VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
FACULTY OF INFORMATION TECHNOLOGY

ÚSTAV INFORMAČNÍCH SYSTÉMŮ
DEPARTMENT OF INFORMATION SYSTEMS

AUTOMATIZACE VÝDEJE A ÚČTOVÁNÍ KÁVY

AUTOMATION AND ACCOUNTING OF COFFEE MACHINE

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE — **JAKUB VÍŠEK**
AUTHOR

VEDOUCÍ PRÁCE — **Ing. JAN PLUSKAL**
SUPERVISOR

BRNO 2019

Vysoké učení technické v Brně
Fakulta informačních technologií

Ústav informačních systémů (UIFS) Akademický rok 2018/2019

Zadání bakalářské práce

Student: **Víšek Jakub**
Program: Informační technologie
Název: **Automatizace výdeje a účtování kávy**
Automation and Accounting of Coffee Machine
Kategorie: Počítačové sítě

Zadání:

1. Seznamte se teoreticky s řešeními pro automatizovaný výdej kávy a účtování. Vytvořte seznam požadavků, dle instrukcí vedoucího, pro implementaci obdobného řešení v domácích podmínkách. Řešení musí obsahovat minimálně správu uživatelů, evidenci skladové kávy a účtování spotřebovaných káv.
2. Navrhněte řešení adresující výstupy z bodu 1. Vyberte vhodné technologie pro implementaci. Popište architekturu navrženého řešení a zvolené IoT komunikační protokoly.
3. Implementujte řešení jako "proof of concept" pro různé typy kávovarů. Řešení bude obsahovat HW pro získání identity uživatele; webovou aplikaci pro správu; REST API jako zdroj dat pro webovou aplikaci tak i pro integraci s aplikacemi třetích stran. SW část řešení bude kontejnerizována a nasaditelná pomocí Dockeru.
4. Proveďte testování a zhodnocení výsledků. Zaměřte se na robustnost systému vůči výpadkům a navrhněte patřičná opatření.

Literatura:

1. Rahman, L. F., Ozcelebi, T., & Lukkien, J. J. (2016, August). Choosing your IoT programming framework: Architectural aspects. In *2016 IEEE 4th International Conference on Future Internet of Things and Cloud (FiCloud)* (pp. 293-300). IEEE.
2. Collina, M., Corazza, G. E., & Vanelli-Coralli, A. (2012, September). Introducing the QEST broker: Scaling the IoT by bridging MQTT and REST. In *Personal indoor and mobile radio communications (pimrc), 2012 ieee 23rd international symposium on* (pp. 36-41). IEEE.
3. Gubbi, J., Buyya, R., Marusic, S., & Palaniswami, M. (2013). Internet of Things (IoT): A vision, architectural elements, and future directions. *Future generation computer systems*, *29*(7), 1645-1660.

Pro udělení zápočtu za první semestr je požadováno:

- Body 1 a 2.

Podrobné závazné pokyny pro vypracování práce viz http://www.fit.vutbr.cz/info/szz/

Vedoucí práce: **Pluskal Jan, Ing.**
Vedoucí ústavu: Kolář Dušan, doc. Dr. Ing.
Datum zadání: 1. listopadu 2018
Datum odevzdání: 15. května 2019
Datum schválení: 4. dubna 2019

Abstrakt

Cílem této práce je návrh a implementace vestavěného a informačního systému pro automatické účtování kávy vydané nápojovým automatem. Řešení je vytvořeno s ohledem na spolehlivost a přenositelnost. Systém může být snadno rozšířen pro další typy prodejních míst než jen kávovary. Výsledky této práce umožňují jednoduché nasazení elektronického účtování zakoupených produktů a demonstrují i zapojení mikropočítače, který ovládá kávovar na základě identifikace uživatelů podle jejich RFID čipů.

Abstract

The goal of this thesis is to design and implement an embedded and information system for automatic accounting of coffee dispensed by a vending machine. Features of the created solution include reliability and portability. The system can be easily extended for use in point of sale applications different to coffee machines. The results enable easy deployment of electronic accounting of purchased products and demonstrate integration of a microcontroller which controls a coffee vending machine on the basis of identifying users by their RFID tags.

Klíčová slova

mikroslužby, Docker, REST API, Raspberry Pi, ESP8266, internet věcí, RFID, automat na kávu, ASP.NET Core, MassTransit

Keywords

microservices, Docker, REST API, Raspberry Pi, ESP8266, Internet of Things, RFID, coffee machine, ASP.NET Core, MassTransit

Citace

VÍŠEK, Jakub. *Automatizace výdeje a účtování kávy.* Brno, 2019. Bakalářská práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedoucí práce Ing. Jan Pluskal

Automatizace výdeje a účtování kávy

Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně pod vedením Ing. Pluskala. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

.....
Jakub Víšek
13. května 2019

Poděkování

Rád bych poděkoval Ing. Janu Pluskalovi, vedoucímu této práce, za jeho vstřícnost, čas a odborné rady, které mi poskytl v rámci zprostředkovaných konzultací. Jeho aktivní zájem o obsah práce a o můj postup při jejím vypracování mi byl velkou motivací.

Obsah

1 Úvod **2**

2 Požadavky na vytvářený systém **3**
2.1 Analýza existujících řešení 3
2.2 Uvažované scénáře použití 6
2.3 Formální specifikace systémových požadavků 6

3 Návrh systému **11**
3.1 Ovlivňující faktory 11
3.2 Návrhové vzory pro koordinaci distribuovaných transakcí 16
3.3 Architektonické celky 17
3.4 Implementační nástroje a technologie 21

4 Implementace systému **25**
4.1 Jádro systému zprostředkovávající REST API 25
4.2 Webová aplikace pro správu 38
4.3 Integrace hardware pro autentizaci uživatelů 39

5 Uvedení do provozu a testování **40**
5.1 Nasazení s využitím kontejnerizace a možnosti škálování 40
5.2 Jednotkové testy 40
5.3 Testy REST API 41
5.4 Opatření pro odolnost vůči výpadkům 42
5.5 Kontrola splnění systémových požadavků 42

6 Závěr **44**

Zkratky **46**

Slovník pojmů **48**

Literatura **50**

A Entity-Relationship diagram **54**

B Analýza konzistence dat **55**

C Klasifikace IoT prvků systému **56**

Kapitola 1

Úvod

Konzumace kávy jako povzbuzovadla je celosvětový společenský fenomén, ke kterému se pojí množství vlastních kulturních zvyklostí[1]. Kofein obsažený v nápoji ve vhodných dávkách potlačuje únavu a zvyšuje pozornost[2], díky čemuž se těší oblíbenosti u široké veřejnosti i u profesí vyžadujících dlouhodobou soustředěnost. Pro mnohé kolektivy jsou přestávky na kávu pravidelné rituály[3] skýtající taktéž příležitost pro společenskou interakci.

Uživatelé kávy mají obecně několik možností, jak si nápoj obstarat. Mohou navštívit kavárnu, což ale může, vzhledem k době trvání obsluhy a obousměrné cesty, zabrat příliš mnoho času. Další možností je vlastní příprava, kde může být pro dosažení dostatečně kvalitního výsledku nutný nákup vybavení nebo získání souvisejících dovedností. Častým kompromisem proto bývají nápojové automaty, které lze provozovat v umístění blízkém zákazníkům a které dokáží produkovat kvalitnější kávu, než by průměrný uživatel zvládl připravit ručně, a navíc to zvládají rychleji.

Pro firemní zákazníky jsou k dispozici komerční řešení kávovarů[4], která zahrnují jejich kompletní servis, údržbu i účtování. Pro menší firmy a pro domácnosti nejsou tato řešení dostupná, ať už kvůli s tím spojeným finančním nákladům nebo kvůli rozhodnutí dodavatele. Uživatelé jsou tak například v rámci zmíněné přestávky na kávu zatíženi nepříjemnými nebo zbytečnými úkony, které je možné automatizovat nebo nahradit, např. vhazováním mincí k platbě, zapisováním útraty do sešitu.

Cílem této práce je vytvořit řešení, které umožní snadné zavedení elektronického účtování a automatizovaného výdeje kávy v malé kanceláři i v domácím prostředí, jako např. v bytě sdíleném studenty. Řešení zpříjemní a zrychlí uživatelům interakci s kávovarem a zároveň poskytne provozovateli zařízení nástroj pro správu a účtování. To bylo i mou motivací k volbě tohoto tématu bakalářské práce, protože se spolubydlícími nejsme spokojeni s našim stávajícím nedigitálním řešením.

Následující text sleduje můj postup při realizaci výše zmíněného řešení. V kapitole 2 se věnuji průzkumu existujících řešení a definici požadavků. Na jejich základě je následně ve 3. kapitole navržena architektura systému obsahující jádro na bázi *microservices*, identifikována některá ze souvisejících rizik a popsán způsob jejich řešení. Zbývající rizika a detaily implementace, která proběhla převážně v jazyce C#, popisuje kapitola 4. V poslední kapitole je pak popsáno provedené testování.

[1] https://en.wikipedia.org/wiki/Coffee#Society_and_culture
[2] https://en.wikipedia.org/wiki/Caffeine
[3] https://en.wikipedia.org/wiki/Coffee#Break
[4] viz Analýza existujících řešení, str. 3

Kapitola 2

Požadavky na vytvářený systém

Pro návrh každého systému je klíčová důsledná analýza požadavků, jež je nutné splnit. Výsledek této práce je plánováno nasadit do provozu ve vybraných pracovnách areálu FIT VUT v Brně. Pro analýzu byly proto směrodatné potřeby tamních uživatelů a provozovatelů zjištěné v průběhu osobních konzultací. Některé požadavky byly zobecněny, aby neztěžovaly případné nasazení systému v odlišných prostředích a aby bylo systém možné snadno upravit v reakci na případnou změnu požadavků.

2.1 Analýza existujících řešení

Kromě požadavků klienta je však pro analýzu cílových požadavků prospěšný i průzkum již dostupných produktů. Skutečnosti, které jejich použití vylučují a které tak podmiňují vznik produktu nového, poskytují cenné informace o poptávaných klíčových vlastnostech. Existující produkty mohou navíc poskytovat funkce, na které zadavatel nepomyslel ale které by přesto uvítal.

Automatický výdejník na kapsle

Řešení dostupné v rámci programu *Nespresso Professional Payment Solutions* podporuje široké množství platebních kanálů – mince, platební karty, RFID čipy a platbu přes PayPal prostřednictvím mobilní aplikace [13]. V té mohou uživatelé rovněž sledovat svůj zůstatek. Provozovateli zařízení jsou přístupné statistiky o prodejích a o stavu zásob. Řešení je však vzhledem k omezení na proprietární typ vydávaných kapslí vhodné výhradně pro použití s kávovary *Nespresso*. Není ani vhodné pro použití v domácnosti, kam je výsledek této práce zamýšleno nasazovat, vzhledem k nezpůsobilosti pro účast v programu *Nespresso Professional*, v rámci kterého je výdejník nabízen pouze firemním zákazníkům.

mocca.vend

Zařízení *mocca.vend* umožňuje zpracování bezhotovostních plateb a jejich účtování [38]. Platby lze provádět různými typy RFID karet. Podporován je jak režim online, kdy zařízení platbu na počkání ověřuje u centrálního serveru, tak v režimu offline, kdy jsou provedené platby ukládány lokálně a synchronizovány po obnovení spojení, případně pověřenou osobou jednou za čas exportovány na paměťový klíč USB. S prodejním automatem zařízení komunikuje protokolem MDB/ICP navrženým pro komunikaci prodejního automatu s jeho periferiemi. Protože však kávovary určené pro použití v domácnostech typicky nezahrnují

funkcionalitu prodejního automatu, je nepravděpodobné, že by tento typ komunikace podporovaly a mohly tak využít tohoto řešení.

Místa prodeje provozované tímto řešením je možné spravovat v aplikaci *mocca.admin*, která poskytuje přístup i ke statistickým datům [37]. K dispozici je také mobilní aplikace *mocca.app*, ve které mohou uživatelé sledovat a navyšovat svůj zůstatek [36]. Mobilní zařízení s podporou technologie NFC je pomocí *mocca.app* možné použít namísto dedikovaného RFID čipu. Obě zmíněné aplikace jsou dodavatelem uváděny jako samostatné produkty a je tudíž možné, že jsou i samostatně účtovány.

Internetový obchod s RFID autentizací

Akbarov ve své diplomové práci popisuje požadavky, případy užití a vývoj systému podobného tomu, jež byl zadán této práci [1]. Uživatelé se v něm mohou autentizovat přiložením RFID čipu a za svůj systémem vedený zůstatek mohou nakupovat položky ve webovém uživatelském rozhraní fungujícím podobně jako internetový obchod. To zajišťuje komunikaci se serverovou částí aplikace pomocí jejího REST API a je provozováno lokálně na počítači Raspberry Pi verze 2B nebo 3, fungujícím jako řídící prvek každého prodejního místa. Jeho nedílnou součásti je kromě již zmíněné RFID čtečky také dotyková obrazovka, která webové rozhraní zpřístupňuje uživateli.

Systém však neuvažuje scénář integrovaného výdejníku zakoupených produktů ani evidenci skladových zásob[1]. Přímo není poskytována ani možnost kontejnerizovaného nasazení aplikace v Dockeru[2], ačkoliv by splnění tohoto požadavku bylo vzhledem k použitým technologiím možné.

Ovládání kávovaru mikrokontrolérem

V rámci školního projektu [9] byl vytvořen program pro obsluhu mikrokontroléru ESP32, který na něm implementuje komunikační role *publish* a *subscribe* protokolu MQTT. V první z nich informuje o změnách stavu připojeného výdejního zařízení, konkrétně kávovaru, nebo o přiložení RFID čipu. Druhá z implementovaných rolí umožňuje mikrokontroléru příjem řídích zpráv jako například „vydej nápoj“, v závislosti na kterých prostřednictvím reléového modulu napojeného na své výstupní porty ovládá řídící logiku samostatného kávovaru.

Srovnání existujících řešení

Automatický výdejník na kapsle představuje jediné řešení, které podle dostupných informací splňuje funkční požadavky popsané v této práci. Není však dostupné požadované cílové skupině a zavádí proprietární uzamčení na konkrétní typ kávových kapslí.

Zbývající řešení splňují funkční požadavky vytvářeného systému jen z části. Mocca.vend oproti ostatním podporuje komunikaci s kávovarem podporujícím standardizovaný protokol a umožňuje offline zpracování transakcí. Internetový obchod s RFID autentizací se odlišuje uživatelským rozhraním na dotykové obrazovce. Ovládání kávovaru mikrokontrolérem neobsahuje žádnou funkcionalitu informačního systému, jedná se však o potenciálně znovupoužitelný program pro mikrokontrolér místa prodeje[3].

[1] viz Funkční požadavky, strana 9, bod 10
[2] viz Požadavky na provoz systému, strana 10
[3] viz Funkční požadavky, strana 7, bod 2

Funkcionalita	Automatický výdejník na kapsle	mocca.vend	Internetový obchod s RFID autentizací	Ovládání kávovaru mikrokontrolérem
Účtování plateb a prodejů	✓	✓	✓	×
Skladová evidence	✓	✓[a]	×	×
Automatizace výdeje	∼[b]	∼[c]	×	✓
Mobilní aplikace	✓	✓[a]	✓[d]	×
Analytické nástroje	✓	✓[a]	×	×
Režim offline	×	✓	×	×

[a]Výrobce nabízí funkcionalitu jako samostatný produkt, který může být prodáván zvlášť.
[b]Zařízení umožňuje automatizovaný výdej kávových kapslí. Vydanou kapsli musí uživatel ručně přesunout do kávovaru a až poté je možné zahájit přípravu nápoje.
[c]Zařízení podporuje automatizaci výdeje produktů z prodejních automatů podporujících komunikaci protokolem Multi-Drop Bus/Internal Communication Protocol.
[d]Uživatelské rozhraní je řešeno formou webové stránky, kterou by bylo možné používat i v mobilním prohlížeči.

Tabulka 2.1: Srovnání funkcí podporovaných zjištěnými existujícími řešeními.

Platební metoda	Automatický výdejník na kapsle	mocca.vend	Internetový obchod s RFID autentizací	Ovládání kávovaru mikrokontrolérem
Virtuální konto[a]	✓	✓	✓	×
Mince	✓	×	×	×
Bankovky	×	✓[b]	×	×
Platební karta	✓	✓[b]	×	×
Mobilní aplikace	✓	∼[c]	×	×
PayPal	✓	×	×	×

[a]Uživatel má v systému veden virtuální peněžní zůstatek, který může využívat k nákupu produktů.
[b]Výrobce nabízí platební metodu jako samostatný produkt, který může být prodáván zvlášť.
[c]Prostřednictvím mobilní aplikace lze platit pouze ze zařízení podporujících technologii NFC.

Tabulka 2.2: Srovnání platebních metod podporovaných zjištěnými existujícími řešeními.

2.2 Uvažované scénáře použití

Na základě funkcí nabízených zkoumanými existujícími řešeními lze vyvodit několik níže popsaných scénářů, které bude vytvářený systém podporovat nebo s jejichž podporou bude při jeho návrhu uvažováno.

Integrovaný kávovar s periodicky splácenou útratou

Všichni uživatelé s RFID kartou registrovanou v systému mohou jejím přiložením zahájit přípravu nápoje. Mikrokontrolér ve spolupráci se serverovou částí aplikace uživatele autentizuje, v případě úspěchu následuje řízení integrovaného kávovaru. V případě detekované poruchy je nákup zrušen. Uživatel není povinen udržovat na svém účtu kladný zůstatek v dostatečné výši, svou zaznamenanou útratu pravidelně splácí provozovateli kávovaru v předem dohodnutých termínech. Místo prodeje ke své činnosti vyloženě nevyžaduje žádné uživatelské rozhraní s výjimkou RFID čtečky, ačkoliv by bylo vhodné pro zobrazování např. aktuálního uživatelského zůstatku nebo zpětné vazby v případě odmítnutí karty či chybového stavu.

Elektronické účtování neintegrovaného kávovaru

Provozovatel kávovaru není schopen zajistit integraci jeho řídící logiky s mikrokontrolérem, ať už z technických (nedostatečná kvalifikace pro zásah do elektrického zařízení) nebo jiných důvodů (kávovar je v pronájmu nebo záruční době). Stávající papírovou evidenci útraty ale po domluvě s uživateli nahrazuje elektronickou. Uživatelé se opět autentizují RFID kartou, v případě úspěchu mohou potvrdit žádost o zapsání útraty, kterou mohou sledovat v k tomu určené webové aplikaci.

Kávovar s kreditovým modelem a komunikací po sběrnici

Kávovar není možné integrovat zásahem do elektronických součástí, lze s ním však komunikovat po standardizované sběrnici. Provozovatel si navíc místo prodeje na dluh přeje, aby mohli uživatelé nakupovat výhradně za předem poskytnuté finanční prostředky. Uživatelům je poskytováno množství platebních kanálů, které do systému poskytovatel integruje nabízeným API.

Automaty se společným účtováním

Systém využívají zaměstnanci několika kanceláří, každá se svým vlastním kávovarem. Několik z nich je však pronajato od stejného externího dodavatele, který zajišťuje jejich servis a pravidelné doplňování. Dodavatel umožní integraci kávovarů do systému, na oplátku si ale žádá přístup k nastavení a agregovaným statistikám míst prodeje, které spadají do jeho kompetence.

2.3 Formální specifikace systémových požadavků

S ohledem na vlastnosti existujících řešení popsaných v kapitole 2.1 a na výčet podporovaných scénářů použití systému z kapitoly 2.2 definuje tato kapitola formální specifikaci požadavků cílového systému.

Funkční požadavky

Jedním z typů systémových požadavků jsou požadavky funkční [12], které definují chování systému v závislosti na vstupech potřebné k poskytování očekávané služby. Výčet entit vedených v systému a vztahů mezi nimi je k dispozici v ER diagramu v příloze A. V této sekci je k jednotlivým prvkům diagramu nabídnut stručný popis a jsou zde rovněž vysvětleny potenciální nejasnosti.

1. Uživatelské účty

   Každému uživateli je v systému vedeno jeho celé jméno a kontaktní e-mailová adresa. Uživatelé s nastaveným příznakem správce mají v systému neomezená přístupová oprávnění. Ztrátou příznaku aktivity lze účet zablokovat a zabránit tak jeho použití při autentizaci.

2. Místa prodeje

   Zařízení sloužící jako koncové body systému, která zajišťují výměnu informací mezi systémem a uživateli za účelem zprostředkování prodeje, jsou modelovány jako místa prodeje. Ke každému z nich jsou vedeny následující informace:

   - název, určený např. k zobrazování v uživatelském výpisu transakcí;
   - nadřazená účetní skupina, do které se mají řadit nově zprostředkované prodeje;
   - seznam nabízených produktů;
   - seznam oprávněných uživatelů, z nichž alespoň jeden je v roli správce a má tak k místu prodeje neomezená oprávnění;
   - seznam položek typu klíč-hodnota, do nichž místo prodeje může ukládat informace o stavu;
   - nepovinně také uplatňovaná prodejní strategie, jinak zděděná z nadřazené účetní skupiny.

3. Autentizační prostředky

   V systému je rozlišováno několik účelů autentizace popsaných v tabulce 2.3, z nichž každý může vyžadovat odlišný druh autentizačního prostředku. Požadavky jednotlivých účelů autentizace byly zohledněny následovně:

   - Ke každému prostředku je vedena použitá autentizační metoda. Hodnotou prostředku se pak rozumí informace, na kterou použitá metoda při autentizaci spoléhá: např. data veřejného klíče v případě asymetrické kryptografie nebo hodnota hašovací funkce v případě hesla.
   - Ke každému prostředku může být vedeno také datum a čas zahájení a konce platnosti, což může např. v případě digitálních certifikátů umožnit plynulé nasazení nových dříve než/poté co starým vyprší doba platnosti.

   Autentizační prostředky je možné přiřadit požadovaným subjektům autentizace — uživatelským účtům a místům prodeje. Přiřazení může být časově omezeno, což je vhodné např. pro RFID čipy propůjčované dočasným uživatelům nebo návštěvám. Pro sdílené autentizační prostředky mohou být užitečné také následující funkcionality reflektované v entitě „UživateliPAP“:

Účel	Popis	Možné prostředky
přihlášení uživatele	Zabránit neoprávněnému přístupu k informacím a manipulaci s nimi, např. k výpisu nákupů ve webovém portálu nebo k nastavení účtu.	heslo, One-Time Password[a], Kerberos[b]
nákup produktu	Vyhledat uživatelský účet, jehož zůstatek má být použit pro úhradu kupní ceny.	sériové číslo RFID čipu, biometrické údaje, příp. i prostředky použitelné k přihlášení uživatele
ověření místa prodeje	Ověřit, že příchozí komunikace skutečně pochází od známého místa prodeje a nikoliv od např. útočníka, pokoušejícího se o neoprávněný vstup do systému.	sdílený tajný klíč, veřejný klíč v případě komunikace využívající asymetrické kryptografie

[a] https://en.wikipedia.org/w/index.php?title=One-time_password&oldid=876139242
[b] https://en.wikipedia.org/w/index.php?title=Kerberos_(protocol)&oldid=876187778

Tabulka 2.3: Účely autentizace uvažované ve vytvářeném systému.

- Možnost označit přiřazení prostředku uživateli jako sdílené, což zabrání jeho použití k jiným účelům, než k autentizaci za účelem zprostředkování prodeje produktu.
- Možnost nastavit k přiřazení jeden nebo více stropů útraty, což zabrání jeho použití k autentizaci za účelem zprostředkování prodeje produktu v případech, kdy by tím došlo k překročení maximální povolené částky za stanovený časový interval.

Každý subjekt autentizace může mít v jednu chvíli přiřazených více platných autentizačních prostředků. Autentizační prostředek však nesmí být současně přiřazen více jak jednomu subjektu.

4. Účetní skupiny

   Účtování prodaných produktů neprobíhá v systému na úrovni jednotlivých míst prodeje, nýbrž na úrovni účetních skupin, do kterých jsou místa prodeje zařazena. V případech, kdy je více míst prodeje provozováno jednou osobou, bude vhodné umožnit jejich agregaci pro účely řízení a přehledu statistik.

   Ke každé účetní skupině jsou podobně jako k místu prodeje vedeny následující údaje:

   - název určený k zobrazování uživatelům;
   - seznam oprávněných uživatelů, z nichž alespoň jeden je v roli vlastníka a má tak k účetní skupině neomezená oprávnění;
   - prodejní strategie uplatňovaná na podřazených místech prodeje, která nemají explicitně nastavenou vlastní.

5. Prodejní strategie

   Systém podporuje alespoň obě prodejní strategie zmíněné v sekci 2.2:

- „kredit“ – uživatelům nebude umožněno provedení nákupu, pokud jejich zůstatek není větší nebo roven kupní ceně produktu;
- „dluh“ – uživatelé mohou provádět nákupy bez ohledu na výši jejich zůstatku.

6. Produkty

   Systém umožňuje vedení produktů definovaných názvem a doporučenou cenou.

7. Nabídky

   Produkty je v místech prodeje možné nabízet ke koupi. K nabídce je možné specifikovat kupní cenu, není-li žádoucí převzít doporučenou cenu nabízeného produktu. Nabídka může být svázána se skladovou položkou a povolit tím vytváření negativních pohybů na skladě s každým potvrzeným prodejem.

8. Prodeje

   O každém prodeji jsou v systému vedeny následující informace:

   - datum a čas vytvoření;
   - prodané množství a zaúčtovaná cena;
   - účetní skupina, do které byl prodej zaúčtován;
   - uživatel, kterému byl prodej zaúčtován;
   - použitý autentizační prostředek;
   - místo prodeje, které prodej vytvořilo.

9. Stav prodeje

   Ke každému prodeji jsou vedeny informace o změnách jeho stavu složené z následujících informací:

   - data a času změny;
   - příčiny změny;
   - nového stavu prodeje;
   - místa prodeje v případě, že změnu vyvolalo;
   - uživatelského účtu v případě, že změnu vyvolal.

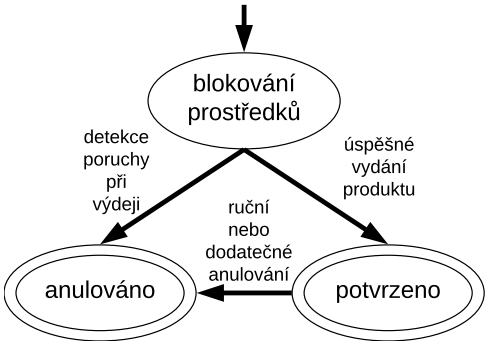
   Možné stavy prodeje a přechody mezi nimi jsou popsány v obrázku 2.1. V kontextu vlastnosti prodeje je „stavem prodeje“ myšlena stavová hodnota z jeho nejnovější změny stavu.

   Systém podporuje nastavení maximální doby, po kterou může prodej zůstat v počátečním stavu. Po jejím uplynutí je takový prodej automaticky anulován.

10. Skladová evidence

    V systému je možné vést skladové položky definované názvem a záznamy o jejich pohybech, které mohou být jednoho ze dvou typů:

    - ručního — vytvořeného uživatelem s ručně zadaným množstvím, např. po ručním naskladnění skladové položky;



Obrázek 2.1: Diagram stavů prodeje a povolených přechodů mezi nimi. Z původního stavu „blokování prostředků“ je prodej změněn do stavu „potvrzeno“ v případě úspěchu nebo „anulováno“ v případě detekce poruchy. Potvrzení nákupu typicky iniciuje místo prodeje, které prodej vytvořilo. Změnu do stavu „anulováno“ může rovněž provést oprávněný uživatel nebo systém samotný v případech, kdy prodej setrvá ve stavu „blokování prostředků“ déle, než povolil správce systému.

- vázaného na prodej — vytvořeného automaticky při změně stavu prodeje na „potvrzeno“ v případě, že byla správcem místa prodeje vytvořena vazba mezi nabídkou a skladovou položkou.

V případě anulování již potvrzeného prodeje systém automaticky odstraní odpovídající pohyb na skladě.

11. Webové uživatelské rozhraní

    Součástí systému je webové rozhraní pro registrované uživatele, které jim umožňuje provádět výše specifikované činnosti příslušné jejich přiřazenému oprávnění.

Požadavky na provoz systému

Splnění níže vyjmenovaných požadavků vycházejících ze zadání práce a z uskutečněných konzultací umožní škálování jednotlivých součástí aplikace dle uznání provozovatele systému. Aplikaci bude totiž jejich důsledkem možné v celku provozovat lokálně, nebo ji z části nebo z celku provozovat v cloudu.

1. Serverová část aplikace a webový portál musí podporovat nasazení v Docker kontejnerech.
2. Aplikace nesmí obsahovat závislost na hostitelském operačním systému, aby bylo její Docker kontejnery možné spustit na Windows 10 i na operačních systémech rodiny Linux.
3. Aplikace nesmí obsahovat závislost na platformě procesoru, aby bylo kontejnery možné spustit jak na platformě x86, tak x86–64 nebo ARM.
4. V důsledku nedostatku výpočetního času nebo přechodné nedostupnosti způsobené alokací virtuálních zdrojů mohou být jednotlivé části aplikace dočasně nedostupné. Aplikace musí tuto skutečnost respektovat a vhodným způsobem reagovat na její výskyt.

Kapitola 3

Návrh systému

Systém specifikovaný v předchozí kapitole je dále potřeba vhodným způsobem dekomponovat na prvky a tyto prvky dále navrhnout. Způsob dekompozice je však do značné míry závislý na množství faktorů, které je z tohoto důvodu nejdříve nutné uvést a popsat.

3.1 Ovlivňující faktory

Jedním z faktorů ovlivňujících návrh vytvářeného systému je požadavek zadání této práce na využití IoT komunikačních protokolů, potažmo samotný koncept IoT. Za ním v této podkapitole následují další faktory, které z něj vychází.

Internet věcí a jeho vztah k navrhovanému systému

Internet věcí je evolucí stávajícího statického internetu do sítě vzájemně propojených „věcí“, jejichž hlavním cílem je umožnit počítačům příjem informací bez lidské obsluhy [8]. Věci za tímto účelem snímají okolní prostředí a interagují s ním, přičemž k tomu využívají standardizované technologie a prostředky, které jsou v internetu již běžně používány. Propojení věcí prostřednictvím sítí na bázi IP je umožňuje řídit a využívat z existující internetové infrastruktury [27].

Pro věci jsou charakteristická hardwarová i softwarová omezení, často až na minimální úroveň, při které je zachována schopnost vykonávat žádanou funkci. Tím způsobený nízký příkon umožňuje *věci* dlouhodobě napájet z baterií nebo z energie v okolí (tzv. *Energy Harvesting*[1]). V režimu nízkého příkonu však zařízení často nemohou využívat některé technologie, jako např. TCP/IP [3], a je tak nutné hledat alternativní řešení. Prvky vytvářeného systému budou omezeny dostupným výpočetním výkonem, předpokládá se však jejich napájení ze sítě, neboť obsluha uživatelů musí probíhat nepřetržitě a v reálném čase.

Architektura microservices

V souladu s definicí IoT uvedenou v předchozí sekci bude vytvářený systém alespoň částečně distribuovaný mezi několik zařízení komunikujících po IP síti. Ve scénáři, kdy jsou do systému zapojeny nejvýše desítky *věcí*, není pravděpodobné, že by pro dosažení přijatelné rychlosti zpětné vazby bylo nutné provádět zvláštní opatření. Připojení prvku do již nasazeného systému ovšem obecně nevyžaduje tolik znalostí, jako jeho nasazení a údržba.

[1] Sběr energie dostupné v okolním prostředí, např. v radiovém nebo infračerveném vlnění.

Nabízí se proto další scénář, ve kterém je jedna instance systému sdílena množstvím různých domácností a kanceláří a ve kterém se proto množství připojených zařízení pohybuje řádově výše, např. ve stovkách nebo tisících. U takového systému lze předpokládat potřebu škálování a ideálně pro něj zvolit takovou architekturu, která škálování vhodně podporuje.

Návrh systému s architekturou *microservices* představuje přístup, kdy je systém vyvinut jako sada malých služeb běžících jako samostatné procesy komunikující s okolím jednoduchým protokolem, jako např. HTTP [7]. Mezi další charakteristiky této architektury patří [35]:

- Každá služba implementuje konkrétní funkcionalitu nebo část problémové domény v dostatečně omezené míře.
- V systému neexistuje centrální datové úložiště, celkový datový model je decentralizován mezi jednotlivé služby. Datový model každé služby přitom vlastní výhradně služba sama, přístup k němu je k dispozici pouze přes její komunikační rozhraní.

Dodržením tohoto přístupu je možné získat následující výhody:

- Datovou suverenitu — každá služba může svá data (vyžadují-li tato data perzistenci) ukládat v SŘBD dle vlastního výběru, nezávisle na zbytku systému.
- Autonomnost — každá služba může být vyvinuta v libovolném programovacím jazyce, být nasazena na libovolné platformě a být škálována samostatně dle potřeby. Jediným předpokladem pro zapojení do systému je podpora definovaného komunikačního rozhraní.
- Izolovanost — změny chování a datových modelů se provádí na úrovni individuálních služeb, které je implementují/vlastní, a není nutné zasahovat do zbytku systému. To platí za předpokladu, že byla službě při návrhu přidělena odpovědnost za dostatečně omezenou část problémové domény.
- Modulárnost — díky volné vazbě na zbytek systému je správně navrženou službu možné nasadit i samostatně nebo ji integrovat v rámci jiného systému, než pro který byla původně vyvinuta.

Komunikace mezi službami

Služby v kontextu architektury *microservices* po přijetí požadavků provádí jejich zpracování a dále mohou odesilateli odpovědět. Způsob odpovědi lze podle časového průběhu klasifikovat do dvou typů [35]:

1. Synchronní způsob odpovědi — sezení s odesilatelem požadavku je udržováno otevřené, než je zpracování požadavku dokončeno. Po odeslání odpovědi může být sezení ukončeno.
2. Asynchronní způsob odpovědi — odesilateli je oznámeno úspěšné přijetí požadavku a poté je sezení uzavřeno. Na odpověď může odesilatel čekat opět dvěma způsoby:
   (a) aktivně, tzv. *polling* – periodicky zasílat synchronní požadavky na zjištění stavu původního požadavku;

(b) pasivně – po vypracování původního požadavku služba zajistí, aby o tom byl odesilatel informován. Součástí tohoto upozornění může být i samotná odpověď.

Pro komunikaci s webovým API, které bude poskytovat i vytvářený systém, se typicky používá protokol HTTP. To odpovídá charakteristice synchronního způsobu odpovědí – webová API typicky odpovídají po nejvýše několika sekundách čekání. V distribuovaných systémech však nelze obecně určit horní časový limit pro zpracování požadavku, neboť kromě doby jeho zpracování je nutné započítat i zpoždění způsobená přechodnými výpadky a chybami komunikace. Klienti protokolu HTTP by tak museli využívat velmi vysokých nebo nekonečných časových limitů odpovědí.

Aktivní čekání na asynchronní odpověď je vhodným způsobem pro komunikaci s API systému, pokud není k dispozici vhodnější komunikační protokol. Nevyžaduje ze strany klienta podporu žádných dalších technologií, generuje ale zbytečnou zátěž na odesilatele, který musí generovat periodické požadavky, i na API, které požadavek musí zpracovat a odpovědět na něj.

Pasivní čekání na asynchronní odpověď odstraňuje nevýhody aktivního čekání, na druhou stranu však nevyužívá již ustaveného komunikačního kanálu a závisí tak na možnosti odesilatele adresovat a na jeho dostupnosti na uvedené adrese. Specifika tohoto způsobu komunikace a řešení s tím spojených problémů lze delegovat na k tomu určený nástroj třetí strany určený k zasílání zpráv.

Message Broker

Message Broker je program, jehož primárním účelem je provádět operace iniciované přijetím formálně definovaných zpráv [41]. V systémech založených na architektuře *microservices* se takový program často používá jako prostředek pro komunikaci mezi službami, což umožňují následující požadované funkcionality:

- **Abstrakce komunikačních protokolů klientů** – každý příjemce/odesilatel zpráv může s message brokerem komunikovat nezávislým protokolem. Služby tak mezi sebou nejsou vázány závislostí na konkrétní komunikační protokol, vzniká pouze závislost na množinu protokolů podporovaných message brokerem.
- **Poskytování implementace různých modelů komunikace** – message broker umožňuje připojeným klientům komunikovat různými modely komunikace, aniž by je museli sami implementovat, jako např. pozorovatel, požadavek s asynchronní odpovědí, notifikace, fronta požadavků.
- **Vytváření pojmenovaných instancí komunikačních kanálů výše uvedených typů** – pojmenováním instancí je zajištěno jejich adresování.
- **Zajištění kvality služeb** – pro message broker je typicky možné nastavit požadovanou úroveň kvality služby z hlediska doručení zpráv:
  1. *best effort* – doručení zprávy není garantováno;
  2. *at least once* – doručení zprávy je garantováno alespoň jednou, připouští se však možnost vícenásobného doručení;
  3. *exactly once* – doručení zprávy je garantováno právě jednou.

Za tímto účelem bývá možné využít možnost perzistence přijatých zpráv do doby, než jsou aplikací zpracovány, aby bylo jejich doručení garantováno i v případě výpadku message brokeru.

- **Škálovatelnost, podpora clusterů** – pro účely vyšší výkonosti a/nebo odstranění SPOF může být jediný logický message broker v aplikaci provozován jako více fyzických spolupracujících uzlů, ať už pro účely vyrovnávání zátěže nebo v režimu aktivního a záložního uzlu.

Komunikace s okolím systému

Za účelem zprostředkování přístupu třetích stran do systému je nutné vyřešit několik otázek:

1. **Komunikační protokol** – různé typy klientů mohou používat/preferovat různé protokoly a serializační formáty dat. Protokol i formát je potřeba transformovat do alternativ použitelných/používaných v systému.

   Tento problém dokáže ze značné části řešit samotný message broker. V případě nutnosti podpory protokolu/formátu dat nepodporovaného samotným brokerem je možné zavést do systému další službu určenou k překladu/transformaci zdrojových požadavků tak, aby s nimi dokázal pracovat.

2. **Granularita služeb** – vzhledem k záměrnému omezování účelu služeb v architektuře *microservices* na dostatečně malé části problémové domény je nepravděpodobné, že by klientům stačilo získat data z jediné služby.

   Jedním z možných řešení je opět zavedení speciální služby k obsluze konkrétní části API, která klientům zajistí vyžádání a agregaci výsledků z jednotlivých služeb. Výhody tohoto přístupu, tj. abstrakce vnitřní kompozice systému a snížení celkové doby obsluhy (komunikací uvnitř systému místo komunikace vnějšího účastníka s jednotlivými službami) převáží nevýhodu v zavedení další služby navíc.

3. **Autentizace a autorizace** – příchozí požadavky musí být před zpracováním autentizovány a autorizovány, aby bylo zabráněno neoprávněnému přístupu a manipulaci se spravovanými daty. Implementací autentizace a autorizace na hranicích systému odpadá potřeba tyto úkony distribuovat do jednotlivých služeb, což by navíc porušovalo princip jejich izolovanosti.

Konzistence dat

Relační SŘBD typicky k zachování konzistence dat nabízí model transakčního zpracování vyznačující se vlastnostmi označovanými v angličtině akronymem ACID [48]:

- *Atomicity* – ve výsledku se buď provedou všechny dílčí operace, ze se transakce skládá, nebo žádná z nich.
- *Consistency* – provedením transakce nesmí být způsobena inkonzistence databáze.
- *Isolation* – souběžně prováděné transakce se navzájem neovlivňují; efekt transakcí bude stejný, jako by byly provedeny sekvenčně.
- *Durability* – po dokončení transakce jsou všechny změny provedené jejími dílčími operacemi trvalé a to i v případě následného výpadku SŘBD.

Konzistenci dat je samozřejmě žádoucí zajistit i v distribuovaných systémech, kde je taktéž možné využít transakčního zpracování, ovšem v jeho distribuované podobě. Umožňuje to například standard *X/Open XA* [43], jež specifikuje koordinaci provádění transakcí založenou na dvoufázovém uzamykacím protokolu. S tímto způsobem realizace distribuovaných transakcí se však v architektuře *microservices* pojí řada problémů popsaných v [30]:

- **Porušení principu datové svrchovanosti jednotlivých služeb.** Služby účastnící se transakce již nemohou perzistenci svých dat zajišťovat libovolným SŘBD, ale výhradně takovým, který podporuje transakční zpracování.
- **Porušení principu volné vazby mezi službami.** SŘBD využívané účastnickými službami musí podporovat koordinaci transakčního zpracování navzájem kompatibilním standardem.
- **Porušení principu izolovanosti jednotlivých služeb.** SŘBD a vnitřní datový model již nejsou prvky, které by služba mohla zapouzdřit, protože je musí zpřístupnit ostatním službám zúčastněným v transakci. To může znesnadnit kontrolu přístupu nebo umožnit manipulaci s daty bez využití rozhraní služby.
- **Nedostupnost uzamčených dat.** Dvoufázový uzamykací protokol typicky využívá pesimistické mechanismy souběžného zpracování, tj. využívající uzamykání dat, se kterými je v rámci transakce manipulováno. Pro transakce s dobou trvání v řádu stovek milisekund to nemusí být problém, při déle trvajících transakcích, ke kterým v distribuovaném prostředí může docházet, to může mít negativní dopad na výkonnost služby způsobený nedostupností uzamknutých informací.
- **Koordinátor transakce je Single Point of Failure** a jeho výpadek může vést k nekonzistentnímu stavu.

Model konzistence dat označovaný v angličtině akronymem BASE (převzato z [32]) vznikl selektivním vynecháním a rozvolněním vlastností ACID. Pro distribuované databáze a architekturu *microservices* je vhodnější zejména proto, že neklade požadavky na použité SŘBD a že nevyžaduje zamykání dat zahrnutých v transakcích [35]. Jeho charakteristiky jsou:

- *Basically Available* – data jsou dostupná v podstatě nepřetržitě;
- *Soft State* – nemusí však být v konzistentním stavu, naopak mohou být zastaralá nebo jen přibližná;
- *Eventual Consistency* – nakonec data v konzistentním stavu budou, typicky však nelze přesně určit, kdy k tomu dojde.

Transakce podle BASE nemusí být oproti ACID transakcím izolované ani atomické. Připouští se možnost pozorovat systém v dočasně nekonzistentním stavu, než je transakce dokončena nebo anulována.

3.2 Návrhové vzory pro koordinaci distribuovaných transakcí

Pro koordinaci distribuovaných transakcí podle BASE budou v systému použity návrhové vzory popsané v této sekci. Pro zajištění spolehlivé konzistence transakcí je ovšem kromě jejich samotné implementace dále nutné:

- **Vhodně navrhnout a perzistovat stav objektů, které je implementují.** Příchozí zprávy signalizující výsledek dílčí transakce je nutné zpracovat, aby změnily vnitřní stav těchto objektů, poté tento stav trvale uložit (např. do relační databáze s využitím transakcí) a až nakonec odeslat prostřednictvím message brokeru výstupní zprávy k zahájení další dílčí transakce, nebo oznámení výsledku. V případě obnovení systému po výpadku mohou být v nedokončené transakce kompenzovány.
- **Zajistit spolehlivost a trvalost zpráv v message brokeru**. Perzistenci zpráv je možné zajistit i mimo samotný aplikační kód — provozováním message brokeru v clusteru několika uzlů[2] a jejich nastavením tak, aby doručování zpráv vždy předcházela jejich perzistence.

Oba přístupy je dále možné kombinovat — je-li dodržena idempotence dopředných a kompenzačních operací dílčích transakcí[3], pak jejich opakované zpracování — v důsledku kteréhokoliv ze zmíněných přístupů — neovlivňuje již konzistentní stav systému.

Návrhový vzor *sága*

Návrhový vzor *sága* představuje posloupnost dílčích transakcí, z nichž každá mění stav jedné autonomní služby [29]. Zpracování počáteční transakce je zahájeno vnějším podnětem. Poté, co je dokončena kterákoliv z dílčích transakcí, je zahájeno zpracování další transakce v pořadí v případě úspěchu, nebo kompenzace dosud provedených transakcí v případě selhání. Každá z dílčích transakcí implementuje dva typy operací:

- dopřednou operaci – mění stav systému v souladu s účelem transakce, např. vytváří novou objednávku;
- kompenzační operaci – mění stav systému tak, aby zrušila platnost dopředné operace, např. nastavení příznaku „objednávka stornována“, odstranění objednávky ze systému, nebo vyvolání chyby a záznam žádosti o ruční kompenzaci ságy lidským uživatelem (objednávka již byla expedována a nelze ji zrušit).

S přihlédnutím k faktu, že komunikační rozhraní služeb je tvořeno publikovanými zprávami, nabízí se dva způsoby, jak návrhový vzor implementovat. Tyto způsoby se liší umístěním kontrolní logiky distribuované transakce:

- *Choreography* — distribuované řízení. Po dokončení vlastní operace služba určí, která služba je odpovědná za další dílčí transakci, a odešle jí zprávu pro její zahájení.

  Tento způsob implementace je vhodný pro transakce, kde výsledek dílčích transakcí neovlivňuje posloupnost budoucích kroků. V opačném případě může dojít k následujícím problémům:

[2] viz sekce 3.1, oddíl Message Broker
[3] viz další sekce Návrhový vzor *sága*

  - Porušení principu izolovanosti služeb — umístěním integrační logiky do služby, která není určena jako integrační a nemá tak mít povědomí o ostatních službách v systému.
  - Nárůst typů nebo složitosti zpráv definovaných v systému — informaci o typu výsledku je nutné předat dále jako hodnotu uloženou ve zprávě nebo v jejím typu.

- *Orchestration* — centralizované řízení. Řízení distribuované transakce zajišťuje zvláštní k tomu určená služba na základě zpráv zaslaných službami odpovědnými za dílčí transakce. Takovou službu budu nazývat jako integrační. Tato varianta zachovává volnou vazbu a izolovanost jednotlivých služeb, zvyšuje však složitost systému zavedením další služby.

  Řídící služba ságy může obsahovat aplikační logiku, která jí může umožnit reagovat na výsledky a selhání dílčích transakcí jinak, než kompenzací všech doposud dokončených dílčích transakcí.

  Taktéž je možné vytvořit řídící službu bez aplikační logiky, která místo jejího řízení pouze sleduje její průběh, který poskytuje ostatním službám nebo např. ukládá do své databáze.

Návrhový vzor *Routing Slip*

Variantu *Choreography* návrhového vzoru sága lze implementovat i tak, aby nedocházelo k porušení principu izolovanosti služeb. Ne všechny distribuované transakce podle BASE totiž nutně vyžadují přítomnost rozhodovací logiky, v případě transakcí typu CRUD naopak stačí jako výsledek transakce zjistit, zda byla úspěšná a jaká byla získána data. Zavedením řídící služby pro každou CRUD transakci by se zbytečně zvyšoval počet služeb v systému.

Proto je možné zavést obecnou řídící službu a specifikovat rozhraní zpráv zahajujících ságy tak, aby zahrnovalo informace potřebné k řízení distribuované transakce – tyto informace lze v angličtině označit jako *routing slip* [10]:

- zásobník zatím neprovedených dílčích transakcí, které mají být provedeny po té aktuální;
- frontu již provedených dílčích transakcí, které mají být kompenzovány v případě selhání té aktuální.

3.3 Architektonické celky

Na základě ovlivňujících faktorů a zvolené systémové architektury lze nyní provést dekompozici formálně specifikovaného zadání na odpovídající prvky. Tomu nejen s ohledem na spojitost charakteru systému s IoT předchází klasifikace systému z pohledu z vyšší úrovně, než z úrovně služeb tvořících jeho jádro.

Komponenty systému a jejich souvislost s IoT

Rahman, Ozcelebi a Lukkien v jejich článku uvádí několik způsobů klasifikace prvků v IoT systémech podle jejich účelu, druhu fyzické reprezentace a specifikací výpočetních parametrů zařízení, kde jsou realizovány [27]. Níže definované prvky vytvářeného systému jsou

podle těchto kritérií klasifikovány v příloze C. Při definici prvků bylo uvažováno všech podporovaných scénářů použití systému uvedených v kapitole 2.2.

1. **RFID čip** — slouží k identifikaci uživatele.
2. **RFID čtečka** – zajišťuje komunikaci s RFID čipem za účelem přečtení jeho sériového čísla a poskytuje tuto informaci zbytku systému.
3. **Displej** — zobrazuje stav místa prodeje a poskytuje zpětnou vazbu uživatelům.
4. **Tlačítka** — rozšiřují uživatelské rozhraní místa prodeje tvořené RFID čtečkou, mohou být vhodná např. pro volbu produktu před přiložením RFID čipu.
5. **Místo prodeje** — alternativa agregující prvky 1–4 do jediného kompozitního prvku nebo prvek, který přijaté systémové zprávy směruje na jednotlivé prvky a naopak.
6. **Jádro na bázi microservices** — zajišťuje řízení ostatních prvků v systému, je odpovědné za poskytování API.
7. Message Broker — umožňuje vzájemnou komunikaci systémových prvků s minimálními vazbami na použité komunikační protokoly, viz příslušný oddíl sekce 3.1.
8. **Klient protokolu Multi-Drop Bus/Internal Communication Protocol** — poskytuje systému stav integrovaného kávovaru a umožňuje jeho řízení překladem systémových zpráv.
9. **Zařízení poskytující IP konektivitu** — jednotlivé prvky je třeba připojit k IP síti. V závislosti na hardwarovém vybavení fyzických zařízení, kde budou systému prvky realizovány, může tuto roli realizovat síťový přepínač nebo např. bezdrátový směrovač.

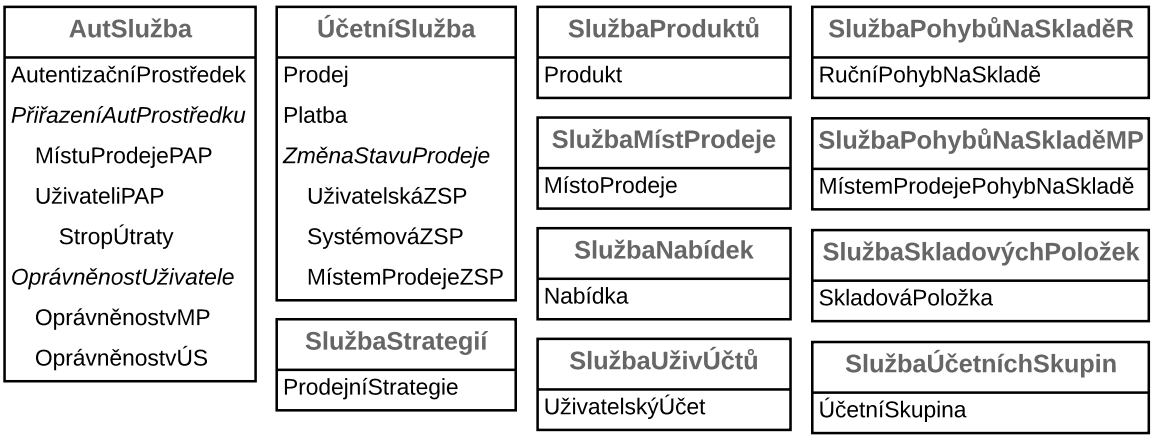
Jádro založené na architektuře microservices

V souladu s doporučením [35] implementovat v každé službě pouze omezený kontext problémové domény jsem při návrhu množiny služeb začal homomorfní transformací entit v diagramu A.1, čímž vznikla množina služeb odpovědných vždy za právě 1 systémovou entitu. U každé služby se předpokládá implementace rozhraní pro základní CRUD operace nad vlastněnými entitami.

Vzhledem k denormalizaci relačního modelu způsobené charakterem distribuované databáze nejsou zaváděny relace cizích klíčů, jejichž implementace by vyžadovala transakční zpracování podle ACID. Relace jsou stále implementovány použitím hodnot primárních klíčů v odkazujících se entitách, ze strany SŘBD však není kontrolována jejich platnost. Vztahy znázorňující dědičnost byly denormalizovány kopií atributů rodičovských entit do entit synovských.

Odchylkou od výše popsaného přístupu k návrhu služeb transformací z entit ER diagramu je služba `ÚčetníSlužba`, do které jsem se rozhodl sloučit vlastnictví entit `Platba`, `Prodej`, `ZměnaStavuProdeje` a z ní odvozené entity, a to z těchto důvodů:

1. **Výhody relačního SŘBD coby úložiště dat o toku finančních prostředků.** Ačkoliv je pomocí metod popsaných v kapitole 3.2 možné zajistit spolehlivou konzistenci dat podle modelu BASE, není možné určit, za jakou dobu bude stav takové



Obrázek 3.1: Diagram systémových služeb odpovědných za část datového modelu. Ke každé ze služeb je kromě jejího pracovního názvu uveden i seznam entit, za jejichž vlastnictví odpovídá a k manipulaci s kterými musí poskytovat odpovídající rozhraní. Kurzívou jsou vysázeny entity, na které v ER diagramu existují vazby dědičnosti. Odpovídající entity s těmito vazbami jsou dále odsazeny levého okraje. Entita `StopÚtraty` je odsazena vzhledem k jejímu charakteru slabé entitní množiny a závislosti na entitě `UživateliPAP`.

databáze konzistentní. V případě selhání libovolné části systému např. při prodeji nebo navyšování uživatelského zůstatku může být taková nedefinovaná prodleva způsobit nepříjemnou uživatelskou zkušenost. Použitím klasického relačního SŘBD sice nelze tuto možnost úplně eliminovat, lze ovšem snížit její pravděpodobnost.

2. **Očekávaná míra výskytu žádostí o získání a změnu uživatelského zůstatku.** Vzhledem k účelu vytvářeného systému předpokládám, že budou žádosti pro manipulaci se zůstatkem jedním z nejčastěji zasílaných typů zpráv. Protože je hodnota uživatelského zůstatku v systému definována jako rozdíl sum plateb přijatých uživatelem a částek utracených za potvrzené prodeje, věřím, že sloučení obou zdrojových informací do jedné služby a jednoho SŘBD pozitivně ovlivní dobu obsluhy zmíněných požadavků.

3. **Úzká souvislost změn stavů prodeje.** Prodeje ovlivňují uživatelský zůstatek jen v případě, že se nachází ve stavu „potvrzeno“. Protože je stav nákupu rovněž definován agregací všech jeho známých stavů[4], je jeho uložení ve stejném SŘBD žádoucí ze stejných důvodů, jako v obou předchozích bodech.

Vzhledem k nutnosti autorizovat požadavky příchozí do systému jsem ze stejných důvodů jako v bodě 2 sloučil vlastnictví entity `AutentizačníProstředek`, entit odvozených od `PřiřazeníAutProstředku` a entit odvozených od `OprávněnostUživatele` do služby `AutSlužba`, která bude v systému zajišťovat autentizaci i autorizaci.

Obrázek 3.1 znázorňuje navrženou množinu služeb vlastnících část systémového datového modelu, kromě kterých však budou v systému přítomny nejméně také integrační služby (např. ty odpovědné za řízení distribuovaných transakcí, viz kapitola 3.2).

[4] viz Funkční požadavky, strana 9, bod 9

Webový portál

Pro uživatele systému bude vytvořen webový portál, na kterém budou mít k dispozici alespoň následující operace:

- Přihlášení.
- Odhlášení a změna hesla přihlášeného uživatele.
- Zobrazení zůstatku přihlášeného uživatele.
- Zobrazení historie nákupů a plateb přihlášeného uživatele.

Uživatelé s přidělenými oprávněními k místům prodeje nebo účetním skupinám budou mít dále k dispozici operace související s částmi systému, k jejichž správě jsou oprávněni, například:

- Přehled k historii prodejů a k agregovaným statistikám míst prodeje a/nebo účetních skupin, které je uživatel oprávněn spravovat.
- Ruční anulování prodejů vytvořených místy prodeje nebo účtovaných v účetních skupinách, které je uživatel oprávněn spravovat.
- Přístup k informacím o skladovém stavu položek, které jsou nabízeny v místech prodeje, které je uživatel oprávněn spravovat. Správa skladových položek, nabídek a vlastností takového místa prodeje.

Pro uživatele s příznakem správce systému bude portál umožňovat všechny výše uvedené operace vhodně rozšířené tak, aby bylo systém a v něm obsažené entity možné spravovat bez zásahu do databází jednotlivých služeb — např. o vytváření a správu uživatelských účtů, míst prodeje nebo autentizačních prostředků a jejich přiřazení.

Přístupový klient pro integrovaný kávovar

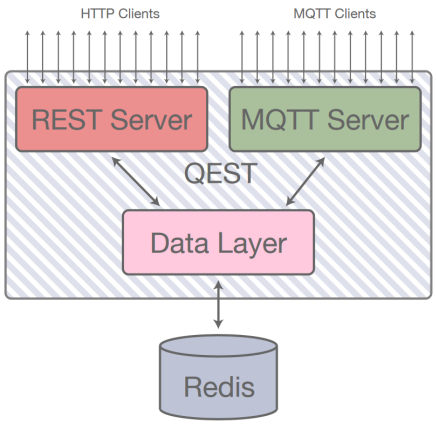
Pro scénář použití „Integrovaný kávovar s periodicky splácenou útratou“ (viz kapitola 2.2) bude připraveno schéma zapojení a řídící program mikrokontroléru založený na [9], který bude zajišťovat následující funkce:

- Komunikaci se čtečkou karet pro zjištění sériového čísla přiloženého RFID čipu.
- Komunikaci s message brokerem pomocí protokolu MQTT potřebnou pro realizaci požadované funkce zprostředkování prodeje.
- Obsluhu výstupních pinů mikrokontroléru v závislosti na přijatých zprávách pro zajištění výdeje kávy.

REST API

Výše zmíněný webový portál má dle zadání využívat REST API poskytované aplikací, konkrétně jejím jádrem na bázi microservices. Representational State Transfer (REST) je množina návrhových omezení, zvaných v angličtině podle jejich autora také jako *Fielding constraints*, která systémům připojeným k síti umožňují vystupovat jako systémy webové [28]. Jejich dodržením lze vytvořit dobře použitelné API na bázi protokolu HTTP s očekávatelným chováním.

Vzhledem ke specifikovanému rozsahu operací, které má webový portál zpřístupňovat, bude REST API zpřístupňovat všechny prvky systému.



Obrázek 3.2: Architektura QEST brokeru. Převzato z [3].

QEST API

Nevýhody protokolu HTTP — ke kterému se REST vztahuje — při použití v prostředí IoT nebo pro komunikaci modelem publish–subscribe[5], lze odstranit implementací QEST — message brokeru podporujícího komunikaci jak prostřednictvím protokolu HTTP, tak některým z protokolů určených k výměně zpráv zmíněným komunikačním modelem, např. MQTT [3]. Při změně stavu některého z publikovaných zdrojů, tj. entity publikované v API pod konkrétním URI, zajistí systém publikování nového stavu do použité databáze typu klíč–hodnota, odkud je implementovaným message brokerem zaslána informace o změně všem klientům, kteří o to požádali kterýmkoliv z podporovaných protokolů.

Při přístupu ke zdroji dostupného prostřednictvím QEST brokeru může dojít k situaci, kdy jeho hodnota nebyla od posledního spuštění aplikace změněna, a použitá databáze typu klíč–hodnota tak dosud neobsahuje jeho aktuální hodnotu. Nabízí se několik způsobů řešení, z nichž bude zvoleno ve fázi implementace systému. Jako možné varianty se nabízí například:

1. Poskytovat prostřednictvím QEST brokeru pouze změněné hodnoty a v případě její absence odpovídat chybou.

2. Rozšířit návrh QEST brokeru z [3] tak, aby použitá databáze typu klíč–hodnota sloužila jako vyrovnávací paměť a aby broker umožňoval aktivní získání chybějící hodnoty.

3.4 Implementační nástroje a technologie

Dalším krokem při návrhu vytvářeného systému je výběr technologií, které budou použity při následné implementaci. U každé z popsaných technologií bude zdůvodněna její relevance k vytvářenému systému a k požadavkům, které pro byly pro systém v předchozí části práce definovány.

[5] viz sekce Komunikace mezi službami, kapitola 3.1, str. 12

.NET Core

„.NET“ je bezplatně dostupná platforma s otevřeným zdrojovým kódem určená k vývoji a provozování platformě nezávislých aplikací [15]. Různé implementace umožňují využití platformy v rozličných prostředích:

1. **.NET Framework** je určen pro desktopové aplikace, služby a webové stránky provozované na operačním systému Windows.
2. **.NET Core** je implementace platformy .NET nezávislá na operačním systému Windows určená k vývoji a provozu serverů, webových stránek a konzolových aplikací také na operačních systémech rodiny Linux a macOS.
3. **Xamarin/Mono** je implementace určená pro vývoj přenositelných mobilních aplikací.

Jednou z předností platformy .NET je její nezávislost na konkrétních programovacích jazycích, lze totiž využít libovolný jazyk umožňující dodržení standardu CLI[6]. Spouštění kódu psaného ve vysokoúrovňových programovacích jazycích je na platformě .NET typické pomocí mezikódu CIL interpretovaného běhovým prostředím CLR[39]. Podle [35] lze v ASP.NET Core v současnosti vyvíjet v jazycích C#, Visual Basic .NET a F#. Systém budu vzhledem ke svým zkušenostem s ním vyvíjet právě v jazyce C#.

Vzhledem k požadavkům na nezávislost vytvářeného systému na platformě a k jeho charakteru webové služby připadá v úvahu pouze implementace .NET Core, pro kterou je dostupný bezplatný framework s otevřeným zdrojovým kódem určený k vývoji webových stránek a služeb — ASP.NET Core [31]. Pro jejich provozování je k dispozici oficiální podpora pro nasazení v Docker kontejnerech [35].

JSON Web Token

JSON Web Token (JWT) je mechanismus pro přenos autorizačních a identifikačních informací [11], který bude v systému využit. JWT jsou kódovány způsobem umožňujícím jejich přenos ve formátu `application/x-www-form-urlencoded` bez nutnosti dalšího kódování, tj. např. jako součást URL nebo jako hodnota hlavičky HTTP.

Takto kódovaný řetězec vždy obsahuje alespoň jeden objekt v notaci JSON, v němž lze uložit autorizační a identifikační informace. Spolu s ním je v JWT obsažena hlavička identifikující verzi JWT a typ použité funkce kontrolního součtu, který je taktéž součástí JWT a k jehož výpočtu je nutné znát tajný klíč, na jehož základě lze rovněž ověřit, že byl přijatý JWT vydán důvěryhodným zdrojem.

JSON objekt obsahující zmíněné autorizační a identifikační informace lze taktéž jako celek šifrovat pomocí asymetrické kryptografie a znemožnit tak jeho čtení třetím stranám [11].

Pro rámec ASP.NET Core je k dispozici oficiální podpora pro autentizaci a autorizaci pomocí JWT, dostupná jako balíček ve službě NuGeT [35].

MariaDB, Entity Framework Core

MariaDB je bezplatně dostupný relační SŘBD s otevřeným zdrojovým kódem, který je zpětně kompatibilní se SŘBD, z jehož zdrojových kódů vznikl – s MySQL [44]. Na platformě

[6] http://www.ecma-international.org/publications/files/ECMA-ST/ECMA-335.pdf

.NET Core lze pro práci s MariaDB vzhledem ke zmíněné zpětné kompatibilitě využít databázového ovladače ADO.NET pro MySQL[7]. Další technologií využitou pro práci s databází bude Entity Framework Core — nástroj pro zprostředkování ORM a verzování databázových schémat, vytvářející další abstrakční vrstvu nad ADO.NET a použitým databázovým ovladačem [16].

RabbitMQ

RabbitMQ je implementace message brokeru splňující požadavky zmíněné v kapitole 3.1:

- Prostřednictvím zásuvných modulů podporuje komunikaci přes množství protokolů, jako např. AMQP, MQTT nebo HTTP [25].
- Poskytuje implementace komunikačních modelů publish–subscribe, front požadavků, RPC a další [23].
- Umožňuje vytváření pojmenovaných kanálů komunikačního modelu publish–subscribe zvané *topics*.
- Podporuje nastavení kvality služeb a vytváření clusterů [24].

Pro komunikaci s RabbitMQ je pro platformu .NET Core poskytována oficiální knihovna tříd, podporující komunikaci protokolem AMQP [22]. RabbitMQ server je možné provozovat i prostřednictvím oficiálního obrazu Docker.

Komunikační protokoly

Systém bude pro komunikaci využívat primárně dosud zmíněné protokoly aplikační vrstvy využívající transportního protokolu TCP/IP, tj.:

1. HTTP — zejména pro komunikaci s poskytovaným REST API;
2. AMQP — pro komunikaci jednotlivých služeb s message brokerem RabbitMQ;
3. MQTT — pro komunikaci s poskytovaným QEST API, případně integračního mikrokontroléru s message brokerem.

Srovnání posledních dvou z těchto protokolů v [2] popisuje jejich vzájemné rozdíly, z nichž některé jsou shrnuty v následujících několika bodech. Oba protokoly jsou primárně určeny k asynchronní komunikaci prostřednictvím výměny zpráv, liší se však rozsahem a podporou souvisejících funkcionalit.

MQTT byl navržen pro příjem a odesílání velkého množství malých zpráv z hlediska objemu dat, na sítích s nízkou šířkou pásma. Podporuje potvrzování přijetí zpráv. Zabezpečených spojení lze dosáhnout využitím TLS, což umožňuje autentizaci klientskými certifikáty. Na aplikační vrstvě MQTT lze pro autentizaci využít uživatelských jmen a hesel, jejich zpracování a vynucování je však ponecháno konkrétním implementacím [18].

Návrh AMQP se snaží zohlednit množství komunikačních modelů, rozšiřitelnost a širší rozsah podporovaných scénářů použití. Na rozdíl od MQTT podporuje na aplikační úrovni fragmentaci zpráv užitečnou pro nespolehlivé sítě s nízkou šířkou pásma nebo také SASL[8]. Implementace standardu AMQP je proto složitější, činí z něj však vhodnější volbu pro složitější případy užití [2].

[7] https://dev.mysql.com/doc/connector-net/en/connector-net-entityframework-core.html
[8] https://en.wikipedia.org/wiki/Simple_Authentication_and_Security_Layer

MassTransit

MassTransit je bezplatně dostupný rámec pro platformu .NET pro vývoj distribuovaných aplikací založených na komunikaci výměnou zpráv [20]. Zprostředkovává abstrakční vrstvu nad konkrétním použitým message brokerem umožňující jeho využívání na základě naprogramovaných definic a s pouze minimální jemu specifickou konfigurací. Navíc poskytuje ve své knihovně tříd pro platformu .NET množství připravených komunikačních modelů, které je rovnou možné využít pro implementaci aplikační logiky, bez nutnosti programovat samotnou komunikaci. V podobě funkcionalit *Courier* a *Automatonymous* jsou rovněž dostupné implementace návrhových vzorů *routing slip* a ságа[9] (v tomto pořadí, viz kapitola 3.2).

Docker

Docker je software s otevřeným zdrojovým kódem umožňující automatizované nasazování aplikací na operačních systémech rodiny Linux a Windows metodikou kontejnerizace [35]. Ta spočívá v nasazování aplikace s využitím balíčků obsahujících všechny závislosti potřebné k jejímu běhu — spustitelnou aplikaci, její závislosti (jako např. knihovny) i potřebnou konfiguraci. Vývoj a ladění kontejnerizované aplikace je tak možné provádět v prostředí totožném s tím produkčním a značně se zjednodušuje proces škálování, který může spočívat v prostém spuštění další instance kontejneru na dalším hostitelském počítači.

Na rozdíl od virtualizace, která taktéž umožňuje automatizované nasazování aplikací, neobsahují kontejnery vlastní hostitelský OS a místo toho využívají prostředky toho, na kterém je spuštěno jádro pro kontejnerizaci, např. Docker. Výhodou tohoto přístupu je menší velikost souborů s kontejnerizačními obrazy, nižší spotřeba prostředků hostitelského OS a kratší doba spouštění. Na druhou stranu využívají všechny kontejnery funkcí stejného jádra hostitelského OS, což snižuje míru vzájemné izolovanosti spuštěných kontejnerů.

Jednotlivé služby popsané v kapitole 3.3 budou implementovány jako samostatné obrazy Docker, aby u nich bylo možné využívat výše popsaných výhod.

PlatformIO

S vývojem programů pro vestavěné systémy se pojí množství vývojových nástrojů a prostředí dodávaných výrobci těchto systémů. Jednotlivá řešení se liší nejen kvalitou a použitelností uživatelských rozhraní, tj. faktory ovlivňujícími rychlost vývoje, ale také obtížností instalace a dobou potřebnou pro přípravu funkčního procesu kompilace a nasazení vytvořených programů.

PlatformIO je řešení, které se výše zmíněné časté problémy snaží eliminovat poskytováním platformě nezávislé sady nástrojů pro jednotný vývoj podporující kromě ESP8266EX, na kterém bude integrace místa prodeje realizována, také množství dalších platforem vestavěných systémů [26]. Nad nástroji určenými k použití v prostředí příkazového řádku, představujícími základní způsob využívání dodaných nástrojů, existují doplňky pro množství IDE, což vývojářům umožňuje využívat jimi preferované vývojové prostředí.

Kromě zmíněných nástrojů je součástí ekosystému PlatformIO také unifikovaný nástroj pro ladění, jednotkové testování a správce knihoven.

[9] Funkcionalita *Automatonymous* umožňuje implementaci stavových automatů, které představují jednu z možností implementace chování návrhového vzoru ságа. Pro perzistenci stavu ság a jejich napojení na jsou v rámci MassTransit dostupné další prostředky.

Kapitola 4

Implementace systému

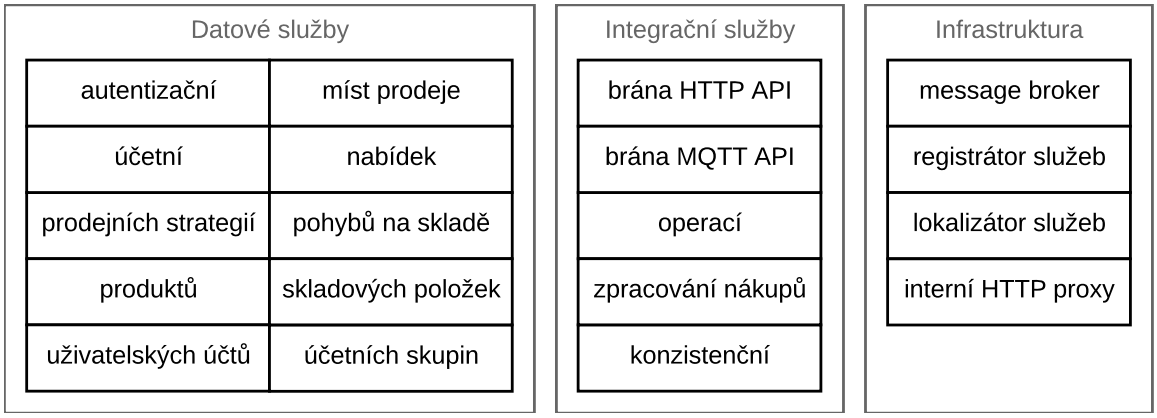
Implementaci systému jsem započal u jádra poskytujícího zdroj dat pro zbytek systémových součástí. Následně byla vyvinuta webová aplikace pro uživatele a správce systému, která využívá jádrem nabízené komunikační rozhraní. Nakonec bylo integrováno řešení pro autentizaci uživatelů a vytvořen program pro ovládání mikrokontroléru integrující výdejník fyzického místa prodeje.

4.1 Jádro systému zprostředkovávající REST API

V souladu s návrhem uvedeným v předchozí kapitole je jádro tvořeno množstvím datových služeb představujících jeho datovou vrstvu. Koordinaci jimi nabízených příkazů a publikovaných událostí pro různé účely dále zajišťují služby integrační. Za interakci systému s okolím jsou zodpovědné služby poskytující API – jedna pro již zmíněné REST API nad protokolem HTTP a jedna pro klienty preferující protokol MQTT. Tyto nejsou vzájemně zaměnitelné — funkcionalita poskytovaná oběma branami API není totožná. Dvojice služeb nakonec zabezpečuje konzistenci dat ukládaných v systému. V diagramu 4.1 je rovněž znázorněna komunikační infrastruktura služeb představující kromě message brokeru také nástroje pro registraci a lokalizaci instancí služeb a interní HTTP API, které lokalizaci služeb odpovědných za žádané koncové body zapouzdřuje.

Datové služby jádra

Datové služby jádra modelují systémové entity v podobě doménových tříd, které neobsahují žádné externí závislosti, jako např. kód odpovědný za serializaci či deserializaci dat. Jejich výhradním účelem je zapouzdřit stav modelovaných entit, který pro čtení publikují prostřednictvím *vlastností* jazyka C#. Pro manipulaci se stavem využívá zbytek systému k tomu určených metod, které ve vhodných případech provádějí taktéž validaci vstupu. Pro tyto účely jsou vhodnými případy validace založené výhradně na ukládané hodnotě nebo aktuálním stavu objektu. Lze např. ověřit, že nastavované zobrazovací jméno je neprázdné. Naopak ověření, zda nastavovaný identifikátor místa prodeje představující vazbu na jinou entitu skutečně existuje, již nespadá do kompetence doménové třídy a musí ji provádět komponenta na vyšší úrovni.



Obrázek 4.1: Diagram komponent jádra systému zprostředkovávajícího REST API. Oproti návrhu datových služeb v diagramu 3.1 byla implementována jediná datová služba pohybů na skladě obsluhující obě z dvojice entit `RučníPohybNaSkladě` a `PohybNaSkladěMP`.

Perzistence dat

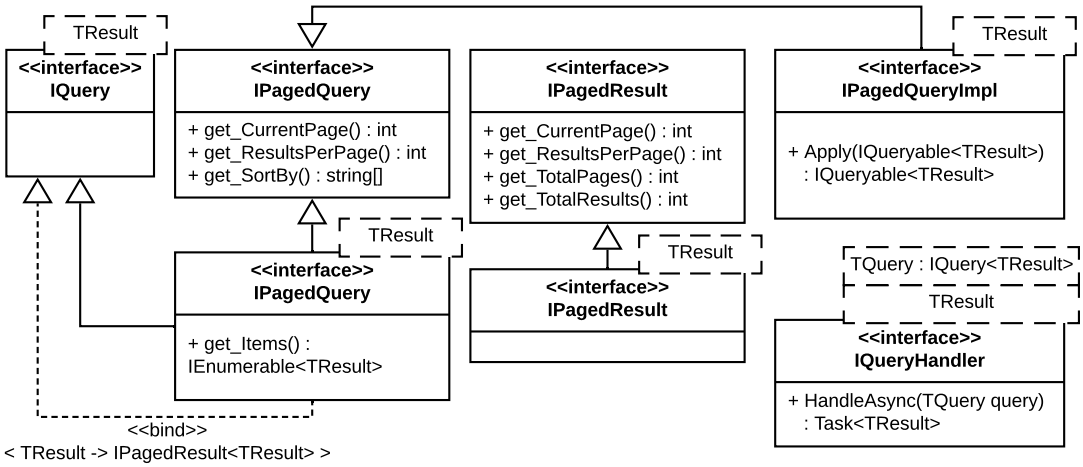
Načítání a ukládání stavu doménových tříd do perzistentního úložiště zajišťují datové služby s využitím knihovny Entity Framework Core pomocí tzv. *code–first* přístupu, kdy je databázové schéma generováno a udržováno na základě programové specifikace. Každá datová služba obsahuje třídu dědící od knihovnou nabízené třídy `DbContext`, ve které vlastnostmi jazyka C# specifikuje výčet doménových entit a úpravou příslušné zděděné metody poskytuje knihovně další informace o schématu, např. o indexech a klíčích. Na základě této třídy knihovna generuje SQL skript pro inicializaci databázového schématu. V případě zásadní změny v implementaci třídy `DbContext` se generují skripty pro dopřednou i zpětnou modifikaci schématu, které lze rovněž rozšířit o vlastní program, je-li např. potřeba dogenerovat obsah nově přidaného sloupce tabulky. Základní interakci s databází lze do určité míry programovat agnosticky k použitému SŘBD, neboť knihovna tříd Entity Framework Core poskytuje také služby ORM založené na sledování změn v instancích tříd datového modelu. Jednodušší, zejména jednotabulkové dotazy, knihovna generuje přepisem konstrukcí LINQ jazyka C#. V případě potřeby lze rovněž spouštět uložené procedury či vlastní příkazy jazyka SQL.

Návrhový vzor repozitář

Ačkoliv rozhraní nabízené knihovnou tříd Entity Framework Core již samo o sobě implementuje návrhový vzor repozitář, jehož účelem je poskytnout rozhraní zapouzdřující vnitřní logiku datového úložiště, obsahují datové služby ještě vlastní vrstvu repozitářů pro jednotlivé entity. Hlavním důvodem k tomuto kroku bylo umožnit přímočařejší jednotkové testování prostřednictvím omezenějšího rozhraní, než poskytuje Entity Framework Core. Umožňuje taktéž případnou výměnu Entity Framework Core za jinou technologii nebo zavedení vyrovnávací paměti způsobem transparentním vůči vyšším vrstvám aplikace.

Command Query Responsibility Segregation (CQRS)

CQRS je technika založená na rozdělení součástí aplikace odpovědných za čtení a za zápis dat [6], která v jistém smyslu zapadá do problematiky očekávání a/synchronních odpo-



Obrázek 4.2: Diagram tříd modelu dotazů a rozhraní obslužné třídy dotazu.

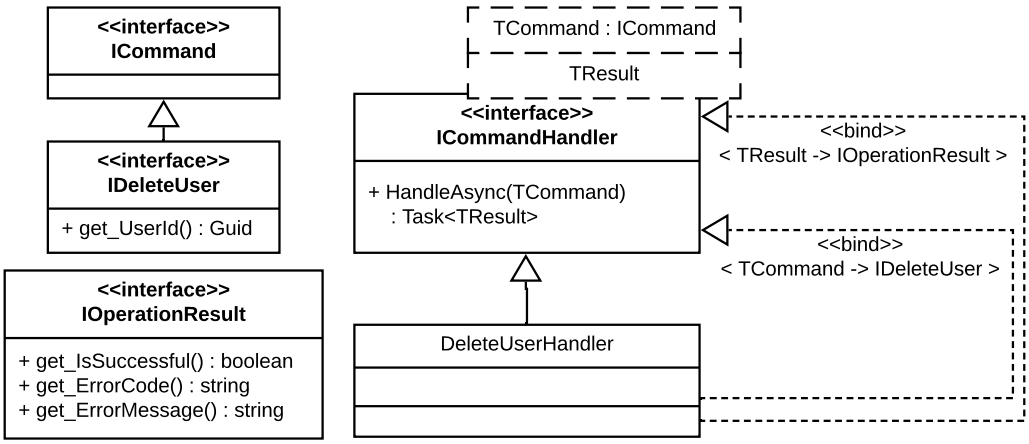
vědí popsané v kapitole 3.1. Oba druhy odpovědí je vhodné realizovat odlišnými protokoly, pro které lze také očekávat odlišné implementace. Aplikováním principu CQRS došlo v systému k vytvoření dvojice nadstaveb nad datovým modelem, ke kterým se pojí jejich vlastní abstrakční vrstvy.

1. **Model dotazů** definující druhy, parametry a návratové hodnoty požadavků na čtení dat ze systému. Konkrétní modely dotazů jsou realizovány třídami implementujícími rozhraní `IQuery`, které typovým parametrem `TResult` předepisují návratovou hodnotu dotazu. Parametry, jako např. identifikátor žádané položky, definují třídy vlastnostmi jazyka C#, znázorněných v diagramu tříd 4.2 metodami s prefixem „get_“. Modely dotazů určené ke čtění více než jedné položky implementují podle zavedené konvence rozhraní `IPagedQuery`, které nad rámec kontraktů `IQuery` zavádí podporu pro vstup a výstup informací o stránkování.

   Směrodatnou a hlavní implementací modelu dotazu je vždy ta využívaná její obslužnou třídou, kterou lze typicky nalézt v datové službě dotazované entity. Odlišuje se také tím, že implementuje rozhraní `IPagedQueryImpl` a v něm definovanou metodu pro aplikaci parametrů dotazu pomocí LINQ. Jednotlivé služby, které dotaz používají ke své funkci, zpravidla používají implementaci odlišnou obsahující pouze potřebnou podmnožinu parametrů dotazu.

2. **Model příkazů** definující druhy a parametry požadavků na manipulaci se systémovými daty. Návratový typ příkazu vždy dědí od rozhraní `IOperationResult` umožňujícího získat informaci, zda byl příkaz úspěšně obsloužen, případně kód a popis zjištěné chyby. Příkazy určené k vytváření instancí entit typicky využívají od něj odvozené rozhraní `IIdentifierResult`, které jej rozšiřuje o identifikátor vytvořené entity, případně `IParentChildIdentifierResult`, které jej rozšiřuje o dvojici identifikátorů.

   Konkrétní modely příkazů jsou realizovány rozhraními implementujícími rozhraní `ICommand`. V systému vždy existuje jen jedna taková implementace, a to v k tomu určeném sestavení `Barista.Contracts`. Služby, které příkazy odesílají, pak interně obsahují třídy, která tato rozhraní implementují, jelikož jsem chtěl využít možností



Obrázek 4.3: Diagram tříd modelu příkazů, rozhraní obslužné třídy příkazu a ukázková obslužná třída dotazu `DeleteUserHandler`.

statické typové kontroly a jelikož v jazyce C# není podporováno vytváření anonymních tříd implementujících rozhraní. Obslužné třídy příkazů, jako např. ta v diagramu tříd 4.3, umístěné zejména v příslušných datových službách manipulovaných entit tyto implementace nevyžadují, neboť vytváření instancí příkazových rozhraní zajišťuje knihovna tříd MassTransit.

Vazby mezi třídami v dotazové i příkazové vrstvě jsou realizovány výhradně prostřednictvím jimi implementovaných rozhraní s využitím techniky vkládání závislostí, což usnadňuje jednotkové testování obslužných tříd příkazů i dotazů.

Rozhraní dotazové vrstvy

Vnější rozhraní dotazové vrstvy každé z datových služeb zajišťují kontrolery ASP.NET Core, které přijímají požadavky příchozí na její interní HTTP API a transformují je na instance příslušných dotazových modelů. Využitím techniky vkládání závislostí je získána instance obslužné třídy, jejíž typový parametr `TQuery` (viz diagram tříd 4.2) odpovídá přijatému typu dotazového modelu. Návratová hodnota získaná invokací obslužené metody je serializována do očekávaného formátu JSON.

Obslužné třídy dotazů typicky vkládají závislost na jeden nebo více repozitářů potřebných pro načtení požadovaných informací. Do odpovědnosti obslužných tříd spadá taktéž konverze výsledků reprezentovaných instancemi doménových tříd na instance tříd sloužících jako Data Transfer Object, které uchovávané informace transformují do podoby vhodné k externí komunikaci a vhodné účelu dotazu. Tato konverze není v obslužných třídách implementována, neboť je delegována na knihovnu tříd AutoMapper, kterou obslužné třídy pouze volají. Mapovací funkce mezi zmíněnými typy tříd jsou konfigurovány jednorázově při spouštění systémových služeb.

Rozhraní příkazové vrstvy

Vnitřní rozhraní příkazové vrstvy je velice podobné tomu k nalezení ve vrstvě dotazové. Obslužné třídy příkazů vkládají kromě závislostí na repozitářích typicky také závislost na rozhraní zapouzdřujícím komunikaci s užívanou infrastrukturou do metod umožňujících

zasílání příkazů a publikování integračních událostí. V případě vazeb obsluhované entity na jinou/jiné vkládá ovšem obslužná třída navíc závislost/i na implementacích rozhraní pojmenovávaných dle konvence `IXyzVerifier`, které prostřednictvím HTTP požadavku metodou `HEAD` na koncový bod odkazované entity kontrolují její existenci před tím, než je vztah realizován uložením identifikátoru do databáze. Klientům je tak zabráněno ve vytvoření od počátku nekonzistentní relace.

Vnější rozhraní dále není k dispozici prostřednictvím interního HTTP API, jako tomu je u dotazové vrstvy, ale prostřednictvím message brokeru. Podobně, jako Entity Framework Core umožňuje práci s různými SŘBD jednotným rozhraním do značné míry nezávisle na konkrétním použitém systému, abstrahuje knihovna tříd MassTransit komunikaci prostřednictvím message brokeru. Při startu každé ze systémových služeb jsou prostřednictvím reflekce vyhledány všechny obslužné třídy dotazů a událostí a pomocí knihovny MassTransit je pro ně nakonfigurován odběr příslušného typu zpráv. Tato konfigurace zahrnuje kontrolu existence požadované topologie a její případné vytvoření.

Použitý message broker RabbitMQ disponuje dvěma typy využívaných komunikačních kanálů:

- **Směrovači** umožňujícími příjem zpráv, které dle svého režimu dále přeposílají do jiných navázaných komunikačních kanálů.
- **Frontami** umožňujícími přihlášení k odběru zpráv, přičemž je zpráva doručena právě jednomu klientovi, který si o ni první požádá.

Pro každý odebíraný typ zprávy, reprezentovaný rozhraním implementujícím `IEvent` nebo `ICommand`, vytváří v message brokeru knihovna tříd MassTransit odpovídající směrovač [19]. Ten je vytvořen také pro každý další bázový typ nebo implementované rozhraní, které typ zprávy implementuje. V případě přihlášení k odběru příkazu `IDeleteUser` z diagramu tříd 4.3 by byla ověřena existence směrovačů `Barista.Contracts:IDeleteUser` a `Barista.Contracts:ICommand`. Tyto směrovače jsou vytvářeny v režimu *fan out*, každá přijatá zpráva je tedy duplikována do všech navázaných komunikačních kanálů.

Nakonec je vytvořena fronta příjemce, kterým je příslušná obslužná třída, a tato je navázána na směrovač odebíraného typu zprávy. Název fronty již negeneruje knihovna MassTransit, systém jej však generuje obdobným způsobem. V případě zpráv implementujících rozhraní `IEvent` je název fronty odvozen od plně kvalifikovaného názvu typu obslužné třídy. V případě zpráv implementujících rozhraní `ICommand` je název fronty založen pouze na úplném názvu typu a není využit název sestavení .NET, ve kterém je typ zprávy definován. Narozdíl od integračních událostí je u příkazů očekáváno, že bude existovat právě jedna implementace jejich obslužné třídy a to v právě jedné systémové službě.

Nevýhodou plynoucí z výchozích jmenných konvencí směrovačů generovaných knihovnou MassTransit pro odebírané typy zpráv je nutnost využívat ve všech službách totožné typy definující zasílané zprávy, což vytváří závislost sdílenou všemi systémovými službami. Ačkoliv lze jmenné konvence nahradit vlastními, rozhodl jsem se je ponechat a vyhnul jsem se tak jinak nevyhnutelné duplikaci kódu rozhraní. Rozhraní zpráv zasílaných prostřednictvím message brokeru jsou proto umístěna v projektu a sestavení `Barista.Contracts`.

Zpracování dotazů, registrace a lokalizace služeb

Každá služba obsahuje v prostoru jmen `Services` rozhraní služeb jí využívaných pro čtení dat ze zbytku systému. Samotná rozhraní i v nich definované metody jsou dekorovány atributy jazyka C# poskytované knihovnou tříd RestEase, které specifikují způsob serializace

```
using System;
using System.Threading.Tasks;
using Barista.Api.Models.SaleStrategies;
using Barista.Api.Queries;
using Barista.Common.Dto;
using RestEase;

namespace Barista.Api.Services
{
    [SerializationMethods(Query = QuerySerializationMethod.Serialized)]
    public interface ISaleStrategiesService
    {
        [AllowAnyStatusCode]
        [Get("api/saleStrategies")]
        Task<ResultPage<SaleStrategy>> BrowseSaleStrategies(
            [Query] DisplayNameQuery query
        );

        [AllowAnyStatusCode]
        [Get("api/saleStrategies/{id}")]
        Task<SaleStrategy> GetSaleStrategy([Path] Guid id);
    }
}
```

Výpis 4.1: Program v jazyce C# definující rozhraní pro čtení dat ze služby prodejních strategií, umožňující zasílat požadavky na dva koncové body HTTP. V první z metod je definována serializace argumentu do podoby dotazu v URL, v druhé je specifikována vazba hodnoty parametru do cesty koncového bodu.

odpovědi, zpracování argumentů metod a umístění koncových bodů HTTP. Na základě těchto informací poskytuje knihovna tříd RestEase běhové dynamické implementace těchto rozhraní zapouzdřující infrastrukturní logiku vykonávaní dotazů jako obyčejná volání metod. Ve výpise 4.1 je uvedena ukázka rozhraní pro službu prodejních strategií.

Správné zpracování dotazu je kritické např. v případech, kdy je dotaz zaslán konzistenční službou (viz dále), která vyhledává entity v nekonzistentním stavu určené k odstranění. Chybně pojmenovaný parametr dotazu v rozhraní tázající se služby by tak mohl způsobit, že pro nerozpoznání nebude předán obslužné třídě a že budou vráceny výsledky, které měl tento parametr vyloučit. Z tohoto důvodu jsou všechny systémové služby vybaveny filtrem ASP.NET Core, který na neúspěšnou vazbu parametrů obsažených v URL reaguje chybovou odpovědí se stavovým kódem HTTP 400.

Po spuštění a dokončení inicializace systémových služeb probíhá z jejich strany registrace do systému Consul využívaného pro jeho funkcionalitu *service discovery*, kdy zodpovídá za udržování seznamu registrovaných služeb. Při registraci sděluje služba následující informace:

1. **Identifikátor služby** — identifikátor unikátní napříč instancemi všech služeb, generovaný náhodně při jejich spuštění. Platný po dobu běhu služby.
2. **Název hostitele a port** — údaje potřebné pro navázání spojení se službou, která se registruje. Tato hodnota se nastavuje v konfiguračním souboru služeb.

3. **Příznaky** — obvykle označují funkcionality nabízené službou, aby ji bylo možné vyhledat podle poskytovaných funkcionalit, namísto jiné pevné vazby, např. na název služby.

4. **Nastavení kontroly zdraví** — interval, v rámci kterého kontroluje Consul zdraví registrované služby, způsob této kontroly a doba, po jejímž uplynutí má být služba odregistrována pro nedostupnost.

Příkazová vrstva je integrována výhradně prostřednictvím message brokeru a klienti tak nemusí adresovat konkrétní cílovou službu, která má příkaz zpracovat. Vyhledávání registrovaných služeb je proto relevantní pouze pro vrstvu dotazovou implementovanou prostřednictvím HTTP API. Proto byla jako způsob kontroly zdraví zvolena kontrola stavového kódu HTTP vraceného službami na speciálním koncovém bodě `ping` poskytovaném doplňkem pro kontroly stavu v ASP.NET Core. Jeho pomocí jsou ověřovány dvě skutečnosti [35]:

- **Živost** — služba je spuštěna a jí poskytovaný HTTP server odpovídá na příchozí požadavky. Vedlejším efektem je ověřeno, že služba při registraci uvedla správný název hostitele a port.

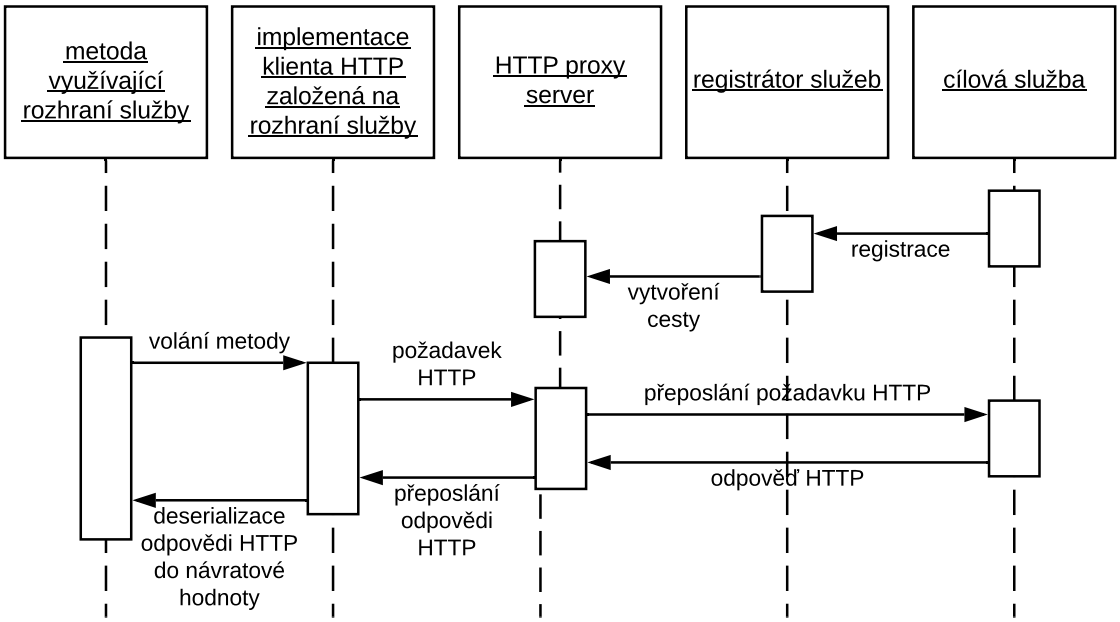
- **Připravenost** — ve výpočtu odpovědi a statového kódu na kontrolním koncovém bodě je zahrnut stav závislostí konkrétní služby. V systémových službách dochází ke kontrole spojení s message brokerem a tam, kde je to vhodné, i s využitým SŘBD.

Na základě seznamu zdravých služeb a jimi definovaných příznaků lze při provádění dotazu využít API poskytované systémem Consul k vyhledání služby, která jej dokáže zpracovat, a HTTP požadavek zaslat jí registrovanému hostiteli a číslu portu. Pro každý dotaz je pak ovšem nutné zaslat ještě jeden adresovací a hlavně vyčkat na jeho odpověď. Zavedení vyrovnávací paměti pro překlady by tato negativa pomohlo zmírnit, do značné míry však komplikuje zdánlivě přímočarý problém zaslání HTTP požadavku.

Z výše zmíněných důvodů využívají systémové služby vnitřní proxy server, na který problém s adresováním delegují. S minimální konfigurací je využit nástroj Fabio umožňující integraci s využívaným systémem Consul pro registraci služeb. Systémové služby při registraci specifikují příznaky ve formátu `urlprefix-/ENDPOINT`, kde `ENDPOINT` je cesta ke každému službou poskytovanému koncovému bodu jejího HTTP API, např. `urlprefix-/api/users`. Na jejich základě pak Fabio poskytuje HTTP proxy, která přichozí požadavky na takto definované cesty přeposílá službám, které je registrovaly [33]. V případě odebrání služby z důvodu selhání kontroly stavu jsou cesty automaticky rušeny, na požadavky s neznámou nebo odebranou cestou odpovídá proxy server stavovým kódem 502. V případě, že je k obsluze koncového bodu dostupná více než 1 služba, je mezi nimi nástrojem Fabio rovným dílem vyrovnávána zátěž. Průběh zpracování dotazů je znázorněn na diagramu v obrázku 4.4.

Analýza konzistence dat

Doposud popsaný mechanismus pro manipulaci s daty uloženými v systému, tedy příkazová vrstva, umožňuje provádění jednotkových operací na jednotlivých entitách. Sám o sobě neumožňuje provádění distribuovaných transakcí ani ošetření s tím souvisejících rizik způsoby popsanými v kapitole 3.2. Na základě ER diagramu dostupného v příloze A byla provedena analýza obsažených vztahů přiřazující každé entitní množině druh na ní závislých vztahů. Její výsledek dostupný v příloze B provádí klasifikaci entitních množin do dvou kategorií:



Obrázek 4.4: Sekvenční diagram zpracování dotazu zaslaného systémovou službou.

1. Entity, jejichž odstranění musí způsobit **kaskádovité odstranění** dalších entit. Do tohoto druhu byly identifikovány entitní množiny `AutentizačníProstředek`, `ÚčetníSkupina`, `MístoProdeje`, `Prodej`, `Produkt`, `SkladováPoložka` a `UživatelskýÚčet`.
2. Pro zbývající entity platí, že jejich odstraněním dojde k **úmyslnému zneplatnění s nimi spojených vztahů**. Uložené identifikátory, které s nimi vztah v databázovém schématu realizují, nebudou nadále odkazovat na platnou entitu. Toto je prohlášeno za očekávaný a požadovaný stav.

Dále jsou v diagramu klasifikovány i jednotlivé vztahy mezi entitními množinami a to podle níže popsaného požadovaného způsobu ošetření nekonzistence. Jako „nadřazená entitní množina“ je v popisu vždy myšlena ta z množin podílejících se na konkrétním vztahu, která je prvního typu (viz klasifikace entitních možin předchozím odstavci).

- **Vyžadující kaskádovité odstraňování** — entitní množiny podílející se na takto klasifikovaném vztahu jsou vlastněny různými datovými službami, kaskádovité odstraňování závislostí na nadřazené entitní množině musí být koordinováno integrační službou.
- **Odstraňované interní implementací** — při odstranění entity z nadřazené entitní množiny by za standardních okolností bylo rovněž nutné koordinovat kaskádovité odstraňování. Způsob, jakým byl E-R diagram transformován na databázové schéma a jakým služba, která jej vlastní, s tímto schématem pracuje, ov3em zajišťuje koordinaci odstraňování autonomně.

  Tato klasifikace byla vztahům vždy přiřazena z jednoho ze dvou možných důvodů:

  - V transformovaném schématu neexistuje rodičovská entitní množina u vztahů typu generalizace–specializace, např. vztah entitních množin `OprávněnostvÚS` a `OprávněnostUživatele` a všechny další vztahy tohoto typu.
  - V transformovaném schématu konzistenci zajišťuje použitý ORM — Entity Framework Core. Synovská entitní množina je v databázovém schématu sice realizována samostatnou tabulkou, manipulaci s ní a zajišťování její konzistence však není nutné řešit přímo.

- **Zachování úmyslné nekonzistence** — odstraněním entity z nadřazené entitní množiny je způsobeno, že vztah odkazuje na neexistující entitu. Tato nekonzistence je v systému úmyslně zachována. Systém o vztahu uvažuje tak, že v minulosti odkazoval na existující entitu, která však byla od té doby odstraněna.

  Zvláštní klasifikací s přívlastkem „kritické“ jsou pak označeny vztahy, kde je jednou ze zúčastněných entitních množin `Prodej` nebo `Platba`. Odstraněním do jedné z nich náležících entit by došlo ke ztrátě informací o pohybu finančních prostředků v systému.

Koordinace kaskádovitého odstraňování

Konzistenci dat po odstranění entity z jedné z účastnických entitních množin příslušně klasifikovaných vztahů zabezpečuje v systému integrační služba `Barista.Consistency`. Pro každou z nadřazených entitních množin zmíněných vztahů existují integrační události, které obslužné třídy jimi obsahovaných entit publikují do systému poté, co jsou odstraněny. Služba obsahuje obslužné třídy pro tyto události, které iniciují transakce využívající návrhový vzor *routing slip* popsaný v kapitole 3.2.

Tyto transakce jsou vždy složeny vždy z $1+n$ dílčích aktivit, kde n je počet souvisejících podřazených entitních množin. Pro každou z těchto n transakcí platí, že:

1. Třída, v níž je implementována, dědí od abstraktní třídy `ConsistencyRemediationActivity` ve jmenném prostoru `Barista.Consistency.Activitites`. Ta definuje dvojici abstraktních metod:

   - `FindAsync`, ve které odvozené třídy implementují vyhledání nekonzistentních entit a vrácení jejich identifikátorů, což je typicky realizováno obyčejným dotazem s podmínkou na shodu identifikátoru reprezentujícího vztah.
   - `RemedyAsync`, ve které odvozené třídy implementují opravu nalezené nekonzistence, což je ve většině případů ekvivalentní příkazu na odstranění nalezené entity[1].

2. V případě, že při běhu metod zmíněných v předchozím bodě dojde k chybě nebo že je nalezena alespoň jedna nekonzistence, inkrementuje transakce celočíselnou proměnnou *RerunRequiredTimes*, která je sdílená napříč celou transakcí.

Poslední dílčí aktivitou v transakci bývá vždy aktivita `RepeatIfRequiredActivity` taktéž z jmenného prostoru `Barista.Consistency.Activities`, jejíž účelem je naplánování opakovaného běhu konzistenční transakce, spočívající v sekvenčním vyhodnocení následující trojice pravidel:

[1] Výjimkou je aktivita odpovědná za rušení reprezentace vztahu „prodejem vyskladňuje“ mezi entitními množinami `SkladováPoložka` a `Nabídka`. Kardinalita vztahu pro nadřazenou entitní množinu je 0/1. Vztah je tedy rušen vynulováním identifikátoru skladové položky u nabídky, který vztah reprezentuje v transformovaném schématu.

- Platí-li, že $RerunRequiredTimes > 0$, je opakovaný běh transakce naplánován na čas $t + t_{\text{drivejsi}}$.
- Platí-li, že $RerunRequiredTimes = 0$ a zároveň $t < t_{\text{udalost}} + t_{\max}$, je opakovaný běh transakce naplánován na čas $t + t_{\text{pozdější}}$.
- Jinak platí, že $RerunRequiredTimes = 0$ a $t \geq t_{\text{udalost}} + t_{\max}$. Opakovaný běh transakce již není naplánován.

Kde:

- $RerunRequiredTimes$ je hodnota již zmíněné celočíselné proměnné sdílené napříč každým během konzistenční transakce.
- t je aktuální čas.
- t_{udalost} je čas vzniku události odstranění entity z nadřazené entitní množiny, která způsobila běh konzistenční transakce.
- t_{drivejsi} je časový interval používaný pro plánování opětovného běhu konzistenčních transakcí, pokud při stávajícím došlo k chybě nebo byla nalezena a opravena alespoň jedna nekonzistence. Jeho hodnotu je možné měnit v konfiguračním souboru služby, výchozí hodnota je 10 sekund.
- t_{pozdejsi} je časový interval používaný pro plánování, proběhla-li transakce v pořádku a nenalezla-li ani jednu nekonzistenci. Jeho hodnotu je opět možné měnit v konfiguračním souboru služby, výchozí hodnota je 60 minut.
- $t_{\max}$ je časový interval znázorňující maximální dobu, po kterou jsou plánovány opětovné běhy konzistenční transakce pomocí t_{pozdejsi}.

Koordinace kaskádovitého vytváření

V průběhu implementace jádra vytvářeného systému byly zjištěny dvě další operace, pro jejichž realizaci bylo nutné koordinovat transakci napříč více datových služeb. Obě entitní množiny `MístoProdeje` a `ÚčetníSkupina` využívají vazbu na entitní množinu `OprávněnostUživatele` a to prostřednictvím specializovaných entit `OprávněnostvMP` a `OprávněnostvUS` v tomto pořadí. Po vytvoření nadřazených entitních množin je nutné spolehlivě zaznamenat oprávněnost uživatele tak, aby v případě přechodného výpadku nedošlo k vytvoření entity, ke které není znám žádný oprávněný uživatel. Takový stav porušuje požadavky uvedené v kapitole 2.3, které stanoví, že právě jeden uživatel musí být oprávněn v roli vlastníka.

Narozdíl od nekonzistencí způsobených odstraněním a ošetřovaných službou `Barista.Consistency` je vytvoření účetní skupiny či místa prodeje operací, u které je žádoucí sledovat její průběh. Lze očekávat, že uživatel, který entitu vytváří, s ní zamýšlí provádět další úkony. S těmi však nemůže pokračovat dříve, než bude entita vytvořena a než k ní bude mít přidělena oprávnění.

Tento typ konzistenčních transakcí koordinuje v systému služba `Barista.Operations`. Implementuje obslužné třídy pro příkazy `IHandleCreationOfAccountingGroup` a `IHandleCreationOfPointOfSale`, jejichž výsledkem je sledovací číslo spuštěné distribuované transakce zodpovědné za popsané dílčí úkoly, které je branou REST API poskytnuto klientovi ve zvláštní hlavičce odpovědi HTTP `X-Operation`. Služba dále poskytuje koncový bod HTTP API, na kterém je pomocí sledovacího čísla možné zjistit stav transakce – zda již byla dokončena a zda proběhla v pořádku, nebo došlo k chybě.

Ostatní operace pro zajištění konzistence

V souladu se systémovými požadavky v bodě 9 kapitoly 2.3 umožňuje systém automatické rušení nákupů, které nebyly místem prodeje potvrzeny v rámci stanoveného časového intervalu. Tato funkcionalita byla realizována ve službě `Barista.Consistency`, neboť se jedná o konzistenční omezení modelované domény – „v systému nesmí existovat prodej, který byl vytvořen před $t_{interval}$ a doposud nebyl potvrzen“. V této službě bylo rovněž možné využít jejího mechanismu plánování událostí popsaného v předchozí sekci o koordinaci kaskádovitého odstraňování.

Realizace funkcionality spočívá ve dvojici integračních událostí, jednoho příkazu a jejich obslužných tříd:

- Obslužná třída integrační události oznamující vytvoření prodeje v systému při svém spuštění naplánuje událost vypršení potvrzovacího časového limitu prodeje na čas $t + t_{interval}$, kde $t_{interval}$ je tento časový limit, jež je možné změnit v konfiguračním souboru služby a jehož výchozí hodnota je 30 sekund.
- Obslužná třída integrační události oznamující vypršení potvrzovacího časového limitu prodeje při svém spuštění zkontroluje stav prodeje a v případě, že se nenachází v koncovém stavu (viz diagram 2.1), zasílá příkaz pro zrušení prodeje.
- Obslužná třída příkazu pro zrušení prodeje, umístěná v účetní službě `Barista.Accounting`, opět kontroluje jeho stav a je-li nekoncový, mění stav prodeje na zrušený.

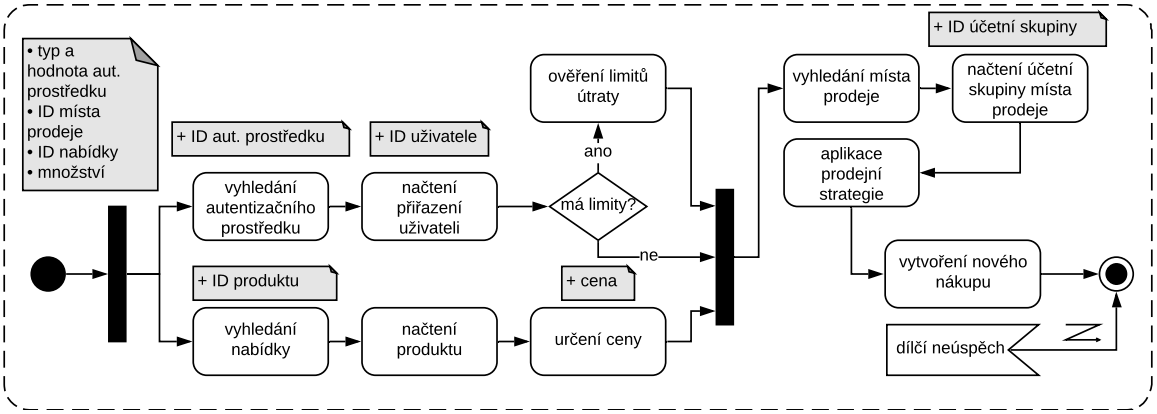
V souladu se systémovými požadavky v bodě 10 kapitoly 2.3 byla ve zmíněné službě vytvořena také obslužná třída reagující na událost změny stavu prodeje na „anulováno“, která dohledává a ruší pohyby na skladě vytvořené v důsledku takových prodejů.

Integrační služba `Barista.Swipe`

Doposud zmíněné datové služby umožňují systému uchovávat informace o modelovaných entitách, dvě další integrační služby poté zabezpečují konzistenci těchto informací. Realizaci očekávaně nejčastějšího případu užití, tedy nákupu produktu v místě prodeje, zabezpečuje služba `Barista.Swipe`. Poskytuje obsluhu pro trojici příkazů, jejichž zasílání je minimální možnou odpovědností softwarového nebo hardwarového zařízení realizujícího místo prodeje:

- Příkaz pro **zpracování nákupu** v místě prodeje, jehož parametry příkazu jsou:
  - Typ a hodnota autentizačního prostředku použitého nakupujícím uživatelem, na jejichž základě systém provádí jeho autentizaci.
  - Identifikátor místa prodeje, ve kterém k prodeji došlo.
  - Identifikátor nabídky, které bylo v místě prodeje využito.
  - Množství produktu, které chce uživatel zakoupit.

  Na příkaz odpovídá systém v případě úspěchu identifikátorem vytvořeného prodeje, které místo prodeje využívá pro oba následující příkazy.
- Příkaz pro **potvrzení uskutečněného nákupu**. Místo prodeje jej zasílá v okamžiku, kdy si je jisté, že výdej zakoupeného produktu proběhl úspěšně, případně ihned po jeho úspěšném zpracování, není-li zjištění úspěchu možné. Parametry příkazu jsou identifikátory místa prodeje a nákupu.



Obrázek 4.5: Průběh obsluhy zpracování nákupu znázorněný pomocí diagramu aktivity. V poznámkách jsou uvedeny informace získávané jednotlivými aktivitami. V případě neúspěchu kterékoliv z aktivit je v obslužném kódu vyvolána výjimka, která jeho činnost ukončuje, což je v diagramu znázorněno obalujícím přerušitelným regionem a přijímačem signálu „dílčí neúspěch“.

- Příkaz pro **zrušení uskutečněného nákupu** a vrácení zablokovaných finančních prostředků zpět uživateli. Místo prodeje jej může zaslat v případě, že detekuje chybu výdeje zakoupeného produktu. Parametry příkazu jsou opět identifikátor místa prodeje a nákupu.

Obsluha příkazu pro zpracování nákupu byla implementována jako několik paralelně běžících operací, znázorněných v obrázku 4.5, zajišťující kromě autentizace uživatele a autorizace platby také získání údajů potřebných k vytvoření prodeje. Využitím Task Parallel Library poskytované rámcovým prostředím .NET spolu s funkcionalitou asynchronních metod jazyka C# je zpracování neblokujícího charakteru a při první detekované chybě je klientovi vrácena informace o neúspěchu.

Implementací zpracování nákupu v podobě distribuované transakce využívající návrhový vzor *routing slip* by se vzhledem k sekvenčnímu zpracování značně prodloužila potřebná doba obsluhy. V případě návrhového vzoru sága by bylo použití konstruktu stavového automatu zbytečně složité. Získaná výhoda možnosti sledování průběhu zpracování by reálně nebyla využita, neboť je očekáváno, že k dokončení zpracování nákupu — ať už s kladným, nebo negativním výsledkem — se očekává v rámci několika sekund.

Služba brány HTTP API

V souladu se zadáním je jádrem aplikace poskytováno REST API sloužící jako zdroj dat pro webovou aplikaci a rozhraní pro integraci nejen aplikací třetích stran, ale také spolu s prací dodávané aplikace pro mikrokontrolér ESP8266EX realizující funkcionalitu místa prodeje. Toho je docíleno definicí veřejného API založeného na autorizaci a provolávání požadavků do ostatních datových a integračních služeb.

K autentizaci a autorizaci uživatelů jsou využívány JWT, jejichž výdej a ověřování brána zajišťuje. Do tokenu je službou ukládán identifikátor autentizačního prostředku použitého k přihlášení, dále pak identifikátor místa prodeje, jde-li o jeho sezení, nebo identifikátor uživatele a hodnota jeho příznaku správce, jde-li o sezení uživatelské. Pro zdroje zpřístup-

ňované na jednotlivých koncových bodech REST API je uplatňován alespoň jeden ze dvou způsobů autentizace popsaných v [35]:

- Autentizace založená na koncovém bodě, vyžadující konkrétní typ sezení (např. koncové body zpřístupňující integrační službu pro zprostředkování nákupů vyžadují sezení místa prodeje) nebo příznak správce (zobrazení úplných informací o uživatelském účtu).

  Pro tento způsob autentizace byla využita infrastruktura ASP.NET Core spočívající v dekoraci celých kontrolerů nebo jeho konkrétních metod představujících jednotlivé koncové body REST API k tomu určenými atributy jazyka C#. Jejich přítomnost při zpracování požadavku uvádí v činnost tzv. autentizační filtry, které v případě zjištění neautorizovaného přístupu přeruší zpracování požadavku a odpovídají chybovou zprávou a odpovídajícím stavovým kódem HTTP 401.

- Autentizace založená na konkrétní entitě, kterou si klient od koncového bodu vyžádal. Uživatel může být např. oprávněným uživatelem k účetní skupině A, manipulace s účetní skupinou B mu však musí být odepřena, přestože se jedná o stejný typ koncového bodu.

  Kontrolery ASP.NET Core vyžadující tento způsob autentizace vkládají závislost na rozhraní ze jmenného prostoru `ResourceAuthorization.Loaders` sestavení `Barista.-Api`, které umožňují načítání úrovně oprávnění uživatele pro konkrétní entity. Jednotlivé metody realizující funkcionalitu koncových bodů poté využívají autentizačních zásad k ověření, že je načtená úroveň dostačující k prováděné operaci. Tyto zásady v současnosti kopírují možné úrovně oprávnění uživatelů k systémovým entitám, první z nich je proto zásadou vlastníka entity a druhá zásadou oprávněného uživatele.

  Pro načtení úrovně oprávnění uživatele je proveden dotaz na příslušnou entitu. V případě, že je bránou prováděna operace nad neexistující entitou, skončí tento dotaz s odpovídající chybou, že entita neexistuje. Z teoretického hlediska by mohl být požadavek odmítnut pro nedostačující oprávnění, jelikož jím klient skutečně nedisponuje. Prakticky jsou chyby zaznamenané při tomto způsobu autentizace předávány klientovi, aby se tak mohl dozvědět, že např. zadal neznámý identifikátor entity.

V bráně REST API jsou definovány veřejné modely dat a dotazů určené k použití třetími stranami. Do této podoby jsou informace získané od datových služeb transformovány stejným způsobem, jako v obslužných třídách dotazové vrstvy — pomocí knihovny tříd AutoMapper. Rozhraní dat požadovaných od klientů na koncových bodech sloužících jako vstupní body příkazové vrstvy jsou branou definovány taktéž. Tato dodatečná úroveň abstrakce je důležitá a výhodná, protože z hlediska vnějšího rozhraní systému není žádoucí prozrazovat jeho vnitřní uspořádání a protože dovoluje provedení vnitřních změn bez ovlivnění vnějšího rozhraní brány API.

Vzhledem k tomu, že se jedná o třídy implementující rozhraní příkazů, obsahují shodné vlastnosti jazyka C#. Koncové body však z bezpečnostních důvodů nastavují hodnoty některých z nich s tím, že jejich případné uživatelem dodané hodnoty jsou přepsány. Toto lze obhájit třeba na příkazu vytvoření entity oprávněného uživatele v účetní skupině, jehož parametry jsou identifikátor účetní skupiny, identifikátor uživatele a požadovaná úroveň oprávnění. Vstupní bod příkazu lze využít zasláním HTTP požadavku s metodou `POST` na koncový bod REST API v umístění `api/accountingGroups/{idSkupiny}/authorizedUsers`. Metoda kontroleru realizující tento koncový bod využívá autentizace založené na konkrétní

entitě, jejíž identifikátor je získán z adresy URL jako `{idSkupiny}`. Uživateli proto nemůže být dovoleno vytvářet záznamy oprávnění pro jinou účetní skupinu, než pro jakou byl požadavek autentizován.

Na koncovém bodě REST API `/swagger` je dostupná dynamicky generovaná dokumentace všech poskytovaných koncových bodů využitím stejnojmenné knihovny tříd.

Služba brány MQTT API

Operace prováděné systémem ve spolupráci s integračním vybavením míst prodeje jsou v systému považovány za asynchronní, neboť mohou být tyto prostředky nedostupné v důsledku výpadku nebo by doba potřebná k jejich provedení činila synchronní přístup nepraktickým, jak již bylo zmíněno v podkapitole o CQRS. V souladu s předpokladem uvedeným v kapitole 3.3, sekci QEST API, bylo při implementaci vybráno a realizováno vhodné řešení spočívající právě ve službě brány MQTT API, která prostřednictvím message brokeru přijímá a vysílá zprávy od a pro integrační prostředky míst prodeje.

Prostřednictvím brány rozhraní REST API je možné zapisovat dvojice klíč-hodnota reprezentující stav místa prodeje. Provádění těchto operací je umožněno rovněž bránou MQTT API. Jedná se tedy o realizaci varianty 1 ze zmíněné sekce o QEST brokeru s tím, že hodnota je vždy načítána z databáze a chybět může jen před jejím prvotním publikováním.

Brána MQTT API umožňuje tedy systémovým službám zasílat zprávy integračním prostředkům míst prodeje a těmto umožňuje publikovat informace o stavu integrovaných zařízení. Pro případné využití ostatních funkcionalit musí být využito REST API.

Centralizovaný sběr diagnostických záznamů

Nejen v průběhu vývoje bylo potřeba sledovat diagnostický výstup jednotlivých systémových služeb. I v produkčním prostředí je důležité mít možnost do tohoto výstupu nahlédnout, statistiky nad kategoriemi zaznamenaných zpráv navíc mohou i ve zdánlivě funkčním systému odhalit přítomnost chyb, které se pouze neprojevily na jeho vnějším chování.

Vzhledem k podpoře kontejnerizovaného nasazení bylo vhodné vyhledat způsob centralizovaného sběru těchto záznamů, aby jejich prohlížení nezahrnovalo nutnost přístupu do hostitelského systému, na němž kontejner běží, a lokalizaci kontejneru s příslušnou službou. V systému je pro tento účel využito nástroje Seq, dostupného i v podobě obrazu Docker, poskytujícího HTTP API pro příjem diagnostických zpráv. Do systémových služeb je integrován prostřednictvím dostupného balíčku služby NuGeT obsahujícího implementaci rozhraní `ILogger`, využívané celou platformou ASP.NET Core. Kromě zpráv pocházejících ze systému jsou tak centralizovaně sbírány i zprávy o zpracování příchozích požadavků a o neošetřených výjimkách, ke kterým ve službě došlo.

4.2 Webová aplikace pro správu

Ačkoliv by webovou aplikaci pro správu bylo možné vytvořit např. s využitím jazyka PHP nebo jiného, který zpracovává obsluhu požadavků na straně webového serveru, jeví se toto poněkud redundantně. Jádro aplikace již obsahuje všechnu funkcionalitu potřebnou k dotazování a manipulaci se systémovými daty. Vytvořená serverová aplikace by tak příchozí požadavky pouze přeposílala jádru a odpověď na nich zpět uživateli. Ke koordinaci po-

žadavků HTTP však lze využít obyčejný webový prohlížeč a v něm podporovaný jazyk JavaScript.

Webová aplikace byla proto vytvořena jako Single Page Application. K jejímu vývoji bylo využito rámcové prostředí Vue.js určené k vývoji webových uživatelských rozhraní tvořených z hierarchicky uspořádaných samostatných celků zvaných komponenty, z nichž se celkové rozhraní nakonec skládá [47]. Seznam vytvořených komponent v podstatě kopíruje seznam veřejných datových modelů definovaných branou REST API, kdy byla vytvořena komponenta pro generování požadavku na stránkovaný výsledek a jeho zobrazení, komponenty pro stručné a detailní zobrazení jednotlivých entit a formuláře pro jejich úpravu.

Komunikaci s jádrem aplikace zajišťuje knihovna tříd Axios, poskytující jednodušší rozhraní klienta HTTP, než je k dispozici v objektu `XMLHttpRequest`. Uživatelé jsou autentizováni pomocí JWT, který je branou REST API na žádost webové aplikace uložen jako soubor cookie s příznakem HTTP–only, což znemožňuje jeho čtení z JavaScriptu a podporuje jeho bezpečnost. K výběru komponent a částí API, které se mají zobrazit, využívá webová aplikace koncového bodu brány REST API `/policies/me`, obsahujícího seznam pojmenovaných autorizačních zásad splňovaných přihlášeným uživatelem.

4.3 Integrace hardware pro autentizaci uživatelů

Jako referenční řešení integrace místa prodeje do vytvářeného systému bylo nakonec převzato řešení vytvořená v rámci projektu do předmětu PDS na FIT VUT v Brně[2] představující novou verzi existujícího řešení popsaného v kapitole 2.1. Řídící program pro mikrokontrolér ESP8266EX zde využívá přítomnosti message brokeru dosažitelného protokolem MQTT, což použitý RabbitMQ umožňuje povolením zásuvného modulu. Vstup a výstup řídícího programu zde obsluhuje služba brány MQTT API popsaná v kapitole 4.1.

Integrace kávovaru probíhá napojením mikrokontroléru na jeho elektroniku. Řídící program má tak prostřednictvím vstupních pinů k dispozici stav LED signalizujících potřebu doplnění zásobníku na vodu a informaci, zda je kávovar zaneprázdněn, nebo je možné zahájit výdej kávy. Prostřednictvím reléového modulu napojeného na výstupní piny pak program dokáže kávovar zapínat a vypínat, zahajovat jeho čištění nebo vydávat kávu.

K autentizaci uživatelů je využita čtečka RFID karet MFRC522, komunikující s řídícím programem prostřednictvím SPI.

Řídící program je do značné míry zapsán deklarativně, což umožnilo použité rámcové prostředí ESPHome založené na PlatformIO popsaném v kapitole 3.4, vytvořené právě k usnadnění tvorby řídících programů především pro mikrokontroléry ESP8266EX a ESP32. Několik stručných programů v jazyce C pouze definuje chování komponent realizujících komunikaci se čtečkou karet a správnou interpretaci úrovní vstupních pinů napojených na řídící elektroniku kávovaru.

[2] https://github.com/nesfit/CoffeeMachineAutomation/tree/master/mcu/

Kapitola 5

Uvedení do provozu a testování

Aplikace byla do testovacího provozu nasazována průběžně spolu s implementaci jednotlivých systémových služeb a součástí. Jejich vývoj taktéž doplňovalo vytváření jednotkových testů ověřujících správné využívání programových závislostí, které jsou stručně popsány v následující kapitole. Dále je popsán zvolený způsob integračního testování celého systému. Místa s možným výskytem výpadků jsou identifikována v předposlední podkapitole, spolu s jejich navrženým a aplikovaným řešením. Závěrem této kapitoly je zhodnoceno splnění požadavků stanovených na začátku této práce.

5.1 Nasazení s využitím kontejnerizace a možnosti škálování

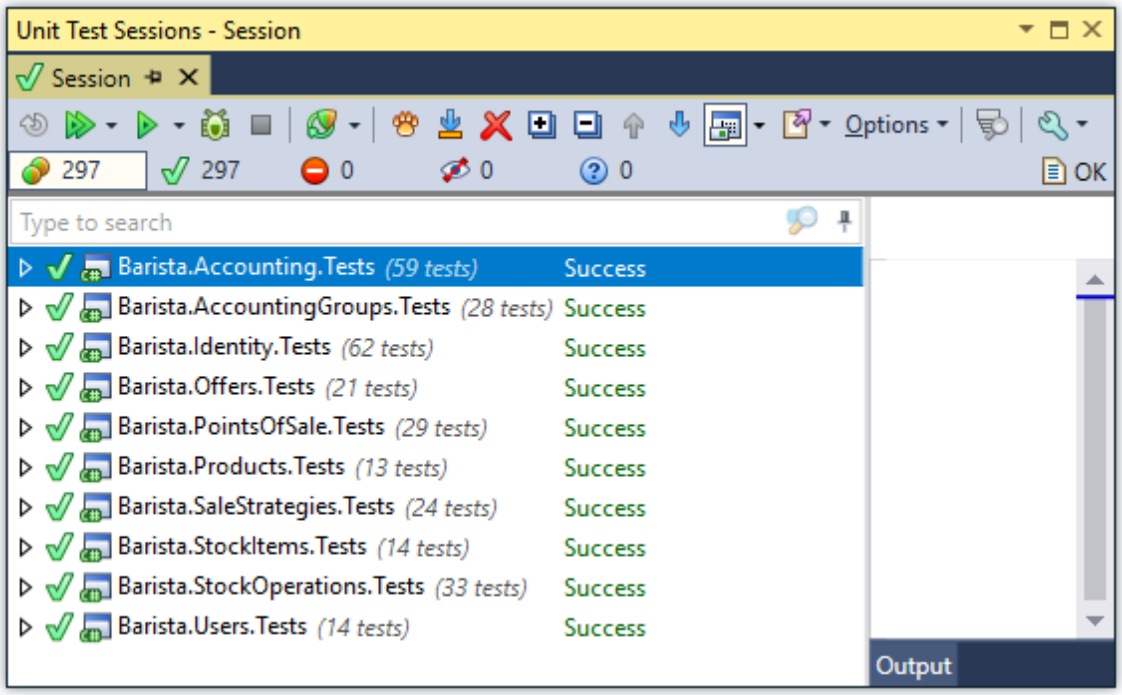
Systém byl vzhledem k zadání práce od začátku vyvíjen se zamýšlenou podporou nasazení v kontejnerech Docker. To se v průběhu vývoje promítlo zejména na výběr použitých technologií, protože všechny tuto podporu sdílí a nebylo tak nutné vytvářet vlastní obrazy Docker ani řešit případné z toho vyplývající potíže. Ve vytvářeném systému je k dispozici konfigurační soubor `docker-compose.yml`, jehož prostřednictvím je po upravení konfigurace služeb možné bezobslužné vytvoření a spuštění všech potřebných obrazů Docker.

Datové služby vnitřně využívají SŘBD MariaDB podporující rozložení zátěže na více instancí databázového serveru jedním z nativně dostupných typů databázových úložišť [14]. Nad použitým SŘBD však aplikace využívá dvou abstrakčních vrstev popsaných v kapitole 4.1 — návrhového vzoru repozitář a dále použitého řešení ORM, tedy Entity Framework Core. Lze jej tedy transparentně zaměnit za jiný, existuje-li pro něj kompatibilní adaptér, nebo změnit implementace systémových repozitářů, aby namísto Entity Framework Core využívaly jiný požadovaný SŘBD.

Systémové služby byly vytvářeny tak, aby neuchovávaly stav. Ve spojení s vhodným způsobem pojmenování směrovačů a front použitého message brokeru a vnitřním HTTP proxy serverem umožňuje bezstavovost služeb provoz libovolného množství jejich souběžně spuštěných instancí a rozložení zátěže mezi ně bez nutnosti dodatečné konfigurace.

5.2 Jednotkové testy

V průběhu vývoje systémových služeb byly k vytvářeným třídám vytvářeny jednotkové testy, spočívající ve vložení falešných implementací rozhraní vkládaných jako závislosti a kontrole, k volání jakých metod v jednotlivých testovaných případech dochází, přičemž



Obrázek 5.1: Snímek obrazovky z uživatelského rozhraní doplňku ReSharper umožňujícího v tomto případě spouštění vytvořených jednotkových testů a zobrazování jejich výsledků.

byly tyto případy vytvářeny podle vnitřní struktury testovaných tříd. K vytváření falešných implementací byla využita knihovna tříd Moq.

Vytvořená sada jednotkových testů pokrývá ve všech systémových službách obslužné třídy příkazů, u kterých ověřuje manipulaci s požadovanou entitou v repozitáři způsobem odpovídajícím účelu příkazu, stav ukládané entity, publikaci integrační události a její obsah. Bylo rovněž dosaženo pokrytí implementací dotazů obsažených v metodě předepsané rozhraním `IQueryImpl`, ověřujících, zda parametry dotazu aplikují správné filtry na zdroj dat. Poslední oblastí pokrytou jednotkovými testy jsou kontrolery ASP.NET Core, u nichž je ověřováno, že na základě dat příchozích součástí HTTP požadavku správně vytváří instance z příkazové a dotazové vrstvy, že tyto instance předává ke zpracování a že návratovou hodnotu správně reprezentuje zpět klientovi.

Jednotkové testy lze spouštět pomocí nástrojů příkazové řádky nebo lze využít uživatelského rozhraní dostupného ve Visual Studiu nebo některém z jeho doplňků, jak je vidět na obrázku 5.1.

5.3 Testy REST API

Hlavní systémové rozhraní poskytované bránou REST API je ověřováno sadou integračních testů, které jej ověřují v testovacím nasazení. Nakládají s ním jako s černou skříňkou, která umožňuje správu modelovaných entit a vhodným způsobem mění svůj stav na základě příchozích požadavků. Kromě již zmíněných oblastí pokrytých jednotkovými testy ověřují testy API i veškerou použitou infrastrukturu a její nastavení — od ASP.NET Core, přes

message broker, MariaDB až po Consul pro registraci služeb a Fabio zprostředkovávající interní HTTP proxy server.

Testovací sada byla vytvořena a je možné ji spouštět nástrojem Postman určeným právě ke konzumaci a testování webových API.

5.4 Opatření pro odolnost vůči výpadkům

V systému existují tři úrovně, ve kterých byla uvažována odolnost vůči výpadkům. První dvě vyplývají z přítomnosti dotazové a příkazové vrstvy, třetí z přítomnosti integračních prostředků míst prodeje, se kterými systém pro plnění svého účelu potřebuje komunikovat. V této sekci jsou dále popsány mechanismy, jakými je odolnosti vůči výpadkům dosaženo.

Dynamické implementace rozhraní dotazové vrstvy zmíněné v kapitole 4.1 využívají nadstavbu nad třídou poskytující výchozí implementaci klienta HTTP, která zachytává odpověď získanou z kontaktovaného serveru. V případě zjištění stavového kódu HTTP udávající chybu vzdáleného serveru (s hodnotou kódu vyšší nebo rovnou 500) je požadavek transparentně vůči uživateli klienta opakován dle nastavení. Pouze v případě neúspěchu všech opakování je o chybě zpraven i klient, který odeslání požadavku HTTP inicioval.

Tímto klientem může být jedna z bran API, které na takovou skutečnost reagují odesláním informace o chybě svému klientovi, nebo obslužná třída dotazové či příkazové vrstvy. Infrastruktura, která jejich spouštění řídí, je rovněž nastavena provádět opakování obsluhy, došlo-li v jejím průběhu k chybě přechodného rázu. Odolnosti vůči dočasným poruchám napomáhá i nastavení message brokeru, který zajišťuje perzistenci zasílaných zpráv, aby byly zachovány v případě jeho pádu. Doručení zpráv je systémovými službami potvrzováno v momentě, kdy je jejich obsluha úspěšně dokončena. Nedoručené zprávy jsou message brokerem zasílány opakovaně, v případě přetrvávajícího neúspěchu jsou zprávy uloženy v chybové frontě, kde je může zobrazit a případně přeposlat správce systému.

Spolehlivost komunikace s místy prodeje je použitou infrastrukturou garantována pouze na hranici brány MQTT API. Spolehlivost příkazů zasílaných systémem do míst prodeje sledují obslužné třídy těchto příkazů aktivním čekáním na předpokládanou odpověď. Spolehlivost informací o stavu zasílaných místy prodeje autonomně spadá do jejich odpovědnosti a může být částečně zajištěna např. použitím spolehlivého síťového protokolu.

5.5 Kontrola splnění systémových požadavků

V kapitole 2 byly zdůvodněny a stanoveny závazné požadavky na vlastnosti vytvářeného systému. Tato kapitola se zaměřuje na jejich rekapitulaci a stručný popis způsobů, jakým byla ve vytvořeném řešení zajištěna jejich shoda.

Podpora stanovených scénářů použití

Vytvořený systém podporuje integraci obou druhů kávovarů uvedených v kapitole 2.2. Pro scénář integrovaného kávovaru bylo poskytnuto referenční řešení, pro scénář neintegrovaného kávovaru by pak řešení bylo vesměs stejné — namísto řízení transakce v reakci na signalizaci řídící logiky kávovaru by pouze byla použita tlačítka nebo jiné ovládací prvky.

Kreditový model byl v systému implementován zavedením prodejních strategií blíže popsaných v bodě 5 kapitoly 2.3.

Scénář automatů se společným účtováním podporuje zavedení účetních strategií popsaných v bodě 4 kapitoly 2.3. Prostřednictvím poskytovaného HTTP API je možné dotazovat

se na místa prodeje zařazená do konkrétních účetních skupin, z nich získávat data o prodejích a tyto dále agregovat.

Funkční požadavky a požadavky na provoz systému

Většina funkčních požadavků stanovených v kapitole 2.3 byla splněna při tvorbě Entity-Relationship diagramu a při jeho transformaci na databázové schéma, což bylo blíže popsáno v sekci Jádro založené na architektuře microservices kapitoly 3.3, neboť stanovovaly výčet informací uchovávaných o jednotlivých modelovaných entitách. Zbývající funkční požadavky byly realizovány následovně, přičemž číslování bodů odpovídá pořadovému číslu ve výčtu funkčních požadavků v kapitole 2.3:

1. Neomezená oprávnění uživatelů s příznakem správce uděluje brána REST API v případě, že k provedení konkrétní činnosti nedostačují uživatelova běžně nastavená oprávnění. Zároveň je o takto provedené elevaci práv odeslána diagnostická zpráva.

   Kontrolu příznaku aktivity uživatelského účtu zajišťuje obslužná třída příkazu pro vyhledání vlastníka autentizačního prostředku, která v takovém případě selže.

2. Vyhledání uplatňované prodejní strategie zajišťuje obslužná třída příkazu pro zpracování prodeje umístšná v integrační službě `Barista.Swipe`.

3. Příznak sdíleného autentizačního prostředku kontroluje taktéž obslužná třída příkazu pro vyhledání vlastníka autentizačního prostředku. Parametrem příkazu je totiž příznak, zda může být přiřazení sdílené. Tento příznak nastavuje dle svých potřeb služba, která provádí autentizaci uživatele.

   V případě přiřazení autentizačního prostředku více než jednomu subjektu, zmíněná obslužná třída rovněž selže. Je tak zajištěno, že autentizační prostředek nemůže být přiřazen více než jednomu subjektu.

7. Vyhledání uplatňované ceny produktu zajišťuje taktéž již zmíněná obslužná třída příkazu pro zpracování prodeje. V případě, že cena není stanovena produktem ani nabídkou, dojde k selhání zpracování nákupu.

9. Anulování nepotvrzených nákupů zajišťuje služba `Barista.Consistency` a bylo blíže popsáno v sekci Ostatní operace pro zajištění konzistence kapitoly 4.1.

10. Rušení pohybů na skladě vytvořených automaticky v návaznosti na potvrzený prodej zajišťuje rovněž služba `Barista.Consistency`.

11. Zobrazování částí uživatelského rozhraní příslušných k oprávnění přihlášeného uživatele umožňuje k tomu určený koncový bod REST API popsaný v sekci Webová aplikace pro správu kapitoly 4.2.

Splnění požadavků na provoz systému vyplývá z implementačních technologií zvolených v kapitole 3.4. Odolnosti vůči výpadkům se věnuje předchozí kapitola 5.4.

Kapitola 6

Závěr

Na základě průzkumu existujících řešení pro automatizovaný výdej a účtování kávy a k tomu určených konzultací bylo na začátku této práce popsáno několik scénářů použití, do kterých bude možné vytvářený systém nasadit a ve kterých jej bude možné využívat bez dalších úprav. Podle nich byly dále definovány funkční a ostatní systémové požadavky a bylo navrženo schéma relační databáze definující systémové entity a vazby mezi nimi.

Ve druhé kapitole byly nejprve pečlivě zváženy faktory ovlivňující návrh samotného systému a jeho prvků vyplývající ze zadání práce. Byla vybrána systémová architektura vhodně splňující stanovené požadavky, dále pak identifikovány problémy, které v důsledku toho vznikly, a popsáno jejich navrhované řešení. Závěrem byly zvoleny komunikační protokoly a technologie odpovídající návrhovým požadavkům, které budou v systému použity.

Ve třetí kapitole jsem navrhl jednotlivé celky vytvářeného systému. Popsal jsem výhody a nevýhody, které ze zvolených architektur a postupů vyplývají, rizika, která přináší, a techniky použité k jejich mitigaci. Vybral jsem technologie a nástroje vhodné k využití při realizaci navržených systémových součástí a svá rozhodnutí zdůvodnil.

Čtvrtá kapitola popsala můj postup při programování jednotlivých součástí systému, návrh jejich vnitřních rozhraní a aplikované techniky. Doplnila dříve identifikovaná rizika o několik dalších objevených až ve fázi implementace a uvedla konkrétní způsoby, jakými bylo zajištěno jejich ošetření. Upravila výčet systémových služeb dříve složený jen ze služeb datových o služby integrační a popsala jejich účel.

Právě vývoj datových služeb považuji za nejnáročnější část této práce. Souběžně s ním totiž vznikala okolní infrastruktura jádra, což vývoj datových služeb zpomalovalo. Programový kód realizující jednotlivé datové služby je navíc téměř totožný, jeho odlišnosti spočívají ve výčtu obsluhovaných entit a atributů. Jeho zápis proto představoval repetitivní činnost vyžadující značnou motivaci.

Poslední kapitola se věnovala možnostem uvedení vytvořeného systému do provozu. Popsala účel a rozsah automatických testů obsažených ve vytvořeném řešení. Vyhodnotila rozsah, v jakém byly splněny úvodem stanovené požadavky.

Věřím, že skutečnosti uvedené v této práci potvrzují, že vytvořené řešení skutečně představuje *proof of concept* systému pro automatizované účtování a výdej plně automatickými prodejními místy, ať už jednotkově či množstevně naceňovaných produktů (podpora jednotkové ceny) nebo v různých prostředích (prodej za kredit nebo na dluh pomocí odlišných prodejních strategií). Oproti zadání byla navíc zavedena podpora pro hierarchické vlastnictví prodejních míst jejich seskupováním do tzv. účetních skupin.

V porovnání s existujícími řešeními zkoumanými v kapitole 2.1 představuje vytvořený systém řešení realizující všechny funkcionality požadované zadáním — účtování plateb

a prodejů, skladovou evidenci a automatizaci výdeje. Poskytovaná REST a MQTT API umožňují integrovat systém s nástroji třetích stran, první z nich představuje také zdroj dat pro účely nejrůznějších analytických nástrojů.

Další práce na vytvořeném systému by mohla spočívat v analýze možného zlepšení výkonu zavedením vyrovnávací paměti do datových a integračních služeb a případná realizace tohoto kroku. Použitá autentizace využívající JWT by rovněž mohla být rozšířena tak, aby umožňovala přihlášení poskytovateli třetí strany bez nutnosti implementace dodatečné integrační služby.

Zkratky

ACID Atomicity, Consistency, Isolation, Durability. 14, 15, 18

AMQP Advanced Message Queuing Protocol. 23

API Application Programming Interface. 4, 6, 12, 13, 14, 18, 20, 23, 25, 28, 29, 31, 34, 36, 37, 38, 39, 41, 42, 43, 44, 48, 49, 56

BASE Basically Available, Soft State, Eventual Consistency. 15, 17, 18

CIL Common Intermediate Language. 22

CLI Common Language Interface. 22

CLR Common Language Runtime. 22

CQRS Command Query Responsibility Segregation. 26, 38

CRUD Create, Replace, Update, Delete. 17, 18

HTTP Hypertext Transfer Protocol. 12, 20, 22, 23, 25, 28, 29, 30, 29, 30, 31, 34, 36, 37, 38, 39, 40, 41, 42, 48

IDE Integrated Development Environment. 24

IoT Internet of Things. 11, 17, 20, 56

IP Internet Protocol. 11, 18, 56

JSON JavaScript Object Notation. 22, 28, 46

JWT JSON Web Token. 22, 36, 39, 45

LED Light-Emitting Diode. 39

LINQ Language Integrated Query. 25, 27

MQTT Message Queuing Telemetry Transport. 4, 20, 23, 25, 38, 39, 42, 44, 48

NFC Near Field Communication. 4

ORM Object-Relational Mapping. 22, 25, 33, 40, 48

OS operační systém. 24

REST Representational State Transfer. 4, 20, 23, 25, 34, 36, 37, 38, 39, 41, 43, 44

RFID Radio Frequency Identification. 3, 4, 6, 7, 8, 18, 20, 39, 48, 49, 56

RPC Remote Procedure Call. 23

SPI Serial Peripheral Interface. 39

SPOF Single Point of Failure. 14, 15

SQL Structured Query Language. 25

SŘBD Systém řízení báze dat. 12, 14, 15, 18, 19, 22, 25, 29, 31, 40

TCP/IP Transmission Control Protocol/Internet Protocol. 11, 23

TLS Transport Layer Security. 23

URI Uniform Resource Identifier. 20

URL Uniform Resource Locator. 22, 30, 37

USB Universal Serial Bus. 3

Slovník pojmů

ADO.NET Abstrakční vrstva platformy .NET pro práci s databázemi [40]. 22

ASP.NET Core Rámcové prostředí pro vývoj webových aplikací na platformě .NET [31]. 28, 30, 31, 37, 38, 41

Autentizační prostředek Posloupnost bajtů, kterou lze spolu s identifikací jejího zdroje použít k autentizaci do systému, viz sekce Funkční požadavky kapitoly 3. 7, 49

Data Transfer Object Model dat přenášených ke klientovi nebo od klienta, založený na interním datovém modelu, avšak uzpůsobený konkrétnímu případu použití a druhu klienta [35]. 28

Docker Platforma pro kontejnerizované nasazování a provoz aplikací, viz sekce Docker kapitoly 3.4. 4, 10, 22, 23, 24, 38, 40, 48

Entity Framework Core Nástroj ORM umožňující vývojářům pracujícím v prostředí .NET Core pracovat s databázovými objekty pomocí objektů .NET [35]. 22, 25, 26, 29, 33, 40

ESP32 Mikrokontrolér vyvinutý společností Espressif Systems [5]. 4, 39, 48

ESP8266EX Mikrokontrolér vyvinutý společností Espressif Systems [4]. 24, 36, 39, 48

ESPHome Nástroj pro vývoj firmwaru pro mikrokontroléry ESP8266EX a ESP32 [46]. 39

Message Broker Software usnadňující integraci systémových komponent komunikujících pomocí zpráv, viz sekce Message Broker kapitoly 3.1. 13, 14, 16, 18, 20, 21, 23, 25, 29, 31, 38, 39, 40, 41, 42, 48, 56

MFRC522 Zařízení k bezkontaktnímu čtení/zápisu RFID karet vyráběné společností NXP Semiconductors [17]. 39

Multi-Drop Bus/Internal Communication Protocol Standardizovaný protokol určený pro komunikaci řídících počítačů prodejních automatů s připojenými periferiemi, jako např. mincovníky [42]. 3, 4, 18, 56

publish–subscribe Komunikační model, ve kterém jsou zprávy namísto konkrétním příjemcům zasílány do pojmenovaných komunikačních kanálů, ze kterých je příjemci, kteří o to stojí, mohou odebírat [35]. 4, 20, 23

QEST Návrh systému pro jednotný přístup ke zdrojům podporujícího protokoly HTTP a MQTT, viz sekce QEST API kapitoly 3.3. 21, 20, 21, 23, 38, 48

Raspberry Pi Jednočipový počítač s procesorem platformy ARM [45]. 4

Single Page Application Webová aplikace, většinu jejíž logiky uživatelského rozhraní zajišťuje programový kód spouštěný ve webovém prohlížeči uživatele, komunikující s webovým serverem zejména prostřednictvím webových API [34]. 39

Task Parallel Library Sada API usnadňující vývojářům využívajícím prostředí .NET programování paralelně běžících úloh [21]. 36

Zdroj autentizačních prostředků Fyzický nebo informační systém poskytující autentizační prostředky, např. čtečka sériového čísla RFID čipu nebo generátor náhodných čísel. 48

Literatura

[1] Akbarov, K. *A SELF SERVICE POS SYSTEM USING RFID AUTHENTICATION*. Tallinn: Tallinn University of Technology, 2017. Diplomová práce. Vedoucí práce Peep-Juhan Ernits. Dostupné na: <https://digi.lib.ttu.ee/i/?7687>.

[2] Cohn, R. *A Comparison of AMQP and MQTT* [online]. 2012 [cit. 30. ledna 2019]. Dostupné na: <https://lists.oasis-open.org/archives/amqp/201202/msg00086/StormMQ_WhitePaper_-_A_Comparison_of_AMQP_and_MQTT.pdf>.

[3] Collina, M., Corazza, G. E. a Vanelli Coralli, A. Introducing the QEST broker: Scaling the IoT by bridging MQTT and REST. In *2012 IEEE 23rd International Symposium on Personal, Indoor and Mobile Radio Communications - (PIMRC)*. Sept 2012. S. 36–41. ISSN 2166-9589.

[4] Espressif Systems. *ESP8266EX Datasheet* [online]. Verze 6.0. 2018 [cit. 29. ledna 2019]. Dostupné na: <https://www.espressif.com/sites/default/files/documentation/0a-esp8266ex_datasheet_en.pdf>.

[5] Espressif Systems. *ESP32 Series Datasheet* [online]. Verze 2.8. 2019 [cit. 29. ledna 2019]. Dostupné na: <https://www.espressif.com/sites/default/files/documentation/esp32_datasheet_en.pdf>.

[6] Fowler, M. *CQRS* [online]. Jul 2011 [cit. 2. května 2019]. Dostupné na: https://martinfowler.com/bliki/CQRS.html.

[7] Fowler, M. *Microservices* [online]. Mar 2014 [cit. 17. ledna 2019]. Dostupné na: <https://www.martinfowler.com/articles/microservices.html>.

[8] Gubbi, J., Buyya, R., Marusic, S. et al. Internet of Things (IoT): A Vision, Architectural Elements, and Future Directions. *Future Generation Computer Systems*. červenec 2012, roč. 29. S. 1645–1660.

[9] Hajdík, T. *Zprovoznění automatizace automatu na vydávání kávy* [online]. Apr 2018 [cit. 29. listopadu 2018]. Dostupné na: <https://github.com/hajdiktomas/PDS-projekt/>.

[10] Hohpe, G. a Woolf, B. *Routing Slip* [online]. 2017 [cit. 24. ledna 2019]. Dostupné na: <https://www.enterpriseintegrationpatterns.com/patterns/messaging/RoutingTable.html>.

[11] Jones and others. *JSON Web Token* [online]. May 2015 [cit. 28. ledna 2019]. Dostupné na: <https://tools.ietf.org/html/rfc7519>.

[12] Kočí, R. a Křena, B. *Analýza a specifikace požadavků*. Sep 2018. Dostupné na: <https://wis.fit.vutbr.cz/FIT/st/cfs.php?file=%2Fcourse%2FIUS-IT%2Flectures%2FIUS2.pdf>.

[13] MAIL & Club Nespresso. *Nespresso Professional Payment Solutions* [online]. 2018 [cit. 17. ledna 2019]. Webová stránka. Dostupné na: <https://www.nespresso.com/pro/cz/cs/pages/payment_solutions>.

[14] MariaDB. *Partitioning Overview — MariaDB* [online]. Jun 2015, 4. dubna 2017 [cit. 13. května 2019]. Dostupné na: <https://mariadb.com/kb/en/library/partitioning-overview/>.

[15] Microsoft Corporation. *What is .NET?* [online]. [cit. 28. ledna 2019]. Dostupné na: <https://dotnet.microsoft.com/learn/dotnet/what-is-dotnet>.

[16] Miller, R. et al. *Overview — EF Core* [online]. 2016 [cit. 29. ledna 2019]. Dostupné na: <https://docs.microsoft.com/en-us/ef/core/>.

[17] NXP Semiconductors. *MFRC522 Product Datasheet* [online]. Verze 3.9. Apr 2016 [cit. 29. ledna 2019]. Dostupné na: <https://www.nxp.com/docs/en/data-sheet/MFRC522.pdf>.

[18] OASIS Open. *MQTT version 3.1.1* [online]. Oct 2014 [cit. 30. ledna 2019]. OASIS standard. Dostupné na: <http://docs.oasis-open.org/mqtt/mqtt/v3.1.1/os/mqtt-v3.1.1-os.html>.

[19] Patterson, C. *Default Topology Conventions — MassTransit* [online]. Jun 2017, 19. dubna 2019 [cit. 13. května 2019]. Dostupné na: <http://masstransit-project.com/MassTransit/understand/default-topology.html>.

[20] Patterson, C., Sellers, D. a Smith, T. *MassTransit Documentation* [online]. Dec 2015 [cit. 29. ledna 2019]. Dostupné na: <https://media.readthedocs.org/pdf/masstransittemp/stable/masstransittemp.pdf>.

[21] Petrusha, R. et al. *Task Parallel Library (TPL)* [online]. 2017 [cit. 13. května 2019]. Dostupné na: <https://docs.microsoft.com/cs-cz/dotnet/standard/parallel-programming/task-parallel-library-tpl>.

[22] Pivotal Software. *.NET/C# RabbitMQ client library — RabbitMQ* [online]. [cit. 29. ledna 2019]. Dostupné na: <https://www.rabbitmq.com/dotnet.html>.

[23] Pivotal Software. *RabbitMQ Tutorials — RabbitMQ* [online]. [cit. 29. ledna 2019]. Dostupné na: <https://www.rabbitmq.com/getstarted.html>.

[24] Pivotal Software. *Reliability Guide — RabbitMQ* [online]. [cit. 29. ledna 2019]. Dostupné na: <https://www.rabbitmq.com/reliability.html>.

[25] Pivotal Software. *Which protocols does RabbitMQ support? — RabbitMQ* [online]. [cit. 29. ledna 2019]. Dostupné na: <https://www.rabbitmq.com/protocols.html>.

[26] PlatformIO. *What is PlatformIO? — PlatformIO 4.0.0a1 documentation* [online]. [cit. 29. ledna 2019]. Dostupné na: <https://docs.platformio.org/en/latest/what-is-platformio.html>.

[27] Rahman, L. F., Ozcelebi, T. a Lukkien, J. J. Choosing Your IoT Programming Framework: Architectural Aspects. In *2016 IEEE 4th International Conference on Future Internet of Things and Cloud (FiCloud)*. Aug 2016. S. 293–300. ISBN 978-1-5090-4052-0.

[28] Richardson, L., Amundsen, M. a Ruby, S. *RESTful Web APIs*. 1. vyd. Sebastopol: O'Reilly Media, Sep 2013. ISBN 978-1-449-35806-8.

[29] Rosa, D. *Saga Pattern* [online]. Jan 2018 [cit. 24. ledna 2019]. Dostupné na: <https://blog.couchbase.com/saga-pattern-implement-business-transactions-using-microservices-part/>.

[30] Rotem Gal Oz, A. *SOA anti-pattern: Transactional Integration* [online]. 2010 [cit. 24. ledna 2019]. Dostupné na: <https://arnon.me/2010/09/soa-antipattern-transactional-integration/>.

[31] Roth, D., Anderson, R. a Luttin, S. *Introduction to ASP.NET Core* [online]. 2019 [cit. 28. ledna 2019]. Dostupné na: <https://docs.microsoft.com/en-us/aspnet/core/?view=aspnetcore-2.2>.

[32] Rychlý, M. a Kolář, D. *NoSQL Databáze* [online]. Oct 2013 [cit. 24. ledna 2019]. Dostupné na: <http://www.fit.vutbr.cz/~rychly/public/docs/slides-nosql-databases/slides-nosql-databases.pdf>.

[33] Schröder, F. et al. *Quickstart – Fabio* [online]. 2016, 18. října 2017 [cit. 13. května 2019]. Dostupné na: <https://github.com/fabiolb/fabio/wiki/Quickstart>.

[34] Smith, S. et al. *Choose Between Traditional Web Apps and Single Page Apps (SPAs)* [online]. 2019 [cit. 13. května 2019]. Dostupné na: <https://docs.microsoft.com/en-us/dotnet/standard/modern-web-apps-azure-architecture/choose-between-traditional-web-and-single-page-apps>.

[35] Torre, C. de la, Wagner, B. a Rousos, M. *.NET Microservices: Architecture for Containerized .NET Applications* [online]. v2.1.03. 2018 [cit. 19. prosince 2018]. Dostupné na: <https://aka.ms/microservicesebook>.

[36] ventopay GmbH. *Mocca.app* [online]. [cit. 17. ledna 2019]. Webová stránka. Dostupné na: <https://ventopay.com/en/mocca/smartphone-application-mocca-app/>.

[37] ventopay GmbH. *Mocca.admin* [online]. Oct 2018 [cit. 17. ledna 2019]. Produktový list. Dostupné na: <https://ventopay.com/wp-content/uploads/2018/08/Product-folder-mocca.admin_.pdf>.

[38] ventopay GmbH. *Mocca.vend* [online]. Oct 2018 [cit. 17. ledna 2019]. Produktový list. Dostupné na: <https://ventopay.com/wp-content/uploads/2018/08/Product-folder-mocca.vend_.pdf>.

[39] Wenzel, M. et al. *What is "managed code"?* [online]. 2016 [cit. 28. ledna 2019]. Dostupné na: <https://docs.microsoft.com/en-us/dotnet/standard/managed-code>.

[40] Wikipedia contributors. *ADO.NET — Wikipedia, The Free Encyclopedia* [online]. 2018 [cit. 29. ledna 2019]. Dostupné na: <https://en.wikipedia.org/w/index.php?title=ADO.NET&oldid=858190230>.

[41] Wikipedia contributors. *Message broker — Wikipedia, The Free Encyclopedia* [online]. 2018 [cit. 24. ledna 2019]. Dostupné na: <https://en.wikipedia.org/w/index.php?title=Message_broker&oldid=876133179>.

[42] Wikipedia contributors. *Multi-Drop Bus / Internal Communication Protocol — Wikipedia, The Free Encyclopedia* [online]. 2018 [cit. 29. ledna 2019]. Dostupné na: <https://en.wikipedia.org/w/index.php?title=Multi-Drop_Bus_/_Internal_Communication_Protocol&oldid=863902279>.

[43] Wikipedia contributors. *X/Open XA — Wikipedia, The Free Encyclopedia* [online]. 2018 [cit. 24. ledna 2019]. Dostupné na: <https://en.wikipedia.org/w/index.php?title=X/Open_XA&oldid=874105655>.

[44] Wikipedia contributors. *MariaDB — Wikipedia, The Free Encyclopedia* [online]. 2019 [cit. 24. ledna 2019]. Dostupné na: <https://en.wikipedia.org/w/index.php?title=MariaDB&oldid=879799025>.

[45] Wikipedie. *Raspberry Pi — Wikipedie: Otevřená encyklopedie*. 2018. [Online; navštíveno 29. 01. 2019]. Dostupné na: <https://cs.wikipedia.org/w/index.php?title=Raspberry_Pi&oldid=16526887>.

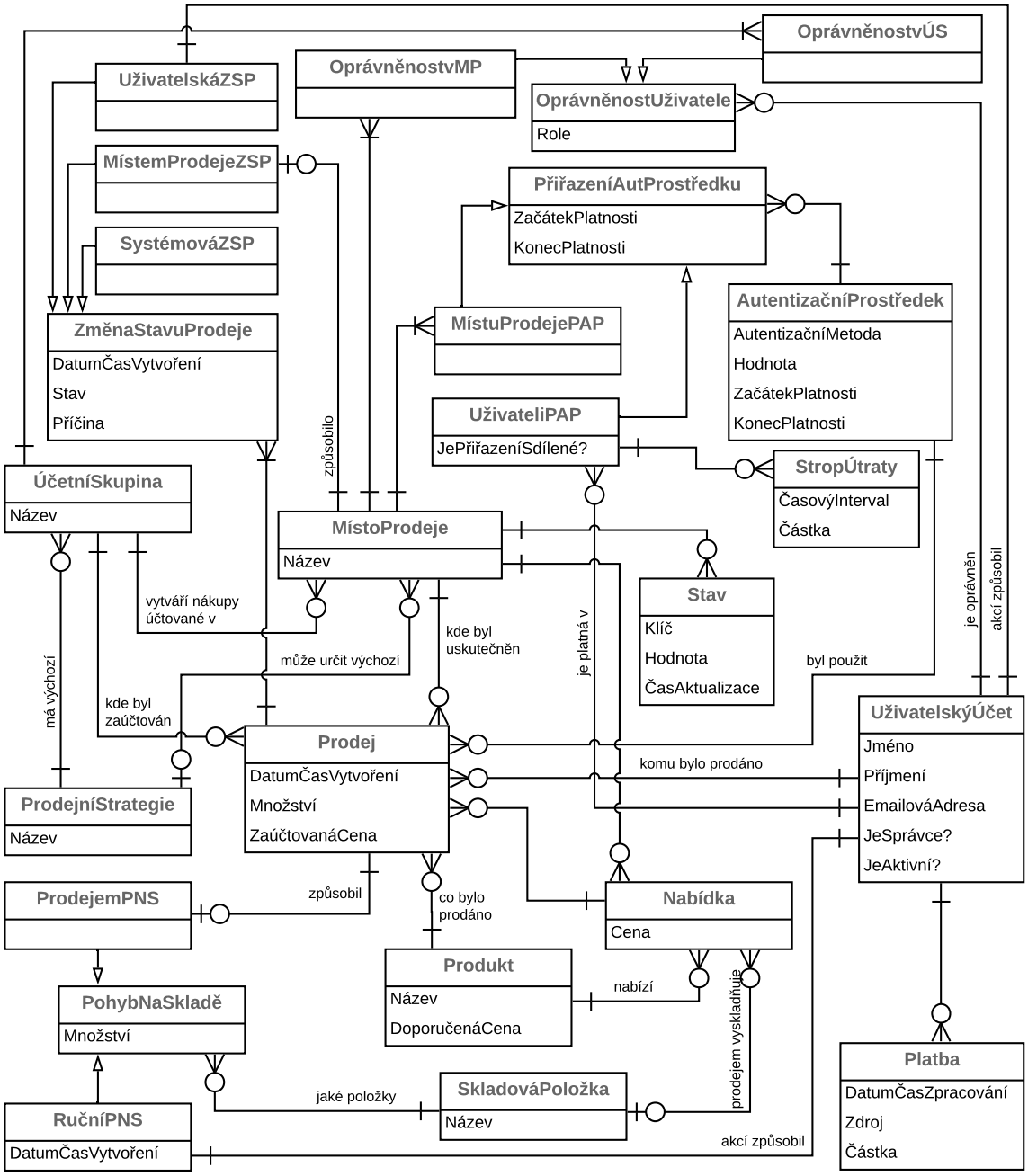
[46] Winter, O. *Getting Started with ESPHome* [online]. Mar 2019 [cit. 13. května 2019]. Dostupné na: <https://esphome.io/guides/getting_started_command_line.html>.

[47] Yuxi Evan You. *Introduction — Vue.js* [online]. Apr 2019 [cit. 13. května 2019]. Dostupné na: <https://vuejs.org/v2/guide/>.

[48] Zendulka, J. a Rudolfová, I. *Studijní opora* [online]. 2006 [cit. 24. ledna 2019]. Dostupné na: <https://wis.fit.vutbr.cz/FIT/st/cfs.php?file=%2Fcourse%2FIDS-IT%2Ftexts%2FIDS_predn.pdf>.

Příloha A

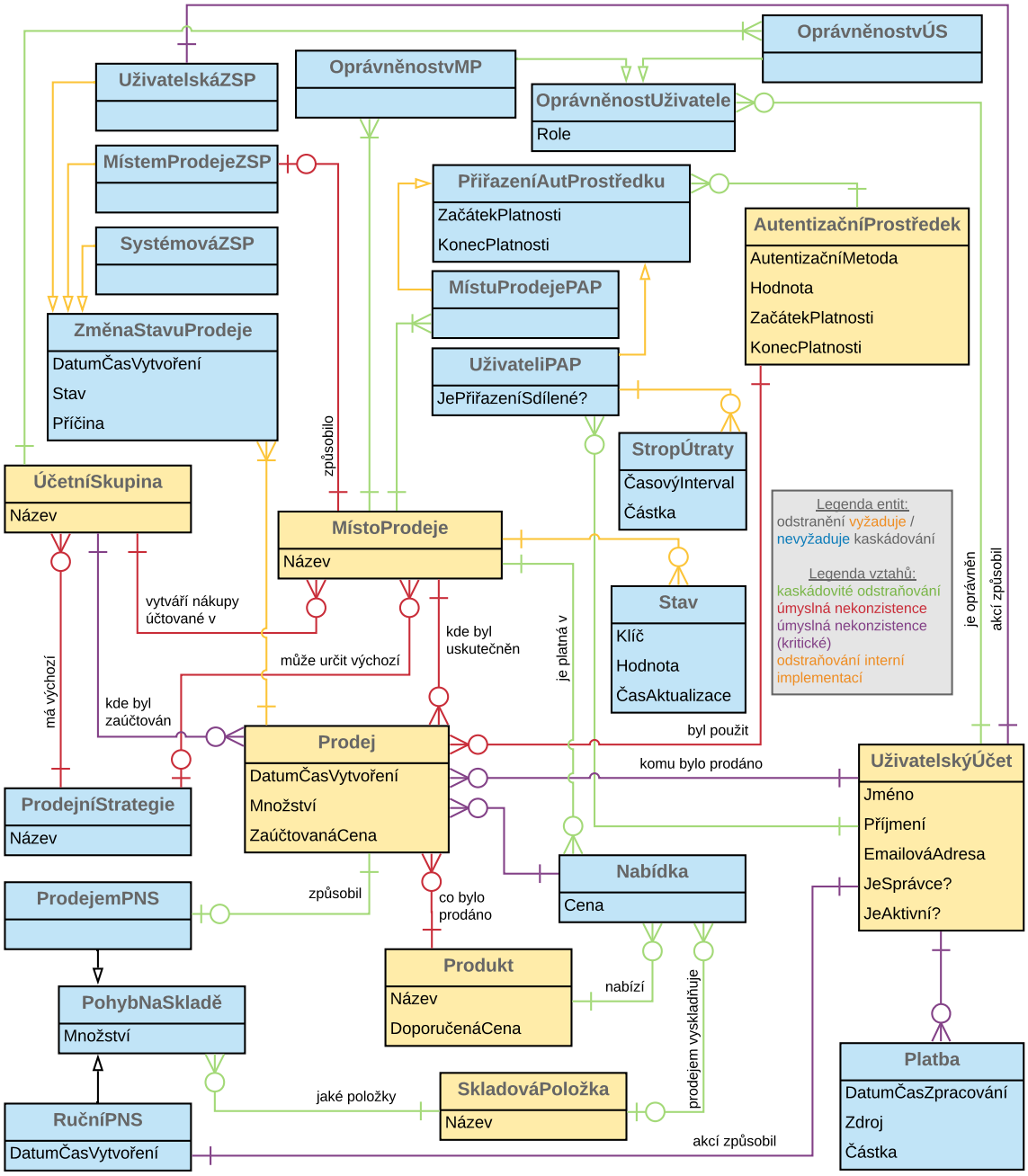
Entity-Relationship diagram



Obrázek A.1: Entity-Relationship diagram vytvářeného systému.

Příloha B

Analýza konzistence dat



Obrázek B.1: Analýza konzistence dat založená na Entity-Relationship diagramu.

Příloha C

Klasifikace IoT prvků systému

Prvek	**Funkcionalita**	**Druh zařízení**	**Třída výpočetního výkonu**
RFID čip	Actuate	T-I	C0
RFID čtečka	Sense	T-S	C1+[a]
Displej	Actuate	E	C1+[a]
Tlačítka	Sense	T-S	C1+[a]
Místo prodeje	Application, Translate	E	C3+[b]
Jádro na bázi *microservices*	Control, Storage, API	F	—
Message Broker	Translate	—	—
Klient protokolu MDB/ICP	Translate, Profile	E	C1+[a]
Zařízení pro IP konektivitu	—	I-S	—

[a] Nejnižší třída výpočetního výkonu schopná minimální IP komunikace [27].
[b] Nejnižší třída výpočetního výkonu schopná IP komunikace a spouštění vlastního řídícího programu [27].

Tabulka C.1: Klasifikace prvků vytvářeného systému podle kritérií specifikovaných [27] tamními tabulkami č. 1, 2 a 3.

Funkcionalita — ***Actuate***: řízení a ovládání hardwarových zařízení; **API**: poskytování API k dostupným funkcím; ***Application***: plná kombinace funkcí na zařízeních, která aplikaci zpřístupňují koncovému uživateli; ***Sense***: obsluha a sběr dat z hardwarových snímačů; ***Control***: určování a nastavování chování prvků IoT systému; ***Profile***: poskytování služeb vycházejících z profilu konkrétního zařízení, a to jak statických (poloha, identita) tak dynamických (stav hardwarových zařízení, zdraví zařízení) Storage: sběr a ukládání dat z připojených zdrojů; ***Translate***: překlad dat mezi dvěma protokoly a směrování dat mezi dvěma sítěmi.

Druh zařízení — **E**: vestavný systém/mikrokontrolér; **F**: serverová farma/cloud; **I-S**: přepínač, zařízení poskytující L2 konektivitu; **T-I**: zařízení umožňující automatickou identifikaci nositele; **T-S**: zařízení fungující jako senzor charakteristické nízkým výpočetním výkonem.

Třída výpočetního výkonu — **C0**: méně než 1 KB nevolatilní paměti, zařízení nedisponuje CPU, např. RFID čipy; **C1**: méně než 64 KB nevolatilní a 2 KB volatilní paměti, 8 bitová architektura, frekvence CPU do 16 MHz, např. Arduino Uno[1]; **C3**: 128-256 KB nevolatilní a 10-128 KB volatilní paměti, 32 bitová architektura, frekvence CPU do 96 MHz, např. MC13224[2].

[1] https://store.arduino.cc/arduino-uno-rev3
[2] https://www.nxp.com/docs/en/data-sheet/MC1322x.pdf