Data management system založený na funkcionálních technológiích

Bc. Gabriel Ečegi

Diplomová práce
2019

Univerzita Tomáše Bati ve Zlíně
Fakulta aplikované informatiky

Univerzita Tomáše Bati ve Zlíně
Fakulta aplikované informatiky
akademický rok: 2018/2019

ZADÁNÍ DIPLOMOVÉ PRÁCE

(PROJEKTU, UMĚLECKÉHO DÍLA, UMĚLECKÉHO VÝKONU)

Jméno a příjmení:	**Bc. Gabriel Ečegi**
Osobní číslo:	**A17281**
Studijní program:	**N3902 Inženýrská informatika**
Studijní obor:	**Informační technologie**
Forma studia:	**kombinovaná**
Téma práce:	**Data management system založený na funkcionálních technologiích**
Téma anglicky:	**A Data Management System Based on Functional Technologies**

Zásady pro vypracování:

1. **Zpracujte literární rešerši na dané téma.**
2. **Popište moderní webové technologie z prostředí vývoje aplikací prosazující funkcionální programování.**
3. **Navrhněte vhodnou architekturu webové aplikace s ohledem na funkcionální technologie.**
4. **Realizujte algoritmy zpracování dat a metadat pro data management systém.**
5. **Vytvořte webovou aplikaci pro zpracování dat v databázi s využitím funkcionálních technologií.**
6. **Zhodnoťte vytvořenou aplikaci a demonstrujte své výsledky a použití data management systému.**

Rozsah diplomové práce:

Rozsah příloh:

Forma zpracování diplomové práce: **tištěná/elektronická**

Seznam odborné literatury:

1. **SYME, Don, Adam GRANICZ a Antonio CISTERNINO. Expert F# 4.0. 4. Apress, 2015. ISBN 978148420740.**
2. **PETRICEK, Tomas. Real-World Functional Programming: With Examples in F# and C#. 1. Manning Publications, 2010. ISBN 9781933988924.**
3. **BUONANNO, Enrico. Functional programming in C#. Shelter Island, NY: Manning Publications, [2017]. ISBN 9781617293955.**
4. **BANKS, Alex a Eve PORCELLO. Learning React: functional web development with React and Redux. Sebastopol, CA: O'Reilly Media, 2017. ISBN 9781491954621.**
5. **DELANEY, Kalen, Bob BEAUCHEMIN, Conor CUNNINGHAM, Jonathan KEHAYIAS, Benjamin NEVAREZ a Paul S RANDAL. Microsoft SQL server 2012 internals. Sebastopol, Calif: O'Reilly Media, [2013]. ISBN 0735658560.**

Vedoucí diplomové práce:	**Ing. Milan Navrátil, Ph.D.** Ústav elektroniky a měření
Konzultant:	**Ing. Hynek Petrla** Holešov, Martinice
Datum zadání diplomové práce:	**3. prosince 2018**
Termín odevzdání diplomové práce:	**15. května 2019**

Ve Zlíně dne 7. prosince 2018

UNIVERZITA TOMÁŠE BATI VE ZLÍNĚ
Fakulta aplikované informatiky
L.S.

doc. Mgr. Milan Adámek, Ph.D.
děkan

prof. Mgr. Roman Jašek, Ph.D.
garant oboru

Prohlašuji, že

- beru na vědomí, že odevzdáním diplomové práce souhlasím se zveřejněním své práce podle zákona č. 111/1998 Sb. o vysokých školách a o změně a doplnění dalších zákonů (zákon o vysokých školách), ve znění pozdějších právních předpisů, bez ohledu na výsledek obhajoby;
- beru na vědomí, že diplomová práce bude uložena v elektronické podobě v univerzitním informačním systému dostupná k prezenčnímu nahlédnutí, že jeden výtisk diplomové/bakalářské práce bude uložen v příruční knihovně Fakulty aplikované informatiky Univerzity Tomáše Bati ve Zlíně a jeden výtisk bude uložen u vedoucího práce;
- byl/a jsem seznámen/a s tím, že na moji diplomovou práci se plně vztahuje zákon č. 121/2000 Sb. o právu autorském, o právech souvisejících s právem autorským a o změně některých zákonů (autorský zákon) ve znění pozdějších právních předpisů, zejm. § 35 odst. 3;
- beru na vědomí, že podle § 60 odst. 1 autorského zákona má UTB ve Zlíně právo na uzavření licenční smlouvy o užití školního díla v rozsahu § 12 odst. 4 autorského zákona;
- beru na vědomí, že podle § 60 odst. 2 a 3 autorského zákona mohu užít své dílo – diplomovou práci nebo poskytnout licenci k jejímu využití jen připouští-li tak licenční smlouva uzavřená mezi mnou a Univerzitou Tomáše Bati ve Zlíně s tím, že vyrovnání případného přiměřeného příspěvku na úhradu nákladů, které byly Univerzitou Tomáše Bati ve Zlíně na vytvoření díla vynaloženy (až do jejich skutečné výše) bude rovněž předmětem této licenční smlouvy;
- beru na vědomí, že pokud bylo k vypracování diplomové práce využito softwaru poskytnutého Univerzitou Tomáše Bati ve Zlíně nebo jinými subjekty pouze ke studijním a výzkumným účelům (tedy pouze k nekomerčnímu využití), nelze výsledky diplomové/bakalářské práce využít ke komerčním účelům;
- beru na vědomí, že pokud je výstupem diplomové práce jakýkoliv softwarový produkt, považují se za součást práce rovněž i zdrojové kódy, popř. soubory, ze kterých se projekt skládá. Neodevzdání této součásti může být důvodem k neobhájení práce.

Prohlašuji,

- že jsem na diplomové práci pracoval samostatně a použitou literaturu jsem citoval. V případě publikace výsledků budu uveden jako spoluautor.
- že odevzdaná verze diplomové práce a verze elektronická nahraná do IS/STAG jsou totožné.

Ve Zlíně, dne 15.5.2019

Gabriel Ečegi, v.r.
podpis diplomanta

ABSTRAKT

Témou diplomovej práce je popis vývoja moderných webových technológií so zameraním na funkcionálne jazyky, knižnice a frameworky. V teoretickej časti je popísané funkcionálne paradigma, funkcionálne jazyky a výhody ich použitia. Teoretická časť tiež popisuje technologické trendy súčasného webového vývoja so zameraním na funkcionálne technológie. V praktickej časti je vytvorená webová aplikácia pre management dát v databázy, ktorá bude demonštrovať použitie funkcionálnych technológií aplikovateľných v praxi. Cieľom tejto práce je priniesť komplexný pohľad do problematiky vývoja moderných funkcionálnych webových technológií a riešenie problematiky ich použitia v line of business aplikáciách.

Klíčová slova: Funkcionálne programovanie, F#, React, GraphQL, Web, TypeScript, .NET Core

ABSTRACT

The topic of the master thesis is a description of the development of modern web technologies, focusing on functional languages, libraries and frameworks. The theoretical part describes the functional paradigm, functional languages and the advantages of their use. The theoretical part also describes the technological trends of the current web development with a focus on functional technologies. In the practical part is created a web application for data management in the database, which will demonstrate the use of functional technologies applicable in practice. The aim of this work is to bring a comprehensive view of the problems of development of the modern functional web technologies and to solve the problems of their use in the line of business applications.

Keywords: Functional programming, F#, React, GraphQL, Web, TypeScript, .NET Core

Ďakujem Ing. Milanovi Navrátilovi, Ph.D. za odborné rady, organizáciu a konzultácie diplomovej práce. Ďakujem tiež Ing. Hynkovi Petrlovi za jeho trpezlivosť a odbornosť pri konzultácií diplomovej práce.

"You can't win an argument. You can't because if you lose it, you lose it; and if you win it, you lose it."

- Dale Carnegie

Prohlašuji, že odevzdaná verze diplomové práce a verze elektronická nahraná do IS/STAG jsou totožné.

OBSAH

ÚVOD

V súčasnej dobe je dominantnou programátorskou paradigmou objektovo orientované programovanie - OOP. Jeho zásady sú známe od 70. rokov 20. storočia a medzi jeho zástupcov patria populárne jazyky ako C++, Java, C#. Avšak populárne riešenie automaticky neznamená najlepšie riešenie. Kritizovanými je často prílišná komplexnosť návrhu, neschopnosť modelovať skutočné situácie z reálneho sveta a považovanie OOP za jediné pravé riešenie problémov. Objektovo orientované programovanie sa tak v poslednej doba stáva terčom väčšej kritiky a s rastúcou popularitou riešení v iných paradigmách, tak ľudia venujúci sa vývoju softvérom hľadajú, objavujú a skúšajú aj iné možnosti.

Jednou z týchto možností je funkcionálne programovanie. Základy funkcionálneho programovania boli položené skôr ako základy OOP, avšak zatiaľ sa mu nepodarilo dosiahnuť takej popularity ako zažilo OOP. Funkcionálne programovanie je na rozdiel od OOP, ktoré je postavené na domnienkach znalosti reálneho sveta, založené na matematických funkciách. Tým že si zakladá na matematike, získava množstvo znalostí nahromadených za stáročia. Tieto riadky však môžu človeka zaoberajúceho sa tvorbou softvéru zneistiť, alebo až odradiť od chuti do učenia sa funkcionálneho programovania. Naopak, funkcionálne programovanie je oveľa jednoduchšie ako sa na prvý pohľad môže zdať.

Funkcionálne programovanie je tak prínosné najmä pre človeka, ktorý prichádza zo sveta OOP. Cieľom teoretickej časti je preto uviesť čitateľa do problematiky funkcionálneho programovania, vysvetliť pojmy s ktorými sa stretne a popísať najznámejšie technológie.

V praktickej časti sa ďalej pracuje s témou funkcionálneho programovania, presnejšie s jeho použitím na vývoj data management systému. Tento systém bude schopný z externej databázy vygenerovať funkčné užívateľské rozhranie pre správu dát. Táto potreba vychádza zo skúseností pri vývoji softvéru, kedy je síce vytvorená funkčná aplikácia, využívaná používateľmi, generujúca dáta. Tejto aplikácií však chýba iné ako databázové rozhranie, ktoré by umožňovalo management dát, či už na prehliadanie dáta, reporting, opravu údajov. V prípade použitia aplikácie by mala byť aj osoba, bez znalosti programovania, schopná vytvoriť funkčnú aplikáciu v priebehu krátkeho času. Osoba vytvárajúca aplikáciu by takto mohla presne nakonfigurovať, ktoré tabuľky a ktoré stĺpce majú byť v aplikácií zobrazené a ktoré údaje by mali byť editovateľné.

I. TEORETICKÁ ČÁST

1 FUNKCIONÁLNE PROGRAMOVANIE

Funkcionálne programovanie je programovacia paradigma, je to rozdielny spôsob premýšľania o programoch ako imperatívna paradigma, ktorá je v dnešnej dobe najrozšírenejšia. Na veľmi vysokej úrovni je to programátorský štýl uprednostňujúci funkcie, pričom sa vyhýba mutácií stavu. [1]

Funkcionálne programovanie dlho inšpirovalo vedcov, študentov a programátorov svojou jednoduchosťou a expresívnosťou. Aplikováné funkcionálne programovanie je na vzostupe, nová generácia silne typových funkcionálnych jazykov dosahuje dospelosť, niektoré konštrukty z funkcionálnych jazykov sú intergrované do jazykov ako C#, Python a Visual Basic.

Existujú tiež silné dôkazy, že funkcionálne programovanie ponúka výrazné zvýšenie produktivity v dôležitých aplikačných oblastiach, ako je napríklad prístup dát, finančné modelovanie, štatistická analýza, machine learning, verifikácia softvéru a bio-informatika. [10]

Funkcionálna paradigma definuje koncepty, ktoré môžeme použiť keď hľadáme riešenie problémov, avšak presne nešpecifikuje, ako by mali byť tieto koncepty reprezentované v programovacom jazyku. Výsledok je, že existuje veľa funkcionálnych jazykov, pričom každý kladie dôraz na rôzne aspekty funkcionálneho štýlu programovania. [33]

V poslednej dobe funkcionálne programovanie stojí pri vzraste popularity deklaratívnych programovacích modelov, obzvlášť pri získavaní dát, reaktívneho a paralelného programovania. [10]

1.1 Funkcie ako hodnoty prvej kategórie

V jazyku, kde sú funkcie hodnoty prvej kategórie, je možné ich použitie ako vstupy alebo výstupy iných funkcií, je možné ich priradiť do premenných alebo môžu byť uložené v kolekciách. Inými slovami, je možné s nimi robiť všetky operácie ako s hodnotami iných typov. [1]

1.2 Funkcie vyšších rádov

Higher order funkcia alebo funkcia vyššieho rádu je funkcia, ktorá berie jednu alebo viac funkcií ako argument, alebo vracia funkciu, prípadne oboje. Termín pochádza z matematiky, kde je rozdiel medzi funkciami a hodnotami braný vážnejšie.

Funkcie vyšších rádov nám umožňujú vytvárať abstrakciu nad funkciami, nie len nad hodnotami ako to je zvykom v imperatívnom programovaní (Obrázok 1). [3]

```
let isGreaterThan a b = b > a

//Higher order funckia
let isGreaterThan10 = isGreaterThan 10

[<EntryPoint>]
let main argv =
    printfn "%b" (isGreaterThan10 5) //false
    printfn "%b" (isGreaterThan10 11) //true
    0
```

Obrázok 1. Funkcia vyššieho rádu [3]

```
let withUserApp token fn = getAppID token |> Option.bind
                           (fun appID -> ApplicationService.getUserApp appID |> fn)

let private itemsOrEmptyList token fn =
            let maybeItems = TokenService.withUserApp token
                                (fun app -> fn app
                                                |> Async.RunSynchronously
                                                |> Array.toList
                                                |> Some)
            match maybeItems with
                | Some items -> items
                | None -> List.empty

let getAllSchemas token = itemsOrEmptyList token
                            (fun app -> SqlTypeService.getAllSchemas app.Connection)
```

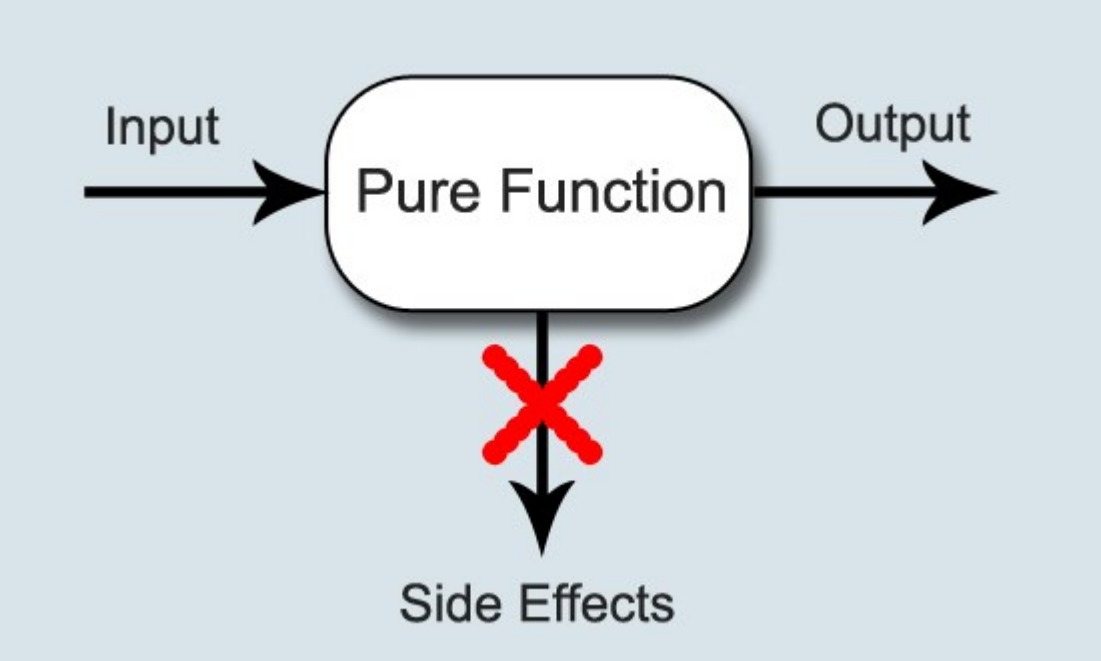
Obrázok 2. Funkcie vyššieho rádu z praktickej časti

Použitím funkcií vyššieho rádu je jednoducho možné riešiť aj na zložitejšie problémy zo sveta (Obrázok 2).

1.3 Vyhýbanie sa mutácii stavu

Vo funkcionálnom programovaní celkovo platí pravidlo vyhýbaniu sa zmenám. Po tom ako je objekt vytvorený, sa už nikdy nezmení a do hodnoty premennej už nikdy nemôže byť priradená nová hodnota. Pojem mutácia indikuje, že hodnota bola niekde v pamäti zmenená. Takéto zmeny sú tiež nazývané deštruktívne zmeny, pretože hodnota, ktorá bola doteraz uložená v premennej je zničená. Tomuto sa pri funkcionálnom programovaní vyhýba, dokonca čisto funkcionálne jazyky updaty vôbec nedovoľujú. Napríklad radenie alebo filtrovanie zoznamov by nemalo upravovať zoznam samotný, ale namiesto toho by malo vrátiť nový, vhodne zoradený a vyfiltrovaný zoznam bez hocijakej úpravy pôvodného zoznamu.

1.3.1 Čisté funkcie



Obrázok 3. Čisté funkcie [4]

Hoci majú funkcie v programovaní blízko k funkciám matematickým funkciám často od nich požaduje niečo, čo v matematike neexistuje. Môže to byť napríklad výpis na obrazovku, spracovanie súboru, interakcia s iným systémom. Ak je teda potreba aby funkcia takto ovplyvnila vonkajší zdroj, túto akciu nazývame ako vedľajší efekt – side effect. Ďalším dôležitým rozdielom je, že matematické funkcie existujú len vo vákuu a ich výsledky sú závislé len na ich vstupe, pričom v programovaní potrebujeme mať prístup k systémovému času,

databázy alebo vzdialenému zariadeniu. Ak takýto kontext existuje, tak to môže znamenať, že možnosti funkcie nie sú presne vyhradené, čo ďalej vedie k zvýšeniu komplexnosti analýzy. Preto funkcie delíme na čisté (pure) a nečisté (impure) funkcie.

Čisté funkcie (Obrázok 3) často pripomínajú matematické funkcie – nerobia nič iné, než vypočítajú výstupnú hodnotu na základe vstupných hodnôt. Aby sa funkcia klasifikovala ako čistá, musí za vyhnúť nasledujúcim akciám:

- Upravuje globálny stav – stav ktorý je viditeľný z iných funkcií
- Upravuje vstupné argumenty
- Vyhadzuje výnimky
- Vykonáva I/O operácie

Výhodami čistý funkcií sú následne možnosti:

- Memoizácie – Uloženie si výsledku operácie – nabudúce stačí ak je výsledok vypočítaný jeden krát
- Lazy evaluation – vyhodnotenie hodnôt len keď je to potreba
- Paralelizmus

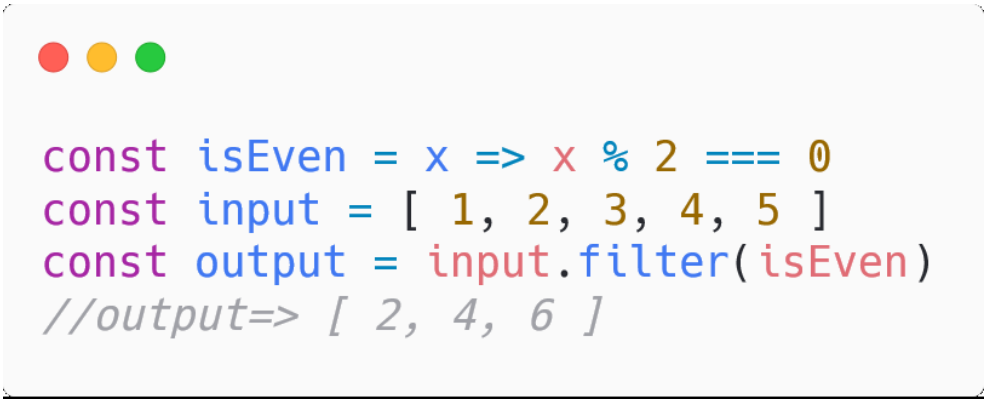
1.3.2 Transformácie dát

Funkcionálne programovanie je príznačné transformáciou dát z jednej formy do druhej, vytváraním transformovaných kópií dát pomocou funkcií. Tieto funkcie zjednodušujú kód a zmenšujú komplexnosť.

1.3.3 Filter

Funkcia Filter je funkcia vyššieho rádu, ktorá vytvára novú kolekciu zo zdrojovej kolekcie. Ako ďalší argument funkcia berie predikát – funkciu vracajúcu boolean hodnotu. Táto funkcia sa vyhodnotí nad každým prvkom kolekcie. Prvok je použitý ako argument funkcie a na základe návratovej hodnoty funkcie je rozhodnuté, či má byť prvok pridaný do novej kolekcie. Pôvodné pole je immutable. [18]

Na obrázku (Obrázok 4) je ukážka algoritmu v JavaScripte, ktorá deklaratívnym spôsobom odfiltrováva nepárne čísla z kolekcie.



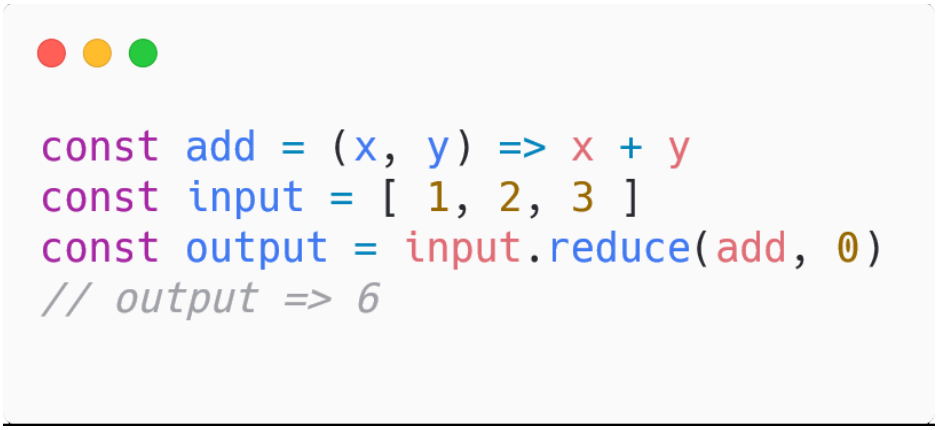
Obrázok 4. Deklaratívne filtrovanie pomocou funkcie Filter [18]

1.3.4 Map

Map je funkcia vyššieho rádu, ktorá vytvára novú kolekciu zo zdrojovej funkcie. Ako druhý argument berie funkciu transformujúcu jednotlivý prvok na nový prvok. Map ako funkcia funktoru je popísaná v kapitole venujúcej sa funktorom. [18]

1.3.5 Fold

Fold alebo tiež Reduce sú funkcie transformujúce kolekciu na hocijakú hodnotu – číslo, objekt, pole, funkciu. Ako druhý argument berie funkciu akumulujúcu výsledok. Ako tretí argument môže brať inicializačnú hodnotu. Na obrázku (Obrázok 5) je deklaratívny algoritmus pre získanie sumy kolekcie.

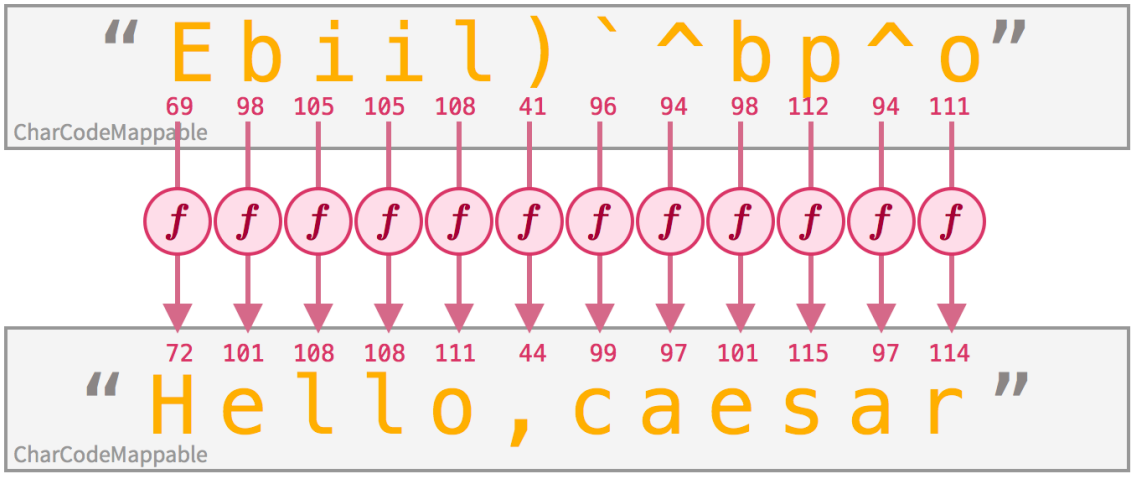


Obrázok 5. Fold/Reduce [18]

1.3.6 Paralelizmus

Jeden zo základných problémov pri paralelizovaní programov je vyriešenie prístupu k zdrojom. V prípade premenných tento problém vo funkcionálnom programovaní neexistuje. V prípade, že máme viacero operácií vykonávaných nad zoznamom hodnôt môžeme vďaka immutabilite zabezpečiť, že pri spustení operácii paralelne dosiahneme vždy správnych výsledkov. Pri vývoji v imperatívnom štýle – explicitná mutácia stavov programu, sa pri zavedení paralelizmu, kvôli novým požiadavkám alebo zvýšeniu výkonnosti vzniká veľa novej práce alebo dokonca bugov. Keď je však program napísaný vo funkcionálnom jazyku toto vylepšenie je možné pridať zadarmo a bezpečne. [1]

1.4 Funktor



Obrázok 6. Funktor [23]

Funktory sú kontajnery, ktoré obsahujú hodnoty a po aplikovaní funkcie na hodnoty sú získané transformované hodnoty v rovnakých kontajneroch (Obrázok 6). Ako funktor klasifikujeme hocijaký typ, na ktorom je definovaná funkcia Map (nazývaná aj Select). [3]

$$Map\colon\ \big(C<T>,\quad (T \to R)\big) \to C<R>$$

C<T> je kontajner obsahujúci hodnoty typu *T*, po aplikácií funkcie $(T \to R)$ ktorej vstupný argumentom je hodnota z kontajneru typu T a výstupnou hodnotou je transformovaná hodnota typu R. Po aplikácií hodnoty získavame kontajner v ktorom je hodnota typu R. [1]

Návrhári jazykov postupne zisťujú že funktory sú dôležité a preto sa začínajú objavovať aj v imperatívnych jazykoch, ako napríklad JavaScript (Obrázok 7) alebo C# (Obrázok 8). [5]

```javascript
var array1 = [1, 4, 9, 16];
const map1 = array1.map(x => x * 2);
console.log(map1);
// expected output: Array [2, 8, 18, 32]
```

Obrázok 7. Map v JavaScripte [18]

```csharp
IEnumerable<int> squares = Enumerable.Range(1, 10).Select(x => x * x);

foreach (int num in squares)
{
    Console.WriteLine(num);
}
// 1 4 9 16 25 36 49 64 81 100
```

Obrázok 8. Select v C# [24]

1.5 Monáda

Monáda (angl. Monad) je monoid v kategórií endofuktorov. [8]

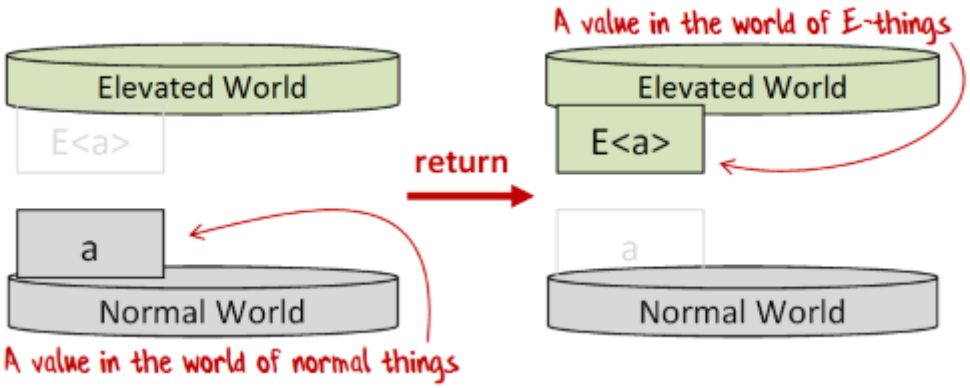
Je to generický kontajner, ktorý musí mať dve funkcie Return a Bind.

$$Return\colon\ T \rightarrow M < R >$$

$$\text{Bind:}\ \left(\mathrm{M} < \mathrm{T} >, (\mathrm{T} \rightarrow C < R >)\right) \rightarrow C < R >$$

[4]

1.5.1 Monadický return



Obrázok 9. Monadický return [9]

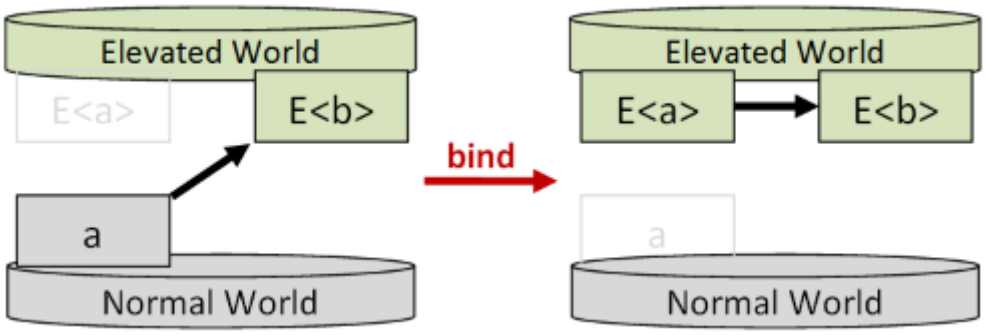
Funkcia *Return* obalí normálnu hodnotu a vyzdvihne ju do monadickej hodnoty. Normálne hodnoty ako int, bool, string sú vyzdvihnuté do monád Option, Enumerable, List, Observable (Obrázok 9). [4]

Algoritmus na vytvorenie monadického returnu býva väčšinou jednoduchý (Obrázok 10).

```
// Monadic Return
// A value lifted to the world of Options
let returnOption x = Some x
// has type : 'a -> 'a option
```

Obrázok 10. Vytvorenie monadického returnu [9]

1.5.2 Monadický bind



Obrázok 11. Bind [9]

Bind berie monadicku hodnotu a funkciu trasformujúcu hodnotu a vracia novú monadickú hodnotu. Využíva sa vtedy, keď chceme pracovať s funkciou, ktorá pracuje s normálnou hodnotou ale my máme k dispozícii len jej monádu. Toto využívame napríklad pri monáde Option (Obrázok 12), alternatíve k null z funkcionálneho programovania, kedy chceme pracovať s hodnotou ktorá „nie je null“, čo však nedokážeme zistiť kvôli monadickému stavu. V tom prípade využijeme monadický bind. [4] [9]

```
module Option =
    // Monadic bind
    // The bind function for Options
    let bind f xOpt =
        match xOpt with
        | Some x -> f x
        | _ -> None
    // has type : ('a -> 'b option) -> 'a option -> 'b option
```

Obrázok 12. Implementácia monadického bindu [9]

```
let getUserID token = token
                      |> getJwtToken
                      |> Option.bind (fun jwtToken -> Some jwtToken.userID)
```

Obrázok 13. Použitie monadického bindu v praktickej časti [Zdroj: vlastný]

2 F#

F# je funkcionálny programovací jazyk pre .NET. Umožňuje písať správny a udržateľný kód. Kombinuje stručnosť, expresívnosť a kompozíciu silne typového programovania s runtime podporov, knižnicami, interoperatibilitu, nástroje a objektový model .NETu.

F# sa odlišuje od ostatných funkcionálnych jazykov tým že využíva imperatívne a objektovo orientované programovanie. Tiež poskytuje chýbajúci vzťah medzi kompilovanými a dynamickými jazykmi, kombinujúc idiómi a programátorské štýly typické pre dynamické jazyky s robustnosťou a výkonom kompilovaných jazykov.

Návrhári jazyka F# prebrali filozofiu návrhu umožňujúcu výber najlepších a najproduktívnejších aspektov týchto paradigiem. F# ich kombinuje, pričom stále kladie dôraz na techniky funkcionálneho programovania. [10]

Programovanie v F# primárne zahrňuje definovanie typov a funkcií ktoré sú automaticky zobecňované a type-inferred. Toto umožňuje lepšie zameranie na doménové problémy a manipuláciu dát, než na detaily programovania.

2.1 Vývoj jazyka

F# zdieľa základné vlastnosti s programovacím jazykom OCaml a je považovaný za „OCaml pre .NET“. F# tiež vychádza z jazyka Haskell, obzvlášť s ohľadom na pokročilé vlastnosti jazyka. Hlavnými dôvodmi podobnosti sú silné prepojenia komunity a návrhárov jazyka.

Napriek podobnostiam z OCaml a Haskell, programovanie v F# je dosť odlišné, obzvlášť v prístupe k type inference, objektovo orientovanému programovaniu, dynamickým technikám. Programovanie často býva viac objektovo orientované, než iné funkcionálne jazyky. Takisto býva viac flexibilné, F# umožnuje .NET techniky ako dynamic loading, dynamic typing, reflexiu a pridáva techniky expression quatation a active patterns.

F# vďačí vývojárom jazykov z rodiny .NET – Visual Basic, C# s ktorých môže F# využívať na zdieľanie techník a stoviek dôležitých knižníc ako napríklad LINQ vytvorených Microsoftom. [10]

2.2 F# a programovanie webov

F# umožňuje budovanie efektívnych, škálovateľných a robustných webových riešení. Programovanie webov je založené na prijímaní a zasielaní odpovedí na jednotlivé HTTP requesty. Toto sa veľmi hodí pre bezstavový, funkcionálny stav. Vlastnosti webových aplikácií v F#:

- Rýchlosť – exekúcia kódu je rýchla, využívajúc natívneho generovania kódu zo scriptov alebo kódu projektu
- Stručnosť – F# je stručný, čitateľný a silnotypový
- Reaktívnosť a škálovateľnosť – asynchrónne programovanie zjednodušuje škálovateľné, reaktívne programovanie
- Prepojiteľnosť – F# je jednoducho prepojiteľný s jazykmi ako C#, JavaScript a TypeScript
- Kompilovateľnosť do JavaScriptu – F# je kompilovateľný do Javascriptu pomocou frameworkov WebSharper a Fable
- Open-source a cross-platform

[11]

2.2.1 ASP.NET Core

ASP.NET Core je cross-platformový, vysoko výkonný, open-source framoweok pre vytváranie moderných, cloudovo založených aplikácií. Na vytvorenie ASP.NET Core Web API projektu stačí vo Visual Studiu použiť šablónu a ako je vidieť na obrázku (Obr. 14), výsledkom je API veľmi podobné C# Web API. [12][13]

ASP.NET Core ponúka:

- Jednotný základ na stavbu web UI a web API
- Testovateľnosť
- Vývoj na Windnow, macOS, Linuxe
- Open-source
- Vstavaná dependency injection
- Schopnosť hostingu na IIS, Nginx, Apache, Docker

[13]

```
[<Route("api/[controller]")>]
[<ApiController>]
type ValuesController () =
    inherit ControllerBase()

    [<HttpGet>]
    member this.Get() =
        let values = [|"value1"; "value2"|]
        ActionResult<string[]>(values)

    [<HttpGet("{id}")>]
    member this.Get(id:int) =
        let value = "value"
        ActionResult<string>(value)

    [<HttpPost>]
    member this.Post([<FromBody>] value:string) =
        ()

    [<HttpPut("{id}")>]
    member this.Put(id:int, [<FromBody>] value:string ) =
        ()

    [<HttpDelete("{id}")>]
    member this.Delete(id:int) =
        ()
```

Obrázok 14. Endpointy ASP.NET Core API [12][13]

2.2.2 Giraffe

Giraffe je F# mikro webový framework pre vývoj webových aplikácií. Bol navrhnutý tak, že je ho možné použiť ako middleware v ASP.NET Core pipeline. Giraffe je určený pre vývojárov, ktorí chcú vyvíjať bohaté webové aplikácie v ASP.NET Core avšak s funkcionálnym prístupom. Giraffe tak stavia na silnej webovej platforme podporovanej Microsoftom a veľkou komunitou vývojárov, získava tak z tohto ekosystému.

Nie je však navrhnutý aby konkuroval, ale len aby dopĺňal ASP.NET Core. Základná idea je stavať na silných základoch ASP.NET Core a použiť jeho stavebné bloky tak aby vývojári mohli získavať z oboch svetov.

Giraffe sa dá považovať ako protistrana ASP.NET Core MVC frameworku.

Na obrázku (Obrázok 15) je vidieť ako sa v praktickej časti kombinuje nastavenie serveru z ASP.NET Core MVC a deklaratívne vytvára aplikáciu pomocou funkcií z Giraffe

```
let webApp =
    choose [
        route "/ping"   >=> text "pong"
        route "/"       >=> htmlFile "/pages/index.html" ]

let configureApp (app : IApplicationBuilder) =
    // Add Giraffe to the ASP.NET Core pipeline
    app.UseGiraffe webApp

let configureServices (services : IServiceCollection) =
    // Add Giraffe dependencies
    services.AddGiraffe() |> ignore

[<EntryPoint>]
let main _ =
    WebHostBuilder()
        .UseKestrel()
        .Configure(Action<IApplicationBuilder> configureApp)
        .ConfigureServices(configureServices)
        .Build()
        .Run()
    0
```

Obrázok 15. Kompletné webové API v Giraffe (endpointy, konfigurácia, spustenie) [14]

2.2.3 SAFE stack

SAFE stack je open-source, zadarmo, flexibilný, funkcionálny stack pre aplikácie pripravené na cloud, ktorý kladie dôraz na silno typové programovanie. Umožňuje vyvíjať webové aplikácie kompletne v F#, bez potreby používať objektovo-orientovaný framework alebo knižnicu alebo byť expertom v CSS a HTML a pritom vyvíjať kvalitné client-side aplikácie. [16]

Komponenty SAFE stack sú vyznačené na obrázku (Obrázok 16).

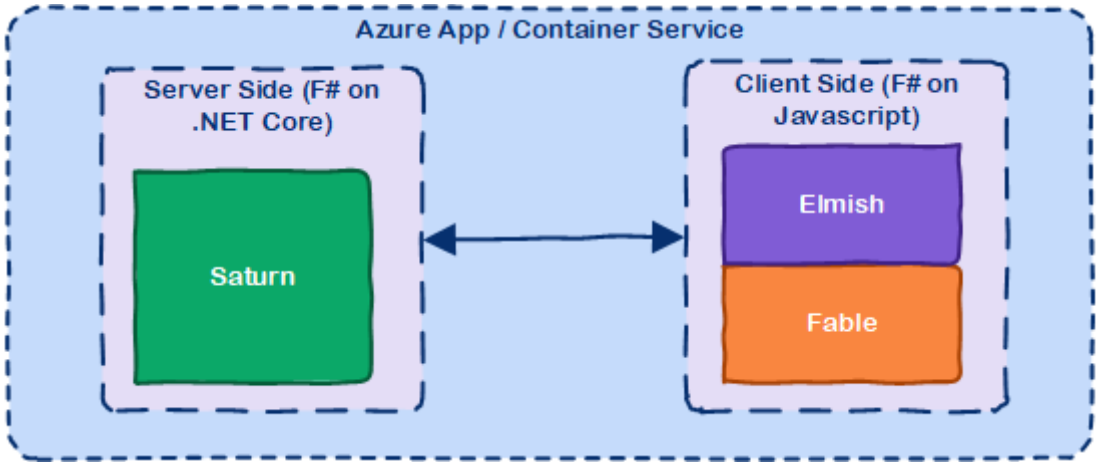
SAFE stack sa skladá zo štyroch hlavných komponent:

- Saturn - webový server bežiaci na .NET Core pre backend služby v F#
- Azure - hostingová platforma poskytujúca jednoduchý, škálovateľný vývoj
- Fable - mechanizmus pre spustenie F# vo webových prehliadačoch –použitie F# na strane klienta
- Elmish - spôsob programovanie užívateľských rozhraní na strane klienta v F#

Výhody SAFE:

- Klient aj serve kompletne v F#
- Tvorba klientských aplikácií bez znalosti JavaScriptu
- Rýchly vývoj vďaka hot module replacement
- Interakcia s Javascriptovými knihovňami, keď je potreba
- Zdieľanie kódu medzi klientom a severom
- Typová bezbečnosť na front-end

[16]



Obrázok 16. SAFE stack [16]

2.2.3.1 Saturn

Saturn je knižnica postavená na Giraffe a ASP.NET Core pre poskytnutie možností absrakcií, ktoré veľmi uľahčujú konfiguráciu webových aplikácii a vytváranie komplexných routes.

Keďže je Saturn postavený na Giraffe, automaticky získava plný prístup k možnostiam Giraffe, ale takisto môže byť kompletne nahradený Girafee. [17]

2.2.3.2 Microsoft Azure

Azure je skupina clodových služieb, ktoré developery a IT profesionáli používajú na tvorbu aplikácií využívajúc globálnu sieť datacentier. [17]

2.2.3.3 Fable

Fable je kompilátor kódu z F# do JavaScriptu, navrhnutý tak aby generoval čistý a štandardný kód pre lepšiu prepojiteľnosť oboch svetov. Takto je možno písať celú aplikáciu v F# a pritom využívať externé JavaScriptové knižnice. [17]

2.2.3.4 Elmish

Elmish model umožňuje vytvárať užívateľské rozhrania využívajúc konštrukty funkcionálneho programovania. Navrhnuté na základe aplikačného modelu jazyka Elm, Elmish využíva návrhový vzor Model-View-Update, vytvárajúci aplikácie, ktoré sú jednoduché. Elmish je postavený na Reacte. [17]

3 REACT

React je populárna knižnica na vytváranie užívateľských rozhraní. Bola vytvorená vo Facebooku, aby vyriešila niektoré výzvy spojené s veľkými, data-driven webovými aplikáciami. React knižnica je malá a vykonáva len určitú časť práce. Súčasťou Reactu nie sú všetky nástroje, ktoré je možno očakávať od tradičného JavaScriptového frameworku. Veľa rozhodnutí o tom aké nástroje z React ekosystému použiť je tak na vývojárovi. Nové nástroje prichádzajú a odchádzajú tak rýchlo, až je problém udržať krok s vývojom.

JavaScript nie je čisto funkcionálnym jazykom, ale funkcionálne techniky sa dajú jednoducho použiť. React uprednostňuje funkcionálne programovanie pre objektovo orientovaným programovaním. Tento posun v myslením vedie k výhodám v oblasti testovateľnosti a výkonu. [18]

```
const { render } = ReactDOM
const Welcome = () => (
  <div id="welcome">
    <h1>Hello World</h1>
  </div> )

render(<Welcome />, document.getElementById('target') )
```

Obrázok 17. Jednoduchá komponenta v Reacte [18]

3.1 Funkcionálne programovanie v Reacte

3.1.1 Výstup z komponent

Hlavným cieľom komponenty v Reacte je vždy vrátiť výstup s metódy *render()*. Metóda *render()* vracia React element, predstavujúci vzhľad komponenty. Inými slovami, React vyžaduje aby každá komponenta mala výstup, týmto sa komponenta podobá viac na funkciu ako objekt: komponenta je niečo, čo vracia výstup na základe vstupu. React komponenty nemusia byť definované ako triedy, ale môžu byť definované ako funkcie (Obrázok 18), do ktorých je možné posielať parametre. Funkcia renderovania musí byť čistá funkcia. [19]

```
const Greeting = (props) => <div>My name is {props.name}!</div>;
```

Obrázok 18. Komponenta ako funkcia [19]

3.1.2 Nemeniteľnosť props

Props sú vstupné parametre React komponent. React garantuje ich nemeniteľnosť, čo zabraňuje vedľajším efektom a vytvára tak priestor pre vznik pure funkcie. [19]

3.1.3 Jednosmerný tok dát

Jednosmerný tok dát (unidirectional data flow) je jednou z významných vlastností Reactu. Toto znamená, že hierarchii komponent musia dáta tiecť z vyššie postavených komponent do nižších. Toto sa môže zdať náročné ak považujeme komponenty za objekty, ak však o nich premýšľame ako o funkciách, tento proces sa zdá prirodzený. [19]

3.1.4 Kompozícia verzus dedičnosť

V objektovo orientovanom programovaní je prirodzené uvažovať o dedičnosti ako o mechanizme špecializácie triedy. Avšak pre React komponenty ako funkcie je tento prístup nevýhodný. [19] Vo Facebooku sú používané tisíce komponent a neexistuje žiaden prípad, kde by bolo odporúčané využívať dedičnosť. Props a kompozícia poskytujú všetku flexibilitu, ktorú je potreba na prispôsobenie vzhľadu a správania komponent explicitným a bezpečným spôsobom. [20]

3.1.5 Komponenty vyšších rádov

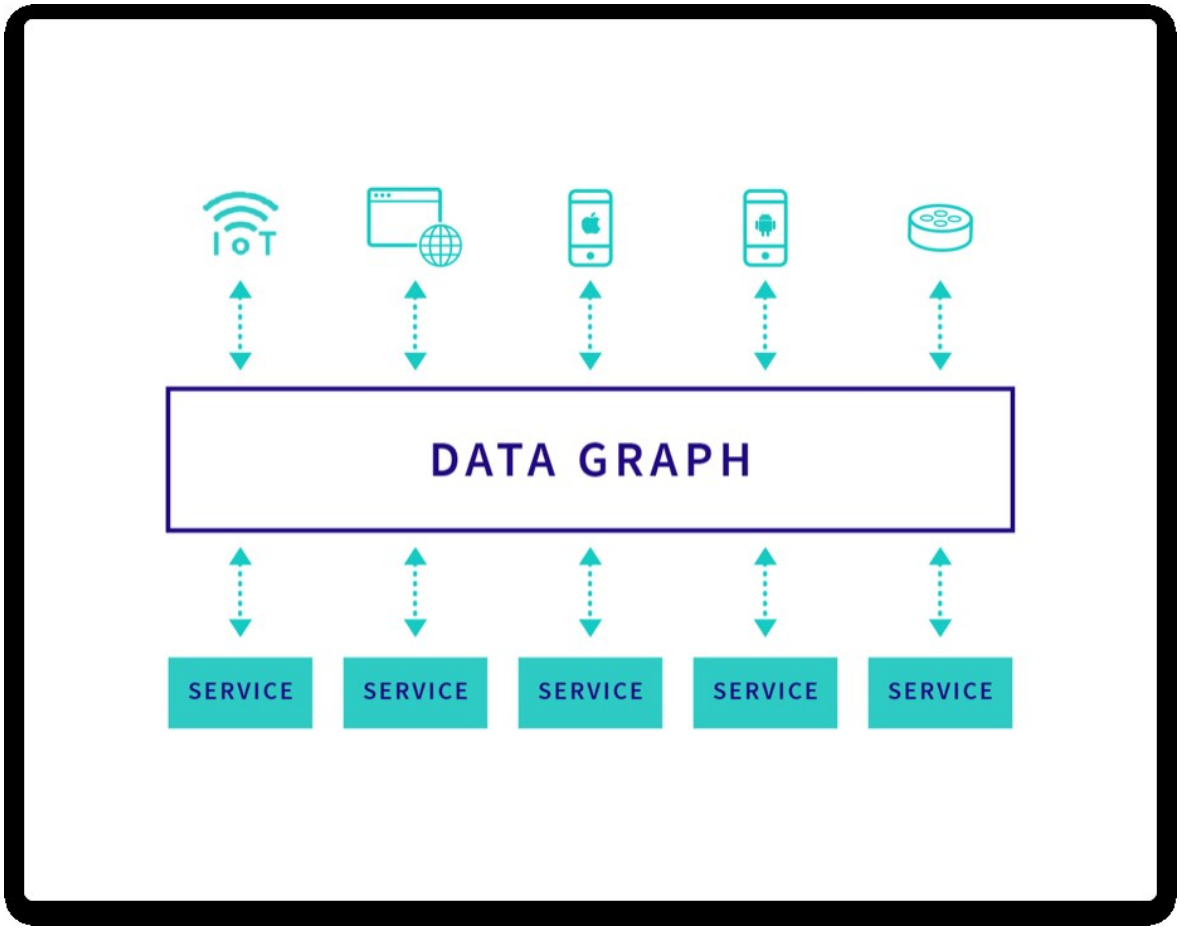
Komponenta vyššieho rádu (Higher order) je v Reacte štandardný návrhový vzor. Dovoľuje znovuvyužitie logiky komponenty v novej komponente. Komponenta vyššieho rádu je funkcia, ktorá berie komponentu ako vstupný parameter a vracia novú komponentu ako výstup. [18]

Môže ísť napríklad o komponentu zobrazujúcu ďalšiu komponentu, iba v prípade, že užívateľ je prihlásený (Obrázok 19).

```
const AuthorizedComponent: React.FunctionComponent<Props> = props => {
    const { component: Component, ...rest } = props;

    const isAuthenticated = React.useContext(TokenContext).state.token !== null;

    const renderComponent: RenderComponent = props =>
        isAuthenticated ? <Component {...props} /> : <Redirect to="/login" />;

    return <Route {...rest} render={renderComponent} />;
};
```

Obrázok 19. Komponenta vyššieho rádu s praktickej časti [Zdroj: vlastný]

4 GRAPHQL



Obrázok 20. GraphQL vrstva medzi klientmi a servermi [32]

GraphQL je dotazovací (query) jazyk pre API a serverside prostredie, využívajúce použitie typového systému definujúceho typy a entity. GraphQL nie je spojené zo žiadnou špecifickou databázou alebo úložným priestorom, namiesto toho sa snaží využívať súčasný kód.

Dotazy na GraphQL môžeme zaradiť do dvoch kategórií – Queries a Mutations.

Query requests predstavujú dotazy o nejaké dáta, pričom ich náplňou nie je upravovať stav dát. Môže ísť napríklad o získanie entít pre výpis, detail entity. V REST API by sa jednalo o dotaz typu GET

Mutation requests predstavujú dotazy, ktorý cieľom je zmeniť stav systému, môže ísť o registráciu, úpravu dát na entite užívateľa alebo zmazanie používateľa. V svete REST by predstavovali dotazy typu POST, PUT, DELETE. [2]

4.1 Porovnanie s REST

Prvý viditeľný rozdiel oproti REST API je ten, že je možné v dotaze definovať potrebné fieldy, tieto musia byť v dotaze explicitne napísané. Fieldy nezahrnuté v dotaze nám GraphQL API nevráti. Takto je možné znížiť počet dát, ktoré musí klient stiahnuť zo serveru - najmä pri optimalizácií aplikácie pre mobilné zariadenia. [25]

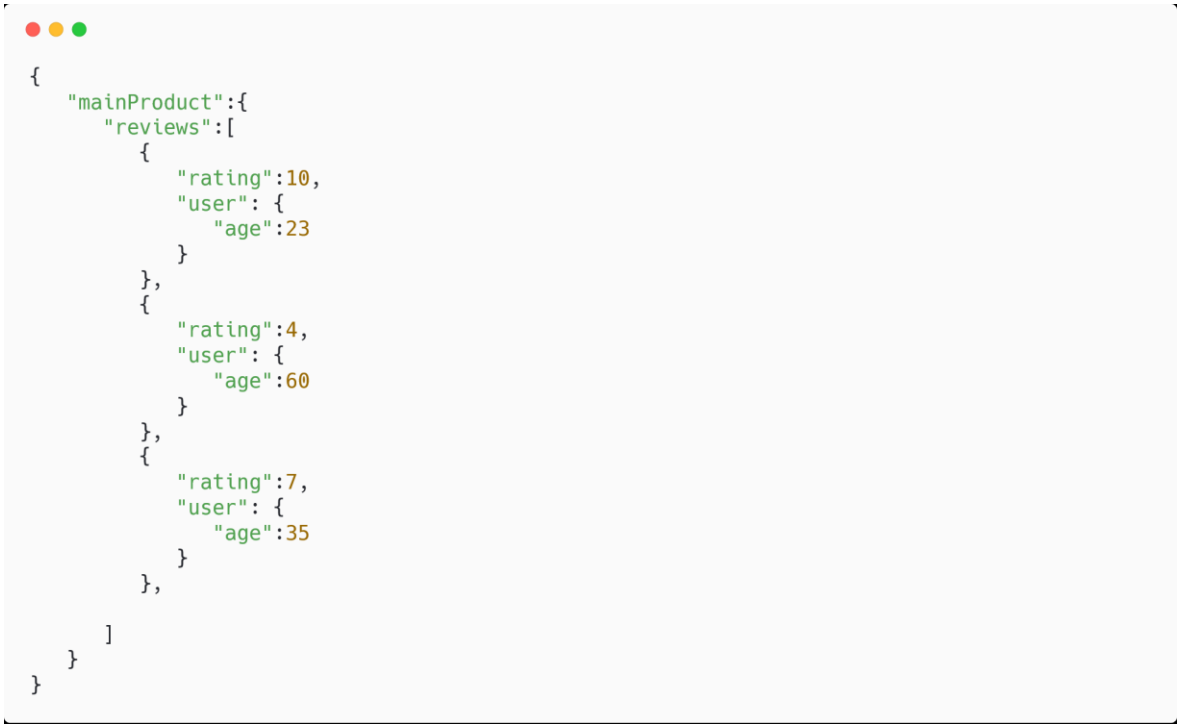
```
type Query {
  mainProduct: Product
}

type User {
  id: ID!,
  name: String!,
  age: Int
}

type Review {
  id: ID!,
  rating: Int!,
  text: String!,
  user: User!
}

type Product {
  id: ID!,
  name: String!,
  priceIncTax: Float!,
  priceExclTax: Float!
}
```

Obrázok 21. Schéma s hodnoteniami a obrázkami [Zdroj: vlastný]

```
{
  mainProduct {
    reviews {
      rating,
      user {
        age
      }
    }
  }
}
```

Obrázok 22. Dotaz na hodnotenia a vek hodnotiacich užívateľov [Zdroj: vlastný]

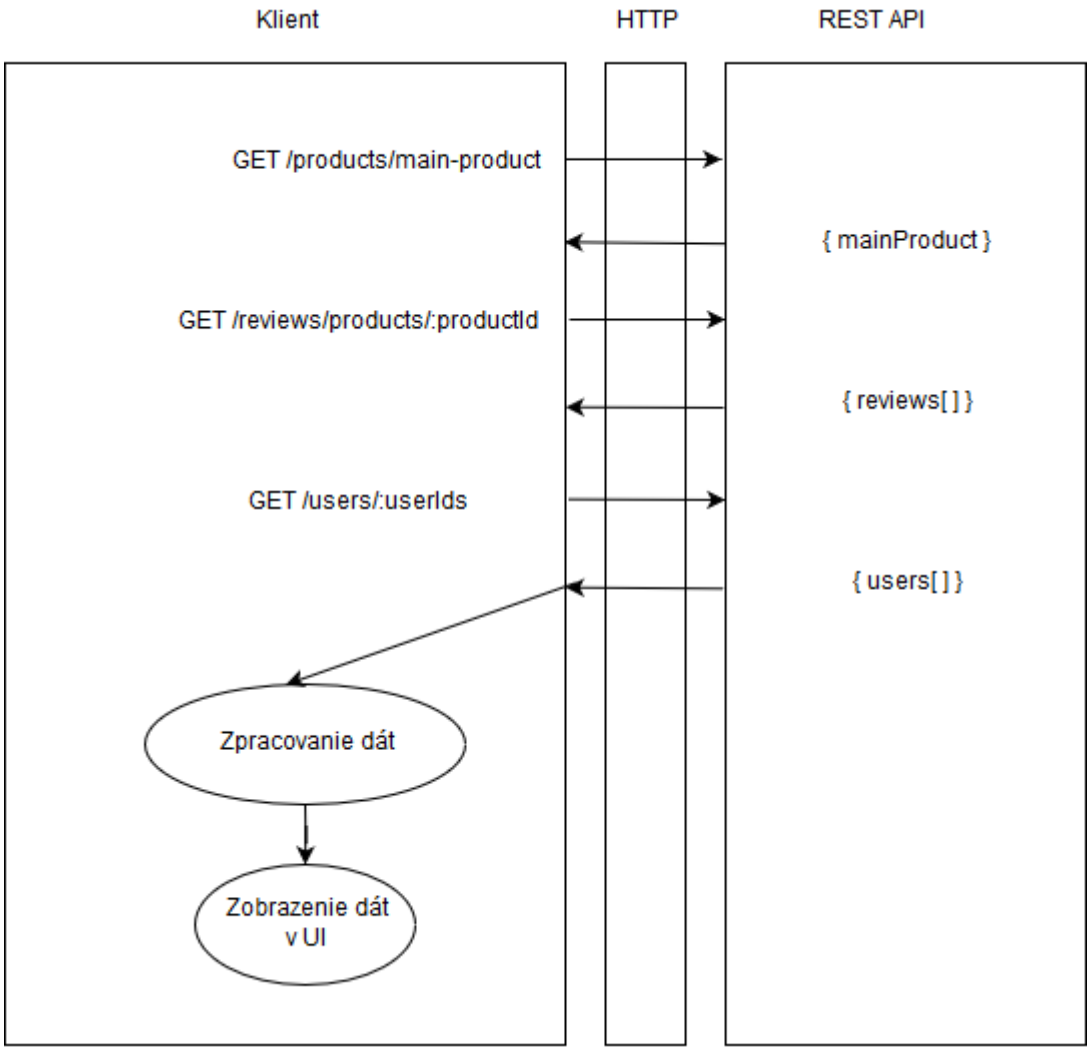
Na prvom obrázku (Obrázok 20) je deklarácia typov používaných pri komunikácií. Na druhom obrázku (Obrázok 22) je dotaz z klienta na server v jazyku GraphQL. Na obrázku (Obrázok 23) je vidieť ďalšiu výhodu GraphQL, schopnosť získať všetky potrebné dáta v rámci jedného dotazu. [25]



```
{
   "mainProduct":{
      "reviews":[
         {
            "rating":10,
            "user": {
               "age":23
            }
         },
         {
            "rating":4,
            "user": {
               "age":60
            }
         },
         {
            "rating":7,
            "user": {
               "age":35
            }
         },

      ]
   }
}
```


Obrázok 23 Odpoveď na hodnotenia a vek hodnotiacich užívateľov

[Zdroj: vlastný]



Obrázok 24. Dotaz na hodnotenia a vek hodnotiacich užívateľov na REST [Zdroj: vlastný]

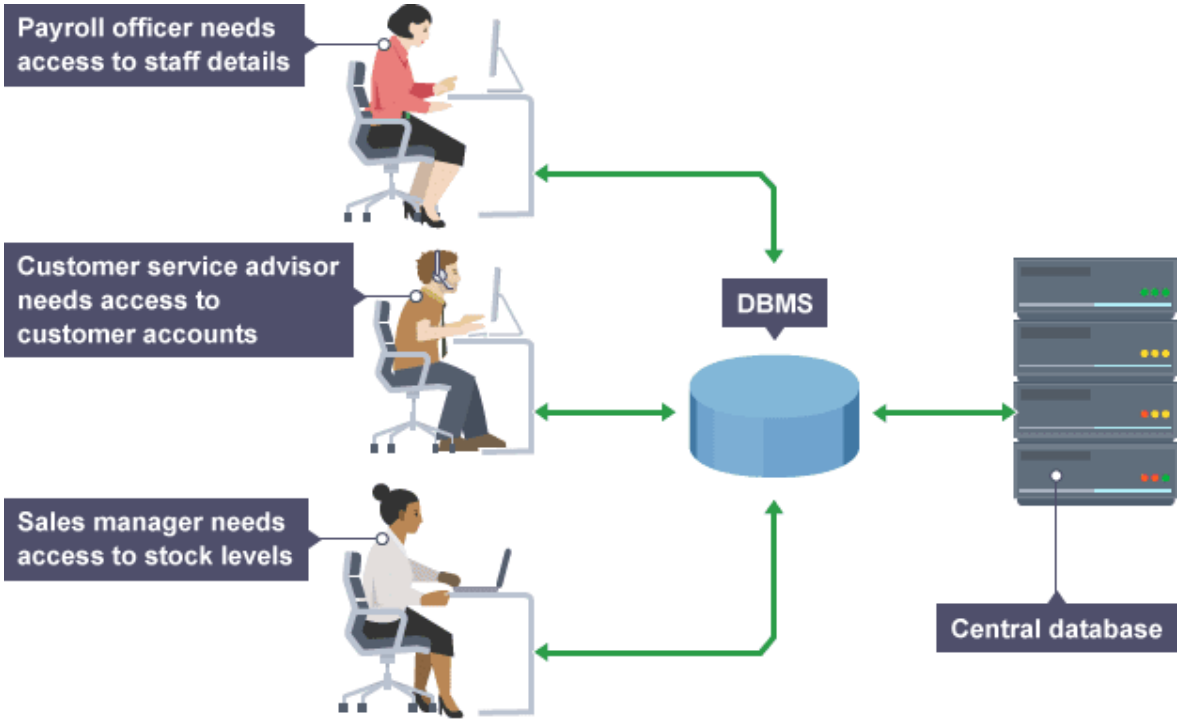
Na obrázku (Obrázok 24) je vidieť získanie rovnakých dát pomocou REST API.

Oproti dotazom na REST API, GraphQL dotaz zvládne všetko čo potrebujú 3 REST dotazy.

Treba poukázať aj na fakt, že v GraphQL dotaze boli vynechané fieldy, ktoré netreba na ďalšie použite u klienta. Toto je obzvlášť dôležité pri dotazovaní sa na zoznam dát, ktorý môže obsahovať desiatky alebo až stovky nadbytočných fieldov.

5 DATA MANAGEMET SOFTWARE

Data management software (DMS) je software, ktorý spracováva dáta a konvertuje ich do jednotného úložiska. Data management systém tiež môže agregovať rozdielne dáta do jednotnej štruktúry ako je napríklad databáza. Vo veľa prípadoch ide o špecifický termín zameniteľný zo širšie používaným termínom, kde data management systém môže smerovať viacero vstupných dát do databáze alebo viacerých databáz.



Obrázok 25. Využívanie data managment systému [26]

Data management systém je druh virtuálnej štruktúry súborov organizujúcich dáta do stromovej štruktúry. Hlavnými jednotkami tejto štruktúry sú metaadresáre, ktoré umožňujú hierarchizovať ostatné objekty a metasúbory. Metasúbory reprezentujú logické zobrazenie výpočtových dát bez ohľadu na ich fyzické umiestnenie. Výpočtové zdroje dát spravované data management systémom môžu byť reprezentované metadátami reprezentujúcimi abstraktný a preskúmateľnej vrstvy zdrojov. [21]

II. PRAKTICKÁ ČÁST

6 ÚVOD DO PRAKTICKEJ ČASTI

Frameworky ako CRUD Admin Generator [35] alebo Forest [36] poskytujú možnosť vygenerovať aplikáciu, v ktorej je možné manažovať dáta, ale buď im chýba dostatočná možnosť konfigurácie, alebo možnosť spolupráce s Microsoft SQL Server. Z tohoto dôvodu sa praktická časť zaoberá vytvorením data management systému, ktorý by umožňoval integráciu s SQL Serverom.

Celkový projekt zastrešujúci aplikáciu je rozdelený do dvoch aplikácií – klient (frontend) a server (backend). Názov projektu je EzAdmin.

Frontend je postavený na knižnici React a je silne ovplyvnený sadou knižníc Apollo GraphQL. Pre zvýšenie kvality kódu je všetok kód napísaný v programovacom jazyku TypeScript, ktorý na rozdiel od JavaScriptu poskytuje typovú kontrolu. React v spojení z TypeScriptom umožňuje písať .tsx komponenty, kde je všetok TypeScriptový kód kontrolovaný statickou analýzou, čo nahradzuje HTML silne typovým jazykom. Túto funkcionalitu v súčasnosti neposkytuje žiadny iný populárny JavaScriptový UI framework.

Backend je napísaný v jazyku F# a postavený na frameworku Giraffe, táto kombinácia je veľmi výhodná pri programovaní bezstavovej komunikácie webového API. Na ukladanie dát, backend využíva NoSQL databázový systém MongoDB.

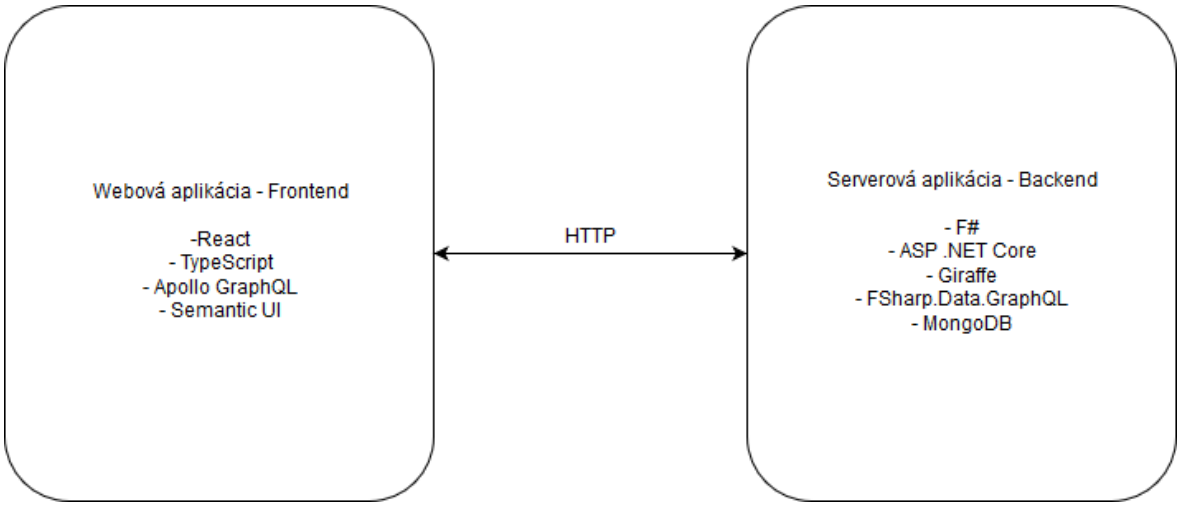
7 ARCHITEKTÚRA APLIKÁCIE

Pri návrhu aplikácie bola zvolená architektúra klient-server. Táto architektúra umožňuje vyššiu modularitu riešenia, čo rozdeľuje zodpovednosť sa získavanie, spracovanie dát a užívateľské rozhranie. V dlhodobom horizonte života aplikácie je takisto výhodou jednoduchšia modernizácia, kedy môže by napríklad klient nahradený novou frontendovou aplikáciou. Rozdelenie na klient-server by zjednodušilo zavedenie na novú platformu, kedy by stačilo na danej platforme vytvoriť klientskú aplikáciu.

Architektúra sa vyznačuje dôrazom použitia čo najmodernejších technológii spolu s dôrazom na písanie bezpečného kódu. Kombinácia zvolených technológií je tak unikátna, čo pri tvorbe prinieslo implementačné výzvy.

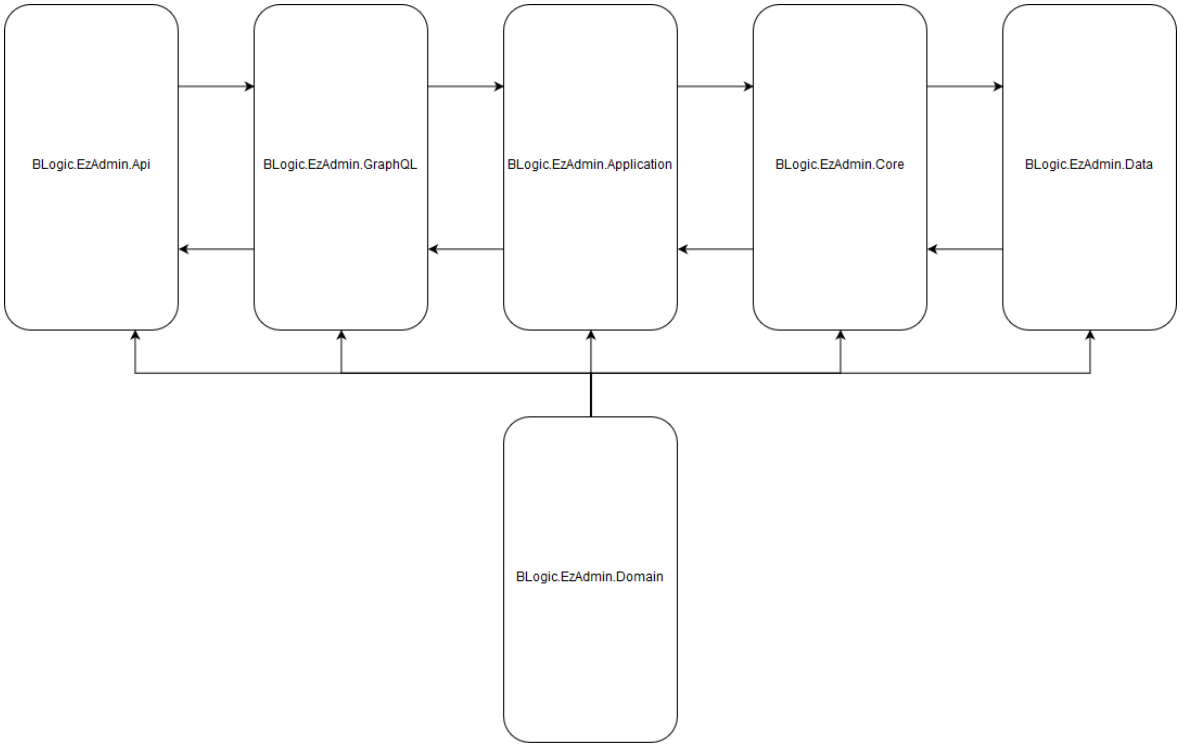
Frontendová časť aplikácie je napísaná v jazyku TypeScript a na zobrazovanie UI využíva knižnicu React. Získavanie dát zo serveru zabezpečuje použitie knižnice Apollo, ktoré spracováva GraphQL dotazy.

Backendová časť je napísaná v F#, je použitý framework Giraffe založený na ASP .NET Core. Na ukladanie schémy sa využíva NoSQL databáza MongoDB.



Obrázok 26. Architektúra aplikácie z praktickej časti [Zdroj: vlastný]

7.1 Architektúra backendu



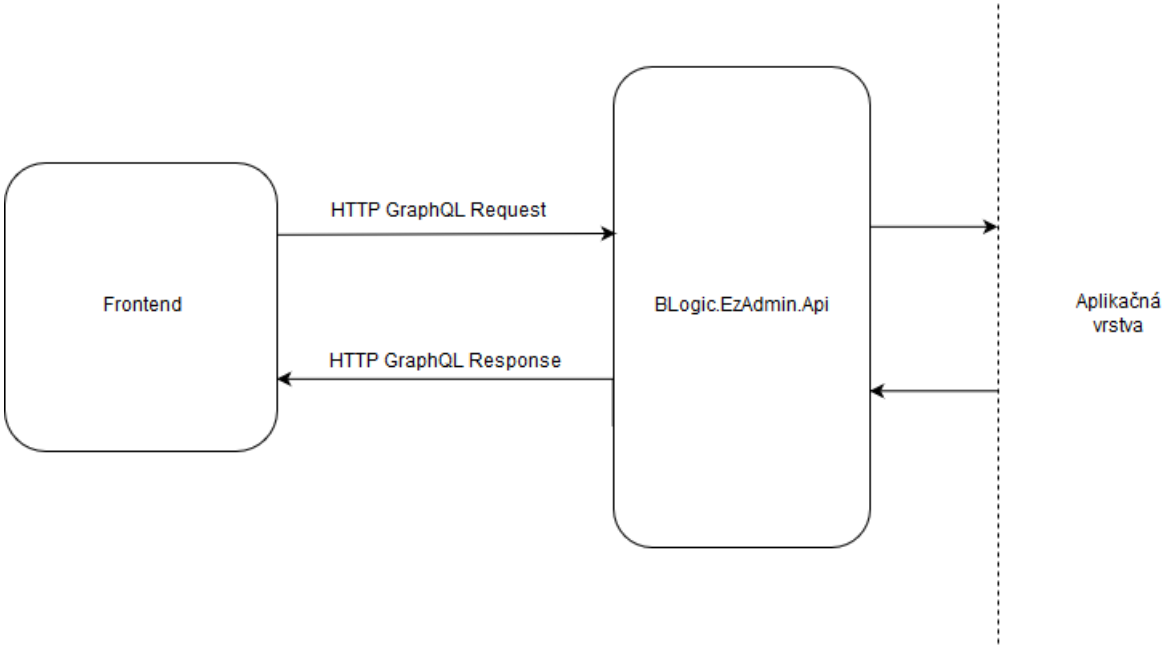
Obrázok 27. Vrstvy backend časti [Zdroj: vlastný]

Backend aplikácie je napísaný v jazyku F# 4.6.2 a postavený na frameworku Giraffe 3.4.0 Giraffe je funkcionálne orientovaný nadstavba nad frameworkom ASP .NET Core 2.1 od Microsoftu.

Architektúra backend časti je vytvorená s viacerých vrstiev. Každá vrstva má svoju zodpovednosť. Usporiadanie vrstiev od vrchu dole je vynútené samotným jazykom. Toto zabraňuje, aby kód z nižšej vrstvy volal kód s vyššej vrstvy. Tento poznatok uľahčuje pochopenie procesov v aplikácií.

Prepojenie aplikácie s klientom, vlastnou databázou a externou databázou je popísané v ďalších kapitolách.

7.1.1 BLogic.EzAdmin.Api



Obrázok 28. BLogic.EzAdmin.Api [Zdroj: vlastný]

BLogic.EzAdmin.Api je najvyššia vrstva aplikácie, ktorá má za úlohu sprostredkovávať komunikáciu s klientom cez HTTP – obsahuje webové API.

Webové API je tenká vrstva, ktorá ma za úlohu prijímanie JSON GraphQL requestu a vrátenie odpovede. API je vytvorené v Giraffe a jeho jednoduchý funkcionálny zápis dokáže definíciu štruktúry API aplikácie definovať na pár riadkov.

Na obrázku (Obrázok 29) je vytvorený objekt reprezentujúci aplikáciu, deklaratívny zápis tu tvrdí, že aplikácia sa skladá z dvoch routes, pričom GraphQL endpoint je dostupný len pre POST requesty.

```
let webApp =
    choose [
        subRoute "/graphql" (choose [ POST >=> graphql ])
        setStatusCode 404 >=> text "Not Found" ]
```

Obrázok 29. Definícia štruktúry API [Zdroj: vlastný]

7.1.2 BLogic.EzAdmin.GraphQL

Za najvyššou vrstvou BLogic.EzAdmin.Api nasleduje BLogic.EzAdmin.GraphQL, ktorá obsahuje schému GraphQL API a spracovanie dotazov z klienta.

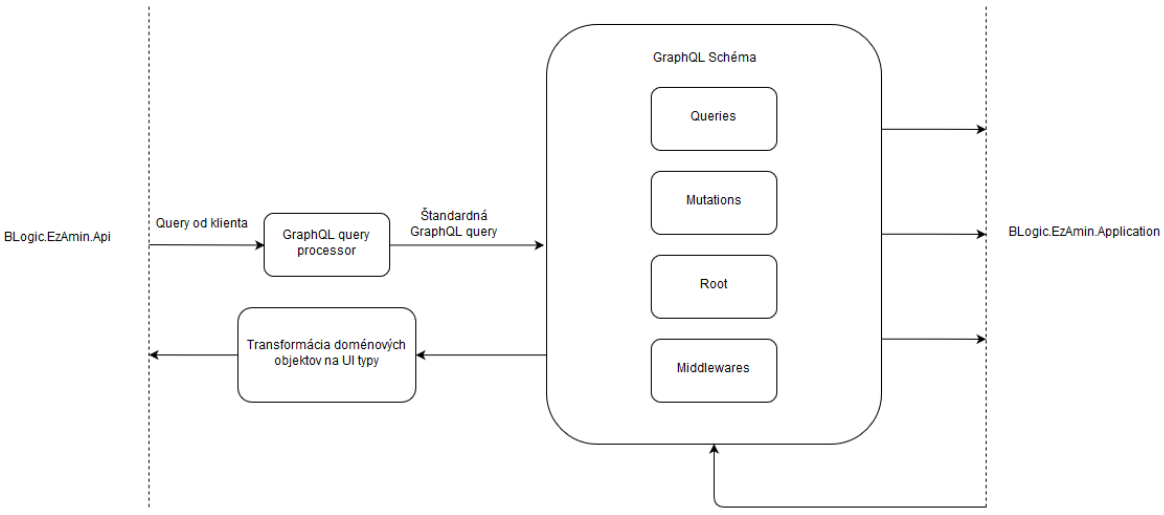
Na definíciu GraphQL schémy sa využíva knižnica FSharp.Data.GraphQL 0.0.18. Táto knižnica umožňuje aj spracovanie objektu predstavujúceho dotaz.

FSharp.Data.GraphQL je F# implementácia GraphQL špecifikácie of Facebooku. Je typovo bezpečná, polia, návratové typy sa kontrolujú počas času kompilácie.

```
let Query =
        Define.Object<Root>(
            name = "Query",
            fields = [
                Define.AuthorizedField("schemas", ListOf (SqlSchemaType), "Get db schemas",
                    fun _ root -> SqlTypesAppService.getAllSchemas root.Token)
            ]
            )
```

Obrázok 30. Definícia poľa typu Query [Zdroj: vlastný]

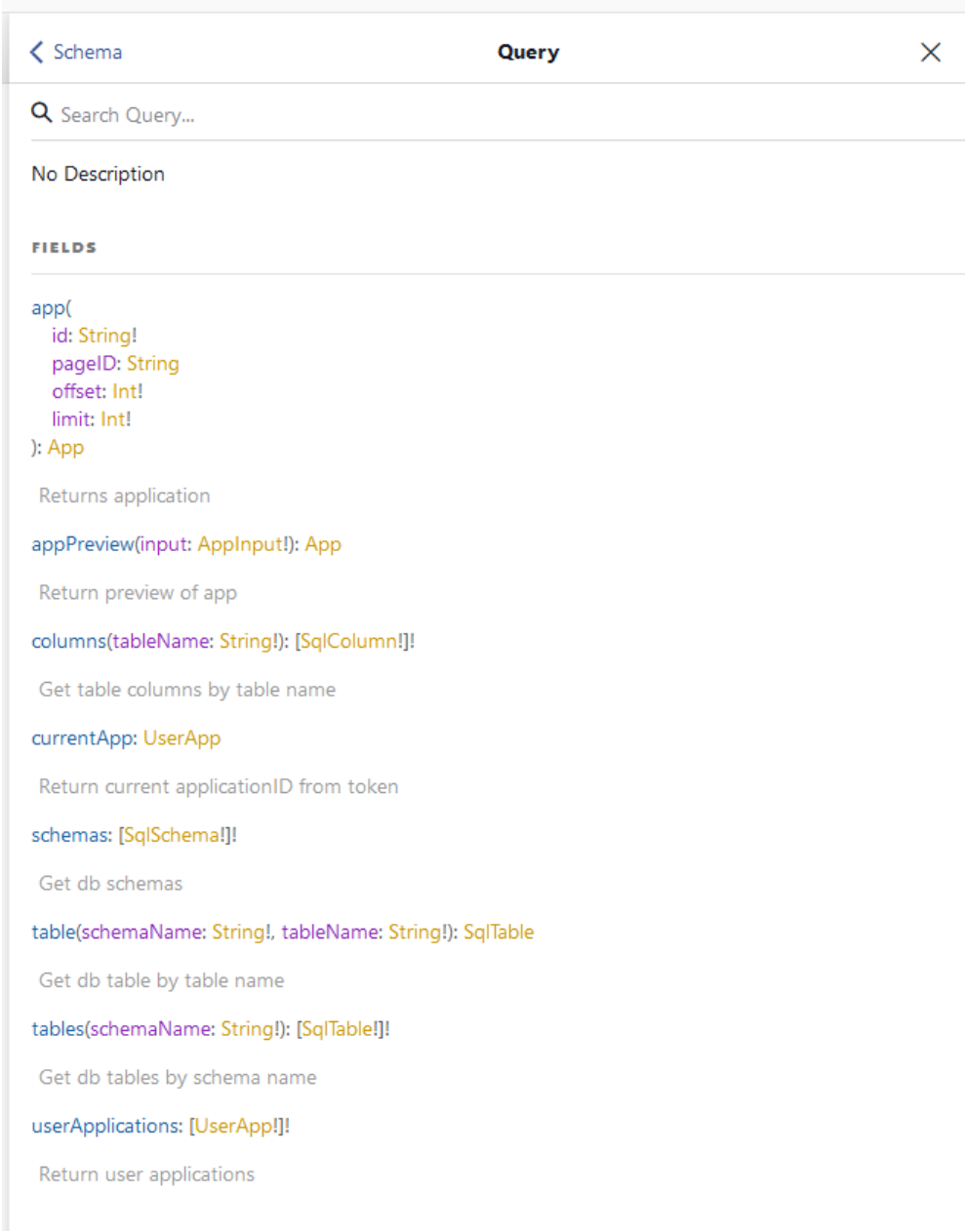
BLogic.EzAdmin.GraphQL rieši aj zabezpečenie zdrojov. Tejto problematike je venovaná špeciálna kapitola.



Obrázok 31. Proces BLogic.EzAdmin.GraphQL [Zdroj: vlastný]

```
let RootType =
        Define.Object<Root>(
            name = "Root",
            description = "The Root type to be passed to all our resolvers",
            isTypeOf = (fun o -> o :? Root),
            fieldsFn = fun () ->
            [
                Define.Field("token", String, "Token of the client",
                fun _ r -> match r.Token with Some v -> v | None -> "")
            ])
```

Obrázok 32. Definícia Root objektu [Zdroj: vlastný]



Obrázok 33. Dokumentácia k GraphQL queries [Zdroj: vlastný]

GraphQL dokáže dynamicky získavať hodnotu filedy na základe requestu. Toto je možné využiť pri väčších objektoch, kedy nie je vždy potreba načítať všetky fieldy. Na obrázku (Obrázok 34) je vidieť dynamické načítavanie zoznamu stĺpcov a referencií do/z tabuľky, toto sa deje asynchrónne.

```
and SqlTableType =
    Define.Object<SqlTable>(
        name = "SqlTable",
        description = "",
        isTypeOf = (fun o -> o :? SqlTable),
        fieldsFn = fun () ->
        [
        Define.Field("tableName", String, "Table name", fun _ (x: SqlTable) -> x.TableName)
        Define.Field("schemaName", String, "Schema name", fun _ (x: SqlTable) ->x.SchemaName)
        Define.Field("columns", ListOf (SqlColumnType), "Columns of table",
            fun ctx (x: SqlTable) -> SqlTypesAppService.getColumns (token ctx) x.TableName)
        Define.Field("referencesToTable", ListOf (SqlReferenceType),
                    "Column references to this table", fun ctx (x: SqlTable) ->
                    SqlTypesAppService.getReferencesToTable (token ctx) x.TableName)
        Define.Field("referencesFromTable", ListOf (SqlReferenceType),
        "Column references from this table to other tables",
            fun ctx (x: SqlTable) -> SqlTypesAppService.getReferencesFromTable (token ctx)
                    x.TableName)
        ]
    )
```

Obrázok 34. Definícia typov s dynamicky resolvovanými fieldami [Zdroj: vlastný]

7.1.3 BLogic.EzAdmin.Application

BLogic.EzAdmin.Application je vrstvou riešiacou získavanie zdrojov vo forme, ktorá môže byť prezentovaná klientovi. Vrstva ďalej rieši overenie a prístup ku zdrojom, ktoré sú zabezpečené podmienkou autorizácie, alebo potrebou mať vybranú vzdialenú databázu.

7.1.4 BLogic.EzAdmin.Core

Táto vrstva je najdôležitejšou časťou aplikácie, primárne sa stará o transformáciu metadát do modelov z ktorých je možné generovať SQL dotazy. Bližší popis je popísaný v kapitole *Engine*.

7.1.5 BLogic.EzAdmin.Data

Repository vrstva zodpovedná za získavanie dát z NoSQL databáze, kde sú uložené schémy aplikácií. Ako NoSQL databáza je použitá MongoDB.

Celá schéma aplikácie sa skladá z veľkého JSON objektu uloženého v MongoDB kolekcii *apps*.

BLogic.EzAdmin.Data sa takisto stará o generovanie a spúšťanie dotazov nad externými databázami.

7.1.6 BLogic.EzAdmin.Domain

Vo vrstve BLogic.EzAdmin.Domain sa nachádzajú všetky definície doménových, UI, SQL, GraphQL a ďalších typov.

Typy sa v F# narozdiel od iných mainstremových imperatívnych jazykov často definujú do jedného súboru. V F# sa jednoduchšie typy definujú na jeden riadok (Obrázok 35).

```
namespace BLogic.EzAdmin.Domain.UiTypes

type [<CLIMutable>] MenuItem = {Name: string; Rank: int; PageID: string}
```

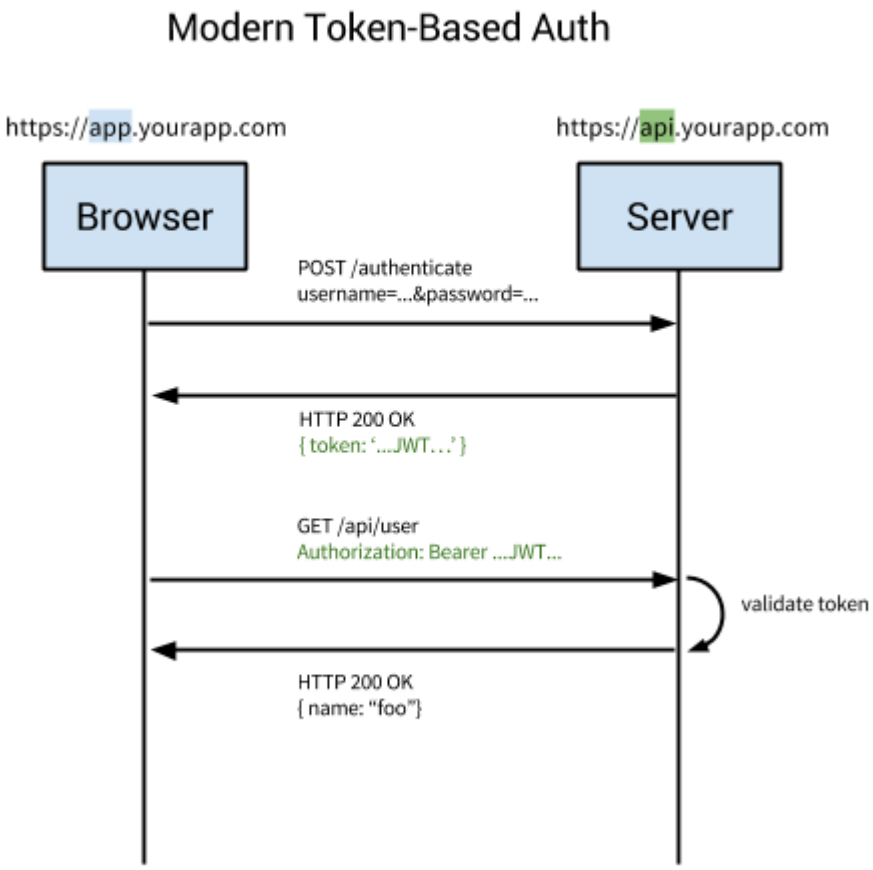
Obrázok 35. Definícia typu MenuItem [Zdroj: vlastný]

7.1.7 Zabezpečenie zdrojov

Pre správne fungovanie aplikácie z praktickej časti je potreba pracovať pod užívateľským účtom, kde si môže vyberať vytvorené aplikácie. Zdroje, s ktorými pracuje teda vyžadujú teda autentifikáciu užívateľa.

7.1.7.1 Autetifikácia pomoucou tokenov

Autentifikácia pomocou tokenov sa stáva čím ďalej používanejšou vďaka rastúcej obľúbenosti SPA (Single Page Applications), webových API (napr. REST, GraphQL) a internetu vecí (IoT). Najčastejším štandardom tokenov je JWT – JSON Web Token. JWT je otvorený štandard, ktorý definuje kompaktný spôsob pre bezpečný presun informácií medzi stranami vo formáte JSON objektu. Informácia je digitálne podpísaná buď pomocou secret s HMAC hashovacieho algoritmu, alebo pomocou verejného a súkromného páru kľúčov s použitím RSA alebo ECDSA. [5]



Obrázok 36. Autentifikačný flow pomocou JWT tokenu [29]

V prvom kroku server požiada klienta o prihlasovacie údaje – napr. pomocou prihlasovacieho formuláru.

V druhom kroku klient svoje prihlasovacie údaje zašle na server pomocou metódy POST. Ak sú správne, tak server vytvorí token a vráti ho klientovi. Klient si token uloží.

V následných krokoch klient zasiela token spolu s requestom – najčastejšie v Authorization header. Server dekóduje JWT token a ak je validný spracuje request.

Keď sa užívateľ odhlási, token je zničený na strane klienta a komunikácia zo serverom nie je potrebná.

7.1.7.2 Štruktúra JWT tokenov

Header

Header sa typicky skladá z dvoch častí – hashovací algoritmus, ktorý bol použitý a potom typ tokenu – u JWT je to vždy JWT. [30]

Payload

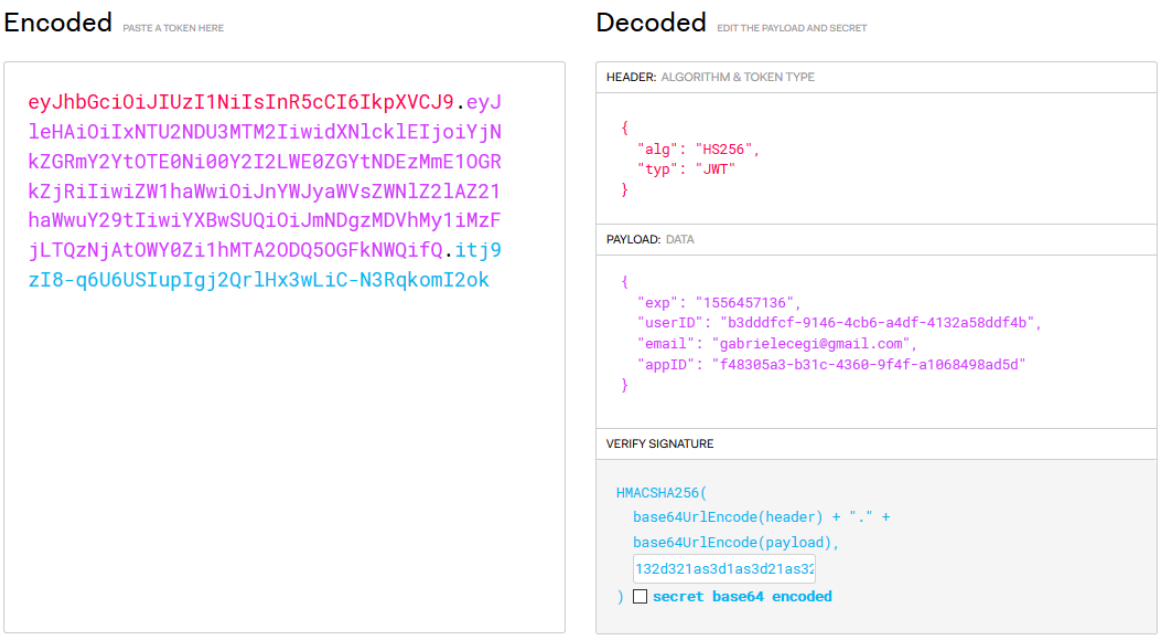
Je druhou časťou tokenu a obsahuje takzvané claims. Claims sú nároky užívateľa a dodatočné dáta. Claims sa rozdelujú do troch skupín

- Registered claims – sú preddefinované štandardom a nie sú povinné alebo implementované pre všetky prípady
- Public claims – vytvorené používateľmi JWT tokenov
- Private claims – dohodnuté medzi klientom a serverom, nenachádzajú sa medzi registered a public cliams

Payload je následne zakódovaný do formátu Base64Url, preto nie je odporúčané do payloadu vkladať citlivé informácie. [30]

Signature

Na vytvorenie signature je potreba header a payload zakódované v base64 a secret. Secret je raťazec znakov uložený na serveri, ktorý sa v žiadnom prípade nesmie dostať do nesprávnych rúk. Preto by mal byť uložený mimo kódu, resp. mimo verzovacieho systému. Ako vidieť na obrázku (Obrázok 37) bol využitý hashovací algoritmus HMACSHA256. Signature sa používa na potvrdenie, že správa nebola zmenená a keď sú tokeny podpísané súkromným kľúčom, môžeme potvrdiť, že odosielateľ je ten za koho sa vydáva. [31]



Obrázok 37. Štruktúra JWT tokenu z praktickej časti [Zdroj: vlastný]

7.1.7.3 Integrácia JWT autentifikácie do GraphQL

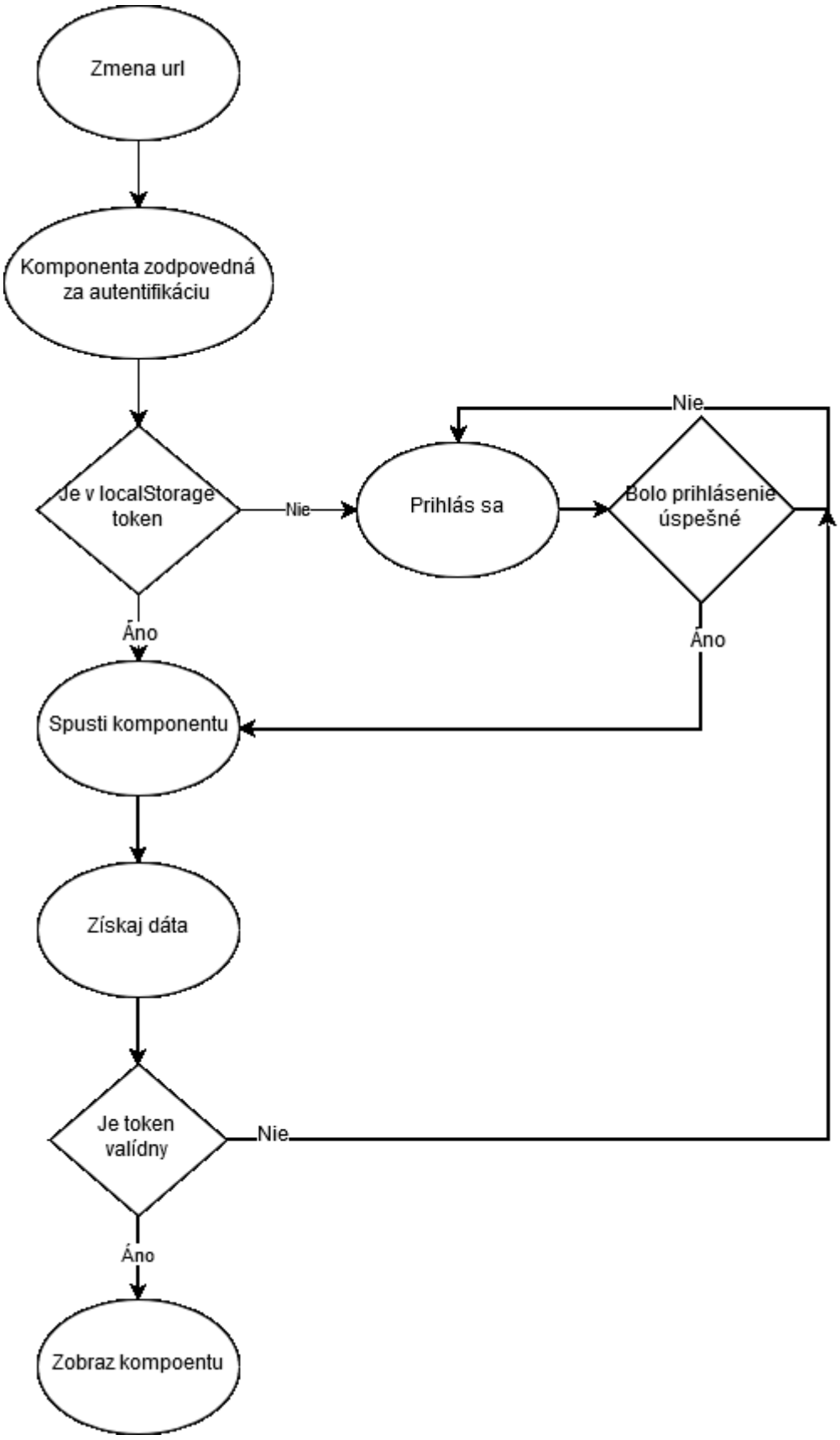
GraphQL v .NET nie je príliš rozšírené, jeden z problémov je to že C# je z veľkej časti OOP. Toto príliš nesedí s funkcionálnou povahou GraphQL. Na druhej strane F#, funkcionálny ale oveľa menej používaný jazyk, je pre GraphQL najlepšou voľbou v prostredí .NET.

GraphQL samotné vôbec nerieši autentifikačný flow a fórum GraphQL knižnice pre F# neponúka rady pre tento problém. Navyše nestačí použitie atribútu Authorize, pretože GraphQL môže obsahovať query fieldy, u ktorých sa treba autentifikovať, ale obsahuje aj fieldy, ktoré môže spustiť aj neprihlásená osoba.

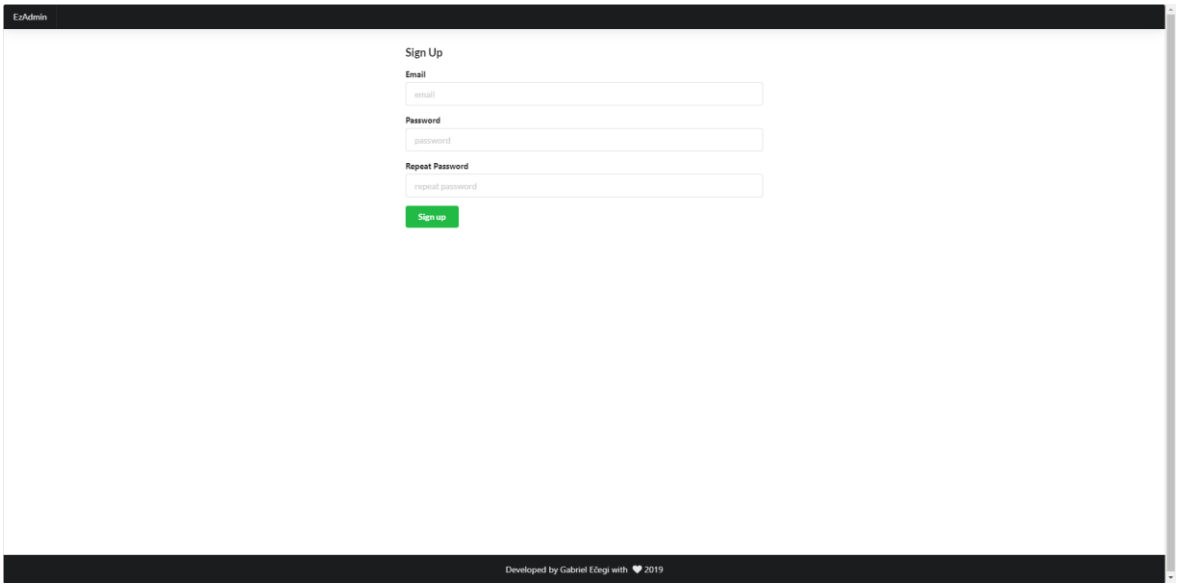
Riešením je pridať nový typ fieldu, ktorý bude validovať či sa token nachádza v hlavičke requestu a ak áno či je validný.

```
Define.AuthorizedField(
    "setAppID",
    LoginResultType,
    "If succesfull returns token",
    [ Define.Input("appID", String); ],
    fun ctx root ->
        ctx.Arg("appID")
        |> TokenService.setAppID root.Token
        |> (fun e -> {  Token = e |>
                                Option.bind (fun token -> sprintf "Bearer %s" token |> Some);
                        ValidationMessage = None})
                    );
```

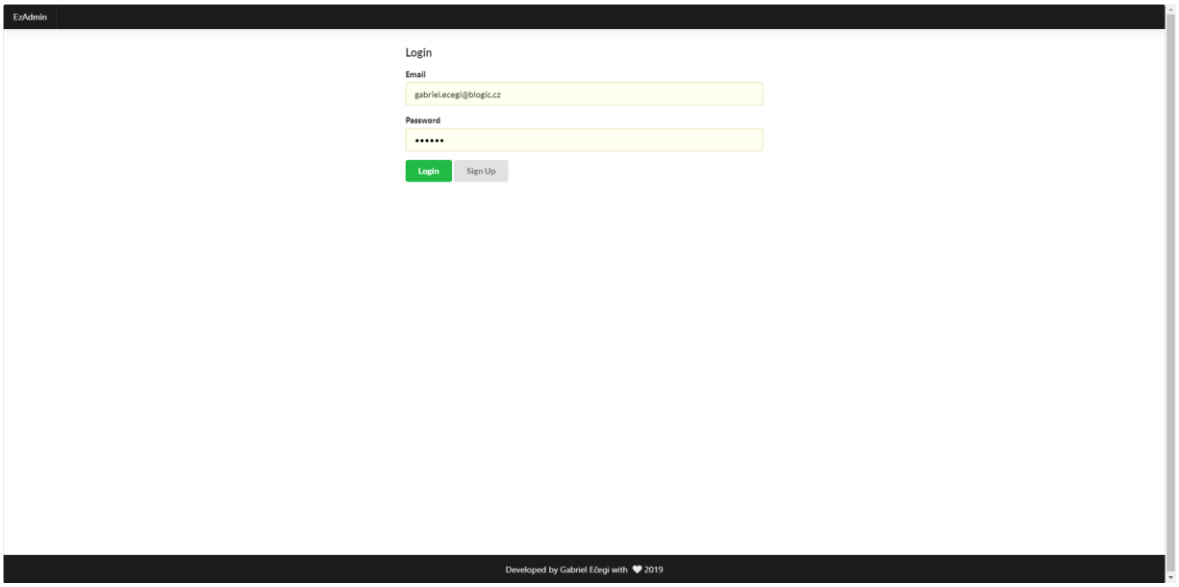
Obrázok 38. Zabezpečený field prestavujúci mutáciu [Zdroj: vlastný]



Obrázok 39. Autentifikačný flow aplikácie [Zdroj: vlastný]



Obrázok 40. Registračný formulár [Zdroj: vlastný]



Obrázok 41. Prihlasovací formulár [Zdroj: vlastný]

7.2 Architektúra frontendovej časti

Frontendová aplikácia je postavená na obľúbenej šablóne Create React App 3.0.1, ktorá umožňuje vygenerovať kompletnú React aplikáciu (verzia 16.8.6) aplikáciu s podporou TypeScriptu 3.4.5, ktorú je možné okamžite spustiť a vytvoriť build produkčnej verzie. Celá architektúra je silne ovplyvnená použitím Apollo GraphQL stacku.

7.2.1 Apollo GraphQL

Apollo GraphQL je sada knižníc umožňujúca integráciu GraphQL do projektu. Projekt z praktickej časti využíva knižnice pre klientské aplikácie ako napríklad *react-apollo 2.5.5, apollo-link-http 1.5.14, apollo-cache-inmemory 1.5.1*. Kombináciou týchto knižníc je možné vytvoriť vo frentodovej aplikácií robustnú silnotypovú komunikáciu s backendom.

7.2.1.1 Generovanie typov

Častým problémom moderných backendových a frontendových aplikácií je zdieľanie definícií typov dátových štruktúr. Typy, ktoré sú vytvorené v backendovej aplikácií sú často napísané v inom programovacom jazyku. Riešením na prenos typov do frontendovej aplikácie môžu byť konvertory medzi jazykmi, ktoré sú však nespoľahlivé a nie všetky typy sú potrebné.

GraphQL ponúka ideálne riešenie. Vďaka dostupnej GraphQL schéme z backendu je možné získať predstavu o štruktúre API - je teda možné algoritmizovať prehľadávanie backendových typov. Toto presne umožňuje npm balíček Apolo CLI 2.11.1, ktorý spája klientskú a servrovú aplikáciu s nástrojmi na validáciu, kompatibilitu a generovanie statických typov pre typovú bezpečnosť.

Frontendová časť komunikuje s backendom kompletne cez pomocou GraphQL, to znamená, že z kódu všetkej komunikácie (queries, mutations), uloženého na frontende, je možné vygenerovať statické typy pre komunikáciu zo serverom.

Prvým krokom je vytvorenie typu výsledku komunikácie na serveri (Obrázok 45). Výsledok je *LoginResult* obsahujúci dva fieldy – *token* a *validationMessage*.

```
LoginResultType =
        Define.Object<LoginResult>(
            name = "LoginResult",
            description = "",
            isTypeOf = (fun o -> o :? LoginResult),
            fieldsFn = fun () ->
            [
                Define.Field("token", Nullable(String), "Token",
                             fun _ (x: LoginResult) -> x.Token)
                Define.Field("validationMessage", Nullable(String), "Validation message",
                             fun _ (x: LoginResult) -> x.ValidationMessage)
            ]
        )
```

Obrázok 42. GraphQL typ deklarovaný na serveri [Zdroj: vlastný]

V backendovej aplikácií treba vytvoriť GraphQL endpoint – mutáciu, predstavujúcu prihlásenie. Endpoint má názov *login* a obsahuje dva parametre typu *String* – *email* a *password*.

```
Define.Field(
                    "login",
                    LoginResultType,
                    "If succesfull returns token",
                    [ Define.Input("email", String); Define.Input("password", String) ],
                    fun ctx _ ->
                            let email = ctx.Arg("email")
                            let password = ctx.Arg("password")
                            SecurityAppService.login email password
                    );
```

Obrázok 43. Definícia mutácie – prihlásenie sa [Zdroj: vlastný]

Vo frontendovej aplikácií je možné napísať mutáciu a nazvať ju *LoginMutation*.

```
mutation LoginMutation($email: String!, $password: String!) {
  login(email: $email, password: $password) {
    token
    validationMessage
  }
}
```

Obrázok 44. Frontend kód GraphQL mutácie [Zdroj: vlastný]

Po spustení nástroja je vygenerovaný kód predstavujúci typy potrebné k staticky typovej komunikácií zo serverom. Obsahuje mutáciu samotnú, návratový typ operácie a dokonca typy parametrov, vďaka čomu sa uľahčí používanie API.

```
/* tslint:disable */
/* eslint-disable */
// This file was automatically generated and should not be edited.

// ====================================================
// GraphQL mutation operation: LoginMutation
// ====================================================

export interface LoginMutation_login {
  __typename: "LoginResult";
  /**
   * Token
   */
  token: string | null;
  /**
   * Validation message
   */
  validationMessage: string | null;
}

export interface LoginMutation {
  /**
   * If succesfull returns token
   */
  login: LoginMutation_login;
}

export interface LoginMutationVariables {
  email: string;
  password: string;
}
```

Obrázok 45. Vygenerovaný kód [Zdroj: vlastný]

Ak v backendovej časti nastane zmena a parameter mutácie bude premenovaný na *login*, tak pri kompilácia frontendovej časti skončí s chybou (Obrázok 44).

```
...EzAdmin/client/src/graphql/mutations/Auth/LoginMutation.ts:
Unknown argument "email" on field "login" of type "Mutation".
...EzAdmin/client/src/graphql/mutations/Auth/LoginMutation.ts:
Field "login" argument "login" of type "String!" is required, but it was not provided.
{ ToolError: Validation of GraphQL query document failed name: 'ToolError' }
```

Obrázok 46. Chyba kompilácie [Zdroj: vlastný]

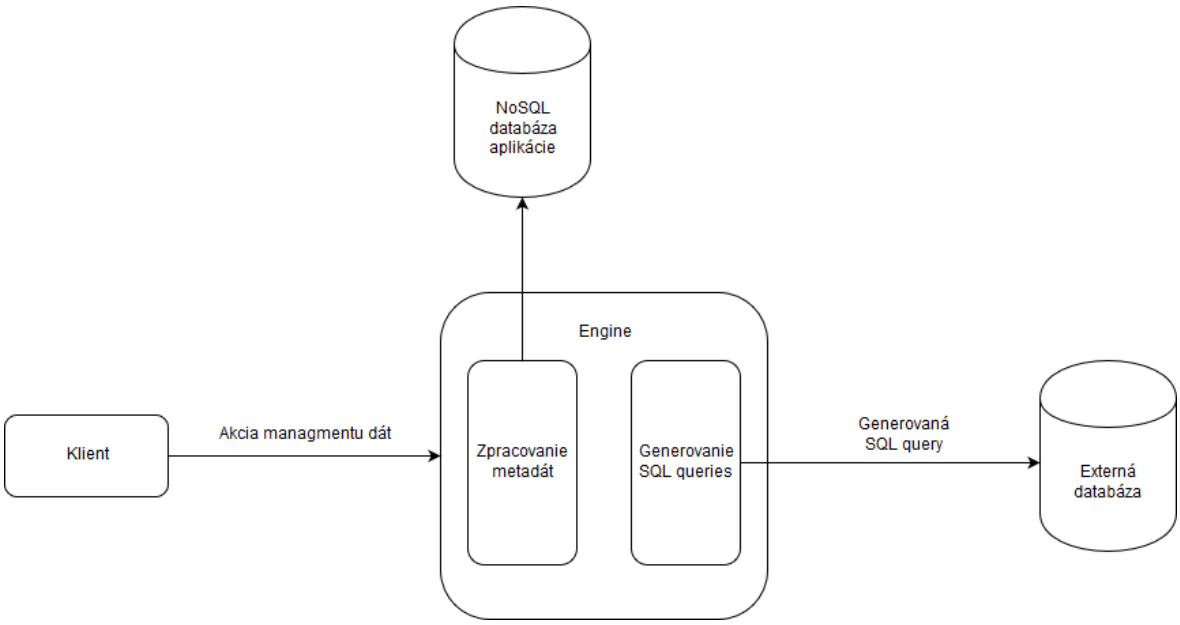
Čo sa na prvý pohľad môže zdať ako chyba a prekážka pri vývoji, je v skutočnosti veľmi dobrý pomocník pri odchytávaní chýb. Bez generovaných typov by bola totiž daná chyba odchytená až v runtime behu aplikácie. Prípadne by sa vyskytla až neskôr ako regresná chyba.

8 SPRACOVANIE DÁT A METADÁT

Najdôležitejšou časťou aplikácie je časť menom Engine. Engine je jadrom, ktoré má na starosti generovanie dynamických dotazov, prácu zo schémou, metadátami a generovaním dotazov pracujúcich s dátami. Moduly, tvoriace Engine sa nachádzajú vo vrstvách BLogic.EzAdmin.Core a BLogic.EzAdmin.Data.

S Enginom pracujú všetky klientské dotazy, ktoré nejakým spôsobom intergujú s externou databázou. Ide tak o všetky akcie, ktoré manageujú dáta – zobrazenie, úprava a zmazanie.

Engine dokáže s externou databázou komunikovať vďaka informáciám získaným z Connection Stringu, ktorý je zadaný pri vytvorení aplikácie. Engine je schopný generovať SQL dotazy, ktorú sú následne spúšťané nad externou databázou.



Obrázok 47. Spolupráca Enginu, klienta, aplikačnej databáze a externej databáze [Zdroj: vlastný]

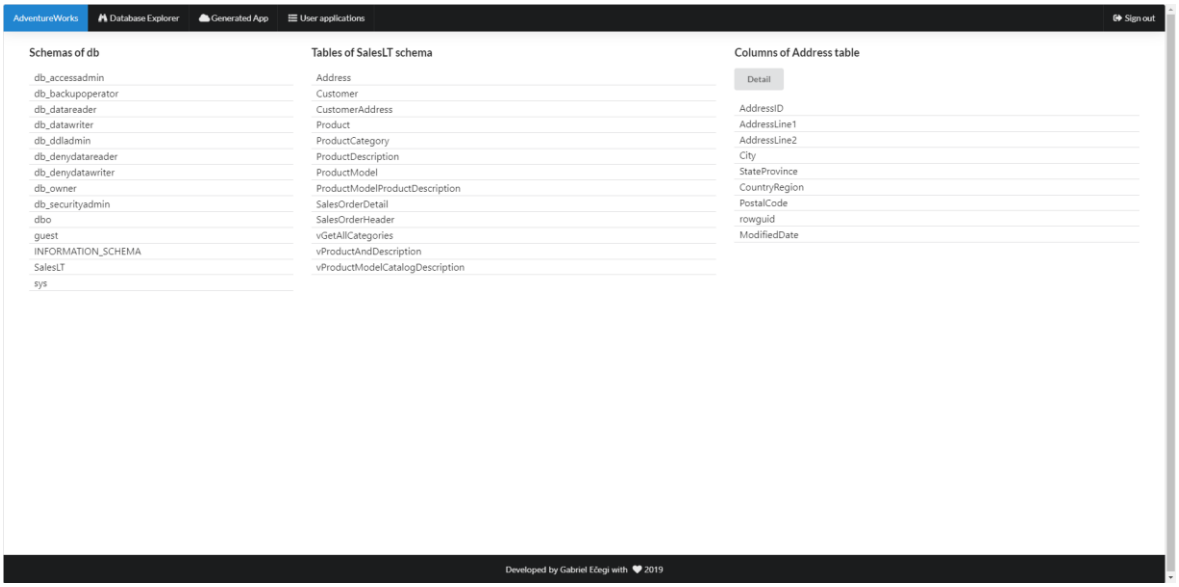
8.1 Získanie metadát

```
exec sp_executesql N'
  SELECT
  [TABLE_SCHEMA] SchemaName,
  [TABLE_NAME] TableName,
  [COLUMN_NAME] ColumnName,
  [DATA_TYPE] DataType,
  CAST(
    CASE WHEN EXISTS(SELECT
      i.name AS IndexName,
      OBJECT_NAME(ic.OBJECT_ID) AS TableName,
      COL_NAME(ic.OBJECT_ID,ic.column_id) AS ColumnName,
      s.name AS [SchemaName]
      FROM sys.indexes AS i
      INNER JOIN sys.index_columns AS ic
      ON i.OBJECT_ID = ic.OBJECT_ID
      AND i.index_id = ic.index_id
      JOIN sys.objects o ON o.object_id = ic.object_id
      INNER JOIN sys.schemas s ON o.schema_id=s.schema_id

      WHERE i.is_primary_key = 1 AND s.name = [TABLE_SCHEMA] AND OBJECT_NAME(ic.OBJECT_ID) =
[TABLE_NAME] AND  COL_NAME(ic.OBJECT_ID,ic.column_id) = [COLUMN_NAME])
        THEN 1
        ELSE 0
      END
  AS BIT) AS IsPrimaryKey  FROM INFORMATION_SCHEMA.COLUMNS

  WHERE [TABLE_NAME] = @TableName
  ',N'@TableName nvarchar(4000)',@TableName=N'Customer'
```

Obrázok 48. Získanie metadát z SQL serveru [Zdroj: vlastný]

SQL server má skupinu tabuliek udržiavajúcich informácie o všetkých objektoch, dátových typoch, nastaveniach a zdrojoch dostupných SQL serveru. Tieto tabuľky sa nazývajú *system base tables*. Pohľady (Views) informačných schém boli pôvodne pohľady na metadáta serveru nezávislé na systémových tabuľkách. Tieto pohľady sa nachádzajú v schéme INFORMATION_SCHEMA. [34]

Vďaka týmto metadátam je možné z externej databáze vybrať informácie o databázových schémach, tabuľkách, stĺpcoch a indexoch. Z indexov sa dajú zistiť relácie medzi tabuľkami.



Obrázok 49. Zobrazenie schém, tabuliek a stĺpcov [Zdroj: vlastný]

8.2 Náhľad generovanej aplikácie

V dizajnér aplikácie (Obrázok 51) je možné vybrať stĺpce, ktoré budú zobrazené v aplikácií na danej stránke. Stĺpce môžu byť vybrané aj z iných tabuliek, ktoré sú s hlavnou tabuľkou prepojené.

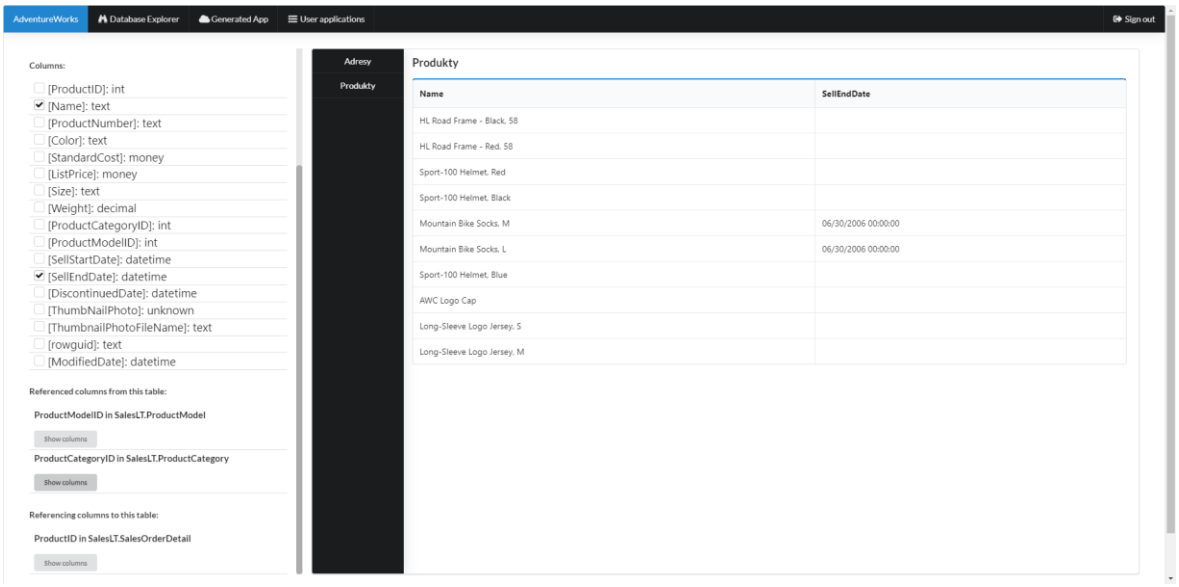
O vybratie správnych dát sa stará časť modulu Engine, ktorá na základe vybraných stĺpcov spojí informácie z metadát a vygeneruje SQL query (Obrázok 50).

Pri každej akcii v dizajnér je odoslaný dotaz na server, kde je ako odpoveď vrátený nový náhľad aplikácie (Obrázok 52). Toto umožňuje naživo modelovať aplikáciu.

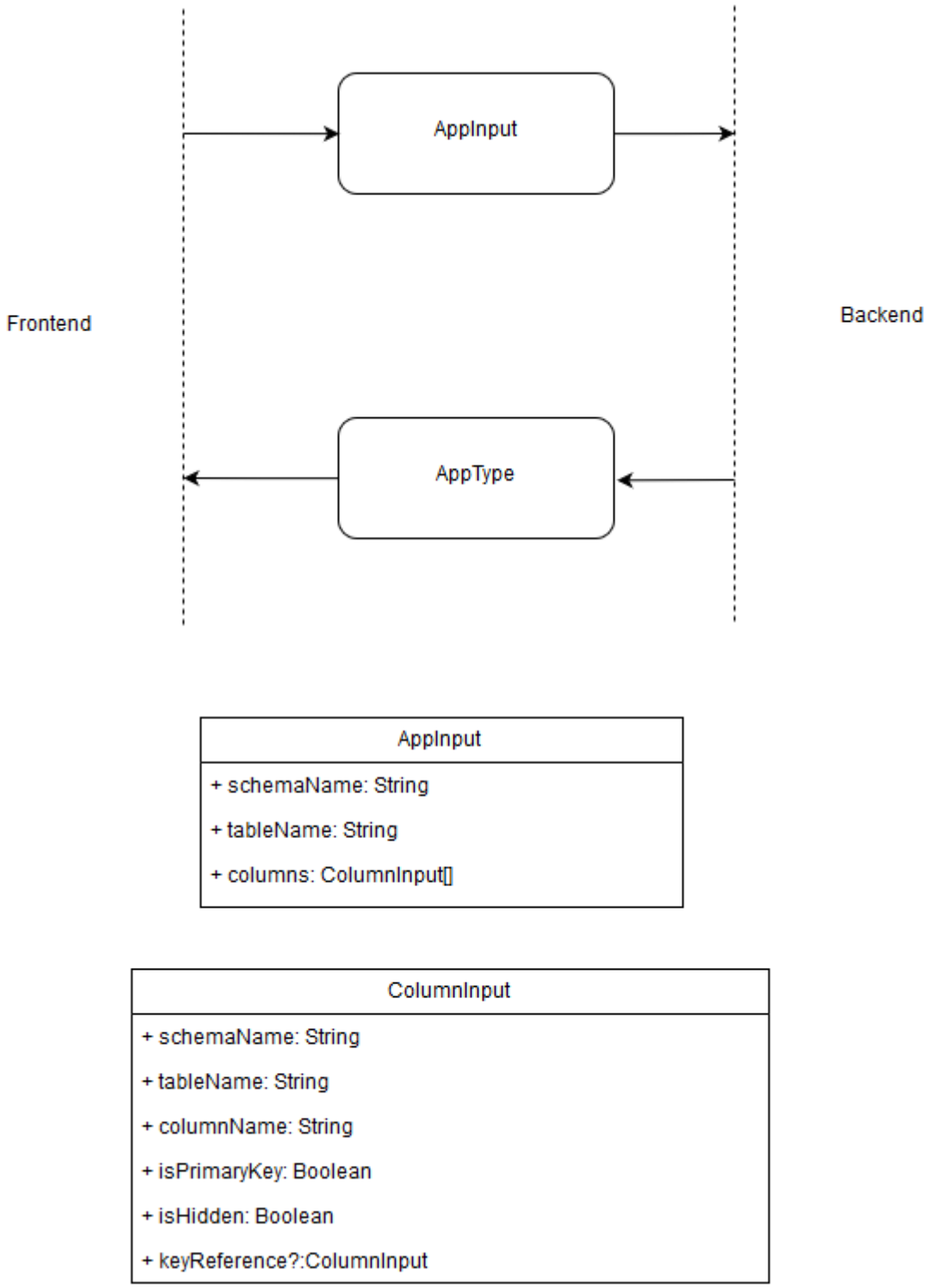
Po ukončení modelovania stránky je možné danú stránku uložiť, týmto bude upravená schéma aplikácie a užívateľ bude presmerovaný do vygenerovanej aplikácie, kde môže zobrazovať, upravovať a editovať dáta.

```
SELECT [MainTable].CityID AS MainTable_CityID
,[MainTable].Name AS MainTable_Name
,[T0].Name AS T0_Name
,[T1].Name AS T1_Name
,[T2].Name AS T2_Name
 FROM App.Cities AS [MainTable]
JOIN App.Regions [T0] ON [MainTable].RegionID = [T0].RegionID
JOIN App.Countries [T1] ON [T0].CountryID = [T1].CountryID
JOIN App.Languages [T2] ON [T1].LanguageCode = [T2].Code
ORDER BY MainTable_CityID
OFFSET 0 ROWS
FETCH NEXT 10 ROWS ONLY
```

Obrázok 50. Vygenerovaná SQL query [Zdroj: vlastný]



Obrázok 51. Dizajnér aplikácie [Zdroj: vlastný]



Obrázok 52. Komunikácia pre zobrazenie náhľadu aplikácie [Zdroj: vlastný]

8.3 Schéma aplikácie


```
{
    "_id": ObjectId("5cbd820775c9cb186402c0c8"),
    "UserID": ObjectId("5cbce052f616d9a0c803f374"),
    "Name": "AdventureWorks",
    "Connection": "Data Source=localhost;Initial Catalog=AdventureWorksLT2016;Integrated
Security=True",
    "MenuItems": {
        "_t": "List`1",
        "_v": [
            {
                "_id": ObjectId("5ccb2fc96ab9329b7c3c25c9"),
                "Name": "Adresy",
                "Rank": 10,
                "PageID": ObjectId("5ccb2fc96ab9329b7c3c25bf")
            },
            {
                "_id": ObjectId("5ccc7eed6ab9329b7c3c2667"),
                "Name": "Produkty",
                "Rank": 20,
                "PageID": ObjectId("5ccc7eed6ab9329b7c3c265d")
            }
        ]
    },
    "Pages": {
        "_t": "List`1",
        "_v": [
            {
                "_id": ObjectId("5ccb2fc96ab9329b7c3c25bf"),
                "Name": "Adresy",
                "Table": {
                    "_id": ObjectId("5ccb2fc96ab9329b7c3c25c8"),
                    "SchemaName": "SalesLT",
                    "TableName": "Address",
                    "Columns": {}
                }
            },
            {
                "_id": ObjectId("5ccc7eed6ab9329b7c3c265d"),
                "Name": "Produkty",
                "Table": {
                    "_id": ObjectId("5ccc7eed6ab9329b7c3c2666"),
                    "SchemaName": "SalesLT",
                    "TableName": "Product",
                    "Columns": {}
                }
            }
        ]
    }
}
```


Obrázok 53. Schéma aplikácie zo skrytými stĺpcami [Zdroj: vlastný]

Schéma aplikácie (Obrázok 53) je uložená ako JSON object v NoSQL databázy.

Schéma obsahuje tieto fieldy:

UserID: ID užívateľa vlastniaceho aplikáciu

Name: Názov aplikácie

Connection: Pripojenie k databázy

MenuItems: Predstavujú položky v ľavom menu, tieto sú klikateľné a odkazujú na objekt Page

Pages: Sú to jednotlivé stránky v aplikácii. Každá stránka má svoj *Table* (Obrázok 54)

Table: Predstavuje tabuľku ktorú je možné zobraziť v aplikácií

Column: Aby mohol byť *Table* vyrendrovaný, je potreba mať zoznam stĺpcov, ktoré sa dajú zobraziť. Na tento účel slúži field *Columns*, predstavujúci stĺpce. Každý stĺpec môže mať referenciu na ďalší stĺpec. Typicky sa jedná o stĺpec z tabuľky, ktorý sa odkazuje na primárny kľúč. V prípade že sa primárny kľúč odkazuje na iný primárny kľúč, jedná sa o prepojenie tabuliek.

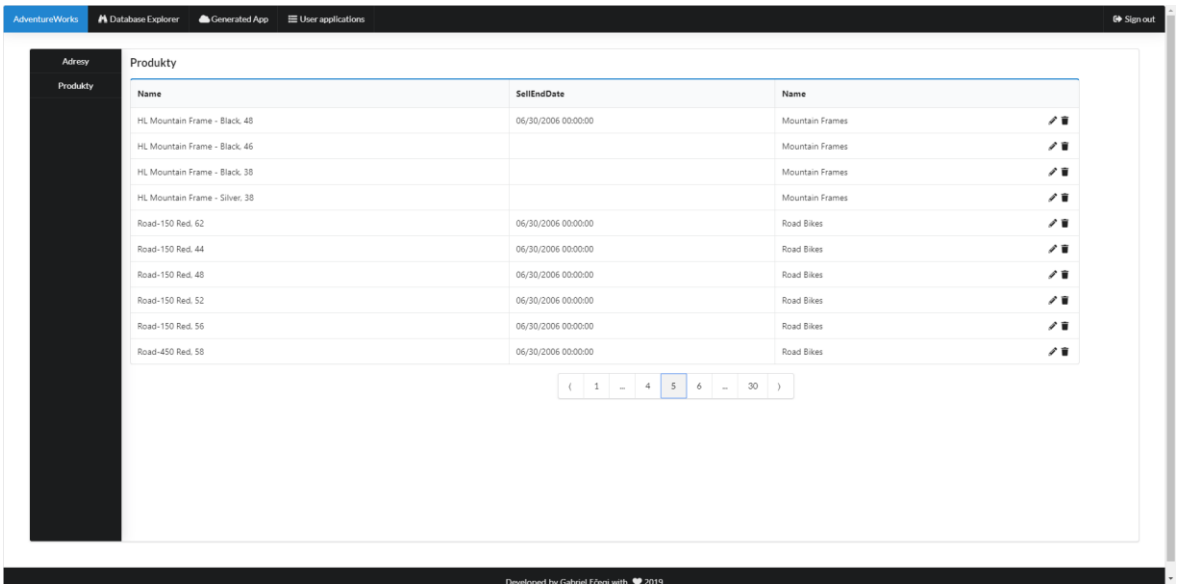
```
"Pages": {
        "_t": "List`1",
        "_v": [

            {
                "_id": ObjectId("5ccc7eed6ab9329b7c3c265d"),
                "Name": "Produkty",
                "Table": {
                    "_id": ObjectId("5ccc7eed6ab9329b7c3c2666"),
                    "SchemaName": "SalesLT",
                    "TableName": "Product",
                    "Columns": {
                        "_t": "List`1",
                        "_v": [
                            {
                                "_id": ObjectId("5ccc7eed6ab9329b7c3c2662"),
                                "ColumnName": "Name",
                                "TableName": "ProductCategory",
                                "SchemaName": "SalesLT",
                                "ColumnType": {
                                    "Case": "Column",
                                    "Fields": []
                                },
                                "IsHidden": false,
                                "Reference": {
                                    "_id": ObjectId("5ccc7eed6ab9329b7c3c2663"),
                                    "ColumnName": "ProductCategoryID",
                                    "TableName": "ProductCategory",
                                    "SchemaName": "SalesLT",
                                    "ColumnType": {
                                        "Case": "ForeignKey",
                                        "Fields": []
                                    },
                                    "IsHidden": true,
                                    "Reference": {
                                        "_id": ObjectId("5ccc7eed6ab9329b7c3c2664"),
                                        "ColumnName": "ProductCategoryID",
                                        "TableName": "Product",
                                        "SchemaName": "SalesLT",
                                        "ColumnType": {
                                            "Case": "Column",
                                            "Fields": []
                                        },
                                        "IsHidden": true,
                                        "Reference": {
                                            "_id": ObjectId("5ccc7eed6ab9329b7c3c2665"),
                                            "ColumnName": "ProductID",
                                            "TableName": "Product",
                                            "SchemaName": "SalesLT",
                                            "ColumnType": {
                                                "Case": "PrimaryKey",
                                                "Fields": []
                                            },
                                            "IsHidden": true
                                        }
                                    }
                                }
                            }
                        ]
                    }
                }
            }
        ]
    }
}
```


Obrázok 54. Ukážka časti schéme reprezentujúcej stránku [Zdroj: vlastný]

8.4 Zobrazovanie dát

Po vybratí tabuliek a ich stĺpcov, ktoré chceme zobraziť, je k dispozícii už vygenerovaná aplikácia. Tu môžeme dáta stránkovať a zobrazovať.

O generovanie SQL dotazov sa stará Engine, ktorý podobným spôsobom ako pri náhľade tabuľky v Dizajnéri spracuje metadáta a vygeneruje príslušnú SQL query zo stránkovaním. (Obrázok 55)



Obrázok 55. Zobrazenie dát vo vygenerovanej aplikácií [Zdroj: vlastný]

8.5 Mazanie dát

Schéma uložené v NoSQL databázy uchováva informácie o identifikátoroch entít, na základe toho je systéme schopný mazať záznamy z vygenerovanej aplikácie.

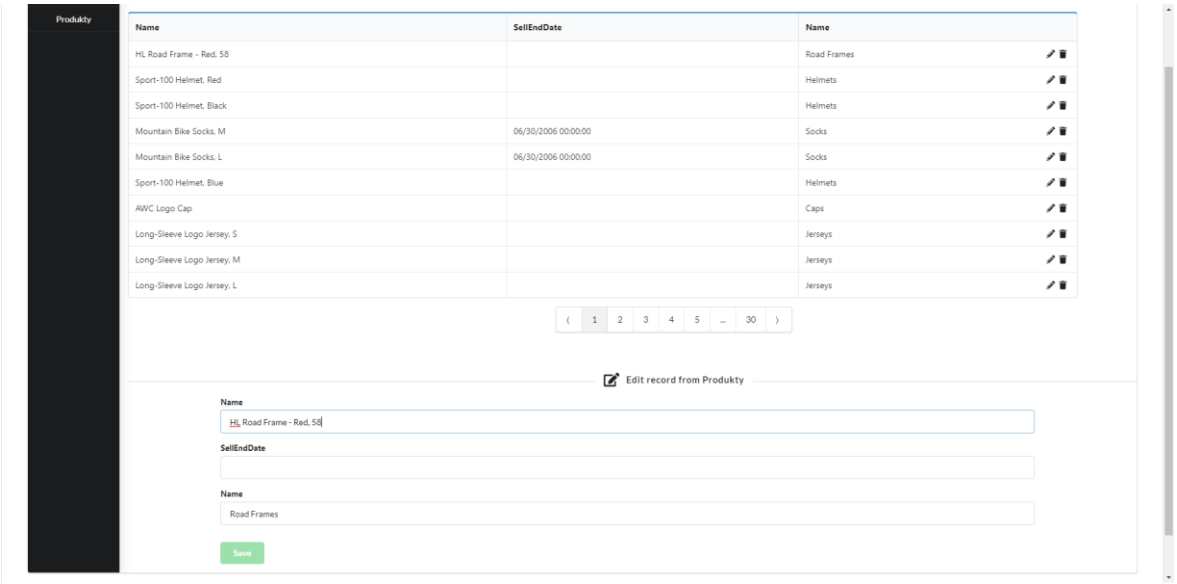
Engine dokáže na základe z informácií z metadát určiť ako zmazať záznam a vygeneruje príslušnú SQL query.

```
DELETE FROM App.Cities WHERE CityID = 1
```

Obrázok 56 Vygenerovaná query na zmazanie záznamu

[Zdroj: vlastný]

8.6 Úprava dát



Obrázok 57. Úprava dát

Zo znalosti stĺpcov dokáže systém pod tabuľkou vygenerovať formulár (Obrázok 57), ktorý sa po úprave poľa dá odoslať. Engine je ďalej schopný upraviť dáta v externej databázy.

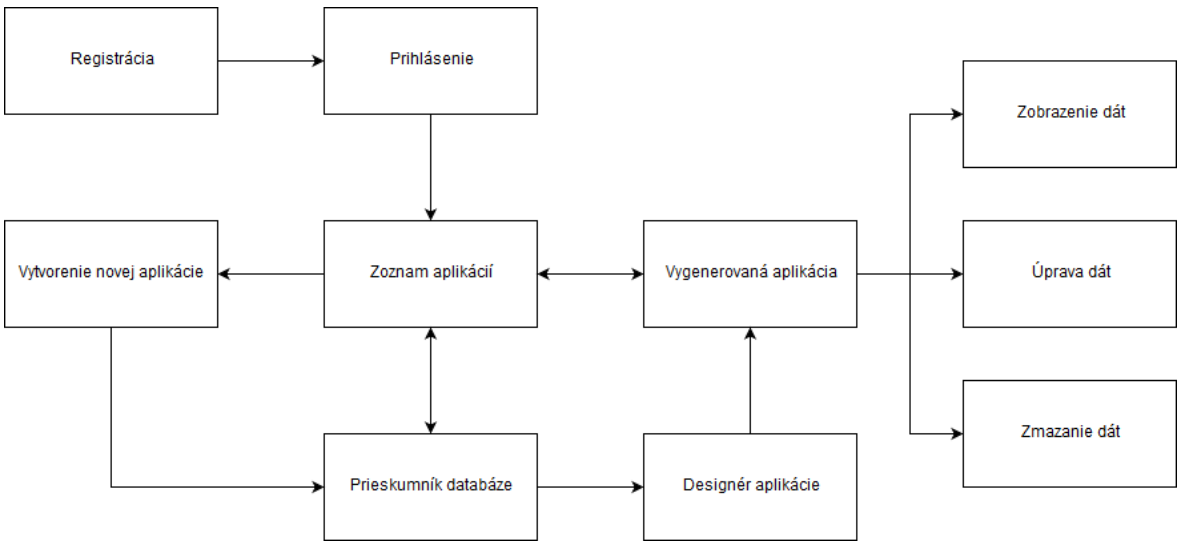
```
UPDATE [MainTable]
SET Name = N'Zlín - mesto'
FROM  App.Cities [MainTable]

WHERE [MainTable].CityID IN (2)

UPDATE [T0]
SET Name = N'Zlínský kraj'
FROM  App.Regions [T0]

WHERE [T0].RegionID IN (SELECT RegionID FROM App.Cities WHERE CityID IN (2))
```

Obrázok 58. Vygenerovaná SQL query úpravy dát [Zdroj: vlastný]

Na obrázku (Obrázok 58) je ukážka vygenerovanej SQL query, ktorá je výstupom modulu Engine a vstupom do externej databáze. Engine zvláda aj zložitejšie úpravy dát – updaty dát v najoinovaných tabuľkách.

Úprava dát je takmer instantná a zmena sa prepíše v UI.

9 VYTVORENÁ APLIKÁCIA



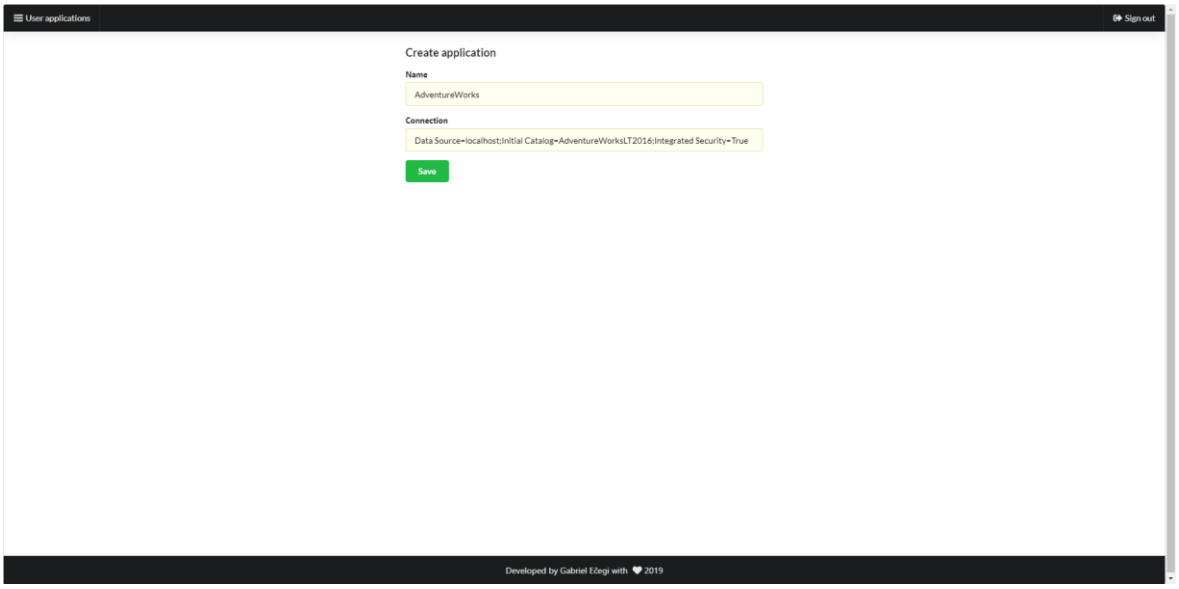
Obrázok 59. Mapa aplikácie [Zdroj: vlastný]

Vytvorená aplikácia je založená na architektúre popísanej v predchádzajúcich častiach. Jej užívateľské rozhranie je dostupné cez webový prehliadač. Užívateľ je v UI schopný vytvoriť nový užívateľský účet, prihlásiť sa cez neho, pridať externú databázu, nakonfigurovať aplikáciu a využívať vygenerovanú aplikáciu.

Užívateľské rozhranie je založené na knižnici Semantic UI, ktorá umožňuje stavať na UI komponentách s moderným UI, responzívnym designom a jednoduchým použitím. Semantic UI poskytuje knižnicu, v ktorej sú vytvorené React komponenty napísané v TypeScripte, čo zvyšuje DX a bezpečnosť aplikácie, keďže sa pri vývoji dajú komponenty konfigurovať cez silno typové React props.

9.1 Použitie data management systému

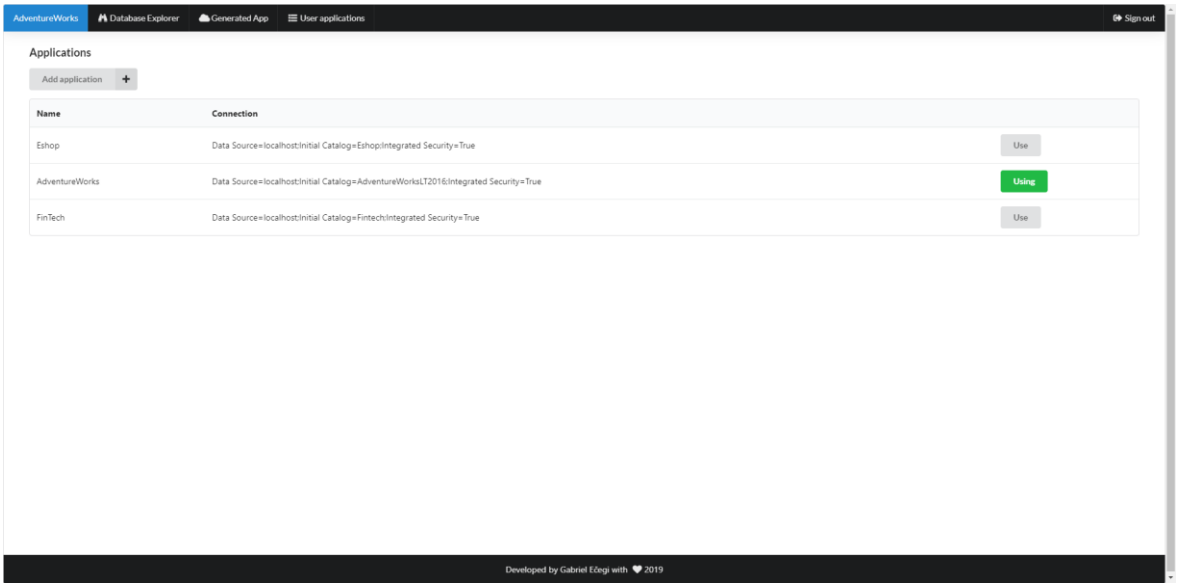
Vytvorená aplikácia je dostupná cez webový prehliadač. Ak chce užívateľ používať aplikáciu, musí mať najprv vytvorený užívateľský účet, ten je možné vytvoriť cez stránku registrácie. Po validnom vyplnení a odoslaní registračného formuláru, je užívateľ zaregistrovaný a je schopný sa prihlásiť.



Obrázok 60. Vytvorenie novej aplikácie [Zdroj: vlastný]

Po prihlásení je presmerovaný na stránku z výberom aplikácie. Ak zatiaľ žiadna aplikácia neexistuje, je ju možno vytvoriť. Na vytvorenie je potreba vyplniť len názov aplikácie a zadať štandardný *Connection String*. Tento obsahuje zdroj, s ktorým bude Engine pracovať. Napríklad pre SQL server je v tvare:

```
Server=<AdresaServeru>;Database=<NázovDatabáze>;Trusted_Connection=True;
```



Obrázok 61. Zoznam dostupných aplikácií [Zdroj: vlastný]

Ak užívateľ vybral aplikáciu, pošle sa na server dotaz s ID vybranej aplikácie a server zapíše do payload autorizačného tokenu ID vybranej aplikácie. Engine tak bude poznať kontext vybranej externej databáze pre ďalšie akcie.

V časti Prieskumník databáze – Database Explorer (Obrázok 49) sú zobrazené databázové schémy. Po vybraní schémy sa zobrazia databázové tabuľky patriace k vybratej schéme. Tu už si užívateľ môže zvoliť tabuľku, ktorú si určí za primárnu pre stavanie novej stránky.

Po vybratí tabuľky je možné prejsť do časti dizajnéra stránky. V dizajnéri (Obrázok 51) je možné vymodelovať stránku, ktorú bude možné používať na prehliadanie, úpravu a mazanie dát. Dizajnér vizualizuje zobrazenie referencií na ďalšie tabuľky do N-tej úrovne, z ktorých je takisto možné vyberať stĺpce. Dizajnér poskytuje aj náhľad vygenerovanej aplikácie, čo urýchľuje návrh, vďaka okamžitej vizualizácií dát.

Ak je užívateľ spokojný s vymodelovanou stránkou, môže stránku uložiť a bude presmerovaný do vygenerovanej aplikácie na stránku, ktorú vytvoril v dizajnéri. Tu už môže manageovať dáta (Obrázok 57).

9.2 Zhodnotenie technickej stránky aplikácie

Pri návrhu aplikácie z praktickej časti boli vybrané čo najmodernejšie technológie, toto so sebou prinieslo problémy vo forme ťažšieho získavania informácií k použitiu. Príkladom je rozhodnutie napísať serverovú časť v programovacom jazyku F#, použitie GraphQL. Množstvo architektonických rozhodnutí tak muselo byť samostatne výmyslených a keďže informácie o podobnom technologickom stacku neboli dostupné, je možné povedať, že je riešenie unikátne.

Celý vývoj aplikácie bol verzovaný verzovacím systémom git, boli dodržiavané princípy čistého kódu, SOLID princípy, čo prispieva k možnosti ďalšieho rozvoja aplikácie.

Použitie moderných technológií však prinieslo aj výhody ako napríklad jednoduchý data fetching vďaka GraphQL, React Hooks namiesto Redux, staticky typová kontrola GraphQL dotazov. Backendové technológie priniesli jednoduché doménové modelovanie v F#, čitateľnejšie algoritmy na spracovanie dát a metadát v F#.

Celkovo je tak možné povedať, že vytvorenie aplikácie založenej na funkcionálnych technológiách je kvôli menšiemu systému náročnejšie, avšak vývoj bol zaujímavejší a riešenie je kvalitnejšie.

9.2.1 Ekosystém F#

Jazyk F# ako taký je jeho užívateľmi chválený, prirovnávaný F# komunitou k C# ako jeho lepší súrodenec. Faktom však je, že komunita je malá a jej členovia sú často nadpriemerne skúsení programátori, ktorí často experimentujú a nevadia im výzvy. Tieto experimenty sú však často len hobby projekty a menšie produkčné aplikácie. Z reálnych diskusií s vývojármi na konferenciách bolo zistené, že vývojári snažiaci sa presadiť F# v pracovných projektoch, často použijú tento jazyk bez znalosti ich nadriadených, klientov, takpovediac nelegálne. Dôvodom je malá snaha software houses o presadenie nových paradigiem, jazykov, technológií do rozbehnutých procesov z dôvodu strachu z neúspechu. Toto sa ďalej odzrkadľuje v menšom počte aktívnych vývojárov a tým pádom malom ekosystéme okolo F#. Pri mojom vývoji som sa tak stretol s druhoradým F# toolingom vo Visual Studio od Microsoft, kde je veľmi dobrá podpora C#, ale F# podpora je citeľne horšia.

Podobne sú na tom F# knižnice, ktoré sú často neudržiavané pôvodným autorom. V rozšírenom ekosystéme by pôvodný autor našiel z radov používateľov jeho knižnice ľudí, ktorí by sa o knižnicu ďalej starali. V prípade malého ekosystému F# sa autor o knižnicu jednoducho prestane starať, čo veľa ľudom nevadí, pretože v F# neprogramujú. Ľudia začínajúci s jazykom zase bývajú odradení od použitia jazyka, pretože F# knižnica je neudržiavaná, avšak knižnica v C# je udržiavaná oveľa lepšie. Stáva sa tak že vývojár si radšej vyberie populárnejší, a nie lepší jazyk.

9.2.2 GraphQL .NET

Veľkou časťou architektúry sú knižnice súvisiace s technológiou GraphQL, ktorá sa napriek svojej krátkej existencii stáva veľmi rozšírenou vo frontendových JavaScript aplikáciách. Jedným z faktorov je otvorenosť frontendových vývojárov k zmenám a novinkám, ďalším je použitie technológie Facebookom vo svojich aplikáciách, čo prináša garanciu podpory technológie v budúcnosti. Problémom je však integrácia do .NET, ktorý o podporu GraphQL štandardu oficiálny záujem nemá. Sú však dostupné open source knižnice od komunity, ktoré dokážu splniť účel, avšak nie je z nich cítiť takú kvalitu akú dosahujú GraphQL knižnice na frontende.

9.3 Zhodnotenie funkčnej časti aplikácie

Aplikácia po funkčnej stránke umožňuje vytvorenie užívateľského účtu a následnú autentifikáciu. V aplikácií je možné vytvoriť a nakonfigurovať prostredie pre aplikáciu, ktorú budeme chcieť vytvoriť a rozširovať. Po vytvorení je užívateľ schopný pracovať s dátami v databázy. Aplikácia tak po funkčnej stránke splňuje časté požiadavky, pre ktoré je treba často programovať zvlášť aplikáciu, slúžiacu len ako administrátorské rozhranie. Aplikácia vytvorená systémom EzAdmin, tak môže nahradiť samostanú aplikáciu a tým ušetriť čas aj financie, keďže nie je potreba programovania, ale len databáze.

Vďaka zvoleným technológiám bude možné rozširovať aplikáciu a pritom v bezpečí zachovať stávajúcu funkčnosť. Aplikácia môže byť rozširovaná o možnosti konfigurácie užívateľského rozhrania vygenerovanej aplikácie. V module používateľov aplikácie môžu byť pridané užívateľské práva tak, aby vybraní používatelia mohli používať vygenerovanú aplikáciu, ale aby už ďalej nemohli upravovať jej vzhľad.

Zaujímavým rozšírením by mohol byť plne automatizovaný export vygenerovanej aplikácie, kedy by sa vygenerovaná aplikácia vystavila na vzdialený server a mohli by k nej mať prístup zákazníci, ktorí by si do vygenerovanej aplikácie zaplatili prístup. Obsluha aplikácie by mohla zákazníkovi vytvoriť a okamžite upravovať vzhľad aplikácie. Toto by bolo finančne menej náročné, ako programovať zákazníkovi systém na mieru.

ZÁVĚR

V oblasti moderného vývoja webových aplikácií je dnes možné vyvíjať aplikácie, ktoré by bolo pred niekoľkými rokmi možné vytvoriť len ako desktopové programy. V tejto oblasti sa čoraz populárnejším stáva funkcionálne programovanie. Množstvo frameworkov, knižníc je dnes založených na funkcionálnych technológiách. Hlavným cieľom diplomovej práce bolo na základe dostupných poznatkov vytvoriť data management systém založený na funkcionálnych technológiách.

V teoretickej časti bola vytvorená literárna rešerš na tému funkcionálne programovanie, jeho základné piliere, populárne knižnice a frameworky využívajúce výhod tejto paradigmy. K základným pilierom a konštruktom boli pridané ukážky kódu, diagramy a grafické vizualizácie. V tejto časti bolo ukázané, ako funkcionálne programovanie využíva svoje základy založené na matematike a matematických funkciách. Ukázalo sa, že tieto koncepty okrem čisto funkcionálnych jazykov začínajú využívať aj pôvodne objektovo orientované jazyky ako JavaScript alebo C#. Tieto sa stávajú výberom toho dobrého ako zo sveta funkcionálneho programovania, tak aj objektovo orientovaného programovania.

Ďalej sa práca venuje jazyku F# a jeho využitiu vo svete webových aplikácii. F# sa ukazuje mladý jazyk, ktorému chýba rozšírená komunita. Na druhej strane je F# známy ako jazyk, ktorý je vytvorený na riešenie moderných problémov s neustále prispievajúcimi členmi komunity, ktorí vytvárajú kvalitný ekosystém jazyka na poli webových technológií.

V práci je tiež popísané použitie funkcionálnych technológií v UI. Tu sa popularite teší knižnica React, ktorá v kombinácií s ďalšími knižnicami ako GraphQL dokáže vytvoriť spojenie technológií pre tvorbu kvalitných webových aplikácií.

Znalosti získané z teoretickej časti sú aplikované v praktickej časti, kde je navrhnutá vhodná štruktúra a architektúra aplikácie, ako pre serverovú časť, tak pre časť klienta. Návrh sa snaží využívať čo najmodernejšie technológie, využívať staticky typový kód a berie ohľad na ďalšiu rozšíriteľnosť aplikácie. V praktickej časti je popísaný modul, ktorý má na starosti prácu s dátami v externej databázy. V popise modulu sú vizualizované, popísané algoritmy spracovávajúce dáta a metadáta v tomto data management systéme.

V praktickej časti je vytvorené užívateľské rozhranie v podobe webovej aplikácie, v ktorom užívateľ môže interaktívne s okamžitým zobrazením zmien, navrhnúť aplikáciu, ktorú na

základe požiadavku systém vygeneruje. Vo vygenerovanej aplikácií je užívateľ schopný dáta zobrazovať, upravovať a mazať.

Následne je v praktickej časti popísaný možný spôsob použitia aplikácie, slúžiaci aj ako návod na obsluhu. Užívateľ aplikácie je tak schopný prepojiť ním navrhnutú aplikáciu s jeho databázou a začať spravovať dáta. Toto je veľmi rýchla, časovo a finančne nenáročná úloha, kedy je užívateľ bez programátorských skúseností vytvárať aplikáciu, ktorú by inak musel vytvárať softvérový vývojár. Vygenerovaná aplikácia tak môže pomáhať osobám s menšou znalosťou SQL, napríklad analytikom a testerom, kedy systém vygeneruje SQL dotazy a zobrazí výsledky operácie za nich.

Aplikácia sa je ako celok zhodnotená po technickej a funkčnej stránke. V hodnotení funkčnej stránky aplikácie je popísané, ako môže byť aplikácia ďalej rozvíjaná. Oblasti rozvoja sú rozsiahle, od rozšírenia konfigurácie vygenerovanej aplikácie až po celý jej export.

SEZNAM POUŽITÉ LITERATURY

[1] BUONANNO, Enrico. *Functional programming in C#*. Shelter Island, NY: Manning Publications, [2017]. ISBN 9781617293955.

[2] GraphQL [online]. Facebook, 2018 [cit. 2018-11-25]. Dostupné z: https://graphql.github.io/

[3] Higher-Order functions [online]. 2019 [cit. 2019-03-24]. Dostupné z: http://functionalprogrammingcsharp.com/higher-order-functions

[4] Functors and Monads [online]. 2019 [cit. 2019-03-24]. Dostupné z: http://functionalprogrammingcsharp.com/functors-monads

[5] Array.prototype.filter() [online]. 2019 [cit. 2019-03-24]. Dostupné z: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/filter

[6] Array.prototype.map() [online]. 2019 [cit. 2019-03-24]. Dostupné z: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/map

[7] Enumerable.Select Method [online]. 2019 [cit. 2019-03-24]. Dostupné z: https://docs.microsoft.com/en-us/dotnet/api/system.linq.enumerable.select?view=netframework-4.7.2

[8] MAC LANE, Saunders. *Categories for the working mathematician*. 2nd ed. New York: Springer, c1998. ISBN 0-387-98403-8.

[9] Understanding map and apply [online]. 2019 [cit. 2019-03-24]. Dostupné z: https://fsharpforfunandprofit.com/posts/elevated-world/#return

[10] SYME, Don, Adam GRANICZ a Antonio CISTERNINO. Expert F# 4.0. 4. Apress, 2015. ISBN 978148420740

[11] Guide - Web Programming with F# [online]. F# Software Foundation and individual contributors, 2019 [cit. 2019-03-31]. Dostupné z: https://fsharp.org/guides/web/

[12] Introduction to ASP.NET Core [online]. Microsoft, 2019 [cit. 2019-03-31]. Dostupné z: https://docs.microsoft.com/en-us/aspnet/core/?view=aspnetcore-2.2

[13] Quickstart: Use Visual Studio to create your first ASP.NET Core web service in F# [online]. Microsoft, 2019 [cit. 2019-03-31]. Dostupné z: https://docs.microsoft.com/en-us/visualstudio/ide/quickstart-fsharp?view=vs-2017

[14] Giraffe: A native functional ASP.NET Core web framework for F# developers. [online]. Microsoft, 2019 [cit. 2019-03-31]. Dostupné z: https://github.com/giraffe-fsharp/Giraffe

[15] Giraffe Documentation [online]. Microsoft, 2019 [cit. 2019-03-31]. Dostupné z: https://github.com/giraffe-fsharp/Giraffe/blob/master/DOCUMENTATION.md

[16] SAFE Documentation [online]. Microsoft, 2019 [cit. 2019-03-31]. Dostupné z: https://safe-stack.github.io/docs/

[17] SAFE Stack overview [online]. Microsoft, 2019 [cit. 2019-03-31]. Dostupné z: https://safe-stack.github.io/docs/overview/

[18] BANKS, Alex a Eve PORCELLO. Learning React: functional web development with React and Redux. Sebastopol, CA: O'Reilly Media, 2017. ISBN 9781491954621

[19] CHIARELLI, Andrea. The functional side of React. In: Medium [online]. Medium, 2018 [cit. 2019-04-04]. Dostupné z: https://medium.com/@andrea.chiarelli/the-functional-side-of-react-229bdb26d9a6

[20] Composition vs Inheritance. React [online]. 2019 [cit. 2019-04-04]. Dostupné z: https://reactjs.org/docs/composition-vs-inheritance.html#so-what-about-inheritance

[21] DATA MANAGEMENT SYSTEM. Poznańskie Centrum Superkomputerowo-Sieciowe [online]. Poznan Supercomputing and Networking Center: Poznan Supercomputing and Networking Center, 2004 [cit. 2019-04-08]. Dostupné z: http://www.man.poznan.pl/coe/documents/Data_Management_WP.pdf

[22] Data Management Software (DMS). Techopedia [online]. 2019 [cit. 2019-04-08]. Dostupné z: https://www.techopedia.com/definition/11363/data-management-software-dms

[23] What is a functor?. Medium [online]. 2015 [cit. 2019-04-23]. Dostupné z: https://medium.com/@dtinth/what-is-a-functor-dcf510b098b6

[24] Enumerable.Select Method. Microsoft [online]. 2019 [cit. 2019-04-23]. Dostupné z: https://docs.microsoft.com/en-us/dotnet/api/system.linq.enumerable.select?view=netframework-4.8

[25] GraphQL vs REST: Overview. Blog Apollo [online]. 2019 [cit. 2019-04-23]. Dostupné z: https://blog.apollographql.com/graphql-vs-rest-5d425123e34b

[26] INTRODUCTION TO DATABASE MANAGEMENT SYSTEM [online]. 2016 [cit. 2019-04-28]. Dostupné z: http://dbmsfortech.blogspot.com/2016/03/introduction-to-database-management.html [27] Guide - Web Programming with F# [online]. F# Software Foundation and individual contributors, 2019 [cit. 2019-03-31]. Dostupné z: https://fsharp.org/guides/web/

[28] JSON Web Tokens vs. Session Cookies: In Practice. Ponyfoo [online]. ponyfoo, 2018 [cit. 2019-04-28]. Dostupné z: https://dzone.com/articles/cookies-vs-tokens-the-definitive-guide

[29] Cookies vs. Tokens: The Definitive Guide. DZone [online]. DZone, 2018 [cit. 2018-11-26]. Dostupné z: https://dzone.com/articles/cookies-vs-tokens-the-definitive-guide

[30] JSON Web Token (JWT). 1. USA: Internet Engineering Task Force (IETF), 2015.

[31] Introduction to JSON Web Tokens. JWT [online]. USA: Auth0, 2018 [cit. 2018-11-26]. Dostupné z: https://jwt.io/introduction/

[32] Why is GraphQL important?. Apollo GraphQL [online]. 2019 [cit. 2019-05-02]. Dostupné z: https://www.apollographql.com/why-graphql

[33] PETRICEK, Tomas a Jon SKEET. *Real-world functional programming: with examples in F# and C#*. Greenwich: Manning, c2010. ISBN 9781933988924.

[34] DELANEY, Kalen, Bob BEAUCHEMIN, Conor CUNNINGHAM, Jonathan KEHAYIAS, Benjamin NEVAREZ a Paul S. RANDAL. Microsoft SQL server 2012 internals. Sebastopol, Calif: O'Reilly Media, [2013]. ISBN 0735658560.

[35] CRUD Admin Generator [online]. 2019 [cit. 2019-05-05]. Dostupné z: http://crud-admin-generator.com/

[36] Forest [online]. 2019 [cit. 2019-05-05]. Dostupné z: https://www.forestadmin.com

SEZNAM POUŽITÝCH SYMBOLŮ A ZKRATEK

OOP	Objektovo orientované programovanie
SQL	Structured Query Language
API	Application programming interface
UI	User interface
IO	Vstup výstup
LINQ	Language Integrated Query
HTTP	Hypertext Transfer Protocol
IIS	Internet Information Services
JS	JavaScript
HTML	Hyper text markup language
CSS	Cascading style sheets
REST	Representational state transfer
DMS	Data managment software
CRUD	Create Read Update Delete
JSON	JavaScript Object Notation
DB	Database
JWT	JSON Web Token
CLI	Command-line interface
ECDSA	Elliptic Curve Digital Signature Algorithm
HMAC	Keyed-hash message authentication code
npm	Node Package Manager
DX	Developer Exprience
FP	Funkcionálne programovanie

SEZNAM OBRÁZKŮ

SEZNAM PŘÍLOH