TECHNICKÁ UNIVERZITA V LIBERCI
Fakulta mechatroniky, informatiky
a mezioborových studií

Zpracování a vizualizace dat z vodárenského přivaděče Bedřichov

Diplomová práce

Studijní program: N2612 Elektrotechnika a informatika
Studijní obor: Informační technologie

Autor práce: **Bc. Lukáš Pelc**
Vedoucí práce: Ing. Jan Kolaja, Ph.D.
Ústav nových technologií a aplikované informatiky

Liberec 2021

TECHNICKÁ UNIVERZITA V LIBERCI
Fakulta mechatroniky, informatiky
a mezioborových studií

Zadání diplomové práce

Zpracování a vizualizace dat z vodárenského přivaděče Bedřichov

Jméno a příjmení: **Bc. Lukáš Pelc**
Osobní číslo: M17000136
Studijní program: N2612 Elektrotechnika a informatika
Studijní obor: Informační technologie
Zadávající katedra: Ústav nových technologií a aplikované informatiky
Akademický rok: **2020/2021**

Zásady pro vypracování:

Upravte stávající stav softwarového řešení sběru, ukládání a grafické prezentace dat z vodárenského přivaděče Bedřichov. Práce by měla splňovat následující body:

1. Analyzujte současný stav měření fyzikálních jevů v přivaděči Bedřichov včetně přenosu a zpracování dat.
2. Vytvořte nástroj pro grafickou prezentaci naměřených dat nezávisle na použitém databázovém systému.
3. Proveďte rešerši, vyberte a instalujte vhodný databázový systém pro ukládání naměřených dat paralelně se stávajícím.
4. Upravte systém tak, aby mohl běžet v ověřovacím režimu, paralelně se stávajícím.
5. Proveďte vyhodnocení zpracování dat stávavajícím a novým systémem.

Rozsah grafických prací: dle potřeby dokumentace
Rozsah pracovní zprávy: 40 – 50 stran
Forma zpracování práce: tištěná/elektronická
Jazyk práce: Čeština

Seznam odborné literatury:

[1] ŠÍR, Karel. *Aplikace pro automatické zpracování veřejně dostupných meteorologických dat do databáze: Application for automatic processing of public available meteorological data into a database*. Liberec: Technická univerzita v Liberci, 2016. Bakalářské práce. Technická univerzita v Liberci.
[2] BĚLOCH, Michal. *Automatizace tvorby grafů v programu Grapher využívající rozhraní k databázi: Automating of graphs creation in Grapher program using the interface to the database*. Liberec: Technická univerzita v Liberci, 2012. Bakalářské práce. Technická univerzita v Liberci.
[3] HOKR, Milan. *Aplikace numerických metod v úlohách transportu látek v podzemních vodách*. Liberec: Technická univerzita v Liberci, 2006. Habilitační práce.

Vedoucí práce: Ing. Jan Kolaja, Ph.D.
Ústav nových technologií a aplikované informatiky

Datum zadání práce: 19. října 2020
Předpokládaný termín odevzdání: 17. května 2021

L.S.

prof. Ing. Zdeněk Plíva, Ph.D.
děkan

Ing. Josef Novák, Ph.D.
vedoucí ústavu

V Liberci dne 19. října 2020

Prohlášení

Prohlašuji, že svou diplomovou práci jsem vypracoval samostatně jako původní dílo s použitím uvedené literatury a na základě konzultací s vedoucím mé diplomové práce a konzultantem.

Jsem si vědom toho, že na mou diplomovou práci se plně vztahuje zákon č. 121/2000 Sb., o právu autorském, zejména § 60 – školní dílo.

Beru na vědomí, že Technická univerzita v Liberci nezasahuje do mých autorských práv užitím mé diplomové práce pro vnitřní potřebu Technické univerzity v Liberci.

Užiji-li diplomovou práci nebo poskytnu-li licenci k jejímu využití, jsem si vědom povinnosti informovat o této skutečnosti Technickou univerzitu v Liberci; v tomto případě má Technická univerzita v Liberci právo ode mne požadovat úhradu nákladů, které vynaložila na vytvoření díla, až do jejich skutečné výše.

Současně čestně prohlašuji, že text elektronické podoby práce vložený do IS/STAG se shoduje s textem tištěné podoby práce.

Beru na vědomí, že má diplomová práce bude zveřejněna Technickou univerzitou v Liberci v souladu s § 47b zákona č. 111/1998 Sb., o vysokých školách a o změně a doplnění dalších zákonů (zákon o vysokých školách), ve znění pozdějších předpisů.

Jsem si vědom následků, které podle zákona o vysokých školách mohou vyplývat z porušení tohoto prohlášení.

17. května 2021 Bc. Lukáš Pelc

Zpracování a vizualizace dat z vodárenského přivaděče Bedřichov

Abstrakt

Diplomová práce popisuje návrh grafické reprezentace dat z vodárenského přivaděče Bedřichov. Cílem je rychlejší a přehlednější práce s daty získávanými v projektu měření fyzikálních jevů v horninovém masivu. V první části popisuje návrh aplikace pomocí Pythonu a webového frameworku Django, následně navazuje jejím nasazením na produkční server. Další části se věnují především zlepšení výkonu a efektivity práce s daty. Výstupem z práce jsou podklady pro úpravu běžící databáze a funkční webová aplikace umožnující manipulaci s měřenými daty a jejich zobrazování.

Klíčová slova: měření fyzikálních veličin, zpracování dat, databáze, webová aplikace, Python, Django, PostgreSQL

Processing and visualization of data from the Bedřichov water supply

Abstract

The diploma thesis describes the design of a graphical representation of data from the Bedřichov water supply. The aim is faster and convenient work with data obtained in the project of measuring physical phenomena in the rock massif. The first part describes the design of the application using Python and the Django web framework, followed by its deployment on a production server. The other sections focus on performance improvement and effictient work with data. The output of the work are materials for editing production database and a functional web application for display and manipulation of measured data.

Keywords: measurement of physical quantities, data processing, databases, web application, Python, Django, PostgreSQL

Poděkování

Mé poděkování patří především Ing. Janu Kolajovi za podporu a čas, který mi věnoval při vedení práce. Dále konzultantu panu doc. Milanu Hokrovi za trpělivé vysvětlení fyzikálního pozadí.

Obsah

Seznam obrázků

Seznam kódů

Seznam zkratek

API	Application Programming Interface, aplikační rozhraní
ASCII	American standard code for information interchange
ASGI	Asynchronous Server Gateway Interface
BRIN	Block range index
CSV	Comma-separated values – Hodnoty oddělené čárkami
CxI	Ústav pro nanomateriály, pokročilé technologie a inovace
DA	Django admin
DTL	Django Template Language
DT	Debug Toolbar
FM	Fakulta mechatroniky, informatiky a mezioborových studií Technické univerzity v Liberci
HTML	Hypertext Markup Language
HTTP	Hypertext Transfer Protocol
ISO	Optical disc image
JSON	JavaScript Object Notation
LTS	Long-term Support – prodloužená podpora
MTV	Model, Template, View
MVC	Model, View, Controller
NTI	Ústav nových technologií a aplikované informatiky
OIS	Oddělení informačních systémů
OMP	Oddělení modelování procesů
ORM	Object-relational mapping – Objektově relační mapování
RSA	Iniciály autorů Rivest, Shamir, Adleman
SFTP	Secure File Transfer Protocol
SSH	Secure Shell
TUL	Technická univerzita v Liberci
UTF	Unicode transformation format
VPN	Virtual private network
WSGI	Web Server Gateway Interface

1 Úvod

Práce byla zadána z podnětu výzkumu na Ústavu nových technologií a aplikované informatiky Fakulty mechatroniky, informatiky a mezioborových studií (NTI FM) a Oddělení modelování procesů Ústavu pro nanomateriály, pokročilé technologie a inovace (OMP CxI) Technické univerzity v Liberci (TUL) pro potřeby zobrazování a analýzy dat z měření probíhajícího ve vodárenském přivaděči Bedřichov. Cílem práce je vytvořit takový systém pro prezentaci dat, který dokáže nahradit současnou webovou aplikaci dostupnou z bedrichov.tul.cz/Tunel. Motivací k náhradě je především potřeba zobrazit data v uživatelsky definovaných grafech a umožnit získávání skutečných fyzikálních hodnot jednotlivých veličin z hodnot naměřených senzory.

Tato práce je členěna do logických celků podle technologií, z tohoto důvodu práce nereflektuje časovou souslednost provedených prací. Ve skutečnosti byl vývoj prací iterativní. Tento přístup se však nehodí pro vysvětlení provedených prací vzhledem k jednotlivým použitým prostředkům. Nevýhodou je chybějící kontext, který si tak musí čtenář v případě potřeby doplnit z ostatních částí dokumentu.

V rámci tématu vodárenského přivaděče Bedřichov vznikl také diplomový projekt. Ten pojednává především o aspektech měření u jednotlivých měřených přístrojů, dále se zaměřuje na výběr programovacího jazyka, webového frameworku a knihovny pro vykreslování grafů. Doporučeními, která byla závěrem tohoto projektu, se tato diplomová práce řídí a informačně z něho čerpá.

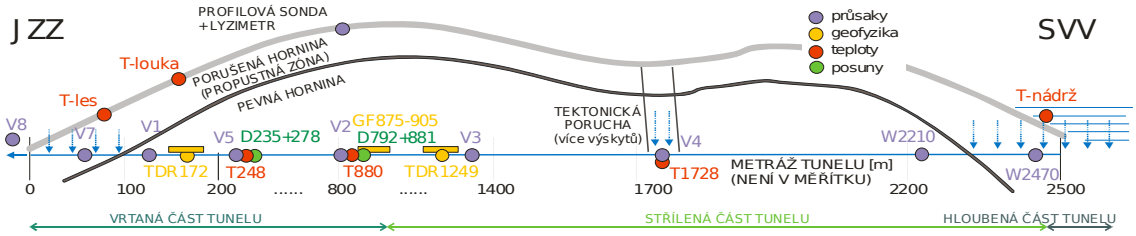
2 Sběr dat

Vodárenský přivaděč Bedřichov je z hlediska měření geologicky významné místo. Jde o tunel, který je zčásti vrtaný a zčásti střílený, je dlouhý přibližně 2600 m a místy vede až v hloubce 150 m žulovým masivem.[1]

Obrázek 2.1 je převzatý z dokumentu [2]. Zde znázorněné body měření pokrývají širší rozsah aktivit, než je zahrnuto ve zpracovávaných datech. Zobrazené označení senzorů slouží pro interní komunikaci výzkumných týmů. Pro tuto práci jednotlivá označení nejsou důležitá, obrázek má však ilustrační hodnotu a lze si podle něho udělat hrubou představu, čím se měření zabývá.[3]

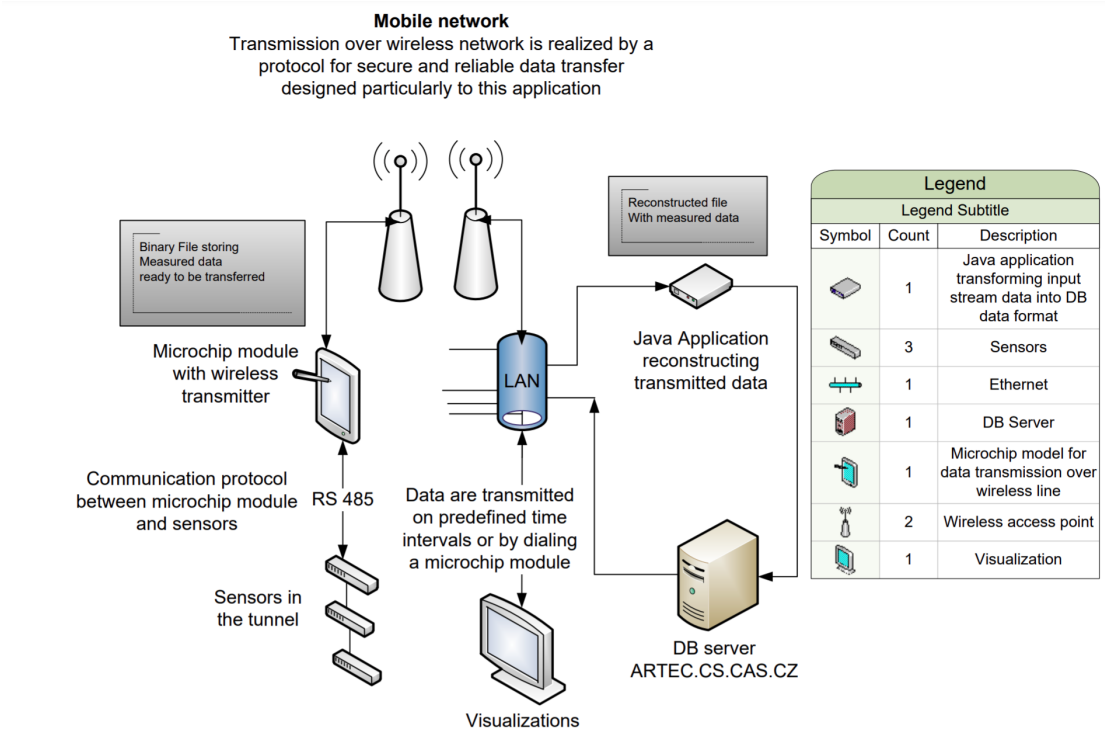
Detailní popis instalovaných měřících zařízeních lze nalézt v závěrečné zprávě [4].

Data z těchto zařízení jsou v pravidelných intervalech přenášena na databázový server. Historicky nebyla pouze jedna cesta, kterou byla data přenášena do databáze. Prvotní instalaci (viz Obrázek 2.2) navrhl Ing. Roman Špánek, Ph.D. Je autorem také java aplikace *RemoteDataReceiver*[5] a již zmíněné, v současné době používané, webové aplikace (dále označované za původní). Aplikace *RemoteDataReceiver* byla nahrazena jednotlivými skripty napsanými unikátně pro každý senzor (viz Obrátek 2.3). Tyto skripty jsou v současné době výhradním prostředkem pro vkládání dat do databáze. Sekundárně je využívána také komerční služba ThinkSpeak, která se specializuje na vizualizaci, analýzu a agregování živých datových toků v cloudu.

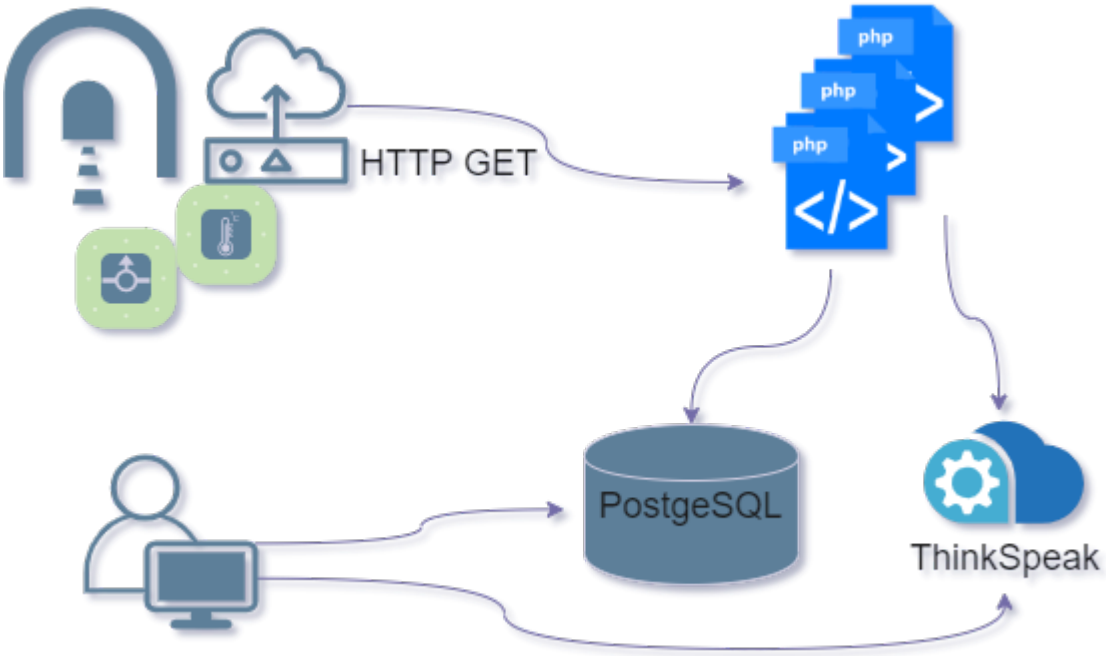


Obrázek 2.1: Názorné schéma přivaděče [2]

V diplomovém projektu byly nastudovány fyzikální vzorce pro převod měřených dat na hodnoty fyzikálních veličin. V přiložených souborech jsou tyto vzorce přehledně shrnuty v tabulce a přiřazeny ke konkrétním senzorům podle ID, jak jsou uloženy v databázi.



Obrázek 2.2: Původní schéma komunikační sítě [**spanek_2011**]



Obrázek 2.3: Současná komunikační síť

3 Architektura aplikace

Aplikace musí být schopná připojení na existující databázový systém.[1] Případně akceptovat ideálně i jakýkoli jiný, pokud dojde k jeho změně. Takové požadavky umí splnit vrstva objektově relačního mapování, dále jen (ORM), která převádí třídy objektového programovacího jazyka na entity databáze. Záleží pouze na zvolené knihovně zprostředkovávající ORM, jaké databázové systémy budou podporovány.

Pro aplikaci bylo již v původním řešení zvoleno webové rozhraní. Takový přístup má řadu výhod, mezi které patří multiplatformnost, snadná distribuce nebo například přístup z jakéhokoli počítače připojeného k internetu nebo také mobilního telefonu. Tyto vlastnosti bylo vhodné zachovat. Volené prostředky byly z toho důvodu vybírány s ohledem na webový vývoj. Hlavním programovacím jazykem byl zvolen Python.

3.1 Obecně

Webová aplikace je kód uložený na serveru (fyzickým nebo virtuálním) a spouštěný webovým serverem (softwarem). Klientský HTTP požadavek na stránku je nejprve směřován na proxy server, který funguje jako prostředník mezi klientem a aplikacemi. Těch tak může být více, proxy může také splňovat funkci vyvažování sítě (load balancer) nebo některé požadavky sama odbavovat.

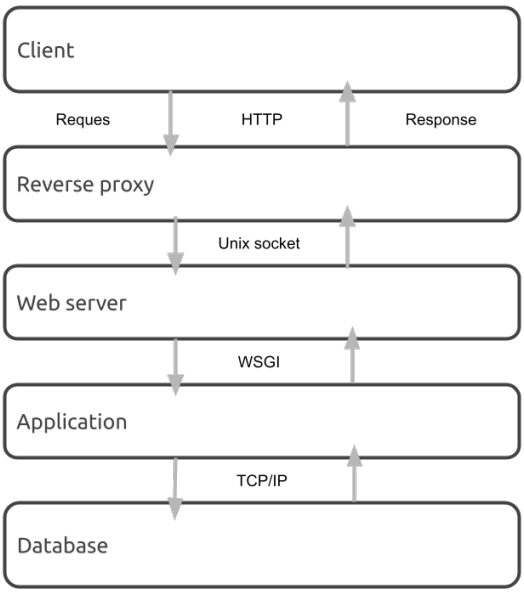
Webový server přijímá požadavek od proxy serveru pomocí meziprocesové komunikace skrze unix socket. Dále komunikuje s aplikací standardizovaným protokolem, kterým je v Pythonu obvykle WSGI (Web Server Gateway Interface).

Aplikace přijme data, uloží je do databáze, vyžádá si potřebná data z databáze, případně provede jakoukoli jinou potřebnou logiku k odbavení požadavku, a svou odpověď směřuje stejnou cestou zpět ke klientovi. Popsanou komunikaci znázorňuje obrázek 3.1.

3.2 Django

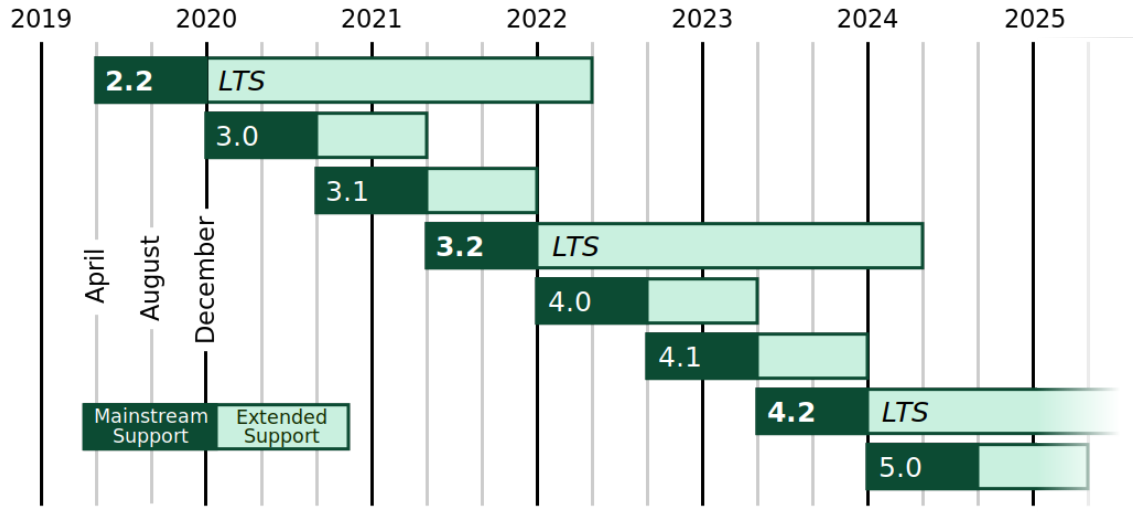
Hlavní platformou pro vývoj bylo vybráno *Django* jakožto výkonný a komplexní webový framework. Django poskytuje zázemí pro mnoho aplikačních scénářů a velmi kvalitní dokumentaci. Projekt předpokládá možné rozšíření a navázání dalších prací, které je tak umožněno díky tomuto výběru a přístupnosti jazyka Python.

[1] Database Management System – (SŘBD) systém řízení báze dat.



Obrázek 3.1: Obecné schéma komunikace

Django má stabilní komunitu vývojářů a předvídatelný plán uvolňování nových verzí. Na obrázku 3.2 je možné vidět, že každé dva roky vyjde verze s prodlouženou dobou podpory (LTS – *Long-term support*).



Obrázek 3.2: Plán vydání nových verzí [**Django-releasnote**]

Každý projekt vytvořený v *Djangu* je členěn do takzvaných *aplikací*, čímž je zajištěna modulárnost, a jeden web tak může být členěn v rámci logických celků, například na *aplikace* obstarávající blog, e-shop nebo chat se zákazníky. Každá tato aplikace je implementována pomocí návrhového vzoru MTV. Zkratka vychází z anglických slov *Model, Template* a *View* a jedná se o modifikaci návrhového vzoru MVC (*Model, View, Contoler*).

Každý požadavek[2] (mimo těch na statické soubory[3]) zpracuje nejprve URL dispečer (v nastavení přiřazený proměnné `ROOT_URLCONF`). Ten směřuje požadavky dále na příslušná *View* jendotlivých *aplikací*, podle seznamu adres `urlpatterns`. Adresy je možné definovat pomocí regulárních výrazů nebo od verze 2.0 také pomocí syntaxe podobné zápisu *fstring* známého z Pythonu, s tím rozdílem, že zde se pro identifikování proměnné užívá ostrých závorek. Zde záleží na pořadí, protože jakmile se jeden z URL vzorů shoduje s požadavkem, Django importuje daný modul (jinou *aplikaci* Django) nebo zavádí příslušné zobrazení (View).

Zobrazení může být funkcí Pythonu nebo třídou, může být také generováno automaticky, tzv. generické View. To se hodí v případě, že není potřeba zásadně manipulovat s daty a veškerá logika může být provedena na úrovni šablonovacího jazyka. V opačném případě je nutné implementací zajistit požadovanou funkčnost. Zobrazení přebírá webový požadavek a vrací webovou odpověď. Touto odpovědí může být HTML webové stránky, přesměrování, JSON dokument nebo jakýkoli jiný soubor. Konvencí je umístění jeho kódu do souboru `view.py`.

Typicky je zde implementován kód, který načítá data z databáze, ke kterým díky *modelům* přistupuje jako k objektům, vytváří z nich kontext požadavku, a ten následně předává šabloně HTML stránky. Po zpracování šablonou předává vyrenderovaný obsah webovému serveru, který jej odesílá na klienta.

Modely jsou třídy jazyka Python. Obsahují proměnné (atributy) a chování dat. Každý model se mapuje do jedné databázové tabulky pomocí objektově-relační vrstvy Djanga. Díky tomu je poskytnuto automaticky generované API pro přístup k databázi. Django ve většině případů nepoužívá knihovny třetích stran, místo nich implementuje svá vlastní řešení. Pokud by však někomu nevyhovovala, je vždy možné je nepoužívat a místo nich integrovat vlastní řešení, nebo řešení třetí strany. Podobně tomu je s vrstvou ORM, kterou je možné zaměnit například za *SQLAlchemy*. Jak Django ORM, tak *SQLAlchemy* jsou pevně svázány s relační algebrou, ze které těží při práci s relačními databázemi. Z toho plyne omezení v použití s některou ze souborových databází. Zde existují knihovny, které toto umí vyřešit a zároveň zachovat stejný přístup k datům skrze implementované modely (například *Django MongoDB Engine* [6]).

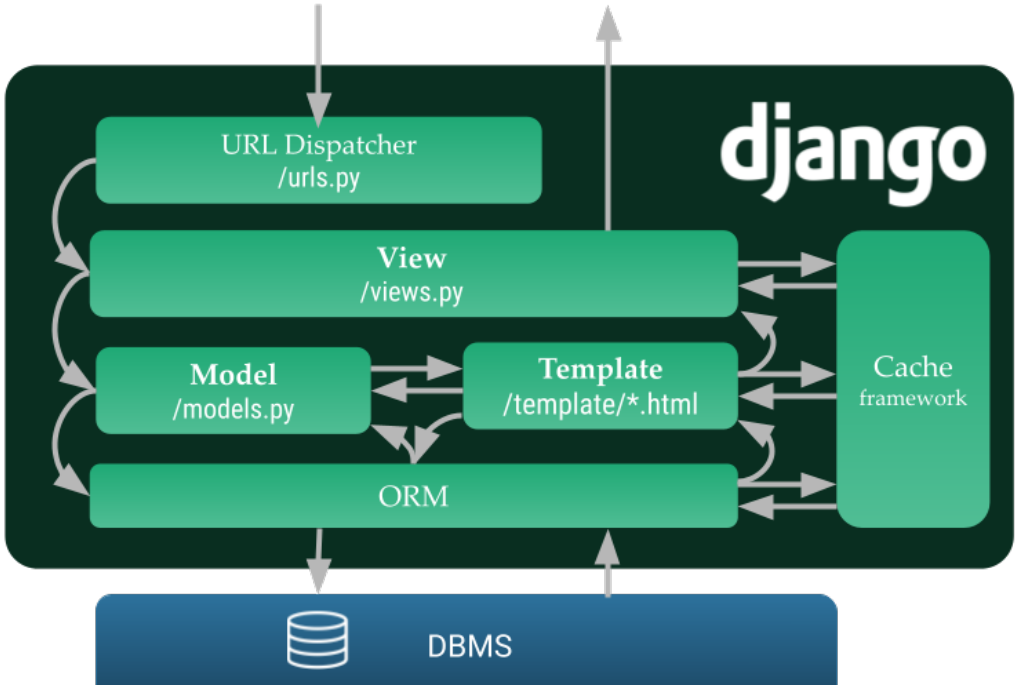
Další důležitou a již zmíněnou součástí je výkonný systém šablon (DTL – *Django Template Language*). I zde je možná jeho výměna, alternativou může být například systém *Jinja2*. Oba systémy jsou velmi podobné, a pokud má čtenář zkušenost se šablonovacími systémy známými z PHP (*Latte*, *Twig*), jistě shledá, že mezi nimi po uživatelské stránce není mnoho rozdílů.

Přestože část aplikační logiky může díky DTL převzít sama šablona, není to dobrý postup. Podle návrhového vzoru MVT by měla být aplikační logika zřetelně oddělena od prezentace. Pro efektivnější návrh šablon podporuje DTL dědičnost i vkládání bloků kódu.

Poslední vrstvou uvedenou na obrázku 3.3 je Cache framework. Django umož-

[2]HTTP requests – požadavek klienta na odpověď serveru. Řídí se protokolem HTTP.

[3]Statickými soubory se rozumí takové soubory, o kterých není nutné programově rozhodovat nebo je modifikovat na straně serveru. Mohou jimi být mediální soubory, kaskádové styly, soubory javascriptu atd.



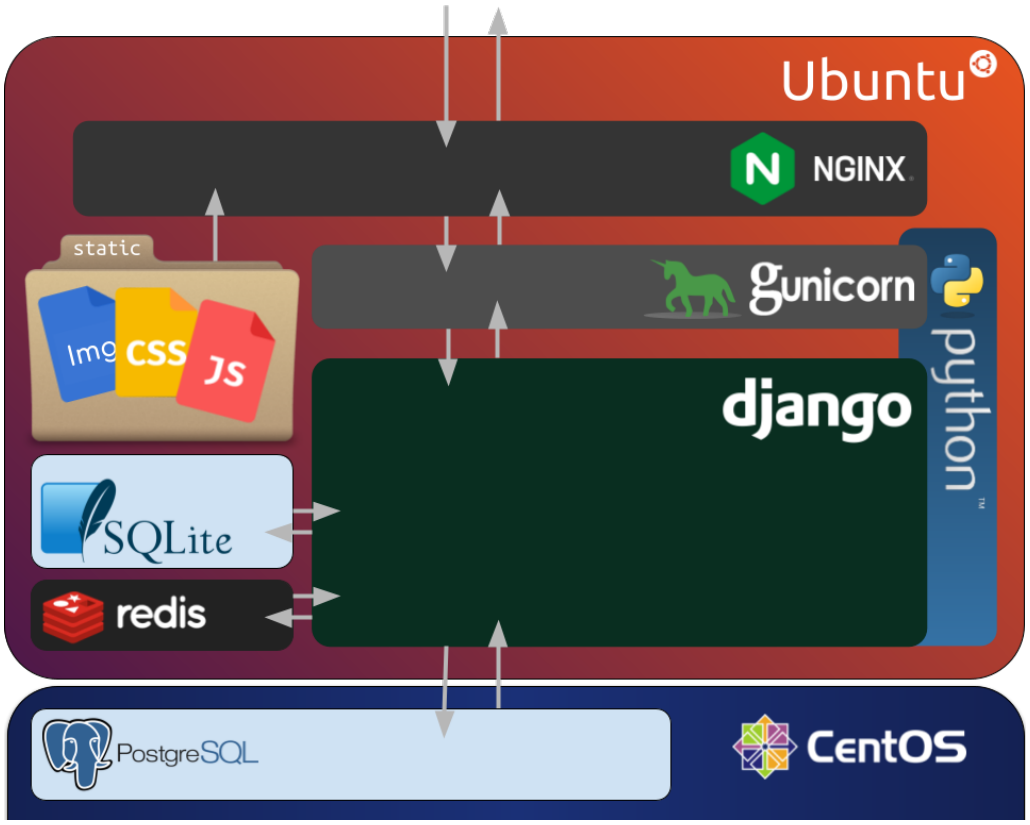
Obrázek 3.3: Architektura Djanga

ňuje využívat mezipaměť prakticky na všech úrovních odbavování dotazu. Je možné do ní ukládat celé pohledy, části stránek v *template* nebo jen jednotlivé dotazy na databázi.[4]

Mezipaměť může být realizována různými backendy. Pro vývoj je vhodná *DummyCache*, která svou funkci pouze předstírá, ale ve skutečnosti nic neukládá. Další možnosti užití jsou popsány v kapitole 6.

Nutno podotknout, že uvedené schéma je značně zjednodušené a do procesu vstupují další vrstvy, jakými jsou *SecurityMiddleware*, *SessionMiddleware*, *AuthenticationMiddleware*, *MessageMiddleware* a další. Uvedené cesty k souborům jsou relativní vzhledem ke kořenovému adresáři *aplikace*. Implementace těchto souborů je nezbytná.

[4]Prostřednictvím hlaviček HTTP dokáže Django také sdělit prohlížeči doporučení, jak pracovat s mezipamětí na úrovni klienta.



Obrázek 3.4: Aplikační struktura

3.3 Projekt zobrazování dat

Následující podkapitoly popisují postup tvorby aplikace. Zvolené aplikační zázemí je graficky znázorněno na obrázku 3.4. Zobrazené závislosti odpovídají obecnému schématu, jak bylo uvedeno v kapitole 3.1

3.3.1 Založení projektu

Django obsahuje dva nástroje pro automatizování některých administrativních úloh. Jsou to `django-admin.py` a `manage.py`. Obecně lze říci, že svou funkcionalitou se vzájemně překrývají, oba přijímají stejné vstupy, a až na výjimky dělají totéž. V ukázce 1 je možné vidět jejich obvyklé použití. Použitím příkazu, na prvním řádku v ukázce, se vygeneruje souborová struktura projektu, její základní nastavení a také již zmíněný skript `manage.py`, který je od této chvíle možné používat pro další kroky.

Vytvoření aplikace pomocí `manage.py`, jak ukazuje řádek 2, má výhodu, že nastavuje proměnnou prostředí `DJANGO_SETTINGS_MODULE` tak, aby ukazovala na *settings.py* daného projektu (dále jen nastavení). Základní struktura složek totiž není pevně daná, a bylo by možné spouštět aplikaci i v jiném adresáři než v tom, který obsahuje také projekt.

```
1 $ django-admin startproject tunel
2 $ python manage.py startapp app
3 $ python manage.py inspectdb > app/models.py
```

Listing 1: Založení projektu

3.3.2 Nastavení databáze

Django je webový framework, který se především hodí pro rychlé a efektivní stavění aplikací takříkajíc na zelené louce. Umožňuje přístup *Code-First*, tento přístup obchází vytváření a správu databáze tím, že umožňuje soustředit se pouze na vývoj aplikace a automaticky generuje skripty pro migraci databáze.

V případě této práce bylo ale nutné zvolit přístup *Database-First*, protože vytvářená aplikace se má pouze připojit na již existující databázi, a pokud možno ji nijak neměnit, aby nevznikla nekompatibilita s jiným běžícím systémem, na této databázi závislým.

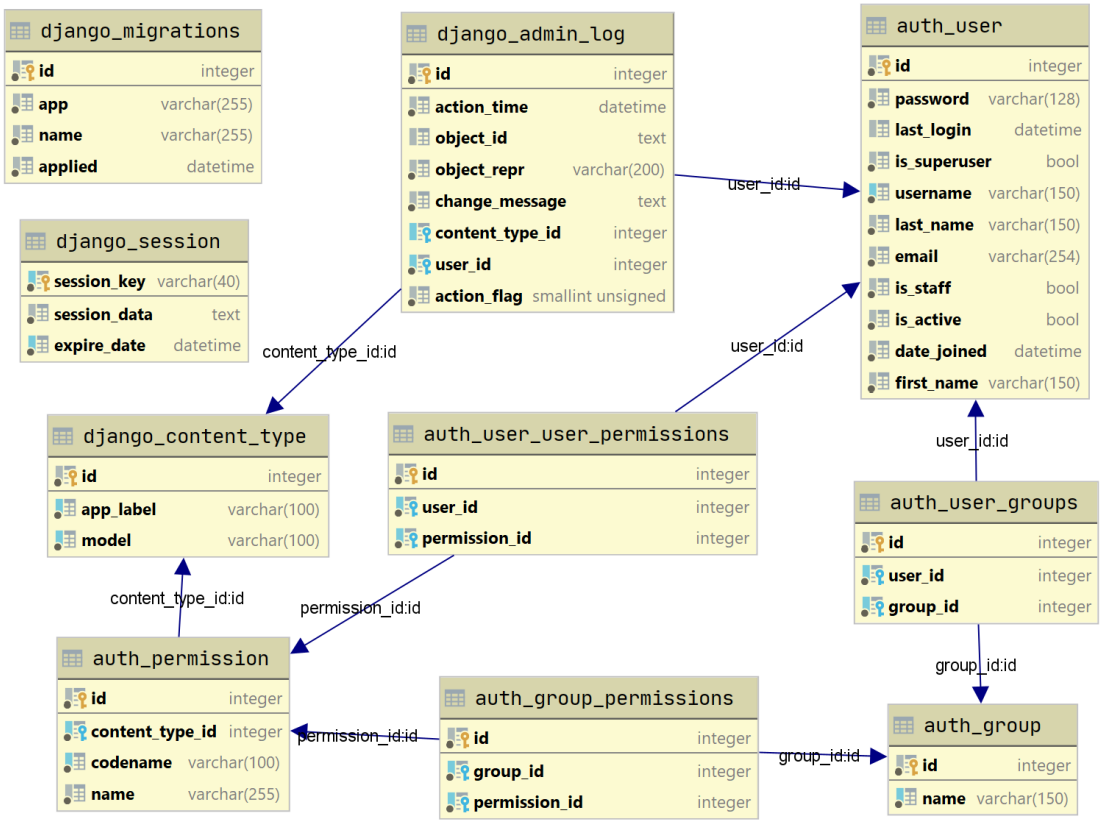
Pro tento přístup Django poskytuje podporu příkazem `inspectdb` (řádek tři v 1), který oskenuje připojenou databázi, a pokusí se k ní vytvořit příslušné objekty, tedy v terminologii Djanga tzv. modely. Výsledek tohoto kroku není dokonalý, ale je velmi nápomocný. Vzniklé modely tak nebylo nutné příliš upravovat, pouze rozšiřovat jejich možnosti a funkcionalitu.

Django samotné k tomu vypíše doporučení:

- "Uspořádejte pořadí modelů.
- Ujistěte se, že každý model obsahuje jedno pole s `primary_key=True`.
- Ujistěte se, že každý *ForeignKey* a *OneToOneField* má parametr `on_delete` nastavený na požadované chování.
- Odstraňte řádky `managed=False`, pokud si přejete, aby Django vytvořilo, upravilo nebo odstranilo tabulku.
- Neváhejte přejmenovat modely, ale nepřejmenovávejte hodnoty `db_table` ani názvy polí."

Tato poslední věta není zcela pravdivá. Pole umožňují předat název atributu `db_column='name_in_table'`, stejně jako je tomu u celých modelů s názvem entity `db_table`. Díky tomu je přejmenovaní možné, ale pouze za předpokladu, že je známý skutečný název v databázi. Protože není žádoucí, aby Django zasahovalo do databáze obsahující data z měření, proměnná `managed` byla ponechána na výchozí hodnotu. Django ale pro ověřování uživatelů potřebuje databázi, kam může ukládat uživatele, skupiny uživatelů, jejich práva, log provedených změn a uživatelské relace.[5] Z toho důvodu byla paralelně s běžící *PostgreSQL* využita ještě databáze *SQLite*.

[5]Session – umožňuje udržovat kontext pro bezstavový protokol HTTP.



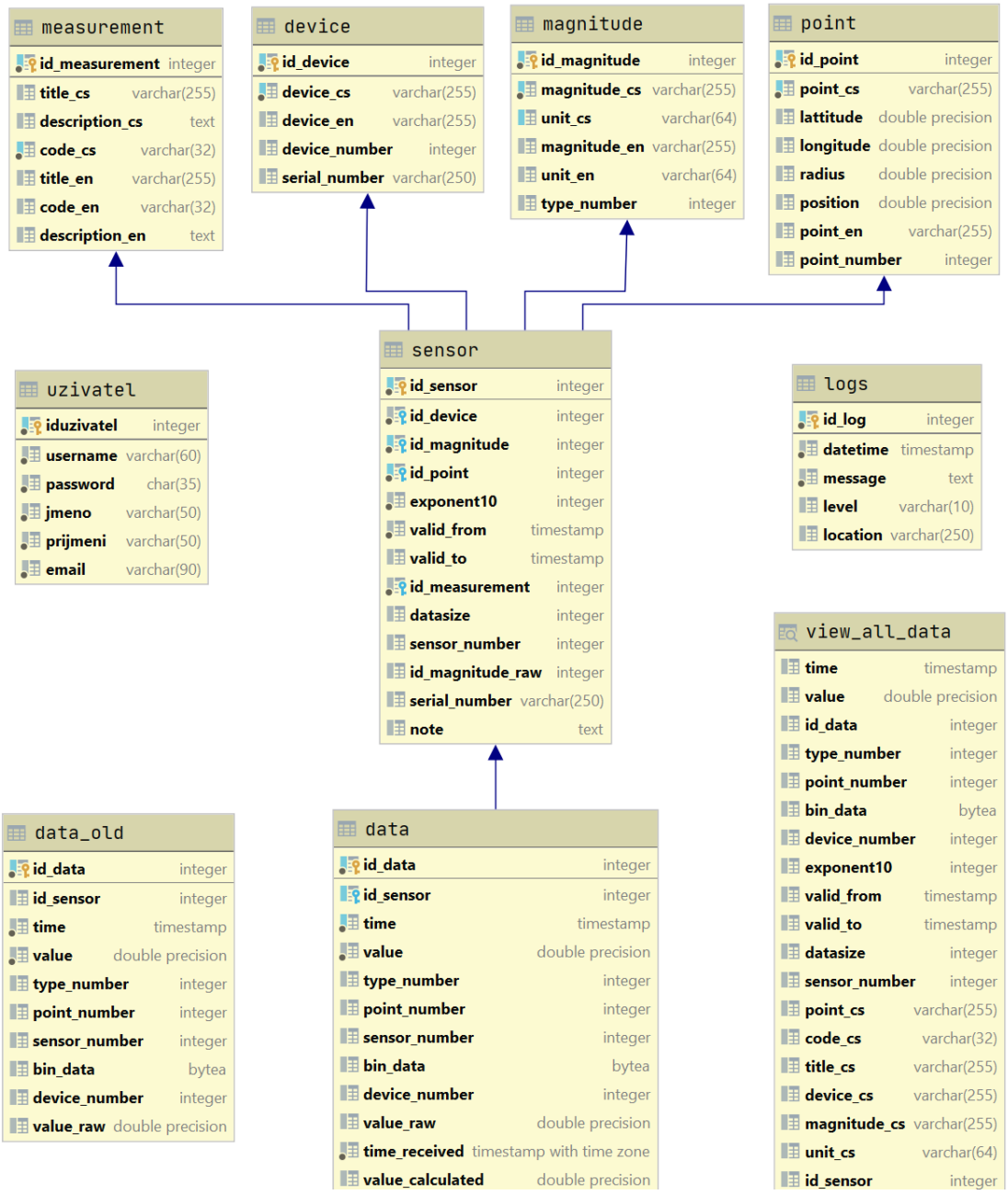
Obrázek 3.5: Správa uživatelů v Django (schéma je generované pomocí UML doplňku pro PyCharm)

Připojení databáze se realizuje pomocí slovníku `DATABASES` v nastavení, obsahujícího parametry spojení. Aby bylo možné mapovat více databází, byly vytvořeny aliasy `'default'` pro *SQLite* a `'tunel'` pro *PostreSQL* v nastavení. Dále implementován objekt *DatabaseRouter* a na něj v nastavení nasměrována odkazující proměnná `DATABASE_ROUTERS = ['tunel.router.PstDatabaseRouter']`. Aby *DatabaseRouter* měl podle čeho rozhodovat, na kterou databázi směřovat požadavky (tedy ke které patří který model) byla ke každému vytvořenému modelu přidána proměnná `app_label = 'tunel'` do objektu *Meta*. Uživatelské modely Django vytváří automaticky (viz Obrázek 3.5) a není třeba je explicitně definovat. Pro rozšíření těchto výchozích modelů se obvykle užívá nová tabulka (model) se spojením 1:1 (*OneToOneField*), ale lze je pochopitelně i kompletně přepsat.

3.3.3 Vytvoření administračního rozhraní

Velkou výhodou Djanga je poskytnutí poloautomatizovaného generování administračního rozhraní. Tento modul (*aplikace*) se nazývá Django admin (dále jen DA).

Čte metadata z modelů a na základě nich poskytuje rozhraní, které lze jednoduše upravovat pomocí nastavování definovaných příznaků. Například pro definování atributů, které se mají zobrazit ve sloupcích tabulky, stačí přidat tyto sloupce do



Obrázek 3.6: Schéma *measurement*

seznamu `list_display`. Filtry, kterými lze omezit rozsah zobrazených výsledků, se přiřazují proměnné `list_filter`. Tato proměnná by měla obsahovat seznam atributů v tabulce nebo odkazů do tabulek v relaci vytvořených pomocí dvojtého podtržení. Například u stránky se zobrazením měřených dat byl vytvořen filtr podle měřené veličiny přidáním řetězce `'id_sensor__id_magnitude'`.

Tato práce si neklade za cíl suplovat dokumentaci Djanga, ani vysvětlovat provedené změny v rozhraní. Případný zájemce jistě rád nahlédne do kódů přiložených v souborech k této práci.

Pro dosažení požadovaných funkcí byly použity následující rozšiřující balíčky: *admin_auto_filters*, *rangefilter*, *import_export*. Pro zajištění dat pro graf byla implementována třída *ChartAdmin* dědící od *admin.ModelAdmin*. Dále byli upraveny výchozí šablony a implementována knihovna *ChartJS* pro zobrazení grafů.

V první fázi byly také implementovány vzorce pro přepočty hodnot. Protože následně došlo ke změně na úrovni databáze, tato část kódu se do výsledné aplikace nepromítla. Bylo možné z ní ale vyjít při implementaci přepočtů hodnot na uživatelské vyžádání. Tato funkce byla vytvořena pomocí rozhraní Django akcí. Změny v databázi jsou provedeny efektivní implementací pomocí hromadného updatu. Pouze u senzorů ID 168 a 184, tedy čítačů pulzů, byla implementace méně efektivní z důvodu velmi složitého vytváření dotazu s použitím funkce window. Funkce Window je překládána na SQL klauzuli `OVER`, slouží k výběru definovaného okénka, neboli v tomto případě k výpočtu nad uloženými záznamy s definovaným posunem.

Vytvořený program má následující funkce:

- Správa uživatelů
- Řízení oprávnění
  - na základě skupin
  - i jednotlivého přidělení
- Log provedených změn uživatelem
- Přístup ke všem uloženým datům podle logiky databáze
  - zobrazování
  - vytváření
  - editace
  - mazání
- Přizpůsobené prostředí s přehledně zobrazenými informacemi
- Možnost vyhledávání
- Import dat
- Export dat
  - na základě filtrů
  - na základě výběru
- Uživatelsky vyvolané přepočítání hodnot
- Agregované zobrazení grafu pro zvolený senzor
  - pro zvolené období
  - agregace po dnech
  - zobrazeno minimum, průměr a maximum
  - přibližování a posun grafu

4 Produkční nasazení

V rámci diplomového projektu, na který tato práce navazuje, byla webová aplikace testována pouze na lokálním počítači a v testovacím režimu také na serveru, na kterém je spuštěný klon databáze. Django nebylo spouštěno pod produkčním serverem, ale pouze pod serverem vývojovým, který je součástí Djanga a samotného Pythonu.

Tento server je nevhodný jak z hlediska bezpečnosti, tak z hlediska výkonu. Oddělení informačních systémů (dále jen OIS) TUL z toho důvodu poskytlo virtuální server pro instalaci potřebného softwarového zázemí.

Následující podkapitoly procházejí potřebné kroky k úspěšnému spuštění webové aplikace, od instalace operačního systému, až po nastavení webového serveru.

4.1 Instalace serveru

OIS používá k virtualizaci produkt *VMware Workstation* společnosti *VMware*. Přístup k virtuálním počítačům je umožněn pomocí webového rozhraní HTML5 klienta jménem *vSphere Client*. Používání *VMware vSphere* klienta je z bezpečnostních důvodů možné pouze přes VPN. Toto omezení platí i pro přístup z vnitřní sítě TUL[7]. Po zapnutí virtuálního stroje v prostředí *vSphere* je možné připojit program *VMware Remote Console*. Ten se instaluje lokálně a umožňuje připojit ISO soubor s instalačním médiem umístěný v lokálním souborovém systému, jako disk k virtuálnímu stroji.

Byla zvolena serverová distribuce Ubuntu v aktuální verzi 20.04.2 *Focal Fossa*. Jedná se o takzvanou LTS verzi s udávanou podporou 5 let.

Instalace proběhla pomocí webové konzole a následně instalován *open-vm-tools*, který je doporučen OIS [7].

Přístup přes webovou konzoli se může zdát značně nepohodlný, nefunguje v něm práce se schránkou a znemožňuje tak kopírování ven, stejně jako vkládání příkazů. Z toho důvodu by bylo výrazně pohodlnější moci pracovat se serverem přes SSH spojení. Nastavení SSH proběhlo již při instalaci, přesto však kdyby čtenář následoval tyto kroky na desktopovém počítači, jsou v kódu 2 uvedeny potřebné kroky.

Ubuntu používá balíčkovací systém *apt*, ten lze ovládat pomocí `apt-get` nebo moderněji pomocí příkazu `apt`. Na řádku jedna kódu 2 je nejprve načten aktuální index repozitáře dostupných programů. Toto je vhodné udělat, aby bylo zajištěno, že jsou instalovány nejnovější verze programů.

Na dalších řádcích je pak znázorněno povolení ve firewallu pomocí příkazu `ufw`. Zde je třeba povolit port, který používá SSH ke spojení. Ve výchozím nastavení je

```
$ sudo apt update             #obnovení indexu repozitáře
$ apt install openssh-server #instalace ssh serveru
$ sudo ufw allow ssh          #povolení portu ve firewallu
$ sudo systemctl status ssh  #kontrola nastavení
```

Listing 2: Nastavení SSH na nové instalaci Ubuntu

```
$ python
Command 'python' not found, did you mean:
  command 'python3' from deb python3
  command 'python' from deb python-is-python3
$ update-alternatives --install /usr/bin/python python
↪  /usr/bin/python3 1
$ python --version
Python 3.8.5
$ sudo apt install python3-pip -y
$ update-alternatives --install /usr/bin/pip pip /usr/bin/pip3 1
$ pip --version
pip 21.1.1 from /usr/lib/python3/dist-packages/pip (python 3.8)
```

Listing 3: Nastavení Pythonu

to port 22.

Dále bylo možné umístit veřejný klíč do domovské složky uživatele, a zajistit tak přihlašování pomocí páru rsa klíčů. Tato možnost je doporučena jak z důvodu většího pohodlí, tak z důvodu zabezpečení. Po zajištění přihlašování pomocí rsa klíče je totiž možné nastavit v souboru `/etc/ssh/sshd_config` proměnnou `PasswordAuthentication` na `no`, a tím zabránit útokům hrubou silou nebo vylákání hesla pomocí phishingu.

Všechny zde uvedené příkazy předpokládají přihlášení do systému pod účtem uživatele s oprávněním správce, nikoli superuživatele.

Django 3.2 podporuje *Python 3.6, 3.7, 3.8* a *3.9. Ubuntu 20.04* již v základní instalaci obsahuje *Python 3.8*. Pokud však uživatel chce spustit interaktivní režim Pythonu, je nutné to udělat voláním příkazu `python3`. V opačném případě se setká s dotazem začínajícím na řádku 3 v 3. Zde je možné nalinkovat Python 3 jako Python (řádek 5), aniž by to vyvolalo kolize v systému.

Analogicky bylo postupováno se správcem balíčků *pip*, jen s tím rozdílem, že ten je potřeba nejprve nainstalovat s oprávněním správce. V použité distribuci operačního systému je výchozím chováním globální instalace do kořenové složky systému. Při použití s jinou distribucí by mohlo být nutné použít přepínač -H. Po vytvoření aliasu kontrola používané verze již hlásí vše korektně.

4.1.1 Django

Zdrojové kódy je možné přenést na server pomocí protokolu SFTP. Aplikace byla tvořena na počítači s OS Windows. K přenesení souborů aplikace byl použit program *WinSCP*. Ten zprostředkovává grafické uživatelské prostředí pro Windows k přenosu souborů tímto protokolem skrze aplikaci PuTTY. Po umístění projektu do domovského adresáře uživatele bylo vytvořeno virtuální prostředí Pythonu a do něj instalovány závislosti projektu. V souboru requirements.txt jsou uvedeny závislosti, které projekt využívá ke svému běhu. Tento soubor lze generovat automaticky pomocí příkazu `pip freeze` s výstupem přesměrovaným právě do tohoto souboru. Uvedený postup lze reprodukovat pomocí příkazů 4.

```
$ sudo pip install virtualenv
$ cd /home/user/project
$ virtualenv venv
$ source venv/bin/activate
(venv) $ # pip freeze > requirements.txt ve vývojovém prostředí
(venv) $ pip install -r requirements.txt
```

Listing 4: Instalace projektu

Na tomto místě se hodí jen malá poznámka ohledně databázového adaptéru *Psycopg*, který zajišťuje spojení mezi *PostgreSQL* a *Pythonem*. Jedná se o wrapper knihovny *libpg* implementovaný v jazyce *C*, existuje ve dvou verzích, v předkompilované *psycopg2-binary* a *psycopg2*. Obě jsou plnou implementací Python DB API 2.0, ale druhá zmíněná je závislá na externích knihovnách. Vývojové prostředí[1] *PyCharm* od *JetBrains* automaticky instaluje psycopg2, ale při nasazení na server byl preferován předkompilovaný balíček *psycopg2-binary* [8].

V tomto bodě je aplikace téměř připravená k otestování pod vývojovým serverem. Aby však mohla běžet pod doménou bedrichov2.tul.cz, je nutné ji přidat do seznamu povolených domén v proměnné `ALLOWED_HOSTS` v souboru settings.py. Tento seznam domén Django používá, aby zabránilo útokům typu *Cross-Site Request Forgery*, *cache poisoning* nebo podvrhnutím odkazu z emailu.[9] Toho lze využít například pro podvrhnutí odkazu pro reset hesla. V tomto souboru se nachází také proměnná `DEBUG`. Django by nikdy nemělo být spouštěno na ostrém serveru, pokud je tato proměnná nastavena na `True`. Zapnutí serveru se provede spuštěním skriptu `manage.py` s přepínačem `runserver`. Tento postup funguje, protože byly přeneseny všechny soubory, včetně vytvořené databáze *SQLite*. To však není podmínkou, pokud by čtenář například chtěl provádět deployment raději pomocí gitu, kde tyto soubory nejsou zahrnuty, je možné následovat kroky 5.

Zde je uveden také příkaz `collectstatic`, ten seskupí všechny statické soubory z různých aplikací a balíčků, které jsou používány v projektu, do složky, která je

[1] IDE (Integrated Development Environment) – textový editor rozšířený o kompilátor nebo interpret, debugger a další funkce usnadňují programátorovi práci.

uvedena v proměnné `STATIC_ROOT`. Toto bude později důležité, aby mohl webový server odbavovat statické soubory přímo.

```
(venv) $ python manage.py makemigrations
↪  #vytvoří skript s příkazy pro migraci
(venv) $ python manage.py migrate
↪  #provede změny v databázi
(venv) $ python manage.py createsuperuser
↪  #pro vytvoření uživatele s administračními právy
(venv) $ python manage.py collectstatic
↪  #seskupí statické soubory
(venv) $ sudo ufw allow 8000
(venv) $ python manage.py runserver 0.0.0.0:8000
↪  #zapne vývojový server; ctrl+c pro vypnutí
```

Listing 5: Spuštění projektu

4.1.2 Gunicorn

Django je pouze webovým frameworkem, k fungování potřebuje webový server. Protože většina webových serverů není psaná nativně v Pythonu, je nutné rozhraní, aby mezi nimi byla možná komunikace. To mu poskytuje rozhraní WSGI (*Web Server Gateway Interface*) nebo ASGI (*Asynchronous Server Gateway Interface*). WSGI je hlavní standard Pythonu pro komunikaci mezi webovými servery a aplikacemi, a z toho důvodu byl vybrán pro nasazení v této práci. ASGI je novější standard vhodný pro použití asynchronních funkcí Pythonu, nově také s podporou v Djangu od verze 3.x. Tuto specifikaci práce nevyužívá.

Hlavními možnostmi, které uvádí dokumentace, jak spustit Django pod WSGI, jsou:

1. Gunicorn,
2. uWSGI,
3. Apache a mod_wsgi.

Všechny tyto servery jsou osvědčené a v praxi využívané pro produkční nasazení. Gunicorn se však v tomto ohledu často uvádí na prvním místě. Mimo jiné ho na svých serverech využívá například český Python hosting Roští.cz a pro tuto práci byl zvolen také.

Po nainstalování Gunicornu do virtuálního prostředí Pythonu byla aplikace opět úspěšně testována. Při reprodukci těchto kroků se není třeba lekat toho, že se zdá, že web nefunguje zcela správně. Je to tím, že chybí nastavení pro poskytování statických souborů. Po testování byl uzavřen zkušební port 8000, jak znázorňují příkazy 6.

```
(venv) $ pip install django gunicorn
(venv) $ gunicorn --bind 0.0.0.0:8000 project.wsgi
(venv) $ sudo ufw deny 8000
```

Listing 6: Spuštění Gunicornu

```
 $ sudo vim /etc/systemd/system/gunicorn.service
[Unit]
Description=gunicorn daemon
After=network.target

[Service]
User=user
Group=www-data
WorkingDirectory=/home/user/web
ExecStart=/home/user/web/venv/bin/gunicorn --access-logfile -
 ↪  --workers 3 --bind unix:/home/user/web/project.sock
 ↪  project.wsgi:application
[Install]
WantedBy=multi-user.target
$ sudo systemctl start gunicorn
$ sudo systemctl enable gunicorn
$ systemctl status gunicorn
```

Listing 7: Nastavení souboru gunicorn.service v systemd

```
$ sudo apt install nginx
$ sudo vim /etc/nginx/sites-available/django
server {
  listen 80;
  server_name 147.230.21.145;
  location /static/ {
        alias /home/user/project/static/;
  }
  location / {
    proxy_pass http://unix:/home/user/project/tunel.sock;
  }
}
$ ln -s /etc/nginx/sites-available/django /etc/nginx/sites-enabled/
$ sudo nginx -t && sudo systemctl restart nginx
$ sudo ufw allow 'Nginx Full'
```

Listing 8: Nastavení směrování NGINX

Aby byl *Gunicorn* automaticky spouštěn po restartu operačního systému, byl nastaven deamon *systemd* a povoleno spouštění podle ukázky 7. Většinou se nastavení nepovede na první pokus. Pokud je zapotřebí znovu editovat soubor *gunicorn.service*, je nutné restartovat démona pomocí `sudo systemctl daemon-reload` a `sudo systemctl restart gunicorn`. Restart serveru *Gunicorn* je také nutné provést po nahrání každé nové verze aplikace, aby se provedené změny projevily.

4.1.3 NGINX

Posledním krokem byla instalace a nastavení *Nginx*, který v této sestavě funguje jako reverzní proxy server. Požadavky na statické soubory přímo odbaví a ostatní požadavky na webové stránky směřuje dále na server *Gunicorn*.

Nginx se po instalaci sám automaticky spouští, k tomu není třeba provést žádné dodatečné kroky, pouze zajistit, aby naslouchal na portu 80 na IP adrese serveru, nasměrovat ho správně na soket *Gunicornu* a na statické soubory, jak je možné vidět v kódu z příkazové řádky 8.

Konfigurační soubor (zde pojmenovaný django) byl uložen do složky sites-available a následně byl na tento soubor vytvořen symbolický odkaz ve složce sites-enabled.

4.1.4 Redis

Instalace Redis serveru byla provedena až jako poslední krok optimalizace. V rámci logiky členění dokumentu však popis jejího nastavení patří na toto místo. Užití Redis jako mezipaměti s Djangem má velice jednoduché nastavení. Stačí ji instalovat a v nastavení zajistit spouštění po restartu pomocí *systemd*.

```
$ sudo apt install redis-server
$ sudo vim /etc/redis/redis.conf
  # kontrola řádku 69: bind 127.0.0.1 ::1
  # změna hodnoty supervised na řádku 147 z no na systemd
$ sudo systemctl restart redis.service
```

Listing 9: Instalace ”key-value” mezipaměti Redis

5 Databázový server

V rámci projektu, na který tato práce navazuje, "bylo zjištěno, že běžící databáze je *PostgreSQL* verze 9.2.24. Tato verze byla vydána v roce 2012 a její podpora byla ukončena v září roku 2017. Nyní je aktuální verze *PostgreSQL 13.1.0*. Od vferze 9.2.x prošla tato databáze vývojem, který opravoval důležité bezpečnostní problémy, přidával nové funkce, ale také zásadním způsobem navyšoval výkon. Optimalizována byla efektivnost zpracování dotazů a zajištěno paralelní zpracování jednoho dotazu na více jádrech procesoru." [3]

Ze strany navrhovatele byl vznesen návrh, jestli by nemohl být nějaký alternativní databázový backend vhodnější. Uvažována byla databáze *InfluxDB*, která se specializuje na časové řady nebo souborová *MongoDB*. *Django* ani jednu z těchto databází nativně nepodporuje. Existují však řešení třetích stran, díky kterým je možné integrovat i tyto databázové systémy. Vytvořená aplikace by díky tomu, za předpokladu malých úprav, neměla stát v cestě výměně databázového systému, pokud by ji v budoucnu někdo z pracovníků OIS poptával.

Žádné alternativní řešení však nebylo testováno, především z toho důvodu, že původní návrh a výběr databáze prováděl Ing. Roman Špánek, Ph.D., který je v oblasti databází, a konkrétně ukládání senzorických dat, odborníkem.

Jím vybraná PostgreSQL je nekomerční, bezplatná a vydávaná pod licencí MIT. Je to objektově-relační databázový systém s více než 30 lety aktivního vývoje, který má pověst spolehlivosti, robustnosti, podpory funkcí a výkonu. [10] Výhodou v sestavě této práce je, že má velmi dobrou podporu funkcí v ORM Djanga.

5.1 Upgrade

Protože bylo rozhodnuto, že databázový engine zůstane PostgreSQL, bylo nutné provést upgrade z verze 9.2 na verzi 13. Byl vysloven předpoklad, že tento upgrade by měl přinést zásadní navýšení výkonu. Důvodem k tomuto předpokladu je, že změn bylo učiněno opravdu hodně, bylo by zbytečné je zde vyjmenovávat, pokud by čtenáře zajímaly, je možné je přehledně nalézt v dokumentaci [**potgres_featurematrix**].

Přestože server s databází, nad kterou probíhaly práce, je pouze klon originálního serveru, nejprve byla provedena záloha nastavení databáze a záloha dat pomocí příkazu `pg_dumpall > measurement_dump.sql`. Dále instalována a inicializována aktuální verze vedle stávající a testován upgrade pomocí příkazu `pg_upgrade` s přepínačem `--check`. Následně se vyskytla řada problémů, po jejichž úspěšném vyřešení mohlo dojít ke spuštění upgradu bez přepínače check. Celý proces je pro možnou

replikaci detailně popsán v dokumentu psaném pomocí syntaxe Markdown v přiložených souborech.

Důležité je spouštět upgrade pomocí skriptu instalovaného s novou verzí.

Alternativní volbou, jak dosáhnout upgradu, by bylo využití dump souboru k vytvoření nové databáze na nové verzi. Realizované řešení má však několik výhod. Je rychlejší a paměťově úspornější a provádí kontrolu kompatibility clustrů, datových typů atd.

5.2 Řešené problémy

Skript, pomocí něhož provádí PostgreSQL 13 upgrade, je dle dokumentace kompatibilní s původní verzí 8.2, přesto se vyskytlo několik problémů, které bylo nutné vyřešit. Po každém neúspěšném pokusu bylo nutné smazat datový adresář nové databáze a inicializovat ho znovu.

Původní adresářová struktura nebyla výchozí, což samo o sobě vedlo k nestandardnímu postupu. Při instalaci nové verze byla tato struktura ctěna a rozšířena pouze o jednu úroveň rozlišující verzi používané databáze. Nebylo také jisté, jestli původní databáze obsahuje rozšiřující balíček *contrib*. Hlubším prohledáním instalovaných služeb na serveru bylo zjištěno, že nikoli, že současný návrh se spoléhá pouze na stabilní a bezpečný základ bez doplňujících balíčků, které nemusí vždy dosahovat takového výkonu, jako základní funkce obsahující PostgreSQL.

Původní databáze používala lokalizaci anglického jazyka a kódování UTF-8, nová se vlivem nastavení serveru inicializovala lokalizací standardního jazyka C a kódováním ASCII. Řešením bylo volání `initdb` s parametrem `--locale=en_US.UTF-8`.

Dokumentace dále uvádí, že pro upgrade z verze 9.2 je nutné spouštět s přepínačem malé `-o`, pomocí kterého lze předat cestu k adresáři s nastavením původní databáze (velké je vyhrazeno pro definování cesty v nové verzi).

V průběhu upgradu skript střídavě zapíná a vypíná obě souběžné verze a v adresáři, nad kterým je proces spouštěn, vytváří dočasné soubory, které mohou být citlivé. Z důvodu bezpečnosti by k tomuto adresáři neměl mít přístup jiný uživatel. V tomto adresáři se také uloží log s případnými chybami a informacemi o průběhu. Zde se také otevírá socket služby *postmaster*. Ve verzi PostgreSQL 9.0 byl parametr vedoucí k tomuto adresáři pojmenovaný `unix_socket_directory`, následně ale došlo k rozšíření možnosti s použitím více adresářů a parametr byl přejmenován na `unix_socket_directories`. V prostředí CentOS z blíže nezjištěného důvodu skript špatně rozpoznává jednotlivé verze, a volání tak proběhne s chybným parametrem. Prvním možným řešením je upravit zdrojové soubory pg_upgrade, následně kompilovat a použít upravenou verzi. Použito však bylo alternativní, méně invazivní řešení, které používá "hack" s podvrženým souborem pg_ctl, který přijímá špatný parametr a volá originální soubor se správným názvem parametru v množném čísle. Takto podvrhnuté soubory byly následně vráceny zpět do výchozího stavu. Provedené příkazy jsou uvedeny v kódu 10.

Následně však socket přesto odmítl navázat spojení. Původní databáze byla z důvodu bezpečnosti nastavena tak, aby nepřijímala žádné lokální požadavky. Proto

```
$ mv /usr/bin/pg_ctl{,-orig}
$ echo '#!/bin/bash' > /usr/bin/pg_ctl
$ echo '"$0"-orig
  ↪  "${@/unix_socket_directory/unix_socket_directories}"' >>
  ↪  /usr/bin/pg_ctl
$ chmod +x /usr/bin/pg_ctl
$ # zrušeno pomocí:
$ mv -f /usr/bin/pg_ctl{-orig,}
```

Listing 10: Podvržení souboru pg_ctl

muselo být změněno nastavení v souboru *pg_hba.conf*.

Dále se objevil problém s tabulkou, která využívá OID (Object Identifier Types). PostgreSQL používá OID jako interní klíče v interních systémových tabulkách. Obecně je OID pužíván jako identifikátor objektu. Existuje více systémových datových typů využívající OID, které nejsou podporovány při upgradu. Původně navržená databáze žádný z těchto typů neobsahovala (všechny tabulky byly vytvořeny s atributem `WITHOUT OIDS`). V předešlých projektech, před touto diplomovou prací, však byla vytvořena tabulka *person*, která takový datový typ využívá. (Zde se vyskytla nesrovnalost s dokumentací. Tabulka využívá typ `regclass`, který sice interně využívá OID, ale `pg_upgrade` by ho podle dokumentace měl umět akceptovat.)

Pomocí klienta pgAdmin byla zjištěna definice tabulky a následně znovu definována bez problematického typu pomocí `ALTER TABLE ... SET WITHOUT OIDS;`.

Aby upgrade proběhl v pořádku, bylo nutné ještě dvakrát žádat správce o navýšení diskového prostoru přidělenému virtuálnímu stroji. Pokaždé o 2 GB a podruhé již migrace na novou verzi proběhla v pořádku.

5.3 Změna v návrhu

V práci [3] bylo navrženo přidat atribut s přepočítanou skutečnou hodnotou měřených veličin. Důvodem k tomu je ujednocení původní nekonzistence v datech. Návrh databáze byl proveden kvalitně v Boyceho–Coddově normální formě, což zajišťuje, že nedochází k redundanci dat pocházejících z funkčních závislostí.[11] Následné užívání databáze však ne vždy zcela ctilo tento návrh a použití jednotlivých atributů si vykládalo po svém. Některé atributy tak nebyly využity vůbec, v jiných docházelo k redundanci dat.

Příkladem takového užití je tabulka *point* ukládající informace o jednotlivých stanovištích, kde probíhá měření. Zde se nacházejí atributy *lattitude*, *longitude* pro záznam GPS pozice. Tato pole však nejsou užívána, protože tunel je ražen v jedné přímce a pro pracovníky NTI je tak jednodušší zaznamenávat pouze vzdálenost od vstupu do tunelu. (Nehledě na fakt, že v žulovém masivu nemá GPS zařízení příjem signálu z družic. Databáze je však navržena obecně, tak aby mohla přijímat i mírně

odlišná data. Například měření z povrchové meteostanice.) Proto je součástí také používaný atribut *position*. V doposud používaném softwaru však tento atribut nebyl pro pracovníky zobrazován odpovídajícím způsobem, a tak byla metráž přidávána také do názvu. Tím docházelo k redundanci dat. Takových příkladů je v užívání databáze více, proto je cílem navrhovaného řešení postupovat v souladu s návrhem databáze i, a to především, potřebami výzkumného ústavu.

Zmíněná nekonzistence v tabulce *data* je způsobena duplikováním skutečně změřené hodnoty senzorem a její reprezentací ve fyzikální veličině. Tabulka pro tento účel obsahuje atributy *value* a *value_raw*, kde první by měl obsahovat přepočtenou a druhý změřenou hodnotu. Tento návrh je velice výhodný, protože pokud by byla ukládána pouze přepočtená hodnota veličiny, údaj o měření by byl nenávratně ztracen. Pokud by naopak byla ukládána pouze surová hodnota měřená senzorem, docházelo by k neoptimální zátěži vlivem nutných výpočtů nad daty.

Současný stav je však takový, že obě pole obsahují stejnou hodnotu, kteroužto je surový údaj z měření. Systém přistupující k těmto datům je následně nucen rozhodovat na základě konkrétního senzoru, zda je nutné data přepočítat, či nikoli. Přestože implementace takového řešení byla provedena, do výsledné aplikace se nepromítla, protože bylo přistoupeno k drobné úpravě databáze. Byl přidán atribut *value_calculated*, který je vzhledem k návrhu databáze nadbytečný. K přidání bylo přistoupeno z důvodu kompatibility, jelikož na databázi již jsou závislé další systémy.

Dále byl vytvořen trigger (v přložených souborech), který se spouští před uložením nového záznamu do tabulky *data* a provede výpočet nutný k získání fyzikální hodnoty pro konkrétní senzor. Tento trigger pro svou komplexnost využívá další dvě obslužné funkce pro výpočet u specifických senzorů. První je pro výpočet průtoku z informace o počtu vylití sklopky s postupně se plnící nádobkou. Druhý pro výpočet průtoku z informace o výšce hladiny vody ve válci s různým počtem výtokových otvorů. Potřeba prvního byla odůvodněna nutností dotazovat se na předešlé hodnoty z měření, a následně rozhodovat, zda došlo k přetečení čítače sklopky, či nikoli. A druhá zobecněním výpočtu pro více umístěných senzorů, které se liší pouze v dílčích parametrech (jmenovitě počtem výtokových otvorů, jejich velikostí a vzdáleností ode dna, případně objemem válce).

6 Optimalizace výkonu

Je důležité mít představu o tom, co je myšleno pod pojmem "výkon". Obvykle to není pouze jedna metrika. Zvýšení rychlosti může být cílem nejzřejmějším, ale někdy je možné hledat další vylepšení výkonu, například nižší spotřebu paměti nebo menší nároky na databázi nebo síť.

V tomto směru vylepšení v jedné oblasti často přináší lepší výkon v jiné, ale ne vždy. Často může být zlepšení jednoho aspektu dokonce na úkor druhého. Například zlepšení rychlosti programu může způsobit, že bude používat více paměti. V ještě horším případě se může stát, že za honbou po rychlosti začne systému docházet přidělená paměť, a provedené optimalizace budou mít ve výsledku opačný efekt.

Z pohledu firmy začíná tato optimalizace metrikou finanční, kdy volí mezi časem programátorů, navýšením výkonu hardwaru a ztrátami vlivem odrazených zákazníků, kteří pro dlouhé čekání opustili webovou stránku (nebo jiný softwarový produkt).

V dalších podkapitolách je ukázán proces odhalování tzv. úzkých hrdel vytvořeného systému, kde je hlavní metrikou pro "výkon" doba odbavení požadavku na server.

6.1 Odhalování problémů

Obecně nejdůležitější je napsat fungující kód, jehož logické funkce jsou potřebné k vytvoření očekávaného výstupu. V této prvotní fázi vývoje softwaru není dobré podléhat přílišné a mnohdy neefektivní optimalizaci. Někdy to však nestačí na to, aby kód fungoval tak efektivně jak by mohl, nebo dokonce, jak je z pohledu uživatele únosné. V takovém případě je potřeba zlepšit výkon kódu, aniž by to ovlivnilo jeho chování, nebo jen minimálně.

Po odeslání požadavku klientem až po vrácení odpovědi procházejí informace mnoha vrstvami, a na každé z těchto vrstev je možné provést nějakou optimalizaci. Z toho důvodu je důležité ujasnit si priority a vědět, která z těchto částí má největší vliv na výsledný výkon z pohledu běžného uživatele.

Obecně lze říci, že Django nabízí mnoho možností, jak dosáhnout cíle[1], ale samotný fakt, že je něco možné, neznamená, že je to nejlepší způsob, jak dojít k výsledku.

Při rozhodování **téměř** vždy platí, že nejlepší vrstva, kde implementovat funkční kód, je ta nejnižší pohodlná. Dokumentace Djanga uvádí příklad s počítáním objektů

[1] V tomto ohledu se nedrží zásad *Zen of Python* kde je uvedeno doslova: "Měl by existovat jeden— a nejlépe pouze jeden —zjevný způsob, jak to udělat."[12] Proto při implementaci sice poskytuje značnou volnost, ale také jasná doporučení.

```
def show_toolbar(request):
    return request.user.is_superuser

DEBUG_TOOLBAR_CONFIG = {
    'SHOW_TOOLBAR_CALLBACK': "ToyBox.settings.show_toolbar",
}
```

Listing 11: Nastavení Debug Toolbaru pro zpřístupnění pouze správci

uložených v databázi. Ten je možné rozšířit a aplikovat na problematiku této práce tak, jak uvádím následovně.

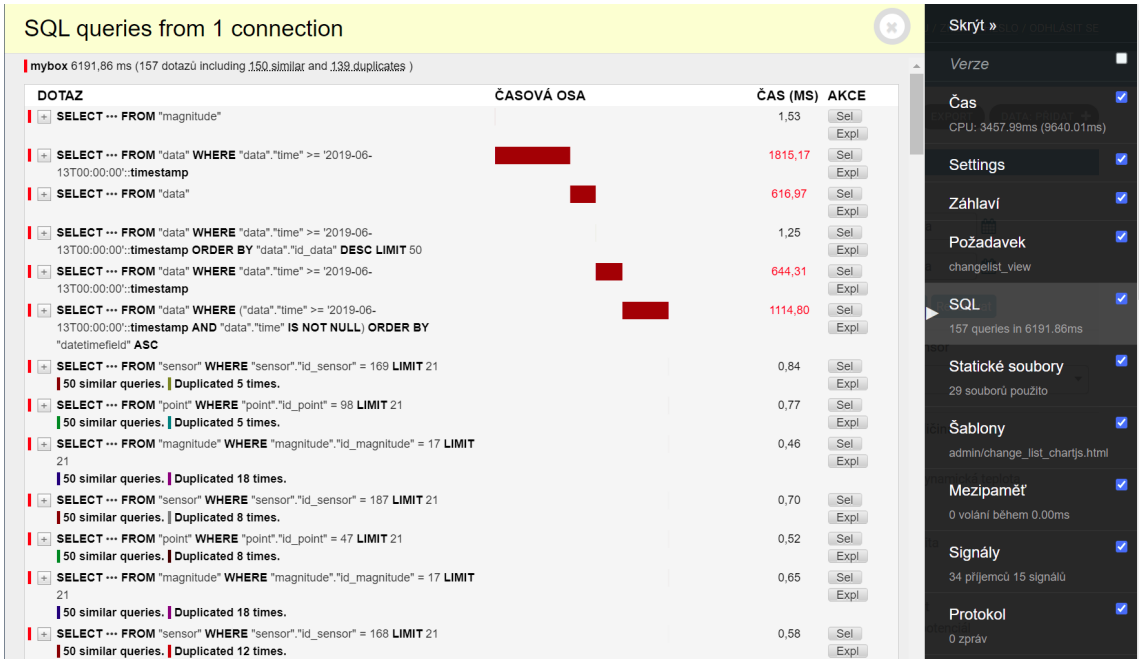
Zadáním je spočítat počet uložených záznamů z měření pro jeden konkrétní senzor. Pokud budeme postupovat v hierarchii možných implementací od klienta postupně do nižších vrstev aplikační logiky, prvním řešením bude spočítat položky pomocí javascriptu přímo na klientovi. To obnáší přenos velkého množství dat z databáze a následně na klientský počítač. Tento příklad, ač se může zdát hloupý, může přinést i řadu výhod, pokud bude následně nad daty vykonávána další logika.

Další možností je spočítat množství záznamů na úrovni šablonovacího jazyka. Takovouto, v rámci možností vhodnou, implementací by bylo `{{data\_by\_sensor|length)}}`. To je o hodně rychlejší, protože odpadá jeden náročný přenos dat na klienta, ale stále pomalé, protože šablonovací jazyk je interpretovaný Pythonem. To přináší další režijní náklady pro již tak interpretovaný jazyk Python.

O něco lepším přístupem je výpočet pomocí `len(data_by_sensor)` na úrovni Pythonu. Ještě je ale možné vyhnout se načítání dat úplně, a výsledek nechat spočítat přímo databázový engine pomocí dotazu, který vygeneruje ORM vrstva voláním `data_by_sensor.count()`. Tento přístup je zdaleka nejlepší, protože databáze jsou vysoce optimalizované, napsané pomocí efektivního jazyka, nikoli interpretovaného, jako je tomu u Pythonu, a hledání provádějí nad indexovanými daty.

Ač jsou tato základní doporučení dobrá, není dobré jen předpokládat, kde v konkrétním kódu spočívá neefektivita. Na úrovni klienta existuje celá řada bezplatných služeb, které mohou analyzovat výkon stránek. Základem jsou *nástroje pro vývojáře*, které bývají součástí internetového prohlížeče. Například v Chrome je možné je vyvolat pomocí klávesové zkratky `ctrl+shift+i`, zde na záložce *Network* Chrome zobrazuje takzvaný *waterfall* načítání stránky. Ten zobrazuje časy načítání pro jednotlivé soubory, ale chybí zde informace o časech zpracování na serveru. V první fázi, tedy po implementaci vzorové aplikace, musel prohlížeč v případě stránky se zobrazením dat čekat na první odpověď ze serveru téměř patnáct sekund. Následné stažení souborů proběhlo přibližně za 1,6 s. To není dobrý výsledek, ale je zřejmé, že začít optimalizovat webový server pro rychlejší přenos souborů k uživateli by nepřineslo zásadní zrychlení vzhledem k trvání odbavení požadavku serverem.

Aby bylo možné ladit výkon na úrovni aplikace, byl integrován oficiálně doporučený doplněk *Django Debug Toolbar* (dále jen DT). Bylo pro-



Obrázek 6.1: Debug Toolbar se zobrazením doby běhu jednotlivých SQL dotazů

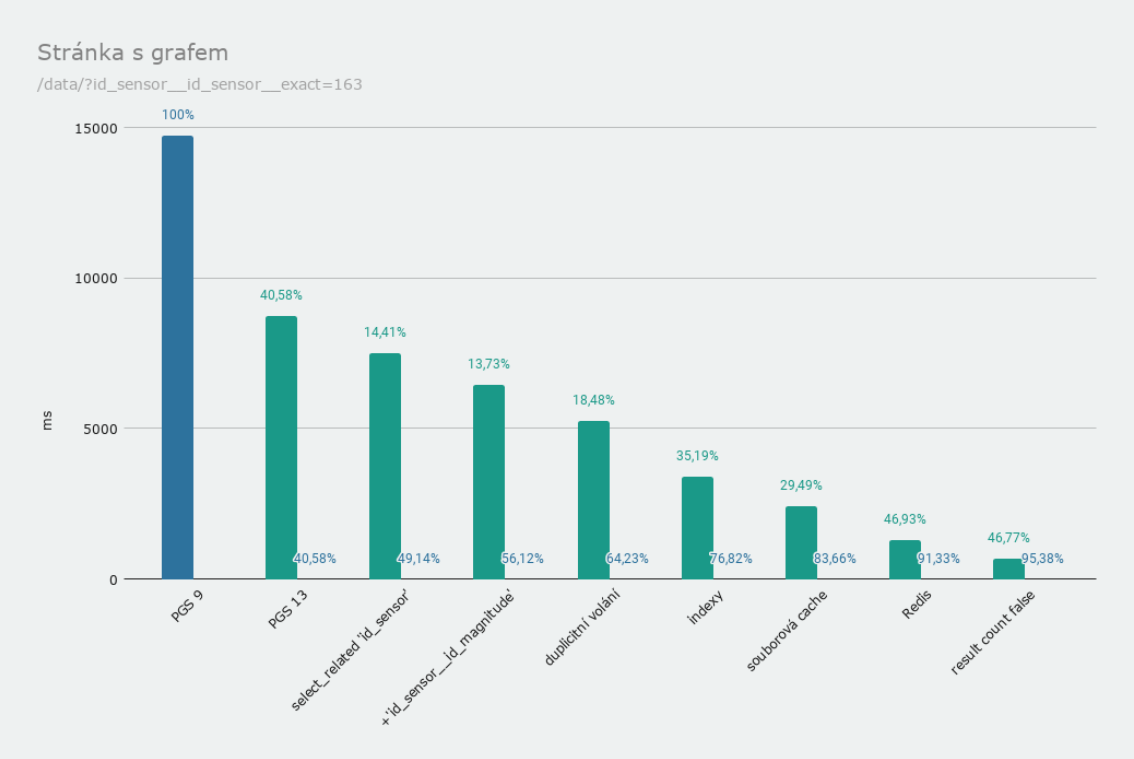
vedeno jeho nastavení pro běh na serveru v provozním prostředí. K tomu bylo zapotřebí do souboru `/etc/nginx/sites-available/gincorn` přidat nastavení hlavičky statických souborů pomocí `add_header Access-Control-Allow-Origin http://bedrichov2.tul.cz;`. Tento krok by nebyl potřebný, pokud by správu statických souborů obstarávalo samo Django. Protože na serveru by Django nikdy nemělo být spouštěno v "debugovacím" režimu, DT zapíná se pouze pokud je tento režim vypnutý a požadavek přichází z interní adresy uvedené v seznamu `INTERNAL_IPS`. Aby bylo toto chování obejito při zachování rozumné míry bezpečí (s přihlédnutím k tomu, že databáze neobsahuje citlivá data), bylo provedeno nastavení podle vzoru 11. Toto nastavení zajišťuje spouštění DT, pokud je přihlášen uživatel s právy superuživatele.[2] Obrázek 6.1 zobrazuje DT na velmi špatně optimalizované stránce. U SQL dotazů dokáže DT zobrazit také informace o plánu jejich vykonávání. Díky tomu lze získat důležité informace o tom, jestli byl použit index tabulky nebo databáze prováděla sekvenční hledání nad celou sadou dat, případně jestli toto hledání bylo paralelizované.

6.2 Provedené změny

Pomocí DT se potvrdil předpoklad, že nejslabším článkem systému jsou pomalé dotazy na databázi. Po provedeném upgradu, popsaném v kapitole 5.1, se průměrná doba trvání dotazů zkrátila přibližně o 42 %.

Pokud je vznesen dotaz na objekty, které jsou s nimi v incidenci, Django nejprve dotaz na všechny objekty a následně pro jeden každý generuje další dotazy

[2] Jde o vyšší oprávnění než oprávnění správce. Superuživatel bývá první vytvořený účet.



Obrázek 6.2: Vliv optimalizací

na odkazované objekty. V případě základní stránky s výpisem dat tak bylo generováno 114 dotazů. Opakovalo se padesát podobných dotazů na senzory, které jsou s jednotlivými daty ve vztahu, tedy do tabulky *sensor* a dále na rozměr dat, tedy do tabulky *magnitude*. Aby bylo takovému chování zabráněno, Django umožňuje předat ORM vrstvě dopředu informaci, že budou tyto objekty potřeba, a Django dotazy sjednotí pomocí spojení INNER JOIN. Objektově lze tohoto chování docílit voláním `select_related('ForeignKeyField', )`. (Pokud jde o spojení, vrací více objektů, například N:N, a nebo zpět na odkazující objekt v relaci 1:N, podobného chování lze dosáhnout pomocí funkce `prefetch_related`. Není nativně podporováno v prostředí DA.) Protože některé chování DA lze upravovat parametricky, pomocí předdefinovaných proměnných, stačí nastavit proměnnou `list_select_related = ('ForeignKeyField', )`. Přidání první tabulky v relaci se promítlo zlepšením přibližně o 14 % a snížením počtu dotazů na 64. Doplnění druhé tabulky se promítlo zlepšením přibližně o dalších 14 % a snížením počtu dotazů znovu z 50 na 14 prováděných dotazů. (Důvodem je stránkování nastavené po 50 položkách.)

Dále byla pomocí DT odhalena chyba prvotní implementace. Vytvořená třída *ChartAdmin*, která je děděna od *admin.ModelAdmin*, přepisuje metodu `changelist_view` a zde chybou návrhu docházelo ke dvojímu volání metody předka. Je s podivem, že takováto hrubá chyba nezatížila systém více a její odstranění se promítlo zvýšením výkonu téměř jen o 18 %.

Také bylo zjištěno, že databáze má vytvořeny pouze automatické indexy nad

primárními klíči.

Vytvoření indexu nad atributem *time* nemělo vliv na rychlost provádění dotazů pro stránku bez časového omezení. Takový případ však uživatelsky není příliš častý, typické použití je zobrazení grafu v definovaném časovém období. Obrázek 6.3 zobrazuje změnu po vytvoření tohoto jednoho indexu. Zobrazené měření bylo provedeno na stránce s výsledky filtrovanými podle času.

Dále byl použit index nad atributem *id_sensor*, a také společný index pro obě pole najednou. I pro stránku bez časového omezení tak šlo využít indexů, a tím došlo ke zrychlení prvního načtení testované stránky o 35 %. Obrázek 6.4 zachycuje plán vykonávání dotazu, kde je možné vidět využití společného indexu (pojmenován *time_id2_btree_idx*). Jde o poměrně kritický dotaz, jehož návratovou hodnotou je počet výsledků odpovídajících filtru. SQL dotazy využívající `count()` dasahují velmi špatných výsledků. Při použití filtru došlo v tomto případě k významnému zlepšení z přibližných 330 ms na 4,5 ms. (Pokud by filtr nebyl použit, dojde k paralelnímu sekvenčnímu procházení celé sady dat, protože databáze by indexy nemohla využít.) Všechny vytvořené indexy jsou typu *BTree*. Testovány byly i jiné, například *BRIN*, ale *BTree* dosahoval nejlepších výsledků. Vytvořené indexy jsou přidanou zátěží při vkládání nových záznamů a také mají paměťové nároky. To je však přijatelná cena za zrychlení dotazů, protože ukládání je rovnoměrně rozloženo v čase a nejsou na něj kladeny nároky na rychlost.

V kapitole 3.2 bylo zmíněno, že Django podporuje různé mezipaměťové backendy. Pro produkci je možné využít mezipaměť založenou na souborovém systému (pomalejší), vlastní operační mezipaměť (paměťově neefektivní), případně využít *Memcached* nebo *Redis*.

Nejprve byla testová souborová mezipaměť s ukládáním na úrovni šablony. Práce s mezipamětí je vidět na zkráceném kódu 12. Blok *cache* přijímá jako první parametr čas v sekundách. V ukázce je možné vidět nastavení, kdy se nová data pro graf mají načítat po uplynulé půl hodině. Obsah pro seznam jednotlivých položek jednou za dvě minuty.

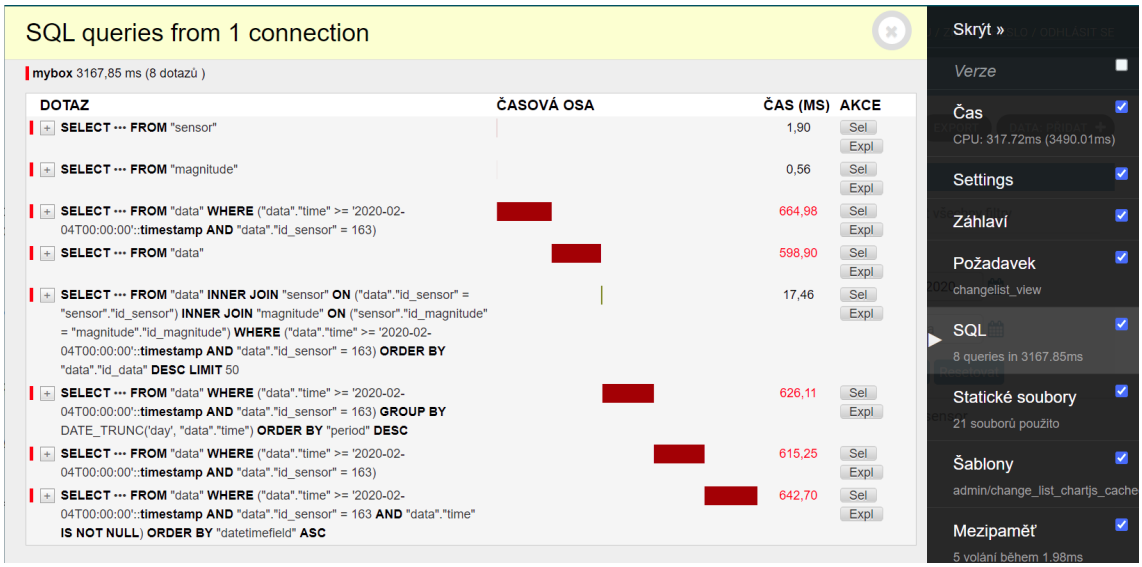
Dále následuje libovolný počet identifikátorů. V ukázce je použit identifikátor v dané šabloně začínající prefixem `admin_`, dále URL adresa společně se všemi předávanými parametry a také kód jazykové lokalizace. (Web v současné chvíli nemá vytvořené jazykové mutace, ale práce byla vytvářena tak, aby je umožňovala.) Data pro graf jsou jazykově nezávislá, z toho důvodu není potřeba je ukládat pod klíčem specifickým pro danou lokalizaci.

Využití ukládání mezivýsledků na lokální disk serveru přineslo zlepšení rychlosti načítání u vzorové stránky téměř o 30 %.

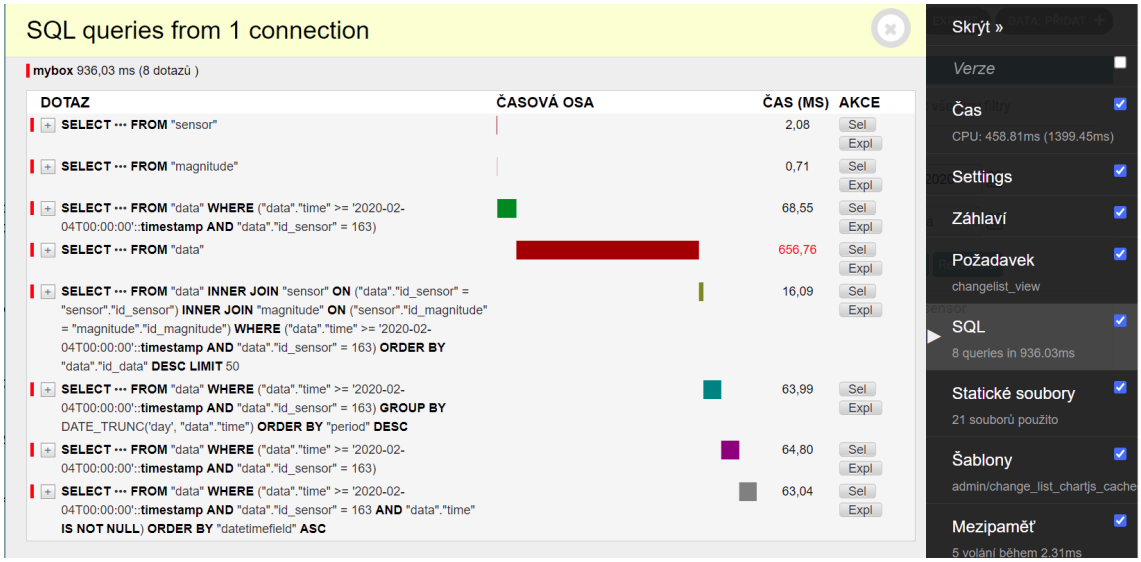
Ukládání dat grafu do mezipaměti z šablony nemělo požadovaný efekt, důvodem je volání a zpracování dat ještě před předáním šabloně. Jinými slovy - načítání dat nevyvolává příslušná šablona, ale již *View*, které šabloně data předává jako kontext. Výsledkem je, že při prvním načtení dojde k provedení dotazu na databázi, následné uložení dat do mezipaměti šablonou. Při příštím načtení jsou ale data z databáze požadována znovu, a předávána šabloně, která ale poskytnutá data nepoužije, a místo toho načítá znovu z mezipaměti.

```
{% extends "admin/change_list.html" %}
{% load static admin_list i18n cache %}
{% get_current_language as LANGUAGE_CODE %}
...
<script>
  ...
  {% cache 1800 admin_chart request.get_full_path %}
    const chartData = {{ chart_data | safe }};
  {% endcache %} {# NEFUNGUJE: data jsou požadována již ve view! #}
  ...
</script>
...
{% block content %}
  ...
  {% cache 120 admin_list_content LANGUAGE_CODE
  ↪  request.get_full_path %}
    {{ block.super }}
  {% endcache %}
{% endblock %}
...
```

Listing 12: Použití mezipaměti v šabloně



(a) Bez indexu



(b) S indexem *Btree*

Obrázek 6.3: Vliv vytvoření indexu nad atributem *time*

Spuštěné SQL

```
SELECT COUNT(*) AS "__count"
  FROM "data"
  WHERE ("data"."time" >= '2021-03-01T00:00:00'::timestamp AND "data"."id_sensor" = 163)
```

Čas

15,12289047241211 ms

Databáze

mybox

QUERY PLAN

```
Aggregate  (cost=135343.82..135343.83 rows=1 width=8) (actual time=330.555..330.673 rows=1 loops=1)
  ->  Gather  (cost=1000.00..135343.82 rows=2 width=0) (actual time=330.546..330.664 rows=0 loops=1)
        Workers Planned: 2
        Workers Launched: 2
        ->  Parallel Seq Scan on data  (cost=0.00..134343.62 rows=1 width=0) (actual time=319.109..319.109 rows=0 loops=3)
              Filter: (("time" >= '2021-03-01 00:00:00'::timestamp without time zone) AND (id_sensor = 163))
              Rows Removed by Filter: 302701
Planning Time: 0.643 ms
Execution Time: 330.787 ms
```

(a) Bez indexu

Spuštěné SQL

```
SELECT COUNT(*) AS "__count"
  FROM "data"
  WHERE ("data"."time" >= '2021-03-01T00:00:00'::timestamp AND "data"."id_sensor" = 163)
```

Čas

15,12289047241211 ms

Databáze

mybox

QUERY PLAN

```
Aggregate  (cost=5112.40..5112.41 rows=1 width=8) (actual time=4.389..4.392 rows=1 loops=1)
  ->  Index Only Scan using time_id2_btree_idx on data  (cost=0.43..5110.31 rows=839 width=0) (actual time=0.144..4.300 rows=416 loops=1)
        Index Cond: (("time" >= '2021-03-01 00:00:00'::timestamp without time zone) AND (id_sensor = 163))
        Heap Fetches: 416
Planning Time: 1.168 ms
Execution Time: 4.488 ms
```

(b) Index: `btree("time", id_sensor)`

Obrázek 6.4: Vliv vytvoření společného indexu na plánování dotazu

7 Závěr

V práci jsou uvedeny postupy vývoje a nasazení aplikace za pomoci frameworku Django, které vyžadují hlubší nastudování dokumentace.

Z existující databáze byly vyrobeny modely, které používá Django pro generování administračního rozhraní. Toto rozhraní bylo dále upraveno dle požadavků výzkumného pracoviště. Implementováno bylo také zobrazování grafů a následně probíhaly práce na zlepšení výkonu aplikace.

Bylo dosaženo zlepšení času načítání o více než 90 % oproti počáteční implementaci. Toho bylo dosaženo pomocí efektivnější práce s databází a načítanými daty, ale také zlepšením výkonu samotné databáze.

Nastudované vzorce byly aplikovány v podobě triggeru, který se spouští před vložením nového záznamu. Aby bylo možné uživatelsky spouštět výpočet i nad již uloženými hodnotami, tyto vzorce byly implementovány také pomocí objektového přístupu uvnitř aplikace *Django*.

Vzniklá aplikace bude sloužit při analýze dat a výzkumu na NTI FM a OMP CxI. Její přínos oproti původnímu řešení spočívá v grafové prezentaci dat, plynulejším chodem systému a přehlednějším interfacem.

Použitá literatura

1. HOKR, Milan. *Poster Bedrichov tunnel - overview of partner institution contributions (in Czech)*. 2016.
2. HOKR, Milan. *Měření a vyhodnocení průsaků ve vodovodním přivaděči Bedřichov*. 2017. seminář TESEUS.
3. BC. LUKÁŠ PELC. *Návrh zpracování a vizualizace dat z vodárenského přivaděče Bedřichov*. 2021. Diplomový projekt. TUL.
4. DOC. ING. MILAN HOKR, PH.D. *TUNEL 2011*. Liberec, 2014-03.
5. ING. ROMAN ŠPÁNEK, PH.D. *Uživatelská příručka: RemoteDataReceiver*. Výzkumné centrum Pokročilé sanační technologie a procesy, [n.d.].
6. *Welcome — Django MongoDB Engine* [online] [visited on 2021-05-12]. Available from: https://django-mongodb-engine.readthedocs.io/en/latest/#.
7. OIS. *Virtul | LIANE* [online] [cit. 2021-04-28]. Dostupné z: `https://liane.tul.cz/cz/uzivatel/virtul`.
8. *Psycopg – PostgreSQL database adapter for Python — documentation* [online] [visited on 2021-05-11]. Available from: https://www.psycopg.org/docs/.
9. DJANGO SOFTWARE FOUNDATION. *Security in Django – documentation* [online]. [N.d.] [visited on 2021-05-03]. Available from: https://docs.djangoproject.com/en/3.2/topics/security/#host-headers-virtual-hosting.
10. GROUP, PostgreSQL Global Development. *PostgreSQL* [PostgreSQL] [online]. 2021-05-14 [cit. 2021-05-14]. Dostupné z: `https://www.postgresql.org/`.
11. ŠPÁNEK R., M. HERNYCH, M. HOKR, P. SVOBODA, P. TYL, M. ŘIMNÁČ, J. ŠTULLER. Bedřichov Tunnel – Continual Automated Measurement Of Physical Quantities, Exploration Geophysics, Remote Sensing and Environment. *ČAAG*. 2011, pp. 73–82. ISSN 1803-1447.
12. *PEP 20 – The Zen of Python* [Python.org] [online] [visited on 2021-05-12]. Available from: https://www.python.org/dev/peps/pep-0020/.