Jihočeská univerzita v Českých Budějovicích
Přírodovědecká fakulta

# Systém pro tvorbu vyúčtování

Diplomová Práce

**Bc. Jakub Vacek, DiS.**

Ing. Jan Píša
Fiedler AMS

České Budějovice 2019

# Bibliografické údaje

Vacek J., 2019: Systém pro tvorbu vyúčtování [Billing system Mgr. Thesis, in Czech.] – 85 p., Faculty of Science, The University of South Bohemia, České Budějovice, Czech Republic.

# Anotace

Tato magisterská práce se zabývá návrhem a implementací systému pro tvorbu vyúčtování. V první části práce jsou definovány zamýšlené funkce systému a analyzována data služeb, které je nutné účtovat. Následně je popsán návrh architektury systému, jeho částí a způsobu řešení jednotlivých problémů. Poté je stručně popsána zvolená sada technologií, s jejíž pomocí je implementován systém vycházející z předchozích částí práce. Ná závěr je popsán způsob testování vytvořeného systému.

**Klíčová slova:** Vyúčtování, Apache Kafka, Typescript, MySQL, Node.js

# Annotation

This master thesis deals with design and implementation of billing system. First, a system function definition is created and service billing data is analysed. Further, the architecture of the system and solutions to all presented problems are described. The selected technology stack is briefly described and then used to implement the system. In the end, a system based on the structural part of the work is implemented and the testing process is presented.

**Key words:** Billing, Apache Kafka, Typescript, MySQL, Node.js

## Prohlášení

Prohlašuji, že svoji diplomovou práci jsem vypracoval samostatně pouze s použitím pramenů a literatury uvedených v seznamu citované literatury. Prohlašuji, že v souladu s § 47b zákona č. 111/1998 Sb. v platném znění souhlasím se zveřejněním své diplomové práce, a to v úpravě vzniklé vypuštěním vyznačených částí archivovaných Přírodovědeckou fakultou, elektronickou cestou ve veřejně přístupné části databáze STAG provozované Jihočeskou univerzitou v Českých Budějovicích na jejích internetových stránkách, a to se zachováním mého autorského práva k odevzdanému textu této kvalifikační práce. Souhlasím dále s tím, aby stejnou elektronickou cestou byly v souladu s uvedeným ustanovením zákona č. 111/1998 Sb. zveřejněny posudky školitele a oponentů práce i záznam o průběhu a výsledku obhajoby kvalifikační práce. Rovněž souhlasím s porovnáním textu mé kvalifikační práce s databází kvalifikačních prací Theses.cz provozovanou Národním registrem vysokoškolských kvalifikačních prací a systémem na odhalování plagiátů.

V Českých Budějovicích dne _____ _____

podpis

# Poděkování

Děkuji především vedoucímu absolventské práce Ing. Janu Píšovi, za trpělivé vedení, cenné rady a čas, který věnoval mé práci. Mé díky patří také Ing. Jindřichovi Fiedlerovi za příležitost realizovat tuto práci pro společnost Fiedler AMS s.r.o. Dále děkuji tvůrcům LaTeX 2$_\varepsilon$[1] za vytvoření systému, který mi značně ulehčil samotné sepsání práce. Děkuji mým blízkým za podporu během studia.


[1] LaTeX 2$_\varepsilon$ je rozšíření systému LaTeX což je kolekce maker pro TeX. TeX je ochranná známka American Mathematical Society.

# Obsah

# Seznam použitých zkratek

| Zkratka | Význam |
|---|---|
| HTTP | Hyper Text Transfer Protocol |
| PNG | Portable Network Graphics |
| TXT | Text File |
| GPRS | General Packet Radio Service |
| GSM | Global System for Mobile Communications |
| NPM | Node Package Mannager |
| GUI | Graphical User Interface |
| XML | Extensible Markup Language |
| HW | Hardware |
| CSV | Comma Separated Values |
| SCADA | Supervisory Control And Data Acquisition |
| DTO | Data Transfer Object |
| SMS | Short Message Service |
| SIM | Subscriber Identity Module |
| PDF | Portable Document Format |
| JSON | JavaScript Object Notation |
| API | Application Programming Interface |
| REST | Representational State Transfer |
| URL | Uniform Resource Locator |
| OOP | objektově orientované programování |
| UUID | Universal Unique Identifier |

# Kapitola 1

# Úvod

V dnešní době existují společnosti, které, mimo svůj hlavní obor podnikání, nabízejí další služby. Tyto služby představuje například zapůjčení SIM karet pro provoz zařízení nabízených společností nebo pronájem místa na datovém serveru pro sběr dat. Jednotlivé služby stejného typu se mohou značně lišit v nákladech, které pro společnost představují a je proto nevhodné je účtovat za pevnou cenu. Je nutné účtovat jednotlivé služby podle skutečných nákladů vynaložených společností na jejich poskytnutí.

Předkládaná magisterská práce se zabývá tvorbou systému, který bude zajišťovat automatické přiřazení jednotlivých služeb zákazníkům a následný výpočet ceny služby za určité časové období. Práce byla zpracována ve společnosti Fiedler AMS. Hlavní motivací pro tvorbu popisovaného systému je snaha o zajištění spravedlivého výpočtu ceny poskytovaných služeb a snížení zátěže účetního oddělení společnosti.

Nejprve jsou definovány funkce, které systém poskytuje. Následně je provedena analýza účtovaných služeb. Na základě analytické části je navržena struktura systému, rozděleny funkce jednotlivých serverů a definována struktura vstupních dat. Poté je čtenář seznámen s některými technologiemi použitými při implementaci vyvíjeného systému. Dále se práce věnuje implementaci jednotlivých serverů a funkcí, které řeší. Na závěr práce jsou popsány technologie použité k testování a samotný průběh testování systému.

Cílem práce je návrh a implementace systému pro tvorbu vyúčtování. Systém bude načítat data účtovaných služeb z několika serverů. Načtená data bude třídit podle zákazníka a odesílat na řídící server. Řídící server bude komunikovat pomocí REST API se serverem zajišťující naceňovaní a také s GUI (návrh Gui není součástí práce). Server pro naceňování bude umožňovat aplikaci různých ceníků. Data budou ukládána v MySQL databázi.

Dílčí cíle práce jsou:

- Analýza a návrh vhodné struktury dat od serverů jednotlivých služeb.
- Implementace serveru načítající data jednotlivých služeb.
- Implementace řídícího serveru, návrh řešení situací kdy nelze určit zákazníka.
- Návrh struktury ceníku a implementace serveru pro naceňování.

Struktura této magisterské práce je následující. V kapitole 2 jsou vymezeny základní funkce vytvářeného systému a popsána analýza služeb, které je nutné účtovat. V závěru kapitoly je popsáno několik základních příkladů užití systému. Kapitola 3 pojednává o návrhu architektury celého systému. Nejprve je popsáno rozdělení systému na jednotlivé části. Následně je čtenář seznámen s funkcí jednotlivých serverů a problémy, které řeší. Kapitola 4 se zabývá implementací jednotlivých serverů celého systému. První část pojednává o technologiích použitých při tvorbě systému. Dále je postupně popsána tvorba serverů systému. U každého serveru je čtenář seznámen se způsobem řešení problémů specifických pro daný server. V kapitole 5 jsou popsány technologie použité při testování systému a následně testování samotné. V závěru práce, tedy kapitole 6, jsou vyhodnoceny cíle práce a naznačen další možný rozvoj systému.

## 1.1 Řešený problém

Práce je zpracována pro společnost Fiedler AMS. Společnost se zabývá výrobou vlastních senzorů, telemetrických sestav pro sběr, analýzu a vizualizaci dat, a řízení procesů pro monitorování životního prostředí [17]. Naprostá většina instalovaných přístrojů obsahuje zabudovaný GSM/GPRS datový modul jehož prostřednictvím zařízení automaticky předává změřená data na hlavní databázový server v internetu [16]. Společnost nabízí

pronájem SIM karet pro tento modul [18]. Vizualizaci dat na databázovém serveru a jejich případný export zajišťuje webová aplikace CloudFM [17]. Aplikace poskytuje možnosti exportu do formátů CSV, TXT a PNG [18]. Pronájem databázového serveru, pronájem SIM, exporty do jednotlivých formátů a další služby jsou v současné době účtovány za pevnou cenu. To není vhodné z důvodu rozdílných nákladů vynaložených společností. Příkladem mohou být dva požadavky o export dat do formátu CSV - první exportuje pouze jeden záznam, kdežto druhý exportuje všechny záznamy, které stanice odeslala. První požadavek potřebuje pouze minimální výpočetní výkon. Oproti tomu druhý požadavek potřebuje výrazně větší výpočetní výkon. Je vhodné aby se tento rozdíl projevil v konečné ceně obou exportů. Zároveň by bylo vhodné aby se cena jednotlivých služeb počítala automaticky, bez nutnosti zásahu zaměstnance společnosti. Je tedy nutné analyzovat služby poskytované společností Fiedler AMS a určit parametry, které ovlivňují cenu služby. Dále je nutné navrhnout strukturu ceníku pro jednotlivé služby a definovat výpočet ceny služby. Před samotným určením ceny musí být služba přiřazena zákazníkovi, kterému bude naúčtována. Tento proces nemusí být vždy úspěšný, vyvíjený systém musí být schopný na tuto situaci reagovat. Systém tedy bude sloužit k automatickému přiřazení účtovaných služeb zákazníkům a výpočtu jejich ceny. Výsledkem bude objekt obsahující služby s přiřazenou cenou za určité časové období, náležící jednomu zákazníkovi. Tento objekt je možné získat pomocí REST API, případně exportovat do pdf.

# Kapitola 2

# Analytická část

Tato kapitola se zabývá analýzou, která předcházela návrhu systému. Nejprve bylo nutné definovat konkrétní funkce poskytované systémem. Následné bylo nutné analyzovat účtované služby. U jednotlivých služeb byly určeny parametry ovlivňující cenu. Na závěr jsou popsány vybrané případy použití systému.

## 2.1 Funkce systému

Definice funkcí systému vychází z konzultací s vedoucím práce, Ing. Janem Tůmou a Ing. Jindřichem Fiedlerem. Ná základě konzultací byla vytvořena množina funkcí s jejich popisem. Z této množiny byly následně vybrány funkce, které je nutné implementovat v první verzi systému a funkce, které budou definovány posléze. První verze systému bude poskytovat následující funkce:

- Systém bude načítat data z připojených serverů v unifikovaném objektu. Tento objekt představuje položku, kterou je nutné účtovat.
- Systém bude přiřazovat příchozí položky zákazníkovi, kterému bude položka účtována a bude umožňovat řešit situace kdy položku nelze přiřadit zákazníkovi.

- Systém bude ukládat nepřiřazené položky, položky čekající na výpočet ceny, parametry nutné pro výpočet ceny položky a již naceněné položky. Tyto data bude možné prohlížet pomocí REST API.

- Systém vypočte cenu obdržených položek za určité časové období. Tento výpočet ovlivňují obdržená data a nastavení od uživatele. Systém následně uloží položky s vypočtenou cenou do databáze k případnému exportu.

- Uživatel bude mít možnost spravovat množinu nastavení pro výpočet ceny položek - ceníků.

- Objekt obsahující množinu naceněných položek bude možné exportovat do PDF.

V budoucnu by bylo vhodné systém rozšířit o následující funkce:

- Systém bude umožňovat výpočet ceny předplacené služby. Zákazník si předplatí nějakou službu, systém poté měsíčně odečítá cenu služby od předplacené částky.

- Systém bude umožňovat vytvářet z dat naceněných položek faktury v určitých formátech. Tyto faktury bude možné exportovat, případně odesílat emailem.

- Systém bude zasílat upozornění o průběhu výpočtu ceny položek, případně stavu faktury emailem.

- Systém bude umožňovat odeslat data naceněných položek v určitém formátu emailem.

## 2.2 Data účtovaných služeb

Následující část se věnuje analýze jednotlivých služeb, které je nutné účtovat. Jsou identifikovány parametry služeb určující cenu dané služby. Na základě této analýzy je určena vhodná struktura dat. Při analýze bylo vycházeno z webových stránek společnosti Fiedler AMS [17] a [16], příručky ke stanici H3 a H7 [15] a konzultací s vedoucím práce. Nejprve jsou popsány služby SMS, email a WhatsApp, které zajišťují informování zákazníka o stavu stanice, déle jsou popsány služby týkající se exportu a importu dat.

### 2.2.1 SMS

Jedná se o SMS zprávu odeslanou ze serveru společnosti Fiedler AMS na základě hodnot přicházejících ze stanice. Cenu SMS určují následující hodnoty:

- použitý operátor
- využívaný tarif
- informace zda je příjemce zprávy v zahraničí či v České Republice
- počet zpráv nutných k přenesení dat

### 2.2.2 Odeslání email a WhatsApp zprávy

Server umožňuje odesílání upozornění pomocí email a WhatsApp zpráv. Cena zprávy je počítána podle počtu odeslaných zpráv.

### 2.2.3 Export dat do CSV

Webová aplikace společnosti Fiedler AMS umožňuje export dat ze serveru do formátu CSV. Tento proces zatěžuje HW společnosti a je proto vhodné jej účtovat. Cenu exportu určují následující hodnoty:

- počet záznamů v souboru
- velikost vyexportovaného souboru
- počet stanic ze kterých export probíhá
- počet streamů využitých při exportu

### 2.2.4 Export dat pomocí API

Webová aplikace společnosti Fiedler AMS umožňuje export dat ze serveru pomocí REST API ve formátu JSON a dalších formátech. Cenu exportu určují následující hodnoty:

- velikost souboru
- počet stanic ze kterých export probíhá
- počet streamů využitých při exportu

### 2.2.5 Export dat z manuálního načtení

Mezi výrobky společnosti patří i zařízení neumožňující fyzické odsílání dat. Ve webové aplikaci společnosti může být vytvořen zástupce takového zařízení a k němu mohou být přiřazena data, které je následně možné exportovat. Cenu exportu určují následující hodnoty:

- velikost souboru
- počet záznamů

### 2.2.6 Import dat na server

Webová aplikace umožňuje importovat data na server. Cenu importu určují následující hodnoty:

- počet záznamů
- objem dat
- počet stanic ze kterých import probíhá
- počet streamů využitých při importu

### 2.2.7 Založení stanice pro manuální odečet

Webová aplikace umožňuje založení stanice pro manuální odečet hodnot. Cenu této služby určuje skutečnost, že došlo k založení stanice určitého typu.

### 2.2.8 Nahrání dat do stanice pro manuální odečet

Webová aplikace umožňuje také nahrání dat a metadat do stanice pro manuální odečet hodnot. Cenu této služby určuje velikost přenášených dat.

### 2.2.9 Datahosting

Společnost Fiedler AMS poskytuje pronájem datového prostoru na serveru pro sběr dat z jednotlivých stanic. Cenu pronájmu určují následující hodnoty:

- typ datahostingu, který se mění podle počtu obsluhovaných stanic
- typ obsluhované stanice

Byly vytvořeny dva typy položek v závislosti na typu datahostingu. Konkrétně se jedná o třídy:

- `StandardDatahosting`
- `PremiumDatahosting`

Obě výše uvedené třídy obsahují informaci o typu obsluhované stanice. Na základě této informace je vypočtena cena služby.

### 2.2.10 Událost

Událost představuje položku, která určitým způsobem zatěžuje servery společnosti. Jedná se například o výpočty vedoucí k odeslání SMS. Cenu události určuje:

- potřebný procesorový čas
- typ události

### 2.2.11 SIM

Společnost Fiedler AMS poskytuje SIM karty k dlouhodobému zapůjčení. Cenu služby za určité časové období určují následující hodnoty:

- typ SIM
- tarif SIM karty

### 2.2.12 Widget

Widget je interaktivní prvek pro zobrazení dat. Společnost Fiedler AMS nabízí vytvoření widgetu na základě specifikace klienta. Takto vytvořené widgety jsou poté použity v uživatelských obrazovkách klienta. Cenu widgetu za určité časové období určuje kategorie widgetu.

### 2.2.13 Vzdálený widget

Vzdálený widget je statické HTML odesláné přes internet přímo do zařízení zákazníka. Cenu určuje opět kategorie widgetu, které se liší od kategorií klasického widgetu. V současné době je jedinou kategorií vzdáleného widgetu metereologický přehled.

### 2.2.14 SCADA obrazovka

Společnost Fiedler AMS nabízí návrh a následný pronájem systému SCADA. Cenu SCADA obrazovky za určité časové období určují následující hodnoty:

- počet stanic ve SCADA obrazovce
- počet senzorů zobrazených ve SCADA obrazovce
- počet prvků ve SCADA obrazovce
- počet zobrazení SCADA obrazovky
- množství dat zpracovaných SCADA obrazovkou
- počet servisních zásahů

### 2.2.15 Online servis SCADA obrazovky

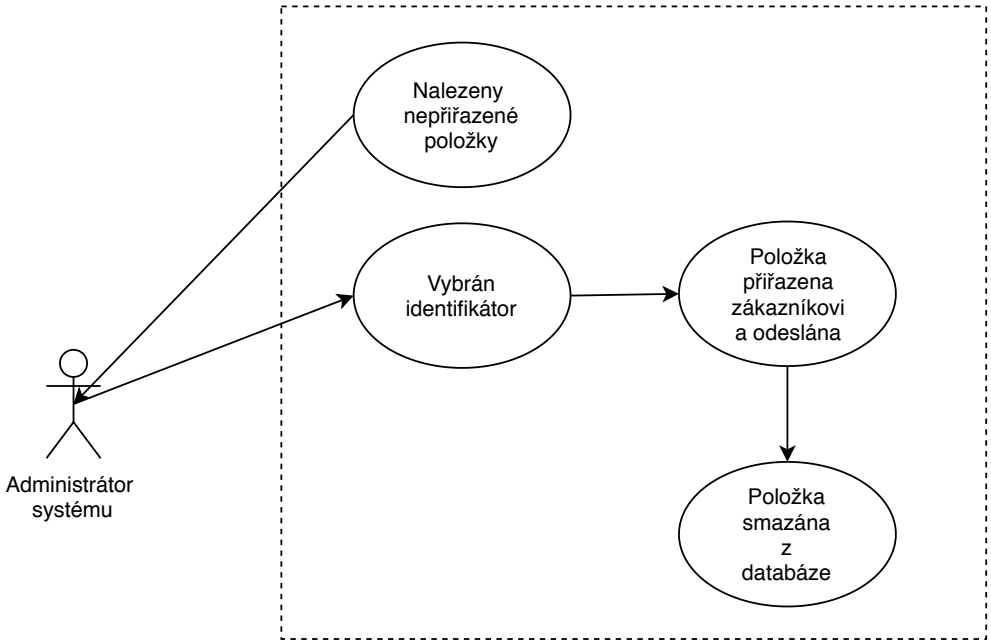
SCADA obrazovky je nutné udržovat případně upravovat dle přání klientů. Cenu servisu SCADA obrazovky určuje doba práce zaměstnance.

## 2.3 Příklady použití systému

Následující část se zabývá popisem vybraných příkladů použití systému z pohledu uživatele. Uživatelem systému je administrátor účetního systému, mezi jeho úkoly patří především správa množiny ceníků a jejich přiřazení zákazníkům. Dále se jedná o řešení situací kdy selhalo automatické přiřazení položky zákazníkovi a export výsledných dat v určitém formátu.

### 2.3.1 Manuální přiřazení položky zákazníkovi

Systém přiřazuje položky zákazníkům automaticky. K selhání přiřazení může dojít ve dvou případech - identifikátor stanice v položce je špatně nebo není přiřazen žádnému zákazníkovi. Pokud automatické přiřazení selže je nepřiřazená položka uložena do databáze. Systém umožňuje tyto položky manuálně přiřadit zákazníkovi pomocí REST API. Na obrázku 2.1 je zobrazeno schéma tohoto příkladu použití.



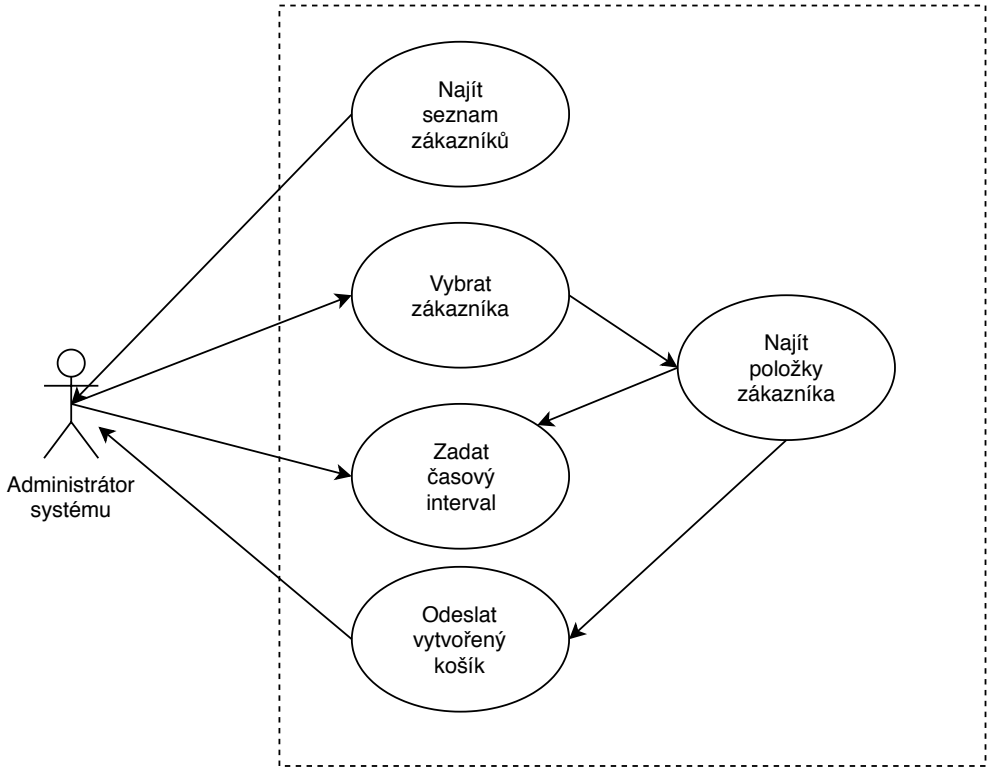

Obrázek 2.1: Manuální přiřazení položky zákazníkovi

Systém odešle kolekci nepřiřazených položek. Administrátor účetního systému vybere položku, kterou chce přiřadit. Následně vybere identifikátor stanice, který má položka obsahovat případně vybere zákazníka jemuž položka náleží. Na server je odesláno id položky a identifikátor stanice případně id zákazníka. Server tyto informace zpracuje -

opraví identifikátor v položce nebo přiřadí identifikátor z položky zákazníkovi. Poté je položka vyřazena z databáze nepřiřazených položek. Výsledkem tohoto případu použití je tedy položka přiřazená konkrétnímu zákazníkovi.

### 2.3.2 Získání košíku určitého zákazníka

Následující případ použití popisuje získání množiny položek určitého zákazníka, které zatím nemají vypočtenou cenu, za určité období. Předpokládá se existence položek, čekající na nacenění, které jsou přiřazeny vyhledanému uživateli. Na obrázku 2.2 je zobrazeno schéma popisovaného případu použití.



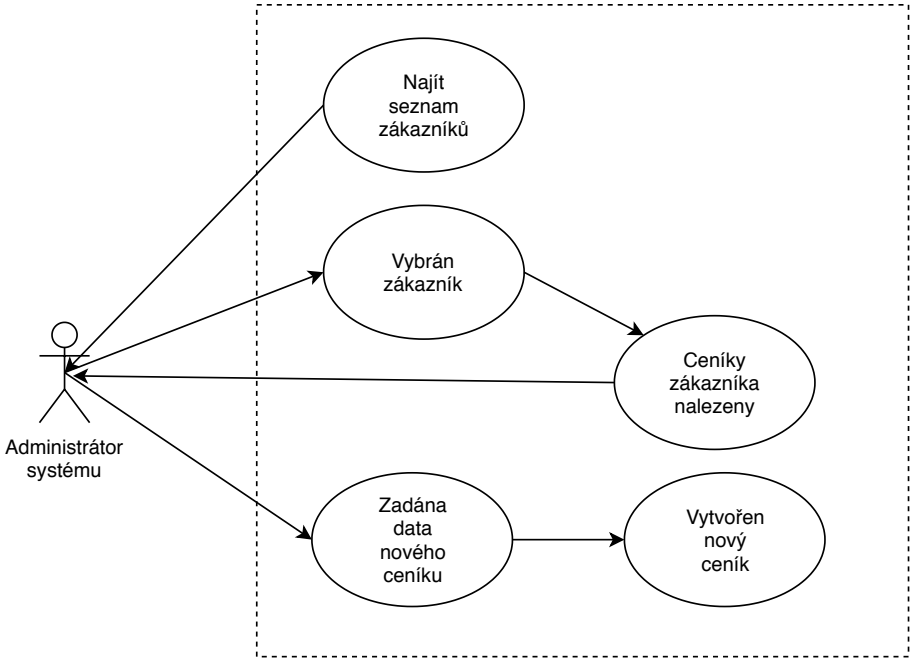

Obrázek 2.2: Získání košíku určitého zákazníka

Ze serveru je načten seznam zákazníků. Administrátor vybere zákazníka jehož položky chce zobrazit. Dále zadá časové rozmezí ve kterém vybírané položky vznikly, případně typ položek. Tento požadavek odešle na server. Z databáze jsou vybrány položky, které byly přiřazeny zákazníkovi, ale ještě nebyla vypočtena jejich cena. Id zákazníka, datum vytvoření položky a případně typ položky musí odpovídat požadavku od uživatele. Vybrané položky jsou vloženy do objektu označovaném jako "košík"a odeslány pomocí

REST API. Administrátor tedy získává objekt sdružující položky stejného zákazníka za určité časové období. Tyto data umožňují administrátorovi systému získat přehled pro jaké položky bude nutné v budoucnu určit cenu.

### 2.3.3 Vytvoření nového ceníku pro určitého zákazníka

Systém umožňuje spravovat množinu ceníků, tento případ použití popisuje vytvoření nového ceníku pro určitého zákazníka. Vytvořený ceník je použit k výpočtu ceny příchozích položek. Na obrázku 2.3 je zobrazeno schéma popisovaného případu.

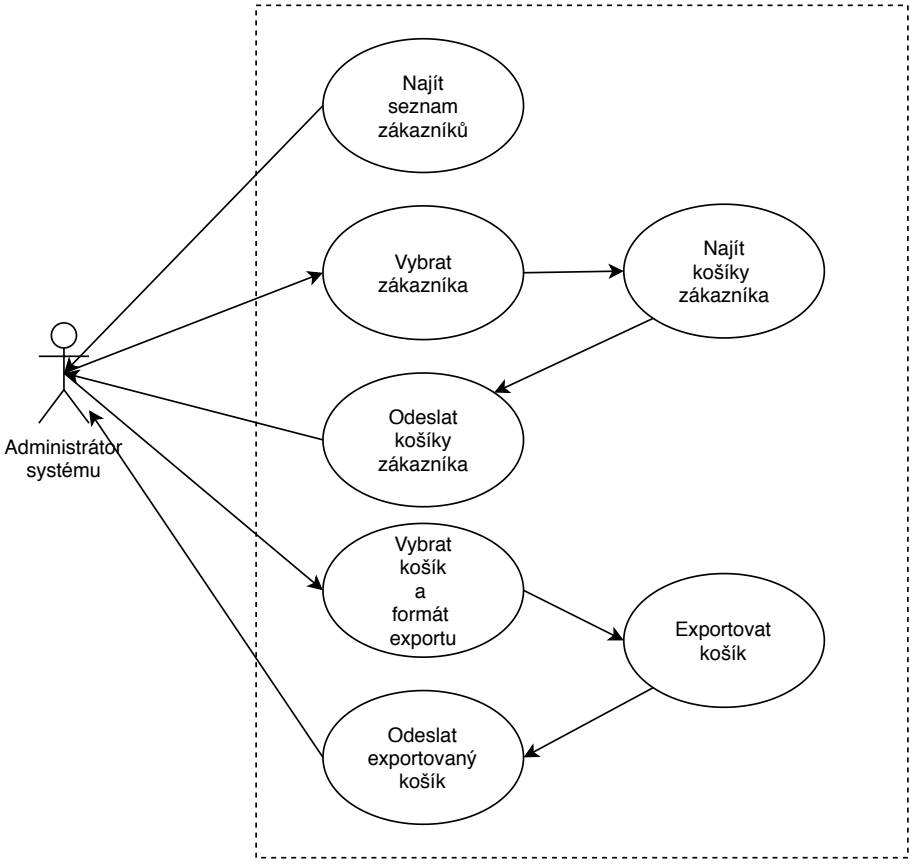



Obrázek 2.3: Vytvoření nového ceníku

Ze serveru je načten seznam zákazníků. Administrátor vybere zákazníka pro kterého chce vytvořit nový ceník. Ze serveru jsou načteny ceníky vybraného zákazníka. Administrátor vybere pro jaký typ položek chce vytvořit ceník a zadá další parametry ceníku, poté potvrdí vytvoření ceníku. Data jsou odeslána na server kde je zkontrolováno zda obsahují parametry odpovídající ceníku pro požadovaný typ položek. Dále je kontrolováno zda zákazník již nemá přiřazený ceník pro tento typ služeb. Následně je vytvořen ceník, který je přiřazen vybranému zákazníkovi a uložen do databáze. Výsledkem tohoto případu použití je vytvořený ceník, který je použit k výpočtu ceny položek určitého typu náležící konkrétnímu zákazníkovi.

### 2.3.4 Export košíku zákazníka s vypočtenou cenou

Tento případ použití popisuje export košíku s vypočtenou cenou v určitém formátu. Na obrázku 2.4 je zobrazeno jeho schéma.




Obrázek 2.4: Export košíku zákazníka

Ze serveru je načten seznam zákazníků. Administrátor vybere zákazníka jehož košíky chce zobrazit. Ze serveru je načten seznam košíků vybraného zákazníka. Dále administrátor vybere košík, který si přeje exportovat a formát exportu. Na server je odesláno id košíku a typ exportu. Z databáze je vybrán košík s odpovídajícím id a převeden do vybraného formátu. Výsledná data jsou odeslána na stranu klienta. Administrátor tedy získá data košíku s vypočtenou cenou v určitém formátu.

# Kapitola 3

# Návrh architektury systému

Tato kapitola pojednává o návrhu architektury celého systému. Nejprve je popsána obecná architektura systému a jeho rozdělení na jednotlivé části. Dále je popsána struktura dat účtované položky, způsob sdílení kódu a komunikace mezi jednotlivými servery. Následně je čtenář seznámen s funkcí jednotlivých serverů a problémy, které řeší.

Následující pasáž o rozsahu 28 listů obsahuje utajované skutečnosti a je obsažena pouze v archivovaném originále diplomové práce uložené na Přírodovědecké fakultě JU.

# Kapitola 4

# Implementace

Tato kapitola se zabývá implementací jednotlivých serverů celého systému. V první části jsou popsány technologie použité při tvorbě systému. Dále je postupně popsána tvorba serverů systému. U každého serveru je popsán způsob řešení problémů specifických pro daný server.

## 4.1 Použité technologie

Tato část se zabývá představením sady technologií, která bude použita k tvorbě systému. Jednotlivé technologie byly vybrány na základě požadavků firmy Fiedler AMS a předpokládaného využití v rámci aplikace s přihlédnutím na zkušenosti autora této práce.

### 4.1.1 NPM

Vytvářený systém obsahuje velké množství závislostí, je proto vhodné využít nástroj pro správu těchto závislostí. Nástroje pro správu závislostí, neboli package managers, slouží k instalaci a správě závislostí v rámci systému. Tento nástroj umožňuje vývojářům sdílet kód, který vytvořili k řešení specifického problému. Tyto části znovupoužitelného kódu se nazývají balíčky (package) nebo moduly. Balíček je obyčejná složka s jedním nebo více soubory, obsahující soubor popisující balíček [4].

Základní myšlenkou je vytvoření malých stavební bloků (balíčků), které řeší jeden problém a řeší jej dobře [3]. Tato myšlenka umožňuje vývoj rozsáhlých aplikací sestavováním těchto malých, sdílených stavebních bloků.

NPM neboli Node Package Manager je nástroj na správu závislostí pro Node.js. NPM je součástí instalace Node.js a používá soubor označovaný jako `package.json`. Tento soubor definuje závislosti a obsahuje informace o balíčku. Příklad souboru `package.json` je zobrazen na 4.1.

```
{
  "name": "MyPackage",
  "version": "1.0.0",
  "dependencies": {
    "my_dep": "^1.0.0",
    "tslib": "^1.9.3"
  },
  "main": "main.js",
  "scripts": {
    "build": "npm build",
    "start": "npm build && npm start",
    "debug": "npm build && npm start --debug",
    "dist": "npm build && npm dist"
  },
  "keywords": [],
  "author": "jakub_vacek",
  "license": "ISC",
  "repository": {
    "type": "git",
    "url": "https://github.com/jakub_vacek/my_package.git"
  },
  "homepage": "https://github.com/jakub_vacek/my_package"
}
```

Listing 4.1: Soubor package.json

NPM značně zjednodušuje správu závislostí případně předávání kódu mezi jednotlivými členy týmu. Při tvorbě rozsáhlé aplikace v Node.js, spoléhající na velké množství závislostí je využívání NPM v podstatě nutností.

### 4.1.2 TypeScript

TypeScript je open source nadstavba jazyka JavaScript, která byla vytvořena firmou Microsoft v roce 2004 [1]. Typescript doplňuje JavaScript o vlastnosti známé z objektově orientovaných jazyků, dále umožňuje využít nejnovější vlastnosti jazyka JavaScript jako jsou asynchronní funkce a zároveň tento kód kompilovat podle starších verzí standardu EcmaScript [2].
TypeScript lze nainstalovat pomocí NPM příkazem `npm install -g typescript`. Soubor `název_souboru.ts` lze přeložit pomocí příkazu `tsc název_souboru.ts`. Výsledkem je soubor `název_souboru.js` obsahující pouze JavaScript. Následuje stručný popis základních vlastností jazyka TypeScript.

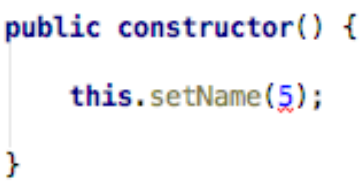
#### 4.1.2.1 Anotace typů

Anotace typů umožňuje zaznamenat zamýšlený návratový typ funkce nebo typ proměnné. Na příkladu 4.2 je zobrazeno využití anotace typů, kód se oproti JavaScriptu liší pouze v hlavičce funkce (`name: string`).

```
function setNamei(name: string) {
    this.name = name;
}
```

Listing 4.2: Anotace typu v jazyce TypeScript

Pokud vývojář zkusí do proměnné `name` přiřadit číselnou hodnotu IDE jej upozorní, že se snaží do proměnné typu String přiřadit číslo (viz. obrázek 4.1).Anotace typů se při kompilaci do JavaScriptu ztrácí, kompilace tedy proběhne bez problémů[2]. Upozornění tedy vývojáře informuje o tom, že vytvořený kód pravděpodobně nebude fungovat tak jak zamýšlel.




Obrázek 4.1: Statická typová kontrola

#### 4.1.2.2 Rozhraní

Rozhraní v TypeScriptu může být použito stejně jako v jiných objektově orientovaných jazycích, ale také k definování struktury objektu [1]. Na příkladu 4.3 je rozhraní použito k definování objektu, který má atributy `id` typu number a `userName` typu string.

```
interface User {
    id: number;
    userName: string;
}
function toString(user: User) {
    return user.userName + user.id;
}
var user = { id: 0,  userName: "testUser" };
document.body.innerHTML = toString(user);
```

Listing 4.3: Rozhraní v jazyce TypeScript

V jazyce TypeScript jsou považovány dva typy za kompatibilní pokud je kompatibilní jejich vnitřní struktura [2], to umožňuje implementovat rozhraní bez nutnosti použít klíčové slovo `implements` [4].

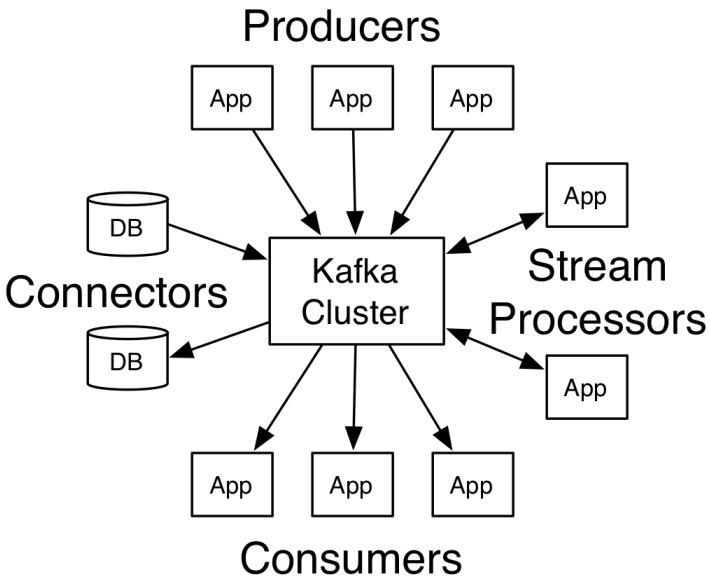
Především statická typová kontrola je ve spojení s linterem (například s TsLint [5]) obzvláště užitečná při vývoji rozsáhlých JavaScript aplikací. To je také důvod proč byla tato technologie zvolena pro vývoj popisovaného systému.

### 4.1.3 Apache Kafka

Apache Kafka je open source distribuovaná streamovací platforma, která poskytuje následující funkce [6]:

- Vytváření a přihlašování se k odběru datových proudů
- Ukládání dat způsobem tolerantním k chybám
- Zpracování datových proudů v okamžik jejich vytvoření

Na obrázku 4.2 je zobrazena struktura systému Apache Kafka, který obsahuje 4 základní API. Jednotlivá API jsou popsána níže.




Obrázek 4.2: API v systému Apache Kafka, převzato z [6]

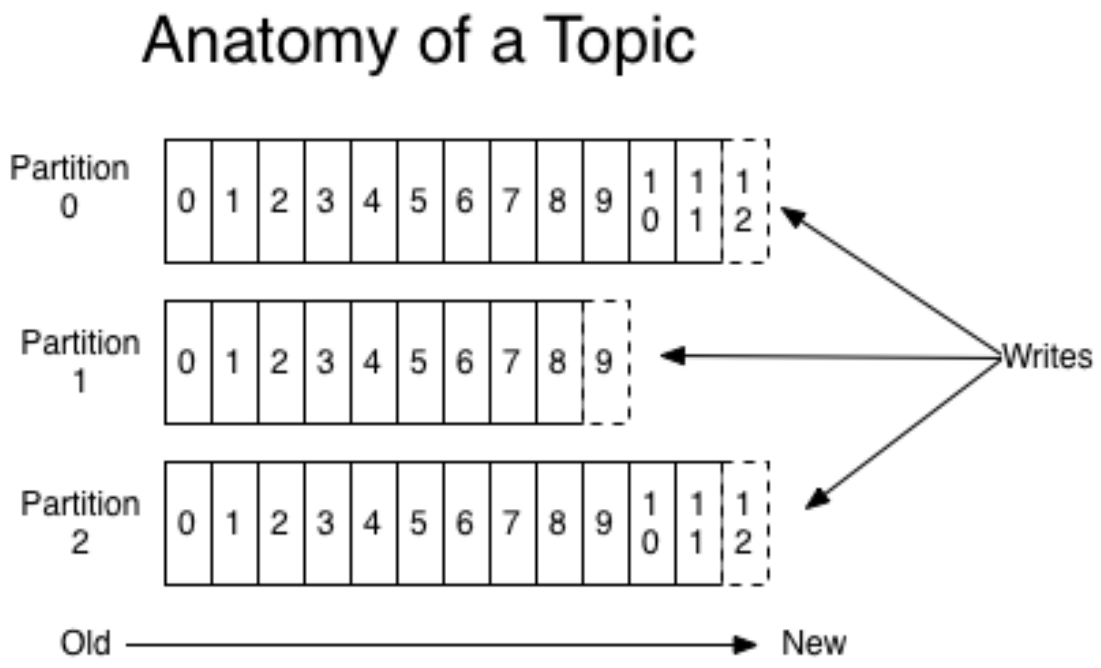
- `Producer API` umožňuje aplikaci vytvářet záznamy v jednom nebo více datových proudech.

- `Consumer API` umožňuje aplikaci přihlásit se k odběru a zpracovávat záznamy v jednom nebo více datových proudech.

- `Streams API` umožňuje aplikaci konzumovat příchozí proud, případně proudy, dat a tento proud přetvářet na výstupní proud dat.

- `Connector API` umožňuje vytvářet a spouštět znovupoužitelné producenty a konzumenty, které se připojují k existující aplikaci. Příkladem může být konektor k relační databázi, který zaznamená každou změnu určité tabulky.

Apache Kafka se spouští jako cluster na jednom nebo více serverech. Cluster obsahuje datový proud záznamů v kategoriích, které se nazývají `topic`. Každý záznam obsahuje klíč, hodnotu a časové razítko [6].
Topic je kategorie do které se ukládají záznamy. K odběru jednotlivých topiců se může přihlásit více aplikací (využívající Consumer API) [6]. Topicy lze dále dělit na části zvané `partition`. Na obrázku 4.3 je zobrazena struktura topicu obsahující 3 partition.




Obrázek 4.3: Struktura topicu, převzato z [6]

Partition je seřazená, neměnná sekvence záznamu, do které se postupně přidávají další záznamy. Každý záznam je identifikován unikátním (v rámci konkrétního partition) sekvenčním číslem zvaným `offset`. Hodnota offsetu je ukládána na straně aplikací, které data zpracovávají. Kafka ukládá publikované záznamy po určitou dobu, kterou lze nastavit při konfiguraci clusteru. Po uplynutí této doby jsou záznamy smazány pro uvolnění místa [6].

Systém Apache Kafka je v popisovaném systému využíván k načítání dat ze serverů jednotlivých služeb a komunikaci serverů mezi sebou. Při lokálním vývoji byl použit Docker [7] s image kafka-docker [9].

### 4.1.4 Nest

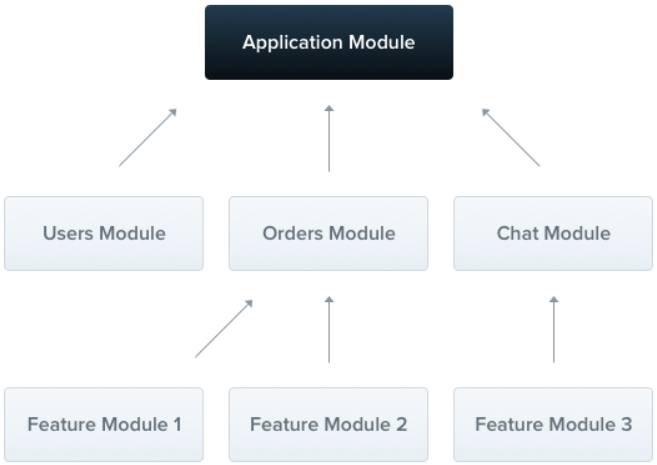
Nest je framework zaměřený na tvorbu Node.js serverových aplikací, který podporuje požívání jazyka TypeScript. Framework využívá HTTP framework Express případně Fastify a také některé funkce OOP jako je například dependency injection. Nest představuje objekt zvaný Modul, pomocí těchto objektů framework usnadňuje návrh architektonicky správně rozdělených aplikací. Níže jsou popsány vybrané vlastnosti frameworku Nest [12].

#### 4.1.4.1 Modules

Modul je třída označená dekorátorem `@Module()`. Tento dekorátor umožňuje frameworku správně uspořádat strukturu aplikace. Vstupem dekorátoru `@Module()` je objekt obsahující následující prvky:

- `Providers` množina tříd, která je automaticky vytvořená při vytvoření modulu a sdílená uvnitř modulu.
- `Controllers` množina tříd označená dekorátorem `@Controller` zpracovávající HTTP požadavky.
- `Imports` množina tříd, které jsou importovány z jiných modulů.
- `Exports` podmnožina `Providers`, které jsou exportovány z daného modulu.

Každá aplikace obsahuje alespoň jeden modul zvaný root module. Od tohoto modulu Nest sestaví aplikační graf. Aplikační graf je vnitřní datová struktura, pomocí které framework určuje vztahy mezi jednotlivými moduly a jejich částmi [12]. Na obrázku 4.4 je zobrazena ukázka aplikačního grafu.




Obrázek 4.4: Aplikační graf, převzato z [12]

Na příkladu 4.4 je zobrazena třída `ExportPdfModule` zajišťují export dat košíku do PDF. Modul exportuje třídu `ExportPdfService` zpracovávající export dat. Tuto třídu mohou importovat jiné moduly a využívat její metody. Modul dále využívá třídu `TableHelper` zajišťující vykreslení košíku do PDF dokumentu.

```
@Module({
    exports: [ExportPdfService],
    providers: [ExportPdfService, TableHelper]
})
export class ExportPdfModule {}
```

Listing 4.4: Modul ExportModule

#### 4.1.4.2 Controllers

Controller je třída označená dekorátorem `@Controller` zajišťující zpracování HTTP požadavků. Dekorátor `@Controller` přijímá jako parametr řetězec označující prefix URL [12]. To umožňuje sdružovat cesty týkající se stejné entity. Na 4.5 je uveden přiklad třídy s dekorátorem `@Controller`.

```
@Controller(CustomerController.PATH)
export class CustomerController {
    @Get('/')
    public async find(@RequestMetadata() meta: HttpRequest, @Query
       ('fetchPackage', ParseStringPipe) fetchPackage?: boolean):
       Promise<Customer[]> {
        return this.service.getMany(meta, fetchPackage);
    }
    @Post('/')
    public async create(@Body() customerData: ValidCustomer,
       @RequestMetadata() meta: HttpRequest): Promise<void> {
        return this.service.create(customerData, meta);
    }
    @Delete('/:customerId')
    public async delete(@Param('customerId', ParseIntPipe)
       customerId: number, @RequestMetadata() meta: HttpRequest):
       Promise<void> {
        return this.service.delete(customerId, meta);
    }
    @Put('/')
    public async update(@Body() customerData: ValidCustomer,
       @RequestMetadata() meta: HttpRequest ): Promise<void> {
        return this.service.update(customerData, meta);
    }
}
```

Listing 4.5: Třída CustomerController

Třída `CustomerController` zajišťuje zpracování požadavků, které se týkají zákazníků. Každá metoda této třídy je označena dekorátorem určující HTTP metodu dotazu a dále specifikující URL dotazu. Ukázka dále obsahuje dekorátor `@param` označující parameter v URL dotazu a také dekorátor `@body` specifikující data dotazu.

#### 4.1.4.3 Pipes

Pipe je třída opatřena dekorátorem `@Injectable` implementující rozhraní `PipeTransform` [12]. Pipe má dvě typická použití: transformaci přichozích dat a kontrolu správnosti příchozích dat. Pipe působí jako prostředník mezi samotným HTTP požadavkem a metodou v třídě označené dekorátorem `@Controller`. Framework předá příchozí data Pipe a po validaci, případně transformaci data pokračují dále. Pokud je při zpracování dat v Pipe vyhozena výjimka data dále nepokračují a klientovy je odesláno hlášení o chybě. Na 4.6 je zobrazena třída implementující rozhraní `PipeTransform`.

```
@Injectable()
export class FeatureValidationPipe implements PipeTransform {
    public async transform(value: Feature): Promise<Feature> {
        const errors: ErrorObject[] | null = SchemaValidator.
           validate(Feature, value);
        if (errors !== null) {
            throw new BadRequestException('Validation failed');
        }
        return value;
    }
}
```

Listing 4.6: Třída FeatureValidationPipe

Třída obsahuje metodu `transform`, kterou vyžaduje rozhraní `PipeTransform`. Tato metoda zajišťuje validaci příchozích dat. Použití validace v praxi je popsáno v **??**.

### 4.1.5 TypeORM

TypeORM je nástroj pro objektově relační mapování navržený pro použití s Node.js s využitím jazyka JavaScript nebo TypeScript [14]. Následuje popis vlastností frameworku TypeORM použitých při tvorbě popisovaného systému.

#### 4.1.5.1 Entity

Entita je označení pro třídu, která je mapována na databázovou tabulku [14]. Na 4.7 je zobrazena třída `AssignedItem` představující model pro TypeORM.

```
@Entity({name: 'assigned_items'})
export class AssignedItem {
@PrimaryGeneratedColumn({name: 'id', type: 'integer'})
    public id!: number;
    @Column({name: 'type', type: 'integer'})
    public type: EItemType;
    @Index('location_id_idx')
    @Column({name: 'location_id', length: 36})
    public readonly locationId: string;
    @Column({ name: 'created', type: 'datetime', precision: 3})
    public readonly created: Date;
    @Column({ name: 'data', type: 'simple-json'})
    public readonly data: TTopicData;
    @ManyToOne(() => CustomerData, (customer: CustomerData) =>
       customer.items, { cascade: true })
    public readonly customerData: CustomerData;
    }
```

Listing 4.7: Třída AssignedItem

Třída `AssignedItem` je opatřena anotací `@Entity` označující třídu jako entitu pro ukládání a vyčítání z databáze. Instanční proměnná `itemId` je označena dekorátorem `@PrimaryGeneratedColumn`. Tento dekorátor označuje primární klíč dané entity a zajistí automatické generování hodnoty proměnné. Další proměnné třídy jsou opatřeny dekorátorem `@Column` určující sloupce tabulky a specifikující typ dat uložených v jednotlivých sloupcích.

Dekorátor `@ManyToOne` označuje vztah mezi tabulkami, v tomto případě instance třídy `AssignedItem` obsahuje jednu instanci třídy `CustomerData`. Na 4.8 je zobrazen stejný vztah z pohledu třídy `CustomerData`. Instance třídy `CustomerData` obsahuje více instancí třídy `AssignedItem`.

```
@OneToMany(() => AssignedItem, (item: AssignedItem) => item.
    customerData)
public readonly items!: AssignedItem[];
```

Listing 4.8: Vztah OneToMany

Pro použití entity je nutné ji nejprve registrovat v konfiguraci TypeORM. NestJs doporučuje použití TypeORM, z tohoto důvodu je konfigurace velmi jednoduchá. Na 4.9 je zobrazen root modul aplikace, která využívá NestJs a TypeORM.

```
@Module({
    imports: [TypeOrmModule.forRoot({
        type: 'mysql',
        host: Configuration.host,
        port: Configuration.port,
        username: Configuration.username,
        password: Configuration.password,
        database: Configuration.database,
        entities: Configuration.entities,
        synchronize: Configuration.synchronize
    })]
})
```

Listing 4.9: Konfigurace TypeORM

Root modul aplikace pouze importuje nastavení databáze a TypeORM. Proměnná `entities` obsahuje cesty k sestaveným entitám. V případě třídy `AssignedItem` cesta vypadá například takto `build/module/dataItem/model/AssignedItem.js`

#### 4.1.5.2 Repository

Repository je objekt, který poskytuje metody pro ukládání a vyčítání jedné entity. NestJS umožňuje vkládat repository pomocí dependency injection. Toho lze docílít použitím dekorátoru `@InjectRepository` s názvem třídy v parametru. Na 4.10 je zobrazeno vložení a použití Repository.

```
@InjectRepository(AssignedItem)
private readonly itemRepository!: Repository<AssignedItem>;

public async create(item: AssignedItem): Promise<void> {
        await this.itemRepository.save(item);
}
public async findById(itemId: number): Promise<AssignedItem> {
   let query: SelectQueryBuilder<AssignedItem>;
   query = this.topicDataItemRepository.createQueryBuilder('item')
      .innerJoinAndSelect('item.customerData', 'customerData')
      .where('itemId = :id', {id: itemId});
   return query.getOne();
}
```

Listing 4.10: Použití Repository

Metoda `create` ukládá intanci třídy `AssignedItem` a ukazuje základní použití Repository. Metoda `findById` vyčítá instanci třídy `AssignedItem` s daným id pomocí objektu `SelectQueryBuilder`. Nejprve je pomocí Repository získán objekt `SelectQueryBuilder` sloužící k tvorbě složitějších dotazů. Následně je vytvořen dotaz, který načte instanci třídy `AssignedItem` včetně dat týkajících se zákazníka. Na závěr je dotaz vykonán.

Následující pasáž o rozsahu 14 listů obsahuje utajované skutečnosti a je obsažena pouze v archivovaném originále diplomové práce uložené na Přírodovědecké fakultě JU.

# Kapitola 5

# Testování systému

V předchozí části práce byl čtenář seznámen s vývojem systému. Následující část se věnuje testování systému. Testování kontroluje zda výsledný kód funguje dle návrhu. Dále, po implementaci případného rozšíření sytému, ověřuje správnou funkci již implementovaných funkcí. Dalším důvodem pro testování je skutečnost, že vytvořený systém pracuje s finančními prostředky. Je tedy nutné ověřit zda nedochází k chybám, které ovlivňují výpočet ceny jednotlivých služeb. Nejprve jsou stručně popsány použité technologie a následně je popsáno testování pomocí unit a integračních testů.

## 5.1 Použité technologie

### 5.1.1 AVA

NPM balíček AVA je test runner pro testování Node.js aplikací [19]. Instalace probíhá pomocí příkazu `NPM install ava`. Následně je nutné přidat do souboru `package.json` script označený `test`, který zajistí spuštění testování pomocí AVA. Další nastavení je možné přidat do `package.json`. Samotné testování se poté spouští pomocí příkazu `npm test`. Jednoduché použití AVA je zobrazeno na 5.1.

```
test('should be bar', async t => {
  const result: string = getBar();
  t.is(result, 'bar');
});
```

Listing 5.1: Testování pomocí AVA

Test představuje funkci `test` z balíčku AVA. Parametrem této funkce je popis testu, který musí být, v souboru obsahující test, unikátní. Dalším parametrem je samotná implementace testu. V příkladu je testováno zda se výsledná hodnota metody `getBar` rovná řetězci `bar`. Další příklady použití AVA jsou zobrazeny v 5.2 a 5.3. Balíček AVA byl zvolen pro svou podporu jazyka TypeScript, možností testovat objekty typu `Promise` a přehledný výstup

### 5.1.2 NYC

NYC je nástroj umožňující zjistit jak dobře testy pokrývají testovaný kód [20]. Instalace opět probíhá pomocí NPM. Spouštění NYC lze provést přidáním výrazu `nyc` do skriptu test v souboru `package.json` jak je zobrazeno na 5.2.

```
"scripts": {
    "test": "nyc ava",
  }
```

Listing 5.2: Použití NYC

Výsledek použití NYC je zobrazen na obrázku 5.1.

```
-----------------------------|----------|----------|----------|----------|-------------------|
File                         |  % Stmts | % Branch |  % Funcs |  % Lines | Uncovered Line #s |
-----------------------------|----------|----------|----------|----------|-------------------|
All files                    |    90.91 |    83.33 |      100 |    88.89 |                   |
 parse-string-pipe.service.ts |    90.91 |    83.33 |      100 |    88.89 |                17 |
-----------------------------|----------|----------|----------|----------|-------------------|
```

Obrázek 5.1: Report pokrytí vytvořený NYC

Na obrázku je zobrazeno pokrytí souboru `parse-string-pipe.service` testy. Jasně ukazuje nedostatečné pokrytí souboru, včetně čísel řádků, který nejsou dostatečně otestovány. To vývojáři umožňuje upravit testy a dosáhnout dostatečného pokrytí.

## 5.2 Unit testy

Unit testy slouží k testování nejmenších funkčních jednotek kódu, což jsou většinou třídy. Testování jednotlivých funkčních jednotek umožňuje včasné odhalení případných chyb [21]. Na 5.3 je zobrazena, pro přehlednost zkrácená, třída `CustomerService`.

```
@Injectable()
export class CustomerService {
    public constructor(private readonly service:
       CustomerPersistanceService) {}

    public async getOne(customerId: number, requestMetadata:
       HttpRequest): Promise<Customer> {
        const customer: Customer | undefined = await this.service.
           findById(customerId);
        if (!customer) throw new NotFoundException();
        return customer;
    }
}
```

Listing 5.3: Třída CustomerService

Třída slouží ke zpracování požadavků na získání dat zákazníka. Konkrétně k získání zákazníka s určitým id. Jedinou závislostí je třída `CustomerPersistanceService` sloužící k vyčítání dat týkající se zákazníka z databáze. Instance této třídy je do instance třídy `CustomerService` vložena pomocí dependency injection. Druhý parametr metody `getOne requestMetadata` je použit k logování zpracovaných požadavků. Toto logování je pro přehlednost vynecháno. Pro otestování této třídy je nutné vytvořit třídu, která nahradí třídu `CustomerPersistanceService` v instanci testované třídy. Třída `CustomerPersistanceServiceStub` je zobrazena na 5.4.

```
class CustomerPersistanceServiceStub {
    public async findById(customerId: number): Promise<Customer |
       undefined> {
        if (customerId === 1) return undefined;
        return new Customer(customerId, 'test', 'test');
    }
}
```

Listing 5.4: Třída CustomerPersistanceServiceStub

Třída obsahuje jedinou metodu vracející, v závislosti na hodnotě parametru `customerId`, testovací data nebo hodnotu `undefined`. Hodnota `undefined` značí, že zákazník nebyl v databázi nalezen. Instance této třídy je následně použita místo instance třídy `CustomerPersistanceService` v samotném testu, který je uveden na 5.5, z testu je vyjmuto testování metody pro získání všech zákazníků.

```
before(async () => {
    let service: CustomerService = new CustomerService( <
       CustomerPersistanceService> new
       CustomerPersistanceServiceStub() );
});
test('get one customer - should be found', async (t:
   ExecutionContext) => {
    const data: Customer = await service.getOne(2, new HttpRequest
       (new HttpMetadata(<any>{})));
    t.deepEqual(data, new Customer(2, 'test', 'test'));
});
test('get one customer - should not be found', async (t:
   ExecutionContext) => {
    const error: NotFoundException =  await t.throwsAsync(async ()
        => {
        await service.getOne(1, new HttpRequest(new HttpMetadata(<
           any>{})));
    });
    t.deepEqual(error, new NotFoundException());
});
```

Listing 5.5: Unit test třídy CustomerService

Nejprve je pomocí metody `before` z balíčku AVA vytvořena instance třídy `CustomerService` s testovací třídou `CustomerPersistanceServiceStub`. AVA zajistí vykonání metody `before` před samotným testováním. Následně je testováno zda metoda `getOne` vrací očekávaná data zákazníka případně chybové hlášení.

Tímto způsobem byly testovány třídy zajišťující hlavní funkce jednotlivých serverů. Tedy třídy zajišťující vyčítání dat z databází, přiřazení položky zákazníkovy a výpočet ceny položky.

## 5.3 Integrační testy

Integrační testování označuje proces, ve kterém jsou jednotlivé části kódu propojeny a testovány k zajištění správné funkce celého systému [22]. Při integračních testech je testována funkce serveru od vstupu serveru (třída zpracovávající požadavky uživatele případně příchozí data ze systému Apache Kafka) až k perzistentní vrstvě systému. Jednoduchým příkladem může být test získání zákazníka s určitým id uvedený na 5.6.

```
before(async () => {
    customerController = new CustomerController(new
       CustomerService(<CustomerPersistanceService> new
       CustomerPersistanceServiceStub()) );
});
test('find one customer - should be found', async (t:
   ExecutionContext) => {
    const data: Customer = await customerController.findById(2,
       new HttpRequest(new HttpMetadata(<any>{})));
    t.deepEqual(data, new Customer(2, 'test', 'test'));
});
test('find one customer - should not be found', async (t:
   ExecutionContext) => {
    const error: NotFoundException =  await t.throwsAsync(async ()
        => {
        await customerController.findById(1, new HttpRequest(new
           HttpMetadata(<any>{})));
    });
    t.deepEqual(error, new NotFoundException());
});
```

Listing 5.6: Integrační test třídy CustomerService

Nejprve je vytvořena instance třídy `CustomerController` do které je vložena instance třídy `CustomerService` obsahující instanci třídy `CustomerPersistanceServiceStub` (zobrazena na 5.4). Následně je testováno zda třída `CustomerController` vrací očekávaná data zákazníka případně chybové hlášení. Integrační testy pokrývají všechny endpointy jednotlivých serverů a třídy zpracovávající data obdržená systémem Apache Kafka.

## 5.4 Pokrytí kódu testy

V tabulce 5.1 je zobrazeno pokrytí kódu jednotlivých serverů testy. Nejvíce pozornosti bylo věnováno testování serveru Pricing Manager, který zajištuje výpočet ceny položek. Nižší pokrytí kódu u Billing Manager serveru je způsobeno vynecháním testování tříd, které zajištují vykreslení košíku s přiřazenou cenou do PDF dokumentu.

| **Jméno serveru** | **Počet testů** | **Pokrytí kódu testy(%)** |
|---|---|---|
| Classifier | 43 | 83 |
| Aggregator | 38 | 80 |
| Pricing Manager | 136 | 86 |
| Billing Manager | 43 | 70 |

Tabulka 5.1: Pokrytí kódu testy

# Kapitola 6

# Závěr

Hlavním cílem předkládané magisterské práce byl návrh a implementace systému ve společnosti Fiedler AMS. Tento cíl, sejně jako ostatní cíle ze zadání, se podařilo splnit.

Nejprve byla v kapitole 2 provedena analýza požadovaných funkcí systému. Následně bylo analýzováno celkem 15 služeb, které je nutné účtovat. Tato analýza se zaměřovala na identifikaci parametrů, které ovlivňují cenu jednotlivých služeb. Poté byl čtenář seznámen s případy použití vytvořeného systému.

Na základě provedené analýzy byla v kapitole 3 navržena architektura systému, která je založena na architektuře mikroslužeb a využívá proudového zpracování dat. Poté byl čtenář seznámen s návrhem struktury dat reprezentující data jednotlivých služeb, způsobem výpočtu jejich ceny a strukturou použitých ceníků. Následně se práce věnovala návrhu jednotlivých serverů. Nejprve bylo popsáno rozdělení jednotlivých serverů na moduly a struktura databáze každého serveru. V závěru kapitoly byly popsány funkce, které jednotlivé servery zajišťují a REST API umožňující práci s daty serveru.

Na začátku kapitoly 4 byl čtenář seznámen se sadou technologií, která byla použita pro tvorbu systému. Jednotlivé technologie byly vybrány na základě požadavků společnosti Fiedler AMS a předpokládaného použití, s přihlédnutím na zkušenosti autora této práce. Následně se kapitola 4 věnovala implementaci návrhu systému z předchozí kapitoly. Nejprve byly popsány NPM balíčky použité ke sdílení kódu mezi servery systému. Poté se práce zabývala implementací jednotlivých serverů, v pořadí, ve kterém servery procházejí zpracovávaná data. Úsek zabývající se Classifier serverem mimo přiřazení zákazníka řešil také komunikaci serverů pomocí systému Apache Kafka. V další části bylo popsáno pravidelné odesílání dat ze serveru. Následující úsek kapitoly byl věnován výpočtu ceny příchozích dat a také validaci jednotlivých ceníků. V závěru kapitoly byl popsán export naceněných dat do souboru ve formátu PDF.

V kapitole 5 byly stručně uvedeny důvody testování a popsány technologie, které byly použity při testování výsledného systému. Následně se kapitola zabývala samotným testováním pomocí unit testů. Na závěr bylo popsáno testování systému pomocí integračních testů.

Havním přínosem práce je vytvoření systému umožňující výpočet ceny služby na základě skutečně vynaložených prostředků na poskytnutí dané služby. Popisovaný systém zajistí spravedlivý výpočet ceny poskytovaných služeb a snížení zátěže účetního oddělení společnosti. Dále je možné předkládanou práci použít jako oporu při návrhu a implementaci systému, který je založen na architektuře mikroslužeb a využívá proudového zpracování dat. Případně při návrhu systému používající obdobné technologie. Práce může také sloužit jako inspirace při řešení validace složitých objektů.

Autorem práce byly zpracovány všechny etapy vedoucí k vytvoření popsaného systému. Zajímavá byla především aplikace výsledků analýz při návrhu systému. Práce také výrazně rozšířila rozhled autora v oblasti technologií použitých při implementaci a testování systému.

Vytvořený systém bude nasazen do prostředí společnosti Fiedler AMS a nadále rozvíjen. Na začátku kapitoly 2 je uveden výčet funkcí o které by bylo vhodné systém v budoucnu doplnit. Tyto funkce budou postupně přidávány. Dále bude vytvořena webová aplikace umožňující správu systému.

# Literatura


[1] FENTON, Steve. *Pro TypeScript: Application-Scale JavaScript Development*. 1. New York: Apress, 2014. ISBN 9781430267911.

[2] *TypeScript* [online]. Redmond: Microsoft, 2012 [cit. 2016-12-19]. Dostupné z: https://www.typescriptlang.org/index.html

[3] *NPM* [online]. Oakland: NPM, 2009 [cit. 2016-12-06]. Dostupné z: https://www.npmjs.com

[4] VACEK, Jakub. *Současný ekosystém Rich Internet Applications v HTML5*. České Budějovice, 2017. Bakalářská. The University of South Bohemia. Vedoucí práce Martin Čížek.

[5] *TSlint* [online]. Palo Alto: Palantir, 2013 [cit. 2019-01-21]. Dostupné z: https://palantir.github.io/tslint/

[6] *Apache Kafka* [online]. n: n, 2018 [cit. 2018-12-10]. Dostupné z: https://kafka.apache.org

[7] *Docker* [online]. San Francisko: Docker, 2013 [cit. 2019-01-29]. Dostupné z: https://www.docker.com/

[8] NPM - Kafka JS. *NPM* [online]. Oakland, California: npm.Inc, 2014 [cit. 2019-01-29]. Dostupné z: https://www.npmjs.com/package/kafkajs

[9] Wurstmeister/kafka. *Https://hub.docker.com* [online]. San Francisco: Docker, 2013 [cit. 2019-01-29]. Dostupné z: https://hub.docker.com/r/wurstmeister/kafka/

[10] *AJV* [online]. UK: AJV, 2015 [cit. 2019-06-25]. Dostupné z: https://ajv.js.org

[11] *PDFkit* [online]. San Francisco, 2014 [cit. 2019-06-25]. Dostupné z: http://pdfkit.org

[12] *NestJs* [online]. Kamil Mysliwiec, 2017 [cit. 2019-06-26]. Dostupné z: https://nestjs.com

[13] MYSLIWIEC, Kamil. Lifecycle Events. *NestJS* [online]. 2017 [cit. 2019-09-16]. Dostupné z: https://docs.nestjs.com/fundamentals/lifecycle-events

[14] *TypeORM* [online]. 2016 [cit. 2019-07-23]. Dostupné z: https://typeorm.io/

[15] Uživatelská příučka k telemetrické stainci H3 a H7. *Webové stránky společnosti Fiedler* [online]. České Budějovice: Fiedler, 2017 [cit. 2019-09-23]. Dostupné z: https://www.fiedler.company/sites/default/files/dokumenty/manual_h7_110b_all_0.pdf

[16] *Fiedler* [online]. České Budějovice: Fideler, 2017 [cit. 2019-09-23]. Dostupné z: https://www.fiedler.company/cs

[17] *Fiedler* [online]. České Budějovice:: České Budějovice, 2019 [cit. 2019-09-23]. Dostupné z: https://public02.fiedler.company/cs

[18] Ceník společnosti 2019. *Web spolčnosti Fiedler* [online]. Fiedler, 2017 [cit. 2019-10-06]. Dostupné z: https://www.fiedler.company/sites/default/files/dokumenty/cenik_2019_v4b.pdf

[19] *GitHub stránky AVA* [online]. online, 2018 [cit. 2019-10-07]. Dostupné z: https://github.com/avajs/ava

[20] *Nyc* [online]. online: Istanbul, 2015 [cit. 2019-10-07]. Dostupné z: https://istanbul.js.org

[21] HUIZINGA, Dorota a Adam KOLAWA. *Automated defect prevention: best practices in software management*. Hoboken, N.J.: IEEE Computer Society, c2007. p. 75. ISBN 978-0-470-04212-0.

[22] ISO/IEC/IEEE 24765:2010(E). *ISO/IEC/IEEE International Standard - Systems and software engineering*. online: ISO/IEC/IEEE, 2010.

[23] RICHARDSON, Chris. What are microservices? *Microservices.io* [online]. online: Chris Richardson, 2019 [cit. 2019-11-04]. Dostupné z: https://microservices.io/index.html

[24] RICHARDSON, Chris. Pattern: Microservice Architecture. *Microservices.io* [online]. online: online, 2019 [cit. 2019-03-05]. Dostupné z: https://microservices.io/patterns/microservices.html

[25] FOWLER, Martin. Microservices: a definition of this new architectural term. *MartinFowler.com* [online]. Chicago, IL: Martin Fowler, 2003, 2014 [cit. 2019-11-05]. Dostupné z: https://martinfowler.com/articles/microservices.html

[26] JAMSHIDI, Pooyan, Claus PAHL, Nabor C. MENDONCA, James LEWIS a Stefan TILKOV. Microservices: The Journey So Far and Challenges Ahead. In: *IEEE Software*. online: IEEE, 2018, **35**(3), s. 24-35. DOI: 10.1109/MS.2018.2141039. ISBN 1937-4194. ISSN 0740-7459. Dostupné z: https://ieeexplore.ieee.org/document/8354433/

[27] XIA, Chuanlong, Guangcan YU a Meng TANG. Efficient Implement of ORM (Object/Relational Mapping) Use in J2EE Framework: Hibernate. *2009 International Conference on Computational Intelligence and Software Engineering*. IEEE, 2009, 2009, **2019**, 1-3. DOI: 10.1109/CISE.2009.5365905. ISBN 978-1-4244-4507-3. Dostupné také z: http://ieeexplore.ieee.org/document/5365905/

[28] KREPS, Jay. The Log: What every software engineer should know about real-time data's unifying abstraction. *LinkedIn Engineering* [online]. online: LinkedIn, 2019, 16.12.2013 [cit. 2019-11-06]. Dostupné z: https://engineering.linkedin.com/distributed-systems/log-what-every-software-engineer-should-know-about-real-time-datas-unifying

[29] MAAS, Gerard a Francois GARILLOT. *Stream Processing with Apache Spark*. Sebastopol, CA: O-Reilly Media, 2019, 29 - 34. ISBN 978-93-5213-858-6.

[30] *GitHub stránky BigNumber.js* [online]. online: online, 2013 [cit. 2019-11-11]. Dostupné z: https://github.com/MikeMcl/bignumber.js

# Seznam obrázků

# Seznam úryvků kódu

# Seznam tabulek

# Příloha A

# Použitý software

**LaTeX** ⟨http://www.miktex.org/⟩

**TeXShop** ⟨http://pages.uoregon.edu/koch/texshop/⟩

**IntelliJ IDEA** ⟨https://www.jetbrains.com/idea/⟩

**Node.js** ⟨https://nodejs.org/en/⟩

**Nest** ⟨https://nestjs.com/⟩

**TypeORM** ⟨http://typeorm.io/⟩

**TypeScript** ⟨https://www.typescriptlang.org⟩

**NPM** ⟨https://www.npmjs.com⟩

**Apache Kafka** ⟨https://kafka.apache.org⟩

**MySQL** ⟨https://www.mysql.com⟩