
# VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

BRNO UNIVERSITY OF TECHNOLOGY

## FAKULTA STROJNÍHO INŽENÝRSTVÍ

FACULTY OF MECHANICAL ENGINEERING

## ÚSTAV AUTOMATIZACE A INFORMATIKY

INSTITUTE OF AUTOMATION AND COMPUTER SCIENCE


# WEBOVÁ APLIKACE PRO VYHODNOCOVÁNÍ INVESTIČNÍCH ZÁMĚRŮ V ODPADOVÉM HOSPODÁŘSTVÍ

WEB APPLICATION FOR EVALUATING INVESTMENT PLANS IN WASTE MANAGEMENT

DIPLOMOVÁ PRÁCE

MASTER'S THESIS


AUTOR PRÁCE — Bc. Roman Červenka

AUTHOR

VEDOUCÍ PRÁCE — Ing. Ladislav Dobrovský

SUPERVISOR



BRNO 2023

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ FAKULTA STROJNÍHO INŽENÝRSTVÍ

# Zadání diplomové práce

| | |
|---|---|
| Ústav: | Ústav automatizace a informatiky |
| Student: | **Bc. Roman Červenka** |
| Studijní program: | Aplikovaná informatika a řízení |
| Studijní obor: | bez specializace |
| Vedoucí práce: | **Ing. Ladislav Dobrovský** |
| Akademický rok: | 2022/23 |

Ředitel ústavu Vám v souladu se zákonem č.111/1998 o vysokých školách a se Studijním a zkušebním řádem VUT v Brně určuje následující téma diplomové práce:

## Webová aplikace pro vyhodnocování investičních záměrů v odpadovém hospodářství

**Stručná charakteristika problematiky úkolu:**

Problematika navazuje na výzkumné aktivity Ústavu procesního inženýrství. Konkrétně se jedná o implementaci existujících matematických modelů do webové aplikace formou nového modulu, který bude rozšiřovat stávající komplexní nástroj o novou funkcionalitu. Tato aplikace bude zaměřena na podporu plánovaní výstavby nových zařízení pro přechod na oběhové hospodářství. Hlavním rysem nově vzniklého modulu bude komplexnost, která zahrnuje celý zpracovatelský řetězec (produkce a složení odpadu, technicko–ekonomické modely, dopravní infrastruktura a legislativní rámec). Konkrétně bude nástroj zaměřen na oblasti:
– Posouzení investičního záměru – návratnost investice, rizika projektu aj.
– Koncepční analýzy z pohledu regionu a státu s vazbou na stanovené cíle v legislativě.
– Databáze obsahující vazby mezi různými druhy odpadu a zařízení včetně reziduálních toků.
– Podpora provozu zařízení – plánování přepravy odpadu, nastavení smluvních kontraktů s ostatními subjekty.

**Cíle diplomové práce:**

Seznámení se strukturou dat a návaznostmi mezi prvky systému.
Návrh a implementace databázového systému.
Návrh komunikace mezi výpočtovým jádrem a uživatelským rozhraním.
Aktivní řízení výpočetních procesů s ohledem na kapacity procesorů a paměti.
Automatizované zpracování výstupů z výpočtového jádra do grafické prezentace.
Integrace v existující webové aplikaci.

**Seznam doporučené literatury:**

CHATURVEDI, Rajneesh, Swati V. CHANDE a Amita SHARMA. Evaluation and Refinement of MVC Web Application Architecture. Journal of Information and Computational Science. 2021, 2021(3). ISSN 1548-7741.

HAKLAY, M. Openstreetmap: User-generated street maps. IEEE Pervasive Computing [online]. 2008, 7(4), 12 [cit. 2022-10-10]. ISSN 1536-1268.

W3 schools: AJAX Introduction [online]. [cit. 2022-10-10]. Dostupné z: https://www.w3schools.com/xml/ajax_intro.asp.

Flask: web development, one drop at a time [online]. [cit. 2022-10-15]. Dostupné z: https://flask.palletsprojects.com/en/2.2.x/

Termín odevzdání diplomové práce je stanoven časovým plánem akademického roku 2022/23

V Brně, dne

L. S.

| | |
|---|---|
| doc. Ing. Radomil Matoušek, Ph.D.<br>ředitel ústavu | doc. Ing. Jiří Hlinka, Ph.D.<br>děkan fakulty |

# ABSTRAKT

Táto diplomová práca sa venuje návrhu a implementácií modulu pre webovú aplikáciu pre optimalizáciu v odpadovom hospodárstve. Hlavnou časťou je návrh užívateľského rozhrania pre zadanie vstupných parametrov a zobrazenie výsledkov optimalizácie v grafickej podobe v mape. Súčasťou práce je taktiež analýza stavu aplikácie a návrhy možných vylepšení.

# ABSTRACT

This thesis is dedicated to design and implementation of module for web aplication for waste management optimization. Main part is design and implementation of user interface for input of optimization variables and visualization of optimization results in graphical map overlay. Part of this thesis is also analysis of current state of application and proposition of possible improvements.



# KLÍČOVÁ SLOVA

Webová aplikácia, odpadové hospodárstvo, React, JavaScript, Python, Flask API, užívateľské rozhranie, kontajnerizácia, Docker

# KEYWORDS

Web aplication, waste management, React, JavaScript, Python, Flask API, user interface, containerization, Docker

ÚSTAV AUTOMATIZACE A INFORMATIKY

2023

# BIBLIOGRAFICKÁ CITACE

ČERVENKA, Roman. Webová aplikace pro vyhodnocování investičních záměrů v odpadovém hospodářství. Brno, 2023. Dostupné také z: https://www.vut.cz/studenti/zav-prace/detail/149697. Diplomová práce. Vysoké učení technické v Brně, Fakulta strojního inženýrství, Ústav automatizace a informatiky. Vedoucí práce Ladislav Dobrovský.

# POĎAKOVANIE

Týmto by som sa rád poďakoval Ing. Ladislavovi Dobrovskému za vedenie práce. Taktiež tímu z Ústavu procesního inženýrství za možnosť spolupracovať na projekte Popelka.
Ďalej by som sa chcel poďakovať rodine a priateľke za podporu, ako pri práci na diplomovej práci, tak aj počas celého štúdia.

# ČESTNÉ PREHLÁSENIE

Prehlasujem, že táto práca je mojím pôvodným dielom, vypracoval som ju samostatne pod vedením vedúceho práce a s použitím odbornej literatúry a ďalších informačných zdrojov, ktoré sú všetky citované v práci a uvedené v zozname literatúry.

Ako autor uvedenej práce ďalej prehlasujem, že v súvislosti s vytvorením tejto práce som neporušil autorské práva tretích osôb, najmä som nezasiahol nedovoleným spôsobom do cudzích autorských práv osobnostných a som si plne vedomý následku porušenia ustanovení § 11 a nasledujúcich autorského zákona č. 121/2000 Zb., vrátane možných trestno právnych dôsledkov.

V Brne dňa 20. 5. 2023 .....

Bc. Roman Červenka

# OBSAH

# 1 ÚVOD

Úsilie o ochranu životného prostredia sa rýchlo zvyšuje spolu s rastom svetovej populácie, čo súvisí s priemyselným rozvojom a zmenami výroby. To platí pre všetky priemyselné sektory, pričom sa kladie osobitný dôraz na znečistenie a nakladanie s odpadmi.

Aktuálny objem odpadov produkovaných po celom svete spôsobuje vážne debaty o účinnosti ich spracovania. Značné množstvo odpadov sa spracováva neefektívne a neekologicky. Odpad však môže predstavovať nielen záťaž pre životné prostredie a ekonomiku, ale aj ekonomickú príležitosť ako druhotný zdroj.

Princípy cirkulárnej ekonomiky integrovanej do odpadového hospodárstva sa považujú za možné riešenie udržateľného rozvoja. Uprednostňujú sa metódy nakladania s odpadmi v jasne danom poradí v súlade s hierarchiou. Zariadenia na energetické využitie odpadov predstavujú strednodobú možnosť nakladania s odpadmi, ktorá zaisťuje udržateľnosť (ihneď po hierarchicky vyšších cieľoch, ako je prevencia alebo získavanie materiálu). Implementácia obehového hospodárstva do existujúcej siete vyžaduje nielen potrebu nových trendov a metód v nakladaní s odpadmi, ale aj testovacie a výpočtové nástroje umožňujúce rýchle vyhodnotenie aktuálnej situácie a modelovanie budúcich stavov.

Česká republika je typickým príkladom štátu, kde práve dochádza k transformácii odpadového hospodárstva smerom k obehovému. V roku 2014 predstavilo Ministerstvo životného prostredia ČR tzv. Plán odpadového hospodárstva na obdobie 2015-2024, ktorý už zohľadňuje nastavené trendy EÚ. Nadväzujúci dokument, ktorý by mal byť pomaly tvorený v súlade s aktuálnou európskou legislatívou, musí zahŕňať jasné postupy a jednotlivé kroky, ako efektívne pracovať s odpadmi. Lokálna legislatíva už implementovala obmedzenia napr. tým, že zakazuje skládkovanie recyklovateľných a využiteľných odpadov od roku 2030 [1].

To sa však netýka iba Českej republiky, ale je to záväzok krajín EÚ: 60 % produkovaného komunálneho odpadu by sa malo do roku 2030 využívať materiálovo. Hlavné ciele a obmedzenia sú definované v smerniciach Európskeho parlamentu a Rady [2] [3].

Simulácia logistických reťazcov za účelom spracovania rôznych typov komunálneho odpadu je v dnešnej dobe veľmi dôležitá. Vzhľadom na to, že mnoho miest a obcí sa snaží minimalizovať množstvo vytváraného odpadu a maximalizovať jeho recykláciu, je nutné využiť moderné technológie a vytvoriť efektívny systém nakladania s odpadom. Existuje mnoho typov odpadu, ktoré sa musia samostatne zbierať a odvážať, často na vzdialené spracovateľské miesta. Jednotlivé odpady majú rôznu hmotnosť a ďalšie vlastnosti ako z pohľadu prepravy, tak aj následného využitia. Pre konkrétny typ odpadu možno teda odhadnúť náklady na prepravu, ktoré sa však môžu výrazne líšiť. Napríklad samostatne sústreďovaný papier alebo plast sú ľahšie a majú väčší objem, zatiaľ čo zmiešaný komunálny odpad má väčšiu hmotnosť a menší objem. Preto je dôležité vytvoriť systém, ktorý dokáže odhadnúť náklady na prepravu odpadu na základe jeho typu a množstva a vhodne ho alokovať do existujúcich a potenciálnych spracovateľských zariadení. Tento systém by mal zahŕňať simulácie logistiky, ktoré by

umožnili optimalizovať trasy prepravy odpadu, znížiť náklady na prepravu a minimalizovať dopady na životné prostredie.

S ohľadom na detail simulácie musí používateľské rozhranie umožňovať optimalizáciu na rôznych úrovniach geografického členenia. Toto rozhranie by malo byť ľahko použiteľné a prispôsobiteľné potrebám konkrétnych oblastí a typu optimalizačnej úlohy.

Keďže kľúčové parametre môžu byť neznáme alebo len odhadnuté, je potrebné vyhodnocovať rôzne scenáre. Neistota môže byť modelovaná napr. stromom scenárov generovaných na základe reálnych dát (napr. dáta o produkcii jednotlivých typov odpadu podľa geografickej lokality). Vyhodnotenie výstupov a rozdiely medzi scenármi potom môžu predstavovať dôležité podklady pre rozhodnutie o podpore výstavby zariadení s novými technológiami vo vhodnej lokalite. Ako príklad môže byť uvedené postupné navyšovanie poplatku za skládkovanie za tonu odpadu v priebehu nasledujúcich rokov. Dôsledkom takéhoto zásahu sú a budú zmeny celého logistického reťazca a súvisiace transformácie koncových zariadení [4].

Diplomová práca sa bude zaoberať vývojom užívateľského rozhrania s možnosťou zobrazovať a vyhodnocovať výsledky logistických optimalizačných úloh. Rozhranie bude členené do troch čiastkových častí:

1. časť pre zadanie parametrov úlohy – voľba riešených typov odpadu, typov a lokalít koncových zariadení, vybranie produkčných uzlov a povolený spôsob prepravy.
2. časť by mala slúžiť k mapovému zobrazeniu vstupných dát a výsledkov jednotlivých simulácií vo forme vektorových dát.
3. časť budú slúžiť k základnému dátovému vyhodnoteniu do súhrnných grafov.

Modul aplikácie ktorým sa bude práca zaoberať, nesie pracovný názov Cireto. Predpokladá sa ďalší vývoj tohto modulu, preto je jedným z cieľov zaistiť upraviteľnosť, rozšíriteľnosť a znovu použiteľnosť komponent a kódu vytvorených v tejto práci.

# 2 POUŽITÉ TECHNOLÓGIE

Voľba použitých technológií bola uskutočnená na základe existujúcej aplikácie. Pre backend to znamená jazyk Python a pre frontend jazyk JavaScript. V tejto práci bude značné sústredenie na frontend. Budú však potrebné zmeny aj v backend časti a v databáze.

## 2.1 Frontend

Frontend je časť aplikácie ktorá beží na strane užívateľa v prehliadači, to znamená grafické rozhranie a JavaScript časť programu. V tejto časti programu sa obvykle nachádzajú dáta ktoré sú špecifické pre daného užívateľa alebo k nim má mať prístup, to zabezpečuje backend ktorý nemôže poslať dáta ktoré nemajú byť konkrétnemu užívateľovi prístupné.

### 2.1.1 React

Existujúca aplikácia využíva open-source knižnicu React. Táto knižnica zjednodušuje tvorbu webových aplikácií. Je možné ju využiť pre jazyky JavaScript aj TypeScript.

React bol vytvorený v roku 2013 spoločnosťou Facebook (dnes Meta). Dnes patrí medzi najpoužívanejšie JavaScript knižnice, podobné frameworky sú napríklad Vue.js, Angular či Svelte.

Výhoda použitia React spočíva, ako vyplýva z názvu, v jeho reaktívnosti. React prekresľuje jednotlivé komponenty keď príde k zmene v zobrazených dátach, čo znižuje počet načítaní a používanie React aplikácií sa preto javí ako veľmi plynulé. Základný stavebným prvkom React aplikácií sú komponenty. Tie zaisťujú znovu použiteľnosť často používaných prvkov (napríklad tlačidlá, nadpisy, tabuľky), to znižuje duplicitu kódu a následne šetrí čas vývojárov a zdroje. [5]

### 2.1.2 OpenStreetMap

OpenStreetMap je open-source projekt ktorého cieľom je poskytovať voľne dostupné geografické dáta. Tento projekt je udržiavaný komunitou a pri použití týchto máp stačí uviesť autora OpenStreetMap a prispievateľov. [6]

### 2.1.3 Leaflet

Leaflet je JavaScript open-source knižnica slúžiaca pre vytvorenie interaktívnych máp vo webových aplikáciách.

Hlavné výhody tejto knižnice sú jednoduchosť použitia, rýchlosť a nízka pamäťová náročnosť.

Túto knižnicu je možné ľahko rozšíriť pomocou pluginov a funguje s akýmkoľvek mapovým serverom. Keďže je ale OpenStreetMap zadarmo, je s touto knižnicou často využívaná. [7]

### 2.1.4 React Leaflet

React Leaflet je knižnica ktorá poskytuje Leaflet funkcionalitu zabalenú do React komponent. Použitie tejto knižnice odstraňuje potrebu vytvárať vlastné mapové komponenty.

## 2.2 Backend

Backend je časť aplikácie ktorá pracuje na serveri.

### 2.2.1 Flask

Flask je open-source framework pre vývoj webových aplikácií v programovacom jazyku Python. Flask je navrhnutý tak, aby bol ľahko použiteľný a flexibilný, a umožňuje vývojárom rýchlo vytvárať webové aplikácie.

Framework poskytuje základné funkcie, ktoré sú potrebné na vývoj webových aplikácií, vrátane podpory HTTP protokolu, URL routingu, integrácie s Jinja2 templating engine pre tvorbu HTML šablón, a mnohých ďalších. Flask je tiež veľmi konfigurovateľný a umožňuje vývojárom používať rôzne moduly a rozšírenia podľa potreby.

Ďalej využíva princíp "mikro" frameworku, čo znamená, že sa snaží byť čo najjednoduchší a minimalistický, aby sa minimalizovala zložitosť a zvýšila flexibilita. To znamená, že Flask nemá integrované ORM alebo databázové adaptéry, ale umožňuje vývojárom používať rôzne moduly a rozšírenia pre prácu s databázami a ďalšími funkcionálne špecifickými oblasťami. [8]

Flask je WSGI framework, web server gateway interface, čo je špecifikácia ako má web server komunikovať s webovými aplikáciami. [9]

## 2.3 DevOps

DevOps časťou aplikácie je spustenie serveru, monitorovanie serveru, konfigurácia či tvorba potrebných nástrojov.

### 2.3.1 GitLab CI/CD

CI/CD znamená:

Continuous Integration – znamená kontinuálne testovanie pridaných zmien. Vývojár pracuje lokálne, pridá výsledok svojej práce do GitLab či iného verzovacieho nástroja. GitLab následne aplikáciu spustí a spustí nad ňou testy. Týmto sa zaisťuje funkčnosť a kvalita pridaného kódu.

Continuous Delivery – je ďalším krokom Continuous Integration. Aplikácia je po otestovaní nahraná na server. Tento krok sa však deje manuálne.

Continuous Deployment – je v CI/CD konečným krokom. Aplikácia po otestovaní je automaticky nahraná na server.
CI/CD znamená že pri každej zmene kódu, v krátkom časovom úseku, môžu užívatelia vidieť nové zmeny. Tento časový úsek je variabilný a pre malé aplikácie to môže byť otázka minút, pre veľmi veľké aplikácie sú to dni aj viac než týždeň. [10]

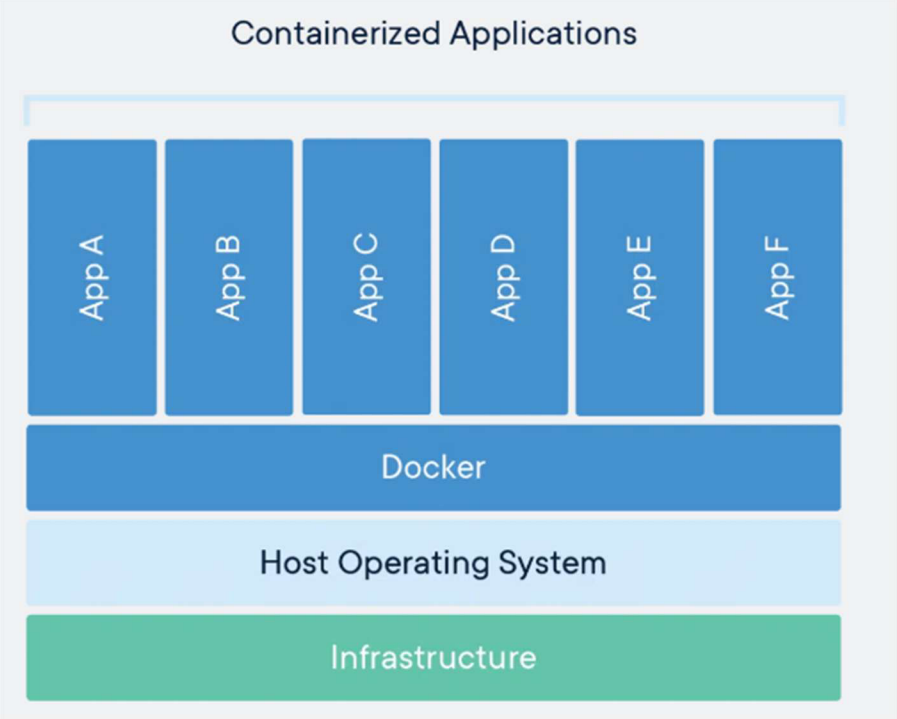
### 2.3.2 Docker

Docker je software, ktorý slúži na kontajnerizáciu aplikácií. Kontajneri sa dajú prirovnať ku virtual machines (virtuálny stroj, VM). Slúžia na dodanie software v ucelenom balíku, toto rieši problém s rozdielnym správaním software na rôznych zariadeniach, pretože takto sa software spúšťa na prakticky totožnom systéme, z hľadiska systému, hardware sa samozrejme môže líšiť. [11]

Keďže však Docker kontajnery nie sú VM, sú značne menej pamäťovo náročné a nepredstavujú až také straty výkonu. Pre všetky zároveň spustené kontajnery slúži len jeden hosťovací operačný systém (Obr. 1). Toto tvrdenie nemusí platiť v niektorých prípadoch pri využití Windows bez Linux VM [12].

Docker sa dnes využíva veľmi často pre rôzne druhy aplikácií, hlavne však pre webové aplikácie a v spojení s Kubernetes.

Existuje tiež Podman, aplikácia podobná Docker, ktorý vznikol neskôr ale je open-source. Ten využíva Open Container Initiative. [13] Táto iniciatíva spoluzaložená Docker a ďalšími, má za cieľ vytvárať otvorené štandardy pre kontajnerizáciu. [14]




Obr. 1 Schéma Docker kontajnerov [12]

### 2.3.3 Kubernetes

Kubernetes, tiež niekedy uvádzaný ako K8s, slúži na riadenie kontajnerov. Poskytuje možnosť automatizovania chodu a spúšťania kontajnerov, kontrolu stavu týchto kontajnerov či rozdelenia záťaže medzi jednotlivé kontajneri. Kubernetes napríklad dokáže upraviť počet kontajnerov s danou aplikáciou podľa záťaže danej skupiny kontajnerov, čo umožňuje jednoduchú škálovateľnosť aplikácií.

# 3 SÚČASNÝ STAV APLIKÁCIE A NAVRHOVANÉ ZMENY

Táto práca nadväzuje na prácu študentov z minulých rokov, je potrebné zanalyzovať aktuálny stav aplikácie a opraviť prípadné chyby či navrhnúť vylepšenia.
Vzhľadom na zameranie tejto práce prevažne na frontend, bude rovnako cielená aj táto analýza.

Výsledkom tejto analýzy budú návrhy na zlepšenia, ktoré budú následne prediskutované s vývojovým tímom. Z diskusie sa vytvoria závery a potrebné kroky, ktoré je nutné vykonať čím skôr ale zároveň aj zmeny ktoré nie je nutné zapracovať okamžite, ale až neskôr či postupne.

## 3.1 Framework

Aktuálne sa v aplikácií nevyužíva framework. Miesto toho sa React aplikácia píše v JS súboroch, ktoré sa ako statický web posielajú na prehliadač užívateľovi.

V modernom webovom vývoji, obzvlášť ak ide o React aplikácie, sa veľa využívajú frameworky všetkých druhov a to z jednoduchého dôvodu. Niekoľko násobne uľahčujú a urýchľujú vývoj aplikácií.

Ako príklad frameworkov používaných spolu s React možno uviesť Next.js, Vite.js či dnes už čiastočne zastaralý Create React App.

Tieto frameworky poskytujú nástroje na spúšťanie aplikačného serveru, zjednodušujú konfiguráciu aplikácie a poskytujú ďalšie nástroje na správu aplikácie.

V aktuálnom stave aplikácie je potrebné posúdiť, či je vhodné takýto framework pridať, aby sa zachoval funkčný stav aplikácie. Zároveň je nutné zhodnotiť či táto zmena stojí za vynaložené úsilie.

## 3.2 NPM – Node package manager

NPM, ktorý je súčasťou Node.js, je používaný ako package manager pre aplikácie písané v JavaScript, niečo ako pip pre Python. Slúži teda na správu knižníc.

Aktuálne sa v aplikácií nepoužíva, knižnice sú manuálne pridávané do aplikácie ako ostatné zdrojové kódy.

Tento prístup vytvára možné problémy. Jedným z nich je náročnosť pridávania nových knižníc do aplikácie. To môže viesť vývojára k vytváraniu vlastných riešení na problém, ktorý bol už vyriešený a značne otestovaný v rámci existujúcej knižnice.

Dôvody sú jasné, šetrí to čas pri vývoji a knižnice ktoré sa používajú už roky sú otestované množstvom iných vývojárov, čiže sú väčšinou spoľahlivejšie a bezpečnejšie.

Ďalším možným problémom nepoužitia NPM je neaktuálnosť použitých knižníc. Ako je spomínané vyššie, knižnice sú pridané do projektu manuálne a zostávajú tam v tejto podobe, až kým ich niekto opäť manuálne neaktualizuje či nezmaže.

Znamená to že sa používajú knižnice ktoré môžu obsahovať chyby či už vo funkčnosti alebo v zabezpečení.

Podobne ako pri vývojovom frameworku, je nutné zohľadniť prípadné vynaložené úsilie na túto zmenu. Použitie NPM by však bolo prínosom pre vývoj aplikácie.

## 3.3 JSX

Keďže React je písaný v JavaScript, bol viazaný na syntax JS. React však prišiel so syntaktickým rozšírením pre JavaScript, ktoré sa nazýva JSX.

Cieľom JSX je pridať HTML syntax do JavaScript, čo zlepšuje čitateľnosť kódu. V praxi je pre React aplikácie takmer vždy používané JSX, čo naznačuje jednoduchosť písania aplikácií týmto spôsobom.

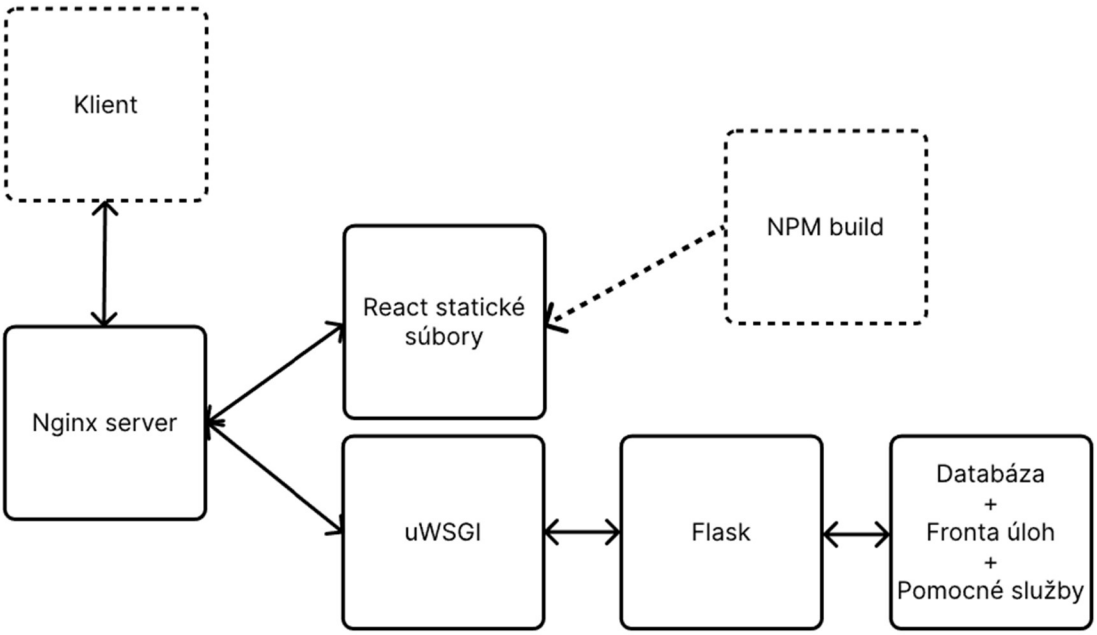
Pridanie JSX do aplikácie je pomerne jednoduché, stačí do závislosti (dependencies) v package.json pridať Babel. Babel je kompilér (compiler) či prekladač, ktorý preloží JSX do čistého JavaScriptu. Tento krok je závislý na pridanie NPM spomínanom vyššie, keďže package.json je súbor, v ktorom sú závislosti projektu.

## 3.4 Frontend a backend

Aktuálne je v aplikácií frontend aj backend prístupný na rovnakom porte pomocou Flask serveru. Všetky požiadavky sa posielajú na tento port a následne aplikácia, konkrétne sa to deje v app.py, rozlíši či sa jedná o požiadavku na API point alebo ide o požiadavku na aplikáciu ako takú, čiže JS a CSS súbory.

Tento prístup má výhodu v tom, že je jednoduchý, pomerný rýchlo naprogramovateľný. Používa sa hlavne pri jednoduchých stránkach, kde sa často ani nenachádza komplexnejší frontend či backend ale je to skôr statické HTML, prípadne JS, ktoré sa odošle na klient žiadne ďalšie požiadavky sa nedejú.

Nakoľko však táto aplikácia značne narástla, bolo by vhodné zvážiť či nie je lepšie, aby frontend a backend aplikácie boli oddelené. Projekt by bol po rozdelení jednoduchší na navigáciu pre programátora, nakoľko je okamžite jasné, kde sa ktorá časť aplikácie nachádza.




Obr. 2 - Navrhovaná zmena architektúry

## 3.5 CORS

Vyššie spomínanými zmenami vznikne problém s CORS - Cross-Origin Resource Sharing. CORS je mechanizmus, ktorý slúži na umožnenie komunikácia webovej aplikácie s aplikáciami z iného zdroja, typicky iná URL či port.

Prehliadač by bez použitia CORS zamedzil takúto komunikáciu kvôli bezpečnosti. V prípade vyššie navrhovaných zmien je to z dôvodu rozdielneho portu, ak je napríklad backend spustený na porte 3000 a frontend na porte 5000, jedná sa o "cross-origin" a prehliadač tieto požiadavky blokuje.

CORS sa povolí pridaním potrebných hlavičiek do odpovede serveru. Taktiež je potrebné povoliť HTTP metódu OPTIONS, ktorú prehliadač pošle ako "pre-flight" požiadavku a server na ňu musí odpovedať s vhodnou hlavičkou. [13]

Všetky tieto zmeny sa vykonajú na serveri a v prípade danej aplikácie je to opäť v súbore app.py. Náhľad na metódy ktoré pridajú potrebné hlavičky možno vidieť nižšie.

```
def build_cors_preflight_response():
    response = make_response()
    response.headers.add("Access-Control-Allow-Origin",
    request.host_url.replace("5000/", "3000"))
    response.headers.add('Access-Control-Allow-Headers', "Content-Type,
        Access-Control-Allow-Headers, Authorization, X-Requested-With")
    response.headers.add('Access-Control-Allow-Methods', "*")
    response.headers.add('Access-Control-Allow-Credentials', 'true')
    return response


def corsify_actual_response(response):
    response.headers.add("Access-Control-Allow-Origin",
```

```
    request.host_url.replace("5000/", "3000"))
    response.headers.add('Access-Control-Allow-Headers', "Content-Type,
        Access-Control-Allow-Headers, Authorization, X-Requested-With")
    response.headers.add('Access-Control-Allow-Methods', "*")
    response.headers.add('Access-Control-Allow-Credentials', 'true')
    return response
```

## 3.6 HTTP metódy

HTTP je protokol ktorý slúži na komunikáciu medzi serverom a klientom (užívateľom). V rámci tohto protokolu sú definované metódy, ktoré sa uvádzajú v hlavičke (headers) požiadavky.

Každá metóda je určená na konkrétny druh úkonu, ktorý sa po poslaní udeje na serveri. Ako príklad možno uviesť GET, ktorý slúži na vyžiadanie dát zo serveru, nenastáva žiadna zmena dát na serveri, poslaním tohto typu požiadavky užívateľ nič nemení. [14]

Na rozdiel od toho, metóda POST je použitá vtedy keď sa zapisujú či vytvárajú nové dáta. Toto je dôležité dodržiavať, nie len preto, že to je akýsi štandard ale zároveň to umožňuje mať na rovnakej URL viacero metód ktoré sa správajú rozlične.

Aktuálne je v aplikácií GET pre frontend časť a POST na všetky API požiadavky, teda aj tie ktoré len dotazujú dáta.

Je vhodné toto zosúladiť so štandardom. Žiadnym spôsobom to však neobmedzuje vývoj a použitie aplikácie a ak ostanú prostriedky pri vývoji, je možné to napraviť.

## 3.7 Autorizácia užívateľa

Po úspešnom prihlásení užívateľa do webovej aplikácie je potrebné autorizovať všetky požiadavky ktoré daný užívateľ pošle. Typicky je to riešené pomocou tokenu, ktorý môže byť uložený v Cookie alebo samostatne v lokálnej pamäti. Podoba tokenu sa môže líšiť, musí byť však šifrovaný aby nebolo možné ho ľubovoľne prepisovať a obchádzať tak zabezpečenie.

V súčasnom stave je v aplikácií toto riešené pomocou tokenu, ktorý sa vytvorí hashovaním údajov užívateľa. Tento hash sa následne uloží v pamäti aplikácie na backende, dočasnej pamäti redis, nie do databázy.

Toto riešenie je bezpečné a nevytvára to žiadne obmedzenia mimo potreby ukladať tento token v pamäti, čo je však v prípade nízkeho počtu užívateľov bezproblémové.

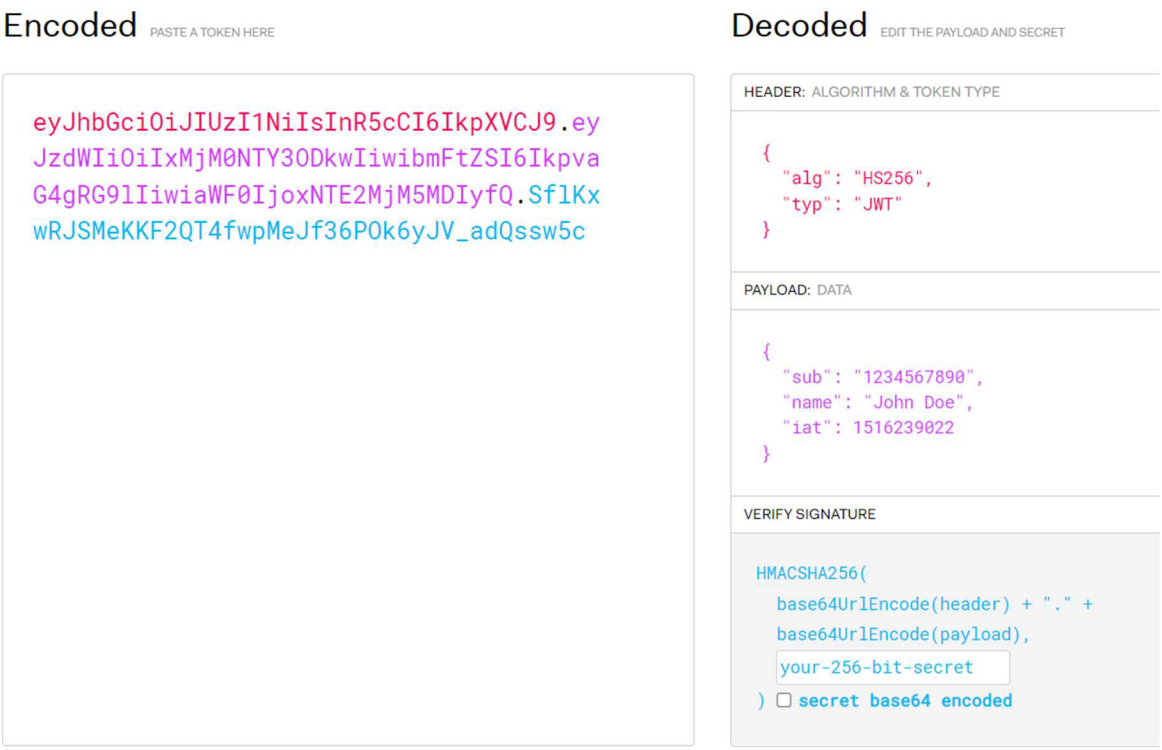
Je vhodné však spomenúť ďalšiu možnosť zabezpečenia, ktorou je JWT - JSON web token. Je to token, z ktorého sa pri čítaní získajú dáta v podobe JSON, teda JavaScript Object Notation, čiže JS objekt.

JWT využíva asymetrické šifrovanie, súkromný kľúč slúži na vytvorenie tokenu s informáciami a verejný kľúč umožňuje čítanie informácií bez znalosti súkromného kľúča. JWT sa často využíva bez verejného kľúča, čiže informácie z tokenu sú čitateľné

verejne. Kľúč je ale aj v tomto prípade podpísaný a je možné overiť či bol vytvorený pomocou daného súkromného kľuča.

Hlavný rozdiel a výhoda je, že ho nie je potreba ukladať, stačí mať v pamäti súkromný kľúč a overiť pravosť JWT. JWT sa skladá z troch častí, header (hlavička), payload (telo, dáta) a podpisu. Tieto tri časti sú oddelené bodkou, ako možno vidieť na Obr. 3. Do payload, dát je možné pridať vlastné pole, čo poskytuje flexibilitu využitia. [15]

Táto zmena nie je potrebná, no v prípade rastu počtu užívateľov sa môže ľahko implementovať.




Obr. 3 - JWT Debugger, ukážka JWT a jeho obsahu

# 4 DISKUSIA A ZAPRACOVANIE ZMIEN

## 4.1 Predstavenie návrhov a diskusia

Návrhy na zmeny boli predstavené vývojovému tímu. Bolo zvážené použitie Create React App oproti použitiu NEXT.js. Po dôslednom zhodnotení sa rozhodlo pre Create React App, hlavne kvôli jednoduchosti zakomponovania do existujúcej aplikácie.

NEXT.js je pokročilejší framework, poskytuje viac funkcionalít a pokrýva celý obe strany webovej aplikácie, frontend aj backend, čo nebolo potrebné. Je však možné že sa v budúcnosti zakomponuje do aplikácie a nahradí Creat React App.

Vzhľadom na nutnosť pridania NPM, v náväznosti na predošlí krok, bolo odsúhlasené aj pridanie NPM.

Pridanie možnosti písať JSX, teda pridanie knižnice Babel, bolo taktiež odsúhlasené. Výrazne to zjednoduší čitateľnosť kódu a vzhľadom na predošlé dva body je pridanie Babel jednoduché a rýchle. Nakoľko je JSX len rozšírením JS, nie je nutné existujúci kód prepracovať na JSX a teda zachovajú sa existujúce komponenty.

Štandardizovanie HTTP metód v požiadavkách bolo odsúhlasené, avšak bude sa zapracovávať postupne, dôležitým krokom je, aby všetky novo pridané funkcie dodržiavali tento štandard.

Ďalšie návrhy na úpravu aplikácie boli pridanie Redux, knižnice, ktorá zjednodušene povedané slúži ako centrálna pamäť pre React frontend. Zhodnotiť kontajnerizáciu s prípadným využitím Kubernetes.

Prechod na TypeScript, tento krok je možné uskutočniť podobne ako pridanie JSX, zatiaľ však nie je nutný.

V poslednom rade bolo vyžiadané preskúmať možnosti využitia GitLab CI/CD. CI/CD, Continuous Integration/Continuous development, znamená nastavenie automatizovaného testovanie a nasadenia novo vytvorených zmien automaticky pri ich integrácií do danej vetvy, typicky main/master vetva.

Je taktiež vhodné spomenúť problém s nedodržiavaním štandardov a dobrých praktík. Ako príklad možno uviesť nedodržiavanie camelCase v JS kóde. Metódy, z ktorých názvu nie je možné pochopiť, na čo daná metóda slúži. Zároveň však tieto metódy nie sú ani dokumentované. Je preto náročné pre nových vývojárov začať pracovať v tomto kóde. Nakoľko je projekt väčšinou riešený v rámci diplomových či bakalárskych prác, je to vcelku zásadný problém a bude sa viac dbať na čitateľnosť kódu.

## 4.2 Pridanie NPM, Create React App a JSX

Po odsúhlasení navrhovaných zmien sa prešlo k ich implementácií. Začalo sa pridaním Node Package Manager (NPM). Bolo potrebné pridať package.json, kde sú zapísané všetky závislosti a ich verzie. Na strane vývojára je potrebné nainštalovať

Node.js, ktorého súčasťou je NPM. Pre inštaláciu závislostí vývojár spustí “npm install” alebo “npm i”.

Po pridaní NPM bolo možné pridať do package.json závislosti pre create react app. Zároveň pre použitie Create React App bolo potrebné upraviť štruktúru projektu, nakoľko ako väčšina frameworkov, aj tu je potrebná konkrétna štruktúra ako vidieť nižšie na Obr. 4.


Obr. 4 Potrebná štruktúra pre framework Create React App [16]

Po pridaní frameworku a NPM, bolo možné pridať potrebné závislosti pre umožnenie použitie JSX. Ako bolo spomínané, jedná sa o pridanie knižnice Babel do závislostí.

V aktuálnom stave projektu boli všetky knižnice súčasťou projektu, nie externá závislosť. Bolo teda vhodné ich z projektu zmazať a pridať ako závislosť do package.json. Výslednú podobu package.json vidieť nižšie. Okrem závislostí (dependencies) boli pridané skripty ktoré slúžia na spustenie, test a build projektu.

---

```
{
  "dependencies": {
    "react": "^18.2.0",
    "react-beautiful-dnd": "^13.1.1",
    "react-dom": "^18.2.0",
    "react-leaflet": "^4.0.1",
    "react-router-dom": "6.8.2",
    "react-router-prompt": "^0.5.2",
    "react-scripts": "^5.0.1",
    "save-dev": "^0.0.1-security"
  },
```

```
  "scripts": {
    "start": "react-scripts start",
    "build": "react-scripts build",
    "test": "react-scripts test --env=jsdom",
    "eject": "react-scripts eject"
  },
  "browserslist": {
    "production": [
      ">0.2%",
      "not dead",
      "not op_mini all"
    ],
    "development": [
      "last 1 chrome version",
      "last 1 firefox version",
      "last 1 safari version"
    ]
  },
  "devDependencies": {
    "@babel/core": "^7.21.3",
    "babel-loader": "^9.1.2"
  }
}
```

## 4.3 Chyby po pridaní frameworku

Po zapracovaní vyššie spomínaných zmien vzniklo v projekte množstvo chýb (error), ktoré bránili chodu aplikácie. V projekte boli všetky závislosti, vrátane React, riešené globálne pre celú aplikáciu. Knižnice boli importované v prvom HTML súbore a boli dostupné v celej aplikácií.

Pri použití frameworku je však potrebné importovať knižnice v každom súbore samostatne. Toto zároveň vytvára čitateľnejší kód, pretože vývojár nemusí hľadať, kedy bola daná knižnica importovaná ale je to jasné už pri prvom pohľade na súbor. Tento problém sa vyriešil pridaním importov všade, kde to bolo potrebné, čo bol časovo náročný, no jednoduchý úkon.

Podobný problém vznikol s CSS súbormi, nakoľko pôvodná implementácia automaticky odoslala ku každému JS súboru aj k nemu náležiaci CSS súbor. V React aplikáciach je však nutné importovať CSS súbory podobne ako knižnice.

Pri nahrádzaní manuálne pridaných knižníc, knižnicami pridanými pomocou NPM nastal konflikt vo verziách knižníc. Tento problém bolo možné riešiť dvoma spôsobmi, nastavením správnej verzie alebo úpravou kódu aby sa používali novšie verzie. Nakoľko nové verzie zvyčajne prinášajú vylepšenia, bolo vhodné pridať novšie verzie a upraviť prípadné konflikty v kóde.

Hlavným problémom bol react-router-dom, nakoľko sa jednalo o prechod z v5 na v6, čiže dve hlavné vydania. Pri prechode z v5 na v6 bol odstránení hook useHistory, ktorý sa skrz `history.push("url")` používal v aplikácií na presmerovanie. Toto je jednoduché nahradiť pretože vo v6 bol pridaný hook useNavigate, ktorý sa pomocou navigate("url") použije rovnako ako presmerovanie.

Ďalšou odstránenou metódou bola prompt, ktorá zabezpečila zablokavanie zvyšku stránky a zobrazila Dialóg s upozornením, prípadne otázkou. Táto metóda nemá vo v6 náhradu, je na to však knižnica react-router-prompt, ktorá sa pridala plnila tento účel.

Verzia v6 je oproti v5 omnoho príjemnejšia na prácu pri vývoji, čiže tento krok priniesol ďalšie zlepšenie vývoja aplikácie.

Pri prechode z v5 na v6 a zakomponovaní JSX, nastal problém s chybným vykresľovaním navbaru a obsahu aplikácie. Čo presne spôsobilo tento problém nie je jasné, no vzhľadom na rozloženie aplikácie bola chyba v routeri. Najrýchlejším riešením bolo napísať nový router s použitím pôvodného routeru ako predlohy pre výslednú funkcionalitu.

Taktiež bolo potrebné pridať presmerovanie, ak užívateľ nie je presmerovaný. V pôvodnej aplikácií toto riešil Flask server, ktorý bez prihlásenia namiesto požadovanej stránky vrátil stránku pre prihlásenie. Vzniknutá komponenta sa volá LoggedInWrapper, nakoľko vytvára obálku nad komponentami v nej.

```
export default function LoggedInWrapper(props) {
    const navigate = useNavigate();

    window.LoaderProgress = LoaderProgress("step-progress-bar-object");
    window.memorysets = {...};

    function initialFetch() {...};

    if (isLoggedIn()) {
        initialFetch();
    }
    else
        navigate(ROUTE.LOGIN);
    return <Outlet/>
```

Túto komponentu vidieť vyššie a jej použitie je zobrazené v kóde nižšie. Metóda initialFetch slúži na prvotné načítanie dát pre aplikáciu. Táto metóda vychádza z použitia pôvodného routeru.

```
class Router extends Component {
    router = createBrowserRouter(
        createRoutesFromElements(
            <Route errorElement={<Error404/>}>
                <Route path={ROUTE.LOGIN} element={<ComponentLogin/>}/>
                <Route element={<Navbar/>}>
                    <Route element={<LoggedInWrapper/>}>
                        <Route path={ROUTE.SVOZ} element={<Svoz/>}/>
                        <Route path={ROUTE.UCET} element={<Ucet/>}/>
                        <Route path={ROUTE.VOZOVY_PARK}
element={<VozovyPark/>}/>
                        <Route path={ROUTE.SBER} element={<Sber/>}/>
```

```
                        <Route path={`${ROUTE.VYPOCET_SVOZ}:id`}
element={<VypocetSvoz/>}/>
                        <Route path={ROUTE.VYPOCTY} element={<Vypocty/>}/>
                        <Route path={ROUTE.CIRETO} element={<Cireto/>}/>
                        <Route element={<AdminWrapper/>}>
                            <Route path={ROUTE.NASTAVENI}
element={<Nastaveni/>}/>
                            <Route path={ROUTE.VZORY_EMAILU}
element={<VzoryEmailu/>}/>
                        </Route>
                        <Route path={ROUTE.ZAMESTNANCI}
element={<Zamestnanci/>}/>
                        <Route path={ROUTE.UVOD} element={<Index/>}/>
                    </Route>
                    <Route exact path="*" element={<Error404/>}/>
                </Route>
            </Route>
        )
    )

    render() {
        return (
            <ContextProvider>
                <ContextConnector>
                    <RouterProvider router={this.router}/>
                </ContextConnector>
            </ContextProvider>
        )
    }
}

ReactDOM.createRoot(document.getElementById("root")).render(<Router/>)
```

# 5 CIRETO

Po zapracovaní všetkých zmien spomínaných v predošlej kapitole sa mohlo prejsť k realizácií modulu, ktorý je hlavným predmetom tejto práce. Bude sa jednať o samostatnú stránku, ktorá nesie názov CIRETO. Tento modul bude slúžiť na zhodnotenie investičných zámerov v odpadovom hospodárstve. Výsledkom bude návrh a následne aj realizácia frontendu a prípadne spojenie so samotnými optimalizačnými modulmi patriaci k tomuto modulu na úrovni backendu. Tie sú však predmetom inej práce.

## 5.1 Návrh vizuálu a rozloženia stránky CIRETO

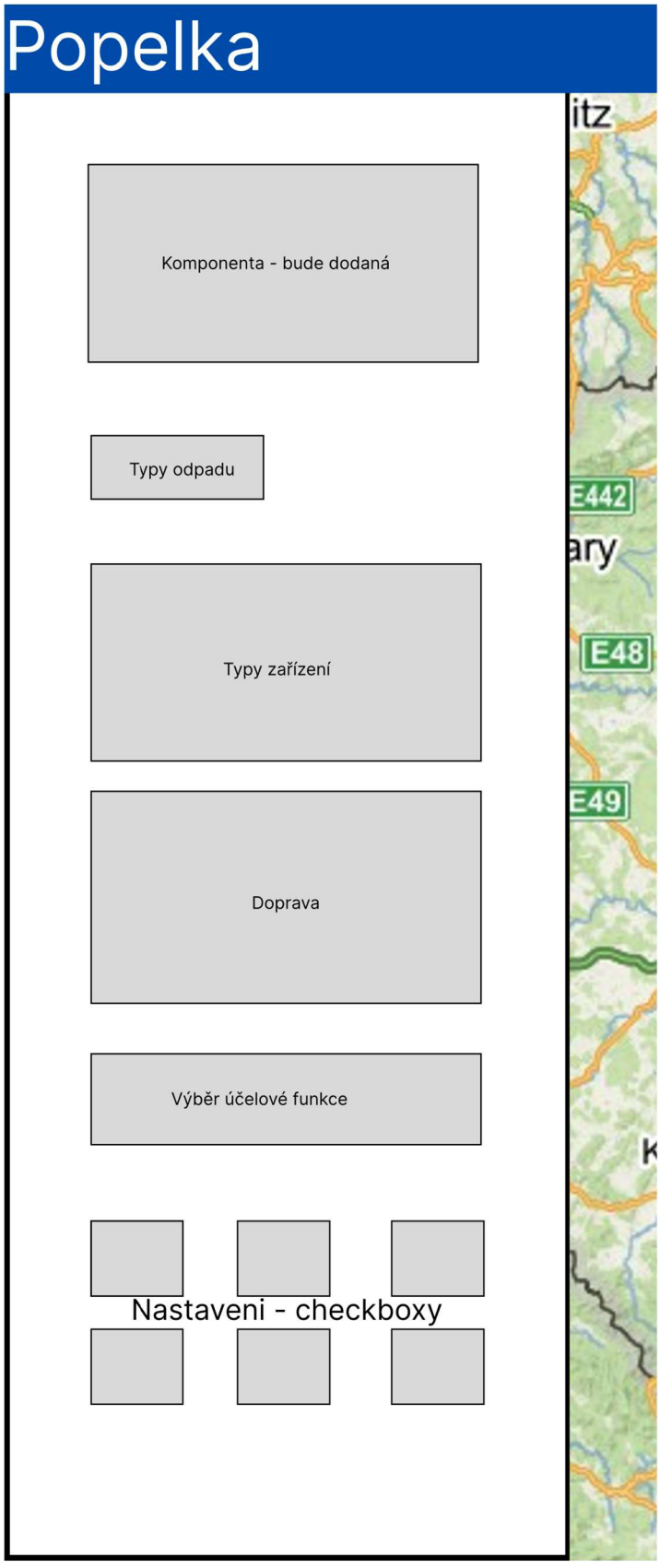
Po diskusii so zadávateľom bol vytvorený náčrt ako by mal vyzerať výsledný frontend. Tento náčrt bol následne vytvorený v programe Figma. Figma slúži na vytváranie náčrtov web-stránok.

Stránku je možné rozdeliť na tri časti, ľavá strana sa bude týkať nastavenia vstupov, pravá bude slúžiť na zobrazenie dát a grafov z výstupu a v strede bude interaktívna mapa so zakreslenými dátami.

Tvorba pravej strany so zobrazením dát vo forme grafov nebude súčasťou tejto práce, nakoľko zo strany zadávateľa nebolo jasné, akú podobu by mali mať tieto grafy a bude to známe až po dokončení inej práce a analýze výstupov danej práce.

### 5.1.1 Nastavenie vstupov

Na ľavej strane stránky sa bude nachádzať časť pre nastavenie vstupov optimalizácie, napríklad typy zariadení, typy odpadov, doprava, atď. Tento návrh je vidieť na Obr. 5. Jedná sa o prvotný návrh rozloženia a výsledná podoba sa môže líšiť. Nakoľko bude tento modul do istej miery univerzálny, bude združovať niekoľko scenárov optimalizácie, budú sa tieto nastavenia vyvíjať s postupným pridávaním funkcionalít.

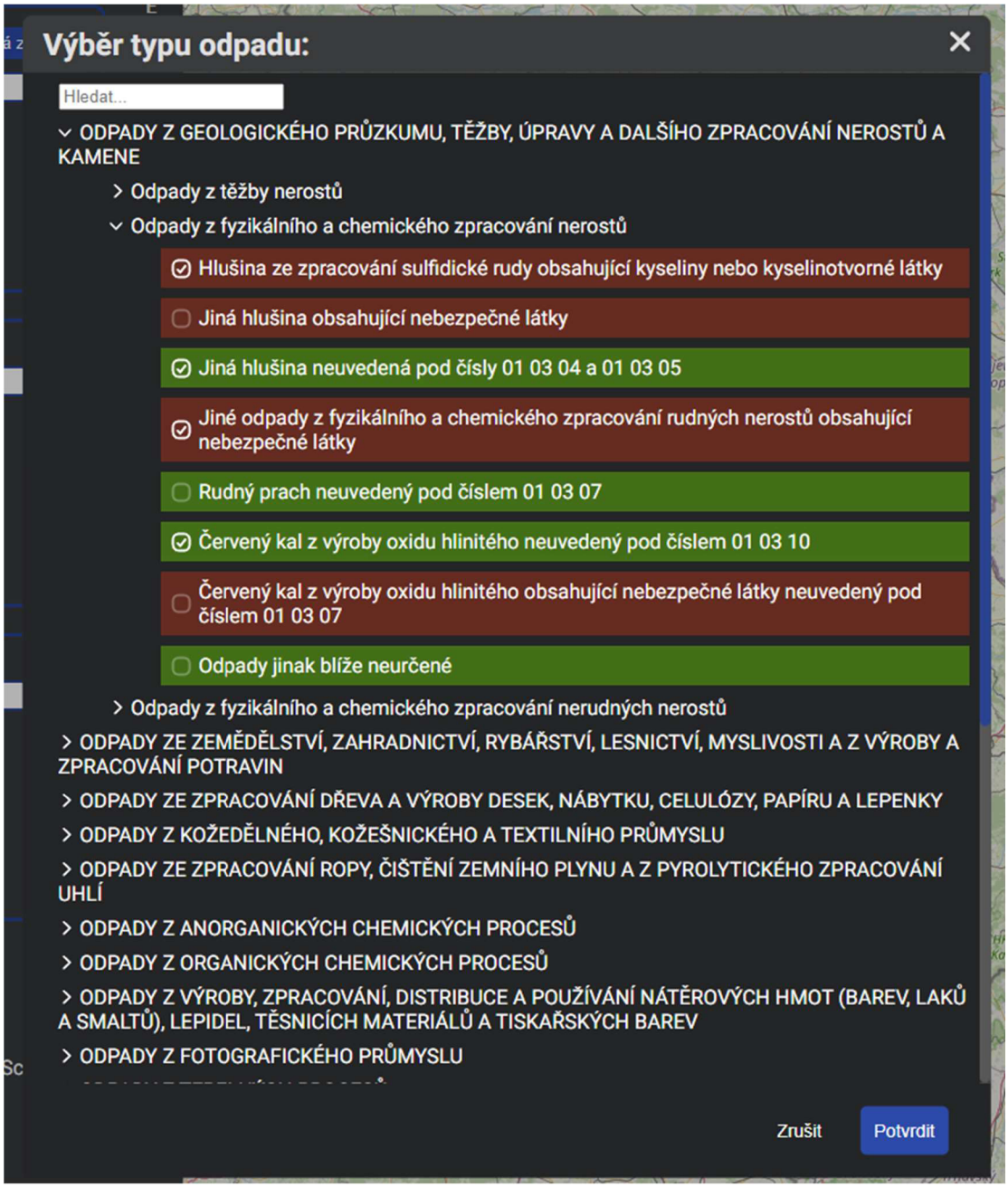



Obr. 5 Návrh nastavenia vstupov

V hornej časti stránky vidieť priestor pre komponentu, ktorá bude neskôr dodaná iným vývojárom.

Box “typy odpadu” bude nahradený tlačidlom ktoré otvorí dialóg s typmi odpadov, tieto typy majú stromovú štruktúru. Hlavná kategória, následne podkategória a potom typ odpadu. Pre tento účel bola vytvorená komponenta, ktorá vychádzala už z existujúcej komponenty, bolo však nutné ju upraviť, preto sa nepoužila pôvodná.

Každý typ odpadu má údaj či je nebezpečný alebo bezpečný. Toto je zobrazené pomocou podfarbenia riadku daného typu červenou alebo zelenou farbou. Toto riešenie je vhodnejšie než len samotný text a zároveň to výrazne oddeľuje jednotlivé riadky. Možno vidieť na Obr. 6.




Obr. 6 Komponenta "Výběr typu odpadu"

Dáta zobrazené na obrázku 10 boli získané vo formáte xlsx zo stránky katalogodpadu.cz . Tie sú dané podľa prílohy 1 Vyhlášky č. 8/2021 Sb.

Zo súboru xlsx, teda Excel, boli dáta exportované do csv, kvôli problémom s formátom niektorých buniek. Tento súbor sa následne použil na vygenerovanie SQL skriptu, pomocou jazyka Python. Skript možno vidieť nižšie.

```
import json

katalog = json.load(open('Katalog-odpadu.json', encoding="utf-8"))

query = "INSERT INTO waste_types_detailed(id, name, parent_id, dangerous) VALUES
"

def createSqlRow(item, parentId):
    id = item["id"]
    if isinstance(id, int):
        id = str(id)
    if isinstance(parentId, int):
        parentId = str(parentId)

    dangerous = "0"
    if 'dangerous' not in item:
        dangerous = "NULL"
    elif(item['dangerous'] == "N"):
        dangerous = "1"

    id = id.replace('*', '')

    row = " \n ('" + id + "', '" + item["name"] + "', '"
          + parentId + "', " + dangerous + "),"
    print(row)
    return row

for i in katalog["waste"]:
    query += createSqlRow(i, "NULL")
    for j in i["items"]:
        query += createSqlRow(j, i["id"])
        for k in j["items"]:
            query += createSqlRow(k, j["id"])

query += ";"
with open('katalog-odpad.sql', 'w', encoding="utf-8") as file:
    file.write(query)
```

V databáze sa vytvorila tabuľka s typmi odpadov. Bolo potrebné premeniť stromovú štruktúru týchto dát na jednotlivé riadky tabuľky. Výsledná tabuľka sa nazvala *waste_types_detailed*, a mala nasledovné stĺpce: *id, name, parent_id, dangerous*. *Id* je identifikátor odpadu, hlavné kategória má formát XX, podkategória tejto kategórie má formát XX YY, a konkrétny typ v tejto podkategórií má formát XX YY ZZ, pričom

na jednotlivých miestach sú čísla. Stĺpec je však typu VARCHAR(8), aby sa zachovali medzery a bolo to ľahko čitateľné v tabuľke.

*Name* je typu VARCHAR(200), *parent_id* označuje kategóriu nad aktuálnym typom a je tiež typu VARCHAR(8), a *dangerous* je BOOLEAN, ktorý označuje nebezpečnosť odpadu. BOOLEAN je v skutočnosti len synonymum pre TINYINT(1), spolu s obmedzením na hodnoty 0 a 1.

Do Python aplikácie sa pridal jednoduchý API point, ktorý tieto dáta z aplikácie pošle na klienta. Dáta z databázy sa získajú spustením SELECT dotazu.

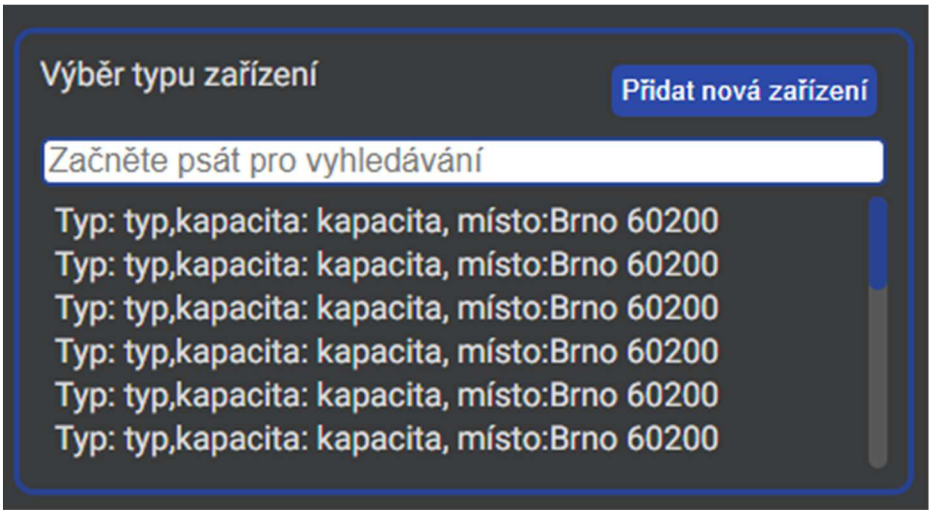
Na strane klienta sa tento zoznam následne pretransformuje z listu na stromovú štruktúru a vykreslí do vyššie uvedenej komponenty.

Bolo zvážené či je vhodnejšie túto transformáciu vykonať na serveri, či na strane klienta. Pre klienta je to minimálna záťaž a užívateľ si toho nevšimne. Pre server by to však s narastajúcim množstvom užívateľov mohla byť rozpoznateľná záťaž. Pre šetrenie vyťaženia serveru je možné čo najviac výpočtovej náročnosti presunúť na klienta. Treba však dbať na to aby to nespomalilo aplikáciu príliš, neboli posielané citlivé dáta či sa neprenášalo priveľa dát. Ale nie vždy to je jednoznačné a neexistuje na to univerzálne pravidlo.

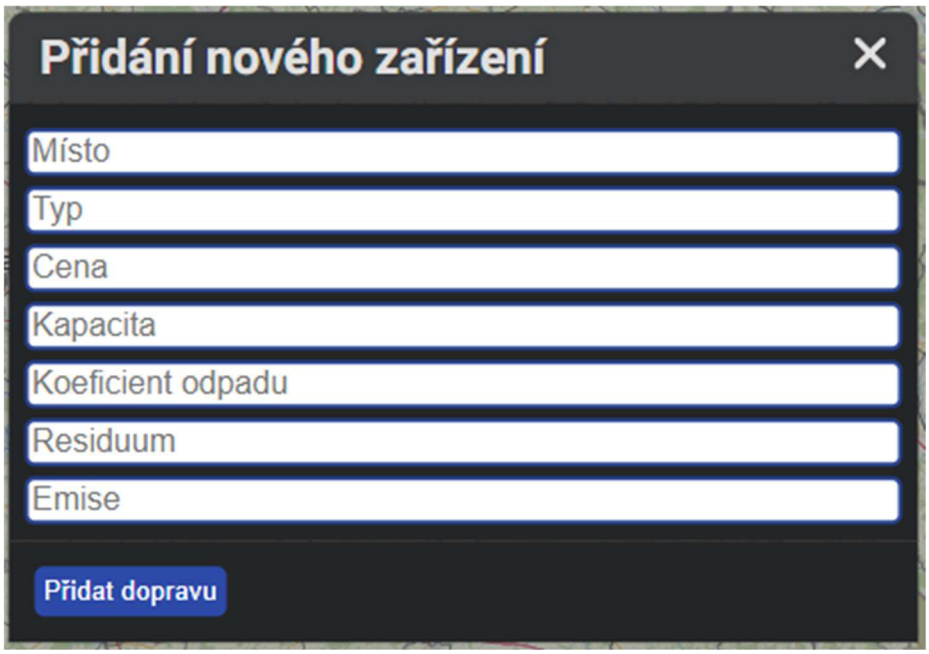
Ďalej bola požiadavka aby si užívateľ mohol predvoliť a uložiť typy odpadov, ktoré často používa, nakoľko prejsť pri každej optimalizácií, všetky typy odpadov je časovo náročné. To je pomerne jednoduché zrealizovať, stačí pridať tabuľku, ktorá bude mať záznam o týchto predvoľbách. Musí obsahovať id užívateľa, meno tejto predvoľby a typy odpadov ktoré obsahuje. K tomu sa vytvorí opäť API point ktorý pošle tieto predvoľby. Stačí posielať typy iba ako označenia odpadov, nakoľko v aplikácií sa už podrobnosti o typoch nachádzajú.

Jednotlivé boxy budú vyplnené konkrétnymi dátami vo forme zoznamu s vyhľadávacím poľom. Nakoľko takýchto boxov bude v aplikácií viacero, bola na to vytvorená samostatná komponenta ktoré sa použije pre typy zariadení, dopravu a výber účelovej funkcie. Náhľad tejto komponenty možno vidieť nižšie (Obr. 7), v tomto prípade ide o výber typu zariadenia aj s tlačidlom pre pridanie dopravy ktoré zobrazí formulár pre vytvorenie.

V čase písania tejto práce neboli dostupné dáta k týmto tabuľkám, nie je to však problémom, dodatočné napojenie na API a databázu je jednoduché a prakticky totožné ako ostatné API pointy, ktoré prevolajú dotaz nad databázou a odošlú dáta. Z tých sa len naformátuje zobrazovaný text a ten sa zobrazí.

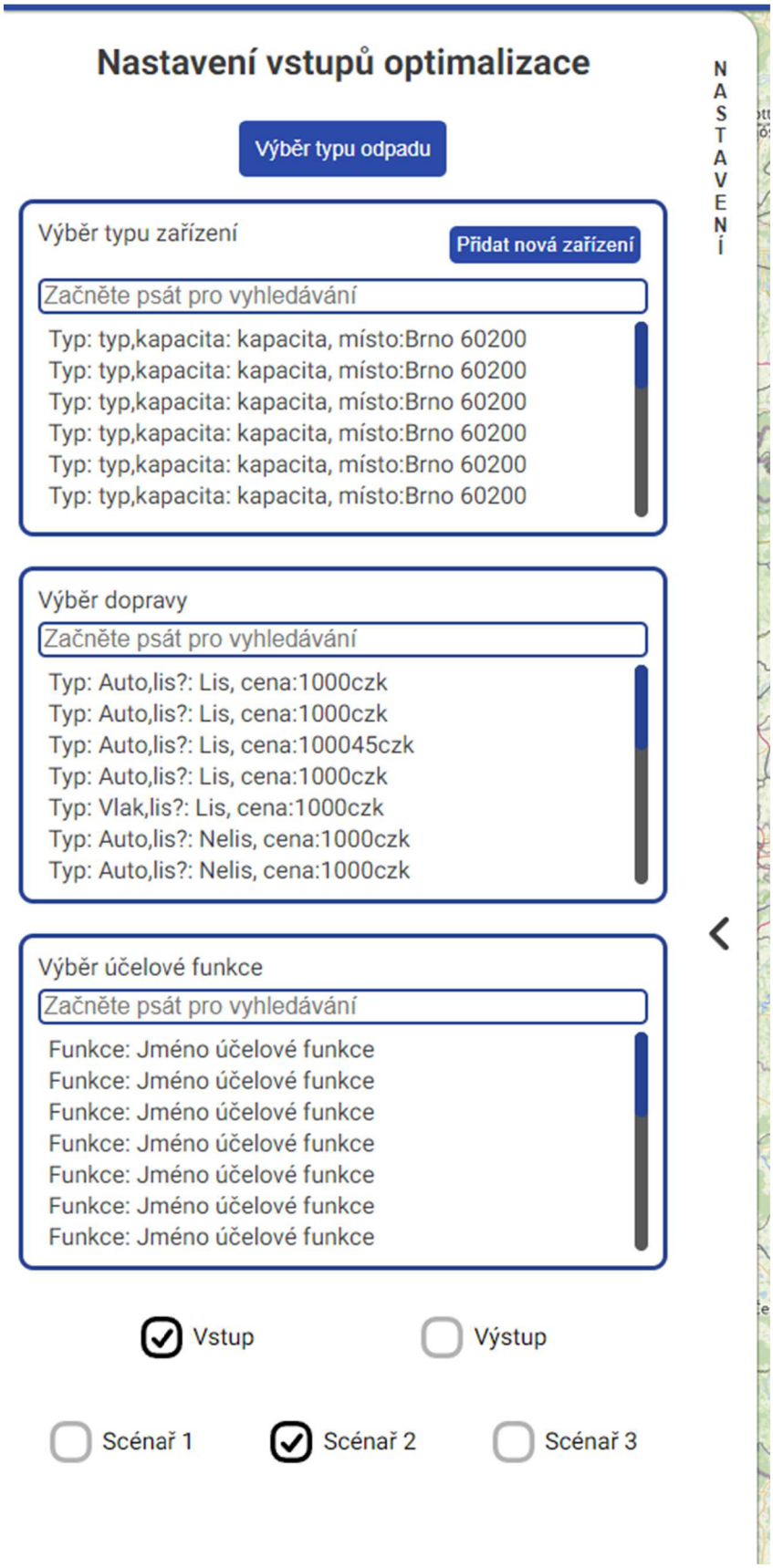



Obr. 7 Komponenta pre zoznam s vyhľadávaním a tlačidlom




Obr. 8 Formulár pre pridanie novej dopravy

Ďalej sa tu budú nachádzať prepínače scenárov, teda predmetu optimalizácie. Pre tento účel boli zvolené "checkboxy" nakoľko z nich je jasné či je daná vec zapnutá alebo nie a sú užívateľsky prívetivé. Na toto bola tiež vytvorená komponenta a počet či rozloženie týchto checkboxov je možné ľahko meniť podľa potreby. Túto komponentu možno vidieť nižšie, už použitú v rámci nastavení (Obr. 9). V tomto prípade môže byť aktívna len jedna komponenta v riadku, je to však možné ľahko zmeniť prepísaním JS funkcie. Často sa pre takéto prípady používa skôr radio button, táto komponenta je však viac vizuálne zhodná so zvyškom aplikácie. Výslednú časť pre nastavenie vstupov možno vidieť nižšie, tentoraz v svetlom móde, zatiaľ bez komponenty, ktorá bude dodaná ako bolo spomínané.



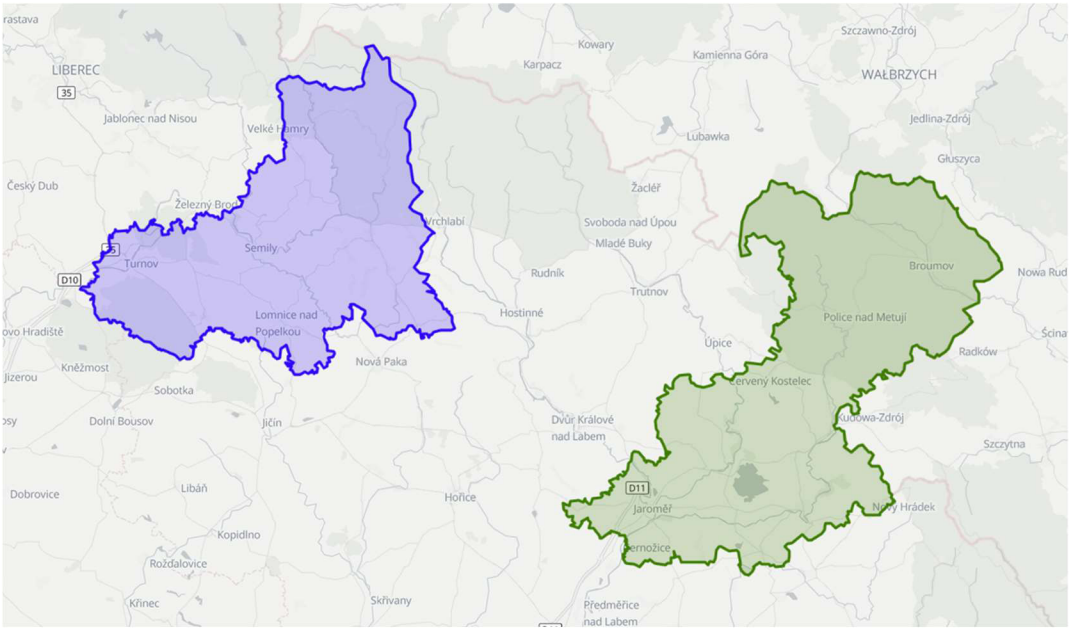

Obr. 9 Komponenta pre nastavenie vstupov optimalizácie

### 5.1.2 Mapa

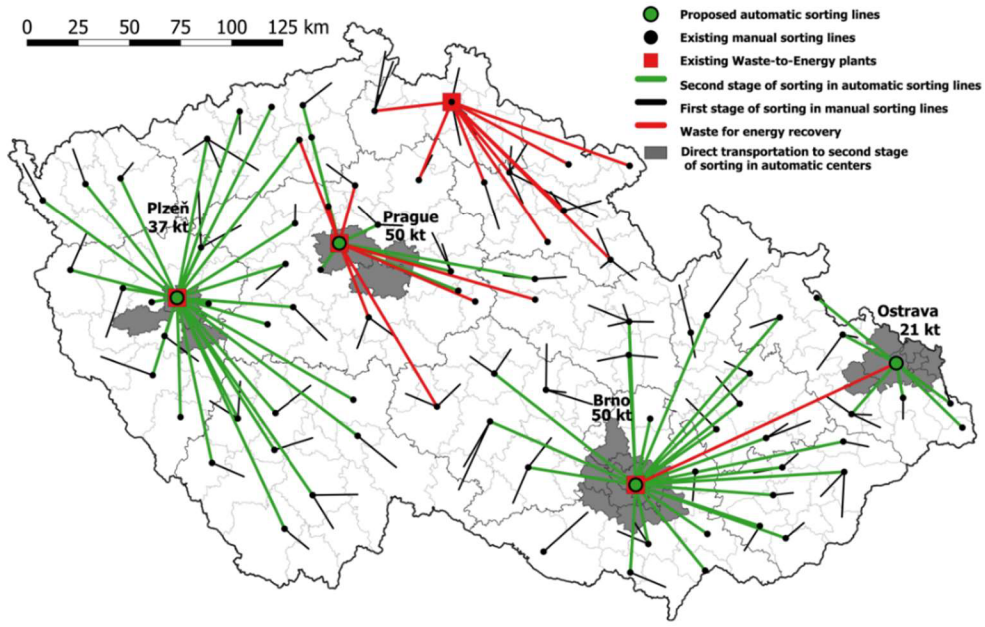
V strede stránky sa bude nachádzať mapa vytvorená pomocou React leaflet. Do tejto mapy sa po skončení optimalizácie budú zakresľovať výsledné dáta. Pôjde o grafické zobrazenie navrhovaných tokov odpadu, zakreslenie polohy zariadení a skládok.

Zadávateľom boli dodané návrhy možnej podoby výsledných máp. Z týchto návrhov sa vyhodnotilo aké komponenty bude potrebné vytvoriť pre zobrazenie dát v mape. Poskytnuté návrhy možno vidieť nižšie na Obr. 10-13.

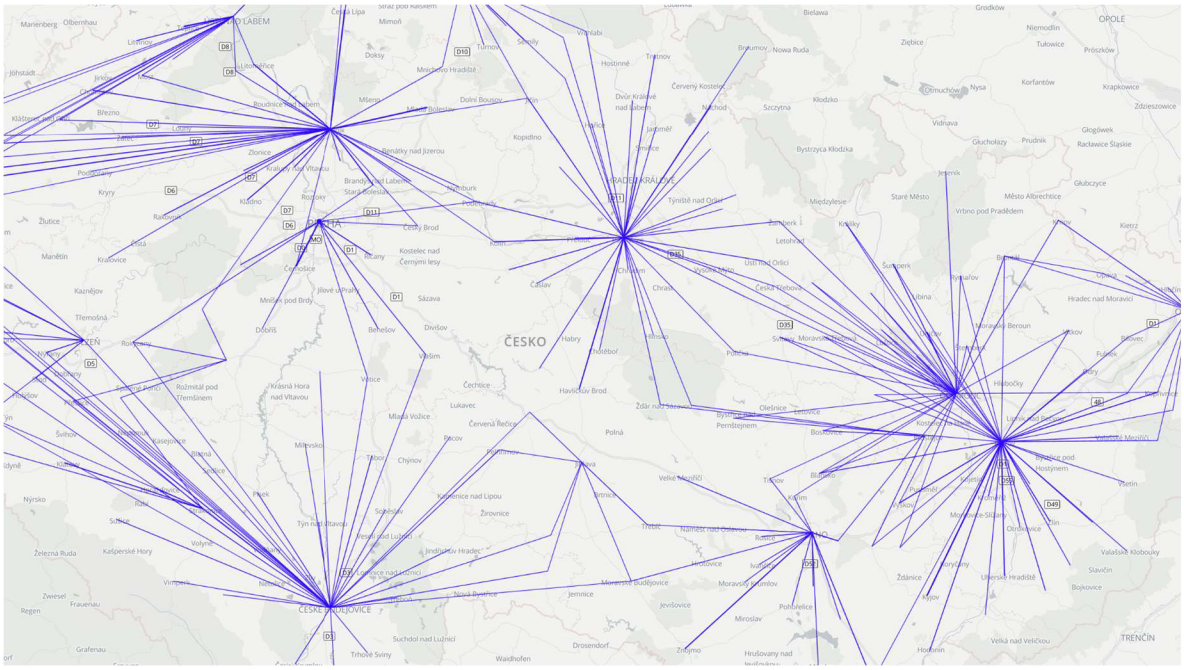
Návrhy na Obr. 10 a 12 boli nahradené vlastnými obrázkami, pretože pôvodné návrhy podliehali utajeniu.



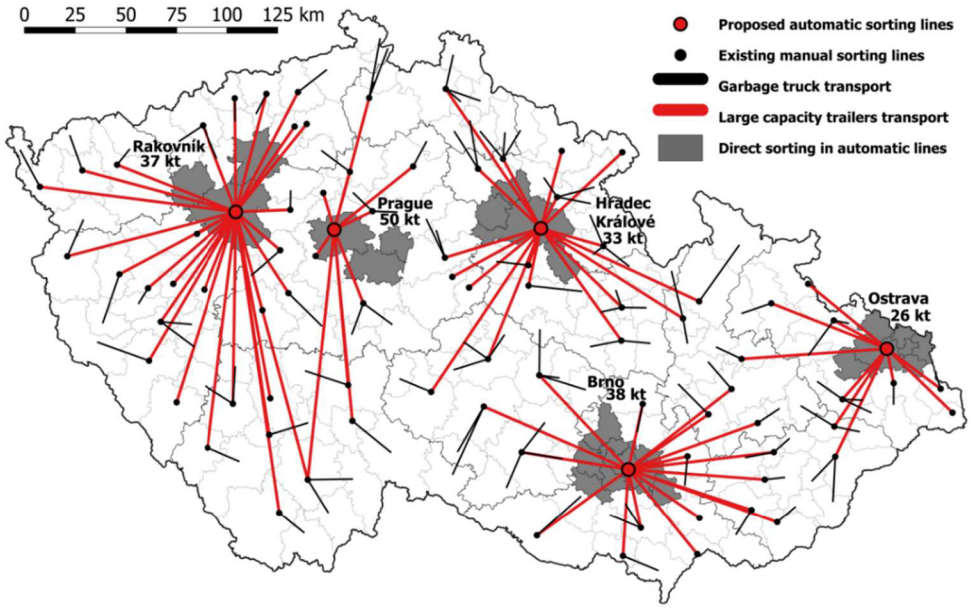

Obr. 10 Návrh mapy s vyznačenými oblasťami podľa zvozovej firmy




Obr. 11 Návrh mapy s tokmi odpadov a zariadeniami na spracovanie




Obr. 12 Návrh mapy s vyznačenými tokmi




Obr. 13 Návrh mapy s vyznačenými tokmi odpadu do triediacich zariadení

Ako je z týchto návrhov zjavné, väčšina dát tvorí graf, orientovaný alebo neorientovaný s ohodnotenými hranami. Hodnota hrán však nie je zobrazená číslom ale bolo požadované, aby to bolo zo zobrazenia zrejmé vďaka hrúbke čiar pre jednotlivé hrany. Hrúbka čiar je ako ohodnotenie nepresné ale ide len o približnú informáciu pre užívateľa, nakoľko podrobné dáta si bude vedieť nájsť v pravej strane stránky.

Pre zobrazenie takéhoto grafu v React Leaflet je potrebné poznať súradnice daných obcí, v poskytnutých príkladových dátach boli zadané len kódy obcí, označované ZUJ, medzinárodne NUTS čo je skratka z Nomenclature of Units for Territorial Statistics.

Pre mapovanie NUTS na súradnice bol vytvorený API point, ktorý získa z databázy a pošle JSON so všetkými nuts a ich súradnicami. Tieto dáta už v databáze existovali s predchádzajúceho vývoja. Dáta boli v dvoch tabuľkách a získali sa dvoma dotazmi.

Následne sa získané zoznamy spojili do jedného a odoslali na frontend. Ďalšie API point budú veľmi podobné, na ukážku je možno vidieť kód pre tento API point nižšie.

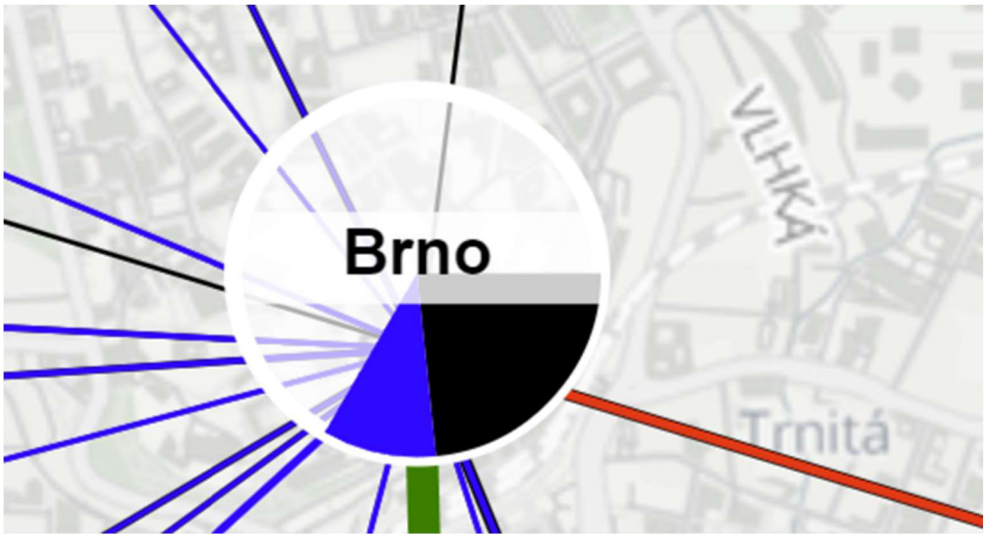
Názov metódy definuje názov pointu v URL. Táto podoba vychádza z už existujúceho kódu v projekte.

```
@point({'system_demo'})
def getmapofnutstocoordinates(payload, user, db):
    with db.cursor() as cursor:
        cursor.execute('SELECT name, nuts_id, latitude, longitude '
                       ' FROM municipal_parts mp')
        data = cursor.fetchall()
        cursor.execute('SELECT name, nuts_id, latitude, longitude '
                       ' FROM municipalities m')
        data.extend(cursor.fetchall())
        data_dict = {}
        for row in data:
            data_dict[row['nuts_id']] = {
                'name': row['name'],
                'latitude': row['latitude'],
                'longitude': row['longitude']
            }

        return utils.json.orjsonify({'success': True, "data": data_dict})
```

Z takto získaných dát bol vytvorený graf pomocou React Leaflet komponenty Line, ktorá vytvorí úsečku medzi dvoma bodmi danými ich súradnicami. Vrcholy grafov boli dočasne vytvorené pomocou komponenty Circle, no v ďalšom kroku budú nahradené.

Ako ďalšia požadovaná komponenta, ktorá nie je zobrazená na prvotných návrhoch, bol kruhový graf, ktorý bude zobrazovať podiel jednotlivých odpadov v daných bodoch, teda zariadeniach. Bude slúžiť aj ako vrchol grafu pre body so zariadeniami. Táto komponenta bola vytvorená ako SVG grafika s názvom obce ku ktorej graf náleží. SVG grafiku je možné zobraziť nad React Leaflet mapou za použitia komponenty SVGOverlay.

Pre SVGOverlay je potrebné tiež zistiť súradnice plochy na ktorej sa bude grafika vykresľovať, konkrétne ľavý horný a pravý dolný roh obdĺžnika. Je to potrebné, pretože táto komponenta slúži na zobrazenie grafiky na mape, nie nad ňou ako napríklad legenda. Pre tento účel sa použije vyššie spomínané mapovanie súradníc od ktorých sa spočítajú súradnice rohov obdĺžnika. Veľkosť je zámerne príliš veľká, pretože tento obdĺžnik slúži len ako neviditeľné plátno a škálovať sa už bude samotná grafika podľa potreby.




Obr. 14 Vytvorená komponenta pre kruhový graf s dátami

Farby jednotlivých odpadov sú definované v samostatnej triede, aby ich bolo možné ľahko zmeniť a týmto spôsobom sa taktiež môžu zdieľať medzi jednotlivými komponentami a nie je nutné tieto farby prepisovať na viacerých miestach. Podobne je riešené aj mapovanie označenia odpadu, ktoré sú v dátach anglicky, napríklad rdf, mmw či slag. Cez túto triedu sa im priradí český názov daného odpadu. Z tejto triedy je tiež možné následne vygenerovať legendu k mape. Kód triedy je nasledovný:

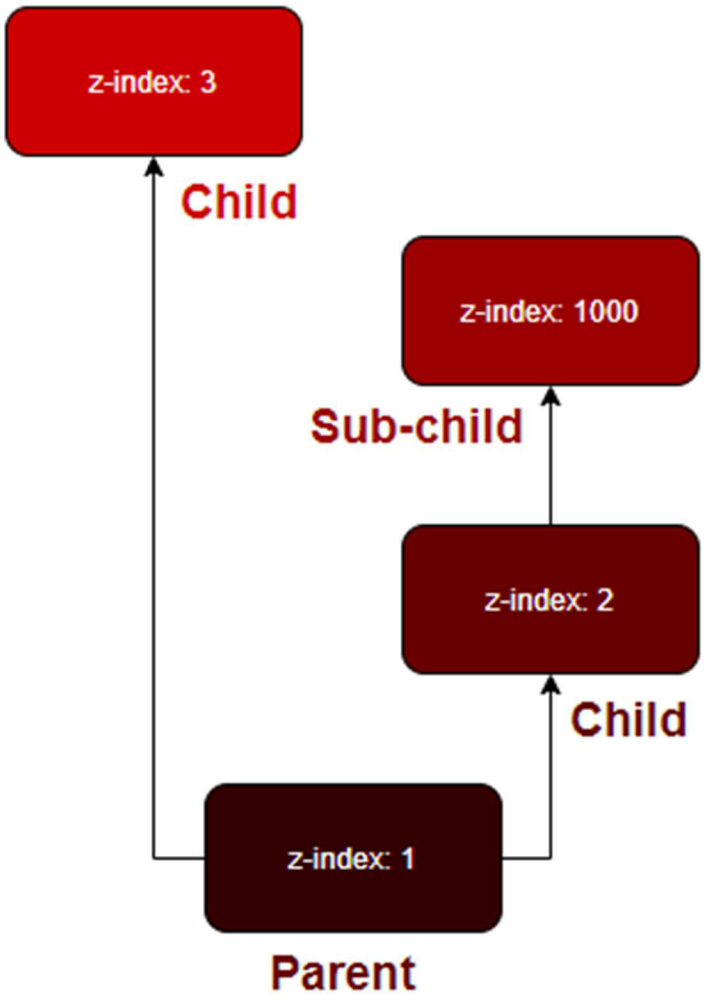
```
export const wasteColor = {
    mmw: "black",
    bulky: "blue",
    rdf: "red",
    slag: "green"
}

export const wasteName = {
    mmw: "Netřízený",
    bulky: "Nadrozměrný",
    rdf: "Stavební",
    slag: "Slag"
}

export const waste = {
    mmw: {name: wasteName.mmw, color: wasteColor.mmw},
    bulky: {name: wasteName.bulky, color: wasteColor.bulky},
    rdf: {name: wasteName.rdf, color: wasteColor.rdf},
    slag: {name: wasteName.slag, color: wasteColor.slag},
}
```

Ako nasledujúca komponenta bola vytvorená vyššie spomínaná legenda. Leaflet, nie React Leaflet, má príklady ako vytvoriť legendu pre mapu. No nakoľko v tejto aplikácii sa využíva React Leaflet, čo je wrapper nad Leafletom, bolo pridanie Leaflet závislosti len kvôli tomuto vyhodnotené ako nadbytočné.

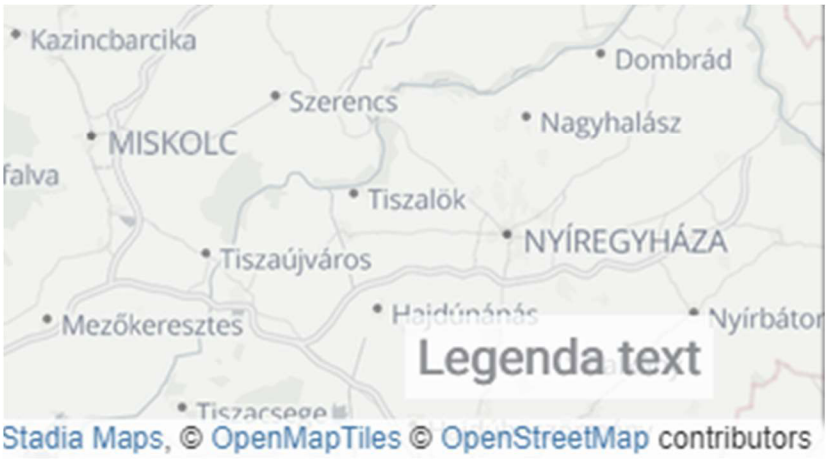
Legenda sa následne implementovala priamejším a jednoduchším spôsobom, a to ako div ktorý je vykreslený v absolútnej pozícií nad mapou. Táto komponenta funguje len ako obal nad potomkami, je teda možné ju znovu použiť na komponenty, ktoré bude potrebné vykresliť nad mapu, nezávisle na súradniciach.

Dôležité v tejto implementácií je, že legenda je príbuzná komponenta mapy, nie potomkom, kvôli spôsobu akým funguje z-index. Toto je graficky vysvetlené na obrázku nižšie a z-index bol nastavený na 9999 čo je náhodná veľká hodnota, ktorá sa podobne zvykne voliť keď je potrebné zaručiť vykreslenie nad príbuzné komponenty stránky.

Výslednú podobu mapy s grafom a legendou je vidieť nižšie. V nasledujúcej sekcii bude druhé zobrazenie, kde sa vyznačujú jednotlivé hranice obcí.




Obr. 15 Vykreslovanie podľa z-indexu [17]




Obr. 16 Komponenta legenda nad mapou

Po zoskladaní týchto komponente do mapy vzniklo prvé zobrazenie dát, ktoré možno vidieť nižšie (Obr. 19).

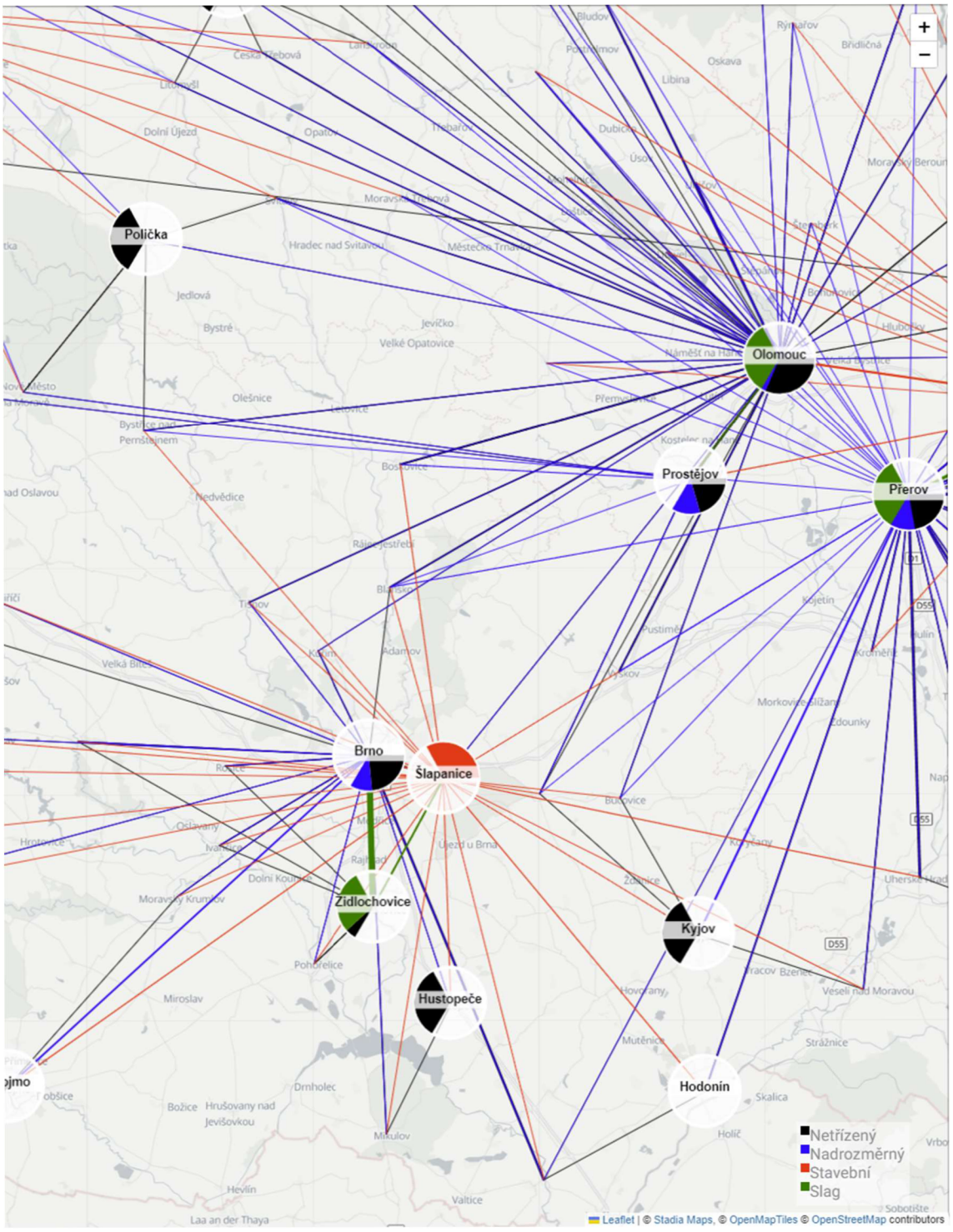
Nakoľko má aj táto komponenta byť znovu použitá inými vývojármi v budúcnosti, bol pridaný JSdoc (Obr. 17), čo je komentár ktorý IDE priradí k danej funkcií či triede a zobrazí vývojárovi (Obr. 18).

```
/**
 *
 * @param values list of {name: string, color: string}
 * @returns {JSX.Element}
 * @constructor
 */
2 usages  Červenka, Roman *
export function LegendListWithColorSquares({values}) {
```

Obr. 17 Príklad písania JSdoc

```
<>
<LegendListWithColorSquares values={Object.values(waste)
<MapContainer
    <ZoomCont
    <StyledMa
export function LegendListWithColorSquares(
    {values}): JSX.Element
Params: values – list of {name: string, color: string}
frontend/.../Legend.js
```

Obr. 18 Zobrazenie JSdoc v IDE




Obr. 19 Zobrazenie dát typu graf s kruhovými grafmi na mape

Ďalej bolo potrebné umožniť zobrazenia hraníc obcí, miest či krajov a prideliť vzniknutému polygónu farbu, napríklad na označenie firmy ktoré spracúva odpad v danej časti krajiny. Toto je možné docieliť pomocou GeoJson dát. Sú to dáta ktoré majú podobu množstva bodov, v podobe súradníc, ktoré tvoria n-uholník.

Tieto dáta je možné nájsť voľne dostupné online, konkrétne pre túto prácu boli dáta získané zo zdroja github.com/siwekm/czech-geojson, kde sa nachádzajú spracované dáta do formátu GeoJson. Autor ich získal z data.gov.cz .

Problém pri týchto dátach je ich množstvo. Napríklad dáta pre všetky obce v ČR, ktorých je v tomto súbore 6258, majú 95 MB. Ak by sa teda mali pri každom načítaní aplikácie načítavať aj tieto dáta v plnom rozsahu, znamenalo by to zdržanie desiatok sekúnd, čo je pri webovej aplikácií veľmi nežiadúce a je potrebné sa tomu vyhnúť, ak je možné.

Väčšinou však nie je potrebné naraz zobraziť tieto dáta pre všetky obce, ale iba pre časť z nich, čo umožňuje násobne zredukovať veľkosť prenášaného JSON súboru.

Bežne nastavený HTTP server dáta komprimuje, no táto komprimácia samotná nie je v tomto prípade dostatočným riešením problému.

Boli zvážené aj iné alternatívy ako zmenšiť veľkosť tohto súboru. Jeden spôsob by bol premenovať kľúče, napríklad miesto "city", by bolo len "c". Toto bolo otestované ale zo súboru ktorý má 95 MB, vznikol súbor o veľkosti 91,5 MB, čo síce je úspora dát, ale príliš malá na to, aby to bolo riešenie problému.

Ďalšou možnosťou je JSON komprimovať napríklad pomocou gzip, táto metóda by však vyžadovala pridanie ďalších knižníc, ďalší výpočetný výkon a nakoľko táto metóda je výhodná hlavne pri opakujúcich sa dátach, bola taktiež vyhodnotená ako nevhodná. Pristúpilo k redukovaniu množstva obcí pre ktoré sa tieto dáta posielajú.

Pre uloženie dát ohraničení jednotlivých obcí, bolo potrebné vytvoriť tabuľku v následovnej podobe. Názov tabuľky bude *municipialities_boundaries* a obsahuje tri stĺpce. *Nuts_id* typu VARCHAR(10), *name* typu VARCHAR(50) a *coordinates* typu JSON, čo je synonymom pre LONGTEXT s pridaným obmedzením, ktoré skontroluje či je daná hodnota typu JSON. Primárny kľúč je *nuts_id*. Ostatné sú unikátne identifikátory.

Bolo potrebné dáta z JSON súboru presunúť do databázy, kde budú uložené a budú sa podľa potreby dotazovať. K vloženiu takého množstva dát bolo potrebné vytvoriť Python skript, ktorý by zautomatizoval tento proces. Pre istú nestabilitu pripojenia k databáze bola posúdená ako vhodnejšia možnosť, vygenerovať SQL skripty, ktoré sa následne manuálne spustia v databáze pomocou programu DBeaver. Podobne ako v predošlom prípade bol na vygenerovanie SQL vytvorený Python skript, ktorý je zobrazený nižšie.

Kvôli množstvu dát boli dáta rozdelené do skupín po 500, čo je množstvo, ktoré je schopná aplikácia DBeaver rozumne zvládnuť spustiť. Pri skupinách po 1000 už DBeaver značne "zamŕzal". Nižšie možno vidieť napísaný Python kód. Výsledný čas na spracovanie dát bol cca 15 sekúnd. Následne je možné tieto dáta jednoducho dotazovať pomocou dotazu:
SELECT * FROM *municipalities_boundaries* WHERE *nuts_id* IN ({zoznam obcí})

```
import json

obce = json.load(open('geoJson/okresy.json', encoding="utf-8"))
print("Obce loaded")
query = "INSERT INTO municipalities_boundaries(nuts_id, name, geometry)
VALUES "
```

```
def addrow(code, name, geometry):
    global query
    geometry= f'{{"type": "Polygon", "coordinates":
                                        {geometry["coordinates"]}}}'
    query += f'\n({code}, \"{name}\", \'"geometry":{geometry}\')'

i=0;
nameid=0
for obec in obce["features"]:
    i+=1
    addrow(obec["nationalCode"], obec["name"], obec["geometry"])
    if i > 500:
        query += ";"
        with open('obce_geojson' + str(nameid) +'.sql', 'w',
encoding="utf-8") as file:
            file.write(query)
        i=0;
        nameid +=1
        query = "INSERT INTO municipalities_boundaries(nuts_id, name,
geometry) VALUES "
    else:
        query += ','

query += ";"

with open('geoJson/okresy_geojson' + str(nameid) +'.sql', 'w',
           encoding="utf-8") as file:
    file.write(query)
```

---

Postup sa opakoval podobne pre okresy a následne aj kraje. Z neznámych príčin však začalo byť nemožné spustiť takto rozsiahle SQL skripty. Databáza pri pokuse o spustenie skriptu so 77 riadkami s okresmi, čo je menej ako 500 pri obciach, prerušila spojenie bez konkrétneho dôvodu. Po menšom prieskume bolo zistené, že nie je odporúčané exekuovať väčšie skripty nad databázou. Lepším prístupom sa ukázalo byť vytvoriť v Python spojenie s databázou a exekuovať tieto príkazy na vloženie dát riadok po riadku.
Výsledný Python kód možno vidieť nižšie.

---

```
import mysql.connector
import json

obce = json.load(open('geoJson/kraje.json', encoding="utf-8"))
print("Loaded")

def addrow(code, national, local, name, geometry):
    query = "INSERT INTO regions_boundaries(id, national_code, local_id,
             name, geometry) VALUES "
    geometry= f'{{"type": "Polygon", "coordinates":
                                        {geometry["coordinates"]}}}'
    query += f'\n(\"{code}\", \"{national}\", \"{local}\", \"{name}\",
                \'"geometry":{geometry}\');'
    try:
```

```
        connection = mysql.connector.connect(host='147.229.135.89',
                                             database='popelka_development'
                                             ,
                                             user='popelka',
                                             password=r"vSpE%*%X3Fe%50PAO")

        cursor = connection.cursor()
        cursor.execute(query)
        connection.commit()
        print(cursor.rowcount, "Record inserted successfully into Laptop
table")
        cursor.close()
    except mysql.connector.Error as error:
        print("Failed to insert record into Laptop table
{}".format(error))

for kraj in obce["features"]:
    addrow(kraj["id"], kraj["nationalCode"], kraj["localId"],
kraj["name"], kraj["geometry"])
```

---

Tento postup bol úspešný. Bolo však pozorované značné spomalenie nahrávania dát, ako tomu bolo pri nahrávaní obcí. Je to preto, že pre každé spojenie sa vytvorilo spojenie na novo a pridal sa len jeden riadok naraz. Bolo by možné tento Python skript zoptimalizovať, ak by bolo potrebné pridať viac dát, no ďalej to už nebolo potrebné. Keď už boli všetky dáta nahrané v databáze, bolo možné vytvoriť API point, ktorý podľa požadovaných obcí poslal dáta ich hraníc.

Ďalšou optimalizáciou bolo vytvorenie úložiska týchto dát na frontende. Tá funguje nasledovne. Ak bude potrebné vykresliť hranice obce, zavolá sa metóda na získanie hraníc, ak má trieda v pamäti tieto dáta, jednoducho ich vráti a hranice sa vykreslia. Ak ich však v pamäti nemá, odošle sa požiadavka na backend a ten tieto dáta odošle na frontend.

Toto úložisko bolo implementované ako jedináčik, anglicky singleton, čo je návrhový vzor, ktorý sa používa, keď je žiadúce mať len jednu inštanciu triedy, ktorá sa následne používa naprieč aplikáciou. Často sa takto implementuje napríklad trieda pre čas, ktorý je vo väčšine prípadov len jeden. Podobný problém riešia rôzne knižnice, napríklad redux, ale implementácia tohto riešenia nebola náročná, tak nebolo nutné pridávať knižnicu.

Pre ďalšiu optimalizáciu práce s týmito dátami sa pridalo použitie local storage, čo je API pre lokálne úložisko. Dáta sa teraz uchovávajú nie len počas chodu aplikácie, ale aj keď užívateľ aplikáciu vypne a znovu zapne.

---

```
import {doAsyncApiGetRequest} from "../../js/utilities";

/**
 * Singleton class used to store and cache geoJson data.
 * Use {@link getBoundsForLandSubdivision} to get geometry data for Nuts
code (ZUJ).
 */
```

```
class GeoJsonMemory {
    constructor() {
        const bounds = localStorage.getItem('geoJsonData');
        if (bounds === null)
            this.bounds = {};
        else
            this.bounds = JSON.parse(bounds)
    }

    /**
     * @param subdivisionId - nuts id for municipality (obec) or national
code for districts (okres) and regions (kraj)
     */
    async getBoundsForLandSubdivision(subdivisionId) {
        if (!this.bounds.hasOwnProperty(subdivisionId))
            await doAsyncApiGetRequest("getboundariesfornuts",
                                       `nuts=${subdivisionId}`)
                .then(response =>
                    this.bounds[subdivisionId] =
response.data['geometry'])
                .then(() => localStorage.setItem('geoJsonData',
JSON.stringify(this.bounds)))
        if (!this.bounds.hasOwnProperty(subdivisionId))
            console.log("Error fetching boundaries for " + subdivisionId)
        return this.bounds[subdivisionId];
    }
}

export const geoJsonMemory = new GeoJsonMemory();
```

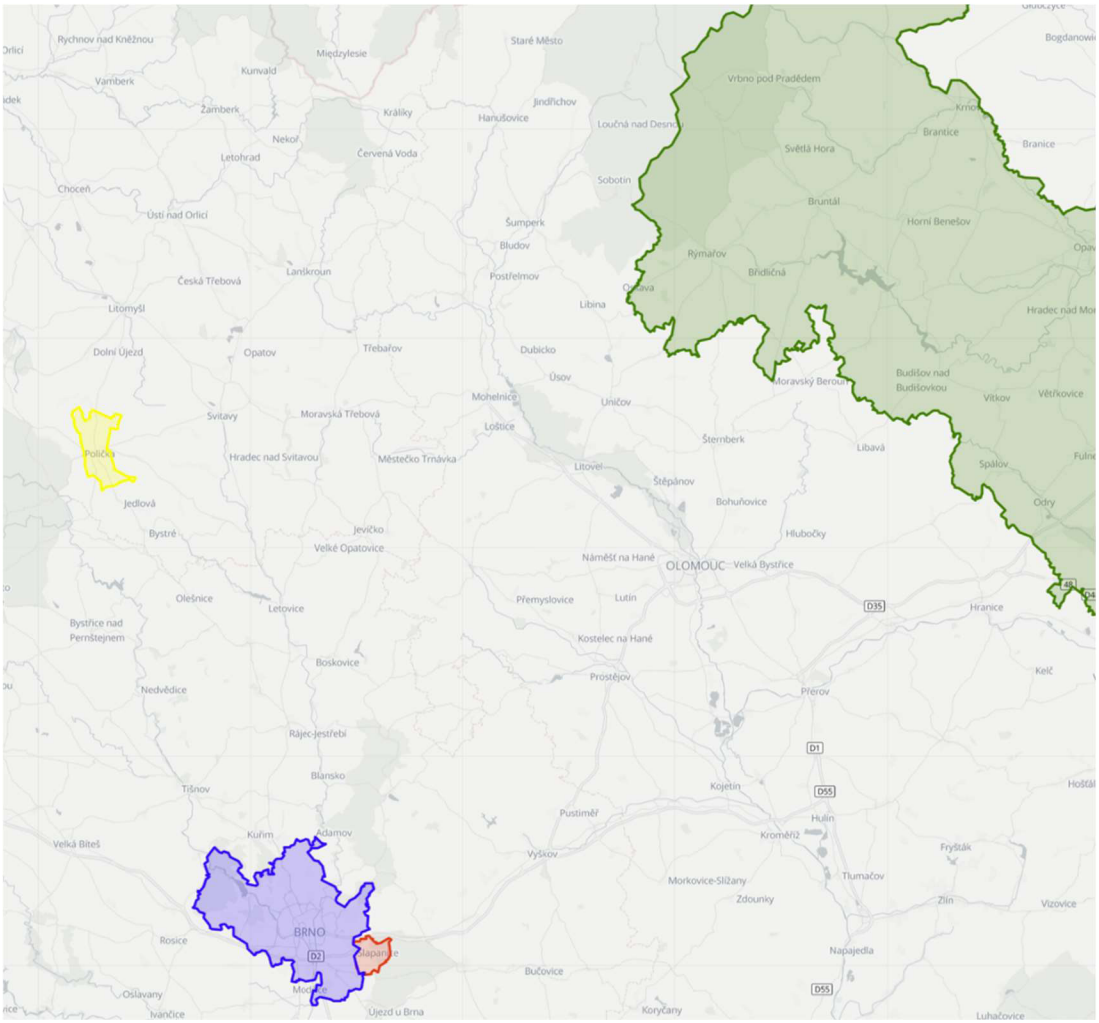
Táto implementácia sa následne využila pri tvorbe komponenty, ktorá tieto dáta vykreslí. Vo vnútri komponenty sa využije GeoJson komponenta z React Leaflet knižnice. Pridá sa však potrebná funkcionalita pre použitie v tejto aplikácií, čím je získanie dát z implementovanej pamäte na základe id obce, okresu alebo kraja. Tento prístup zredukuje potrebnú prácu budúceho užívateľa tejto komponenty na minimum. Nižšie možno vidieť kód tejto komponenty a tiež výslednú grafickú podobu v aplikácií (Obr. 20).

```
import React, {useState} from "react";
import {geoJsonMemory} from "../geoJsonMemory";
import {GeoJSON} from "react-leaflet";

/**
 * @param color of displayed region
 * @param subdivisionId - nuts id for municipality (obec) or national code
for districts (okres) and regions (kraj)
 */
export default function CountrySubdivisionBoundsOverlay({color = "pink",
subdivisionId}) {
    const [bounds, setBounds] = useState(null);
    geoJsonMemory.getBoundsForLandSubdivision(subdivisionId)
                 .then(value => setBounds(value))
```

```
return bounds && <GeoJSON style={{color: color}}
                          data={
                              {
                                  "type": "FeatureCollection",
                                  "features": [
                                      {
                                          "type": "Feature",
                                          "properties": {},
                                          "geometry": bounds
                                      }
                                  ]

                              }}/>
```

Komponenta sa použije nasledovne:

```
<CountrySubdivisionBoundsOverlay color={"yellow"}
                                 subdivisionId={578576}/>
```




Obr. 20 Vykreslenie ohraničení územných celkov v mape

Týmto sa končí práca na module Cireto v tejto diplomovej práci. Nasledujúca kapitola je cielená na zvýšenie kontroly projektu a zjednodušenie nasadenia aplikácie.

# 6 GITLAB CI/CD A DOCKER

Táto časť sa bude venovať zavedeniu zmien, zameraných na zjednodušenie údržby a spustenia aplikácie. Táto časť v práce sa v praxi označuje ako DevOps, čo je metodika, ktorá vytvára spojenie medzi vývojármi a IT odborníkmi. Obvykle to zahŕňa konfigurácie, vytváranie nástrojov či skriptov, automatizáciu procesov a ďalšie.

Všetky tu vykonané zmeny sa budú vykonávať pre aktuálnu verziu aplikácie, t.j. pred pridaním zmien z predošlej kapitoly.

## 6.1 Docker

Pre potreby zjednodušenia nasadenia aplikácie, či už na serveri alebo lokálne, bolo potrebné nakonfigurovať Docker. Pre úplné zjednodušenie sa nakonfiguroval aj Docker compose, čo je orchestrácia nad samotnými docker konfiguráciami.

Pre každú aplikáciu je spravidla vždy jeden súbor s konfiguráciou, Dockerfile. Ten obsahuje informácie ako má Docker aplikáciu spustiť, v ako kontajneri, prípadne, s akými premennými. Pri väčšine aplikácií je ale potrebné spustiť viacero aplikácií, backend, frontend alebo databáza. Prípadne, ak by išlo o ešte viac členenú architektúru, napríklad microservice architektúra, môže sa jednať o desiatky malých aplikácií. Čím vzniká problém časovej náročnosti manuálneho spustenia všetkých týchto aplikácií.

Tento problém rieši docker compose, predtým docker-compose, ale dnes je to súčasť docker. Pre docker compose sa vytvorí konfiguračný súbor, kde sa nadefinujú všetky aplikácie, ktoré majú byť spustené, prípadne s ďalšími parametrami.

Pre aktuálny stav aplikácie sú potrebné dve konfigurácie. Jedna je pre Flask aplikáciu a druhá pre redis. Nakoľko redis nie je potrebné viac konfigurovať, postačí obraz (image) kontajneru s redis a zvyšná konfigurácia sa vykoná cez docker compose.

Dockerfile pre Flask aplikáciu vyzerá nasledovne:

```
# syntax=docker/dockerfile:1

FROM Python:3.10 as base

WORKDIR /Python-docker
ADD ./ ./

COPY requirements.txt requirements.txt
RUN pip install -r requirements.txt
#--------production--------------
FROM base as prod

CMD flask run -h 0.0.0.0 -p 5000

#---------debug-----------------
FROM base as debug

RUN pip install debugpy
```

```
CMD Python -m debugpy --listen 0.0.0.0:5678 -m flask run -h 0.0.0.0 -p5000
```

*FROM* určuje, z akého obrazu (image) aplikácia vychádza. Tie je možné nájsť na stránkach docker hub.
*WORKDIR* je pracovná zložka na disku kontajneru.
*ADD* pridá súbory z aktuálnej zložky na disku Pc na disk kontajneru.
*RUN* spustí príkaz, v tomto prípade *pip install* nainštaluje Python závislosti.
Tu sa konfigurácie rozdeľujú na produkčnú a debug konfiguráciu. Ktorá sa použije, sa určí v docker compose neskôr.
Obe ale spustia Flask aplikáciu. S tým rozdielom, že pri debug sa nainštaluje debugpy, ktoré slúži na debugovanie a spustí sa zároveň s Flask aplikáciou.

Teraz je potrebné nastaviť docker compose, ktorý sa konfiguruje v súbore *docker-compose.yaml*, ktorý vidieť nižšie.

```yaml
version: '3.6'

services:
  app:
    build:
      context: .
      target: debug
    environment:
      - REDIS_HOST=redis
    ports:
      - 5000:5000
      - 5678:5678
    healthcheck:
       test: [ "CMD", "curl", "-f", "http://localhost:5000" ]
       interval: 30s
       timeout: 10s
       retries: 5
  redis:
    image: redis:latest
    command: ["redis-server", "--bind", "redis", "--port", "6379"]
    ports:
      - 6379:6379
```

*Version* udáva verziu docker compose a zároveň aj docker engine. V tomto prípade verzia 3.6 je kompatibilná s docker engine verzie 18.02.0+.
*Services* označuje aplikácie, ktoré sa spustia, v tomto prípade sú to app, čo je Flask aplikácia a redis. App aj redis sú zvolené voľne, tento názov budú mať aj po spustení v docker zozname kontajnerov, je ich nutné zvoliť vhodne.
*Context* označuje pracovnú zložku, kde sa nachádza aj Dockerfile.
*Target* označuje konfiguráciu v Dockerfile, tu je to debug, ktorú vidieť v Dockerfile vyššie.
*Ports* označuje porty, ktoré budú prístupné aj mimo kontajner, anglicky sa označujú published. Je tu definované aj ich presmerovanie, tu je to bez presmerovania, teda z 5000

na 5000. Bolo by možné napísať len 5000, ale toto je predpripravené na presmerovanie, ktoré bude nutné na produkcii. V tom prípade to bude 80:5000, čiže 80 port bude dostupný na server a 5000 je port v kontajneri. Port 5678 je dostupný pre debug, ten musí byť odstránený na produkcií.
*Command* v časti redis spustí daný príkaz, ten naviaže redis na port 6379, bez tohto príkazu sa nedalo pripojiť na redis.

Je nutné spomenúť príkaz *environment*, ktorý nastavuje systémové premenné. Táto konkrétna premenná je názov redis kontajneru, čo je aktuálne redis. Bez tejto premennej flask aplikácia vezme východiskovú hodnotu 127.0.0.1, čiže localhost. Pri spustení priamo na Pc, mimo docker je toto správne, v kontajneri však localhost odkazuje na daný kontajner, čo je nesprávne, keďže pri tejto konfigurácií sa redis nachádza v inom kontajneri.

Po tejto konfigurácií sa spustenie aplikácie značne zjednodušilo. Je potrebné spustiť len dva, respektíve tri príkazy.
Voliteľný je *docker compose stop,* tento príkaz zastaví kontajneri, nie je potrebný, ak je to prvé spustenie.

Následne stačí spustiť *docker compose build,* pre zostavenie kontajnerov. A príkazom *docker compose up -d* sa spustí celá konfigurácia, *-d* slúži na spustenie bez trvalého pripojenia konzole, anglicky detached.

Pre úplné zjednodušenie bol z týchto príkazov vytvorený bash script *deploy.sh*, ktorý vykoná všetky tri príkazy. Tento script sa následne využije pre GitLab CI/CD, ale ak by sa nastavenie CI/CD ukázalo byť problematické, aj manuálne spustenie je týmto značne zjednodušené.

Čo však zatiaľ táto Docker konfigurácia neposkytuje, je tzv. „hot-swap“. Ide o výmenu kódu za chodu aplikácie. Je to veľmi užitočné pri vývoji a debugovaní. V budúcnosti by bolo vhodné to pridať, ak sa má Docker používať aj pri vývoji v plnom rozsahu. Aktuálne sa to však nepodarilo nakonfigurovať. Nie je to však zásadný problém, pretože pri spustení bez Docker má Python túto vlastnosť, ako interpretovaný jazyk. Výhoda oproti tomu by bola, že v Docker pracuje aplikácia rýchlejšie než vo Windows.

## 6.2 GitLab CI/CD

Ako posledná časť tejto práce bolo potrebné nastaviť a otestovať automatické nasadenie aplikácie. Nakoľko sa ako verzovací systém používa GitLab, je vhodné využiť jednu z jeho funkcionalít, ktorou je GitLab CI/CD. Vysvetlenie tejto funkcie možno nájsť na začiatku práce v sekcií 2.3.1.

Pre aktuálne potreby projektu je však sústredenie na automatické nasadenie a spustenie, nie na automatické testovanie zmien.

Pre umožnenie využívania GitLab CI/CD je potrebné mať rozhranie, kde bude môcť CI/CD pracovať, nazýva sa to *runner*, respektíve *gitlab-runner*. Možnosti sú dve.

Prvou je využiť GitLabom poskytnutých runnerov, ktorý sú do 400 minút mesačne zdarma, je však nutné aby každý vývojár, ktorý ho použije mal overený účet pomocou platobnej karty.

Druhá možnosť je nastaviť a spustiť vlastnú inštanciu GitLab *runnera*. Je preto potrebné nainštalovať *gitlab-runner* a zaregistrovať ho v GitLab projekte. Tento proces trval cca 30 minút a bol otestovaný vo VM vo Windows a taktiež aj v Docker. Na server sa bude inštalovať neskôr, vzhľadom na potrebu fyzického prístup k serveru. Bolo by to možné uskutočniť aj cez SSH, to však aktuálne nie je k dispozícií.

Po nastavení a registrovaní gitlab-runneru je možné nakonfigurovať CI/CD. To sa nakonfiguruje v súbore gitlab-ci.yml. YML je rovnaký formát ako YAML, ktorý bol konfigurovaný pri Docker compose.

Ten už je v repozitári so základnou konfiguráciou. Ostáva tam pridať spustenie skriptu, ktorý sa vytvoril vyššie pre Docker compose. Tento skript sa musí spustiť cez SSH, ktoré je potrebné nakofigurovať. Nie je možné ho spustiť priamo na zariadení, pretože aj keď by bol gitlab-runner na rovnakom serveri, pravdepodobne by bol vo VM alebo Docker. Je síce možnosť mať runner ako SHELL executor, ale v tom prípade nie je možné využívať iné funkcionality, ako napríklad testovanie.

Nasadenie GitLab CI/CD vyžaduje pozastavenie aplikácie na serveri, preto tieto zmeny neboli otestované na produkčnom serveri. Vykoná sa to však v blízkej budúcnosti.

## 6.3 GitLab

Nastavenie automatizovaného nasadenia aplikácie v predošlej kapitole vytvorilo nové riziko. Všetky zmeny vo vetve *production* sa automaticky prejavia na produkčnom serveri. Je teda možné, aby ktorýkoľvek vývojár poškodil túto aplikáciu. Pri malých projektoch sa tento fakt zvykne zanedbávať, no pri raste aplikácie je potrebné zabezpečiť, aby takýto prístup mali len osoby zodpovedné za projekt či osoby dostatočne skúsené a dôveryhodné.

Dobré pravidlo je udeľovať oprávnenia len v minimálnej nutnej miere. Tým sa zamedzí riziko ohrozenia chodu aplikácie, ktoré by mohlo znamenať značné škody či už finančné alebo poškodenie reputácie inštitúcie.

Odporúčané riešenie je zamedziť priamej úprave kódu v produkčnej vetve. Kód by sa tam mohol meniť len skrz merge request, čo je v GitLab žiadosť o pridanie kódu a zároveň aj revízia kódu. Toto nastavenie je však dostupné len pre platenú verziu GitLab.

Dostupné riešenie je zmeniť role všetkým aktuálnym členom z „maintainer“ na „developer“ a vetvu upraviť na chránenú (protected branch) s povolenou zmenou len vybranými pracovníkmi, ktorým sa rola ponechá na úrovni „maintainer“. Role v GitLab fungujú hierarchicky. A to nasledovne, od najmenej oprávnení po najviac: guest, reporter, developer, maintainer a owner.

K týmto rolám náležia aj iné oprávnenia, tie však aktuálne nie sú dôležité. Owner je aktuálne len jeden a je to vlastník celého repozitáru, nakoľko projekt spadá pod jeho účet. Owner má absolútnu kontrolu nad celým projektom, je vhodné to takto ponechať a zamerať sa len na rozlíšenie na role maintainer a developer.

# 7 ZHODNOTENIE A DISKUSIA

Túto prácu je možné rozdeliť na tri časti. Prvou bolo pridanie NPM, create-react-app frameworku, pridanie JSX, teda šlo o zmeny zamerané na spríjemnenie práce vývojára. Tohto sa dosiahlo a následný vývoj modulu Cireto bol značne rýchlejší než za predchádzajúceho stavu.

Čo by sa však v tejto oblasti mohlo ešte zlepšiť je napríklad prechod z JavaScript na TypeScript, čím by sa zaistilo statické typovanie v čase transpilácie, čo by zlepšilo stabilitu aplikácie a obmedzilo vznik bugov.

Ďalším návrhom, ktorý sa však nepodarilo zapracovať, je štandardizovanie HTTP požiadaviek a odpovedí API. Metódy GET, POST, PATCH a pod. majú každá dané použitie. Aj keď to nie je vždy jednoznačné, určite by bolo možné zlepšiť ich použitie v aplikácii. Odpovede API sú tiež problematické, API pointy vždy vracajú kód 200 OK, no reálny kód sa nachádza až v tele odpovede. Toto je tiež neštandardné. Oba problémy by vyriešilo nasledovanie REST API štandardu.

Ako bolo v analýze stavu spomenuté, bolo by možné tiež zmeniť autorizáciu z aktuálne používaného hashovaného tokenu na JWT, čím by sa odstránila potreba udržiavať v pamäti tieto tokeny a získala by sa väčšia flexibilita. V aktuálnom stave aplikácie má však táto zmena malý prínos, preto pravdepodobne nebude vykonaná.

Ďalšia časť práce bola návrh a implementácia modulu Cireto. Vzhľadom na rozsah tohto modulu bolo od začiatku práce jasné že nebude dokončený v rámci tejto jednej záverečnej práce. Podarilo sa však vytvoriť základ, na ktorom bude možné ďalej tvoriť tento modul.

Komponenty ktoré boli vytvorené sú: kruhový graf v mape, zoznam s vyhľadávaním, graf vykreslený nad mapou, vykreslenie ohraničení územia v mape, zoznam odpadov so stromovou štruktúrou. Pri tvorbe týchto komponent bol cieľ aby boli znovu ľahko použiteľné. To sa podarilo hlavne v prípade kruhového grafu, vykreslení hraníc územia a zoznamu s vyhľadávaním. Zoznam odpadov nebolo možné vytvoriť dostatočne znovu použiteľný, vzhľadom na špecifickosť tohto zobrazenia.

Všetky vyššie spomínané komponenty boli následne použité v prvotných návrhoch zobrazenia v module. Je však vhodné spomenúť, že sa vytvorené zobrazenie bude meniť, rovnako ako sa s vývojom aplikácie menia aj požiadavky užívateľov.

Na záver bola vytvorená konfigurácia pre Docker a Docker compose, ktorá by mala zjednodušiť hlavne údržbu aplikácie zo strany práce so serverom. K tomu sa pridalo nastavenie GitLab CI/CD ktoré by malo v budúcnosti umožniť úplné automatizovanie aktualizácie verzií aplikácie.

K tomu boli pridané potrebné zmeny v repozitári projektu na zaistenie bezpečnosti projektu.

Ak bude projekt však naďalej rásť, je nutné zvážiť či by nebolo vhodné sprísniť akým spôsobom sa kód pridáva. Ako bolo spomínané, na to je potrebná prémium verzia GitLab.

V začiatkoch práce bolo zvažované použitie Kubernetes, ten by mohol pomôcť vyriešiť problémy so škálovaním, túto možnosť by však bolo potrebné viac preskúmať.

Je tiež vhodné zvážiť prechod na niektorú z cloudových služieb, teda poskytovateľov serverov. Prípadne zabezpečiť lepší hardware vzhľadom na náročnosť optimalizačných výpočtov, ktoré v aplikácií prebiehajú. S nárastom množstva užívateľov je pravdepodobné, že výpočetný výkon nebude postačovať.

Ďalším možným riešením problémov s výkonom, okrem zvýšenia dostupnej výpočtovej výkonnosti, by bola optimalizácia Python kódu.

Optimalizácia optimalizačných algoritmov by však mohla byť náročná, ak nie úplne nemožná. Je však možné pokúsiť sa zrýchliť jazyk, ako taký. Jedným možným riešením by bolo použitie kompilácie Python kódu. Na toto sa používa napríklad knižnica Numba, ktorá pri prvom zavolaní metódy kompiluje Python kód.

Použitie tejto knižnice je však často náročné, množstvo Python knižníc nie je podporovaných v týchto kompilovaných triedach a tým pádom sú obmedzené možnosti, kde využiť kompiláciu.

Výrazne náročnejšie a radikálnejšie riešenie by bolo prejsť na iný jazyk. Python je veľmi obľúbený kvôli svojej jednoduchej syntaxi a dynamickému tipovaniu. Toto však nie je zadarmo a Python preto nepatrí medzi najrýchlejšie jazyky. Možno hovoriť preto o C alebo C++, avšak písať komplexné optimalizačné úlohy v týchto jazykoch by mohlo byť náročnejšie.

Z pohľadu webového vývoja sa ponúkajú jazyky používané v najväčších webových aplikáciách, tými sú Java a C#. Tie sú rýchlejšie než Python a keďže sa často používajú pre webové aplikácie, sú k ním k dispozícií veľmi kvalitné frameworky pre webový vývoj.

Pri zhodnotení realít tohto projektu sú však aj tieto jazyky nevhodné na využitie v plnom rozsahu aplikácie. Mohli by však slúžiť pre API, autorizáciu, všetko súvisiace s webovou časťou aplikácie. Nutné však poznamenať, že skúsení vývojári pre Java a C# sú menej dostupní a to by taktiež znamenalo finančné zaťaženie projektu.

Na tomto projekte spolupracujú študenti a akademický pracovníci. Je možné predpokladať, že im známe jazyky budú tie, ktoré sa používajú na výpočty či prácu s dátami. To sú napríklad Matlab, Octave, Python, R či Julia.

Z týchto jazykov sa javí Julia ako vhodný kandidát. Je to JIT (just-in-time, práve v čas) kompilovaný jazyk, ktorý bol vyvinutý so zameraním na výkonnosť a je tiež často využívaný pre vedecké účely. Tu však nie je možné vyvodiť jasný záver a pri takto zásadnej zmene v projekte by bolo vhodné prediskutovať ju s akademickým vedením projektu.

# 8 ZÁVER

Hlavným cieľom tejto práce bolo navrhnúť a implementovať riešenie pre nový modul v systéme zaoberajúcim sa odpadovým hospodárstvom. Bolo zrejmé, že nebude možné vytvoriť celý modul v krátkom časovom úseku tejto práce.

Podarilo sa však vytvoriť základnú funkcionalitu, ktorá poskytuje možnosť zadať vstupné parametre pre optimalizáciu a následne výsledky optimalizácie zobraziť na mape. Spôsob akým boli komponenty vytvorené by mal zabezpečiť znovupoužiteľnosť pri ďalšom vývoji. Nakoľko jedinou konštantou vo vývoji je zmena, je toto zásadný parameter pre budúcnosť projektu.

Uskutočnená implementácia mala presah od frontendu, teda užívateľského rozhrania, cez tvorbu API pointov v Python Flask aplikácií, až po dotazovanie a pridávanie z a do databázy. Pre pridávanie dát boli tiež použité Python skripty vzhľadom na objem spracúvaných dát.

Ďalej sa podarilo modernizovať projekt, pridaním JSX, create-react-app frameworku a NPM. Týmto sa zlepšil a zjednodušil vývoj aplikácie, čím sa umožnilo lepšie a rýchlejšie dodávať nové funkcionality do aplikácie.

Na koniec práce bol pridaný Docker spolu s Docker compose, čo zjednodušilo spustenie aplikácie či už pri vývoji alebo na produkčnom serveri.

K tomu bola pripravená konfigurácia GitLab CI/CD, ktorá plne zautomatizuje nasadzovanie aplikácie na produkčný server pri každej aktualizácií.

Práca na tomto module a spolupráca s vedením projektu, bola prínosná a zaujímavá. Nešlo len o vývoj aplikácie ako taký, ale aj o porozumenie problematike odpadového hospodárstva.

Vývoj modulu bude ďalej pokračovať, hlavným bodom nasledujúceho vývoja bude spojenie vytvoreného rozhrania s optimalizačnými modulmi riešenými v inej diplomovej práci. Následne by sa mohlo pracovať na zlepšení prispôsobiteľnosti, pridať filtrovanie či ukladanie často používaných konfigurácií. S rastom počtu užívateľov v budúcnosti bude tiež potreba riešiť škálovanie aplikácie.

# ZOZNAM POUŽITEJ LITERATÚRY


[1] Zákon č. 541/2020 Sb., Zákon o odpadech. Účinnost od 1. 1. 2021. [cit. 9. Máj 2023], *Zákon č. 541/2020 Sb., Zákon o odpadech. Účinnost od 1. 1. 2021. [cit. 9. 5. 2023] Dostupné z: https://www.zakonyprolidi.cz/cs/2020-541.*

[2] *Směrnice Evropského parlamentu a Rady (EU) 2018/851 ze dne 30. května 2018, kterou se mění směrnice 2008/98/ES o odpadech (Text s významem pro EHP).*

[3] *Směrnice Evropského parlamentu a Rady (EU) 2018/850 ze dne 30. května 2018, kterou se mění směrnice 1999/31/ES o skládkách odpadů (Text s významem pro EHP).*

[4] K. J. Š. R. N. V. P. P. Hrabec D., „Circular economy implementation in waste management network design problem: a case study (2020),“ *Central European Journal of Operations Research, 28 (4), pp. 1441 – 1458, DOI: 10.1007/s10100-019-00626-z,* 5. Jún 2019.

[5] Meta Open Source, „React,“ [Online]. Dostupné z: https://react.dev. [Cit. 18. Marec 2023].

[6] „OpenStreetMap,“ [Online]. Dostupné z: https://www.openstreetmap.org/about. [Cit. 18. Marec 2023].

[7] „Leaflet,“ [Online]. Dostupné z: https://leafletjs.com [Cit. 18. Marec 2023].

[8] PalletsProject, „Flask,“ [Online]. Dostupné z: https://flask.palletsprojects.com. [Cit. 10. Apríl 2023].

[9] WSGI, [Online]. Dostupné z: wsgi.readthedocs.io/en/latest/what.html. [Cit. 10. Apríl 2023].

[10] Gitlab, „Gitlab - Docs,“ [Online]. Dostupné z: https://docs.gitlab.com/ee/ci/introduction/index.html#continuous-integration. [Cit. 10. Máj 2023].

[11] Wikipedia, [Online]. Dostupné z: en.wikipedia.org/wiki/Docker_(software). [Cit. 10. Apríl 2023].

[12] Docker, [Online]. Dostupné z: docker.com/resources/what-container/. [Cit. 10 Apríl 2023].

[13] Podman, „Podman,“ [Online]. Dostupné z: podman.io. [Cit. 22. Máj 2023].

[14] The Linux Foundation, „Open Containers Initiative,“ [Online]. Dostupné z: https://opencontainers.org/. [Cit. 22. Máj 2023].

[15] Mozzila, [Online]. Dostupné z: developer.mozilla.org/en-US/docs/Web/HTTP/CORS. [Cit. 10. Apríl 2023].

[16] W3 School, [Online]. Dostupné z: w3schools.com/tags/ref_httpmethods.asp. [Cit. 10. April 2023].

[17] „JWT,“ [Online]. Dostupné z: jwt.io. [Cit. 10 Apríl 2023].

[18] „Create React App,“ [Online].
Dostupné z: create-react-app.dev/docs/folder-structure/. [Cit. 10. April 2023].

[19] A. Morel, „Stack overflow,“ 3 Máj 2022. [Online]. Dostupné z: https://stackoverflow.com/a/50164499. [Cit. 7. Máj 2023].

# PRÍLOHY

Vzhľadom na utajenie zdrojových kódov projektu, nie je možné ich poskytnúť.