Univerzita Hradec Králové

Fakulta informatiky a managementu

Katedra informačních technologií

Moderní webová aplikace založená na Spring Frameworku

(Webová aplikace pro elektronickou evidenci tržeb)

Diplomová práce

Autor: Pavel Komeštík

Studijní program: N1802 – Aplikovaná informatika

Studijní obor: Aplikovaná informatika – kombinovaná

Vedoucí práce: Ing. Pavel Kříž Ph.D.

Hradec Králové Duben 2017

Prohlášení:

Prohlašuji, že jsem Diplomovou práci vypracoval samostatně a že jsem v seznamu použité literatury uvedl všechny prameny, z kterých jsem vycházel.

V Hradci Králové dne 2.5.2017 Pavel Komeštík

Poděkování:

Děkuji vedoucímu bakalářské práce Ing. Pavlu Křížovi, Ph.D. za metodické vedení práce a velmi aktivní přístup při řešení vzniklých problémů.

Anotace

KOMEŠTÍK, P. *Moderní webová aplikace založená na Spring Frameworku.* Hradec Králové, 2017. Diplomová práce na fakultě informačních technologií Univerzity Hradec Králové. Vedoucí diplomové práce Ing. Pavel Kříž Ph.D. 75 stran.

Tato diplomová práce se zabývá tvorbou webové aplikace s využitím moderních webových technologií. Jako základní stavební prvky aplikace byly vybrány frameworky Spring a AngularJS. Práce popisuje implementaci jednotlivých uživatelských cílů aplikace za pomoci zmíněných frameworků a nastiňuje jejich praktický přínos při samotné tvorbě kódu.

Popisovaná aplikace je vytvořena jako nástroj pro správu objednávek v restauraci či podobné provozovně a zároveň umožňuje evidenci tržeb podle platného zákona. V samotném textu práce je tedy popsána jak tvorba aplikace, tak i způsob komunikace s portálem elektronické evidence tržeb.

Vytvářená aplikace využívá principu tzv. multitenance, jehož implementace je popsána v praktické části práce. Dále práce popisuje zabezpečení aplikace proti neoprávněnému přístupu k rozhraní serverové části.

V části věnované klientské části aplikace je popsáno využití frameworku AngularJS při tvorbě uživatelského rozhraní v prostředí webového prohlížeče. V závěru práce se autor vyjadřuje k nejnovějším možnostem webových aplikací, jako jsou progresivní webové aplikace, či využití nových API webových prohlížečů pro přístup k hardwaru.

Klíčová slova

Webová aplikace, Spring Framework, AngularJS Framework, elektronická evidence tržeb

Annotation

KOMEŠTÍK, P. *Moderní webová aplikace založená na Spring Frameworku.* Hradec Králové, 2017. Diplomová práce na fakultě informačních technologií Univerzity Hradec Králové. Vedoucí diplomové práce Ing. Pavel Kříž Ph.D. 75 p.

The diploma thesis deals with creating modern web application using current web technologies. Frameworks Spring and AngularJS were selected as a basic building blocks of the application. The work describes the implementation of the various use cases and outlines the practical benefit of used frameworks when coding.

Described application is designed as a tool for managing orders in a restaurant or similar establishment and allows electronic registration of sales under applicable law. The thesis describes the creation of the application and the way how to communicate with the portal of electronic registration of sales.

The application uses the principle of so-called "Multi-tenant", which implementation is explained in the practical part. It also describes the application security to prevent unauthorized access to the REST interface on server side.

The last part is dealing with user interface in the web browser and the framework AngularJS is described. In conclusion, the author expresses the latest capabilities of Web applications, such as progressive web applications, or use of the new API in web browsers to access the hardware.

Keywords

Web application, Spring Framework, AngularJS Framework, electronic evidence of sales

Obsah

Seznam obrázků

Seznam ukázek kódu

1 Úvod

Vývoj aplikací v dnešní době klade velké nároky na jejich tvůrce především v oblastech škálování aplikace, její rozšiřitelnosti, udržovatelnosti a možnosti běhu na různých platformách. Tyto požadavky jako i některé další jsou důvodem pro vznik různých nových přístupů, nástrojů a frameworků, které mají jejich naplnění usnadnit.

Prvním zásadním krokem na cestě k modernímu způsobu programování byl bezpochyby přechod od procedurálního programování k objektovému paradigmatu. Objektový přístup nám přinesl jiný pohled na strukturu programu a umožnil tvorbu složitých aplikací, které přitom zůstávají dobře udržovatelné a rozšiřitelné. Tato změna však přinesla také nutnost podrobně pochopit děje a vztahy mezi objekty, které v dané doméně existují. Z tohoto důvodu vznikly další přidružené oblasti jako například objektové modelování.

Ve světě objektového modelování je struktura programu chápána jako zjednodušený obraz skutečného světa a jednotlivé objekty, vystupující v dané doméně jsou převáděny na třídy v kódu. Instance těchto tříd pak mají všechny relevantní znaky a vlastnosti svých předloh.

Hlavní principy objektového přístupu jako jsou například dědičnost, zapouzdření a polymorfismus tak pomohly ke vzniku nástrojů, které tvorbu kódu výrazně zjednodušují. Tyto nástroje mají většinou za cíl transparentně poskytnout funkcionalitu, která je stejná pro mnoho aplikací a umožnit tak programátorovi soustředit se výhradně na psaní produkčního kódu. Hlavními zástupci takovýchto nástrojů jsou především aplikační frameworky.

Aplikační frameworky nám tedy umožňují za pomoci objektového přístupu splnit nároky dnešních aplikací na rozšiřitelnost, udržovatelnost a škálování. Dalším důležitým prvkem při návrhu moderní aplikace je však také přenositelnost na různé platformy. S tímto problémem je možné se vypořádat různými způsoby. Původním řešením byla kompilace zdrojového kódu pro každou cílovou platformu, avšak toto řešení vyžaduje tvorbu různých kompilátorů, jejichž vývoj rozhodně není snadný. Dále existuje možnost využití takzvaných interpretovaných programovacích jazyků, kdy na cílové platformě běží interpret daného jazyka. Typickým příkladem je jazyk Java a jeho JVM (Java Virtual Machine).

V dnešní době se v této oblasti dostává do popředí také možnost využití prostředí webových prohlížečů. S jejich pomocí je totiž možné provozovat jeden kód na mnoha platformách, bez jakékoli úpravy. Nezanedbatelnou výhodou je také fakt, že pro změny v aplikaci není nutná distribuce ke každé instanci zvlášť. Samotná klientská aplikace může být načtena ze serveru provozovatele kdykoli je to nutné.

Webové prohlížeče se v dnešní době snaží poskytnout podporu pro provoz aplikací i v off-line režimu a také umožnit lepší přístup k prostředkům samotného zařízení. Programy napsané v jazyce Javascript tak mohou fungovat zcela nezávisle na internetové síti, a zároveň realizovat propojení s aplikačním serverem. Stejně jako při tvorbě běžných aplikací i zde existuje řada různých frameworků, které tvorbu webových aplikací usnadňují.

Tato práce se bude snažit demonstrovat možnosti využití moderních nástrojů dostupných pro programovací jazyky Java a Javascript při tvorbě webové aplikace komunikující s aplikačním serverem. Budou zde popsány hlavní přínosy uvedeného přístupu a na příkladech ukázány některé implementační detaily.

2 Cíl práce a způsob řešení

Hlavním cílem této diplomové práce je vytvoření moderní webové aplikace pro správu zboží a elektronickou evidenci tržeb s využitím vhodných postupů a nástrojů, které umožní splnit aktuální požadavky na bezpečnost, udržovatelnost, rozšiřitelnost, škálovatelnost a přenositelnost vytvořené aplikace.

Pro naplnění těchto požadavků byly zvoleny následující postupy a nástroje:

- Pro analýzu a návrh bude použit jazyk UML (Unified Modeling Language) a UP (Unified Process).
- Při implementaci bude použit přístup programování řízeného testy (Test Driven Development).
- Serverová část aplikace bude vytvořena v programovacím jazyce Java s využitím prostředků frameworku Spring.
- Klientská část aplikace využije pro svůj běh prostředí webového prohlížeče a bude napsána v jazyce Javascript. Zde bude využit framework AngularJS a pro grafickou část pak Angular Materials.

3 Analýza stávajících řešení

V době vzniku původního záměru, kdy ještě neexistovala elektronická evidence tržeb, měla vyvíjená aplikace poskytovat především služby pro správu zboží a vytváření objednávek v běžné restauraci. Ještě před několika lety, a především před příchodem elektronické evidence tržeb, byla v této oblasti velmi slabá konkurence a nabízená řešení měla velice daleko k dnešní představě o ergonomii a uživatelském komfortu. Existovalo pouze několik málo řešení nabízejících například funkce pro mobilního číšníka či vzdálenou správu. Také grafická stránka těchto aplikací byla velice zastaralá a některá z řešení fungovala dokonce pouze v textovém režimu, tak jak to dnes ještě můžeme vidět například na pokladnách v některých obchodních řetězcích.

Původním obchodním záměrem tedy bylo zaplnit tuto mezeru na trhu modernějším a kvalitnějším produktem za výrazně nižší cenu.

S příchodem elektronické evidence tržeb se však otevřel obrovský trh pro podobné softwarové nástroje, což vedlo ke vzniku velkého množství dostatečně kvalitních produktů za přijatelné náklady.

Bylo tedy nutné zamyslet se nad novým záměrem, který by odpovídal změněné situaci a bylo třeba nastínit vlastnosti současných řešení a jejich cenové relace.

V současnosti nabízené produkty lze z hlediska jejich softwarového řešení rozdělit do dvou kategorií, a to na produkty fungující jako webové služby a potom na klasické desktopové programy.

3.1 Webové služby

U dostupných webových služeb pro elektronickou evidenci tržeb jde především o nástroje nabízející pouze možnost odeslání jednotlivé tržby na portál správce daně, a to vyplněním formuláře na webové stránce. Tyto služby tak nenabízí možnost správy zboží, či jakoukoli další funkcionalitu a jsou většinou poskytovány zdarma.

- **Harmonik**

Tato služba je dostupná na webové stránce harmonik.cz a nabízí zdarma odeslání tržby pomocí webového formuláře.

- **Fiket**

Nabízí odeslání tržby ve třech krocích opět pomocí formuláře zdarma na webu fiket.cz

- **Info Office**

Tato společnost nabízí webovou aplikaci pro elektronickou evidenci tržeb, a to za cenu přibližně 600 Kč za jeden rok jejího používání. Aplikace umožňuje registraci jednoho obecného typu zboží, u kterého se případně mění cena pro každou tržbu. Aplikace není bez připojení k internetu funkční.

3.2 Desktopové aplikace

Dostupné desktopové aplikace pro elektronickou evidenci tržeb nabízí již daleko širší možnosti, než je tomu u webových služeb. Těchto aplikací existuje opravdu velké množství a do následujícího přehledu je vybráno jen několik reprezentativních případů.

- **Markeeta**

V popisu uvedeném na webové stránce produktu (markeeta, 2017) je uvedeno především, že produkt nabízí rychlou a jednoduchou obsluhu bez prodlev, kompletní řešení EET bez paušálu za nejnižší cenu na trhu, nadstandardně dlouhou záruku 24 měsíců a především diskrétnost.

K dalším výhodám patří konfigurátor pokladny, díky němuž zákazník dostane přesnou nabídku podle oboru svého podnikání. Zároveň si může vybrat, zda si pokladní program rovnou celý koupí, nebo se rozhodne pro paušální poplatek.

Cenově je nabídka rozdělena podle nabízené funkcionality do tří balíčků.

Balíček *Basic* nabízí 150 položek zboží, 2 uživatele, off-line provoz, podporu 24/7, kategorie zboží, vzdálenou správu, zálohy, přehledy, cizí měny. Cena je 290 Kč za měsíc.

Balíček *Standard* přidává pokladní knihu, otevřené účty a skladové hospodářství za cenu 550 Kč za měsíc.

Poslední Balíček *Pro* navíc nabízí recepty, grafické rozložení stolů a docházku za cenu 850 Kč za měsíc.

- **KASA FIK**

Dalšího zástupce desktopových systémů nabízí společnost KASA FIK. Z popisu produktu na webu prodejce (KASA FIK, 2017) vyplývá především, že u tohoto produktu neplatí zájemce paušální platby, ale jednorázově produkt zakoupí. Funkcionalita je opět rozdělena do cenově odstupňovaných balíčků.

Balíček *Free* umožní především provádět storna, zobrazit historii účtenek, odeslat účtenku emailem, ručně odeslat neodeslané platby a provoz v off-line režimu. Tento balíček je nabízen zdarma.

Balíček *Klasik* navíc nabízí 200 položek zboží, evidenci příjmů a výdajů, otevřené objednávky, platbu kartou, uzávěrky, podporu cizích měn, záloha do souboru nebo on-line a vzdálenou správu. Cena tohoto balíčku je 3999 Kč.

Balíček Plus nabízí navíc neomezené množství položek zboží, evidenci zákazníků, přihlášení až 3 zaměstnanců, omezení uživatelských práv a skladové hospodářství, a to za cenu 4999 Kč.

- **O2 eKasa**

Posledním vybraným zástupcem je produkt od společnosti O2 nazvaný eKasa Air. Teno produkt je nabízen ve dvou variantách.

O2 Mobilní eKasa Air pro mobilní telefon s operačním systémem Android a úhlopříčkou do sedmi palců s možnostmi pro vzdálenou správu, připojení platebního terminálu, podporu bluetooth tiskárny, vytvoření seznamu zboží, přidání slevy a akční nabídky. Tato varianta je nabízena za cenu 349 Kč měsíčně.

O2 eKasa Air pro tablet s operačním systémem Android a úhlopříčkou od sedmi palců výše. V této variantě je navíc nabízeno skladové hospodářství, připojení čtečky čárových kódů, podpora cizích měn, mapa stolů a seznam a výběr stálých zákazníků. Cena této varianty je 449 Kč měsíčně.

3.3 Obchodní záměr

Z provedené analýzy nabízených produktů je patrné, že prostor pro kvalitní webovou službu pro elektronickou evidenci tržeb na trhu stále existuje, a to především díky malé funkcionalitě dostupných webových řešení a vysoké ceně těch desktopových. Obchodním záměrem vytvářené aplikace je tedy nově vytvoření produktu nabízejícího funkcionalitu blížící se desktopovým aplikacím za výrazně nižší cenu.

4 Teoretická východiska

Při tvorbě popisované aplikace byl pro serverovou část zvolen programovací jazyk Java a framework Spring, zatímco pro klientskou aplikaci jazyk Javascript a framework AngularJS. V této části se práce bude zabývat především možnostmi těchto nástrojů a nastíní také vlastnosti jejich hlavních konkurentů. Na konci této kapitoly bude popsán způsob fungování elektronické evidence tržeb a nástroje použité při implementaci této funkcionality.

4.1 Technologie serverové části aplikace

Pro tvorbu moderní aplikace je třeba využít nejnovějších postupů a poznatků v oblasti objektového modelování a také nalézt nejvhodnější přístupy a techniky pro samotnou implementaci. V následujících odstavcích budou popsány hlavní požadavky na programovací jazyk serverové části aplikace a jeho případná rozšíření, které umožní naplnění stanovených cílů práce. Dále pak samotný programovací jazyk Java a framework Spring.

4.1.1 Hlavní požadavky na programovací jazyk serverové části

Tyto požadavky jsou hlavními nikoli však jedinými nároky pro výběr programovacího jazyka pro serverovou část aplikace. Ve skutečnosti byl samotný výběr jazyka Java ovlivněn především dlouholetými zkušenostmi programátora s tímto jazykem.

- **IoC a Dependency Injection**

Podle (Schaefer, a další, 2014) je IoC (Inversion of Control) technikou pro vytváření a správu závislostí komponent.

Tradičně pokud nějaká komponenta programu potřebuje k naplnění svých závazků instanci jiné třídy, pak tuto buď vytvoří použitím klíčového slova *new*, nebo odkaz na ní získá pomocí továrny objektů tzv. *object factory*. Při použití dependency injection je potřebná instance dodána při běhu programu nějakým externím procesem. Toto chování, vkládání závislostí za běhu programu, vedlo k tomu, že Martin Fowler přejmenoval IoC na mnohem více výstižné dependency injection.

Pro tvorbu moderní aplikace je podpora dependency injection stěžejní, protože pomáhá naplnit hned dva ze zmíněných cílů této práce. Těmito dvěma cíli jsou udržovatelnost a rozšiřitelnost. Díky dependency injection je možné vytvářet velmi volné vazby mezi komponentami a tím zjednodušit případnou výměnu jejich implementace a také je třeba méně obslužného kódu pro vytváření závislostí, což kód zpřehledňuje a umožňuje programátorovi soustředit se pouze na programování produkční funkcionality.

- **ORM**

Objektově relační mapování je technika umožňující transparentní mapování objektů v programu na záznamy v databázi. S pomocí ORM je tak programátor odstíněn od nutnosti psát kód pro komunikaci s databází a o propojení dat a objektů v programu se stará ORM framework. Takovýchto frameworků existuje v jazyce Java celá řada.

V Java Enterprise Edition existuje koncept tzv. Entity Beans, které ORM zajišťují, vlastní implementace nabízí například frameworky Hibernate či EclipseLink. V Javě navíc vznikla specifikace JPA (Java Persistence API), která se stává standardem pro ORM v jazyce Java a zmíněné frameworky toto abstraktní API implementují a rozšiřují.

Hlavním přínosem pro naplnění cílů práce je zde opět především snížení množství obslužného kódu a tím zpřehlednění aplikační logiky.

- **AOP**

Aspect Oriented Programming (AOP) je jedním z moderních programovacích postupů. Využitím IoC a proxy objektů či Java Reflection API je možné vynutit vykonání kódu obalujícího volání jakékoli metody jakéhokoli objektu. Tato technika umožňuje velice efektivně (z pohledu struktury a množství kódu) provádět opakované činnosti, které by jinak bylo nutné implementovat na mnoha místech aplikace. Typickým příkladem zde může být transakční zpracování, či logování chování aplikace.

AOP je při tvorbě samotných aplikací používáno jen zřídka a pro aplikační logiku je používání aspektů z důvodu nepřehlednosti přímo nevhodné. Na druhé straně pro aplikační frameworky, jakým je například framework Spring, je AOP stěžejní. Velké množství funkcionality, kterou tento framework nabízí je na AOP přímo závislé.

- **Convention over Configuration**

Udržovatelnost a rozšiřitelnost aplikace z velké části definuje její rozsah a složitost. Moderní aplikační frameworky se tedy snaží mít co nejmenší nároky na konfiguraci s cílem minimalizovat množství kódu nutného k jejich použití.

Convention Over Configuration je designové paradigma, které říká, že množství rozhodnutí, které musí programátor udělat pro použití frameworku by mělo být minimální, ale přesto si nastavení musí zachovat potřebnou flexibilitu. V praxi to znamená, že framework by měl obsahovat sadu výchozích hodnot nastavení a programátor by se o nastavení měl zajímat až ve chvíli, kdy cítí potřebu něco na chování frameworku změnit.

4.1.2 Programovací jazyk Java

Za základní nástroj pro vývoj serverové části aplikace byl zvolen programovací jazyk Java. Tento jazyk ve spojení s frameworkem Spring nabízí potřebné nástroje pro dosažení všech stanovených cílů této diplomové práce.

- **Historie jazyka**

Tato část byla zpracována s využitím (Schildt, 2012 stránky 22-25).

Programovací jazyk Java navazuje na bohaté dědictví jazyků C a C++ a přidává jim rysy a vlastnosti odrážející aktuální stav v oblasti programování. V reakci na vývoj on-line prostředí se jazyk Java zaměřuje na poskytnutí nástrojů k efektivnímu programování pro vysoce distribuovanou architekturu. Jazyk Java vymysleli v roce 1991 James Gosling a jeho spolupracovníci pod původním názvem „OAK". Tento název byl autory změněn v roce 1995. Hlavní motivací pro vznik jazyka byla potřeba softwaru nezávislého na platformě, pro použití v nejrůznějších zařízeních spotřební elektroniky. V takovéto elektronice se totiž používá mnoho typů výpočetních jednotek a jazyk C++ vyžadoval pro každý typ vlastní kompilátor. Tvorba takového kompilátoru je však drahá a časově náročná, proto tvůrci Javy hledali lepší řešení.

V době, kdy se pracovalo na detailech Javy se vynořil další aspekt, který hlavně stojí za jejím prudkým rozvojem a tím byla celosvětová webová síť. Samotný web totiž také vyžadoval přenositelné programy, a tak se jazyk Java původně určený do dálkových ovládání a topinkovačů dostal do popředí mezi hlavními programovacími jazyky.

- **Vývoj Javy**

Tato část byla zpracována s využitím (Schildt, 2012 stránky 15-18).

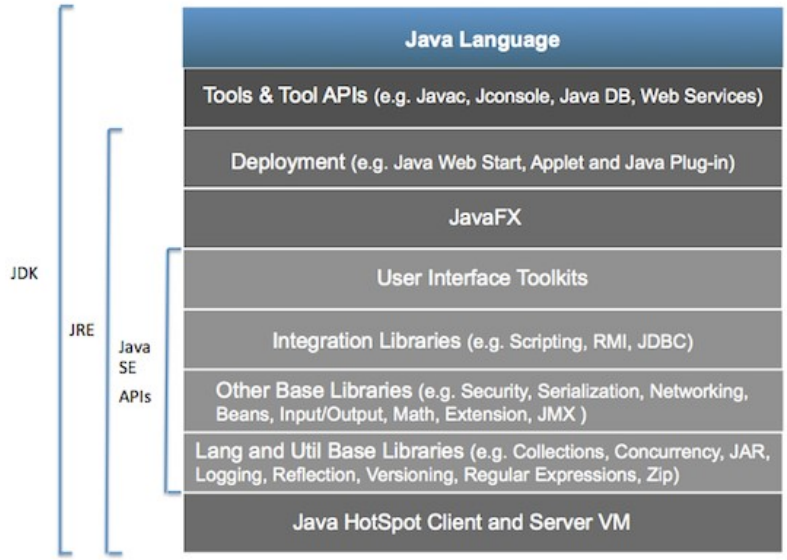
První oficiální verze 1.0 vyšla v roce 1996. Další verze 1.1 vyšla o rok později a přidala do jazyka Java podporu vnořených tříd, reflexi a Unicode verze 2.0. Dále pak JDBC, Java RMI (Remote method invocation), rozšíření AWT a JIT (Just In Time) kompilátor pro Microsoft Windows. V roce 1998 se změnilo pojmenování na Java 2 a název verze na J2SE. Část názvu SE od této chvíle slouží pro odlišení základní platformy od J2EE (Enterprise Edition) a J2ME (Micro Edition). Dále došlo v průběhu vývoje k implementaci mnoha dalších vlastností reflektujících aktuální stav programovacích jazyků. K nejrevolučnějším změnám nicméně došlo především v roce 2004 ve verzi 5.0 (1.5). Zde bylo změněno číslování verzí z 1.5 na 5.0 a došlo ke generačnímu skoku v jazyce. Od verze SE 6 se v označení vypouští druhá číslice a oficiální název je tedy Java SE 6. Java SE 7 začlenila mimo jiné upgrady běhového prostředí systému, který nyní podporuje i jiné jazyky než jen Javu.

- **Současný stav**

V roce 2014 přinesla Java SE 8 vybrané prvky z funkcionálního programování jako jsou lambda funkce či proudové zpracování dat. V průběhu roku 2017 má dojít k vydání verze Java SE 9, která přinese podporu pro reaktivní programování.

- **Java Standard Edition (SE)**

Na následujícím obrázku je možné sledovat rozdělení jednotlivých technologií zastoupených v Java SE.



Obrázek 1 - Koncepční diagram Java SE, Zdroj:
http://www.oracle.com/technetwork/topics/newtojava/documentation/index.html

JRE nabízí knihovny, virtuální stroj Javy a další komponenty nezbytné pro běh aplikací napsaných v jazyce Java. Toto běhové prostředí může být distribuováno společně s aplikací, což takovou aplikaci činní nezávislou (Oracle Company a, 2017).

JDK obsahuje JRE současně s dalšími nástroji pro vývojáře jako překladače a ladící programy, které jsou užitečné pro vývoj aplikací v jazyce Java (Oracle Company a, 2017).

- **Java Enterprise Edition (EE)**

Java Enterprise Edition (Java EE) je standardem v oblasti tvorby webových a podnikových aplikací. Nabízí nové prostředky pro rozšíření podpory HTML5, zvyšuje produktivitu vývojáře a vylepšuje způsoby, kterými je možné naplnit podnikové cíle. Vývojářům nabízí prostředky pro tvorbu lépe rozšiřitelných

aplikací, umožňuje jim psát méně kódu, který nesouvisí s business logikou a má lepší podporu pro webové aplikace a frameworky (Oracle Company b, 2017).

- **Java ME**

Java Micro Edition se používá hlavně pro embedovaná a mobilní zařízení a podle webu společnosti Oracle je použita v 5ti bilionech SIM a Smart karet, 3 bilionech mobilních headsetů, 80ti milionech televizních přijímačů, a mnoha dalších. Dělí se na Java SE Embedded pro zařízení s 32 MB paměti, Java ME Embedded pro zařízení s 8 MB paměti a Java Embedded Suite pro zařízení připojující se k databázi.

Souhrnně lze říci, že v současné době je jazyk Java jedním z nejrychleji se rozvíjejících programovacích jazyků a obsahuje všechny nástroje potřebné pro tvorbu moderních aplikací.

4.1.3 Java Enterprise Edition

Tato část byla zpracována s využitím (Oracle Company, 2016) a (Parsons, 2012).

Java Enterprise Edition se snaží přinášet vývojářům silné API, které jim pomáhá k snadnějšímu a rychlejšímu vývoji výkonných aplikací za současného snížení komplexnosti tohoto vývoje. Java EE používá zjednodušený programovací model. Konfigurační XML jsou nepovinné a vývojář může jednoduše vložit informace o konfiguraci jako anotaci přímo do kódu. Java EE aplikační server pak nakonfiguruje komponenty přímo při nasazování aplikace a za jejího běhu.

- **Historie**

Java Professional Edition (předchůdce JEE) byla oznámena společností Sun v roce 1998. První verze pro obecné použití byla verze 1.2, která již obsahovala podporu pro EJB (Enterprise JavaBeans), JTA (Java Transaction API), RMI (Remote Method Invocation) a další. Většina těchto technologií přetrvala ve svých vylepšených verzích v Java EE dodnes. Verze 1.3 přinesla nová vylepšení jako například JAXP (Java API foe XML Processing). Pro podporu webových služeb pak vylepšenou verzi Java Servlet a novinky JSP (Java Server Pages) JSTL (Java Server Pages Standard Tag Library) a také JAAS (Java Authentication and Authorization Service). V dalších verzích byla dále vylepšována podpora webového prostředí jako odpověď na jeho rychlý rozvoj. Rozšířena byla podpora formátu XML a také přišla alternativa pro JSP v podobě JSF (Java Server Faces).

Původní verze Javy EE však postupem času začaly mít potíže se stížnostmi na velikost celého frameworku a nutnost použití všech částí v každém projektu, i když nebyly třeba. Takový framework se nazývá anglickým výrazem „heavyweight" framework. Opakem jsou pak modulární tzv. „lightweight"

frameworky jako Spring, Struts či Hibernate, které se stávali stále populárnějšími a pod jejich tlakem muselo dojít ke změně v přístupu a použití obecnějších API jako například JPA místo původních Entity Beans. Došlo také ke změně číslování verzí a z původní Java EE 1.5 se stala Java EE 5.

Od verze Java EE 6 je zde patrná snaha o standardizaci a propojení konceptů a funkcí s frameworkem Spring. Toto vedlo k otevření možností pro vývojáře třetích stran k vytváření nezávislých implementací jednotlivých Java Specifikací.

Aktuální verze Java EE 7 vydaná v červnu roku 2013 pokračuje v trendech nastavených v Java EE 6. Rozšiřuje tedy možnosti vývojářů při výběru vhodné implementace jednotlivých Java specifikací a zaměřuje se na webové prostředí.

4.1.4 Spring framework

Ve své podstatě je Spring framework souborem užitečných open source projektů sdružených v jednom modulárním frameworku. Tento kompilát však nemá za cíl nahradit Javu EE, jako spíše existovat vedle ní.

Jak uvádí (Schaefer, a další, 2014) je Spring obvykle popisován jako typický *lightweight* framework pro vytváření aplikací v Jazyce Java. Tento na první pohled stručný popis vystihuje hned dvě velmi zásadní funkcionality frameworku Spring.

Za prvé, Spring může být použit při tvorbě jakékoli aplikace v jazyce Java. Je tedy možné ho použít při tvorbě samostatně spustitelné aplikace stejně dobře, jako aplikace pro web, či Java EE aplikace.

Za druhé, Spring jako zástupce modulárních frameworků nám umožňuje použít vždy jen přesně ty knihovny, které aktuálně potřebujeme. Jedinou povinnou částí je samotný IoC kontejner obsažený v Spring Core.

Pro splnění cílů této práce je také velmi důležitá další z vlastností tohoto frameworku a tou je jeho neinvazivnost. Podle (Schaefer, a další, 2014) neinvazivnost znamená, že pro využití benefitů, které nám Spring Core přináší jsou třeba minimální či vůbec žádné změny v samotném kódu aplikace. Zde je však třeba upozornit, že toto se týká především samotného Spring Core a další komponenty jako například Spring Data Access vyžadují daleko těsnější provázání.

- **Historie**

Tato část práce byla zpracována s využitím (Schaefer, a další, 2014).

První veřejné vydání frameworku Spring byla verze 0.9, která byla vytvořena na základě knihy: Expert One-on-One: J2EE Design and Development napsané

Rodem Johnsonem v roce 2002. Spring Core, Spring Context, Spring DAO, Spring ORM, Spring AOP, Spring Web a Spring Web MVC, tedy hlavní komponenty použité pro splnění cílů této práce, byly součástí frameworku již od verze 1.

Další vývoj však velmi usnadnil jejich použití především standardizací jednotlivých anotací podle JSR. Verze 2 přinesla podporu pro Java Persistence API (JPA), což je abstraktní vrstva pro ORM. A verze 2.5 pak nové anotace podle JSR-250. Verze 3.0 přinesla podporu pro novinky v Java 5 jako jsou generické typy, varargs a další. Tato verze také přinesla z pohledu této práce velmi důležitou podporu pro REST rozhraní a validace pomocí anotací z JSR-303. Zatím aktuální verzí je verze 4.0, která přináší podporu pro novinky z Java 8 a dále vylepšuje a rozšiřuje možnosti samotného frameworku.

- **Spring Core**

Samotné jádro frameworku Spring je samo o sobě skvělý nástroj přinášející především podporu pro dependency injection, nicméně samotné jádro tomuto frameworku také umožňuje snadnou integraci s velkým množstvím další funkcionality. Spring tak může přinášet prostředky pro všechny vrstvy aplikace, a přitom stále zůstat dostatečně modulárním.

- **Spring ORM**

Přístup k datům a objektově relační mapování jsou ve světě Javy velmi diskutovanými tématy. Spring nabízí podporu pro standardní JDBC prostřednictvím zjednodušeného API, ale především pro pokročilé metody ORM prostřednictvím Hibernate, JDO a JPA. Pro zpracování dat je také velmi důležitým prvkem správa transakcí. Spring nabízí podporu jak pro imperativní, tak deklarativní kontrolu transakcí.

- **Spring MVC**

Spring nabízí širokou podporu také v oblasti programování pro web. Spring MVC je framework využívající návrhový vzor MVC (Model View Controller). Nabízí široké možnosti pro jednoduchou konfiguraci předávání dat z modelu do webové prezentační vrstvy aplikace. Použitím tohoto frameworku získá programátor rozsáhlou flexibilitu při výběru implementace frontendu aplikace. Na této vrstvě je možné využít jak JSP (Java Server Pages) a JSTL (Java Standard Tag Library), tak další frameworky prezentační webové vrstvy, jako jsou například Apache Velocity, FreeMaker, Apache Tiles a XSLT. Další nabízenou možností je pak místo samotné implementace prezentační vrstvy na straně serveru využít stále populárnější rozhraní REST, či SOAP.

- **Spring Boot**

Projekt Spring Boot původně vznikl jako nástroj pro snadné spuštění ukázkových aplikací vytvořených ve frameworku Spring. Tento svůj účel plní i nadále avšak dnes je samotnými tvůrci označován za plně produkční řešení (Spring, 2016). Spring Boot přináší do Spring frameworku koncept „Convention over Configuration“ a odbourává potřebu explicitní konfigurace prostředí pro spuštění aplikace.

Spring Boot společně se svými startovacími balíčky obsahuje všechny potřebné nástroje a prostředky pro samostatný provoz aplikace, a to v případě použití Spring MVC i včetně samotného aplikačního serveru. Celé prostředí je implicitně nastaveno tak, aby bylo možné aplikaci okamžitě spustit, bez potřeby cokoli konfigurovat. Jakékoli nastavení však může programátor kdykoli upravit podle vlastního uvážení a není oproti běžnému použití v tomto nijak omezován.

- **Validace a JSR-303**

Validace atributů jednotlivých entit obsahujících aplikační data je jedním z velkých bodů v každé aplikaci. Je vhodné, aby validační pravidla byla uplatněna vždy, nezávisle na tom, zda je zdrojem požadavku na manipulaci s nimi backend či frontend aplikace. Jako řešení tohoto problému nabízí Spring vlastní validační API prostřednictvím interface *Validator*.

V rámci JCP (Java Community Process) byla také navržena specifikace pro validaci objektů pod JSR-303. Spring v základním nastavení používá validátor Hibernate, který má plnou podporu pro tuto specifikaci.

4.2 Technologie klientské části aplikace

V této části budou uvedeny hlavní technologie použité při tvorbě klientské aplikace v prostředí webového prohlížeče. Jedná se tedy především o programovací jazyk Javascript a jeho frameworky, prostředky prohlížeče pro off-line aplikace a knihovnu Forge použitou při implementaci požadavků na šifrování položek zpráv systému pro elektronickou evidenci tržeb.

4.2.1 Programovací jazyk Javascript

Jazyk Javascript se poprvé objevil roku 1995 a jeho hlavním účelem bylo ověřování vstupů od uživatele na straně webového prohlížeče (Zakas, 2009).

Podle (Zakas, 2009) se před rokem 2001 velká skupina programátorů domnívala, že jazyk Javascript není dostatečně seriózní. Avšak mnozí z těch, kteří dospěli až k plnému porozumění tohoto jazyka, došli k názoru, že chyba je spíše na straně špatných implementací ve webových prohlížečích.

Po roce 2001 se však situace ohledně nekvalitních implementací již natolik vylepšila, že většina předchozích námitek mohla být zapomenuta. Weboví vývojáři začali tedy naplno využívat výhod, které jim Javascript přináší. V této době začal být vývojáři také objevován objekt *XMLHttpRequest* ležící v srdci technologie Ajax a v prohlížečích se objevilo nové paradigma interakce uživatelů ve stylu desktopových aplikací.

Podle (Zakas, 2009) jsou termíny Javascript a ECMAScript často nesprávně zaměňovány. Jazyk Javascript je ve své komplexnosti daleko více, než co definuje standard ECMA-262. Tento standard je sice jádrem jazyka Javascript, nicméně kompletní implementaci Javascriptu doplňují ještě modely dokumentu (DOM) a prohlížeče (BOM).

Jazyk Javascript a model dokumentu (DOM) poprvé umožnili změnu vzhledu stránky bez jejího načtení ze serveru, což způsobilo jistou revoluci v chápání webových aplikací a nakonec dalo vzniknout frameworkům pro single-page aplikace jakým je například i framework AngularJS.

4.2.2 Framework AngularJS

Tato část práce byla zpracována s využitím (Smith, 2015).

AngularJS framework je nástrojem pro vytváření single-page aplikací podle vzoru model-view-controller (MVC) v prostředí webového prohlížeče. Single-page aplikace mají za cíl přinést uživateli podobný pocit, jaký má při používání čistě desktopových aplikací. Využito je přitom především možností jazyka Javascript a prostředků HTML.

S pomocí tohoto frameworku je tedy možné vytvořit ve webovém prohlížeči uživatelskou aplikaci, která sama provádí veškeré renderování HTML kódu stránky a se serverem komunikuje pomocí REST rozhraní. Tento přístup přináší výhodu v podobě přenesení velké části výpočetních nároků aplikace na klienta. Framework je tedy vhodný pro prostředí, kde se počítá s velkým počtem aktivních klientů.

AngularJS navíc přináší do jazyka Javascript některé nové prvky objektově-orientovaných jazyků jako je například dependency injection. Veškerá aplikační logika je zde tvořena tzv. moduly, které jsou buď součástí frameworku, nebo byly vytvořeny programátorem pro konkrétní aplikaci. Tyto moduly mohou mezi sebou vytvářet vazby pomocí dependency injection, což napomáhá organizaci celého kódu a přináší přehlednější strukturu aplikace.

Framework AngularJS navíc přináší další nástroje pro práci s HTML kódem a objektovým modelem dokumentu (DOM). Datový model aplikace je synchronizován se zobrazením v rámci tzv. digest cyklů. Během každého z těchto cyklů dojde ke zjištění změn v datovém modelu aplikace a tyto změny jsou přeneseny do zobrazení.

Při tvorbě samotné aplikace byl použit framework AngularJS ve verzi 1.6. Je třeba zmínit, že existuje novější nástupce tohoto frameworku pod názvem Angular v současné verzi 4.0. Tento framework značně mění celé pojetí původního AngularJS a prozatím se netěší takové oblibě mezi vývojáři jako původní verze. Oba tyto frameworky jsou prozatím vyvíjeny současně.

4.2.3 Angular materials

Jak uvádí oficiální web projektu (Google, 2017), je Angular Materials zároveň frameworkem pro vytváření uživatelských rozhraní a také implementací specifikace Material Design od společnosti Google. Tento projekt přináší soubor znovupoužitelných, dobře otestovaných a přístupných komponent, které jsou pomocí direktiv frameworku AngularJS převáděny do běžného HTML kódu a vytvářejí tak obrazové prvky uživatelského rozhraní.

Použití tohoto frameworku ve vytvářeném projektu přináší významné zjednodušení práce při tvorbě uživatelského rozhraní a také zvýšení produktivity výsledného SW právě z důvodu známého vzhledu a rozložení prvků uživatelského rozhraní.

4.2.4 Service Worker API

Podle (Mozilla, 2017) vystupuje service worker v roli proxy objektu zařazeného mezi webovou aplikaci v prohlížeči a samotnou síť. Cílem tohoto API je mimo jiné umožnit efektivní vytváření aplikací nabízejících širokou off-line funkcionalitu.

Služba service worker umožňuje převzít kontrolu nad jakýmkoli síťovým požadavkem a provést akce nutné pro udržení funkčnosti aplikace i v případě nedostupnosti připojení k síťovému serveru.

4.2.5 PouchDB

Podle (Harvey, a další, 2017) je PouchDB databáze napsaná v jazyce Javascript, umožňující tvorbu aplikací běžících v prostředí webového prohlížeče, které jsou funkční i bez připojení k síti. PouchDB je implementací databáze CouchDB v jazyce Javascript snažící se co nejvíce napodobit CouchDB API v prostředí webové aplikace.

Nerelační databáze PouchDB je podle (Slater, 2014) skvělým nástrojem pro replikaci dat mezi vzdáleným zdrojem a prohlížečem.

Tato databáze umožní na straně klienta práci s daty v off-line režimu a také s pomocí CouchDB na straně serveru snadnou synchronizaci dat mezi klienty. Při využití tohoto přístupu je třeba vzít také v úvahu fakt, že všechny instance databáze nemusí v každé chvíli obsahovat přesně stejná data, a to z toho důvodu, že k synchronizaci dojde pouze, pokud má konkrétní klient dostupné připojení k aplikačnímu serveru.

4.2.6 Knihovna Forge

Pro zajištění potřebných kryptografických funkcí na straně klienta bude v projektu použita open source knihovna Forge, která je podle svého popisu (digitalbazaar, 2017) nativní implementací TLS (Transport Layer Security) v jazyce Javascript.

Tato knihovna obsahuje funkce potřebné pro práci s certifikátem uživatele ve formátu PKCS#12 a další algoritmy použité pro výpočty kódů elektronické evidence tržeb v režimu off-line.

4.3 Elektronická evidence tržeb

V této části práce bude stručně popsán způsob fungování elektronické evidence tržeb. Není cílem práce podrobně rozebrat všechny aspekty zákona o elektronické evidenci, avšak budou zde popsány základní postupy a požadavky, které jsou pro fungování vytvářené aplikace vitální.

Podle oficiálního webu Generálního finančního ředitelství je postup evidování tržby následující:

1. Podnikatel zašle datovou zprávu o transakci ve formátu XML Finanční správě.

2. Ze systému finanční správy je zasláno potvrzení o přijetí s unikátním kódem FIK (fiskální identifikační kód).
3. Podnikatel vystaví účtenku (včetně FIK), kterou předá zákazníkovi.
4. Zákazník si může ověřit svoji účtenku na Daňovém portále.
5. Podnikatel si může ověřit tržby evidované pod jeho jménem ve webové aplikaci Elektronická evidence tržeb.

V případě, že se nepodaří datovou zprávu odeslat, z důvodu výpadku internetového připojení, je podnikatel povinen takto učinit nejpozději do 48 hodin od přijetí platby. Zákazníkovi v takovém případě vystaví podnikatel účtenku s kódem PKP místo kódu FIK.

V případě, že dojde k chybě při komunikaci během transakce, pak podnikatel postupuje stejně, jako v případě výpadku internetu, tj. zašle údaje o evidované tržbě serveru Finanční správy bezprostředně po obnovení spojení, nejpozději do 48 hodin od uskutečnění evidované tržby. Zde nezáleží na tom, zda správce daně již zprávu obdržel, a zařízení podnikatele pouze neobdrželo potvrzení, či zda zpráva o evidované tržbě nebyla vůbec doručena.

V případě poruchy pokladny poplatník neplní povinnosti vyplývající ze zákona o evidenci tržeb, pokud však prokáže, že učinil vše pro to, aby mohl začít opět co nejdříve tržby evidovat, nebude nijak postihován. Zprávy o platbách je pak poplatník povinen zaslat dodatečně.

Podle zákona o elektronické evidenci tržeb § 21 musí poplatník nastavit mezní dobu odezvy na více než 2 sekundy. Mezní dobu odezvy zákon specifikuje jako časový úsek mezi pokusem o odeslání údajů o evidované tržbě a přijetím fiskálního identifikačního kódu pokladním zařízením zákazníka. Při překročení mezní doby odezvy se postupuje jako v případě výpadku internetu.

Součástí evidované tržby jsou také kódy PKP (podpisový kód poplatníka) a BKP (bezpečnostní kód poplatníka). Tvorbu těchto kódů upravuje vyhláška o způsobu tvorby podpisového kódu poplatníka a bezpečnostního kódu poplatníka.

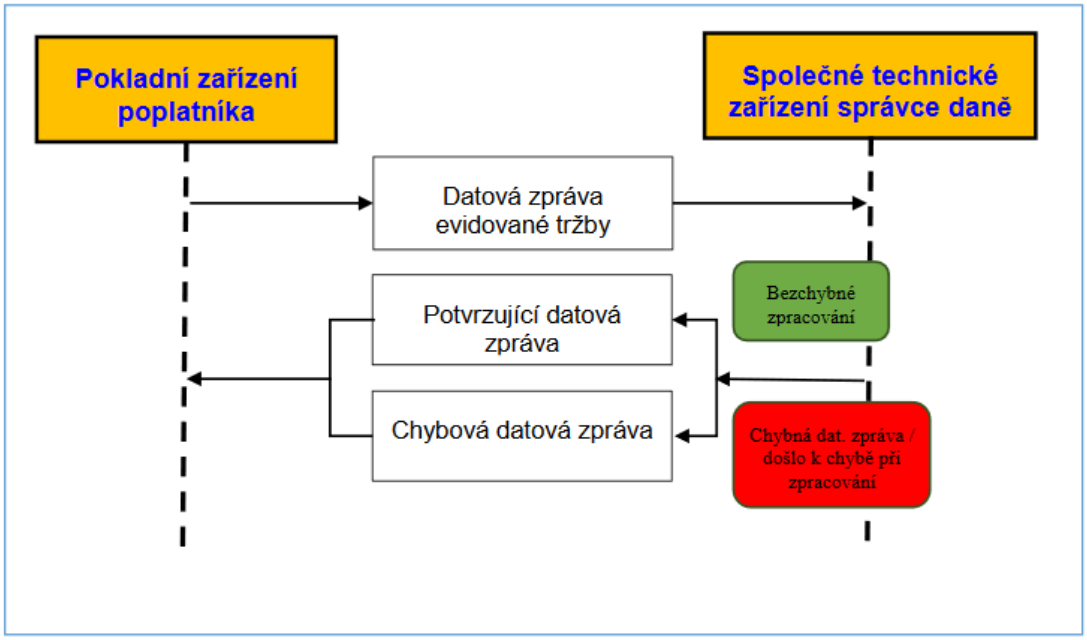
Podle §1 této vyhlášky je podpisový kód poplatníka tvořen podepsaným otiskem řetězce údajů o evidované tržbě a vyjádřen v kódování Base64.K vytvoření otisku řetězce údajů o evidované tržbě se použije kryptografická hashovací funkce SHA-256. Otisk řetězce údajů o evidované tržbě se podepíše podpisovým schématem RSASSA-PKCS1-v1_5 pomocí soukromého klíče, který náleží k používanému certifikátu pro evidenci tržeb platnému ke dni evidované tržby.

Bezpečnostní kód poplatníka je podle §2 uvedené vyhlášky tvořen otiskem podpisového kódu poplatníka vyjádřeného v osmibitovém kódování a je vyjádřen v kódování Base16 ve formě 5 skupin po 8 znacích oddělených pomlčkou, kterou

je znak s dekadickým kódem 45 v kódování Unicode. K vytvoření otisku podpisového kódu poplatníka se použije kryptografická hashovací funkce SHA-1.

Certifikát pro evidenci tržeb umožní podle zákona o elektronické evidenci tržeb § 15 poplatníkovi získat správce daně, přičemž jednomu poplatníkovi může být vydáno více certifikátů. Podle § 16 je poplatník povinen zacházet s certifikátem tak, aby nedošlo k jeho zneužití.

Formát a strukturu údajů o evidované tržbě popisuje dokument *Elektronická evidence tržeb – Formát a struktura údajů o evidované tržbě* dostupný na webu Generálního finančního ředitelství. Schéma komunikace můžeme vidět na obrázku 2.



Obrázek 2 - Komunikační scénář, Zdroj: Elektronická evidence tržeb – Formát a struktura údajů o evidované tržbě

Všechny 3 datové zprávy (datová zpráva evidované tržby, potvrzovací datová zpráva, chybová datová zpráva) mají společný základní datový formát daný protokolem SOAP (Simple Object Access Protocol), tj. aplikační XML datové struktury jsou vloženy do tzv. těla SOAP obálky (<*SOAP Envelope Body*>).

Datová zpráva evidované tržby musí splnit následující požadavky:

- Pro realizaci elektronického podpisu zprávy je využit standard WS-Security 1.0 a XML Digital Signature.
- Vlastní digitální podpis musí být vložen do SOAP obálky datové zprávy, a to v sekci hlaviček WS-Security. Odkaz na podepisovaný objekt

(element *<soap: Body>*) je realizován referencí s využitím relativního odkazu v rámci SOAP zprávy.

- Je požadován algoritmus *Exclusive C14N* kanonizace podepisovaného objektu (Exclusive XML Canonicalization Version 1.0, https://www.w3.org/TR/xml-exc-c14n/).

- Pro výpočet otisku (digest) podepisovaného objektu (element *<soap: Body>*) pro elektronický podpis SOAP zprávy je požadován hashovací algoritmus *SHA256* (http://www.w3.org/2001/04/xmlenc#sha256).

- Pro elektronický podpis SOAP zprávy je požadován algoritmus *RSA-SHA256* (http://www.w3.org/2001/04/xmldsig-more#rsa-sha256).

- X509 certifikát náležející k privátnímu klíči použitému pro realizaci elektronického podpisu datové zprávy evidované tržby včetně SOAP obálky musí být přiložen v elementu *BinarySecurityToken* v rámci sekce WS-Security hlavičky SOAP zprávy (typ http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-soap-message-security-1.0#Base64Binary) ve formátu X509v3 (typ http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-x509-token-profile-1.0#X509v3). Z digitálního podpisu je tento certifikát standardními prostředky referencován.

5 Návrh moderní webové aplikace

Pro splnění požadavků na snadnou údržbu, možnosti rozšiřování a rychlý vývoj aplikace je vhodné použít při jejím návrhu aktuální postupy a využívat moderní návrhové vzory.

V podkapitolách této části práce budou vysvětleny, či alespoň nastíněny, možnosti, které nám dnešní nástroje a poznání nabízí a jejich praktický přínos při tvorbě aplikace.

5.1 Jazyk UML a UP

S příchodem objektově orientovaných programovacích jazyků došlo také k vývoji samotných postupů při analýze a návrhu aplikací. Objektový přístup totiž přináší mimo jiné i daleko vyšší potřebu pochopení cílové domény, než tomu bylo dříve. Tento problém dal vzniknout novým nástrojům, jako jsou UP (Unified Process) a UML (Unified Modeling language). Jazyk UML byl navržen proto, aby spojil nejlepší existující postupy softwarového inženýrství a je navržen tak, aby jej mohly implementovat všechny nástroje CASE (computer-aided software engineering) (Arlow, a další, 2011). Specifikaci jazyka UML spravuje skupina OMG (Object Management Group). Jazyk UML byl za obecný průmyslový standard přijat v roce 1997. Zatím poslední verze UML 2.5 byla vydána v červnu roku 2015. Hlavními novinkami oproti předchozí verzi má být podle (OMG, 2016) zjednodušení návrhu a tři nové strukturální diagramy. Diagram modelů, diagram manifestací a diagram síťové architektury.

Podle (Arlow, a další, 2011) je jazyk UML nástrojem poskytujícím syntaxi pro objektově orientovaný návrh, zatímco UP je standardní osnovou pro proces tvorby aplikace, která usnadňuje vypracování objektově orientované analýzy a návrhu.

Jak uvádí (Mellor, 2004), lze model UML použít na třech úrovních:

- Za prvé, jako skicu pro vytvoření představy o řešeném problému, která nemá moc velkou hodnotu a nebývá přidávána do dokumentace, ale v prvních fázích vývoje může pomoci k ujasnění představy o řešeném problému.
- Za druhé, jako detailní plán, který je již formálnějším a přesnějším vyjádřením daného problému. Model UML je na této úrovni aktivně využíván a uchováván a k jeho tvorbě a úpravám je již nutné využívat skutečné modelovací nástroje jakým je například software Enterprise Architect společnosti Sparx Systems.
- Za třetí, lze jazyk UML použít k vytvoření spustitelného kódu pomocí architektury MDA (Model Driven Architecture). Jedná se o nejformálnější

vyjádření a modely UML jsou zde opatřeny takovými detaily, že je možný překlad programu přímo z modelu.

Podle (Arlow, a další, 2011) je možné návrh v jazyce UML s použitím UP rozdělit na část požadavků, analýzy a návrhu.

Požadavek zde představuje jednotku funkcionality, kterou uživatel (zadavatel) od implementace požaduje. V jazyce UML jsou takovéto požadavky specifikovány uživatelskými případy užití jednotlivých aktérů systému. Standard UP popisuje techniky a metody jakými tyto aktéry a jejich uživatelské cíle identifikovat.

Analýza jednotlivých požadavků pak vede k návrhu analytických modelů. Tyto modely obsahují analytické třídy a jsou již formálním popisem struktury aplikace, i když na vysoké úrovni abstrakce. V analytickém modelu jsou také znázorněny relace mezi jednotlivými analytickými třídami a vyjádřeno použití dědičnosti a polymorfismu. Podle (Arlow, a další, 2011) by správně navržená analytická třída měla splňovat následující požadavky:

- Název implikuje účel analytické třídy,
- jedná se o hrubou abstrakci, která definuje specifický prvek problémové domény,
- mapuje jasně identifikovatelnou vlastnost problémové domény,
- obsahuje malou množinu odpovědnosti (dodržuje princip takzvané Single Responsibility),
- je velmi soudržná (data a metody spolu úzce souvisí),
- obsahuje minimum vazeb na jiné třídy.

Ve fázi návrhu se analytické třídy převádějí na třídy návrhové a upřesňují se analytické relace. Návrhová třída by měla obsahovat detailní popis atributů (včetně názvu, typu, viditelnosti a povinně i implicitní hodnoty) a také úplnou sadu metod vytvořených podle operací definovaných v analytické třídě, ze které vychází. U každé metody je specifikována viditelnost, typ návratové hodnoty a typy parametrů. Podle (Arlow, a další, 2011) by správně formulovaná návrhová třída měla splnit následující podmínky:

- Je úplná a dostačující pro plnění svého účelu (Single Responsibility),
- je jednoduchá,
- je vysoce soudržná,
- je bez těsných vazeb (dodržuje návrhový vzor Loosely Coupled Component).

V návrhové fázi projektu jsou dále vytvářeny entitně relační diagramy (E/R diagramy) databázových struktur. Vzhledem k použití JPA ve vyvíjené aplikaci jsou entitní třídy přesným obrazem datových struktur databáze. Proto je při

jejich modelování využito E/R diagramů jako znázornění datového modelu aplikace (a tedy i vztahů mezi entitními třídami).

Prostředkem pro modelování procesů v systému jsou takzvané diagramy aktivit. Podle metodiky UP lze diagramy aktivit použít na více místech a nabízejí univerzální mechanismus pro modelování dynamiky systému kdekoli je to vhodné.

Při samotné tvorbě aplikace byl jazyk UML použit jako forma detailního plánu a k samotné implementaci byl použit programovací přístup založený na testování tzv. Test Driven Development.

5.2 Test driven development

Při tradičním způsobu vývoje software je pomocí analýzy požadavků navržena funkcionalita jednotlivých komponent, která je následně implementována a až poté jsou napsány jednotkové a integrační testy, které ověřují správnost implementace.

Při implementaci řízené testováním jsou poslední dva kroky procesu obráceny a testy jsou napsány ještě před samotnou implementací. Samotný kód je postupně vytvářen tak, aby jednotlivé metody plnily zadané testy. Tento postup přináší podle (Madeyski, 2010) několik výhod. Mimo jiné například:

- Konstantní odezvu na implementaci nové funkcionality,
- rozčleňuje problém na malé lépe zvládnutelné části,
- dodává odvahu pro refaktorování kódu,
- zaručuje určitou kvalitu a robustnost aplikace,
- testy jsou chápány, jako další forma dokumentace.

Test driven development, jako agilní způsob programování kódu, může být vzhledem ke své flexibilitě použit v jakékoli metodologii vývoje software, včetně těch, které obsahují detailní designové fáze předcházející samotné implementace. Definice testů mohou totiž posloužit přímo jako popis chování jednotlivých komponent systému.

Podle analýzy publikované v roce 2014 (Mäkinen, a další, 2014), kde bylo hodnoceno deset různých interních a externích kvalitativních aspektů z různých empirických studií, přináší TDD snížení počtu defektů a vede k produkování lépe udržovatelného kódu v porovnání s tradičními postupy. Části implementovaného kódu jsou navíc menší co do rozsahu i komplexnosti. Zatímco udržování takto vytvořeného kódu je jednodušší, jeho prvotní tvorba zabere větší množství času.

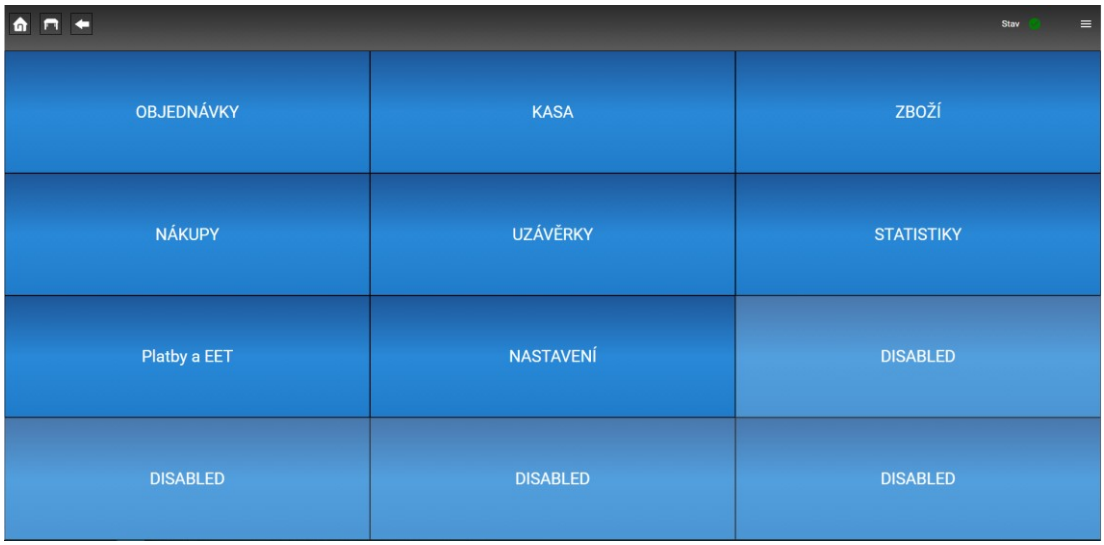
6 Popis a struktura aplikace

Vytvářená aplikace umožní uživatelům především elektronickou evidenci tržeb, ale také provádění dalších úkonů spojených s provozem restauračního zařízení s cílem získání přehledu o prodejích, tržbách, nákupech a dalších významných statistických datech. S ohledem na plnění povinností nařízených zákonem o elektronické evidenci tržeb bude možné aplikaci využívat v omezeném režimu i bez připojení k serverové části. O stavu aplikace a aktuálním režimu je uživatel informován prostřednictvím stavového panelu v záhlaví aplikace.

Uživatelské rozhraní je tvořeno s ohledem na co nejlepší ergonomii a na specifické požadavky jednotlivých úrovní uživatelů.

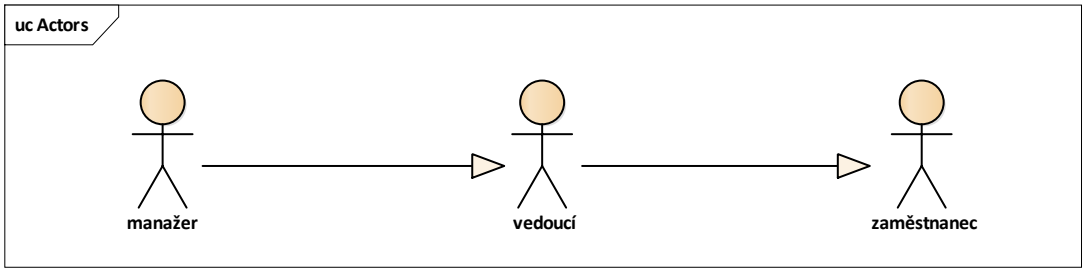
6.1 Funkcionalita pro jednotlivé role uživatelů

Možnosti celého uživatelského rozhraní jsou organizovány do systému karet na úvodní obrazovce, jak lze sledovat na obrázku 3. Podle role uživatele se zobrazí pouze patřičné karty.



Obrázek 3 - Úvodní obrazovka uživatelského rozhraní

Hlavní role v systému budou zaměstnanec, vedoucí a manažer. Mezi těmito rolemi existuje vztah generalizace, jak ukazuje následující diagram.

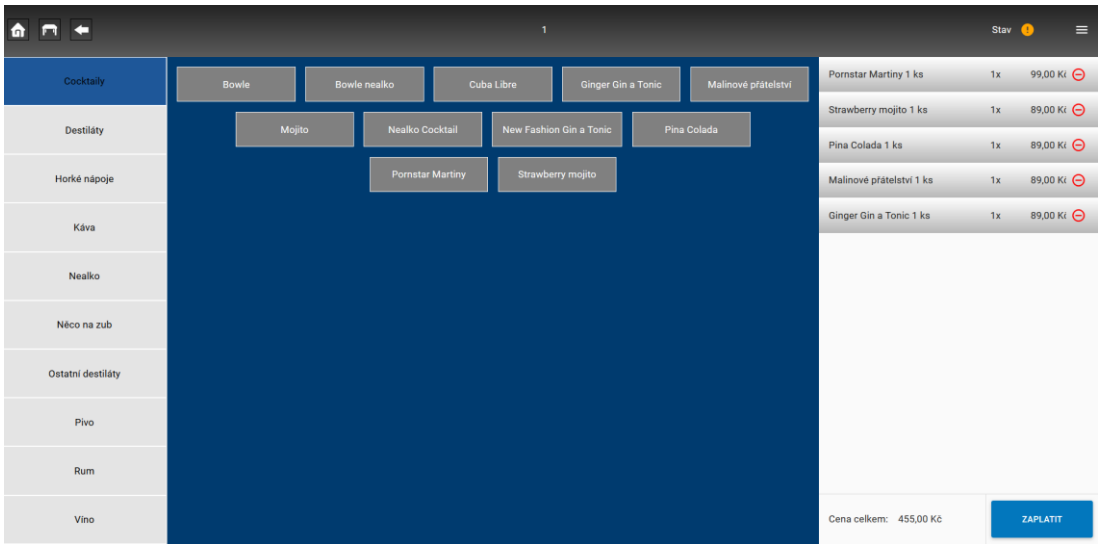


Obrázek 4 - Uživatelské role v systému

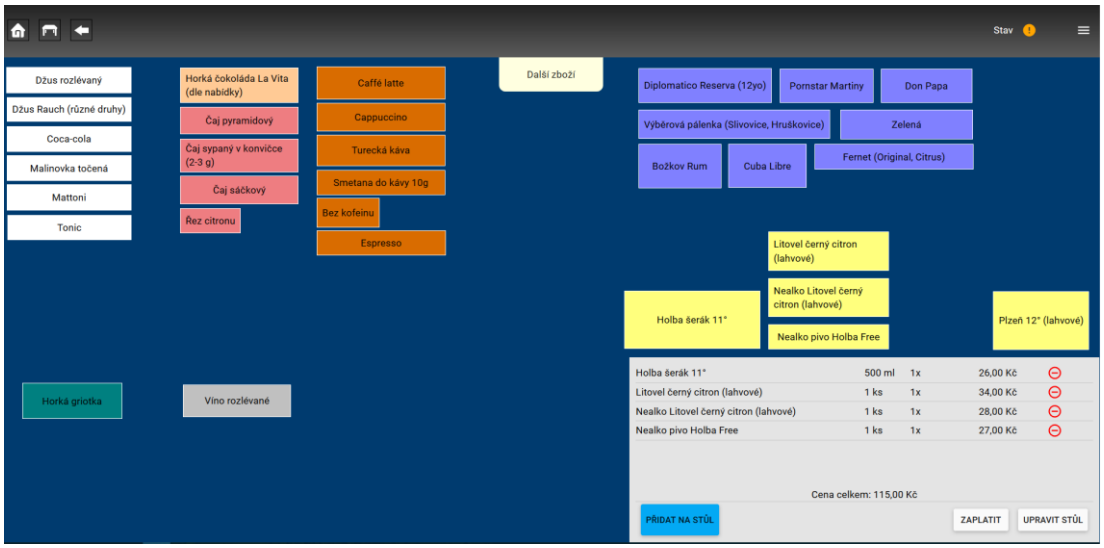
Z pohledu běžného zaměstnance (obsluhy) umožní aplikace především vytváření a správu objednávek a přijímání plateb. Při přijetí každé platby je provedeno její zaevidování podle zákona o evidenci tržeb. Uživatel také může platby procházet, případně znovu odeslat platbu, kterou se nepodařilo odeslat automaticky v době jejího přijetí od zákazníka. Jedním z hlavních požadavků na aplikaci je všechny tyto základní činnosti co nejvíce zjednodušit a zpřehlednit s ohledem na časovou tíseň, ve které se obsluha často nachází.

Z důvodu co největší flexibility a přizpůsobení se uživateli existují v aplikaci dva způsoby přidání zboží na objednávku. První možností je otevření seznamu stolů se zobrazenými aktuálními objednávkami a po otevření příslušného stolu kliknutím na položku v seznamu zboží, jak lze sledovat na obrázku 5. Druhou možností je přidání zboží prostřednictvím karty kasa. Toto zobrazení nabízí možnost vytvoření rychlé objednávky, kterou kupující okamžitě uhrazuje, nebo vytvoření objednávky, která je poté v dialogovém okně přiřazena některému ze stolů. V tomto zobrazení má uživatel možnost rozložit si na pracovní ploše nejčastěji používané (prioritní) zboží a barevně jednotlivé položky odlišit, jak lze sledovat na 6.

Zaměstnanec je v praxi běžná obsluha a s ohledem na zavedenou praxi nemá možnosti pro zobrazení statistických dat o prodejích ani možnost provádět storna.



Obrázek 5 – Uživatelské rozhraní pro úpravu objednávky



Obrázek 6 - Uživatelské rozhraní pro úpravu objednávky *KASA*

Dále aplikace nabídne funkcionalitu pro vedoucího směny, který bude mít možnost provádět všechny úkony, které provádí obsluha a navíc možnosti k vytváření uzávěrek, jejich procházení a především vytváření storno plateb. Storno platby často v podnicích tohoto typu vytvářejí právě vedoucí směny a nikoli obsluha. Podle zákona o evidenci tržeb znamená vytvoření storna pouze odeslání záporné částky na portál elektronické evidence tržeb a z pohledu aplikace tedy není nutné vázat storna na jednotlivé platby. Vytvoření uzávěrky znamená vytvořit záznam o datu a čase uzávěrky, její celkové hodnotě a položkách které obsahuje. Uzávěrky jsou nutné především při předávání provozu další směně, kdy je třeba zjistit kolik jakého zboží prodala směna předchozí. Ani na této úrovni uživatele není běžné prohlížení statistických dat provozovny.

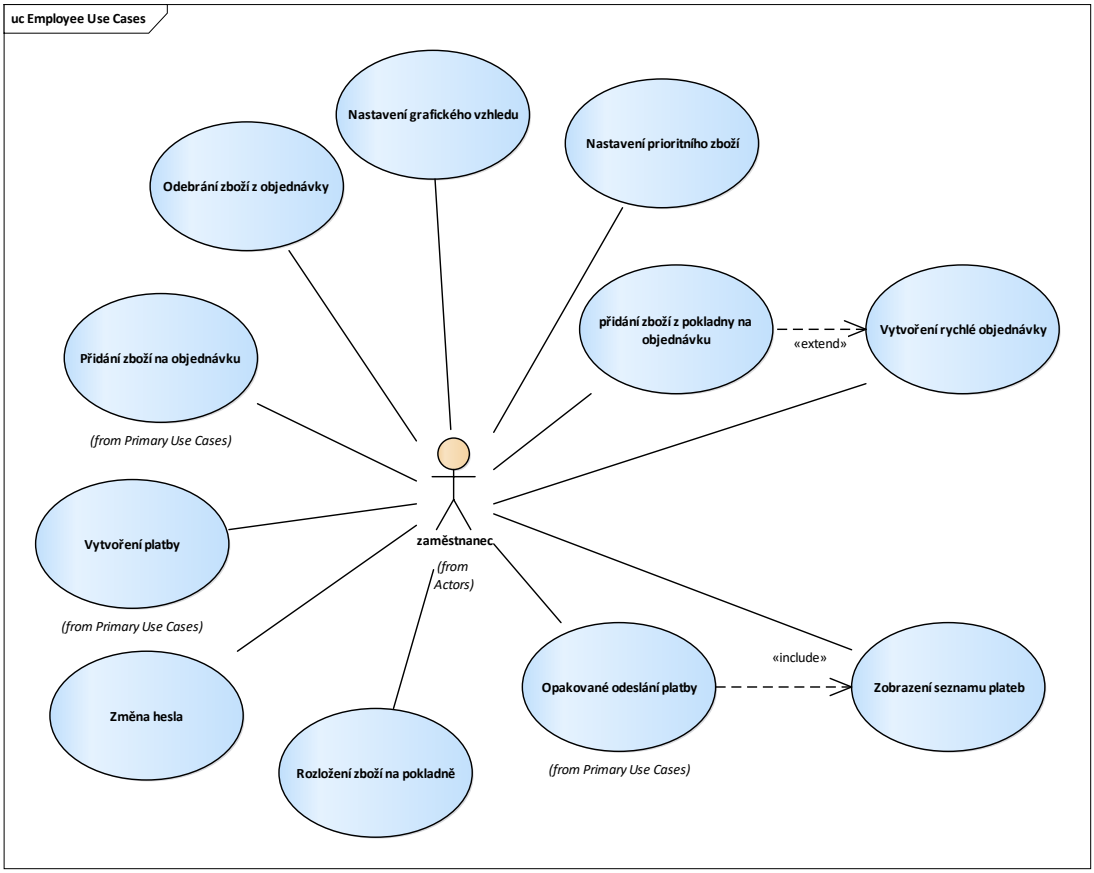
Nejvyšším stupněm v hierarchii uživatelů bude manažer (v praxi často majitel) podniku, který bude mít možnost přidávat ostatní uživatele a spravovat jejich práva. Dále bude mít tento uživatel k dispozici veškerou funkcionalitu nižších rolí a navíc prostředky k zadávání nákupů, editaci zboží a cen, přidávání a odebírání stolů, registraci nové pokladny, nastavení dat o provozovně a certifikátu pro elektronickou evidenci tržeb, prohlížení statistik a další. Zadávání nákupů slouží k porovnání historických cen daného druhu zboží a také ke sledování stavu zásob.

Společně veškerá tato funkcionalita umožní sledování chodu podniku, nabídne nástroje pro zefektivnění jeho provozu a současně zabezpečí splnění podmínek zákona o elektronické evidenci tržeb.

6.2 Uživatelské cíle a jejich stručný popis

V této podkapitole budou stručně popsány jednotlivé uživatelské cíle, jejichž podrobné scénáře byly využity při tvorbě aplikace. Tyto uživatelské cíle byly identifikovány pomocí analýzy cílové oblasti a prostřednictvím rozhovorů se zadavateli aplikace.

6.2.1 Uživatelské cíle pro roli zaměstnanec



Obrázek 7 - Uživatelské cíle pro roli zaměstnanec

- **Nastavení grafického vzhledu**

Prostřednictvím tohoto případu užití může uživatel přes kartu nastavení změnit rozložení grafického vzhledu stolů.

- **Rozložení zboží na pokladně**

Uživatel prostřednictvím menu na obrazovce karty *KASA* spustí editační režim, ve kterém je možné rozložit prioritní zboží na ploše zobrazení. Uživatel může měnit velikost jednotlivých položek a také jejich barvu či velikost písma.

- **Nastavení prioritního zboží**

Uživatel může prostřednictvím karty *ZBOŽÍ* nastavit, které zboží bude prioritní, a tím pádem zobrazené na ploše karty *KASA*.

- **Přidání zboží na objednávku**

Uživatel otevře kartu objednávek a přes dlaždici stolu přidá zboží na aktuální objednávku kliknutím na zboží v seznamu. Pokud má vybrané zboží více variant (např. různá množství téhož zboží), zobrazí systém před přidáním zboží na objednávku okno pro výběr varianty.

- **Vytvoření rychlé objednávky**

Uživatel má možnost prostřednictvím karty *KASA* vytvořit rychlou objednávku tím, že vybere požadované zboží z plochy, či seznamu zboží. Tento seznam je možné zobrazit pomocí vysouvacího panelu zboží kliknutím na tlačítko *Další zboží* na kartě *KASA*.

- **Přidání zboží z pokladny na objednávku**

Tento případ užití rozšiřuje vytvoření rychlé objednávky o možnost jejího přesunutí či přidání k nějaké existující objednávce na jednom ze stolů.

- **Odebrání zboží z objednávky**

Uživatel vybere přes kartu objednávek patřičný stůl. Na seznamu zboží objednávky odebere požadované zboží kliknutím na tlačítko odebrání zboží.

- **Vytvoření platby**

Pro vytvoření platby je nutné ze seznamu stolů vybrat požadovaný stůl a na jeho obrazovce kliknutím na tlačítko zaplatit otevřít možnosti platby. Na obrazovce platby je zobrazen seznam zboží na objednávce a uživatel má možnost vybrat zboží, které hodlá zákazník zaplatit. Po vybrání zboží uživatel stiskem tlačítka

ZAPLATIT vyvolá dialogové okno, ve kterém může zadat přijatou částku a systém mu zobrazí sumu, kterou má zákazníkovi vrátit. Poté uživatel stiskem tlačítka pro tisk účtenky vytiskne zákazníkovi doklad se všemi náležitostmi požadovaným zákonem o evidenci tržeb.

Platbu je možné také vytvořit přes kartu kasa po vytvoření rychlé objednávky.

- **Zobrazení seznamu plateb**

Zobrazení seznamu plateb je pro běžného uživatele v roli zaměstnanec nutné především pro opakované odeslání nezaevidované platby. Platbu má uživatel povinnost zaevidovat nejpozději v okamžik jejího přijetí od zákazníka. Pokud však z důvodu výpadku služeb portálu pro elektronickou evidenci tržeb či nedostupnosti internetového připojení dojde k chybě při odesílání platby, je tato platba zobrazena na seznamu plateb jako neodeslaná. V takovém případě musí uživatel po obnovení služeb platbu odeslat v zákonné lhůtě znovu.

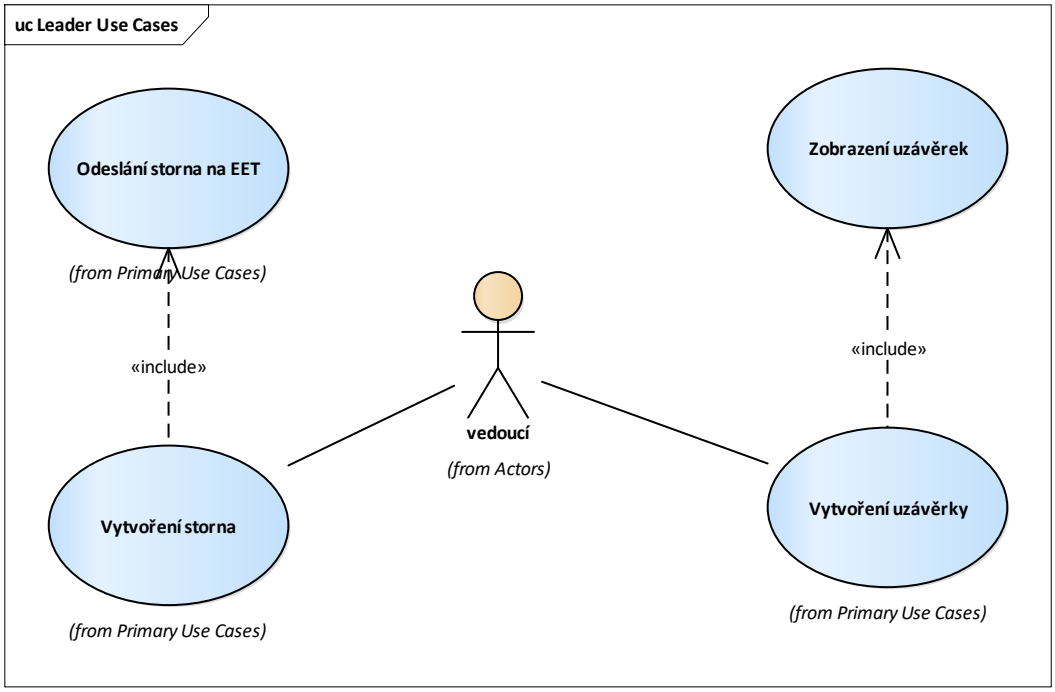
- **Opakované odeslání platby**

Uživatel prostřednictvím karty platby a EET otevře zobrazení seznamu plateb. Neodeslané platby jsou graficky zvýrazněny a na svém řádku mají tlačítko pro opakované odeslání platby.

- **Změna hesla**

Uživatel prostřednictvím menu v horní části obrazovky otevře dialog pro změnu hesla. Po vyplnění požadovaných údajů systém uloží v šifrované podobě heslo do databáze.

6.2.2 Uživatelské cíle pro roli vedoucí



Obrázek 8 - Uživatelské cíle pro roli vedoucí

- **Zobrazení uzávěrek**

Uživatel zobrazí seznam všech uzávěrek v systému pomocí karty uzávěrek. Po zobrazení seznamu má uživatel možnost výběrem konkrétní uzávěrky zobrazit seznam zboží, které uzávěrka obsahuje.

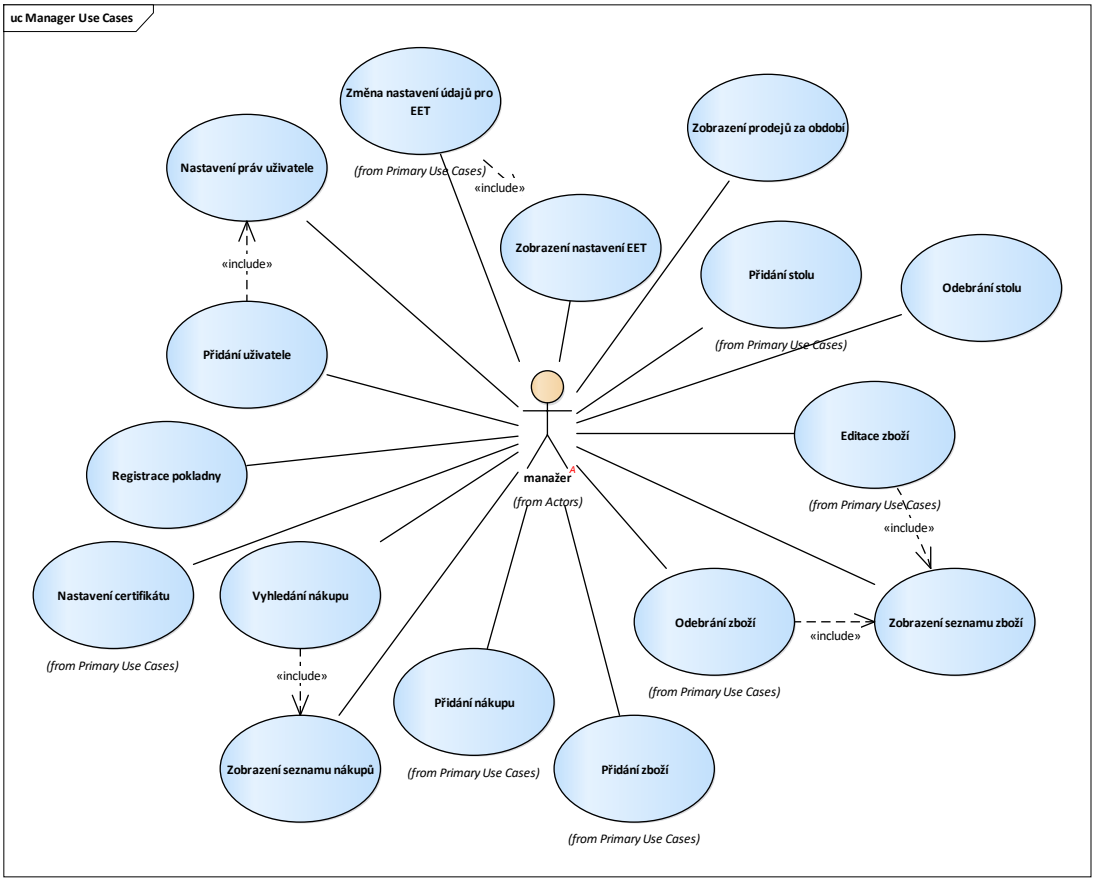
- **Vytvoření uzávěrky**

Uzávěrky vytváří vedoucí směny na kartě uzávěrek. Po vytvoření uzávěrky je možné zjistit její celkovou hodnotu a zobrazit seznam zboží uzávěrky.

- **Vytvoření a odeslání storna**

Vedoucí může na kartě storna vytvořit novou zápornou platbu. Tato platba je uložena stejně jako obyčejné platby za zboží. Součástí této aktivity, je také odeslání daného storna na portál elektronické evidence tržeb.

6.2.3 Uživatelské cíle pro roli manažer



Obrázek 9 - Uživatelské cíle pro roli manažer

Uživatel v roli manažer má k dispozici kromě zde uvedených uživatelských cílů také veškerou funkcionalitu nižších rolí.

- **Zobrazení seznamu nákupů**

Uživatel zobrazí seznam nákupů provozovny přes kartu *NÁKUPY*. Po otevření karty je zobrazeno okno se seznamem nákupů. Seznam obsahuje název zboží, datum, a cenu nákupu.

- **Vyhledání nákupu**

Prostřednictvím tohoto případu užití může uživatel na kartě *NÁKUPY* vyhledat nákupy se stejným zbožím a získat tak přehled o vývoji nákupních cen, případně porovnat ceny v historii s aktuální nabídkou.

- **Přidání nákupu**

Uživatel přidá nákup vyvoláním dialogového okna přidání nákupu. V tomto okně uživatel zadá název kupovaného zboží, jeho množství a cenu. Dále uživatel uvede datum nákupu. Po uložení je nákup zobrazen na seznamu nákupů.

- **Zobrazení seznamu zboží**

Uživatel výběrem karty *ZBOŽÍ* vyvolá zobrazení rozhraní pro administraci zboží. Systémem je zobrazen seznam zboží řazený do kategorií. Výběrem požadované kategorie a zboží může uživatel vyvolat zobrazení detailu vybrané položky. Tento detail obsahuje množství a cenu zboží a také historii těchto cen.

- **Přidání zboží**

Pro přidání zboží vyvolá uživatel dialogové okno, v němž vyplní údaje o zboží. Těmito údaji se rozumí název zboží a jednotky ve kterých bude uváděno.

- **Editace zboží**

Po výběru patřičného zboží a zobrazení jeho detailu má uživatel možnost upravit cenu zboží, čímž se předešlá cena uloží do historie cen. Uživatel má také možnost přidat nové množství a nastavit jeho cenu.

- **Odebrání zboží**

Po zobrazení seznamu zboží uživatel otevře pomocí tlačítka dialog pro odebrání zboží. V tomto dialogu systém přednastaví poslední vybrané zboží. Uživatel buď potvrdí smazání přednastaveného zboží, nebo vybere jiné zboží pro smazání ze seznamu.

- **Přidání stolu**

Uživatel na kartě objednávek pomocí tlačítka vyvolá dialog pro přidání stolu. Systém vyzve uživatele k zadání jména stolu. Uživatel zadá jméno a odešle stůl k uložení pomocí tlačítka *ULOŽIT*.

- **Odebrání stolu**

Na kartě objednávek vyvolá uživatel pomocí tlačítka dialog pro odebrání stolu. Systém zobrazí dialog pro odebrání stolu s vysouvacím seznamem stolů. Uživatel vybere požadovaný stůl a potvrdí jeho smazání.

- **Změna nastavení údajů pro EET**

Uživatel může na kartě *NASTAVENÍ* přes záložku *EET* změnit údaje potřebné pro elektronickou evidenci tržeb. Jedná se o id pokladního zařízení, daňové identifikační číslo poplatníka, a id provozovny. Uživatel vybere požadovanou položku a systém zobrazí dialogové okno s popisem položky a vstupním polem pro její změnu. Po potvrzení dialogu je hodnota položky uložena do databáze. Tyto hodnoty aplikace využívá i v off-line režimu, a proto jsou po změně nové hodnoty uloženy i do lokálního úložiště prohlížeče.

- **Nastavení certifikátu**

Uživatel má na kartě *NASTAVENÍ* prostřednictvím záložky *EET* možnost nahrání či změny certifikátu. Uživatel vyvolá dialog pro přidání či změnu certifikátu stiskem příslušného tlačítka. Systém zobrazí dialog pro nahrání nového souboru a zadání hesla. Po vyplnění údajů a potvrzení dialogu uživatelem systém uloží certifikát jak na straně serveru, tak na straně klienta.

- **Registrace pokladny**

Tento případ užití je spuštěn při prvním přihlášení uživatele z daného zařízení (prohlížeče). Systém otevře dialog pro přidání zařízení, ve kterém uživatel vyplní či potvrdí údaje o provozovně zadané při registraci, nahraje certifikát pro elektronickou evidenci tržeb a provede úvodní nastavení uživatelského rozhraní.

- **Přidání uživatele**

Na kartě *NASTAVENÍ* přes záložku *UŽIVATELÉ* vyvolá uživatel příslušným tlačítkem přidání nového uživatele. Systém zobrazí dialog pro zadání jména uživatele. Po potvrzení dialogu je založen nový uživatel bez jakýchkoli oprávnění.

- **Nastavení práv uživatele**

Na seznamu uživatelů vyvolá uživatel v roli manažera příslušným tlačítkem dialog pro nastavení práv jiného uživatele. Na zobrazeném dialogu je zobrazen seznam oprávnění v systému. Uživatel vybere příslušná oprávnění a potvrdí dialog. Systém přidělí vybraná oprávnění danému uživateli.

- **Odebrání uživatele**

Odebrání uživatele vyvolá uživatel v roli manažera pomocí příslušného tlačítka na seznamu uživatelů. Systém zobrazí potvrzovací dialog. Uživatel potvrdí smazání.

- **Zobrazení prodejů za období**

Prostřednictvím karty *STATISTIKY* je možné zobrazit seznam prodaného zboží za dané období. Uživatel zadá období, za které si přeje prodeje zobrazit a systém zobrazí dané položky. U každé položky je zobrazen název, množství, počet kusů, kategorie a celková cena. Také je zobrazena souhrnná statistika s údaji o celkovém počtu druhů zboží a celkové hodnotě prodejů za dané období.

6.3 Datové struktury

Vzhledem k relační povaze ukládaných dat byla pro uložení dat serverové části aplikace zvolena SQL databáze. V následující části textu jsou uvedeny požadavky na integritní omezení a struktura uložení dat.

Každá tabulka (entita) v databázi má jako primární klíč automaticky generované id. Pokud existuje soubor sloupců (atributů), který jednoznačně identifikuje daný záznam (instanci entity) v tabulce, pak je tento soubor označen unikátním indexem.

Datovou strukturu aplikace lze rozdělit na několik logicky souvisejících celků.

6.3.1 Datová struktura zboží provozovny

- **Kategorie**

Zboží je uspořádáno do stromové struktury, jejíž nejvyšší úrovní jsou kategorie. Kategorie představují v aplikaci prostředek pro organizaci zboží. Jméno kategorie je v tabulce kategorií jedinečné a musí být uvedeno. Kategorie obsahuje zboží, ale je možná existence kategorie bez zboží. Z funkčního hlediska taková kategorie nemá význam, nicméně její existence je vhodná pro jednodušší vytváření kategorií, kdy je vytvořena prázdná kategorie a až později je do ní přidáváno zboží.

- **Zboží**

Zboží představuje druh prodávané suroviny. V jedné kategorii nesmí být více zboží se stejným názvem. Každé zboží má v atributu typ uloženu hodnotu, která reprezentuje, zda je zboží prodáváno v jednotkách objemu, váhy, či v kusech. Realizaci vztahu s entitou kategorie představuje povinný cizí klíč do tabulky kategorií. Vytvářené zboží tedy musí vždy patřit do některé kategorie. Každé zboží musí mít dále uveden název, hodnotu a DPH.

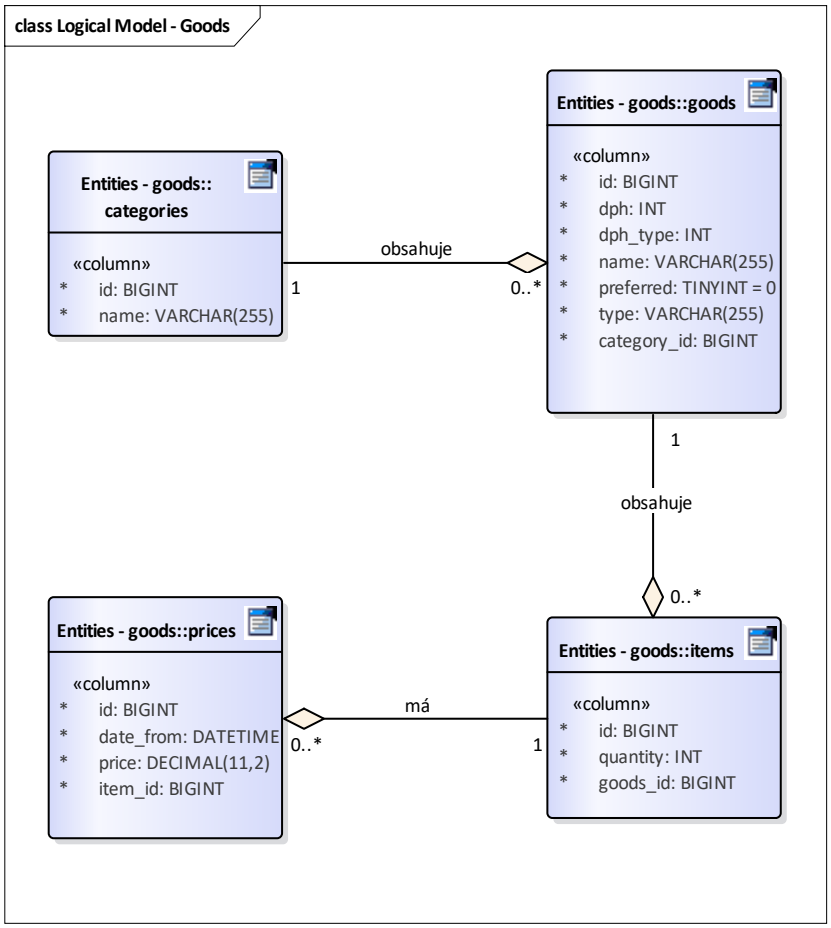
- **Item (množství)**

Zboží obsahuje kolekci záznamů typu *Item*. Tyto entity reprezentují jednotlivá množství stejného typu zboží. V jednom zboží nesmí být více položek tohoto typu se stejným množstvím. Povinnou hodnotou je zde množství a vazba na zboží je vyjádřena cizím klíčem do tabulky zboží. Tento klíč je opět povinný, takže každá entita typu *Item* patří k nějakému existujícímu zboží.

Jedním z požadavků na aplikaci je zobrazení historie cen jednotlivých položek nacházejících se v seznamu zboží. Z tohoto důvodu byla vytvořena samostatná entita typu *Cena*.

- **Cena**

Nejnižším stupněm v hierarchii je entita *Cena*. Entita typu cena obsahuje povinné atributy *date_from* a *price*.



Obrázek 10 - E/R diagram zboží

6.3.2 Datová struktura plateb

Platby jsou v aplikaci vytvářeny pomocí aplikace klientské části (ve webovém prohlížeči). Tito klienti si drží stav svých otevřených objednávek a platbu odesílají na server až při uzavření objednávky. Vzhledem k častým změnám v cenách, či názvech jednotlivých druhů zboží neexistuje v aplikaci pevná vazba mezi entitou platby a zbožím. Při přidání zboží na objednávku je vytvořena nová entita typu *order_item* a jsou do ní zkopírovány údaje o zboží. Tímto způsobem zůstanou data platby neměnná. Pokud by existovala vazba například cizím klíčem na entitu ceny, pak by při změně ceny došlo i ke změně v již uskutečněné objednávce, což je nežádoucí.

- **Platba**

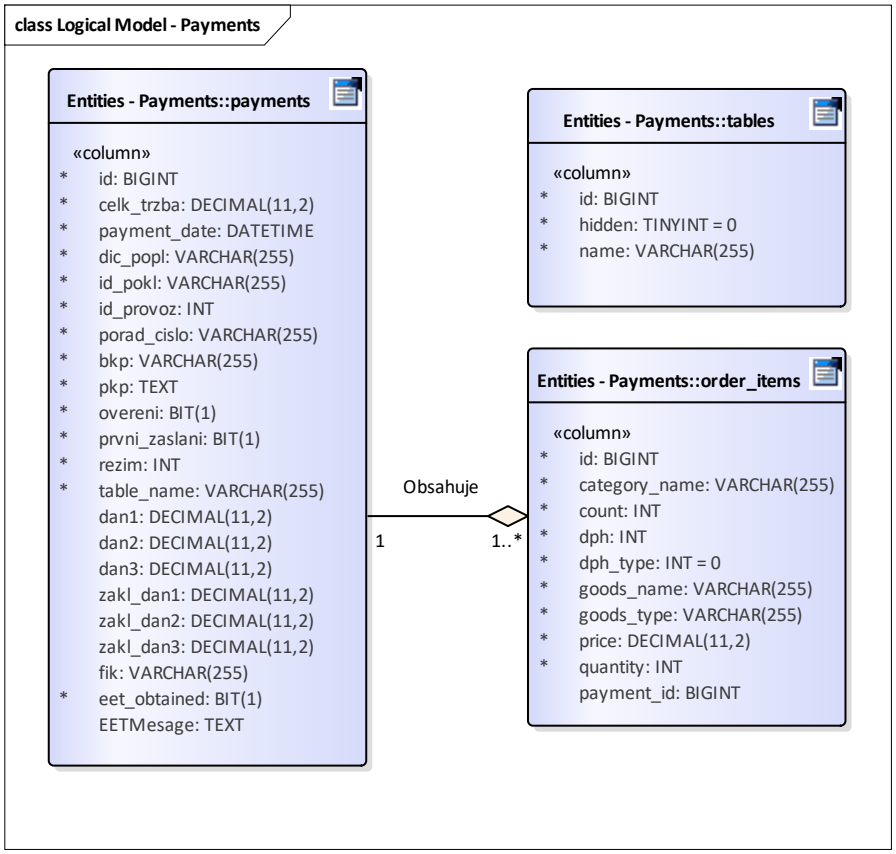
Platba reprezentuje data o platbě, kterou uživatel uskutečnil a zároveň je tato struktura abstrakcí ze struktury dat potřebných pro evidování platby na portálu elektronické evidence. Všechny vlastnosti této entity jsou povinné kromě vlastností *table_name*, *dan1 – 3*, *zakl_dan1 – 3*, *fik*, *eet_obtained* a *EETMessage*. Atribut *table_name* obsahuje název stolu, na kterém byla vytvořena původní objednávka. Daňové položky jsou relevantní pouze pokud je majitel provozovny plátcem DPH a položky *fik*, *eet_obtained* a *EETMessage* slouží k uložení výsledku odeslání platby na portál správce daně. Každá platba obsahuje nejméně jednu položku typu *order_item*.

- **Položka objednávky**

Každá položka objednávky obsahuje informace o zboží, ze kterého byla vytvořena. Jedná se o povinné atributy *category_name*, *dph*, *dph_type*, *goods_name*, *price* a *quantity*. Povinný atribut *count* obsahuje informaci o počtu položek daného typu. Vztah s entitou platby je vyjádřen povinným cizím klíčem na tuto entitu. V tabulce položek objednávky musí být unikátní soubor hodnot názvu zboží, množství a jména kategorie.

- **Stůl**

Slouží k uložení informace o jednotlivých stolech v provozovně. Obsahuje povinný atribut *name* se jménem daného stolu a atribut *hidden*, který je využit při zobrazování seznamu stolů.



Obrázek 11 - E/R diagram plateb

6.3.3 Datová struktura nastavení

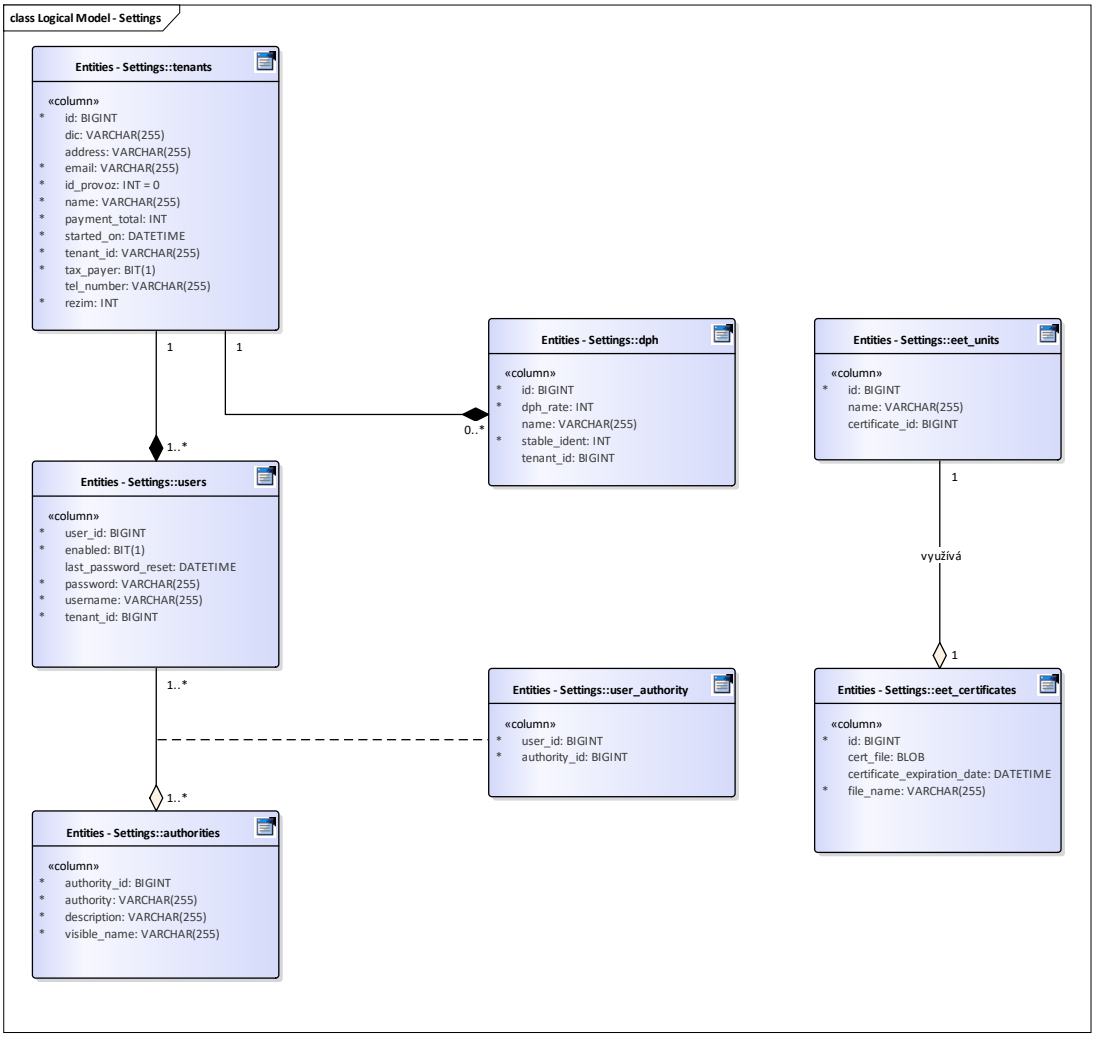
- **Tenant**

Entita typu *Tenant* zastupuje v aplikaci jednu provozovnu. Z hlediska elektronické evidence tržeb je nutné, aby každý tenant uvedl informace o identifikačním čísle provozovny a svém DIČ. Velmi důležitou položkou je zde také identifikátor tenanta *tenant_id*, které odlišuje jednotlivé tenanty v rámci aplikace.

- **Uživatel**

Entita typu *User* zastupuje jednotlivé uživatele v rámci jednoho klienta. Tito uživatelé mají k dispozici kolekci entit typu *Authority*, která definuje práva a role každého uživatele.

Další entitou aplikace je entita *EETUnit*, která reprezentuje jednotlivá zařízení používaná v rámci aplikace. Tato zařízení potřebují pro svou funkcionalitu informace o certifikátu provozovny. Vztah mezi certifikátem a jednotkou je 1 k 1.

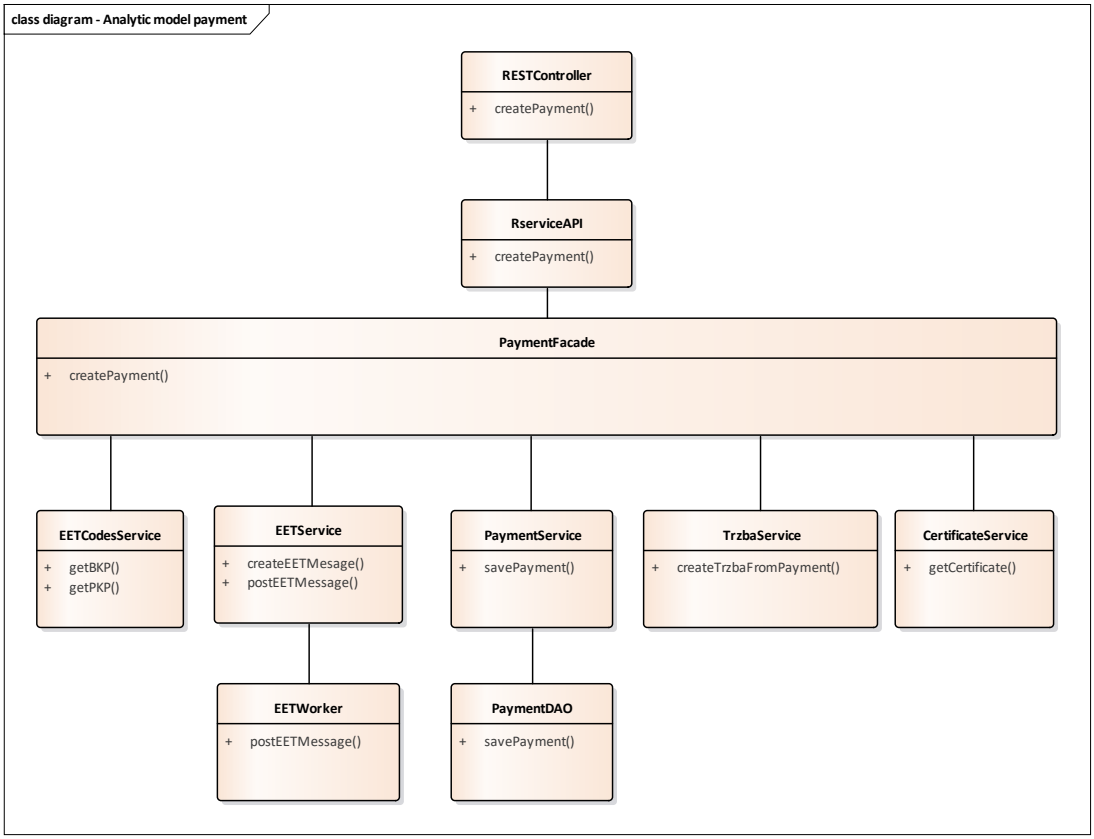


Obrázek 12 - E/R diagram pro nastavení aplikace

6.4 Příklad tvorby analytického modelu

Při tvorbě analytického modelu je třeba vycházet z analýzy daného problému, která může být reprezentována například případem užití. Podrobným rozborem případu užití jsou identifikovány důležité funkce, které by měl software poskytnout a ty poslouží jako podklad pro tvorbu jednotlivých metod a tříd analytického modelu aplikace. Při tomto postupu je také vhodné brát v úvahu moderní návrhové vzory, které mohou navrhovanou strukturu aplikace ujednotit a zjednodušit.

Na obrázku 13 můžeme sledovat analytický model komponent pro naplnění uživatelského cíle *vytvoření platby* v pokročilé fázi návrhu. Jsou zde identifikovány hlavní komponenty systému a názvy jednotlivých metod vystihují odpovědnost dané komponenty.



Obrázek 13 – Diagram analytického modelu platby

Nejvyšším bodem hierarchie tříd je zde kontrolér, který představuje samotné REST rozhraní pro komunikaci s webovým prohlížečem.

Vstupním bodem funkcionality celé aplikace je pak *RserviceAPI*. V tomto bodě je možné připojení různých komunikačních vrstev, pokud by bylo třeba zaměnit rozhraní REST například za RMI (Remote Method Invocation).

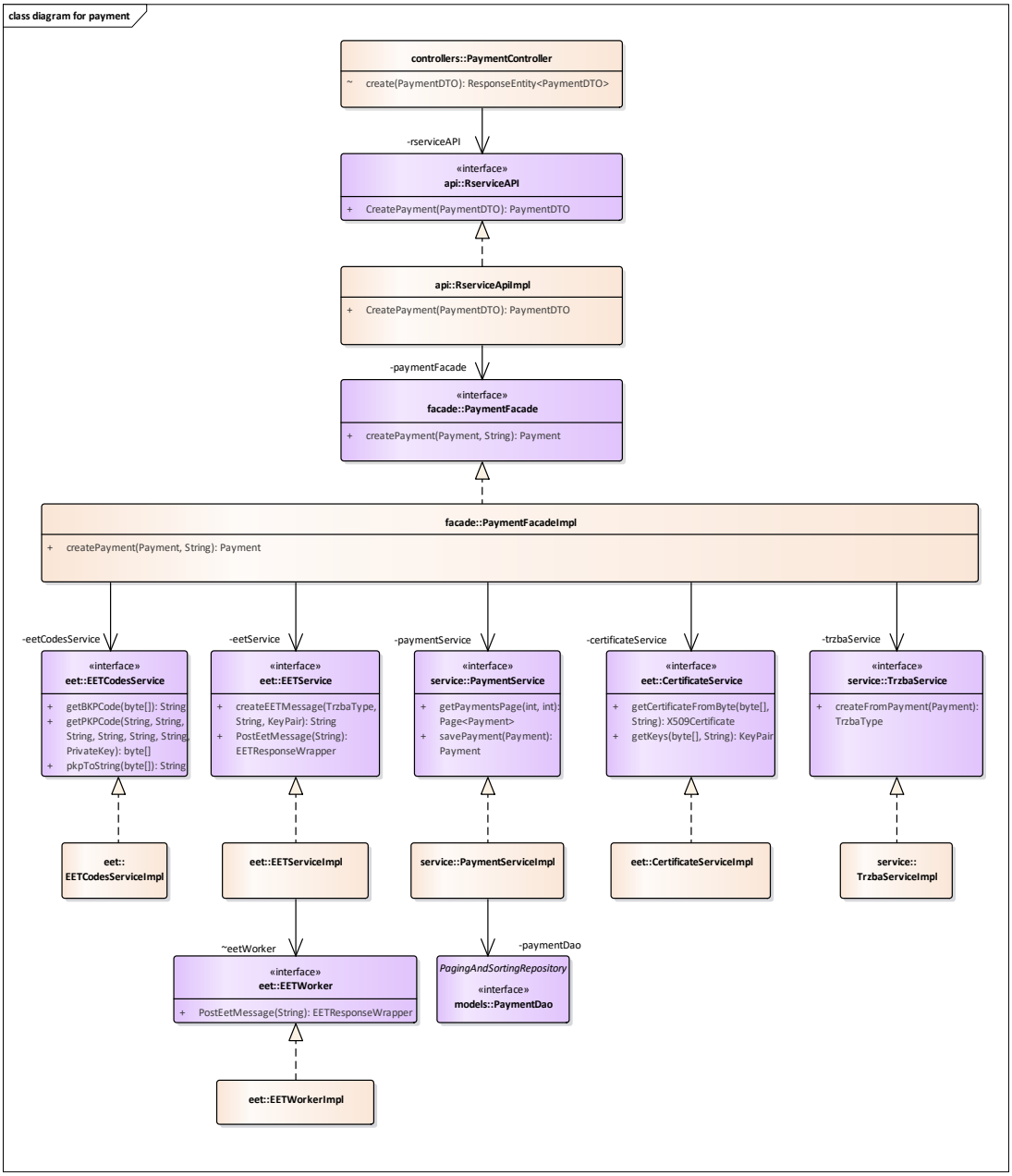
Dále je zde využito návrhového vzoru *Facade*, který je určen pro zastřešení určité části funkcionality aplikace. Jednotlivé metody zde představují vlastní uživatelské cíle, které má tato část naplňovat. Vzhledem k tomu, že zde máme přehled o funkcionalitě, kterou uživatel v dané chvíli požaduje, je na této úrovni vhodné řešit autorizaci uživatele pro použití jednotlivých metod.

Dále zde máme několik tříd představujících samotnou aplikační logiku, jejichž spoluprací je možné uživatelský záměr provést. Pro tyto třídy se vžilo označení servis a jejich cílem je poskytnout služby pro malou a ohraničenou část funkcionality.

Nejníže v hierarchii jsou především třídy zastřešující přístup k datům umožňující především CRUD operace. Tyto třídy jsou navrženy podle návrhového vzoru DAO (Data Access Object) později usnadní případnou výměnu datového zdroje.

6.5 Příklad tvorby návrhového modelu

Při převodu analytických tříd do jejich návrhové podoby je především třeba přesně specifikovat jejich vazby a vlastní funkcionalitu. Na obrázku 14 lze sledovat návrhové třídy vytvořené z předchozího analytického modelu. Tyto třídy již mají plně specifikované parametry a návratové hodnoty jednotlivých metod a jsou připraveny k implementaci. Zde je třeba ještě zmínit, že celý proces analýzy, návrhu a implementace je iterativní a v jakémkoli jeho bodě může být nutné upravit vyšší vrstvy tak, aby reflektovali změny vynucené níže.



Obrázek 14 - Diagram návrhového modelu pro platbu

Správně a co nejjednodušeji vytvořené návrhové třídy a jejich vazby jsou základním předpokladem pro snadnou udržovatelnost a rozšiřitelnost aplikace. Při tvorbě návrhových tříd bylo využito několika moderních návrhových vzorů a postupů jako jsou: volně propojené komponenty, Single Responsibility a DTO (Data Transfer Objects). Praktická spolupráce navržených komponent při provádění uživatelského cíle *vytvoření platby* bude uvedena v dalších částech práce.

- **Volně propojené komponenty**

Jednotlivé implementační třídy jsou závislé pouze na rozhraních ostatních komponent, které ke svému běhu potřebují a tím pádem nemají žádné povědomí o jejich vlastní implementaci.

- **Single responsibility**

Tento princip říká, že každá třída by měla mít co nejmenší a co nejlépe ohraničenou oblast zodpovědnosti. Dodržení tohoto principu mimo jiné velmi usnadňuje testování takovýchto tříd v jednotkových testech.

- **Data Transfer Objects (DTO)**

Tyto objekty usnadňují přenos dat mezi serverem a klienty, a to většinou tam, kde je třeba pro jedno zobrazení poskytnout více různých entit, či jejich seznamů. Tímto způsobem je možné jinak samostatné dotazy na server spojit do dotazu jediného. Pro konverzi mezi DTO a běžnými entitami je možné použít již vytvořené nástroje jako je například knihovna *ModelMapper*. DTO objekty jsou použity pouze pro přenos dat a objevují se proto pouze na úrovni API celého projektu a v jednotlivých kontrolérech REST rozhraní.

7 Implementace klíčových částí aplikace

V této části práce budou popsány klíčové aspekty celé aplikace a jejich implementační řešení.

7.1 Test Driven Development

Jak již bylo řečeno v úvodní části práce, byly všechny klíčové komponenty implementovány s pomocí programovacího přístupu založeného na testování. V této části bude takovýto postup prakticky prezentován na komponentě *EETCodesService*, která slouží pro výpočet kódů PKP a BKP.

Prvním krokem pro implementaci je definice kontraktu této služby. Kontrakt je zde definován pomocí zápisu dokumentace v Javě (Javadoc). Každá metoda má ve svém popisu definovánu svou zodpovědnost, parametry, návratovou hodnotu a své chování v případě chyby, jak lze sledovat v kódu 1.

Pro validaci parametrů je využita anotace *@Validated* a u každého parametru, je poté dalšími anotacemi nastavena samotná validace. V případě nevalidní hodnoty dojde k výjimce typu *MethodConstraintViolationException*, která je v kontraktech jednotlivých metod vždy zmíněna.

Kód 1 - Kontrakty komponenty pro výpočet kódů PKP a BKP

```
/**
 * Tato sluzba je urcena pro vypocet bezpecnostniho kodu poplatnika PKP a privatniho
 * kodu poplatnika PKP.
 */
@Validated
public interface EETCodesService {

    /**
     * Slouzi pro vypocet kodu PKP. Podle §1 vyhlášky o způsobu tvorby podpisového
     * kódu poplatníka a bezpečnostního kódu poplatníka.
     * @param dicPopl - Danove cislo poplatnika
     * @param idProvoz - id provozu
     * @param idPokl - id pokladniho zarizeni
     * @param poradCis - poradove cislo uctenky
     * @param datTrzby - datum uskutecneni trzby
     * @param celkTrzba - celkova castka trzby
     * @param privateKey - privatni klic uzivatele
     * @return  byte[] pkp
     * @throws MethodConstraintViolationException v pripade neplatnych argumentu
     */
    byte[] getPKPCode(@NotBlank String dicPopl, @NotBlank String idProvoz, @NotBlank
                      String idPokl, @NotBlank String poradCis, @NotBlank String
                      datTrzby, @NotBlank String celkTrzba, @NotNull PrivateKey
                      privateKey);

    /**
     * Kod BKP je tvoren otiskem kodu PKP podle §2 vyhlášky o způsobu tvorby
     * podpisového kódu poplatníka a bezpečnostního kódu poplatníka.
     * @param pkp - kod PKP v osmibitovém kódování ve forme 5 skupin po 8 znacích
     * oddělených pomlčkou, kterou je znak s dekadickým kódem 45 v kódování Unicode.
     * @return kod BKP v Base16
     * @throws MethodConstraintViolationException v pripade neplatnych argumentu
     */
    String getBKPCode(@NotNull byte[] pkp);

    /**
```

```
     * Slouzi pro prevod PKP z pole bajtu na String
     * @param byte[] pkp
     * @return String pkp
     * @throws MethodConstraintViolationException v pripade neplatne hodnoty
     * argumentu pkp
     */
    String pkpToString(@NotNull byte[] pkp);

}
```

Jakmile máme definován kontrakt celého interface, je možné přistoupit k tvorbě testů jednotlivých metod. Pro testování validace jednotlivých parametrů je třeba spouštět test v kontextu Spring IoC kontejneru, z tohoto důvodu všechny testy rozšiřují abstraktní třídu *AbstractTest*, která provede zavedení kontextu pomocí anotací *@RunWith* a *@SpringApplicationConfiguration* viz kód 2.

Kód 2 - Abstraktní třída testu pro zavedení kontextu

```
@RunWith(SpringJUnit4ClassRunner.class)
@SpringApplicationConfiguration(classes = Application.class)

public abstract class AbstractTest {

    protected Logger logger = LoggerFactory.getLogger(this.getClass());

}
```

Kód 3 - Jednotkový test pro EETCodesService

```
public class EETCodesServiceTest extends AbstractTest {

    @Autowired
    EETCodesService eetCodesService;
    @Autowired
    CertificateService certificateService;

    @Test(expected=MethodConstraintViolationException.class)
    public void testArgumentsValidation() {

        eetCodesService.getPKPCode("CZ1212121218", "273", "/5546/RO24", 0/6460/ZQ42",
                                   "2016-08-05T00:30:12+02:00", "34113.00", null);
    }

    @Test
    public void pkpGeneration(){

        KeyPair keypair;
        File certFile = new File("src/test/resources/EETTests/testCertificate.p12");
        String pass = "eet";

        try {
            keypair = certificateService.getKeys(
                                        Files.readAllBytes(certFile.toPath()), pass);
        } catch (Exception e) {
            throw new IllegalArgumentException(
                                    "cannot read certificate with pass: " + pass, e);
        }

       byte[] pkpValue = eetCodesService.getPKPCode("CZ1212121218", "273",
                                                    "/5546/RO24", "0/6460/ZQ42",
                                                    "2016-08-05T00:30:12+02:00",
                                                    "34113.00",
                                                    keypair.getPrivate());

        String pkp = eetCodesService.pkpToString(pkpValue);

        Assert.assertEquals("expected pkp code", expectedPKP, pkp);
```

```
        String bkp = eetCodesService.getBKPCode(pkpValue);

        Assert.assertEquals("expected bkp code", expectedBKP, bkp);

    }

}
```

Hlavním účelem testování je ověřit, zda metody daného interface plní svůj kontrakt. V případě tohoto testu to znamená, že vypočtené hodnoty obou kódů jsou správné a při použití testovacího certifikátu vrátí testované metody očekávané hodnoty viz kód 3. Tyto hodnoty prezentované proměnnými *expectedPKP* a *expectedBKP*, společně s testovacím certifikátem, byly získány pro účely testování z webu Generálního finančního ředitelství.

Dalším aspektem, který je třeba otestovat je to, zda se metody chovají podle svého kontraktu i v případě zadání nevalidních dat. K tomuto účelu slouží metoda *testArgumentsValidation* uvedená v kódu 3, která testuje správnost vyvolané chyby při zadání *null* hodnoty jako jednoho z argumentů. Zde by bylo vhodné testovat validaci každého z argumentů nicméně pro zpřehlednění ukázky je testována pouze jedna z variant.

Poslední částí procesu je již samotná implementace dané služby a provedení potřebných iterací postupu, dokud implementace neprojde všemi vytvořenými testy. Samotná implementace zde nebude uvedena z důvodu své rozsáhlosti.

7.2 Multitenancy

Jak již bylo řečeno, *Tenant* zastupuje v aplikaci jednu provozovnu. Aplikace je navržena tak, aby měl každý tenant v databázi své vlastní schéma. Z hlediska implementace tak potřebujeme tenanta jednak identifikovat a také mu přidělit vlastní zdroje pro přístup k jeho schématu.

Pro identifikaci je každému tenantovi přiděleno jedinečné identifikační číslo. Toto číslo je v aplikaci získáváno dvěma způsoby. Pokud tenant ještě není přihlášen, tak toto id musí uvést v URL při přihlášení. Pokud již přihlášen je a má svou vlastní *session*, pak je využit kontext uživatele uložený v objektu *SecurityContextHolder*. Získané id tenanta je poté využito pro změnu datového zdroje. Spring Security je nakonfigurováno tak, aby autentizace uživatele proběhla oproti nastavenému datovému zdroji, který obsahuje informace a přihlašovací údaje různých uživatelů jednoho tenanta.

Prvním krokem při implementaci multitenantní aplikace s využitím frameworku Spring je nastavení Spring Security tak, aby k autentizaci uživatele byla použita databáze s informacemi o uživatelských účtech. Uživatel je v aplikaci reprezentován entitou *User*, která se z hlediska uložení neliší od ostatních

datových entit aplikace. Konfigurační třídu pro nastavení *Spring Security* demonstruje kód 4.

Aby Spring při autentizaci uživatele použil ORM a získal informace o uživatelských účtech z databáze, je třeba nastavit do kontextu aplikace vlastní implementaci služby *UserDetailsService*, jak lze sledovat v kód 4.

Kód 4 - Konfigurace Spring Security

```
@Configuration
@Order(SecurityProperties.ACCESS_OVERRIDE_ORDER)
public class WebSecurityConfig extends WebSecurityConfigurerAdapter {

    @Autowired
    private ApplicationContext appContext;

    @Autowired
    MyUserDetailService userDetailService;

    @Override
    public UserDetailsService userDetailsServiceBean() throws Exception {
        return userDetailService;
    }
}
```

Samotná implementace interface *UserDetailsService* není nijak složitá a vyžaduje implementaci pouze jediné metody viz kód 5.

Kód 5 - Implementace *UserDetailsService* pro vlastní DAO.

```
@Service
@Transactional
public class MyUserDetailService implements UserDetailsService {

    @Autowired
    UserDao userDao;

    @Override
    public UserDetails loadUserByUsername(String username) throws
                                                        UsernameNotFoundException
    {

        User user = userDao.findByUsername(username);
        if (user == null) {
            throw new UsernameNotFoundException(String.format("No user found with
                                                    username '%s'.", username));
        } else {
            return user;
        }

    }
}
```

Tímto způsobem zajistíme, že Spring použije vlastní implementaci *UserDao*, která využívá ORM k získání informací o uživateli. Spring použije informace o uživatelském účtu k autentizaci uživatele a pokud je autentizace úspěšná, tak uloží informace o přihlášeném uživateli do zmíněného objektu

SecurityContextHolder. Takto nakonfigurovaná aplikace by však měla stále stejný zdroj dat a sloužila by pouze jedinému tenantovi.

Pro přepínání mezi datovými zdroji a tím i tenanty musíme ještě provést konfiguraci datových zdrojů. Datové zdroje jsou organizovány v kolekci *MultitenantDataSourceMap*, jejíž implementace je uvedena v kódu 6 a pro jejich správu a přepínání slouží *AbstractRoutingDataSource*. Tato abstraktní třída reprezentuje datový zdroj umožňující přepínání mezi podřízenými datovými zdroji a je součástí Spring frameworku. Tuto třídu rozšiřuje vlastní třída *MultitenantDataSource*, která přepisuje metody potřebné k přepínání datových zdrojů viz kód 8. Pro start aplikace a její funkcionalitu bez přihlášeného tenanta je také nutné nakonfigurovat výchozí datový zdroj, jak ukazuje kód 7.

Kód 6 - *MultitenantDataSourceMap* pro správu datových zdrojů

```
@Component
public class MultitenantDataSourceMap {

    private Map<Object,Object> dataSourceMap;

    public MultitenantDataSourceMap()
    {
        dataSourceMap=new HashMap<Object,Object>();
    }
    public void addDataSource(String session, DataSource dataSource)
    {
        this.dataSourceMap.put(session,dataSource);
    }
    public Map<Object,Object> getDataSourceMap()
    {
        return dataSourceMap;
    }
    public void removeSource(String session)
    {
        dataSourceMap.remove(session);
    }
}
```

Kód 7 - Konfigurace datového zdroje

```
@Configuration
public class DataSourceConfiguration {

    @Autowired
    private MultitenantDataSourceMap multitenantDataSourceMap;

    @Bean
    public DataSource dataSource() {

        MultitenantDataSource dataSource = new MultitenantDataSource();
        dataSource.setDefaultTargetDataSource(defaultDataSource());
        dataSource.setTargetDataSources(multitenantDataSourceMap.getDataSourceMap());
        dataSource.afterPropertiesSet();

        return dataSource;
    }


    private DataSource defaultDataSource() {

        DataSourceBuilder dataSourceBuilder = new
                                DataSourceBuilder(this.getClass().getClassLoader())
```

```
                .driverClassName(properties.getDriverClassName())
                .url(properties.getUrl())
                .username(properties.getUsername())
                .password(properties.getPassword());

        if(properties.getType() != null) {
            dataSourceBuilder.type(properties.getType());
        }
        return dataSource;
    }
}
```

Nyní máme nastaven multitenantní datový zdroj aplikace, který umožňuje změnu aktivního datového zdroje podle potřeby. Další částí konfigurace je zabezpečení samotného přepínání zdrojů. K tomuto účelu slouží abstraktní metody *determineCurrentLookupKey* a *determineTargetDataSource* zmíněného *AbstractRoutingDataSource*. Dále je uvedena celá implementace třídy *MultitenantDataSource*, která rozšiřuje *AbstractRoutingDataSource* a implementuje uvedené metody viz kód 8.

Kód 8 - Implementace *AbstractRoutingDataSource* pro dynamické přepínání datových zdrojů

```
public class MultitenantDataSource extends AbstractRoutingDataSource {

    @Autowired
    MultitenantDataSourceMap multitenantDataSourceMap;

    @Autowired
    private DataSourceProperties properties;

    @Override
    protected Object determineCurrentLookupKey() {

        String tenantName = (String) TenantContext.getCurrentTenant();

        DataSource existingDatasource = (DataSource) multitenantDataSourceMap
                                                .getDataSourceMap().get(tenantName);

        if(existingDatasource == null && tenantName != null){

            DataSource datasource = DataSourceBuilder
                    .create()
                    .username(properties.getUsername())
                    .password(properties.getPassword())
                    .url(datasourceUrlPrefix + TenantContext.getCurrentTenant() +
                                                            datasourceUrlPostfix)
                    .driverClassName(properties.getDriverClassName())
                    .build();

            org.apache.tomcat.jdbc.pool.DataSource dataSource =
                                (org.apache.tomcat.jdbc.pool.DataSource)datasource;
            multitenantDataSourceMap.addDataSource(
                               (String)TenantContext.getCurrentTenant(),datasource);
        }

        return tenantName;
    }

    @Override
    protected DataSource determineTargetDataSource(){

        String tenantName = (String)determineCurrentLookupKey();
        DataSource dataSource = (DataSource) multitenantDataSourceMap
```

```
                                                .getDataSourceMap().get(tenantName);

        Authentication authentication = SecurityContextHolder.getContext()
                                                              .getAuthentication();

        if (authentication != null && !(authentication instanceof
                                                    AnonymousAuthenticationToken)) {
            if(tenantName == null){
                return null;
            }
        }
        if(dataSource == null)
            return super.determineTargetDataSource();

        return dataSource;
    }

}
```

Metoda *determineCurrentLookupKey* slouží ke specifikaci klíče, pod kterým má být datový zdroj hledán v kolekci datových zdrojů. V těle metody je získáno id tenanta a pokud datový zdroj pod takovýmto id ještě neexistuje, tak je vytvořen. Metoda *determineTargetDataSource* pak slouží k nalezení samotného datového zdroje v kolekci zdrojů podle získaného id tenanta. V těle metody je také ověřeno, že proběhla autentizace uživatele.

Zatím bylo vysvětleno použití vlastní databázové struktury jako úložiště uživatelů pro autentizaci pomocí Spring Security a dále pak nastavení datového zdroje pro přístup k databázi, který umožňuje přepínat mezi jednotlivými databázemi tenantů. K získání informace o tom, na jakého tenanta (databázi) má datový zdroj přepnout používá *MultitenantDataSource* objekt *TenantContext* a jeho metodu *getCurrentTenant*. Zbývá tedy vysvětlit fungování tohoto objektu a způsob nastavení aktuálního tenanta.

Jak bylo uvedeno v úvodu této části práce, je možné získat id tenanta buď z URL požadavku, nebo po přihlášení z bezpečnostního kontextu aplikace *SecurityContextHolder*. K tomuto účelu je využito systému filtrů, které mají možnost obalit zpracování požadavku vlastní funkcionalitou.

První z filtrů je spouštěn před autentizací uživatele a má za cíl získat id tenanta z adresy URL viz kód 9.

Kód 9 - Filtr pro získání id tenanta z URL

```
public class PreAuthtFilter implements Filter {

    @Override
    public void doFilter(ServletRequest req, ServletResponse res,
                         FilterChain chain) throws IOException, ServletException {

        TenantContext.setCurrentTenant(req.getParameter("tenantId"));



        try {
            chain.doFilter(req, res);
        }finally {
            TenantContext.setCurrentTenant(null);
        }
    }

}
```

Zde je vhodné upozornit na skutečnost, že požadavek je předán na další zpracování v rámci bloku *try* a v bloku *finally* je nastaven tenant na *null*. Toto je pro správnou funkcionalitu velmi důležité, protože jinak by mohlo dojí k situaci, kdy kvůli výjimce při běhu zpracování nebude tenant vymazán. To by znamenalo bezpečnostní riziko.

Druhý filtr slouží pro získání id tenanta z bezpečnostního kontextu aplikace a je zařazen za autentizaci uživatele viz kód 10.

Kód 10 - Filtr pro získání id tenanta z bezpečnostního kontextu

```
public class PostAuthFilter implements Filter {

    @Autowired
    UserDao userDao;

    @Override
    public void doFilter(ServletRequest request, ServletResponse response,
                         FilterChain chain) throws IOException, ServletException {

        Object principal = SecurityContextHolder.getContext()
                                                .getAuthentication()
                                                .getPrincipal();

        if(principal instanceof User) {

            User user = (User) principal;
            String tenantVContextu = (String) TenantContext.getCurrentTenant();

            if(tenantVContextu != null &&
                          !tenantVContextu.equals(user.getTenant().getTenant_id()))
            {
                HttpServletRequest req = (HttpServletRequest) request;
                req.logout();
                HttpServletResponse res = (HttpServletResponse) response;
                res.setStatus(401);
                return;
            }
            TenantContext.setCurrentTenant(user.getTenant().getTenant_id());

        }else {
            TenantContext.setCurrentTenant(null);
        }
```

```
        try {
            chain.doFilter(request, response);
        }finally {
            TenantContext.setCurrentTenant(null);
        }
    }
}
```

Tento filtr vyzvedne objekt uživatele z bezpečnostního kontextu a nastaví jeho id do instance typu *TenantContext*. Pokud by již bylo toto id nastaveno, tak to znamená, že již jednou přihlášený uživatel zadal do URL id tenanta. Toto je nestandardní situace a pokud by se navíc toto id tenanta v URL neshodovalo s id tenanta v bezpečnostním kontextu bude provedeno odhlášení a vrácen chybový kód 401.

Oba tyto filtry jsou v aplikaci registrovány standardním mechanismem pro registraci filtrů ve frameworku Spring. Je třeba však upozornit na nutnost dodržení správného pořadí filtrů při jejich registraci.

Zbývá doplnit implementaci třídy *TenantContext*, která drží po dobu zpracování požadavku nastavené id tenanta viz kód 11.

Kód 11 - *TenantContext* pro přístup k id tenanta

```
public class TenantContext {

    private ThreadLocal<Object> currentTenant = new ThreadLocal<>();

    public void setCurrentTenant(Object tenant) {
        currentTenant.set(tenant);
    }

    public Object getCurrentTenant() {
        return currentTenant.get();
    }
}
```

Tato implementace využívá pro uložení informace o id tenanta třídu *ThreadLocal*, která reprezentuje API pro nastavení a přístup k proměnným přístupným kdekoli ve stejném vláknu programu.

7.3 Elektronická evidence tržeb

V této části bude popsána implementace elektronické evidence tržeb jak na straně klienta, kde vznikají samotné objednávky a dochází k výpočtům kódů PKP a BKP, tak na straně serveru, který zajišťuje komunikaci s portálem elektronické evidence tržeb.

7.3.1 Vytvoření datové zprávy

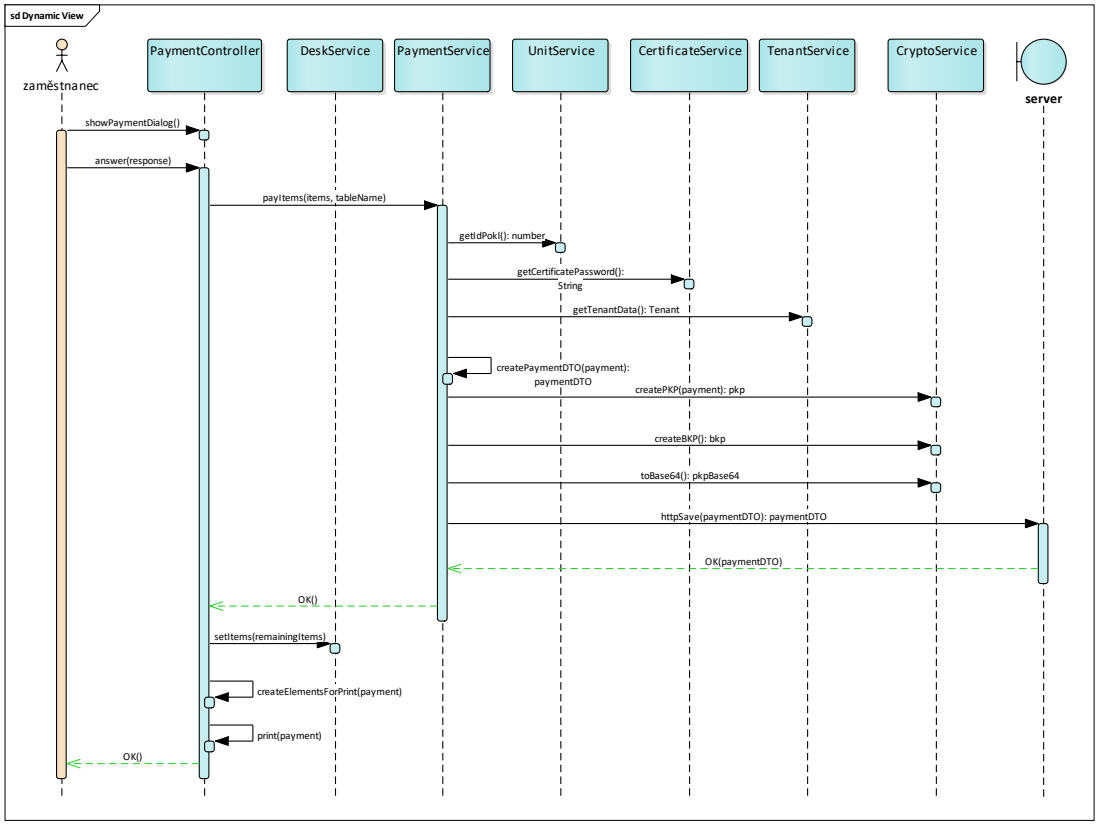
Pro vytvoření datové zprávy elektronické evidence tržeb je třeba spolupráce klientské a serverové části aplikace. V případě, že je vše v pořádku, vytvoří klient transfer objekt platby z patřičné objednávky a odešle zprávu serveru. Server

provede kontrolu položek a kódů v transfer objektu a vytvoří objekt tržby pro odeslání na portál správce daně. Server poté odešle tržbu a podle obdržené odpovědi odpoví klientovi buď chybovou zprávou, nebo opět transfer objektem platby obsahujícím uloženou platbu společně s údaji získanými od portálu správce daně (typicky kód FIK).

Pro off-line funkcionalitu je nutné ukládat neodeslané platby na straně klienta. Pokud tedy klientská aplikace po odeslání platby serveru obdrží chybovou zprávu, či dojde k výpadku komunikace, uloží klient data o platbě na své straně.

- **Klientská část aplikace**

Na obrázku 15 můžeme sledovat sekvenční UML diagram vytvoření a zaslání platby v klientské části aplikace



Obrázek 15 - Sekvenční UML diagram pro odesílání platby – klientská část

Uživatel pro odeslání platby vybere z objednávky požadované zboží a vyvolá dialog platby. Systém zobrazí dialog a uživatel ho potvrdí. Na základě položek vybraných uživatelem k platbě vytvoří systém data transfer objekt platby.

Zde je z pohledu implementace klíčovou částí výpočet kódů PKP a BKP. Tyto kódy je nutné vypočítat na straně klienta, protože poplatník je povinen je uvést na účtence i v případě výpadku spojení se systémem elektronické evidence.

Kód 12 ukazuje způsob výpočtu PKP v jazyce Javascript s použitím funkcí knihovny Forge.

Kód 12 - Vytvoření kódu PKP na straně klienta

```
function createPKP(paymentDTO){

    var plaintext = paymentDTO.dicPopl + '|' + paymentDTO.idProvoz +
                '|' + paymentDTO.idPokl + '|' + paymentDTO.poradCis +
                '|' + paymentDTO.datTrzby +
                '|' + paymentDTO.celkTrzba;

    var pkcs12B64 = certificateService.getCertificate();
    var pass = forgeService.decryptAES(
    certificateService.getCertificatePassword())

    var p12Der = forge.util.decode64(pkcs12B64);
    var p12Asn1 = forge.asn1.fromDer(p12Der);
    var p12 = forge.pkcs12.pkcs12FromAsn1(p12Asn1, pass);

    var bags = p12.getBags({bagType:
    forge.pki.oids.pkcs8ShroudedKeyBag});

    var bag = bags[forge.pki.oids.pkcs8ShroudedKeyBag][0];
    var key = bag.key;

    var md = forge.md.sha256.create();
    md.update(plaintext, 'utf8');

    var signature = key.sign(md);

    return forge.util.encode64(signature);

}
```

Nejdříve je vytvořen text, ze kterého bude výpočet proveden. Poté je pomocí komponenty pro správu certifikátu získán certifikát a heslo. Z certifikátu je dále získán privátní klíč poplatníka. Vytvořený text je převeden pomocí hashovací kryptografické funkce SHA256 a podepsán získaným privátním klíčem. Nakonec je kód převeden do Base64 kódování.

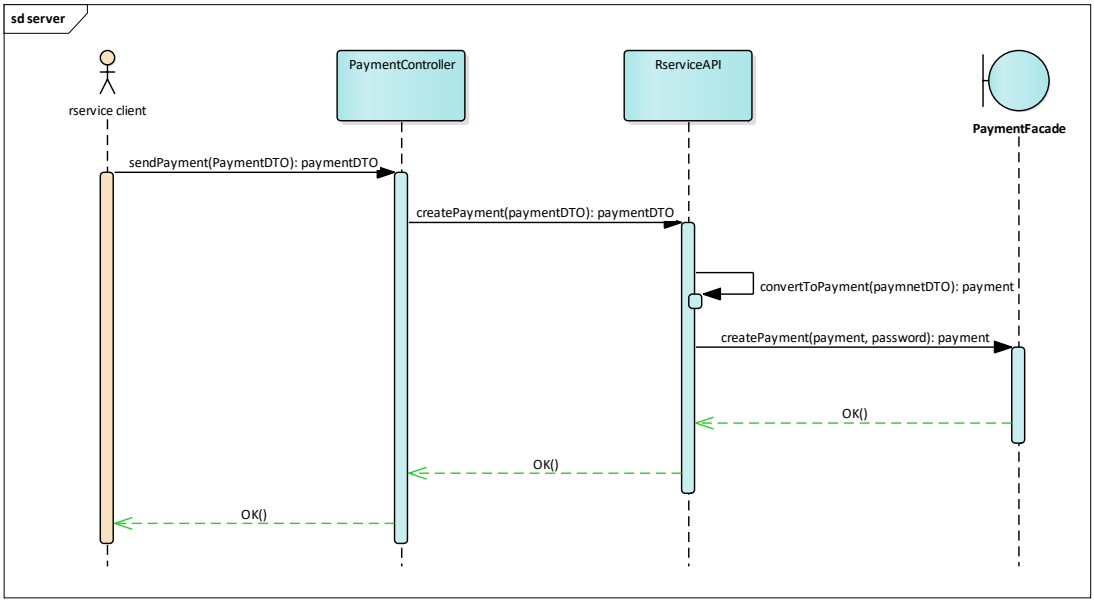
Bezpečnostní kód poplatníka je pak vypočten z hodnoty PKP viz kód 13.

Kód 13 - Vytvoření kódu PKP na straně klienta

```
function _createBKP(pkpValue){

    var md = forge.md.sha1.create();
    md.update(pkpValue);

    var cryptedpkp = md.digest();
    var hexbkp = cryptedpkp.toHex();

    var bkp = hexbkp.substring(0,  8)+ "-" +
              hexbkp.substring(8,  16) + "-" +
              hexbkp.substring(16, 24) + "-" +
              hexbkp.substring(24, 32) + "-" +
              hexbkp.substring(32, hexbkp.length);

   return bkp;

}
```

Kód PKP je převeden pomocí algoritmu SHA1 a poté vyjádřen hexadecimálně, nakonec jsou přidány požadované pomlčky.

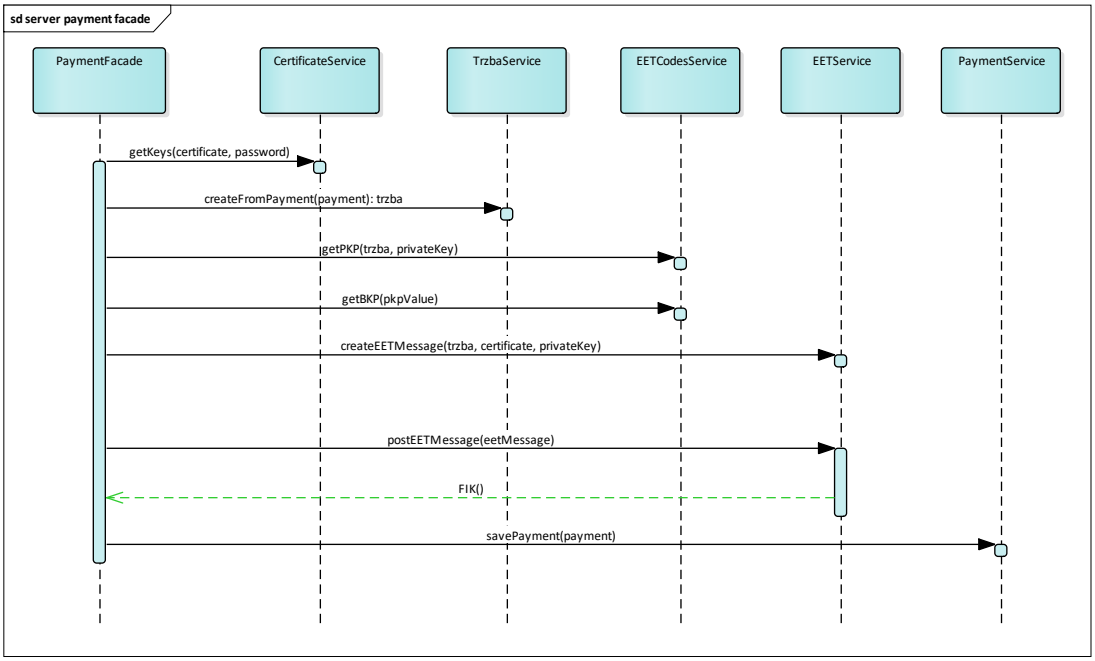
- **Serverová část**



Obrázek 16 – Sekvenční UML diagram pro zpracování platby – serverová část

Serveru je zaslán požadavek na provedení platby a server prostřednictvím kontroléru platby tento požadavek přijme. Požadavek obsahuje data, která jsou z formátu JSON převedena do data transfer objektu Javy. Tento objekt je použit jako parametr volání metody pro vytvoření platby v instanci třídy *RserviceAPI*. Pro samotné zpracování požadavku je volána metoda instance třídy *PaymentFacade*. V případě úspěšného vytvoření a odeslání platby je odeslána odpověď klientovi. Tato odpověď obsahuje data o uložené a zpracované platbě.

Samotný servis *PaymentFacade* však potřebuje k úspěšnému splnění úlohy provést několik dílčích kroků, jak můžeme sledovat na obrázku 17.



Obrázek 17 – Sekvenční UML diagram pro zpracování platby

Nejprve je nutné získat z úložiště certifikát uživatele a z něj privátní klíč. K tomuto účelu slouží instance třídy *CertificateService*. Poté je nutné vytvořit instanci třídy *Trzba*. Třída *Trzba* je vygenerována pomocí schématu xsd, které je součástí popisu rozhraní elektronické evidence tržeb. Potom je pomocí instance typu *EETService* vygenerována SOAP zpráva a tato je odeslána na portál elektronické evidence tržeb.

Z pohledu implementace jsou zajímavé části pro vygenerování SOAP zprávy a její samotné odeslání. Pro generování tříd ze schématu xsd byl použit mapovací nástroj JAXB 2.0. Pro vytvoření zprávy je tedy použita funkce uvedená v kódu 14.

Kód 14 - Vytvoření elementu tržby SOAP zprávy

```
private JAXBElement<TrzbaType> createTrzba(TrzbaType trzba) {

   ObjectFactory objectFactory = new ObjectFactory();
   return objectFactory.createTrzba(trzbaType);

}
```

Dále je použita *MessageFaktory* z balíčku *javax.xml.soap* k vytvoření SOAP zprávy a přidán element *<SOAP-ENV:Header>* viz kód 15.

Kód 15 - Vytvoření výchozí SOAP zprávy

```
SOAPMessage soapMessage = MessageFactory.newInstance().createMessage();
SOAPPart soapPart = soapMessage.getSOAPPart();
SOAPEnvelope soapEnvelope = soapPart.getEnvelope();
SOAPHeader header soapEnvelope.getHeader();

soapEnvelope.addHeader();
```

Do takto vytvořené hlavičky SOAP zprávy jsou přidány elementy *<wsse:Security>* a *<wsse:BinarySecurityToken>* viz kód 16.

Kód 16 - Přidání elementů SOAP zprávy

```
SOAPHeaderElement wsseHeaderElement =
                                        header.addHeaderElement(soapEnvelope.createName(
                                                "Security", "wsse", "http://docs.oasis-
                                                 open.org/wss/2004/01/oasis-200401-wss-
                                                 wssecurity-secext-1.0.xsd"));

wsseHeaderElement.setMustUnderstand(true);
SOAPElement binarySecurityTokenElement = wsseHeaderElement.addChildElement(
                                                              "BinarySecurityToken","wsse");
binarySecurityTokenElement.addTextNode(encodedCert);
```

Podobně je přidán do SOAP zprávy element *<SOAP-ENV:Body >* a do něj samotná zpráva tržby viz kód 17.

Kód 17 - Vložení tržby do SOAP zprávy

```
SOAPBody soapBody = soapEnvelope.getBody();
marshaller.marshal(trzbaSoap,document);
soapBody.addDocument(document);
```

Další částí zprávy je její podpis. Podepisuje se pouze element *Body* viz kód 18. Je zde použita *XMLSignatureFactory* z balíčku *javax.xml.crypto.dsig*. Parametry *signedInfo* a *keyInfo* jsou nastaveny tak, aby odpovídaly zadání pro elektronickou evidenci tržeb.

Kód 18 - Nastavení *XMLSignatureFactory* pro podepsání zprávy

```
String providerName = System.getProperty("jsr105Provider",
                                          "org.jcp.xml.dsig.internal.dom.XMLDSigRI");

XMLSignatureFactory sigFactory = XMLSignatureFactory.getInstance("DOM", (Provider)

Class.forName(providerName).newInstance());
XMLSignature sig = sigFactory.newXMLSignature(signedInfo, keyInfo);
```

Nakonec je podpis vložen do elementu *</SOAP-ENV:Header>* a celá zpráva je převedena na hodnotu typu *String*.

Odeslání samotné SOAP zprávy se provádí s využitím knihoven Apache HttpComponents. Pro splnění požadavků na maximální dobu odezvy je nutné nastavit maximální dobu čekání na připojení a také maximální dobu čekání na data po navázání spojení. Nastavení těchto parametrů ilustruje kód 19.

Kód 19 – Nastavení pro odeslání SOAP zprávy

```
RequestConfig requestConfig = RequestConfig.copy(defaultRequestConfig)
                                .setSocketTimeout(socketTimeout)
                                .setConnectTimeout(connectTimeout)
                                .build();

httpPost.setConfig(requestConfig);
```

Odeslání zprávy a převod odpovědi na Java objekt pak s použitím lambda funkce ukazuje kód 20.

Kód 20 - Odeslání SOAP zprávy a zpracování odpovědi

```
ResponseHandler<EETResponseWrapper> responseHandler = response -> {/*…*/}
EETResponseWrapper eetResponseWrapper = httpclient.execute(httpPost,
responseHandler);
```

7.4 Off-line funkcionalita

V off-line režimu nabídne aplikace omezenou funkcionalitu s cílem umožnit elektronickou evidenci tržby i v případě výpadku služeb aplikačního serveru, či internetového připojení. Pro splnění těchto požadavků je nutné zabezpečit uložení některých dat na straně klienta. K tomuto účelu bylo využito Service Worker API pro uložení provozních dat (data o zboží), localStorage webového prohlížeče pro uložení údajů o tenantovi a PouchDB pro uložení aktuálních objednávek a neodeslaných tržeb daného zařízení.

7.4.1 Přihlášení uživatele a registrace služby Service Worker

Po úspěšném přihlášení uživatele je zaregistrována služba service worker, která při své inicializaci (fáze životního cyklu *install*) provede načtení statického obsahu a dat o zboží a stolech do své paměti. Tato data se v případě, že aplikace má spojení s aplikačním serverem obnovují po uplynutí stanoveného intervalu, jinak po obnovení připojení. Obnovení dat v paměti probíhá na pozadí, což znamená, že každý dotaz, jehož URL je registrována ve službě service worker je přerušen a data jsou načtena z paměti. V případě, že byl překročen mezní limit pro invalidaci dat v paměti, odešle na pozadí služba service worker dotaz na aplikační server a aktualizuje data ve své paměti. Tento přístup však nemá plnou podporu v objektu, který reprezentuje paměť prohlížeče, jelikož neexistuje možnost, jak zjistit, kdy byla data do paměti uložena. Z tohoto důvodu bylo nutné vytvoření datové struktury, která drží informaci o posledním čase aktualizace každé URL. Dále je obsah paměti také aktualizován při každém dotazu aplikace, který mění data na serveru (typicky POST, PUT, DELETE). Tímto způsobem je zajištěna jak platnost dat, tak rychlá odezva.

Další data, která aplikace nutně pro off-line funkcionalitu potřebuje, jsou data o vlastním tenantovi. Tato data jsou také získána po přihlášení uživatele a aplikace je obdrží společně s odpovědí na odeslané identifikační údaje. Data jsou uložena do lokálního úložiště prohlížeče (localStorage), odkud jsou aplikací načítána, kdykoli je to nutné. Pokud aplikace požaduje data o tenantovi a tato nejsou v lokálním úložišti dostupná, je uživatel odhlášen. V případě, že je aplikace v módu off-line, a data o tenantovi nejsou dostupná, je toto reportováno uživateli jako chyba a je mu sděleno, že pro obnovení funkčnosti aplikace je třeba připojení k aplikačnímu serveru. Samotnou implementaci služby service worker ilustruje kód 21.

Kód 21 - Implementace služby Service Worker

```
var CACHE_NAME = 'my-site-cache-v1';
var CACHE_DATA_INVALID_TIMEOUT = 1000; //1 sec
var nextUpdates = {};

var StaticResourcesToCache = [

    '' //zastupuje root webu (prejde na index.html)
    'css/angular_app.css',

];

Var dataResourcesToCache = [
    'rest/category',
    'rest/table',

];

function getNextUpdateTime(url){

    var nextUpdate = new Date().getTime() -1;
    for (var key in nextUpdates) {
        if(url === String(key)){
            nextUpdate = nextUpdates[key];
        }
    }
    return nextUpdate;

}

function fetchToCacheMany(urlArr, target){

    var promises = [];
    urlArr.forEach(function (url, index) {
        promises.push(
            fetchToCache(target.registration.scope + url)
        );
    });
    return Promise.all(promises);

}

function fetchToCache(request){

    return fetch(request, {credentials: 'include'})
        .then(function (webResponse) {
                if (webResponse && webResponse.status == 200 &&
                    webResponse.type == 'basic') {
                    return caches.open(CACHE_NAME)
                        .then(function (cache) {
                            nextUpdates[request.url] =
                                   new Date().getTime()
                                   + CACHE_DATA_INVALID_TIMEOUT;

                            cache.put(request, webResponse);
                        });
                }else{
                    throw webResponse.status;
                }
        });

};

self.addEventListener('install', function(event) {

    event.waitUntil(
        caches.open(CACHE_NAME)
            .then(function(cache) {
                var promiseArr = [];
```

```
                promiseArr.push(fetchToCacheMany(
                            StaticResourcesToCache, event.target));
                promiseArr.push(fetchToCacheMany(
                            dataResourcesToCache, event.target));
                return Promise.all(promiseArr);
            })
    );
});

self.addEventListener('fetch', function(event) {

    event.respondWith(
        caches.match(event.request)
            .then(function(response) {
                    if (response) {
                        var nextUpdateSec =
                               getNextUpdateTime(event.request.url);
                        if (nextUpdateSec < new Date().getTime()) {
                            fetchToCache(event.request)
                        }
                        return response;
                    }

                    return fetch(event.request)
                        .then(function (response) {
                            if(event.request.method !== 'GET') {
                              fetchToCacheMany(dataResourcesToCache,
                                               event.target);
                            }
                            return response;
                        })
            })
    );
});
```

7.4.2 PouchDB a správa aktuálních objednávek

Pro zabezpečení off-line funkcionality aplikace je nutné držet aktuální stav objednávek na straně klienta. K tomuto účelu byla v aplikaci využita databáze PouchDB. Použití této databáze ve spojení s frameworkem AngularJS sebou přináší několik aspektů, které je nutné vyřešit. Předně se zde jedná o propagaci změn databáze do grafické části aplikace, a tudíž o propojení změny v databázi s digest cyklem frameworku Angular. Dále je nutné udržovat správnou verzi daného dokumentu, protože dokumentová databáze je na verzích závislá především v ohledu optimistických zámků. K tomuto účelu byl vytvořen modul *deskService* viz kód 22.

Kód 22 - Implementace služby pro správu objednávek na straně klienta

```
rserviceServices.factory('deskService', function($rootScope, $log, itemService,
                                                  pouchService, alertService){
    var db = pouchService.db;
    var _actualState = {};
    _actualState.actualOrder = {graphicItems:[], totalPrice: 0, _id: null};

    function _changeDesk(deskId){
        _actualState.actualOrder = {graphicItems:[], totalPrice: 0, _id: "table_" +
                                    deskId};
        return updateActualOrder();
    }
```

```
    function updateActualOrder(){
        return db.get(_actualState.actualOrder._id)
            .then(function (resp) {
                $rootScope.$apply(function () {
                    $log.debug('updating actual order from db', resp);
                    _actualState.actualOrder = resp;
                })
            })
            .catch(function (err) {
                $log.warn('Error getting deskState from pouch! NONEXIST?', err);
            })
    }

    db.changes({
        since: 'now',
        live: true
    }).on('change', updateActualOrder);

});
```

Řešením obou problémů je využití funkce *changes* na objektu databáze. Tato funkce umožňuje zaregistrovat tzv. *callback* funkci, která je volána vždy při změně dat v databázi a v rámci této funkce je poté spuštěn digest cyklus AngularJS frameworku pomocí metody *$apply* na objektu *$rootScope*.

Samotná data aktuální objednávky jsou poté zpřístupněna pomocí proměnné *_actualState*. Tímto způsobem se komponenty využívající tento servis nemusí zabývat spouštěním digest cyklu ani správným číslem verze dokumentu v databázi. Pokud navíc budou klienti a jejich databáze mezi sebou synchronizováni, tak budou data aktuální na každém z klientů.

8 Shrnutí výsledků

Vytvořená aplikace poskytuje klientům velmi podobnou funkcionalitu jako další podobné produkty na trhu a díky přesunutí velké části zátěže na jednotlivé klientské PC vykazuje velmi malé nároky na výkon serveru na jednoho klienta. Také se ukazuje, že vytvoření aplikace s touto funkcionalitou je v silách jednoho programátora. Tyto dva aspekty umožňují poskytovat podobné služby podstatně levněji, než je tomu u ostatních komerčních produktů.

Možné zjednodušení aplikace nabízí využití některé z implementací standardu WS-Security, se kterým by měla být služba pro elektronickou evidenci tržeb kompatibilní. Bohužel se v současné podobě služby a jednotlivých implementací nepodařilo tímto způsobem vytvořit validní SOAP zprávu, kterou by portál elektronické evidence přijal.

Více než roční zkušenosti s reálným provozem ukazují, že spolehlivost a robustnost aplikace jsou dostatečné a funkcionalita se blíží větším a mnohem dražším nativním aplikacím.

Hlavní zaznamenané problémy většinou vyplývají ze samotné koncepce webové aplikace.

V první řadě zde existuje jistá nechuť uživatelů používat webovou aplikaci v okně prohlížeče, a to zvláště pokud aplikace pracuje s citlivými daty uživatele. V případě nativní aplikace, která je také pouhým klientem některé cloudové služby podobný problém nenastává. Tento problém subjektivního dojmu z aplikace by mohl být do budoucna vyřešen lepší podporou pro aplikace ve webových prohlížečích. V tomto ohledu se zdá být krokem zpět zrušení podpory tzv. Chrome apps pro Windows, Linux a Mac OS. Tato funkcionalita ve webovém prohlížeči Chrome umožňovala spuštění webové aplikace samostatnou ikonou na ploše operačního systému a její běh v okně imitujícím vzhled nativní aplikace. Do budoucna však lze očekávat, že podobnou podporu webových aplikací budou nabízet všechny webové prohlížeče, stejně jako podporu pro lepší komunikaci se samotným hardware pomocí nových API.

Dalším problémem je pak závislost aplikace na samotném webovém prohlížeči. Podpora různých prohlížečů je stále velmi složitá vzhledem k rozdílným implementacím Javascriptu.

Pohodlný provoz aplikace je také do jisté míry závislý na zkušenostech uživatele se samotným prohlížečem. Nejvíce řešených problémů se týkalo nedostupnosti internetového připojení, ovládání samotného prohlížeče, nebo vůbec práce s PC všeobecně.

9 Závěry a doporučení

V průběhu návrhu a implementace se osvědčily všechny vybrané nástroje a potvrdila se jejich vhodnost pro tvorbu moderní webové aplikace.

Postupy definované v UML a UP se podařilo bez větších problémů aplikovat na skutečné problémy a jejich přínos pro přehlednost celého řešení je neoddiskutovatelný.

Použité frameworky Spring a AngularJS splnily všechna očekávání a umožnily použití pokročilých programovacích technik jak v serverové, tak i v klientské části aplikace.

Implementace pomocí vývoje řízeného testy se stala hlavním nástrojem pro vybudování robustní aplikace a všechny výhody tohoto přístupu popsané v teoretické části se potvrdili v praxi.

Společně tyto nástroje umožnili dosažení všech stanovených cílů této diplomové práce a vytvořená aplikace je plně funkční a použitelná v produkčním prostředí. Podařilo se také dodržet požadavek na nízkou poměrnou zátěž serveru na jednoho klienta a tím také na nižší výslednou cenu nabízených služeb oproti konkurenčním řešením.

Hlavním doporučením k současnému stavu je implementovat zabezpečení komunikace s portálem elektronické evidence tržeb pomocí některé z implementací WS-Security standardu a tím dále zlepšit strukturu aplikace.

Dále je třeba rozšiřovat současnou funkcionalitu v reakci na nové požadavky uživatelů především ohledně různých zobrazení statistických dat a jejich exportů.

10 Citovaná literatura

[1] **Arlow, Jim a Neustadt, Ila. 2011.** *UML2 a unifikovaný proces vývoje aplikací.* Brno : Computer press, as, 2011. ISBN 978-80-251-1503-9.

[2] **Ashmore, Derek C. 2014.** *The Java EE architect´s handbook: how ro be a successful application architect for Java EE applications.* Bolingbrook : DVT Press, 2014. ISBN 978-0-9729548-8-4.

[3] **Česko.** Vyhláška č. 269 ze dne 16.08.2016 o způsobu tvorby podpisového kódu poplatníka a bezpečnostního kódu poplatníka. In Sbírka zákonů č. 269/2016 České republiky. ISSN 1211-1244.

[4] **Česko.** Zákon č. 112 ze dne 16.03.2016 o evidenci tržeb. In Sbírka zákonů České republiky.

[5] **digitalbazaar. 2017.** Forge. *github.* [Online] 2017. [Citace: 10. 04 2017.] Dostupné z: https://github.com/digitalbazaar/forge.

[6] **Generální finanční ředitelství. 2016.** Elektronická evidence tržeb - Formát a struktůra údajů o evidované tržbě. *Web etržby.* [Online] 13. 10 2016. [Citace: 12. 03 2017.] Dostupné z: http://www.etrzby.cz/assets/cs/prilohy/EET_popis_rozhrani_v3.1.1.pdf.

[7] **Generální finanční ředitelství. 2017.** etržby - elektronická evidence tržeb. *Web etržby.* [Online] 07. 03 2017. [Citace: 25. 03 2017.] Dostupné z: http://www.etrzby.cz.

[8] **Google. 2017.** *Angular Material.* [Online] 2017. [Citace: 22. 02 2017.] Dostupné z: https://material.angularjs.org/latest/.

[9] **Harvey, Dale a Nolan, Lawson. 2017.** pouchdb - The Database that Syncs. *pouchdb.* [Online] 2017. [Citace: 15. 04 2017.] Dostupné z: https://pouchdb.com/.

[10] **KASA FIK. 2017.** funkce pokladního systému. *KASA FIK.* [Online] 2017. [Citace: 12. 04 2017.] https://www.kasafik.cz/web/cs/pokladni-system-funkce/.

[11] **Madeyski, Lech. 2010.** *Test-Driven Development.* Berlin : Springer Berlin Heidelberg, 2010. ISBN 978-3-642-04287-4.

[12] **Mäkinen, Simo a Jürgen, Münch. 2014.** *Effects of Test-Driven Developmen: A Comparative Analysis of Empirical Studies.* Berlin : Springer Berlin Heidelberg, 2014. ISBN 978-3-319-03601-4.

[13] **markeeta. 2017.** O markeete. *markeeta.* [Online] 2017. [Citace: 12. 04 2017.] Dostupné z: https://www.markeeta.cz/o-markeete.

[14] **Mellor, Stephen. 2004.** Agile MDA. [Online] 2004. Dostupné z: http://www.omg.org/mda/mda_files/AgileMDA.pdf.

[15] **Mozilla. 2017.** Service Worker API. *Mozilla Deevloper Network.* [Online] 2017. [Citace: 01. 04 2017.] Dostupné z: https://developer.mozilla.org/en-US/docs/Web/API/Service_Worker_API.

[16] **OMG. 2016.** Unified Modeling Language. [Online] 2016. [Citace: 28. 03 2017.] http://www.omg.org/spec/UML/index.htm.

[17] **Oracle Company a. 2017.** Java Platform Standard Edition 8 Documentation. *Oracle.* [Online] 2017. [Citace: 20. 02 2017.] Dostupné z: https://docs.oracle.com/javase/8/docs/.

[18] **Oracle Company b. 2017.** New to Java Programming Center - Documentation. *Oracle.* [Online] 2017. [Citace: 25. 02 2017.] Dostupné z: http://www.oracle.com/technetwork/topics/newtojava/documentation/documentation-jsp-142568.html.

[19] **Oracle Company. 2016.** Overview (Release 7). *Oracle.* [Online] 2016. [Citace: 01. 03 2017.] Dostupné z: https://docs.oracle.com/javaee/7/tutorial/overview.htm.

[20] **Parsons, David. 2012.** *Foundational Java.* London : Springer, 2012. ISBN 978-1-4471-2478-8.

[21] **Schaefer, Chris, Ho, Clarence a Harrop, Rob. 2014.** *Pro Spring.* New York : Apress, 2014. ISBN 978-1-4302-6151-3.

[22] **Schildt, Herbert. 2012.** *Java 7 Výukový kurz.* Brno : Computer Press, 2012. ISBN 978-80-251-3748-2.

[23] **Slater, Noah. 2014.** An Introduction To PouchDB. *Engine Yard.* [Online] 2014. [Citace: 15. 04 2017.] Dostupné z: https://blog.engineyard.com/2014/an-introduction-to-pouchdb.

[24] **Smith, Chris. 2015.** Angular Basics. *angularjsbook.* [Online] 2015. [Citace: 25. 03 2017.] Dostupné z: http://www.angularjsbook.com/.

[25] **Spring. 2016.** spring-boot. [Online] 2016. [Citace: 18. 02 2017.] https://projects.spring.io/spring-boot/.

[26] **Zakas, Nicholas, Z. 2009.** *Javascript pro webové vývojáře.* Brno : Computer press, 2009. ISBN 978-80-251-2509-0.

Obsah přiloženého CD

CD obsahuje elektronickou verzi práce ve formátu PDF a soubory samotné aplikace. V adresáři *src* se nachází zdrojové kódy. Soubor *pom.xml* obsahuje informace o závislostech pro nástroj Maven.

V souboru *README.docx* je popsán postup pro zprovoznění projektu v prostředí InteliJ IDEA a způsob přidání aplikace na plochu PC s pomocí prohlížeče Chrome.