Univerzita Hradec Králové
Fakulta informatiky a managementu
Katedra informatiky a kvantitativních metod

Distribuovaný pokladní EET systém

Diplomová práce

Autor: Martin Zezula
Studijní obor: AI-2

Vedoucí práce: Ing. Pavel Kříž, Ph.D.

Hradec Králové Srpen 2020

Prohlášení:

Prohlašuji, že jsem diplomovou práci zpracoval samostatně a s použitím uvedené literatury.

V Hradci Králové dne 12.8.2020 Martin Zezula

Poděkování:

Děkuji vedoucímu diplomové práce Ing. Pavlovi Křížovi, Ph.D. za metodické vedení práce.

Anotace

Práce popisuje návrh a implementaci distribuovaného EET systému pro maloobchodní použití. Systém se skládá ze systému pro centrální správu a pokladní aplikace. Při návrhu se uplatňují principy distribuovaných systémů, které jsou v práci detailně popsány. Práce také uvádí základní fungování a principy práce s daty otevřené platformy Serenity, která je při vývoji použita. Je vysvětlena právní a technická stránka elektronické evidence tržeb. Implementace demonstruje využití platformy Serenity, principy a synchronizaci dat v distribuovaných systémech a komunikaci s technickým zařízením správce daně za účelem elektronické evidence tržeb. Výsledný systém implementuje alespoň některé funkce, které implementují jiná komplexní EET řešení a je připraven pro další rozšíření funkcionality. Výsledný systém bude nasazen v několika potravinových a hospodských zařízeních.

Annotation

Title: Distributed EET cashier system

This paper describes the design and implementation of distributed EET (electronical evidence of incomes) cashier system for small businesses. The first part of the system is the central management server and the second part is the cashier application. In the design of the system we consider principles of distributed systems which are described in detail in this paper. Basics and principles of open-source platform Serenity are introduced, and are used in the implementation of the system. Both the legal and technical parts of the EET architecture are explained. The implementation of the system demonstrates the usage of Serenity platform, principles and data synchronization in distributed system, and communication with Czech Republic tax administration system for electronical evidence of incomes. The final system implements some of the major functions similar to other complex EET solutions and is prepared for extending functionality. The final system is planned to be deployed in grocery stores and restaurants.

Obsah

Seznam obrázků

1 Úvod

1.1 Cíl práce

Hlavním cílem této diplomové práce je vytvořit pokladní EET systém s podporou synchronizace a vzdálené správy. Po dokončení bude systém nasazen jako náhrada již existujícího systému do několika menších provozoven, především potravin a hospod. Původní EET systém vyvinutý autorem této práce měl velice omezené možnosti práce s daty, malou možnost rozšíření o nové funkce a především postrádal možnost vzdálené správy. Na popud uživatelů původního systému bude navržen nový systém, který vyřeší nedostatky původního systému. Toto je i hlavní motivací této diplomové práce.

1.2 Navrhovaný systém

Výsledný systém umožní majiteli, dále jen správci, spravovat síť jeho prodejen nebo i firem. Bude tedy moci spravovat všechny jednotky jeho firemní struktury. Základní jednotky pro správu budou firmy, prodejny a pokladny. Správce bude moci vytvářet účty pro zaměstnance tak, aby měl přehled o každém jeho zaměstnanci. Uživatel bude mít své vlastní přihlašovací údaje pro přihlášení do systému. Díky tomuto členění bude možné sledovat informace na různých úrovních. Systém tedy bude schopný zobrazit grafy a reporty pro jednotlivé zaměstnance, prodejny nebo i firmy a to pro různá časová období. Přehlednost a usnadnění práce s daty je jeden z nejdůležitějších aspektů navrhovaného systému.
Systém se bude skládat ze dvou základních částí:

1. Pokladní aplikace
2. Centrální systém

Pokladní aplikace bude fungovat jako pokladna na prodejnách a bude podporovat obvyklé funkce pokladního systému. Bude tedy podporovat zadávání tržeb, základní správu a tisk účtenek. Aplikace bude odesílat provedené platby do elektronické evidence tržeb. Centrální systém bude komunikovat se všemi pokladnami, synchronizovat data a umožnovat správu celé organizace daného zákazníka. Systém

na pokladních zařízeních bude ukládat data pouze o dané pokladně a bude synchronizovat data s centrálním serverem. K centrálnímu systému bude mít přístup správce a bude zde mít data synchronizovaná ze všech pokladních zařízení. Centrální systém bude navržen pro více tenantů. Tenanti na sobě budou zcela nezávislí a nebudou o sobě sdílet žádné informace. Každý zcela oddělený zákazník systému bude představovat jednoho tenanta.

Dalším důležitým aspektem systému bude univerzálnost tak, aby měl každý zákazník možnost upravit fungování dle jeho požadavků. Příkladem je formát účtenky pro tisk. Systém by měl být připraven pro možná rozšíření či integraci s ostatními systémy dle možných budoucích požadavků zákazníků.

Využití takového systému je myšleno především pro majitele více menších provozoven. Majitel potřebuje získat důležité informace o prodeji a tyto informace využít k plánování zásobování či sledování trendů a vývoje jeho podniku. Příkladem jsou grafy a reporty, například tržeb za různá období. Bez takového systému majitel musí manuálně získat data z jednotlivých prodejen. Následné získání pro něj důležitých informací je velmi obtížné a časově náročné. Systém navržený v této práci by měl tuto práci zastat, bude totiž efektivně pracovat s daty a umožní je filtrovat, efektivně v nich vyhledávat a zobrazovat reporty.

1.3 Synchronizace a distribuované systémy

Tato kapitola uvádí problematiku distribuovaných systémů, tak jak ji popisuje [1].

Počítačové sítě jsou dnes přítomny všude. Internet je složen z mnoha menších sítí. Mobilní sítě, korporátní sítě, kampusové sítě, domácí sítě. Všechny z nich dohromady sdílí charakteristiky jež je dělají relevantními subjekty distribuovaných systémů.

Distribuovaný systém je dle definice jak ji popisuje [1] takový systém, ve kterém komponenty na počítačích připojených k síti komunikují a koordinují své akce pouze pomocí předávání zpráv. Tato definice pokrývá celou škálu systémů, ve kterých jsou většinou nasazeny počítače připojené k síti. Jak [1] dále popisuje, hlavní motivací pro vývoj a použití distribuovaných systémů je sdílení zdrojů. Výraz „zdroj" je docela abstraktní, ale nejlépe vystihuje škálu věcí, které mohou být většinou sdíleny v zasíťovaném počítačovém systému. To zahrnuje hardwarové komponenty

jako například disky a tiskárny až po softwarové entity jako soubory, databáze a datové objekty všech druhů. Z definice lze vyvodit hned několik následků:

Souběžnost: Komponenty distribuovaného systému běží zároveň. Je třeba zajistit koordinaci jejich akcí.

Žádné globální časování: Koordinace akcí komponent distribuovaného systému je většinou závislá na myšlence sdíleného času. Problémem je omezená přesnost synchronizace času mezi komponentami systému.

Nezávislé selhání: Všechny počítačové systémy mohou selhat. Distribuované systémy mohou selhat mnoha způsoby, a to i v důsledku chyby sítě. Další charakteristikou je, že pouze některé komponenty mohou selhat a ostatní komponenty o tom nemusí vědět.

Mnoho služeb, které dnes považujeme za samozřejmé jsou založeny na principech distribuovaných systémů:

Finance a komerce: Příkladem mohou být firmy jako Amazon a eBay a platební technologie jako PayPal nebo i bankovní systémy spolu s internetovým bankovnictvím.

Web: Růst WWW zapříčinil rozvoj vyhledávacích nástrojů jako je Google či Yahoo. Dalším příkladem jsou sociální sítě jako Facebook nebo služby pro sdílení obsahu jako YouTube.

Transport a logistika: S využitím lokačních technologií jako je GPS v navigačních systémech. Moderní automobily samy o sobě můžeme považovat za komplexní distribuované systémy, stejně tak i ostatní dopravní prostředky.

Jak můžeme vidět, distribuované systémy zahrnují mnoho významných technologických vývojů posledních let a tak i pochopení fungování moderního vývoje celkově.

1.4 Alternativy a existující řešení

Pokladních systémů je na trhu opravdu mnoho a faktorů při jejich výběru nespočet. Proto je třeba se při výběru zamyslet nad tím jaké máme na systém požadavky a seznámit se podporovanými funkcemi. Jedna z možností jak si utvořit ucelenou představu jsou srovnání a recenze. Proto tato kapitola čerpá především z [2] a [3].

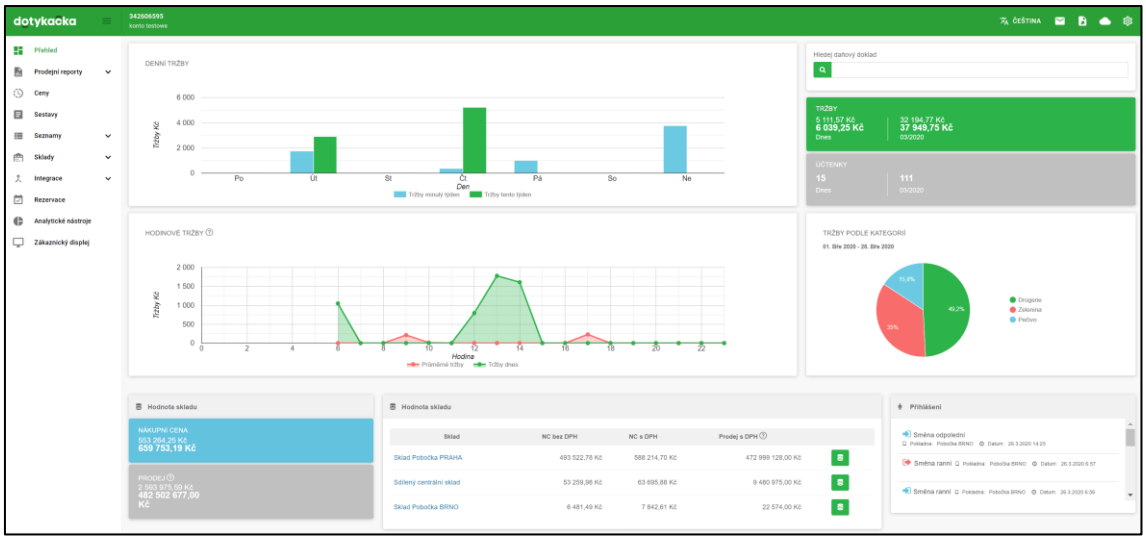
Základem je samozřejmě pokladní systém s podporou elektronické evidence tržeb, a to i v režimu offline. Pokladní systémy musí být schopné zaevidovat platbu i pokud nejsou připojeny k internetu. Důležitým faktorem při výběru systému jsou podporované platformy. Velmi oblíbenou volbou jsou mobilní aplikace pro systémy Android a iOS v kombinaci s chytrým telefonem nebo tabletem. Tato možnost většinou představuje levnější řešení. Pro robustnější využití se většinou doporučují pokladny se zabudovaným systémem. V nabídce jsou jak stolní tak i mobilní EET pokladny. Příkladem může být stolní dotyková pokladna od firmy Dotykačka na Obr. 1.

Obr. 1 Stolní dotyková pokladna Dotykačka
Zdroj: https://www.dotykacka.cz/

Některá zařízení mohou mít i zabudovanou tiskárnu. Mobilní pokladna s tiskárnou je tedy vhodnou volbou především pro restaurační zařízení. Další možností je desktopový klient nebo webová aplikace. Dalším důležitým faktorem při výběru EET systému jsou podporované funkce. Mnoho prodejců nabízí software podporující například skladové hospodářství, kategorizovaný produktový katalog, správu dodavatelů, pomoc s inventurou. Některé softwary také podporují integraci s dalšími pokladními periferiemi, jako například čtečkou čárových kódů či váhou.

Některé funkce lze ocenit především v oblasti gastronomie, jako například mapa stolů, docházkový systém.

Pro tuto práci je zajímavá především možnost vzdálené správy. Mnoho dodavatelů bohužel nenabízí možnost demoverze vzdálené správy.



Obr. 2 Vzdálená správa Dotykačka
Zdroj: https://admin.dotykacka.cz/

Na Obr. 2 můžeme vidět ukázku ve formě webového portálu pro správu od firmy Dotykačka. Hned v úvodním přehledu můžeme vidět grafy a přehledné informace o spravovaném podniku. Vzdálená správa Dotykačka podporuje správu produktů, skladových zásob, dodavatelů, zákazníků, zaměstnanců a rezervací. Podporuje také různé reporty a analytické nástroje.

Dále se již nebudeme zabývat konkrétními řešeními. Funkcí a řešení je opravdu mnoho. V naší implementaci centrálního systému pro vzdálenou správu se pokusíme implementovat alespoň ty nejdůležitější funkcionality.

2 Distribuovaný systém

V této kapitole bude shrnuta základní problematika a principy distribuovaných systémů dle [1],[4] a [5].

Technologie jsou dnes všudypřítomné. Se zvýšením dostupnosti se počet chytrých zařízení se kterými se každý den setkáváme rapidně zvýšil. Rozvoj nezahrnuje pouze použití v korporátní sektoru, ale také použití v osobním životě. S rozvojem se také rozvinul internet a počítačové sítě, na které dnes narazíme v mnoha různých formách. Většina zařízení, která využíváme jsou připojeny k internetu a komunikují se vzdálenými servery za účelem zisku dat. Příkladem můžou být sociální sítě, aplikace pro počasí, multiplayerové hry a cloudová uložiště. Toto je i základní představa k jednoduchému pochopení co výraz distribuovaný systém znamená.

2.1 Definice a základní charakteristiky

Definicí distribuovaného systému je mnoho a každá z nich pokrývá nějakou část celkové oblasti. Některé definice si mezi sebou dokonce odporují. Pro tuto práci bych rád uvedl definici jak je popsána v [1]:

Distribuovaný systém je takový systém, ve kterém komponenty na počítačích připojených k síti komunikují a koordinují své akce pouze pomocí předávání zpráv.

Tato definice pokrývá celou škálu systémů, ve kterých jsou většinou nasazeny počítače připojené k síti. Raději než dále definovat distribuovaný systém se zaměříme na jejich důležité vlastnosti. Jednou z důležitých charakteristik je skutečnost, že rozdíly mezi zařízeními a způsoby jakými mezi sebou komunikují jsou pro uživatele většinou nedůležité. Další charakteristikou je, že distribuované systémy by měli být relativně snadno rozšiřitelné. Tato charakteristika je přímým následkem faktu, že je distribuovaný systém složen z nezávislých komponent. Distribuovaný systém by měl zůstat dostupný i v případě, že nějaká z částí dočasně vypadne. Uživatelé by si neměli povšimnout, když je nějaká část systému nahrazována, nebo přidávána za účelem navýšení celkové kapacity. Vzhledem k vlastnostem distribuovaných systémů vyvstávají v jejich designu problémy a výzvy jež je třeba překonat.

2.2 Vlastnosti

2.2.1 Heterogenita

Internet poskytuje uživatelům přístup ke službám a aplikacím skrz heterogenní skupinu počítačů a sítí. Jak uvádí [1], heterogenita (rozdílnost částí celku) se vztahuje především na:

- Počítačové sítě
- Počítačový hardware
- Operační systémy
- Programovací jazyky
- Různé implementace vývojářů

Přesto, že internet je složen z mnoha různých částí, jejich rozdíly jsou schovány za faktem, že všechny komponenty k nim připojené využívají internetové protokoly ke komunikaci mezi sebou. Například počítač připojený přes Wifi implementuje jiný protokol pro připojení k internetu než počítač připojený přes Ethernet.
Datové typy můžou být reprezentovány jiným způsobem na různém hardware. Jak uvádí [1], například alternativy ve způsobu bitového řazení číselného datového typu. S těmito rozdíly je třeba se vypořádat pokud jsou zprávy zasílány mezi aplikacemi, které běží na různém hardwaru.
Přestože počítačové operační systémy připojené k síti musí implementovat internetové protokoly, nemusí implementovat stejné aplikační rozhraní těchto protokolů. Jak uvádí [1], například volání pro výměnu zpráv v Unixových systémech se liší od volání v systémech Windows.
Programovací jazyky se mohou lišit ve způsobu reprezentace datových typů a struktur. Vývojář musí řešit tyto problémy pokud očekává komunikaci s aplikací vyvinutou v jiném programovacím jazyce.
Aplikace vyvinuté jinými vývojáři spolu nemohou komunikovat, pokud nevyužívají stejné struktury při komunikaci. Za účelem úspěšné společné komunikace je třeba specifikovat společnou množinu struktur pro výměnu zpráv mezi různými implementacemi. Proto je dobré vytvořit dokumentaci, která stanovuje protokol komunikace a struktury k ní potřebné.

2.2.2 Otevřenost

Tato kapitola popisuje rozšiřitelnost a otevřenost systému včetně vývoje třetí strany, tak jak ji popisuje [1].

Otevřenost počítačového systému je charakteristika, která specifikuje zda je možné ho rozšířit nebo upravit. Otevřenost distribuovaných systémů především specifikuje úroveň možnosti přidání nových služeb a klientských aplikací.

Otevřenosti nelze dosáhnout pokud specifikace a dokumentace distribuovaného systému a jeho klíčového rozhraní nebudou přístupny vývojářům, kteří by je použili k rozšíření systému.

Navzdory tomu, publikování dokumentace je pouze první krok k rozšíření systému. Největší výzvou je komplexita systému s mnoha komponentami, který je rozšiřován více nezávislými vývojáři.

Shrnutí otevřenosti sytému dle [1]:

- Klíčová rozhraní systému musí být publikována.
- Otevřené distribuované systémy jsou založeny na poskytnutí jednotného komunikačního mechanismu a rozhraní k přístupu ke zdrojům.
- Otevřené distribuované systémy mohou být konstruovány z heterogenních částí. Každá část musí využívat specifikovaný mechanismus komunikace.

2.2.3 Bezpečnost

Mnoho informačních zdrojů spravovaných a manipulovaných v distribuovaných systémech má vysokou cenu pro jejich uživatele. Bezpečnost těchto informací je tedy velice důležitá. Dle [1], bezpečnost informačních zdrojů se skládá ze 3 komponent:

- Důvěrnost – Přístup pouze oprávněným osobám
- Integrita – Prevence proti úpravě a narušení dat
- Dostupnost – Poskytnutí informací při potřebě

Vzhledem k tomu, že spolu aplikační komponenty distribuovaného systému komunikují skrze internet, vzniká bezpečnostní riziko.

Dle [1] uvedeme příklady, kdy uživatelé zasílají nebo přistupují k datům spravovaným serverem, což znamená zasílání informací skrze síť:

- Lékař přistupuje k nemocničním datům pacientů nebo přidává data nová.
- Při online platbách uživatelé zasílají čísla platebních karet přes internet.

Ve zmíněných případech je důležité zajistit, že je komunikace obsahující citlivá data uskutečněna bezpečným způsobem. Bezpečnost však nezahrnuje pouze šifrování obsahu komunikace, zahrnuje také ověření identity uživatele, který zprávu zasílá. V prvním případě server potřebuje ověřit, že komunikuje opravdu s lékařem, který má k datům přístup. V druhém případě uživatel, jež se chystá sdělit informace o platební kartě, potřebuje ověřit identitu obchodu nebo banky se kterou se potýká. Obě problematiky lze řešit použitím šifrovacích technik při vývoji řešení.
Dalším problémem spojeným s bezpečností je možnost DoS útoků. Útočník, který chce přerušit fungování služby z nějakého důvodu. Toho může útočník dosáhnout nepřetržitým dotazováním na službu, která při jejím vykonávání vyvíjí zátěž na zdroje celého systému. Systém je tedy natolik zatížen požadavky útočníka, že není schopen dostatečně obsloužit požadavky ostatních uživatelů. Prevence takových útoků je většinou možná řešit omezením počtů požadavků.

2.2.4 Škálovatelnost

Počet zařízení, které využíváme každý den se rychle zvyšuje. Škálovatelnost je proto také jeden z nejdůležitějších aspektů při návrhu distribuovaného systému. Distribuované systémy jsou efektivní v malém i velkém měřítku. Systém může fungovat například pouze v malé interní síti, nebo také v celém internetu. Systém lze označit jako škálovatelný pokud zůstane efektivní při výrazném zvýšení zdrojů a počtu jeho uživatelů. Dále je také nutné zmínit administrativní škálovatelnost. Pokud systém začne být masivně využíván, je stále třeba ho efektivně spravovat a řídit. Je požadováno aby se systém a jeho aplikační software nemusel měnit pokud dojde k výraznému navýšení počtu jeho uživatelů. Tato problematika však nemá jednoduché řešení. Jak uvádí [4], škálovatelné systémy se často potýkají se ztrátou výkonu se zvýšením jejich velikosti.

2.2.5 Transparentnost

Dle [1], definujeme transparentnost jako zatajení rozdělení komponent distribuovaného systému uživateli tak, že systém je vnímán raději jako celek než jako množina nezávislých komponent. Vybrané typy transparentnosti, které uvádí [1]:

- *Přístupová transparentnost* – přístup k lokálním i vzdáleným zdrojům identickými operacemi.
- *Lokační transparentnost* – zdroje jsou přístupné bez znalosti jejich fyzického nebo síťového umístění.
- *Konkurenční transparentnost* – více procesů může operovat konkurenčně s využitím sdílených zdrojů bez vzájemného rušení.
- *Mobilní transparentnost* – možnost pohybu zdrojů a uživatelů bez narušení fungování systému a jeho uživatelů.
- *Škálová transparentnost* – systém může být rozšířen bez změny jeho struktury a aplikačních algoritmů.
- *Chybová transparentnost* – uživatelé a aplikační programy dokončí své akce navzdory chybě hardwarové nebo softwarové komponenty.

[1] dále uvádí, že nejdůležitějšími typy jsou přístupová a lokační transparentnost. Jejich přítomnost nebo absence nejvíce ovlivňuje utilizaci distribuovaných zdrojů.

2.3 Komunikace

Komunikace mezi procesy je jednou z nejdůležitějších částí distribuovaných systémů. Samotný význam distribuovaných systémů je zakořeněn v komunikaci procesů na různých počítačích, a i proto je zkoumání možných způsobů jejich komunikace velice důležité. Komunikace v distribuovaných systémech je vždy založena na nízkoúrovňovém předávání zpráv, které nabízí síť skrze kterou komunikují komponenty systému. Řešení takové komunikace může být vcelku náročné, a to především pokud komunikace probíhá přes síť s tak nespolehlivou komunikací jako je internet.

V této kapitole představíme charakteristiky a vybrané základní modely komunikace, které uvádí [4] a [6]

- Předáváním zpráv
- Voláním vzdálené procedury

2.3.1 Charakteristika

[1] uvádí některé důležité charakteristiky mezi procesové komunikace, které je třeba brát v úvahu při návrhu a vývoji distribuovaného systému.

- Synchronnost
- Spolehlivost
- Posloupnost

V synchronní formě komunikace se komunikující procesy synchronizují při každé zprávě. Proces odesílající zprávu čeká na její odpověď. V asynchronní formě komunikace proces odesílající zprávu nečeká na odpověď. Použití jedné či druhé formy má své výhody i nevýhody a jejich použití většinou závisí na okolnostech a potřebách komunikace.
Spolehlivost komunikace odkazuje především na její validitu a integritu. Komunikace může být považována za spolehlivou pokud i přes ztrátu některých dat při komunikaci přes síť je doručená celá zpráva. Integrita představuje především předpoklad, že by zprávy při komunikaci měly dorazit nepoškozené a bez duplikací. Příkladem může být kontrast mezi použitím protokolu UDP a TCP pro komunikaci.
Posloupnost je myšlena především skutečností, že některé aplikace požadují, aby byly zprávy při komunikaci přijaty v pořadí, ve kterém byly odeslány od odesílatele. Pokud nejsou zprávy přijaty v pořadí, ve kterém byly odeslány, typicky to znamená chybu aplikace.

2.3.2 Reprezentace dat

Při komunikaci mezi procesy je třeba také určit jakou formou budou předávaná data reprezentována. Především pokud jsou zasílány komplexní objekty, je třeba aby zvolená reprezentace poskytla všechny potřebné vlastnosti k jejich vyjádření. Po přijetí je třeba převést data zpět do podoby objektů. Jako známé možnosti při vývoji systémů můžeme zmínit například:

- XML
- JSON
- Serializované objekty

- Binární forma

2.3.3 Předávání zpráv

Komunikace pomocí předávání zpráv je základem většiny meziprocesových komunikací v distribuovaných systémech. Tento koncept využívá nejnižší úrovně abstrakce a vyžaduje aby aplikační vývojář věděl jak identifikovat zdrojový a cílový proces, jednotlivé zprávy a datové typy očekávané v těchto zprávách.
[6] uvádí, že paradigma komunikace pomocí předávání zpráv v základní formě využívá operace **send()** a **receive()**. Syntaxe je obvykle následující:

- **send**(příjemce, zpráva)
- **receive**(odesílatel, zpráva)

Operace **send**() vyžaduje jméno přijímacího procesu a data zprávy jako parametry. Operace **receive**() vyžaduje jméno procesu, který zprávu zasílá a poskytuje buffer pro data zprávy.
V systému, který využívá komunikaci pomocí předávání zpráv, je povinností aplikačního vývojáře kontrolovat přesun dat mezi procesy a kontrolovat synchronizaci těchto procesů.

2.3.4 Volání vzdálené procedury (RPC)

Předávání zpráv nutí vývojáře explicitně kontrolovat pohyb dat při komunikaci. Při komunikaci pomocí volání vzdálené procedury, již tato explicitní kontrola není potřeba. Poskytuje také zvýšenou úroveň abstrakce a sémantiku podobnou volání lokální procedury.
Spočívá ve volání procedury ve vzdáleném procesu. Informace jsou přeneseny v parametrech volání a přeneseny zpět jako výsledek procedury. Základní paradigma dle [6] využívá operaci **call**() s následující syntaxí:

- **call** vzdálená procedura(parametry, data odpovědi)

Klientský proces blokuje volání operace **call()** dokud není přijata odpověď.

2.4 Synchronizace a časování

Distribuovaný systém se skládá z kolekce procesů, které mezi sebou komunikují výměnou zpráv. Častou problematikou se kterou se setkáváme při návrhu a vývoji distribuovaného systému je synchronizace dat a řešení možných konfliktů. Čas hraje při synchronizaci důležitou roli. Jelikož více částí distribuovaného systému může v jeden čas nezávisle na sobě měnit tatáž data, vzniká možnost různých konfliktů. K synchronizaci a vyřešení takových konfliktů je potřebné vědět posloupnost úprav tak, abychom mohli určit, která část dat je nejnovější. V distribuovaných systémech postrádáme globální čas a jeho synchronizace není triviálním úkolem. Přesnost času je pro mnoho systémů kritická.

2.4.1 Fyzické hodiny

Většina počítačových zařízení má logické obvody k udržování času. Fungování je obvykle založeno na frekvenci přesně zpracovaných křemenných krystalů. Pod napětím krystal osciluje v definované frekvenci. Přesto, že je frekvence vcelku stabilní, není žádná záruka, že krystaly na jiných zařízeních budou operovat na úplně stejné frekvenci.

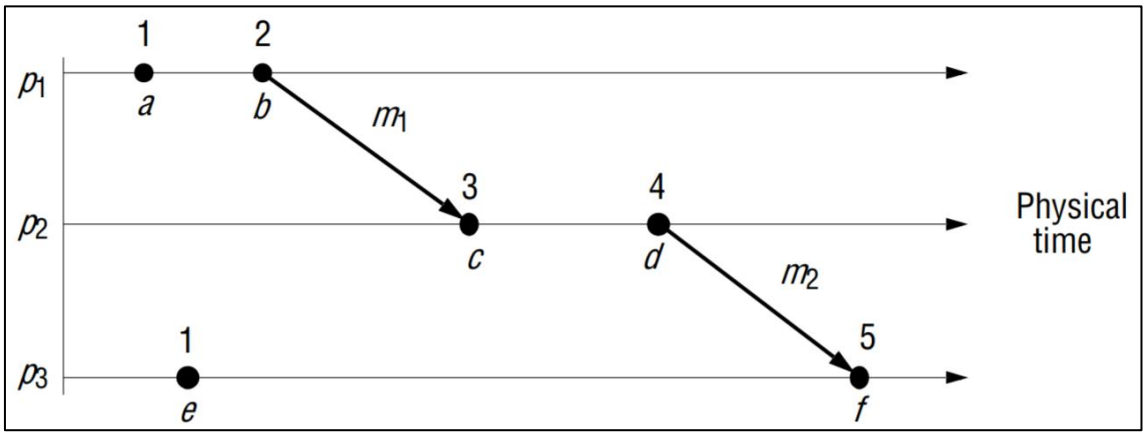
Za těchto podmínek, především v systémech pracujících v reálném čase, způsob synchronizace hodin s reálným časem je náročným úkolem. Eliminace rozdílu frekvence na různých zařízení není možná. Jednou z možností jak synchronizovat hodiny je získat aktuální čas z autoritativního externího zdroje. Toto nazýváme externí synchronizací. K takovéto synchronizaci se nabízí například Network Time Protokol (NTP).

2.4.2 Logické hodiny

V této kapitole představíme logické hodiny, které uvádí [7]. I při použití synchronizace fyzických hodin, jejich přesnost není zcela dokonalá. Při i jen velice malém rozdílu může nastat problém a nebude možné určit posloupnost událostí. K vyřešení tohoto problému můžeme využít logické hodiny, které jsou založeny na událostech raději než na čase. Logické hodiny jsou také nazývány Lamportovi hodiny podle jeho autora. Časovou hodnotu události nazýváme Lamportovo časové razítko.

Předpokládejme, že daný systém je složen z množiny procesů. Každý proces má sekvenci událostí. Zaleží na typu aplikace jakou povahu mají jeho události. V některých systémech to může být úprava části dat, v jiných zase třeba volání procedury. Logické hodiny přiřazují čísla (časová razítka) událostem podle pořadí jak se staly. Procesy aktualizují své hodiny L a přenášejí jejich hodnotu při komunikaci.

- LC1
    - Hodiny L_i je jsou inkrementovány při každé události procesu
    - $L_i = L_i + 1$
- LC2
    - Když proces zasílá zprávu m, zašle také informaci o stavu L_i jako t
    - Když proces přijme zprávu vypočítá $L_j = max(L_j, t)$ a aplikuje LC1



Obr. 3 Příklad fungování logických hodin
Zdroj: [1]

Jak lze pozorovat naObr. 1 Obr. 3, problematika nastává pokud více procesu obsahuje události se stejným časovým razítkem. V takovém případě není možné říci, která z událostí se udála dříve. Pro události $e1$ a $e2$ pak můžeme říci $e1 < e2 \Rightarrow L(e1) < L(e2)$, neplatí přitom však $L(e1) < L(e2) \Rightarrow e1 < e2$.

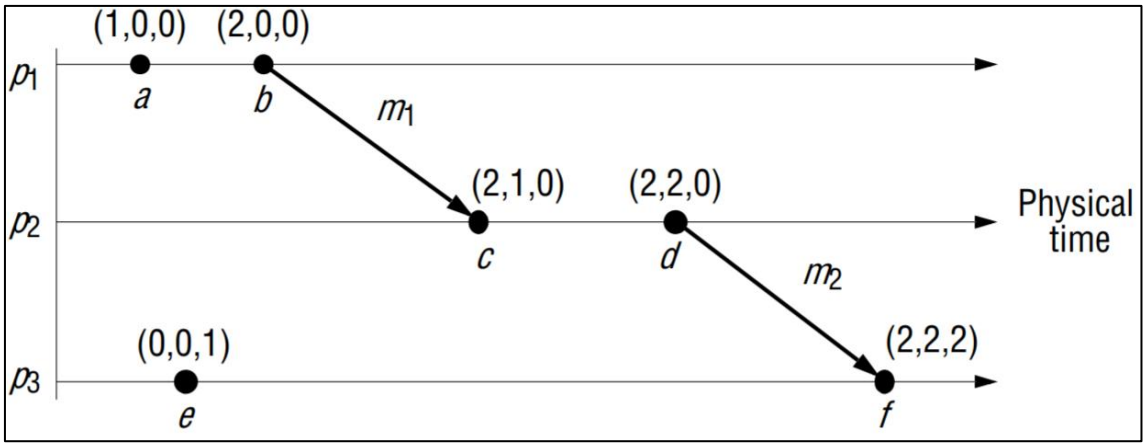
2.4.3 Vektorové hodiny

Vektorové hodiny řeší nedostatky Lamportových hodin zmíněných v předchozí kapitole. Vektorové hodiny pro systém o N procesech je pole N celých čísel. Každý proces udržuje své vlastní hodiny, které používají k razítkování lokálních událostí.

Podobně jako u Lamportových hodin, procesy při komunikaci zasílají hodnotu vektorových hodin. Pravidla pro aktualizaci vektorových hodin:

- VC1: Při každé události procesu p_i je inkrementováno $V_i[i] = V_i[i] + 1$
- VC2: p_i zasílá $t=V_i$ v každé zprávě
- VC3: při přijetí zprávy nastavuje $V_i[j] = max(V_i[j], t[j])$ pro každé $j=1,2,..,N$

Pro vektorové hodiny V_i, $V_i[i]$ je počet událostí, které proces p_i orazítkoval a $V_i[j]$ $(j \neq i)$ je počet událostí, které se staly na p_j a potencionálně ovlivnily p_i.



Obr. 4 Příklad fungování vektorových hodin
Zdroj: [1]

Z Obr. 4 můžeme také vyvozovat, že pro události *e1* a *e2* platí $e1 < e2 \Rightarrow V(e1) < V(e2)$ i $V(e1) < V(e2) \Rightarrow e1 < e2$.

3 Elektronická evidence tržeb

Kapitola se věnuje elektronické evidenci tržeb, známou jako EET, a její technologické stránce. Informace jsou čerpány především ze specifikace, kterou uvádí [8] a zákon č. 112/2016 Sb., o evidenci tržeb (dále jen ZoET).

Jak uvádí [8], elektronická evidence tržeb je převratný systém, který má jako hlavní cíl narovnat podnikatelské prostředí v České republice. Zajišťuje, aby poctivý živnostníci a podnikatelé nebyli znevýhodněni oproti těm, kteří neplatí daně. Obdobné formy evidence tržeb fungují již v přibližně dvaceti evropských zemích.

EET se snaží vytvořit férové prostředí takové, kde mají všichni stejné podmínky a ve kterém plnění povinností neznamená konkurenční nevýhodu. Proto je elektronická evidence tržeb již od počátku zamýšlena jako plošné opatření hotovostních tržeb.

Dalším efektem je lepší výběr daní. [8] také uvádí, že podle Českého statistického úřadu dosahovaly před spuštěním elektronické evidence tržeb nevykázané tržby 160 mld. ročně.

Osobou povinnou evidovat tržby je poplatník

- daně z příjmů fyzických osob a
- daně z příjmů právnických osob.

Pokud takovému subjektu plynou evidované tržby (§ 3 ZoET). Evidovanou tržbou je platba uskutečněná na území České republiky, která splňuje formální náležitosti (§ 5 ZoET) a která zároveň zakládá rozhodný příjem (§ 6 ZoET).

Evidenci tržeb nepodléhají tržby uskutečněné:

- platební kartou
- převodem z účtu na účet
- složením hotovosti na účet v bance
- inkasem
- barterem

Zákon o evidenci tržeb dále v § 12 explicitně vylučuje z evidence tržeb:

- státu

- České národní banky
- příspěvkové organizace
- držitele poštovní licence
- územního samosprávného celku

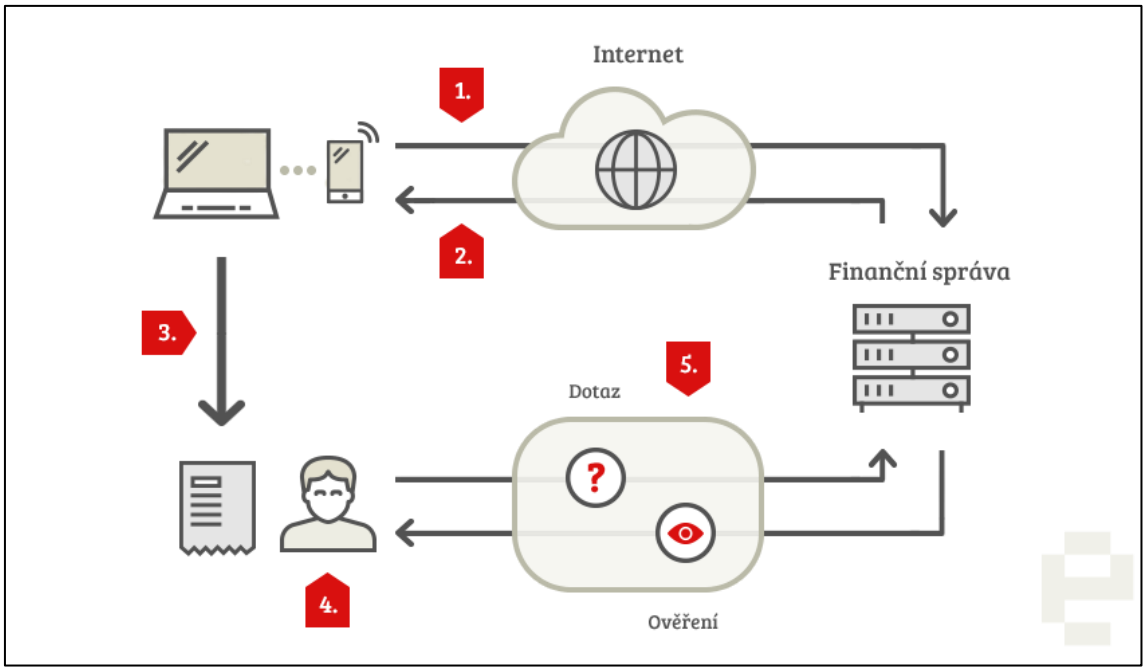
Podnikatelé se do elektronické evidence zapojují postupně ve fázích:

1. fáze - od 1. prosince 2016 - ubytovací a stravovací služby
2. fáze - od 1. března 2017 - maloobchod a velkoobchod
3. a 4. fáze - od 1. ledna 2021 – ostatní činnosti

Ve sbírce zákonů vyšla novela zákona o úpravách v oblasti evidence tržeb v souvislosti s vyhlášením nouzového stavu z důvodu pandemie coronaviru COVID-19. Novela nabývá účinnosti ke dni 3. června 2020. Novela tohoto zákona prodlužuje odklad elektronické evidence tržeb až do konce roku 2020. Povinnost evidovat tržby vzniká všem subjektům k datu 1. ledna 2021.

3.1 Schéma fungování

V [8] se uvádí, že technické řešení vyžaduje zařízení, které dokáže elektronicky komunikovat přes internet (PC, pokladní systém, mobilní telefon, tablet) a připojení k internet pro provedení komunikace. Volbu pokladního zařízení a pokladního software ponechává zákon na uvážení podnikatele. Podnikatel tedy může zvolit řešení odpovídající zvláštnostem jeho podnikatelské činnosti, pokud dané řešení splňuje povinnost odeslat údaje o evidované tržbě datovou zprávu a vydat účtenku. Není tedy nutné pořizovat nijak speciá certifikované registrační pokladny.



Obr. 5 Obecné schéma evidence tržeb
Zdroj: [8]

[8] popisuje obecné schéma fungování EET na Obr. 5. Bod 1 představuje evidenci transakce podnikatele do systému finanční správy ve formě XML zprávy. Na základě této evidence je ze systému finanční správy zasláno potvrzení o přijetí s fiskálním identifikačním kódem (bod 2 na Obr. 5). Bod 3 představuje následné vystavení účtenky s fiskálním kódem podnikatelem zákazníkovi. Zákazník si dále může ověřit svou účtenku na daňovém portále a podnikatel si může ověřit své tržby ve webové aplikaci elektronické evidence tržeb.

3.2 Koncová zařízení

Systém evidence tržeb je koncipován jako otevřený a to z hlediska hardwarového i softwarového řešení. Záměrem tedy není zavádět pokladní aplikace, které by podléhali standardu Ministerstva financí. Stát tedy nebude vyhlašovat žádné výběrové řízení na dodavatele pro koncová pokladní zařízení.

Softwarové i hardwarové řešení je oprávněn nabízet kdokoliv a není k tomu potřeba žádná certifikace výrobce či dodavatele nebo jejich produktů. Stejně tak servisní firmy nepodléhají kontrole či certifikaci. Je na podnikatelích jaký pokladní systém si pro evidenci zvolí. K tomu aby mohl podnikatel evidovat tržby postačí jakékoliv počítačové zařízení, které je schopné elektronicky komunikovat přes internet.

Zařízení musí být tedy technicky vybavené tak, aby bylo schopno odeslat požadované informace o evidované tržbě a vytisknout účtenku s fiskálním identifikátorem zákazníkovi.

3.3 Účtenka

Jak bylo zmíněno v 3.2, koncové zařízení musí být schopné vytisknout účtenku zákazníkovi. Podle § 18 ZoET je poplatník povinen nejpozději při uskutečnění evidované tržby vystavit účtenku tomu, od koho evidovaná platba plyne. § 20 ZoET dále specifikuje údaje, které je poplatník povinen uvádět na účtence:

- Celkovou částku tržby
- Pořadové číslo účtenky
- Datum a čas přijetí tržby
- Označení pokladního zařízení, na kterém je tržba evidována
- Označení provozovny, ve které je tržba uskutečněna
- Daňové identifikační číslo (DIČ)
- Fiskální identifikační kód (FIK)
- Bezpečnostní kód poplatníka (BKP)
- Údaj, zda je tržba evidována v běžném nebo zjednodušeném režimu

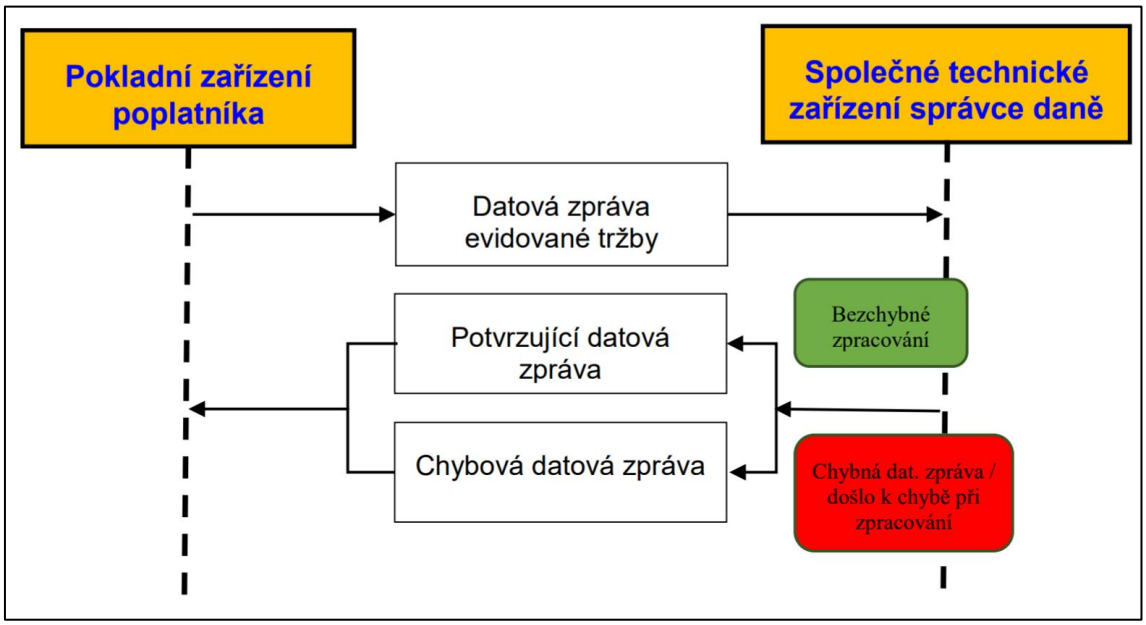
3.4 Komunikace

Pokladní zařízení komunikuje s technický zařízením správce daně ve formě datových zpráv, které obsahují informace o evidované tržbě. Pokud datová zpráva a její informace vyhovují kritickým kontrolám, které provádí technické zařízení správce daně, je na straně technického zařízení správce daně vytvořena potvrzující zpráva a následně zaslána zpět na zařízení, které datovou zprávu zaslalo.

Pokud bude datová zpráva evidující tržbu vyhodnocena jako nevyhovující kritickým kontrolám nebo pokud dojde k chybě při zpracování, technické zařízení správce daně odpoví chybovou hláškou se specifikací povahy chyby, pokud to povaha chyby umožní.

Komunikace mezi pokladním zařízením a technickým zařízením správce daně probíhá tedy stylem: *požadavek/odpověď (request/response)*. Odpověď technického

zařízení správce daně slouží především k potvrzení přijetí a zaevidování zasílané tržby, případně k upozornění na její odmítnutí. Potvrzovací a původní zpráva jsou spolu svázány bezpečnostním kódem poplatníka (BKP) a také číslem datové zprávy, které učil poplatník. Potvrzovací zpráva také slouží k zaslání fiskálního identifikačního kódu, který generuje a eviduje technické zařízení správy daně. FIK je pro každou přijatou a evidovanou tržbu unikátní.



Obr. 6 Scénář komunikace
Zdroj: [8]

3.5 Datové zprávy

Všechny datové zprávy využívané pro komunikaci jsou ve společném datovém formátu, který je daný protokolem SOAP (Simple Object Access Protocol). Datová struktura je tedy ve formátu XML a je vložena do SOAP obálky.

Datová zpráva je SOAP XML struktura obsahující údaje stanovené pro odesílání údajů o evidované tržbě. Data tržby jsou uložena ve vnitřní struktuře v XML elementu *<Tržba>*, která je obsažena těle SOAP zprávy (SOAP body). V hlavičce SOAP zprávy bude uložen XML podpis a certifikát, k němuž patří příslušný privátní klíč, který byl použit pro vytvoření XML podpisu.

Data o evidované tržbě jsou uložena v elementu *<Trzba>*. Tento element obsahuje 3 vnořené elementy: *<Hlavicka>, <Data> a <KontrolniKody>*(Obr. 7).

```
<eet:Trzba>
      <eet:Hlavicka atributy ... />
      <eet:Data atributy ... />
      <eet:KontrolniKody>
            hodnoty ...
      </eet:KontrolniKody>
</eet:Trzba>
```

Obr. 7 Datová zpráva evidované tržby v XML formátu
Zdroj: [8]

Přehled vybraných položek datové zprávy o evidenci tržeb v elementu <*Hlavicka*>:

UUID zprávy (uuid_zpravy)

Univerzální unikátní identifikátor datové zprávy generovaný pokladním zařízením poplatníka jednoznačně identifikuje datovou zprávu. Jedná se o řetězec o délce 36 znaků a má formát dle RFC 4122.

Příklad:

b3a09b52-7c87-4014-a496-4c7a53cf9125

Datum a čas odeslání zprávy (dat_odesl)

Datum a čas odeslání zprávy je okamžik, kdy pokladní zařízení odeslalo datovou zprávu evidované tržby. Formát je určen datovým typem DateTime dle ISO 8601.

Příklad:

2019-05-10T01:38:48+02:00

První zaslání údajů o tržbě (prvni_zaslani)

Jedná se o příznak s hodnotami *true* nebo *false*, který určuje zda se jedná o první zaslání dané evidované tržby.

Mód ověření (overeni)

Příznak označující mód ověření odesílání datových zpráv evidovaných tržeb. Podle hodnoty je zpráva zpracována v ostrém nebo ověřovacím módu.

Přehled vybraných položek datové zprávy o evidenci tržeb v elementu <*Data*>:

- DIČ poplatníka
- Označení provozovny
- Označení pokladního zařízení poplatníka
- Pořadové číslo účtenky
- Datum a čas tržby
- Finanční položky tržby

K finančním položkám patří především informace o částkách. Tedy informace o základu daně, dani a celkové částce pro jednotlivé daňové kategorie.

3.6 Kontrolní kódy

Podpisový kód poplatníka (PKP) je elektronický podpis vybraných údajů evidované tržby. Z technického hlediska, PKP je elektronický podpis textového řetězce složeného z vybraných údajů evidované tržby. Podpis je vytvořen na pokladním zařízení poplatníka pomocí jeho privátního klíče. Privátní klíč tvoří pár s veřejným klíčem. Oba tyto klíče jsou součástí X509 certifikátu, který byl vytvořen finanční správou a je jedinečný pro každého poplatníka. Veřejný klíč je vložen do hlavičky SOAP datové zprávy. K vytvoření PKP a XML podpisu musí být použit tentýž privátní klíč.

Položky vybrané k tvorbě PKP:

- DIČ poplatníka
- Označení provozovny
- Označení pokladního zařízení
- Pořadové číslo účtenky
- Datum a čas platby
- Celková částka tržby

Výpočet PKP začíná ve vytvoření podpisového řetězce z vybraných položek. Podpisový řetězec je vytvořen zřetězením vybraných položek v kódování ASCII s použitím oddělovače „|“ (znak s ASCII dekadickou hodnotou 124) mezi jednotlivými položkami. Příklad takto složeného podpisového řetězce:

```
"CZ72080043|181|00/2535/CN58|0/2482/IE25|2016-12-
07T22:01:00+01:00|87988.00"
```

Z takto vytvořeného řetězce se poté vypočte otisk (hash) algoritmem SHA256, který se následně elektronicky podepíše algoritmem RSASSA-PKCS1- v1_5 podle RFC 3447, s použitím stejného certifikátu a klíče, který bude použit pro podpis celé datové zprávy. Přesná definice způsobu výpočtu PKP je uvedena ve vyhlášce č. 269/2016 o způsobu tvorby podpisového kódu poplatníka a bezpečnostního kódu poplatníka.

Výsledný text je následovně algoritmem Base64 do textového řetězce, který je pak vložen do datové zprávy v elementu <*pkp*>. Výsledný textový řetězec má délku 344 znaků.

Bezpečnostní kód poplatníka (BKP) je otisk hodnoty PKP algoritmem SHA1. Tedy hodnotu BKP lze kdykoliv zkonstruovat pouze při znalosti PKP. Otisk je vytvořen z PKP ve formě řetězce oktetů.

Výpočet bezpečnostního kódu poplatníka začíná vytvořením otisku (hash) řetězce oktetů PKP pomocí algoritmu SHA1. Tím vznikne řetězec oktetů o délce 160 bitů. Tento řetězec je následně zakódován hexadecimálně do textového řetězce, který má 40 hexadecimálních číslic. Textový řetězec je následně upraven do cílového tvaru vložením pomlčky („-“) mezi následující číslice:

- 8. a 9.
- 16. a 17.
- 24. a 25.
- 32. a 33.

Tedy 40 hexadecimálních číslic je rozděleno do pěti bloků po osmi číslicích oddělených pomlčkou. Výsledný text má tedy 44 znaků. Tento výsledný text je pak vložen do datové zprávy v elementu <*bkp*>.

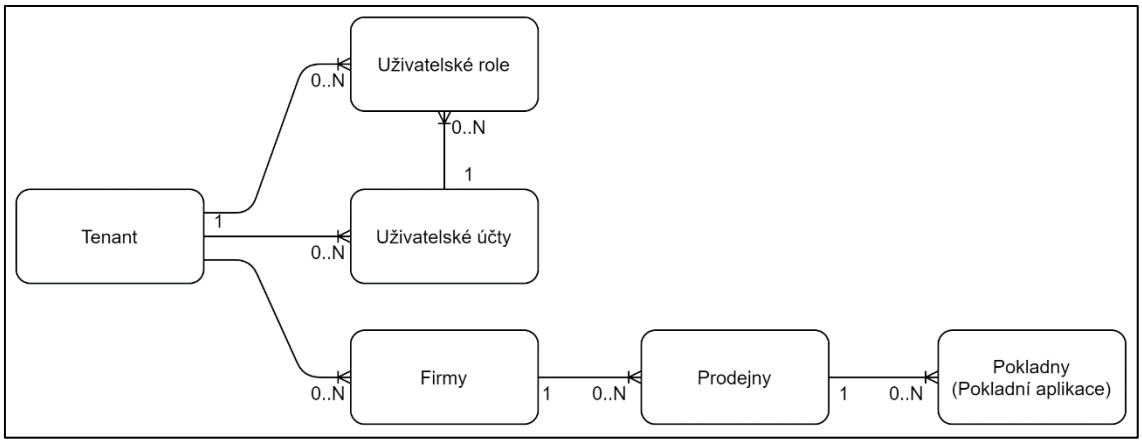
Přesná definice způsobu výpočtu BKP je uvedena ve vyhlášce č. 269/2016 o způsobu tvorby podpisového kódu poplatníka a bezpečnostního kódu poplatníka.

4 Návrh řešení

4.1 Struktura systému

Navrhovaný systém, jak bylo již popsáno v úvodu této práce, se skládá z pokladní aplikace a centrální serverové aplikace. Pokladní aplikace by měla sloužit především pro zadávání plateb a serverová aplikace především pro vzdálenou správu celé struktury. Pokladní aplikace bude synchronizovat data se serverovou aplikací po každé zadané platbě. Komunikace bude tedy stylem klient-server. Serverová aplikace bude navržena jako webová aplikace.

Systém také podporuje více tenantů, kteří jsou na sobě nezávislí. Každý tenant má své vlastní firmy. Firmy obsahují obchody a obchody mají své pokladny. Pokladnu představuje jedno zařízení s naší pokladní aplikací. Tenant má také své vlastní účty a jejich přístupové role. Systém je v základu navržen tak, že centrální serverová aplikace je jedna a bude obsahovat data všech tenantů. Všechny pokladní zařízení se k ní budou připojovat a synchronizovat s ní data. Do centrální serverové aplikace se bude moci vzdáleně přihlásit. Hierarchii navrhovaného systému jak byla popsána můžeme vidět na Obr. 8.



Obr. 8 Hierarchie navrhovaného systému
Zdroj: vlastní zpracování

Pokladní aplikace slouží především k zadávání plateb a následnou synchronizaci se serverovou aplikací, ale měla by být také soběstačná i bez serverové aplikace a měla by umožnit alespoň základní správu účtů a práv. Vzhledem k tomu se množina datových struktur potřebných ukládat na pokladním zařízení podobá množině datových struktur, které bude potřebovat serverová část systému. Správa a

uživatelské rozhraní bude v obou částech obdobné. Vzhledem k těmto skutečnostem jsem se rozhodl navrhnout jednu aplikaci, která bude zastávat obě zmíněné role. Serverová i pokladní část bude tedy ve formě webové aplikace. V konfiguračním souboru bude možné vybrat zda má aplikace fungovat jako pokladní nebo serverová pro vzdálenou správu. Mód, ve kterém bude aplikace spuštěna se bude lišit především v chování a možnostech správy.

4.2 Použité technologie a systémové nároky

Jak již bylo popsané, aplikace bude ve formě webové aplikace. Vzhledem k mým pracovním zkušenostem využijeme pro vývoj ASP.NET Core a platformu Serenity, která bude podrobně popsána v kapitole 5. Pro vývoj v klientské části a rozhraní bude využit jazyk Typescript.

Vzhledem k použití ASP.NET Core bude možné aplikaci využít na operačních systémech (specifikováno v [9]):

- Windows 7 a vyšší (x64, x86)
- Windows Server 2008 R2 a vyšší (x64, x86)
- Linuxové distribuce (Ubuntu, Debian, Red Hat a další) (x64, ARM32, ARM64)
- macOS 10.13 a vyšší

Hlavním cílem však je navrhnout aplikaci pro operační systém Windows. Cílem je, aby navržená aplikace plynule fungovala na i na notebookách nižší cenové třídy. Minimální hardwarové požadavky bude možné určit po dokončení implementace.

4.3 Ukládání dat a databáze

Systém bude ukládat data o tenantech a jejich obchodech, pokladnách a platbách. Jelikož bude třeba se efektivně dotazovat na data, zvolíme k uložení dat relační databázi.

Pokladní aplikace slouží především k zadávání plateb a serverová aplikace především ke vzdálené správě. Pokladní aplikace by však měla být schopná pracovat samostatně a umožnit alespoň základní správu k tomu potřebnou. Jak už bylo zmíněno, obě části systému tedy budou obsahovat obdobné datové struktury.

Systém je multi-tenantní a serverová aplikace bude ukládat data od všech tenantů, tedy všech jeho obchodů a jednotlivých pokladen. S přibývajícím počtem tenantů bude tedy nutné ukládat velké množství dat. Vzhledem k tomu a nutnosti efektivního dotazování na data, se nabízí využití robustních relačních databází. Naopak pokladní aplikace bude obsahovat pouze zlomek dat, a tak by využití robustní relační databáze mohlo být naopak zbytečností a přítěží při instalaci.
Vzhledem k tomu by bylo dobré, aby aplikace měla možnost využít různé možnosti ukládání dat. Pro pokladní aplikaci se například nabízí možnost využití vestavěné databáze SQLite nebo SQL Server Compact.
Možnost využití různých databázových systémů je popsána v kapitole 5.

4.4 Funkční požadavky

Pokladní aplikace bude ve formě webové aplikace. Pro její jednoduché a rychlé ovládání budou implementovány klávesové zkratky. Především při zadávání plateb by všechny úkony měly být možné provést pomocí klávesnice tak, aby byla obsluha co nejrychlejší. Zadávání plateb na pokladních zařízeních je často repetitivní práce, především v potravinách. Proto je třeba operace k tomu potřebné co nejvíce zjednodušit.
Platforma Serenity (kapitola 5), kterou použijeme pro implementaci našeho systému nabízí pro její základní uživatelské rozhraní překlady hned do několika světových jazyků. Čeština však není jedním nich. Bude tedy třeba vytvořit překlad pro Český jazyk. Dále povolíme v systému volbu anglického jazyka.

4.5 Uživatelé a práva

Systém dovolí majiteli sítě prodejen, dále jen správci, vytvářet uživatelské účty pro jeho zaměstnance. Díky tomu bude mít přehled o změnách nebo například o tom, jaký zaměstnanec kolik utržil. Síť prodejen může být rozsáhlá, a tak může být správa rozdělena mezi více managerů, kteří spravují určité prodejny. Pro řízení přístupu uživatelů bude tedy třeba vybudovat systém práv pro různé úkony v systému. Správce bude moci přiřazovat práva jednotlivým uživatelům. Každá jednotka bude označovat zda uživatel může vykonat nějakou akci. Pro zachování maximální

flexibility vytvoříme velký počet práv, pro specifické úkony. Tato práva by měla mít podobnou úroveň abstrakce tak, aby mezi nimi nevznikaly závislosti a hierarchie. Příkladem takových práv může být:

- Vytvoření uživatelského účtu
- Vytvoření prodejny
- Smazání prodejny
- Úprava informací prodejny
- Export dat z prodejny
- Vytvoření platby
- Zobrazení denního souhrnu
- Přihlášení do systému pro správu (serverové aplikace)

Jelikož se očekává veliký počet různých práv, bylo by nepraktické tato práva přidělovat manuálně. Pro zjednodušení přidělování práv uživateli, vytvoříme uživatelské role. Uživatelská role bude mít nastavena jaká práva obsahuje. Příklad uživatelských rolí:

- Pokladník
    - Vytvoření platby
- Manager
    - Export dat z prodejny
    - Zobrazení souhrnů
    - Úprava informací prodejny
- Administrátor
    - Všechna práva

Správci chceme ponechat maximální flexibilitu, a tak bude moci tyto role vytvářet, upravovat a přiřazovat uživatelským účtům svých zaměstnanců. Z toho vyplývá, že i role budou specifická pro každého tenanta v systému.
Jak již bylo zmíněno, síť prodejen může být rozsáhlá, a tak by bylo vhodné také řídit pro jaké prodejny má uživatel práva, například omezit účet pro pokladníka tak, aby měl právo se přihlásit pouze na pokladním zařízení některých prodejen.

4.6 Tisk a účtenka

Dalším požadavkem, který musí splňovat pokladní zařízení, je schopnost vytisknout účtenku. V kapitole 3.3 jsou uvedeny údaje, které je třeba vytisknout na účtenku. Obecnou praxí u pokladních zařízení je využívat řádkové tiskárny. Proto se i náš návrh zaměří především na využití řádkových tiskáren.

Je třeba si uvědomit, že řádkové tiskárny mají omezené možnosti stylování a formátování textu. Jejich odlišnost od klasických stránkových tiskáren je především ve faktu, že tisknou po jednotlivých řádcích, zatímco stránkové tiskárny tisknou pouze celé stránky. Dalším faktem, který je třeba si uvědomit je, že řádkové tiskárny mohou využívat různou šířku papíru. Obvyklá šířka je například 58mm a 80mm. Jednotlivé modely řádkových tiskáren se mohou lišit také v podporovaných funkcích, jako například možnosti tisku čárového kódu, nebo možnosti zaslání signálu pro otevření do pokladního šuplíku. Náš systém bude vyhledávat tiskárny, které jsou nainstalované v systému.

Požadavky na vzhled účtenky se můžou u každého zákazníka lišit. Z tohoto důvodu by bylo dobré navrhnout správu vzhledu účtenky tak, aby si každý uživatel mohl vzhled jeho účtenky změnit. Účtenka však obsahuje údaje o platbě, a tak je třeba vymyslet způsob, jak umožnit úpravy vzhledu účtenky a způsob jakým budou umístěny informace o platbě. Tento problém vyřešíme pomocí šablon. Šablona bude v textovém formátu a bude představovat design účtenky. Text bude obsahovat statický text, který uživatel využije například k oddělení jednotlivých částí a odkazy na informace ze skutečné platby. Příklad takové šablony můžeme vidět na Obr. 9.

```
**********************
   {NazevPoplatnika}
         {DIC}
**********************
FIK: {FIK}
BKP: {BKP}
Účtenka: {cislouctenky}
Datum: {datum}
**********************
Daň                 Cena
{for p in polozky}
{p.Dan}         {p.cena}
{end}
**********************
Celkem          {celkem}
{drawerKick}
```

Obr. 9 Příklad šablony účtenky
Zdroj: Vlastní zpracování

Platba může také obsahovat údaje, kterých je dynamický počet. Vypsání i takových údajů na účtenku je požadováno. Příkladem můžou být jednotlivé položky nákupu a informace o jejich ceně a daňovém zařazení (Obr. 9). Pro zpracování takových údajů bude šablonový nástroj muset podporovat iteraci v jednotlivých položkách. Pro další flexibilitu by bylo dobré implementovat funkce pro usnadnění formátování různých datových typů. Dále by bylo dobré implementovat funkce pro vyvolání speciálních funkcí, jako je například vysunutí šuplíku (*„drawerKick“* Obr. 9).

4.7 Evidence na daňový portál

Jak již bylo popsáno v kapitole 3, pokladní zařízení musí být schopné elektronicky komunikovat s technickým zařízením správce daně.

Evidence tržby je provedena pomocí zaslání datové zprávy s informacemi o tržbě technickému zařízení správce daně. Součástí této datové zprávy jsou i vypočtené bezpečnostní kódy, jejichž specifikace je uvedena v kapitole 3.6. Pro vytvoření těchto bezpečnostních kódů je třeba pracovat s hashovacími funkcemi a s certifikáty poplatníka. Náš systém tedy bude muset umět načíst certifikát poplatníka a použít ho k vytvoření bezpečnostních kódů a k podepsání datové zprávy jako takové.

Datové zprávy jsou ve SOAP XML formátu s danou strukturou. Náš systém tedy bude muset umět tuto strukturu vytvořit a doplnit do ní příslušné informace.

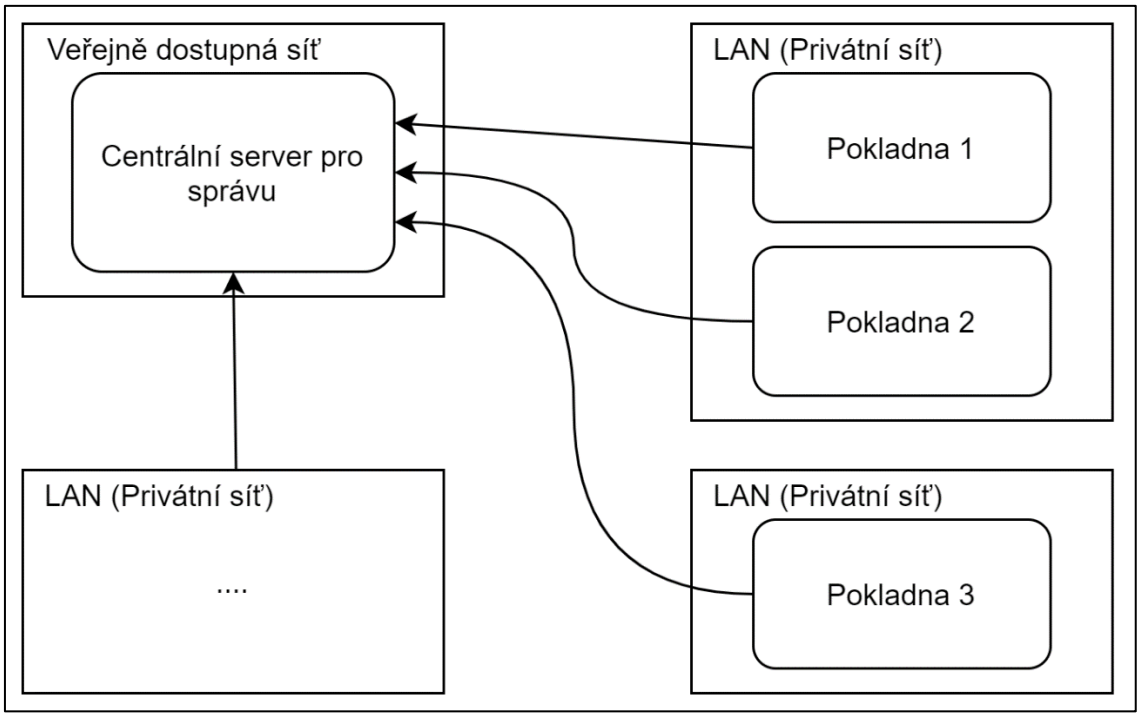
Technické zařízení správce daně zasílá jako odpověď k datové zprávě o evidenci potvrzující datovou zprávu obsahující fiskální identifikační kód, případně informace o chybě. Náš systém tedy bude muset umět číst odpovědi tak, aby mohl efektivně využít informace v nich obsažené.

4.8 Synchronizace

Jednou z důležitých částí našeho systému je synchronizace dat mezi centrálním serverem a pokladními aplikacemi, které představují pokladní zařízení. V kapitole 2 jsme uvedli problematiku s níž musíme při navrhování systému počítat.

4.8.1 Komunikace

Jak bylo zmíněno v kapitole 2.3, máme více způsobů jak uskutečnit komunikaci mezi jednotlivými procesy v distribuovaném systému. V našem systému využijeme způsob volání vzdálené procedury. Tento způsob se přímo nabízí, jelikož bude systém implementován jako webová aplikace. Tedy komunikace bude provedena voláním vzdáleného endpointu. Komunikace bude tedy stylem *request-response*.
Dále je třeba se zamyslet, jakým směrem bude komunikace provedena. Je třeba si uvědomit, že pokladní zařízení budou fungovat v provozovnách jednotlivých zákazníků a budou umístěny na různých fyzických místech (Obr. 10). Možnost připojení směrem od centrálního serveru k pokladnímu zařízení je tedy nepravděpodobné. Pokladní zařízení se tedy budou dotazovat na centrální server a ten jim bude odpovídat (*request-response*).



Obr. 10 Schéma komunikace
Zdroj: vlastní zpracování

4.8.2 ACID

Při synchronizaci je nutné zajistit zachování konzistence dat (anglicky *consistency*). Především pokud budou naše entity obsahovat relace mezi sebou. Dále je důležité, aby se při synchronizaci mezi dvěma zařízeními přenesla data celá nebo vůbec žádná. Tedy chceme zachovat atomicitu synchronizace. Po ukončení synchronizace bychom si měli být jisti, že jsou data skutečně přenesena a že již nemohou být ztraceny. Toto označujeme jako trvalost (anglicky *durability*). Poslední problematikou, kterou je třeba zmínit je izolovanost. Synchronizace s centrálním serverem může probíhat najednou s více aplikačními zařízeními. V tomto případě si musíme být jisti, že jednotlivé synchronizace se mezi sebou neovlivní v průběhu přenosu dat. Čtveřici těchto vlastností dohromady označujeme jako vlastnosti *ACID*:

- Atomicita (Atomicity)
- Konzistence (Consistency)
- Izolovanost (Isolation)
- Trvalost (Durability)

Většina databázových systémů vlastnosti *ACID* implementuje a tak se nabízí přenechání zajištění těchto vlastností přímo na databázovém systému. Tedy provést synchronizaci dat v rámci databázové transakce.

Přijatá novější data budou uložena do databáze v rámci transakce. Problém ale nastává právě v kontrole jejich konzistence, kterou provádí databázový systém. Některé záznamy mohou mít relace (cizí klíče) na záznamy, které ještě nebyly vytvořeny. Proto budeme muset řídit posloupnost uložení záznamů do databáze tak, abychom neporušili její konzistenci.

4.8.3 Časování

Protože navrhovaný systém bude obsahovat data, které bude moci konkurenčně upravovat, je nutné zajistit aby bylo možné určit, která verze je právě aktuální. K řešení tohoto problému využijeme především informace obsažené v kapitole 2.4.

Centrální server pro správu bude obsahovat vcelku všechna data o celém systému a pokladní zařízení bude mít k dispozici pouze data, která jsou pro dané zařízení relevantní. V případě kdy nějaké zařízení v systému upraví data, je nutné určit zda ostatní zařízení již tuto změnu obdrželi.

Podobně jako ve vektorových hodinách zmíněných v kapitole 2.4.3 budeme číslovat události. Událost bude představovat úprava nebo přidání nějakých dat. Příkladem může být třeba vytvoření tržby na pokladním zařízení nebo třeba vytvoření nového uživatele na centrálním serveru pro správu. Každý prvek v systému bude své události číslovat lokálně a bude také udržovat informace o tom, jaké číslo události získal jako poslední od jiných zařízení při synchronizaci. Při synchronizaci mezi dvěma stranami, obě strany vědí jakou získaly poslední aktualizaci od druhé strany. Při synchronizaci se tedy přenášejí pouze opravdu změněná data.

4.8.4 Bezpečnost a přenášené informace

Navrhovaný systém bude uchovávat a přenášet důvěrná data různých zákazníků (tenantů). Bezpečnost navrhovaného systému je tedy velice důležitá. Problematiku shrnuje kapitola 2.2.3.

Při začátku používání našeho systému zákazník nejprve vytvoří organizační struktury, jako firmy a obchody. Dále je ale třeba připojit pokladní zařízení. Při

připojení se bude muset pokladní zařízení nějakým způsobem identifikovat a identifikovat zákazníka (tenanta) k němuž se připojuje. Pro tento účel bude mít zákazník dočasný přístupový kód, který se bude měnit po každém použití. Pokladní zařízení se tedy připojí k centrálnímu serveru a uvede svou identifikaci, identifikaci zákazníka (tenanta) a jeho přístupový kód. Centrální server si uloží identifikaci pokladního zařízení a již nebude vyžadovat přístupový kód zákazníka při další synchronizaci.

Navrhovaný systém bude obsahovat více na sobě nezávislých zákazníků (tenantů), jejichž data by měla být oddělena a v bezpečí proti neautorizovaným úpravám. Při synchronizaci bychom měli kontrolovat, že upravovaná data jsou relevantní pro dané pokladní zařízení a pro tenanta k němuž je dané pokladní zařízení připojeno. Při synchronizaci tedy budeme provádět kontrolu toho, že data pocházejí od daného tenanta a jsou relevantní pro dané pokladní zařízení.

Pokladní zařízení potřebuje pouze informace, které jsou potřeba pro jeho fungování. Například pokladní zařízení nepotřebuje informace o platbách z jiného pokladního zařízení i přes to, že obě zařízení patří jednomu zákazníkovi. Proto při synchronizaci bude centrální systém zasílat aktualizace pouze dat relevantních k danému pokladnímu zařízení.

Při synchronizaci bychom měli zajistit důvěrnost přenášených dat. Proto by spojení mělo být provedeno pomocí protokolu HTTPS.

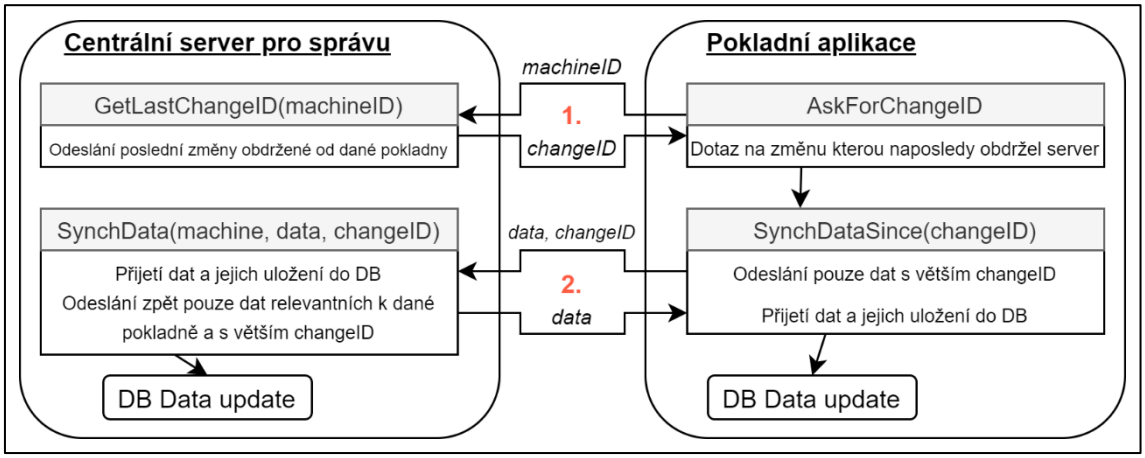
4.8.5 Souhrn

Na Obr. 11 můžeme vidět zjednodušené schéma průběhu synchronizace mezi pokladním zařízením a centrálním serverem pro správu. Schéma je vytvořeno na základě předchozího obsahu této kapitoly (4.8).

Bod 1. na Obr. 11 představuje vyvolání komunikace na straně pokladního zařízení. Pokladní zařízení se dotáže centrálního serveru na poslední změnu (*changeID*), kterou získal od daného pokladního zařízení. Centrální server se na základě identifikace pokladního zařízení podívá na poslední změnu, kterou od něj obdržel a odpoví pokladnímu zařízení.

V 2. bodě pokladní zařízení následně odešle všechna změněná data od poslední změny, kterou obdržel centrální server. Při odeslání také přidá informaci o tom,

jakou poslední změnu (*changeID*) pokladní zařízení obdrželo od centrálního serveru. Centrální server přijme data, zkontroluje zda jsou relevantní pro dané pokladní zařízení a provede aktualizaci v databázi. V této chvíli si také uloží novou hodnotu poslední změny (*changeID*), kterou přijal od daného pokladního zařízení. Následně centrální server odpoví pokladnímu zařízení a zašle mu data změněná od poslední změny, kterou přijala pokladní aplikace. Pokladní data uloží do jeho databáze a uloží si také informaci o poslední změně, kterou přijalo od centrálního serveru. Tímto synchronizace končí.



Obr. 11 Schéma synchronizace
Zdroj: vlastní zpracování

5 Serenity

5.1 Objektově relační mapování - ORM

Reprezentace dat je jedním z důležitých aspektů při vývoji softwaru a práci s daty. Udává nám jak na data nahlížíme a může se lišit v mnoha aspektech. Jak popisuje [10] a [11], každá reprezentace má své limity, jelikož může mít svou vlastní formu abstrakce a implementovat různé organizační principy a vazby. Ve výpočetní technice se setkáváme s mnoha různými reprezentacemi. Každá reprezentace má významný vliv jak na design tak i na samotný vývoj výpočetních systémů. Jak [10] dále popisuje, kombinace technologií založených na různých reprezentacích dat může být problém pro ty, kteří jsou zodpovědní za návrh a vývoj takovýchto výpočetních systémů. Dle [10] na takovýto problém referuje jako na impedanční nesoulad.

Relační reprezentace se osvědčila při vývoji databází, zatímco většina dnes populárních programovacích jazyků pro vývoj komplexních systémů je založena na objektové bázi. Popularita těchto technologií využívajících různé reprezentace dat v základním aspektu vývoje výpočetních systémů znamená, že se nevyhnutelně využívají dohromady. Rozdíl v abstrakci a programovacím jazyku znamená problém při kombinaci těchto technologií.

Objektově relační aplikace kombinuje objekty jak z relačního tak i objektového modelu. Objektově relační aplikace je tedy taková aplikace, která je typicky vyvinuta v objektovém jazyce a využívá relační databázi pro práci s daty. Vývojář takovéto aplikace se tedy při vývoji musí potýkat s již výše zmíněným impedančním nesouladem. Tento problém je v dnešní době dobře znám a jsou dostupná různá řešení a koncepty, které jsou obecně známa jako objektově relační mapování.

V kontextu vývoje objektově relační aplikace je jeden z cílů ORM strategie izolovat vývojáře za použití objektově orientovaného programovacího jazyka od potřeby pochopení SQL jazyka, schématu databáze a její sémantiky. Vývojář se nepotřebuje soustředit na to jak uložit objekty, ale na to co a kdy uložit či načíst.

Taková strategie je typická pro ORM řešení jakou je Hibernate a Oracle TopLink.

Platforma Serenity pracuje s relačními databázemi a provádí mapování dat do objektové reprezentace. Hlavním rozdílem však je, že neizoluje vývojáře od použití

jazyka SQL. Objekty a jejich mapování v platformě Serenity mohou obsahovat i spojení více databázových objektů. Mapování tedy spíše vytváří databázový pohled (view) na data obsažená v databázových tabulkách. Detailní vysvětlení způsobu mapování bude uvedeno v kapitole 5.4.

5.2 Platforma Serenity

Kapitola popisuje fungování jak jej popisuje [12]. Serenity je open source ASP.NET Core MVC platforma, která je vyvinuta na open source technologiích. Pro klientskou stranu využívá jazyku TypeScript, což je nadstavba velice populárního JavaScriptu. Platforma je zaměřena především na zjednodušení vývoje a snížení ceny údržby. Je kladena především snaha na snížení množství kódu, který je vývojář nucen napsat a snížení času stráveného na opakujících se úkonech.

Serenity se nejlépe úplatní při vývoji business aplikací s mnoha typy datových objektů nebo jako administrativní rozhraní. Jejích funkcí lze ale efektivně využít při vývoji i jiných typů aplikací.

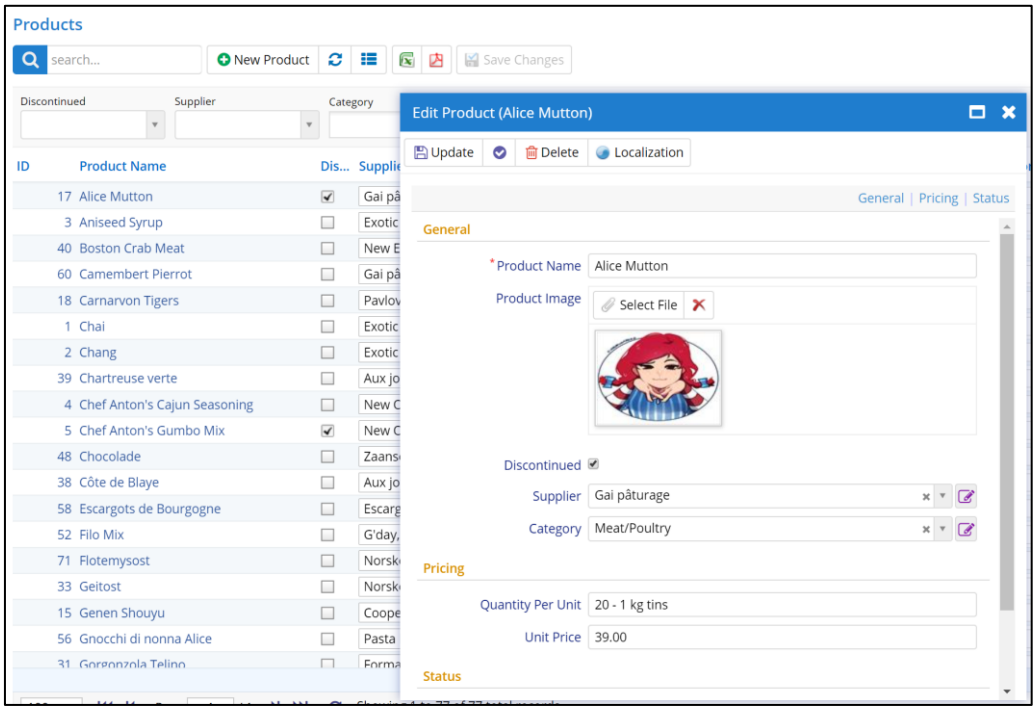
Jak autor [12] popisuje dále jeho motivaci. V aplikacích, které využívají rozsáhlé databázové struktury, které obsahující velký počet entit musí vývojář většinou pro každou z těchto entit vytvořit její objektovou definici a mapování. Dalším opakujícím se úkolem je definice rozhraní pro zobrazení, úpravu a práva pro úpravu daného typu entity. Tyto opakující se úkony a kopírování již existujícího kódu vedou k chybám a zvýšení času pro správu, úpravu a implementaci nové funkcionality. Z těchto problémů vzešel nápad na platformu, která vyřeší tento problém, sníží čas implementace na minimum a připraví pro vývojáře již předpřipravené typy formulářů pro úpravu dat a zobrazení. Důraz je přitom kladen na obecnost a flexibilitu tak, aby vše bylo možné upravit pro specifickou aplikaci. Kompatibilita je také jeden z hlavních záměrů platformy. Nezávislost na typu použité databáze je také zaručena.

5.3 Setup projektu a generátor entit

Pro začátek projektu je dobré využít šablonu projektu SERENE dostupnou ve Visual Studiu. Po vytvoření máme v projektu již vytvořené entity, endpointy, dialogy a datové gridy pro testovací databázi Northwind. Vytvořené jsou již i entity pro správu

uživatelů a jejich práv a překladů zobrazovaných textů. Základní implementace poskytuje překlady základního uživatelského rozhraní do 12 světových jazyků. V inicializaci jsou již nakonfigurovány služby a závislosti pro autentizaci a autorizaci a jsou připraveny funkce pro dvoufaktorovou autentizaci. [12] také uvádí jak implementovat různé typy autentizace, jako například Windows autentizaci. Po vytvoření projektu aplikace vypadá stejně jako samotné demo Serenity dostupné na https://serenity.is/demo/. Projekt již také obsahuje základní ukázky práce s rozhraním, entitami a službami. V konfiguračním souboru jsou nastaveny výchozí připojovací řetězce do databáze. Jako výchozí databáze je nastavena MSSQL Server LocalDB, tedy data jsou ukládána do souboru přímo v adresáři projektu.

Součástí platformy je generátor entit a přidruženého kódu s názvem sergen. Generátor je utilita pro příkazový řádek. Generátor načte připojovací řetězce z konfiguračního souboru a zobrazí všechny databázové tabulky a pohledy pro výběr. Po výběru databázového objektu je možné zvolit název entity a jaké části se pro ni mají vygenerovat. Je možné vygenerovat objektovou entitu, službu s endpointy pro manipulaci s entitou a uživatelské rozhraní, které zahrnuje datový grid pro zobrazení a dialog pro úpravu.



Obr. 12 Datový grid a dialog
Zdroj: vlastní zpracování – Serenity Demo

```
[ConnectionKey("Default"), Module("Administration"), TableName("Users")]
[DisplayName("Users"), InstanceName("User")]
[ReadPermission(PermissionKeys.Security)]
[ModifyPermission(PermissionKeys.Security)]
[LookupScript(Permission = PermissionKeys.Security)]
public sealed class UserRow : LoggingRow, IIdRow, INameRow, IIsActiveRow
{
    [DisplayName("User Id"), Identity]
    public Int32? UserId
    {
        get { return Fields.UserId[this]; }
        set { Fields.UserId[this] = value; }
    }

    [DisplayName("Username"), Size(100), NotNull, QuickSearch, LookupInclude]
    public String Username
    {
        get { return Fields.Username[this]; }
        set { Fields.Username[this] = value; }
    }

    [DisplayName("Password Hash"), Size(86), NotNull, Insertable(false), Updatable(false),
MinSelectLevel(SelectLevel.Never)]
    public String PasswordHash
    {
        get { return Fields.PasswordHash[this]; }
        set { Fields.PasswordHash[this] = value; }
    }

    [DisplayName("Display Name"), Size(100), NotNull, LookupInclude]
    public String DisplayName
    {
        get { return Fields.DisplayName[this]; }
        set { Fields.DisplayName[this] = value; }
    }

    [DisplayName("Email"), Size(100)]
    public String Email
    {
        get { return Fields.Email[this]; }
        set { Fields.Email[this] = value; }
    }
```

Obr. 13 Ukázka vygenerované entity User
Zdroj: vlastní zpracování

Jak lze vidět na Obr. 13 generátor vygeneruje entitu a správně určí datový typ jejích vlastností. Lze si také všimnout, že jsou také vygenerovány položky z tabulek na které má tato entita cizí klíč. Mapování je dosaženo pomocí anotací. Je také možné přidat další anotace pro úpravu chování aplikace:

- Column(name) – Identifikuje sloupec v tabulce v databázi.
- Expression(hodnota) – Definuje SQL výraz.
- NotNull – Při vytvoření a úpravě objektu se zkontroluje, že hodnota není nulová.
- Size(velikost) – Omezuje délku řetězce.

- DisplayName(text) – Nastavuje text pro zobrazení v datovém gridu.
- QuickSearch – Indikuje, že je vlastnost zahrnuta do vyhledávání v datech.
- DefaultValue(hodnota) – Pokud je hodnota při vytvoření nulová, tato hodnota je použita jako výchozí.
- Insertable(bool) – Indikuje zda je možné hodnotu vložit.
- Updatable(bool) – Indikuje zda je možné hodnotu změnit.
- TypEditoru – Lze nastavit editor, který je pak použit při zobrazení v dialogu.
- ForeignKey(tabulka, sloupec) – Specifikuje jakou tabulku a sloupec referuje.
- Left/RitghJoin(alias) – Specifikuje spojení.

Generátor tedy především usnadňuje práci při přidávání nových typů entit do systému. Vzhledem k tomu je čas potřebný k vývoji alespoň základního rozhraní a služeb zcela minimální.

5.4 Mapování a práce s daty

Jak už bylo uvedeno v předchozích podkapitolách, mapování je provedeno pomocí anotací. Pro vytvoření, úpravu a zisk dat jsou připravené objekty, které tyto informace uvedené v anotacích efektivně využívají k složení SQL dotazů. Tyto objekty budeme dále nazývat jako handlery. Díky těmto handlerům je možné provádět velice komplexní SQL dotazy bez potřeby znalosti SQL jazyka. Tyto handlery také zajišťují vynucení práv pro úpravu a čtení dat na základě práv a rolí uživatele. Handlery tedy zprostředkovávají:

- Práva přístupu uživatelů k datům
- Kontroly omezení – NOTNULL, LENGTH
- Vyhledávání – složený WHERE
- Filtrování dat – WHERE
- Řazení dat – ORDER BY
- Stránkování – OFFSET, FECH

Handlery jsou zcela otevřené a vývojář je může upravit dle jeho potřeby. Pro handlery je také možné přidat chování (*Behavior*), které přidá některé kroky v procesu například vytvoření entity. Toto je velice užitečné pokud je třeba přidat

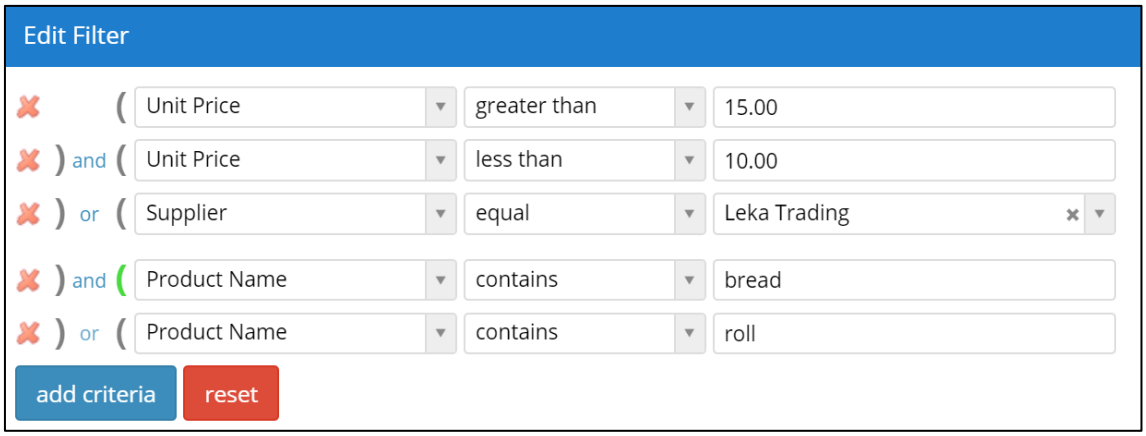
obdobné chování pro skupinu entit. Příkladem může být logování změn nebo kontrola přístupu na základě tenanství. V implementaci tenantství chování kontroluje, že se uživatel nedostane k datům nějakého jiného uživatele.
Entita může obsahovat spojení a tedy i sloupce z jiných tabulek v databázi a může tedy působit obdobně jako databázový pohled. Spojení je také možné vytvořit i na sub dotaz. Tímto se entita může stát velmi komplexní a výsledný SQL dotaz na velký objem dat zdrojově náročný. I s tímto problémem handler efektivně pracuje a SQL dotaz složí pouze na data, která jsou vyžadována například uživatelským dialogem.

5.5 Fluent Migrator

Platforma Serenity využívá Fluent Migrator ke spouštění migrací databáze. Tím je zajištěna i možnost bezproblémového přechodu mezi verzemi výsledného systému, jelikož migrace upravující databázovou strukturu jsou součástí zdrojového kódu aplikace.
Migrace jsou definované jako třídy jazyka C#, které jsou potomky třídy *Migration*. Každá migrace má v anotaci třídy uveden svůj unikátní identifikátor. Migrace mohou být spuštěny přímo z aplikace nebo pomocí externího programu *Migrate.exe*. K uchování informace o migracích, které již proběhly využívá Fluent Migrator tabulku *VersionInfo*.

5.6 Uživatelské rozhraní

Většina uživatelského rozhraní je připravena pro práci s daty. Dialogy a datové gridy jsou také stejně jako ostatní části platformy přizpůsobené k co nejjednodušší úpravě chování. Na základě anotací zmíněných v předchozích podkapitolách se uživatelské rozhraní adaptuje dle datového typu a vazeb na další entity. Data jsou jinak formátována a prvky pro jejich výběr jsou také určeny dle okolností. Jak si lze všimnout na obrázku 1, pokud upravuji vlastnost entity, která je cizím klíčem jako je například dodavatel, selektor pro výběr hodnoty nabízí hodnoty z cílové tabulky.



Obr. 14 Filtrovací dialog
Zdroj: vlastní zpracování

Jak lze vidět na Obr. 14 platforma poskytuje komplexní filtrování dat. Pro rychlé filtrování jsou připravené prvky přímo v datovém gridu.

Většina uživatelského rozhraní je tvořena takzvanými widgety, které vytváří většinu HTML obsahu stránky. Data zobrazovaná uživateli jsou zobrazována pomocí těchto widgetů, které data získávají pomocí volání endpointů (*AJAX*). V důsledku těchto přístupů HTML pohled vytvářený serverem tedy obsahuje jen minimum dat.

Platforma při sestavení projektu provádí transformaci klientských i serverových struktur. Tedy pokud server vrací klientu data, klientský kód má informace o jeho struktuře a datových typech a naopak. To velice usnadňuje práci při vývoji a minimalizuje vznik chyb.

6 Implementace

Kapitola se věnuje postupu implementace systému, který byl navržen v kapitole 4. Zaměříme se především na nejzajímavější problematiky, které jsou uvedeny v kapitolách 3 a 5. Implementace by měla zároveň adoptovat principy uvedené v kapitole 2.

Ukázky v této kapitole ukazují jen nejvýstižnější části implementace. Většina práce, která byla provedena při implementaci výsledného systému bude pouze znázorněna. Pro plné pochopení architektury a implementace našeho systému je třeba prozkoumat zdrojový kód v příloze 1.

6.1 Setup projektu

Implementaci našeho systému zahájíme vytvořením projektu ve vývojovém prostředí Microsoft Visual Studio. Pro vytvoření projektu je třeba stáhnout a nainstalovat šablonu projektu platformy Serenity. Na https://marketplace.visualstudio.com/ vyhledáme šablonu s názvem Serene.

Po vytvoření projektu ve Visual Studiu je projekt připraven k prvnímu spuštění. Projekt již obsahuje implementaci autorizace a autentizace. Dále jsou připraveny entity pro uživatele, uživatelské role a uživatelská oprávnění. Při spuštění webové aplikace již můžeme spravovat všechny zmíněné typy entit.

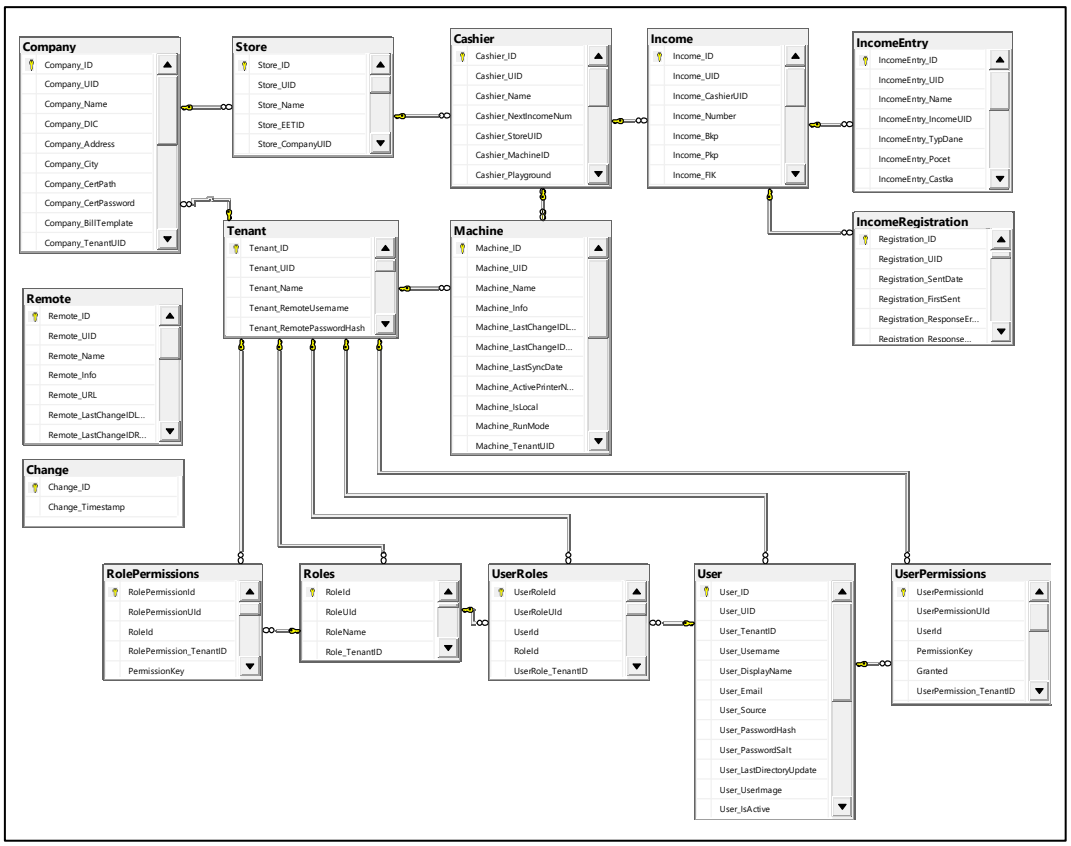
V základním nastavení je použita SQL Server Express LocalDB. Pro změnu použité databáze je třeba změnit připojovací řetězce v souboru appsettings.json.

Platforma Serenity podporuje zobrazení uživatelského prostředí ve 12 světových jazycích. Pro možnost rozšíření či úpravu existujících jazyků jsou připraveny konfigurační soubory. Čeština není podporovaným jazykem. Vzhledem k tomu je třeba vytvořit konfigurační soubor s překladem.

Překlad se nachází v příloze 1. v souboru ZEET.Web\wwwroot\Scripts\site\texts\site.texts.cs-CZ.json. V naší implementaci také zachováme podporu pro anglický jazyk.

6.2 Datový model a migrace

Po vytvoření projektu již máme předpřipravené nějaké základní entity pro práci s uživateli a oprávněními. Dalším krokem implementace bude vytvořit datový model a migrace pro vytvoření struktur v databázi.



Obr. 15 Datový model
Zdroj: Vlastní zpracování – Microsoft SQL Server Management Studio

V horní části diagramu na Obr. 15 můžeme vidět entity již zmiňované v předchozích kapitolách. Tabulka ***Tenant*** označuje jednoho nezávislého zákazníka v našem systému. Každý *Tenant* má své vlastní firmy (*Company*). Tabulka ***Company*** obsahuje informace o těchto firmách, jako například název, DIČ, certifikát pro EET. Tyto informace jsou pak dále využívány při odesílání tržeb elektronické evidence a při tisku účtenky. Každá firma může dále obsahovat více prodejen (*Store*). Tabulka ***Store*** obsahuje informace o prodejně. Obsahuje například identifikaci dle daňového portálu EET. V každé prodejně následně očekáváme více pokladních zařízení (*Cashier*). Tabulka ***Cashier*** obsahuje informace o pokladních zařízeních. Uchovává například informaci o číslování tržeb. Fyzické zařízení představuje tabulka

Machine. Záznam v této tabulce identifikuje přímo fyzické zařízení a uchovává informace o posledních synchronizovaných změnách. Mezi *Machine* a *Cashier* existuje vazba 1:N. Tato vazba umožňuje, aby se na jednom fyzickém zařízení dalo přepínat mezi více pokladními zařízeními. Základem této myšlenky je skutečnost, že někteří uživatelé systému požadují, aby bylo možné zařízení používat ve více prodejnách a to protože zařízení sezóně využívají na více místech. Tedy na jednom zařízení je tedy možné se přepnout mezi více logickými pokladními zařízeními. Poslední nezbytnou částí je tabulka tržeb (***Income***). Tržba má identifikaci pokladny, na které byla provedena a položky potřebné pro EET, jako jsou bezpečnostní kódy PKP, BKP a ověřovací kód FIK. Tržba také obsahuje jednotlivé její položky, které jsou uchované v tabulce ***IncomeEntry***. Tyto položky obsahují informace o počtu, ceně a o daňové kategorii, která se na ně vztahuje. Záznamy mohou být také rozepsány na účtence vydávané zákazníkovi. Registrace EET tržby se může opakovat a to z důvodu možného selhání komunikace. Pro evidenci těchto pokusů se využije tabulka ***IncomeRegistration***.

Ve spodní části diagramu na Obr. 15 můžeme vidět entity spojené především s autorizací a autentizací. Tabulka ***User*** představuje uživatele našeho systému. Uživatelská oprávnění představuje tabulka ***UserPermission***, která obsahuje seznam oprávnění, která má uživatel přidělen. Identifikace samotného oprávnění je v atributu ***PermissionKey***. Informace o tom, zda je oprávnění uživateli povoleno či zamezeno, je dáno v atributu ***Granted***. Uživatel může mít také přidělené uživatelské role (***UserRole***). Uživatelská role obsahuje své vlastní oprávnění (***RolePermission***). Výsledná oprávnění uživatele jsou složena z množin, které poskytují jeho role a jeho vlastní oprávnění. Tento systém umožní uživatele zařadit do rolí, ale je stále možné přiřadit mu role specificky pouze pro něj.

Jak si lze všimnout většina tabulek má vazbu k tabulce ***Tenant***. Tato vazba bude zajišťovat multitenancitu. Tedy každý tenant má své uživatele a definuje své vlastní uživatelské oprávnění, uživatelské role a oprávnění uživatelských rolí.

Tabulka ***Change*** slouží pro správu změn a využívá se pro synchronizaci. Klientská část aplikace komunikuje s centrální částí systému. Pro takovou komunikaci je potřeba zadat URL centrálního systému, na které se pokusí klientská část komunikaci vytvořit. Pro uchování této informace existuje tabulka ***Remote***.

Všechny vazby mezi tabulkami jsou na sloupce s datovým typem GUID. Při synchronizaci mezi zařízeními bude nutné zaručit, že identifikace záznamů bude unikátní. Proto namísto číselného identifikátoru budeme využívat GUID.
Většina zmíněných tabulek navíc obsahuje společnou množinu sloupců:

- *Log_InsertDate* – datum vytvoření záznamu
- *Log_InsertUserId* – autor záznamu
- *Log_UpdateDate* – datum poslední úpravy
- *Log_UpdateUserId* – autor poslední úpravy
- *Log_DeleteDate* – datum smazání záznamu
- *Log_DeleteUserId* – autor smazání záznamu
- *Log_Synchable* – příznak označující zda je možné záznam synchronizovat
- *Log_ReadOnly* – příznak označující zda je možné záznam měnit

Při synchronizaci mezi zařízeními bude nutné synchronizovat informaci o smazaných záznamech. Proto budeme využívat pouze logické smazání záznamů. Logické smazání se tedy provede doplněním sloupce *Log_DeleteDate*.
Jak již bylo popsáno v kapitole 5.5 platforma Serenity využívá migrační framework FluentMigrator. Migrační systém zaručí bezproblémový přechod mezi verzemi výsledného systému. Pro vytvoření naší datové struktury v databázi tedy vytvoříme migrace.

```
[Migration(20191016130000)]
public class DefaultDB_20191016_130000_Company : AutoReversingMigration
{
    public override void Up()
    {
        this.Create.Table("Company")
            .WithColumn("Company_ID").AsInt32().Identity().PrimaryKey().NotNullable()
            .WithColumn("Company_UID").AsGuid().Unique().NotNullable().WithDefault(SystemMethods.NewGuid)
            .WithColumn("Company_Name").AsString(250).NotNullable()
            .WithColumn("Company_DIC").AsString(12).NotNullable()
            .WithColumn("Company_Address").AsString(255).Nullable()
            .WithColumn("Company_City").AsString(255).Nullable()
            .WithColumn("Company_CertPath").AsString(200).Nullable()
            .WithColumn("Company_CertPassword").AsString(200).Nullable()
            .WithColumn("Company_BillTemplate").AsString(2000).Nullable()
            .WithColumn("Company_TenantUID").AsGuid().NotNullable()
                .ForeignKey("Tenant", "Tenant_UID")
            .WithColumn("Log_InsertDate").AsDateTime().NotNullable()
            .WithColumn("Log_InsertUserId").AsGuid().NotNullable()
            .WithColumn("Log_UpdateDate").AsDateTime().Nullable()
            .WithColumn("Log_UpdateUserId").AsGuid().Nullable()
            .WithColumn("Log_DeleteDate").AsDateTime().Nullable()
            .WithColumn("Log_DeleteUserId").AsGuid().Nullable()
            .WithColumn("Log_ChangeID").AsInt64().Nullable()
            .WithColumn("Log_Synchable").AsBoolean().NotNullable().WithDefaultValue(true)
            .WithColumn("Log_ReadOnly").AsBoolean().NotNullable().WithDefaultValue(false);
    }
}
```

Obr. 16 Migrace vytvoření tabulky Company
Zdroj: vlastní zpracování – výsledný systém

Na Obr. 16 můžeme vidět migraci pro vytvoření tabulky *Company*. V migraci jsou specifikovány datové typy, výchozí hodnoty a primární a cizí klíče.

6.3 Generování entit

Jak bylo popsáno v kapitole 5.3 platforma Serenity obsahuje generátor entit z databáze, který vytvoří základní C# a Typescript třídy pro základní operace. Tento generátor nám vygeneruje pro každou databázovou strukturu:

Entita

Představuje třídu, která se rovná jednomu záznamu v databázové tabulce. Obsahuje atributy a specifikuje mapování do relační databáze. Atributy již mají správě určené datové typy. Takto vygenerovaná třída bude mít například pro tabulku *Cashier* název *CashierRow.*

Page

Třída obsahující endpoint s návratem view.

Endpoint

Třída obsahující endpoity, které vracejí JSON. Tyto endpointy jsou volány pouze Ajax voláním a poskytují možnost CRUD operací s entitou.

Repository

Endpoint sám neobsahuje logiku pro práci s daty. Volá metody v repositáři, který vykonává všechny CRUD operace.

Grid

Pro zobrazení dat v aplikaci se využívá Typescript grid, který načítá data AJAX voláním endpointu a následně je zobrazuje.

Dialog

Pro vytváření a úpravu jednotlivých záznamů je vytvořena třída formuláře (Typescript). Formulář načítá data a ukládá změny opět pomocí AJAX volání edpointu.

Generátor nám tedy již vygeneruje všechny třídy potřebné k CRUD operacím. Generátor spustíme pomocí příkazu v adresáři projektu.

dotnet sergen g

Generátor následně nabídne dostupné struktury v databázi. Vybrat lze tabulky nebo pohledy (*view*). Po vybrání struktury pro vygenerování je třeba zvolit modul, do kterého bude entita zařazena. Jako poslední je třeba zadat, jaké třídy se mají pro strukturu vygenerovat.

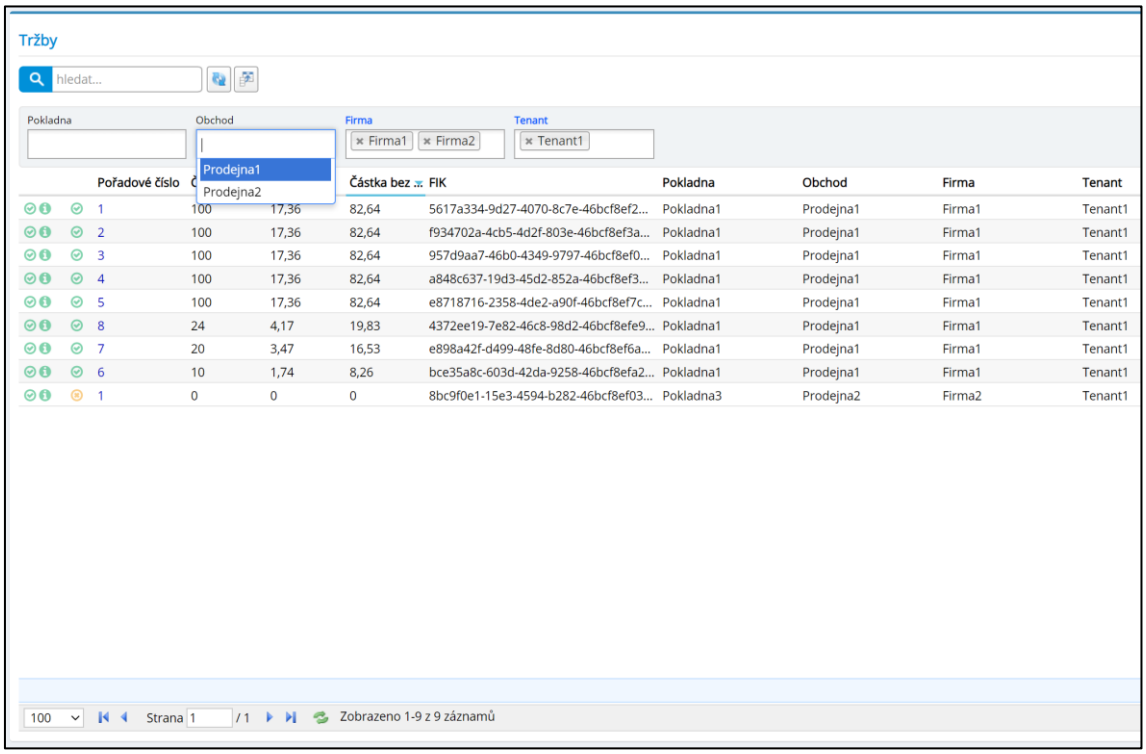
Po vygenerování již existuje třída například *CompanyRow*. Třída obsahuje anotace, které specifikují mapování a omezení. Možné anotace byly uvedeny již v kapitole 5.3 a na Obr. 13.

Jak bylo zmíněno v kapitole 6.2, některé datové struktury (Entity) obsahují stejnou podmnožinu atributů pro logování. Proto vytvoříme předka *LoggingRow*, který bude obsahovat tyto společné atributy. Například třída *CompanyRow* tedy bude potomkem *LoggingRow*. Toto nám také usnadní práci při implementaci specifického chování pro entity tohoto typu.

6.4 Práce s daty a uživatelské rozhraní

Platforma Serenity nabízí nástroje pro efektivní práci s daty. Na Obr. 17 lze vidět grid pro zobrazení tržeb. V datech lze vyhledávat pomocí rychlého vyhledávání.

Panel pro stránkování můžeme vidět ve spodní části gridu. Po kliknutí na název sloupce v hlavičce gridu se provede seřazení dat dle daného sloupce.



Obr. 17 Grid tržeb
Zdroj: Vlastní zpracování – výsledný systém

Na Obr. 17 si můžeme také všimnout rychlých filtrů. Rychlé filtry, které mají být použity v gridu vybereme přidáním anotace *QuickFilter* na atributy v třídě *Row* dané entity (*IncomeRow*). Dalším užitečným nástrojem je komplexní filtrování, které již bylo zmíněno v kapitole 5.6 a na Obr. 14.

Kapitola 5.4 popisuje základní princip fungování *Handlerů* pro práci s daty. Tyto handlery jsou generické a tak fungují pro všechny typy entit. Pro úpravu chování, při například ukládání dat, jsou připravené tzv. *Behaviors*. *Behavior* specifikuje pro jaké typy entit je platný. Tedy uplatní se pouze u daných typů. Behavior pak dále provádí úkony v různých fázích procesu, například uložení záznamu.

V kapitole 6.2 již bylo zmíněno, že většina datových struktur (Entit) obsahuje stejnou podmnožinu atributů týkajících se informací o vytvoření, úpravě, smazání a autorech těchto činností. Tyto položky nemůže uživatel volně měnit a tak je nutné, aby je systém určoval sám. K takovému úkonu využijeme *Behavior*.

```
public class LogBehavior : ISaveBehavior, IDeleteBehavior,IListBehavior,IRetrieveBehavior, IImplicitBehavior {
    private ILoggingRow logRow;
    public bool ActivateFor(Row row){
        logRow = row as ILoggingRow;
        return logRow != null;
    }
    public void OnReturn(ISaveRequestHandler handler){
        if(Machine.CurrentTrans(handler.Connection).RunMode == RunMode.client)
            Dependency.Resolve<Sync.SyncService>().ClientInvokeSync(handler.Connection);
    } ... obdobně implementováno pro IDeleteRequestHandler
    public void OnPrepareQuery(IListRequestHandler handler, SqlQuery query) {
        query.Where(logRow.DeleteDateField.IsNull());
    } ... obdobně implementováno pro IRetrieveRequestHandler
    public void OnSetInternalFields(ISaveRequestHandler handler) {
        var row = handler.Row;
        if (handler.IsUpdate) {
         logRow.UpdateDateField[row]=DateTimeField.ToDateTimeKind(DateTime.Now, logRow.UpdateDateField.DateTimeKind);
            logRow.UpdateUserIdField[row] = ((UserDefinition)Authorization.UserDefinition).UserUId;
        }
        else if (handler.IsCreate) {
            if(!row.IsAssigned(logRow.SynchableField)) logRow.SynchableField[row] = true;
          logRow.InsertDateField[row]=DateTimeField.ToDateTimeKind(DateTime.Now,logRow.InsertDateField.DateTimeKind);
            logRow.InsertUserIdField[row] = ((UserDefinition)Authorization.UserDefinition).UserUId;
        }
        if(logRow.SynchableField[handler.Row] == true){
            long changeId = (long)handler.Connection.InsertAndGetID<ChangeRow>(new ChangeRow(){
                Timestamp = DateTime.Now });
            logRow.ChangeIDField[handler.Row] = changeId;
        }
    }
    public void OnValidateRequest(ISaveRequestHandler handler) {
        if(handler.IsUpdate && logRow.ReadOnlyField[handler.Old] == true)
            throw new AccessViolationException("Read only!");
        }
    } ... obdobně implementováno pro IDeleteRequestHandler
...
}
```

Obr. 18 Log behavior
Zdroj: Vlastní zpracování – výsledný systém

Na Obr. 18 můžeme vidět, že *LogBehavior* bude platný pouze pro entity typu *ILoggingRow* a že automaticky nastavuje atributy.

Jak již bylo zmíněno, provádíme pouze logické smazání záznamů. K implementaci takového chování použijeme také *LogBehavior*. Ten zaručí, že při listování dat se přidá podmínka o nulovosti atributu *Log_Deleted* do klauzule *where*. Při pokusu o smazání záznamu provede pouze doplnění hodnoty do *Log_Deleted* a záznam fyzicky nesmaže.

6.5 Oprávnění a tenantství

Výsledný systém musí zaručit spolehlivou nezávislost a oddělení dat jednotlivých tenantů. Pro implementaci takového chování se opět nabízí využití *Behavior*.

Vyžadované chování:

- Uživateli budou poskytnuta data pouze od jeho tenanta
- Uživatel může měnit data pouze svého tenanta

```
public class TenantBehavior
         :ISaveBehavior, IDeleteBehavior,IListBehavior,IRetrieveBehavior, IImplicitBehavior
{
    private ITenantRow tenantRow;
    public bool ActivateFor(Row row){
        tenantRow = row as ITenantRow;
        return tenantRow != null;
    }
    public void OnPrepareQuery(IListRequestHandler handler, SqlQuery query){
        if(!Authorization.HasPermission(ManagementPermissionKeys.SuperAdmin))
            query.Where(new Criteria(tenantRow.TenantField) ==
                  ((UserDefinition)Authorization.UserDefinition).TenantUId);
    } ... obdobně také pro IRetrieveRequestHandler, ISaveRequestHandler, IDeleteRequestHandler
    public void OnSetInternalFields(ISaveRequestHandler handler){
        var row = handler.Row;
        //pokud nemá právo volit tenanta nebo tenant není vybrán
        if(!Authorization.HasPermission(ManagementPermissionKeys.SuperAdmin)
             || tenantRow.TenantField.AsObject(row) == null)
            tenantRow.TenantField[row] = ((UserDefinition)Authorization.UserDefinition).TenantUId;
    }
    public void OnValidateRequest(ISaveRequestHandler handler){
        if(handler.IsUpdate &&
         tenantRow.TenantField[handler.Old] !=((UserDefinition)Authorization.UserDefinition).TenantUId){
             Authorization.ValidatePermission(ManagementPermissionKeys.SuperAdmin);
        }
    } ... kontrola oprávnění změny. Obdobně také pro IDeleteRequestHandler
    public void OnAfterSave(ISaveRequestHandler handler) {
        var fldId = (Field)(handler.Row as IIdRow).IdField;
        var fields=handler.Row.GetFields().Where(f=>
            f.CustomAttributes.Any(att=>att.GetType()==typeof(TenancyCheckAttribute)));
        if(!fields.Any()) return;
        Row r = (Row)Activator.CreateInstance(handler.Row.GetType());
        new SqlQuery().Dialect(handler.Connection.GetDialect()).From(r).Select(fields.ToArray())
            .WhereEqual(fldId, fldId.AsObject(handler.Row)).GetFirst(handler.Connection);
        foreach(var p in fields){
            var val = (Guid?)p.AsObject(r);
            if(val.HasValue && val != tenantRow.TenantField[handler.Row]) throw new UnauthorizedAccessException();
        }
    } ... kontrola cizích klíčů (stejný tenant)
...
}
```

Obr. 19 Tenant behavior
Zdroj: Vlastní zpracování – výsledný systém

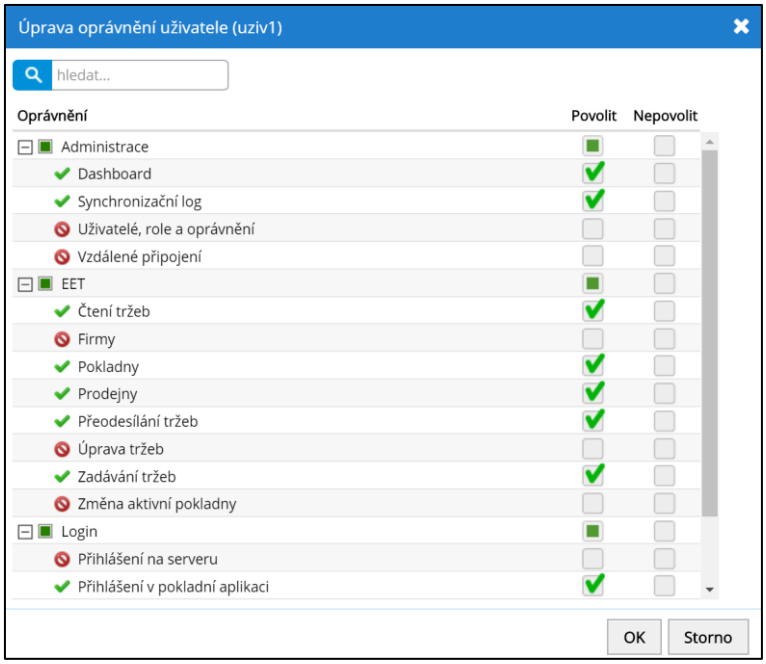
Na Obr. 19 můžeme vidět opět přidání podmínky při listování dat z databáze. Dále můžeme vidět, že *TenantBehavior* také kontroluje zda existující záznam má stejného tenanta jako uživatel nebo automaticky doplňuje tenanta při vytvoření nového záznamu. Tímto je zaručeno, že uživatel může pracovat pouze s daty svého tenanta. V poslední metodě na Obr. 19 můžeme vidět kontrolu cizích klíčů. Metoda se provádí až po uložení záznamu do databáze, ale stále ve stejné transakci. Pokud tedy nastane chyba (vyvolaná kontrolou) všechny provedené změny jsou vráceny zpět (*rollback*).

V metodě se načte řádek, který byl právě vytvořen a je provedeno ověření, že všechny referované záznamy jsou ze stejného tenanta. Pro upřesnění, které atributy se mají kontrolovat využíváme *TenancyCheckAttribute*, který je specifikován na atributech v *Row* nějaké entity.

Jak již bylo zmíněno v kapitole 4.5, výsledný systém by měl podporovat uživatelská práva a role. Sadu uživatelských práv, která ovlivňuje možnosti uživatele je pevně stanovená implementací systému. Práva mohou být specifikována na různých místech:

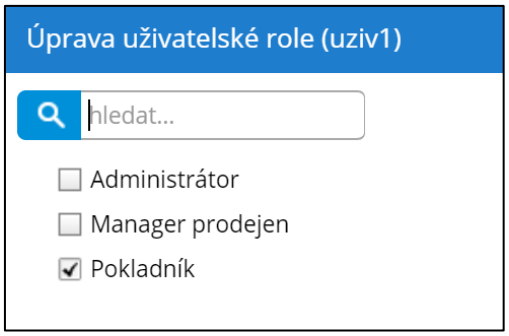
- Kontroler – pro zobrazení stránky
- Služba – pro volání
- Row – ovlivňuje handler pro zprostředkování CRUD operací

Každé právo umožňuje množinu různých úkonů v systému. Zákazník systému (tenant) může poté přiřazovat tyto práva svým uživatelům. Dialog pro přiřazení oprávnění uživateli můžeme vidět na Obr. 20.



Obr. 20 Přiřazení oprávnění uživateli
Zdroj: Vlastní zpracování – výsledný systém

Pro zjednodušení si může zákazník vytvořit uživatelské role. Uživatelské roli přiřadí sadu povolených oprávnění. Každému uživateli tedy již nemusí specifikovat oprávnění, ale pouze mu přiřadí jednu nebo více rolí.



Obr. 21 Přiřazení rolí uživateli
Zdroj: Vlastní zpracování – výsledný systém

Funkcionalita pro přiřazení rolí je již poskytnuta platformou Serenity. Ukázku přiřazení uživatelských rolí můžeme vidět na Obr. 21.

6.6 Pokladní zařízení a centrální systém

Navrhovaný systém se skládá z aplikace pro pokladní zařízení a centrálního serveru pro správu. Jak již bylo zmíněno v kapitole 4.1, vzhledem k tomu, že množina datových struktur a operací s nimi je v obou částech systému velice obdobná, bude navržena pouze jedna aplikace zastávající obě role. Pro výběr módu ve kterém má aplikace operovat, využijeme konfigurační soubor.

```
public class ClientRunModeAttribute: Attribute, IResourceFilter{
    public ClientRunModeAttribute(){}
    public void OnResourceExecuted(ResourceExecutedContext context){}
    public void OnResourceExecuting(ResourceExecutingContext context) {
        if(CurrentRunMode.Current != RunMode.client){
            if (Authorization.IsLoggedIn){
                context.Result = new Result<ServiceResponse>(new ServiceResponse {
                    Error = new ServiceError {
                        Code = "AccessDenied",
                        Message = LocalText.Get("Authorization.AccessDenied")
                    }
                });
                context.HttpContext.Response.StatusCode = 400;
            }
            else {
                context.Result = new Result<ServiceResponse>(new ServiceResponse {
                    Error = new ServiceError {
                        Code = "NotLoggedIn",
                        Message = LocalText.Get("Authorization.NotLoggedIn")
                    }
                });
                context.HttpContext.Response.StatusCode = 400;
            }
        }
    }
}
```

Obr. 22 Atribut pro omezení módu aplikace
Zdroj: Vlastní zpracování – výsledný systém

Uživatelské rozhraní a chování se však v mnoha částech liší. V jednotlivých módech chceme omezit zobrazení některých částí a přístup k endpointům. Například v serverové části nechceme, aby bylo možné vytvářet tržby. Pro zamezení přístupu k různým akcím edpointů vytvoříme atribut pro filtrování požadavků. Implementaci atributu, který povoluje přístup pouze v módu pokladní aplikace, můžeme vidět na Obr. 22. Obdobně vypadá i atribut pro omezení přístupu pouze pro mód centrálního serveru.

Příklad využití atributu pro omezení vytvoření tržby můžeme vidět na Obr. 23.

```
public class IncomeController : ServiceEndpoint
{
...
    [ClientRunMode]
    public RetrieveResponse<IncomeRegistrationRow> Create(IUnitOfWork uow,
IncomeCreateRequest request)
    {
        return new MyRepository().Create(uow, request);
    }
...
```

Obr. 23 Použití atributu pro omezení přístupu
Zdroj: Vlastní zpracování – výsledný systém

Stejně tak jako platby, v serverové části se nedají vytvářet logické pokladny. Vytvoření pokladny musí být provedeno přímo v aplikaci pokladního zařízení (módu). Přístup do endpointu provedeme obdobně jako u vytvoření platby. Dále je potřeba zamezení zobrazení tlačítka pro přidání poklady.

```
@Serenity.Decorators.registerClass()
export class CashierGrid extends TenantBaseGrid<CashierRow, any> {
...
    protected getButtons(){
        let btns = super.getButtons();
        if(Authorization.machineDefinition.RunMode == Common.RunMode.client)
            return btns;
        return  btns.filter(b=>b.cssClass != "add-button")
    }
...
}
```

Obr. 24 Omezení zobrazení tlačítka (Typescript)
Zdroj: Vlastní zpracování – výsledný systém

Pro takové omezení využijeme třídu gridu *CashierGrid*, která byla vygenerovaná platformou Serenity. Úpravu funkcionality provedeme přepsáním metody *getButtons* a v závislosti na aktuálním módu vyfiltrujeme tlačítka k zobrazení (Obr. 24).

6.7 Synchronizace

Jednou z důležitých částí našeho systému je synchronizace dat mezi centrálním serverem a pokladními aplikacemi, které představují pokladní zařízení. V kapitole

4.8 byl navržen způsob synchronizace. Komunikace je provedena stylem *request-response*.

První krok komunikace mezi pokladní aplikací a centrálním serverem je navázání vazby k nějakému tenantovi. Pro první navázání je třeba znát identifikaci tenanta (*GUID*) a jeho tajný kód (*Secret*). Metoda, která je vyvolána při navázání na straně serveru je na Obr. 25.

```
public SyncDataResponse ServerConnectToTenant(IUnitOfWork uow, SyncDataRequest request){
    bool alreadyExists = new SqlQuery().From(new MachineRow()).Select("1").WhereEqual(MachineRow.Fields
.MachineUid, request.Machine.MachineUid).Exists(uow.Connection);
    if(alreadyExists)throw new ValidationError("Machine already exists");

    var tRow = new TenantRow();
    if(!new SqlQuery().From(tRow)
    .SelectTableFields()
    .WhereEqual(TenantRow.Fields.UID, request.Tenant.UID)
    .WhereEqual(TenantRow.Fields.ConnectionSecret, request.Tenant.ConnectionSecret)
    .WhereEqual(TenantRow.Fields.Synchable,true)
    .Where(new Criteria(TenantRow.Fields.DeleteDate).IsNull()).GetFirst(uow.Connection))
        throw new ValidationError("Bad credentials");

    request.Machine.IsLocal = false;
    request.Machine.TenantUID = tRow.UID;
    request.Machine.ClearAssignment(MachineRow.Fields.MachineId);
    request.Machine.ClearAssignment(MachineRow.Fields.LastChangeRemoteGot);
    clearNotTableFields(request.Machine);
    new SqlInsert(MachineRow.Fields.TableName).Set(request.Machine).Execute(uow.Connection);

    //změna secret po každém připojení
    new SqlUpdate(TenantRow.Fields.TableName).Set(TenantRow.Fields.ConnectionSecret,Guid.NewGuid())
    .WhereEqual(TenantRow.Fields.UID, request.Tenant.UID).Execute(uow.Connection);

    var res = new SyncDataResponse();
    getDataToUpdateSince(uow.Connection,res,0,tRow.UID.Value,request.Machine.MachineUid.Value);
    res.ServerChangeID = uow.Connection.Max<ChangeRow>(ChangeRow.Fields.Id,Criteria.True);
    res.Tenant = tRow;
    return res;
}
```

Obr. 25 První připojení k centrálnímu serveru
Zdroj: vlastní zpracování – výsledný systém

Podobně jako ve vektorových hodinách zmíněných v kapitole 2.4.3 budeme číslovat události. Událost bude představovat úprava, smazání nebo přidání nějakých dat. Číslování událostí obstará *LogBehavior*. Implementaci můžeme vidět na Obr. 18

v metodě *OnSetInternalFields*. Každý prvek v systému bude své události číslovat lokálně a bude také udržovat informace o tom, jaké číslo události získal jako poslední od jiných zařízení při synchronizaci.

Proces vychází z návrhu v kapitole 4.8.5 a na Obr. 11. Synchronizaci vyvolává pokladní aplikace tak, že se dotáže serveru na poslední *ChangeID*, které od této pokladní aplikace obdržel. Pokladní aplikace následně odešle v požadavku všechny změny od zmíněného *ChangeID* a zároveň přidá informaci o posledním *ChangeID*, které obdržela od serveru. Serverová část tyto změny přijme a provede jejich uložení do databáze. Server do odpovědi zahrne všechny změny od *ChangeID*, které požadovala pokladní aplikace. V posledním kroku pokladní aplikace uloží data přijatá v odpovědi do databáze. Část zdrojového kódu, který provádí synchronizaci na pokladní aplikaci můžeme vidět na Obr. 26.

```
private ServiceResponse ClientInvokeSync(IUnitOfWork uow, ServiceRequest request)
{
    RemoteRow remote = uow.Connection.TryFirst<RemoteRow>(new Criteria(RemoteRow.Fields.DeleteDate).IsNull());
    if(remote == null) throw new ValidationError("No remote available");
    var machine = Machine.CurrentTrans(uow.Connection);
    var machineUID = machine.MachineUid.Value;
    var client = new MyJsonServiceClient(remote.Url);
    var resp = client.Post<GetLastChangeIDResponse>("/services/Sync/Sync/ServerGetLastChangeID",
        new GetLastChangeIDRequest(){
            MachineUID = machineUID});

    var clientLastChID = resp.ChangeID; //poslední ChangeID které ode mě server dostal
    var req = new SyncDataRequest();
    req.Machine = machine;
    req.ServerChangeID = machine.LastChangeFromRemote ?? -1; //poslední ChangeID co jsem přijmul od serveru
    req.ClientChangeID = uow.Connection.Max<ChangeRow>(ChangeRow.Fields.Id,Criteria.True); //zasílám všechny změny
    getDataToUpdateSince(uow.Connection,req,clientLastChID,machine.TenantUID.Value,machineUID,log);
    var dataResp = client.Post<SyncDataResponse>("/services/Sync/Sync/ServerSyncData",req);

    long lastChangeID = 0;
    dataResp.Users.ForEach(u=>lastChangeID  = createOrUpdate(uow,u,machine.TenantUID.Value,cleanUsers:true));
    dataResp.Roles.ForEach(u=>lastChangeID  = createOrUpdate(uow,u,machine.TenantUID.Value));
    dataResp.UserPermissions.ForEach(u=>lastChangeID  = createOrUpdate(uow,u,machine.TenantUID.Value));
    dataResp.UserRoles.ForEach(u=>lastChangeID  = createOrUpdate(uow,u,machine.TenantUID.Value));
    dataResp.RolePermission.ForEach(u=>lastChangeID  = createOrUpdate(uow,u,machine.TenantUID.Value));
    dataResp.Companies.ForEach(u=>lastChangeID  = createOrUpdate(uow,u,machine.TenantUID.Value));
    dataResp.Stores.ForEach(s=>lastChangeID  = createOrUpdate(uow,s,machine.TenantUID.Value));
    dataResp.Cashiers.ForEach(s=>lastChangeID  = createOrUpdate(uow,s,machine.TenantUID.Value));
    dataResp.Incomes.ForEach(s=>lastChangeID  = createOrUpdate(uow,s,machine.TenantUID.Value));
    dataResp.IncomeEntries.ForEach(s=>lastChangeID  = createOrUpdate(uow,s,machine.TenantUID.Value));
    dataResp.IncomeRegistrations.ForEach(s=>lastChangeID= createOrUpdate(uow,s,machine.TenantUID.Value));
    dataResp.Users.ForEach(u=>updateUsers(uow,u));

    var latestServerGot = uow.Connection.Max<ChangeRow>(ChangeRow.Fields.Id,Criteria.True);
    //uložení informací
    new SqlUpdate(MachineRow.Fields.TableName)
        .Set(MachineRow.Fields.LastChangeFromRemote,dataResp.ServerChangeID)
        .Set(MachineRow.Fields.LastChangeRemoteGot,latestServerGot)
        .Set(MachineRow.Fields.LastSyncDate,DateTime.Now)
        .Execute(uow.Connection);

    //dám serveru vědět jaké bylo poslední ChangeID které přijal (jeho změny vytvořily nové ChangeID)
    var req2= new UpdateLastChangeIDRequest(){
        ChangeID= latestServerGot,
        RemoteChangeID = dataResp.ServerChangeID,
        MachineUID = machineUID
    };
    client.Post<ServiceResponse>("/services/Sync/Sync/ServerUpdateLastChangeID",req2);
    return new ServiceResponse();
}
```

Obr. 26 Synchronizace (zdrojový kód)
Zdroj: vlastní zpracování – výsledný systém

Vzhledem k vazbám v datovém modelu (Obr. 15) je při synchronizaci změn třeba dbát na pořadí v jakém budeme změny ukládat do databázového systému. Pokud bychom tak neudělali, porušovali bychom tím referenční integritu a databázový systém by vyhodil chybu. Proto určíme pořadí změn:

- Uživatelé (*User*)
- Uživatelská oprávnění (*Role, RolePermission, UserRole, UserPermission*)
- Firmy (*Company*)
- Prodejny (*Store*)
- Pokladny (*Cashier*)
- Tržby (*Income, IncomeEntry, IncomeRegistration*)

Při synchronizaci je také třeba vyřešit problémy zmíněné v kapitole 4.8.4. Především potřebujeme zaručit, že nebude možné upravovat či číst data k nímž nemá uživatel oprávnění (data jiných tenantů). K takovému opatření použijeme obdobné kontroly jako na Obr. 19.

Synchronizace je vyvolána po každé změně nebo přidání záznamu, který je určen k synchronizaci. O vyvolání se stará *LogBehavior* v metodě *OnReturn* (Obr. 18). Změny je možné provádět i v centrální serverové aplikaci. Bohužel vyvolání synchronizace ve směru od centrální serveru k pokladnímu zařízení není možné. Proto pokladní aplikace periodicky vyvolává synchronizaci v intervalu, který je specifikovaný v nastavení aplikace. Implementace je provedena ve formě úlohy na pozadí (*IHostedService*).

6.8 EET

V kapitole 3 již byla uvedena architektura elektronické evidence tržeb. Implementace datové komunikace bude v našem systému vycházet především z informací uvedených v [8].

6.8.1 Zadávání plateb

Pro co nejjednodušší zadávání plateb je v pokladní aplikaci připravena stránka, kterou můžeme vidět na Obr. 33. Tržba se skládá z jednotlivých položek. Pokladník zadá částku položky a stiskne klávesovou zkratku pro připsání do určité daňové třídy (21%,15%,…) dle typu položky. Pro dokončení tržby se odešle request do endpointu, který data zpracuje a uloží do databáze. Následně jsou data předána k vytvoření datové zprávy.

6.8.2 Datová zpráva

Komunikace s technickým zařízením správce daně je provedeno v podobě SOAP XML zpráv. V kapitole 3 již zmiňujeme problematiku komunikace.

Pro vytvoření datové zprávy o tržbě použijeme předem vytvořenou šablonu tak, aby ji systém nemusel vytvářet pokaždé znovu. Do šablony následně doplníme požadované informace o platbě.

Většina informací, které jsou obsaženy v datové zprávě není třeba vypočítávat a stačí je pouze doplnit. Výjimkou jsou bezpečnostní kódy BKP a PKP. K vytvoření těchto bezpečnostních kódů je nutné použít certifikát poplatníka. Fyzickou cestu a heslo k certifikátu je uvedeno v záznamu firmy (*Company*). Pomocí *JOIN* v *IncomeRow* získáme tyto informace v jednom objektu. Útržek kódu načítající certifikát poplatníka můžeme vidět na Obr. 27.

```
try{
    cert = new X509Certificate2(newIncome.CertPath,newIncome.CertPassword,X509
KeyStorageFlags.Exportable);
}
catch(Exception ex){
    throw new Exception("Chyba při načítání certifikátu - "+ex.Message,ex);
}
```

Obr. 27 Načtení certifikátu poplatníka
Zdroj: vlastní zpracování – výsledný systém

Kapitola 3.6 popisuje položky a způsob vytvoření BKP a PKP. To jak jej vytváří náš systém můžeme vidět na Obr. 28.

```
public static void FormPKPandBKP(IncomeRow income, X509Certificate2 cert){
    string dic = income.CompanyDIC.ToString();
    string idProvozovny = income.StoreEETID.ToString();
    string idPokladny = income.CashierName;
    string poradCis = income.Number;
    string datPrijetiPlatby =  income.InsertDate.Value.ToString("yyyy-MM-ddTHH:mm:sszzz");
    string celkCastka = String.Format("{0:0.00}",income.TrzbaCelkem).Replace(",",".");
    string pkpString = $"{dic}|{idProvozovny}|{idPokladny}|{poradCis}|{datPrijetiPlatby}|{celkCastka}";

    RSACng privateKey = (RSACng)cert.PrivateKey;
    RSACryptoServiceProvider privateKey1 = new RSACryptoServiceProvider();
    privateKey1.ImportParameters(privateKey.ExportParameters(true));

    var data = Encoding.UTF8.GetBytes(pkpString);
    byte[] signature = privateKey.SignData(data,HashAlgorithmName.SHA256,RSASignaturePadding.Pkcs1);
    var signatureText1 = System.Convert.ToBase64String(signature);
    byte[] signature2 = privateKey1.SignData(data,"SHA256");
    var signatureText2 = System.Convert.ToBase64String(signature2);

    var hash = new SHA1Managed().ComputeHash(signature);
    var bkp= BitConverter.ToString(hash).Replace("-","");
    foreach( var i in new int[]{8,16+1,24+2,32+3}){
        bkp =bkp.Insert(i,"-");
    }
    income.Pkp = signatureText2;
    income.Bkp = bkp;
}
```

Obr. 28 Vytvoření BPK a PKP

Zdroj: vlastní zpracování – výsledný systém

Následně se z těla datové zprávy vytvoří otisk (*DigestValue*), který je vložen do podpisové části hlavičky (*SignatureInfo*). Z podpisové části (*SignatureInfo*) je následně vytvořen podpis použitím certifikátu. Tento podpis je vložen do hlavičky datové zprávy (*SignatureValue*). Do hlavičky zprávy (*BinarySecurityToken*) se také přidá samotný certifikát, který byl použit k podpisům.

Datová zpráva je následně odeslána na URL produkčního nebo testovacího prostředí (Playground). Příznak udávající, které z prostředí se má využít je obsažen v záznamu pokladny (*Cashier*).

```
HttpWebRequest webRequest = (HttpWebRequest)WebRequest.Create(url);
var content = Encoding.UTF8.GetBytes(registration.SoapMessage);
webRequest.Headers.Add("SOAPAction", "http://fs.mfcr.cz/eet/OdeslaniTrzby");
webRequest.ContentType = "text/xml;charset=\"utf-8\"";
webRequest.Accept = "text/xml";
webRequest.Method = "POST";
webRequest.ContentLength = content.Length;
webRequest.GetRequestStream().Write(content);
var response = webRequest.GetResponse();
try{
    string soapResult = "";
    using (StreamReader rd = new StreamReader(response.GetResponseStream())){
        soapResult = rd.ReadToEnd();
    }
    XDocument doc = XDocument.Parse(soapResult);
    string textChyby = "";
    string textVarovani = "";
    textChyby = string.Join(", ",doc.Descendants().Where(e=>e.Name.LocalName == "Chyba").Select(e=>
        $"Kód {e.Attributes("kod").First().Value} - {(string)e}"
    ));
    textVarovani = string.Join(", ",doc.Descendants().Where(e=>e.Name.LocalName == "Varovani").Select(e=>
        $"Kód {e.Attributes("kod_varov").First().Value} - {(string)e}"
    ));
    var fik = doc.Descendants().Where(e=>e.Name.LocalName=="Potvrzeni").Select(e=>e.Attributes("fik").First().Value);
    if(fik.Any()) registration.ResponseFIK = fik.First();
    if(textChyby.Length > 0) registration.ResponseError = textChyby;
    if(textVarovani.Length > 0) registration.ResponseWarning = textVarovani;
}
catch(Exception e){
    registration.ResponseError = e.Message + e.StackTrace;
}
```

Obr. 29 Odeslání datové zprávy
Zdroj: vlastní zpracování – výsledný systém

Technické zařízení správce daně odpoví na datovou zprávu. V případě chyby poskytne v odpovědi informace specifikující chybu. V případě úspěšné evidence navrátí fiskální identifikační kód tržby. Odeslání datové zprávy a následné procesování odpovědi můžeme vidět na Obr. 29.

6.8.3 Účtenka a tisk

Jeden z požadavků na pokladní aplikaci je tisk účtenek evidovaných tržeb. Mnoho zákazníků může vyžadovat svůj vlastní vzhled účtenky a různé informace obsažené na účtence. Vzhledem k tomu bylo v kapitole 4.6 navrženo šablonování formátu účtenky. Každý zákazník tedy bude moci specifikovat svou vlastní šablonu.

Pro implementaci využijeme šablonový jazyk Scriban. Scriban poskytuje direktivy a funkce pro práci s objekty, poli objektů. Nabízí také funkce pro aritmetiku a formátování. Scriban přijímá objekty s daty a funkcemi. Šablona obsahuje direktivy jazyka, které pracují s těmito objekty a vytváří finální podobu. Šablona je specifikována pro každou firmu (*Company*).

```
{{-drawer_kick-}}
{{-align_center-}}
************************
* {{center Trzba.FirmaNazev 22 }} *
* {{center Trzba.FirmaAdresa 22 }} *
* {{center Trzba.FirmaMesto 22 }} *
************************
* {{center Trzba.ObchodNazev 22 }} *
************************
{{-if !Trzba.FIK}}
PKP: {{Trzba.PKP}}
{{-else}}
FIK: {{Trzba.FIK}}
{{-end}}
BKP: {{Trzba.BKP}}
DIC/IC: {{Trzba.DIC}}
Provozovna: {{Trzba.ObchodEETID}}
Pokladna: {{Trzba.PokladnaNazev}}
Uctenka: {{Trzba.PoradoveCislo}}
Datum: {{date.to_string Trzba.DatumVytvoreni `%d. %m. %Y %T`}}
================================
Polozka          Sazba      Cena
{{-for Polozka in Trzba.Polozky}}
    {{~string.pad_right Polozka.Nazev 16}}
    {{-string.pad_left Polozka.TypDane 6}}
    {{-string.pad_left Polozka.Castka 10}}
{{-end}}
================================
Sazba   Zaklad     DPH    Celkem
{{-if Trzba.Trzba21 != 0}}
    {{~string.pad_right "21%" 6}}
    {{-string.pad_left Trzba.Trzba21BezDPH 8}}
    {{-string.pad_left Trzba.Trzba21DPH 8}}
    {{-string.pad_left Trzba.Trzba21 10}}
{{-end}}
{{-if Trzba.Trzba10 != 0}}
    {{~string.pad_right "10%" 6}}
    {{-string.pad_left Trzba.Trzba10BezDPH 8}}
    {{-string.pad_left Trzba.Trzba10DPH 8}}
    {{-string.pad_left Trzba.Trzba10 10}}
{{-end}}
{{-if Trzba.Trzba15 != 0}}
    {{~string.pad_right "15%" 6}}
    {{-string.pad_left Trzba.Trzba15BezDPH 8}}
    {{-string.pad_left Trzba.Trzba15DPH 8}}
    {{-string.pad_left Trzba.Trzba15 10}}
{{-end}}
{{-if Trzba.Trzba0 != 0}}
    {{~string.pad_right "0%" 6}}
    {{-string.pad_left Trzba.Trzba0 26}}
{{-end}}
--------------------------------
          Celkem
{{- string.pad_left Trzba.TrzbaCelkem 16}}
--------------------------------
{{-line_feed 5}}
```

Obr. 30 Výchozí šablona účtenky

Zdroj: vlastní zpracování – výsledný systém

Výchozí šablonu pro účtenku můžeme vidět na Obr. 30. V šabloně se dotazujeme na objekt *Trzba*, který představuje *IncomeRow* tržby. Dále si můžeme všimnout volání funkcí pro formátování textu a datumů, které jsou součástí jazyka Scriban. V našem systému také specifikuje speciální funkce, které jsou určené pro přidání kontrolních znaků pro ovládání řádkové tiskárny (*Line printer*). Například funkce *drawer_kick* přidá kontrolní kód pro otevření pokladního šuplíku připojeného k tiskárně.

Kód použití k renderování výsledné účtenku z šablony můžeme vidět na Obr. 31.

```
private string FillTemplateInternal(IncomeRow row, string templateText){
    var scribanObj = Generation.GetScriptObject(row,typeof(IncomeRow));
    ScriptObject so = new ScriptObject();
    so.Add("Trzba",scribanObj);
    so.Import(typeof(ScribanFunctions));
    var context = new TemplateContext { MemberRenamer = member => member.Name };
    context.PushGlobal(so);

    var t = Template.Parse(templateText);
    string resultText = t.Render(context);
    string[] lines = resultText.Split(Environment.NewLine);
    return resultText;
}
```

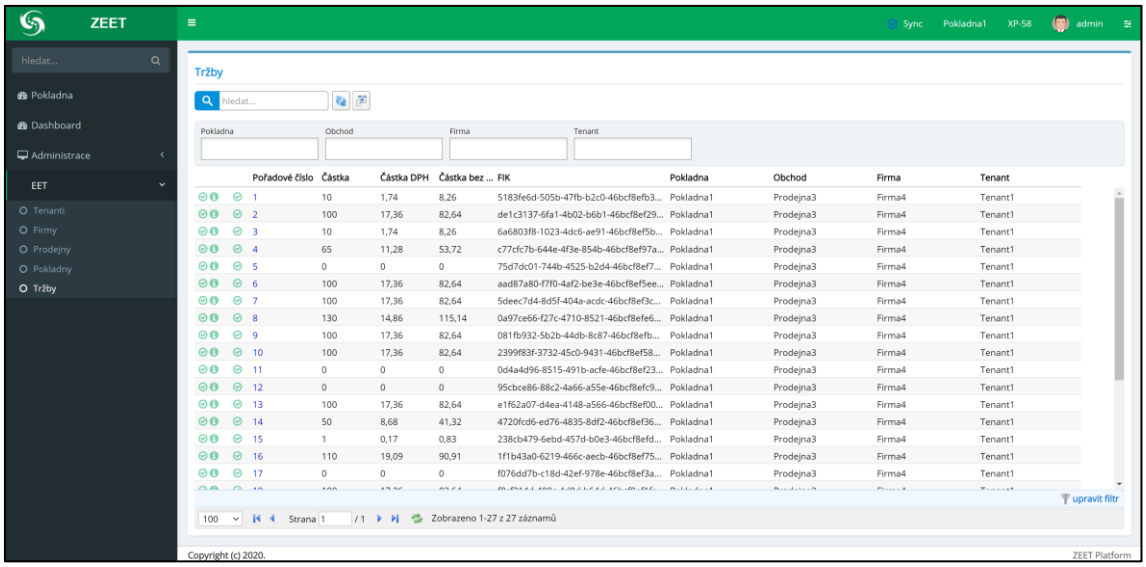
Obr. 31 Renderování účtenky
Zdroj: vlastní zpracování – výsledný systém

V našem systému se zaměřujeme na použití řádkových termálních tiskáren, jelikož jsou ve většině místech nejpoužívanějším způsobem tisku. Proto je naše implementace uzpůsobena především pro fungování s řádkovou tiskárnou. Jelikož řádkové tiskárny se řídí jinou sadou kontrolních kódů, stejné řešení nefunguje pro jiné typy tiskáren.

K tisku jsou vyhledány nainstalované tiskárny v systému. Na Obr. 32 můžeme v horním panelu vidět tlačítko pro vybrání aktivní tiskárny. Při odeslání tržby se automaticky použije vybraná tiskárna. V seznamu tržeb (Obr. 32) je u každého záznamu tlačítko pro opětovný tisk tržby. Tisk v našem systému byl testován s tiskárnou Xprinter XP58-IIN.

6.9 Uživatelské rozhraní

Náš systém obsahuje mnoho funkcí a nebudeme se dále zabývat jejich podrobným rozborem. Pro hlubší pochopení dalších funkcí lze prozkoumat zdrojový kód v příloze 1.

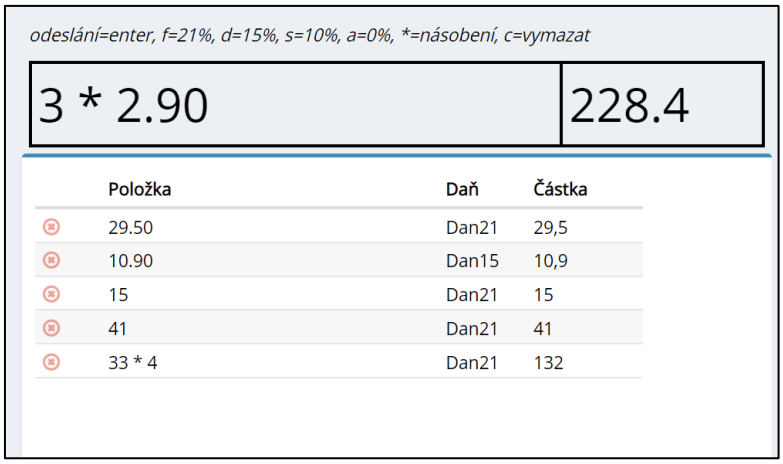


Obr. 32 Tržby
Zdroj: vlastní zpracování – výsledný systém

Na Obr. 32 můžeme vidět ukázku ze seznamu tržeb. Každý záznam v gridu obsahuje ikonky pro informaci o jejich aktuálním stavu a případné akce. První ikona indikuje úspěšnou evidenci na daňový portál. V případě, že tržba ještě není evidována, tato ikona slouží pro opětovné odeslání tržby na technické zařízení správce daně. Druhá ikona slouží k opakovanému tisku účtenky.

Třetí ikona je společná pro všechny entity podléhající synchronizaci. Ikona označuje stav synchronizace záznamu. Například pokud vytvoříme novou tržbu, ikona dané tržby bude indikovat, že záznam ještě nebyl synchronizován s centrálním serverem. V pravém horním panelu stránky můžeme vidět tlačítko s ikonou označující celkový stav synchronizace s centrálním serverem. Stisknutí tlačítka manuálně vyvolá synchronizaci s centrálním serverem.

Pro správu aktivní pokladny a tiskárny obsahuje horní panel tlačítka pro výběr. Dále můžeme vidět informaci o přihlášeném uživateli. Při kliknutí na uživatelské jméno se zobrazí menu pro správu a odhlášení.



Obr. 33 Zadávání tržeb
Zdroj: vlastní zpracování – výsledný systém

Pro zadávání tržeb pomocí klávesnice je připravena obrazovka podporující funkce obdobné kalkulačce. Pokladník využívá čísla k zadání částek a definované klávesy k připsání položky do jednotlivých daňových kategorií. V pravé horní části se nachází celkový součet.

7 Závěr

Motivací této práce byl vývoj nového EET pokladního systému, který má nahradit stávající pokladní systém fungující v několika menších provozovnách. Hlavním důvodem byla nedostatečná schopnost správy dat a vzdáleného přístupu. Pro lepší představu o možnostech EET systémů jsme si představili některé existující EET systémy a jejich nejzajímavější funkce.

Vzhledem k tomu, že náš systém synchronizuje data mezi více zařízeními, představili jsme si problematiky distribuovaných systémů. Představili jsme možné způsoby časování, které jsou nutné pro synchronizaci, způsoby komunikace mezi zařízeními a vlastnosti distribuovaných systémů, které je nutné brát v potaz při vývoji. Nabyté informace byli využity především v návrhu systému a synchronizace.

Jedním z požadavků na cílový systém byla podpora elektronické evidence tržeb. Vzhledem k tomu jsme si uvedli základní legislativu a požadavky týkající se elektronické evidence. K lepšímu pochopení jsme si vysvětlili schéma komunikace mezi pokladní aplikací a technickým zařízením správce daně. Z technické dokumentace ([8]) jsme zjistili strukturu datové zprávy pro zaslání evidované tržby. Důležitá část se týká vytvoření bezpečnostních kódů a podepsání datové zprávy certifikátem poplatníka.

Z nabyté teorie byla navržena architektura systému. Byly stanoveny požadavky a jejich možná řešení. Při návrhu jsme vycházeli z funkcí moderních EET pokladních systémů. Velká část návrhu byla věnována především způsobu synchronizace.

Vzhledem k mým předchozím zkušenostem jsem zvolil použití open-source platformy Serenity. Tato platforma se osvědčila především v rychlosti vývoje, otevřenosti a práci s daty. Serenity připraví vývojáři mnoho funkcionalit pro CRUD operace, generování kódu na základě datového modelu a správy uživatelů a jejich oprávnění. Vzhledem k množství funkcí, které platforma poskytuje, jsem jí v této práci věnoval detailní představení. Zaměřili jsme se především na způsob objektově relačního mapování, které platforma využívá. Dále jsme si také představili generátor tříd z datového modelu a uživatelské rozhraní, které umožňuje operace s daty.

Vzhledem k velikosti výsledného systému byly v implementační části zmíněny pouze ty nejdůležitější části zdrojového kódu. Implementační část popisuje

postupný vývoj a formování výsledného systému. Pro lepší pochopení se věnuji i založení projektu a generování tříd pomocí generátoru platformy Serenity. Dále jsou popsány problematiky, které byly třeba překonat. Uvedeny jsou části zdrojového kódu použité k elektronické evidenci tržeb, k vyřešení tenancity a oprávnění a k řešení synchronizace mezi zařízeními.

7.1 Další vývoj

Výsledný systém byl otestován v testovacím provozu a prozatím není trvale nasazen v cílových prodejnách. Před plným nasazením bude nutné provést hloubkové testování tak, aby byla zajištěna spolehlivost a bezpečnost systému. Vzhledem k tomu je možné, že se na základě testování nebo na popud uživatelů výsledná podoba systému ještě změní.

V budoucnu by bylo dobré adoptovat některé další funkce, které můžeme vidět u konkurenčních řešení. Příkladem může být správa skladových zásob, správa produktů a jejich cen, rozhraní API pro propojení s dalšími systémy. Náš systém je na nové funkce velmi dobře připraven a jeho rozšíření by nemělo narušit stávající funkce.

Dalším velkým krokem by bylo rozšíření systému i na další platformy. Logicky se nabízejí dotyková přenosná zařízení.

7.2 Vyhodnocení

Práce popisuje všechny aspekty potřebné k vývoji výsledného systému. Přes množství uvedené teorie, bylo třeba navrhnout vlastní originální řešení, které není podobné žádnému jinému systému. Z toho důvodu může tato práce být námětem či inspirací při vývoji dalších distribuovaných systémů.

Výsledný systém splňuje hlavní cíle vytyčené v úvodu této práce. Zákazníkům jsou poskytnuty funkce pro práci s daty a vzdálenou správu. Vzhledem k robustnosti systému a jeho mnoha funkcím jsme splnili očekávání.

8 Seznam použité literatury

[1] G. Coulouris, J. Dollimore, T. Kindberg, and G. Blair, *Distributed Systems: Concepts and Design*, 5th ed. USA: Addison-Wesley Publishing Company, 2011, ISBN: 0132143011.

[2] 5nej.cz, "SROVNÁNÍ EET POKLADEN," 2020. https://www.5nej.cz/srovnani-eet-pokladen/.

[3] I. Polášková, "Porovnávací test a recenze EET pokladen 2020," 2020. https://www.arecenze.cz/eet-pokladny/.

[4] A. S. Tanenbaum and M. Van Steen, *Distributed systems: principles and paradigms*. Prentice-Hall, 2007, ISBN: 0132392275.

[5] D. Bermbach and J. Kuhlenkamp, "Consistency in distributed storage systems: An overview of models, metrics and measurement approaches," *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, vol. 7853 LNCS. pp. 175–189, 2013, doi: 10.1007/978-3-642-40148-0_13.

[6] J. Silcock, J. Silcock, and A. Goscinski, *Message Passing, Remote Procedure Calls and Distributed Shared Memory as Communication Paradigms for Distributed Systems*, vol. 28. Deakin University, School of Computing and Mathematics, 1995. Available: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.95.2490&rep=rep1&type=pdf

[7] L. Lamport, "Time, Clocks, and the Ordering of Events in a Distributed System," in *Communications of the ACM*, vol. 21, no. 7, 1978, pp. 558–565, doi: 10.1145/359545.359563.

[8] Finanční správa, "Elektronická evidence tržeb," 2016. https://www.etrzby.cz/cs/technicka-specifikace.

[9] Microsoft, "Závislosti a požadavky .NET Core," 2020. https://docs.microsoft.com/cs-cz/dotnet/core/install/dependencies?tabs=netcore22&pivots=os-windows (accessed May 26, 2020).

[10] C. Ireland, D. Bowers, M. Newton, and K. Waugh, "Understanding object-relational mapping: A framework based approach," *Int. J. Adv. Softw.*, vol. 2, no. 2, pp. 202–216, 2009, [Online]. Available: http://oro.open.ac.uk/19555/.

[11] E. O'Neil, "Object/Relational mapping 2008: Hibernate and the entity data model (EDM)," in *Proceedings of the ACM SIGMOD International Conference on Management of Data*, 2008, pp. 1351–1356, doi: 10.1145/1376616.1376773.

[12] V. Ceylan, "Serenity Developer Guide." https://play.google.com/books/reader?id=vjlECwAAQBAJ&hl=cs&pg=GBS.PP1.

9 Přílohy

1) Zdrojový kód implementovaného systému

Obsah adresářové struktury:

/ZEET.sln *– soubor řešení*
/ZEET/ZEET.Web/
- ***App_Data/*** *- adresář pro aplikační data (šablony, sqlite DB, certifikáty)*
- ***Initialization/*** *- třídy pro inicializaci (Startup.cs, Program.cs, …)*
- ***Migrations/*** *- migrace Fluent Migrator*
- ***Modules/*** *- adresář obsahující entity a většinu kódu funkcionality*
  - ***Administration/*** *- obsahuje entity administrativní části*
    - ***User/***
    - ***Role/***
    - ***Language/***
    - ***…***
  - ***EET/*** *- obsahuje entity EET části a přidružené funkcionality*
    - ***Tenant/***
    - ***Company/***
    - ***Store/***
    - ***Cashier/***
    - ***Income/***
    - ***…***
  - ***Common/*** *- obecné a generické funkcionality*
    - ***Helpers/***
    - ***UI/***
    - ***Behavior/***
    - ***…***
- ***Views/*** *- pohledy (.cshtml)*
- ***wwwroot/*** *- statické soubory pro UI (css, js, libs, překlady, …)*
- ***appsettings.json*** *– konfigurační soubor*
- ***ZEET.Web.csproj*** *– soubor projektu*

UNIVERZITA HRADEC KRÁLOVÉ
Fakulta informatiky a managementu
Akademický rok: 2019/2020

Studijní program: Aplikovaná informatika
Forma studia: Kombinovaná
Obor/kombinace: Aplikovaná informatika (ai2-k)

Podklad pro zadání DIPLOMOVÉ práce studenta

Jméno a příjmení:	**Bc. Martin Zezula**
Osobní číslo:	**I1800748**
Adresa:	Nad Strží 157, Trutnov – Volanov, 54101 Trutnov 1, Česká republika
Téma práce:	Distribuovaný pokladní EET systém
Téma práce anglicky:	Distributed EET cashier system
Vedoucí práce:	Ing. Pavel Kříž, Ph.D. Katedra informatiky a kvantitativních metod

Zásady pro vypracování:

Cíl: Navrhnout a implementovat distribuovaný multitenantní pokladní systém v .NET s využitím frameworku Serenity. Systém bude podporovat více pokladen v jedné firmě a bude automaticky synchronizovat veškeré nastavení, učivatelské účty a provedené platby. Na klientu poběží v běžném webovém prohlížeči.
Osnova:

1. Úvod
2. Distribuovaný systém
3. Návrh řešení
4. Framework Serenity
5. Implementace
6. Výsledky
7. Závěr

Seznam doporučené literatury:

https://serenity.is/docs/

Podpis studenta: Datum:

Podpis vedoucího práce: Datum: