VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA ELEKTROTECHNIKY A KOMUNIKAČNÍCH TECHNOLOGIÍ

FACULTY OF ELECTRICAL ENGINEERING AND COMMUNICATION

ÚSTAV MIKROELEKTRONIKY

DEPARTMENT OF MICROELECTRONICS

UNIVERZÁLNÍ DATOVÝ SKLAD PRO IOT APLIKACE

UNIVERSAL DATA STORAGE FOR IOT APPLICATIONS

DIPLOMOVÁ PRÁCE

MASTER'S THESIS

AUTOR PRÁCE — Bc. Matej Kadlíček

AUTHOR

VEDOUCÍ PRÁCE — doc. Ing. Pavel Šteffan, Ph.D.

SUPERVISOR

BRNO 2022

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
FAKULTA ELEKTROTECHNIKY A KOMUNIKAČNÍCH TECHNOLOGIÍ

Diplomová práce

magisterský navazující studijní program **Mikroelektronika**

Ústav mikroelektroniky

Student: Bc. Matej Kadlíček ***ID:*** 203243

Ročník: 2 ***Akademický rok:*** 2021/22

NÁZEV TÉMATU:

Univerzální datový sklad pro IoT aplikace

POKYNY PRO VYPRACOVÁNÍ:

V rámci diplomové práce navrhněte a implementujte webovou aplikaci pro zobrazení naměřených dat. Vytvořte online serverový prostor s databází pro ukládání naměřených dat. Realizujte vlastní API, které bude zprostředkovávat přenos dat z databáze do koncové aplikace. Vytvořte přihlašovací systém pro identifikaci jednotlivých uživatelů aplikace. Otestujte funkčnost navrženého systému.

DOPORUČENÁ LITERATURA:

Podle pokynů vedoucího práce.

Termín zadání: 7.2.2022 ***Termín odevzdání:*** 24.5.2022

Vedoucí práce: doc. Ing. Pavel Šteffan, Ph.D.

doc. Ing. Lukáš Fujcik, Ph.D.
předseda rady studijního programu

Abstrakt

Diplomová práca sa zaoberá vytvorením webovej aplikácie dátového skladu pre testovacie potreby interných projektov. Cieľom práce bolo vybrať programátorské nástroje na tvorbu webovej stránky, urobiť návrh funkcionality, vzhľadu, dátového úložiska a následne návrh implementovať.

Práca je rozdelená na 5 kapitol. Prvá je venovaná výberom programátorských nástrojov na tvorbu serverovej časti. Druhá kapitola je venovaná návrhu databázového úložiska. V tretej kapitola sa nachádza návrh požiadaviek na funkcionalitu a návrh používateľského rozhrania. Vo štvrtej kapitole sa nachádza implementácia aplikácie. Piata kapitola je venovaná testovaniu.

Kľúčové slová

backend, frontend, databáza, zabezpečenie, návrh, implementácia

Abstract

The master's thesis is written about the topic of data warehouse and the technology associated with it. The aim of the theses was to select programming tools for creating a website. Next aim was to design the functionality, appearence and datastorage. Subsequent implementation of design was done after. Theses was divided into 5 chapters. The first one includes selections of programming tools for a creating a backend. Second chapter includes design of database storage. Third chapter includes design of functionality and user interface. Fourth chapter includes implementation of the application and the last one includes testing.

Keywords

backend, frontend, database, security, design, implementation

Bibliografická citácia

KADLÍČEK, Matej. Univerzální datový sklad pro IoT aplikace. Brno, 2022. Dostupné také z: https://www.vutbr.cz/studenti/zav-prace/detail/139388. Diplomová práce. Vysoké učení technické v Brně, Fakulta elektrotechniky a komunikačních technologií, Ústav mikroelektroniky. Vedoucí práce Pavel Šteffan.

Prehlásenie autora o pôvodnosti diela

Meno a priezvisko študenta:	*Matej Kadlíček*
VUT ID študenta:	*203243*
Typ práce:	*Diplomová práca*
Akademický rok:	*2021/22*
Téma záverečnej práce:	*Univerzálny dátový sklad pre IoT aplikácie*

Prohlašuji, že svou závěrečnou práci jsem vypracoval samostatně pod vedením vedoucí/ho závěrečné práce a s použitím odborné literatury a dalších informačních zdrojů, které jsou všechny citovány v práci a uvedeny v seznamu literatury na konci práce.

Jako autor uvedené závěrečné práce dále prohlašuji, že v souvislosti s vytvořením této závěrečné práce jsem neporušil autorská práva třetích osob, zejména jsem nezasáhl nedovoleným způsobem do cizích autorských práv osobnostních a jsem si plně vědom následků porušení ustanovení § 11 a následujících autorského zákona č. 121/2000 Sb., včetně možných trestněprávních důsledků vyplývajících z ustanovení části druhé, hlavy VI. díl 4 Trestního zákoníku č. 40/2009 Sb.

.

V Brně dne: 3.januára.2022 ----------

podpis autora

Poďakovanie

Ďakujem vedúcemu diplomovej práce doc.Ing. Pavel Šteffan Ph.D. za podnety k zlepšeniu kvality diplomovej práce a za podporu počas semestra pri písaní diplomovej práce.

V Brně dne: 3.januára.2022 ----------

podpis autora

Obsah

ZOZNAM OBRÁZKOV

ZOZNAM TABULIEK

ZOZNAM ZDROJOVÝCH KÓDOV

ÚVOD

Veľa študentov prichádza pri tvorbe bakalárskych a diplomových prácach do styku s množstvom dát, ktoré chcú mať uložené v databáze, a takisto ich chcú graficky zobraziť. V dnešnej dobe existuje množstvo produktov či služieb, ktoré ponúkajú dané možnosti pre úložisko dát či ich zobrazenie. Tieto služby väčšinou ponúkajú obmedzené možnosti. Ak ich chceme využiť naplno je potrebné za službu zaplatiť.

Preto vznikla myšlienka vytvoriť webovú aplikáciu, ktorá bude uskladňovať namerané hodnoty, a takisto zobrazovať ich grafickú reprezentáciu. Táto aplikácia bude slúžiť na interné účely pre vedúceho práce a jeho budúcich študentov. Aplikácia bude disponovať prihlasovacím systémom, vďaka ktorému bude môcť byť regulovaný počet používateľov.

Diplomová práca opisuje výber jednotlivých technológii využitých pre potreby serverovej časti a používateľského rozhrania. Taktiež sa v nej nachádza návrh na vytvorenie databázového modelu vďaka, ktorému bude zabezpečené správne ukladanie nameraných dát.

Ďalej je v práci rozobraná implementácia celej aplikácie, v ktorej sa nachádza popis jednotlivých funkcionalít, popis rozloženia používateľského rozhrania či popis dátovej štruktúry na získanie nameraných údajov od študentov.

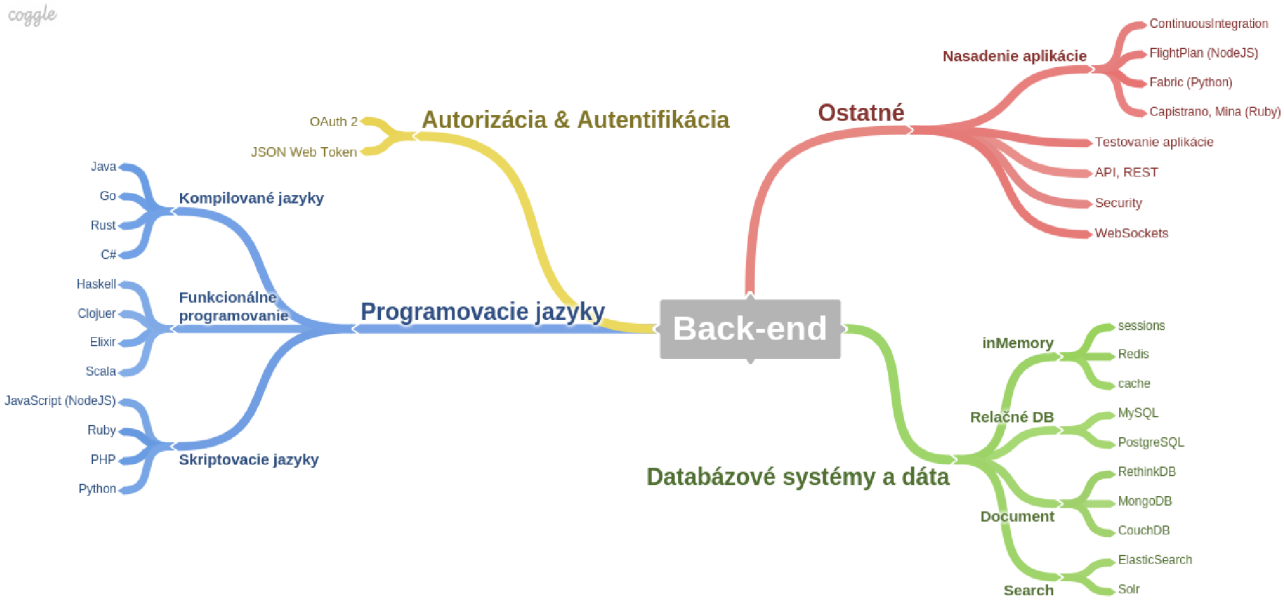
Posledná časť diplomovej práce sa venuje testovaniu základných funkcionalít webovej aplikácie. Jeho cieľom je dokázať, že navrhnuté používateľské rozhranie funguje a komunikuje spolu so serverovou časťou a takisto ukázať, že navrhnutá funkcionalita robí to čo sa od nej čakalo.

1. TEORETICKÝ ROZBOR SERVEROVEJ ČASTI APLIKÁCIE

Prvá kapitola sa zaoberá výberom nástrojov na tvorbu serverovej logiky aplikácie. Výber nástrojov spočíva vo výbere programovacieho jazyka, databázového systému a knižníc. Jedná sa o knižnice zaoberajúce sa vývojom webových aplikácií a komunikáciou programovacieho jazyka s vybranou databázou.

1.1 Čo je to backend?

Ako backend sa označuje časť webovej aplikácie, ktorá slúži k administrácii webu a k spracovaniu dát. Táto časť úzko spolupracuje so serverom a databázou. Na obrázku 1.1 je vidieť schéma najpoužívanejších technológií pre backend: [1]



Obrázok 1.1: Príkladová štruktúra back-endového systému [1]

1.2 Python a jeho knižnica Flask

Python je dynamický interpretovaný programovací jazyk. To znamená, že sa kód prekladá až za behu. Taktiež je to objektovo orientovaný jazyk, čo znamená, že všetky dátové štruktúry sa považujú za objekty, ktoré majú svoje vlastnosti, metódy.

Jedná sa o rozšírený programovací jazyk, ktorý má mnohé využitia, jedným z nich je tvorba webových aplikácií. [2,3]

Na tvorbu webových aplikácií je možné použiť knižnicu nazývanu Flask. Ide o tzv. microframework, ktorý je navrhnutý tak, aby jadro aplikácie ostalo jednoduché.

Miesto toho, aby ponúkal abstrakčnú vrstvu pre podporu práce s databázami, len podporuje rozšírenia na pridanie týchto schopností do webovej aplikácie.

Flask je založený na súprave nástrojov Werkzeg WSGI a šablónovom engine Jinja2. [4]

1.2.1 WSGI

WSGI je skratka pre Web Server Gateway Interface, čo znamená rozhranie pre bránu webového servera. Jedná sa o špecifikáciu rozhrania pomocou ktorého server a aplikácia komunikujú. Je používané ako štandard pre vývoj webových aplikácií v jazyku Python. [5]

1.2.2 Werkzeug

Jedná sa o súpravu nástrojov WSGI, ktorá implementuje požiadavky, objekty odpovedí a pomocné funkcie. To umožňuje postaviť na ňom webový framework. Flask používa Werkzeug ako jeden zo svojich základných pilierov.

1.2.3 Jinja2

Jinja2 je rýchly, výrazný a populárny šablónový nástroj určený pre Python. Systém webových šablón kombinuje šablónu so špecifickým zdrojom údajov na vykreslenie dynamickej webovej stránky.

1.3 Knižnica SQLAlchemy

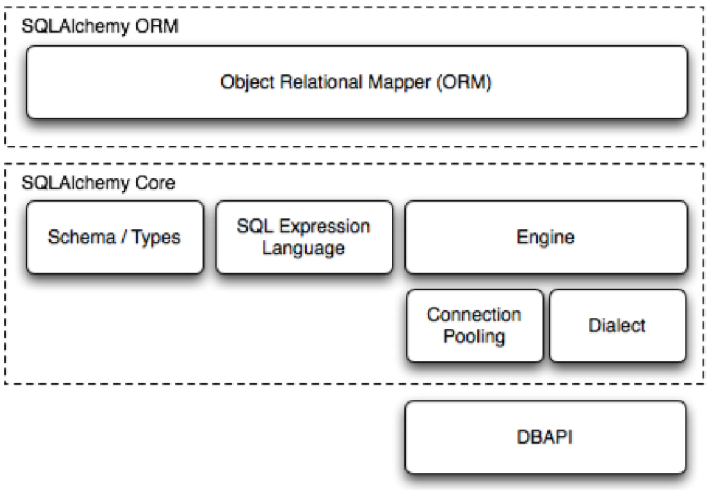
SQLAlchemy je relačný mapovač objektov (ORM). Jedná sa o knižnicu využívanú jazykom Python, ktorá pracuje s dátami uloženými v SQL databáze ako s objektami jazyka Python. Vďaka tomu je možné pomocou kódu písaného v Pythone vytvárať, vyčítavať, aktualizovať či vymazávať dáta miesto písania priameho SQL kódu. To pomáha zrýchliť čas vývoja a zmenšovať náchylnosť na chyby pri manažovaní dát. Poskytuje celú sadu nástrojov navrhnutých pre efektívny a vysokovýkonný prístup k databáze, ktoré sú upravené do programovacieho jazyka Python.

Táto knižnica pozostáva z dvoch primárnych častí, a to sú Core a ORM. Core je samo o sebe plne funkčnou súpravou SQL nástrojov, ktoré poskytuje hladkú vrstvu abstrakcie cez širokú škálu implementácií databázových aplikačných programových rozhraní určených pre Python (DBAPI), ako aj využitím jazyka výrazov SQL, ktorý umožňuje písanie výrazov SQL jazyka pomocou štruktúr jazyka Python.

SQLAlchemy reprezentuje schému databáze dvomi spôsobmi. Buď využitím popisovacieho jazyka dát (DDL),využívajúceho syntaxe SQL jazyka, ktorý sa používa na samotné vytvorenie databázovej schémy, alebo preskúmaním existujúcich schém pomocou objektov jazyka Python mapujúcich databázovú štruktúru. [6]

ORM je voliteľné nadstavba, ktorá je postavený nad jadrom.

Táto nadstavba poskytuje prostriedky na asociáciu používateľom nadefinovaných tried jazyka Python s databázovými tabuľkami. Toto zahŕňa systém, ktorý synchronizuje všetky zmeny stavov medzi objektami a ich príslušným riadkom. Jedná sa teda o vyššiu abstrakciu používania SQLAlchemy, čo ju radí medzi užívateľmi ako široko využívanú a preto bude ďalej bližšie popísaná. Rozdelenie SQL je zobrazené na obrázku 1.2 :



Obrázok 1.2: Vnútorné rozdelenie knižnice SQLAlchemy

1.3.1 Vytvorenie pripojenia

Východiskovým bodom je realizácia spojenia medzi databázou a našou backendovou časťou aplikácie. Spojenie sa realizuje vytvorením inštancie enginu (ďalej ako motor) funkciou **create_engine()**. Táto funkcia obsahuje v argumente dátový typ string, ktorý popisuje názov používaného druhu relačnej databázy, vybraného DBAPI, prihlasovacie údaje do databázy, port/IP adresu na ktorej databáza beží a nakoniec názov schémy danej databázy. Ilustračný príklad je vidieť na zdrojovom kóde 1.1:

```
engine = create_engine("dialect+driver://user:password@host/name")
```

Zdrojový kód 1.1: Vytvorenie enginu

, kde **dialect** reprezentuje vybraný typ relačnej databázy, **driver** vybraný typ DBAPI, **user** a **password** prihlasovacie údaje, **host** reprezentuje port/IP adresu a **name** názov danej schémy.

- **DBAPI**

Je skratkou pre špecifikáciu aplikačného programového rozhrania určeného pre databázy a Python. Jedná sa o široko využívané špecifikácie v Pythone na definovanie bežných vzorov používania pre všetky balíky pripojenia k databáze. DBAPI je nízkoúrovňové API používané aplikáciou vytvorenou v Pythone na komunikáciu s databázou.

1.3.2 Metadáta

Metadátami sa nazývajú triedy jazyka Python, ktoré reprezentujú prvky reálnej databázovej schémy ako tabuľky či stĺpce. Pri používaní ORM je proces, ktorým deklarujeme metadáta zvyčajne kombinovaný s procesom deklarovania mapovania tried.

Proces mapovania tried asociuje mapovanú triedu so schémou v databáze. V tomto procese musia triedy prejsť cez funkciu **mapper().** Tento proces asociuje mapovanú triedu s databázovou tabuľkou. Mapovaná trieda teda obsahuje atribúty-metadáta, ktoré budú spojené so stĺpcami v reálnej databáze.

Funkcia **mapper()** je vyvolaná použitím objektu nazývaného **registry**(ďalej Register). Register slúži ako základ pre udržiavanie kolekcie mapovaných objektov a poskytuje niekoľko konfiguračných postupov.

Existujú 3 základné konfigurácie mapovania a to deklaratívne mapovanie pomocou základne, deklaratívne mapovanie pomocou dekorátora a imperatívne(klasické) mapovanie.

- **Deklaratívne mapovanie pomocou základne:**

Jedná sa o najtypickejšie mapovanie v modernom SQLAlchemy. V tomto druhu mapovania je využívaná funkcia **declarative_base()**, pomocou ktorej vytvoríme triedu Base(ďalej len Základňa). Základňa bude pridelená do triedy s metadátami, ktoré budú mapovať reálnu tabuľku z databáze. Názorná ukážka vytvorenia Základne a jej priradenia do metatriedy je zobrazená v zdrojovom kóde 1.2 :

```
Base= declarative_base()


class User(Base):
    __tablename__= "user_table"
    id = Column(Integer, primarykey=True)
    name =Column(String(20))
    fullname= Column(String(30))
```

Zdrojový kód 1.2: Vytvorenie základne

Funkcia **declarative_base()** je v skutočnosti skratkou pre prvotné vytvorenie konštruktora Registra **registry()** a následné vytvorenie Základne pomocou Registrovej metódy **.generate_base()**. Táto ekvivalentnosť je zobrazená na zdrojovom kóde 1.3:

```
Base = declarative_base #ekvivalent

mapper_registry = registry()
Base = mapper_registry.generate_base()
```

Zdrojový kód 1.3: Rôzne spôsoby deklarovania základne

- **Deklaratívne mapovanie pomocou dekorátora triedy**

Jedná sa o alternatívu k použitiu deklarácie pomocou Základne. Rozdiel spočíva v tom, že proces deklaratívneho mapovania je ku triede aplikovaný explicitne, a to pomocou dekorátora triedy. Ten je vytvorený pomocou metódy funkcie **registry()** nazývanej **.mapped**.

Vytvorenie deklaratívneho mapovania pomocou dekorátora triedy je zobrazené na zdrojovom kóde 1.4:

```
@mapper_registry.mapped
class User:
    __tablename__:"user_table"
```

Zdrojový kód 1.4: Vytvorenie dekorátora

Táto forma mapovania je užitočná pri kombinovaní deklarácie tried, najmä s modulom dataclass. Tento modul poskytuje dekorátor triedy **@dataclass**, ktorý slúži k automatickému generovaniu základných metód ako sú __init__() či __repr__(). Dekorátor je využívaný na zoskenovanie atribútov, ktoré danú triedu definujú. Hierarchia použitia dekorátorov je nasledovná, najskôr je použitý dekorátor @dataclass, pričom dekorátor @mapper_registry.mapped je využitý až po úplnom skonštruovaní triedy.

- **Imperatívne(klasické) mapovanie**

Je druh mapovania, ktorý využíva metódu **.map_imperatively()** od funkcie **registry().** Mapovaná trieda neobsahuje žiadne atribúty popisujúce tabuľku v databáze. Obsahuje len atribút pass, ktorý v jazyku Python značí prázdnosť danej štruktúry. Atribúty popisujúce databázovú tabuľku sa nachádzajú v triede **Table**. Táto trieda obsahuje informácie ako názov databázovej tabuľky, názvy stĺpcov či ich dátové typy. Imperatívne mapovanie je zobrazené na zdrojovom kóde 1.5:

```
mapper_registry= registry()

user_table= Table("user", mapper_registry.metadata,
    Column("id", Integer, primary_key= True),
    Column("name", String(50)),
    Column("fullname", String(50)),
    Column("nickname",String(12))
    )

class User:
    pass

mapper_registry.map_imperatively(User,user_table)
```

Zdrojový kód 1.5: Imperatívne mapovanie

Či už je využité mapovanie deklaratívnym alebo imperatívnym štýlom, objekt Register za nás vytvorí konštruktor __init__(), a to ku všetkým mapovaným triedam, ktoré explicitne nemajú priradený svoj vlastný. Vďaka tomuto bude argument novovytvoreného objektu z mapovanej triedy akceptovať všetky atribúty, ktoré sú v danej triede pomenované. Príklad vytvorenia objektu z mapovanej triedy je zobrazený v zdrojovom kóde 1.6:

```
u1=User(name="John", fullname="Doe", nickname="JD")
```

Zdrojový kód 1.6: Vytváranie objektu mapovanej triedy

,kde User značí mapovanú triedu a v argumente sú červeným zobrazené kľúče a zeleným ich hodnoty.

1.3.3 ORM Session

Session(ďalej len Relácia) je základný transakčno/databázový interaktívny objekt, ktorý sa využíva v ORM štýle. Hlavnou úlohou Relácie je nadviazať spojenie s databázou a reprezentovať akúsi prevádzaciu zónu pre všetky objekty, ktoré sú s ňou spájané počas doby jej trvania. Taktiež poskytuje rozhranie pre dotazovanie sa na databázu pomocou klúčových príkazov ako **SELECT či INSERT** a ďalších, ktoré pracujú s objektami mapovanými pomocou ORM Relácie.

- **Stavy objektov v Reláciach**

Objekty sa v Relácii môžu nachádzať v 4 stavoch. Jedná sa o stav prechodný, čakajúci, pretrvávajúci , oddelený.

-*prechodný stav*: nový objekt nemá žiadnu databázovú identitu a nebol zatiaľ priradený k žiadnej Relácii.

-*čakajúci stav*: objekt stále nemá žiadnu databázovú identitu, no bol pridaný k Relácii

-*pretrvávajúci stav*: objekt už má databázovú identitu(napr. primárny kľúč) a je momentálne priradený k Relácii. Hociktorý objekt, ktorý bol buď v čakajúcom stave a teraz je vložený do databázy, alebo je z databázy načítavaný do Relácie je v pretrvávajúcom stave.

-*oddelený stav:* objekt má databázovú identitu, no už nie je spojený s Reláciou. Objekt bol buď odstránený, alebo bola Relácia uzatvorená. Tento stav sa zvyčajne používa pri prenášaní objektov medzi jednotlivými Reláciami.

- **Vytvorenie spojenia Relácie s databázou**

Relácia potrebuje na nadviazanie spojenia s databázou zdroj pripojenia. Ako zdroj stabilizovanej transakcie na pripojenie je využitý motor, ktorý je spojený s Reláciou cez jej argument.

Vytvorenie spojenia Relácie s databázou je možné aj pomocou kontextového manažéra vytvoreného pomocou kľúčového slova with.
Výhodou je presné vymedzenie pôsobenia Relácie, čo pomáha udržiavať prehľad. Taktiež je Relácia na konci automaticky uzatvorená. Z tohto dôvodu nie je potrebné volať metódu Relácie **.close()** slúžiacu na zatvorenie Relácie. Príklad kontextového manažéra je zobrazený v zdrojovom kóde 1.7:

```
with Session.begin() as session:
    session.add(some_object)
```

Zdrojový kód 1.7: Kontextový manažér

,kde session.begin() začína transakciu, a session.add pridáva do databázy objekty.

- **Dopytovanie**

Dopytovanie sa na databázu v ORM môže byť vykonávané pomocou príkazu **select**, ktorý má v argumente názov mapovanej triedy či mapovanú triedu a názov jej stĺpca. Druhou metódou je využitie metódy query na objekt Relácie **session.query**. Principiálny rozdiel medzi týmito dvomi metódami nie je. Jediným rozdielom je podpora príkazu select v budúcnosti SQLAlchemy, pričom session.query v budúcnosti prestane byť podporované.

Oba typy dopytovania používajú podporné metódy ako:

.filter() – argument obsahuje kritérium filtrovania v podobe udania mapovanej triedy a metódy jej inštancie, ktorá reprezentuje daný stĺpec v reálnej databázovej tabuľke,

.order_by() – argument obsahuje zoraďovacie kritérium aplikované na mapovanú triedu a metódu jej inštancie,

.all() – vráti n-ticu výsledkov daných dopytovaním,

.first() – vráti prvý výsledok dopytovania,

.one() – vráti práve jeden výsledok, v prípade viacerých výsledkov či žiadneho výsledku vráti error.

- **Vytváranie objektov mapovanej triedy**

Po vytvorení mapovanej triedy je možné vytvárať objekty patriace danej vytvorenej triede. Tieto objekty sú využívané v rámci transakcie na úkony ako pridávanie, mazanie či aktualizovanie. Dané objekty obsahujú v argumente kľúče a ich hodnoty. Príklad objektu patriaceho do triedy User je zobrazený na zdrojovom kóde 1.8:

```
user1= User(name="John")
```

Zdrojový kód 1.8: Objekt mapovanej triedy

,kde name je kľúč a “John“ je jeho hodnota.

Vďaka tomu , že mapovaná trieda automaticky generuje __init__() konštruktor, je v argumente vytvoreného objektu danej triedy možné použiť ako kľúč názov mapovaných stĺpcov z databázovej tabuľky.

- **Priradenie objektu k Relácii**

Objekty je možné priradiť k Relácii dvomi spôsobmi a to pomocou metód:

.add() – slúži na priradenie jedného objektu do Relácie,

.add_all() – sa využíva v prípade potreby priradenia viacerých objektov do Relácie

Obe metódy obsahujú v argumente daný objekt na priradenie.

- **Odosielanie objektov do databázy**

Po pridaní objektov do Relácie pomocou metódy .add() alebo add_all() však ešte neprichádza k ich odoslaniu do databázy. S odosielaním objektov do databázy sú spojené dve metódy, a to **.flush()** a **.commit()**

- **flush()** slúži na uskutočnenie série operácií spojených s databázou ako je vkladanie či mazanie. Databáza ich však uchováva ako čakajúce operácie v transakcii. Zmeny sa zatiaľ neuložia natrvalo do databáze a ani sa nezobrazia pre potreby iných transakcií, až pokým nie je použitá metóda .commit()
- **commit()** táto metóda potvrdí vykonávané operácie do databázy. Objekty sú teraz viditeľné v databáze a sú taktiež viditeľné pre potreby iných transakcií.

Pri využití kontextového manažéra nie je potrebné používať metódu .flush() ani .commit(), nakoľko jeho použitím je na konci bloku transakcia automaticky potvrdená pomocou .commit().

- **Aktualizovanie či vymazanie objektu z databázy**

ORM prístup umožňuje využiť vstavané funkcie, pre aktualizovanie či mazanie objektov databázy. Pre ich využitie treba importovať z knižnice sqlalchemy funkcie **update()** a **delete()**.

Operácia update obsahuje v argumente názov mapovanej triedy, klauzulu **.where()**, ktorá bližšie špecifikuje kritérium nájdenia správneho záznamu v databáze a ako poslednú časť obsahuje metódu **.values()**, ktorá má v argumente novú hodnotu pre daný kľúč reprezentujúci stĺpec z databázovej tabulky. Operácia delete má štruktúru rovnakú ako operácia update, s rozdielom vynechania metódy .values().

Štruktúra je znázornená v zdrojovom kóde 1.9:

```
stmt= update(User).where(User.name == "some_name").values(name="some_name1")
stmt= delete(User).where(User.name == "some_name")
```

Zdrojový kód 1.9: Aktualizovanie a mazanie objektu

,kde stmt predstavuje premennú, do ktorej sa uloží výsledok operácie.

Po uložení daných operácií do premennej je táto premenná predaná do argumentu funkcie **session.execute()**. Táto funkcia vykoná konštrukciu SQL výrazu uloženého do premennej.

- **Uzatvorenie transakcie**

Zatvorenie transakcie medzi databázou a backendom je vykonávané pomocou metódy objektu Relácie **.close()**.

Táto metóda odstráni všetky ORM mapované objekty z Relácie a rozviaže transakčné zdroje z objektu **engine**, ku ktorému je Relácia viazaná a ktorý obsahuje pripojovacie údaje k databáze. Po opätovnom pripojení sa automaticky daný engine priradí k objektu Relácie a nová transakcia môže začať.

Typicky sa odporúča, aby bola transakcia čo najkratšia a aby bolo jasne poznať, kde začína a kde končí. K jasnému určeniu rozsahu transakcie nám slúži kontextový manažér. [7]

1.4 Aplikačné programové rozhranie

Tento názov je vo svete technológii známy pod skratkou API. Jedná sa o mechanizmus, ktorý umožňuje aplikácii alebo službe pristupovať k zdrojom v rámci inej aplikácie, či služby. Aplikácia alebo služba, ktorá pristupuje, sa nazýva **klient.** Aplikácia alebo služba obsahujúca zdroj sa nazýva **server**.

Títo účastníci komunikujú na základe requestu (ďalej ako požiadavka) a response (ďalej ako odpoveď).

Jedným z najznámejších API je REST API. Táto architektúra umožňuje pristupovať k dátam a robiť nad nimi CRUD(Create, Retrieve, Update, Delete) operácie. Na vykonávanie týchto operácii sa využívajú metódy komunikačného protokolu HTTP - GET,POST,PUT a DELETE.

Veľká výhoda REST API je, že je ho možné používať pri hociktorom programovacom jazyku a podporuje tiež rôzne formáty dát. Nato, aby takto všestranne fungovalo, musia však byť dodržané určité pravidlá:

- **Jednotné rozhranie** – všetky požiadavky na API pre rovnaký zdroj musia mať rovnakú štruktúru, bez ohľadu nato, odkiaľ požiadavka pochádza**.**
- **Bezstavovosť** – každá požiadavka musí obsahovať všetky informácie potrebné na jej spracovanie od API**.**
- **Možnosť využitia pamäte cache** – dáta by mali byť uložiteľné do vyrovnávacej pamäte na strane klienta alebo servera. Odpoveď servera musí však obsahovať informáciu o tom, či je ukladanie do vyrovnávacej pamäte povolené pre daný zdroj. Ak je toto povolené, je možné dáta uložiť do pamäte a použiť ich neskor pri ekvivalentných žiadostiach.

- **Oddelenie serverovej a klientskej časti systému –** pri návrhu REST API musia byť klientské a serverové aplikácie nezávislé. Jediná informácia, ktorú by mal klient poznať je URI požadovaného zdroja.

Podobne by serverová aplikácia nemala upravovať klientskú aplikácie inak, než jej vracať požadované údaje cez komunikačný protokol. [8] [9]

Príklad URI je zobrazený na obrázku 1.3 :

Obrázok 1.3: Príklad štruktúry URI

2. ŠTRUKTÚRA DÁTOVÉHO MODELU

Táto kapitola sa venuje návrhu a štruktúre databázového systému vytvoreného v relačnej databáze MySQL. Štruktúra databázy je reprezentovaná entitno-relačným diagramom. Dôležitou súčasťou aplikácie je aj jej bezpečnosť, ktorá môže byť zaistená pomocou JSON Web Tokenu v kombinácii s bezpečnostným prenosovým protokolom HTTPS.

2.1 MySQL

MySQL je relačný databázový systém vytvorený firmou MySQL AB. Keďže sa jedná o relačný systém, údaje sú uložené v tabuľkách, ktoré sú vzájomne prepojené vzťahmi.

MySQL sa vyznačuje ľahkou implementáciou, pričom ju je možné inštalovať na Linux aj Windows. Masívne sa rozšíril medzi ľudí aj vďaka tomu, že poskytuje okrem komerčnej verzie aj bezplatnú licenciu. Pre spoluprácu s jazykom Python poskytuje knižnice na vzájomnú komunikáciu ako použitá MySQLAlchemy.

Pri optimálnom množstve dát má MySQL oproti konkurencii výhodu. Tá sa však začína strácať pri veľkom množstve uložených dát, nakoľko rýchlosť potom výrazne klesá. [10]

Výhody a nevýhody MySQL:

+ open source databáza,
+ rýchlosť pri optimálnom množstve dát,
+ veľká rozšírenosť v komunite,
+ veľký počet komunikačných knižníc s programovacími jazykmi,
- pri veľkom množstve dát klesá rýchlosť.

2.2 Dátový model

Dátový model ponúka kompletný prehľad o využitých tabuľkách v databáze či atribútoch daných tabuliek a ich dátových typov. Tabuľky sú medzi sebou previazané rôznymi vzťahmi. Základné vzťahy, ktoré je možné využiť sú **1:N** a **M:N**.

Na vytvorenie vzťahu medzi tabuľkami je potrebné definovať pojmy ako primárny kľúč (PK) a neznámy kľúč (FK).

Primárny kľúč je vytvorený zo stĺpca v tabuľke, ktorý bude jednoznačne identifikovať každý záznam. Tabuľka s PK sa nazýva **parent table**.

Vytváranie PK by malo byť podriadené nasledujúcim pravidlám:

- **Krátky rozsah** – PK by mal byť čo najmenší, ideálne v číselnom formáte kvôli väčšej kompaktnosti oproti znakovým formátom. Vďaka jednoduchému číselnému formáte budú zjednodušené operácie nad databázou, ako napr. dopytovanie. [11]

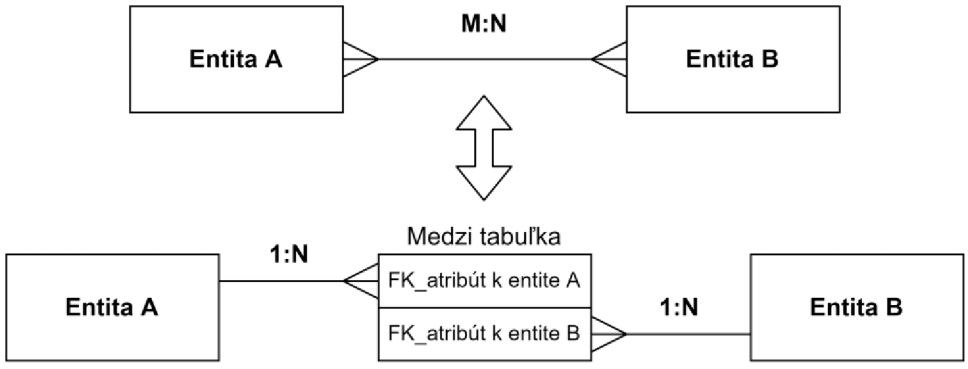
- **Nemennosť** – primárny kľúč by sa po prvotnom stanovení nemal meniť, nakoľko môže byť použitý v indexoch aplikácie pre dopytovanie a je previazaný k FK z inej tabuľky.

Neznámy kľuč sa vytvára zo stĺpca v tabuľke, ktorý bude referenciou na PK inej tabuľky. Keďže odkazuje na PK inej tabuľky, funguje ako krížový odkaz medzi tabuľkami, čím vytvára medzi nimi prepojenie. Tabuľka s FK sa nazýva child table.

Vzťah 1:N popisuje chovanie, kde je 1 prvok z entity A (parent table) previazaný s N prvkami z entity B (child table), pričom N patrí do množiny N+{0}. Naopak 1 prvok z entity B môže byť previazaný maximálne s 1 prvkom z entity A.

Vzťah M:N popisuje chovanie, kde 1 prvok z entity A je previazaný s N prvkami entity B a naopak N prvkov z entity A môže byť previazaných s 1 prvkom entity B. Prakticky to vyzerá tak, že medzi entitami, u ktorých je potrebný vzťah M:N je vytvorená medzi-tabuľka, ktorá obsahuje FK pre obe požadované entity, a tým pádom je možné vzťah M:N rozložiť na dva vzťahy 1:N. [12]

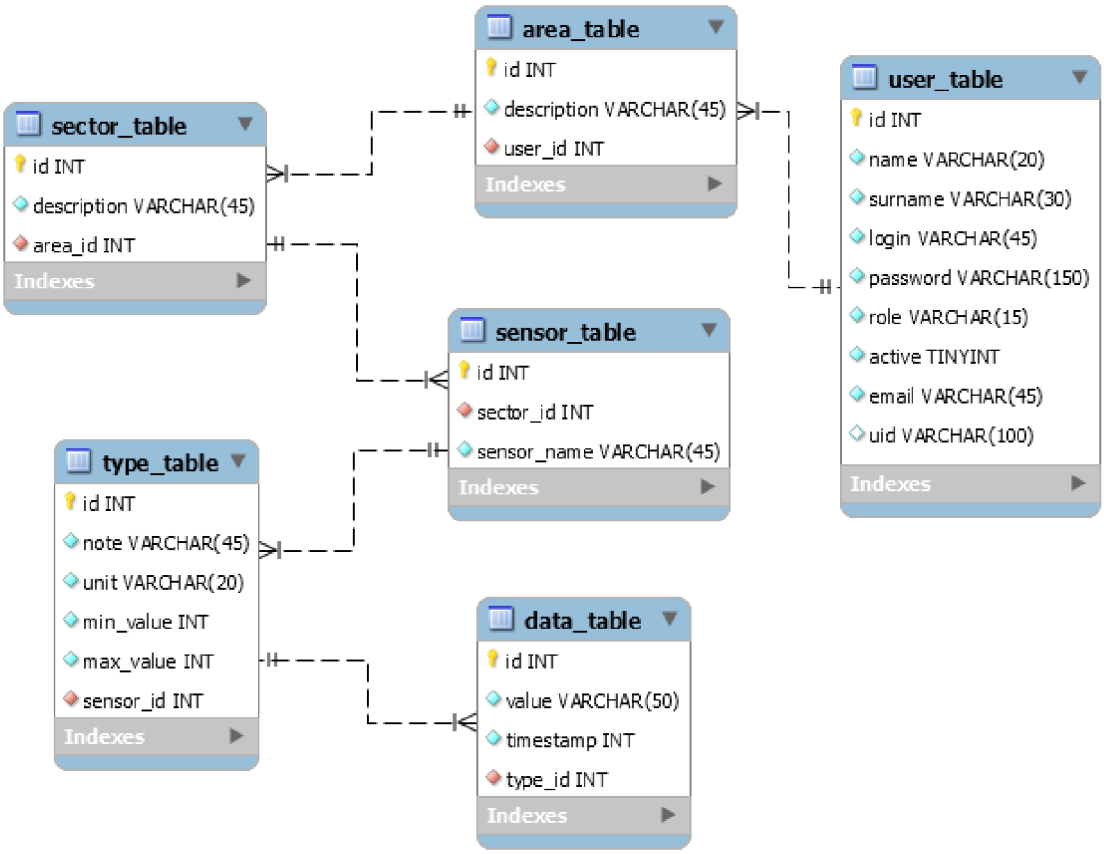
Ukážka vzťahu M:N je na vidieť na obrázku 2.1 :



Obrázok 2.1: Grafické znázornenie vzťahu M:N

Dátový model je možné vyjadriť pomocou entitno-relačného diagramu. Jedná sa o množinu objektov, ktoré pomáhajú na konceptuálnej úrovni opísať používateľskú aplikáciu s cieľom následnej špecifikácie štruktúry databázy. Množinu objektov predstavujú entity a ich vzájomné vzťahy-relácie. [13]

Entitno-relačný diagram použitého dátového modelu pre zadanú prácu je zobrazený na obrázku 2.2:



Obrázok 2.2: Entitno-relačný diagram využitého dátového modelu

2.3 Využité časti dátového modelu v aplikácii

Dátový model sa skladá z 5 hlavných častí, ktoré obsahujú entity popisujúce štruktúru databázy.

- Model pre ukladanie prijatých dát z meraní:
  - entita: data_table.
- Model definujúci názov použitého meracieho zariadenia:
  - entita: sensor_table.
- Model definujúci vlastnosti použitého meracieho zariadenia:
  - entita: type_table.
- Model definujúci informácie o užívateľoch:
  - entita: user_table.
- Model definujúci umiestnenie meracích senzorov:
  - entity: area_table, sector_table.

2.3.1 Model pre ukladanie prijatých dát z meraní

Tento model a jeho entity sú určené priamo pre klienta, ktorý bude pomocou aplikačného protokolu HTTP komunikovať s danou entitou a posielať do nej údaje o meraní. Tento model obsahuje nasledujúcu entitu:

- **data_table**

Daná entita popisuje, aké údaje majú byť klientom posielané na uloženie do databázy. Jedná sa o údaje nameranej číselnej hodnoty, času kedy bol údaj nameraný a typu meranej veličiny.

2.3.2 Model definujúci informácie o použitých meracích senzoroch

Model obsahuje vlastnosti meracích senzorov. Je potrebný pre kategorizáciu jednotlivých meraní a taktiež pre úkony vykonávané nad dátami. Tento model obsahuje 2 entity:

- **sensor_table**

Entita, ktorá v sebe obsahuje názov senzora a prepojenie na sektor, v ktorom sa daný senzor nachádza.

- **type_table**

Entita, ktorá obsahuje výpis vlastností daného senzora ako je jednotka merania, názov typu meranej veličiny, minimálna a maximálna hodnota rozsahu merania.

2.3.3 Model definujúci informácie o užívateľoch

Model obsahuje záznamy všetkých používateľov aplikácie a informácie o používateľských údajoch. Tento model obsahuje nasledujúcu entitu:

- **user_table**

Daná entita popisuje zoznam užívateľov. Tí sú jednoznačne identifikovaní pomocou **ID**. Taktiež obsahuje údaje o mene a priezvisku.

Ďalej entita obsahuje prihlasovacie atribúty ako **login** a **password**, pomocou ktorých sa užívateľ bude prihlasovať.

Registračný systém je vymyslený spôsobom, kde sa pri registrácií zobrazí administrátorovi v systéme záznam o novom používateľovi, ktorý ho môže potvrdiť. Na potvrdenie slúži atribút **active**.

Každý užívateľ disponuje určitými právami. Tieto práva popisuje atribút **role**. Tento atribút môže nadobudnúť 2 stavy a to :

1. **admin**: má prístup ku správe používateľov,
2. **user**: má prístup ku svojej konfiguračnej časti a môže graficky zobrazovať všetky svoje namerané hodnoty.

2.3.4 Model definujúci umiestnenie meracích senzorov

Model obsahuje informácie o tom, na akom mieste daný senzor vykonáva meranie. Merania sú rozdelené podľa nasledujúcich entít:

- **area_table**

Daná entita širšie vymedzuje priestor pôsobenia u daného meracieho zariadenia. Príkladom širšieho vymedzenia môže byť byt.

- **sector_table**

Daná entita užšie špecifikuje priestor pôsobenia u daného meracieho zariadenia. Príkladom užšej špecifikácie môže byť miestnosť v byte.

2.4 Zabezpečenie komunikácie

Zabezpečenie komunikácie je dôležité z hľadiska uchovania dátového toku len medzi klientom a serverom. Je potrebné, aby server dokázal rozoznať pravosť klienta a neposkytol dáta falošnému klientovi. Zabezpečenie komunikácie je možné vykonať viacerými spôsobmi. V kapitole budú spomenuté 2 spôsoby šifrovania a to JWT, HTTPS .

2.4.1 Json Web Token (JWT)

Jedná sa o otvorený štandard RFC 7519, ktorý definuje kompaktný spôsob na bezpečný prenos informácií medzi účastníkmi komunikácie vo forme objektu JSON. Tieto informácie sú dôveryhodné a overené, nakoľko sú digitálne podpísané. JWT môže byť podpísaný dvomi spôsobmi. Buď pomocou tajného kľúča použitím HMAC algoritmu ,alebo pomocou verejného/súkromného kľúča pomocou asymetrickej šifry RSA.

JWT sa za kompaktný považuje vďaka jeho veľkosti. Tá umožňuje jeho odoslanie pomocou URL, htttp metódy POST či vnútri http hlavičky. S jeho malou veľkosťou sa uplatňuje takisto veľká rýchlosť.

- **Štruktúra JWT**

JWT sa skladá z 3 častí, a to **hlavička, payload a podpis.** Navzájom sú oddelené bodkou.

1. **Hlavička**- Pozostáva typicky z 2 častí : typ tokenu a použitý hashovací algoritmus(RSA, HMACSHA256)

Príklad hlavičky je vidieť na obrázku 2.3:

```
{
  "alg": "HS256",
  "typ": "JWT"
}
```

Obrázok 2.3: Príklad hlavičky JWT [10]

Ďalej je hlavičkový JSON kódovaní pomocou Base64URL, a takto tvorí prvú časť JWT.

2. **Payload**- V tejto časti sú obsiahnuté nároky. Jedná sa o vyhlásenia o entite a ďalších metadátach. Existujú 3 typy nárokov, a to vyhradené, súkromné a verejné.

-*vyhradené* : jedná sa o preddefinované nároky, ktoré nie sú povinné, ale odporúčané. Predpokladá sa, že poskytujú súbor užitočných tvrdení. Príkladom sú nároky ako exp(expiračný čas), sub(predmet), aud(publikum). Všetky vyhradené nároky majú obmedzenú dĺžku na 3 znaky, vďaka čomu zaberá JWT menej pamäte.

-*verejné*: jedná sa o nároky, ktoré sú definované tými, čo JWT používajú. K predídeniu kolízií by mali však byť vymyslené nároky definované vo JSON IANA webovom registri tokenov.

-*súkromné*: jedná sa o vlastné nároky vytvorené na zdieľanie informácií medzi stranami, ktoré súhlasia s ich používaním.

Príklad payloadu je vidieť na obrázku 2.4:

```
{
  "sub": "1234567890",
  "name": "John Doe",
  "admin": true
}
```

Obrázok 2.4: Príklad payloadu JWT [14]

Po vytvorení časti payload je opäť využité kódovanie pomocou Base64URL a tak tvorí payload druhú časť JWT.

3. **Podpis**: Táto časť je vytváraná serverom a overuje pravosť JWT. Vytvára sa spojením zakódovaných častí hlavičky, payloadu a tajného maximálne 256bitového výrazu, ktorý pozná len server. Tieto všetky časti sú dané ako argument pre hashovací algoritmus(RSA, HMAC).

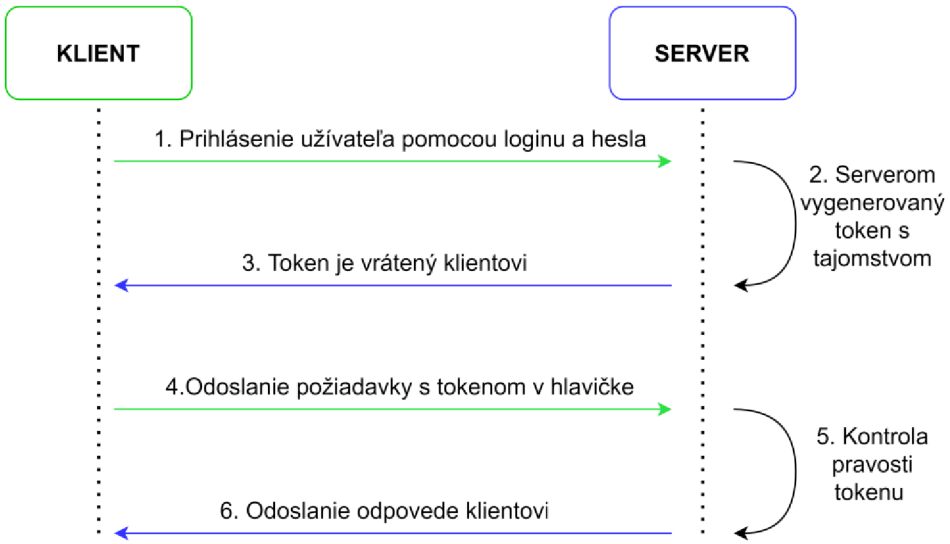
Po spojení všetkých častí vznikne kódovaný reťazec, ktorého časti sú oddelené bodkou. Tento string je ľahko použiteľný pri komunikácii pomocou HTTP protokolu či HTML. Ukážka kódovaného reťazca je na obrázku 2.5:

```
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.
eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6IkpvaG4
gRG9lIiwiaXNTb2NpYWwiOnRydWV9.
4pcPyMD09olPSyXnrXCjTwXyr4BsezdI1AVTmud2fU4
```

Obrázok 2.5: Príklad stringu JWT zakódovaného pomocou Base64URL a hashovacieho algoritmu HMAC [14]

JWT funguje ako autorizačný nástroj, a to znamená, že prichádza na scénu až po autentifikácii používateľa, napríklad pomocou prihlásenia cez meno a heslo.
Jeho hlavnou prednosťou je, že používateľ nemusí každý raz pri požiadavke na server prikladať všetky osobné údaje, aby ho server vedel identifikovať a verifikovať. Používateľ dostane od servera po prihlásení token, ktorý vždy priloží do hlavičky pri každej ďalšej požiadavke. Server následne overí pravosť informácii v tokene a ak sa zhoduje s tými, ktoré sám poslal klientovi tak klientovi pošle odpoveď na jeho žiadosť.

Opísaný mechanizmus je zobrazený na obrázku 2.6:



Obrázok 2.6: Mechanizmus JWT

Spolu s JWT sa odporúča používať zabezpečenie komunikácie pomocou protokolu HTTPS, pretože je dôležité zabezpečiť jeho prenos medzi klientom a serverom proti útoku z cudzích strán. [14]

2.4.2 HTTPS

Jedná sa o skratku pre Zabezpečený hypertextový prenosový protokol. Oproti HTTP protokolu ponúka zabezpečený prenos dát. To znamená, že prenášané dáta sú šifrované ,a tak sa znižuje riziko zneužitia odosielaných údajov, zámeny obsahu či odposluchu komunikácie. HTTPS komunikácia prebieha na TCP/IP porte 433.

Šifrovanie prebieha využitím protokolu SSL(Secure Socket Layer) alebo TLS (Transport Layer Security). Oba tieto protokoly využívajú architekturu klient-server, čiže sú vhodné pre danú aplikáciu. Dané protokoly majú 2 hlavné účely:

-overiť, či klient skutočne komunikuje priamo so serverom,

-zaistiť, že jedine server dokáže prečítať našu požiadavku a jedine klient dokáže prečítať odpoveď servera.

Oba protokoly spadajú pod druh asymetrickej kryptografie. Princíp tohoto druhu kryptografie je založený na dvojici kľúčov- verejného a súkromného. Verejný kľúč slúži k šifrovaniu, zatiaľ čo súkromný kľúč slúži k rozšifrovaniu.

SSL funguje tak, že verejný kľúč je serverom odoslaný klientovi a súkromný kľúč ostáva na serveri, kde je v bezpečí.
Akékoľvek údaje odoslané z webovej stránky šifrované pomocou verejného kľúča môžu byť dešifrované iba serverom, čím je vytvorená bezpečná komunikácia 1:1. [15] [16] [17]

Dané certifikáty bývajú platené. Nájdu sa však aj bezplatné skúšobné verzie, ktoré však majú nižšiu dobu platnosti, poprípade menej vlastností. Poskytovateľom takýchto bezplatných certifikátov sú napríklad firmy ZeroSSL, či SSL.com, ktoré ponúkajú SSL certifikáty zadarmo na 90 dní s možnosťou obnovy. [18] [19]

3. NÁVRH POŽIADAVIEK NA APLIKÁCIU A GRAFICKÚ ŠTRUKTÚRU

Táto kapitola sa zaoberá funkcionálnou špecifikáciou danej aplikácie. Sú to rozobrané požiadavky na aplikáciu, prípady využitia aplikácie používateľom a taktiež sa tu nachádza návrh používateľského rozhrania aplikácie vytvorený v dizajnérskej aplikácii Figma, ktorý následne bude v implementačnej časti vytváraný pomocou javascriptovej knižnice React.js.

3.1 Špecifikácia požiadaviek na aplikáciu

Tieto požiadavky boli zostavené na základe požadovaných funkcií, ktoré musí aplikácia obsahovať a vykonávať. Pri tvorbe funkčných požiadaviek sa vychádzalo z jej hlavnej úlohy, a to vizualizácie nameraných dát. Všetky požiadavky sú zobrazené v tabuľke 3.1:

Tabuľka 3.1: Funkčné požiadavky

Funkčné požiadavky	*Popis*
prihlásenie užívateľa	Používateľ sa musí prihlásiť kvôli všetkým ďalším úkonom
zobrazenie relevantných údajov	Prihlásenému užívateľovi sa zobrazia len tie dáta, na ktoré má práva.
grafické zobrazenie nameraných veličín	Výsledky meraní budú zobrazené v grafe spolu s údajmi o čase namerania.
zobrazenie doplňujúcich údajov	Zobrazenie doplňujúcich údajov ako priemerná, minimálna a maximálna hodnota.
možnosť výberu grafu meraní jednotlivého senzoru	Možnosť používateľa zvoliť si analýzu dát či už grafickú alebo textovú pre každý použitý senzor.
možnosť zmeny rozsahu časovej osy	Používateľ si môže zvoliť v akom období chce zobraziť namerané hodnoty.
senzorový manažment	Používateľ bude môcť pridať a odstrániť senzor pridať a odstrániť typ

3.2 Nefunkčné požiadavky

Tieto požiadavky sa týkajú doplnkových požiadaviek a nárokov na aplikáciu. Definovanie týchto požiadaviek je spojené so špecifikáciou využitia , platformy či zabezpečenia. Požiadavky sú zobrazené v tabuľke 3.2:

Tabuľka 3.2: Nefunkčné požiadavky

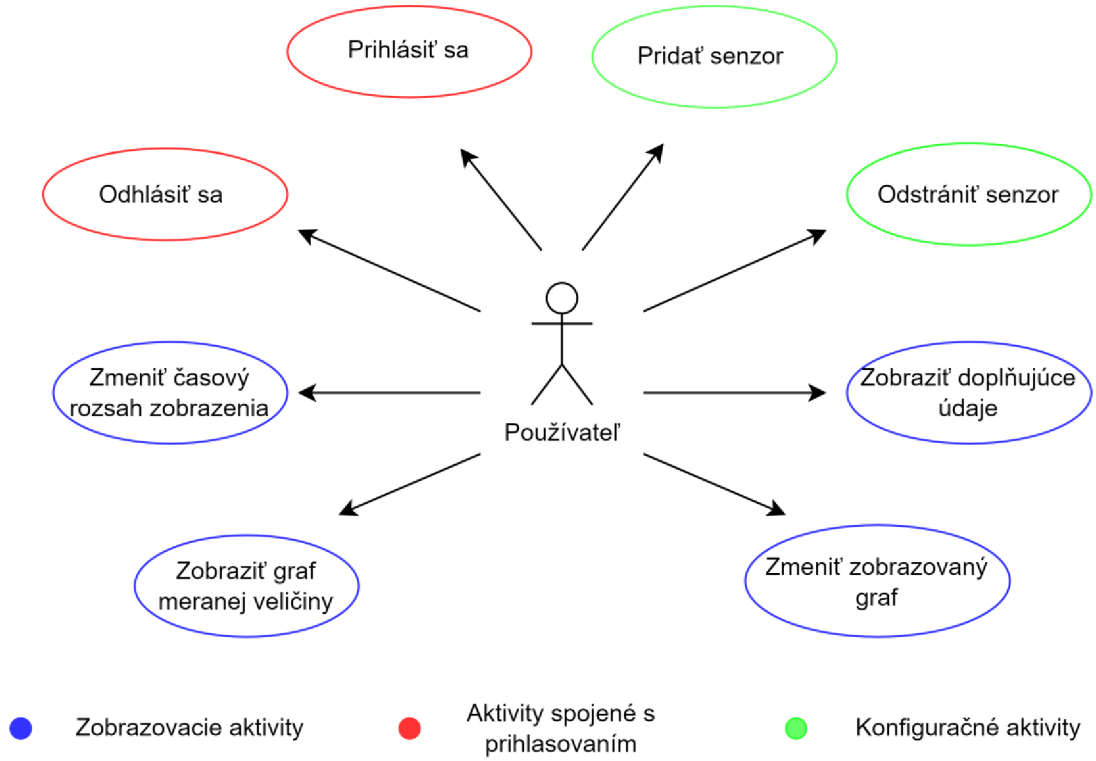
Nefunkčné požiadavky	Popis
vývoj webovej aplikácie	
implementácia zabezpečenia komunikácie	Zabezpečenie komunikácie klient-server pomocou JWT a SSL protokolu
implementácia responzívneho dizajnu	Webová aplikácia bude disponovať responzívnym dizajnom

3.3 Prípady použitia a návrh grafickej štruktúry aplikácie

Na základe funkčných požiadaviek je možné identifikovať jednotlivé prípady použitia aplikácie. V tejto podkapitole bude ukázaný opis jednotlivých prípadov použitia. Taktiež tu bude rozobraný opis grafickej štruktúry.

3.3.1 Prípady použitia aplikácie

Jedná sa o akcie, ktoré môžu daný používatelia vykonávať. Prípady použitia sú navrhnuté tak, aby obsiahli všetky funkčné požiadavky kladené na aplikáciu. Ich súhrn je zobrazený na obrázku 3.1:



Obrázok 3.1: Možnosť aktivít pre používateľa

3.3.2 Bližšia špecifikácia prípadov použitia

Prihlásiť sa – Používateľ zadá meno a heslo čím bude autentifikovaný serverom a prihlásený do webovej aplikácie.

Odhlásiť sa – Používateľ pri skončení práce v aplikácií klikne na tlačidlo, ktoré vykoná jeho odhlásenie a vrátenie na pôvodnú prihlasovaciu obrazovku.

Zobraziť graf meranej veličiny – Používateľ sa preklikne do obrazovky s grafom nameranej veličiny.

Zmeniť zobrazený graf – Používateľ sa môže v prípade využitia viacerých senzorov prepínať medzi jednotlivými grafickými zobrazeniami.

Zmeniť časový rozsah zobrazenia – Používateľ si môže zvoliť z viacerých možností : posledná hodina, posledných 24h, posledný týždeň.

Odstrániť senzor – V prípade potreby môže používateľ odstrániť senzor, pričom je zabezpečené kaskádové mazanie. Toto mazanie zabezpečí to že pri zmazaní záznamu z parent table (senzory) budú zmazané všetky nadväzujúce záznamy z child table (merania).

Pridať senzor – V prípade potreby môže používateľ pridať nový senzor na meranie, ktorý zaradí do príslušnej oblasti a sektoru. Takisto senzoru používateľ priradí veličiny, ktoré dokáže merať spolu s doplnkovými informáciami.

3.4 Návrh grafickej štruktúry aplikácie

Štruktúra aplikácia vychádza z vymedzenia funkčných požiadaviek a jednotlivých prípadov použitia aplikácie používateľom. Aplikácia bude rozdelená do viacerých blokov, kde každý blok bude predstavovať samostatnú obrazovku. Podľa potreby budú niektoré obrazovky obsahovať nadväzujúce podobrazovky. Aplikácia bude rozdelená na nasledujúce obrazovky :

Pre účel prihlásenia do systému:

- prihlasovacia obrazovka,
- registračná obrazovka.

K dispozícii pre používateľa:

- plavná obrazovka,
- grafická obrazovka,
- konfiguračná obrazovka.

K dispozícii pre administrátora:

- údržba používateľov.

3.4.1 Obrazovky pre účel prihlásenia

- **Prihlasovacia obrazovka**

V tejto obrazovke bude mať používateľ možnosť sa prihlásiť pomocou loginu a hesla, čím mu bude umožnené vstúpiť do hlavnej obrazovky aplikácie. Taktiež je možné sa z tejto obrazovky dostať na registračnú obrazovku.

- **Registračná obrazovka**

Do tejto obrazovky bude budúci používateľ presmerovaný po kliknutí na tlačidlo Registrovať sa v prihlasovacej obrazovke.

V tejto obrazovke prebieha registrácia budúceho používateľa vyplnením potrebných údajov o mene osoby, emailovej adrese a prihlasovacích údajov. Po vyplnení údajov, je vytvorená autentifikačná požiadavka na administrátora k potvrdeniu registrácie.

3.4.2 Obrazovky pre používateľa

Každá obrazovka sa bude skladať z dvoch častí:

Menu (ľavá časť obrazovky)

Prvú časť pokrýva navigačný panel MENU, v ktorom používateľ vidí zoznam všetkých obrazoviek, ktoré má možnosť využiť. Zoznam obrazoviek bude nasledovný:

- **Hlavná stránka,**
- **grafy,**
- **konfigurácia.**

Taktiež sa tu bude nachádzať tlačidlo na odhlásenie používateľa.

Hlavná časť (pravá časť obrazovky)

Hlavná časť zobrazuje obsah jednotlivých častí navigačného panelu. Ďalej budú popísané jednotlivé hlavné časti :

- **Hlavná obrazovka**

Po úspešnom prihlásení bude používateľ presmerovaný do hlavnej obrazovky.

V Hlavnej časti aplikácie sa nachádza prehľadová tabuľka so zoznamom všetkých pridaných senzorov daného používateľa. Tabuľka obsahuje okrem zoznamu aj dodatočné informácie o senzore ako jednotka merania, či rozsah meranej veličiny.

V tabuľke bude taktiež možnosť konkrétny senzor zmazať.

- **Grafická obrazovka**

V hornej časti grafickej obrazovky budú k dispozícii nastavovacie komponenty . Tieto komponenty budú rozdelené do 2 skupín. Prvá bude mať za úlohu vybrať rozsah zobrazenia a dátum. V druhej skupine sa bude postupným výberom cez výber oblasti, sektora, senzora a typu meranej veličiny potvrdzovať výber želaných dát na zobrazenie do grafu.

Pod nastavovacou časťou sa bude nachádzať graf, ktorý podľa nastavenia zobrazí používateľovi namerané dáta daného senzora.

- **Konfiguračná obrazovka**

V hlavnej časti konfiguračnej obrazovky sa budú nachádzať 3 formuláre:

1. **Formulár na pridanie oblasti** : používateľ bude mať možnosť pridať názov jeho novej oblasti.
2. **Formulár na pridanie sektora** : používateľ si vyberie oblasť, do ktorej chce budúci sektor priradiť a vyplní názov sektora.
3. **Formulár na pridanie senzora** : používateľ si vyberie oblasť a sektor, v ktorej chce mať budúci senzor priradený a vyplní potrebné informácie k úspešnému pridaniu senzora.

3.4.3 Obrazovka pre administrátora

Obrazovka sa bude skladať z dvoch častí:

1. Menu (ľavá časť obrazovky)

Prvú časť pokrýva navigačný panel MENU, v ktorom používateľ vidí zoznam všetkých obrazoviek, ktoré má možnosť využiť. Zoznam obrazoviek bude nasledovný:

- **Konfigurácia**

Taktiež sa tu bude nachádzať tlačidlo na odhlásenie používateľa.

2. Hlavná časť (pravá časť obrazovky)

Hlavná časť zobrazuje obsah jednotlivých častí navigačného panelu. Ďalej budú popísané jednotlivé hlavné časti :

- **Konfiguračná obrazovka**

Po úspešnom prihlásení bude používateľ presmerovaný do hlavnej obrazovky.

V Hlavnej časti aplikácie sa nachádza prehľadová tabuľka so zoznamom všetkých používateľov. V tejto tabuľke môže administrátor zmeniť stav aktívnosti účtu konkrétneho používateľa. Pri aktivácií účtu bude používateľovi odoslaný potvrdzovací email. Pri deaktivácii účtu bude odoslaný deaktivačný email.

3.5 Figma

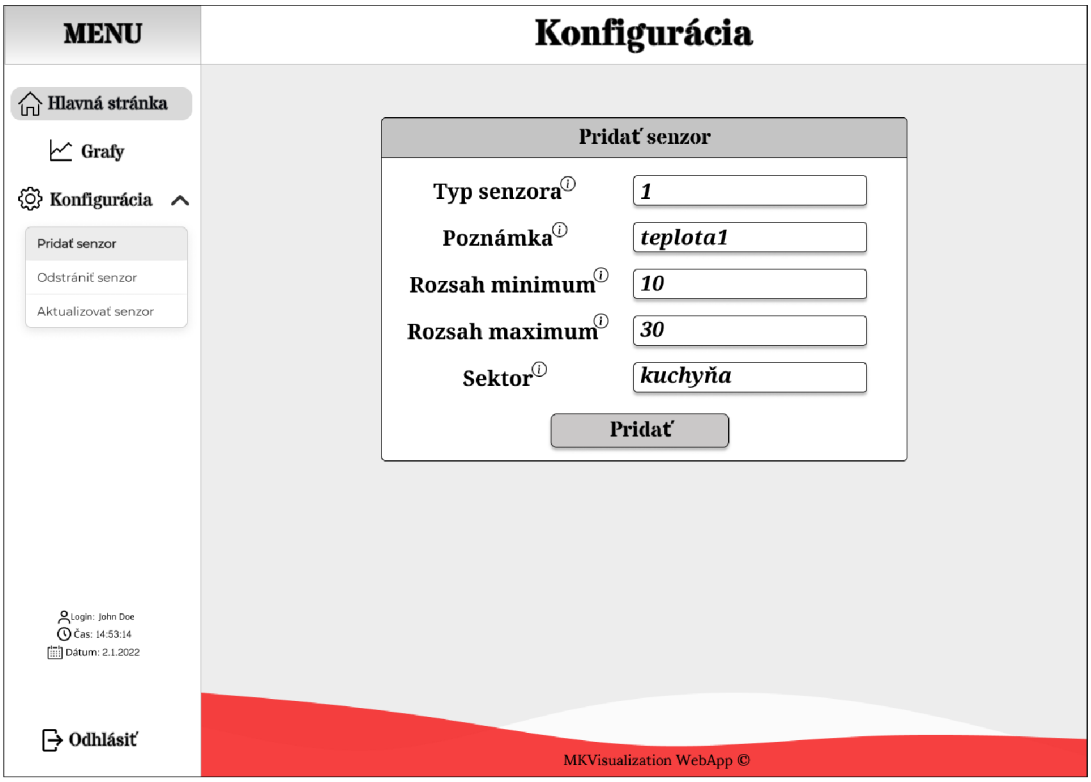
Na účel vytvorenia návrhu aplikácie bola použitá webová aplikácia Figma, ktorá slúži ako návrhársky grafický nástroj. Používateľ má k dispozícii rôzne geometrické tvary, škálu farieb, štýly, parametre a grafické komponenty, vďaka ktorým je možné vytvoriť unikátny dizajn.

Hlavná výhoda použitia návrhárskeho nástroja je následná možnosť využiť navrhnutý dizajn pri implementácii. To je možné pomocou exportovania použitých štýlov.

Výhody danej platformy:

- **Beží online v prehliadače** – zaniká potreba inštalácie programu.
- **Možnosť kolaborácie pomocou pracovnej skupiny** – vhodné najmä pre firmy či iné inštitúcie, nakoľko na návrhu môžu pracovať viacerí používatelia naraz.
- **Zdieľané knižnice komponentov -** možnosť vytvorenia zdieľanej knižnice komponentov, ktorej obsah uvidia všetci ľudia vo danej pracovnej skupiny.
- **Brainstorming** – Figma ponúka priestor na zdieľanie myšlienok a ideí vzájomne medzi komunitou vo forme FigJam**. [20]**

Príklad návrhu dizajnu aplikácie je zobrazený na obrázku 3.2 :



Obrázok 3.2: Príklad návrhu konfiguračnej obrazovky pre pridanie senzora

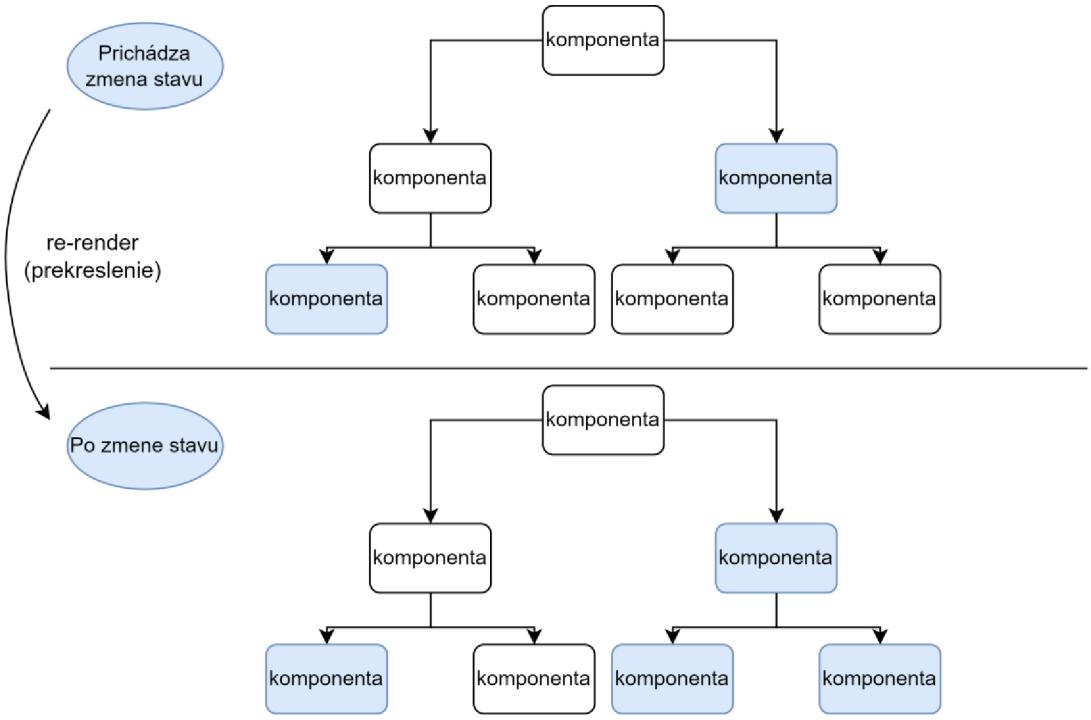
3.6 React.js

Jedná sa o javascriptovú knižnicu vynájdenú firmou Facebook. Slúži pre tvorbu užívateľského rozhrania UI. Základný stavebný prvok knižnice React.js tvoria komponenty, ktoré majú svoje vlastnosti (props) a spravujú svoj vnútorný stav (state). React používa JSX syntax, čo je XML/HTML syntax , ktorá produkuje React elementy. Táto syntax je následne prevedená na jazyk javascript kompilérom Babel. Výhodou je, že v jednom súbore dokáže používateľ písať JSX syntax aj javascript syntax a Babel všetko pretransformuje na spoločný javascript.

Využitie JSX nie je nutné. Táto syntax je však v dnešnej dobe veľmi využívaná vďaka jej prehľadnosti a jednoduchosti.

React funguje na princípe prekresľovania len tých komponentov/elementov, u ktorých prišlo k zmene stavu pri porovnaní s predchádzajúcim stavom. [21] [22]

Princíp prekresľovania je zobrazený na obrázku 3.3 :



Obrázok 3.3: Princíp prekresľovania dotyčných uzlov

Výhody React.js :

- **rýchlosť,**
- **jednoduchosť pri vytváraní dynamických webových aplikácií,**
- **opakovane použiteľné komponenty,**
- **silná komunita používateľov- dostupný prístup k informáciám.**

4. IMPLEMENTÁCIA APLIKÁCIE

Táto kapitola je zameraná na popis implementácie aplikácie. Popis zahŕňa serverovú aj klientskú časť.

4.1 Komunikačná štruktúra

Nakoľko budú požiadavky na ukladanie dát do databázy posielané od viacerých klientov, bolo potrebné vymyslieť univerzálnu dátovú štruktúru, vďaka ktorej bude možné rozoznať správne uloženie dát pre daného klienta. Údaje medzi klientom a serverom sú prenášané v textovom formáte JSON, ktorý bol zvolený pre jeho ľahkú čitateľnosť a taktiež ponúka ľahkú implementáciu pre klienta.

4.1.1 Dátová štruktúra požiadavky

Požiadavka obsahuje jednoznačný identifikátor pre určenie, komu dáta patria. Tento identifikátor je UID. Je to špeciálne vygenerovaný reťazec znakov, ktorý je danému senzoru určený priamo pri jeho vytváraní v používateľskom rozhraní. Každý senzor má typy veličín, ktoré dokáže merať, preto sa nachádza v požiadavke tiež. Na základe týchto dvoch údajov je jasné, ako je potrebné dáta uložiť. Okrem údajov o **type meranej veličiny** sa ukladá **hodnota nameranej veličiny** a **čas namerania**.

Príkladová štruktúra požiadavky je zobrazená na zdrojovom kóde 4.1 :

```
{
"uid": "5105b4f7-2616-493d-9871-ede1e7e1e5ae",
"data": [
{
"value": 69,
"timestamp":16513512,
"type": "forecast_temperature"
},
{
"value": 99,
"timestamp":16513512,
"type": "outside_temperature"
},
{
"value": 42,
"timestamp":16513512,
"type": "road_temperature"
}]
}
```

Zdrojový kód 4.1: Štruktúra požiadavky pre uloženie dát

4.1.2 Dátová štruktúra odpovedi

V prípade, že server dostane celú štruktúru odpovede bez vynechania jedinej veci, dáta budú uložené do databázy a klientovi je vrátená štruktúra, ktorá je zobrazená na zdrojovom kóde 4.2 :

```
{"status":true, "message":"Data inserted to database"}
```

Zdrojový kód 4.2: Štruktúra odpovede pre platnú požiadavku

4.2 Architektúra serverovej časti aplikácie

Kapitola je zameraná na vysvetlenie štrukturalizácie serverovej časti aplikácie. Táto časť bola vytvorená pomocou programovacieho jazyka Python za použitia knižnice Flask. Popis daných technológií sa nachádza v kapitole č.1.

4.2.1 Controller

V danej časti sú popísané jednotlivé koncové body pre každý zdroj dát – tzv. **endpointy.** Popis bude vykonaný v tvare **HTTP metóda:/URL.**

Endpointy sa dajú logicky rozdeliť podľa zdrojov ku ktorým pristupujú. Rozdelenie vyzerá nasledovne:

Endpointy pre **používateľa**:

- **POST:/api/user/sign-up** – slúži pre zaregistrovanie používateľa.
- **POST:/api/user/activate** – slúži pre potvrdenie registrácie používateľa ako aj pre deaktiváciu jeho účtu.
- **POST:/api/user/auth** – slúži pre prihlásenie používateľa do systému a na vygenerovanie autentifikačného tokenu pre potreby posielania dátových požiadaviek na server.
- **GET:/api/user/all** – slúži pre získanie zoznamu všetkých používateľov.

Endpointy pre **oblasť**:

- **GET:/api/area** –slúži na získanie všetkých oblastí pre daného používateľa.
- **POST:/api/area/add** – slúži na pridanie oblasti, ktorá bude patriť jednoznačne jednému používateľovi.
- **GET:/api/area/sectors** – slúži na získanie zoznamu oblastí a zoznamu všetkých sektorov patriacich ku jednotlivým oblastiam.

Endpointy pre **sektor**:

- **GET:/api/sector/getAreasSector** – slúži na získanie všetkých sektorov, ktoré patria danej oblasti.

- **POST:/api/sector/add** – slúži na pridanie sektora, ktorý bude patriť jednoznačne jednej oblasti.
- **GET:/api/sector** – slúži na získanie zoznamu všetkých sektorov priradených danej oblasti.

Endpointy pre **senzor**:

- **POST:/api/sensor/add** – slúži na pridanie senzora, ktorý bude jednoznačne patriť do jedného sektoru.
- **GET:/api/sensor** – slúži na získanie zoznamu všetkých senzorov, ktoré patria danému sektoru.
- **GET:/api/sensor/all** – slúži na získanie informácii o všetkých senzoroch pre prehľadovú tabuľku
- **DELETE:/api/sensor**– slúži na vymazanie daného senzora.

Endpointy pre **typy meraných veličín**:

- **GET:/api/sensor_type** – slúži na získanie zoznamu všetkých typov meraných veličín pre daný senzor.

Endpointy pre **dáta**:

- **POST:/api/data/add** – slúži na pridanie merania do databázy.
- **POST:/api/data/chart** – slúži na získanie správnych dát pre potreby grafu.

4.2.2 Model

V danej časti sa nachádza popis tried jazyka Python reprezentujúce dátové štruktúry, ktoré mapujú reálne databázové tabuľky a ich obsah. Pre interné účely sa nazývajú namapované triedy modely.

Taktiež sa tu nachádzajú triedy reprezentujúce pomocné dátové štruktúry (DTO), ktoré majú za úlohu zoskupiť informácie z viacerých dátových štruktúr do jednej, ktorá bude použitá pri odpovedi klientovi.

Väčšina modelov má vytvorené serializačné funkcie, ktoré slúžia na prispôsobenie štruktúry, ktoré budú vracať funkcie používajúce dané modely.

Každá trieda popisuje databázovú tabuľku z dátovej štruktúry, ktorá bola navrhnutá v kapitole 2. Štruktúra dátového modelu a je zobrazená na obrázku č.5. Prehľad daných modelov je uvedený v prehľadovej tabuľke 4.1 :

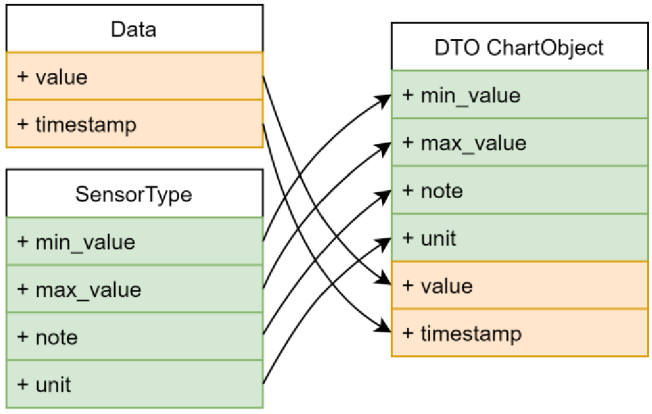
Tabuľka 4.1: Modely a ich referenčné databázové tabuľky

Model	Referenčná databázová tabuľka
User	user_table
Area	area_table
Sector	sector_table
Sensor	sensor_table
SensorType	type_table
Data	data_table

DTO triedy

1. ChartObjectDTO- jedná sa o pomocnú triedu, ktorá združuje hodnoty o vybranej veličine daného senzora, ktorá bude zobrazená na grafe.
2. ChartValueDTO- jedná sa o pomocnú triedu, ktorá priamo obsahuje veličiny zobrazované v grafe, kde na ose x je to čas nameranej hodnoty a ose y nameraná hodnota.
3. RealValueDTO- jedná sa o pomocnú triedu, ktorá slúži na overenie správnosti pri priraďovaní časov merania do správnych intervalov pre zobrazenie.
4. AreaWithSectorsDTO – jedná sa o pomocnú triedu, ktorá slúži na výpis všetkých oblastí a k nim priradených sektorov.
5. SectorsDTO – je pomocná trieda, ktorá združuje pre AreaWithSectorsDTO všetky sektory patriace k jednej oblasti a pripája ich do jej štruktúry.

Ako ukážkový príklad bola zvolená trieda ChartObjectDTO, kde je možné vidieť ako jej obsah, tak jej vznik. Príklad je zobrazený na obrázku 4.1:



Obrázok 4.1: Vznik DTO triedy ChartObject

4.2.3 Services

Popis API obsahuje funkcie, ktoré na základe požiadavky vyťahujú z databázy dáta a parsujú ich na odpoveď, ktorá bude zobrazená klientovi .

Pri vybraných kľúčových funkciách boli z dôvodu lepšej názornosti či zložitosti funkcie zostrojené vývojové diagramy, ktoré vyjadrujú priebeh funkcie.

- **get_all_users()**

Popis: jedná sa o funkciu, ktorá vracia zoznam všetkých registrovaných používateľov. Ide o administrátorskú funkciu.

Funkcionalita:

1. Na databázu je zavolaný dopyt, ktorý vracia zoznam všetkých používateľov webovej aplikácie. Ak je zoznam prázdny, funkcia vráti správu : **Žiadny používateľ nebol nájdený a kód 200.**

- **auth_user()**

Popis: jedná sa o funkciu, ktorá zisťuje, či je používateľ pokúšajúci sa o prihlásenie oprávnený.

Vstupné parametre:

- login (prihlasovacie meno),
- password (heslo).

Funkcionalita:

1. Vytiahnuť riadok používateľa s daným loginom z databázy spolu so všetkými informáciami o jeho účte.
2. Overiť, či je daný login zaznamenaný v databáze. Ak nie je funkcia vráti správu: **Používateľ s daným loginom neexistuje a vráti kód 401.**
3. Overiť, či zadané heslo súhlasí s heslom z databázy. Ak nie je, funkcia vráti správu: **Zlé heslo a vráti kód 401.**
4. Overiť, či je používateľ aktivovaný administrátorom. Ak nie je, funkcia vráti správu: **Váš účet ešte nie je aktivovaný a vráti kód 403**.
5. Ak prešiel používateľ krokmi 2-4 v poriadku, je potrebné okresať vytiahnuté informácie z kroku 1. To má za úlohu serializačná funkcia serialize_token_payload, ktorá je zobrazená na zdrojóm kóde 4.3 :

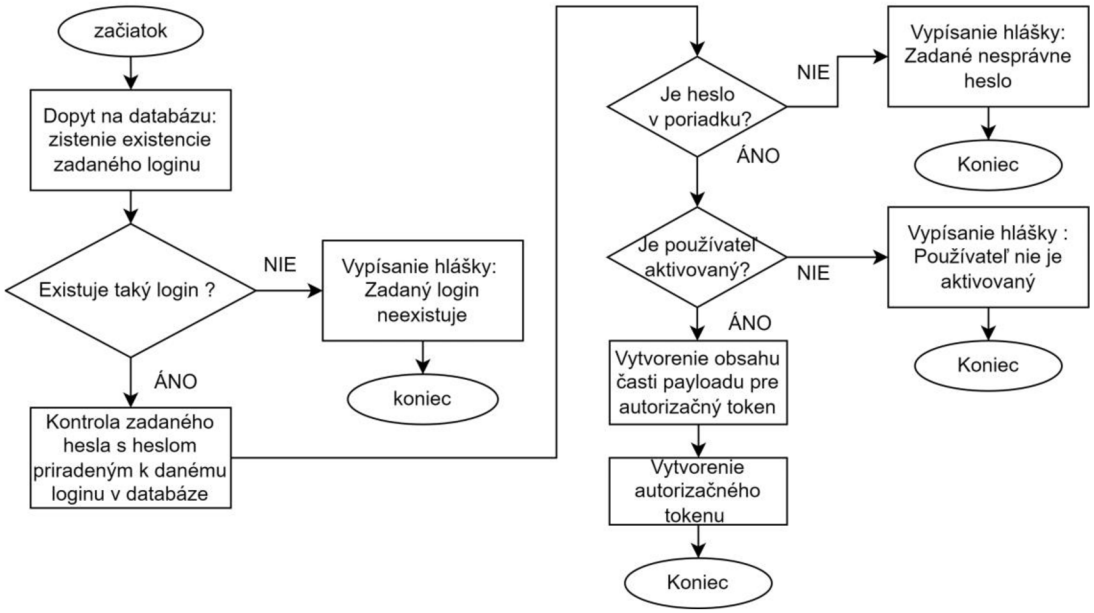
```
@staticmethod
    def serialize_token_payload(user):
        return {'login': user.login,
                'user_id': user.id,
                "active": user.active if hasattr(user, 'active') and
user.active else False}
```

Zdrojový kód 4.3: Serializačná funkcia pre obsah tokenu

6. Funkcia auth_user() teda pri úspešnej autorizácii používateľa vracia **štruktúru s prvkami login, user_id, active a kód 200.**

7. Vrátená štruktúra je totiž potrebná pre autentifikáciu klienta pri odosielaní požiadaviek na server.
8. Autentifikačná štruktúra je daná ako argument funkcii, ktorá vytvorí prístupový token. Daná funkcia na vytvorenie prístupového tokenu bola použitá z knižnice JWT.

Jednotlivé kroky danej funkcie sú zosumarizované v prehľadovom vývojovom diagrame, ktorý je zobrazený na obrázku 4.2 :



Obrázok 4.2: Vývojový diagram prihlasovacej funkcie

- **sign_up_user()**

Popis: jedná sa o funkciu, ktorá má za úlohu zaregistrovať klienta.

Vstupné parametre:

- name (meno),
- lastname (priezvisko),
- login (prihlasovacie meno),
- password (heslo),
- email.

Funkcionalita:

1. Na databázu je zavolaný dopyt s cieľom zistiť, či dané používateľské meno už neexistuje. Ak áno, funkcia vráti správu : **Používateľ s daným loginom už existuje.**
2. Ak je login jedinečný, prichádza na rad hashovacia funkcia, ktorá vytvára unikátny hash pre heslo.

Tento krok je urobený z hľadiska bezpečnosti, nakoľko by sa nemal ukladať do databázy nechránený údaj o hesle.

3. Vytvorenie objektu triedy User, ktorý je naplnený vstupnými dátami.
4. Vytvorenie kontextového manažéra, v ktorom je zavolaná na pridávaný objekt z kroku 3. metóda .add() na pridanie objektu do databázy
5. Úspešný priebeh funkcie sign_up_user() vracia správu : **Registrácia prebehla úspešne, prosím čakajte na konfirmačný email.**

- **change_users_active_state()**

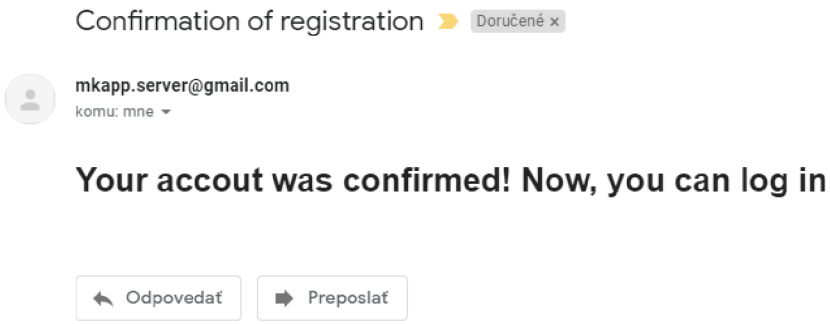
Popis: jedná sa o funkciu, ktorá má dva prípady použitia. Buď klientovi potvrdí registráciu, alebo klienta deaktivuje . Aký prípad nastane je dané z vyhodnotenia aktivačnej premennej. Jedná sa o administrátorskú funkciu.

Vstupné parametre:

- user_id (identifikátor používateľa),
- is_active (ukazovateľ aktívnosti účtu).

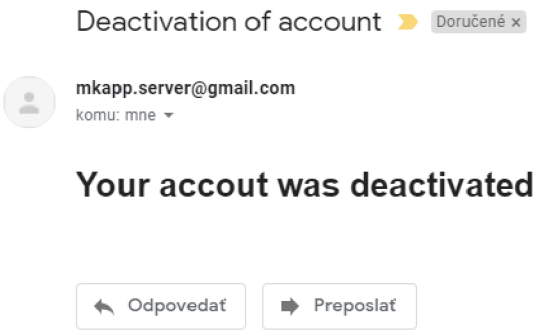
Funkcionalita:

1. Na databázu sa zavolá dopyt s cieľom vytiahnuť z databázy podľa user_id objekt daného používateľa. Ak taký používateľ neexistuje, funkcia vráti správa **: Používateľ nebol nájdený**.
2. Ak príde vo vstupných dátach kľúč is_active s hodnotou True, tak je následne poslaný konfirmačný email. Ukážka tohto emailu je na obrázku 4.3 :



Obrázok 4.3: Potvrdzovací email určený pre registráciu používateľa

3. Ak príde vo vstupných dátach kľúč is_active s hodnotou False, tak je následne poslaný deaktivačný mail. Ukážka tohto mailu je na obrázku 4.4 :



Obrázok 4.4: Deaktivačný email potvrdenie registrácie používateľa

O posielanie emaila sa stará SMTP(Simple Mail Transfer Protocol) server , ktorý na fungovanie potrebuje nasledujúce údaje:

- **Port, host**: SMTP podporuje mnohé emailové služby. V diplomovej práci bola zvolená služba gmail.
- **Sender**: odosielateľom je email vytvorený na účely administrácie: mkapp.server@gmail.com
- **Password**: heslo k administračnému emailu.
- **Subject**: predmet emailu
- **Message**: správa poslaná v tele emailu

Po získaní vyššie popísaných údajov prichádza vytvorenie spojenia na SMTP server- pripojenie na Gmail. Pri úspešnom spojení je odoslaný mail používateľovi.

- **get_all_user_areas()**

Popis: jedná sa o funkciu, ktorá má za úlohu získať zoznam všetkých oblastí patriacich konkrétnemu používateľovi.

Vstupné parametre:

- user_id (identifikátor používateľa).

Funkcionalita:

1. Na databázu je zavolaný dopyt s cieľom zistiť všetky oblasti, ktoré patria danému používateľovi. Pri zistení neprítomnosti aspoň jednej oblasti vracia funkcia správu : **Používateľ nemá žiadnu oblasť a kód 200**.
2. V prípade, že používateľ má aspoň jednu oblasť, je objekt tejto oblasti poslaný ako argument serializačnej funkcii.

Daná funkcia je zobrazená na zdrojovom kóde 4.4:

```
@staticmethod
def serialize_area(area):
    return {"description":area.description, "id": area.id}
```

Zdrojový kód 4.4: serializačná funkcia pre oblasť

3. Funkcia get_all_areas() vracia pole takto serializovaných oblastí a kód 200.

- **get_areas_with_sectors()**

Popis: jedná sa o funkciu, ktorá má za úlohu získať všetky oblasti, ku ktorým má daný používateľ prístup a takisto všetky sektory, ktoré patria do týchto oblastí.

Vstupné parametre:

- user_id (identifikátor používateľa).

Funkcionalita :

1. Na databázu je zavolaný dopyt, ktorý zistí všetky oblasti patriace danému používateľovi a uloží ich do premennej . Ak sa žiadna oblasť nenašla, funkcia vráti správu : **Používateľ nemá žiadnu oblasť a kód 200**.
2. Je vytvorené prázdne pole, do ktorého sa budú ukladať objekty DTO triedy AreaWithSectorsDTO.
3. Nastáva cyklické prechádzanie objektov získaných z dopytu z bodu 1, v ktorom je vytvorený objekt typu AreaWithSectorsDTO združujúci informácie o oblasti. Taktiež je tu vytvorený dopyt na databázu, vďaka ktorému sú zistené všetky sektory patriace danej oblasti. Výsledok dopytu uložený v objekte typu SectorDTO je zlúčený s objektom typu AreaWithSectorsDTO a funkcia vracia zoznam oblastí daného používateľa a ich sektorov.

- **add_area()**

Popis: jedná sa o funkciu, ktorá ma za úlohu pridať používateľovi oblasť do databázy.

Vstupné parametre:

- user_id (identifikátor používateľa),
- description (názov oblasti).

Funkcionalita:

1. Vstupné dáta sú použité pri vytváraní objektu triedy Area.
2. Je vytvorený kontextový manažér, ktorý otvára reláciu, pridá objekt do databázy a reláciu uzavrie.
3. Funkcia vráti správu : **Oblasť s id {} bola vložená do databázy, kód 201**.

- **add_sector()**

Popis: jedná sa o funkciu, ktorá má za úlohu pridať sektor, ktorý jednoznačne patrí len jednej oblasti.

Vstupné parametre:

- area_id (identifikátor oblasti),
- description (názov sektora).

Funkcionalita:

1. Z požiadavky prichádzajú vstupné parametre, ktoré sú uložené do premenných.
2. Je vytvorený objekt triedy Sector, ktorého sú vstupné dáta predané vo forme argumentu.
3. Je vytvorený kontextový manažér, ktorý otvára reláciu, pridá objekt do databázy a reláciu zatvára.

- **get_all_sectors_for_area()**

Popis funkcie: jedná sa o funkciu, ktorá má za úlohu získať zoznam všetkých sektorov patriacich pod vybranú oblasť.

Vstupné dáta:

- area_id (identifikátor oblasti)

Funkcionalita:

1. Na databázu je zavolaný dopyt, ktorého výsledkom je pole objektov triedy Sector.
2. Toto pole je cyklicky prechádzané a každý prvok je serializačnou funkciou upravený na potrebnú podobu s názvom sektora a s identifikátorom ID.

- **get_all_sensors_for_sectors()**

Popis funkcie: jedná sa o funkciu, ktorá má za úlohu získať zoznam všetkých senzorov patriacich pod vybraný sektor.

Vstupné dáta:

- sensor_id (identifikátor sektora).

Funkcionalita:

1. Na databázu je zavolaný dopyt, ktorého výsledkom je pole objektov triedy Sensor.
2. Toto pole je cyklicky prechádzané a každý prvok je serializačnou funkciou upravený na potrebnú podobu s názvom senzora a s identifikátorom ID.

- **get_all_sensor_types_for_sensor()**

Popis funkcie: jedná sa o funkciu, ktorá má za úlohu získať zoznam všetkých typov meraných veličín patriacich ku vybranému senzoru.

Vstupné dáta:

- sensor_id (identifikátor senzora).

Funkcionalita:

1. Na databázu je zavolaný dopyt, ktorého výsledkom je pole objektov triedy SensorType.
2. Toto pole je cyklicky prechádzané a každý prvok je serializačnou funkciou upravený na potrebnú podobu s názvom meranej veličiny a s identifikátorom ID.

- **get_all_sensors_for_user()**

Popis funkcie: jedná sa o funkciu, ktorá má za úlohu vytvoriť objekt, ktorý bude naplnený informáciami o názve senzoru a informáciami ohľadne meraných veličín. Tento objekt bude následne poslaný k uloženiu do databázy.

Vstupné dáta:

- user_id (identifikátor používateľa).

Funkcionalita:

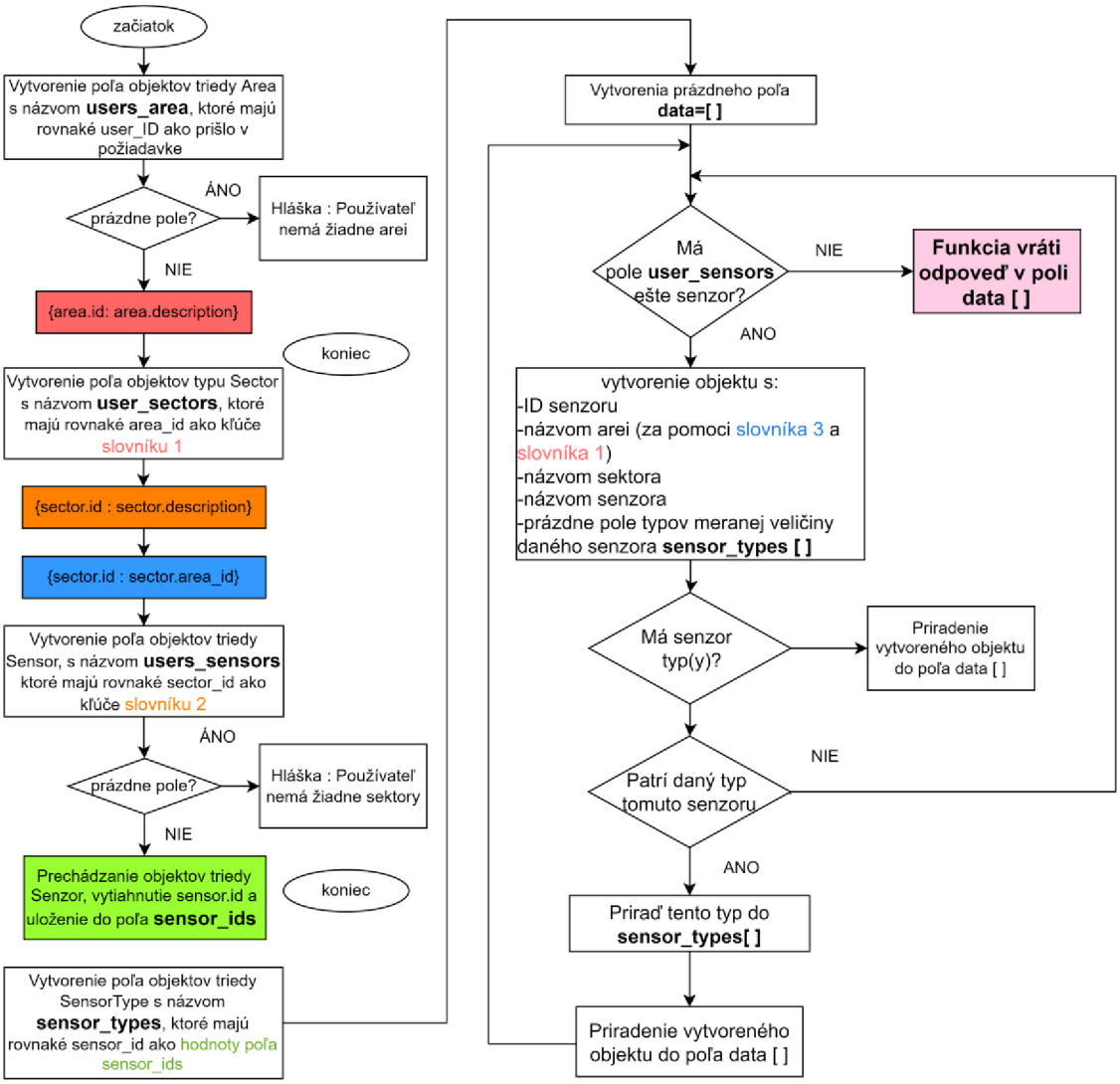
1. Z požiadavky prichádza vstupný parameter, ktorý je použitý v kontextovom manažéri, kde prebieha dopyt na databázu na vrátenie všetkých oblastí daného usera.
2. Dopyt vráti pole objektov triedy Area. Ak je pole prázdne, funkcia vráti správu: **Používateľ nemá žiadne oblasti a kód 200.**
3. V prípade, ak používateľ má viac ako 0, tak for cyklus prechádza pole objektov triedy Area, ktoré sú ukladané ako slovník kde kľúčom je **ID oblasti** a hodnotou **názov oblasti**.
4. Je vykonaný ďalší dopyt na databázu , v ktorom sa získavajú z databázy tie sektory, ktoré majú area_ID (cudzí kľúč na prepojenie s tabuľkou Area) rovnaké ako je ID oblasti zo slovníka z bodu 3.
5. Dopyt vráti pole objektov triedy Sector. Ak je pole prázdne, funkcia vráti správu: **Používateľ nemá pre zadanú oblasť žiadne sektory a kód 200.**
6. V prípade, že používateľ má viac ako 0 sektorov, tak for cyklus prechádza pole objektov triedy Sector. Výsledok for cyklu je uložený ako slovník, kde kľúčom je **ID sektora** s hodnotou **názov sektora**.
7. Rovnako ako v bode 6 s jediným rozdielom, a to tým, že slovník má **kľúč ID sektora** a hodnotu **area_ID**.
8. Je vykonaný ďalší dopyt na databázu, v ktorom sa získavajú tie senzory, ktoré majú sector_ID (cudzí kľúč na prepojenie s tabuľkou Sector) rovnaké ako je ID sektora zo slovníka z bodu 6.
9. Dopyt vráti pole objektov triedy Sensor. Ak je pole prázdne, funkcia vráti správu: **Používateľ nemá pre zadaný sektor žiadne senzory a kód 200.**
10. V prípade, že má používateľ viac ako 0 senzorov, tak for cyklus prechádza pole objektov triedy Sensor. Z for cyklu sú vytiahnuté ID daných senzorov a sú uložené do poľa **sensor_ids**.

11. Je vykonaný ďalší dopyt na databázu, v ktorom sa získavajú typy meraných veličín, ktoré majú sensor_ID(cudzí kľúč na prepojenie s tabuľkou Sensor) rovnaké ako prvky poľa sensor_ids z kroku 10.
12. Posledným krokom je skonštruovať odpoveď klientovi. Na tento účel je vytvorené prázdne pole **data**, ktoré bude obsahovať všetky potrebné informácie. Na naplnenie tohto poľa je vytvorený for cyklus prechádzajúci pole objektov triedy Senzor z kroku 9, v ktorom je vytvorený slovník združujúci všetky informácie ako **: ID senzora, názov oblasti, názov sektora, názov senzora a pole typov meraných veličín daného senzora**. Za zmienku stojí získanie hodnoty pre názov oblasti a sektoru.

    Pri názve oblasti sa najskôr vytiahla hodnota predstavujúca area_ID zo slovníka z bodu 7, aby bola táto hodnota následne použitá na vytiahnutie hodnoty predstavujúcu názov oblasti zo slovníka z bodu 3.

    Pri názve sektora bola použitá slovníková metóda .get(), ktorá vyťahuje hodnotu z daného slovníka. Ako argument jej bol zadaný sector_id a vyťahoval sa podľa neho názov sektora zo slovníka z bodu 6.

    Kvôli zložitosti funkcie a jej lepšej názornosti bol zostrojený vývojový diagram. Nachádza sa na obrázku 4.5 :



Obrázok 4.5: Vývojový diagram pre funkciu get_all_sensors_for_user()

- **get_chart_data ()**

Popis: jedná sa o funkciu, ktorá má za úlohu na základe zvoleného rozsahu ,rozlíšenia, senzoru a typu meranej veličiny získať potrebné dáta na grafické zobrazenie.

Vstupné parametre:

-sensor_id (identifikátor senzoru),

-sensor_type_id (identifikátor typu meranej veličiny),

-interval (hodina, deň, týždeň , mesiac, rok),

-from (od),

-to (do).

Funkcionalita :

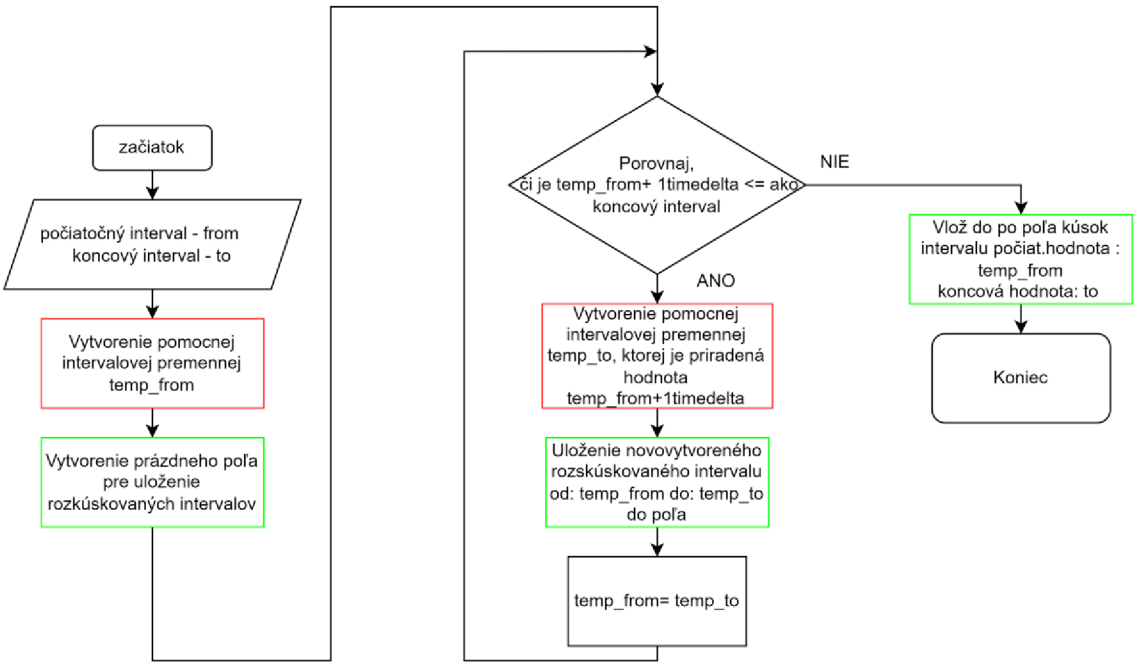
1. Vo funkcii je vytvorená konštanta obsahujúca skupinu slovníkov predstavujúcich intervaly.

Intervaly je možné zvoliť na hodinový, dňový, týždňový, mesačný a ročný. Hodnoty slovníkov sú dané funkciou timedelta, ktorou disponuje knižnica datetime.
Timedelta má za úlohu určiť podľa toho, aký si zvolíme interval, či sa bude pripočítavať hodina alebo deň, alebo týždeň atď. Príklad intervalu s timedeltou je zobrazený na zdrojovom kóde 4.5 :

```
'day': timedelta(days=1)
```

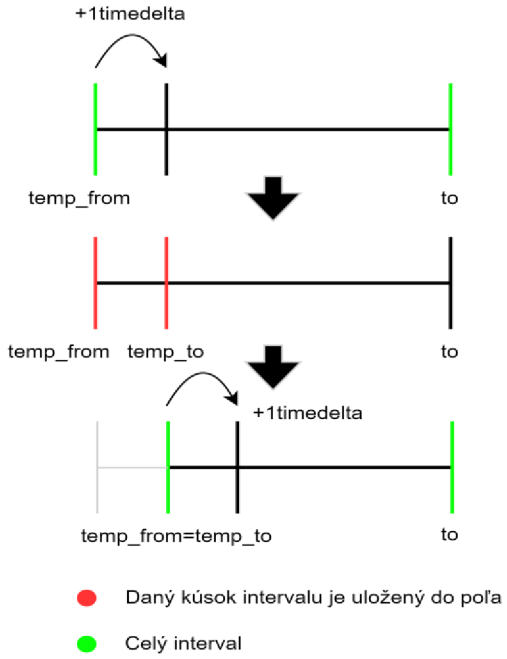
Zdrojový kód 4.5: Dňový interval s timedeltou 1

2. Zo vstupných dát sa zistí, či sa zadaný interval od klienta zhoduje z tými z ponuky servera. Ak nie, funkcia vypíše správu : **Zle špecifikovaný parameter intervalu a kód 400.**
3. Intervalové parametre from, to z požiadavky klienta sú uložené do premennej. Z parametra from je urobená pomocná premenná **temp_from**.
4. Ďalej bolo potrebné rozdeliť zvolený interval na menšie rovnaké diely. Na tento účel bol vytvorený algoritmus, ktorý porovnáva počiatočnú hodnotu intervalu s koncovou. Základom je smyčka, ktorá trvá dovtedy, než je počiatočná hodnota menšia ako koncová. V nej je porovnávané, či aj po pričítaní jednej timedelty daného intervalu k počiatočnej hodnote intervalu je menší či rovní koncovej hodnote. Pre daný algoritmus bol pre názornosť vytvorený vývojový diagram, ktorý je zobrazený na obrázku 4.6 :



Obrázok 4.6: Vývojový diagram pre rozkúskovanie zvoleného intervalu

Podľa vývojového diagramu vyzerá zmenšovanie intervalu nasledovne, ako na obrázku 4.7 :



Obrázok 4.7: Postupné zmenšovanie daného intervalu na jednotlivé kúsky

5. Výsledok z vývojového diagramu a obrázku č.14 je, že daný interval je ako pole po malých a rovnakých dielikoch uložený do premennej.
6. Z požiadavky od klienta sa uloží do premenných aký senzor a aký typ meranej veličiny sa bude zobrazovať.
7. Na databázu je zavolaný dopyt pre vybranie zadaného typu meranej veličiny, ak existuje. Ak nie, funkcia vráti správu : **Typ meranej veličiny pre daný senzor neexistuje a kód 200.**
8. Vybraný senzor je použitý v pomocnej triede **ChartObject,** ktorá má na starosť vrátiť klientovi dáta podľa výberu senzora a typu meranej veličiny ku grafickému zobrazeniu podľa zadania. Daná pomocná trieda je vidieť v zdrojovom kóde 4.6 :

```
class ChartObject():
  def __init__(self, sensor_type_id:int, note: str, unit: str, max, min):
        self.sensor_type_id = sensor_type_id
        self.note = note
        self.unit = unit
        self.max = max
        self.min = min
        self.values: List[ChartValue] = []
```

Zdrojový kód 4.6: DTO trieda ChartObject

9. V tomto momente chýba pomocnej triede už len zaplnenie hodnotami prislúchajúcimi k rozkúskovaným intervalom, ktoré sú uložené do poľa v kroku 5. K naplneniu bude slúžiť ďalšia pomocná trieda **ChartValue.**
10. K tomu bolo potrebné vo for cykle prechádzať dané pole rozkúskovaných intervalov a ku každému zistiť z dátovej tabuľky, ktoré dáta prislúchajú v danom intervale danému senzoru.
11. Keďže môže daný rozkúskovaný interval reprezentovať v grafe len jedna hodnota, bolo potrebné všetky získané dáta z daného intervalu spriemerovať. **Výsledná priemerná hodnota je vždy priradená počiatku rozkúskovaného intervalu.**
12. Výsledná priemerná hodnota, ako aj počiatočná hodnota rozkúskovaného intervalu sú uložené do premennej, ktorá je objektom pomocnej triedy **ChartValue**. Štruktúra triedy je zobrazená na zdrojovom kóde 4.7 :

```
class ChartValue():
    def __init__(self, average: float, date_label_unix: str,
date_label_dt ) -> None:
        self.average = average
        self.date_label_unix = date_label_unix
        self.date_label_dt = date_label_dt
        self.real_values: List[RealValue] = []
```

Zdrojový kód 4.7: DTO pomocná trieda ChartValue

Daná trieda vracia priemernú hodnotu k zobrazeniu na os x ako aj čas, ktorý bol k hodnote priradený na os y. Tento čas bol pre potreby overenia funkčnosti celej funkcie v dvoch formátoch. Jeden, ten ktorý bude slúžiť ako hodnota na os y, bol vo formáte unix timestampu. Jedná sa o počet sekúnd od roku 1970. Druhý formát bol v pythonovskom datetime formáte, a to z dôvodu čitateľnosti dátumu pre potreby overenia funkčnosti. [23]

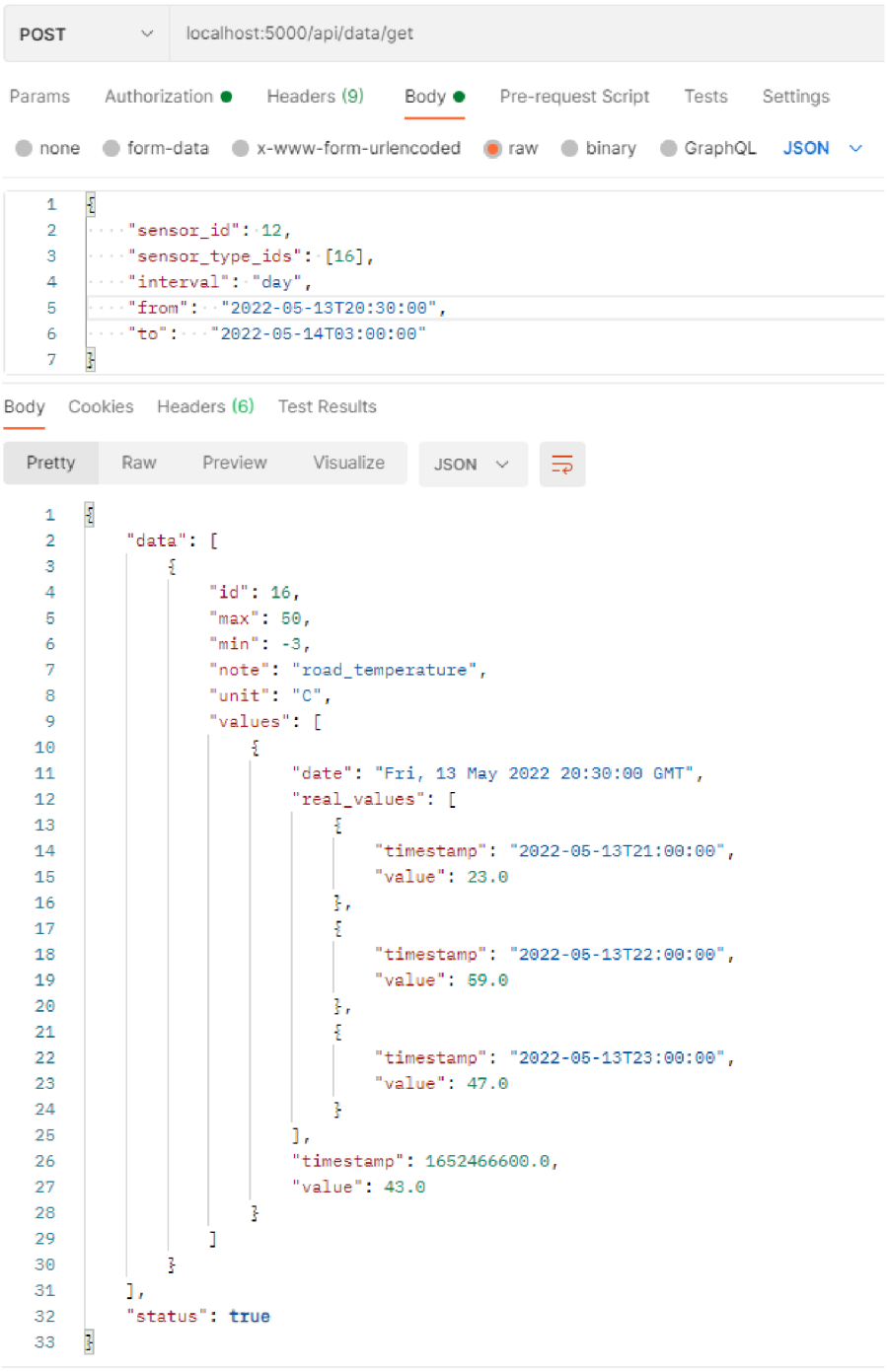
Keďže algoritmus je navrhnutý tak ,aby vypočítaná priemerná hodnota bola priradená práve k počiatočnej hodnote, bola pre dokázanie správnosti vytvorená funkcionalita, ktorá zobrazí reálny čas ako aj hodnotu namerania danej veličiny v tomto čase.

Formuláciu reálneho času a nameranej hodnoty v odpovedi klientovi má na starosť pomocná trieda RealValue, ktorá je zobrazená v zdrojovom kóde 4.8 :

```
class RealValue():
    def __init__(self, value: float, date_label: str) -> None:
        self.value = value
        self.date_label = date_label
```

Zdrojový kód 4.8: Pomocná trieda RealValue

Ukážka príkladu ,ktorý dokazuje funkčnosť riešenia je zobrazená na obrázku 4.8 :



Obrázok 4.8: Požiadavka a odpoveď overujúca funkčnosť riešenia

,kde bol stanovený interval na deň, rozsah od 13.5.2022 20:30 do 14.5.2022 03:00. Daný rozsah je menší ako jeden deň, takže nejde už deliť na menšie intervaly.

Tým pádom je vrátená len priemerná hodnota hlavného intervalu, ktorá sa vypočítala z hodnôt pri časoch, ktoré v ňom ležia. Takisto je vidieť, že daná priemerná hodnota je priradená k začiatku intervalu, presne tak ako, bol algoritmus navrhnutý.

- **delete_sensor()**

Popis: jedná sa o funkciu, ktorá vymazáva z databázy záznam o danom senzore.
Vstupné dáta:

- sensor_id (identifikátor senzora).

Funkcionalita:

1. Z požiadavky príde ID senzora, ktoré si uložíme do premennej.
2. Vytvorenie kontextového manažéra, v ktorom je vykonaný dopyt na databázu, kde pomocou metódy **.delete()** prichádza k odstráneniu daného senzora.

- **add_sensor()**

Popis: Jedná sa o funkciu, ktorá má na starosti pridanie nového senzora patriaceho konkrétnemu používateľovi do databázy.
Vstupné dáta:

- sector_id (identifikátor sektora)
- sensor_name (názov senzora)
- sensor_types(podrobnosti o type meranej veličiny)

Funkcionalita:

1. Nastáva uloženie vstupných parametrov do premenných.
2. Generovanie UID, čo je identifikátor, ktorý je unikátny pre každý senzor.
3. Vytvorenie objektu triedy Sensor obsahujúceho ID sektora v ktorom sa senzor nachádza, názov senzora a UID. Tento objekt je následne pridaný do databázy. Pri absencii hociktorej vstupnej veličiny sa pridanie do databázy nepodarí a funkcia vypíše správu : **Senzor {názov} nemôže byť pridaný a kód 200.**
4. Cyklické prechádzanie typov meraných veličín zo vstupných dát, ktoré sú ukladané do objektu triedy SensorType obsahujúci informácie ako rozsah, jednotka či názov meranej veličiny. Tento objekt je v cykle pridávaný do databázy.
5. V prípade úspešného vloženia senzora aj s jeho typmi je vrátená správa : **Senzor {nazóv} s jeho typmi {názov typov} bol úspešne pridaný a kód 201.**

- **add_data()**

Popis: Jedná sa o funkciu, ktorá má na starosti vkladanie dát do databázy.
Vstupné dáta:

- štruktúra požiadavky z kapitoly 4.1.1.

Funkcionalita:

1. Zo vstupných dát je uložený do premennej unikátny identifikátor senzora. Ak sa vo vstupných dátach nenachádza, funkcia vypíše správu : **Dáta nemôžu byť vložené, chýba unikátny identifikátor a kód 200.**
2. Je otvorená relácia s databázou, kde pomocou kontextového manažéra prichádza k dopytu na senzor, ktorý má rovnaké UID ako prišlo vo vstupných dátach. Výsledok dopytu je uložený do **objektu triedy Sensor**.
   Ak žiadny senzor neodpovedá zadaným kritériám, funkcia vypíše správu : **Nie je možné nájsť senzor pre zadané UID, dáta nie je možné vložiť a kód 200**.
3. Na databázu je volaný ďalší dopyt, konkrétne na type meranej veličiny daného senzora, u ktorého sa porovnáva získané ID z objektu triedy Sensor z bodu č.2. Výsledok je uložený do **objektu triedy SensorType**. Ak žiadny typ daný senzor nemá, funkcia vypíše správu : **Pre zadaný senzor nie je možné nájsť žiadny typ meranej veličiny. Dáta nemôžu byť vložené a kód 200**.
4. Keď už je známe, ku ktorému senzoru a typu meranej veličiny dáta patria, je pomocou metódy .add pristúpené k pridaniu dát do databázy.

4.3 Používateľské rozhranie aplikácie

Nasledujúca kapitola sa zaoberá implementáciou používateľského rozhrania. Na implementáciu bola použitá kombinácia knižnice React.js, ktorej popis sa nachádza v kapitole č.3, s typovým jazykom Typescript.
Pri vytváraní používateľského rozhrania boli použité aj balíčky ako **Material-ui** slúžiaci na použitie štandardizovaných komponentov, **ReactHookForm** za účelom vytvárania formulárov na použitých komponentoch. Tieto formuláre slúžia na spracovanie a odosielanie dát do serverovej časti. Na účely validácie formulárov bola použitá validačná knižnica **Yup** a jedným z najdôležitejších použitých balíčkov je **ReactRouter** slúžiaci na navigáciu naprieč aplikáciou.
Aplikácia sa skladá z viacerých komponentov, ktoré sú vytvorené genericky, teda tak ,aby sa daný komponent mohol používať viackrát pre rôzne časti aplikácie.

4.3.1 Komponenty

Ako komponent sa rozumie funkcia, ktorá na základe vstupných parametrov a vnútornej vybudovanej logiky vráti HTML štruktúru, ktorá je zobrazená v užívateľskom rozhraní.

Ako bolo spomenuté, komponenty boli vytvorené genericky, aby bolo rovnaký komponent možné použiť vo viacerých častiach aplikácie. Vďaka tomu prišlo k zvýšeniu prehľadnosti, čitateľnosti kódu a zlepšeniu orientácie v aplikácií, ako aj k ľahšiemu hľadaniu prípadných chýb.

V aplikácii boli použité komponenty ako **tabuľka, formulár na vyplnenie, výberový formulár** a ďalšie.

Ako demonštračný príklad na ukázanie štruktúry bol zvolený komponent formulára, ktorý získava textový vstup od používateľa. Nachádza sa na zdrojovom kóde 4.9 :

```
export function FormTextField({ name, label, sx, type }: Props) {
    const form = useFormContext()

    if (!form) {
        throw new Error('No form found. Please wrap component with form
provider.')
    }

 return (
     <Controller
         name={name}
         control={form.control}
         render={({ field }) => (
             <Box display="flex" flexDirection="column" mb={2}>
                 <TextField
                     {...field}
                     label={label}
                     sx={sx}
                     type={type}
                     variant="standard"
                     error={!!form.formState.errors[name]}
                     fullWidth
                     defaultValue={''}
                  />
                 <Typography>
                   {form.formState.errors[name]?.message ?? ''}
                 </Typography>
               </Box>
            )}
        />
    )
}
```

Zdrojový kód 4.9: Príklad generického komponentu

,kde v názve funkcie je vidieť, že sa jedná o komponent TextFormField, ktorý sa bude využívať v rámci formulárov. Vstupné parametre danej funkcie sa nazývajú Props a používajú sa na konfiguráciu komponentu. V tomto prípade sú to: ***name, label, sx, type***. Použitím typescriptu je nutné definovať každému vstupnému parametru typ, aby sa zaručilo priradenie hodnôt so správnymi typmi a nedochádzalo pri vývoji ku kompilačným chybám. Následne je použitý komponent Controller z balíčka ReakHookForm, ktorý poskytuje prepojenie vytvoreného komponentu s formulárom, pričom boli využité komponenty z balíčka Material-ui.

Celá spomenutá štruktúra je obalená komponentom Box, ktorý slúži na štylistiku. Ako komponent pre získanie používateľského vstupu sa používa komponent TextField, ktorý získa vlastnosti zadaním vstupných parametrov do komponentu FormTextField. Na rovnakom princípe sú vytvorené aj ďalšie komponenty použité v aplikácii.

4.3.2 Moduly

Nato, aby používateľ či admin mohol robiť operácie nad aplikáciou slúžia moduly.
Jedná sa o časť aplikácie, ktorá vymedzuje logický celok a má osobité užívateľské rozhranie. Moduly obsahujú komponenty, ktoré slúžia na získanie dát či zobrazenie výsledných dát.

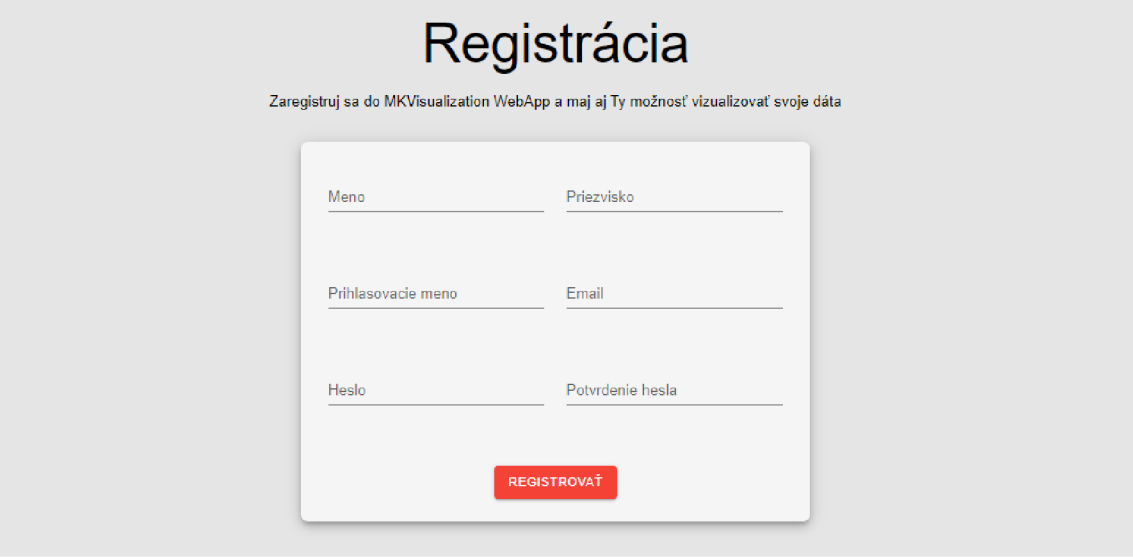
Okrem komponentov moduly obsahujú popísanú funkcionalitu k tomu, aby mohli komponenty správne fungovať.

Aplikácia je rozdelená na 6 modulov:

- **modul Registrácia**

V module registrácia sú opäť použité komponenty formulára na textový vstup slúžiace na združenie potrebných dát o používateľovi, ktorý sa chce zaregistrovať. Vybudovaná logika sa postará o to, aby sa používateľ, ktorý sa zaregistroval, objavil v tabuľke administrátorovi. Ten mu na základe toho bude môcť potvrdiť registráciu a umožniť mu tak prístup do aplikácie.

Ukážka modulu registrácie sa nachádza na obrázku 4.9 :



Obrázok 4.9: Modul registrácia

- **modul Prihlásenie**

V module prihlásenie sa nachádzajú komponenty formulára na textový vstup, ktoré slúžia na vyplnenie prihlasovacích údajov.

Vybudovaná logika po úspešnom prihlásení presmeruje používateľa do jadra aplikácie.

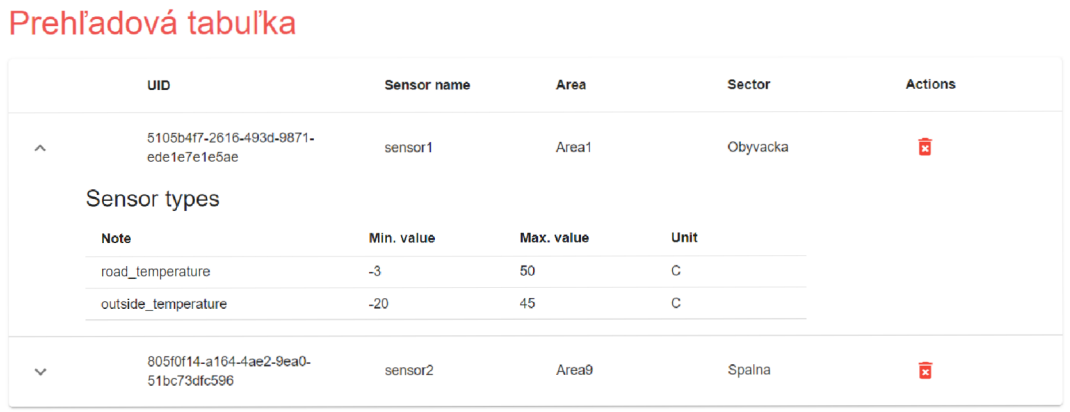
Ukážka modulu prihlásenie je na obrázku 4.10 :

Obrázok 4.10: Modul prihlásenie

- **modul Hlavná stránka**

V module hlavná stránka sa nachádza komponent tabuľky, ktorý má za úlohu zobrazovať prehľadové informácie o počte a vlastnostiach senzorov jednotlivého používateľa. Takisto tu používateľ má možnosť daný senzor zmazať.

Ukážka modulu hlavnej stránky sa nachádza na obrázku 4.11 :



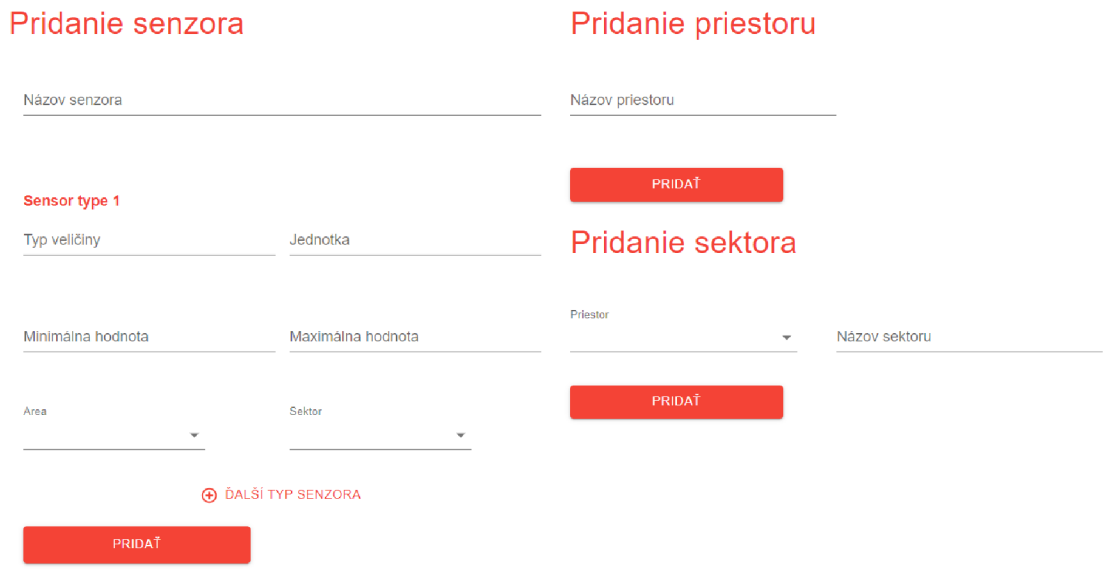
Obrázok 4.11: Modul hlavnej stránky

- **modul Konfigurácia**

V module konfigurácia sa nachádzajú tri logické časti, ktoré zo štrukturálneho hľadiska obsahujú **formuláre,** či už s textovým vstupnom alebo s výberom. Sú to:

1. Pridanie senzora: ak chce používateľ pridať nový senzor, v tejto časti vyplní všetky informácie týkajúce sa názvu nového senzora a informácie týkajúce sa meraných veličín potrebných pre pridanie.
2. Pridanie oblasti: ak chce používateľ pridať novú oblat, v tejto časti vyplní názov a potvrdí pridanie.
3. Pridanie sektora: ak chce používateľ pridať nový sektor, v tejto časti vyplní názov a priradí sektoru danú oblasť, v ktorej sa bude senzor nachádzať a potvrdí pridanie.

Ukážka modulu Konfigurácia je na obrázku 4.12 :



Obrázok 4.12: Modul konfigurácia

- **modul Grafy**

V module grafy sa nachádzajú formulárové komponenty na výber jednotlivého typu meranej veličiny, periodicity a dátumového rozsahu. Nižšie sa nachádza komponent grafu, ktorý na základe používateľom vybraných parametrov vykreslí namerané dáta patriace konkrétnemu senzoru a jeho meranej veličine.

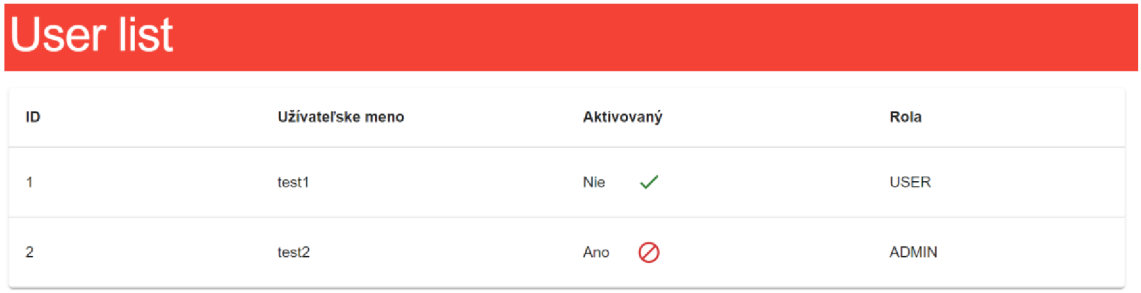
Ukážka modulu Grafy je na obrázku 4.13 :

Obrázok 4.13: Modul grafy

- **modul Admin**

V module admin sa nachádza komponent tabuľky, ktorá slúži ako prehľadová tabuľka všetkých používateľov, ktorí boli zaregistrovaní. Hlavná funkcionalita je však potvrdzovanie registrácie či deaktivácia účtu podľa stĺpca Aktivovaný. Ak by administrátor úspešne zmenil stav aktívnosti účtu daného používateľa, serverová logika túto zmenu zaregistruje a odošle používateľovi email, na základe ktorého bude používateľ informovaný buď o zmene stavu jeho účtu.

Ukážka modulu Admin je na obrázku 4.14 :



Obrázok 4.14: Modul admin

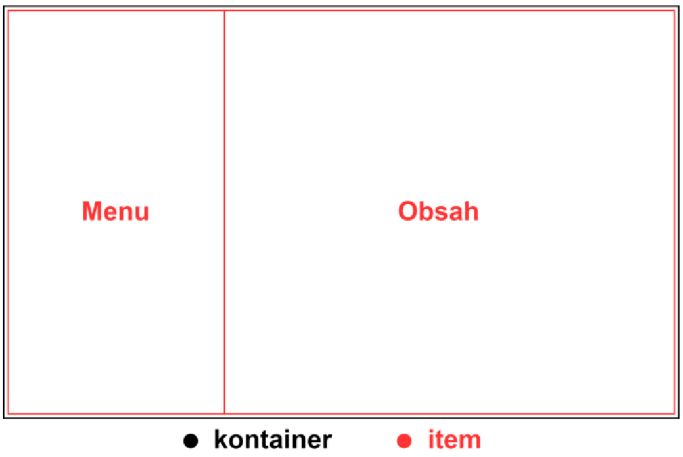
4.3.3 Rozloženie obrazovky

Pri tvorbe rozloženia obrazovky boli použité komponenty **Grid** a **Box**.

Grid je komponent, ktorý môže mať 2 stavy. Buď môže slúžiť ako **kontajner,** ktorý obaľuje obsah, alebo môže byť použitý ako **item**, teda prvok obalený kontajnerom. Vďaka gridu je možné rozdeliť obrazovku na 12 dielov, kde každému itemu je možné priradiť pomernú časť z celkového počtu dielov. Pri tvorbe základného layoutu sa rozdelila obrazovka na 2 itemy: jeden slúži pre navigačný panel- **MENU** a má šírku 3/12 dielov a druhý slúžiaci na **zobrazenie obsahu** jednotlivých položiek z menu pozostáva zo zvyšných 9/12 dielikov.

Do itemov je ďalej možné vnoriť ďalší kontajner, ktorý bude mať zase svoje itemy. Takýmto spôsobom sa vytváralo rozloženie v jednotlivých moduloch. Obrazovka bola tak rozdelená na viac sekcií, pričom tak vznikol prehľadný a účelný dizajn.

Základné rozdelenie je možné vidieť na obrázku 4.15 :



Obrázok 4.15: Základné rozloženie používateľského rozhrania

Box je komponent, slúžiaci ako obaľovač obsahu. Jeho význam spočíva v tom, že nie je potrebné, aby sa rovnaká štylizácia používala na každý komponent zvlášť. Komponenty sú obalené do Boxu a tomu sa daná štylizácia uplatní, tým pádom všetky jeho komponenty túto štylizáciu dedia. Takisto je vďaka boxu možné manipulovať s rozložením komponentov, ktoré box obaľuje.

4.3.4 Navigácia

Webové aplikácie vytvorené pomocou React.js fungujú na princípe SPA(Single Page Application. Čo znamená, že na server sa pošle požiadavka, ktorá vráti HTML stránku spolu so skompilovaným typescript kódom, ktorý kontroluje React aplikáciu. HTML stránka je pôvodne prázdna. Jej obsah React vloží dynamicky podľa top-level komponentov, ktoré sú nato vytvorené. Pri navigácii sa ďalej neposielajú ďalšie požiadavky na server. Ďalej navigáciu preberá balíček **React-router**.

Využitie react routeru v práci je možné vidieť na zdrojovom kóde 4.10 :

```
function App() {
    return (
        <BrowserRouter>
            <UserContextProvider>
               <Routes>
                  <Route path={ROUTES.signIn} element={<SignIn />} />
                  <Route path={ROUTES.signUp} element={<SignUpForm />} />

                  <Route path={ROUTES.admin} element={<Admin />} />

                  <Route path={ROUTES.home} element={<Layout />}>
                    <Route index element={<Homepage />} />
                    <Route element={<Charts />} path={ROUTES.charts} />
                    <Route element={<Config />} path={ROUTES.config} />
                  </Route>

                  <Route
                     element={<>200 | Stranka neexistuje </>}
                     path={ROUTES[200]}
                  />
               </Routes>
            </UserContextProvider>
        </BrowserRouter>
```

Zdrojový kód 4.10: Štruktúra React routera

, kde je vidieť že na najvyššej úrovni sa nachádza komponent **BrowserRouter**, ktorý slúži na obalenie routovaného obsahu. O úroveň nižšie sa nachádza komponent **Routes**, ktorý obaľuje jednotlivé trasy (**Route**). Ako je vidieť trasy sa môžu navzájom vnoriť. Trasa ROUTES.home obsahuje len " **/dashboard**" a nazýva sa parent. Všetky vnorené trasy dedia z parentu "**/dashboard**" a za neho pridávajú názov komponentu, ktorý budú zobrazovať. Buď **Homepage, Charts** alebo **Config.** Prvá vnorená trasa nemá cestu ale má atribút index. Ten určuje túto trasu ako predvolenú trasu pre trasu parent. Vnorenie trás je urobené z dôvodu, že tieto trasy majú jednotné rozloženie obrazovky a treba ich preto logicky oddeliť. Tieto vnorené trasy sú určené len pre bežných používateľov. Preto v nich nastáva overenie, či sa na tieto trasy nesnaží prejsť administrátor.

Ďalej sa tu nachádzajú trasy na prihlásenie a registráciu, ktoré sú prístupné ako používateľovi s rolou user tak aj používateľovi s rolou admin. Prihlasovacia obrazovka musí byť prístupná pre obe role, a to preto, aby mal možnosť prihlásenia sa do aplikácia ako bežný používateľ, tak aj administrátor, pričom podľa role im bude zobrazený povolený obsah.

Takisto sa tu nachádza aj trasa, ktorá je prístupná len administrátorovi na ceste ROUTES.admin, kde sa nachádza tabuľka používateľov, kde administrátor môže pridať alebo odobrať používateľovi prístup do aplikácie.

4.3.5 Formuláre

Aplikácia je z veľkej časti tvorená práve formulármi. Prostredníctvom formulárov sa získavajú dáta, ktoré sa následne posielajú na server, kde sú vytvorené API, ktoré na základe prijatých dát vykonávajú rôzne funkcionality. Pri vytváraní formulárov bol použitý balíček **ReactHookForm.**

ReactHookForm používa na reprezentáciu formulárov vlastné komponenty. Dovoľuje však použiť formuláre aj na komponentoch z externých knižníc, ako je Material-ui, z ktorej boli v práci využité takmer všetky komponenty.

Nato, aby mohli vo formulároch použité komponenty z externých knižníc slúži v ReactHookForme komponent **Controller.** Tento komponent obaľuje komponent z externej knižnice.

Obsahuje argumenty ako:

- **name** : popisujúci unikátny názov pre daný komponent,
- **control** : registruje daný komponent do funkcie useForm.

Základnou funkciou, ktorú musí formulár obsahovať je funkcia **useForm**. Jedná sa o funkciu z API ReactHookFormu. Táto funkcia obsahuje metódy, ktoré sú potrebné k správnej funkčnosti formulára. Jadrom je metóda register. Táto metóda slúži na zaregistrovanie komponentu, ktorého cieľom je fungovať ako formulár, do funkcie useForm. Vďaka registru je možné robiť operácie nad formulárom ako validácia či odoslanie. Pri práci s externými komponentovými knižnicami je registrovanie komponentu v réžii spomínaného Controllera, kde na registráciu slúži metóda **control.**

Na odosielanie dát slúži metóda **handleSubmit**, ktorá v argumente obsahuje potvrdzovaciu funkciu, ktorá je napísaná používateľom.

V práci je formulár použitý vo viacerých formách. Prvou formou je textový/číselný vstup. Túto formu využívajú napríklad komponenty na pridávanie sektora či senzora. Druhou formou je výber možností, ktorý sa uplatňuje napríklad pri pridávaní sektora, kde je potrebné vybrať danú oblasť, v ktorej sa sektor bude nachádzať. Treťou formou je výber dátumu, ktorý sa nachádza pri zvolení dátumového rozsahu zobrazenia nameraných dát.

Príklad formulára je vidieť v sekcii 4.3.1 na zdrojovom kóde č.17.

4.3.6 Validácia formulárov

Úlohou formulárov je sprostredkovať dáta v takej štruktúre, na akú je pripravená serverová časť. Na to, aby bolo zaručené, že príde na server predpokladaná štruktúra, sa používa validácia. Vo validácii sa kontroluje, či je zadaný správny dátový typ údajov.

Takisto je možné kontrolovať tvar zadaného dátového typu. Špecifickým príkladom je email, u ktorého je potrebné, aby bol zadaný v určitom tvare.

Na validáciu formulárov bol použitý balíček **yup**. Tento balíček umožňuje vytváranie validačných schém. Príklad validačnej schémy je ukázaný na zdrojovom kóde 4.11 :

```
const schema = yup.object({
    username: yup.string().required('Používateľské meno je povinné'),
    password: yup.string().min(12, 'Min 12 znakov').required('Heslo je
povinné'),
})
```

Zdrojový kód 4.11: Validačná schéma prihlásenia

, kde má validačná schéma tvar yup objektu. Obsah objektu tvoria validačné pravidlá pre jednotlivé atribúty formulára. Formát validačného pravidla je **{kľúč:hodnota}**, pričom kľúčom je vždy názov atribútu, ktorému sa stanovujú validačné pravidlá a hodnotou je typ atribútu (string,number...) a dané validačné pravidlo.

Je vidieť, že všetky atribúty sú zvolené ako povinné, nakoľko sa očakávajú v požiadavke klienta na server.

ReactHookForm podporuje validácie pomocou externých knižníc vďaka funkcii **resolver**. Tá je vložená ako argument do základnej funkcie useForm. Resolver obsahuje veľa typov externých validačných knižníc, preto je mu daný typ **yupResolver**, ktorý sa používa pre validačnú knižnicu yup. Ako argument sa yupResolveru vloží daná schéma zo zdrojového kódu č. 19.

Ukážka implementácie validácie do balíčka ReactHookForm je na zdrojovom kóde 4.12 :

```
const form = useForm<SignUpValues>({
     resolver: yupResolver(schema)
 })
```

Zdrojový kód 4.12: Implementácia validácie

5. TESTOVANIE

Táto kapitola je venovaná testovaniu aplikácie, ktoré spočívalo v overení základných funkcionalít systému. Počas celého procesu vývoja aplikácie boli používané konzolové výpisy, ktoré slúžili na sledovanie vykonávaných krokov aplikácie. Vďaka ich použitiu bolo možné odstrániť logické chyby a výrazne optimalizovať jednotlivé funkcionality aplikácie. Vo finálnej verzii aplikácie boli všetky konzolové výpisy odstránené.

Testovací scenár č.1:

- **Akcia** – pokus o registráciu nového používateľa
- **Vstup** – správne vyplnené formuláre s textovým vstupom
- **Výstup** – aplikácia uložila do databázy nového používateľa a pridala mu status neaktívny. Následne je nový používateľ zobrazený u administrátora v prehľadovej tabuľke používateľov, kde mu môže administrátor aktivovať účet.

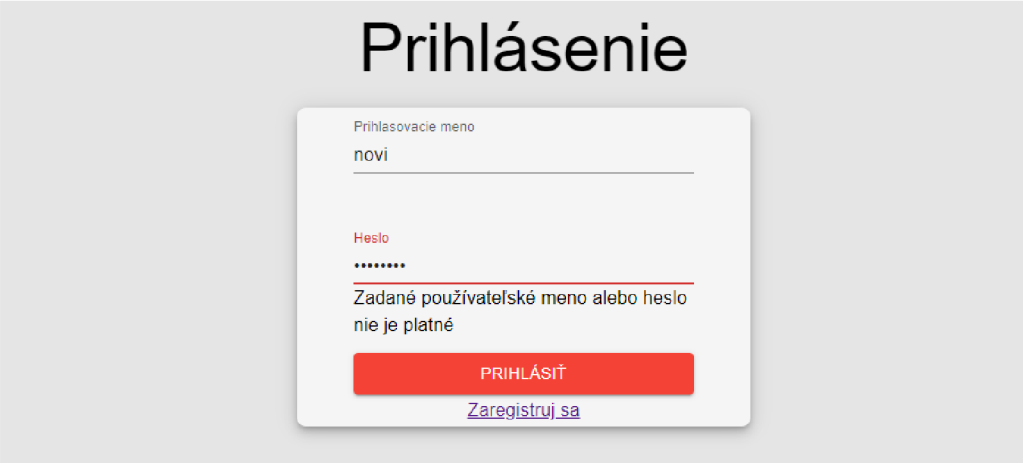
Testovací scenár č.2:

- **Akcia** – pokus o registráciu nového používateľa
- **Vstup** – nesprávne vyplnené formuláre
- **Výstup** – aplikácia neumožní používateľovi potvrdiť registráciu. Pri zle vyplnených formulároch systém zobrazí informačné hlášky ako na obrázku 5.1 :

Obrázok 5.1: Nesprávny pokus o registráciu

Testovací scenár č.3:

- **Akcia** – pokus o prihlásenie pomocou zlých prihlasovacích údajov.
- **Vstup** – vyplnené formuláre s textovým vstupom.
- **Výstup** – používateľovi bol odmietnutý prístup do aplikácie a bola mu zobrazená hláška ako na obrázku 5.2:



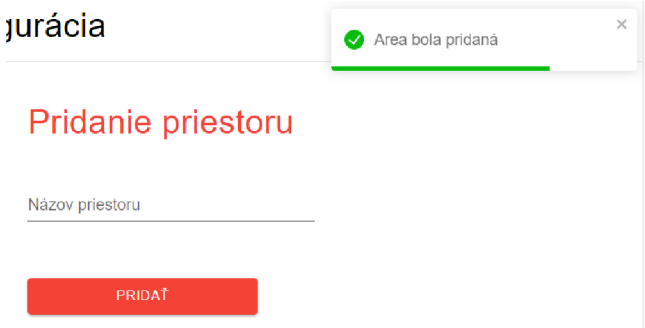
Obrázok 5.2: Nesprávny pokus o prihlásenie

Testovací scenár č.4:

- **Akcia** – pokus o prihlásenie so správnymi údajmi.
- **Vstup** – vyplnené formuláre s textovým vstupom.
- **Výstup** – používateľ bol úspešne prihlásený a presmerovaný do aplikácie.

Testovací scenár č.5:

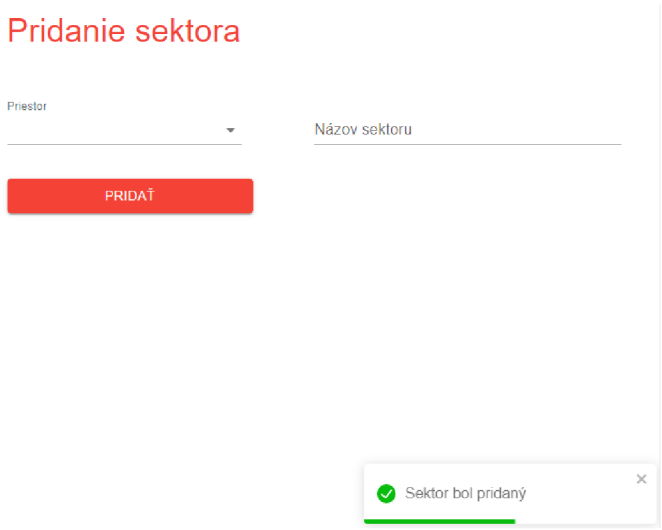
- **Akcia** – pridanie oblasti cez formulár v module Konfigurácia.
- **Vstup** – vyplnený formulár s textovým vstupom.
- **Výstup** – oblasť je úspešne pridaná do databázy , formulár je po pridaní vyčistený a používateľ je o úspešnom pridaní oboznámený správou, aká sa nachádza na obrázku 5.3 :



Obrázok 5.3: Správny výsledok pridania oblasti

Testovací scenár č.6:

- **Akcia** – pridanie sektora cez formulár v module Konfigurácia.
- **Vstup** – vyplnené formuláre s textovým vstupom a s výberom.
- **Výstup** – sektor je úspešne pridaný do databázy , formulár je po pridaní vyčistený a používateľ je o úspešnom pridaní oboznámený správou, aká sa nachádza na obrázku 5.4 :



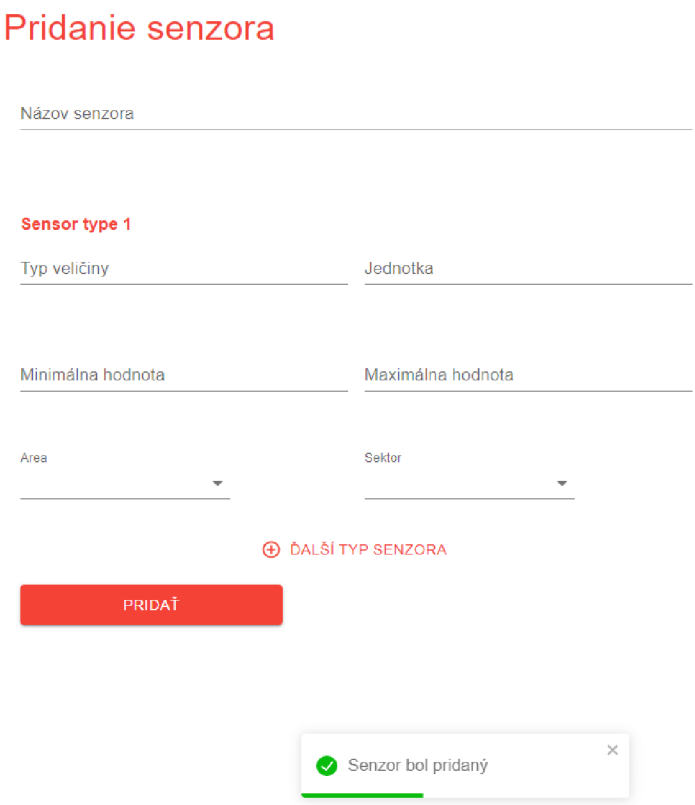
Obrázok 5.4 Správny výsledok pridania sektora

Testovací scenár č.7:

- **Akcia** – zmena vyplnených formulárov v module Grafy.
- **Vstup** – vyplnené formuláre s výberom.
- **Výstup** – po zmene obsahu jedného formulára z nasledujúcich štyroch, ktoré sú v tomto poradí : výber oblasti, výber sektoru, výber senzoru, výber typu meranej veličiny , sa všetky nasledujúce formuláre premažú a ponúkajú nové možnosti patriace novému obsahu daného formulára.

Testovací scenár č.8:

- **Akcia** – pridanie senzora cez formulár v module Konfigurácia.
- **Vstup** – vyplnené formuláre s textovým vstupom a s výberom.
- **Výstup** – senzor je úspešne pridaný do databázy , formulár je po pridaní vyčistený a používateľ je o úspešnom pridaní oboznámený správou, aká sa nachádza na obrázku č. :



Obrázok 5.5 Správny výsledok pridania senzora

Testovací scenár č.9:

- **Akcia** – zmena stavu aktivity účtu vykonaná administrátorom.
- **Vstup** –stĺpec Aktivovaný v prehľadovej tabuľke používateľov, ktorý ponúka akciu aktivácie či deaktivácie účtu.
- **Výstup** – pri stlačení akčného tlačidla pri používateľovi s neaktívnym účtom mu je účet aktivovaný a používateľ je o tom oboznámený emailom, ktorý je na obrázku 4.3. Pri používateľovi s aktívnym účtom mu je po kliknutí na akčné tlačidlo účet deaktivovaní. Používateľ je o tejto skutočnosti opäť upozornený emailom, ktorý sa nachádza na obrázku 4.4.

ZÁVER

Cieľom diplomovej práce bolo navrhnutie a následná implementácia webovej aplikácie slúžiacej ako Univerzálny dátový sklad pre IoT aplikácie pre využitie na interné účely vedúceho práce a jeho budúcich študentov.

Návrh serverovej časti spočíval vo výbere technológií, ktoré budú použité na vytvorenie funkčnej stránky aplikácie. Jednalo sa o výber programovacieho jazyka Python a knižnice SQLAlchemy, vďaka ktorej dokáže Python komunikovať s databázou a vykonávať nad ňou operácie v podobe dopytov. Ďalej sa v tejto časti nachádza popis použitého komunikačného modelu medzi klientom a serverom v podobe REST API, ktorý podporuje štandardné metódy komunikačného protokolu HTTP ako GET,POST, DELETE.

Pri návrhu databázovej štruktúry bola použitá relačná databáza MySQL, ktorá slúži ako úložisko dát. Databáza bola štrukturalizovaná do viacerých tabuliek, ktoré na seba navzájom nadväzujú a vytvárajú potrebnú štruktúru pre potrebu dopytov zo serverovej časti. Táto štruktúra je v práci reprezentovaná EER diagramom, ktorý zobrazuje obsah jednotlivých tabuliek a popisuje vzájomné prepojenia týchto tabuliek.

V ďalšej časti sa nachádza súhrn požiadaviek na aplikáciu, ktoré určovali aké funkcionality bude aplikácia obsahovať. Tieto požiadavky vymedzovali aktivity spojené s prihlasovaním, zobrazovaním dát a či senzorovou konfiguráciou. Na základe týchto požiadaviek bolo navrhnuté používateľské rozhranie, na ktoré bol použitý dizajnérsky nástroj Figma. V tomto nástroji bol zostrojený návrh jednotlivých obrazoviek používateľského rozhrania slúžiacich na reprezentáciu aktivít vyplývajúcich z požiadaviek na aplikáciu.

Realizáciou celkového návrhu bola následná implementácia, ktorú je možné rozdeliť na serverovú a klientskú časť aplikácie. Implementácia serverovej časti popisuje navrhnutý štandardizovaný formát prijímaných správ. Tento formát je využitý na prenos nameraných dát medzi rôznymi hardvérovými modulmi tretích strán a serverom. Serverová logika prijaté dáta následne uloží do databázy. V tejto časti sa ďalej nachádza popis jednotlivých funkcií a algoritmov, ktoré majú za úlohu plniť požiadavky stanovené v návrhu. Ku zložitejším funkciám boli zostrojené vývojové diagramy, ktoré umožňujú lepšiu grafickú názornosť popisovaných funkcionalít. Implementácia používateľského rozhrania, ktorá je vytvorená v technológii React.js, sa zaoberá vysvetlením základných stavebných častí aplikácie ako sú komponenty a ich validácia, logicky oddelené časti používateľského rozhrania tzv. moduly a navigácia naprieč aplikáciou.

Záver práce je venovaný testovaniu, ktoré na základe niekoľkých testovacích scenárov overuje správne fungovanie aplikácie.

Zoznam použitej literatúry

[1] HERKO, Lubomír. Learn2Code: Frontend vs Backend: v čom je rozdiel? [online]. 28.5.2017 [cit. 2022-01-03]. Dostupné z: https://www.learn2code.sk/blog/front-end-vs-back-end

[2] GILLIS, Alexander S. Object-oriented programming (OOP): What is object-oriented programming?. TechTarget [online]. [cit. 2022-05-23]. Dostupné z: https://www.techtarget.com/searchapparchitecture/definition/object-oriented-programming-OOP

[3] What is Python? Executive Summary [online]. [cit. 2022-05-23]. Dostupné z: https://www.python.org/doc/essays/blurb/

[4] What is Flask Python. Python Tutorial [online]. [cit. 2022-01-03]. Dostupné z: https://pythonbasics.org/what-is-flask-python/

[5] Introduction. WSGI [online]. [cit. 2022-01-03]. Dostupné z: https://wsgi.tutorial.codepoint.net/

[6] Data Definition Language (DDL). Technopedia [online]. 21.9.2020 [cit. 2022-01-03]. Dostupné z: https://www.techopedia.com/definition/1175/data-definition-language-ddl

[7] SQLAlchemy [online]. [cit. 2022-01-03]. Dostupné z: https://www.sqlalchemy.org/

[8] REST APIs. IBM: cloud [online]. [cit. 2022-05-23]. Dostupné z: https://www.ibm.com/cloud/learn/rest-apis

[9] HANÁK, Drahomír. Stopařův průvodce REST API [online]. [cit. 2022-05-23]. Dostupné z: https://www.itnetwork.cz/programovani/nezarazene/stoparuv-pruvodce-rest-api/

[10] BALOGH, P. Webová aplikace HelpDesk a synchronizace dat. Brno: Vysoké učení technické v Brně, Fakulta strojního inţenýrství, 2012. 79 s. Vedoucí diplomové práce Prof. RNDr. Ing. Jiří Šťastný, CSc

[11] CHAPLE, Mike. Čo je primárny kľúč. Eyewanted [online]. [cit. 2022-01-03]. Dostupné z: https://sk.eyewated.com/co-je-primarny-kluc/

[12] Three Types of Relationships in an ERD Diagram. RelationalDBDesign [online]. [cit. 2022-01-03]. Dostupné z: https://www.relationaldbdesign.com/database-design/module6/three-relationship-types.php

[13] Entitno-relačný model. Truni [online]. [cit. 2022-01-03]. Dostupné z: https://pdf.truni.sk/e-ucebnice/databazove-systemy1/data/b68b3315-4936-4edd-8e84-f66a4b6fdd2a.html?ownapi=1

[14] Get started with JSON Web Tokens. Auth0 [online]. [cit. 2022-01-03]. Dostupné z: https://auth0.com/learn/json-web-tokens/

[15] KOĎOUSKOVÁ, Barbora. HTTPS V KOSTCE: CO TO JE, JAK FUNGUJE A JAK NA NĚJ PŘEJÍT. Rascasone [online]. 17.11.2021 [cit. 2022-01-03]. Dostupné z: https://www.rascasone.com/cs/blog/co-je-https-http-ssl-tls

[16] FEREK, M. Bezpečená komunikace mezi data loggerem a databazovým serverem. Brno: Vysoké učení technické v Brně Fakulta elektrotechniky a komunikačních technologií, 2011. 81 s. Vedoucí diplomové práce Ing. Radomír Svoboda, Ph.D..

[17] HEATON, Robert. How does actually HTTPS works? Robertheaton [online]. 27.3.2014 [cit. 2022-01-03]. Dostupné z: https://robertheaton.com/2014/03/27/how-does-https-actually-work/

[18] SLL.com: Free SSL [online]. [cit. 2022-01-03]. Dostupné z: https://www.ssl.com/certificates/free/

[19] ZeroSSL [online]. [cit. 2022-01-03]. Dostupné z: https://zerossl.com/

[20] BRACEY, Kezz. What Is Figma?: Figma. Envatotuts+ [online]. 26.10.2018 [cit. 2022-05-23]. Dostupné z: https://webdesign.tutsplus.com/articles/what-is-figma--cms-32272

[21] Introducing JSX. React [online]. [cit. 2022-05-23]. Dostupné z: https://reactjs.org/docs/introducing-jsx.html

[22] MÁCA, Jindřich. Úvod do React. ITnetwork [online]. [cit. 2022-05-23]. Dostupné z: https://www.itnetwork.cz/javascript/react/zaklady/uvod-do-react/

[23] The Current Epoch Unix Timestamp. Dan's Tools [online]. [cit. 2022-05-23]. Dostupné z: https://www.unixtimestamp.com/

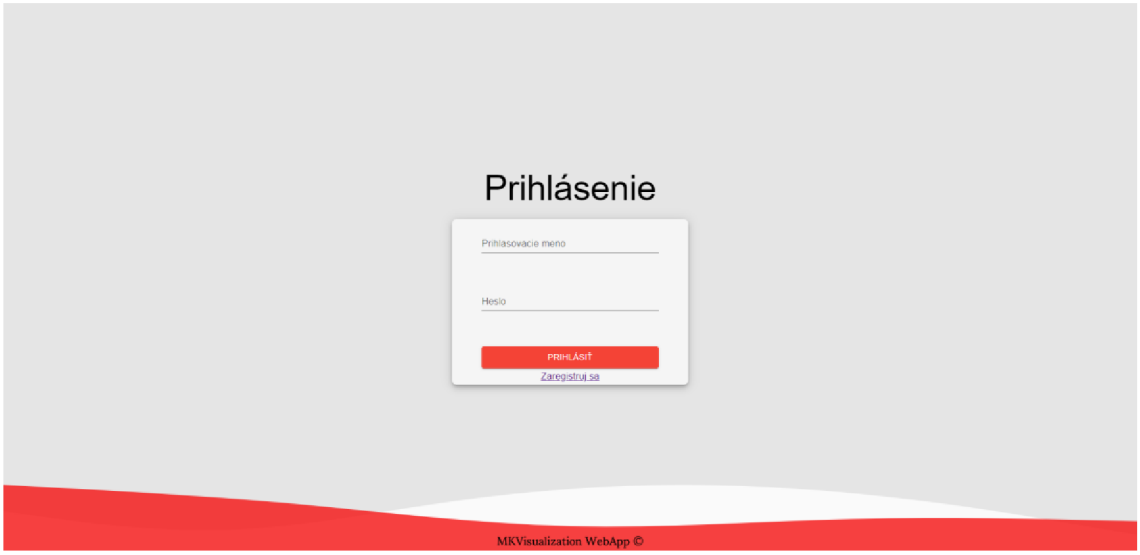
ZOZNAM SKRATIEK

API	Application programming interface.
DDL	Data Definition Language
DTO	Data Transfer Object
HTTP	Hypertext Transfer Protocol
ID	Identificator

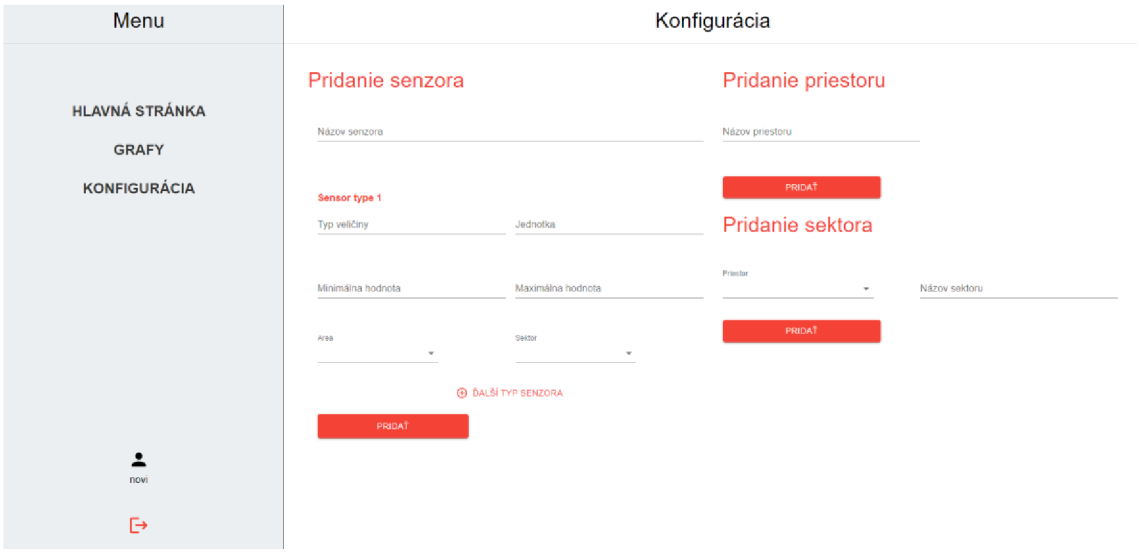
Příloha A - Ukážka používateľského rozhrania

A.1 Hlavná stránka

A.2 Prihlasovacia stránka



A.3 Konfiguračná stránka



A.4 Stránka pre grafy