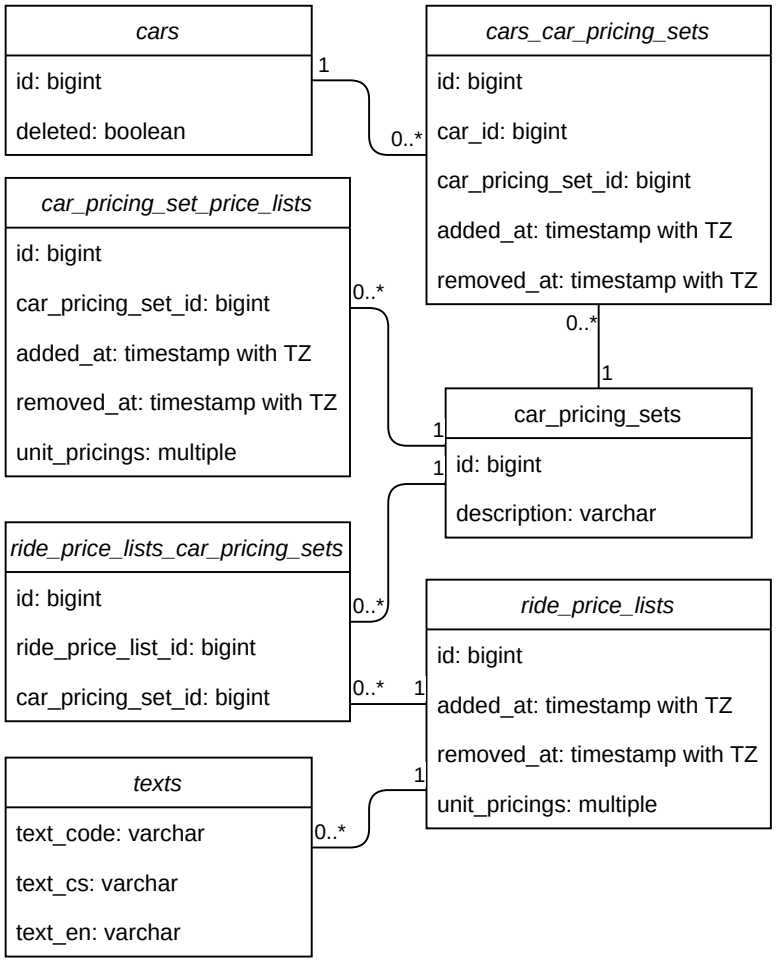
  - Získání textu popisujícího zvolený jízdní ceník pro usnadnění výběru ceníku před rezervací.

- Správa přiřazení uživatelů do uživatelských skupin
  - Přiřazení uživatele do uživatelské skupiny.
  - Úprava přiřazení uživatele do uživatelské skupiny. Čas ukončení přiřazení uživatele do skupiny nemůže být v minulosti.
  - Získání seznamu uživatelských skupin, které může uživatel použít pro nadcházející rezervaci.

- Zjišťování ceny rezervací.
  - Zjištění průběžné ceny rezervace.
  - Zjištění finální ceny rezervace.
  - Zjištění ceny potenciální rezervace na základě poskytnutých parametrů s možností volby ceníku.

- Správa slev a přiřazení jednotlivým uživatelům.
  - Získání všech slev.
  - Získání detailu konkrétní slevy.
  - Vytvoření nové slevy.
  - Přiřazení slevy uživateli. Bude možné specifikovat seznam uživatelů, kterým má být sleva přiřazena.
  - Získání detailu o čerpání slev vybraným uživatelem.

- Synchronizační rozhraní pro duplikované entity, které v mikroslužbě slouží pouze ke čtení.
  - Synchronizační rozhraní pro přidání automobilu.
  - Synchronizační rozhraní pro odebrání automobilu.
  - Synchronizační rozhraní pro přidání nového uživatele.
  - Synchronizační rozhraní pro odebrání uživatele.
  - Synchronizační rozhraní pro přidání uživatelské skupiny.
  - Synchronizační rozhraní pro úpravu uživatelské skupiny.
  - Synchronizační rozhraní pro texty použité pro popis jízdních ceníků.



Obrázek 3.5: Navržené databázové schéma související se skupinovými ceníky

3.8.2 Databázový model

Obrázky 3.5 a 3.6 popisují databázové schéma související s cenotvorbou. Diagramy byly rozděleny přes tabulku `ride_price_lists`. Tabulka `cars` bude replikou z mikroslužby spravující automobily. Auta budou nově přiřazovaná do skupin (tabulka `car_pricing_sets`) manuálně, ne na základě modelu. Přiřazení bude reprezentováno záznamem ve vazební tabulce. Rovněž bude umožněno přiřazení do skupiny časově omezit (parametry `added_at` a `removed_at`).

Parametr `unit_pricings` tabulky `car_pricing_set_price_lists` popisuje parametry související s definicí jednotlivých složek ceny. Mezi ně patří *cena za kilometr*, *cena za hodinu výpůjčky* a *maximální počet placených hodin za 24 hodin rezervace* pro hodinový ceník a *počet volných kilometrů na 24 hodin rezervace*, *cena za kilometr nad limit*, *cena za minutu jízdy*, *cena za minutu*

stání a *maximální cena za minuty za 24 hodin rezervace* pro minutový ceník. Pro oba ceníky existuje parametr *minimální cena*.

Tabulka `ride_price_lists` reprezentuje jízdní ceníky. Pod parametrem `unit_pricings` se skrývají stejné parametry jako u skupinových ceníků a navíc definuje příznak určující, zda se jedná o definici hodinového nebo minutového ceníku. Dále obsahuje příznak určující, zda se jedná o relativní nebo absolutní ceník a příznak, zda je ceník určen na jízdy zdarma. Jízdní ceníky mají vztah ke složkám ceny (tabulka `price_components`). Tabulka obsahující složky ceny obsahuje informace nutné k párování složek ceny na popisující text položek faktur. Tabulka `price_component_type` definuje typ složky ceny (např. cena za minutu jízdy). Jednotlivé typy složek ceny mohou mít různé popisy na faktuře pro různé jízdní ceníky, respektive uživatelské slevy.

Tabulky `texts`, `users` a `user_groups` jsou replikami tabulek z ostatních mikroslužeb spravující příslušné entity. Tabulka `users_user_groups` je spravována touto mikroslužbou a udává interval a celkový a zbývající počet možných použití uživatelské skupiny. Tabulku textů odkazuje jízdní ceník. Do odkazovaného textu doplní své jednotkové ceny a vrátí ho při dotazu na popis ceníku.

Data pro výpočet ceny rezervace jsou uložena v tabulce `reservation_data` a vypočtená celková cena je uložena do tabulky `reservation_pricing`. Vypočtené ceny za jednotlivé složky jsou uloženy jako jednotlivé záznamy pro každou složku ceny v tabulce `reservation_price_components`.

V tabulce `discounts` je uložena sleva. Parametr `data_defining_discount` skrývá množinu parametrů popisující jednotlivé vlastnosti slev. Slevy přiřazené uživatelům jsou reprezentovány v tabulce `user_discounts`. Tabulka `reservation_pricings_discounts` ukládá informace o čerpání uživatelských slev.

3.8.3 Proces oddělení

Proces oddělení bude víceméně dodržovat proces popsaný v 3.4. Mikroslužba bude nejprve pracovat nad stejným schématem, který se používá v monolitické aplikaci. Po oddělení a odstranění bude schéma postupně zmigrováno na novou verzi popsanou v předchozí kapitole.

1. **Vytvoření replik přenášených tabulek** – V databázi monolitické aplikace vytvoříme nové tabulky pro udržování informací, které bude vyžadovat nová mikroslužba. Rozhraní pro aktualizaci změny v tabulkách, u kterých bude pokračovat replikace dat i po oddělení bude pečlivě odděleno, aby bylo později snadné tuto aktualizaci provádět s pomocí HTTP požadavku na novou mikroslužbu. Nová mikroslužba bude i po oddělení spravovat repliky entit v tabulkách automobilů, uživatelů, uživatelských skupin a rezervačních dat pro výpočet ceny.

2. **Vytvoření nové mikroslužby** – Následně vytvoříme novou mikroslužbu, která bude pracovat s vlastní databází a zatím může poskytovat jen synchronizační rozhraní pro všechny replikované tabulky.

3. **Přesměrování replikací** – Funkcionalitu pro replikaci tabulek přesměrujeme na rozhraní nové mikroslužby.

4. **Vytvoření nových tabulek** – V oddělené databázi nové mikroslužby vytvoříme tabulku `reservation_pricings` pro ukládání celkové ceny a tabulku `reservation_pricing_components` pro ukládání vypočtené výše ceny za jednotlivé složky.

5. **Implementace funkcí** – Do nové mikroslužby implementujeme funkce, které se starým modelem může poskytovat.

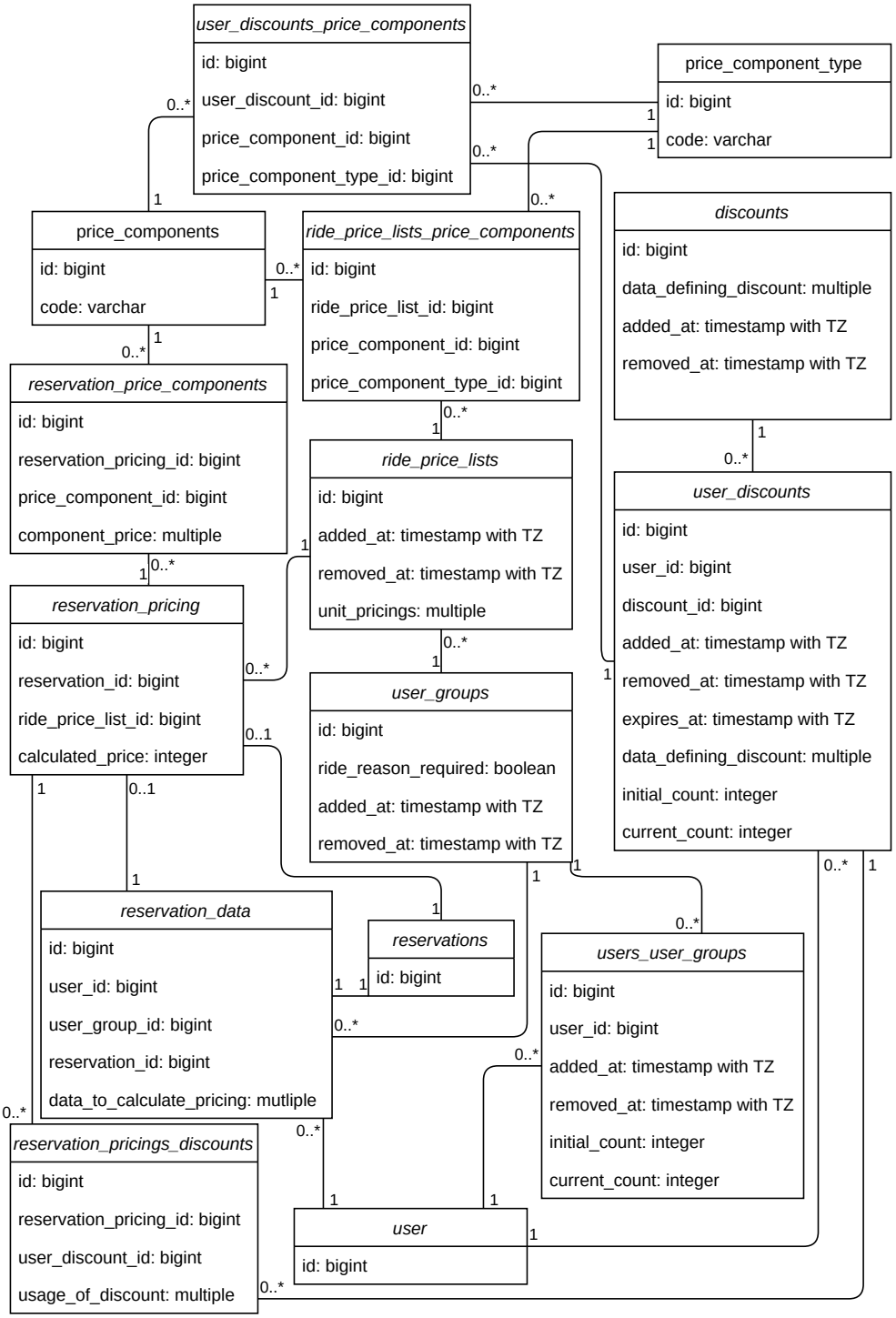
6. **Přesměrovávání na novou mikroslužbu** – Přesměrujeme volání implementovaných funkcionalit do nové mikroslužby.

7. **Ukončení dočasných replikací** – Vypneme dočasné replikace tabulek, které se budou mazat z původní aplikace (tabulky reprezentující skupinové a jízdní ceníky a vazební tabulky mezi modely automobilu a jízdními ceníky a mezi uživateli a uživatelskými skupinami). Smažeme dočasná rozhraní, která sloužila k jejich replikaci.

8. **Odstranění staré funkcionality a dat** – Z monolitu odstraníme funkcionalitu podporující oddělenou funkcionalitu a z databáze odstraníme data, která v monolitu již nebudou potřeba.

9. **Implementace nových funkcí** – Do nové mikroslužby implementujeme funkcionalitu pro nové skupinové ceníky (nevyužívající modely automobilů) a ukončíme replikaci této tabulky z monolitické aplikace. Dále implementujeme funkcionalitu spojenou s tabulkami udržující popisy složek ceny a upravíme rozhraní pro tvorbu jízdních ceníků.

10. **Implementace slev** – Provedeme implementaci upravených slev a odstraníme funkcionality slev z jízdních ceníků.



Obrázek 3.6: Navržené databázové schéma související s jízdními ceníky a slevami

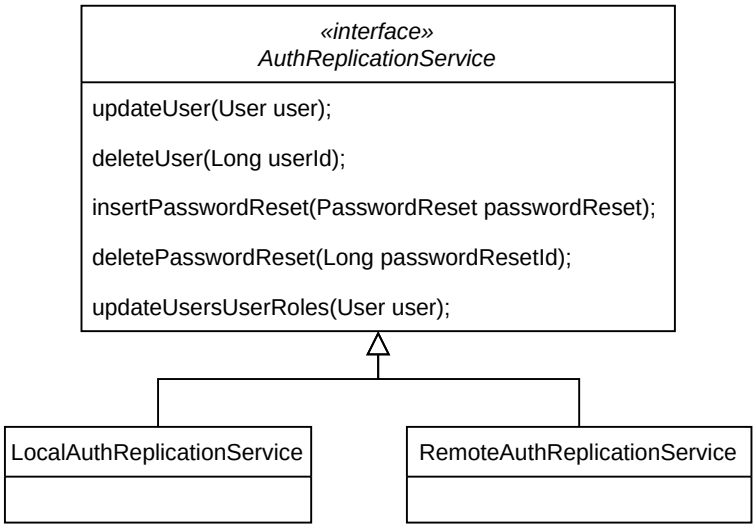
Kapitola 4

Implementace

U všech funkcionalit spadajících do nových mikroslužeb, na které se měla práce zaměřit, byla zahájena transformace. Vzhledem k tomu, že byl zvolen přístup, ve kterém se snažíme duplikovat data, abychom se nemuseli ptát ostatních mikroslužeb, začalo se prvním testovacím krokem – lokální replikace souvisejících dat. Čtení dat je mnohem častější operace než jejich úprava či vkládání. Je tedy ve většině případů efektivnější mít data vždy dostupná ve vlastní databázi než se na ně dotazovat jiné mikroslužby. S tímto krokem souviselo vyhledání míst, kdy se data mění a změnu provést i v lokální kopii dat. Před spuštěním lokálních kopií je nutné zkopírovat stávající data do nových tabulek. Musíme také zajistit, aby se v době mezi zkopírováním dat a spuštěním replikací neprovedli žádné změny. Po určitě době zkontrolujeme, že se data replikovala správně a budeme moci pokračovat dalším krokem.

Dalším krokem je příprava rozhraní nových mikroslužeb pro příjem změn v duplikovaných datech. Pro tento účel je vytvořena definice API v OAS3, která se rovněž používá pro vygenerování kostry serverové části kódu. Definice rozhraní pro replikaci do jiné databáze se zatím vytvořila pouze pro mikroslužbu na autentizaci a autorizaci uživatele. Její vytvoření pro ostatní mikroslužby společně s popisem v kapitolách 3.6.3, 3.7.3 a 3.8.3 je relativně přímočaré a bude moci být provedeno i jiným členem týmu. Součástí tohoto kroku je také nutné provést vytvoření tabulek v nových databázích, které budou udržovat replikovaná data. Pro tento účel se využijí připravené SQL skripty z prvního kroku transformace (pouze upravíme názvy tabulek).

Třetím krokem je implementace replikace do nových databází s pomocí připraveného rozhraní ve druhém kroku. Pro data replikovaná do mikroslužby na autentizaci a autorizaci se vytvořila alternativní implementace (viz obrázek 4.1) služby pro replikaci dat do lokálních replik. Tato implementace využívá vygenerovaného klientského kódu z OAS3. Provedení replikace se provádí asynchronně v jiném vlákně, podobně jako v případě lokální replikace, abychom replikací neopožďovali odpovědi na řešené požadavky. Podobně se bude postupovat i u zbylých mikroslužeb.



Obrázek 4.1: Implementace rozhraní pro replikace dat

Pokračujeme přesunem funkcí z monolitu do mikroslužeb. Součástí tohoto kroku je vytvoření definice rozhraní v OAS3. Zatím bylo vytvořeno pro mikroslužbu na autentizaci a autorizaci a mikroslužbu pro správu notifikací.

Dalším krok zahrnuje přesměrování vykonávání funkcí do nových mikroslužeb, tj. úprava monolitické aplikace v místech volání součástí, které byly přesunuty do nových mikroslužeb. Tato úprava zatím nebyla provedena. Bude vhodné provést úpravu tak, aby se nejprve otestovalo, že všechny funkce probíhají správně a to tak, že se operace bude provádět jak v původním monolitu (který bude stále ta část systému, která má pravdu), tak i v nových mikroslužbách (nejlépe postupovat iterativně po jedné mikroslužbě) a vyhodnotí se, že výsledky operací jsou stejné. Musíme při tom dát pozor, abychom z mikroslužeb neprováděli nechtěné operace, například abychom neprovedli duplicitní pokus o provedení platby. Po určitém čase provedemé vyhodnocení, zda se mikroslužba chová tak, jak má – stejně jako monolitická aplikace a provádění operací přenecháme čistě na mikroslužbě a stane se tak zdrojem pravdy. Pro operace dostupné zvenku (z klientských aplikací / z administrátorské aplikace), které zahrnují pouze funkce z nové mikroslužby se na přesměrování může využít úprava API Gateway. Jedná se o jednoduchou úpravu rozšířené specifikace OAS3 – úprava cílového serveru na vyřešení požadavku (parametr `uri` v rámci parametru `x-amazon-apigateway-integration`).

V této fázi se již nevyužívá funkcionalita v monolitické aplikaci a tudíž může být odstraněna společně s daty, která byla přesunuta do mikroslužeb. Společně s odstraněním staré funkcionality se může odstranit i kód pro dočasnou replikaci, který se už také nevyužívá.

Nakonec se provedou úpravy v implementacích mikroslužeb tak, aby od-

povídali návrhu z kapitoly 3. Například úprava mikroslužby spravující cenotvorbu bude nově dělit automobily do skupin nezávisle na modelu auta. Implementace se upravuje až po oddělení, aby bylo při oddělování snazší provádět menší kroky a nemuseli jsme rovnou začít využívat celou mikroslužbu s jinou aplikační logikou a s daty netriviálně transformovanými z monolitické databáze.

Pro vývoj mikroslužeb se vytvořil společný základ (viz konec kapitoly 3). Tento společný kód je udržován v odděleném git repozitáři. Všechny ostatní mikroslužby mají tento repozitář nastavený jako alternativní vzdálený repozitář a mohou si tak snadno zavést změny do své kopie společného kódu `git pull template master --allow-unrelated-histories`.

4.1 Použitá existující řešení

V rámci monolitické aplikace se používali vybrané existující knihovny a nástroje. Pro usnadnění vývoje mikroslužeb bylo rozhodnuto o využití některých z nich a některých nových, přičemž různé mikroslužby mohou využívat jiné knihovny. Následuje seznam použitých existujících řešení.

Java Play Framework Jak již bylo zmíněno pro tvorbu jednotlivých mikroslužeb byl zvolen Java Play Framework. Byl zvolen, neboť většina členů týmu je s tímto frameworkem obeznámena a jeho používání je relativně snadné.

OpenAPI generátor Pro generování kostry serverového kódu a klientského kódu se využívá OpenAPI generátor, který umí generovat kód přímo pro Java aplikace v Play Frameworku. Generátor funguje na principu Mustache šablon [46]. Některé šablony byly upraveny pro potřeby projektu. Vygenerování kódu se pak spouští z příkazové řádky s pomocí příkazu `java -jar $generator batch -- $path_to_configuration`, kde proměnná `generator` odkazuje na sestavený JAR generátoru a parametr `path_to_configuration` je cesta k souboru s konfigurací pro generátor. Nastavuje mj. volbu generátoru, cestu, kam se mají soubory vygenerovat, cestu ke specifikaci apod.

JOOQ generátor Slouží pro generování tříd souvisejících s komunikací s relační databází. Je nastaven tak, aby generoval DAO třídy a POJO třídy reprezentující záznamy v databázi. Pro komplexnější dotazování vlastní jazyk podobný SQL. Jednoduchý dotaz lze vidět ve výpisu 1.

Knihovny pro Swagger UI Spolu s vygenerovaným rozhraním z API specifikace se generuje rozhraní, které umožní zobrazení specifikace ve uživatelský přátelské formě Swagger UI.

Retrofit2 Pro klientský kód se použila knihovna retrofit2.

```
ctx().select(AppVersions.APP_VERSIONS.fields())
    .from(AppVersions.APP_VERSIONS)
    .where(AppVersions.APP_VERSIONS.CLIENT_TYPE
        .eq(clientType)
        .and(AppVersions.APP_VERSIONS.VERSION_CODE
            .eq(versionCode)))
    .fetchInto(AppVersions.class);
```

Výpis kódu 1: Databázový dotaz s pomocí JOOQ DSL

javax.validation Obsahuje anotace pro validaci vstupních (případně výstupních) objektů přicházejících skrze vystavené rozhraní. Anotace se automaticky generují společně s DTO.

com.fasterxml.jackson Knihovna pro snadnou práci s JSON. Využivá se pro serializaci a deserializaci DTO.

guice Dle doporučení Play Framework byla zvolena knihovna guice jako systém pro vkládání závislostí.

postgresql Knihovny pro připojení k relační databázi.

evolutions Výchozí systém pro podporu migrací databáze v Play frameworku.

JUnit Framework pro psaní jednotkových testů.

Mockito Framework pro tvorbu zástupných objektů pro testovací účely.

play.test Součást Play Framework usnadňující testování běžící aplikace pro testovací účely.

firebase-admin Posktuje rozhraní pro posílání push notifikací.

play-mailer Součást Play Framework umožňující snadné posílání emailů.

jaxb-api Knihovna pro práci s XML pro použití v komunikaci s SMS bránou.

jbcrypt Hashování uživatelských hesel a porovnávání přihlašovacích údajů.

java-jwt Tvorba a verifikace JWT.

4.2 Dokumentace

Nezbytnou součástí implementace je dokumentování API. Jak již bylo zmíněno, využívá se pro generování částí kódu a zároveň jako informace, jak se mají klientské aplikace s aplikací integrovat. Pro snadné psaní definic se využívá nástroj Apicurito [47]. Umožňuje definovat většinu používaných parametrů. Neumí definovat veškerá omezení pro validaci parametrů, ale to lze

```
@Test
public void
throwIfAnyFeatureAlreadyAssigned_allNewFeatures_shouldSucceed()
throws AlreadyExistsException {
    AppVersionsDao appVersionsDaoMock
        = mock(AppVersionsDao.class);
    AppVersionService appVersionService
        = new AppVersionService(appVersionDaoMock);
    StoreAppVersionDto dto = new StoreAppVersionDto();
    dto.setClientOs("anyOS");
    dto.setFeatures(Collections.singletonList("anyFeature"));
    when(
        appVersionsDaoMock.getForClientTypeAndTag(
            any(String.class),
            any(String.class)
        )
    )
    .thenReturn(Collections.emptyList());
    appVersionService.throwIfAnyFeatureAlreadyAssigned(dto);
}
```

Výpis kódu 2: Jednotkový test se zástupným objektem

dodefinovat ručně přímo v JSON definici rozhraní. API dokumentace, která bude vytvořena pro všechny vstupní body se použije pro vytvoření rozhraní v systému Amazon API Gateway. Většina metod je opatřena Javadoc komentářem pro snazší pochopení funkcí metod včetně popisu parametrů a návratové hodnoty. Dokumentace, jak vytvářet nové funkcionality a nové mikroslužby, je průzběžně přidávána do interní wiki projektu Uniqway.

4.3 Testování

Při vývoji mikroslužeb je kladen velký důraz na testovatelnost. Po naklonování repozitáře se šablonou pro mikroslužby je možné snadno začít psát testy všech plánovaných úrovní (jednotkové testy, funkční testy, testy rozhraní).

Jednotkové testy by měli testovat malou část funkcionality, obvykle kód jedné metody. Všechny jednotkové testy jsou ve společném balíčku `unit` a poté jsou rozděleny do balíčků podle toho, v jakém balíčku je umístěna třída, která je testem testována. Výpis kódu 2 ukazuje, jak lze definovat zástupný objekt, nadefinovat jeho chování a použit při testování.

Další formou jednotkového testování je s pomocí parametrizovaných testů. V takovém případě definujeme seznam seznamu parametrů, kde každý seznam je použit pro jeden běh. V ukázce 3 lze vidět testování metodu služby spra-

```
@Parameterized.Parameters(name = "{index}: {0} {1}")
public static Collection<Object[]> data() {
    return Arrays.asList(new Object[][]{
            {"iOS/13.5.1", "iOS"},
            {"Android/10", "Android"},
            {"", ""},
            {null, ""}
    });
}

@Test
public void getClientType_shouldEqualExpectedClientType() {
    DeviceTokens deviceToken = new DeviceTokens();
    deviceToken.setOs(os);
    assertEquals(expectedClientType,
        deviceTokenService.getClientType(deviceToken));
}
```

Výpis kódu 3: Jednotkový parametrizovaný test

vující FCM tokeny. Prvním parametrem je vstupní hodnota, druhý parametr je očekávaná hodnota. Pro zprovoznění parametrizovaného testu musíme vytvořit konstruktor přijímající seznam parametrů a nad třídu přidat anotaci `@RunWith(value = Parameterized.class)`.

Pro komplexnější testování se využívá běžící aplikace. Pro otestování konkrétní třídy lze s pomocí vkládání závislostí získat instanci a test provést nad touto instancí. Pro získání instance třídy `DeviceTokensDao` provedeme `app.injector().instanceOf(DeviceTokensDao.class)`, kde proměnná `app` je poděděná z nadtřídy od Play Framework. Při tomto testování může aplikace komunikovat s databází, proto byla vytvořena logika, která umožní definici počáteční sady dat. Obsah databáze se pročišťuje po každém testu. V ukázce 4 lze vidět test, který ověřuje, že smazání entity opravdu proběhne.

Poslední formou automatického testu je test rozhraní. V takové situaci se aplikaci pošle HTTP požadavek a ověřuje se, jak se s ním vypořádá. Využívá stejnou instanci aplikace jako funkční testy, proto je snadné ověřit výsledek operace, i když nemáme vystavené rozhraní pro změněná data (např. ukončení rezervace provede mnoho úkonů, které bychom musel ověřovat mnoha voláními na různá rozhraní). Při těchto testech je nutné vytvořit zástupné objekty pro klientská rozhraní, která může aplikace pro řešení daných požadavků využívat. V ukázce 5 lze vidět nastavení instance zástupného objektu do systému vkládání závislostí. V ukázce 6 lze vidět, jak lze provést požadavek na vybrané rozhraní aplikace

Pro testování kooperace mikroslužeb se použijí existující API testy psané

```
public class DeviceTokensDaoTest extends WithDatabase {

    private DeviceTokensDao dao() {
        return app.injector().instanceOf(DeviceTokensDao.class);
    }

    @Test
    public void deleteByToken_existingToken_shouldDelete() {
        assertEquals(2L, dao().findAll().size());
        dao().deleteByToken("TOKEN");
        assertEquals(1L, dao().findAll().size());
    }
}
```

Výpis kódu 4: Funkční test pracující s databází

```
@Override
protected Application provideApplication() {
    com.google.inject.Module testModule =
            new AbstractModule() {
                @Override
                public void configure() {
                    bind(ApiClient.class)
                        .toInstance(apiClientMock);}
                }
            };
    return new GuiceApplicationBuilder()
                .overrides(testModule).build();
}
```

Výpis kódu 5: Vkládání instance zástupného objektu do systému vkládání závislostí

```
Http.RequestBuilder request
    = Helpers.fakeRequest().method(POST).uri("/sms");
request.bodyJson(Json.toJson(dto));
Result result = route(app, request);
```

Výpis kódu 6: Provedení požadavku

v jazyce Python. Při testování kooperace jde především o otestování, zda se služby najdou a navzájem si rozumí. Podniková logika není předmětem těchto testů.

Závěr

Cílem práce bylo navrhnout a implementovat proces transformace backendového systému služby sdílení automobilů z monolitické architektury do architektury mikroslužeb.

Cíl práce byl velmi ambiciózní, i přesto bylo zadání splněno. Analýza architektury mikroslužeb poskytla užitečné informace pro návrh a implementaci transformace. Před navržením procesu transformace byla zanalyzována aktuální serverová aplikace, aby se navržené kroky transformace mohli detailně zaměřit na konkrétní problémy. Proces transformace byl navržen pro všechny funkcionality, na které se měla práce zaměřit. Proces transformace byl zahájen pro vybrané funkcionality, bohužel nebylo možné transformaci dokončit před odevzdáváním práce z důvodu komplexity, časové náročnosti a jelikož se změny provádí na neustále vyvíjejícím se projektu. Provedení transformace zbylých mikroslužeb bude díky této práci relativně snadná i pro jiného člena vývojového týmu. Jedna z plánovaných mikroslužeb je téměř připravena na započetí využití v produkčním prostředí. Pro ostatní plánované mikroslužby byly započaty první kroky k oddělení. Pro nově vzniklé mikroslužby jsou psány automatické testy a dokumentace poskytovaného rozhraní. Další dokumentace je tvořena ve formě Javadoc a v rámci interní wiki.

Z důvodu práce na fungujícím projektu se postupovalo obezřetně. Byla kladena snaha na vyvarování se chybám, které by mohli negativně ovlivnit uživatele. I proto byl kladen důraz na poctivé a opakovatelné testování. Pro snazší orientaci v kódu a dle zadání práce byla tvořena dokumentace.

Bude se pokračovat v oddělování mikroslužeb. Bude se dokončovat oddělování mikroslužeb, které se započalo v rámci této práce. Rovněž se bude pokračovat v oddělování dalších funkcionalit ze zbytku monolitické aplikace.

Bibliografie

1. VACHULA, Richard. *Databáze a rozhraní pro modul automobilu pro systém sdílení automobil více uživateli*. 2017. bakalářská práce. FEL ČVUT.
2. DÖRNER, Petr. CARSHARING: JEZDIT AUTEM, ALE NEVLASTNIT HO. *ŠKODA Storyboard* [online]. 2019 [cit. 2021-03-11]. Dostupné z: `https://www.skoda-storyboard.com/cs/inovace/mobilita/carsharing-jezdit-autem-ale-nevlastnit-ho/`.
3. TÝM UNIQWAY. Náš příběh [online] [cit. 2021-03-11]. Dostupné z: `https://uniqway.cz/about`.
4. PROUZA, Petr. *Rozšíření Android aplikace pro uživatele systému sdílení automobilů o reward shop*. 2019. bakalářská práce. FIT ČVUT.
5. RAVAS, Filip. *Scalability of Car Sharing System*. 2019. diplomová práce. FEL ČVUT.
6. Webová služba (web service). *ManagementMania.com* [online]. 2016 [cit. 2021-03-11]. Dostupné z: `https://managementmania.com/cs/webova-sluzba`.
7. FOWLER, Martin. Software Architecture Guide [online]. 2019 [cit. 2021-03-11]. Dostupné z: `https://martinfowler.com/architecture/`.
8. RICHARDSON, Chris. The Scale Cube [online]. 2020 [cit. 2021-03-12]. Dostupné z: `https://microservices.io/articles/scalecube.html`.
9. PRASAD, RadhaKrishna. Application Scalability — How To Do Efficient Scaling [online]. 2019 [cit. 2021-03-12]. Dostupné z: `https://dzone.com/articles/application-scalability-how-to-do-efficient-scalin`.
10. RICHARDSON, Chris. Pattern: Monolithic Architecture [online]. 2019 [cit. 2021-03-12]. Dostupné z: `https://microservices.io/patterns/monolithic.html`.

11. RICHARDSON, Chris. Pattern: Microservice Architecture [online]. 2020 [cit. 2021-03-12]. Dostupné z: https://microservices.io/patterns/microservices.html.

12. RICHARDSON, Chris. Pattern: Decompose by business capability [online]. 2020 [cit. 2021-03-12]. Dostupné z: https://microservices.io/patterns/decomposition/decompose-by-business-capability.html.

13. RICHARDSON, Chris. Pattern: Decompose by subdomain [online]. 2020 [cit. 2021-03-12]. Dostupné z: https://microservices.io/patterns/decomposition/decompose-by-subdomain.html.

14. Domains and Subdomains. *DDD - The Domain Driven Design* [online]. 2019 [cit. 2021-03-11]. Dostupné z: https://thedomaindrivendesign.io/domains-and-subdomains/.

15. FOWLER, Martin. BoundedContext [online]. 2014 [cit. 2021-03-12]. Dostupné z: https://martinfowler.com/bliki/BoundedContext.html.

16. RICHARDSON, Chris. Pattern: Database per service [online]. 2020 [cit. 2021-03-12]. Dostupné z: https://microservices.io/patterns/data/database-per-service.html.

17. RICHARDSON, Chris. Pattern: Saga [online]. 2020 [cit. 2021-03-12]. Dostupné z: https://microservices.io/patterns/data/saga.html.

18. RICHARDSON, Chris. Pattern: Event sourcing [online]. 2020 [cit. 2021-03-12]. Dostupné z: https://microservices.io/patterns/data/event-sourcing.html.

19. RICHARDSON, Chris. Pattern: Transactional outbox [online]. 2020 [cit. 2021-03-12]. Dostupné z: https://microservices.io/patterns/data/transactional-outbox.html.

20. RICHARDSON, Chris. Pattern: API Composition [online]. 2020 [cit. 2021-03-12]. Dostupné z: https://microservices.io/patterns/data/api-composition.html.

21. RICHARDSON, Chris. Pattern: Command Query Responsibility Segregation (CQRS) [online]. 2020 [cit. 2021-03-12]. Dostupné z: https://microservices.io/patterns/data/cqrs.html.

22. RICHARDSON, Chris. Pattern: API Gateway / Backends for Frontends [online]. 2020 [cit. 2021-03-12]. Dostupné z: https://microservices.io/patterns/apigateway.html.

23. RICHARDSON, Chris. Pattern: Service registry [online]. 2020 [cit. 2021-03-12]. Dostupné z: https://microservices.io/patterns/service-registry.html.

24. RICHARDSON, Chris. Pattern: Client-side service discovery [online]. 2020 [cit. 2021-03-12]. Dostupné z: https://microservices.io/patterns/client-side-discovery.html.

25. RICHARDSON, Chris. Pattern: Server-side service discovery [online]. 2020 [cit. 2021-03-12]. Dostupné z: https://microservices.io/patterns/server-side-discovery.html.

26. 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/.

27. RICHARDSON, Chris. Pattern: Remote Procedure Invocation (RPI) [online]. 2020 [cit. 2021-03-12]. Dostupné z: https://microservices.io/patterns/communication-style/rpi.html.

28. RICHARDSON, Chris. Pattern: Messaging [online]. 2020 [cit. 2021-03-12]. Dostupné z: https://microservices.io/patterns/communication-style/messaging.html.

29. RICHARDSON, Chris. Pattern: Single Service Instance per Host [online]. 2020 [cit. 2021-03-12]. Dostupné z: https://microservices.io/patterns/deployment/single-service-per-host.html.

30. RICHARDSON, Chris. Pattern: Multiple service instances per host [online]. 2020 [cit. 2021-03-12]. Dostupné z: https://microservices.io/patterns/deployment/multiple-services-per-host.html.

31. RICHARDSON, Chris. Pattern: Health Check API [online]. 2020 [cit. 2021-03-12]. Dostupné z: https://microservices.io/patterns/observability/health-check-api.html.

32. RICHARDSON, Chris. Pattern: Distributed tracing [online]. 2020 [cit. 2021-03-12]. Dostupné z: https://microservices.io/patterns/observability/distributed-tracing.html.

33. RICHARDSON, Chris. Pattern: Audit logging [online]. 2020 [cit. 2021-03-12]. Dostupné z: https://microservices.io/patterns/observability/audit-logging.html.

34. RICHARDSON, Chris. Pattern: Log aggregation [online]. 2020 [cit. 2021-03-12]. Dostupné z: https://microservices.io/patterns/observability/application-logging.html.

35. RICHARDSON, Chris. Pattern: Circuit Breaker [online]. 2020 [cit. 2021-03-12]. Dostupné z: https://microservices.io/patterns/reliability/circuit-breaker.html.

36. MISHRA, Deepanshu. Cross-cutting Concern [online]. 2019 [cit. 2021-03-12]. Dostupné z: https://medium.com/anatta-design/cross-cutting-concern-aadf4f51a5c1.

37. RICHARDSON, Chris. Pattern: Microservice chassis [online]. 2020 [cit. 2021-03-12]. Dostupné z: https://microservices.io/patterns/microservice-chassis.html.

38. FOWLER, Martin. MonolithFirst [online]. 2015 [cit. 2021-03-12]. Dostupné z: https://martinfowler.com/bliki/MonolithFirst.html.

39. DEHGHANI, Zhamak. How to break a Monolith into Microservices [online]. 2018 [cit. 2021-03-12]. Dostupné z: https://martinfowler.com/articles/break-monolith-into-microservices.html.

40. TODKAR, Praful. How to extract a data-rich service from a monolith [online]. 2018 [cit. 2021-03-12]. Dostupné z: https://martinfowler.com/articles/extract-data-rich-service.html#Step1.IdentifyLogicAndDataRelatedToTheNewService.

41. GOOGLE DEVELOPERS. *Set up a Firebase Cloud Messaging client app on iOS* [online]. 2021 [cit. 2021-04-14]. Dostupné z: https://firebase.google.com/docs/cloud-messaging/ios/client.

42. GOOGLE DEVELOPERS. *Set up a Firebase Cloud Messaging client app on Android* [online]. 2021 [cit. 2021-04-14]. Dostupné z: https://firebase.google.com/docs/cloud-messaging/android/client.

43. OPENAPI-GENERATOR CONTRIBUTORS. *OpenAPI Generator* [online]. 2021 [cit. 2021-04-18]. Dostupné z: https://github.com/OpenAPITools/openapi-generator.

44. DATA GEEKERY™ GMBH. *Great Reasons for Using jOOQ* [online]. 2021 [cit. 2021-04-18]. Dostupné z: https://www.jooq.org/.

45. GOOGLE DEVELOPERS. *About FCM messages* [online]. 2021 [cit. 2021-04-14]. Dostupné z: https://firebase.google.com/docs/cloud-messaging/concept-options#ttl.

46. WANSTRATH, Chris. *mustache - Mustache processor* [online]. 2014 [cit. 2021-04-14]. Dostupné z: https://mustache.github.io/mustache.1.html.

47. RED HAT, INC. What is Apicurito? [online]. 2021 [cit. 2021-03-12]. Dostupné z: https://www.apicur.io/apicurito/.

Příloha A

Seznam použitých zkratek

API Application Programming Interface

AWS Amazon Web Services

CQRS Command-Query Responsibility Segregation

DAO Data Access Object

DDL Data Definition Language

DNS Domain Name System

DSL Domain Specific Language

DTO Data Transfer Object

FCM Firebase Cloud Messaging

GP Global Payments

HTML HyperText Markup Language

HTTP HyperText Transfer Protocol

IDE Integrated Development Environment

JAR Java ARchive

JOOQ jOOQ Object Oriented Querying

JSON JavaScript Object Notation

JVM Java Virtual Machine

JWT JSON Web Token

POJO Plain Old Java Object

OAS3 OpenAPI Specification version 3

ORM Object Relational Mapping

REST Representational State Transfer

RPC Remote Procedure Call

SBT Simple Build Tool

SMS Short Message Service

SMTP Simple Mail Transfer Protocol

SOAP Simple Object Access Protocol

SQL Structured Query Language

TZ Time Zone

URL Uniform Resource Locator

VPC Virtual Private Cloud

WSDL Web Services Description Language

XML eXtensible Markup Language

Příloha **B**

Obsah přiloženého CD

- `readme.txt` ..... stručný popis obsahu přiloženého média
- `src`
  - `impl` ..... zdrojové kódy implementace a automatických testů
  - `thesis` ..... zdrojová forma práce ve formátu LaTeX
  - `documentation` ..... externí dokumentace vzniklého řešení
- `text` ..... text práce
  - `DP_Petr_Prouza_2021.pdf` ..... text práce ve formátu PDF
  - `TZP_Petr_Prouza.pdf` ..... zadání práce ve formátu PDF