ZADÁNÍ DIPLOMOVÉ PRÁCE

I. OSOBNÍ A STUDIJNÍ ÚDAJE

Příjmení: **Zářecký** Jméno: **Pavel** Osobní číslo: **384532**

Fakulta/ústav: **Fakulta elektrotechnická**

Zadávající katedra/ústav: **Katedra počítačů**

Studijní program: **Otevřená informatika**

Studijní obor: **Softwarové inženýrství**

II. ÚDAJE K DIPLOMOVÉ PRÁCI

Název diplomové práce:

Vylepšení a přesun restauračního systému na platformu ASP.NET Core

Název diplomové práce anglicky:

Restaurant System Upgrade and Migration to ASP.NET Core Platform

Pokyny pro vypracování:

Analyzujte technologická omezení existující implementace serverové části restauračního systému CashBob[2] postaveného na JAVA technologiích. Nastudujte problematiku vývoje aplikací pro platformu ASP.NET [4] a analyzujte výhody/nevýhody přesunu stávající aplikace na tuto platformu.
Nově navrhněte a naprogramujte serverovou část tak, aby odstraňovala zásadní nedostatky stávajícího řešení. Práci provádějte iterativně, průběžně dokumentujte, testujte a nasazujte.
Do systému doplňte podporu elektronické evidence tržeb EET a podporu tisku účtenek.
Testování kompatibility přeprogramovaných rozhraní provádějte především pomocí existující frontendové Android aplikace[3]. Výsledný systém porovnejte s existujícími zavedenými systémy.

Seznam doporučené literatury:

[1] LARMAN, Craig a Chris RUPP. Applying UML and patterns: introduction to object-oriented analysis and design and iteractive development. 3rd ed. New Jersey: Prentice-Hall, 2005, xviii, ISBN 01-314-8906-2.
[2] ČERVENKA, Tomáš. Webové rozhraní restauračního systému, Praha, 2016. Dostupné také z: http://hdl.handle.net/10467/64854
[3] ZÁŘECKÝ, Pavel. Vývoj aplikací na platformě Android, Praha, 2016. Dostupné také z: http://hdl.handle.net/10467/64634
[4] ESPOSITO, Dino. Programming asp.net core. Indianapolis, IN: Microsoft Press, 2018. ISBN 9781509304417.

Jméno a pracoviště vedoucí(ho) diplomové práce:

Ing. Martin Komárek, kabinet výuky informatiky FEL

Jméno a pracoviště druhé(ho) vedoucí(ho) nebo konzultanta(ky) diplomové práce:

Datum zadání diplomové práce: **26.02.2019** Termín odevzdání diplomové práce: **05.01.2021**

Platnost zadání diplomové práce: **19.02.2021**

Ing. Martin Komárek podpis vedoucí(ho) práce	podpis vedoucí(ho) ústavu/katedry	prof. Ing. Pavel Ripka, CSc. podpis děkana(ky)

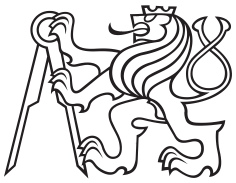
III. PŘEVZETÍ ZADÁNÍ

Diplomant bere na vědomí, že je povinen vypracovat diplomovou práci samostatně, bez cizí pomoci, s výjimkou poskytnutých konzultací.
Seznam použité literatury, jiných pramenů a jmen konzultantů je třeba uvést v diplomové práci.

. _____
Datum převzetí zadání

_____
Podpis studenta

Diplomová práce



České
vysoké
učení technické
v Praze

F3

Fakulta elektrotechnická
Katedra počítačů

Vylepšení a přesun restauračního systému na platformu ASP.NET Core

Pavel Zářecký

Školitel: Ing. Martin Komárek
Obor: Softwarové inženýrství
Zaměření: Otevřená informatika
Leden 2021

Poděkování

Chtěl bych zde poděkovat vedoucímu své práce, Ing. Martinu Komárkovi, za jeho vedení a rady. Rád bych zde také poděkoval svým rodičům a příbuzným za jejich dlouhodobou podporu při mém studiu a mé díky patří i všem ostatním, kteří mě při studiu podporovali.

Prohlášení

Prohlašuji, že jsem předloženou práci vypracoval samostatně, a že jsem uvedl veškeré použité informační zdroje v souladu s Metodickým pokynem o dodržování etických principů při přípravě vysokoškolských závěrečných prací.

V Praze, 5. ledna 2021

Abstrakt

Cílem diplomové práce je analýza, vylepšení a přesun restauračního systému CashBob na platformu ASP.NET Core. Systém CashBob vznikal v předchozích letech iterativně, především v rámci závěrečných prací studentů ČVUT.

Klíčová slova: ASP.NET Core, Cloud, REST, EET, Iterativní vývoj

Školitel: Ing. Martin Komárek

Abstract

The main goal of this thesis is the analysis, improvement and transfer of the restaurent system CashBob to the ASP.NET Core platform. The Cashbob system was created iteratively over the last years, mainly in the form of theses of the ČVUT students.

Keywords: ASP.NET Core, Cloud, REST, EET, Iteration based development

Title translation: Improvement and transfer of a restaurant payment system to the ASP.NET Core platform

Obsah

Obrázky

Tabulky

Zkratky

Zkratka	Význam
REST	Representational state transfer
API	Application program interface
XML	Extensible markup language
ASP	Active server pages
EET	Elektronická evidence tržeb
RMI	Remote method invocation
JAVA SE	Java standard edition
FIK	Diskální identifikační symbol
SOAP	Simple object access protocol
HTTP	Hypertext transfer protocol
MVC	Model, view, controller
DTO	Data transfer object
SQL	Structured query language
CRUD	Create, Read, Update, Delete
JSON	JavaScript object notation
DLL	Dynamically linked libary
PDF	Portable document format
URI	Uniform resource identifier
UI	User interface
DAO	Data access object
JPA	Java persistence API
BKP	Bezpečnostní kód poplatníka
PKP	Podpisový kód poplatníka
PAAS	Platform as a service
RQ	Requirement
EF	Entity Framework
SSL	Secure Sockets Layer
LINQ	Language Integrated Query
JWT	Json Web Token
HTML	Hypertext Markup Language

Kapitola 1
Úvod

Ve své bakalářské práci jsem se, mimo jiné, věnoval vývoji Android aplikace pro číšníky, komunikující přes REST rozhraní s pokladním systémem CashBob[Zá16]. Navazoval jsem převážně na práci jiného studenta[Hog16], který již vytvořil funkční základ aplikace.

Vzhledem k tomu, že v nedávné době vstoupila v platnost povinnost elektronické evidence tržeb – EET[1], bylo by nutné k zajištění aktuálnosti upravit nejen tento funkční základ aplikace, ale i serverovou část tak, aby EET podporovaly.

A dále také proto, že se v následujících kapitolách ukáže, že pro současný kód (až na samotný základ pokladního modulu) již nadále efektivně neexistuje dokumentace, bylo v rámci zadání rozhodnuto, že bude aplikace přepsána a znovu navržena tak, aby byly zohledněny nedostatky současné verze a aplikace tak byla vylepšena.

Přepsání aplikace jako takové otevřelo také možnost převést aplikaci na platformu jinou než Java, pokud by toto bylo výhodné - vybrán byl framework ASP.NET Core[cor], který navazuje na původní ASP.NET, ale na rozdíl od něj je nejen multiplatformní, ale hlavně také open-source. v Následujících kapitolách tedy bude tento framework porovnán s technologiemi stojícími na Javě, ve kterých je současná aplikace psána.

1.1 Pokladní systém Cashbob

CashBob jako takový je studentský projekt, který již za dobu svého vývoje doznal spousty změn, jmenovitě například přesun od lokální desktop aplikace postupně ke kombinaci více modulů komunikujících přes rozhraní Java RMI[rmi] až k zavedení standardizovaného multiplatformního API REST[res]. Ve všech svých iteracích se ale primárně vždy jednalo o desktop aplikaci, a to i v případě své poslední verze vytvořené Tomášem Červenkou v rámci jeho bakalářské práce[Če16].

[1] Zákon vstoupil pro pohostinství v platnost 1. prosince 2016 [eetb]

1.2 Motivace

V zadání této diplomové práce mám stanoveny mimo jiné tyto hlavní body:

- Analýza technologických a implementačních omezení existující implementace
- Analýza výhod/nevýhod přesunu aplikace na platformu ASP.NET Core
- Návrh nové aplikace odstraňující hlavní omezení aplikace současné, ideálně bez zavedení nových nedostatků

V současné době také panuje trend, při kterém se aplikace a systémy, historicky provozované lokálně, stěhují na Cloud[clo], za účelem co největšího zjednodušení přístupu uživateli a odstranění nutnosti vlastního technického zázemí - serverů. Toto je možné především proto, že stabilní, vysokorychlostní a především neomezený internet je v současné době (v České republice) již téměř samozřejmostí[int]. Tento trend je již možné pozorovat i v oblasti pokladních systémů pro malé podniky - existuje totiž hned několik takovýchto aplikací s cloud funkcionalitou[pok]. Využil bych tedy rád tuto příležitost a aplikaci navrhl tak, aby uživateli nabízela cloud funkcionalitu, obdobnou té, kterou již nabízejí konkurenční aplikace. Funkcionalitu obsaženou v těchto konkurenčních aplikacích proto zohledním při návrhu funkčních požadavků a připravím jednoduchou rešerši.

1.3 Struktura dokumentu

Dokument je standardně dělen na analýzu, prvotní návrh, jednotlivé iterace vývoje a zhodnocení práce. Součástí analýzy je seznam důvodů pro přechod na platformu ASP.NET Core, prvotní návrh pak zohledňuje celkovou funkcionalitu, kterou bych si přál v průběhu vývoje implementovat. Pokud by se v průběhu iterací ukázalo, že některá navržená funkcionalita by měla být změněna, budou jednotlivé iterace také obsahovat pozměněný návrh. Testování funkcionality pak bude probíhat průběžně v každé z iterací. Závěrem aplikaci porovnám s konkurenčními produkty na českém trhu a shrnu celkový výsledek práce.

S ohledem k jednomu se zadání, ve kterém mi bylo určeno nastudovat problematiku vývoje aplikací pro ASP.NET Core a také vzhledem k tomu, že jsem nikdy v minulosti nepřišel do kontaktu ani s prostředím .NET, ani s programovacím jazykem C#, využiji inkrementální způsob vývoje i k tomu, abych v každé inkrementaci popsal (pro mě) nově nalezené způsoby řešení a ostatní problémy, na které jsem při práci narazil a způsob jejich řešení.

Kapitola 2
Analýza

Představím platformu ASP.NET Core, včetně srovnání s platformou Java SE + Spring, ve které je původní aplikace implementována. Závěrem bude analýza výhod a nevýhod, které by případný přesun aplikace na novou architekturu představoval.

2.1 Popis problému a analýza stávající aplikace

Jak již bylo nastíněno v úvodu, restaurační systém Cashbob sestává ze dvou části - Serverové aplikace psané v jazyce Java a lehkého klienta pro číšníky, který běží na tabletu s operačním systémem Android a číšník ho má vždy k dispozici, spolu s přenosnou Bluetooth tiskárnou pro tisk účtenek.

Serverová část aplikace je samostatným funkčním celkem, který obsahuje jak funkční část aplikace, tedy backend který se stará o zpracování a ukládání dat, tak uživatelské rozhraní, které je přístupné ve webovém prohlížeči. Implementován je i tisk - každou účtenku je ale nutné samostatně odsouhlasit, jelikož webové aplikace běžící v prohlížeči většinou nepovolují práci s tiskárnou přímo - je nutné použit uživatelské rozhraní prohlížeče. Serverová část systému nadále poskytuje zabezpečené REST API pro komunikaci s Android aplikací pro číšníky.

Android aplikace je pouze tenkým klientem - tedy její funkčnost zcela závisí na komunikaci se severem, bez něj neposkytuje vůbec žádnou funkcionalitu.

2.1.1 Analýza serverové části

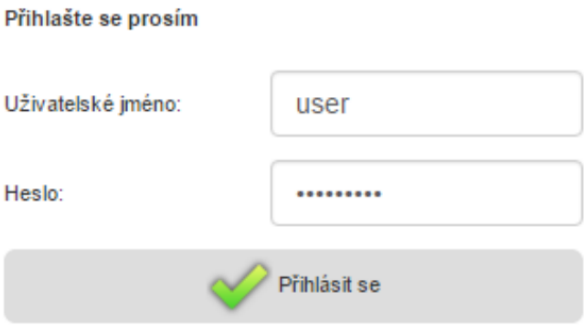
Funkcionalita aplikace Cashbob

Následuje představení funkcionality aplikace. Uživatelské rozhraní je přístupně pomocí prohlížeče, žádné jiné uživatelské rozhraní dostupné není. Domovská obrazovka na obrázku 2.1

Zabezpečení. Aplikace samozřejmě pro funkci vyžaduje autentizaci i autorizaci uživatele. Verze aplikace pana Červenky toto částečně splňuje, je možné se přihlásit jako vlastním podniku nebo číšník, viz obrázek 2.2. Bohužel jsem

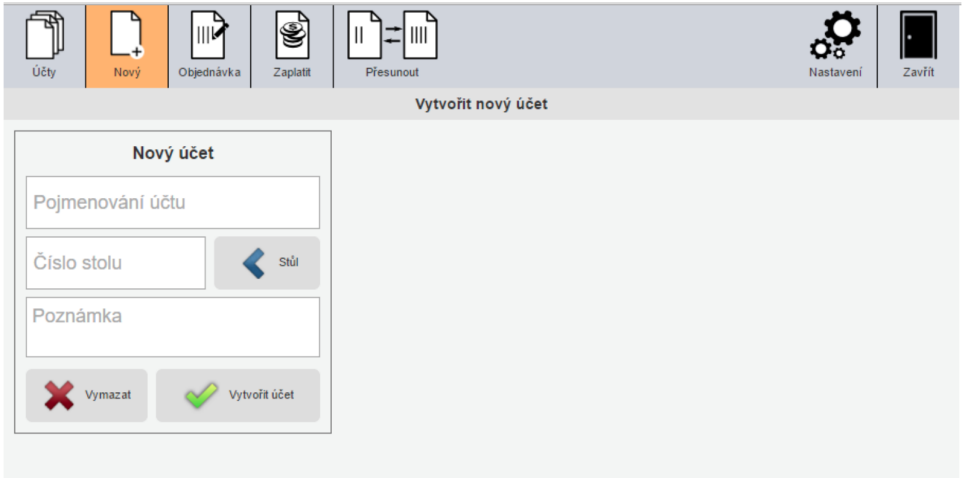
Obrázek 2.1: Domovská obrazovka původní aplikace Cashbob

ale v práci nenalezl způsob, jak se registrovat - tato funkcionalita ve webovém rozhraní chybí, přesto že na domovské obrazovce pro to tlačítko existuje, ale nic nedělá.

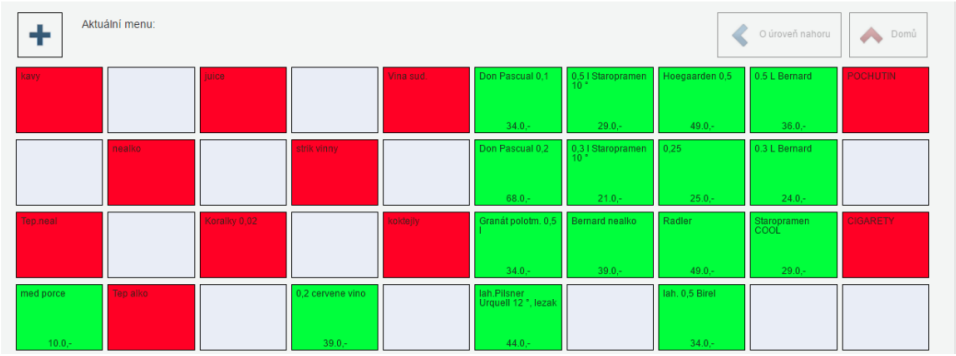


Obrázek 2.2: Přihlášení do aplikace

Účty. Základní jednotkou aplikace jsou tzv. účty, které imitují fyzické účty běžné v restauračních zařízeních. Každému účtu je možné přiřadit stůl (obrázek 2.3), čímž se automaticky vygeneruje i jméno daného účtu, pro jednodušší dohledání za chodu. toto jméno lze upravit, v případech kdy by existovaly dva souběžné účty u jednoho stolu. Každému účtu lze pak přiřazovat objednávky hostů - k tomu slouží rozhraní v podobě mřížky obsahující sortiment podniku (Obrázek 2.4). Tento sortiment lze libovolně rozmístit (při přihlášení jako vlastník podniku). Nezaplacené položky lze mezi účty přesouvat.



Obrázek 2.3: Tvorba nového účtu



Obrázek 2.4: Menu

Placení a stornování. Z každého účtu je možné zaplatit vybrané položky, nebo vše najednou. Platba je poté uložena a v případě existence výchozí tiskárny je vytištěna účtenka. Je možné i stornovat položky, nebo zrušit celý účet bez placení (to je možné i v případě role číšník).

Další správa. Implementace Tomáše Červenky pozbývá většinu další funkcionality, která by se dala od aplikace pro restaurační zařízení očekávat, jako například účetnictví, nebo alespoň zobrazení zaplacených objednávek, či uzavřených účtů. Tisk účtenek je implementován, ale jak již bylo zmíněno, není obsažen v rámci prohlížeče, ale na straně serveru - tiskne se na výchozí tiskárně, výstup účtenky je ve formátu PDF s nastavenou šířkou stránky pro tisk na účtenkových tiskárnách.

Nastavení aplikace se omezuje na tvorbu stolů.

REST rozhraní. Aplikace k dálkovému ovládání všech funkcionalit a ke komunikaci používá rozhraní REST (Representational state transfer) - jedná se o architekturu, která staví na HTTP volání GET a POST a přidává

ještě DELETE a PUT, tedy kompletní CRUD model. Přenášená data jsou reprezentována pomocí zdrojů, kde každý má svou HTTP adresu (URI)[res]. Obsahem těla HTTP volání jsou pak tzv. JSON (JavaScript Object Notation) objekty, které reprezentují přenášená data.

REST	CRUD	Popis
POST	Create	Vytvoření nového zdroje
PUT	Update	Úprava existujícího zdroje
GET	Retrieve	Získání zdroje
DELETE	Delete	Smazání zdroje

Tabulka 2.1: Popis REST metod[Zá16]

Závěr analýzy uživatelského rozhraní a funkcionality. Serverová aplikace umožňuje základní funkce pokladny, tedy tvorba účtů, objednávky, placení, stornování, editace položek menu, tisk účtenky. Už ale neumožňuje nastavení parametrů podniku, jako je i jen jméno podniku. Neobsahuje žádnou správu účetnictví, bez přístupu k REST API přímo se není možné ani registrovat nebo měnit pravomoce účtů.

2.1.2 REST rozhraní

Následuje výčet a popis jednotlivých REST zdrojů:

Manipulace účtů

```
{
    "uri":"http://localhost:9000/rest/account/1",
    "table":null,
    "id":1,
    "name":"First account",
    "createdate":1325372400000,
    "note":null,
    "user":null,
    "category":null,
    "opened":false,
    "orders":[
        {
            "uri":null,
```

Obrázek 2.5: JSON účtu

Zdroj: GET rest/account/
Vrátí všechny otevřené účty

Zdroj: GET rest/account/id
Vrátí informace o účtu specifikovaného id. JSON objekt již obsahuje i seznam otevřených objednávek.

Zdroj: POST rest/account/
Vytvoří nový účet

Zdroj: POST rest/account/id/order
Přebírá seznam objednávek a přidá jej na účet ve stavu přijato/nezaplaceno

Zdroj: PUT rest/account/id
Změní specifikovaná pole účtu, nemění seznam objednávek, ty proto mohou být v JSON objektu prázdné

Zdroj: PUT rest/account/id/payItems
Přebírá seznam objednávek k zaplacení a tyto objednávky a archivuje na straně serveru a vytiskne účtenku. Jednotlivé objednávky jsou koncipované jako jakýsi JSON "obal"pro jednotlivé položky (menuItem) a navíc obsahuje ještě celkovou cenu objednávky a datum.

Zdroj: PUT rest/account/id/moveItems/id2
Přesune nezaplacené objednávky z jednoho účtu do druhého

Adresa: DELETE Rest/Account/id
Smaže účet.

Manipulace menu

Adresa: GET Rest/menuItem/
Vrátí seznam všech aktivních položek menu.

Adresa: GET Rest/menuItem/id
Vrátí informace o položce specifikovaného id.

Adresa: POST Rest/menuItem/
Vytvoří novou položku.

Adresa: PUT Rest/menuItem/id
Upraví položku. Jelikož se archív objednávek na tyto položky odkazuje, neměl by nikdy být měněna celá položka, pouze provedeny mírné úpravy. Každá objednávka má proto specifikovanou vlastní cenu, aby nebylo ovlivněno účetnictví podniku.

Adresa: DELETE Rest/Item/id
Smaže položku

```
{
            "isMenu":true,
            "name":"Jídelní lístek",
            "nameShort":null,
            "price":null,
            "quantity":null,
            "parentMenu":null

        },
        {
            "uri":"http://localhost:9000/custmenunode/60",
            "custMenuNodeId":60,
            "itemId":25,
            "isMenu":false,
            "name":"Kofola",
            "nameShort":null,
            "price":36.0,
            "quantity":"1",
            "parentMenu":{
                "uri":null,
                "menuid":2,
                "name":"nealko",
                "date":1351852894000,
                "isCustMenu":false
            }
        }
    ]
}
```


Obrázek 2.6: JSON pole položek menu


```
{
    "uri":"http://localhost:9000/rest/table/2",
    "id":2,
    "tablenumber":16,
    "numberofplaces":4
}
```


Obrázek 2.7: JSON stolu

Manipulace stolů

Zdroj: GET rest/table/
Vrátí všechny stoly

Zdroj: GET rest/table/id
Vrátí informace o stolu specifikovaného id.

Zdroj: POST rest/table/
Vytvoří nový stůl

Zdroj: PUT rest/table/id

Změní specifikovaná pole stolu

Adresa: DELETE Rest/table/id
Smaže stůl

Manipulace uživatelských účtů


```
{
  "uri":"http://localhost:9000/rest/user",
  "users":[
    {
        "uri":"http://localhost:9000/rest/user/admin",
        "id":1,
        "version":1,
        "firstName":"Admin",
        "lastName":"predefined",
        "username":"admin",
        "password":null,
        "personalId":"1",
        "credit":0.0,
        "role":null,
        "updatePassword":null
    }
  ]
}
```


Obrázek 2.8: JSON uživatelských účtů

Zdroj: GET rest/user
Vrátí všechny stoly

Zdroj: GET rest/user/id
Vrátí informace o uživateli specifikovaného id.

Zdroj: GET rest/user/username
Vrátí informace o uživateli specifikovaného uživatelského jména.

Zdroj: GET rest/user/id/transactions
Vráti transakce uzavřené daným uživatelem

Zdroj: GET rest/user/id
Vrátí účty vytvořené daným uživatelem

Zdroj: POST /rest/user
Registruje uživatele

Zdroj: PUT rest/user/id/password/oldPassword
Smění uživateli heslo

Ostatní zdroje

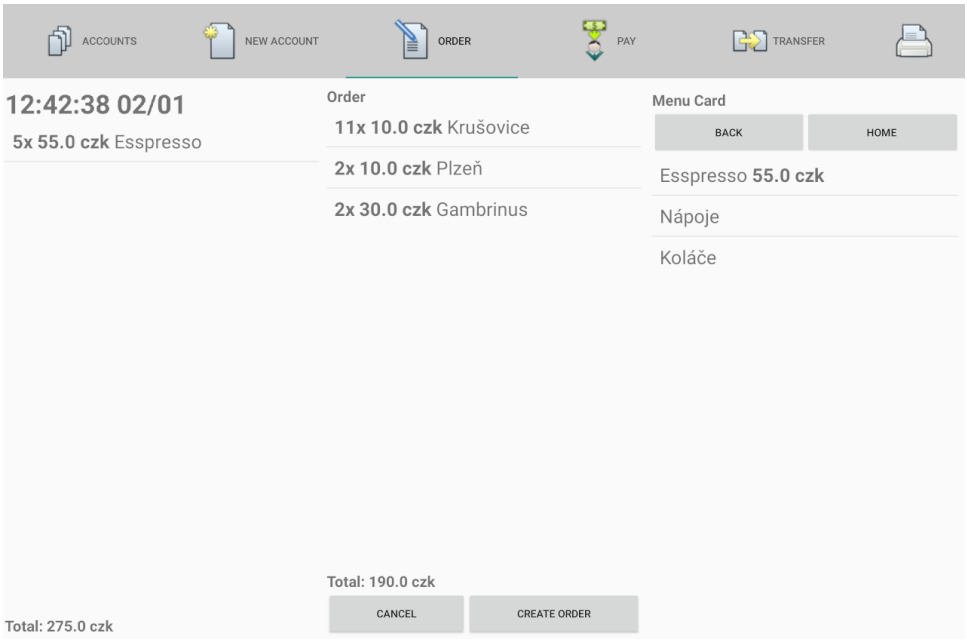
Aplikace technicky dále obsahuje i další REST zdroje, ty ale nejsou žádným způsobem využity, a to ani registrace, jak již bylo zmíněno v předchozích odstavcích. Dále se tedy budu zabývat pouze funkcionalitou, kterou aplikace opravdu uživateli zpřístupňuje.

2.1.3 Analýza Android aplikace

Aplikaci původně psanou Tomášem Hogenauerem jsem již analyzoval[Zá16], včetně testů funkčnosti rozhraní REST, ve své bakalářské práci. Pro úplnost zde tedy shrnu výsledky této původní analýzy. Ve své bakalářské práci jsem také již analyzoval i platformu Android, pro kterou je aplikace určena - budu se tedy na ni případně odkazovat.

Určení

Jedná se o přenosnou aplikaci pro číšníky, která slouží jako tenký klient. Na tomto klientovi je možné vytvořit nový zákaznický účet, přiřadit mu číslo stolu, přidat objednávky (viz obrázek 2.9), zaplatit objednávky, uzavřít účet, či přesouvat objednávky mezi účty. Klient je bezstavový a při každé změně aktivity se data načítají znovu přes rozhraní REST ze serveru.



Obrázek 2.9: Aplikace pro Android

2.2 Analýza vhodnosti přesunu na ASP.NET Core

ASP.NET Core (dále jen framework Core) je multiplatformní open-source framework který kombinuje ASP.NET MVC (pro webové aplikace) a ASP.NET Web API (REST). Nejedná se tedy pouze o programovací jazyk - toto je tedy největší rozdíl oproti čisté Javě, ve které je psána původní aplikace. ASP.NET Core totiž nespecifikuje ani použitý jazyk, aplikace je možné kromě obvyklého C# psát například i v C++, nicméně pro tento jazyk nejsou k dispozici žádné automatizované nástroje[cor].

Za účelem rozhodnutí o vhodnosti přesunu aplikace z původního prostředí, srovnám jednotlivé aspekty aplikace dané jejím programovacím jazykem, frameworkem, knihovnami a konkrétní implementací s tím, jak by tyto aspekty mohly být implementovány v novém prostředí frameworku Core. Závěrem poté zhodnotím tyto zlepšení (nebo pohoršení), zda-li jsou dostatečně pozitivní a umožní jednodušší vývoj a rozšířitelnost aplikace v budoucnu a zrychlení prototypování. Konkrétně se jedná o tyto aspekty:

- Jednotnost programovacích jazyků mezi komponentami aplikace 2.2.1
- Podpora ze strany knihoven a frameworků
  - Perzistence dat 2.2.2
  - Implementace autentizace a autorizace 2.2.3
  - Implementace uživatelského rozhraní 2.2.4
  - Ošetření vstupů 2.2.5
  - Přímá podpora REST API 2.2.6
  - Přímá podpora technologií pro implementaci komunikace se serverem finanční správy 2.2.7
- Náklady spojené s vývojem 2.2.8
- Nasaditelnost 2.2.9

2.2.1 Jednotnost programovacích jazyků mezi komponentami aplikace

Původní aplikace je psána v jazyce Java 8 s použitím Frameworku Play! ve verzi 2.4. Webové rozhraní aplikace je ale psáno kombinací vložených tagů frameworku Play"a také JavaScriptu. Ten je využit vždy, když je třeba asynchronně získávat data, nebo editovat proměnné na straně klienta. Je tedy nutné pro programování UI a backendu použít dva programovací jazyky.

Nutnost použití JavaScriptu pro inicializaci proměnných:

```
$(document).ready(function() {
            $('#order').addClass("top-row-icon-active")
        });

        var account = @Html(Json.stringify(Json.toJson(account)));
        var menu = @Html(Json.stringify(Json.toJson(menuList)));
```

Proměnné stránky se inizializují jednoduše pomocí bloku code:

```
@code {
    private Account[] accounts;

    protected override async Task OnInitializedAsync() {
        accounts = await Http.GetFromJsonAsync<Account[]>("rest/account"
    );
    }

}
```

Řešení dle frameworku Core

Framework Core má na tento problém odpověď v podobě na něm stavějícím frameworku Blazor, konkrétně v podobě klientské aplikace využívající WebAssembly, což znamená, že při prvním spuštění aplikace se pomocí jediného malého JavaScriptu stáhne mikrojádro ASP.NET Core, které pak běží v prohlížeči a komunikuje se serverovou aplikací pomocí REST rozhraní. To dále znamená, že je možné využít sdílenou kódovou základnu mezi oběma moduly klient/server, což velmi usnadní vývoj uživatelského rozhraní. Kód v C# se jednoduše vloží mezi HTML tagy a sám je může i generovat.

Blazor pak sám podporuje tzv. Razor komponenty, které je možné importovat do projektu pomocí správce balíku ve Visual Studiu, a ještě tak významně snížit nutnost použití HTML.

Nakonec pak toto řešení také umožňuje použití takovéto klientské aplikace i offline, kde může akumulovat jednotlivé požadavky na server a odeslat je po obnovení internetového připojení.

2.2.2 Perzistence dat

Data jsou v původní aplikaci do databáze ukládána pomocí frameworku Hibernate, pomocí abstrakce Spring JPA. Databázový objekt, DAO, tedy zajišťuje ukládání a dotazování databáze, například takto:

```
Query q = JPA.em().createNamedQuery(this.getNamedQuery("findOpenedByName
    "));
```

kde NamedQuery, tedy pojmenovaný dotaz vypadá například takto:

```
@NamedQuery(name = "Account.findOpenedByName", query = "SELECT a FROM
    Account a where a.opened = true and UPPER(a.name) = UPPER(:name)"),
```

Dotaz je tedy nutné specifikovat pomocí standardního SQL dotazu, což může být problém kvůli útoku pomocí SQL injekce (o tom dále v následujících odstavcích). Core na rozdíl od toho využívá ve svém EntityFrameworku tzv. LinQ dotazů, které simulují jazyk SQL, ale místo dotazování se nad tabulkami se odkazujeme pomocí tzv. lambda výrazů přímo pomocí atributů entit - toto zjednodušuje přístup k datům, jelikož není nutné rozlišovat jejich původ, jak jsou uložena (a odpadá nutnost používat DAO nebo DTO objekty, pracuje se přímo s entitami)[lin]:

```
accountItem.Orders = accountItem.Orders.OrderBy(o => o.Item.Id).ThenBy(o
    => o.IsPaid).ToList();
```

Definice entit

Atributy entit musí být v původní implementaci popsány explicitně, například:

```
@Id
@GeneratedValue(strategy = IDENTITY)
@Column(name = "ACCOUNTID", unique = true, nullable = false)
private Long accountid;
```

kdežto v Core jednoduše stačí:

```
public long Id { get; set; }
```

a EntityFramework automaticky vygeneruje schema s unikátním klíčem Id. Samozřejmě je možné také použít anotace k upřesnění, ale nejsou nutné.

2.2.3 Implementace autentizace a autorizace

Autentizace je v původní aplikaci implementována pouze za použití tzv. Basic autentizace, která spočívá v odesílání uživatelského jména a hesla v Base64 formátu s každým REST voláním. Toto je základní chování použitého frameworku a bylo by tedy pracné toto změnit. ASP.NET Core místo toho k autentizaci implicitně používá tzv JWT bearer tokeny, kde uživatelské jméno a heslo jsou na server zaslány pouze jednou a poté je vygenerován Bearer token, který je šifrovaný privátním klíčem na straně serveru a mlže obsahovat více identifikátorů, například IP adresu připojujícího se uživatele, aby nebylo možné "ukrást"sezení, pokud by se útočník k tomuto tokenu dostal. Délku sezení (platnost tokenu) je možné nastavit. Klient poté při komunikaci do hlavičky REST volání vloží tento šifrovaný token, který jméno a heslo neobsahuje. Stejným způsobem mimo jiné funguje implicitně i přihlašování a sezení klientské webové aplikace Blazor, není tedy nutné psát autentizaci pro REST API a webového klienta zvlášť.

Co se autorizace týče, ta je v obou frameworcích řešena obdobně, tedy pomocí anotací. Zde je dobré zmínit, že implementace Tomáše Červenky má kód pro autorizaci zakomentovaný a momentálně tedy nefunkční, všechny účty mají stejné pravomoce.

2.2.4 Implementace uživatelského rozhraní

Oba frameworky řeší implementaci uživatelské rozhraní (tedy dynamické generování webových stránek) podobným způsobem, Core Blazor tedy ještě k tomu přidává podporu nativního (C#) kódu na straně klienta za užití WebAssembly.

Navíc, jelikož klient technicky neběží na stejném zařízení jako server, je možné pro získávání a odesílání dat prostě použít opět rozhraní REST, místo běžného MVC server modelu, který implementuje původní Cashbob aplikace.

2.2.5 Ošetření vstupů

Ověření platnosti atributů

Jedná se o ověření dodržování datových typů, např. zabránění uživateli zadat řetězec do pole, kde má být číslo. Na rozdíl od původní implementace, tato funkcionalita je v Core implicitní - pokud je datový typ Entity int, zadání řetězce neprojde validací. Java implementace k tomuto potřebuje anotace. Řešení validace v Core vyžaduje pouze:

```
<DataAnnotationsValidator />
```

v HTML kódu uvnitř form, při zavolání REST API se potom na straně klienta automaticky objeví chybová hláška.

SQL Injection

Jedná se o vkládání dodatečných SQL dotazů do vstupních polí formulářů, které pak neošetřený kód na serveru odešle do databáze, což může mít závažné následky. V původní aplikaci je toto ošetřeno vkládání pomocí metody setParameter, která automaticky vstup ošetřuje:

```
q.setParameter("from", from);
```

V Core je tomu obdobně, jen není potřeba nastavovat parametry, díky použití LinQ místo SQL se parametry můžou vložit přímo do dotazu, viz 2.2.2.

2.2.6 Přímá podpora REST API

Jelikož přímá podpora REST rozhraní není v Java ani v Spring obsažena, je v původní aplikaci použit framework Play!, který implementuje funkcionalitu komunikace pomocí protokolu HTTP. Ne tedy přímo REST, ale pouze dokáže zpracovávat HTTP zprávy. Serializace a de-serializace, URI dekódování a volání správných metod je psáno zcela manuálně. ASP.NET Core API naproti tomu již má plnou podporu REST, není nutné nastavovat vůbec nic, stačí vytvořit Controller třídu pro každý REST zdroj a jednotlivé metody jsou pak automaticky dekódovány dle jména metody v kódu, nebo upřesněny pomocí anotací:

2.2.7 Přímá podpora technologií pro implementaci komunikace se serverem finanční správy

V roce 2016 vstoupila v platnost nutnost živnostníků odesílat po každé transakci data o ní na server finanční správy. Jedná se především o celou tržbu v korunách včetně základu daně a daně samotné, DIČ obchodníka,

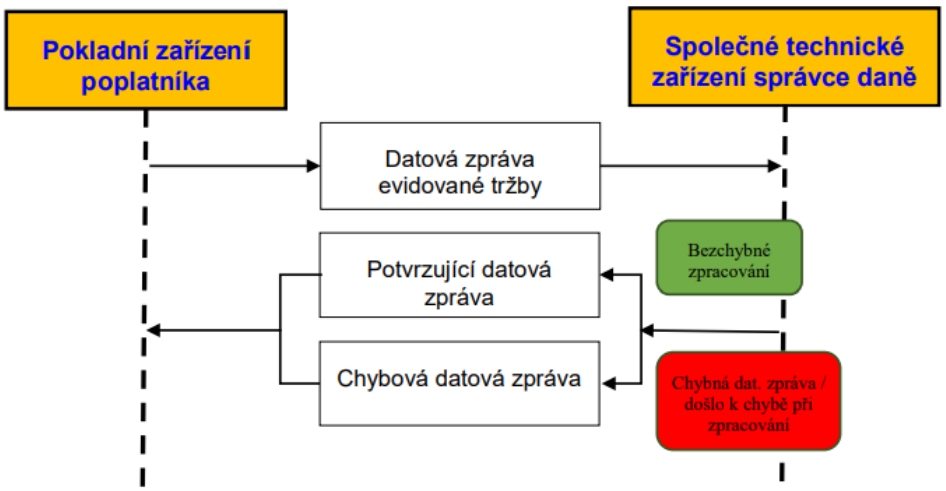
jméno a číslo podniku a podpis pomocí certifikátu. Jméno, číslo a certifikát vydává Finanční správa živnostníkovi na požádání na svém portálu. Všechny 3 údaje jsou povinnou součástí zmíněné zprávy a ta je bez nich neplatná [eeta].

Zpráva se na server finanční správy posílá pomocí HTTP vrstvy podobně jako tomu je tomu u REST API, REST se zde ale nepoužívá, místo něj byly zvoleny tzv. SOAP zprávy, což je vlastně XML dokument zasílaný v těle HTTP, tedy žádné další informace v hlavičce, nepoužívají se ani GET a POST metody.

Podepisování SOAP objektu probíhá pomocí privátního klíče již zmíněného certifikátu obchodníka. Podepisování probíhá dle standardu WS-Security. Dále je nutné vypočítat dva kontrolní součty tak, jak je stanoveno Vyhláškou č. 269/2016 Sb. [bkp]

Pokud je zpráva dobře formovaná a obsahuje vše potřebné, odešle server EET nazpět tzv. FIK, tedy fiskální identifikační kód. Ten je nutné, spolu s BKP a dalšími povinnými údaji umístit na účtenku [uct]. V takovém případě se jedná o tzv. Běžný režim. V případě, že nefunguje komunikace se serverem EET, může být účtenka vystavena v tzv. zjednodušeném režimu, kdy na ní místo FIK bude uveden kód PKP. Prodejce je pak povinen údaje o transakci na server EET odeslat dodatečně po obnově spojení. Z toho všeho plyne, že je nutné ukládat všechny tyto údaje do databáze, aby bylo možné údaje později odeslat [eeta].

Jedná se tedy o velmi komplexní proceduru. Bohužel, ASP.NET Core neobsahuje téměř žádnou funkcionalitu, která by ji usnadnila. V původním .NET stále ještě nástroje pro práci se SOAP existují, v Core již ale byly odstraněny, jelikož tento framework počítá pouze s komunikací pomocí REST API. Naštěstí ale bylo EET v platnosti již dostatečně dlouhou dobu na to, aby se objevily dostatečně robustní knihovny i pro .NET, které komunikaci automatizují.



Obrázek 2.10: Schema komunikace se serverem finanční správy[eeta]

2.2.8 Náklady spojené s vývojem

Původní aplikace je psaná v Javě, která je zcela zdarma, včetně běžných vývojových nástrojů.

ASP.NET Core

ASP.NET Core framework je také open-source, multiplatformní a zdarma. Pokročilé vývojové nástroje sady Visual Studio Community je také možné používat zdarma i za účelem vývoje zpoplatněné aplikace, pokud bude daná aplikace vyvíjena maximálně pěti fyzickými osobami v týmu. Pokud by tým byl větší než 5 osob, musely by být nástroje zpoplatněny [vst]. Visual Studio Community ovšem není jediným nástrojem pro vývoj ASP.NET Core, aplikaci je možné vyvíjen i pomocí nástroje Visual Studio Code, který je zcela zdarma a open-source bez jakýchkoliv limitací. Z pohledu nákladů na vývoj tedy ASP.NET Core není oproti Javě znevýhodněno.

2.2.9 Nasaditelnost

Jako stand-alone aplikace

Původní aplikaci je možné přeložit do standardního JAR balíku, který ke spuštění potřebuje pouze nainstalovanou Javu. ASP.NET aplikace můžou být přeloženy podobným způsobem, při kterém pak k chodu vyžadují nainstalovaný balík .NET knihoven požadované verze (v případě aplikace, která bude výstupem této práce by se jednalo o .NET Core 3.1)

Jako PaaS služba

PaaS[paa] v Cloud computing znamená "platform as a service", tedy není k dispozici celý operační systém, ale pouze runtime v podobě potřebných frameworků (jako Java nebo ASP.NET), na které je nutné aplikaci nahrát.

2.3 Závěr

Analýza rozdílů mezi frameworky Java/Spring/Play! a ASP.NET Core/Blazor odhalila, že přesto, že funkční rozdíly mezi nimi jsou minimální a je možné implementovat veškerou potřebnou funkcionalitu v obou, jednoduchost implementace jednoznačně vítězí u ASP.NET Core. Ať už je to implementace databázového modelu 2.2.2, implicitní zabezpečení API 2.2.3, ošetření vstupů 2.2.5, či implementace samotného rozhraní, na kterém aplikace staví svou komunikaci, ASP.NET Core umožňuje mnohem rychlejší vývoj, především implementace změn do datového modelu je velmi triviální, Entitiy manager tohoto frameworku nespoléhá na anotace nebo konfigurační soubory, vystačí si se samotnou strukturou psaného kódu.

Jelikož je v plánu další rozšíření aplikace, především tedy implementace EET a nasaditelnost na vzdáleném runtime prostředí (Cloud typu PaaS[paa]),

rozhodl jsem se co nejobjektivněji s přihlédnutím ke zmíněním aspektům, přenést aplikaci na ASP.NET Core.

Kapitola 3

Návrh

Aby bylo možné začít navrhovat strukturu modelů a REST URI aplikace, je nutné nejdříve stanovit funkční a nefunkční požadavky na nový server. U požadavků budu vycházet primárně z funkcionality, kterou již původní Cashbob obsahuje a navíc se zaměřím také na rešerši již existujících pokladních systémů s podporou přenosných zařízení/tiskáren a EET(Vzhledem k naprosté nutnosti podporovat EET, toto splňují prakticky všechny).

Vzhledem k tomu, že jsem začal na projektu pracovat bez jakýchkoliv znalostí platformy .NET, složitosti jednotlivých požadavků jsem nemohl určit, jelikož jsem nevěděl, kolik času a úsilí bude třeba k získání potřebných znalostí – vše jsem se musel nejdříve prostudovat v uživatelské dokumentaci. Dokonce i samotný programovací jazyk C# byl pro mě úplnou novinkou. Jak jsem již zmínil v úvodu 1.3, součástí každé iterace bude i popis nalezení řešení dané problematiky.

Aplikace bude koncipována jako cloudové řešení - tedy uživatel nebude mít z principu aplikaci nainstalovanou na svém zařízení, ale bude k ní přistupovat přes webové, nebo REST rozhraní - to by bylo realizováno pomocí dodávané android aplikace pro číšníky, která obsahuje veškerou funkcionalitu pro chod podniku. Úskalí tohoto řešení je závislost na internetovém připojení. Jelikož ale samotný systém EET internetové připojení stejně vyžaduje[1], nevidím to osobně jako překážku.

3.1 Rešerše existujících systémů

Existuje spousta systémů[pok], většina pouze jako lokální aplikace pro přenosné zařízení, některé nabízené i jako celkové zázemí s centrální správou, tedy podobě jako je tomu u systém Cashbob. Jednotlivé systémy s podporou centrální správy tedy porovnám a závěrem shrnu funkcionalitu, kterou subjektivně, z hlediska konkurenceschopnosti, budu muset ještě introdukovat.

3.1.1 Hledání aplikací s podporou Android

Aplikace jsem nalezl pomocí obchodu s aplikacemi Google Play, hledáním výrazu "EET pokladna"a obsažením těch, které podporují komunikaci se

[1] Existuje i tzv. offline režim[eetb]

serverem:

Aplikace	Dotykačka	MiniPOS	Kasa FIK
Zpoplatnění	Ano	Ano	Ano
Webová aplikace	Ano	Ano	Ano
Android aplikace	Ano	Ano	Ano
iOS aplikace	Ne	Ne	Ne
Desktop aplikace	Ne	Ne	Ne
EET běžný režim	Ano	Ano	Ano
EET zjednodušený	Ano	Ano	Ano
Historie tržeb	Ano	Ano	Ano
Správa zásob	Ano	Ano	Ano
Mapa stolů	Ano	Ano	Ano

Tabulka 3.1: Funkce aplikací s centrální správou

3.2 Závěr rešerše

Při srovnání aplikací vyšly najevo některé nedostatky, nebo spíše chybějící funkcionalita:

3.2.1 Mapa stolů

Webová, ani Android aplikace nepodporují rozestavění stolů do mapy, ze které si číšník může vybrat. Tato funkcionalita jednoznačně ulehčuje práci a bylo by dobré jí v Android aplikaci implementovat

3.2.2 Platby kartou

V roce 2020 již většina Čechů pravidelně platí kartou[pla]. Jakýkoliv pokladní systém by ji tedy měl podporovat. Umožnění placení kartou ale není součástí zadání a nebude se tedy prozatím implementovat, scope této práce je především přesun stávající funkcionality na platformu ASP.NET Core.

3.2.3 Stav mobilní aplikace

Při srovnání uživatelského rozhraní aplikací nabízených na Google Play a Android aplikace Cashbob, je zjevný alespoň jeden hlavní nedostatek, a tedy nemožnost zobrazení menu v mřížce. Původní Cashbob aplikace toto ve webovém rozhraní splňuje. Jelikož se v této práci zaměřím na přesun serverové části a pouze zajištění funkčnosti s Android aplikací, nebudu uživatelské rozhraní této aplikace měnit, nebude to tedy součástí návrhu.

3.2.4 Administrace

Jak jsem již v analýze zmínil, aplikace původní aplikace Cashbob umožňuje administrace podniku pouze pomocí volání REST, ne již užitím uživatelského

rozhraní.

3.2.5 Vliv zjištění na návrh

Tuto funkcionalitu určitě zahrnu při tvorbě funkčních/nefunkčních požadavků, implementačně ale dostanou jen nízkou prioritu, jelikož tyto funkce původní aplikace neobsahuje a prioritou je, jak již bylo zmíněno, aplikaci hlavně přesunout na platformu Core.

3.3 Funkční požadavky

RQ 1 – Perzistence

Zadavatel Pavel Zářecký
Priorita Vysoká

Aplikace bude automaticky ukládat veškeré potvrzené změny provedené v datech pomocí REST rozhraní, protože data nesmí být v nekonzistentním stavu (při výpadku proudu by došlo ke ztrátě pracovních dat).

RQ 1.1 – Ukládání všech objednávek

Zadavatel Pavel Zářecký
Priorita Vysoká

Aplikace bude ukládat historii objednávek pod jednoznačným ID za účelem automatizovaného účetnictví a pro potřeby EET (např. tzv. offline režim). Původní aplikace správu účetnictví podporuje pouze pomocí REST API, ne však v uživatelském rozhraní. Ukládá ale zaplacené transakce, nová aplikace toto bude podporovat též.

RQ 2 – Správa sortimentu

Zadavatel Pavel Zářecký
Priorita Vysoká

Aplikace bude umožňovat správu sortimentu, jelikož se jedná o nedílnou součást aplikací podobného zaměření, jak odhalila rešerše a také proto, že původní aplikace toto obsahuje.

RQ 2.1 – Správa položek menu

Zadavatel Pavel Zářecký
Priorita Vysoká

Systém bude umožňovat výběr aktuální nabídky a její kategorizaci do menu, aby byl umožněn co nejrychlejší výběr číšníkem za provozu podniku.

RQ 2.2 – Správa skladu

Zadavatel Pavel Zářecký
Priorita Nízká

Původní aplikace umožňuje velmi omezenou správu skladových zásob, modul skladu je ale naprosto oddělený od pokladního systému a tedy dle mého názoru pozbývá smyslu, v budoucnosti by bylo dobré implementovat správu skladu vztaženou k prodávanému sortimentu, nicméně tento požadavek má jen nízkou prioritu.

RQ 3 – Manipulace objednávek a účtů

Zadavatel Pavel Zářecký
Priorita Vysoká

Aplikace bude umožňovat vytváření nových objednávek, jelikož se jedná o nedílnou součást aplikací podobného zaměření.

RQ 3.1 – Platba zboží

Zadavatel Pavel Zářecký
Priorita Vysoká

Aplikace bude automaticky ukládat veškeré potvrzené změny provedené v datech pomocí REST rozhraní, protože data nesmí být v nekonzistentním stavu (při výpadku proudu by došlo ke ztrátě pracovních dat)

RQ 3.2 – Jednoznačné určení objednávek za provozu

Zadavatel Pavel Zářecký
Priorita Vysoká

Aplikace bude umožňovat číšníkovi označit objednávku tak, aby ji bylo možné za provozu jednoznačně určit (jméno, číslo stolu, ...).

RQ 3.3 – Přesun objednávek mezi účty

Zadavatel Pavel Zářecký
Priorita Vysoká

Bude možné přesouvat nezaplacené objednávky mezi aktivními účty, protože zákazník může chtít zaplatit část účtu samostatně.

RQ 4 – Granularita oprávnění

Zadavatel Pavel Zářecký
Priorita Vysoká

Aplikace bude rozlišovat několik úrovní oprávnění, aby číšníci nemohli měnit citlivé informace.

RQ 5 – Souběžnost přihlášených uživatelských účtů

Zadavatel Pavel Zářecký
Priorita Vysoká

Aplikace bude muset podporovat spoustu přihlášených uživatelských účtů, protože se bude jednat o centralizované cloud řešení nabízené více zákazníkům výsledného produktu.

RQ 6 – Offline provoz

Zadavatel Pavel Zářecký
Priorita Střední

Jelikož bude mít zákazník k dispozici jediné hardwarové řešení - tedy svůj tablet, který provozuje pouze tenký klient, mlže se stát, že vypadne internetové připojení. Aby byla zachována funkce pokladního zařízení, bude tenký klient (jak webová aplikace, tak Android klient) obsahovat schopnost ukládat REST požadavky na zásobník a později je zpracovat po obnovení spojení.

RQ 7 – Evidence tržeb

Zadavatel Martin Komárek
Priorita Vysoká

Systém bude podporovat elektronickou evidenci tržeb, jelikož restaurační zařízení dle zákona [eetc] musí zajistit. nutnou součástí EET v základním režimu je také tisk účtenek.

3.4 Nefunkční požadavky

RQ 8 – Platforma ASP.NET

Zadavatel Martin Komárek

Aplikace bude implementována na platformě ASP.NET Core v jazyce C#, jelikož užití tohoto frameworku vyplynulo z analýzy.

RQ 9 – Rychlost operací

Zadavatel Pavel Zářecký

Aplikace musí být optimalizovaná pro rychlost, aby nedocházelo ke zdržování zákazníku pomalými reakcemi klientské aplikace

3.5 Závěrem

Jednotlivé požadavky budou dle zadání průběžně implementovány, testovány a nasazovány iterativně, součástí každé iterace tedy bude seznam požadavků, které byly splněny, včetně technických detailů jakým způsobem toho bylo dosaženo. V závěru zprávy poté shrnu do jaké míry byly požadavky splněny.

Kapitola 4

Iterace vývoje

Tato kapitola bude obsahovat popis jednotlivých iterací vývoje. Součástí každé z těchto iterací je Návrh nově implementované funkcionality, popis samotné implementace, testování a závěr, který může vést k zohlednění nových skutečností v následných iteracích.

4.1 Aplikační vrstva

Nejdříve je nutné navrhnout aplikační vrstvu. To znamená funkční základ, který bude komunikovat s Android aplikací pomocí rozhraní REST. Iterace se bude zabývat implementací požadavky RQ1, RQ1.1, RQ2 a RQ2.1.

4.1.1 Návrh

V původní aplikaci jednotlivé JSON objekty, posílané a přijímané pomocí REST, velmi těsně kopírují strukturu aplikačních entit. Navrhl jsem tedy obdobné vazby mezi entitami, viz 2.1.2. Kompletní sada entit je vidět na obrázku 4.1

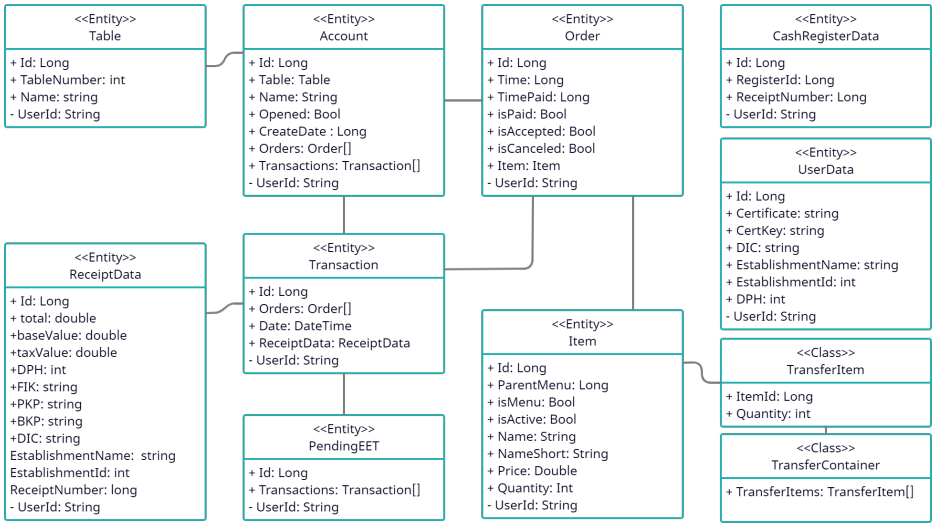
Každá entita, kterou může uživatel zprostředkovat pomocí REST API obsahuje i jedinečný identifikátor uživatele, uživatele, jelikož aplikace bude uchovávat data více podniků najednou. Pole UserId není v JSON objektech obsažené, ani není možné jej dodat pomocí metod POST/PUT.

ReceiptData. Obsahuje veškeré informace nutné k pozdějšímu odeslání EET dat. Jednotlivé položky byly určeny dle specifikace.[uct]

Transaction. Archivace transakcí. K informacím z účtenky přidává ještě seznam objednávek a datum uskutečnění transakce.

CashRegisterData. Dle specifikace EET je nutné evidovat i číslo účtenky, které musí být ze spojité logické řady, specifické pro každé zařízení. Do databáze se tedy budou ukládat data pro každou pokladnu (Android zařízení s nainstalovanou Cashbob aplikací pro číšníky) zvlášť.

UserData. Obsahuje data o podniku, včetně dat pro automatizaci komunikace EET (Certifikát a klíč).



Obrázek 4.1: Entity a pomocné třídy aplikace

4.1.2 Implementace

Entity aplikační vrstvy

Jak jsem již nastínil v Analýze, entity v ASP.Net Core stačí implementovat jako jednoduché třídy s veřejnými atributy. Například implementace entity UserData:

```
public class UserData {
        public long Id { get; set; }
        [JsonIgnore]
        public string UserId { get; set; }    //each user has their own
    set of entities, given that the app is cloud based and multi-user,
    multi-customer
        public string Certificate { get; set; }
        public string CertKey { get; set; }
        [MinLength(10, ErrorMessage = "DIC must have at least 10
    characters, include the CZ prefix")]
        [MaxLength(12, ErrorMessage = "DIC must have at most 12
    characters, include the CZ prefix")]
        public string DIC { get; set; }
        public string EstablishmentName { get; set; }
        public int EstablishmentId { get; set; }
        [Range(1, 100, ErrorMessage = "Allowed {0} values are between
    {1} and {2}.")]
        public int DPH { get; set; }

    }
```

Anotace v hranatých závorkách zde slouží k ošetření vstupů. [JsonIgnore] pak atribut vynechá ze serializace JSON objektů, jak bylo zmíněno v předešlých odstavcích.

Perzistence

EntityFramework. V ASP.NET Core se pro práci s entitami používá Entity Framework Core[ef] (dále jen EF). Díky němu není nutné pracovat se žádnými databázovými objekty (DAO), které jsou běžné např. v Java Frameworku JPA/Hibernate. Stačí vytvořit vytvořit EF například takto:

```
namespace Cashbob_JWT_WASM.Server.Data
{
    public class ApplicationDbContext : IdentityDbContext
    {
        public ApplicationDbContext(DbContextOptions options) : base(
    options){
        }

        public DbSet<Account> Accounts { get; set; }
        public DbSet<Order> Orders { get; set; }
        public DbSet<Item> Items { get; set; }
        public DbSet<Table> Tables { get; set; }
        public DbSet<Menu> Menu { get; set; }
        public DbSet<UserData> UserData { get; set; }
        public DbSet<CashRegisterData> CashRegisterData { get; set; }
        public DbSet<PendingEET> PendingEET { get; set; }

    }
}
```

NuGet package manager. Tvorba schematu databáze a vytvoření tabulek je také automatické, stačí po každé změně entit zavolat následující příkaz v NuGet Package Manageru, který je součásti vývojových nástrojů Visual Studio všech verzí:

```
Add-Migration (Jmeno migrace)
Update-Database
```

příkaz vytvoří migraci, která automaticky převede databázi z předchozího schematu (migrace) do nového konzistentního stavu a provede patřičné změny.

Spojení s databází. Ve vstupním bodu programu stačí přidat databázový kontext mezi využité služby a vše ostatní je již zřízeno automaticky:

```
services.AddDbContext<ApplicationDbContext>(options =>
                    options.UseSqlServer(Configuration.
    GetConnectionString("DefaultConnection")));
```

DefaultConnection odkazuje do Konfiguračního souboru aplikace "appsetings.json", který je načten před startem aplikace a umožňuje explicitně konfigurovat aspekty programu, například definovat globální konfigurace. Pro funkčnost tento konfigurační soubor nutný není, jde pouze u usnadnění přístupu, aby nebylo nutné konfigurační proměnné psát přímo do kódu. Tento konfigurační soubor je také možné editovat po přeložení a nasazení aplikace na webový server a tedy změnit umístění databáze, přihlašovací jméno, nebo heslo.

Controllery REST API

Pro zřízení REST API Controlleru stačí vytvořit třídu, která dědí Controller třídu frameworku. Pomocí anotací pak je nastavena cesta URI. Jednotlivé metody pak stačí opět pomocí anotací ustanovit jako zdroje pro REST API, například Controller pro UserData:

```
[Authorize]
    [Route("rest/[controller]")]
    [ApiController]
    public class UserDataController : Controller {
        private readonly ApplicationDbContext _context;
        private readonly UserManager<IdentityUser> _userManager;
```

Poté v konstruktoru předáme parametr typu DatabaseContext, který slouží k abstraktizaci práce s databází

```
        public UserDataController(ApplicationDbContext context,
    UserManager<IdentityUser> userManager) {
            _context = context;
            _userManager = userManager;
        }
```

A konečně metoda sloužící k implementaci REST zdroje:

```
        [HttpGet]
        public async Task<ActionResult<UserData>> Get() {
            UserData userData = (await _context.UserData.Where(s => s.
    UserId == User.Identity.Name).ToListAsync()).FirstOrDefault();
        *** Zkraceno ***
                await _context.SaveChangesAsync();

            return userData;
        }
```

Zde je také vidět jednoduchost práce s dotazováním databáze.

Úprava Android aplikace

Vzhledem k tomu, že jsem entity nové aplikace navrhl tak, aby co nejvěrněji kopírovaly strukturu původní aplikace, velké zásahy do Android aplikace nebyly nutné. Několik změn ale muselo být provedeno. Například původní aplikace vždy, když vracela pole JSON objektů, obalila ho do dalšího objektu obsahujícího i URI zdroje. Jelikož je REST rozhraní původní aplikace ručně psané, toto nebylo problém implementovat. API ASP.NET Core aplikací je ale z větší části automaticky generované, upravovat tuto strukturu je sice možné, ale pozbývalo by to smyslu, jelikož jeden z hlavních důvodů přesunu aplikace na ASP.NET Core je tato určitá automatizovanost. Proto jsem musel změnit uspořádání JSON de-serializace v Android aplikaci tak, aby dokázala zpracovat JSON pole bez zmíněného obalu.

Dále aplikace, jak bylo zmíněno v analýze, používá pro autentizaci jméno a heslo zasílané v Base64 formátu s každým požadavkem. Tuto autentizaci REST API nové serverové části ignoruje. a Účelem testování API "naživo"jsem

prozatím v aplikaci odstranil kód pro přihlašování (který už dále nebude možné využít, REST API ve frameworku Core používá jiný systém autentizace). Aplikace tedy v rámci této iterace plně komunikuje s novou serverovou částí.

4.1.3 Testování

Testování proběhlo dvojím způsobem, V průběhu implementace REST.API jsem pro testování používal aplikaci Postman, po dokončení implementace všech API jsem také funkčnost otestoval pomocí aplikace pro Android.

Postman

Postman (viz obrázek 4.2)je aplikace umožňující manuální HTTP komunikaci, včetně sestavení HTTP dotazů, včetně hlavičky a těla a poté získání odpovědi, ve stejném formátu. Každou takovouto metodu lze připravit, pojmenovat a uložit. Hlavní výhodou tohoto programu je ale schopnost automatizovat testování API za pomocí vzdálených "unit"testů, které jsou psány v Javě a přímo pracují s JSON objekty a testují za pomocí assert funkcionality. Díky tomu nebylo nutné psát unit testy přímo v serverové aplikaci.

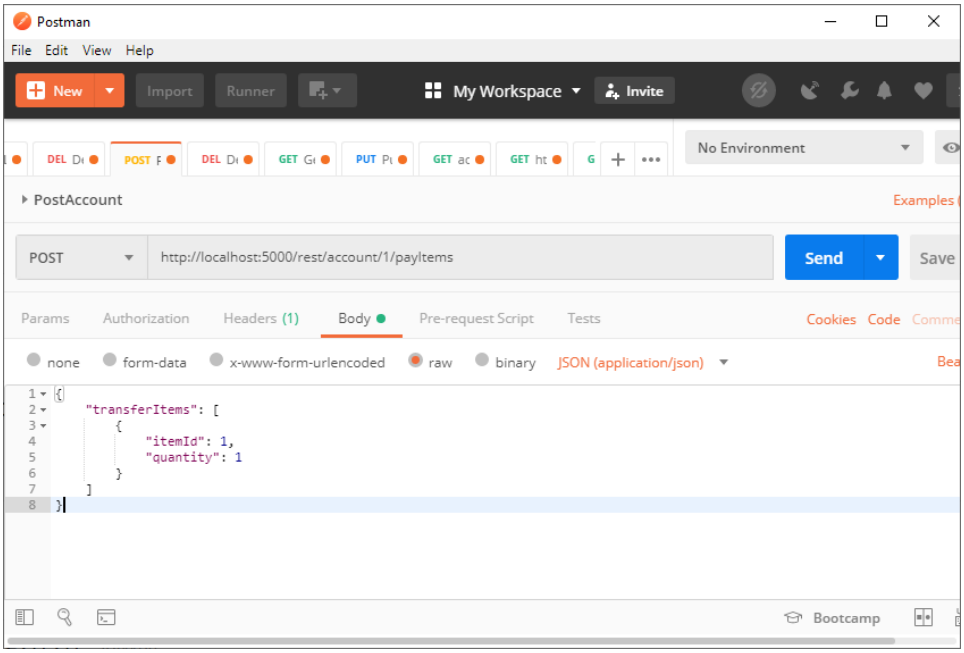
Příklad testovacího kódu

Testování zdroje /rest/table vypadá například takto, jelikož metoda pro tvorbu stolu automaticky vrací daný stůl jako JSON objekt:

```
pm.test("vkladani stolu", function () {
  pm.response.to.have.status(200)
  pm.expect(jsonData.data.cislo).is.to.equal(2);
  pm.expect(jsonData.data.name).is.to.equal("Stul u okna");
  pm.expect(jsonData).to.have.property('id');
  pm.expect(jsonData).to.not.have.property('userId');
});
```

4.1.4 Závěr iterace

Výsledkem iterace je aplikační vrstva, která již obsahuje nové funkční REST rozhraní 4.1, tedy pouze ty zdroje, které využívá Android klient. Není například možné registrovat nebo přihlásit uživatele, tuto funkcionalitu budu implementovat v následující iteraci.



Obrázek 4.2: Testování platby

4.2 Zabezpečení aplikace

Následující iterace se bude zbývat implementací zabezpečení, tedy požadavku RQ4

4.2.1 Návrh

Framework Core poskytuje robustní podporu zabezpečení ve formě frameworku Core Identity. Tento framework spravuje uživatelské účty, oprávnění i autorizaci pomocí anotací v REST API Controlleru. Využiji jej tedy v implementaci serverové části aplikace.

Do aplikační vrstvy budou přidány nové entity, pro zajištění předávání všech potřebných informací mezi klientem a serverem pomocí REST API. Tyto entity slouží výhradně jako šablony pro automatickou serializaci/de-serializaci požadavků.

- "LoginModel"pro odesílání přihlašovacích údajů
- "RegisterModel"pro odesílání registračních údajů

Pro reprezentaci uživatele stačí implicitní uživatel Indentity frameworku, protože již v základu podporuje vše potřebné, včetně stanovení a autorizace rolí.

Metoda	Zdroj
GET	rest/account/
GET	rest/account/id
POST	rest/account/
POST	rest/account/id/order
PUT	rest/account/id
POST	rest/account/id/payItems/cashRegister
POST	rest/account/idFrom/transferTo/idTo
DELETE	rest/account/id
GET	rest/Item
GET	rest/Item/id
POST	rest/Item
PUT	rest/Item/id
DELETE	rest/Item/id
GET	rest/Table
GET	rest/Table/id
POST	rest/Table
GET	rest/UserData
POST	rest/UserData

Tabulka 4.1: Popis nově implementovaných REST metod

4.2.2 Implementace

Nejdříve jsem implementoval pomocné entity, např. entita pro odesílání přihlašovacích údajů:

```
public class LoginModel
    {
        [Required]
        public string Email { get; set; }
        [Required]
        public string Password { get; set; }
        public bool RememberMe { get; set; }
    }
```

Registrace uživatelů probíhá pomocí zabudovaného REST zdroje /api/login, není tedy třeba přihlašování explicitně implementovat, to je již ve frameworku zařízeno. Ve verzi ASP.NET Core 3.1, ve které projekt implementuji, ale implicitní registrační formulář vytváří uživatele, který se může autentizovat pouze pomocí JWT Bearer tokenu, který je zašifrován asynchronní šifrou a implicitně je také vyžadováno zabezpečení pomocí host a klient IP adresy, toto je nežádoucí chování, jelikož např. Android klient poběží na arbitrární IP adrese. Proto jsem vytvořil vlastní jednoduchou registrační metodu která přebírá implicitního uživatele a registruje jej pro autentizaci pomocí standardního JWT tokenu:

```
[HttpPost]
public async Task<IActionResult> Post([FromBody]RegisterModel model) {
  var newUser = new IdentityUser { UserName = model.Email, Email = model.
```

```
        Email };
      var result = await _userManager.CreateAsync(newUser, model.Password);
      if (!result.Succeeded) {
        var errors = result.Errors.Select(x => x.Description);
        return Ok(new RegisterResult { Successful = false, Errors = errors
        });
      }
      return Ok(new RegisterResult { Successful = true });
    }
```

Po přihlášení poté stačí s každým REST požadavkem zasílat i JWT token, který neobsahuje uživatelské heslo

Úprava Android aplikace

V Android aplikaci musela být upravena přihlašovací aktivita, jelikož pomocné objekty na přenos hesla, i adresa zdroje URI jsou nyní jiné. Dále byl Apache HTTP klient v Android aplikaci přenastaven tak, aby upřednostňoval šifrované SSL spojení pomocí protokolu HTTPS, jelikož prvotní přihlášení stále přenáší jméno a heslo v plain-textu.

Jelikož bearer token je také Base64 řetězec, nemusela se upravovat funkcionalita autentizace v ostatních částech aplikace, pouze se změnil název parametru z "Basic"na "Bearer".

4.2.3 Testování

Pro účely testování jsem vytvořil novou sadu Unit testů, které zkouší registraci, přihlašování. Zaměřil jsem se především na testování autorizace REST zdrojů a také na použití nestandardních vstupů.

4.2.4 Závěr

Aplikace nyní obsahuje upravenou implementaci robustního frameworku Core Identity, který zajišťuje zabezpečení.

SSL funguje v aplikaci automaticky, není nutné nic nastavovat, jelikož bude aplikace nasazena na serveru *.azurewebsites.net a Visual Studio při nasazování vše automaticky nastaví. Toto by neplatilo, pokud by se aplikace někdy v budoucnu nasazovala na jiný webhosting, nebo lokálně.

4.3 Administrační rozhraní

Tato iterace se týká požadavků RQ 2, RQ 2.1, RQ 3, RQ4 a RQ5.

4.3.1 Návrh

Při vývoji s užitím frameworku Core se běžně v MVC modelu používá jako View framework Razor. Tento framework uvádí vkládání C# kódu do HTML dokumentů jednotlivých HTML stránek ve formě komponent

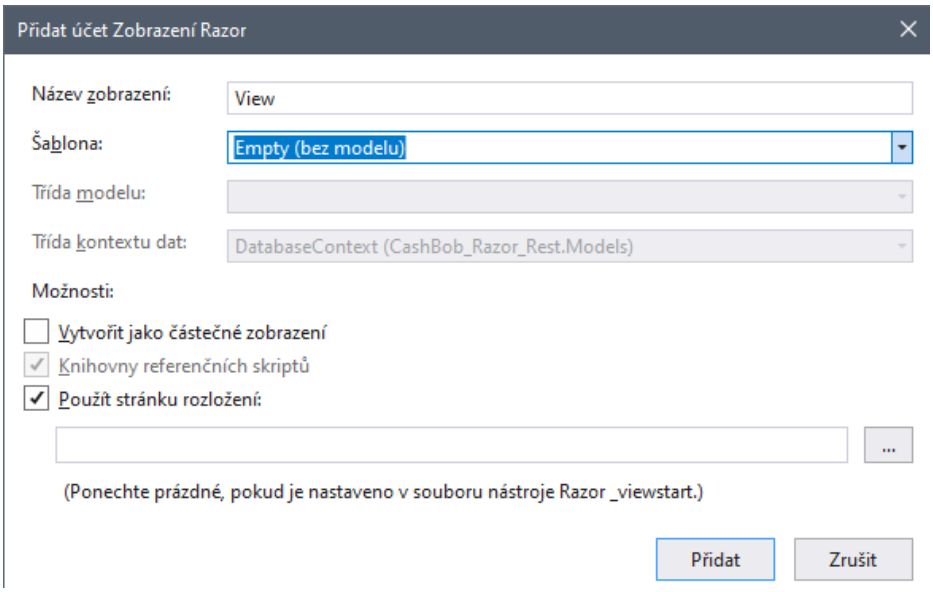
*.razor. Jednotlivé komponenty se mapují na webový server pomocí následující hlavičky na začátku každého souboru:

```
@page "/accounts"
@model CashBob_Razor_Rest.Pages.ItemPages.EditModel
```

Parametr @page určuje URI mapování, pomocí @using je možné využít jmenné prostory aplikace jako v jakémkoliv jiném *.cs souboru se zdrojovými kódy C#. Dále @model odkazuje na přidružený *.cs soubor se třídou dědící od PageModel, která implementuje metody pro práci s aplikační vrstvou.

Scaffolding

Razor komponenty je možné automaticky generovat pomocí funkcionality Scaffolding, obsažené ve Visual studiu (viz obrázek 4.3). Vygeneruje se kompletní CRUD model pro danou entitu.



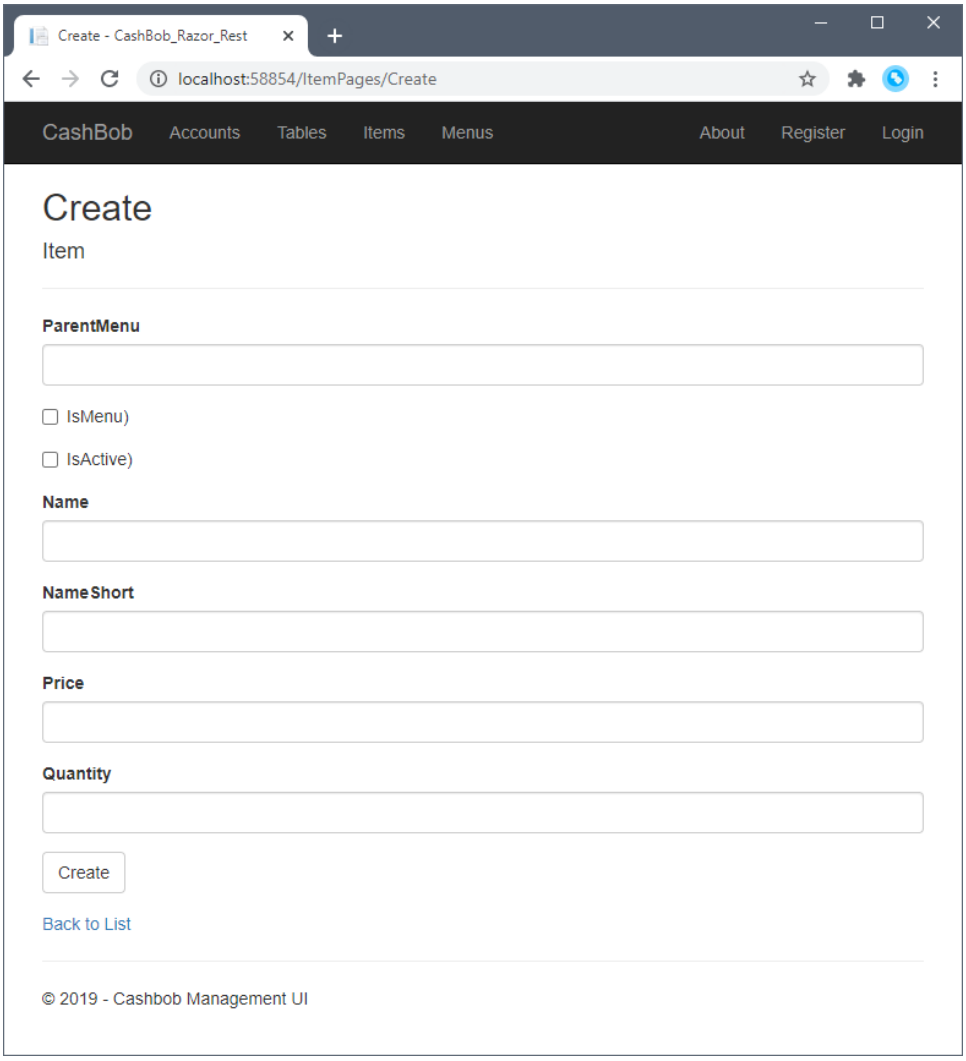
Obrázek 4.3: Scaffolding entit

4.3.2 Implementace

Pro každou entitu byla pomocí scaffoldingu vygenerována sada CRUD operací. Bohužel jsem při této operaci také zjistil, že scaffolding dokáže generovat CRUD operace pouze pro primitivní typy, vlastní třídy jsou ignorovány. Ukázka scaffoldingu pro operaci nad položkami menu viz obrázek **??**

4.3.3 Závěr

Po konzultaci s vedoucím práce panem Ing. Komárkem jsem se rozhodl ukončit tuto iteraci předčasně, ze dvou důvodů. V první řadě By generování uživatelského rozhraní tímto způsobem vyžadovalo implementace dvojí sady



Obrázek 4.4: Výsledná vygenerovaná stránka pro editaci položky menu

Controllerů aplikační vrstvy, jedna sada pro REST API, druhá pro Razor, což by vlastně vnitřní implementaci aplikace ještě zkomplikovalo, v porovnání s původní implementací serveru. A za druhé, scaffolding tímto způsobem ani nedokáže automaticky generovat objektové typy, úplně je ignoruje. V rámci další iterace tedy pozměním návrh struktury aplikace.

4.4 Nová verze administračního rozhraní

Tato iterace se opět týká požadavků RQ2, RQ2.1, RQ3, RQ4 a RQ5, jelikož předchozí iterace skončila neúspěchem.

4.4.1 Návrh

Místo Razor frameworku, který potřebuje vlastní separátní model entit, jsem se rozhodl použít framework Blazor, jelikož umožňuje použití existujícího REST rozhraní a také mlže běžet i v offline režimu.

Blazor

Blazor aplikace ve formě WebAssembly klienta ve webovém prohlížeči jsou ve frameworku Core novinkou[blaa], první verze se objevily v roce 2020 [blab]. Jak jsem již v návrhu popsal, může sdílet kódovou základnu s REST API serverem a dokonce může být daným serverem i distribuován, stačí zadat internetovou adresu serveru a aplikace se automaticky v podporovaných prohlížečích stáhne, uživatel vůbec nepozná rozdíl v porovnání s klasickými webovými stránkami psanými v HTML a JavaScriptu, které musí být generovány na serveru.

4.4.2 Implementace

Nejdříve jsem vytvořil nový ASP.NET Blazor projekt, do kterého jsem přemístil entity a Controllery z předchozí iterace. Nový projekt obsahuje dvě aplikace - Serverovou část a také nového Blazor klienta, který je již implicitně nastaven tak, aby komunikoval se serverem pomocí REST rozhraní.

Razor komponenty

Blazor pro používá Razor komponenty pro vykreslování uživatelského rozhraní. Hlavička každé komponenty, kterou jsem použil pro práci s CRUD, obsahuje:

```
@page "/accounts"
@using Cashbob_JWT_WASM.Shared
@using Microsoft.AspNetCore.Authorization
@attribute [Authorize]
@inject HttpClient Http
```

Kde je nově přítomna anotace [Authorize], která funguje stejně jako stejná anotace v Controlleru REST API, tedy pokud uživatel není přihlášen a autorizován, stránka se nezobrazí a objeví se chybová hláška "Not authorized"(Tu je možné upravit, případně použít přesměrování, v projektu jsem ale implementoval výchozí stav)

Nakonec je do každé komponenty pomocí @inject introdukováno API pro práci s REST, jelikož má každá komponenta nyní přístup do sdílené kódové základny Cashbob_JWT_WASM.Shared, je implementace vykreslování dat triviální, například vykreslení interface stolu:

```
@if (results == null) {
    <Loading />
} else {
    <table class="table">
        <thead>
            <tr>
                <th>Id</th>
```

```
                    <th>Table</th>
                    <th>Name</th>
                </tr>
            </thead>
            <tbody>
                @foreach (var result in results) {
                    <tr>
                        <td>@result.Id</td>
                        <td>@result.TableNumber</td>
                        <td>@result.Name</td>
                    </tr>
                }
            </tbody>
        </table>

        <Button Type="ButtonType.Link" Color="Color.Primary" To="tables/
        create">
            <Icon Name="IconName.Add" />
             Create new Table
        </Button>
    }
```

A kód pro získání stolů z REST rozhraní vypadá následovně, metoda se zavolá při inicializaci komponenty:

```
    @code {
        private Cashbob_JWT_WASM.Shared.Table[] results;

        protected override async Task OnInitializedAsync() {
            results = await Http.GetFromJsonAsync<Cashbob_JWT_WASM.Shared.
        Table[]>("rest/table");
        }

    }
```

Objekt Http implementuje všechny metody REST API, pro odeslání dat stačí např. použít PostAsJsonAsync a metoda dokonce automaticky i entity serializuje. Není tedy zapotřebí vůbec žádný JavaScript, kód klienta je psaný čistě v C#.

Autentizace

Autentizace probíhá ve dvou fázích - autentizace jednotlivých stránek pomocí [Authorize] a poté autentizace jednotlivých REST API probíhá přímo na serveru - při selhání autorizace nebo autentizace server zašle v odpovědi kód 401, na který aplikace Blazor mlže reagovat. Jelikož kód klienta je všechen na straně klienta, autentizace jednotlivých stránek má tedy spíše smysl vizuální, všechna důležitá autorizace probíhá na straně serveru.

Jelikož všechna komunikace se serverem probíhá jen a pouze pomocí REST API, je autorizace řešena stejným způsobem jako u Android klienta - pomocí Bearer tokenu, který se uloží jako cookie do prohlížeče podobným způsobem, jako Session cookie u běžné internetové stránky.

Získání Bearer tokenu pomocí uživatelem zadaného jména a hesla:

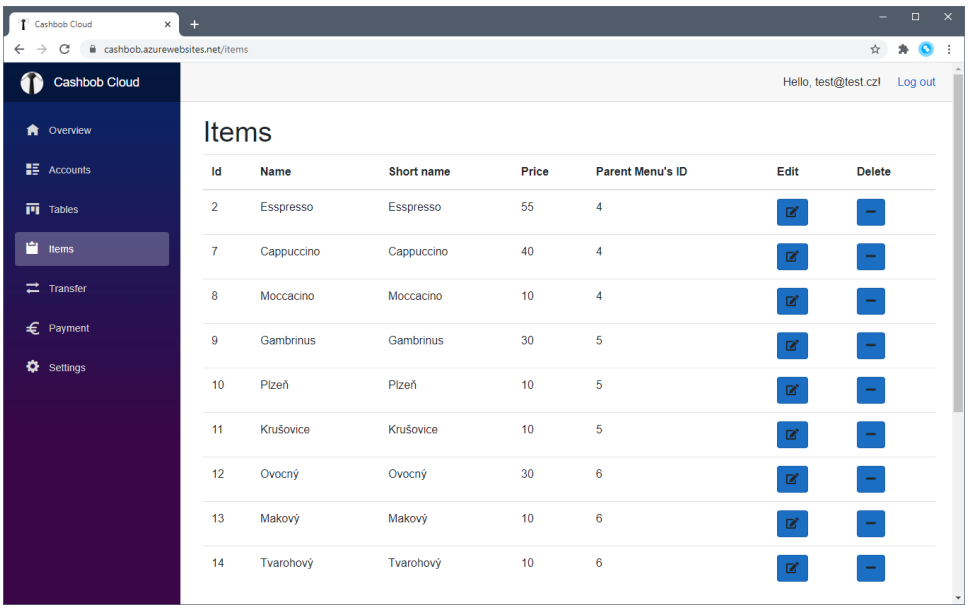
```
var loginAsJson = JsonSerializer.Serialize(loginModel);
var response = await _httpClient.PostAsync("api/Login", new
    StringContent(loginAsJson, Encoding.UTF8, "application/json"));
var loginResult = JsonSerializer.Deserialize<LoginResult>(await response.
    Content.ReadAsStringAsync(), new JsonSerializerOptions {
    PropertyNameCaseInsensitive = true });
```

Uložení tokenu jako cookie a přihlášení se:

```
await _localStorage.SetItemAsync("authToken", loginResult.Token);
((ApiAuthenticationStateProvider)_authenticationStateProvider).
    MarkUserAsAuthenticated(loginModel.Email);
httpClient.DefaultRequestHeaders.Authorization = new
    AuthenticationHeaderValue("bearer", loginResult.Token);
```

4.4.3 Testování

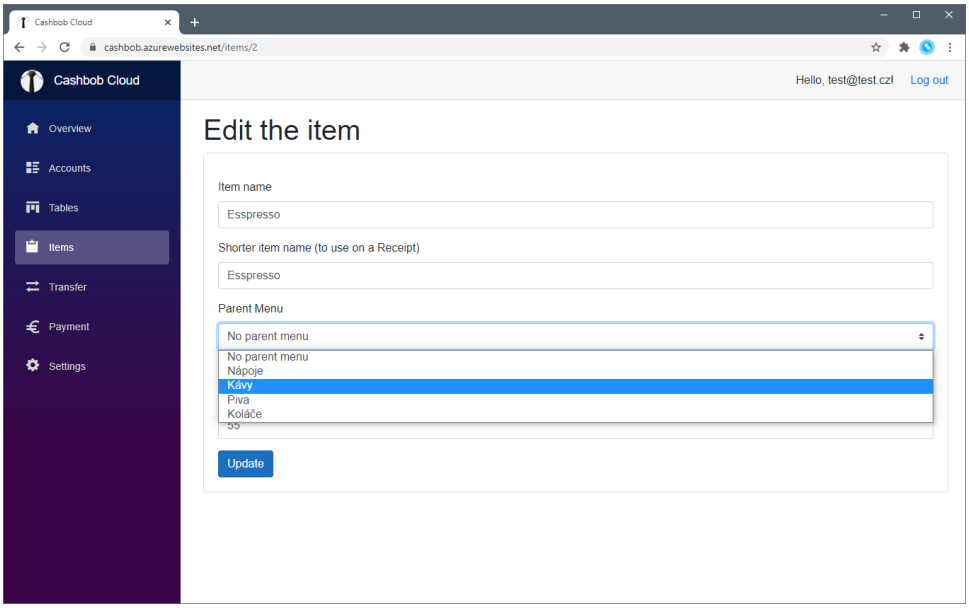
Aplikace v implementované podobě funguje pouze jako tenký klient, data z REST API zobrazuje a odesílá nové JSON objekty. A rozhraní API je již průběžně automaticky testováno pomocí nástroje Postman, jak jsem popsal v první iteraci. Jediná nová funkcionalita vyžadující testování je tedy uživatelského rozhraní samotné - to jsem testoval manuálně pomocí prohlížečů Google Chrome, Microsoft Edge a Mozilla Firefox. Ná obrázcích 4.5 a 4.6 je vidět výsledná podoba uživatelského rozhraní.



Obrázek 4.5: Zobrazení administračního rozhraní

4.4.4 Závěr

Závěrem této iterace je tedy plně funkční správa podniku - je možné vytvářet položky menu, přiřadit je do kategorií a nastavit informace o podniku, včetně



Obrázek 4.6: Editace položky menu

dat potřebných pro funkci EET. V budoucnu by bylo vhodné zavést automatizované testy uživatelského rozhraní, jako například Selenium [sel]. Plně implementovány byly RQ2, RQ2.1 a RQ5. Došlo k částečné implementaci RQ3- Je možné vytvářet nové účty, zbytek funkcionality je dostupný pouze v Android aplikaci.

4.5 Evidence tržeb EET

Iterace se soustředí na implementaci požadavku RQ7.

4.5.1 Návrh

Jak jsem v analýze technologií zmínil, ASP.NET má v rámci API utility pouze pro serializaci a de-serializaci XML zpráv (SOAP objektů), ne už ale pro jejich podepisování, tato funkcionalita byla opomenuta při transformaci z původního uzavřeného ASP.NET frameworku. To znamená, že mám při návrhu komunikace dvě možnosti - naprogramovat podepisování sám, nebo použít otevřenou knihovnu. Jelikož EET používá pro podepisování poměrně komplexní systém kontroly schémat WS-Security[wss], rozhodl jsem se místo toho použít knihovnu s licencí MIT (tedy nijak nebrání v případnému zpoplatnění).

Knihovna

Jedná se o knihovnu

```
https://drive.google.com/drive/folders/0B2B4_OfsI25paTB2R0NNM1hqMzg
```

Knihovna pro chod vyžaduje plné prostředí .NET a proto několik API z něj knihovna nemůže v menším frameworku Core najít. Volané API jsem proto nahradil buď ekvivalentním z Core, nebo kompletně přepsal. Detaily mých zásahů popíši v implementaci.

4.5.2 Implementace

Jelikož knihovna používá API které v Core neexistuje, musel sem místo přeloženého DLL vložit do projektu zdrojové kódy ve složce EET a ty následně upravit.

Třída SoapSigner

První problémy nastaly ve třídě SoapSigner, který se stará o podpis Soap objektu dle standardu WS-Security. Api frameworku Core neobsahuje popis šifry RSA/AES256. Samotný šifrovací algoritmus existuje, ale popis této kombinace chybí, musel jsem jej tedy doplnit:

```
    public RSAPKCS1SHA256SignatureDescription() {
                KeyAlgorithm = typeof(RSACryptoServiceProvider).FullName;
                DigestAlgorithm = typeof(SHA256Managed).FullName;
                FormatterAlgorithm = typeof(RSAPKCS1SignatureFormatter).
        FullName;
                DeformatterAlgorithm = typeof(RSAPKCS1SignatureDeformatter).
        FullName;
    }
```

Získání privátního klíče z certifikátu

Naproti tomu ke zjednodušení došlo při přepisu kódu pro získávání privátního klíče z certifikátu poplatníka - Core implementuje zjednodušenou metodu GetRSAPrivateKey():

```
    signedXml.SigningKey = certificate.GetRSAPrivateKey();
```

namísto

```
    var key = (RSACryptoServiceProvider)certificate.PrivateKey;
    var enhCsp = new RSACryptoServiceProvider().CspKeyContainerInfo;
    var cspparams = new CspParameters(enhCsp.ProviderType, enhCsp.
        ProviderName, key.CspKeyContainerInfo.KeyContainerName);
    rsaKey = new RSACryptoServiceProvider(cspparams);
    signedXml.SigningKey = rsaKey;
```

Použití knihovny

Do knihovny stačí dodat všechna potřebná data o poplatníkovi, certifikát a data o tržbě, včetně data a čísla účtenky. Poté komunikace s EET proběhne automaticky a získá se kód FIK, pokud je připojení úspěšné. Pokud ne, je výstupem alespoň kód PKP, který je pak možné vytisknout na účtenku místo FIK.

```
bool result = eet.OdeslaniTrzby();
```

Data účtenky

Struktura účtenky, která je po zaplacení odeslána zpět na pokladnu jako serializovaný JSON objekt:

```
//Creating Receipt
            ReceiptData RD = new ReceiptData {
                baseValue = (double)baseValue,
                DIC = currUserData.DIC,
                DPH = currUserData.DPH,
                EstablishmentId = currUserData.EstablishmentId,
                EstablishmentName = currUserData.EstablishmentName,
                taxValue = (double)taxValue,
                total = (double)totalPaid,
                FIK = ""
            };

            //EET info in receipt
            if (USE_EET && eet.Potvrzeni != null) {
                RD.FIK = eet.Potvrzeni.fik;
            } else {
                RD.PKP = eet.PKP;
            }
            RD.BKP = eet.BKP;
```

Archivace pro zjednodušený režim

Aplikace všechny data vygenerovaná při tvorbě účtenky uloží jako instanci objektu Transaction do příslušného účtu Account a uloží je do databáze. Pokud v průběhu komunikace se servery EET došlo k chybě, tedy byly například nedostupně, uloží se reference na Transaction také do pomocného databázové entity PendingEET, odkud aplikace může dávkově tyto transakce zpracovat po obnovení připojení.

4.5.3 Testování

Testování funkčnosti výsledného REST API proběhlo opět v rámci aplikace Postman, kdy jsem do Unit testů jednoduše přidal také podporu JSON atributů v účtence spjatých s EET (Tedy FIK, PKP, BKP). Funkci zjednodušeného režimu jsem testoval odpojením od internetu pro vygenerování účtenky s PKP místo FIK a následně opětovným připojením a odesláním dat dodatečně.

4.5.4 Závěr

Výsledkem iterace je plně funkční odesílání tržby na server finanční správy. Data o transakci jsou uložena v rámci účtu a data o účtence odeslána zpět pokladní Android aplikaci k vytisknutí. Pokud by nebylo k dispozici připojení

k internetu, je možné data uložit ke zpracování později v tzv. offline zjednodušeném režimu, ve kterém je pak na účtence prozatím vytisknut kód BKP místo FIK. Zbýva už jen implementovat tisk v Android aplikaci, což bude náplní příští iterace. Android aplikace v původním stavu implementovala tisk PDF, jelikož toto PDF generoval v původní aplikaci Server a soubor odeslal zpět klientovi. v Nové implementaci se zasílají jen textová data k vytisknutí, proto bude nutné tisk upravit.

4.6 Tisk účtenek

Pokračování implementace RQ7.

4.6.1 Návrh

Jak jsem zmínil v předchozí sekci, Původní Android aplikace obsahuje tisk PDF. Implementováno toto bylo pomocí proprietární knihovny pro tiskárny STAR - toto je tedy zcela nevyhovující, jelikož moderní Bluetooth tiskárny, jako např Cashino PTP-II, kterou jsem použil pro testování, podporují tisk jednoduchého datového proudu pomocí Bluetooth socketu - nejsou nutné žádné proprietární ovladače, potřebná funkcionalita je již obsažená přímo v systému Android, je nutné pouze inicializovat spojení.

Z projektu tedy odstraním knihovny:

```
*.printing.sdk.*
*.printing.PrintUtils
```

4.6.2 Implementace

Implementace spočívá v automatickém vybrání spárovaného zařízení s podporou BluetoothSocket (tedy tiskárny) a otevření onoho socketu.

Příprava výstupního proudu

Nejdříve je nutné získat Bluetooth BluetoothSocket spárovaného zařízení:

```
BluetoothSocket socket = null;
socket = (BluetoothSocket) device.getClass().getMethod("
    createRfcommSocket", new Class[]{int.class}).invoke(device, 1);
socket.connect();
```

Následně získáme výstupní datový proud:

```
OutputStream btOutputStream = socket.getOutputStream();
```

A následně již stačí do datového proudu vypisovat řetězec ve formě bytů:

```
btOutputStream.write(encodeNonAscii(text).getBytes());
```

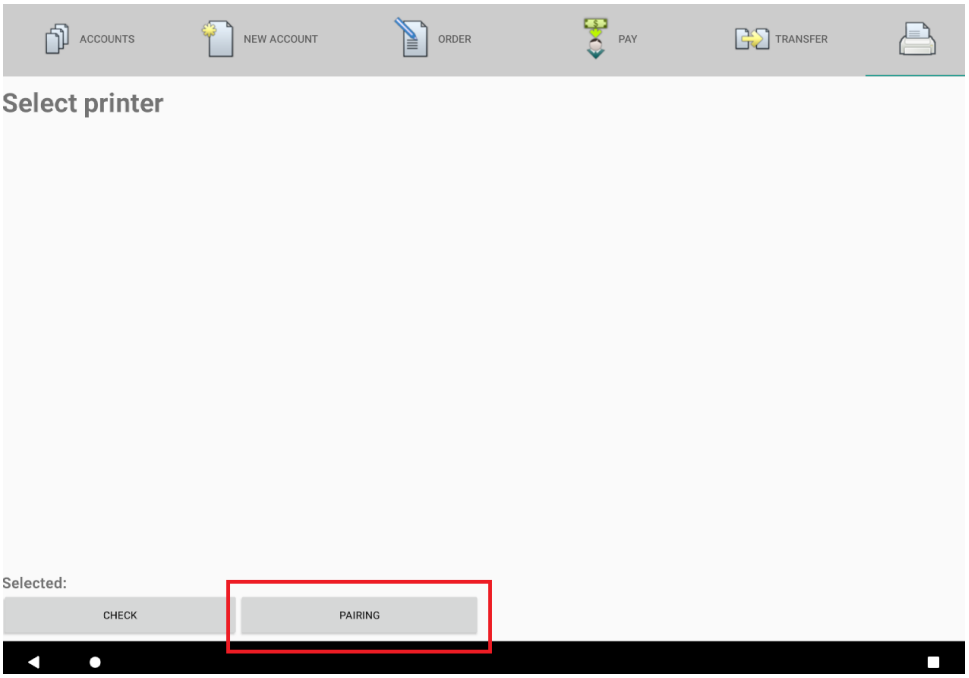
Tiskárna začne text tisknout okamžitě po přijetí, nový řádek se vytiskne jednoduše pomocí speciálního formátovácího znaku /n.

Samotný tisk

Aplikace již měla uživatelské rozhraní přizpůsobené k tisku - tisk pomocí proprietární knihovny jsem již jednou implementoval v rámci své bakalářské práce [Zá16]. Nahradil jsem tedy pouze tiskovou funkcionalitu v místech, kde se původně volaly metody staré knihovny. Tiskárna se připojuje automaticky, stačí ji spárovat se zařízením, k tomu slouží tlačítko v sekci aplikace pro nastavení, viz obrázek 4.7.

Příklad tisku účtenky

```
mPrinter.printText(rd.EstablishmentName);
mPrinter.addNewLine();
mPrinter.printText("  DIC: " + rd.DIC+"");
mPrinter.addNewLine();
mPrinter.printText("Table: " + rd.EstablishmentId);
mPrinter.addNewLine();
mPrinter.printText(" Date: " + new SimpleDateFormat("HH:mm:ss dd/MM/yyyy
    ").format(Calendar.getInstance().getTime()));
mPrinter.addNewLine();
mPrinter.printText("   RN: " + rd.receiptnumber);
mPrinter.addNewLine();
mPrinter.printText("------------------------------");
mPrinter.addNewLine();
```



Obrázek 4.7: Interface pro práci s tiskárnou

4.6.3 Testování

Tisk byl testován pomocí tiskárny Cashino PTP-II [cas]. Byly testovány různé scénáře, například pokus o tisk bez spárované tiskárny, tisk s více spárovanými

zařízeními, vypnutí tiskárny během tisku, automatické obnovení spojení.

4.6.4 Závěr

Implementací podpory tisku byl splněn RQ7. Systém je nyní plně připraven elektronicky evidovat tržby na serveru EET a vytisknout účtenku. Vydání EET účtenky je nutnou součástí fyzické transakce [vyd], proto by nemělo být možné tuto uzavřít na straně klienta bez vytisknutí. Proto, pokud by při prvotním požadavku o zpracování platby selhal tisk, existuje v Android aplikaci ještě tlačítko, které vytiskne účtenku pro poslední platbu. Číšník tedy může znovu připojit tiskárnu a vydat účtenku zákazníkovi.

Kapitola 5

Závěr a zhodnocení práce

Závěrem bych rád zhodnotil výsledek a míru splnění zadání této diplomové práce. Vývoj aplikace probíhal iterativně. Každá iterace s sebou nesla nový cyklus návrhu, implementace, testování a zhodnocení. Díky tomu bylo možné design aplikace postupně upravovat tak, aby výsledná aplikace co nejlépe splňovala očekávání. Například při první iteraci, ve které jsem se seznamoval s novým programovacím jazykem i frameworkem, jsem zjistil, že tzv. scaffolding pomocí Core MVC frameworku Razor není tou správnou cestou. Automatizovala by se sice základní CRUD struktura prezenční vrstvy aplikace, což by zrychlilo prototypování, musela by se ale implementovat aplikační logika dvakrát - jednou pro MVC a zvlášť také pro API. Nalezl jsem proto lepší řešení - Použít místo MVC framework Blazor. Ten vlastně serverovou část aplikace rozdělil na dva moduly, které spolu komunikují pouze přes rozhraní REST API. A vzhledem k tomu, že Blazor klient běží celý za pomocí WebAssembly technologie v prohlížeči, bude možné webovou část systému provozovat i čistě offline za pomocí cache a aktualizovat data jen, když bude potřeba změnit menu podniku. Jak bylo zmíněno v příslušné kapitole, tato funkcionalita zatím není implementována jelikož toto nebylo součástí Scope projektu, implementace by ale případně měla být triviální.

Byla plně implementována funkcionalita EET, včetně ukládání certifikátu poplatníka, dat podniku, odesílání podepsané XML zprávy na server finanční správy ČR a získání nazpět kódu FIK. V případě nemožnosti komunikace se serverem finanční správy aplikace pracuje v zjednodušeném offline režimu, ve kterém je zákazníkovi vydána účtenka pouze s kontrolními součty a serverová část systému pak data o nekompletní transakci uloží do databáze, odkud je pak možné data načíst a zprávy o tržbách dávkově odeslat.

Systém je plně připraven k nasazení jako Cloud služba, kdy poplatník daně lokálně používá pouze Android modul aplikace jako pokladnu s přenosnou tiskárnou na účtenky. Administrace probíhá pomocí web aplikace ve webovém prohlížeči. Zde je možné spravovat podniková data, jako položky menu, stoly a data potřebná k fungování EET.

V převodu na ASP.NET Core jsem měl i vlastní zájem. CashBob je studentský projekt, který již za dobu svého vývoje doznal spousty změn, jmenovitě například přesun od lokální desktop aplikace postupně ke kombinaci více modulů komunikujících přes rozhraní Java RMI[rmi] až k zavedení standardi-

zovaného multiplatformního API REST. Bohužel, nejnovější iterace aplikace je sestavena pomocí nástrojů (framework Play!), které nejsou tak běžně používány a při mých vlastních pokusech o překlad jsem se z počátku potýkal s problémy. Po konzultaci s vedoucím práce a uvážení dalších technologických omezení programovacího jazyka a frameworků, jsem se tedy rozhodl převést aplikaci k Microsoft platformě .NET - ještě v rámci semestrálního projektu, který předcházel tuto diplomovou práci. Zároveň se jednalo o příležitost vyzkoušet si převod aplikace z jednoho prostředí do druhého, při zachování funkcionality veřejných API původního řešení. Myslím tedy, že jsem si z těchto důvodů z projektu mnoho odnesl.

5.1 Míra splnění zadání

- Byla analyzována technologická a implementační omezení původní implementace serverové části systému CashBob
- Byla analyzovány výhody/nevýhody přechodu na platformu ASP.NET
- Byla navržena nová serverová aplikace CashBob, která odstraňuje technologické nedostatky původní implementace. Aplikace byla vyvíjena iterativně.
- Byla implementována plná podpora EET evidence tržeb a aplikace nyní podporuje tisk účtenek.
- Srovnání funkcí výsledného systému vůči podobným zavedeným systémům je v následujícím odstavci.

5.2 Srovnání s konkurenčními aplikacemi

V kapitole 3.1 jsem vypracoval souhrn rešerše aplikací podobných systému CashBob. Na základě funkcionality, kterou jsem při ní zkoumal jsem upravil funkční požadavky aplikace tak, aby v případě jejich splnění, systém podporoval obdobnou funkcionalitu. Bohužel, nepodařilo se toto z časových důvodů splnit a z vyčtených funkcí aplikace nepodporuje následující:

- Platby platební kartou (nebylo součástí Scope)
- Webová aplikace
- Přístup k historii tržeb pomocí UI
- Správa zásob (nízká priorita)
- Mapa stolů (Android aplikace byla pouze upravena pro fungování s pozměněným API)

5.3 Možné budoucí vylepšení

Implementovány byly téměř všechny funkční a nefunkční požadavky, až na některé výjimky. Například správa skladu bohužel není implementována. Tato funkcionalita chybí i v původním systému Cashbob, nepřikládal jsem tedy tomuto požadavku prioritu. Do budoucna by ale bylo vhodné tuto funkcionalitu doimplementovat, za účelem zvýšení konkurenceschopnosti systému. Také jsem bohužel musel dle zadání upřednostnit plnou funkcionalitu REST API a již jsem neměl čas implementovat funkcionalitu pokladny na webovém rozhraní. Jako pokladna tedy v současné době slouží pouze Android aplikace, což opět snižuje hodnotu aplikace mezi konkurencí. Na závěr by také bylo dobré provést revizi uživatelského rozhraní Android aplikace, alespoň změnou zobrazení menu, místo jednoduchého jednorozměrného listu, dvourozměrná mřížka s možností uspořádat si kategorie manuálně tak, jak je to v původní webové aplikaci.

5.4 Nasazení aplikace

Serverová část aplikace je ve stavu, ve kterém byla při odevzdání projektu, nasazena na webovém serveru:

```
https://cashbob.azurewebsites.net
```

Je samozřejmě možné se k ní připojit z upravené verze Android aplikace, která je také součástí výstupu této diplomové práce.

Literatura

[bkp] *Vyhláška č. 269/2016 sb.*, https://www.etrzby.cz/assets/cs/prilohy/Vyh_2016-269_sb0104-2016.pdf, [Online; navštíveno 12.12.2020].

[blaa] *Introduction to asp.net core blazor*, https://docs.microsoft.com/cs-cz/aspnet/core/blazor/?view=aspnetcore-5.0, [Online; navštíveno 18.12.2020].

[blab] *What's behind the hype about blazor?*, https://stackoverflow.blog/2020/02/26/whats-behind-the-hype-about-blazor/, [Online; navštíveno 21.12.2020].

[cas] *Cashino ptp-ii dual bt*, https://www.eet-tiskarny.cz/eshop/ptp-ii-58mm-eet-erecept-mobilni-tiskarna.html, [Online; navštíveno 20.12.2020].

[clo] *Cloud computing*, https://cs.wikipedia.org/wiki/Cloud_computing, [Online; navštíveno 22.5.2019].

[cor] *Introduction to asp.net core*, https://docs.microsoft.com/cs-cz/aspnet/core/?view=aspnetcore-2.2, [Online; navštíveno 22.5.2019].

[Če16] Tomáš Červenka, *Webové rozhraní restauračního systému*, 2016.

[eeta] *Eet popis rozhraní*, https://www.etrzby.cz/assets/cs/prilohy/EET_popis_rozhrani_v3.1.1.pdf, [Online; navštíveno 22.5.2019].

[eetb] *etržby- elektronická evidence tržeb*, https://www.etrzby.cz/, [Online; navštíveno 17.5.2019].

[eetc] *Zákon č. 112/2016 sb.*, https://www.mfcr.cz/cs/legislativa/legislativni-dokumenty/2016/zakon-c-112-2016-sb-26768, [Online; navštíveno 8.12.2020].

[ef] *Entity framework core*, https://docs.microsoft.com/cs-cz/ef/core/, [Online; navštíveno 21.12.2020].

[Hog16] Tomáš Hogenauer, *Bakalářská práce tomáše hogenauera*, 2016.

[int] *Vývoj internetu v Čr za posledních 10 let*, https://www.lupa.cz/pr-clanky/vyvoj-internetu-v-cr-za-poslednich-10-let/, [Online; navštíveno 22.5.2019].

[lin] *Linq*, https://cs.wikipedia.org/wiki/LINQ, [Online; navštíveno 22.5.2019].

[paa] *Platforma jako služba*, https://azure.microsoft.com/cs-cz/overview/what-is-paas/, [Online; navštíveno 13.12.2020].

[pla] *Češi a platební styk 2019*, https://cbaonline.cz/cesi-a-platebni-styk-2019, [Online; navštíveno 14.12.2020].

[pok] *Nejlepší eet pokladna 2019*, https://www.5nej.cz/srovnani-eet-pokladen/, [Online; navštíveno 22.5.2019].

[res] *Rest api tutorial*, https://restfulapi.net/rest-architectural-constraints/, [Online; navštíveno 22.5.2019].

[rmi] *Java remote method invocation*, https://cs.wikipedia.org/wiki/Java_remote_method_invocation, [Online; navštíveno 22.5.2019].

[sel] *Seleniumhq browser automation*, https://www.selenium.dev/, [Online; navštíveno 2.1.2021].

[uct] *Vzorová účtenka podle zákona o evidenci tržeb*, https://www.etrzby.cz/assets/cs/prilohy/vzorova_uctenka.pdf, [Online; navštíveno 12.12.2020].

[vst] *Visual studio community*, https://visualstudio.microsoft.com/cs/vs/community/, [Online; navštíveno 27.12.2020].

[vyd] *Je prodávající povinen mi vystavit doklad o koupi?*, https://www.coi.cz/faq/2-je-prodavajici-povinen-mi-vystavit-doklad-o-koupi/, [Online; navštíveno 13.12.2020].

[wss] *Ws-security*, https://www.ibm.com/support/knowledgecenter/SSTTDS_11.0.0/com.ibm.etools.mft.doc/ac55630_.html, [Online; navštíveno 3.1.2021].

[Zá16] Pavel Zářecký, *Vývoj aplikací na platformě android*, 2016.

Příloha A

Seznam příloh

Zdrojové kódy systému Cashbob. sources.zip

Tato práce ve formátu PDF. thesis-zarecky.pdf