FAKULTA INFORMAČNÍCH TECHNOLOGIÍ ČVUT V PRAZE

Zadání diplomové práce

Název:	Systém pro podporu správy chytrých fotovoltaických elektráren
Student:	Bc. Matěj Jehlička
Vedoucí:	Ing. Marian-Daniel Rolník
Studijní program:	Informatika
Obor / specializace:	Softwarové inženýrství
Katedra:	Katedra softwarového inženýrství
Platnost zadání:	do konce letního semestru 2023/2024

Pokyny pro vypracování

Cíl diplomové práce:
Cílem práce je navrhnout a vytvořit prototyp systému s webovým UI, který podporuje procesy spojené se správou chytrých fotovoltaických elektráren (dále jen FVE). Systém je svým zaměřením cílen především na správu komunitních FVE.

Kroky pro naplnění cílů diplomové práce:
1. Seznamte se s prostředím FVE a nezbytnými procesy spojené se správou takových FVE.
2. Specifikujte požadavky a zpracujte rešerši existujících řešení.
3. Navrhněte vlastní řešení a implementujte jeho funkční prototyp.
4. Berte v potaz následující požadavky:
    1. možnosti napojení na různé řídící komponenty a čidla FVE,
    2. škálovatelnost řešení.
5. Vhodně otestujte jednotlivé části aplikace.
6. Zhodnoťte výsledné řešení a navrhněte možnosti budoucího rozvoje.

Elektronicky schválil/a Ing. Michal Valenta, Ph.D. dne 10. prosince 2022 v Praze.

F8

Fakulta informačních technologií
Katedra softwarového inženýrství

Diplomová práce

Systém pro podporu správy chytrých fotovoltaických elektráren

Bc. Matěj Jehlička

leden 2024

dp.jehlicka.eu

Vedoucí práce: Ing. Marian-Daniel Rolník

Poděkování / Prohlášení

Děkuji svému vedoucímu diplomové práce, Ing. Marianu-Danielu Rolníkovi, za pomoc, čas a cenné rady, kterých se mi v průběhu psaní práce dostalo.

Prohlašuji, že jsem předloženou práci vypracoval samostatně a že jsem uvedl veškeré použité informační zdroje v souladu s Metodickým pokynem o dodržování etických principů při přípravě vysokoškolských závěrečných prací.

Beru na vědomí, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., autorského zákona, ve znění pozdějších předpisů. V souladu s ust. § 2373 odst. 2 zákona č. 89/2012 Sb., občanský zákoník, ve znění pozdějších předpisů, tímto uděluji nevýhradní oprávnění (licenci) k užití této mojí práce, a to včetně všech počítačových programů, jež jsou její součástí či přílohou a veškeré jejich dokumentace (dále souhrnně jen „Dílo“), a to všem osobám, které si přejí Dílo užít. Tyto osoby jsou oprávněny Dílo užít jakýmkoli způsobem, který nesnižuje hodnotu Díla a za jakýmkoli účelem (včetně užití k výdělečným účelům). Toto oprávnění je časově, teritoriálně i množstevně neomezené. Každá osoba, která využije výše uvedenou licenci, se však zavazuje udělit ke každému dílu, které vznikne (byť jen zčásti) na základě Díla, úpravou Díla, spojením Díla s jiným dílem, zařazením Díla do díla souborného či zpracováním Díla (včetně překladu) licenci alespoň ve výše uvedeném rozsahu a zároveň zpřístupnit zdrojový kód takového díla alespoň srovnatelným způsobem a ve srovnatelném rozsahu, jako je zpřístupněn zdrojový kód Díla.

V Praze dne

.....

Abstrakt / Abstract

Práce se zabývá doménou chytrých fotovoltaických elektráren v kontextu sdílení elektrické energie. Popisuje základní komponenty a procesy provozu takových elektráren. Dále se zaměřuje na aktuální a nově vznikající legislativu týkající se komunitní energetiky v České republice. Na základě této analýzy je v práci popsán návrh prototypu systému pro podporu správy chytrých fotovoltaických elektráren. Návrh je koncipován tak, aby bylo možné prototyp snadno rozšiřovat. Zaměřuje se jak na datovou, tak technologickou stránku systému. Dále se práce zabývá implementací navrženého prototypu. Realizovaný prototyp systému se skládá z několika mikroslužeb, které jsou napsané v jazyce Java. Implementováno je také uživatelské rozhraní pro zobrazování evidovaných informací. Díky otevřenému API a vhodně zvoleným technologiím lze evidovaná data asynchronně zpracovávat pomocí Spark Jobs. Úlohy automatického zpracování je možné definovat, mimo jiné, v programovacím jazyce Scala. Definované úlohy lze spouštět automatizovaně s využitím platformy Apache Airflow.

Klíčová slova: fotovoltaická elektrárna, komunitní energetika, mikroslužba, databáze časových řad, Java, Apache HBase, JanusGraph, Docker, Apache Spark, Apache Airflow

This thesis explores the domain of smart photovoltaic power plants in the context of electric energy sharing. It describes the basic components and processes of such power plants. Furthermore, it focuses on current and emerging legislation related to community energetics in the Czech Republic. Based on this analysis, the design of a system prototype to support the management of smart photovoltaic power plants is carried out. The design is conceived to allow easy expansion of the prototype. It addresses both the data and technological aspects of the system. The thesis also deals with the implementation of the proposed prototype. The implemented prototype system consists of several microservices written in Java. A user interface for displaying recorded information is also implemented. Thanks to an open API and appropriately chosen technologies, the recorded data can be asynchronously processed using Spark Jobs. Automatic processing tasks can be defined in the Scala programming language. Defined tasks can be automatically executed using the Apache Airflow platform.

Keywords: photovoltaic power plant, community energetics, microservice, time series database, Java, Apache HBase, JanusGraph, Docker, Apache Spark, Apache Airflow

Title translation: System for the management support of smart photovoltaic power plants

Obsah /

Tabulky / Obrázky

Kapitola 1
Úvod

Fotovoltaické elektrárny jsou v současnosti na vzestupu zájmu jak u odborné, tak i laické veřejnosti. Je tomu tak zejména kvůli snaze zemí Evropské unie redukovat svou uhlíkovou stopu a zapojit do energetických soustav větší podíl obnovitelných zdrojů, a tím postupně snižovat závislost na fosilních zdrojích energie. Důsledkem těchto kroků je i rostoucí popularita fotovoltaických elektráren mezi právnickými i fyzickými osobami. Tyto skupiny jsou stále více motivovány k implementaci tohoto obnovitelného zdroje elektrické energie skrz zajímavé dotační programy. Neopomenutelným faktem, který také přispěl k prudkému nárůstu uživatelů fotovoltaických elektráren, je stále rostoucí cena energií na evropském trhu a s ní rostoucí potřeba zajistit si alespoň částečnou energetickou soběstačnost.

Technologie nutné pro provoz fotovoltaických elektráren se stávají masově dostupné, avšak stále je potřeba při stavbě projektů myslet na návratnost investic do nich vložených. Je proto nezbytně nutné, aby investor svou pozornost věnoval kromě technicko-fyzikálním parametrům instalované fotovoltaické elektrárny (např. druh panelů, jejich náklon a počet; umístění panelů z hlediska orientace světových stran; celkový výkon atp.) i řízení všech komponent fotovoltaických elektráren. Řízení komponent fotovoltaických elektráren má přímý dopad na její efektivitu, její životnost a finanční návratnost. Příkladem efektivního řízení může být např. prodej elektrické energie distributorovi, její okamžitá spotřeba či odpojení komponent fotovoltaické elektrárny v případě hrozby zásahu blesku.

K rozšíření zájmu o fotovoltaické elektrárny mezi bytovými družstvy a SVJ přispěly novela vyhlášky o Pravidlech trhu s elektřinou a novela energetického zákona, které umožňují sdílet vyrobenou elektrickou energii z jednoho obnovitelného zdroje mezi více odběrných míst (resp. bytových jednotek) – tzv. komunitní energetika. Práce se také věnuje procesu spojenému s tímto rozdělováním.

Vývoj systému pro správu chytrých fotovoltaických elektráren je zásadní v době, kdy se komunitní energetika teprve začíná prosazovat v praxi. Práce tedy cílí na vytvoření funkčního prototypu systému s webovým uživatelským rozhraním, který bude sloužit pro podporu procesů spojených se správou chytrých fotovoltaických elektráren, se zaměřením na komunitní fotovoltaické elektrárny. Především se jedná o sběr dat a jejich poskytování v unifikovaném formátu pro další zpracování. Rozhraní systému pro práci s procesy musí být dostatečně obecné, aby přidávání a modifikování procesů bylo možné provádět dynamicky. Spouštění procesů nesmí být závislé na interakci s uživatelem.

Práce je strukturována do několika kapitol. V první části (kapitola číslo 2) se práce zabývá analýzou a popisem samotné struktury fotovoltaické elektrárny, novou a stávající legislativou spojenou s obnovitelnými zdroji elektrické energie (se zaměřením především na fotovoltaické elektrárny a komunitní energetiku). Dále jsou zde definovány požadavky na navrhovaný systém a jsou představena vybraná existující řešení, jejichž funkcionality se alespoň z části kryjí s těmito požadavky. V závěru kapitoly je popsána analýza zpracovávaných dat.

Další kapitola (kapitola číslo 3) je zaměřena na návrh samotného řešení. Navazuje na provedenou analýzu a dodržuje všechny principy a poznatky, které jsou v ní popsány. První část kapitoly popisuje výběr a zvolení architektury, technologií, programovacího jazyka a frameworku. Dále je popsán způsob ukládání dat, která bude systém evidovat. V poslední části kapitoly jsou popsány návrhy jednotlivých mikroslužeb a uživatelského rozhraní.

Na návrh navazuje kapitola 4, popisující provedenou implementaci. Kapitola popisuje – datový model a ukládaní dat, implementace databáze časových řad, implementace systému (mikroslužeb), implementace uživatelského rozhraní, způsob definice a využití automatických úloh, lokání nasazení pro potřeby vývoje a simulaci provozu. Popis implementace se zaměřuje zejména na doménově specifické problémy a řešení, která nejsou obecně známá. Kapitola je uzavřena popisem budoucího rozvoje systému.

Předposlední kapitola (kapitola číslo 5) se zaměřuje na testování implementace. Obsahuje popis metod testování softwaru a technologií, které je možné pro testování použít. Následně popisuje testování provedené implementace s využitím těchto technologií a metod. I zde je text doplněn ukázkami kódů.

Výstupy práce jsou diskutovány v jejím závěru (kapitole 6). V něm jsou shrnuty všechny provedené kroky a výsledné zhodnocení naplnění cílů a definovaných požadavků.

Kapitola 2
Analýza

Základní prerekvizitou vývoje softwaru je tvorba analýzy problémové domény a zajištění důsledného splnění cílů projektu. Toho lze dosáhnout několika metodikami, které jsou využívány jak na straně samotného softwarového inženýrství, tak projektového řízení. Vzhledem k zaměření této práce zde bude kladen větší důraz na část týkající se softwarového inženýrství. Proto bude pro modelování, a tam kde je to vhodné, zvolena notace UML umožňující vytvoření srozumitelných diagramů pro potřeby softwarového návrhu a následně i implementace. Pomocí této notace lze mimo jiné zachytit funkční požadavky, nefunkční požadavky a případy užití. Dále pomocí relační matice lze popsat vztahy mezi nimi. Těmito informacemi se vymezí jasné cíle pro provedení návrhu a implementace, které reflektují cíl práce. Jednotlivé části implementace poté budou mapovány na funkční požadavky a případy užití. Díky tomu tak bude možné vyhodnotit, zda byly naplněny všechny případy užití (business požadavky) na systém, a tedy splněny cíle práce.

Ačkoliv záměrem této práce není návrh fotovoltaické elektrárny (FVE), ani jiných jejích součástí, přesto je nutné popsat alespoň její základní komponenty, procesy a chování. Díky tomu bude možné identifikovat prvky FVE, které dokážou při vlastním provozu poskytnout relevantní data, jež mohou být následně vytříděna, ukládána a dále zpracovávána.

Provedená analýza popisuje typy FVE, jejich základní komponenty a jejich obecné možnosti pro zefektivnění provozu. Tyto informace jsou poté dány do kontextu s aktuálním energetickým zákonem ovlivňujícím provoz FVE a s vyhláškou upravující pravidla trhu s elektřinou. Legislativní rámec této problematiky je velice dynamický. Obě tyto legislativy prošly v době psaní této práce změnami, které se týkaly zavedení komunitní energetiky. V této kapitole je uveden příklad sdílení vyrobené elektrické energie v rámci bytového domu, který demonstruje jednu z těchto novel. Stejný příklad je použit pro demonstraci v kapitole zabývající se návrhem i kapitole zabývající se implementací jako proces, který lze v systému podporovat.

V dalších částech této kapitoly jsou definovány požadavky na navrhovaný prototyp systému. Získaná specifikace vyplývající z analýzy je následně porovnána s již existujícími řešeními, jejichž funkcionalita se alespoň z části kryje s vytvořenými požadavky. U těchto systémů je vždy zhodnoceno, zda-li jsou vhodnými konkurenty systému navrhovaného v této práci, případně jaké nedostatky nebo vylepšení oproti němu mají.

V poslední části této kapitoly je analyzována datová část navrhovaného systému. V návaznosti na vzorový příklad sdílení elektrické energie v bytovém domě, uvedený v sekci 2.5.2, je rozebrána povaha těchto dat. Tato analýza bude sloužit jako podklad pro návrh datového modelu, případně volbu vhodného datového úložiště a souvisejících podpůrných softwarových technologií pro uvažovaný systém.

2.1 Fotovoltaické elektrárny

Fotovoltaické elektrárny se řadí do tzv. obnovitelných zdrojů elektrické energie (OZE), které jsou podle zákona o životním prostředí definovány následovně: *Obnovitelné přírodní zdroje mají schopnost se při postupném spotřebovávání částečně nebo úplně obnovovat, a to samy nebo za přispění člověka. Neobnovitelné přírodní zdroje spotřebováváním zanikají.* [1]

Jedná se o přímý způsob výroby elektrické energie ze slunečního záření (zachycováním fotonů). Jednou z hlavních výhod této výroby je nezávislost na externí mechanické energii, elektrické energii apod. Další výhodou je to, že je možné provozovat takovouto výrobu téměř na libovolném místě, protože instalace solárních panelů nevyžaduje extrémně specifické podmínky, jako je tomu u jiných OZE (např. výroba elektrické energie z větru nebo vody). Tyto výhody jsou však vyváženy malou účinností a celkově nízkými výkonovými parametry samotné výroby (efektivita přeměny se v současné době u fotovoltaických článků pohybuje v rozmezí 13–19,3 % [2]).

2.1.1 Provozní režimy

Fotovoltaické elektrárny lze provozovat ve dvou základních režimech, resp. ve třech, kde poslední vzniká kombinací dvou základních. Prvním základním režimem je *ostrovní režim* (off-grid), ve kterém je celý sytém nezávislým (samostatně fungujícím) celkem nepřipojeným do distribuční sítě. Dále je možné FVE provozovat v *síťovém režimu* (on-grid), ve kterém je FVE připojena do distribuční sítě přes elektroměr distribuční společnosti. [3] Schematicky je zapojení FVE v těchto dvou režimech zobrazeno na obrázku 2.1.

Dále je možné FVE používat *hybridním způsobem*, v rámci kterého bude systém obsahovat akumulátor a zároveň bude připojen do distribuční soustavy. Vyrobená elektřina tak nemusí být spotřebována ihned, zároveň přebytky nemusejí být ihned prodávány distributorovi, ale lze je uložit do akumulátoru. Rozhodování, zda bude elektřina prodávána, nebo ukládána, může být řízeno ručně nebo automatizováno. *Hybridní způsob* provozu má také výhodu v případě výpadku elektrické energie, kdy je možné dodávat např. do domu elektrickou energii uloženou v akumulátoru, stejně jako v případě *ostrovního systému*. [4]

Dodávce elektrické energie z akumulátoru při výpadku se říká *backoff* a nelze takto napájet spotřebiče s velkou spotřebou, např. spotřebiče pro ohřev vody. Tato možnost bývá využívána především pro zařízení podporující životní funkce nebo pro zátěže s malou spotřebou, jako jsou zabezpečovací zařízení nebo osvětlení.

Ukládání elektrické energie do akumulátoru není jediný možný způsob, jak přebytečnou energii spotřebovat. Vzhledem k snadné převoditelnosti elektrické energie na tepelnou je např. možné z přebytků ohřívat vodu v bojleru nebo bazénu. Další možností může být dobíjení elektromobilu (kde fakticky jde o ukládání elektrické energie do akumulátoru, jen takového, který není součástí FVE). Možností je tedy několik a vždy záleží na konkrétní instalaci FVE, jak bude s vyrobenou energií nakládat.

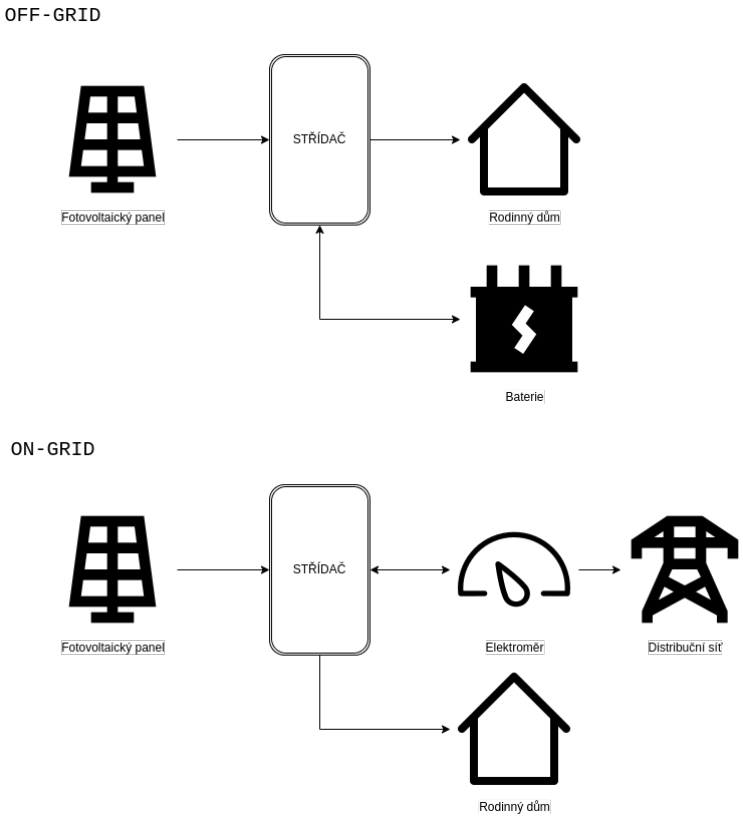
Ostrovní režim

Ostrovní systémy jsou používány tam, kde není dostupná infrastruktura pro distribuování elektrické energie. Zároveň je ale nutné, aby zátěž, která odebírá energii z takového zdroje, byla malá [3]. Jinak nemůže být takovýto systém efektivní. Výroba elektrické energie je totiž nestabilní, a tak v době velkých přebytků je potřeba energii uschovat do již zmíněného akumulátoru. Zároveň pokud výroba kolísá opačným směrem, tedy

energie je vyráběno méně, slouží akumulátory k pokrytí tohoto nedostatku. Další možností řešení tohoto nedostatku, který již nespadá do obnovitelných zdrojů, je zapojení do celého systému elektrocentrálu vyrábějící elektrickou energii z fosilních paliv [5]. Kromě toho, že takto vyrobená elektrická energie není ekologická, je navíc drahá [5]. Kvůli finanční nákladnosti a relativně nízkým možnostem pro uložení elektrické energie je provozování takovéto elektrárny (např. pro bytový dům se čtyrčlennou rodinou) bez změny návyků ve spotřebě elektrické energie nereálné.

Síťový a hybridní režim

V druhém způsobu zapojení, *síťový režim*, je FVE oproti *ostrovním systémům* připojena k distribuční síti [3]. Pokud vyráběný výkon nestačí pokrýt okamžitou spotřebu zátěže, je možné využít akumulátory (stejně jako v případě *ostrovních systémů*), ale také je možnost zbytek elektrické energie dokoupit od distributora. Naopak, pokud je vyráběna elektrická energie, která není okamžitě spotřebována, není nutné ji ukládat do akumulátorů, ale je možné prodat ji distributorovi. Taková elektrárna, která je napojena na distribuční síť a zároveň je možné ji provozovat nezávisle, je také nazývána jako *hybridní*.



Obrázek 2.1. Konceptuální schéma zapojení FVE on-grid i off-grid

2.1.2 Komponenty

Žádná fotovoltaická elektrárna se neobejde bez dvou základních komponent: fotovoltaických panelů a střídače. Jejich zapojení ve FVE je zachyceno na schématu porovnání provozních režimů na obrázku 2.1. Zjednodušeně lze konstatovat, že panely elektrickou energii vyrábějí v surové podobě a střídač ji přetváří do podoby, ve které ji lze využívat

obvyklým způsobem. Další doplňkovou komponentou, která však není nutná, je akumulátor pro ukládání přebytků elektrické energie a pokrytí poklesů ve výrobě. Součástí elektrárny mohou být i další prvky (např. senzor teploty) které slouží k monitoringu, případně vylepšují její parametry. Nejedná se už ale o kořenové součásti, bez kterých by nebylo možné elektrárnu provozovat.

Fotovoltaický panel

Jednou ze zásadních komponent fotovoltaické elektrárny je fotovoltaický panel. Ten přeměňuje solární energii na elektrickou, generuje stejnosměrné napětí. Jedná se o polovodičové součástky, které se liší především ve zpracování křemíku. Existují dva základní typy fotovoltaických článků: monokrystalický a polykrystalický.

Fotovoltaický panel je složen z několika fotovoltaických článků, které jsou sério-paralelně zapojeny tak, aby dodávaly požadované stejnosměrné napětí. Ve fotovoltaických elektrárnách jsou poté panely (složené z několika článků) zapojovány sériově do tzv. stringů [6]. Tímto řetězením se dosahuje optimálního napětí pro provoz s daným typem střídače. Pro zvýšení výkonu FVE se jednotlivé stringy zapojují paralelně – navyšování proudu.

Každý fotovoltaický panel má optimální pracovní bod, označovaný zkratkou MPP (Maximum Power Point), ve kterém dodává nejvyšší výkon. Tento pracovní bod vychází z voltampérové charakteristiky fotovoltaického panelu. Pracovní bod je ovlivňován několika faktory (jako je teplota panelu, míra jeho osvitu atd.). Pro dosažení maximálního energetického zisku se musí parametry napětí a proud regulovat – výstupy panelů se musejí zatěžovat. Za tuto konfiguraci jsou ve fotovoltaických elektrárnách zodpovědné střídače. [6]

Fotovoltaické panely se nachází ve venkovním prostředí, jejich výkon je tak determinován vnějším okolím, zejména počasím. V letním období jsou panely náchylné zejména na pyl, ten však jde smýt vodou. V zimním období je obecně intenzita slunečního záření nižší. Zároveň může být výkon snížen s přibývajícím sněhem. Zatímco panely, které jsou umístěné v určitém sklonu, jsou do jisté míry „samočisticí“, panely, které jsou instalované vodorovně, jsou na znečištění náchylné nejvíce. [7]

Střídač

Druhou komponentou, která je důležitá pro připojení solárních panelů do distribuční sítě, resp. libovolný provoz FVE, je střídač (lze jej nazývat také měničem). Ten přeměňuje stejnosměrný proud z fotovoltaických panelů na střídavý, v případě připojení do zmíněné distribuční sítě na $U_{ef} = 230/400V$, $f = 50Hz$. Zmíněný převod musí být co nejvíce efektivní, aby nedocházelo k přílišným ztrátám vyrobené energie [8]. Efektivita přeměny je závislá jak na kvalitě, tak na typu střídače. Každý střídač má rozsah optimálního napětí, ve kterém je jím prováděná transformace elektrické energie nejefektivnější [8].

Pro maximalizaci výkonu fotovoltaických panelů používají střídače tzv. maximum power point tracking (MPPT). Sledují bod maximálního výkonu panelů, který se dynamicky mění v závislosti na intenzitě osvitu a teplotě (bez přístupu k datům o těchto veličinách) [6]. Fakticky se jedná o změnu výstupního odporu střídače a je hledáno takové napětí a proud, při kterém dodává panel nejvyšší výkon [8]. Jeden střídač tímto nastavením ovlivňuje vždy všechny připojené panely (případně pouze panely v jednom stringu, má-li jich připojených více). Tato funkce není standardní pro všechny střídače na trhu. Další dělení střídačů je podle toho, zda mohou umožňují připojení k distribuční síti. Ty, které tuto možnost nemají, operují v tzv. *ostrovním režimu* (2.1.1). V něm

se veškerá vyrobená energie ihned spotřebovává a případné přebytky se ukládají do akumulátorů. Střídače, které mohou být napojeny na distribuční síť, mohou případné přebytky elektrické energie odevzdat do distribuční sítě nebo, stejně jako předchozí, je ukládají do akumulátorů. Tyto střídače operují v tzv. *hybridním režimu* (2.1.1). Další rozdělení střídačů je podle toho, jak distribuují vyrobený výkon.

Existují střídače, které výkon rozdělují symetricky do všech třech fází a střídače, které umožňují vyrobený výkon rozdělovat asymetricky [9]. Způsob distribuce vyrobeného výkonu může mít přímý vliv na návratnost investice, zejména kvůli způsobu účtování vyrobené elektrické energie. Způsob účtování se však aktuálně nachází v přelomové fázi.

V prosinci 2023 byla přijata novela energetického zákona označovaná jako Lex OZE II. Ta upravuje, mimo jiné, způsob měření dodané elektrické energie do distribuční sítě. Měření spotřeby a distribuce tak nebude prováděno fázově (asymetricky), ale součtem všech tří fází (symetricky). Novela nabude účinnosti 1. 7. 2024. [10]

Do července 2024 je v České republice měřena spotřebovaná energie asymetricky, tzv. po fázích. Tedy každá fáze je vyhodnocována zvlášť a velký přetok na jedné fázi tak neznamená „energii zdarma“ na fázi jiné. Pokud tedy je veškerá vyrobená energie dodávána do fáze L1 a na fázi L2 je zapnut spotřebič, který svou spotřebou nepřekračuje aktuálně vyráběný výkon, i tak je nutné veškerou elektrickou energii potřebnou pro jeho provoz nakoupit od dodavatele, zatímco všechna vyrobená energie je prodána. Toto chování je neefektivní a nevede ani k žádné finanční úspoře. [11]

V malé FVE je zpravidla pouze jeden střídač. U větších fotovoltaických farem se panely spojují do tzv. stringů (panelů zapojených do série), kde každý z nich má svůj střídač [6]. Toto zapojení je efektivnější, protože není nutné hledat nejúčinnější pracovní bod pro všechny panely, ale pouze pro jejich podmnožinu.

Jelikož existuje nespočet výrobců a druhů střídačů, nelze jednoznačně určit unifikované rozhraní, přes které je lze ovládat. Stejně tak nelze určit, jakým způsobem z nich lze získávat data. Některé střídače poskytují HTTP API (např. od značky GoodWe [12] nebo Growatt [13]). Většina však umožňuje připojení přes sběrnici ModBus. V závislosti na typu zařízení lze přes tuto sběrnici číst různé provozní data střídače a ovládat jeho stav. Zpravidla však všechny střídače poskytují informaci o aktuálním výkonu, který do nich přitéká (tedy aktuálním vyráběném výkonu pomocí fotovoltaických panelů), a výkonu zátěže (tedy aktuálním odebíraném výkonu). U vícefázových střídačů mohou být výkony měřeny zvlášť na jednotlivých fázích. Tyto informace budou jedny z hlavních, které budou v systému zpracovávány.

Wattrouter

Wattrouter je zařízení pro směrování toku elektrické energie [14]. Paralelou v počítačových sítích by mu byl směrovač, který rozhoduje, kam bude packet předán. V zapojení FVE je zbytnou komponentou a v některých případech je jeho funkcionalita přímo nahrazována střídačem. Určuje, na jaký jeho výstup bude dodána elektrická energie, která mu přichází na vstupu [14]. Tímto způsobem lze například řídit, zda má být vyrobená elektrická energie lokálně spotřebována, uložena do baterií nebo má být prodána distributorovi. Lze tak maximalizovat spotřebu elektrické energie přímo v místě její výroby. Přebytky lze spotřebovávat např. pro ohřev vody nebo nabíjení baterií. Není nutné se omezovat pouze na jedno zařízení, ale lze tímto způsobem směrovat vyrobenou elektřinu dle potřeby mezi několika zátěžemi.

Akumulátor

Akumulátory v systému FVE slouží k ukládání přebytků energie, která byla vyrobena a nebyla přímo spotřebována, případně k vyrovnání rozdílu mezi spotřebou a výrobou. Sleduje se u nich několik základních parametrů, zejména: kapacita, výkon, životnost (počet nabíjecích cyklů). Kapacita udává, kolik energie lze do akumulátoru uložit. S rostoucí kapacitou však roste i cena. Výkon určuje, jaký maximální výkon lze z akumulátoru dostat, závisí na maximálním vybíjecím proudu akumulátoru. [15]

2.2 Řízení fotovoltaické elektrárny

V sekci 2.1.1 byly popsány způsoby provozu fotovoltaické elektrárny. Pokud je elektrárna provozovaná v tzv. *hybridním režimu*, tedy vyrobenou energii umí jak dodávat do distribuční sítě, tak ukládat do akumulátoru, je nutné vybrat, který z těchto dvou režimů bude používán. V zásadě mohou být dva cíle tohoto výběru. Jedním je získat co největší soběstačnost, tedy odebrat od distributora co nejméně elektrické energie. Druhým cílem je maximalizovat finanční úsporu za odběr elektrické energie, což nutně neznamená odebrat od distributora co nejméně elektrické energie.

Přebytečnou energii lze prodávat za předem domluvené fixní ceny a nebo za tržní cenu na spotovém trhu [16]. Cena na spotovém trhu odráží aktuální poptávku a nabídku na trhu. V čase, kdy FVE vyrábí nejvíce energie (přes poledne, kdy slunce svítí nejvíce), je cena energie zpravidla nejnižší, a naopak ráno a večer je cena nejvyšší.

Pro zajištění co nejmenších nákladů za elektrickou energii je tedy nutné správně volit mezi prodejem, spotřebováním, uložením vyrobené energie a nákupem od dodavatele. Jak bylo popsáno v sekci 2.1.2, přepínání mezi těmito stavy je řízeno střídačem (případně wattrouterem), proto je možné stav řídit i dálkově (strojově). Rozhodování, v jakém režimu má kdy FVE pracovat není univerzální a vždy musí respektovat konkrétní instalaci. Do rozhodování vstupuje několik faktorů: výkupní cena elektrické energie, nákupní cena elektrické energie, množství vyráběné energie, predikce výroby elektrické energie. V případě automatizace je tedy nutné brát v potaz všechny tyto parametry, což vede k velice komplexnímu řešení.

2.3 Energetický zákon

V České republice upravuje podmínky podnikání v energetických odvětvích zákon č. 458/2000 Sb., o podmínkách podnikání a o výkonu státní správy v energetických odvětvích a o změně některých zákonů (energetický zákon) [17]. Pokud je výrobna (v tomto případě FVE) připojena k distribuční síti, je nutné mít jako provozovatel licenci od Energetického regulačního úřadu (ERÚ). Výjimkou jsou takové instalace, které nepřesahují výkon 50 kW a vyrobená energie je určena pro vlastní spotřebu.

Tyto změny byly provedeny na základě přijetí zákona č. 19/2023 Sb. [18] (označovaného jako Lex OZE I) a nabyly účinnosti dne 24. 1. 2023. Do té doby byl rozhodující výkon 10 kW. Tento limit stále platí pro provozování tzv. mikrozdrojů. To jsou takové zdroje, které jsou připojeny do distribuční sítě, ale není jim umožněno prodávat přetoky. Veškerá energie tak musí být spotřebována v místě výroby. Zároveň tato novela přinesla i změny ve stavebním zákoně, kde pro malé obnovitelné zdroje do stejného výkonu 50 kW není potřeba rozhodnutí o umístění stavby ani územní souhlas. Pokud výrobna nezasahuje do nosných konstrukcí budovy a nebo nemění její způsob užívání, není potřeba pro takové změny stavební povolení ani ohlášení stavebnímu úřadu. [19]

Všechny tyto změny přispívají k jednoduššímu a levnějšímu vybudování obnovitelných zdrojů energie a dosažení energetické soběstačnosti budov. Pro provozovatele se také sníží náklady za nákup elektrické energie od dodavatele.

Kromě této novelizace byla také schválena další novela – změna energetického zákona, díky které je možné provozovat tzv. komunitní energetiku (sdílet vyrobenou elektrickou energii s využitím veřejné distribuční a přenosové soustavy). Této problematice se podrobněji věnuje sekce 2.5.

2.4 Nákup a prodej elektrické energie

Společně se změnami v energetickém zákoně došlo také ke změnám ve vyhlášce o Pravidlech trhu s elektřinou [20]. Ta nově od 1. 1. 2023 umožňuje identifikovat jedno odběrné místo dvěma EAN kódy. Tento kód je jednoznačným identifikátorem místa a pro každý takový může být přidělen nejvýše jeden distributor. Tato změna umožňuje zákazníkovi mít zvlášť identifikátor pro nákup a zvlášť pro prodej elektrické energie – nemusí tak prodávat elektrickou energii stejnému distributorovi, od kterého energii odebírá. Dříve tato možnost nebyla bez licence možná.

Elektrickou energii lze od distributora nakupovat buď za fixní, nebo tržní cenu. S prodejem elektrické energie je to složitější. Distributoři vykupují přebytky buď za fixní, nebo velkoobchodní cenu. Kromě výkupu přebytků elektrické energie nabízejí také tzv. virtuální baterie. Fakticky se jedná také o výkup elektrické energie, ale stejný výkon, který byl do sítě dodán, lze zdarma odebrat zpět. Ne vždy to je však zdarma úplně. Zdarma je elektřina samotná, ale od distribučního poplatku a dalších poplatků spojených s odběrem není odběratel osvobozen. Navíc tato služba bývá zpoplatněna paušální částkou. Tato možnost je lepší pro uživatele, kteří nechtějí příliš řešit, jak s vyrobenou elektřinou naloží, pouze chtějí snížit celkovou platbu za elektrickou energii. [16]

2.5 Komunitní energetika

Komunitní energetika je přístup k výrobě a distribuci energie (především) z obnovitelných zdrojů, který není závislý na žádném centrálním subjektu. Tímto konceptem lze podpořit decentralizaci produkce energií. Dále bude uvažována komunitní energetika v kontextu výroby elektrické energie, zejména pak FVE.

Komunitní elektrárny jsou vlastněny, provozovány a spravovány skupinami subjektů (komunitou). Každý subjekt využívá část vyrobené energie pro svou potřebu. Tím snižuje množství odebrané elektrické energie z veřejné distribuční sítě. Motivací pro vytváření energetických komunit však není jen úspora finančních prostředků. Roli zde hraje také environmentální faktor. Protože není nutné dodávat tak velké množství elektrické energie z centrálních elektráren, snižují se ztráty vznikající při přenosu energie. Zároveň komunitně vyrobená energie pochází z obnovitelných zdrojů. Další výhodou je větší energetická nezávislost komunity.

Ke sdílení vyrobené elektrické energie využívají komunity veřejnou distribuční a přenosovou soustavu [21]. To však vede k problém na straně distributorů, kteří tyto soustavy spravují. U obnovitelných zdrojů nelze zcela regulovat výrobu. Ta je ovlivněna okolními vlivy (v případě FVE slunečním svitem). Je tedy nutné monitorovat stav distribuční soustavy a vhodně mu přizpůsobovat dodávky z centrálních elektráren, případně dynamicky měnit zátěž sítě, aby byla zajištěna stabilita sítě. V budoucnu k tomu může napomoci další chystaná novela energetického zákona, označovaná jako Lex OZE III [22]. Ta by měla být zaměřena na akumulaci a agregaci flexibility. V praxi by se mělo

jednat o automatické ovládání zátěže v síti nízkého napětí. U domácností se může jednat např. o tepelná čerpadla, bojlery apod. [23]

2.5.1 Legislativa

Od 1. 1. 2023 je možné v České republice vytvořit spolupracující skupinu zákazníků, odběratelů elektrické energie, kteří budou mít společnou výrobnu elektrické energie a mohou se o ni dělit. Novela vyhlášky o Pravidlech trhu s elektřinou [20] od tohoto data zavádí úpravu, jakým způsobem lze rozdělit vyrobenou elektřinu v bytovém domě mezi jeho obyvatele.

Každému zákazníkovi, který se rozhodne v bytovém domě účastnit této „specifické formy sdílení“ zůstávají všechna práva, například právo na volbu a změnu dodavatele elektřiny. Navíc má tento zákazník možnost si zvolit v jakém poměru bude v rámci spolupracující skupiny zákazníků spotřebovávat elektřinu, vyrobenou ve společné výrobně elektřiny, většinou solární elektrárně (například doplněné o systémy ukládání vyrobené elektřiny), instalované obvykle na střeše bytového domu. Na takto spotřebované elektřině z vlastní výroby uspoří zákazníci obchodní i regulovanou platbu vztaženou na MWh, tedy na objem spotřebované elektřiny. Spotřebu vyrobené elektřiny a záznam průběhu, bude provádět příslušný provozovatel distribuční soustavy (dále jen „PDS“), který hodnoty o vyrobené elektřině zaznamená, zpracuje, vyhodnotí a následně předá operátorovi trhu a obchodníkovi ke zúčtování každému ze zákazníků. [24]

Před touto novelou toto nebylo možné a vyrobená elektrická energie mohla být buď spotřebována na jednom odběrném místě, nebo přes něj prodána distributorovi. Touto novelou je umožněno, že subjekty, které obývají či jinak sdílí objekt, ve kterém se nachází více odběrných míst, mohou instalovat sdílenou výrobnu elektrické energie a užívat ji jako vlastní. Doposud musela být jedna fotovoltaická elektrárna spjata s právě jedním odběrným místem. Pokud chtělo více odběrných míst sdílet elektřinu, muselo pro tento účel vytvořit nové, speciální, odběrné místo (např. pro celý bytový dům).

Dne 1. 12. 2023 byla poslaneckou sněmovnou schválena další novela energetického zákona (tzv. Lex OZE II), která umožňuje vytvářet skupiny sdílení. Původní znění bylo zveřejněno Ministerstvem průmyslu a obchodu (MPO) 24. 2. 2023, s plánovanou účinností pro sdílení elektřiny k 1. 7. 2023. Kvůli připomínkování a průtahům v legislativním procesu se účinnost pro sdílení posunula o rok, až na 1. 7. 2024. Samotná novela nabyla účinnosti 1. 1. 2024. [10]

Komunity mohou sdílet elektrickou energii napříč distribučními soustavami. Výrobna elektrické energie tak nemusí být fyzicky spjata s odběrnými místy využívající elektrickou energii, kterou vyrobila. Oproti předchozí novele totiž odpadá nutnost fyzické blízkosti a zapojení jednotlivých odběrných míst do jedné rozvodné skříně. Každý zákazník může být součástí pouze jedné skupiny sdílení, přičemž skupina sdílení může být dvou typů. Základním rozdílem mezi nimi je maximální možný počet členů a územní omezení, na kterém mohou elektrickou energii sdílet.

Prvním typem skupiny sdílení je tzv. *aktivní zákazník* [25]. Jedná se o zákazníka, který vyrábí elektrickou energii a přetoky dodává do distribuční sítě. Vyrobenou přebytečnou energii poté může sdílet s až deseti dalšími zákazníky, kteří mohou být kdekoliv na území České republiky.

Druhým typem je *energetické společenství* [25], které je zaměřeno na sdílení většího množství energie větší skupinou zákazníků. Novela umožňuje společenství mít až jeden tisíc členů, kteří mohou sdílet vyrobenou energii na území maximálně tří sousedících obcí s rozšířenou působností. Všichni členové se budou moci společně podílet na výstavbě větších elektráren, nejen solárních, ale i jiných, které využívají k výrobě

obnovitelné zdroje (např. větrné elektrárny). To vede k rozdělení nákladů na výstavbu a zvýšení dostupnosti takovýchto výroben.

Územní omezení a omezení na počet členů je u *energetického společenství* pouze dočasným řešením, které by mělo skončit v červenci roku 2024. Do té doby bude totiž Elektroenergetické datové centrum zajišťující sdílení elektrické energie (EDC) provozováno v prozatímním režimu. Omezení, které bude přetrvávat i po ukončení prozatímního provozu, je možnost jednoho odběrného místa být členem pouze jedné skupiny sdílení. [10]

2.5.2 Sdílení energie v bytovém domě

Jak již bylo zmíněno, od 1. 1. 2023 (kdy vešla v účinnost novela Lex OZE I) lze jednoduše sdílet vyrobenou energii v bytovém domě. Dokonce není potřeba souhlas všech bytových jednotek. Sdílení se může účastnit pouze část domu. Podmínkou však je, že elektrická energie bude sdílena bez využití distribuční soustavy, tedy odběrná místa musejí být fyzicky propojena v jedné hlavní pojistkové skříni. Všechna energie, která bude dodána do distribuční sítě, bude brána jako dodávka, a ne energie pro sdílení. [24]

Sdílení s využitím distribuční a přenosové soustavy (sdílení podle novely Lex OZE II) zde nebude popisováno. Stejně tak se mu nebude konkrétně věnovat žádná další část práce. Z hlediska rozpočtu vyrobené energie, a tedy monitoringu, se jedná o stejný proces jako sdílení, které tyto sítě nevyužívá.

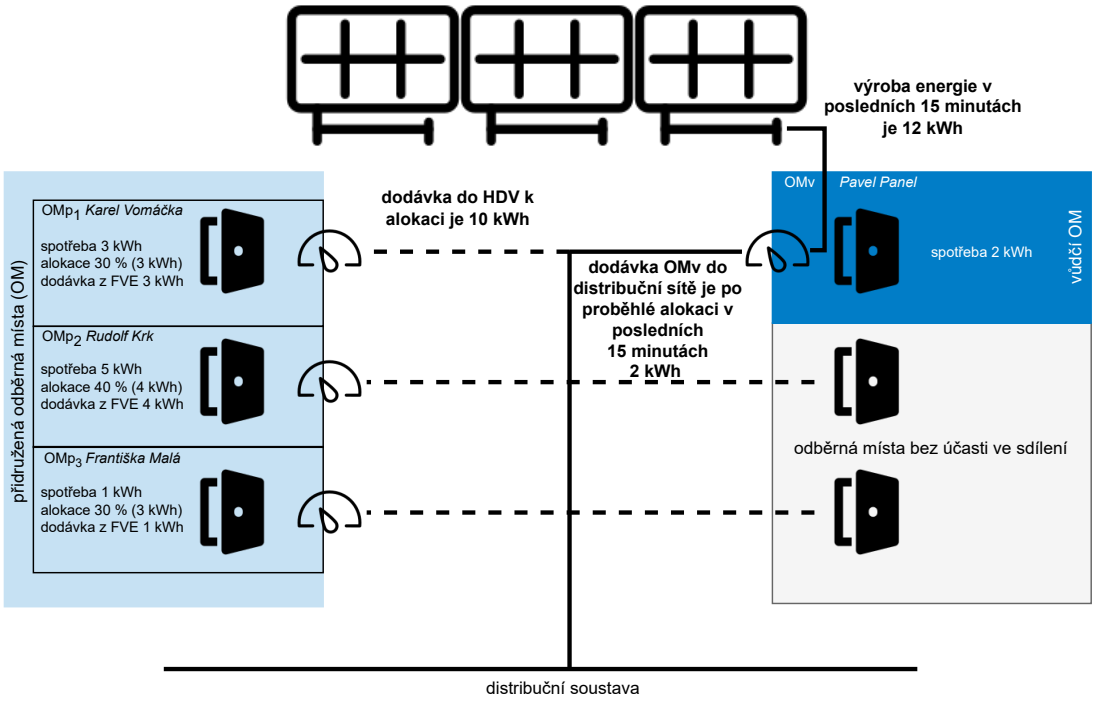
Odběrná místa, která se účastní sdílení elektrické energie, jsou dvojího typu.

- *Vůdčí odběrné místo* je takové místo, ke kterému je fyzicky připojena FVE, dodává elektrickou energii ostatním místům sdílení a jsou přes něj realizovány přetoky do distribuční sítě. Může být pouze jedno.
- *Přidružená odběrná místa* jsou ostatní odběrná místa, která se účastní sdílení. Mají mezi sebou definovaný alokační poměr (tzv. alokační klíč), ve kterém jim je rozdělena přebytečná vyrobená elektřina. Nemusí se tedy všechna dělit rovným dílem.

Rozpočet přebytečné vyrobené energie *vůdčího místa* mezi *přidružená místa* podle alokačního klíče, který provádí dodavatel, aby získal cenu odebrané (resp. dodané) elektrické energie pro každé odběrné místo, je prováděn se čtvrthodinovou granularitou. Výpočet tedy neoperuje s okamžitou spotřebou (resp. výrobou), ale se spotřebou (resp. výrobou) za 15 minut.

Na obrázku 2.2 je zachycen modelový příklad takového sdílení. V domě je instalována jedna výrobna elektrické energie ve formě FVE. V domě je šest odběratelů, přičemž na sdílení vyrobené elektrické energie se shodli ale jen čtyři odběratelé. Tito odběratelé si zvolili jedno *vůdčí odběrné místo* (označené jako *OMv*). Toto místo je fyzicky připojeno k FVE. Zbylá místa jsou označována jako *přidružená odběrná místa* (*OMp*). Dále se všichni zúčastnění musejí domluvit na tzv. alokačním klíči, resp. jakým poměrem bude mezi jednotlivá přidružená odběrná místa rozdělena přebytečná (nespotřebovaná vůdčím odběrným místem) elektrická energie. V tomto příkladu je to 30 % pro OMp_1, 30 % pro OMp_2 a 40 % pro OMp_3. Vyhodnocování spotřebované elektrické energie probíhá ve čtvrthodinových intervalech, tzn. že každých 15 minut je odečtena spotřebovaná energie na *OMp* a vyrobená energie na *OMv*. Ve vyúčtování distributora je poté od spotřebované energie *OMp* odečten jemu příslušný podíl vyrobené energie. Toto výsledné množství je poté bráno jako odebrané od distributora a je zákazníkovi účtováno. Model této skupiny sdílení je dále v textu uveden i v jiných kontextech.

Modelovým příkladem může být následující. FVE vyrobila za 15 minut 12 kWh elektrické energie. Na *OMv* byly v tomto časovém úseku spotřebovány 2 kWh, přetok do



Obrázek 2.2. Schéma sdílení vyrobené elektrické energie v bytovém domě [24]

sítě je tedy 10 kWh. Tato energie bude rozdělena mezi OMp. První OMp_1 spotřebovalo 3 kWh v daný časový úsek a mělo alokováno také 3 kWh, tím pádem mu nebude účtován žádný odběr. Druhé OMp_2 spotřebovalo 5 kWh, ale mělo alokováno pouze 4 kWh z vyrobené energie. Spotřebovalo tedy celou svou alokaci z FVE, a navíc mu bude účtován odběr 1 kWh. Konečně OMp_3 spotřebovalo 1 kWh, z vyrobené energie mělo také alokováno 3 kWh, nebude mu účtován žádný odběr. Zbylé 2 kWh zůstanou nevyužity. Jelikož bylo vyrobeno 12 kWh, ale spotřebováno pouze 10 kWh, OMv v daném časovém úseku tedy dodalo do distribuční sítě 2 kWh. Se ziskem z přetoku je poté naloženo podle interních směrnic daného subjektu. [24]

2.6 Specifikace požadavků

V předchozí části této kapitoly byly popsány všechny aspekty spojené s provozováním fotovoltaické elektrárny a to jak technologické (sekce 2.1), tak právní (sekce 2.3 a 2.5). Nedílnou součástí analýzy pro vytvoření systému, který bude pracovat s rozsáhlou doménou FVE, je definování, resp. sběr, konkrétních požadavků.

Požadavky jsou rozděleny do dvou skupin: funkční – které popisují očekávanou funkcionalitu systému, nefunkční – které popisují omezení systému, která musejí být dodržena. Všechny získané požadavky jsou v následujících sekcích uvedeny výčtem. Každý požadavek je opatřen krátkým upřesňujícím popiskem. Rozsah a zaměření požadavků odpovídá cíli práce, kterým je navrhnout funkční prototyp systému.

2.6.1 Funkční požadavky

Seznam funkčních požadavků popisuje očekávanou funkcionalitu systému. Všechny tyto požadavky se týkají pouze navrhovaného prototypu, nikoliv finálního systému, který případně na základě této práce vznikne.

V některých funkčních požadavcích je zmíněn termín datový objekt. V kontextu tohoto systému se jedná o ucelenou skupinu atributů, která je přiřazena určité entitě a dohromady vytváří sémanticky úplnou informaci. Lze si jej také představit jako objekt klasického objektového návrhu známého z programovacích jazyků, jako je např. Java.

- **F1 – Evidence datových objektů.**
  Systém bude umožňovat evidenci datových objektů, včetně uchování historie změn. Evidované objekty budou typované, každý typ bude obsahovat předem definovanou množinu vlastností. Každá vlastnost bude mít specifikovaný datový typ, případně validační podmínky, které nebudou závislé na ostatních datech. Datové objekty bude možné mazat, upravovat, přidávat a zobrazovat jejich vlastnosti.

- **F2 – Evidence vazeb mezi objekty.**
  Systém bude umožňovat evidenci vazeb mezi datovými objekty, včetně uchování historie změn. Evidované vazby budou typované, každý typ bude obsahovat předem definovanou množinu vlastností. Každá vlastnost bude mít taktéž specifický datový typ, případně validační podmínky, které nebudou závislé na jiných vlastnostech ani evidovaných datech. Dále bude mít každý typ vazby možnost definice integritního omezení na počáteční a koncový typ datového objektu, případně skupinu těchto typů. Vazby mezi objekty bude možné mazat, upravovat, přidávat a zobrazovat jejich vlastnosti.

- **F3 – Evidence časových řad.**
  Systém bude umožňovat evidenci časových řad. Jednotlivé časové řady bude možné logicky propojit s evidovanými objekty. Dále bude systém umožňovat vyhledávat určité časové úseky dat v těchto řadách. Kromě dat bude umožňovat systém i vyhledávání jednotlivých časových řad podle značek, případně logicky navázaných objektů.

- **F4 – Dotazování na evidované datové objekty, vazby a časové řady.**
  Systém bude umožňovat unifikovaný způsob dotazování na evidované objekty, jejich vazby a data z časových řad. U objektů a vazeb bude možné využívat tzv. traverzování – vyhledávání vzorů v databázi. Zároveň bude možné vyhledávat podle vlastností a typu objektů, resp. vazeb. U časových řad bude možné vyhledávat podle jejich názvu a k ním navázaným objektům. Všechna tato vyhledávání bude možné vykonávat k určitém časovém kontextu. Bude se jednat o časový rozsah, po jehož celou dobu musejí být platné všechny vyhledané objekty, vazby a body v časové řadě.

- **F5 – Napojení na externí datové zdroje.**
  Systém bude poskytovat rozšiřitelný způsob napojení na externí datové zdroje. Data získaná z těchto zdrojů bude možné v systému evidovat – buďto jako datové objekty (vazby) nebo jako data v časových řadách. Dále bude systém poskytovat rozšiřitelný způsob aktivního odesílání dat do externích zdrojů.

- **F6 – Definice úloh / business procesů.**
  Systém bude poskytovat rozšiřitelný způsob definování business procesů, které nebudou závislé na konkrétní implementaci zbytku systému. Všechny procesy budou mít přístup ke všem datům evidovaných v systému. Zároveň jim bude umožněno automaticky provádět všechny podporované operace s těmito daty. Takovéto procesy bude možné spouštět jak ručně, tak automaticky. Bude existovat log spuštění jednotlivých procesů, který bude obsahovat čas spuštění, průběh procesu a jeho výsledek. Jednotlivé definice bude možné do systému dynamicky přidávat, odebírat je a měnit jejich chování.

2.6.2 Nefunkční požadavky

Nefunkční požadavky definují chování systému, které musí dodržovat po celou dobu svého běhu. Těmito požadavky není, resp. by neměla být, ovlivněna jeho funkcionalita. Žádný z těchto požadavků nedefinuje chování z pohledu business vrstvy. Požadavky se zaměřují na realizační, technickou, část sytému.

- **N1 – Systém bude využívat mikroservisní architekturu.**
  Systém bude implementovaný s využitím mikroservisní architektury. Jednotlivé části funkcionality budou vhodně rozděleny tak, že je bude možné nezávisle na sobě škálovat.
- **N2 – Systém bude možné provozovat na operačním systému GNU/Linux.**
  Systém bude spustitelný na stroji s operačním systémem GNU/Linux.
- **N3 – Systém bude možné nasadit do kontejnerů Docker.**
  Systém a všechny jeho přidružené služby, komponenty, bude možné provozovat pomocí virtualizace OS v kontejnerech nástroje Docker.
- **N4 – Systém bude možné provozovat v single-node režimu na jednom stroji.**
  Systém nebude závislý na fyzickém propojení ani architektuře stroje (strojů), na kterých má být provozován. Bude umožňovat také provoz pouze na jednom stroji, na kterém budou spuštěny všechny jeho součásti, včetně závislých služeb.
- **N5 – Dotazování na data v systému bude možné pomocí unifikovaného dotazovacího jazyka.**
  Systém bude agregovat přístup ke všem evidovaným datům do jednoho místa API. Na tato data se bude možné dotazovat pomocí GraphQL.
- **N6 – Uživatelské rozhraní bude implementované jako webová aplikace.**
  Uživatelské rozhraní systému bude poskytováno formou jednostránkové webové aplikace (SPA).
- **N7 – Systém bude možné horizontálně škálovat.**
  Zásadní komponenty systému bude možné provozovat ve více instancích. Dynamická změna počtu instancí jedné komponenty nebude mít vliv na funkcionality poskytované systémem.

2.7 Specifikace případů užití

Každý případ užití popisuje akci, kterou může uživatel se systémem provést. Oproti požadavkům tak nedefinuje samotné chování (omezení) systému, ale spíše to, jak má být využíván. Každého případu užití se účastní jeden, nebo více aktérů. Každý případ užití je většinou složen z několika akcí, které jsou zachyceny jeho scénářem. Pro lepší pochopení je vhodné jednotlivé případy užití opatřit ještě krátkým doplňujícím popisem. Všechny případy užití, kromě UC12, jsou zaměřeny na obecné procesy, nutné pro libovolnou rozšiřující funkcionalitu systému. Případ užití UC12 se zaměřuje na konkrétní proces, na kterém je demonstrováno reálné využití systému.

Stejně jako u požadavků, i případy užití reflektují, že cílem práce má být pouze funkční prototyp systému.

Všechny specifikované případy užití jsou popsány v sekci 2.7.2 a zachyceny v diagramu případu užití na obrázku 2.3. Aktéři figurující v jednotlivých případech užití jsou popsáni v sekci 2.7.1.

2.7.1 Aktéři

V případech užití figurují tzv. aktéři. Jde o obecné persony, které se systémem interagují, přičemž interakce může být na několika úrovních. Aktéři mohou dané případy užití iniciovat nebo být pouze součástí procesu, kterého se případ užití týká. V případech užití navrhovaného prototypu systému figurují celkem tři aktéři – *Uživatel*, *Administrátor* (speciální případ uživatele) a *Systém*.

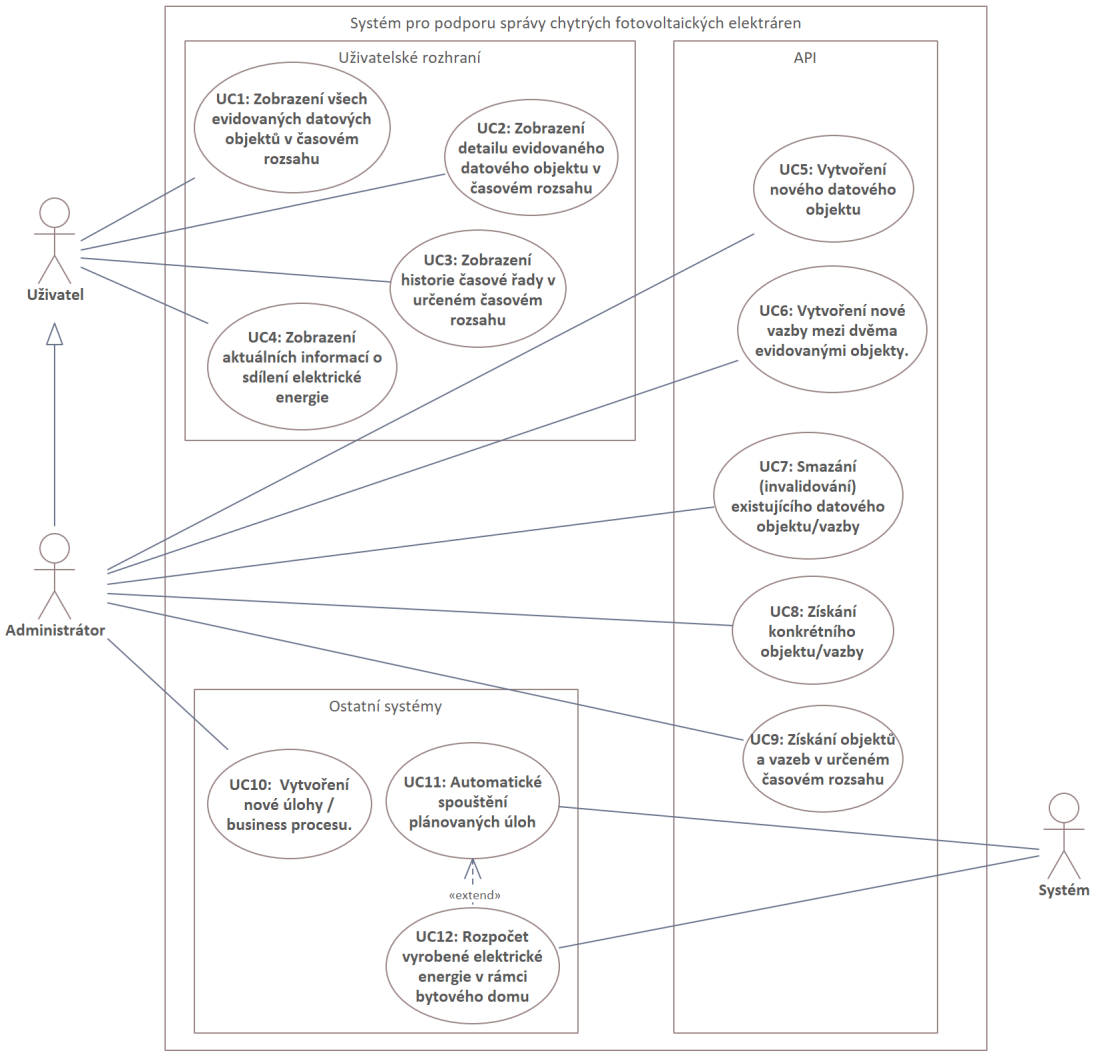
- *Uživatel* je aktér, který interaguje se systémem výhradně přes grafické webové rozhraní. Tento aktér využívá pouze předem vytvořené obrazovky, přes které se dívá na uložená data. Nevkládá do systémů žádná nová data, nepracuje nijak s žádným aplikačním rozhraním.
- *Administrátor* je rozšířením aktéra *Uživatel*. Se systémem také interaguje přes grafické webové rozhraní, ve kterém může zobrazovat aktuální data uložená v systému. Navíc má možnost a znalost přistupovat na aplikační rozhraní jednotlivých komponent systému. Přes ně může provádět všechny dostupné operace s datovými objekty, které se netýkají dat měření. *Administrátor* nemůže pracovat ani přes API s daty měření. K nim může přistupovat pouze přes webové rozhraní.
- *Systém* je posledním aktérem zasahujícím do případů užití. V tomto případě se vždy jedná o stroj, který nemůže být nikdy nahrazen fyzickou osobou. Tento aktér může spravovat všechna data evidovaná v sytému přes aplikační rozhraní jednotlivých komponent. Přes tato rozhraní se také může na všechna data dotazovat.

2.7.2 Seznam případů užití

V této sekci je uveden seznam případů užití, které definují způsob používání navrhovaného systému. Každý z nich je opatřen krátkým shrnujícím popisem. Případy užití, jejichž aktérem je `Uživatel`, popisují interakci s uživatelským rozhraním. Případy užití, jejichž aktérem je `Systém`, popisují automaticky vykonávané procesy nebo procesy reagující na události, které nejsou vyvolané uživatelem. Zbytek případů užití je zaměřen na využívání aplikačních rozhraní. U těchto případů je iniciálním aktérem vždy `Administrátor`.

Některé scénáře obsahují klíčová slova **Pokud** a **Jinak**. Těmito klíčovými slovy jsou vyjádřeny podmínky a části scénářů jimi označené se nemusejí provést vždy.

- **UC1 – Zobrazení všech evidovaných datových objektů v časovém rozsahu.**
  Aktér bude moct zobrazit přehled všech evidovaných datových objektů v zadaném časovém rozsahu. Tento přehled bude dostupný jako jedna z hlavních funkcionalit v menu. Výchozí časový rozsah je vždy celý den, ve kterém aktér provádí operace.
  Aktéři: Uživatel
  Scénář:
  1. V menu klikne aktér na položku reprezentující zobrazení všech evidovaných objektů.
  2. Systém zobrazí všechny evidované objekty.
  3. **Pokud** aktér požaduje změnit časový rozsah, ve kterém požaduje zobrazit data, může tak učinit pomocí komponenty výběru data a času rozsahu. Svůj výběr potvrdí kliknutím na tlačítko *OK*.
  4. Systém zobrazí všechny evidované objekty, které jsou platné v zadaném časovém rozsahu.



Obrázek 2.3. Diagram případů užití

- **UC2 – Zobrazení detailu evidovaného datového objektu v časovém rozsahu.**
  Aktér bude moct zobrazit detail vybraného evidovaného datového objektu. V rámci tohoto detailu budou zobrazeny informace o vlastnostech daného objektu, jeho vazbách na další objekty a seznam časových řad, které jsou pro tento objekt evidovány. Zobrazená data musejí být platná po celou dobu zvoleného rozsahu (musejí být vytvořena dříve než začátek rozsahu a ukončena (invalidována) později (nebo nikdy) než je konec časového rozsahu).
  Aktéři: Uživatel
  Scénář:

  Prvotní podmínkou pro tento scénář je, že aktér se již nachází v přehledu všech evidovaných datových objektů.

  1. Aktér vybere požadovaný datový objekt z přehledu, u kterého chce zobrazit detailní informace – vlastnosti. Volbu provede kliknutím na tlačítko *Show*.
  2. Systém zobrazí detailní informace o vybraném datovém objektu – jeho vlastnosti, vazby na další objekty a seznam časových řad, které jsou k němu evidovány. V tomto zobrazení bude systém respektovat časový rozsah, který byl nastaven v přehledu všech evidovaných datových objektů.

3. **Pokud** aktér požaduje změnit časový rozsah, ve kterém mají být informace zobrazeny, může tak učinit pomocí komponenty výběru data a času rozsah. Svůj výběr potvrdí kliknutím na tlačítko *OK*.
4. Systém upraví detailní zobrazení informací tak, že zobrazí detaily objektu platné v zadaném časovém rozsahu.

- **UC3 – Zobrazení historie časové řady v určeném časovém rozsahu.**

Aktér bude moct přistupovat k datům časových řad uložených v systému. Tato data bude možné zobrazit vždy ve vybraném časovém rozsahu. K jednotlivým časovým řadám spojených s některým z evidovaných objektů bude možné přistoupit z detailu tohoto objektu.

Aktéři: Uživatel

Scénář:

Prvotní podmínkou pro tento scénář je, že aktér se již nachází v detailu datového objektu.

1. Aktér vybere kliknutím na název časové řady tu, jejíž data chce zobrazit.
2. Systém zobrazí data z příslušné časové řady v časovém rozsahu, ve kterém byl zobrazen detail objektu.
3. Aktér může změnit časový rozsah pomocí komponenty výběru data a rozsahu času, ve kterém požaduje zobrazit data. Svůj výběr potvrdí kliknutím na tlačítko *OK*.
4. Systém zobrazí data z příslušné řady v zadaném časovém rozsahu.

- **UC4 – Zobrazení aktuálních informací o sdílení elektrické energie.**

Aktér bude moct zobrazit detailní informace o sdílení elektrické energie. Součástí tohoto zobrazení bude rozpočet vyrobené elektrické energie mezi jednotlivá místa, dále informace o tom, kolik elektrické energie bylo prodáno do sítě jako přetok, kolik energie bude fakturováno jednotlivým odběrným místům. Všechny tyto informace budou zobrazeny v požadovaném časovém rozsahu s patnáctiminutovou granularitou (stejná granularita, jakou definuje zákon). Jak zobrazit tyto informace popisuje tento případ užití.

Aktéři: Uživatel

Scénář:

Prvotní podmínkou je, že se aktér nachází v přehledu všech skupin sdílejících elektrickou energii. Zároveň má vybraný časový rozsah, ve kterém chce zobrazit detailní informace.

1. Aktér vybere požadovanou skupinu sdílející elektrickou energii kliknutím na její alokační klíč.
2. Systém zobrazí detailní informace o vybrané skupině sdílení.

- **UC5 – Vytvoření nového datového objektu.**

Aktér bude moct vytvářet nové datové objekty definovaných typů. Vytvářený objekt bude muset splňovat definované schéma.

Aktéři: Administrátor

Scénář:

1. Aktér odešle HTTP POST požadavek na určený endpoint. V těle bude obsažena informace o typu nově vytvářeného objektu společně s jeho vlastnostmi.
2. Systém přijme požadavek, zvaliduje jednotlivé položky vlastností, zda odpovídají validačním kritériím pro daný typ objektu.

(i) **Pokud** vše odpovídá požadavků, systém přidá nový objekt do evidence. V odpovědi odešle nově vytvořený objekt včetně jeho identifikátoru a jiných generovaných vlastností.
(ii) **Jinak** systém o neúspěšné validaci informuje v odpovědi.

- **UC6 – Vytvoření nové vazby mezi dvěma evidovanými objekty.**
  Aktér bude moct vytvářet nové vazby definovaných typů mezi evidovanými objekty. Vytvářená vazba bude muset splňovat definované schéma a validační kritéria.
  Aktéři: Administrátor
  Scénář:
  1. Aktér odešle HTTP POST požadavek na určený endpoint, v jehož těle budou všechny informace o nově zakládané vazbě. Dále bude požadavek obsahovat identifikátor počátečního a koncového objektu nově evidované vazby.
  2. Sytém přijme požadavek a zvaliduje vlastnosti nově zakládané vazby. Společně s tím provede také validaci integritních omezení vazeb na incidenční objekty.

  (i) **Pokud** všechny validace skončí úspěšně, systém přidá novou vazbu mezi objekty do evidence.
  (ii) **Jinak** o neúspěšné validaci systém informuje v odpovědi. Součástí odpovědi budou také informace, proč nebyla validace úspěšná.

- **UC7 – Smazání (invalidování) existujícího datového objektu/vazby.**
  Aktér bude moct smazat (invalidovat) evidovaný objekt nebo vazbu. Tato operace bude ovlivňovat platnost datového objektu.
  Aktéři: Administrátor
  Scénář:
  1. Aktér odešle HTTP POST požadavek na určený endpoint. V URL bude uveden identifikátor datového objektu, resp. vazby, která má být smazána.

  (i) **Pokud** je požadovaný objekt (resp. vazba) evidován a není označen za smazaný, systém tak učiní. Jako čas smazání doplní systém časovou značku přijmutí dotazu. Systém v odpovědi odešle aktuální stav všech vlastností mazaného objektu, resp. vazby.
  (ii) **Jinak** systém o neúspěšném pokusu o smazání informuje v odpovědi.

- **UC8 – Získání konkrétního objektu/vazby.**
  Aktér bude moct získat detailní informaci o konkrétním datovém objektu, případně vazbě. U vazby bude možné získat také detailní informace o počátečním a koncovém datovém objektu.
  Aktéři: Administrátor
  Scénář:
  1. Aktér odešle HTTP GET požadavek na určený endpoint, v URL bude specifikován identifikátor objektu, resp. hrany, kterou uživatel požaduje.
  2. Sytém přijme požadavek.

  (i) **Pokud** je požadovaný objekt (resp. hrana) evidován, poskytne systém tato data aktérovi v odpovědi.
  (ii) **Jinak** o neúspěchu informuje v odpovědi.

- **UC9 – Získání objektů a vazeb v určeném časovém rozsahu.**
  Aktér bude moct získat informaci o vazbách daného objektu v zadaném časovém rozsahu. Ke každé vazbě bude moct získat její počáteční a koncový objekt.
  Aktéři: Administrátor
  Scénář:
  1. Aktér odešle HTTP POST požadavek na určený endpoint. V jeho těle bude uveden dotaz v implementací poskytovaném dotazovacím jazyku.
  2. Sytém přijme požadavek.

     (i) **Pokud** je dotaz validní, vykoná požadované příkazy a vrátí všechny vyhovující objekty/vazby.
     (ii) **Jinak** o nesprávném formátu dotazu informuje v odpovědi.

- **UC10 – Vytvoření nové úlohy / business procesu.**
  Aktér bude moct vytvořit novou úlohu, kterou bude systém automaticky spouštět, nebo bude spuštěna manuálně, jednorázově, uživatelem.
  Aktéři: Administrátor
  Scénář:
  1. Aktér implementuje požadovanou logiku úlohy. Ve specifikovaném formátu ji nahraje do systému. Součástí definice, případně nahrání, musejí být všechna potřebná metadata, jako jsou časy, ve kterých má být úloha spouštěna apod.
  2. Systém detekuje, má-li nová úloha být automaticky spuštěna.

     (i) **Pokud** ano, bude ji systém automaticky asynchronně spouštět v definovaných časech.
     (ii) **Jinak** systém úlohu spustí jednou a poté ji nebude dále automaticky vykonávat.

- **UC11 – Automatické pouštění úloh / business procesů.**
  Systém bude moct automaticky spouštět definované úlohy / business procesy.
  Aktéři: Systém
  Scénář:
  1. U některé z definovaných úloh je splněna časová podmínka pro její spuštění.
  2. Systém spustí výpočet úlohy. Dále s ní již nijak neinteraguje, nezpracovává její výsledky ani stav.

- **UC12 – Rozpočet vyrobené elektrické energie v rámci bytového domu.**
  Systém bude poskytovat automatizovanou možnost rozpočtu vyrobené elektrické energie mezi jednotlivá odběrná místa podle platné legislativy.
  Aktéři: Systém
  Scénář:
  1. U automatické úlohy vykonávající rozpočet vyrobené elektrické energie byla splněna časová podmínka.
  2. Systém spustí rozpočet. S prováděným výpočtem dále již nijak neinteraguje.
  3. Spuštěná úloha uloží výsledky operace do sytému.

2.8 Mapování požadavků na případy užití

Mapování požadavků na případy užití (případně opačně) slouží v návrhu softwaru k propojení dvou pohledů. Co od navrhovaného softwaru požaduje uživatel (případy užití) a jaké funkcionality má navrhovaný software skutečně poskytovat (funkční požadavky). V mapování by měl každý případ užití odpovídat jednomu nebo více funkčním požadavkům. Na nefunkční požadavky nejsou případy užití mapovány, ty slouží pouze k řízení implementace a pomoci při návrhu systému, nijak neovlivňují chování systému z pohledu uživatele.

Pomocí tohoto mapování lze odhalit, že některé funkční požadavky nebudou nikdy využity (nejsou spojeny s žádným případem užití). Takovéto požadavky buďto byly opomenuty při tvorbě případů užití, nebo se jedná o požadavky skutečně nadbytečné. Dále lze odhalit opačný případ, kdy některý případ užití není spojen s žádným funkčním požadavkem. Tento stav může indikovat, že funkční požadavky kladené na software nejsou úplné a je potřeba je rozšířit. Případně se může jednat o takové případy užití, které nezapadají do kontextu daného systému. Nastane-li jeden z popsaných případů, je nutné inkriminované funkční požadavky, nebo případy užití znovu analyzovat.

Mapování je zachyceno pomocí relační matice. Sloupce reprezentují jednotlivé funkční požadavky, řádky případy užití. Vyplněná hodnota v matici znázorňuje vazbu typu `realization` mezi případem užití v řádku a funkčním požadavkem ve sloupci. Vyplněná buňka matice má následující význam – *Daný případ užití potřebuje (alespoň z části) pro svou realizaci určitý funkční požadavek.* Při čtení po řádcích lze matici reprezentovat takto – *Daný případ užití potřebuje pro svou realizaci tyto funkční požadavky.* Relační matice je uvedena v tabulce 2.1.

	F1	F2	F3	F4	F5	F6
UC1	x			x		
UC2	x			x		
UC3			x			
UC4	x	x		x		
UC5	x					
UC6		x				
UC7	x	x				
UC8	x	x		x		
UC9			x	x		
UC10						x
UC11						x
UC12	x	x	x	x		x

Tabulka 2.1. Relační matice případů užití a požadavků

Z relační matice 2.1 plyne, že každý případ užití potřebuje pro svou realizaci minimálně jeden funkční požadavek. Nebyl tedy analyzován žádný takový případ užití, který by nebylo možné realizovat (alespoň částečně) s navrženou funkcionalitou. Pro funkční požadavky je situace jiná.

Funkční požadavek F5 nemá vazbu s žádným případem užití. Tento funkční požadavek tak není potřebný pro žádnou popsanou akci. Implementace funkcionalit popsaných tímto požadavkem tak může být interpretována jako zbytečná. Nicméně není tomu tak. Požadavek není pokryt, protože celý návrh se týká jenom prototypu systému. Pokud

by tento sytém byl dále rozšiřován, byla by tato funkcionalita potřebná. Jedná se tedy o rezervu pro zajištění větší flexibility a rychlejší reakci na případné budoucí případy užití.

2.9 Existující řešení

V této sekci budou popsána existující řešení, která odpovídají, alespoň z části, požadavkům navrhovaného systému. Jsou to taková řešení, která nejsou cílena pouze na jeden typ FVE, resp. nejsou společně dodávána s konkrétním typem FVE, a jsou v rámci možností univerzální. U každého systému jsou popsány jeho základní vlastnosti, které jsou zhodnoceny s ohledem na požadavky navrhovaného systému. První dva popisované systémy: Homeassistant a OpenEMS jsou zástupci open-source projektů, které si uživatel musí sám nakonfigurovat a zprovoznit. Třetí zmíněný, SunDayGate, je jednoduchý, komerční, monitorovací systém pro malé FVE. Poslední systém, Domy Sobě, je komplexní komerční řešení.

2.9.1 Homeassistant

Homeassistant [26] je automatizační open-source platforma, primárně cílená na domácnosti nebo jiné automatizace menšího rozsahu. Mezi nadšenci do chytrých domácností je tato platforma velice populární, má podporu celosvětové komunity. Jistě tomu také přispívá fakt, že pro menší instance lze celou platformu provozovat na jednodeskovém, velice levném, počítači RaspberryPi.

Do platformy lze integrovat datové zdroje a zároveň odtud ovládat chytrá zařízení. Ať už jde o osvětlení, či měření spotřeby vody, pro většinu IoT zařízení existuje integrace právě do této platformy. Výjimkou tak nejsou ani některé střídače, které lze snadno bez větší znalosti do Homeassistant přidat, číst z nich data a ovládat je.

Jelikož se jedná o automatizační platformu, může si uživatel na základě dat, která platforma sbírá, nakonfigurovat automatické ovládání chytrých zařízení. Homeassistant podporuje pro jednoduché integrace grafické rozhraní. Složitější automatizace lze poté konfigurovat přes specifický DSL v notaci YAML.

Přestože platforma poskytuje komplexní nástroje pro automatizaci, neumožňuje sofistikovanější evidenci komponent nebo jiných objektů. Nelze ukládat např. senzory, z nichž sbírá data, tak, aby bylo možné sledovat jejich historii, případně logické či fyzické vazby. V porovnání s požadovanými vlastnostmi systému (2.6) splňuje tato platforma ty, které se zaměřují na sběr a dotazovaní na časové řady. Požadavky na evidenci datových objektů s jejich historií a vazbami není s pomocí této platformy možné naplnit. Dále není tato platforma vhodná pro větší instalace a je špatně škálovatelná.

2.9.2 OpenEMS

Jedná se o open-source projekt, který poskytuje modulární platformu pro spravování energetických instalací, jako jsou FVE. Umožňuje monitorovat, ovládat a integrovat již existující řešení do jedné platformy. Výhodou této integrace je možnost analyzovat všechna data na jednom místě a na základě nich ovládat připojená zařízení a rozhodovat, zda má být vyrobená energie prodána distributorovi, uložena nebo spotřebována. [27]

Architektura platformy je rozdělena na dvě části. První je OpenEMS Edge, který slouží jako integrační brána, která komunikuje se zařízeními FVE, ovládá je a sbírá o nich data. Data ukládá jako časové řady do InfluxDB. Druhou částí je OpenEMS Backend, který agreguje data z několika OpenEMS Edge. Nad těmito daty umožňuje vytvářet algoritmy, které je mohou zpracovávat a dále analyzovat. Získané informace

z těchto analýz poté zpětně zpřístupňuje jednotlivým instancím OpenEMS Edge, které na základě nich mohou např. automatizovat některé procesy.

Platforma poskytuje velice širokou škálu protokolů pro integrování s jinými službami a zařízeními. Pro některá, často používaná, zařízení jsou integrace již implementovány. Pro ostatní poskytuje otevřené rozhraní umožňující napojení nových komponent. Výhodou také je, že projekt je napsán v high-level programovacím jazyce Java, a tak je přístupný větší škále uživatelů. Konceptuálně je projekt zaměřený na stejný případ užití jako navrhované řešení v této práci, včetně integrace na externí systémy a datové zdroje. Příkladem externích datových zdrojů je např. předpověď počasí a solárního osvitu. Na základě nich platforma predikuje výrobu elektřiny na 24 hodin dopředu společně s predikcí spotřeby na stejný časový úsek. Snaží se tak usnadnit a co nejvíce automatizovat správu vlastních výroben elektrické energie.

Nesoulad s navrženými požadavky na systém a touto platformou je především v evidenci objektů. Lze evidovat pouze připojené komponenty, ne už jejich historii a vazby. Celkově lze evidovat pouze aktuální stav. Nicméně OpenEMS teoreticky umožňuje tuto funkcionalitu doimplementovat (zejména tedy protože se jedná o open-source projekt). Jednalo by se ale o zásah do interních procesů, a ne pouze o rozšíření, které podporuje.

2.9.3 SunDayGate

SunDayGate [28] je automatizační software pro monitoring malých fotovoltaických elektráren, který na pozadí využívá software Promotic. Umožňuje uživateli zobrazit aktuálně vyráběný výkon, aktuální spotřebu na jednotlivých fázích, stavy nabití akumulátoru a další. Dále umožňuje například zobrazovat na základě informace o množství aktuálně vyráběné energie, jaké přístroje (podle jejich výkonu) je možné používat, aby byla energie stále čerpána z fotovoltaiky.

Funkcionality jsou omezeny pouze na dva typy střídačů Goodwe ET a Goodwe ET plus. Software umožňuje tyto typy střídačů ovládat a přepínat mezi stavy ukládání energie do baterií a prodávání energie distributorovi. Toto přepínání umí automatizovat na základě několika parametrů: nabití baterie, napětí FV panelů, kalendáře. Hlavním rozdílem oproti požadavkům na navrhovaný systém je architektura samotného softwaru. Zde se jedná o desktopovou monolitní aplikaci, která sbírá data a ovládá FVE pouze tehdy, je-li zapnuta. Konsekvencí tohoto návrhu je poté fakt, že všechna naměřená data systém ukládá pouze lokálně, na disk. Nelze je tak přímo pomocí tohoto nástroje odesílat do systému třetích stran pro další zpracování. Zároveň není možné data analyzovat ani lokálně s podporou aplikace. Problém je také s integrací dalších datových zdrojů do systému. Celkově jde spíše o monitorovací systém s velice základní možností automatizace bez možností rozšíření o více komplexní operace a nástroje. Nesplňuje tak požadavky, které má splňovat navrhovaný systém.

2.9.4 Domy sobě

Řešení pro komunitní energetiku poskytované projektem Domy sobě [29] je částmi své funkcionality velice podobné navrhovanému systému. Umožňuje spravovat komunitní FVE, dynamicky přidávat a odebírat odběrná místa ze sdílení a jejich monitoring. Všechna potřebná měřicí zařízení a čidla jsou spojena do jednoho řešení a přístupná přes službu Dominiq, poskytovanou stejným projektem.

Zaměření projektu ale není čistě pouze na komunitní energetiku a FVE. Součástí projektu je i možnost zapojení nejrůznějších čidel, jak elektrické energie, tak např. spotřebované vody. Poskytuje kompletní řešení (hardwarové i softwarové) pro dálkové odečty. Do svého ekosystému umožňuje přidat bezpečností čidla apod. Data z nich

agreguje do již zmíněné aplikace Dominiq. Jelikož jde o uzavřené řešení, ke kterému není dostupná dokumentace a propagační informace o aplikaci jsou také velice strohé, není možné dále analyzovat všechny funkcionality.

Domy sobě poskytují pomoc také s legislativním procesem dotací kolem obnovitelných zdrojů elektrické energie (dotační program Nová zelená úsporám). Jedná se tedy o komplexní projekt, jehož výhodou může být právě široký záběr působení, který zákazníkovi je schopen pomoci s libovolným problémem (nejen) komunitní energetiky. Hlavní nevýhodou je uzavřenost řešení, které neumožňuje žádným způsobem evidovat jednotlivé části FVE. Dále pak nemožnost napojení na externí služby, stejně tak jako poskytování dat do externích systémů. Tím pádem není možné se získanými daty dále pracovat.

2.10 Zpracovávaná data

V požadavcích na systém není jasně definováno, která data bude nutné v systému evidovat a dále zpracovávat. Jsou uvedeny pouze abstraktní struktury. Nicméně protože navrhovaný systém má sloužit pro podporu správy FVE, analýza předpokládá, že evidovaná data budou právě z této domény. Zároveň případ užití UC12 popisuje požadavek na rozpočet vyrobené elektrické energie v rámci bytového domu. Pro jeho splnění je nutné mít všechna potřebná data. V tomto případě se jedná o souhrny spotřebované a vyrobené elektrické energie. Tyto informace je možné získat z elektroměrů, nemusejí být nutně poskytovány distributorem (případně lze využít data poskytovaná střídačem). V instalacích mohou existovat vlastní předřazené elektroměry, které poskytují integrační rozhraní a umožňují dálkové odečítání dat.

Je nutné myslet ale i na další rozvoj systému. Rozpočet vyrobené energie v bytovém domě je jen jednou z mnoha možností výpočtů. Obecně pro operace (a procesy) budou v systému evidována i další data (např. elektrické veličiny). Způsob získávání těchto dat není nijak blíže specifikován. Z popisu komponent FVE (sekce 2.1.2) je ale zřejmé, že většina informací o stavu FVE půjde získat z již zmíněného střídače nebo zmíněných elektroměrů. Tomuto se však více věnuje sekce 2.10.2.

Struktura, ve které budou data do systému importována, také není pevně stanovena a s různými výrobci různých zařízení se může velice lišit. Je tedy nutné myslet na to, že návrh tohoto procesu musí být dostatečně obecný a rozšiřitelný – data mohou mít různou strukturu a nebudou do systému importována v uniformním formátu.

2.10.1 Vytěžované a zpracovávané informace

V systému budou zpracovávány dva hlavní druhy dat – pozorované veličiny (data poskytované komponentami FVE, případně jinými senzory nebo aplikacemi) a komponenty FVE společně s jejich vazbami (případně další datové objekty, jako je alokační klíč). Oba druhy mají svá specifika a nelze je zpracovávat stejným způsobem.

Při popisu komponent (v sekci 2.1.2) FVE bylo zjištěno, že jádrem každé FVE je tzv. střídač. K němu jsou připojeny ostatní zásadní komponenty, jako fotovoltaické panely či baterie. Zároveň slouží jako místo dodávky elektrické energie z FVE. Také umožňuje ovládat, v jakém režimu bude elektrárna pracovat – zda bude prodávat vyrobenou energii, nebo ji bude ukládat do baterií, nebo bude spotřebována připojenou zátěží. Střídač má dále k dispozici informace ze všech připojených komponent, případně sám některé veličiny měří. Lze z něj získat informace o aktuálně vyráběném výkonu fotovoltaickými panely, o aktuálním příkonu připojené zátěže, o stavu nabití baterií, ale i o tom, kolik elektrické energie je dodáváno z distribuční sítě (pokud to jeho zapojení

umožňuje). Sledovat množství dodané a odebrané elektrické energie lze taktéž pomocí chytrých elektroměrů. Z nich lze získávat informace jak o aktuální spotřebě (dodávce), tak o spotřebě (dodávce) za určitý časový úsek (liší se v závislosti na typu elektroměru). Dalším zdrojem dat pro systém mohou být přídavné senzory. Ty však nejsou pro provoz fotovoltaické elektrárny nutné, proto nemusejí být v každé instalaci dostupné. Nejčastějšími senzory mohou být senzory teploty, vlhkosti či osvitu. Se ztrátou přesnosti lze taktéž pro tyto informace využít externí datové zdroje.

Všechna data, která budou postačující pro základní funkcionalitu systému, aby byly naplněny všechny případy užití popsané v sekci 2.7.2, lze získat ze střídače a elektroměrů. Ačkoliv některé veličiny lze odvodit z jiných (např. výkon lze získat z napětí a proudu), pro zjednodušení a urychlení dotazování je vhodné ukládat je duplicitně. Seznam veličin, které je možné získat z běžného střídače, je uveden v tabulce 2.2. Tento seznam byl zpracován na základě informací poskytovaných GoodWe API [12]. Zároveň jde o základní elektrické veličiny, které lze případně měřit i externími měřicími přístroji.

název	popis	jednotka
okamžitá výstupní napětí FV panelů	napětí na výstupu panelů / napětí na vstupu střídače	V
okamžitý výstupní proud FV panelů	proud na výstupu panelů / proud na vstupu střídače	A
okamžité výstupní napětí střídače	napětí na výstupu střídače	V
okamžitý výstupní proud střídače	proud na výstupu střídače	A
okamžitý výkon zátěže	kolik elektrické energie je spotřebováváno připojenou zátěží	W
okamžitý dodávaný výkon z distribuční sítě	kolik je dokupováno elektřiny, aby byla poryta spotřeba	W
spotřebovaný výkon zátěží	spotřebovaný výkon v určeném časovém horizontu	Wh
vyrobený výkon FV panely	vyrobený výkon v určeném časovém horizontu	Wh
dodaný výkon do distribuční sítě	dodaný výkon v určeném časovém horizontu	Wh
napětí baterie	napětí nakrátko na svorkách baterie	V
stav nabití baterie	poměr uložené energie vůči maximálnímu množství, které je možné uložit	1

Tabulka 2.2. Seznam základních veličin, které lze získat ze střídače (GoodWe API [12])

Další data, která budou v systému zpracovávána, se týkají fyzického a logického propojení komponent FVE, případně dalších datových objektů, které bude potřeba evidovat. Tato data nebudou odečítána (získávána z externích zdrojů), ale budou do systému zadávána uživatelem. Tyto informace budou určeny především pro zpětnou kontrolu a monitorování změn v systému a pohled na jeho aktuální stav. Také je bude

možné použít pro identifikace zdroje měřených dat a případně další libovolné aplikace, které budou potřebovat pracovat s doménou konkrétní instalace fotovoltaické elektrárny.

2.10.2 Využití a dotazování vytěžovaných dat

Dostatečná analýza využití vytěžených dat je prerekvizitou dalšího návrhu. Od těchto informací se bude odvíjet také výběr databázových strojů vhodných pro implementaci. Ze sekce 2.10.1 vyplývá, že vytěžované informace budou dvojího druhu. Jedny s charakteristikou časových řad a druhé závislostního grafu.

U časových řad (v této práci také označovaných jako měření nebo metrika) mají obecně všechny informace platnost pouze v momentu, kdy byly získány, resp. naměřeny. Nikdy se nebudou aktualizovat, pouze se budou přidávat nová data na konec řady. Z povahy těchto informací pozdější aktualizace ani nedává smysl. Příkladem z domény FVE může být hodnota napětí na výstupu fotovoltaického panelu (nebo jiné měření elektrické veličiny, stejně tak jako aktuální teplota). Ta je platná v daný okamžik měření. Z pohledu naměřené hodnoty nemůže nikdy v budoucnu dojít ke změně. Změna by znamenala zásah do minulosti, což je z podstaty kontinuity času nesmyslné.

Všechny dotazy nad daty v časových řadách budou analytického charakteru. Lze předpokládat časté provádění agregací, zejména vytváření časových. Těmito agregacemi lze totiž „nastavit“ (snížit) detail dotazovaných informací (např. průměrovat hodnotu napětí na fotovoltaickém panelu během jedné hodiny). Toho bude možné využít jak pro ad-hoc dotazy vytvářené uživatelem, který je může požadovat například kvůli grafické vizualizaci, tak pro další analytické zpracování, jako je např. výpočet spotřeby mezi jednotlivými odběrnými místy, jehož příklad byl uveden v sekci 2.5.2. V případě budoucí potřeby automatického řízení FVE mohou být tato data použita jako vstup pro rozhodování (např. jako trénovací pro algoritmy strojového učení pro predikci výroby elektrické energie). Časové řady tedy budou zpracovávány tzv. OLAP [30] (online analytical processing) přístupem. Nebudou probíhat modifikace dat, ale budou se nad nimi provádět časově a výpočetně náročné dotazy a operace.

Dalším důležitým dotazováním bude vyhledávání mezi vazbami datových objektů (komponent FVE), a to jak s aktuální verzí objektů a vazeb, tak s libovolnou historickou verzí. I toto je totiž jeden z požadavků na systém. Fyzická propojení mezi zařízeními (a případně jinými entitami v systému) lze reprezentovat jako graf. Jednotlivá propojení mohou být jak fyzická, tak jakákoliv abstraktní (např. datové spojení). Z důvodu odlišného charakteru dat oproti předchozím časovým řadám bude nejspíš vhodné zvolit jiný datový engine než pro data měření. Nicméně to není předmětem této sekce a podrobněji bude tento problém rozebrán v následující kapitole. V těchto datech se bude často vyhledávat propojení komponent platné k určitému datu, nejčastěji potom aktuálně platné spojení. Tyto informace budou sloužit pro zjišťování změn v systému, např. při hledání potenciálního zdroje problému.

Kapitola 3
Návrh

V této kapitole jsou popsány jednotlivé části návrhu. V úvodu je představena zvolená architektura. Následuje popis technologií, které mohou být použity v dalších fázích návrhu. Technologie byly vybrány v návaznosti na provedenou analýzu. Dále je popsán vybraný programovací jazyk a framework. Pro jeho výběr byly brány v potaz uvažované technologie.

Následně je představen návrh datového metamodelu objektů a vazeb, včetně základního konceptu udržování konzistence a zajištění integritních omezení. Datový návrh dále popisuje ukládání dat časových řad.

V závěru kapitoly je zpracován návrh jednotlivých mikroslužeb. Pro každou z nich jsou z navrhovaných technologií vybrány ty, které budou použity pro následnou implementaci. Součástí návrhu mikroslužeb je také návrh uživatelského rozhraní. Pro každou jeho obrazovku je uveden wireframe. Dále jsou jednotlivé obrazovky dány do kontextu s analyzovanými případy užití.

Jedním z cílů práce je vytvořit systém tak, aby byl dobře rozšiřitelný, resp. škálovatelný. Proto bude v návrhu přistoupeno k mikroservisní architektuře, zároveň se jedná i o jeden z nefunkčních požadavků – N1. Mikroservisní architektura sice přináší několik problémů, především zajištění spolehlivé komunikace mezi mikroslužbami a v případě více instancí jedné mikroslužby load-balancing. I přesto má tento přístup nesporně jednodušší možnost škálování. Oproti monolitní architektuře se také snáze implementují nové funkcionality, často jako nové služby. Každá mikroslužba vždy obsluhuje pouze jednu business funkcionalitu. Návrh služeb nelze vytvořit pouze na základě funkčních (business) požadavků na systém. Některé požadavky mohou být natolik blízké, že budou naplněny jednou službou. V případě požadavků popsaných v sekci 2.6.1 bude vhodné spojit do jednoho logického celku (služby) evidenci datových objektů (F1) a evidenci vazeb mezi datovými objekty (F2).

Datové zdroje (např. senzory, externí API) obecně neposkytují stejné rozhraní a nekomunikují stejným protokolem. Stejně tak není unifikovaný formát, ve kterém data poskytují. Návrh tak bude muset zohlednit možnost jednoduché rozšiřitelnosti o konektory, integrace atp. Toho lze dosáhnout oddělením ukládání a načítání. Každá tato funkcionalita bude obsluhována jednou mikroslužbou. V případě potřeby nových integrací bude stačit rozšířit funkcionalitu jedné mikroslužby, případně se pro tento účel implementuje nová.

3.1 Mirkoservisní architektura

Hlavním záměrem přístupu k tvorbě softwaru s dodržením mikroservisní architektury je rozdělování požadovaných funkcionalit do služeb, které jsou poté implementovány jako samostatné aplikace, nazývaná mikroslužba. Každá mikroslužba by měla poskytovat jednoduché rozhraní, často HTTP API, přes které je přístupná její logika pro ostatní mikroslužby. [31]

Z tohoto popisuje je zřejmé, že implementace softwaru s využitím mikroservisní architektury je časově více náročné. Každá mikroslužba totiž obsahuje řadu funkcionalit, které je nutné vyvinout. Zejména se jedná o poskytované aplikační rozhraní. Hned vzápětí je podobný problém s klienty pro přístup na ostatní mikroslužby. Zároveň je velice důležité implementovat je odolně vůči chybám [31].

Protože jsou jednotlivé služby rozděleny do zvláštních aplikací, které se poté podílejí na vykonávání složitějších business procesů, je nutné udržovat přehled, jaké operace v které mikroslužbě patří ke kterému vykovávaní jakého business procesu. Toho lze dosáhnout pomocí tzv. korelačního identifikátoru, který je vytvořen s prvotním dotazem a poté propagován s každým dalšími dotazy skrze všechny mikroslužby. V logování jednotlivých služeb pak lze trasovat celý průtok jednoho dotazu. Aby nebylo nutné při sledování některých dotazů procházet více logovacích zdrojů, existují nástroje jako např. OpenTelemetry [32], které všechny tyto informace agregují a umožňují je zobrazit v ucelené formě.

Testování mikroslužeb může být v některých případech také velice náročné. Především prvotní časová investice do vytvoření simulací (mock) všech služeb, které jsou pro běh testované mikroslužby potřeba. Simulování je důležité zejména pro jednotkové testy. Pro integrační testy je naopak poměrně snadné nasadit celý systém (nebo jeho část), která je pro ně potřeba, a nad ním spouštět potřebné testy. Ačkoliv i v tomto typu testů je částečně nutné simulovat některé závislosti.

Odměnou za takto vyložené úsilí je jednoduchá škálovatelnost s vysokou granularitou. Každá služba může běžet ve více instancích a dynamicky lze jejich počet měnit podle potřeby (zejména podle zátěže systému). Celý systém tak bude navrhován právě s využitím mikroservisní architektury – tím bude pokryt nefunkční požadavek na použití mikroservisní architektury (N1).

Samotné škálování spouštěním jedné služby ve více instancích je pouze prvním krokem. K tomu, aby tento mechanismus byl alespoň základně funkční, je ještě potřeba distribuovat požadavky na službu mezi všechny její instance. K tomuto účelu slouží tzv. load balancer, který musí mít informaci o všech běžících instancích, ideálně jejich vytížení, aby mohl požadavky efektivně distribuovat. K zjišťování, jaké instance, které služby jsou spuštěny slouží service discovery. Může se jednat například o službu (service registry), do které se při spuštění každá instance zaregistruje, případně ji zaregistruje aplikace třetí strany (např. clustering framework Kubernetes) [33]. Pokud jiná služba (např. load balancer) potřebuje vědět, jak jsou dostupné služby a jejich instance, může se na tyto informace dotázat. Na všechny tyto aspekty je také nutné v dalším návrhu myslet.

Využitím mikroservisní architektury při návrhu a implementaci prototypu systému bude možné snáze splnit jeden z cílů, kterým je požadavek na škálovatelnost (N7).

3.2 Technologie

Před samotným návrhem jednotlivých mikroslužeb systému budou v této sekci popsány technologie, které by během něj mohly být využity. Při výběru technologií byly brány v potaz informace získané v kapitole 2, Analýza. Jednalo se zejména o informace z oblasti funkčních požadavků na systém společně s případy užití.

Ke každé technologii je uveden její krátký popis a základní popis funkcionalit, které nabízí, případně problémů, které je s její pomocí možné řešit. V závěru každé technologie je poté dále uvedeno, jak lze danou technologii využít v dalším návrhu.

3.2.1 Apache Hadoop

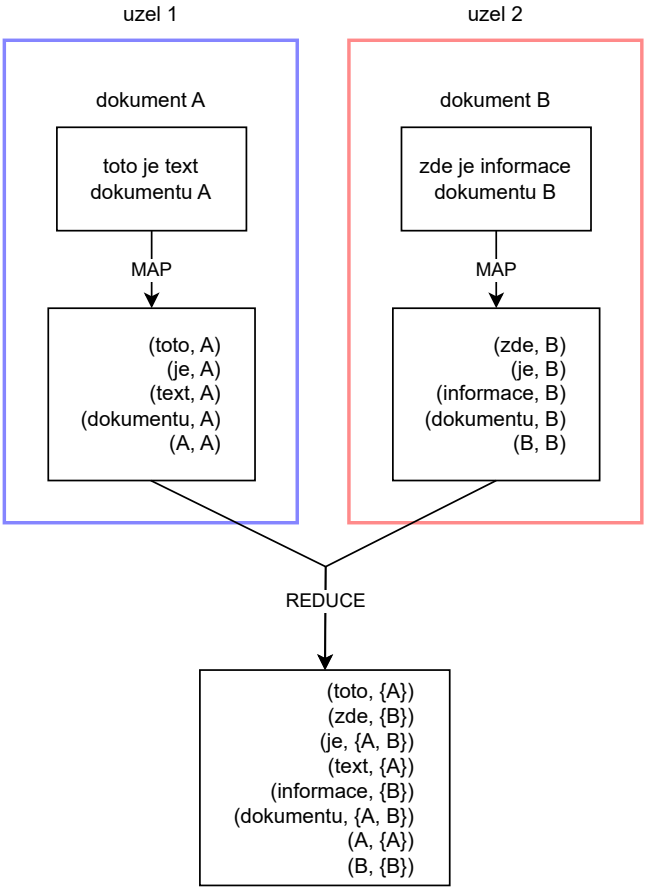
První popisovanou technologií je Apache Hadoop [34]. Bude poskytovat běhové prostředí pro některé použité technologie. Jedná se o software pro škálovatelné distribuované výpočty. Skládá se z několika specializovaných knihoven, které dohromady tvoří univerzální framework pro paralelní zpracování velkého množství dat. Dvě, které budou důležité pro implementaci, jsou Hadoop distributed filesystem (HDFS) a Hadoop MapReduce.

Hadoop distributed filesystem používají pro ukládání dat všechny knihovny v Apache Hadoop. Tento filesystém ukládá data – jak název napovídá – distribuovaně na několik strojů (uzlů). Funguje v režimu Master-Slave, tedy jeden uzel (*NameNode*) řídí přístup do celé filesystému, ostatní uzly (*DataNode*) poté pouze ukládají a poskytují soubory, nijak už neřeší přístupová práva ani metadata k uloženým souborům [35]. To umožňuje jednoduše navyšovat kapacitu bez nutnosti využívat specifický hardware. Stačí pouze stroj (nový výpočetní uzel), na kterém lze spustit instanci HDFS, tedy takový, na kterém lze provozovat Javu. Jelikož s tímto filesystémem nebude implementace nikde pracovat přímo, nebudou další, detailnější, informace dále rozpracovány.

Hlavním principem, který Apache Hadoop využívá při zpracování, je MapReduce framework, který implementuje stejnojmenný výpočetní model. Umožňuje tak rozdělit složitý výpočet do několika menších, které lze poté distribuovat mezi několik výpočetních uzlů. Při jeho použití jsou vstupní data rozdělena do nezávislých celků, nad kterými lze provádět nezávislé operace (funkce poskytující tyto operace se nazývá *Map*). Jelikož je provádění těchto operací na sobě nezávislé, lze je distribuovat na několik výpočetních uzlů. Vypočtené výsledky jednotlivých částí (z několika výpočetních uzlů) jsou poté sloučeny do jednoho. Funkci implementující toto spojení se říká *Reduce*. Aby bylo možné spojit spolu správné mezivýsledky, jsou data reprezentována jako uspořádaná dvojice $(key, value)$. Spojují se vždy data se stejnou hodnotou *key*. Operace *Reduce* může být provedena hned po operaci *Map* na stejném výpočetním uzlu. Tato optimalizace je vhodná v případě, je-li aktuálně zpracovávaný dataset malý, ušetří se tak síťové I/O operace, které jsou spojené s přesunem dat mezi výpočetními uzly. O operaci *Reduce* provedené na stejném uzlu, na kterém byl proveden *Map*, se poté říká *Combine* [36].

Jednoduchým příkladem využití výpočetního modelu MapReduce je výpočet zpětného indexu. Postup je schematicky zachycen na obrázku 3.1. Vstupem algoritmu pro výpočet zpětného indexu je množina textových dokumentů, výstupem je poté pro každé slovo množina identifikátoru dokumentů, ve kterých se dané slovo vyskytuje. Při výpočtu je každý dokument zpracováván jedním výpočetním uzlem. V první části každý uzel vytvoří množinu uspořádaných dvojic (a, b), kde a je indexované slovo a b je identifikátor dokumentu. Ve druhé části je proveden *Reduce*. Oba výsledky jsou zkombinovány dohromady tak, že jsou opět vytvořeny uspořádané dvojice. První položkou je stejně jako u předchozího kroku indexované slovo. Druhou položkou je množina identifikátorů všech dokumentů, ve kterých se dané slovo nachází. Tato množina vznikla spojením dvou podmnožin, které byly poskytnuty z výpočetních uzlů.

Samotné použití technologie Apache Hadoop nebude pro další návrh ani následnou implementaci vhodné, nicméně se jedná o technologii, nad kterou jsou postaveny další, jejichž použití naopak vhodné bude. Těmito technologiemi jsou Apache Spark popsané v sekci 3.2.2 a Apache HBase popsaná v sekci 3.2.3.



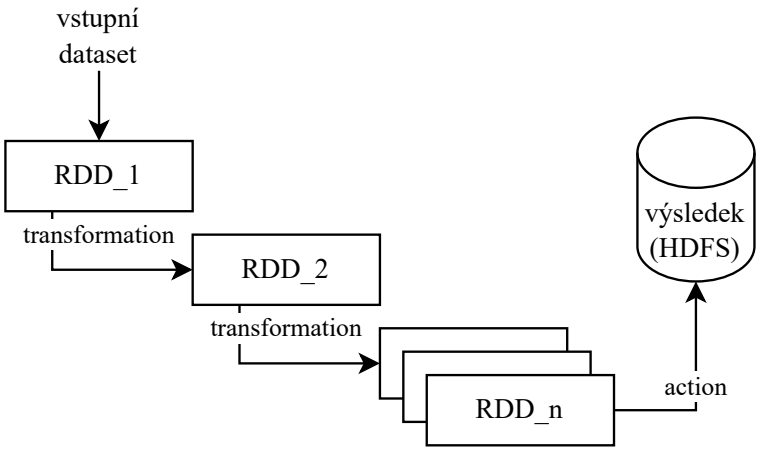
Obrázek 3.1. Příklad výpočtu zpětného indexu pomocí MapReduce na dvou uzlech

3.2.2 Apache Spark

Apache Spark je analytický engine pro distribuované zpracovávání velkého množství dat. Ve většině případů užití jsou v něm výpočty rychlejší než v Apache Hadoop využívajícím MapReduce framework [37], a to zejména proto, že většina jich je prováděna v paměti, nebo jsou výsledky ukládány do cache. Skládá se z několika hlavních komponent, kde dvě zásadní jsou Spark core a vysokoúrovňové knihovny (SparkSQL, Spark Streaming atd.). Kompletní schéma high-level architektury Apache Spark je zobrazeno na obrázku 3.3.

V porovnání s MapReduce frameworkem v Apache Hadoop pracuje Apache Spark s daty odlišným způsobem. Neimplementuje výpočetní model MapReduce, ale pracuje s daty jako s abstraktní datovou strukturou nazývanou Resilient Distributed Dataset (RDD) [38]. Ta obsahuje kolekci elementů (záznamů), která může být rozdělena do několika částí (každá část může být na jiném serveru v clusteru) a zpracovávána paralelně. Zdrojem záznamů může být soubor na distribuovaném filesystému HDFS, přímo kolekce v řídícím programu, libovolný jiný podporovaný engine pro ukládání dat (příkladem mohou být enginy uvedené v posledním řádku ve schématu 3.3) nebo jiné RDD. Další vlastností této abstraktní struktury je neměnnost (immutability) a dostupnost jen pro čtení (read-only). Pokud se tedy má s daty provést nějaká modifikace, je nutné vytvořit nový dataset s upravenými hodnotami. Odtud tedy plyne možnost vytvářet RDD z jiných RDD.

Celkově nad daty načtenými v RDD lze provádět dva typy operací: transfromace (transformation) a akce (action). Transformace modifikuje záznamy v datasetu, a jelikož je dataset immutable, vytváří se tak nový, který obsahuje modifikovaná data. Akce nevytváří nový dataset, ale určitým způsobem z něj získává data. Může jít například o agregace nebo o vyčítání prvních n záznamů z datasetu. Výsledky všech akcí jsou předávány řídícímu programu [38]. Akce jsou terminující operátor, tedy nelze za žádnou akci řetězit transformaci. Schéma řetězení transformací je zobrazeno na obrázku 3.2.



Obrázek 3.2. Posloupnost transformací nad RDD

Pro zvýšení efektivity prováděných operací jsou transformace a akce pouze předpisem, jak se vstupním datasetem pracovat. Jejich vykonávání je tzv. lazy. Tedy žádná transformace není vykonána, dokud nejsou skutečně potřeba data, které poskytuje na svém výstupu. Jinými slovy, žádná transformace není provedena do té doby, dokud není vykonána nějaká akce. Navíc, je-li známé, jak budou data zpracovávána, lze také výsledky některých operací cachovat. Zejména jde o takové operace, které jsou volány vícekrát se stejnými parametry a jejichž výpočet je nezávislý na stavu. Tuto optimalizaci také Apache Spark ve svých výpočtech provádí.

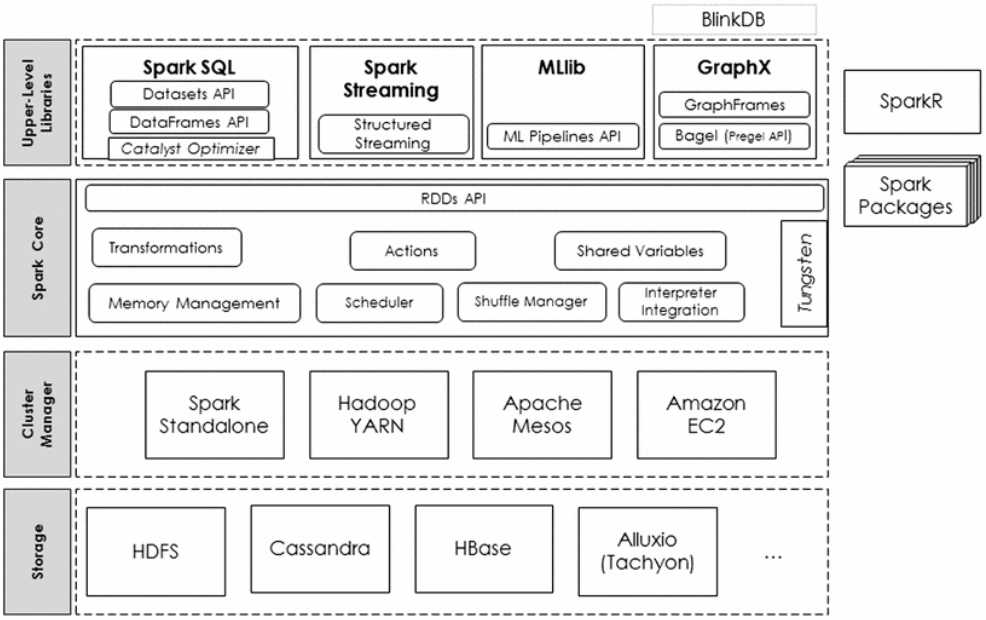
Operace a pořadí jejich provádění (závislosti) mohou mít téměř libovolnou strukturu. Musejí se však řídit základním pravidlem, aby bylo možné deterministicky určit, v jakém pořadí je provést: Závislosti použitých operátorů (transformací a akcí) musejí tvořit acyklický graf.

Aplikační rozhraní vyšších úrovní

V úvodní sekci bylo popsáno základní aplikační rozhraní, resp. datová struktura RDD, se kterým Apache Spark pracuje. S dalším vývojem však přišla řada nadstaveb, které poskytují nad RDD další, vyšší vrstvu (ve schématu 3.3 nejvrchnější řádek). Jde například o SparkSQL, které umožňuje jednotlivé RDD transformovat do tzv. DataFrames, které jsou obdobou relačních tabulek [39]. DataFrames oproti RDD pracují také se schématem uložených data a tím mohou lépe optimalizovat dotazy. SparkSQL totiž poskytuje klasickou funkcionalitu dotazování přes SQL.

Využití v dalším návrhu

Technologie Apache Spark poskytuje komplexní řešení nejen pro distribuované výpočty. Zároveň umožňuje napojení na množství datových zdrojů, ať už se jedná např. o databáze nebo souborový systém. Těchto vlastností lze využít v dalším návrhu jako



Obrázek 3.3. Apache Spark stack architektura [37]

součást vytváření automatických business procesů, které budou zpracovávat systémem evidovaná data.

3.2.3 Apache HBase

Apache HBase je distribuovaný databázový NoSQL engine, který je postaven na technologie Apache Hadoop [40]. Tato technologie byla popsána v sekci 3.2.1. Díky tomu má HBase několik společných vlastností s Apache Spark (popsaný v sekci 3.2.2). Lze ji jednoduše horizontálně škálovat. Dále má také ve výpočetním clusteru podporu a lze ji používat jako datový zdroj. Díky těmto technologickým vlastnostem je vhodným kandidátem pro použití v implementaci.

Z pohledu kategorizace databázových strojů lze Apache HBase zařadit mezi wide-column databáze. Záznamy jsou ukládány do tabulek. Každý záznam v tabulce je poté jednoznačně identifikovatelný pomocí *rowkey*, reprezentovaného jako neinterpretované byty. Může jím být tedy cokoliv a zodpovědnost za serializaci a deserializaci je ponechána na klientech. Záznamy v tabulce jsou řazeny podle tohoto klíče lexikograficky.

Způsob, jakým ukládá data, tak lze chápat jako dvoudimenzionální key-value uložiště. V první dimenzi lze jednoduše vyhledávat pouze podle unikátního klíče každého záznamu, tzv. *rowkey*. Veškerý ostatní uložený obsah je pro tento krok nečitelný. Tímto způsobem se vybere řádek, který bude dále zpracováván. Ve druhé dimenzi, když už je řádek známý, lze přistupovat k uloženým datům podle sloupců. Ty jsou rozděleny do tzv. *column-families* (rodin sloupců). V prostředí programovacího jazyka Java je lze přirovnat k namespaces. V těchto rodinách jsou poté už samotné názvy sloupců, tzv. *qualifiers*. Kombinací *column-family* a *qualifier* lze získat konkrétní sloupec. Pomocí takto vybraného sloupce a již známého *rowkey* lze získat konkrétní hodnotu uloženou v databázi. Kvůli takovémuto designu je vyhledávání záznamů efektivní pouze podle *rowkey*. Pro efektivní vyhledávání podle jiných klíčů je nutné data indexovat. Nicméně Apache HBase neposkytuje žádný způsob indexace. Tato zodpovědnost je přenesena na uživatele, který ji v případě potřeby implementuje. Často je pro tento účel využívána jiná databáze, např. ElasticSearch.

Názvy *column-families* musejí být definovány při vytváření tabulky a obsahovat pouze tisknutelné znaky. Také by se měly udržovat v co nejkratší možné formě, neboť je jejich název ukládán společně s každým záznamem. Zároveň by jedna tabulka neměla obsahovat více než tři *column-families*. [40]

Dalším specifikem Apache HBase je možnost implicitního udržování verzí ukládaných dat. Doposud byl popisován jako unikátní identifikátor každé uložené informace jako kombinace *rowkey*, *column-family* a *qualifier*. Je-li potřeba udržovat historii dat, lze do tohoto unikátního klíče přidat ještě jednu hodnotu a tou je verze reprezentovaná UNIX časovou značkou. Časové značky jsou seřazeny v sestupném pořadí a díky nim lze udržovat historii změn v každé jednotlivé buňce [40].

Distribuované nasazení

Doposud popsané vlastnosti jsou platné jak pro standalone, tak distribuované nasazení. S distribuovaným nasazením přicházejí další problémy, se kterými se musí HBase vypořádat. Jedná se především o způsob, jakým jsou distribuována uložená data v rámci clusteru. Kompletní vlastnosti distribuovaného nasazení zde nebudou popisovány, protože nijak neovlivňují další návrh. Jednou vlastností, která jej ale ovlivnit může, je způsob samotného rozložení dat v tabulkách na jednotlivých serverech clusteru.

Všechny záznamy každé tabulky jsou rozděleny přes clusteru do shardů právě podle klíče každého řádku. Pro rovnoměrné zatížení jednotlivých serverů clusteru při náhodném dotazování je tak vhodné mít mezi ně data každé tabulky rovnoměrně náhodně rozdělená. Toho lze dosáhnout např. pomocí prefixu klíče záznamu, který bude výsledek hashovací funkce, jejíž vstupem bude klíč záznamu bez tohoto prefixu. Rozdělení jednotlivých záznamů pak lze řídit typem hashovací funkce.

Využití v dalším návrhu

Všechny zmíněné vlastnosti databáze HBase je možné využít pro ukládání naměřených dat, časových řad. Jak bylo zmíněné v popisu technologie, v této databázi lze efektivně vyhledávat pouze podle klíče každého záznamu. Z tohoto důvodu bude ještě nutné provést jeho návrh s ohledem na jeho použití.

3.2.4 JanusGraph

Pro ukládání entity společně se vztahy mezi nimi, které nemají pevnou strukturu, je vhodné zvolit specifický typ databáze. Konkrétně grafovou databázi. V systému budou mít podle analýzy v sekci 2.10 takovouto strukturu data reprezentující propojení jednotlivých komponent FVE.

Grafových databázových strojů existuje celá řada, např. Neo4j, NebulaGraph nebo – jak napovídá název této sekce – JanusGraph. Koncept ukládání dat mají všechny stejný. Ukládají dvě hlavní komponenty tvořící graf, hrany a vrcholy. Vrcholy reprezentují entity, zatímco hrany vztahy mezi nimi. Oboje je jednoznačně identifikovatelné pomocí syntetického identifikátoru, který si jednotlivé enginy vytváří. Zpravidla je pak možné oboje označit značkou (*label*) a tím vrchol zařadit do skupiny, hrana poté získává sémantický význam. Dále je možné jak na hranu, tak vrchol, přiřadit libovolné množství vlastností (*properties*), které jsou nejčastěji reprezentovány jako mapa, tedy key-value. Tyto vlastnosti více specifikují ukládanou informaci. Na základě všech těchto informací lze v grafu vyhledávat. U vyhledávání je zásadní výhoda grafové databáze, oproti jiným typům databází, tzv. traverzování – hledání vzorů, chození po grafu.

V čem se však různé typy databází liší, je způsob interního uložení dat a způsob jejich dotazování. Neo4j a NebulaGraph se v tomto ohledu velice podobají. Oba mají

proprietární implementaci backendu pro ukládání dat. Dotazování se v Neo4j provádí pomocí jazyka Cypher, který, jak zmiňuje dokumentace [41], je SQL pro grafovou databázi. Jeho zápis je díky ASCII Art like zápisu snadno čitelný i pro člověka. Velice podobné dotazování nabízí i NebulaGraph [42].

V některých zmíněných ohledech se ale JanusGraph liší. Pro dotazování a úpravy používá dotazovací jazyk Gremlin, který je součástí frameworku Apache TinkerPop [43]. Tento framework je podporovaný řadou databázových strojů, mimo jiné i již zmíněným Neo4j [44]. Všechny operace s databází jsou spjaty s transakcí. Transakce sdružuje několik operací. Každá transakce vždy pracuje se stavem databáze, který byl platný v době jejího vytvoření, tzv. snapshot. Jednotlivé transakce mohou být spouštěny paralelně a jsou od sebe navzájem izolované.

JanusGraph nemá pro ukládání dat vlastní implementaci, ale využívá již hotové wide-column, resp. key-value, databáze. Tím od nich může podědit některé užitečné vlastnosti, především jde o možnost provozovat JanusGraph v distribuovaném prostředí, snadnou horizontální škálovatelnost a vysokou dostupnost. Podporovanými datovými enginy jsou např. BerkleyDB, Apache Cassandra, ScyllaDB či Apach HBase. Další výhodou využívání existujících databázových strojů jako datového backendu je možnost provozovat JanusGraph bez nutnosti provozovat další servery. Celá databáze může pracovat nad již existující databází a stane se tak pouze další přístupovou vrstvou. Stejným způsobem lze rozšířit JanusGraph o indexační backendy, jako je Apache Solr nebo ElasticSearch. Poté je možné kromě již popsaného dotazování využívat např. fulltextové vyhledávání a všechny ostatní vlastnosti zmíněných enginů [45].

Indexování

Aby dotazování bylo efektivnější a nemusela se vždy procházet všechna uložená data, umožňuje JanusGraph vytvářet dva druhy indexů: grafové indexy a relační indexy. Grafové indexy jsou globální pro celý graf a umožňují efektivně přistupovat k uzlům a hranám podle jejich vlastností. Jsou rozděleny do dvou podkategorií. První z nich, *composite*, vytváří mapování některé vlastnosti (resp. uspořádané n-tice vlastností) vrcholu nebo hrany na jejich unikátní identifikátor. Při vyhledávání podle tohoto indexu musí být přesná shoda v hodnotách hledaných vlastností [46]. Tato možnost je tedy vhodná především pro takové vlastnosti, které mají konečnou množinu hodnot, kterých mohou nabývat. Zároveň takto lze zaručit unikátnost hodnot některé z vlastností na úrovni grafu. Pokud je potřeba v indexu vyhledávat jiným způsobem než úplnou shodou (např. vyhledávání hodnot s určitým prefixem), je nutné využít druhý typ, *mixed* index. Pro vytváření, spravování a vyhledávání v *mixed* indexu používá JanusGraph zmíněný indexační backend, tedy např. ElasticSearch [46]. Identifikátory konkrétních uzlů a hran splňující požadavek jsou vyhledány v externím indexačním enginu, podle nich jsou v JanusGraph vyhledány příslušné hrany a vrcholy, které jsou poté vráceny jako výsledek dotazu. *Mixed* indexy jsou vhodné také v případech, je-li v dotazu používáno řazení podle hodnoty určité vlastnosti, resp. vlastností.

Druhým typem indexů, které lze nad databází vytvářet, jsou *vertex-centric* indexy neboli vztahové indexy. Tyto indexy jsou uloženy lokálně (nepotřebují indexační backend) a jsou specifické pro každý jeden vrchol. I tento index se rozděluje na dva další, specializované: *edge* indexy (indexy hran) a *property* indexy (indexy vlastností). Je možné je specifikovat explicitně nebo ponechat na samotné databázi, aby je spravovala [46]. Jak indexy hran, tak indexy vlastností slouží pro urychlení traverzovacích operací (procházení grafu podle patternů) a definují se vždy pro určitou skupinu uzlů se stejnou značkou (*label*). U prvního indexu, *indexu vlastností*, jde o urychlení vyhledávání

v případě traverzování podle vlastností hran. Index totiž umožňuje rychlý přístup ke konkrétní hraně právě pomocí některé z jejích vlastností. Například lze takto vyhledávat všechny hrany, jejichž vlastnost reprezentující čas vytvoření je v dotazovaném rozsahu. Druhý typ *vertex-centric* indexů, index vlastností, je vhodné použít v případě, je-li potřeba efektivně traverzovat po vrcholech na základě jejich vlastností. Jde o podobný index jako *composite*, ale lze v něm vyhledávat i jinak než na přesnou shodu, u čísel například podle rozsahu.

Využití v dalším návrhu

JanusGraph poskytuje kompletní možnosti pro ukládání grafově orientovaných dat. Jelikož takovouto strukturu budou mít v systému evidované datové objekty a jejich vazby, bude vhodné tuto technologii využít právě pro ně. Vzhledem k požadavkům na validování a strukturu datových objektů a hran nebude možné tuto technologii použít v surové formě, pouze jako databázi. Ačkoliv JanusGraph poskytuje možnosti definice integritních omezení pomocí indexů, nelze využít pouze je. Indexy jsou totiž špatně modifikovatelné a rozšiřitelné a při každé změně je nutné manuálně provést reindexaci všech dat.

Integrace s frameworkem Apache TinkerPop je dalším důvodem, proč je vhodné zvolit tuto technologii pro ukládání dat. Pokud by v budoucnu byla shledána za nevyhovující, bude možné využít stejnou logiku dotazování, pouze dojde ke změně backendu pro ukládání dat.

3.2.5 ElasticSearch

V sekci o grafové databázi JanusGraph (3.2.4) byl ElasticSearch zmíněn jako jedna z možností indexačního backendu. Jedná se o distribuované dokumentově orientované uložiště, které umožňuje indexování různých typů dat. Ukládaná data jsou strukturována jako dokument obsahující key-value záznamy. Každý záznam takového dokumentu může být indexován jiným způsobem, zejména podle datového typu, který je v něm uložen. Například textové položky mohou být indexovány pro fulltextové vyhledávání pomocí tzv. zpětných indexů, nebo jako klíčová slova. Indexy číselných hodnot ukládá do BKD stromů (k-dimenzionální B-strom) [47]. Operace indexování a vyhledávání jsou postaveny nad Apache Lucene [48], indexační a vyhledávací knihovnou. U všech indexů si navíc ElasticSearch drží základní statistiky, jako je např. celkový počet dokumentů.

Jednotlivé dokumenty jsou rozděleny do tzv. *indexů*. Ve světě relačních databází se tento koncept dal přirovnat k tabulkám. Index specifikuje rodinu dokumentů podobných vlastností, tedy podobné struktury. ElasticSearch dokáže pracovat se strukturou dokumentů dynamicky,a není proto požadováno dopředu definovat schéma dokumentů. Nicméně ne vždy je mapování odvozeno správně. Navíc v některých případech není nutné indexovat všechny položky, které dokument obsahuje. Z tohoto důvodu je v případě, kdy je struktura dokumentů známá, lepší mapování definovat explicitně. Každý dokument je jednoznačně identifikován položkou *id*, která může být syntetická, generovaná přímo datovým enginem nebo může jít o přirozený identifikátor definovaný uživatelem. Příklad dokumentu uloženého v ElasticSearch v indexu *animals* je uveden v kódu 3.1.

Struktura dokumentů neboli *mapping* je definována na úrovni jednotlivých indexů [49]. Všechny položky v jednom indexu mají stejnou definici struktury. Pro každou položku dokumentu v indexu specifikuje, jakého má být datového typu. Podle toho se poté engine ElasticSearch rozhoduje, jak ji bude indexovat. Není nutné, aby byly indexovány všechny položky. Pokud dokumenty obsahují data, která nebudou nikdy na

```
{
  "_index": "animals",
  "_id": 1,
  "_version": 2
  "_source": {
    "species": "dog".
    "name": "Rex",
    "age": 4,
    "sound": "wofwof"
  }
}
```

Kód 3.1. Ukázka dokumentu indexovaného pomocí ElasticSearch

přímo dotazována, ale bude se k nim přistupovat až po nalezení samotné dokumentu, lze pro takové části indexování potlačit. Tato volba je zrychluje tzv. zaindexování nového dokumentu a zároveň může snížit paměťovou stopu dokumentů. Naopak, pokud to struktura dat vyžaduje, lze některé položky indexovat více způsoby (jako více datových typů).

Definováním struktury dokumentů lze ošetřit, že nebudou vloženy žádné takové dokumenty, které tuto strukturu nesplňují. Případně lze dynamicky dodávat nové položky, bude-li se struktura dokumentů měnit. Základní *mapping* použitelný pro zaindexování zmíněného dokumentu 3.1 je uveden v kódu 3.2. V něm jsou uvedeny pouze základní podporované datové typy [50]. Za zmínku stojí *keyword* a *fields*, kde na první pohled nemusí být jasný rozdíl v jejich významu. *Fields* určuje další způsoby indexování jedné položky (reprezentuje informaci jako jiný datový typ). Zároveň určuje, jak se takto indexovaná položka bude jmenovat. V ukázce je tímto typem označena položka *raw* – ke které lze přistupovat jako *species.raw*. Položka *species* bude tedy indexovaná jako *text*, tak pod názvem *raw* také jako typ *keyword*. Tento typ je vhodný pro pevně strukturovaná data, které se v dotazech využívají převážně na přesnou shodu. Případně se podle nich často provádějí agregace. Lze také říci, že je vhodný pro taková data, která nabývají spočetného množství hodnot. V ukázce je typem *keyword* označena položka *species* – druh zvířete. Předpokládané dotazování pro položkou tak může být například hledání všech psů jedné rasy atp.

Z pohledu fyzického uložení dat jsou *indexy* rozděleny do shards (každý shard obsahuje kus indexu), které jsou rozprostřeny po clusteru. Zároveň existuje několik replik každého shardu [51]. Hodnoty shardingu a replikace je možné konfigurovat. Tento mechanismus využívá ElasticSearch pro zvýšení dostupnosti dat, lze je číst z více zdrojů a zároveň v případě ztráty dat u některého z node je lze jednoduše obnovit.

Uložená a indexovaná data jsou uživateli zpřístupněna přes REST API. Pro dotazování je používán doménově specifický jazyk zapisovaný jako JSON dokument. Podle datového typu jednotlivých položek, resp. způsobu jejich indexování, je možné podle nich vyhledávat a filtrovat dokumenty. Většina dotazů, která je nad uloženými daty prováděna, je dokončena téměř v reálném čase. Složitější dotazy, které trvají déle, lze vykonávat asynchronně. Clusteru je předán dotaz a klient se poté periodicky dotazuje (polling) na stav vykonávaného dotazu. Ve chvíli, kdy se dotáže na stav a dotaz již doběhl, může si klient stáhnout výsledek.

```
{
  "animals" {
    "mappings": {
      "properties": {
        "species": {
          "type": "text",
          "fields": {
            "raw": {
              "type": "keyword"
            }
          }
        },
        "name": {
          "type": "text"
        },
        "age": {
          "type": "integer"
        },
        "sound": {
          "type": "text",
          "index": false
        }
      }
    }
  }
}
```

Kód 3.2. Ukázka mappingu použitého pro indexování dokumentu 3.1

Využití v dalším návrhu

Ačkoliv tato technologie poskytuje kompletní možnosti pro ukládání a vyhledávání v dokumentech, nebude pro další návrh využita explicitně, pouze jako případný indexační backend pro JanusGraph. Jedná se ale o technologii, která by mohla být využita, např. pokud by bylo nutné systém rozšířit o evidenci složitějších dokumentů s možností vyhledávání. Případně se může jednat o alternativu pro ukládání dat časových řad.

3.2.6 Apache Kafka

Apache Kafka [52] je streamovací platforma. Řeší problémy, které vznikají při on-line zpracovávání velkého objemu dat. Přijímá data od více producentů, poté je ukládá do front. Z těch poté může několik konzumentů odebírat přijaté zprávy. Platforma tak zajišťuje buffering a brání v přehlcení klientů a zároveň může distribuovat přijaté zprávy mezi více klientů a fungovat jako load balancer. V případě potřeby navýšení výkonu lze platformu horizontálně škálovat.

Platforma pracuje s daty jako s takzvanými zprávami (*messages*). Každá zpráva je producentem (*producer*) – klient posílající zprávy do Apache Kafka – odeslána *brokeru* do určité fronty. Fronty se nazývají *topics*. Oproti messaging systémům, jako je např. RabbitMQ, mají zprávy definovanou dobu, po kterou jsou v *topicu* uloženy a nejsou mazány po potvrzení jejich doručení. Zprávy mohou být před odesláním serializovány do binární podoby, a tím snížena jejich velikost. Podporované formáty serializace jsou např. Protobuf [53] nebo Apache Avro [54]. Zároveň je tímto mechanismem ošetřena

validace struktury dat. Pokud příchozí zpráva nejde deserializovat dle definovaného, například Avro, schématu, je považována za neplatnou a není do fronty zařazena. Dále je zajištěno, že příjemce zprávy bude znát její strukturu.

Příjemci zpráv se nazývají konzumenti (*consumers*). Zprávy jim nejsou doručovány asynchronně (push model), ale musejí se aktivně doptávat na *broker* a vyčítat z něj zprávy – v Apache Kafka jsou *brokers* pasivní. Konzumenti jsou rozděleny do logických celků nazývaných *consumer groups*. Ty jsou důležité například v případě, existuje-li více konzumentů jedné fronty, kteří zpracovávají příchozí zprávy stejným způsobem, a je potřeba zajistit, aby každá zpráva byla doručena nejvýše jednomu z nich. Každá zpráva je totiž doručena každé skupině konzumentů maximálně jednou, resp. maximálně jednomu konzumentovi z každé skupiny [55]. Vhodný počet konzumentů je úzce spjatý s tzv. *partitioning* konzumovaného *topic*.

Při velkém množství příchozích dat je možné paralelizovat jejich zpracování přidáním více konzumentů jednoho *topic*. Tito konzumenti budou sdílet stejnou skupinu. Tedy každá zpráva bude doručena právě jednomu z nich. Aby bylo možné tuto operaci provést, musí být *topic* rozdělen do více *partitions*. Používaný název – fronta – není úplně správný, Kafka nezaručuje, že zprávy budou doručeny v pořadí, v jakém byly do *topicu* zařazeny. Tato vlastnost je zaručena pouze na úrovni těchto *partitions*. Jejich počet se definuje při vytváření *topic*. Zde přichází provázanost počtu konzumentů a zmíněných *partitions*. Každému konzumentovi ze skupiny je přidělena jedna část *topic* v případě, jsou-li tyto dva počty shodné. Bude-li více částí než konzumentů, některý konzument bude získávat zprávy z více částí. Naopak, bude-li více konzumentů než částí *topicu*, dostane každý konzument nejvýše jednu část, ze které bude získávat zprávy. Nutně tedy zůstane některý konzument nevyužitý, nebude zpracovávat žádné zprávy.

Dále musí každý konzument kvůli vlastnosti zmíněné v předchozím odstavci, že se zprávy po přečtení nemažou, udržovat informaci o tom, které zprávy už přečetl a které nikoliv. Jelikož jeden konzument čte vždy z jednoho *partition*, ve kterém jsou zprávy seřazeny, stačí mu pro tento účel udržovat pouze „zarážku“ za poslední přečtenou zprávou (neboli *offset*). Pro případ, že by došlo k výpadku konzumenta, a tím ke ztrátě informace o *offset* (v době výpadku mohl zprávy zpracovat jiný konzument ze skupiny), je tato informace replikována do speciálního *topicu* `__consumer_offsets`, ze kterého si ji může kdykoliv znovu přečíst [56].

V případě velké zátěže *brokeru*, tedy velkého množství příchozích zpráv, lze vytvořit cluster, ve kterém bude více *brokers*. Každý z nich poté bude spravovat pouze několik *partitions* několika *topics*. Nebo-li části jednoho *topicu* budou rozděleny mezi několik *brokers*. Jednotlivé části lze v clusteru také replikovat. Tím lze zajistit větší odolnost proti chybám. V případě výpadku jednoho *broker* je jeho činnost nahrazena ostatními, kteří mají repliku *partitions*, které obsluhoval.

Kafka Streams

Použitím knihovny Kafka Streams lze provádět tzv. *stream processing* [57]. Oproti předchozímu popisu, který se týkal spíše *messagingu*, není jeho hlavním cílem předávání informací mezi systémy, ale zpracování streamu dat. Během tohoto zpracování jsou data ochuzována, modifikována a obohacována. Kafka Stream aplikace kontinuálně čtou ze vstupních *topics*, provádějí transformace a zapisují výsledky do výstupních *topics*. Jednotlivé fronty tak slouží k předávání (mezi)výsledků mezi jednotlivými transformacemi. Dělí se do tří typů, vstupní, výstupní a průběžné (vstupně/výstupní). Vstupní, z pohledu streamu, pouze poskytují data a nelze do nich zapisovat. Do výstupních lze naopak

pouze zapisovat. A do průběžných lze zapisovat i z nich číst. Jinými slovy, každý průběžný *topic* obsahuje výsledek nějaké transformace a je vstupem pro jinou transformaci.

Transformace prováděné nad streamem jsou pouze předpisem, jak s daty zacházet. Jsou vyhodnocovány až v případě, je-li to skutečně potřeba. Dále posloupnost transformací musí tvořit orientovaný acyklický graf. Všechny tyto vlastnosti jsou shodné s vlastnostmi transformací RDD v Apache Spark popsaných v sekci 3.2.2.

Serializační systém – Apache Avro

V sekci 3.2.6 bylo zmíněno, že zprávy odesílané na *broker* mohou projít určitým způsobem komprese a serializací do binární podoby podle předem definovaného schématu. Všechny serializační systémy poskytují svůj doménově specifický jazyk pro definici dokumentů. Nijak se neliší v konceptuálním pojetí serializace. Z toho důvodu zde bude popsán pouze jeden, a to Apache Avro. Je jedním z podporovaných formátů v Apache Kafka.

Fungování Avro schémat je přímočaré. Uživatel definuje strukturu v doménově specifickém jazyce v notaci JSON. Tento předpis slouží pro vygenerování objektu, serializátoru a deserializátoru. Příklad takové definice je uveden v kódu 3.3. Každá položka je definována svým jménem a datovým typem. Jednotlivá schémata lze do sebe libovolně zanořovat. Jedna položka může mít více možných datových typů. Toho lze využít v případě, je-li potřeba takovou položku označit jako nepovinnou. Pouze jí bude přidělen druhý datový typ, null. Příkladem takového použití je *sound* v kódu 3.3. Možné je také využívat komplexnější datové typy: pole, mapa (v podstatě zanořené schéma), výčtové typy (enum) atd. Příklad použití výčtového typu je uveden opět v ukázce kódu 3.3 u položky *gender*. Výchozí hodnoty (`default`: `unknown`) jsou uvedeny kvůli zajištění dopředné a zpětné kompatibility.

Schema registry

Aby měl jak *producer*, tak *consumer* stejná schémata, lze je distribuovat pomocí tzv. *schema registry*. Zároveň je možné jednotlivá schémata v tomto registru verzovat. Každý klient může používat jinou verzi. Z toho důvodu je také vhodné zachovávat zpětnou kompatibilitu a nevytvářet změny, které zabraní klientům se starší verzí schématu jejich deserializaci. Příkladem takového registru může být Confluent Schema Registry [58].

Ačkoliv je jeho primární určení udržovat schémata pro komunikaci s Apache Kafka, lze jej také využít pro ukládání jiných dokumentů. Podporované formáty jsou JSON, AVRO a PROTOBUF. Pro manipulaci s uloženými dokumenty a jejich registraci je možné využít poskytované HTTP API, případně knihovny, které jsou dostupné pro řadu jazyků, mj. pro Javu.

Využití v dalším návrhu

Apache Kafka, zejména se zmíněným streamovým zpracováním dat, je vhodnou technologií pro využití při vkládání nových dat do systému. Systém by touto technologií získal robustní vstupní bod, do kterého je možné nahrávat požadavky (nová data), které nemusejí být ihned zpracovány, ale lze je zpracovávat postupně podle volného výpočetního výkonu. Zároveň by s pomocí této technologie bylo možné jednoduše rozšiřovat zpracování surových vstupních dat novými transformacemi (streamy).

3.2.7 Apache Airflow

Odečítání dat ze senzorů a jiných komponent, společně s řízením definicí automatických úloh (a jiných business procesů), bude muset být prováděno periodicky a automatizo-

```
{
  "namespace": "com.example.avro.animals",
  "type": "record",
  "name": "AnimalEnity",
  "fields": [
    {
      "name": "species",
      "type": "string"
    },
    {
      "name": "age",
      "type": "int"
    },
    {
      "name": "sound",
      "type": ["null", "string"]
    },
    {
    "name": "gender",
      "type": {
        "type": "enum",
        "name": "Gender",
        "symbols": [
          "female",
          "male",
          "unknown"
        ],
        "default": "unknown"
      },
      "default": "unknown"
    }
  ]
}
```

Kód 3.3. Ukázka definice Avro schématu

vaně. Stejně tak bude potřeba spouštět analytické výpočty, které zpravidla potrvají dlouho. Všechny tyto operace je nutné koordinovat. Jednotlivé operace získávání dat se také budou moct lišit podle typu komponenty FVE. Jednotlivé komponenty budou požadovat různé intervaly odečítání atp. Pro koordinaci těchto operací je vhodné zvolit službu, která poskytuje možnost popisovat takovéto procesy a spouštět jejich vykonání.

Apache Airflow [59] je platforma umožňující definovat, plánovat a monitorovat workflow. Ty se definují pomocí skriptů v programovacím jazyce Python. Používá se tak high-level jazyk, takže je možné využít všechny jeho funkcionality včetně případného rozšíření o další knihovny.

Workflow se skládá z operátorů, které jsou řetězeny za sebe tak, aby tvořily acyklický orientovaný graf. Tím je zaručeno, že každé workflow bude možné vykonat v konečném čase a nedojde k zacyklení. Jednotlivé operátory si mohou mezi sebou předávat informace, např. výsledky operací, které mohou reagovat. Bez dalších rozšíření jsou poskytovány operátory na komunikaci protokolem HTTP, spouštění Bash skriptů atd. [60]

Přidáním rozšíření lze získat další operátory, jako SparkSubmitOperator, který umožňuje odesílat jobs do Apache Spark clusteru [61]. Speciálním typem operátorů jsou poté senzory. Poskytují pouze jednu funkcionalitu, a to čekání na událost. Například takto lze čekat na dokončení výpočtu, který byl spuštěn, periodickým dotazováním na jeho stav v určitém časovém intervalu [62].

Pro zobrazení workflows, logů jejich exekucí a dalších informací má Apache Airflow dostupné webové rozhraní. Přes něj lze také jednotlivá workflows spouštět, zároveň jim zde předávat i parametry pro spuštění. Další možností spuštění je použít časovač. Každé workflow může mít definované periodické spouštění. Pokud je potřeba exekuci spustit externím systémem, je za tímto účelem poskytováno HTTP API, přes které lze operaci provést. Zároveň rozhraní poskytuje přístup k celé platformě Airflow, takže z něj lze získat mimo jiné i informace o všech běžích daného workflow, jejich stavu atp.

Využití v dalším návrhu

Apache Airflow lze pro další návrh použít už jako hotovu doplňkovou službu. S využitím této služby bude možné snadno spouštět Apache Spark Jobs (pomocí zmíněného rozšíření SparkSubmitOperator) na zpracování dat uložených v systému. Případně budou možné přímo v Airflow, které bude mít přístup ke komponentám systému, implementovat nové business procesy. Díky jazyku Python, ve kterém lze psát skripty (DAG) pro Airflow, je tímto způsobem možné zpřístupnit API komponent, které půjde obsluhovat bez znalosti vnitřní architektury systému.

3.2.8 JSON Schema

Oproti předchozím není *JSON Schema* technologie, která by sama o sobě poskytovala funkcionality. Jedná se o deklarativní jazyk, kterým je možné popsat strukturu objektů. Syntaxe vychází z notace JSON.

JSON Schema je možné použít několika způsobit. Lze s jeho pomocí definovat datové objekty pro komunikaci mezi službami, tzv. DTOs (Data Transfare Objects). Toto využití je vhodné zejména při návrhu několika služeb, které mezi sebou komunikují, a tudíž potřebují unifikované a předem známé struktury zpráv. Existují i knihovny, které z *JSON Schema* definic generují datové objekty pro konkrétní programovací jazyk (např. pro Javu, TypeScript apod.) [63]. Lze tedy jednoduše vygenerovat interface pro komunikaci mezi službami.

Dalším použitím této technologie je možnost validovat JSON dokumenty oproti definovaným schématům. *JSON Schema* poskytuje konstrukty pro definování integritních a validačních kritérií, na základě kterých lze validace provádět. Zvalidované JSON dokumenty poté obsahují předem známé objekty předem daných typů. Zároveň, pokud dokument nesplňuje definované schéma, lze jednoduše najít jeho porušení a případně informovat, ve kterých místech dokument definované schéma porušuje.

Všechna schémata mohou mít rodičovské schéma, které definuje strukturu svých potomků. Takto lze vytvořit hierarchickou strukturu schémat, kdy každý rodič bezprostředně ovlivňuje strukturu svých potomků. Zároveň existují meta-schemata (např. `draft-7` nebo `2020-12` [64]), která obsahují definice jazyka JSON Schema. Z nich budou jednotlivá implementovaná JSON schémata vycházet.

Využití v dalším návrhu

Tento deklarativní jazyk je možné využít zejména pro definici struktur datových objektů a jejich vazeb. S pomocí těchto definic je možné navrhnout komplexní datové struktury a na jejich základě poté například validovat JSON dokumenty reprezentující příchozí

požadavky. Technologii bude možné uplatnit zejména v návrhu datového modelu a na něj navazujících problémech.

3.3 Programovací jazyk, framework

Dosavadní návrh se zabýval obecnými koncepty v kontextu potenciálně použitelných technologií pro další návrh systému. Dále je také nutné zvolit konkrétní programovací jazyk, případně celý framework, ve kterém bude systém implementován. Jelikož navrhovaný systém má být funkčním prototypem, je vhodné zvolit takové prostředky, které maximálně usnadní práci, zrychlí vývoj a zajistí flexibilitu při častých úpravách. Není totiž vyloučeno, že některá část implementace bude v blízké době upravena a může dojít i ke změně používané technologie, pokud se z jakéhokoliv důvodu bude jevit jako nevyhovující. Tyto požadavky přímo směřují na použití frameworku.

Nejprve však k volbě programovacího jazyka. Systém má být možné provozovat na prostředí s OS Linux – nefunkční požadavek N2. Dále budou použity technologie, jako JanusGraph, Apache Spark, ElasticSearch, Apache HBase, které jsou všechny implementované v programovacím jazyce Java (případně jiném jazyce pracujícím nad JVM). Všechny také poskytují nativní aplikační rozhraní pro Javu. Proto pro zachování soudržnosti a zaručení případné jednoduchosti v interoperabilitě jednotlivých technologií bude pro implementaci zvolen také jazyk Java.

3.3.1 Backend

V úvodu sekce bylo nastíněno, že pro rychlejší a snadnější vývoj je vhodné využít framework, který řeší mnohé, často se opakující problémy. Známým frameworkem pro zvolený jazyk Java je Spring [65]. Pro své služby jej využívá např. společnost Netflix [66]. Hlavním konceptem využívaným v tomto frameworku je Inversion of Control (IoC). Framework poskytuje IoC kontejner, do kterého ukládá všechny komponenty, tzv. Beans. Pomocí tohoto mechanismu pak může programátorovi nabídnout jednoduché využívání Dependency Injection (DI), protože má všechny potřebné závislosti uložené ve zmíněném kontejneru. Tento velice zjednodušený popis je hlavní výhodou používaní Spring. Programátor si pouze definuje jednotlivé funkční celky (Beans), vytvoří mezi nimi závislosti, ale už se nestará o proces, jakým se tyto závislosti vyřeší. Z jeho pohledu jsou automaticky získány a vloženy přesně tam, kde je definoval. Další výhodou IoC kontejneru je možnost jednoduše měnit implementaci určitého funkčního celku. Pokud je dodržen kontrakt (interface), stačí pouze poskytnout IoC kontejneru novou implementaci (Bean), ta se automaticky dostane pomocí DI do všech míst v kódu, kde je požadována, a není potřeba další úprava. Stejný mechanismus lze využít například pokud je rozdílná implementace určité funkcionality pro vývojové a produkční prostředí, zpravidla připojení do databáze atp. Podle potřeby se IoC kontejneru poskytne požadovaná implementace kontejneru.

Vzhledem k popularitě Spring frameworku existuje nespočet rozšiřujících knihoven, které slouží k řešení nejčastějších problému, se kterými se setkává mnoho implementací, jako je načítání konfigurace, implementace REST kontrolérů, klienti do SQL/NoSQL databází, zabezpečení aj. Pro široce využívané technologie proto existují oficiální knihovny pro rozšíření Spring frameworku. Nejvíce populárním rozšířením je Spring Boot.

Spring Boot poskytuje vše potřebné pro vytváření standalone aplikací. Programátor je tak oproštěn od nutnosti zabývat se samotným sestavením spustitelného Java archivu. Dále poskytuje vestavěný aplikační server, možnost používání tzv. starters závislostí. Ty automaticky poskytnou IoC kontejneru všechny potřebné Beans, které jsou

předkonfigurované pro základní použití jimi poskytovaných funkcionalit. Všechny tyto vlastnosti se snaží odstínit programátora od nutnosti psát boilerplate kód a umožnit mu zaměřit se především na implementaci business logiky.

Jedním z příkladů knihovny, která poskytuje i zmíněný starter, je Spring Boot Actuator [67]. V analýze v sekci 3.1 byl mimo jiné zmíněn problém se sledováním jednotlivých mikroslužeb a trasování průtoku dotazů systémem (několika microservisami). Právě k tomuto účelu slouží zmíněná knihovna. Její součástí je fasáda Micrometer [68], která umožňuje napojení na různé nástroje, mimo jiné na zmíněný OpenTelemetry.

Dalším rozšířením, které je postaveno nad Spring Boot, je Spring Cloud. To je určeno pro vytváření systémů s mikroservisní architekturou, případně jiných distribuovaných systémů. Oproti Spring Boot už neřeší podporu implementace jednotlivých (mikro)servis, ale jejich vzájemné propojení. Opět je distribuovaný jako knihovna, která využívá koncept starter. Lze s jeho pomocí jednoduše vytvořit service discovery či loadbalancing bez nutnosti psaní velkého množství kódu (problémy mikroservisní architektury popsané v sekci 3.1). Existuje oficiální implementace těchto vzorů v knihovně s názvem Eureka od společnosti Netflix [69].

Kromě Spring a jeho rozšíření existují další frameworky, které cílí na vytváření distribuovaných systému. Je jím např. Micronaout [70]. Ten poskytuje stejné informace jako Spring Boot. Nativně však podporuje service discovery a loadbalancing, který je nutné do Spring Boot dodat rozšířením. Hlavním rozdílem mezi těmito dvěma frameworky je způsob vytváření DI. Zatímco Spring (potažmo Spring Boot) dělá tyto operace až za běhu, Micronaout řeší DI v čase sestavování aplikace. Tím předchází běhovým chybám, kdy nelze najít správnou implementaci, a zároveň zrychluje start aplikace.

Pro implementaci bude zvolen framework Spring Boot. Oproti frameworku Micronaut má větší podporu komunity, což implikuje i větší množství rozšiřujících knihoven. Nedostatky s procesy, které jsou nezbytné pro mikroservisní architekturu a ve frameworku Micronaout jsou nativně podporovány, lze vyřešit využitím rozšíření Spring Cloud.

3.3.2 Uživatelské rozhraní

Pro uživatelské rozhraní bude nutné volit jinou technologii a programovací jazyk než pro backendové služby. Pro snazší implementaci, rozšiřitelnost a udržitelnost je vhodné, stejně jako u backendových služeb, využít některý z existujících frameworků. Většina takovýchto frameworků je postavena nad jazykem JavaScript. Jsou jimi např. Vue.js, React či Angular. Nicméně existují i frameworky pro jiné jazyky, např. Nette pro PHP.

Pro implementaci uživatelského rozhraní bude zvolen jazyk JavaScript (resp. jeho typované rozšíření TypeScript), zejména kvůli jeho přetrvávající popularitě. Z vyjmenovaných frameworků bude poté použit Angular. Strukturou svého kódu, podporou dependency injection atp., je mírně podobný zvolenému frameworku Spring Boot pro backendové služby. Výhodou JavaScript frameworku je také jeho produkční výstup. Jedná se většinou o SPA (single page application), kterou je snadné nasadit. Taková aplikace se stará o všechny nezbytné procesy, jako je vykreslování, dotazování na data či směrování. V případě fullstack vývoje – který je pro navrhovaný systém nezbytný – je tedy ze zmíněných důvodů Angular vhodný.

3.4 Datový model

Tato sekce je zaměřena na popis datového modelu ve všech částech systému. Popisuje, jak budou data ve vybraných technologiích ukládána. Návrh datového modelu je rozdělen podle typu zpracovávaných dat na dvě části – ukládání datových objektů se vztahy

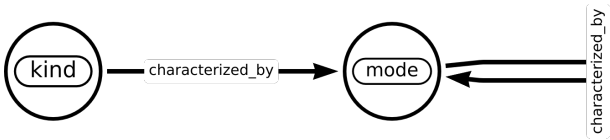
a ukládání dat časových řad. Pro lepší přehlednost jsou u každého popisu přidána schémata, která napomáhají k pochopení navrhovaných struktur, konceptů způsobu uložení a práce s daty.

3.4.1 Ukládání vztahů mezi objekty

Datové objekty a jejich vztahy budou ukládány do databáze JanusGraph (popsané v sekci 3.2.4). V návrhu a tedy i v následné implementaci na ně bude nahlíženo jako na graf, kde vrcholy budou reprezentovat jednotlivé datové objekty a hrany vztahy mezi nimi.

Z pohledu systému budou datové objekty rozděleny do několika kategorií. Především se bude jednat o kategorie spojené s doménou FVE. Některé kategorie budou ale více obecné. Každá kategorie bude reprezentovat datové objekty (např. komponenty FVE) podobných vlastností (např. všechny takové komponenty FVE, které poskytují nějaká data). Zároveň všechny tyto kategorie budou nést rigidní informace o daném objektu – takové informace, které se po celou dobu existence objektu v systému nebudou měnit (např. výrobní číslo, výrobce atp.). Na základě některé z těchto informací, případně kombinace několika z nich, bude možné každý objekt jednoznačně identifikovat. Tyto kategorie budou v databázi reprezentovány vrcholy se *značkou* `Kind`.

Udržovat pouze neměnné informace není dostačující. Všechny informace, které se mohou měnit, ale jsou navázány na konkrétní datový objekt, budou uloženy separátně jako vrcholy se *značkou* `Mode`. Těmto vrcholům je ale potřeba přiřadit zdroj identity, jinak by sémanticky nedávaly smysl, nešlo by určit, ke které entitě patří. Tím pádem by v nich ani nešlo správně vyhledávat. K propojení dodatečných informací (vrcholů `Mode`) s entitou, ke které patří bude sloužit hrana `characterized_by`. Pokud bude potřeba mít přídavné informace (uložené ve vrcholech `Mode`) s větší granularitou, lze stejnou hranou, `characterized_by`, svázat dva vrcholy typu `Mode`. Jelikož neposkytují identitu, bude vždy jednoznačně rozlišitelné, ke kterému vrcholu `Kind` informace patří. Schematicky je tato struktura uvedena na obrázku 3.4.



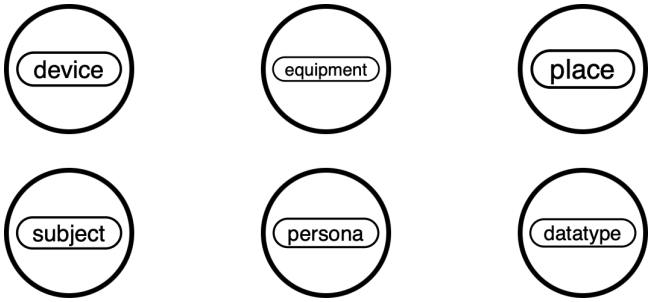
Obrázek 3.4. Propojení vrcholů `Kind` a `Mode`

Tento návrh také umožňuje vést historii změn přidružených informací pomocí mechanismu popsaného na začátku této sekce. Pro názornost je zde uveden příklad, jak bude probíhat update informací uložených ve vrcholu typu `Mode`: *Vytvoří se nový vrchol se značkou* `Mode`. *Hrana, která vede k upravovanému vrcholu, se označí za neplatnou – vyplní se vlastnost* `deleted_at` *a nastaví se příznak* `deleted` *na* `true`. *Nový vrchol se propojí novou hranou* `characterized_by` *začínající ve stejném vrcholu, ve kterém začíná zneplatněná hrana. Tato hrana bude označena za platnou (nebude mít vyplněnou vlastnost* `deleted_at` *a příznak* `deleted` *bude nastaven na* `false`*).*

Vrcholy typu `Kind` mohou být instancí některého z následujících podtypů: `Equipment`, `Device`, `Place`, `Subject`, `Persona`, `Datatype` (schematicky zobrazeno na obrázku 3.5). Podtypy budou v databázi realizovány pomocí *značky* odpovídající jejich názvu s prefixem odpovídajícím *značce* jejich rodiče. Jde tedy o hierarchické uspořádání, všechny zmíněné typy jsou potomky typu `Kind`. Z pohledu systému budou typu `Device` všechny komponenty FVE, které poskytují aktuální informace o svém stavu (nebo o stavu jiné

komponenty). Oproti tomu `Equipment` budou takové komponenty, které neposkytují žádné informace, nemají žádný datový výstup. Jsou to především pasivní části FVE, jako např. panely. Poslední dva podtypy se již nevztahují ke komponentám FVE. Vrcholy typu `Place` budou určovat fyzická místa (jako je např. odběrné místo) a vrcholy typu `Subject` budou označovat fyzické či právnické subjekty (např. skupina sdílení). Dále vrcholy typu `Persona` specifikují uživatele (osobu) určitého typu. `Datatype` mohou být objekty, které nesou pouze obecnou datovou informaci (např. alokační klíč).

Pro vrcholy typu `Mode` nebude navrhovaná žádná specializace, neb tyto nebudou potřeba při realizaci navrhovaného prototypu. Nelze je však úplně opomenout, dodávají zde totiž kontext, jakým způsob lze datový model dále rozšiřovat. Případné podtypy typu `Mode` by byly stejně jako u předchozího typu `Kind` rozlišeny v databázi *značkou* odpovídající jejich názvu s prefixem odpovídajícím *značce* jejich kategorie.



Obrázek 3.5. Specializace typu `Kind`

V databázi bude nutné udržovat ještě další vazby mezi objekty. Ty už nebudou mít více hlavních typů, jako tomu u objektů. Existovat bude pouze jeden, se *značkou* `relation`. Jak již bylo nastíněno, bude existovat hrana `characterized_by`, která bude směřovat z vrcholu typu `Kind` (případně `Mode`) do vrcholu typu `Mode`. Další vazby budou již spojeny s jednotlivými podtypy `Kind`. Budou více restriktivní. Hrana `wire_with` bude reprezentovat fyzické propojení a může být vedena z `Equipment` do `Device`, stejně tak jako mezi dvěma vrcholy typu `Equipment` nebo `Device`. Pro logické propojení (především datové, případně reprezentující jiný nehmotný tok) bude použita hrana `feed_from`, která má následující omezení – může být vedena z vrcholu typu `Device` do vrcholu typu `Equipment`. Dalším typem hrany je `install_at`, která může být vedena z vrcholu `Equipment` nebo `Device` do libovolného vrcholu `Place`. Sémanticky zachycuje, že daná komponenta (počáteční vrchol) je fyzicky umístěna na konkrétním místě (koncový vrchol). S vrcholem typu `Place` je svázán také typ hrany `managed_by`, která může vést ze zmíněného typu vrcholu do vrcholu typu `Subject`. Touto vazbou lze zachytit správce, případně jiný subjekt svázaný s daným místem (vrchol `Place`). Posledním typem hrany je `act_as`. S využitím této hrany je možné v modelu zachytit zastupitelnost jednotlivých subjektů (hrana mezi dvěma vrcholy typu `Subject`), případně pomocí ní specifikovat typ daného subjektu (hrana vedená z vrcholu typu `Subject` do vrcholu typu `Persona`).

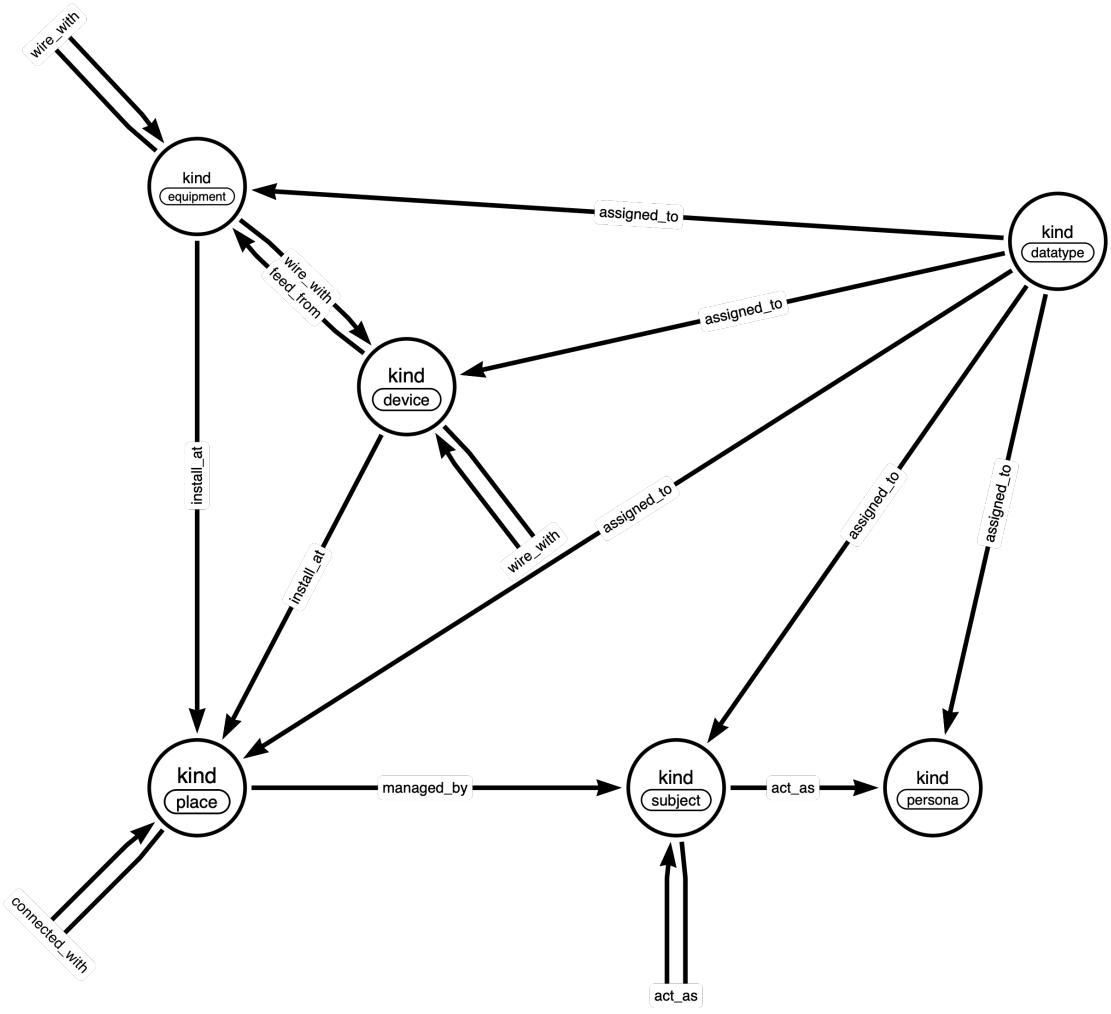
Schéma celého metamodelu typů objektů a vazeb mezi nimi je zachyceno na obrázku 3.6. Pro přehlednost je zanedbaná vazba `characterized_by` ze všech typů na `Mode`. Dále jsou stejné informace shrnuty v tabulce 3.1.

Integritní omezení

Všechny principy popsané v sekci 3.4.1 definovaly omezení na ukládaná data. Při vkládání nových informací (a případně při jejich úpravách) je potřeba validovat, aby byla

hrana	počáteční vrcholy	koncové vrcholy
characterized_by	equipment, device, place, subject, persona, datatype	mode
install_at	equipment, device	place
connected_with	place	place
wired_with	equipment, device	equipment, device
feed_from	device	equipment, device
act_as	subject	subject, persona
managed_by	place	subject
assigned_to	datatype	equipment, device, place, subject, persona

Tabulka 3.1. Navržené vazby datového metamodelu



Obrázek 3.6. Schéma navrženého metamodelu objektů a vazeb

tato omezení dodržena. K tomu bude potřeba mít schéma se všemi omezeními jedno-

značně definované, uložené a přístupné všem částem systému, které jej budou potřebovat. Schéma nesmí být závislé na použité databázové technologii.

Definice struktury a integritních omezení bude muset mít následující parametry. Pro každý `Kind`, každý `Mode` a každou hranu bude existovat jeden záznam. U hran bude záznam obsahovat název, typ možného koncového a počátečního vrcholu. Pro oba konce hrany také aritu (kolik hran stejného typu může vycházet z jednoho vrcholu, resp. končit v jednom vrcholu) a konečně seznam vlastností, které lze hraně přiřadit, které je nutné hraně přiřadit a jakého typu má která vlastnost být. U vrcholů (`Kind`, `Mode` a jejich podtypů) bude struktura záznamu jednodušší. Bude obsahovat pouze název a seznam vlastností, které mohou být objektu přiřazeny, které musejí být objektu přiřazeny a jakého typu má která vlastnost být. Jde o stejný princip, jako byl popsán u hran. Popsanou strukturu lze efektivně ukládat, mimo jiné, do libovolné relační databáze. S jejím využitím by ale definice schémat byly závislé na použité databázové technologii, což není žádoucí.

Tato integritní omezení však lze bez úhony reprezentovat také jako strukturovaný JSON dokument, pomocí kterého by bylo možné validovat strukturu a typy vlastností hran a uzlů ukládaných do databáze. K tomuto účelu je možné využít zmíněný deklarativní jazyk JSON Schema (popsaný v sekci 3.2.8), pro který existují validátory pro celou škálu programovacích jazyků. Stejné schéma lze také použít pro ověřování počátečních a koncových vrcholů hrany, stejně tak pro kontrolu počtu hran. Zde sice nelze využít implementované validátory, neboť nejde čistě pouze o validaci struktury a hodnot JSON dokumentu. Pro tuto validaci je nutné brát v potaz kontext získaný ze samotné databáze. Nicméně tuto funkcionalitu lze snadno doimplementovat. Další výhodou oproti relační (či jiné) databázi je možnost automatického zpracování definic. Lze z nich např. generovat Java třídy. JSON Schema také umožňuje vytvářet hierarchickou strukturu, což přesně koresponduje s návrhem datového modelu popsaným v sekci 3.4.1.

Přidávat integritní omezení, případně další podtypy `Kind`, resp. `Mode`, nebo hranu, lze dynamicky a není nutný kvůli takovéto změně žádný zásah do kódu ani do schématu zvolené databáze JanusGraph. Pouze se definuje nové JSON schéma, podle kterého se bude validovat nový vrchol, resp. hrana. Tato vlastnost dává možnost jednoduché rozšiřitelnosti systému pro evidování dalších objektů zájmu.

Při definici nových a modifikaci stávajících dokumentů je potřeba brát v potaz vlastnosti databáze JanusGraph, která bude použita. Jedná se zejména o způsob interní evidence schématu a jiných integritních omezení. Databáze eviduje pro každý název vlastnosti její datový typ (vlastnost se stejným názvem musí mít stejný typ u všech vrcholů). Tento typ nelze v budoucnu změnit bez provedení ruční reindexace dat. Přidávání, potažmo i odebírání, nových položek ze schématu lze provádět bez problémů. Po těchto operacích není potřeba provádět reindexaci dat. Změny množin povolených typů koncových a počátečních vrcholů nijak nezasahují do interní evidence JanusGraph a lze je tak z tohoto pohledu modifikovat bez omezení. Opět zde bude nutné myslet na to, že odebráním záznamu z této množiny může dojít k narušení integrity dat. Po modifikaci zůstanou v databázi uloženy všechny objekty a hrany, z nichž některé již nemusejí splňovat upravené schéma.

3.4.2 Ukládání dat do časových řad

V kapitole 2 bylo popsáno, jaká data bude potřeba vytěžovat a ukládat a k jakým dotazům budou poté použita. Stejně tak byla specifikována jejich abstraktní struktura, resp. povaha. Pro ukládání časových řad, není vhodné použít stejnou databázi, jako bude použita pro ukládání vztahů mezi objekty. Tato sekce se věnuje více podrobně, jak data

časových řad strukturovat a ukládat s ohledem na zvolenou technologii Apache HBase (3.2.3). Samozřejmě existují již implementované specializované databázové stroje pro tento typ dat, jako je např. InfluxDB [71]. Nicméně pro udržení konzistence a minimalizaci technologií nutných pro provoz systému bude pro návrh (a následnou implementaci) zvolen zmíněný datový engine Apache HBase. Zároveň je touto volbou poskytnuta obecná databáze, do které případně lze ukládat libovolná jiná data nebo v budoucnu rozšířit implementaci evidence časových řad tak, jak by to v jiných databázích nebylo možné.

Při volbě, jak data ukládat, je nutné vzít v potaz vlastnosti používané technologie a požadavky na práci s ukládanými daty. Jelikož je HBase v zjednodušené představě multidimenzionální key-value databáze (její základní rysy byly popsány v sekci 3.2.3), je nejprve nutné navrhnout klíč (rowkey), pod kterým se budou data ukládat. Tato hodnota je jediná, podle které lze v databázi efektivně vyhledávat (bez nutnosti provádět při dotazech full-table scan, případně vytvářet externí indexování), je nutné, aby obsahovala všechny informace, podle kterých bude potřeba ukládaná data filtrovat.

Evidence časových řad bude implementována jako Java knihovna zprostředkovávající přístup do databáze HBase. Tuto knihovnu budou využívat všechny části systému, které budou chtít pracovat s evidovanými časovými řadami. Implementací tohoto návrhu bude splněn funkční požadavek F3.

Návrh identifikátoru záznamů

Protože budou ukládány časové řady, musí být součástí identifikátoru (klíče) časová značka s časem, ve kterém byla daná hodnota naměřena. Časová značka bude reprezentována v UNIX timestamp formátu v milisekundách (počet milisekund od počátku epochy). Dále bude nutné mít v klíči unikátní identifikátor časové řady (v dalším textu označovaný také jako UID), do které patří. Struktura společně s příkladem jednoho takového identifikátoru je uvedena v kódu 3.4. Podle tohoto složeného identifikátoru lze určit, které záznamy spolu souvisí, patří do stejné časové řady. Tyto dvě informace jsou postačují pro jednoznačné určení záznamu a zpětnou replikaci celé časové řady.

```
struktura        <UID časové řady><časová značka>
rowkey           1231676488175123
123              UID časové řady
1676488175123    časová značka
```

Kód 3.4. Návrh struktury jednoduchého unikátního identifikátoru záznamu s ukázkou

Ačkoliv by byla popsaná struktura klíče dostatečná, je vhodné myslet na možnost zvýšení granularity, se kterou lze časové řady strukturovat. Jako motivace pro rozšíření návrhu může být, že bude možné uložit stejnou veličinu (stejné UID) např. pro několik různých entit (datových objektů). V takovém případě by musela být rovnou součástí UID informace o entitě, ke které sémanticky daná časová řada patří. Využívání takového rozdělení lze podpořit obecnějším návrhem identifikátoru záznamu v časové řadě (klíče, resp. rowkey). Postačující bude rozšířit dosavadní návrh. Na konec klíče bude možné přidat libovolné množství značek, uspořádaných dvojic `<klíč, hodnota>`. K jednoznačnému identifikování časové řady tak bude potřeba znát UID a n uspořádaných dvojic, značek. Pro jednotlivé záznamy časové řady se stejně jako v původním návrhu přidá časová značka. Značky, uspořádané dvojice, mohou být použity pro ukládání cizích klíčů do jiných databází, rozdělování časových řad do skupin atp. Příklad jednoho takto navrženého identifikátoru záznamu společně se strukturou je uveden v kódu 3.5.

```
struktura       <UID časové řady><časová značka><značka 1><značka 2>...
rowkey          1231676488175123tag1val1tag2val2
123             UID časové řady
1676488175123   časová značka
tag1:val1       značka 1 <klíč:hodnota>
tag2:val2       značka 2 <klíč:hodnota>
```

Kód 3.5. Návrh strukunikátnítury rozšířeného unikátního identifikátoru záznamu s ukázkou

Dalším faktorem, který bude nutné zohlednit v návrhu, je pořadí částí v unikátním identifikátoru. Zde je opět nutné vzít v potaz zvolenou technologii. V Apache HBase jsou záznamy řazeny lexikograficky podle klíče [72]. Vyhledávání záznamů v tabulce lze omezit pouze na určitou podmnožinu řádků definovanou počátečním a koncovým klíčem. Další možností je poté využít *RowFilter* s vybraným komparátorem (např. *RegexStringComparator*). Takto lze vybrat podmnožinu řádků, které nejsou bezprostředně po sobě jdoucí, ale jsou rozesety po celé délce tabulky a splňují definované predikáty (např. regulární výraz) [72]. V klientské knihovně existuje několik implementací takového filtrování (resp. komparátorů). Implementačně jednodušší je první varianta (určení počátečního a koncového klíče), neboť stačí nastavit pouze dva příznaky a není třeba sestavovat celý mechanismus pro filtrování řádků. Oproti tomu druhá varianta poskytuje možnost simulování sekundárního indexu. Samozřejmě nejefektivnějšího vyhledávání lze dosáhnout kombinací obou přístupů.

Jak bylo zmíněno v předchozím odstavci, na unikátní identifikátor má vliv na pořadí ukládání záznamů. Pokud by všechny záznamy jedné časové řady byly v tabulce bezprostředně za sebou, stačilo by pouze určit počáteční a koncový identifikátor záznamu, a tím by bylo veškeré dotazování hotové. Přímočarým řešením by tak bylo přizpůsobit tomuto požadavku návrh klíče a usnadnila by se následná realizace dotazování. Ovšem už zde je také vhodné myslet na to, aby návrh nebránil rozšiřitelnosti systému. Zvolená technologie Apache HBase, pokud je provozována v cluster módu, rozděluje všechny záznamy každé své tabulky mezi jednotlivé region servery právě podle klíče [72]. To by v budoucnu znamenalo, že všechny záznamy (případně většina) jedné časové řady by byly uloženy na jednom region serveru. Tím pádem by nedocházelo ke správné paralelizaci dotazů, a distribuování zátěže, a servery by byly přetěžovány. Není tedy zcela nutné udržovat záznamy jedné časové řady u sebe. Dokonce, podle zmíněných vlastností HBase, to může být s dalším rozvojem systému nežádoucí. V případě clusteru je nejlepším stavem mít jednu časovou řadu rovnoměrně rozdělenou mezi všechny servery. Toto rozdělení lze vyřešit pomocí vhodného hash vypočteného z klíče a následně umístěného jako jeho prefix. Nicméně to je věcí dalšího rozšiřování systému, a nikoliv prototypu.

Pořadí jednotlivých částí klíče tedy není nutné dále navrhovat tak, aby zachovávalo u sebe záznamy jedné časové řady. Lepší je vytvořit návrh umožňující využití dostupných filtrovacích mechanismů. Ze stejného důvodu zde přichází na na řadu vyřešit ještě jeden problém, který byl doposud přehlížen, a tím je délka UID a značek. Databáze HBase umožňuje dotazování na klíče uložených záznamů několika způsoby. Ten nejsnazší byl již několikrát zmíněn a je jím prosté určení počátečního a koncového klíče. Dalším způsobem umožňujícím vybrat záznamy z tabulky, tentokrát už bez nutnosti aby se nacházely bezprostředně za sebou, je dotazování pomocí regulárních výrazů. Vráceny jsou poté všechny záznamy, jejichž klíč splňuje zadaný předpis. Pro toto dotazování je ale vhodné (či dokonce nutné) znát délky jednotlivých částí klíče. Implementace filtrů je poté snazší. Kromě vyhledávání pomocí regulárního výrazu existuje ještě další filtr,

který je vhodný pro vyhledávání v takto strukturovaných klíčích záznamů. Jedná se o fuzzy filtr. I pro jeho použití je nutné znát délku jednotlivých částí klíče. Pro vyhledávání je totiž potřeba definovat bytovou masku. Ta označuje, které byty jsou ve hledaných klíčích pevně dané a které variabilní. K bytové masce je potřeba poskytnout ještě skutečnou hodnotu klíče, podle které se určí, jaké hodnoty mají nabývat byty označená maskou za neměnné. Ostatní části klíče, volné byty, mohou mít libovolnou hodnotu.

Délky částí je nutné volit tak, aby poskytovaly dostatečné množství kombinací bitů pro uložení potřebného množství různých hodnot. Na druhou stranu je nutné myslet na velikost klíče a neukládat zbytečně mnoho informací, které se nikdy nevyužijí. Vhodná délka pro tento systém může být stejná jak pro UID, tak pro klíč a pro hodnotu značky, a to 3 byty. Pro značky je tak rezervováno 2^{24} různých kombinací pro klíče, stejně tak může být různých hodnot (hodnoty budou sdílené mezi všemi klíči). Protože i pro UID je zvolena stejná délka 3 bytů, bude jich možné uložit 2^{24} různých. Časová značka je implicitně omezena na 8 bytů, neboť se jedná o počet milisekund od počátku epochy, jak bylo zmíněno v úvodu navrhování. Celková délka klíče se tak odvíjí od počtu značek, minimálně však 12 bytů.

Konečný návrh klíče jednoho záznamu časové řady je takový, aby části, u kterých je známá pevná délka, byly na začátku. Za tímto prefixem pak bude následovat část s variabilním počtem položek – značky. Základní vlastnosti, které z tohoto návrhu plynou, jsou – záznamy jedné časové řady nebudou bezprostředně za sebou, záznamy každé časové řady budou seřazeny podle časové značky (HBase řadí záznamy lexikograficky a pokud se budou řádky lišit pouze v časové značce, které je striktně neklesající, tedy pokud nebude využit hash). Schéma navrženého klíče je shodné s uvedeným v ukázce kódu 3.5). Pro větší přehlednost je shrnuta celá struktura v kódu 3.6.

```
struktura      <UID časové řady><časová značka><značka 1><značka 2>...
délka v bytech         3               8          3+3       3+3
```

Kód 3.6. Návrh struktury rozšířeného unikátního identifikátoru záznamu s ukázkou

Návrh struktury ukládané hodnoty

Druhou částí návrhu ukládání dat do časových řad je způsob uložení samotné hodnoty. Ta samozřejmě nebude součástí klíče, ale ten na ni bude odkazovat. Struktura, na kterou odkazuje klíč záznamu je opět key-value. Jedná se tedy o druhou key-value dimenzi. Klíčem je zde tzv. *column-family* a hodnotou je poté další, již třetí a poslední, key-value záznam, kde klíčem je tzv. *qualifier* a hodnotou libovolné neinterpretované byty. Až zde bude uložena samotná hodnota. Je tedy třeba navrhnout strukturu těchto dvou zbývajících dimenzí a zajistit správné ukládání a následnou zpětnou interpretaci bytů.

Column-families budou pro ukládání hodnot potřeba dvě. Hodnota je ukládána jako neinterpretované byty, a proto je potřeba kromě ní samotné uložit další metadata, která pomohou při zpětné interpretaci, tedy při čtení dat. Možností by také bylo udržovat metainformace v jiné tabulce, případně jinde v aplikaci, a v případě potřeby se na ně doptávat. Tento přístup má dvě zásadní nevýhody, kvůli kterým nebude použit. Tou první je nutnost vyvíjet a spravovat další aplikaci, případně zajišťovat integritu v další tabulce. Kromě integrity by bylo nutné dále řešit zpětnou kompatibilitu, tedy i verzování, aby bylo možné číst záznamy, které byly zapsány s jinou než aktuální verzí schématu (metadat). Druhou nevýhodou je složitější dotazování a zvýšená režie. Pro

přečtení uložených data by bylo zapotřebí provést další dotaz do jiné aplikace, případně jiné tabulky.

První *column-family* s název `value` bude sloužit pro uložení samotné hodnoty. Druhá *column-family* s názvem `meta` bude sloužit pro ukládání případných metadat, jako je např. typ uložené informace apod. Množina *qualifiers* pro *column-family* `meta` bude obsahovat dvě položky – `value` a `type`. První bude obsahovat skutečně naměřenou hodnotu a druhá název typu (případně jednotky) uložené hodnoty. Jelikož mají být názvy *column-families* nejkratší možné, budou v implementaci použita pouze první písmena názvu. *Qualifiers* pro ukládání hodnoty uložení metadat zde nebudou nijak specifikovány, jde pouze o prostor, který může být využit pro ukládání implementačně závislých informací. Schematicky je popsaná struktura zobrazena v tabulce 3.2.

column family	value	meta
qualifiers	value type	*rezervováno pro implementaci*

Tabulka 3.2. Schéma HBase tabulky pro ukládání hodnot časových řad

3.5 Mikroslužby systému

Systém se bude skládat z několika mikroslužeb – komponent. Každá mikroslužba bude zodpovědná za poskytování jedné ucelené funkcionality. Tato funkcionalita (mikroslužba), případně kooperace několika mikroslužeb, bude naplňovat požadavky na systém definované v sekci 2.6. V této sekci jsou jednotlivé mikroslužby popsány z hlediska architektury. Je popsán jejich hlavní účel, vzájemná komunikace a logické vazby. Zároveň jsou zmíněny technologie, které bude vhodné využít při jejich implementaci. Všechny tyto informace budou následně vstupem pro samotnou realizaci.

V úvodu kapitoly bylo řečeno, že je vhodné spojit dva funkční požadavky do jednoho logického celku (požadavek na evidenci datových objektů (F1) a vztahů mezi nimi (F2)). Funkcionalita, kterou tyto požadavky definují, bude obsluhována jednou mikroslužbou (*Verta*). Při jejím návrhu bude využit datový model navržený v sekci 3.4.1.

Společně s tímto modelem byl v téže sekci také navrhnut způsob, jak ukládat datové objekty a jejich vztahy a poskytovat je pro další zpracování. Odtud vyvstala potřeba evidovat schémata pro určité typy ukládaných objektů a vazeb kvůli validaci a kontrole integritních omezení. Poskytovaní a evidence těchto informací by měla být zodpovědností separátní mikroslužby (*Query Resolver*), neboť se jedná o ucelenou funkčnost.

Další ucelenou funkcionalitou, kterou lze naplnit jednou službou, je evidence časových řad (F3). Pro navrženou databázi časových řad postavenou nad datovým enginem Apache HBase nebude dále navrhovaná žádná mikroslužba. Tato funkcionalita bude následně implementována jako knihovna (wrapper) nad HBase API.

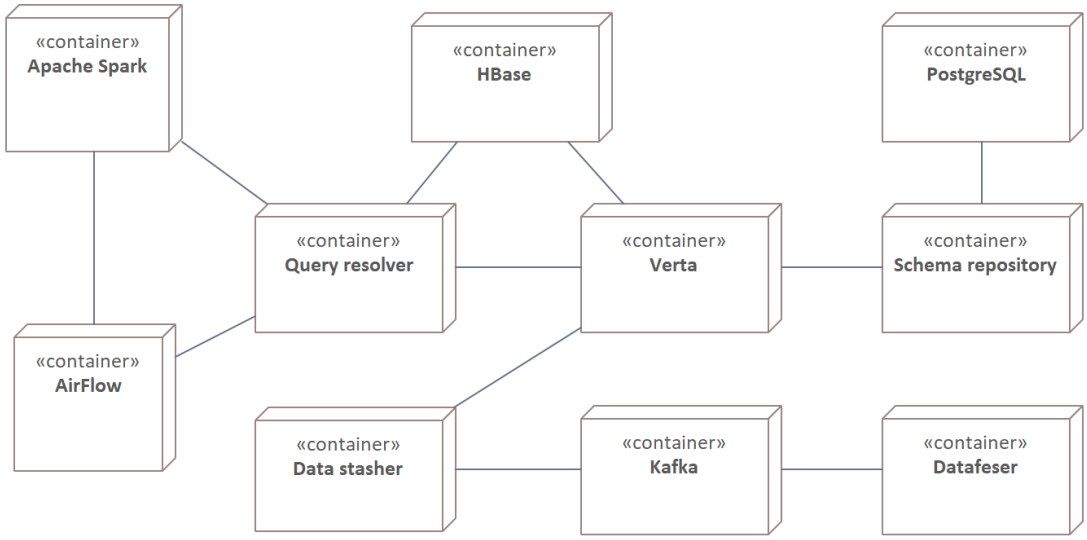
Ukládání dat do databáze časových řad bude rozděleno na dvě části, přičemž každá část bude navržena jako samostatná mikroslužba. Dohromady budou naplňovat funkční požadavek na napojení systému na externí datové zdroje (F5). Tento návrh byl již nastíněn v úvodu kapitoly. Jedna mikroslužba (*Data Feser*) bude sloužit pro samotné odečítání dat z externích zařízení a systémů. Tato data bude poskytovat druhé mikroslužbě (*Data Stasher*), která je bude zpracovávat a ukládat do databáze časových řad, odkud budou dostupná pro další zpracování.

Poslední mikroslužbou, kterou bude nutné navrhnout, je query gateway (*Query Resolver*), která bude umožňovat přístup ke všem datům evidovaným v systému. Tato

mikroslužba bude sloužit pro unifikaci dotazování na všechna data, která budou systémem evidována (tedy jak na evidenční data objektů a vazeb, tak na data časových řad). Zároveň se bude jednat o službu, na kterou budou přistupovat aplikace třetích stran. Bude se tedy jednat o vstupní bod systému.

Ačkoliv uživatelské rozhraní lze také chápat jako jednu z mikroslužeb, nebude v této sekci uvedeno. Bude mu však věnována pozornost ve speciální sekci, ve které budou diskutovány všechny zásadní body jeho návrhu.

Všechny navržené mikroslužby systému včetně jejich logických propojení jsou schematicky zobrazeny na obrázku 3.7. Součástí schématu jsou také závislé služby. Zachycené vztahy reprezentují, že mezi propojenými mikroslužbami (resp. mikroslužbou a závislou službou) bude nutné komunikovat, předávat si data.



Obrázek 3.7. Schéma logického propojení mikroslužeb systému a závislých služeb

3.5.1 Schema Repository

V sekci 3.4.1, zabývající se způsobem ukládání vztahů mezi objekty, je zmíněna nutnost kontrolovat (a definovat) integritní omezení. Aby byl systém dynamický a šlo tato omezení přidávat bez nutnosti zásahu do kódu jednotlivých mikroslužeb, případně změny struktury databáze, je nutné definice všech těchto omezení udržovat externě. Pro tento účel bude sloužit mikroslužba *Schema Repository*, která bude spravovat definice všech objektů a vazeb. Delegováním této funkcionality na samostatnou mikroslužbu lze odstranit přímé závislosti jednotlivých mikroslužeb, případně duplikaci definic mezi mikroslužbami. Takto bude zajištěno, že všechny mikroslužby, které budou potřebovat schémata objektů a vazeb, budou mít k dispozici stejné definice. Dále se tímto způsobem zvýší úroveň abstrakce a nebude nutné vždy ukládat definice do stejného perzistentního uložiště, důležité bude pouze splnění definovaného kontraktu – interface.

Definice objektů a vztahů spravovaných mikroslužbou *Schema Repository* budou poskytovány ve formátu deklarativního jazyka JsonSchema [73]. Detaily tohoto jazyka byly diskutovány v sekci 3.2.8. Mezi jeho hlavní přednosti, které budou využity touto mikroslužbou, patří, že je definován v textovém formátu. Schémata lze ukládat jako text, tedy omezení na datové uložiště je minimální. Může se jednat jak o objektovou databázi, tak např. i pouze o textové soubory uložené na souborovém systému. Nicméně

v implementaci bude vhodné zvolit určitý typ databáze, kterou bude využívat i jiná mikroslužba, aby nebylo nutné spravovat mnoho rozdílných technologií. Výhodami JsonSchema, které budou využity na straně klientů této mikroslužby, jsou možnosti validace JSON dokumentu oproti schématu a možnost generovat ze schémat POJOs (Plain Old Java Objects).

Mikroslužba bude poskytovat aplikační rozhraní komunikující protokolem HTTP. Rozhraní bude dodržovat architekturu REST. Bude poskytovat možnost pro přidání, získaní, smazání a úpravu schémat. Aplikační rozhraní bude také umožňovat dotaz na všechna uložená schémata. Schémata budou verzována, proto úprava schématu bude provedena jako vložení nové verze. Zároveň bude možné ostatní zmíněné operace provádět nad libovolnou verzí libovolného schématu. Tímto návrhem bude zaručena kompatibilita schémat. Klient tak může využívat starší verzi určitého schématu. Není však nijak validována zpětná kompatibilita nově vytvářených verzí schématu. Tato zodpovědnost je přenechána na klientech. Obecně však lze předpokládat, že schémata nebudou zpětně kompatibilní.

Dotazování na všechna schémata budou využívat všechny mikroslužby, které s nimi budou pracovat. Načtení proběhne při startu aplikace a pak periodicky, v daných časových úsecích, aby došlo k propagaci případné změny. Takto se zajistí, že mikroslužba *Schema Repository* nebude přetížena dotazy na jednotlivá schémata, ale zároveň se nestane úzkým hrdlem celého systému. Při jejím výpadku lze běžící systém po omezenou dobu provozovat.

Tato mikroslužba se bude podílet na naplnění funkčních požadavků F1 a F2. Pro evidenci objektů a vazeb bude poskytovat předpisy jednotlivých datových typů, na základě kterých budou prováděny validace.

3.5.2 Verta

Pro ukládání vztahů komponent FVE, datových objektů a vztahů mezi nimi bude sloužit mikroslužba s názvem *Verta*. S ní je úzce spjat datový model navržený v sekci 3.4.1 a mikroslužba *Schema Repository* (3.5.1). Mikroslužba bude využívat jako perzistentní uložiště grafovou databázi JanusGraph (popsaná v sekci 3.2.4).

Úlohou této mikroslužby v systému bude poskytovat aktuální a historické informace o komponentách FVE, jiných evidovaných objektech a jejich vztazích. Data bude poskytovat přes aplikační rozhraní komunikující protokolem HTTP. Přes něj bude také možné provádět vkládání, úpravu, mazání a invalidaci. Dále bude mikroslužba validovat všechny ukládané objekty a vztahy podle JSON schemat definovaných ve *Schema Repository*, aby byl dodržen navržený datový model podle sekce 3.4.1. Každé JSON schéma bude definovat objekt nebo hranu, kterou lze do *Verta* uložit. Tím bude zajištěno, že v databázi budou pouze objekty a hrany předem známého typu a vlastností, tedy bude zajištěna integrita dat na této nejnižší možné úrovni. Validačním kritériem tak bude moct být libovolný konstrukt, který je podporovaný v JSON schema. U vztahů bude navíc možné definovat počáteční a koncový typ objektu. Aby nebyla přímá závislost mezi touto mikroslužbou a *Schema Repository*, *Verta* bude při startu načítat všechna schémata do paměti a poté bude pracovat s touto kopií (snapshot). Pro zajištěná konzistence si bude tuto kopii aktualizovat. Tímto se zajistí tzv. eventual konzistence schémat – schémata nejsou konzistentní v každém okamžiku běhu aplikace, ale v některých časových bodech ano.

Dotazování na uložená data bude možné podle všech atributů všech objektů. Dále bude možné dotazování podle vazeb mezi nimi. Všechny typy podmínek půjde spojovat logickými spojkami `or`, `and`. Dále bude možné aplikovat unární operaci `not` pro negaci

výroku. Přes tento dotazovací jazyk budou k uloženým datům přistupovat všechny ostatní komponenty. Zároveň přes něj budou získávat data automatické úlohy.

Implementací mikroslužby s dotazovacím jazykem v navrhovaném rozsahu budou splněny následující funkční požadavky – F1 a F2. Zároveň bude svým dotazovacím jazykem přispívat k naplněné funkčního požadavku F4. Pro jeho plné splnění však bude potřeba implementace několika dalších funkcionalit obsažených v jiných mikroslužbách.

3.5.3 Data Feser

Název mikroslužby je odvozen od její navrhované funkcionality. Mikroslužba bude získávat data z chytrých zařízení FVE a jiných systémů (data fetching) a zároveň bude poskytovat služby pro odesílání dat do těchto zařízení a systému (data sending). Odtud název *Data Feser*.

Data z jednotlivých komponent FVE je nutné aktivně odečítat. Proto musí být součástí systému mikroslužba, která bude tento odečet provádět. Jelikož každý střídač, senzor nebo jiné zařízení poskytuje jiné rozhraní pro přístup k jeho datům, bude nutné zajistit, aby tato mikroslužba byla snadno rozšiřitelná o další způsoby komunikace. Možností, jak toto udělat, je připravit aplikaci, pro kterou bude možné vytvářet zásuvné moduly (plugins). Druhou možností je vytvořit pouze dostatečně abstraktní rozhraní, které bude každý způsob komunikace dodržovat. Oba způsoby mají své výhody a nevýhody. První způsob je vhodný pro produkční prostřední, protože je sdíleno pouze rozhraní a není nutné sdílet celý kód. Nicméně je složitější na implementaci. Druhý způsob vyžaduje sdílení celého kódu, zároveň po každé změně je nutné sestavit aplikaci znovu. Nicméně je jednoduší na realizaci, a jelikož je navrhován prototyp systému, který nebude sdílen třetím stranám, bude vhodné v implementaci využít právě jej.

Všechny odečtené hodnoty se nebudou ukládat rovnou do databáze, ale před uložením se data obohatí o další informace. Obohacování dat však bude obstarávat mikroslužba *Data Stasher* popsaná v sekci 3.5.4. V této sekci je také objasněn celý proces obohacování.

Aby bylo odečítání co nejrychlejší a oddělila se přímá závislost dvou mikroslužeb, všechna data budou ukládána do Kafky (topic `metering`). Uložení dat tak nebude provádět dlouhé blokující volání. Zároveň nedostupnost mikroslužby provádějící obohacování dat nezpůsobí zablokování možnosti nahrávat nová data ze senzorů a ostatních komponent do systému.

Druhá funkcionalita, kterou bude mikroslužba poskytovat, je inverzní k první. Bude umožňovat odesílání dat do jiných systému, případně zařízení. Z dosavadního textu vyplynulo, že jedna taková integrace již bude existovat pro Apache Kafka. Nicméně pro větší univerzálnost bude *Data Feser* poskytovat, stejně jako pro odečítání, rozšiřitelné rozhraní, s jehož pomocí bude možné implementovat komunikaci pomocí libovolného protokolu. Tato funkcionalita může být využita např. při nahrávání nového firmwaru do zařízení atp.

Navržené funkcionality napojení na externí služby, které bude *Data Feser* poskytovat, naplňují funkční požadavek F5.

3.5.4 Data Stasher

V sekci 3.5.3 bylo zmíněno, že obohacování dat bude provádět separátní mikroslužba. Tou je právě *Data Stasher*. Naměřená data z komponent FVE, která byla uložena do Kafka topic, budou touto mikroslužbou obohacena a dále zpracována. K naměřeným datům budou přidány další informace a zároveň se pro každou hodnotu provede validace, zda ukládaná hodnota obsahuje všechny potřebné informace v požadovaném formátu.

Definice struktury bude prováděna pro každý topic pomocí AVRO schématu. Schéma definuje strukturu zpráv, které jsou ukládány v rámci jednoho topicu.

Základní tok naměřené hodnoty (případně jiné ukládané hodnoty do systému) mikroslužbou *Data Stasher* bude následující. Předpokladem je, že některá mikroslužba naplní Kafka topic surovými daty, která získala z externích zdrojů. Z tohoto, vstupního, Kafka topic, nazývaného např. `metering`, ve kterém jsou surová data, se načte záznam pro další zpracování. Tento záznam se obohatí o potřebné informace a případně zvaliduje. Pokud je obohacený záznam validní, uloží se do dalšího topic, např. s názvem `waiting_for_stash`. Odtud pak budou záznamy ukládány přímo do *databáze časových řad*.

Samotné obohacování je v podstatě denormalizace, případně přidávání cizích klíčů do jiných datových zdrojů. U dat časových řad reprezentujících měření může být obohacování např. přidání jednotky podle schématu. Co však bude nutné v implementaci provádět, je mapování názvů časových řad na jejich UID (tří bytový unikátní identifikátor popsaný v sekci 3.4.2). S tímto interním identifikátorem bude pracovat pouze databáze, pro zbytek systému bude možné mít pro jednotlivé časové řady libovolně dlouhý, lidsky čitelný, identifikátor.

Proces obohacování může přidávat i komplexnější data, jako jsou informace, se kterou verzí systému, případně verzí validačního schématu, byla data zpracována. Dále lze tato data doplnit některou informací, která je poskytována mikroslužbou *Verta*, jako je typ objektu, ke kterému naměřená data patří apod. Všechno toto duplikování informací slouží pro zjednodušení a zrychlení dotazování.

Do procesu obohacování mohou vstoupit také další transformace. Jedním zástupcem může být kontrola duplicity dat. Tato kontrola by probíhala stejným způsobem jako obohacování. Vstup pro obohacování dat by tak nebyl Kafka topic se surovými informacemi, ale už s předzpracovanými daty, která by obsahovala informaci, zda se jedná o duplicitu, či nikoliv.

Koncept streamového zpracování dat, kdy jsou data po každé jedné transformaci uložena do dalšího topicu, odkud jsou dále zpracována jinou funkcí, je snadno rozšiřitelný. Lze přidat libovolnou novou funkcionalitu. Zároveň lze jednoduše definovat, kolik instancí které transformace má existovat, a tím zajistit škálování každé operace zvlášť (N7). Vhodnou technologií pro implementaci toho zpracování jsou Kafka Streams popsané v sekci 3.2.6.

Popsaný návrh mikroslužby *Data Stasher* tak naplňuje část funkčního požadavku týkajícího se ukládání dat do evidovaných časových řad. Konkrétně se jedná o F3. Jelikož tato mikroslužba bude ukládat data do systému tak, že je bude možné snadno prohledávat, podílí se tento návrh také na splnění funkčního požadavku F4.

3.5.5 Query Resolver

Navrhovaný systém obsahuje několik zdrojů informací, na jejichž data se bude možné dotazovat. V analýze byla tato data rozdělena na dvě kategorie – časové řady a datové objekty s vazbami. Každá kategorie je přístupná pomocí jiné mikroslužby systému. Obě budou konzumovány dalšími částmi systému, zejména uživatelským rozhraním.

Tato data budou přístupná přes jedno aplikační rozhraní v unifikovaném formátu, aby bylo používání systému snazší pro uživatele (klienty). Zároveň se tímto přidá vrstva do zabezpečení systému, neboť budou vystaveny pouze některé funkcionality, které jsou pro uživatele nezbytně nutné. Další výhodou je možnost implementovat několik různých API pro různé konzumenty dat ze systému. Jako příklad může být rozdílné API pro mobilní aplikaci, pro webovou aplikaci či pro aplikace třetích stran. Pro prototyp však

bude existovat pouze jedna implementace, kterou bude používat uživatelské rozhraní a zároveň bude poskytnuto pro možnost integrace (např. pro automatické úlohy).

Vhodná technologie pro implementaci API, které umožňuje dotazování do několika datových zdrojů, je GraphQL. Jde o obecný dotazovací jazyk, kterému je definováno datové schéma – je tedy typovaný. Dotazy jsou odesílány ve formátu JSON a musejí splňovat definované schéma. Zároveň přímo v dotazu jsou specifikovány informace, které klient požaduje. Odpověď tak neobsahuje data, která nejsou pro klienta relevantní. Zároveň je možné spojit několik dotazů dohromady. Tím je možné snížit datový tok po síti. V klasickém REST API by byl dotaz na dva typy entit rozdělen na více dotazů, které by směřovaly na různé endpointy. Z těchto dotazů by pak byl sestaven výsledný dataset.

V GraphQL je možné pro každou položku každého datového typu specifikovat její data resolver. Jedná se o metodu, která bude data pro zvolenou položku získávat. Zároveň má každý data resolver dostupný kontext, takže zná parametry. Pokud jde o zanořenou položku, tak zná výsledek rodičovského dotazu. Například pokud by bylo potřeba získat naměřená data pro určitý datový objekt (komponentu FVE), data resolver obstarávající data o časových řadách (naměřených datech) by měl v kontextu od rodiče (datového objektu), ke které komponentě má naměřená data hledat.

Druhou signifikantní výhodou této technologie je jednoduchost implementace na straně backendu. Stačí pouze implementovat zmíněná data resolvers a není nutné starat se o správný návrh enpointů, struktur návratových objektů atp.

Tento návrh mikroslužby *Query Resolver* umožňuje splnění požadavku F4. Požadavek může být splněn, zejména protože data budou do systému ukládána v takovém formátu a takovým způsobem, že bude možné implementovat potřebné resolvers a spojovat data z více zdrojů – z komponenty *Verta* a *databáze časových řad* – požadovaným způsobem.

3.6 Uživatelské rozhraní

Navržené mikroslužby popsané v předchozích sekcích se nijak neomezovaly na některé konkrétní případy užití, jako je např. sdílení elektrické energie, ale přispívaly k vytvoření obecných procesů zachycených ve funkčních požadavcích. V případě uživatelského rozhraní je tomu jinak. Jelikož toto rozhraní má sloužit jako výstupní bod pro zobrazení dat uživateli, který nemusí znát celou logiku, která za nimi stojí, je nutné přizpůsobit toto rozhraní konkrétním případům užití. V opačném případě by bylo nutné řádné zaškolení do dané domény, neboť data by se zobrazovala obecně a jejich kontext by nebyl ve všech případech zřejmý. Nevýhodou toho však je, že při změně využívání systému bude muset být uživatelské rozhraní přepracováno.

Již v analýze, v sekci 2.5.2, byl uveden příklad sdílení elektrické energie v bytovém domě. Na toto sdílení bude také navazovat návrh uživatelského rozhraní. Jelikož celý navrhovaný systém je prototypem, bude uživatelské rozhraní poskytovat základní pohledy na data uložená v systému. Bude zobrazovat vztahy mezi uloženými datovými objekty, např. komponentami FVE, ale nebude se omezovat jen na ně. Mezi těmito vazbami bude možné interaktivně procházet, a tím docílit snadného pochopení topologie uložených dat. Dále bude uživatelské rozhraní umožňovat zobrazení jednotlivých časových řad v zadaném časovém rozmezí. Poslední informací, která půjde v uživatelském rozhraní zobrazit je detail sdílení elektrické energie, ve kterém bude možné pro jednotlivá přidružená odběrná místa a hlavní odběrné místo vidět, kolik energie které místo spotřebovalo a kolik energie mu bylo podle alokačního klíče přiděleno.

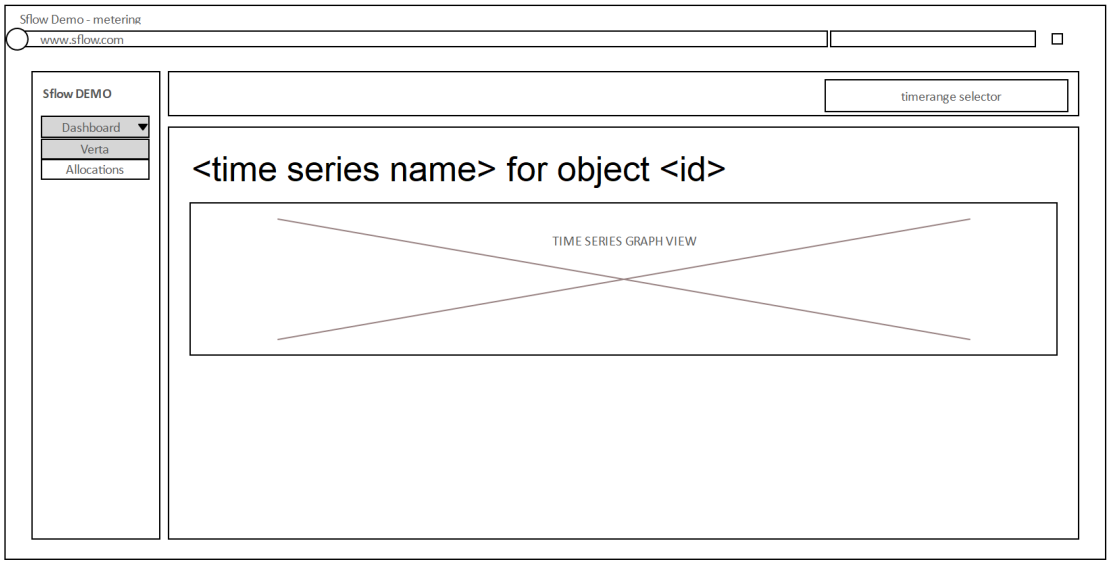
3.6.1 Wireframes

Návrh uživatelského rozhraní je zachycen pomocí wireframes. Návrhu každé obrazovky uživatelského rozhraní je věnována jedna sekce. Ta kromě samotného modelu obsahuje i jeho popis a případy užití, které naplňuje.

Společnými prvky pro všechny wireframes je levé (hlavní) menu pro navigaci mezi jednotlivými obrazovkami. Dále záhlaví obsahující breadcrumb menu pro lepší orientaci uživatele, na které obrazovce se právě nachází a jaká k ní vedla cesta. Posledním společným prvek umístěným v záhlaví, konkrétně v pravé části, je menu pro výběr časového rozsahu. Pomocí něho bude možné vybrat rozsah, se kterým bude systém zpracovávat uživatelské požadavky. Tento prvek se podílí na naplnění případů užití, které požadují pracovat s daty pouze v zadaném časovém rozsahu. Tento požadavek má většina z nich, vizte sekci 2.7.2.

Přehled časové řady

Systém má mít možnost zobrazení dat z jednotlivých časových řad. Tato funkcionalita je vyžadována v případu užití UC3, částečně také v UC4 pro zobrazení detailu sdílení elektrické energie. Návrh uživatelského rozhraní pro takové použití je uveden na obrázku 3.8.



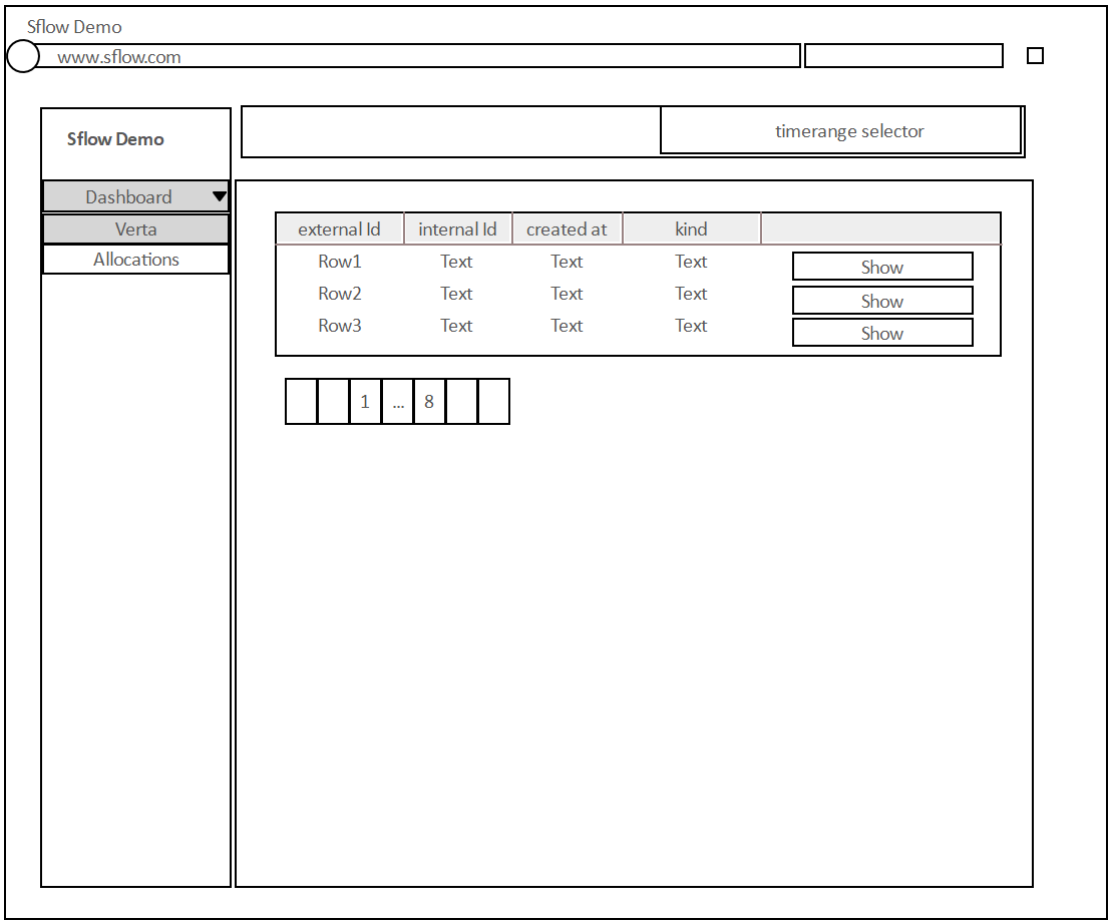
Obrázek 3.8. Wireframe uživatelského rozhraní zobrazení detailu časové řady

Wireframe zobrazení časové řady obsahuje pouze jednu komponentu navíc oproti společným. Jedná se o grafické znázornění časové řady spojnicovým grafem. Tento graf nemusí nutně obsahovat pouze jednu časovou řadu, ale může jich zobrazovat několik přes sebe. Takové zobrazení umožňuje jednoduché vizuální porovnání hodnot různých časových řad v různých bodech. Zobrazená data musejí respektovat časový rozsah vybraný pomocí společné komponenty v pravém horním rohu.

Přehled evidovaných datových objektů

Návrh obrazovky uživatelského rozhraní, která bude sloužit pro zobrazení všech informací o evidovaných datových objektech, lze vidět na obrázku 3.9. Zobrazení tohoto přehledu je přímo uvedeno jako jeden z případů užití, konkrétně UC1.

Wireframe obsahuje tabulku, ve které budou zobrazeny všechny evidované objekty. Jelikož se jedná o přehledovou obrazovku, jsou v tabulce pro každý objekt uvedeny



Obrázek 3.9. Wireframe uživatelského rozhraní pro zobrazení datových objektů uložených v komponentě `Verta`

pouze základní informace, jako je interní a externí identifikátor, druh objektu a časové značky jejich vytvoření a smazání. Dále bude možné zobrazit všechny vazby vybraného objektu kliknutím na řádek tabulky, ve které je uveden. Přehled se zobrazí také jako tabulka přímo pod řádkem, na který bylo kliknuto. Tato tabulka bude obsahovat objekty, které jsou s vybraným objektem spojeny vazbou, k nim bude zobrazovat stejné informace jako přehledová tabulka všech objektů. Navíc bude uveden typ použité vazby.

Dále bude možné z každého záznamu provést proklik na jeho detail (kliknutím na tlačítko *Show*) – UC2. Návrh obrazovky, na kterou budou směrovat tyto odkazy pro zobrazení detailu objektu, je zachycen ve wireframe na obrázku 3.10.

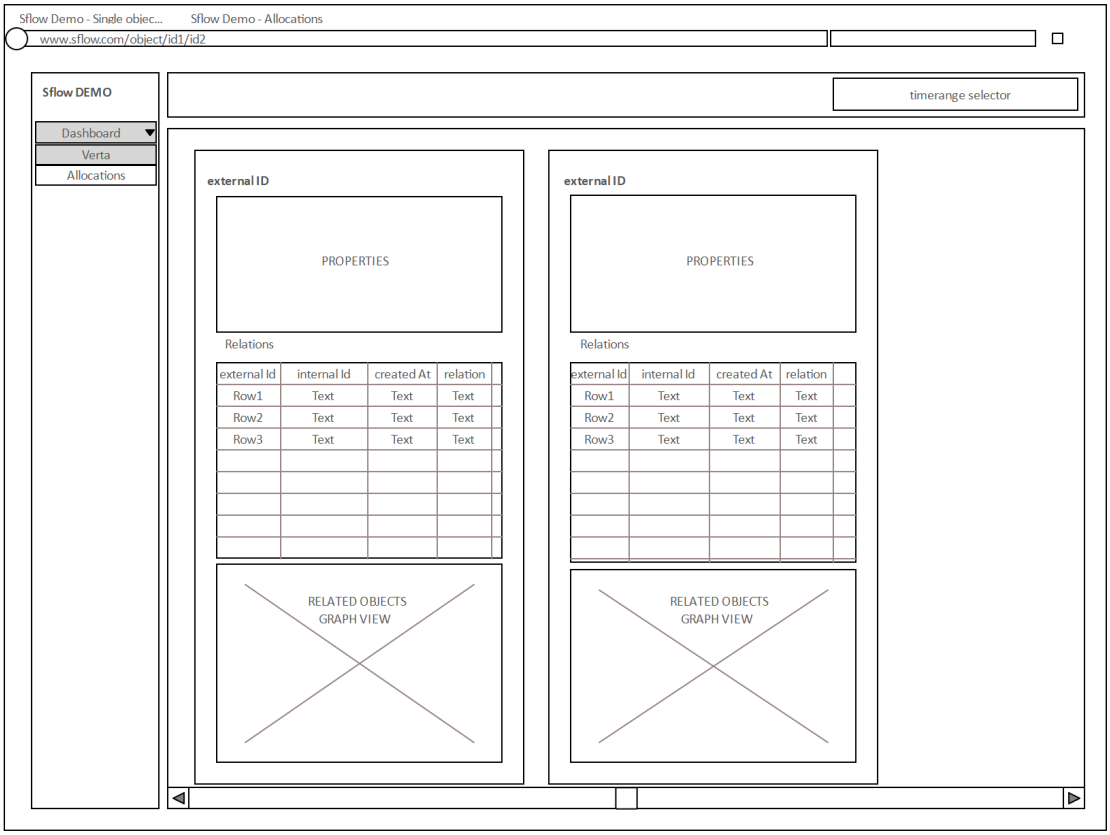
Detail datových objektů

Obrazovka pro zobrazení detailu datového objektu bude přístupná přes odkazy v přehledu datových objektů (wireframe na obrázku 3.9). Na této obrazovce již budou uvedeny všechny vlastnosti vybraného objektu. Její wireframe je možné vidět na obrázku 3.10 a pokrývá případ užití UC2.

V návrhu je možné vidět detail dvou datových objektů. Každý objekt je na separátní kartě. Tento návrh podporuje snadné porovnání jejich vlastností a vazeb. Na kartě každého objektu je v hlavičce uveden jeho externí identifikátor (neexistuje-li, bude uveden interní identifikátor). Pod hlavičkou je seznam všech vlastností daného objektu. Součástí těchto vlastností jsou i informace, které časové řady jsou s daným datovým

objektem spjaty. Přes tento přehled lze přejít na obrazovku s přehledem vybrané časové řady, wireframe na obrázku 3.8.

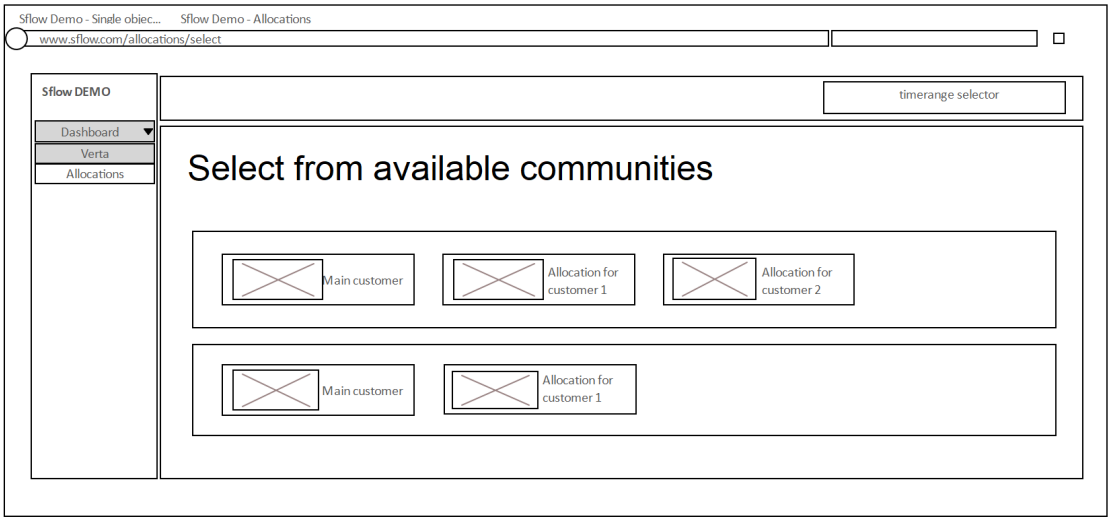
Pod vlastnostmi je v návrhu obrazovky seznam všech vazeb zobrazovaného objektu na ostatní evidované objekty. Stejná informace je také znázorněna graficky ve spodní části karty. Tyto dvě komponenty, tabulka a graf zobrazující vazby objektů, budou poskytovat možnost prokliku na navázané objekty. Právě tímto způsobem bude možné zobrazit novou kartu s objektem vpravo od stávající, přesně tak, jak je zachycuje wireframe na obrázku 3.10. Z každé karty bude možné zobrazit vždy jen jeden navazující objekt. Nicméně počet karet nebude nijak omezen a bude takto možné zobrazit celou cestu v grafu závislostí. Případné přetečení mimo zobrazovací plochu bude řešeno horizontálním posuvníkem.



Obrázek 3.10. Wireframe uživatelského rozhraní zobrazení detailu datového objektu uloženého v komponentě `Verta`

Výběr skupiny sdílení elektrické energie

V systému může být evidováno více skupin sdílení elektrické energie. Tato obrazovka (wireframe na obrázku 3.11) zobrazuje jejich souhrn a umožňuje přejít na jejich detail (na obrazovku s přehledem sdílení elektrické energie). U každé skupiny je jako první uvedeno vůdčí odběrné místo. Pokud je s tímto místem spojený zákazník, je zobrazeno jeho jméno, jinak je zobrazen externí identifikátor daného místa. Za vůdčím místem jsou uvedena všechna přidružená odběrná místa. Opět je zde uvedeno buď jméno zákazníka, nebo externí identifikátor odběrného místa. Navíc je zde ještě uvedena alokace (podíl z vyrobené energie). Podobné zobrazení je také použito v detailu pro zobrazení alokačního klíče.



Obrázek 3.11. Wireframe uživatelského rozhraní s výběrem skupiny sdílení elektrické energie pro zobrazení detailu

Kliknutím na libovolné odběrné místo bude možné přejít na detail skupiny, jehož je členem (obrazovka je zachycena na obrázku 3.12).

Přehled sdílení elektrické energie

Přehled sdílení elektrické energie bude dostupný z hlavního, postranního, menu v levé části obrazovky. V záhlaví obrazovky bude, stejně jako u ostatních, možné nastavovat časový rozsah, ve kterém mají být zobrazena všechna data. Wireframe s návrhem této stránky je zobrazen na obrázku 3.12 a pokrývá případ užití UC4.

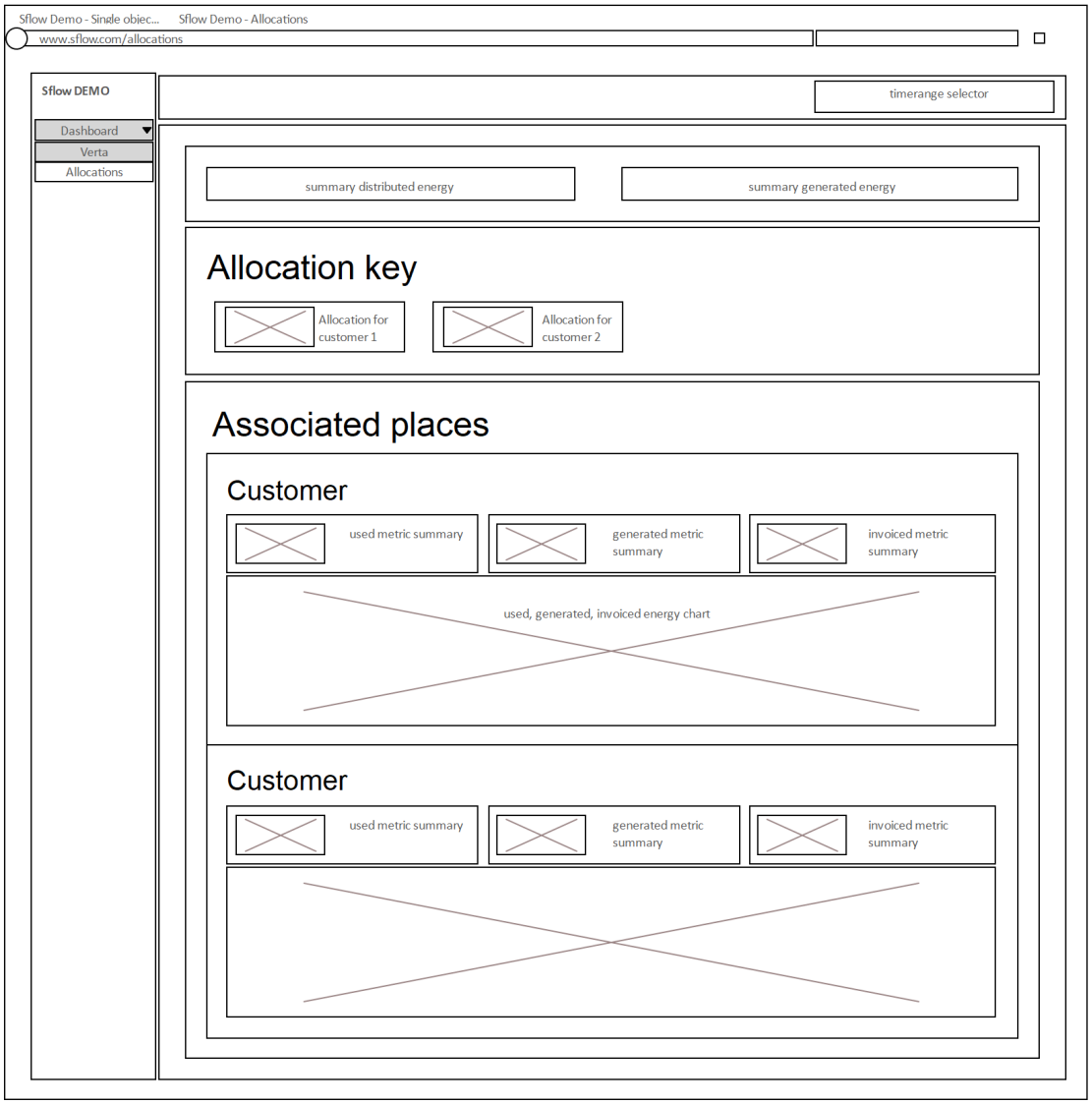
V horní části obrazovky bude přehled vyrobené energie a distribuované energie (přetoky do distribuční sítě). Pod těmito informacemi se bude nacházet alokační klíč, ve kterém bude uveden alokovaný poměr vyrobené energie pro každé odběrné místo.

Přehled jednotlivých odběrných míst bude uveden pod alokačním klíčem. Přehled každého odběrného místa bude obsahovat jméno zákazníka, který je pro dané odběrné místo evidovaný, případně identifikátor odběrného místa, nebude-li přiřazen žádný zákazník. Dále bude zobrazen přehled vyrobené, spotřebované, využité energie v rámci každého odběrného místa. Součástí toho přehledu bude také odhadované množství fakturované energie.

Tato data budou zobrazena formou spojnicového grafu jako souhrn za zvolený časový rozsah agregovaný do patnáctiminutových intervalů. Toto rozdělení respektuje časové úseky, které bude využívat distributor pro výpočet spotřebované energie (popsané v sekci 2.5.2). Kliknutím na souhrn bude možné přejít na zobrazení časové řady (wireframe 3.8), které bude obsahovat graf zvoleného měření s daty agregovanými do 15minutových intervalů pro stejný objekt, kterému patřil souhrn. Toto zobrazení bude také respektovat zvolený časový rozsah. Kromě detailního rozpadu jednotlivých řad bude také pro každou z nich zobrazen souhrnný součet za vybraný časový úsek.

3.7 Automatické úlohy

V případech užití UC10 a UC11 a funkčním požadavku F6 byla analyzována potřeba definovat a automaticky spouštět úlohy zpracování evidovaných dat. Pro evidenci jednotlivých úloh, jejich automatické i manuální spouštění bude použita technologie Apache



Obrázek 3.12. Wireframe uživatelského rozhraní pro zobrazení detailu sdílení elektrické energie

Airflow (popsaná v sekci 3.2.7). V této technologii lze také definovat samotné procesy. Nicméně je možné používat pouze programovací jazyk Python. Protože všechny ostatní části systému budou implementovány v jazyce Java, bylo by vhodnější využívat jej (případně alespoň jiný JVM jazyk) i pro tento účel.

Tento nedostatek lze kompenzovat kombinací s technologií Apache Spark, která umožňuje definovat jednotlivé úlohy zpracování dat v programovacím jazyce Java (případně Scala a Python). Dále také umožňuje distribuovat výpočty na jednotlivé servery v clusteru. Toho bude možné využít v budoucnu, kdy poroste množství dat evidovaných v systému, a tím množství dat, které bude nutné těmito úlohami zpracovávat.

Funkční požadavek a případy užití týkající se automatického zpracování dat úlohami tak nebudou pokryty žádnou implementovanou mikroslužbou, ale pouze využitím zmíněných technologií. V případě potřeby využití některé jejich funkcionality, jako je např. manuálního spouštění úloh, bude potřeba integrace na jejich API.

3.8 Shrnutí

V této kapitole byly popsány technologie, které vhodné pro návrh jednotlivých mikroslužeb, případně takové technologie, které by napomohly naplnit funkční požadavky. U každé z nich byly popsány její základní vlastnosti v kontextu navrhovaného systému. V návaznosti na tyto technologie a informace o nich získaných byl zvolen jako vhodný programovací jazyk pro implementaci návrhu jazyk Java a framework Spring Boot. Pro uživatelské rozhraní byl zvolen jazyk JavaScript s frameworkem Angular. Dále byl navrhnut rozšiřitelný datový meta model pro popis domény komunitních FVE. Společně s tím byl navrhnut i způsob ukládání časových řad do databáze HBase. Tento návrh se zaměřil zejména na unikátní identifikátory jednotlivých záznamů tak, aby bylo možné v časových řadách efektivně vyhledávat. Součástí něj byl také návrh ukládání samotných naměřených hodnot. Zejména se jednalo o způsob, jakým bude možné do databáze ukládat data různých datových typů.

Na základně těchto informací byly následně navrženy samotné mikroslužby. Každá mikroslužba má za úkol jednu ucelenou funkcionalitu. Díky tomu lze namapovat samotné mikroslužby na funkční požadavky, na jejichž naplnění se podílejí. Matice tohoto mapování je uvedena v tabulce 3.3. V řádcích jsou uvedeny názvy navržených mikroslužeb, ve sloupcích jsou funkční požadavky. Znak `x` lze poté interpretovat následovně: *Daná mikroslužba (sloupec) se podílí na naplnění konkrétního funkčního požadavku (řádek)*. Jedna mikroslužba se může podílet na naplnění více funkčních požadavků. Zároveň k naplnění jednoho funkčního požadavku může být zapotřebí více mikroslužeb. Funkční požadavek F6 není v matici spojen s žádnou mikroslužbou, protože jeho naplnění je navrženo pomocí spojení existujících systémů.

Některé funkční požadavky nejsou pokryty žádnou navrženou mikroslužbou, ale některou ze závislých služeb. Jedná se o funkční požadavek F6. Tento funkční požadavek je plně pokryt vhodným využitím technologií Apache Airflow a Apache Spark.

Celkově bude navržený systém obsahovat 5 mikroslužeb a uživatelské rozhraní. Dále bude jeho součástí knihovna pro práci s databází HBase, která bude poskytovat možnosti pro ukládání a vyhledávaní dat v časových řadách. Všechny tyto části systému budou implementovány podle provedeného návrhu.

mikroslužba / funkční požadavek	Schema registry	Verta	Data Stasher	Data Feser	Query Resolver
F1	x	x			
F2	x	x			
F3			x		
F4					x
F5				x	
F6					

Tabulka 3.3. Mapování navržených mikroslužeb na funkční požadavky

Kapitola 4
Implementace

V kapitole 3, zabývající se návrhem řešení, byla pro systém zvolena mikroservisní architektura a byly popsány jednotlivé návrhy mikroslužeb. Dále byl navržen datový model ukládání datových objektů a jejich vazeb. Datům se návrh věnoval také v kontextu časových řad. Pro ně byla vybrána technologie HBase a navržena struktura klíče.

V úvodu této kapitoly je popsána struktura kódu a základní koncepty, které jsou využívány v celé implementaci. Dále popisuje rozšíření navrženého datového modelu o konkrétní prvky FVE. K tomuto rozšíření je uvedena ukázka, jak s jeho pomocí namodelovat bytový dům, který využívá sdílení elektrické energie z FVE. Tento model navazuje na případ sdílení popsaný v sekci 2.5.2.

Na navržený model navazuje popis implementace Gradle pluginů pro generování POJOs z definovaných schémat. Tyto POJOs je možné využívat např. při implementaci úloh zpracovávajících evidovaná data. Kromě popisu, jak je implementována validace a ukládání objektů a vazeb, je také popsána konkrétní implementace vyhledávání v *databázi časových řad* a její typový systém. Součástí tohoto popisu jsou ukázky Java API poskytovány zmíněnou databází, na kterých jsou demonstrovány implementované funkcionality.

Dále tato kapitola popisuje navržené komponenty více do detailů a cílí především na detaily implementační, které jsou z hlediska řešeného problému zajímavé. U každé komponenty je dále uvedeno, jakým způsobem interaguje se zbytkem systému, případně zda komunikuje se softwarem/hardwarem třetích stran. Kromě samotných mikroslužeb je součástí této kapitoly také popis implementace *databáze časových řad*, která je poskytována jako knihovna, ne jako samostatná aplikace.

Na popis jednotlivých technologií navazuje popis implementace úloh zpracovávajících data evidovaná systémem. Zejména se jedná o popis implementace rozpočítávání sdílení elektrické energie, který je základním prvkem pro monitoring sdílené FVE. Není popisována samotná logika výpočtu, ale spíše integrace na komponenty systému.

V závěru kapitoly je uvedeno, jakým způsobem ji lze nasadit na lokální stroj s využitím nástroje Docker. A také jsou zde diskutovány podněty pro budoucí rozvoj.

4.1 Koncepty využívané v implementaci

V souladu s analýzou a vybraným programovacím jazykem a frameworkem je každá backendová funkcionalita implementována v jazyce Java ve verzi 17 – konkrétní používaná distribuce Javy je Amazon Corretto 17.0.8. Komponenty, resp. mikroslužby, jsou implementovány s využitím frameworku Spring Boot. Pro zajištění co nejpohodlnějšího vývoje je přistoupeno k použití frameworku pro mikroservisní architektury Spring Cloud [74]. Jedná se o nadstavbu nad navrženým frameworkem Spring Boot. Ve Spring Cloud jsou ale navíc obsaženy podpůrné služby pro implementaci distribuovaných sytémů jako service-discovery, load-balancing atp.

Tento framework není využíván v částech implementace, které mají spíše knihovní podobu. Jedná se o všechnu implementaci v adresáři common (např. *databáze časových*

řad). Často se totiž jedná o klienty, a tím pádem je vhodné, aby byly dostupné bez závislosti na frameworku.

Struktura kódu je u všech mikroslužeb podobná. Zásadní části kódu, povětšinou se jedná o interfaces nebo jiné veřejné kontrakty, jsou opatřeny dokumentačními komentáři. Všechny se řídí třívrstvou architekturou – prezentační, business a datová vrstva. Komunikaci s okolím (příjímání dotazů) zajišťuje prezentační vrstva. Ta je implementována pomocí kontrolérů. Jedná se o Java třídy anotované `@Controller` (konstrukt frameworku Spring Boot). Z pravidla jsou umístěny v adresáři `controller`, který je v kořenovém adresáři každé komponenty. Kód vykonávající business logiku dané mikroslužby je umístěn v adresáři `service` na stejné úrovni. Názvy ostatních adresářů poté již nelze takto generalizovat a vždy je nutné je chápat v kontextu dané mikroslužby.

Implementace je verzována pomocí nástroje Git [75] v jednom repozitáři, tzv. monorepozitář. Všechny aplikace – jednotlivé mikroslužby – a implementované knihovny jsou tak uloženy v jednom repozitáři. Tento přístup zvyšuje přehlednost a centralizuje sledování změn v celém systému. Mimo jiné také usnadňuje automatické procesy, jako je kontinuální integrace a nasazování.

Repozitář je rozdělen do dvou částí – `apps` a `common`. Všechny mikroslužby jsou umístěny v adresáři `apps`. Projekty, které se v něm nacházejí, nemají mezi sebou žádné závislostí vazby. Sdílené části kódu, jako jsou například definice komunikačních protokolů, případně klienti mikroslužeb, jsou umístěny v adresáři `common`. Z adresářové struktury tak lze snadno poznat, který kód je sdílený mezi více mikroslužbami a který nikoliv. Zároveň lze snadno předcházet špatným rozhodnutím v podobě přímých závislostí mikroslužeb, a to případně i automaticky.

4.2 Datový model a jeho konzistence

V návrhu systému bylo zmíněno, že všechny datové struktury zpracovávané v systému a jejich vazby budou definovány v deklarativním jazyku JSON schema. Ten poskytuje několik základních konstrukcí, s nimiž lze definovat požadovanou strukturu JSON dokumentů, ale také různá omezení na jednotlivé hodnoty (pro textové položky např. pomocí regulárního výrazu nebo omezení na nejmenší možné číslo). Skutečné JSON dokumenty lze podle těchto definic validovat. Zároveň lze z této definice vygenerovat Java třídy (tzv. POJOs) reprezentující dané schéma. Vygenerované třídy pak lze použít v kódu pro zajištění typování příchozích (resp. odchozích) zpráv, stejně tak jako pro deserializaci (resp. serializaci) z JSON.

V implementaci systému struktura JSON schémat koresponduje s datový modelem uvedeným v návrhu 3.4.1 (případně rozšířením popsaným v sekci 4.2.1). Všechna schémata společně s vygenerovanými POJOs jsou zabaleny jako jeden sdílený modul (`:common:schema`), který mohou využívat všechny části systému.

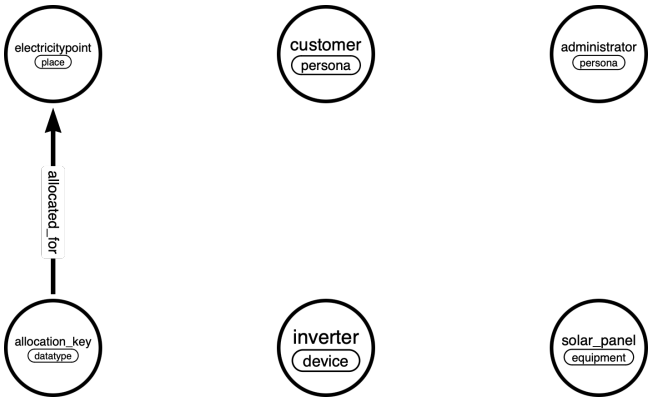
4.2.1 Definice datového modelu

V implementaci je definováno několik kořenových schémat, od kterých dědí všechny ostatní. Každé kořenové schéma určuje skupinu objektů, které spolu logicky souvisí. Základní čtyři jsou: `datatype` popisující datové typy, `element` popisující fyzické a abstraktní objekty, `relation` specifikující vztahy mezi objekty a `mode` upřesňující libovolné objekty podle návrhu v datovém modelu.

Od těchto schémat dědí další, která buďto reprezentují konkrétní typy objektů, nebo jsou více specifickou abstrakcí. Například od kořenového schématu `element` dědí všechny specializace typu `Kind` (jako je `device`, `equipment` atp.) popsané v návrhu

a uvedené ve schématu na obrázku 3.5. Až od této druhé abstraktní vrstvy jsou děděny konkrétní objekty, jako `inverter`, `switchboard` atp. Konkrétní objekty, které využívá implementace, jsou zachyceny ve schématu na obrázku 4.1. Od nich pak může v databázi existovat několik instaci. U každého objektu je uveden jeho rodič. Zároveň je součástí schématu i jedna rozšiřující vazba.

Vlastnosti jednotlivých hran a objektů zde nejsou uvedeny, ani dále v textu. V případě specifických případů, kdy je tomu jinak, jsou důležité vlastnosti zmíněny přímo v textu. Ostatní poté lze nahlédnout ve zmíněném sdíleném modulu `:common:schema` ve zdrojovém kódu.



Obrázek 4.1. Schéma rozšířeného modelu použitého pro implementaci

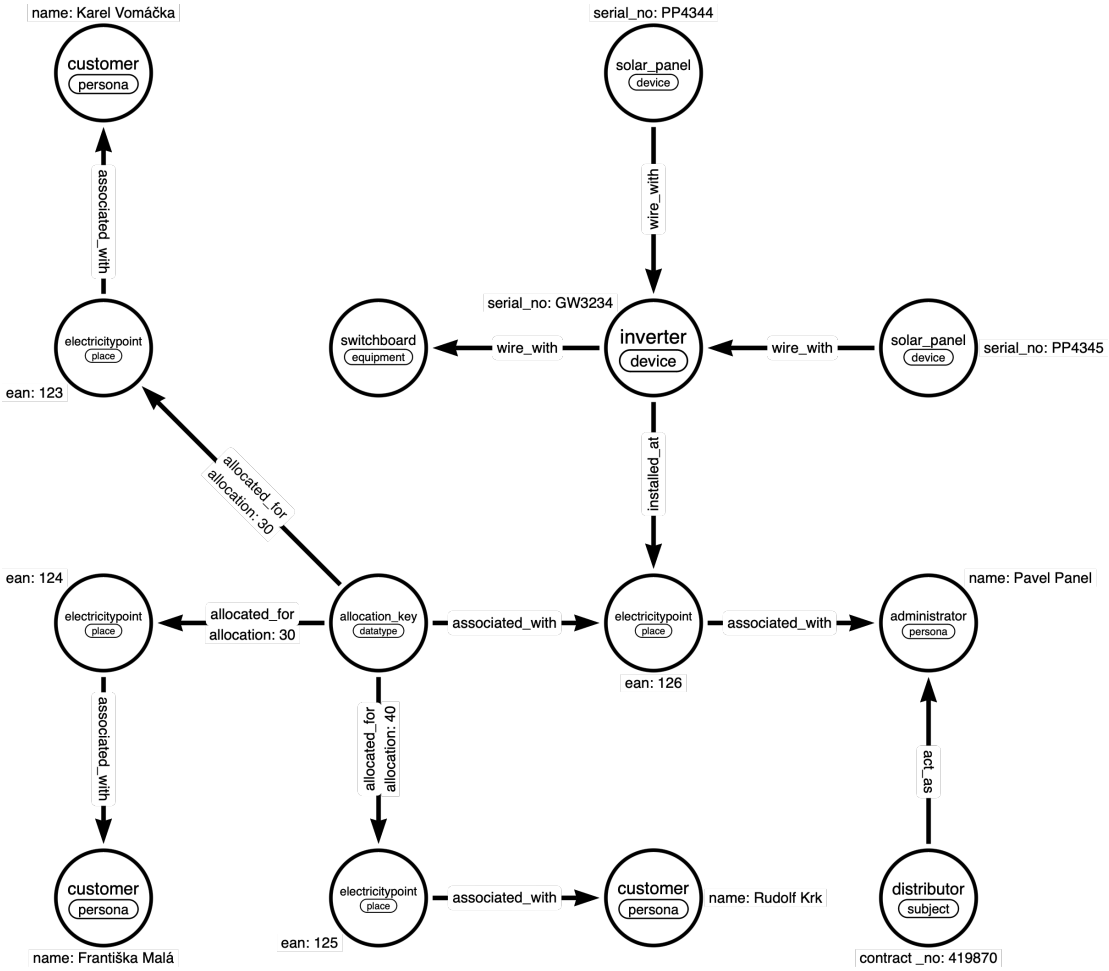
4.2.2 Instance bytového domu

V sekci 2.5.2 byl popsán modelový příklad sdílení energie v bytovém domě. Stejný příklad sdílení je uveden v této sekci, pouze je dán do kontextu navrženého datového modelu ze sekce 4.2.1. Schematicky je model bytového domu sdílejícího elektrickou energii zachycen na obrázku 4.2. Tento model přímo reflektuje strukturu uložení dat v grafové databázi *Verta*.

Na schématu lze vidět, že všechna odběrná místa (`eletricitypoint`) jsou sdružena kolem objektu instance typu `allocation_key` (alokační klíč). To je vhodné například v dotazování. Pokud bude potřeba získat všechna odběrná místa, která jsou součástí sdílení, stačí získat příslušnou instanci alokačního klíče. Vazby mezi alokačním klíčem a odběrným místem obsahují informaci o tom, jaký podíl vyrobené elektrické energie má dané odběrné místo alokováno. Každé odběrné místo má poté přiřazeno jednoho zákazníka – `customer` (v případě vůdčího místa je typ objektu zákazníka `administrator`), který je s ním smluvně spjat. Hlavní odběrné místo má navíc vazbu se zařízením typu `inverter` (střídač). Střídač má poté vazby na další komponenty FVE (ty už do systému neposkytují žádné informace).

4.2.3 Generování POJOs

Pro generování Java tříd z JSON schémat existuje několik knihoven. Pro implementaci byla zvolena knihovna *jsonschema2pojo* (zejména kvůli přetrvávajícímu udržování komunitou). Bohužel ale tato knihovna nepodporuje možnost definovat v JSON schématech vícenásobnou dědičnost vůbec, jednoduchou dědičnost poté jen částečně. Jelikož datový model vyžaduje použití dědičnosti (reprezentuje skutečný svět, kde dědičnost, resp. kategorizace dat, je jeho běžnou součástí), např. `inverter` je potomkem `device`, který je potomkem `element`. Z hlediska jazyka JSON schema dává tato vlastnost smysl,



Obrázek 4.2. Schéma datového modelu sdílené FVE v bytovém domě

jeho cílem není popisovat objekty, ale pouze definovat omezení, které objekty musejí splňovat.

Ve schématu je možné slučovat jednotlivé validace pomocí odkazů (direktiva `$ref`) a direktivy `all_of`. Tato direktiva při validaci zajistí, že JSON dokument musí splňovat definované schéma, ale i všechna schémata odkazované ve zmíněné direktivě. S využitím této vlastnosti a zmíněné knihovny pro generování lze dosáhnout požadovaného stavu, generovaná POJOs půjdou definovat jako hierarchická struktura následujícími dvěma kroky.

1. dereferencovat odkazovaná schémata (a to i rekurzivně)
2. všechny položky (dereferencované) direktivy `all_of` sloučit s rodičovským schématem

Obě operace jsou rekurzivní a prováděny průchodem stromu závislostí od kořene k listům. Od listu se poté zpětně sestavuje výsledné schéma. V druhém kroku je ještě nutné rozhodnout, jak se má generování zachovat v případě duplicitního klíče jak v rodičovském, tak referencovaném schématu. Z hlediska implementace dává smysl zachovat obojí, pouze doplnit sufix s pořadovým číslem (případně jiným unikátním identifikátorem) nebo, stejně jako u objektového návrhu, přepsat rodičovskou definici implementací potomka (overriding).

Výsledkem této operace je validní JSON schéma, které přijme (resp. bude požadovat za validní) stejnou množinu JSON dokumentů jako původní JSON schéma. Neobsahuje však reference na jiná schémata a neobsahuje žádnou direktivu `all_of`. Jedná se tedy o flatten původního schématu. Takto modifikovaná schémata jsou uložena do komponenty *Schema Repository*, která je následně poskytuje ostatním mikroservisám. Dále lze z těchto schémat již snadno vygenerovat POJO, které bude obsahovat všechny položky všech referencovaných schémat. Nevýhodou tohoto řešení je duplikace informací a zbavení se dědičnosti i na úrovni Java tříd. Pro opakující se typ v několika JSON schématu se pro každý jeden výskyt vygeneruje nový datový typ. Např. pro dvakrát referencované schéma budou vygenerovány dvě jeho implementace.

Obrázek 4.3. Schématické zobrazení hledaného vzoru dotazem 4.7

Příklad spojování schémat

Na obrázku 4.3 je zachycen stav dvou JSON schémat v průběhu generování flatten schématu, ze kterého je následně generováno POJO. Kroky `1` a `2` jsou pouze definice schémat, které do procesu vstupují. První schéma (`parent.schema.json`) obsahuje

referenci na definici, která je uvnitř stejného souboru. Tato reference je zvýrazněna červenou barvou. Druhé schéma (`example.schema.json`) obsahuje referenci na první soubor. Tato reference je zvýrazněna modrou barvou.

Další popis se bude týkat případu, kdy popsaný algoritmus bude spuštěn na druhý soubor – `example.schema.json`. V prvním kroku se provede dereferencování, tedy za červenou referenci v kroku `2` bude substituováno dereferencované schéma `parent.schema.json`. Toto schéma však také obsahuje referenci, proto je nutné ho nejprve dereferencovat. Reference odkazuje pouze na vnitřní definici, která již neobsahuje další reference. Je tedy možné provést pouze substituci. Výsledek této operace je zobrazen v kroku `3`.

Nyní je možné provést substituci v `example.schema.json`, které záviselo na `parent.schema.json`. Provede se tedy substituce modré reference za dereferencované schéma z kroku `3`. Výsledek této operace je zobrazen v kroku `4`.

V tento moment je výsledný soubor dereferencovaný, neobsahuje odkazy na jiné soubory ani na své vlastní definice. Nicméně schéma je pořád zanořeno v direktivě `all_of`. V tento moment se tedy provede druhý popisovaný krok, a to spojení všech schémat v direktivě `all_of` s rodičovským schématem. Výsledek této operace je zobrazen v kroku `5`.

Díky barevnému znázornění referencí jde vidět, že reference z `parent.schema.json` se tímto mechanismem přenesla do výstupního souboru v dereferencované podobě jako nová vlastnost JSON dokumentu. Původní a výsledný soubor tak označí za validní stejnou množinu schémat jako původní schéma.

4.3 Databáze časových řad s využitím HBase

V sekci 3.4.2 byl diskutován a navrhnut způsob, jakým je vhodné ukládat naměřená data do databázového stroje HBase. Funkcionalitami a rozsahem implementace *databáze časových řad* byl splněn funkční požadavek F3. Implementace se drží návrhu a využívá definovanou strukturu klíče (rowkey) a tabulky. Samozřejmě bylo nutné některé navrhované části více specifikovat a přizpůsobit implementačním detailům použité databáze. Tato sekce zevrubně popisuje tyto potřeby.

V analýze byla navrhnuta struktura klíče řádků – rowkey. Stejně jako tam, i v implementaci bylo implementování logiky vytváření klíče vstupním místem do problematiky. V tomto případě však již muselo být přihlédnuto ke způsobu, jakým je možné v HBase filtrovat, resp. dotazovat, data. I toto však již bylo nastíněno v návrhu. Veškeré dotazování je implementováno pomocí tzv. *Scan*, resp. *Get*. Oba přijímají jako parametr *Filter* (resp. jejich seznam). Jediný rozdíl je v tom, že *Scan* lze aplikovat na rozsah řádků (vše mezi počátečním a koncovým rowkey), zatímco *Get* je aplikovatelný pouze na jeden řádek (jedno rowkey).

Pro operace *Scan* a *Get* omezuje *Filter* množinu řádků, která bude těmito operacemi zpracována. Do operace *Scan* tak nevstoupí všechny řádky mezi počátečním a koncovým rowkey, ale jsou nejdříve profiltrovány operací *Filter*. Zpracovávané řádky tak nemusejí být bezprostředně následující. V implementaci se provádí zejména filtrování podle řádků. Jsou proto využity implementace *RegexStringComparator* a *FuzzyRowFilter*.

4.3.1 Struktura identifikátoru záznamů

Návrh identifikátoru záznamů naměřených hodnot neboli rowkey neboli klíč záznamu byl diskutován v sekci 3.4.2. Implementace tento návrh respektuje, nicméně v jednom

bodě se mírně liší. Zachovány jsou všechny části klíče, včetně jejich pořadí. Dále je zachován koncept mapování hodnot na unikátní identifikátory pevně dané délky.

Rozdílem je implementace mapování a ukládání značek. V návrhu bylo pro každou značku rezervováno 6 bytů – tři pro hodnotu a tři pro klíč. Aby nebylo nutné v implementaci neměnně definovat tuto délku, je před každý klíč a hodnotu značky umístěn byte navíc, který určuje jejich délku. Jedna značka tak zabírá vždy o 2 byty více. Touto implementací je dosaženo konfigurovatelnosti a optimalizace počtu možných značek. Aktuální implementace *databáze časových řad* neumožňuje tento počet měnit dynamicky, je nutné aby se po prvním uloženém záznamu dále neměnil. Tato vlastnost je důsledkem implementace filtrování záznamů popsaným v sekci 4.3.2. Druhým implementovaným mechanismem, který nebyl v návrhu popsán, je řazení značek. Opět i tato vlastnost je spjata s implementací filtrování. Všechny značky jsou seřazeny podle identifikátoru klíče značky. Pokud je shodný, dále se řadí podle identifikátoru hodnoty značky.

4.3.2 Filtrování záznamů

Návrh klíče provedený v sekci 3.4.2 nezaručuje, že hodnoty (reprezentované jako řádky v tabulce) jedné časové řady (definované unikátním identifikátorem a značkami) budou vždy uloženy bezprostředně za sebou. Zaručeno je pouze, že budou uloženy ve správném pořadí, tedy seřazeny podle času. HBase řadí záznamy lexikograficky, a proto pokud se budou řádky lišit pouze v časové značce, která je striktně neklesající, budou vždy seřazeny. Z toho důvodu není možné v implementaci použít vyhledávání záznamů pouze pomocí *Scan*, který umožňuje pouze definovat počáteční a koncový řádek, resp. jejich klíče. Výsledkem tohoto dotazu jsou pak všechny řádky nacházející se v zadaném rozsahu. Protože není zaručeno, že všechny řádky jedné časové řady budou vždy následovat za sebou a nebudou mezi nimi řádky jiné časové řady, je nutné využít jiný způsob filtrování záznamů.

UID	časová značka	značka	hodnota
M1	123	A:1	78
M2	123	B:1	90
M1	124	A:1	70
M1	125	A:1	23
M1	126	A:1	98
M2	126	B:1	12

Tabulka 4.1. Ukázka překrývajících se záznamů dvou časových řad M1 a M2

K dotazování pouze na záznamy jedné časové řady lze využít *RowFilter* s některým z poskytovaných komparátorů. Pro složené klíče se hodí *RegexStringComparator*. Nicméně ten musí projít všechny záznamy v dané tabulce a vybrat pouze ty, které vyhovují zadanému komparátoru. Tento přístup je dostačující v začátcích, ale s rostoucím počtem řádků bude mít tento přístup neblahý vliv na výkon vyhledávání. Jednoduchou optimalizací, aby nemusela být procházena celá tabulka, je určení počátečního a koncového klíče. Využívání volby rozsahu, ve kterém se mají data vyhledávat je v dalším textu implicitně považován za použitý (není-li uvedeno jinak). Další optimalizací je filtrování řádků pomocí *FuzzyRowFilter*.

FuzzyRowFilter má na vstupu vzor (částečně vyplněný) klíč a bytovou masku. Umožňuje skenování v tzv. fast-forward režimu, kdy nenačítá všechny řádky, ale přeskakuje ty, jejichž klíč neodpovídají zadanému vstupu (vzoru a masce). Bytová maska určuje,

které byty ze vstupního vzoru jsou fixně dané a které mohou nabývat libovolné hodnoty. Tímto filtrováním lze docílit rychlejšího vyhodnocování dotazu. Díky správnému návrhu, který přesně definuje délky jednotlivých částí klíče, je využití tohoto způsobu vyhledávání v implementaci nejen možné, ale dokonce jednoduché. Bez pevně definovaných velikostí jednotlivých částí by nebylo možné určit, které byty patří ke které z částí, a tudíž by nebylo možné v masce definovat, které byty jsou pevné a které variabilní. Oproti jiným možnostem klade tento filtr nepřímo omezení na stejnou délku všech dotazovaných klíčů (případně alespoň stejnou délku prefixu těchto klíčů).

Databáze poskytuje celkem tři základní typy dotazů. Pro každý typ je uvedena ukázka kódu. V ukázkách jsou vynechány detaily vytváření některých objektů a jsou nahrazeny skutečnou hodnotou, se kterou se poté pracuje. Takové části jsou uvedeny v komentářích (`/* */`). Typy dotazů jsou co do implementace velice podobné a jsou přístupné přes sjednocené Java API.

Prvním typem dotazu je dotazování na celou (nebo část) časovou řadu, u které je předem známé UID měření a všechny značky (příklad takového dotazu je uveden v ukázce kódu 4.1). Jelikož všechny záznamy jedné časové řady mají stejné UID a stejné značky (klíče se liší pouze v časové značce), je délka klíčů všech hledaných záznamů stejná. Dokonce je dopředu známá. Známé je také, na které pozici se nachází jaká značka (značky jsou seřazeny vzestupně podle klíče, pokud je klíč stejný, tak podle hodnoty klíče). Pro dotazování na jednu časovou řadu tak lze využít jak *FuzzyRowFilter*, tak *RowFilter* se *RegexStringComparator*. V implementaci jsou pro tento typ dotazů využity oba zmíněné filtry najednou. Tím se zvyšuje efektivita dotazování, protože není nutné procházet celou tabulku, ale díky *FuzzyRowFilter* jsou některé části přeskočeny.

```
GetDataPoints
    .builder()
    .context(context)
    .table(/* metering */)
    .input(
        DataPointsRange
            .builder()
            .metering(/* 123 */)
            .tags(/* tag1:val1 */)
            .from(/* 1684251461 */)
            .to(/* 1694251461 */)
            .tagsMustExactlyMatch(true)
            .build()
        )
    .build();
```

Kód 4.1. Ukázka dotazu na časovou řadu podle UID a značek (s přesnou shodou)

Druhým typem dotazu, který využívá také kombinaci obou dvou zmíněných filtrů, je dotazování na konkretní měření (UID) obsahující požadovanou značku, případně několik značek. Jedná se o stejný typ dotazu jako předchozí, jen není nutné, aby záznamy, které mají být vráceny, obsahovaly pouze zadané značky, mohou obsahovat i další. Oproti předchozímu typu dotazu zde není zaručeno, že všechny klíče budou stejně dlouhé. Není ani možné určit, na které pozici se hledaná značka nachází. Přesto ale lze najít společný prvek pro všechny hledané klíče. Je jím stejně dlouhý prefix, složený z UID

a časové značky. Právě s tímto prefixem bude pracovat *FuzzyRowFilter* a bude jím zajištěno přeskočení záznamů, které nemají hledané UID. Na další filtrování však v tomto případě *FuzzyRowFilter* použít nelze a implementace se v něm spoléhá na *RowFilter* se *RegexStringComparator*.

```
GetDataPoints
    .builder()
    .context(context)
    .table(/* metering */)
    .input(
        DataPointsRange
            .builder()
            .metering(/* 123 */)
            .tags(/* tag1:val1 */)
            .from(/* 1684251461 */)
            .to(/* 1694251461 */)
            .tagsMustExactlyMatch(false)
            .build()
        )
    .build();
```

Kód 4.2. Ukázka dotazu na časovou řadu podle UID a značek (bez přesné shody)

V ukázkách kódu 4.1 a 4.2 je vidět, že první dva typy dotazů jsou shodné až na parametr `tagsMustExactlyMatch`. V případě hledání konkrétních záznamů časové řady obsahuje dotaz vyčerpávající výčet všech značek. Tím pádem je znám jejich počet. Zároveň každá značka se stejným klíčem i hodnotou může být v klíči záznamu pouze jednou. Proto lze pouze porovnat počet hledaných značek s počtem značek každého záznamu. Je-li tento počet shodný, klíč záznamu obsahuje právě a jen požadované značky. Rozhodnutí, zda se mají počty značek porovnávat, a tedy, zda je hledána přesná shoda ve značkách, je řízeno uživatelem, resp. jím poskytnutou informací, v dotazu.

Posledním, třetím, typem dotazu je dotazování na záznamy pouze podle značek bez specifikace UID. Stejně jako v předchozích dvou případech je i pro tento typ dotazu možné využít zmíněné dva filtry. V případě, že uživatel požaduje přesnou shodu ve značkách, musí tento požadavek v dotazu specifikovat (stejně jako u druhého typu dotazu v atributu `tagsMustExactlyMatch`). Pokud je požadována přesná shoda značek, porovnává se jejich počet. Příklad jednoho takového dotazu je uveden v ukázce kódu 4.3.

Všechny dotazy používají stejnou implementaci, liší se pouze v parametrech. Vždy se jedná o kombinaci dvou typů filtrů – *RowFilter* se *RegexStringComparator* a *FuzzyRowFilter*. Regulární výraz, který je použit pro *RegexStringComparator*, je dynamicky vytvořený na základě vstupních parametrů. Obsahuje konkrétní UID (pokud se jedná o první dva typy, jinak nikoliv) a konkrétní značku, příp. značky. Příklad regulárního výrazu pro UID `123` a značku `tag1:val1` je uveden v ukázce kódu 4.4. V ukázce je také uvedeno mapování skutečných hodnot na jejich unikátní identifikátory předem dané délky. Důvod tohoto mapování byl popsán v návrhu v sekci 3.4.2. Dále je v implementaci všech dotazů využit ještě poslední typ filtrování, který byl zmíněn pouze v návrhu v sekci 3.4.2, a tím je určení počátečního a koncového klíče záznamu, mezi kterými se má vyhledávat. Tento výběr slouží pro zúžení prostoru vyhledávání a zároveň pro výběr dat časové řady ve specifickém časovém rozsahu.

```
GetDataPoints
    .builder()
    .context(context)
    .table(context.tsdbConfiguration().getTableMetering())
    .input(
        DataPointsRange
            .builder()
            .tags(metering1.tags())
            .from(Timestamp.builder().unix(0L).build())
            .to(Timestamp.builder().unix(9000L).build())
            .tagsMustExactlyMatch(false)
            .build()
        )
    .build();
```

Kód 4.3. Ukázka dotazu na časovou řadu podle značek

```
unikátní ID měření          123         001
značka 1 <klíč:hodnota>     tag1:val1   004:008

(?s)^\\x5CQ\\x00\\x00\\x01\\x5CE.*\
(?:.{8})*\\x5CQ\\x03\\x00\\x00\\x04\\x03\\x00\\x00\\x08\\x5CE(?:.{8})*$
```

Kód 4.4. Ukázka regulárního výrazu pro filtrování záznamů

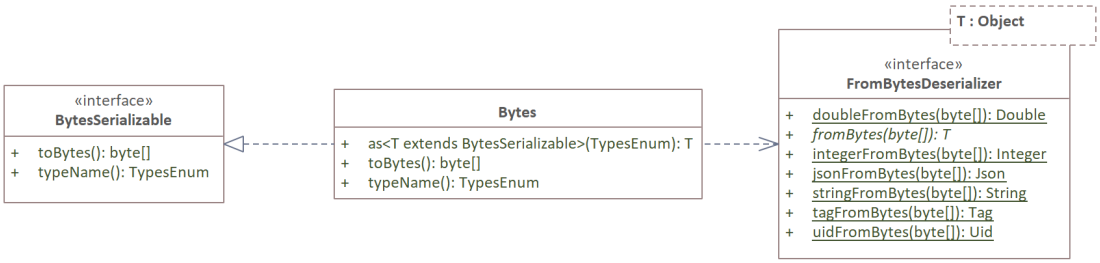
4.3.3 Ukládání dat

Všechna měření (časové řady) jsou ukládána do jedné tabulky s názvem `metering`. Schéma tabulky je shodné s navrhovaným, které je uvedeno v tabulce 3.2. Aby bylo poskytované rozhraní typované a nepřenášela se povinnost zachovávat informace o uložených typech dat na klienta, je společně s každým záznamem ukládána informace o uloženém typu do buňky `meta:type` (column family:qualifier). Typování je prováděno na úrovni jednoho záznamu, není tedy zaručeno, že všechny záznamy jedné časové řady budou mít pod stejným column-family a qualifier hodnotu stejného typu. Zaručení této integrity je přenecháno na klientovi, pokud ji vyžaduje. Znalost datového typu je také potřeba při deserializaci, protože HBase ukládá veškerá data jako neinterpretované pole bytů. Zde je opět vhodné mít informaci o typu uloženou přímo v databázi. Není nutné dotazovat se do jiného systému, případně si někde bokem vést druhou databázi o tom, pod kterým colmun-family a qualifier pro jakou časovou řadu je uložen který datový typ (to by navíc vedlo na nutnost využívat stejný datový typ v jedné časové řadě).

Z pohledu implementace je typový systém navržen tak, že každý typ, který má být uložen do databáze, musí implementovat interface *BytesSerializable*. Tím je zajištěno, že daný typ bude možné převést do pole bytů a zároveň je možné získat informaci o datovém typu. Uživatel tak s využitím implementovaného klienta nemůže do databáze uložit data, která by neuměl přečíst. Datový typ ukládaný do databáze je reprezentován jako výčtový typ. Možností by bylo také použít přímo kvalifikovaný název Java třídy, nicméně v takovém případě nelze kontrolovat v čase kompilace, zda je třída s tímto názvem dostupná, a tím pádem by mohlo docházet k běhovým chybám. Využitím výčtového typu se implementace tomuto neduhu snaží předejít tím, že v případě přidání nového typu

je pro jeho serializaci, resp. deserializaci, nutné přidat jeho reprezentaci do zmíněného výčtového typu. Proto v místech, kde se na základě jeho hodnoty rozhoduje, jaká implementace bude použita, již nebude uveden vyčerpávající výčet, tím pádem nastane chyba již při sestavování. Nicméně tento mechanismus ochrání programátora pouze před chybami spojenými s vývojem nových datových typů. Nijak nezajišťuje zpětnou ani dopřednou kompatibilitu. Pokud tedy bude uložen nový datový typ, který bude následně přečten starší verzí klienta, která daný datový typ nemá implementovaný, dojde k běhové chybě. Schematicky je tento návrh tříd zobrazen v diagramu na obrázku 4.5.

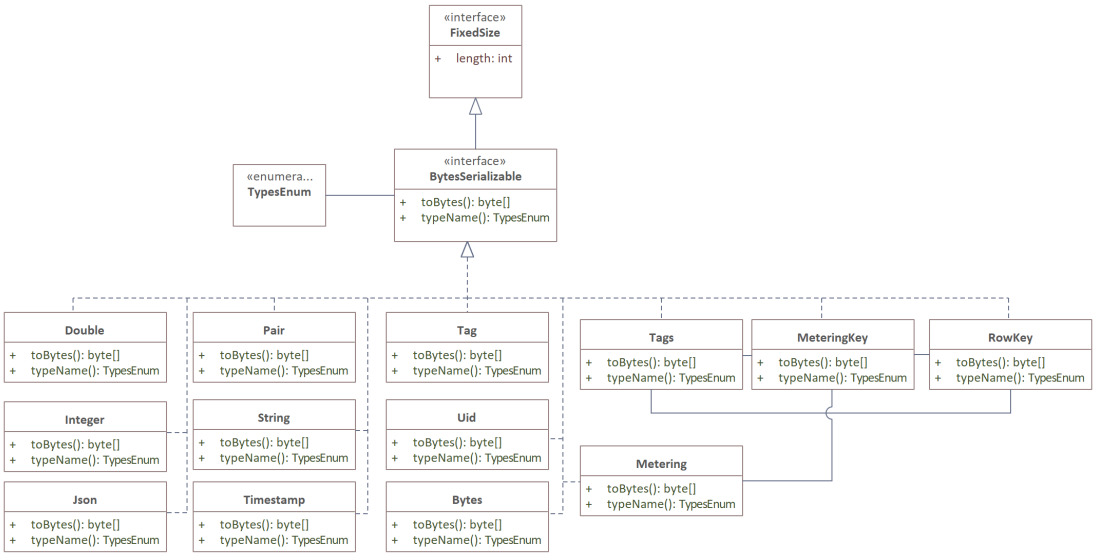
Pro deserializaci by bylo vhodné použít stejný mechanismus, tedy všechny typy by implementovaly určitý interface *BytesDeserializable*, který by požadoval implementaci metody, která by převedla pole bytů na daný datový typ. Bohužel takovýto koncept není možné v typovém systému jazyku Java provést. Metoda by musela být statická, tím pádem by nemohla být bez implementace, a tím pádem by nebylo možné ji využít ve zmíněném interface či jiném abstraktním konstruktu. Druhou možností by mohla být existence servisní třídy, která by obstarávala samotnou deserializaci. V implementaci je však deserializace vyřešena ještě jiným způsobem, a to následovně. Pro každý deserializovatelný typ musí existovat implementace generického funkcionálního interface *FromBytesDeserializer*. Bohužel nelze při kompilaci kontrolovat, zda tato implementace skutečně existuje, či nikoliv. Implementace je vždy provedena jako statický objekt, který je veřejnou metodou interface *FromBytesDeserializer*. Implementace všech deserializací jsou tak na jednom místě a je jednoduché přidat implementaci pro nový deserializovatelný typ. V podstatě se jedná o statickou servisní třídu. Schéma tříd implementujících deserializaci je zobrazeno na obrázku 4.4.



Obrázek 4.4. Diagram tříd deserializace typů uložených v HBase

Výběr správného deserializátoru pro daný uložený datový typ je řízen samotnou instancí *Bytes*, která obsahuje serializovaná data načtená z databáze. Správný deserializátor vybírá podle typu dat, která obsahuje. Tuto informaci získá z databáze podle hodnoty uložené v column-family `meta` v qualifier `type`. Zde se projevuje výhoda ukládání datového typu společně s každým záznamem, která byla popsaná výše. Typ je uložen společně s každým záznamem, a tím pádem je jednoduché vybrat podle něj správný deserializátor. V případě nového typu tak není nutné měnit strukturu ukládaných dat ani provádět jiný zásah do databáze. Pouze se do dané buňky zapíše identifikátor nového datového typu.

Přidávání nových typů, které lze do databáze ukládat, je z pohledu kódu velice snadné. Stačí pouze rozšířit zmíněný interface a přidat statickou metodu pro deserializaci. U případného odebírání datového typu je nutné myslet na to, že taková změna není zpětně kompatibilní. Použití datového typu je vždy podmíněno existencí zmíněných implementací serializace a deserializace v dané verzi *databáze časových řad*. Na tuto skutečnost je nutné myslet při používání databáze, zejména pak, je-li provozována



Obrázek 4.5. Diagram tříd typů ukládaných do HBase

na více strojích (ve více aplikacích), kde na každém není spuštěna stejná verze. Nejlepší však je se těmto nekompatibilním změnám zcela vyhnout.

Ukázka uloženého záznamu

Ukázkový záznam datového typu `Integer` s hodnotou `1`, uložený v řadě s názvem `test`, které odpovídá UID `123`, v čase `12345678` a se značkami `tag1:val1` (klíči značky odpovídá UID `001` a hodnotě UID `002`) a `tag2:val1` (klíči značky odpovída UID `002`, hodnota je stejná jako u předchozí značky, tedy její UID je `001`) je zobrazen v tabulce 4.2.

rowkey	123123456784001400140024001	
column family	value	meta
qualifiers	value=1	type=`integer`

Tabulka 4.2. Ukázka záznamu v tabulce měření

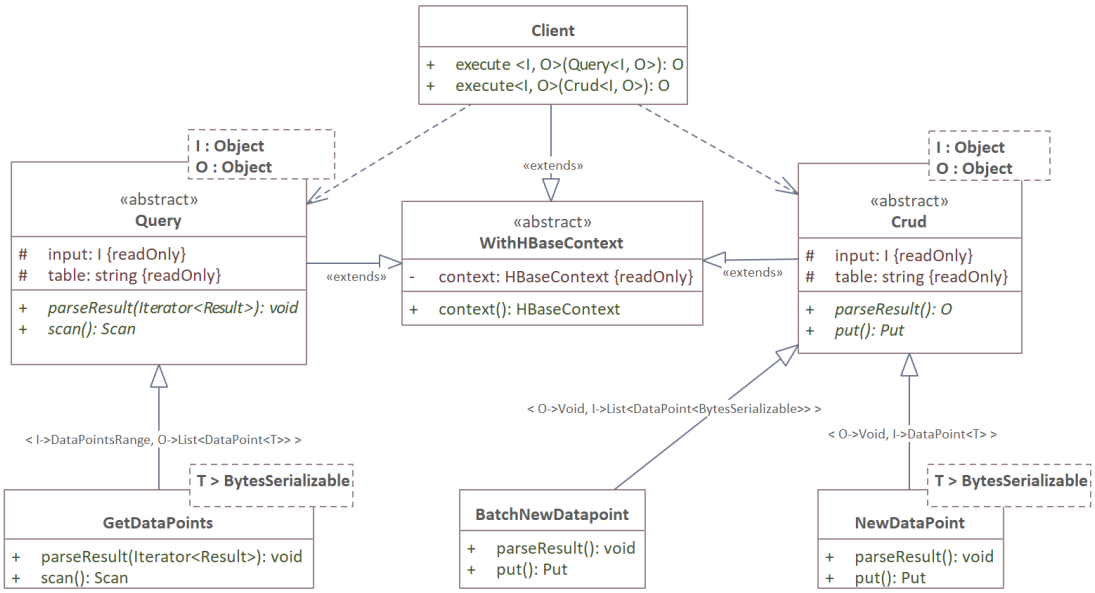
4.3.4 Aplikační rozhraní

Databáze časových řad poskytuje Java API, s jehož pomocí umožňuje vykonávat dva typy dotazů. První druh provádí modifikace dat (CRUD operace), druhým typem dotazů je možné provádět pouze dotazy. Obě kategorie jsou implementovány jako abstraktní třídy s generickými typy určující vstupní a výstupní typ dané operace, resp. dotazu. Diagram tříd TSDB klienta je zobrazen na obrázku 4.6. V diagramu jsou zachyceny pouze části, které se přímo týkají zápisu, resp. čtení dat. Jsou vynechány pomocné typy dotazů.

Jelikož HBase neumožňuje vytváření transakcí, je nutné synchronizaci zápisu a změn v databázi řešit na vyšší úrovni. Rozdělení dotazů na tyto dvě kategorie umožňuje oddělit operace, které lze provádět paralelně (dotazování) a které ne (CRUD operace). V systému by tedy měl existovat právě jeden klient, kterým se bude do databáze zapisovat. Klientů pro dotazování může existovat více. Z pohledu mikroservis by měla do databáze zapisovat pouze komponenta *Data Stasher*, všechny ostatní data pouze čtou. V případě, že potřebuje jiná komponenta zapsat data do systému, odešle je do Kafka

topicu, odkud je asynchronně zpracuje *Data Stasher*. Tento přístup je využívaný např. ve Spark Jobs, pro ukládání výsledků vykonávaných operací.

Poskytované rozhraní je plně typované, tedy není nutné udržovat externě informaci o tom, jaký datový typ byl do daného místa v tabulce uložen. Informace z databáze je vždy správně deserializována do příslušného datového typu. S každým záznamem je do databáze ukládán i typ uložené informace (toto je potřeba, protože HBase nijak neinterpretuje uloženou hodnotu). Typ je uložen do column-family `meta`, qualifier `type`. Ukládanou hodnotu typu poskytuje každý typ implementující interface *BytesSerializable* (vizte sekci 4.3.3).



Obrázek 4.6. Zjednodušený diagram tříd klienta a dotazů do TSDB

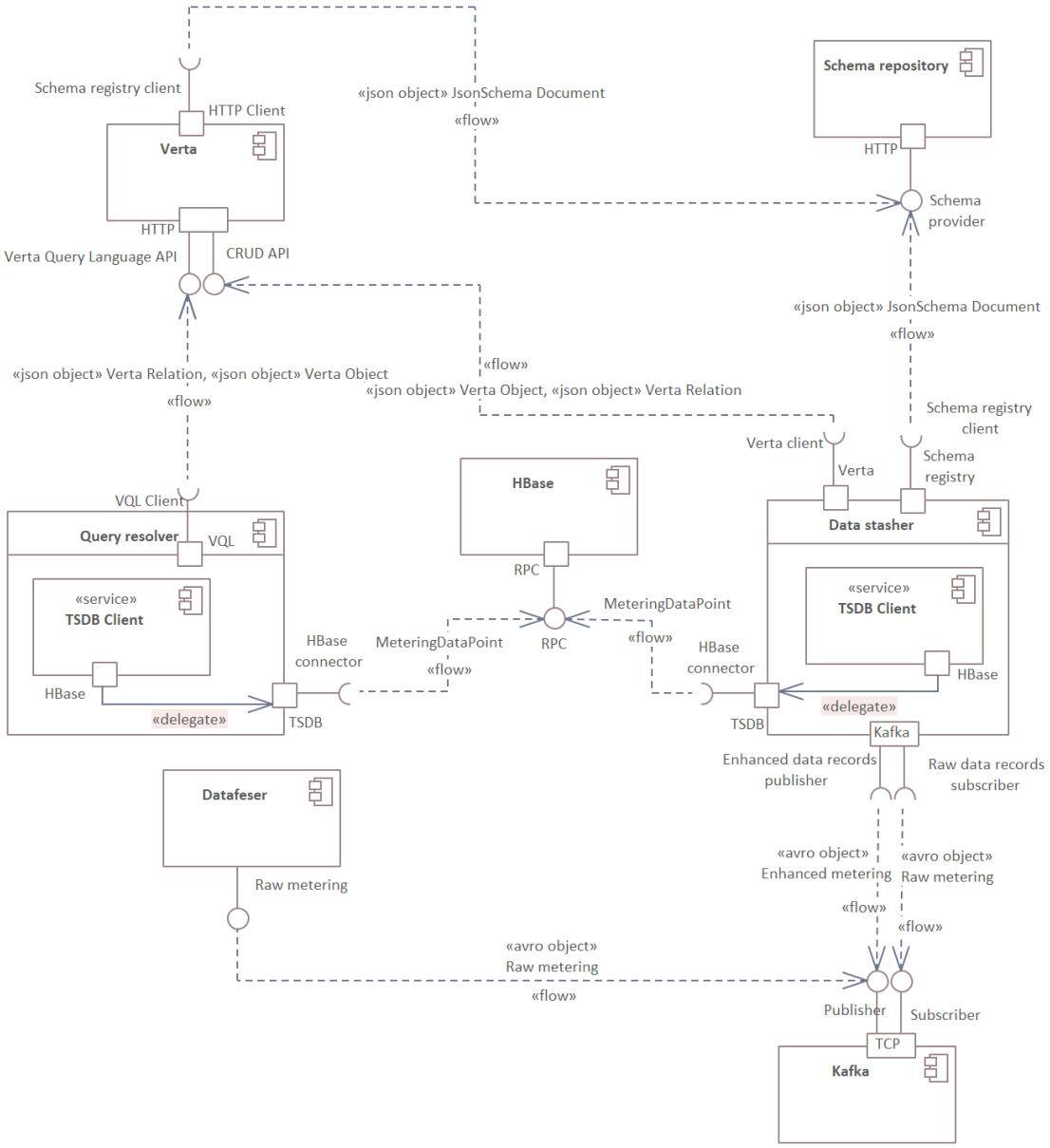
4.4 Popis implementovaných mikroslužeb

Implementace byla provedena podle návrhu architektury a jednotlivých komponent ze sekce 3.5 s využitím technologií popsaných v sekci 3.2. Stejně jako v návrhu jsou zde popsány v samostatných sekcích jednotlivé komponenty. U každé komponenty jsou popsány nejzásadnější části implementace, které jsou v mnohých případech podpořeny diagramy v notaci UML.

Popsány jsou také logické vazby mezi mikroslužbami. Zejména jak si mezi sebou předávají data. Shrnutí propojení jednotlivých mikroslužeb je uvedeno na obrázku 4.7. U každé vazby jsou uvedeny názvy datových objektů, které jsou přes ně posílány. Protože implementovaný sytém dodržuje mikroservisní architekturu a jednotlivé komponenty v diagramu reprezentují logické celky poskytující určitou funkcionalitu, reprezentuje každá komponenta v diagramu jednu mikroslužbu (případně jinou závislou službu). V komponentách *Query resolver* a *Data Stasher* je navíc uvedena služba TSDB Client, která obstarává komunikaci s databází a poskytuje funkcionalitu *databáze časových řad*.

4.4.1 Schema Repository

Implementace mikroslužby poskytující JSON schémata pro zbytek systému využívá pro ukládání datový engine PostgreSQL. Schémata budou ukládána v dereferencovaném formátu, tzn. ve stejném formátu, jako je využit pro generování POJOs popsaný v sekci



Obrázek 4.7. Diagram komunikace implementovaných mikroslužeb

4.2.3. Datové schéma databáze mikroslužby je velice jednoduché, obsahuje pouze jednu tabulku `documents`, ve které jsou uloženy všechna schémata ve všech verzích. Každý záznam je jednoznačně identifikovaný názvem schématu `kind` (odpovídající typu objektu, pro který je definován) a verzí `version`. Samotné schéma je pak uloženo ve sloupci `content`.

Pro operace se záznamy poskytuje *Schema Repository* HTTP REST API. Záznamy lze vkládat pouze jednotlivě s případným specifikováním konkrétní verze. Pokud dané schéma v požadované verzi není v databázi, provede se vložení, jinak je vrácena chybová hláška. Nebo lze schémata vkládat bez specifické verze, potom je schématu přiřazena nejvyšší verze daného `kind` schématu inkrementovaná o jedna. Pokud v databázi neexistuje žádný záznam pro schéma daného `kind`, je jako verze použito číslo 1. Vkládaná schémata mohou být libovolného charakteru, nijak se nekontroluje kompatibilita s před-

chozí verzí. Jediná validace, která při vkládání probíhá, je, že nové schéma splňuje draft `https://json-schema.org/draft-07/schema`.

Přístup ke schématům je umožněn přes zmíněné HTTP REST API. Lze přistupovat jak ke všem schématům v poslední verzi, tak ke konkrétnímu schématu v poslední verzi, tak ke konkrétnímu schématu v konkrétní verzi. Přístup ke všem schématům v poslední verzi je nejčastěji využívaným endpointem. Ostatní služby, které potřebují schémata, jej využívají při svém startu a při periodickém obnovování.

4.4.2 Verta

Pro přístup a správu dat uložených v grafové databázi JanusGraph (popsaná v sekci 3.2.4) je určena komponenta *Verta*. Tato komponenta implementuje základní dotazovací jazyk umožňující filtrování a vyhledávání uložených objektů. Do této databáze jsou ukládány fyzické komponenty FVE, ale i logické struktury, u kterých je vhodné sledovat jejich vazby mezi sebou nebo na části FVE. Co vše je do databáze ukládáno, je popsáno v sekci 3.4.1. Komponenta umožňuje všechny CRUD operace s objekty i hranami.

Verta je implementována v jazyce Java s využitím frameworku Spring Boot a knihovny Project Reactor [76] umožňující zpracovávat příchozí požadavky v reaktivních streamech. Stejně tak umožňuje odesílat asynchronní požadavky. Pro využívání implementovaných služeb vystavuje API komunikující protokolem HTTP. Příchozí dotazy obsluhují kontroléry, které parsují požadavky a předávají ke zpracováním servisní vrstvě. Ta vykonává samotnou logiku. V případě potřeby je tak jednoduché změnit chování jednotlivých kontrolérů.

Dále tato komponenta kontroluje ukládaná data proti JSON schématům, která získává ze *Schema Repository*. Pro zajištění co největší izolace a minimalizace závislosti na ostatních mikroslužbách jsou do paměti při startu aplikace načtena všechna schémata. Snapshot schémat je poté periodicky aktualizován. Každé schéma obsahuje vlastnosti, které daný typ objektu musí obsahovat společně s jejich typem, jakého musí být jejich hodnoty. Pro hrany navíc omezuje typy koncových a počátečních objektů. Díky této kontrole jsou data v databázi ukládána v předem definovaném a konzistentním formátu. Zároveň je tímto mechanismem zajištěna definice a následné dodržování integritních omezení.

Unikátní identifikátory objektů (unikátní identifikátory v databázi JanusGraph) jsou použity také jako cizí klíče do *databáze časových řad*. Podle těchto identifikátorů poté lze jednoznačně určit, ke kterému objektu daná časová řada patří. Unikátní identifikátor objektu je poté součástí identifikátoru každého záznamu dané časové řady jako značka `object`. Díky udržovaní historie všech změn v komponentě Verta je možné i v případě smazání objektu vyhledávat data k němu naměřená.

Aplikační rozhraní

Jak již bylo zmíněno v návrhu, implementace komponenty *Verta* poskytuje HTTP API. Přes něj lze provádět CRUD operace s vrcholy i hranami. Dále poskytuje dotazování na data pomocí Verta Query Language (VQL) popsaného níže.

Funkcionality poskytované jednotlivými endpointy přímo naplňují, nebo se podílejí na naplnění některých případů užití. Přehled endpointů namapovaných na případy užití, které naplňují, je uveden v tabulce 4.3.

Příklad příchozího požadavku pro vytvoření nového objektu je uveden v ukázce kódu 4.5. Pro vytváření hrany je dotaz velice podobný, pouze se specifikuje ID počátečního a koncového objektu (vizte ukázku kódu 4.6). Klíč `kind` z ukázky reprezentuje typ objektu. Podle něj implementace validuje objekt vůči schématu. Pod klíčem `properties`

HTTP metoda	endpoint	případ užití
POST	`/edge/create`	UC6
PUT	`/edge/invalidate/<id>`	UC7
GET	`/edge/<id>`	UC8
POST	`/vertex/create`	UC5
PUT	`/vertex/invalidate/<id>`	UC7
GET	`/vertex/<id>`	UC8
GET	`/vertex/related/<id>`	UC2
GET	`/vertex/relations/<id>`	UC2
POST	`/match`	UC2

Tabulka 4.3. Mapování endpointů aplikačního rozhraní komponenty Verta

```
POST http://<verta>/vertex/create
Content-Type: application/json

{
  "kind": "device:inverter",
  "properties": {
    "ean": "23413234545",
    "vendor": {
      "name": "GoodWe"
    },
    "externalId": "ean"
  }
}
```

Kód 4.5. Ukázka dotazu pro zaevidování nového střídače

```
POST http://<verta>/edge/create
Content-Type: application/json

{
  "kind": "relation:allocation_for",
  "sourceVertexId": "ODM4NA==",
  "targetVertexId": "NDA5NjQxMTI=",
  "properties": {
    "allocation": 40
  }
}
```

Kód 4.6. Ukázka dotazu pro zaevidování nové vazby mezi objekty

jsou uloženy všechny vlastnosti objektu, které musejí splňovat požadovanou strukturu definovanou schématem.

Způsob ukládání dat

Pro dodržení navrženého datového modelu v sekci 3.4.1 používá mikroslužba *Verta* JSON schémata, která získává z komponenty *Schema Repository*. Při startu si načte všechna schémata do paměti, vytvoří snapshot a následně je periodicky aktualizuje.

K dosažení integrity dat musí implementace tato schémata správně využívat a vyžadovat jejich využívání. Toho dosahuje pomocí validace při ukládání nových záznamů do databáze. Před každým vložením jsou objekty a hrany validovány. V případě nekonzistence nejsou data uložena a je vrácena chybová hláška popisující, kde došlo k porušení schématu.

Všechny požadavky, které do komponenty přicházejí, jsou ve formátu JSON a obsahují jednoznačný identifikátor typu objektu, resp. hrany, kterého se týkají. Tato informace je označována jako `kind`. U každého přijatého dotazu je nejprve zvalidována struktura, zda všechny vlastnosti, které jsou pro daný objekt, resp. hranu, poskytnuty, odpovídají jak datovými typy, tak samotnou hodnotou definovanému schématu pro zpracovávaný typ objektu – `kind`. Pokud neexistuje schéma pro přijatý datový typ, není do databáze vložena žádná hodnota a je vrácena chybová hláška o neexistenci definice pro požadovaný `kind`. Každý dotaz je validován podle JSON schématu, které je pro daný `kind` evidováno. Díky tom je předem jednoznačně určena struktura každého dotazu. Pro validaci JSON objektů (požadavků) oproti JSON schématům využívá implementace knihovnu Vert.x Json Schema [77].

K ukládání využívá mikroslužba v implementaci databázový engine JanusGraph, který umožňuje ke každému vrcholu přiřadit `label` a dalších n key-value vlastností (vizte 3.2.4). Stejné informace umožňuje přiřadit ke každé hraně. Všechny objekty datového modelu jsou ukládány do JanusGraph jako vrcholy a hrany grafu, proto musejí být převedeny do zmíněné struktury. Každému novému objektu přiřadí databáze JanusGraph unikátní číselný identifikátor. Pro potřeby systému je zakódován do Base64. V této formě je pak používaný jako vstup do dotazů, pro vytváření hran, invalidaci, dotazování atp. Obdobně je tomu i u hran. Těm databáze také přiřadí jednoznačný, teď už alfanumerický, identifikátor. Pro potřeby systému je stejně jako identifikátor objektů zakódován do Base64. V této formě slouží pro stejné účely jako zakódovaný identifikátor objektů.

Jako `label` se do databáze JanusGraph ukládá informace, která byla označena v návrhu datového modelu (3.4.1) jako *značka* – jednoznačný identifikátor typu objektu mající hierarchickou strukturu. Databáze umožňuje ukládat tuto informaci pouze jako datový typ string, proto není nutné implementovat žádné další mechanismy serializace a deserializace. Oproti tomu u vlastností je možné uložit do databáze jako hodnotu každého páru instanci libovolné Java třídy implementující interface Serializable. Nicméně je doporučeno využívat nativně podporovaných datových typů ,[]. Klíč každého páru musí být, stejně jako `label`, vždy datového typu string. Kvůli možnosti ukládat libovolné datové typy je pro vlastnosti implementován mechanismus serializace a deserializace založený na knihovně Jackson [78]. Stejně jako u *databáze časových řad* se i zde ukládá pro každou vlastnost její datový typ. Tomu by se zde dalo vyhnout, jelikož celá struktura je vždy uložena v JSON schématu. Ačkoliv by ji šlo ve zpětné deserializaci dat z databáze použít, protože se s každým objektem ukládá i verze schématu použitá při jeho uložení, není to nutné a zbytečně by docházelo ke zpomalení v důsledku vyhledávání správného schématu a vlastností v něm. Jelikož se všechna data validují při uložení do databáze, implementace předpokládá, že data uložená v databázi mají správný formát, tedy je stačí pouze načíst, jednotlivé vlastnosti deserializovat a sestavit tak zpětně výsledný objekt.

Každý datový typ použitý v implementaci musí implementovat interface *PropertyValue*. Ten zaručuje, že bude možné hodnotu daného typu serializovat a uložit do databáze. Zároveň pro každý datový typ existuje továrna, která ho umí vytvořit z datové repre-

zentace, ve které je uložen v databázi. Opět jde o velice podobný mechanismus, který je implementovaný v *databázi časových řad*.

K vlastnostem každého objektu a hrany jsou automaticky přidány další informace, aby bylo možné zajistit historizaci, jak bylo navrženo v sekci 3.4.1. Názvy těchto parametrů odpovídají těm uvedeným v návrhu. Jsou to `created_at` obsahující časovou značku vytvoření objektu (resp. hrany), `deleted_at` obsahující časovou značku konce platnosti objektu (resp. hrany), `deleted` obsahující informaci, zda byla u objektu (resp. hrany) nastavena časová značka určující konec platnosti.

Historie změn

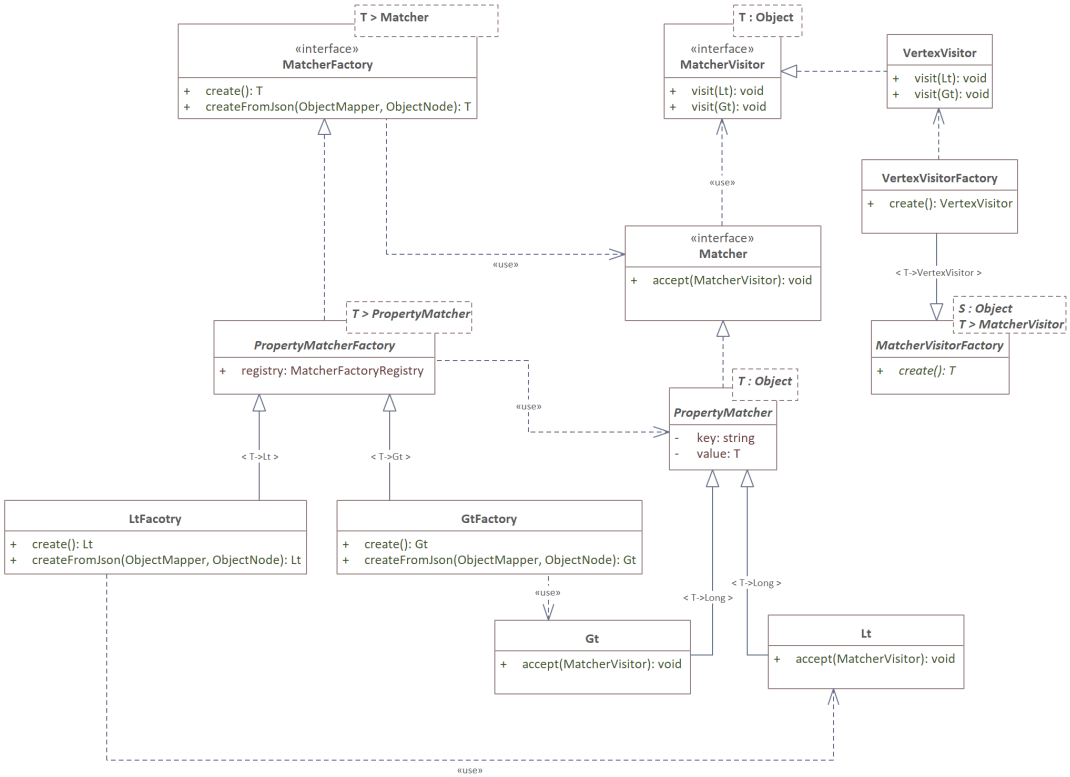
Při vytvoření je objektu i hraně automaticky přiřazena speciální vlastnost s názvem `created_at` obsahující časovou značku vzniku. Tato vlastnost slouží k vyhledávání podle času přidání, zároveň určuje, od kdy je daný objekt (případně hrana) platný. Odstranění vazeb a objektů je rozděleno do dvou úrovní. První úroveň je úplné smazání, při kterém dojde k odstranění informace z databáze a nelze k ní již nikdy přistoupit ani zjistit, že v ní byla uložena. Dojde tak k porušení historie a data jsou nenávratně ztracena. Tato operace je dostupná jako HTTP metoda DELETE. Druhou úrovní mazání je invalidace, ta je méně invazivní. Hrana či objekt nejsou smazány z databáze, ale je jim přiřazena vlastnost `deleted_at` s časovou značkou odpovídající době smazání. Invalidace je dostupná jako HTTP POST metoda. Takto lze v databázi udržovat historii změn. Zároveň je možné se dotazovat pouze na platné informace, takové, které nemají vyplněnou vlastnost `deleted_at`, ale i na informace platné v určitém časovém okamžiku v historii. V takovém případě se porovnává jak vlastnost `created_at` se začátkem vyhledávaného intervalu, tak `deleted_at` s koncem vyhledávaného intervalu. Pro zrychlení dotazování je v případě invalidace vazbě i objektu nastavena speciální, pomocná vlastnost `deleted` na hodnotu `true`. Klienti tak mohou získat snapshot stavu databáze v libovolném čase.

Dotazovací jazyk Verta Query Language

Pro dotazování nad uloženými daty poskytuje *Verta* dotazovací jazyk. Pomocí něho lze filtrovat objekty a hrany podle jejich vlastností a typu. Vyhledávat lze podle přesné shody, nebo podle toho, zda hodnota dané vlastnosti obsahuje hledaný text. Dále je možné použít pro filtrování regulární výraz. Číselné hodnoty (ale i textové, dává-li to smysl) lze filtrovat pomocí porovnávacích operátorů, *je větší než* (`gt`), *je menší než* (`lt`), případně v inkluzivních variantách. Jednotlivé dílčí dotazy je možné spojovat logickými výrazy `and` a `or`.

Dále je v dotazovacím jazyce implementováno kromě dotazování pomocí atributů tzv. traverzování. To je vhodné pro hledání objektů podle jejich vazeb, případně podle vlastností těchto vazeb, které do objektu vstupují, resp. z něj vystupují. Dotazovací jazyk je implementován tak, aby byl snadno rozšiřitelný o další funkcionality (operátory). Veškerá sémantika je řešena v jedné třídě s využitím návrhové vzoru Visitor, lze ji tedy jednoduše v případě potřeby zaměnit za jinou. Konfigurace, která třída se má používat, se provádí při spuštění aplikace. Aktuální implementace překládá všechny dotazy do jazyka Gremlin (dotazovacího jazyka pro JasnusGraph), kterým se poté provádí samotné dotazování do databáze. Ukázka diagramu tříd reflektující implementaci vybraných operátorů je uvedena na obrázku 4.8. Ve schématu jsou uvedeny pouze dva operátory `gt` a `lt`, ostatní jsou implementovány obdobně.

V kódu 4.7 je zachycen dotaz v jazyce VQL, který demonstruje použití tohoto jazyka. Tento dotaz vrátí všechny objekty, které jsou spojeny odchozí hranou s vlastností

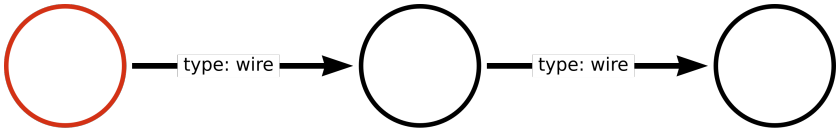


Obrázek 4.8. Diagram tříd vybraných operátorů pro vyhledávání

`type`, jejíž hodnota obsahuje řetězec `wire` s objekty, které jsou spojeny odchozí hranou s vlastností `type`, jejíž hodnota obsahuje řetězec `wire` s jiným objektem. Schematicky je dotazovaná struktura zachycena na obrázku 4.9. Dotazem by byl vrácen červený objekt (uzel), tedy ten nejvíce vlevo.

```
{
  "matcher": {
    "action": "out",
    "property": {
      "action": "containing",
      "key": "type",
      "value": "wire"
    },
    "traversal": {
      "action": "out",
      "property": {
        "action": "containing",
        "key": "type",
        "value": "wire"
      }
    }
  }
}
```

Kód 4.7. Ukázka dotazu v dotazovacím jazyku VQL



Obrázek 4.9. Schematické zobrazení hledaného vzoru dotazem 4.7

Validace při úpravách v komponentě Verta

Popis validace byl částečně zmíněn v popisu ukládání dat, neboť je s ním úzce spjat. Všechna data, která se do databáze ukládají, jsou nejprve validována, aby byl dodržen definovaný datový model z návrhu 3.4.1. Jak bylo zmíněno u ukládání dat, komponenta má v paměti vždy načtený snapshot všech JSON schémat v poslední verzi, který se průběžné aktualizuje. Jednotlivá schémata představují datové typy, které lze v komponentě *Verta* spravovat. Každý typ objektu, resp. hrany, který má být do databáze uložen, musí být definován v právě jednom schématu. Každé schéma obsahuje definice vlastností, které může daný objekt obsahovat, dále jakého typu jednotlivé vlastnosti jsou a zda jsou povinné, či nikoliv. Ukázka jednoho takového JSON schématu je uvedena v kódu 4.8. Jedná se o dereferencované schéma (popis jak tento proces je uveden v sekci 4.2.3) definující datový typ střídač (angl. inverter).

4.4.3 Data Feser

Návrh komponenty *Data Feser* byl popsán v sekci 3.5.3. Komponenta poskytuje SDK (Standard Development Kit), které lze využít pro jednoduché rozšíření implementace nových komunikačních protokolů. Pro potřeby prototypu není implementována komunikace pomocí žádného konkrétního protokolu. Všechna data, která tato komponenta získává, pocházejí ze simulátoru střídače a elektroměrů odběrných míst (vizte sekci 4.8).

Spuštění procesu odečtení (získání) dat není řízeno samotnou komponentou, ale poskytuje tuto funkcionalitu přes aplikační rozhraní. Přijímané požadavky na odečtení musejí obsahovat interní identifikátor objektu, pro který mají být data odečtena, název časové řady, do které mají být odečtená data uložena. Dále musí obsahovat typ měření, který specifikuje, jakým protokolem a jakým způsobem je možné tato data odečíst. Na základě typu měření rozhodne implementace mikroslužby *Data Feser* o tom, jak budou data odečtena.

Operace spuštění procesu odečte data a samotné vykonávání úlohy je tak od sebe logicky oddělené. O plánování, kdy (a nyní i jak) se mají která data z kterých zařízení odečíst, se starají skripty (tzv. DAG) definované v aplikaci Apache Airflow.

Na obrázku 4.10 je uveden sekvenční diagram, který zachycuje odečtení dat ze střídače (inverter). Diagram reprezentuje případ, kdy externí služba Apache Airflow spustí odečet v mikroslužbě *Data Feser*. Ta získá data ze střídače, transformuje je do unifikovaného formátu a odešle do Kafka topic `metering`, odkud si data vyzvedne pro další zpracování *Data Stasher*.

4.4.4 Data Stasher

Komponenta *Data Stasher* operuje se surovými daty, která vstupují do systému. Tato data jsou získávána např. ze senzorů, střídače nebo jiných externích zdrojů. Obecně se nemusí jednat pouze o data měření – data časových řad. Mohly by takto být přidávány do evidence také nové datové objekty, vazby atp. Nicméně pro potřeby prototypu je toto omezení dostačující. *Data Stasher* tedy zpracovává pouze data měření, resp. nové záznamy časových řad.

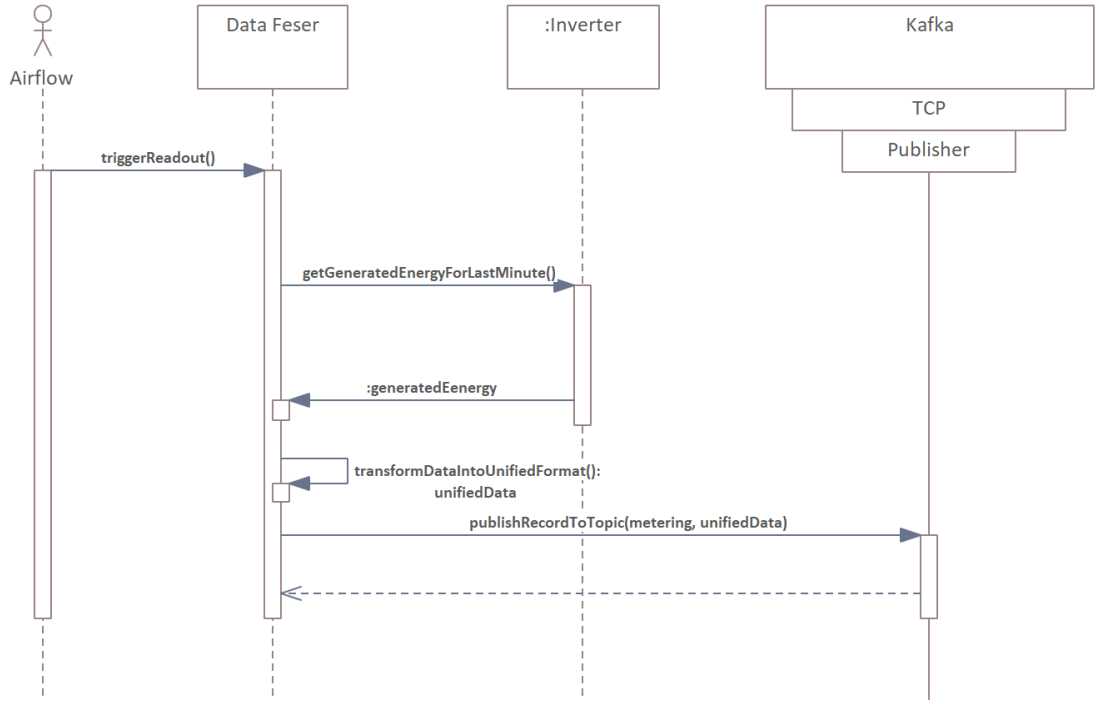
```
{
  "$id" : "inverter.schema.json",
  "kind" : "device:inverter",
  "additionalProperties" : false,
  "type" : "object",
  "properties" : {
    "ean" : {
      "kind" : "datatype:ean",
      "name" : "ean",
      "pattern" : "^([0-9]{1,19})$",
      "description" : "EAN (European Article Number)",
      "type" : "string",
      "javaType" : "eu.jehlicka.fve.repository.schema.datatype.Ean"
    },
    "vendor" : {
      "kind" : "datatype:vendor",
      "name" : "vendor",
      "description" : "Vendor",
      "additionalProperties" : false,
      "type" : "object",
      "properties" : {
        "name" : {
          "pattern" : "^([a-z]|[A-Z]){1,100}$",
          "type" : "string"
        }
      },
      "required" : [ "name" ],
      "javaType" : "eu.jehlicka.fve.repository.schema.datatype.Vendor"
    }
  }
}
```

Kód 4.8. JSON schéma datového typu střídač (angl. inverter)

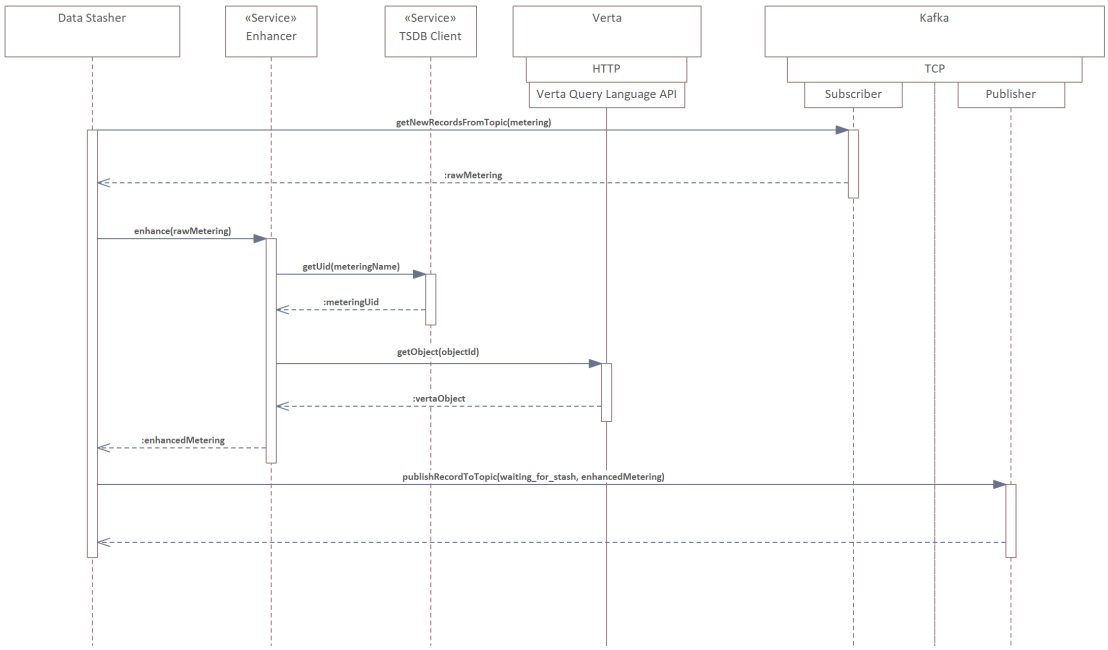
Data měření jsou do systému uložena komponentou *Data Feser* (4.4.3). Před uložením však nejsou nijak validována, proto nemusejí být ve správném formátu, dokonce nemusejí být ani validní. *Data Stasher* tato data validuje a obohacuje o další informace, které získává z ostatních komponent systému. Validace je prováděna pouze na úrovni struktury ukládaných dat. Samotné hodnoty nejsou nijak validovány. Tato funkcionalita je buďto místem pro budoucí rozvoj, nebo lze validaci dělat dodatečně pomocí automatických úloh.

Vstupní data vybírá komponenta z Kafka topicu a zpracovává je pomocí Kafka Streams (technologie popsaná v sekci 3.2.6). Zmíněná validace struktury dat je prováděna pomocí AVRO schémat. Každý topic, kterým data procházejí, má pomocí schématu definovanou obecnou strukturu dat, s nimiž pracuje.

Implementace komponenty obohacuje data o identifikátor časové řady. Jednotlivé záznamy přicházejí pouze s názvem, nikoliv s UID, časové řady, do které mají být uloženy. Mapování názvu na UID je možné získat pomocí TSDB klienta. Druhé obohacení prováděné mikroslužbou *Data Stasher* je získání všech vlastností objektu, ke kterému má být daný záznam přiřazen. Tyto informace jsou uloženy v komponentě *Verta* a je potřeba



Obrázek 4.10. Sekvenční diagram odečtu dat ze střídače



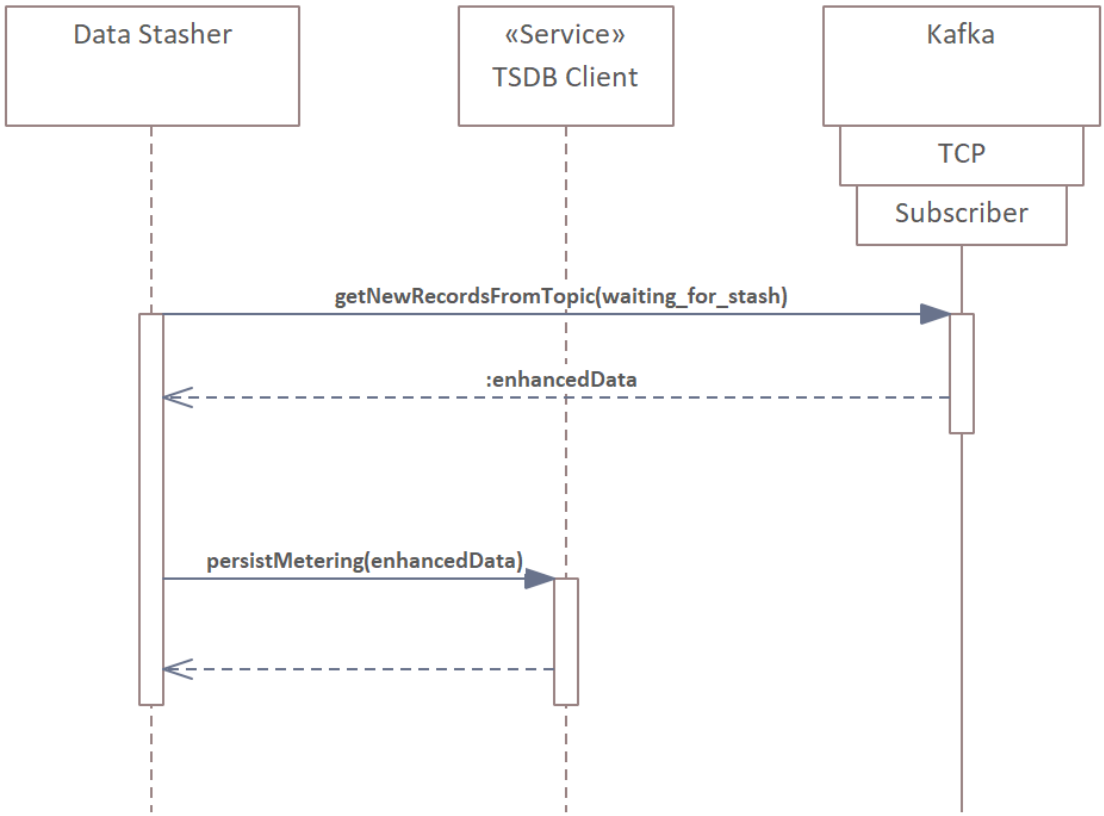
Obrázek 4.11. Sekvenční diagram obohacování dat měření

se na ně dotázat přes poskytované rozhraní. Sekvenční diagram průběhu obohacování dat je uveden na obrázku 4.11.

Po obohacení o zmíněné informace nejsou data přímo uložena do TSDB, ale jsou odeslána do Kafka topic `waiting_for_stash`. Tímto implementace demonstruje obecně rozšiřitelný koncept, ve kterém každý stream může provést pouze část obohacení, uložit data opět do Kafky, odkud je vyzvedne další stream, který data dále zpracuje. Až

nakonec, když jsou data ve formátu a kvalitě vhodné pro uložení do perzistentní databáze, ze které jsou přístupná dále v systému, je tak učiněno. Topic, ze kterého už se neprovádí žádné transformace, ale data se ukládají do perzistentního uložiště, je již zmíněný `waiting_for_stash`.

Sekvenční diagram ukládání obohacených dat je uveden na obrázku 4.12. Data jsou vyzvednuta z Kafky a uložena do TSDB pomocí TSDB klienta.



Obrázek 4.12. Sekvenční diagram ukládání obohacených dat měření

4.4.5 Query Resolver

Všechna data, která jsou v systému spravována, musejí být dostupná nejen pro další zpracování aplikacím třetích stran, ale také pro další analytické operace prováděné samotným systémem. Jelikož jsou data uložena ve více databázích a přístupná přes více komponent, je nutné přístup k nim sjednotit. Tímto sjednocujícím prvkem je právě mikroslužba *Query Resolver*. Implementace této komponenty poskytuje GraphQL API, přes které lze přistupovat jak k datům časových řad, tak k datům uloženým v grafové databázi. Tato data lze také dávat do vzájemného kontextu (spojovat pomocí unikátních identifikátorů datových objektů, které jsou v časových řadách jako značka). Zároveň je poskytován lidsky (i strojově) čitelný popis aplikačního rozhraní, ze kterého je možné generovat datové typy pro mnoho programovacích jazyků.

GraphQL schéma je definováno manuálně a pouze na obecné úrovni. Jeho součástí tak nejsou informace o konkrétních typech objektů ze *Schema Repository* nebo názvech časových řad atp. Všechny tyto parametry jsou do dotazů vkládány jako textové řetězce, a proto je potřeba je před vytvářením dotazů znát buď z externích zdrojů, případě se na ně dotázat jiným dotazem.

Pro uživatele poskytovaného API je dotazování na objekty, hrany a měření zcela transparentní, jako by vše bylo uloženo koherentně na jednom místě v systému. O zajištění tohoto chování se v implementaci starají tzv. data resolvers, základní konstrukt pro získávání dat přes GraphQL. V implementaci je použit Spring boot starter pro GraphQL [79], který přidává podporu pro práci s GraphQL, včetně zmíněných data resolvers. S využitím této podpory lze jednou anotací (`@SchemaMapping(typeName = <name>)`) definovat pro GraphQL typ třídy obsahující data resolvers pro jednotlivé položky reprezentované metodami anotované třídy. Jednotlivé metody jsou poté opatřeny anotací `@SchemaMapping(field = <name>)` určující, ke které položce daného typu patří.

Každý resolver má k dispozici kontext dotazu, zná všechny jeho parametry a jelikož se jedná o stromovou strukturu, tak i výsledky vyhledávání rodičů. Tím je efektivně zajištěno, že se implementace doptává pouze na data, která jsou v dotazu požadována. Nevýhodou této implementace, která však plyne z použité technologie, je nutnost udržovat průběžné výsledky a následně i celý výsledek v paměti. Pro složité dotazy na velké množství dat tak může být vyžadováno velké množství paměti.

Jelikož dotazovací jazyk GraphQL je obecným konceptem, který lze používat na libovolné datové schéma dostupné z libovolných datových zdrojů, nemůže nativně podporovat žádné operace závislé na dynamické struktuře dat a vše je nutné implementovat. V mikroslužbě *Query Resolver* je takovou operací např. agregace měřených hodnot v zadaných časových úsecích. Implementace umožňuje hledaný časový rozsah rozdělit do časových úseků definované délky (buckets). Na data v každém časovém úseku je aplikována agregační funkce. Dostupné jsou: aritmetický průměr, suma, maximum, minimum. Díky této implementaci lze vytvářet dotazy, které se dotazují na průměrnou či celkovou hodnotu v každých 15 minutách v zadaném časovém intervalu. Tento typ dotazů je potřebný pro rozpočet vyrobené elektrické energie. Výpočty agregací neprovádí *databáze časových řad*, ale jsou delegovány na *Query Resolver*.

Implementovaný dotazovací jazyk umožňuje používat ve všech GraphQL dotazech obecný filtr podle interního identifikátoru datového objektu. Dále je možné vyhledávat data platná v určitém časovém rozsahu. U objektů je platnost určena časovou značkou vytvoření a časovou značkou invalidování. Data v časových řadách jsou platná od doby jejich vzniku dále.

U datových objektů je možné provádět dotazy na jejich vlastnosti, vazby a sousední objekty. V traverzování, vyhledávání návazných objektů, lze k určení přechodu použít typ vazby (opět se na ni aplikuje globální filtr existence). V případě potřeby složitějšího dotazu lze využít funkcionality poskytované dotazovacím jazykem VQL mikroslužby *Verta*. Díky tomu je možné traverzovat i podle vlastností objektů a vazeb.

Na data časových řad nelze přistupovat na přímo, ale je nutné se dotazovat vždy v kontextu některého objektu. Implementace časové řady je označována jako metrika (metrics). Tento název vychází z faktu, že v implementaci obsahují časové řady nejčastěji měření nějaké veličiny. Příklad GraphQL dotazu pro získání dat o měření k určitému objektu je uveden v kódu 4.9. Dotaz obsahuje dva parametry – `ids` specifikuje identifikátory objektů, nad kterými se má dotaz provést (pro která požadujeme měření), a `metrics` specifikuje, jaká měření mají být vrácena. Samotný dotaz obsahuje filtrování objektů podle identifikátorů měření předaných jako zmíněný parametr. Pro každý objekt je vráceno interní ID a jeho typ. Tato data jsou dostupná přes mikroslužbu *Verta*. Zbytek dotazu se týká filtrování dat měření. Tyto informace jsou uloženy v jiném datovém zdroji, v *databázi časových řad*. Data každého měření jsou seskupena do 15minutových intervalů, z každého intervalu je spočtena suma, která je dotazem vrá-

cena společně s časovou značkou počátku intervalu, ze kterého byla suma vypočtena, a názvem měření.

```
query GetObjectsMetering($ids: [String], $metrics: [String!]!) {
  objects(filter: {ids: $ids}) {
    internalId
    kind
    metricsSet(ids: $metrics) {
      groupByTime(size: 15, unit: "MINUTES") {
        timestamp: key
        values {
          metering: name {
            value
          }
          sum {
            ... on NumberValue {
              value
            }
          }
        }
      }
    }
  }
}
```

Kód 4.9. GraphQL dotaz pro získání dat o měřeních k objektům

4.5 Uživatelské rozhraní

Uživatelské rozhraní (UI) bylo implementováno podle návrhu provedeného v sekci 3.6. Jsou implementovány všechny navržené obrazovky. Protože byl dodržen stanovený rozsah a funkcionality, jsou splněny všechny případy užití, ve kterých je iniciálním aktérem uživatel. Příručka pro práci s tímto rozhraním je uvedena v příloze C.

Pro implementaci je použit programovací jazyk TypeScript společně s frameworkem Angular. Dále je použita knihovna, resp. grafický vzor, CoreUI [80], který poskytuje základní kostru webové aplikace, společně s interaktivními grafickými elementy, které je možné používat. Tato volba značně zrychlila implementaci a odbourala nutnost vytváření vlastního grafického návrhu.

Všechna data, která jsou v jednotlivých obrazovkách uživatelského rozhraní zobrazována, jsou načítána z komponenty *Query Resolver*. Dotazování probíhá pomocí GraphQL endpointu. Pro TypeScript, jazyk, ve kterém je implementováno UI, existují generátory kódu, které z GraphQL schématu vygenerují datový typ. Stejným způsobem je možné vygenerovat datové typy také pro odpovědi na dotazy. Této vlastnosti je využito také v implementaci.

Pro uchování stavu aplikace a reakci na uživatelské akce je v implementaci využita knihovna NgRx [81]. Jedná se o knihovnu, která podporuje ukládání a spravování globálního stavu webové aplikace. V implementaci jsou některé části globálního stavu (např. nastavení časového rozsahu) ukládány do local storage, aby při opětovném načtení stránky s aplikací byla tato nastavení zachována.

Hlavní výhodou knihovny NgRx je možnost asynchronního čtení a modifikování globálního stavu aplikace. Modifikace jsou prováděny pomocí akcí, které je možné vyvolat z libovolné části aplikace. Reakce na ně jsou poté definovány na jednom místě. Obsluha každé akce je implementována jako funkce bez vedlejších efektů, která má na vstupu aktuální stav aplikace a na jejím výstupu je nový, modifikovaný stav. Ostatní části aplikace zůstávají bez zásahu.

Data z globálního stavu používají jednotlivé obrazovky jako zdroj informací. Reagují na změnu stavu dynamicky. Pokud dojde ke změně stavu, dojde také k překreslení dané komponenty. Veškerá interakce s aplikací je zpracovávána pomocí této knihovny.

4.6 Automatické úlohy

Implementované mikroslužby poskytují pouze obecné procesy bezprostředně spjaté s ukládáním, resp. obohacováním, dat. Nedefinují však žádné konkrétní business procesy. Ty je potřeba řídit externě. Díky této vlastnosti implementovaného systému nejsou žádná omezení, jak mohou být uložená data zpracována, resp. jaké výpočty s nimi lze provádět.

Pro definici úloh s business procesy, které budou vykonávat operace s evidovanými daty, není implementována žádná speciální komponenta, ale je použito hotové řešení Apache Airflow. Je dodržen návrh ze sekce 3.7. V Airflow je nastaven konektor na mikroslužbu *Data Feser*, pomocí kterého lze odesílat požadavky na odečtení dat u komponent FVE. V případě potřeby lze přidat i konektory na další části systému. Dále je nastaven konektor na Apache Spark pro spouštění výpočtů.

Pro potřeby prototypu jsou implementovány tři úlohy (DAGs) v Airflow. Dvě pro odečítání dat – vyrobené a spotřebované energie. A jedna pro spuštění rozpočtu vyrobené elektrické energie mezi jednotlivá odběrná místa. Samotný výpočet je poté implementovaný jako Spark Job.

Touto konfigurací je docíleno možnosti implementovat úlohy, které budou koordinovat spolupráci jednotlivých komponent, případně spouštět potřebné procesy v implementovaném systému. Úlohy je možné nakonfigurovat tak, aby byly spouštěny automaticky. Tím je naplněn funkční požadavek F6.

4.6.1 Rozpočet vyrobené elektrické energie

Rozpočet je implementován jako Apache Spark Job v jazyce Scala, který lze spouštět v clusteru zmíněné technologie. Automatické spuštění každou hodinu je nastaveno pomocí úlohy (DAG) v Apache Airflow. Je dodržen popsaný architektonický návrh, v rámci kterého jsou business výpočty odděleny od logiky mikroservis sytému. Samotný výpočet přesně koresponduje s tím, jak jej definuje novela vyhlášky o Pravidlech trhu s elektřinou [20] (příklad popsaný v sekci 2.5.2). Jeho výstupem je informace o tom, kolik energie bylo vyrobeno pro každé přidružené místo (rozdělení vyrobené energie podle alokačního klíče), dále informace o tom, kolik každé odběrné místo své alokované energie využilo (rozdíl alokované a spotřebované energie se spodní hranicí na nule), kolik elektrické energie bude každému odběrnému místu fakturováno (rozdíl spotřeby a výroby se spodní hranicí na nule) a poté souhrnná informace pro všechna místa, a to kolik z vyrobené energie bylo prodáno jako přetok do distribuční sítě.

Všechny výsledky výpočtu jsou ukládány do příslušných časových řad. Z pohledu sytému se vykonávání úlohy chová jako aplikace třetí strany a nová data ukládá do Kafky, odkud jsou systémem dále zpracována. Stejně tak přistupuje k datům. Opět je vykonávání úlohy z pohledu systému aplikace třetí strany, která k datům přistupuje

přes komponentu *Query Resolver*, která agreguje všechny datová zdroje. Díky tomu je celý výpočet zcela nezávislý na vnitřní implementaci systému, využívá pouze jeho aplikační rozhraní.

Při dotazování na data přes komponentu *Query Resolver* se projevila výhoda použité technologie GraphQL. Pro dotazování je možné z GraphQL schématu vygenerovat všechny potřebné datové struktury. A to jak pro dotazy, tak konkrétní odpovědi na ně. Tím je zajištěno typování při tvorbě dotazů, ale také při čtení jejich odpovědí. Zároveň nebyla potřeba implementovat DTOs (Data Transfer Objects) pro jednotlivé dotazy a odpovědi. K vygenerování byl v implementaci použit plugin Caliban [82] pro build tool SBT [83].

Jelikož je nutné, aby byla výpočetní úloha postupně spouštěna nad všemi daty, je součástí implementace také mechanismus, kterým výpočet pozná, do kterého časového bodu již data zpracoval a kde má nový výpočet započít. Zároveň se tímto zajistí, že stejná data nebudou zpracována vícekrát. To by teoreticky nevadilo, neboť by nemělo docházet k modifikaci již uložených informací, ale jedná se o zbytečný výpočet. Technicky je tato pomyslná zarážka výpočtu pouze záznam ve speciální časové řadě, ve které se s každým spuštěním vytvoří nový záznam určující, do kterého časového bodu byla data zpracována, resp. do kterého časového bodu byla rozpočítána vyrobená elektrická energie. V případě potřeby také lze pomocí těchto informací zjistit, kdy byl spuštěn výpočet. Tento koncept je obecný a samozřejmě jej lze aplikovat na libovolný podobný výpočet.

4.7 Lokální nasazení

Součástí implementace jsou také konfigurace pro nástroje, které podporují automatizované operace s vyvíjeným kódem, jako je automatické testování či spouštění. Jedná se především o konfigurační soubory pro virtualizaci (izolaci procesů) nástroje Docker. S jejich využitím lze pro každou mikroslužbu vytvořit vlastní oddělené běhové prostředí.

Zde se projevuje jedna z výhod monorepozitáře. Všechny potřebné konfigurace pro lokální nasazení snapshot verzí mikroslužeb je možné mít na jednom místě, a tím usnadnit celý proces.

4.7.1 Nasazení pomocí nástroje Docker

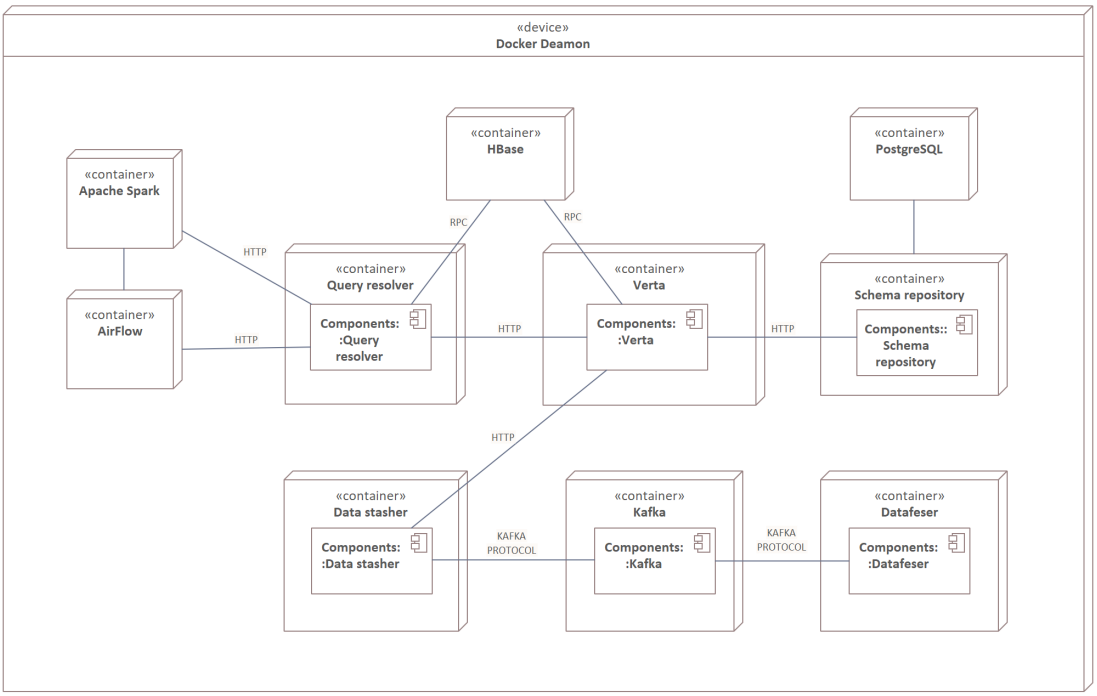
Pro potřeby vývoje a integračního testování je vhodné vytvořit prostředí, které bude spustitelné na lokálním stroji. Takové prostředí nesmí být závislé na žádném externím serveru, na kterém by běžela některá ze služeb systému (případně jiná, na kterých je systém závislý). Dále nesmí být závislé na platformě, na které bude spuštěno. Všechny změny tak půjdou při vývoji rychle nasadit do lokálního prostředí a nebude každou změnu, kterou chce vývojář nasadit a jakýmkoliv způsobem vyzkoušet její funkčnost, nahrávat na vzdálený server. V takovémto prostředí lze také provádět integrační testy, a to opět bez nutnosti vytvářet novou infrastrukturu na lokálním (či vzdáleném) stroji (resp. strojích).

K vytvoření zmíněného prostředí je v implementaci použit nástroj Docker, který izoluje jednotlivé služby do vlastních běhových prostředí, tzv. kontejnerů. Ty jsou na sobě nezávislé. Z pohledu služeb je chování stejné, jako by běžely fyzicky na jiných zařízeních.

Každá mikroslužba systému má ve svém kořenovém adresáři vlastní `Dockerfile` popisující běhové prostředí dané služby. Všechna prostředí, ve kterých je spuštěna mikroslužba implementovaná v jazyce běžícím nad JVM, využívají image `sflow-base`,

který obsahuje JDK pro Javu verze `17.0.8`. Pokud by tak např. došlo k tomu, že všechny služby budou potřebovat stejný balíček, jinou verzi JDK atp., bude zapotřebí tuto změnu provést pouze na jednom místě. Ostatní služby, které potřebují pro svůj běh jiné prostředí, využívají již hotových, veřejně dostupných images.

Schematicky je nasazení jednotlivých komponent pomocí nástroje Docker zobrazeno s využitím deployment diagramu na obrázku 4.13. Pokud je součástí některého kontejneru implementovaná komponenta, je v diagramu uvnitř tohoto kontejneru zachycena. Dále jsou součástí diagramu asociace mezi zmíněnými komponentami, případně kontejnery, pokud jejich součástí není vlastní implementace.



Obrázek 4.13. Schéma lokálního nasazení

4.7.2 Nasazení komponent systému

Samotné rozdělení mikroslužeb do separátních běhových prostředí není však jediná věc, kterou implementovaný mechanismus pro lokální nasazení obsahuje. Dále je součástí tohoto procesu trojice konfiguračních souborů pro nástroj Docker Compose [84]. Jeden pro samotné mikroslužby systému (popsané v sekci 4.4), druhý pro nasazení Apache Airflow, třetí potom pro nasazení Apache Spark.

Oproti `Dockerfiles` už nedefinují běhová prostředí, ale popisují, jak jsou jednotlivé kontejnery (služby) systému mezi sebou propojeny, dále definují jejich závislosti. Před startem každé služby tak bude zajištěno, že budou spuštěny i závislé služby (v rámci systému). V těchto konfiguračních souborech jsou také definovaná mapování adresářů jednotlivých služeb na adresáře hostitelského systému pro zajištění perzistentního uložení dat. V neposlední řadě je pak pro každou službu definováno mapování síťových portů mezi hostitelským systémem a kontejnery.

Jelikož se jedná o lokání nasazení pro potřeby vývoje a testování, jsou porty všech služeb exportovány a namapovány na hostitelský systém. Lze k nim tedy přistupovat i mimo kontejnery. Pro samotný běh v Dockeru toto však není nutné. Všechny služby

jsou propojeny v jedné lokání virtuální síti, přes kterou komunikují. Vystavení portů je zde tedy opravdu pouze pro debugging, monitorovací či testovací účely.

4.7.3 Nasazení závislých služeb

Stejně jako u samotného systému, tak i u závislých služeb, jako jsou Apache Airflow nebo Apache Spark, je vhodné, aby běžely v izolovaných kontejnerech, nezávisle na hostitelském systému. Pro většinu takovýchto služeb existuje veřejně dostupný Docker image, který stačí pouze vhodným způsobem nastavit pomocí systémových proměnných. Pro některé služby existuje ukázkový `Dockerfile` a konfigurační soubor pro Docker Compose upravitelný podle potřeb uživatele. V případě Apache HBase však neexistuje ani jeden ze zmíněných způsobů v oficiální variantě. Existuje několik komunitních pokusů, které jsou buď upraveny přímo pro konkrétní použití, nebo jsou dlouhodobě neudržované. Z tohoto důvodu je v implementaci pro lokální nasazení Apache HBase použit vlastní image, který se sestavuje přímo na lokálním stroji a není využit již některým z existujících veřejných repozitářů. Při vytváření tohoto image je použita metoda multi-stage build, při které jsou jednotlivé kroky vytváření výsledného image od sebe odděleny, výsledkem každé jedné fáze je sub-image, ze kterého si jeho následovníci vezmou (zkopírují) pouze data, která potřebují. V tomto konkrétním případě je build rozdělen do tří stages. V rámci prvního kroku je stažen Apache HBase archiv, v druhém kroku je tento stažený soubor převzat a extrahován. Ve třetím kroku je poté spuštěn samotný program. Výsledky (images) předchozích kroků jsou uloženy pro případné další použití. Pokud tedy bude probíhat build Apache HBase podruhé, nebude nutné znovu stahovat a extrahovat všechny potřebné soubory, bude stačit pouze znovu sestavit poslední (třetí) krok.

Obě závislé služby jsou nakonfigurovány tak, aby byly ve stejné lokální síti, jako jsou mikroslužby systému. Při komunikaci s nimi tak není nutné mapovat síťové porty závislých služeb na porty hostitelského systému, případně takto namapované porty využívat. Veškerou komunikaci lze provozovat přes zmíněnou lokální síť spravovanou nástrojem Docker. Výhodou tohoto použití je mimo jiné to, že není nutné znát adresy, které jsou po zapnutí přiřazeny jednotlivým službám, ale lze k nim přistupovat podle jejich názvu.

4.7.4 Spuštění lokálního nasazení

V předchozím textu o lokálním nasazení systému a závislých služeb bylo zmíněno, že je použito několik souborů nástroje Docker Compose. Jednou možností, jak spustit všechny služby, je sjednotit všechny soubory do jediného. Tím se zároveň přidá možnost definovat závislosti všech služeb (i z doposud jiných konfiguračních souborů). Nevýhodou tohoto spojení je velikost výsledného souboru. Jednalo by se o dlouhý, nepřehledný soubor.

Jelikož systém je schopen fungovat i bez závislých služeb Apache Airflow a Apache Spark, které zajišťují spouštění a vykonávání definovaných procesů (vizte 4.6), není nutné definovat závislosti jimi poskytovaných služeb na mikroslužbách systému a naopak. Proto mohou zůstat oddělené v jiných konfiguračních souborech. Tento přístup je zvolen také v implementaci.

Pro zajištění správného spuštění všech mikroslužeb systému a závislých služeb je tedy potřeba volání jednotlivých příkazů nástroje Docker Compose agregovat, v implementaci je tato agregace vytvořena pomocí Bash scriptu. Tento skript zároveň umožňuje spustit build a lokální nasazení pro dvě hostitelské architektury – x86 a ARMv8. Dále

tento skript umožňuje naplnit systém relevantními daty, např. pro demonstraci funkčnosti.

4.8 Simulace provozu

Pro validaci funkcionalit, vývoj a demonstraci je pro implementovaný systém vytvořena podpora pro simulování sdílení elektrické energie bez potřeby napojení na skutečná zařízení. Simulace zachovává všechny procesy a průchody dat jednotlivými mikroslužbami. Navíc poskytuje simulátor pro odečítání vyrobené a spotřebované energie na jednotlivých odběrných místech.

Simulátor vystavuje HTTP API, kterým poskytuje data o aktuální spotřebě, resp. výrobě, s granularitou na jednu minutu. Každý dotaz na něj tak vrací simulovaná, agregovaná data za poslední celou minutu ve watthodinách. Data o spotřebě a výrobě negeneruje náhodně, ale na základě ukázkových dat za jeden den. K těmto datům přidává náhodný šum. Hodnoty vygenerované pro stejný čas tak nejsou shodné. Se simulátorem komunikuje komponenta *Data Feser*, která provádí samotný odečet dat.

Procesy odečítání jsou poté řízeny pomocí DAG (automatických úloh) v Apache Airflow (opět stejně, jak by tomu bylo ve skutečném provozu) spouštěnými automaticky v nastavených intervalech. Každý DAG pro simulaci odečtu obsahuje operátor umožňující odesílat HTTP požadavky a je nastaven tak, aby spustil odečítání vyrobené a spotřebované elektrické energie na předem známé množině objektů (zařízení). Objekty jsou v požadavcích referencovány pomocí interních identifikátorů vytvořených při uložení do databáze. Jejich výčet je nutné do Apache Airflow definovat jako proměnnou po nahrání demonstračního modelu sdílení elektrické energie do komponenty *Verta*.

Dále je pro podporu simulovaného provozu implementována aplikace, která obsluhuje a nastavuje schéma databáze HBase pro ukládání časových řad. Zároveň ale také umožňuje naplnit systém (komponentu *Verta*) demonstračními daty. Po spuštění vytvoří objekty a vazby zachycující model sdílené FVE v bytovém domě. Jedná se o stejný model, který je uveden na obrázku 4.2. Aplikací je možné vytvořit libovolné schéma. Obsahuje funkcionalitu pro statické referencování jednotlivých objektů bez znalosti jejich interních identifikátorů, které jsou normálně potřeba pro vytvoření vazeb. Jednotlivé objekty a vazby jsou definovány pomocí JSON dokumentů. Místa, ve kterých by byl v normálním dotazu použit interní identifikátor, jsou nahrazena referencí. Aplikace si z těchto schémat vytvoří topologicky uspořádaný seznam dotazů na vytvoření vazeb a objektů. Z něj následně jednotlivé položky odesílá do komponenty `Verta`, eviduje si mapování názvu referencí na unikátní identifikátory již vytvořených objektů a vazeb. Získané identifikátory poté doplňuje na příslušná místa v dalších dotazech.

Díky tomuto návrhu lze vytvořit libovolný datový model a ten následně nahrát do systému. Obecnost této implementace je vhodná zejména pro budoucí použití, kdy je předpokládané, že budou probíhat změny v datovém modelu a poroste i jeho komplexita.

4.9 Vyhodnocení

V rámci implementace byl vytvořen rozšiřitelný prototyp systému pro podporu správy komunitní FVE. Funkcionalita implementovaného řešení má dostatečný rozsah, aby pokryla definované požadavky v sekci 2.6.1.

Systém je implementován s využitím mikroservisní architektury (nefunkční požadavek N1). Jednotlivé mikroslužby lze spustit ve více instancích, a tím systém škálovat.

Tím je naplněn nefunkční požadavek N7. Mikroslužby systému jsou realizovány v programovacím jazyce Java s využitím frameworku Spring Boot. Jelikož se jedná o populární framework s řadou rozšiřujících knihoven, které jsou přímo pro něj optimalizovány, dává tato volba prostor pro snadnou rozšiřitelnost. Knihovny pokrývají mnohé standardní problémy, proto bude další rozvoj a implementace nových funkcionalit rychlejší a pohodlnější.

Systém lze snadno napojit na externí datové zdroje, a to jak hardwarové (senzory apod.), tak softwarové (předpověď počasí apod.). K tomuto napojení poskytuje jednu rozšiřitelnou mikroslužbu – *Data Feser* 4.4.3. V případně jiné specifické potřeby lze řešení díky jeho architektuře snadno rozšířit implementací nové mikroslužby. Díky použité architektuře je také možné systém horizontálně škálovat spuštěním více instancí některých mikroslužeb. Tímto implementace naplňuje funkční požadavek F5.

Data získaná z externích zdrojů jsou komponentou *Data Stasher* uložena do databáze časových řad. Tato databáze je implementována jako Java knihovna zprostředkovávající přístup do databáze Apache HBase. Poskytuje API pro vytváření časových řad a vyhledávání časových řad podle názvu a značek. Také umožňuje do časových řad přidávat nové záznamy. V záznamech jednotlivých časových řad je možné filtrování podle časového rozsahu. Tato implementace poskytuje data pro případy užití UC3 a naplňuje funkční požadavek F3.

Mikroslužba *Verta* zprostředkovává funkcionalitu pro evidenci datových objektů a vazeb mezi nimi. Implementováno je přidávání nových objektů a vazeb (UC5 a UC6). Kromě vytváření je možné objekty a vazby také invalidovat (UC7). Dále je implementován dotazovací jazyk, který umožňuje vyhledávat datové objekty a vazby podle jejich vlastností, tedy i podle času vytvoření a invalidace (poskytuje data pro UC1 a UC2). U vyhledaných objektů je možné získat detailní informace o jejich vlastnostech a vazbách na další objekty (UC8 a UC9). Implementace mikroslužby *Verta* naplňuje funkční požadavky F1 a F2. Společně s databází časových řad tato mikroslužba naplňuje funkční požadavek F4.

Pro jednotný přístup k datům je implementována mikroslužba *Query Resolver*. Ta poskytuje GraphQL API, přes které lze provádět dotazy jak na datové objekty a jejich vazby, tak na data měření (časové řady). *Query Resolver* naplňuje nefunkční požadavek N5. Tato mikroslužba poskytuje data také pro business procesy (automatické úlohy zpracování dat).

Business procesy lze implementovat buď jako Spark Job, který je automaticky spouštěn platformou Apache Airflow, nebo přímo v této platformě. Správná integrace technologií Apache Spark a Apache Airflow do sytému naplňuje případy užití UC10, UC11 a funkční požadavek F6. Použití těchto technologií je demonstrováno na implementovaném business procesu rozpočtu elektrické energie (naplňuje případ užití UC12).

Pro celý systém byla vytvořena hierarchie Docker a Docker Compose konfiguračních souborů umožnujících nasadit všechny mikroslužby a závislé služby (jako jsou databáze) pomocí nástroje Docker (nefunkční požadavek N3). Vše je tak možné v případě potřeby jednoduše nasadit na jeden stroj (nefunkční požadavek N4) – např. pro lokální vývoj. Zároveň díky volbě jazyka Java není implementace vázána na žádný konkrétní operační systém (nefunkční požadavek N2).

I přesto, že se návrh a implementace zaměřovaly pouze na sdílení elektrické energie v bytovém domě, tedy neřešily komunitní sdílení s využitím distribuční a přenosové sítě, je možné výsledný systém použít i na evidování takového druhu sdílení. Všechny procesy rozpočítávání energie zůstávají shodné. Případné procesní změny lze díky rozšiřitelnému návrhu (automatickým úlohám) snadno modifikovat.

4.10 Budoucí rozvoj

Implementované funkcionality systému naplňují definované cíle, nicméně to neznamená, že není možné systém dále rozšiřovat. Jsou implementovány všechny hlavní funkcionality, jako je sběr dat, jejich zpracování pomocí Spark Jobs a následné zobrazení v grafickém rozhraní.

Budoucí rozvoj se tak může ubírat dvěma směry – rozšiřování business procesů, tedy implementací úloh pro zpracování dat, a vylepšením technického řešení.

Systém je připraven na integraci dalších datových zdrojů (např. měřících zařízení). Aktuálně je napojen na simulátor (vizte sekci 4.8). Systém provádí analýzu nad aktuálně měřenými daty (rozpočet vyrobené elektrické energie podle alokačního klíče). I v této části je systém připraven na rozšíření. Další rozvoj se proto může ubírat směrem přidávání nových metrik a nových způsobů analýzy dat. Systém lze chápat jako framework. Díky jeho dostatečně obecné implementaci základních mechanismů pro práci s daty je rozšiřování funkcionality snadné, často pouze pomocí implementace vhodné úlohy v Apache Airflow (resp. Apache Spark).

Příkladem business rozšíření je automatické řízení režimu FVE, kterým by systém řešil, jak má být s vyrobenou elektřinou naloženo, zda má být prodávána jako přetoky do sítě, zpracovávána či ukládána do akumulátorů. K těmto rozhodnutím by systém potřeboval napojení i na další datové zdroje. Technické řešení by bylo vhodné rozšířit o širší škálu možností pro export dat. Implementace sice poskytuje přístup ke všem informacím přes aplikační rozhraní, ale data jsou poskytována pouze ve formátu JSON. V budoucím rozvoji by bylo vhodné doplnit systém o možnost exportování dat do CSV souborů. Tím by se také rozšířila možnost jeho integrace.

Z technických vylepšení se jedná o implementaci autorizace uživatelů a mikroslužeb pro provolávání funkcionalit poskytovaných přes jednotlivá API. K tomuto účelu může být použita rozšiřující knihovna Spring Security, která je dostupná pro použitý framework Spring Boot.

Další místem vhodným pro další rozvoj je implementace administrátorského rozhraní, přes které bude možné spravovat evidované objekty a přidávat nové do systému. V implementovaném prototypu je tato funkcionalita pokryta pouze aplikačními rozhraními jednotlivých mikroslužeb, přes které lze tyto operace provádět. Tato rozhraní je také možné využít k získaní kompletního přehledu o datovém modelu. V dalším rozvoji je tak možné vizualizovat uživateli, které vazby a objekty může vytvářet společně s jejich vlastnosti a integritními omezeními.

Ačkoliv se tato práce zabývala především komunitními FVE, není nutné omezovat se pouze na ně. Obdobné mechanismy lze implementovat na téměř libovolné výrobny elektrické energie a nejen na ně. Jde především o to, s jakými externími zdroji bude systém pracovat a jak budou evidovaná data zpracovávána (pomocí Spark Jobs). Další rozvoj se tak může ubírat i tímto směrem úzce souvisejícím se vznikem energetických společenství.

Kapitola 5
Testování

Testování softwaru napomáhá k zajištění jeho kvality, lepší kvality implementace a v mnoha případech i k lepšímu pochopení kódu. Je také nutností pro udržitelnost a další rozvoj systému. V testech jsou vždy použity konstrukty implementovaného kódu, tím mohou dát případným dalším programátorům návod, jak používat některé aplikační rozhraní, metody apod.

Prvotním výstupem testování je informace, kolik testů proběhlo úspěšně a kolik neúspěšně (resp. poměr těchto dvou skupin). Tato metrika není příliš vypovídající. Například pokud bude implementován pouze jeden test, který navíc skončí úspěšně, bude úspěšnost testování stoprocentní, ale efektivně nedojde k otestování téměř žádného kódu. Z toho důvodu je nutné brát tuto metriku v širším kontextu. Zejména s procentuálním pokrytím kódu testy. Tato metrika určuje, jaká část implementace je spuštěna (tedy i otestována) v rámci testovacího procesu. Pokud je pokrytí vysoké (ideálně stoprocentní) a zároveň je procento neúspěšných testů nízké (ideálně nulové), je možné se domnívat, že implementace je řádně otestována. Samozřejmě záleží na kvalitě jednotlivých testů. Zda testují extrémní případy (edge cases) atp., to už z těchto metrik vyčíst nelze.

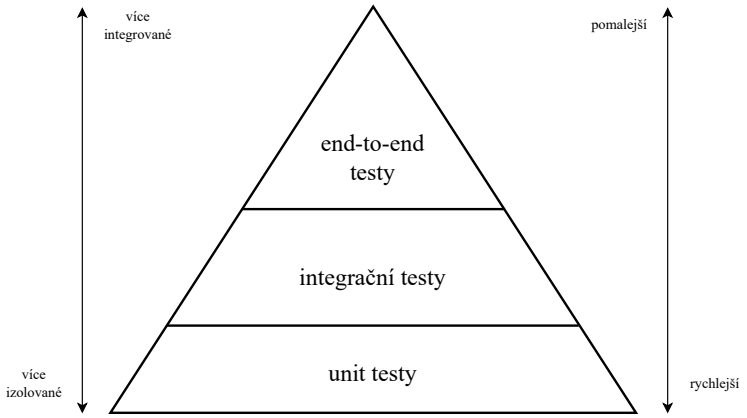
Testy není nutné psát až jako doplněk k určité implementaci. Lze je také použít jako nástroj pro přemýšlení nad strukturou budoucí implementace, tedy psát nejprve testy a až poté samotnou logiku. Tímto přístupem je zaručeno, že implementovaný kód bude dostatečně abstraktní, aby jej bylo možné otestovat. Tento způsob vývoje je nazýván jako Test Driven Development (Testy Řízený Vývoj), zkráceně TDD. V implementaci systému tento přístup nebyl použit. Testy byly doplňovány souběžně s implementací.

V této kapitole jsou popsány základní kategorie testů, jejich charakteristiky a využití. Kapitola dále popisuje technologie vhodné pro testování a testovací frameworky. Dále jsou uvedeny způsoby testování použité u jednotlivých mikroslužeb, případně jiných částí implementace, popsané v kapitole 4. V závěru kapitoly je zhodnoceno otestování implementace.

5.1 Testovací pyramida

Automatizované testy jsou jedním ze základů pro udržitelný software. Jedním z konceptů, které zachycují pojetí automatizovaných testů, je testovací pyramida. S tímto konceptem přišel Mike Cohn ve své knize *Succeeding with Agile* [85] [86]. Tato pyramida se skládá ze 3 stupňů – Unit Tests, Service Tests a E2E Tests – schematicky je tato hierarchie zachycena na obrázku 5.1. Každý stupeň reprezentuje jeden typ testů. Níže položené testy jsou rychlejší a pokrývají malý kus kódu (malý funkční celek). Oproti tomu testy v horní části pyramidy jsou více komplexní a jejich vykonání trvá déle (testují komunikaci několika funkčních celků, a to i mezi více procesy).

Koncept testovací pyramidy se může zdát pro moderní přístupy příliš jednoduchý a může být považován za zavádějící. Naproti tomu je na tomto konceptu dobré stavět základy při implementaci jednotlivých testů. Při implementaci testů je vhodné řídit se



Obrázek 5.1. Testovací pyramida [86]

jednoduchými pravidly, které testovací pyramida definuje. Jedním z těchto pravidel je psát testy s odlišnou granularitou. Druhým pravidlem je psát tolik testů, na jaké úrovni pyramidy se zrovna nacházíme. Čím výše na pyramidě se programátor nachází, tím méně testů by mělo vzniknout. Pro účely vytvořeného systému byly UI testy nahrazeny E2E testy, které více zapadají do konceptu testovaného systému. [87]

Testy přibývají v různých částech projektu různou rychlostí. V raných částech jsou psány pouze unit testy, následně jsou přidány integrační testy a konečně E2E testy. Závislost času vývoje projektu a počtu testů je zobrazen na obrázku 5.2. Tento postupný růst počtů testů v jednotlivých kategoriích je podmíněn jejich určením. Je samozřejmé, že v počátku vývoje, kdy neexistuje více služeb, které by spolu mohly komunikovat, a není implementována téměř žádná business logika, nemají E2E žádný smysl, a proto nejsou používány.

Podle typu testu se také liší rychlost jeho vykonávání. Není dogma, že vykonávání integračního testu musí nutně trvat delší dobu než vykonávání unit testu a obdobě u ostatních. Nicméně opět je toto chování spíše přímým důsledkem toho, jak dané typy testů fungují a co vše potřebují pro svůj běh. Signifikantním zpomalením integračních testů je vytváření testovacího prostředí, spouštění závislých služeb apod.

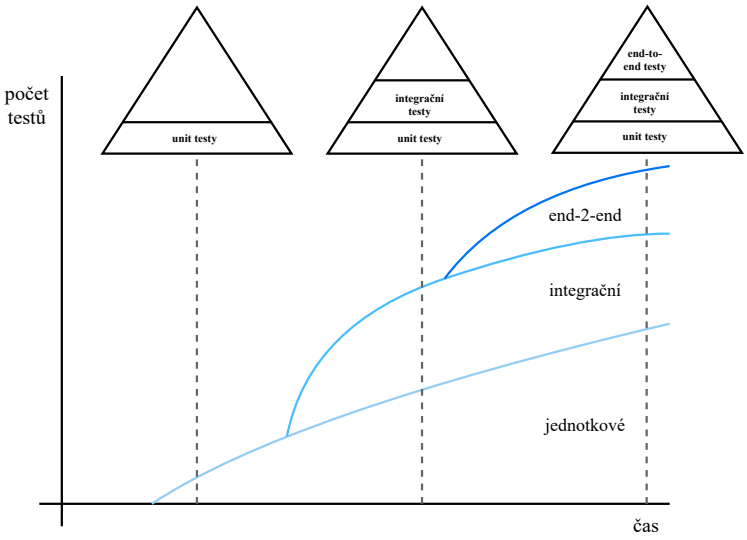
Od toho se také odvíjí jejich počty. Jednodušších a rychlejších testů je více, složitějších a komplexnějších méně. Přesně tak, jak definuje testovací pyramida. Řádové rychlosti vykonání testů podle typu jsou uvedeny v tabulce 5.1.

typ testu	řádová rychlost
unit	0.01–0.001 s
integration	1 s
end-to-end	10 s

Tabulka 5.1. Řádová rychlost vykonávání testů podle typu

5.2 Jednotkové testy

Jednotkovými testy (unit tests) se v testování softwaru rozumí takové testy, které testují funkčnost izolovaných jednotek systému. Účelem těchto testů je zjistit, zda části softwaru pracují dle požadavků. Cílí především na testování algoritmického chování, vstupu a výstupu metod (funkcí) atp. Typicky se tento typ testů spouští před integrováním



Obrázek 5.2. Závislost počtu testů na době od začátku projektu a jejich uvedení do kontextu testovací pyramidy [87]

nové funkcionality do zbytku systému. Jsou často spouštěny samotným programátorem, který tak snadno může ověřit, že jím implementovaná část softwaru odpovídá požadavkům. Stejně tak se tyto testy zpravidla spouští automaticky v tzv. CI/CD pipelines při vytvoření nového commitu v Git repozitáři [88].

Při testování oddělených částí kódu jsou všechna data potřebná k vykonání testované operace tzv. mokována – nahrazena statickým vstupem. Jde například o volání síťových služeb, v jednotkových testech se nevolá skutečná služba, ale testovanému kódu se pošle jako vstupní parametru statická odpověď, která však odpovídá požadavkům testu (nemusí být vždy validní). Mockováním se nahrazují technologické závislosti testovaného kódu a tím dochází k jeho izolaci. Vykonávání testu je díky tomu deterministické a nezávislé na jiných částech implementace. Proto také lze snadno identifikovat příčinu případného selhání testu.

Kromě testování scénářů, které mají skončit úspěchem (např. testování pouze s validními vstupy), tzv. happy paths, je také vhodné testovat chování kódu při použití neočekávaných vstupů, tzv. unhappy paths. Častým místem, kde je vhodné využít testování unhappy paths, jsou uživatelské vstupy. Tímto testováním lze odhalit další chyby a předejít případným problémům, kdy např. testovaný algoritmus nedetekuje nesprávná data a bude pro ně vracet nesprávné výsledky. [89]

Jednotkové testy by měly být spouštěny vždy, když je provedena změna v kódu (např. nový commit v Git repozitáři). V případě nové funkcionality se tím ověří správnost implementace z pohledu business požadavků, naopak v případě opravy již existující části kódu se těmito testy kontroluje, zda nedošlo k narušení již implementované funkcionality a to nejen v místě změny, ale celém softwaru. Tomuto případu použití jednotkových testů se říká regresní testování. [90]

5.3 Integrační testy

Integrační testování se oproti předchozímu, jednotkovému testování, zabývá ověřováním propojení komponent. Testuje především komunikaci a výměnu dat mezi izolovanými částmi kódu. Cílem tohoto testování je identifikovat problémy v propojení více komponent, případném mapování výstupů jedné komponenty na vstupy komponenty druhé.

Pokud na sobě závisí více komponent, je možné testovat celou sérii komunikací mezi několika komponentami [91].

Za integrační testování lze považovat i testování aplikačního rozhraní systému. Testuje se, zda příchozí požadavky na dané endpoints (u HTTP API jsou to URL adresy, resp. jejich části) jsou obsluhovány správnými komponentami. Zda tyto komponenty správně reagují na nesprávné vstupy a zda vrací očekávaná chybová hlášení nebo požadovaná data. I v integračním testování je tedy vhodné testovat jak happy, tak unhappy paths.

Tímto způsobem tak lze otestovat, zda implementované aplikační rozhraní dodržuje definovanou strukturu a ostatní aplikace s ním mohou bez problému pracovat. V případě více verzí aplikačního rozhraní lze takto například otestovat zpětnou kompatibilitu apod.

Cílem integračních testů je tak zachytit případné chyby v komunikaci izolovaných částí systémů. Integrační testy kontrolují, aby jednotlivé mikroslužby dodržovaly definovaná rozhraní. Těchto vlastností lze využít při regresním testování, ve kterém je testováno, že nová (nebo upravená) funkcionalita nenarušila původní strukturu (a tím i funkčnost) komunikace komponent, např. u aplikačního rozhraní. [91]

5.4 End-to-End testy

End-To-End (případně End2End nebo E2E, česky také koncové testy) zachycují všechny aspekty systému od začátku do konce. Tyto testy simulují scénáře reálného průchodu systémem. Pro jejich vykonání je potřeba nasazení kompletního systému. V případech, které to umožňují, je možné provádět testy pouze nad určitým segmentem systému. Těmito testy je možné odhalovat chyby při průchodu uživatelským rozhraním, úzká hrdla toku dat v systému atp. [86]. Dále je tyto testy možné využívat pro kontrolu splnění (business) požadavků [86].

U moderních systémů skládajících se z desítek (i stovek) komponent, které zajišťují celkovou funkčnost, je testování (resp. kontrola) komplexních procesů, které vyžadují vzájemnou komunikaci několika komponent, náročným úkolem. Integračními testy lze otestovat jednotlivé komunikace mezi mikroslužbami. Nelze však testovat celý business proces vyžadující interakci několika částí systému. Do jisté míry tedy mohou E2E testy zastupovat ty integrační. Důležité je zmínit, že se nejedná o jejich náhradu. V E2E je složitější odhalit skutečného původce případné chyby. Stejně tak často nelze otestovat extrémní případy (edge cases) všech interesovaných částí procesu. Tyto případy mohou být v řetězu komunikace odfiltrovány a nejsou použity pro další komunikaci. V případě změny implementace se však může stát, že dříve filtrované edge cases se použijí pro komunikaci, a tím dojde k selhání testu. Hledání původce tohoto problému pak může být složité. Takovéto chyby by byly integračními testy odhaleny, protože se zaměřují pouze na jedno (malou skupinu) rozhraní, na kterém jsou schopny podchytit všechny možné (i extrémní) případy.

E2E testování je prováděno s využitím takových částí systému, které systém vystavuje jako vstupní body pro interakci s uživatelem nebo jiným systémem. Může se jednat o grafické uživatelské rozhraní, ale také o API [86]. Testy vykonávané nad těmito částmi systému jsou komplexní a testují ucelené scénáře simulující uživatelské (klientské) průchody.

K podpoře E2E testování existuje řada nástrojů. Pro testování webových rozhraní se jedná např. o Selenium [92]. Aplikační rozhraní lze testovat např. pomocí platformy

Postman [93]. Pomocí těchto nástrojů pro E2E je možné otestovat, zda případy užití odpovídají skutečnému, implementovanému, chování systému.

5.5 Technologie pro testování

K podpoře jednotlivých testovacích metod existují technologie a frameworky. V kapitole 2 v sekci 3.3, zabývající se výběrem implementačního jazyka a frameworku, byl zvolen Spring Boot jako hlavní framework použitý pro implementaci všech komponent. S ohledem na tuto skutečnost byly pro testování vybrány takové technologie, které jsou tímto frameworkem podporovány a je jim umožněno integrování do celého ekosystému [94]. Zároveň jsou tyto technologie používané i mimo Spring Boot. Každé technologii se věnuje jedna podsekce.

5.5.1 JUnit

Obecně lze provádět jednotkové testování bez využití knihoven. Existuje však několik frameworků, které psaní jednotkových testů usnadňují a poskytují další analytické nástroje, jako procentuální pokrytí kódu testy apod. Dále poskytují možnost porovnávání očekávaných a skutečných hodnot nejrůznějších datových typů v programátorsky přívětivém DSL (Domain Specific Language). Oproti normálním asertacím poskytne takovéto porovnání v případě selhání, mimo jiné, informaci o očekávaném a skutečném vstupu. Jedním z frameworků pro jednotkové testování je JUnit [95].

JUnit je open-source framework pro jednotkové testování určený pro jazyk Java [95]. Pro stejný jazyk, ve kterém jsou implementovány všechny komponenty systému. JUnit patří do rodiny frameworků určených k podpoře tvorby jednotkových testů, známé jako xUnit [96].

Zmíněný framework podporuje vytváření testů, které označuje jako *test suites* (případy testování), pomocí anotací metod (`@Test`), které obsahují samotnou logiku testů. Dále poskytuje sadu metod určených k validaci výstupu testované funkcionality oproti předpokládanému výsledku (zmíněné asertace). Další výhodou využití JUnit je možnost implementované testy automaticky spouštět s pomocí JUnit Platform (jedna z částí JUnit frameworku). JUnit Platform automaticky vyhledá testy (anotované metody). Vyhledané testy umožňuje filtrovat podle názvu či balíčku (package) a následně je spouštět [95].

Kromě anotace metod, které obsahují testy, obsahuje JUnit i další anotace. Pomocí nich je možné definovat chování, které se má provést před vykonáváním všech testů (`@BeforeAll`), před vykonáním každého z testů (`@BeforeEach`). Stejně tak je možné anotovat metodu, která bude obsahovat kód, který se vykoná po všech testech (`@AfterAll`), nebo který se vykoná po každém testu (`@AfterEach`). Tyto operace jsou vhodné např. pro nastavení připojení do databáze a její naplnění daty. Po testech je takto možné smazat všechna data, která byla vytvořena pro potřeby testů (nebo během nich) a nadále již nejsou k užitku.

Díky těmto vlastnostem je možné psát jednotkové testy efektivněji. Programátor není nucen řešit problémy porovnávání výsledků, spouštění testů apod., ale může veškerou svoji pozornost věnovat implementaci logiky testovacích scénářů.

5.5.2 Mockito

V popisu kategorií testů bylo několikrát zmíněno, že není nutné vždy vytvářet celý kontext programu pro otestování pouze izolované části funkcionality. Některé komponenty, které nejsou v testu přímo využívány, lze simulovat, a tím naplnit závislostní

předpoklady testované komponenty. Simulování komponent bez znalosti jejich vnitřní implementace a struktury lze dosáhnout v jazyce Java pomocí poskytovaného Reflection API. Prostřednictvím tohoto API se pro účel testování definuje na potřebné metody jejich návratová hodnota bez návaznosti na vnitřní implementaci.

I zde, stejně jako pro jednotkové testování, existuje několik frameworků, které umožňují jednoduché simulování objektů bez znalosti zmíněného Reflection API. Jedním z nich je Mockito [97]. Jedná se o open-source framework pro vytváření simulovaných objektů určených pro použití v testování kódu napsaného v jazyce Java. Napomáhá izolovat jednotlivé funkční celky od závislostí na zbytku kódu. Simulované objekty lze definovat buď pomocí poskytovaného API, nebo lze využít anotace, s jejichž pomocí lze dosáhnout lepší čitelnosti kódu.

V simulování objektů se používají dvě základní techniky. V jedné se simuluje chování objektu, resp. jeho metod a vlastností. Tato akce se nazývá stubbing [97]. V případě jejího využití není volána skutečná implementace objektu, vše je zpracováváno simulovaným objektem. Příklad vytvoření takto simulovaného objektu je uveden v kódu 5.1. V něm je simulován objekt `Dog`, v rámci kterého namísto volání implementace metody `sound()` je simulovaným objektem vrácena hodnota `wuf`. Na simulovaném objektu lze také ověřovat, které operace s ním byly provedeny, které metody byly kolikrát zavolány apod. Ve stejné ukázce kódu je ověřeno, že metoda `sound()` byla zavolána právě jednou.

```
Dog dog = mock(Dog.class);
when(dog.sound()).thenReturn("wuf"); //stubbing
System.out.println(dog.sound());     //following prints "wuf"
verify(dog, times(1)).sound();
```

Kód 5.1. Ukázka simulace objektu ve frameworku Mockito

Druhou technikou je využívaní tzv. spies. V jejich případě není nahrazován celý objekt simulací, ale pozoruje se volání skutečných implementací. Sledovaný objekt je tak obalený dekorátorem, díky kterému lze ověřovat např. kolikrát byla určitá metoda zavolána, zda vůbec došlo k interakci se sledovaným objektem apod. Využití této techniky však nijak nekoliduje se simulací chování [97]. Lze tedy definovat objekty, které z části využívají reálnou implementaci a z části je jejich chování simulované. Příklad toho, jak lze sledování objektů využít, je uveden v kódu 5.2. Stejně jako v příkladu 5.1 je zde využit objekt `Dog`, ve kterém se sleduje počet volání jeho metody `sound()`.

```
Dog dog = new Dog();
Dog spy = spy(Dog.class);
verify(dog, times(0)).sound();
spy.sound();
verify(spy, times(1)).sound();
```

Kód 5.2. Ukázka sledování interakce s objektem ve frameworku Mockito

5.5.3 Cucumber

Cucumber [98] je open-source testovací framework, který podporuje Behavior Driven Development (BDD) – psaní testů bez nutnosti technické znalosti testované problematiky.

BDD je proces vývoje softwaru, jehož cílem je vyplnit mezeru mezi technickou a business vrstvou projektu [99]. Na základně business požadavků jsou specifikovány scénáře, které by měl implementovaný kód splňovat. Scénáře jsou popsané v lidsky čitelné podobě bez technických detailů. Díky tomu mohou testy psát i netechničtí uživatelé. Formát zápisu scénářů je závislý na konkrétním testovacím frameworku.

V konečném důsledku lze tedy konstatovat, že BDD vývoj softwaru je obdobou TDD, pouze s tím rozdílem, že definování testů, resp. testovacích scénářů, je prováděno v lidsky čitelné podobě a usnadňuje tak jejich tvorbu pro netechnické uživatele.

Definování business testovacích scénářů má i další výhody v komerčním vývoji softwaru. Při vytváření softwaru s využitím BDD se nejprve definují scénáře, které odsouhlasí zákazník a které budou následně předmětem akceptačního testování. Jak bylo zmíněno, tyto testy se definují v lidsky srozumitelném formátu, proto není nutné, aby zákazník měl technickou znalost dané problematiky.

Ve frameworku Cucumber se pro definici scénářů používají klíčová slova *Given, When, Then*, která reprezentují, ve stejném pořadí, vstupní podmínky, akci, výsledek po provedení akce. Příklad takto definovaného scénáře je uveden v kódu 5.3. Scénář definuje stažení souboru `test.txt` z Minio serveru.

```
Feature: Download a file from Minio
  Scenario: Download an existing file
    Given I have a file "test.txt" in the bucket "my-bucket"
    When I download the file "test.txt" from the bucket "my-bucket"
    Then I should receive the contents of the file as an InputStream
```

Kód 5.3. Ukázka testovacího scénáře ve frameworku Cucumber

Na základně této business definice testu se následně implementuje logika nutná pro vykonání testu, která je závislá na implementaci softwaru. Ve frameworku Cucumber se pro každý krok, resp. každé využití klíčového slova *Given, When* nebo *Then*, definuje metoda (implementace). Definice se provádějí pomocí anotací. Pro každý test vždy existuje kontext, ve kterém je možné udržovat informace. Kontextem je vždy instance třídy obsahující anotované metody. Není tedy nutné nijak kontext předávat, stačí pouze pracovat s třídními proměnnými. Implementace pro ukázkový scénář uvedený v kódu 5.3 je zobrazena v kódu 5.4.

5.5.4 Postman

Oproti předchozím popsaným technologiím pro testování není Postman [93] pouze knihovnou, která poskytuje rozšiřující možnosti pro programátora, ale sama o sobě neposkytuje žádnou funkcionalitu. Postman je komplexní nástroj pro práci s aplikačními rozhraními. Umožňuje vytvářet zprávy (dotazy) pro několik aplikačních protokolů, např. MQTT či HTTP. Zároveň umožňuje jednotlivé dotazy seskupovat do kolekcí. Lze ho využít jak pro navrhování aplikačních rozhraní (kontraktů mezi službami), tak pro testování. Dotazy v kolekcích je možné automaticky spouštět a jejich výsledky porovnávat s očekávanými.

Dále Postman umožňuje pro tyto kolekce vytvářet mock API servery. To jsou taková aplikační rozhraní, která nemají implementovanou žádnou logiku a pouze odpovídají na dotazy předem definovanými odpověďmi. Tento přístup je vhodný pro testování, v rámci kterého není potřeba mít spuštěny všechny závislé služby. Ty jsou nahrazeny právě těmito jednoduchými implementacemi. Testovaná komponenta tak získá potřebná data, ale není nutné kvůli tomu startovat mnoho aplikací. Testy jsou poté rychlejší a eliminují se chyby jiných komponent.

V testování systémů najde tato technologie místo zejména pro testování kontraktů – ověřování struktury dotazů, resp. odpovědí, případně jako simulátor některých komponent, resp. jejich HTTP API. Postman lze použít také jako framework pro vykonávání E2E testů.

```
@Given("I have a file {string} in the bucket {string}")
public void iHaveAFileInTheBucket(String fileName, String bucketName)
                                                        throws Exception {
    String contents = "Hello, world!";
    InputStream stream = new ByteArrayInputStream(
        contents.getBytes(StandardCharsets.UTF_8)
    );

    when(minioClient.getObject(any(GetObjectArgs.class)))
        .thenReturn(
            new GetObjectResponse(
                Headers.of(Map.of()),
                bucketName,
                "region",
                "object",
                stream
                )
        );

    this.bucketName = bucketName;
    this.fileName = fileName;
}

@When("I download the file {string} from the bucket {string}")
public void iDownloadTheFileFromTheBucket(String fileName,
                                          String bucketName) {
    try {
        this.inputStream = minioService.getFile(bucketName,fileName);
    } catch (Exception e) {
        e.printStackTrace();
    }
}

@Then("I should receive the contents of the file as an InputStream")
public void iShouldReceiveTheContentsOfTheFileAsAnInputStream()
                                            throws IOException {
    Assertions.assertEquals(
        "Hello, world!", convertInputStreamToString(this.inputStream)
    );
}
```

Kód 5.4. Ukázka implementace testovacího scénáře ve frameworku Cucumber z kódu 5.3

5.6 Testování implementace

V této sekci jsou diskutovány jednotlivé přístupy k testování mikroslužeb a jiných částí implementace popsaných v kapitole 4. K testování jsou použity technologie popsané v sekci 5.5 a na základě vhodnosti jsou pro jednotlivé části implementace vytvořeny jednotkové (5.2) a integrační (5.3) testy. Pro uživatelské rozrhaní byl vytvořen příklad psaní end-to-end (5.4) testům. Pro každou mikroslužbu je vygenerován report pokrytí

kódu testy. Pro lepší přehlednost jsou z reportů vyjmuty třídy, které neposkytují žádnou logiku a nedává smysl je testovat. Jedná se především o výjimky, DTOs (Data Trasfare Objects), třídy zajišťující konfiguraci atp. Obrázky jednotlivých reportů jsou uvedeny v příloze D.

5.6.1 Databáze časových řad

Ačkoliv databáze časových řad není implementována jako samostatná komponenta, ale jde o knihovnu, kterou používají některé mikroslužby, je jednou ze zásadních částí systému. Proto je vhodné ji řádně otestovat. Pro databázi jsou napsány jak jednotkové, tak integrační testy. Jednotkové testy pokrývají základní typový systém databáze. Testují serializaci a deserializaci různých datových typů. Dále je jednotkovými testy pokryta implementace rowkey (popsaná v sekci 4.3.1). Testována je mimo serializace a deserializace klíče také jeho struktura. Testy ověřují, zda implementace správně dodržuje navržené pořadí jednotlivých částí klíče, jejich délku atp.

Implementace databáze časových řad je úzce spjata s databázovým enginem, do kterého ukládá data. Ačkoliv jednotkové testy pokrývají funkcionalitu tvorby dotazů do HBase, resp. testují vstupní regulární výrazy pro vyhledávání, pro zlepšení odhalování chyb jsou implementovány také integrační testy, které testují veškeré funkcionality proti běžící instanci HBase. Před spuštěním testů je tak nutné mít tuto instanci lokálně nasazenou.

Integrační testy pokrývají vkládání záznamů do databáze a základní dotazy. Aby bylo tyto testy možné rozlišit od jednotkových, jsou všechny třídy, které je obsahují, označeny anotací `@Tag('integration')`. Díky těmto anotacím lze oddělit spouštění obou druhů testů.

Před každým testováním dotazů je nejprve databáze uvedena do předem definovaného konzistentního stavu. S každým testem se nevytváří nová instance HBase, protože její nastartování je časově náročné. Před každým testem jsou tak nejprve smazány všechny tabulky, následně jsou tabulky znovu vytvořeny a je do nich vložen testovací dataset, který nemusí být pro všechny testy stejný. Toto chování je zajištěno pomocí dvou abstraktních tříd. První `AbstractTSDBTest` pouze zajišťuje testům, které ji rozšiřují, jednoduché napojení do testovací instanci HBase. Druhá abstraktní třída `WithPreparedSeries`, která dědí od `AbstractTSDBTest`, zajišťuje pro třídy, které ji rozšiřují, automatické inicializování schématu databáze společně s nahráním testovacích časových řad před každým testem.

Dotazování do databáze je otestováno jak na triviálních, tak i komplexních případech, ve kterých databáze obsahuje několik časových řad, jejichž záznamy se překrývají, záznamy jedné časové řady nejsou uloženy bezprostředně za sebou (případ popsaný v implementaci a zobrazený v tabulce 4.1). Ukázka integračního testu pro testování, zda je horní mez časového úseku vyhledávání exkluzivní, je uvedena v kódu 5.5 (test předpokládá iniciální nastavení databáze, které je prováděno vždy před každým testem – náleží třídě, která rozšiřuje `WithPreparedSeries`).

Implementace databáze časových řad obsahuje neotestované části, které jsou zpravidla spjaty s neočekávaným průchodem systému. Nejsou otestovány všechny výjimky atp. Dosavadní pokrytí se zaměřuje především na happy path. Testování ostatních cest průchodu systémem je ponecháno na budoucím rozvoji.

```
@Test
public void Upperbound_In_Timerange_Should_Be_Exclusive() {
    var query = GetDataPoints.builder()
            .context(context)
            .table(context.tsdbConfiguration().getTableMetering())
            .input(DataPointsRange.builder()
                    .metering(metering1.metering())
                    .tags(metering1.tags())
                    .from(Timestamp.builder().unix(101L).build())
                    .to(Timestamp.builder().unix(120L).build())
                    .build()
            )
            .build();

    var res = context.client().execute(query);
    assertEquals(2, res.size());
    assertEquals(series1.get(1), res.get(0));
    assertEquals(series1.get(2), res.get(1));
}
```

Kód 5.5. Test vyhledávání v časové řadě

5.6.2 Schema Repository

Schema Repository poskytuje JSON schemata pro ostatní mikroservisy přes HTTP REST API. Funkcionality, jejichž testování dává smysl jsou především následující dvě – správná struktura dotazů a ukládání dat do databáze.

Obě zmíněné funkcionality jsou otestovány pomocí Spring Boot testů, které zajišťují načtení celého aplikačního kontextu, včetně zpřístupnění kontrolérů obsluhující dotazy pro HTTP komunikaci. Samotné testování tak probíhá odesíláním HTTP požadavků pomocí klienta určeného pro testování. Takto jsou otestovány funkcionality dotazování na schéma, na schéma v určené verzi a na všechna schémata v aktuální verzi. Před každým testem se vždy uvede databáze do známého stavu. Smaže se vše uložené a nahrají se do ní předem známá množina schémat v několika verzích.

Testování funkcionality vkládání nových schémat, resp. nových verzí již evidovaných schémat, se provádí obdobným způsobem. Odesílají se HTTP požadavky na příslušné endpoints, které jsou obsluhování kontrolérem. Následně je zpětně ověřeno, zda byla data do databáze uložena správně.

Všechny popsané testy spadají do kategorie integračních. Testují strukturu dotazů, tedy poskytované API, společně s napojením na databázový stroj.

5.6.3 Verta

Hlavní funkcionalitou mikroslužby *Verta* je zprostředkovávat přístup do grafové databáze přes poskytované HTTP API a udržovat integritu ukládaných dat. Pro dotazování poskytuje Verta specifický dotazovací jazyk, které bylo nutné otestovat.

Mikroslužba Veta obsahuje sadu jednotkových testů. Část testů je zaměřena na otestování parsování dotazovacího jazyka. Testy jsou pokryty všechny základní operace a i jejich skládání. Příklad jednoho takového testu je uveden v kódu 5.6. Další jednotkové testy jsou zaměřeny na přidružené funkcionality, jako jsou konvertory identifikátorů. Pro implementaci všech jednotkových testů byl použit framework JUnit.

Pro zlepšení a větší udržitelnost kódu, je mikroslužba Verta opatřena integračními testy. Jelikož je celá implementována ve frameworku Spring Boot, který poskytuje nástroje pro snadné využití dependency injection, je nutné v testech využívat podpůrný testovací modul Spring Boot Test, který umožňuje injektování speciálních, testovacích nebo simulovaných, instancí (resp. implementací) komponent kódu. Pro integrační testy byla vytvořena abstraktní třída `AbstractVertaTest`, která nahrazuje všechny potřebné závislosti jejich simulacemi. Zároveň poskytuje in memory verzi grafové databáze JanusGraph. Integrační testy tak mohou komunikovat přes stejné databázové API (Apache Gremlin), jako komunikuje mikroslužba při normálním běhu.

Pomocí integračních testů je ověřováno, zda aplikační rozhraní dodržuje stanovenou strukturu. Tedy jestli je struktura dotazů a odpovědí v souladu s předpokladem. Díky tomu lze snadno odhalit případnou změnu, která by mohla zapříčinit přerušení komunikace ostatních komponent s komponentou Verta, a tím narušit fungování celého systému. Dále integrační testy testují samotnou komunikaci s databázovým enginem JanusGraph (jeho in memory verzi). Testování této interakce je prováděno přes několik vrstev a tím testy pokrývají širší spektrum funkcionality. Kromě samotného ukládání, mazání a úpravy vrcholů a hran v databázi, je testována validace objektů a hran, parsování dotazů a validace. Implementace totiž nesmí umožnit uložení nevalidních dat.

5.6.4 Data Feser

Implementace této mikroslužby není rozsáhlá, proto ani množství testů není velké. Zároveň jednotlivé testy netestují příliš složitou logiku. Proto byl pro testování této mikroslužby použit kromě JUnit a Mockito využit framework Cucumber. Tento framework byl zvolen pouze zde jako technologie pro otestování, která bude v případě pozitivního ohlasu použita i v dalších částech systému.

Pro testované části mikroslužby *Data Feser* jsou vytvořeny `.feature` soubory obsahující popis scénářů v lidsky čitelné podobě. Scénáře jsou rozděleny podle poskytovaných služeb. Je otestován konektor na generátor simulující data ze střídače a odběrných míst a odesílání odečtených do Kafky. Dále byl otestován výběr implementace získávání dat podle typu obdrženého v požadavku.

5.6.5 Data Stasher

Hlavní technologií použitou v komponentě *Data Stasher* je Kafka Streams. Pro možnost testování této technologie jednotkovými testy je poskytována knihovna umožňující simulovat vše potřebné, aby nebylo nutné spouštěn Kafka cluster. Pro testování tak lze vytvořit v paměti všechny potřebné objekty, které budou simulovat činnost zúčastněných komponent. Výhodou toho přístupu je možnost testovat kód bez velkého výpočetního výkonu a oproštění od konfigurace dalších služeb. Zároveň lze snadněji lokalizovat případnou chybu, než v testech s využitím koncové technologie. Samozřejmě takové testy jsou pro komplexní otestování funkcionality také nutné, stejně jako u ostatních komponent implementovaného systému.

Zmíněnou simulací v paměti, lze jednoduše definovat celý mechanismus obohacování dat, popsaný v sekci 4.4.4 bez složitého simulování a abstrahování celé funkcionality do složitého modelu. Za pomoci knihovnou poskytovaného API je sestavena totožná topologie Kafka streams, jako v normálním kódu. Při normálním spuštění je topologie složena plně automaticky, každý Kafka stream je poskytován Spring Boot aplikačnímu kontextu jako Bean, pro testovací účely je potřeba builder topologie předat ručně. Samotné sestavení poté již probíhá s využitím kódu aplikace, tedy výsledná testovaná

```
@Test
public void Match_complex_query() {
    Matcher matcher = Or.builder().subMatchers(
        List.of(
            ExactMatch.builder()
                .label("testLabel")
                .build(),
            Containing.builder()
                .key("property_name")
                .value("evic")
                .build(),
            And.builder().subMatchers(List.of(
                    StartingWith.builder()
                        .key("nested")
                        .value("s")
                        .build(),
                    Lt.builder()
                        .key("number_prop")
                        .value(42L)
                        .build()
            )).build()
        )
    ).build();
    matcher.accept(visitor);
    var reference = __.or(
        __.hasLabel("testLabel"),
        __.has("property_name", containing("evic")),
        __.and(
            __.has("nested", startingWith("s")),
            __.has("number_prop", lt(42L))
        )
    );
    assert visitor.getBuffer().equals(List.of(reference));
}
```

Kód 5.6. Test překladu z dotazovacího jazyka komponenty Verta

topologie je vždy totožná. Výhodou tohoto přístupu je absolutní kontrola nad sestavenou strukturou. Je možné vytvořit konektory na jednotlivé topics apod.

Do topologie sestavené pro testovací účely jsou manuálně poslána data a na konci transformace jsou validována oproti očekávaným výstupům. Dále je testována detekce nevalidních zpráv a chybové stavy, které z takové situace mohou plynout. Pro ověřování správnosti výsledků a definice testů je použit framework JUnit.

5.6.6 Query Resolver

Implementace mikroslužby *Query Resolver* je velice úzce spjata s datovými zdroji, ze kterých získává data. Její hlavní úlohou je spojovat data z těchto zdrojů a poskytovat je přes GraphQL API v uceleném formátu. Dává tedy smysl otestovat, zda poskytované rozhraní využívá správné metody pro získávání dat, případně poskytuje všechny očekávané funkcionality. Samotná logika metod provádějících dotazování je úzce spjata

s datovými zdroji a bez nich (nebo jejich simulací) ji nelze jednoduše otestovat. Pro testování jsou implementace těchto metod nahrazeny jednoduššími, které neimplementují všechny funkcionality filtrování a vyhledávání. Toto nahrazení lze provést velice snadno, protože všechny konektory na datové zdroje dodržují kontrakty pomocí interfaces.

Testy jsou implementovány s využití knihovny Spring Boot GraphQL Test [100]. Tato knihovna přidává rozšíření pro Spring Boot Test framework. S její pomocí je možné simulovat dotazování na GraphQL endpoint pomocí testovacího HTTP klienta. Všechny takto odeslané dotazy jsou vyhodnocovány stejným kódem, který je použit při provozu aplikace. Mockovány jsou pouze klienti časové databáze a komponenty *Verta*. Simulovaný obsah těchto databází se vytvoří vždy před spuštěním testů.

U testovacích dotazů se ověřuje struktura odpovědí a hodnoty ve vybraných položkách. Příklad testu, který využívá zmíněnou knihovnu pro testování dotazu na všechny existující objekty, je uveden v kódu 5.7.

```
@Test
void Query_all_objects() {
    WebTestClient client =
            MockMvcWebTestClient.bindToApplicationContext(context)
                    .configureClient()
                    .baseUrl("/graphql")
                    .build();

    HttpGraphQlTester tester = HttpGraphQlTester.create(client);
    tester.document("""
                    query GetAllObjects {
                      objects {
                        internalId
                      }
                    }""")
            .execute()
            .path("data.objects[0:].internalId").entityList(String.class)
            .contains("NQ==", "NAQ==", "NDQ==");
}
```

Kód 5.7. Test GraphQL dotazu na všechny existující objekty

5.7 Shrnutí

V této kapitole zaměřené na testování byly popsány základní přístupy k testování softwaru, které byly následně aplikovány na implementovaný systém. U každé komponenty byl popsán způsob, jakým bylo přistoupeno k jejímu otestování. V případech, které byly posouzeny za přínosné, byly uvedeny také příklady testovacího kódu. Tyto ukázky obsahují využití jak podpůrných testovacích knihoven, tak vlastních implementací. V každé logické části implementace (mikroslužbě, knihovně) je vždy otestována minimálně její základní funkcionalita. Dále byl pro každou komponentu vytvořen report pokrytí testů. Jednotlivé reporty jsou uvedeny v příloze D.

Testování webového rozhraní uživatelem nebylo provedeno. Tento druh testování je vhodný pro komplexnější rozhraní, která poskytují více funkcionalit. Dále bylo bráno v potaz, že tento druh testování je velice časově náročný a implementovaný systém je

pouze prototyp, jehož funkcionality se můžou velice rychle měnit. Navíc hlavním výstupem implementace je backendová část systému, na kterou je v celé práci kladen větší důraz. Nicméně je velice pravděpodobné, že by takové testování odhalilo nedostatky, které v uživatelském rozhraním mohou být. Aby nezůstalo uživatelské rozhraní neotestované, byly vytvořeny základní průchody, které testují správné načtení stránky. Testy jsou implementovány pomocí frameworku Selenium.

Při testování mikroslužeb byl kladen větší důraz na integrační testování oproti jednotkovému. Zejména kvůli tomu, že implementace jednotlivých komponent nejsou striktně algoritmického charakteru, ale pouze přispívají svou částí do obecného procesu. Zároveň je tato volba kompromisem mezi dobou vytváření testů a jejich přínosem. Pomocí integračních testů je totiž zároveň pokryta většina funkcionalit, které by mohly být otestovány jednotkovými testy. Případnou chybu implementace tak integrační testy odhalí stejně, pouze její lokalizace je složitější. Chybný integrační test je totiž často až důsledkem jiné chyby, která přímo nesouvisí s logikou, na kterou test cílí. Zároveň jedna chyba může ovlivnit průběh několika integračních testů. U jednotkových testů je proces lokalizace chyby přímočarý, protože každý jednotkový test testuje (nebo by měl testovat) jednu ucelenou funkcionalitu, kterou nelze dále dělit. Pokud některý z takových testů selže, je zřejmá chybná část implementace. Jednotkové testy mohou lépe otestovat extrémní případy, které jsou ve většině implementací problematické.

Dalším důvodem, proč je kladen větší důraz na integrační testy, je mikroservisní architektura celého systému. V ní je důležité zajistit, že všechny služby budou poskytovat svá API v očekávaném formátu, aby byla možná jejich vzájemná interakce. Chyba ve struktuře aplikačního rozhraní by totiž zapříčinila nefunkčnost dané komponenty, což by mohlo mít za následek nefunkčnost celého systému.

Kapitola 6
Závěr

Tato práce se zabývala návrhem a vytvořením funkčního prototypu systému s webovým uživatelským rozhraním, který slouží pro podporu procesů spojených se správou chytrých fotovoltaických elektráren.

V první části práce byla představena analýza domény fotovoltaických elektráren. V této analýze bylo přistoupeno k popisu technických částí, komponent a způsobu provozu fotovoltaických elektráren. Součástí byla také analýza množiny dat, která lze z fotovoltaických elektráren získat, a bylo popsáno, jak lze takováto data využít. Dále byl uveden aktuální stav energetického zákona a vyhlášky o Pravidlech trhu s elektřinou. V rámci tohoto kontextu byly diskutovány také připravované (nebo krátce účinné) novely, které ovlivňují možnosti provozu tzv. komunitní energetiky. Příklad skupinového sdílení elektrické energie z fotovoltaické elektrárny byl demonstrován na bytovém domě. Stejná instance bytového domu byla použita při popisu návrhu a implementace.

Na základě podkladů z analýzy domény fotovoltaických elektráren a komunitní energetiky byly vypracovány požadavky na systém a případy užití. Jimi byl řízen následný návrh. Hlavními požadavky byly: evidence datových objektů a jejich vazeb, evidence dat časových řad, škálovatelnost řešení, univerzálnost sběru dat z různých zdrojů (komponent). Dalším důležitým požadavkem byla možnost definovat business procesy, které budou evidovaná data automaticky asynchronně zpracovávat.

Druhá část práce se zabývala návrhem prototypu systému. V ní byly nejprve popsány technologie, které byly na základě získaných požadavků posouzeny jako potenciálně vhodné pro další návrh a implementaci. Na základě definovaných požadavků a vybraných technologií byly zvoleny programovací jazyk Java a framework Spring Boot (resp. Spring Cloud) jako vhodné prostředky pro realizaci. Návrh se také věnoval datovému modelu. Mimo jiné zde byl navržen rozšiřitelný mechanismus definic datového schématu, který je nezávislý na databázovém enginu a umožňuje definovat základní integritní omezení pomocí deklarativního jazyka JSON Schema.

Jako databáze pro ukládání datových objektů a jejich závislostí, které mají být v systému evidovány, byl zvolen JanusGraph s využitím Apache HBase. Pro databáze časových řad byla z důvodu udržení konzistence technologií zvolena databáze Apache HBase. Tato volba vedla k nutnosti detailního návrhu ukládání jednotlivých položek (záznamů časových řad) tak, aby v nich bylo možné vyhledávat.

Poslední část návrhu byla zaměřena na samotnou architekturu systému. Případy užití byly rozděleny do kategorií vyžadujících stejnou funkcionalitu. Na základě tohoto rozdělení bylo navrženo několik mikroslužeb. Každá navržená mikroslužba poskytuje jednu specifickou funkcionalitu. Všechny mikroslužby dohromady tvoří funkční celek, který poskytuje univerzální prostředí pro práci s daty v analyzovaném rozsahu. Pro samotné business procesy bylo navrženo použití technologie Apache Spark a Apache Airflow. Tím bylo dosaženo oddělení technické a business funkcionality, což zajišťuje univerzálnost návrhu systému.

Implementace prototypu systému, která byla představena ve třetí části práce, se striktně držela provedeného návrhu. Byla vytvořena JSON schémata definující datový

model sdílení elektrické energie v bytovém domě. Implementací navržených mikroslužeb byl vytvořen univerzálně rozšiřitelný prototyp systému pro správu komunitních fotovoltaických elektráren, který umožňuje sběr dat z několika zdrojů a jejich následné automatické zpracování pomocí Spark Jobs (příp. Airflow DAG). Implementován byl také modelový algoritmus pro automatické rozpočítávání vyrobené elektrické energie v rámci skupiny sdílení. V závěru popisu implementace bylo provedeno celkové zhodnocení splnění analyzovaných požadavků. Implementace plně pokrývá všechny funkční i nefunkční požadavky, stejně jako plně pokrývá případy užití. Dále byla pro implementovaný prototyp systému vytvořena možnost kompletního nasazení pomocí nástroje Docker, a to včetně všech závislých služeb.

Čtvrtá část práce se zabývala popisem metod testování, včetně popisu testování samotné implementace. Pro všechny implementované mikroslužby byly napsány integrační a jednotkové testy. Tyto testy mohou sloužit kromě validace implementace také k odhalování chyb při dalším rozvoji nebo změnách. Případně mohou být využity jako vstupní bod pro programátory, kteří by chtěli implementaci rozšířit.

Praktickým výstupem práce je prototyp systému pro podporu správy chytrých fotovoltaických elektráren. Díky univerzálnosti jeho implementace je možné integrovat do prototypu různé datové zdroje (ať už hardwarové, jako jsou senzory, čidla apod., tak softwarové, jako je např. předpověď počasí). Všechna tato data umožňuje prototyp systému evidovat a provádět nad nimi automatické úlohy. Zároveň poskytuje všechna evidovaná data přes GraphQL API. Z tohoto zdroje konzumuje informace pro zobrazení i grafické rozhraní.

Výsledný prototyp systému je otestovaný na simulovaných datech a připraven pro pilotní běh. Před uvedením do produkčního provozu je nutné přizpůsobit uživatelské rozhraní pro práci se systémem pro koncového uživatele. Zároveň je nutné, aby byly zohledněny některé zmíněné podněty budoucího rozvoje.

Literatura

[1] Česko. *§ 7 odst. 2 zákona č. 17/1992 Sb., o životním prostředí - znění od 1. 7. 2017* [online]. [vid. 28. 3. 2023]. Dostupné na https://www.zakonyprolidi.cz/cs/1992-17#p7-2.

[2] Haselhuhn, Ralf a Petr Maule. *Fotovoltaické systémy: energetická příručka pro elektrikáře, techniky, instalatéry, projektanty, architekty, inženýry, energetiky, manažery, stavitele, studenty, učitele, ostatní odborné a profesní soukromé nebo veřejné instituce a zájemce o fotovoltaický obor a energetickou nezávislost*. Plzeň: Česká fotovoltaická asociace, 2017. ISBN 978-80-906281-5-1.

[3] Zilvar, Jiří. *Střešní fotovoltaika – jak funguje a co od ní očekávat?* [online]. [vid. 3. 4. 2023]. Dostupné na https://oze.tzb-info.cz/fotovoltaika/22067-stresni-fotovoltaika-jak-funguje-a-co-od-ni-ocekavat.

[4] Zilvar, Jiří. *Hybridní fotovoltaická elektrárna s domácí baterií* [online]. [vid. 3. 4. 2023]. Dostupné na https://oze.tzb-info.cz/fotovoltaika/20041-hybridni-fotovoltaicka-elektrarna-s-domaci-baterii.

[5] Zilvar, Jiří. *Ostrovní dům s fotovoltaikou a baterií* [online]. [vid. 3. 4. 2023]. Dostupné na https://oze.tzb-info.cz/fotovoltaika/20042-ostrovni-dum-s-fotovoltaikou-a-baterii.

[6] Bechník, Bronislav. *Nejpoužívanější pojmy ve fotovoltaice* [online]. [vid. 3. 4. 2023]. Dostupné na https://oze.tzb-info.cz/fotovoltaika/11772-nejpouzivanejsi-pojmy-ve-fotovoltaice.

[7] ČEZ, a.s. *Jak se starat o solární panely, aby byl jejich výkon co největší?* [online]. [vid. 3. 4. 2023]. Dostupné na https://www.cez.cz/cs/clanky/fotovoltaika/jak-se-starat-o-solarni-panely-aby-byl-jejich-vykon-co-nejvetsi-174016.

[8] Czech RE Agency, o.p.s. *Fotovoltaický střídač - účinnost není vše* [online]. [vid. 3. 4. 2023]. Dostupné na https://oze.tzb-info.cz/fotovoltaika/5571-fotovoltaicky-stridac-ucinnost-neni-vse.

[9] Lázoková, Eva. *Symetrický, nebo asymetrický střídač pro fotovoltaiku?* [online]. [vid. 1. 9. 2023]. Dostupné na https://www.woltair.cz/blog/fotovoltaika/symetricky-nebo-asymetricky-stridac-pro-fotovoltaiku.

[10] Frank Bold Advokáti, s.r.o. *Sdílení energií v Česku je tu. Jak bude po schválení Lex OZE 2 v praxi fungovat?* [online]. [vid. 10. 12. 2023]. Dostupné na https://www.fbadvokati.cz/cs/clanky/9399-sdileni-energii-v-cesku-je-tu-jak-bude-po-schvaleni-lex-oze-2-v-praxi-fungovat.

[11] Šuvarský, Jaroslav. *Měření po fázích je pro vlastníky malých fotovoltaik problém. Jak mu předejít?* [online]. [vid. 3. 4. 2023]. Dostupné na https://oze.tzb-info.cz/fotovoltaika/16878-mereni-po-fazich-je-pro-vlastniky-malych-fotovoltaik-problem-jak-mu-predejit.

[12] GoodWe Technologies Co., Ltd. *GoodWe API introduction*. [vid. 25. 3. 2023]. Dostupné na https://community.goodwe.com/solution/PV%20System%20Solution/API-introduction.

[13] Growatt New Energy Technology Co., Ltd. *Growatt API Interface Documentation* [documentation]. [vid. 25. 3. 2023]. Dostupné na https://growatt.pl/wp-content/uploads/2020/01/Growatt-Server-API-Guide.pdf.

[14] SOLAR controls s.r.o. *Wattrouter - základní popis funkce* [online]. [vid. 4. 4. 2023]. Dostupné na https://solarcontrols.cz/cz/wattrouter_function.html.

[15] Mujgoš, Michal. *Baterie pro fotovoltaiku: Kompletní průvodce pro 2023* [online]. [vid. 4. 4. 2023]. Dostupné na https://evolty.cz/fve/baterie-k-fotovoltaice/.

[16] Hladík, Richard. *Výkupní cena elektřiny z fotovoltaických elektráren 2023* [online]. [vid. 28. 3. 2023]. Dostupné na https://evolty.cz/fve/vykupni-cena-elektriny-z-fve/.

[17] Česko. *Zákon č. 458/2000 Sb., o podmínkách podnikání a o výkonu státní správy v energetických odvětvích a o změně některých zákonů (energetický zákon)* [online]. [vid. 28. 3. 2023]. Dostupné na https://www.zakonyprolidi.cz/cs/2000-458.

[18] Česko. *Zákon č. 19/2023 Sb., zákon, kterým se mění zákon č. 458/2000 Sb., o podmínkách podnikání a o výkonu státní správy v energetických odvětvích a o změně některých zákonů (energetický zákon), ve znění pozdějších předpisů, a další související zákony*. [vid. 28. 3. 2023]. Dostupné na https://www.zakonyprolidi.cz/cs/2023-19.

[19] Česko. *§ 103 odst. 1 písm. e) zákona č. 183/2006 Sb., o územním plánování a stavebním řádu (stavební zákon) - znění od 24. 1. 2023* [online]. [vid. 28. 3. 2023]. Dostupné na https://www.zakonyprolidi.cz/cs/2006-183#p103-1-e.

[20] Česko. *Vyhláška č. 404/2022 Sb., kterou se mění vyhláška č. 408/2015 Sb., o Pravidlech trhu s elektřinou, ve znění pozdějších předpisů* [online]. [vid. 28. 3. 2023]. Dostupné na https://www.zakonyprolidi.cz/cs/2015-408.

[21] Česko. *§ 27e odst. 1 zákona č. 458/2000 Sb., o podmínkách podnikání a o výkonu státní správy v energetických odvětvích a o změně některých zákonů (energetický zákon) - znění od 1. 7. 2024* [online]. [vid. 10. 12. 2023]. Dostupné na https://www.zakonyprolidi.cz/cs/2000-458#p27e-1.

[22] Česko. *Návrh zákona, kterým se mění zákon č. 458/2000 Sb., o podmínkách podnikání a o výkonu státní správy v energetických odvětvích a o změně některých zákonů (energetický zákon), ve znění pozdějších předpisů, a další související zákony* [software]. [vid. 20. 10. 2023]. Dostupné na https://odok.cz/portal/veklep/material/KORNCVVBQQNX/.

[23] Michalčáková, Anna. *Lex OZE 3: zákon, který může ušetřit miliardy a umožní rychlejší rozvoj obnovitelných zdrojů* [software]. [vid. 20. 10. 2023]. Dostupné na https://frankbold.org/zpravodaj/kategorie/aktualne/lex-oze-3-zakon-ktery-muze-usetrit-miliardy-a-umozni-rychlejsi-rozvoj-obnovitelnych-zdroju.

[24] Energetický regulační úřad. *Výroba elektřiny v bytovém domě a její rozdělení mezi jednotky od roku 2023* [online]. [vid. 28. 3. 2023]. Dostupné na https://www.eru.cz/vyroba-elektriny-v-bytovem-

dome-jeji-rozdeleni-mezi-jednotky-od-roku-2023 ? fbclid = IwAR1KSkoLpJ5DSTtI2yr2Fg__0e9HSJGBVV-A3J5kdRRprMN3wZMZjuNbg6I.

[25] Ministerstvo průmyslu a obchodu. *Komunitní energetika* [online]. [vid. 10. 12. 2023]. Dostupné na https://www.energiezamene.cz/komunitni-energetika.

[26] *Homeassistant* [software]. [vid. 1. 4. 2023]. Dostupné na https://www.home-assistant.io.

[27] OpenEMS Association e.V. *OpenEMS* [software]. [vid. 1. 4. 2023]. Dostupné na https://openems.io/.

[28] Majíčková, Gabriela. *SunDayGate* [software]. [vid. 1. 4. 2023]. Dostupné na https://www.sundaygate.cz.

[29] Domysobe.cz. *Domy sobě* [online]. [vid. 10. 12. 2023]. Dostupné na https://www.domysobe.cz/komunitni-energetika.

[30] Pedersen, Torben Bach. Multidimensional Modeling. In: LING LIU a M. TAMER ÖZSU, editoři. *Encyclopedia of Database Systems*. Boston, MA: Springer US, 2009. s. 1777–1784. ISBN 978-0-387-39940-9. Dostupné na DOI 10.1007/978-0-387-39940-9_229. Dostupné na https://doi.org/10.1007/978-0-387-39940-9_229.

[31] James Lewis, Martin Fowler. *Microservices* [online]. [vid. 10. 12. 2023]. Dostupné na https://martinfowler.com/articles/microservices.html.

[32] The OpenTelemetry Authors. *OpenTelemetry* [software]. [vid. 28. 3. 2023]. Dostupné na https://opentelemetry.io.

[33] Richardson, Chris. *Pattern: Service registry* [online]. [vid. 10. 12. 2023]. Dostupné na https://microservices.io/patterns/service-registry.html.

[34] The Apache Software Foundation. *Apache Hadoop* [software]. [vid. 20. 4. 2023]. Dostupné na https://hadoop.apache.org.

[35] Apache Software Foundation. *HDFS Design*. [vid. 20. 4. 2023]. Dostupné na https://hadoop.apache.org/docs/stable/hadoop-project-dist/hadoop-hdfs/HdfsDesign.html.

[36] Apache Software Foundation. *MapReduce tutorial*. [vid. 20. 4. 2023]. Dostupné na https://hadoop.apache.org/docs/stable/hadoop-mapreduce-client/hadoop-mapreduce-client-core/MapReduceTutorial.html.

[37] Salloum, Salman, Ruslan Dautov, Xiaojun Chen, Patrick Xiaogang Peng a Joshua Zhexue Huang. Big data analytics on Apache Spark. *International Journal of Data Science and Analytics*. Nov, 2016, ročník 1, č. 3, s. 145–164. ISSN 2364-4168. Dostupné na DOI 10.1007/s41060-016-0027-9.

[38] The Apache Software Foundation. *RDD Programming Guide*. [vid. 3. 4. 2023]. Dostupné na https://spark.apache.org/docs/latest/rdd-programming-guide.html.

[39] Armbrust, Michael, Reynold S. Xin, Cheng Lian, Yin Huai, Davies Liu, Joseph K. Bradley, Xiangrui Meng, Tomer Kaftan, Michael J. Franklin, Ali Ghodsi a Matei Zaharia. Spark SQL: Relational Data Processing in Spark. In: *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*. New York, NY, USA: Association for Computing Machinery, 2015. s. 1383–1394. SIGMOD '15. ISBN 978-1-4503-2758-9. Dostupné na DOI 10.1145/2723372.2742797. Dostupné na https://dl.acm.org/doi/10.1145/2723372.2742797.

[40] The Apache Software Foundation. *Apache HBase* [software]. [vid. 21. 3. 2023].

[41] Neo4j, Inc. *Neo4j Graph Data Platform* [software]. [vid. 21. 3. 2023]. Dostupné na https://neo4j.com/.

[42] VEsoft, Inc. *NebulaGraph* [software]. [vid. 21. 3. 2023]. Dostupné na https://www.nebula-graph.io.

[43] The Apache Software Foundation. *Apache TinkerPop* [software]. [vid. 30. 4. 2023]. Dostupné na https://tinkerpop.apache.org.

[44] The Apache Software Foundation. *Apache TinkerPop* [online]. [vid. 30. 4. 2023]. Dostupné na https://tinkerpop.apache.org/providers.html.

[45] The Linux Foundation. *JanusGraph* [software]. [vid. 30. 4. 2023]. Dostupné na https://janusgraph.org.

[46] The Linux Foundation. *Indexing for Better Performance* [oneline]. [vid. 30. 4. 2023]. Dostupné na https://docs.janusgraph.org/schema/index-management/index-performance/#composite-index.

[47] Elasticsearch B.V. *Data in: documents and indices* [online]. [vid. 28. 3. 2023]. Dostupné na https://www.elastic.co/guide/en/elasticsearch/reference/current/documents-indices.html.

[48] The Apache Software Foundation. *Apache Lucene* [software]. [vid. 28. 3. 2023]. Dostupné na https://lucene.apache.org.

[49] Elasticsearch B.V. *Mapping* [online]. [vid. 28. 3. 2023]. Dostupné na https://www.elastic.co/guide/en/elasticsearch/reference/current/mapping.html.

[50] Elasticsearch B.V. [vid. 21. 4. 2023]. Dostupné na https://www.elastic.co/guide/en/elasticsearch/reference/current/mapping types.html.

[51] Elasticsearch B.V. *Scalability and resilience: clusters, nodes, and shards* [online]. [vid. 28. 3. 2023]. Dostupné na https://www.elastic.co/guide/en/elasticsearch/reference/current/scalability.html.

[52] The Apache Software Foundation. *Apache Kafka* [software]. [vid. 28. 3. 2023]. Dostupné na https://kafka.apache.org.

[53] Google LLC. *Protocol Buffers* [software]. [vid. 29. 3. 2023]. Dostupné na https://protobuf.dev.

[54] The Apache Software Foundation. *Apache Avro* [software]. [vid. 29. 3. 2023]. Dostupné na https://avro.apache.org.

[55] Garg, Nishant. *Learning Apache Kafka - Second Edition*. Birmingham, UK: Packt Publishing, 2015. Community Experience Distilled. ISBN 978-1-78439-309-0. Dostupné na https://search.ebscohost.com/login.aspx?direct=true&db=e000xww&AN=959977&lang=cs&site=ehost-live.

[56] Narkhede, Neha, Gwen Shapira a Todd Palino. *Kafka: The Definitive Guide Real-Time Data and Stream Processing at Scale*. 1st vyd. O'Reilly Media, Inc., 2017. ISBN 1491936169.

[57] The Apache Software Foundation. *Apache Kafka Streams* [software]. [vid. 28. 3. 2023]. Dostupné na https://kafka.apache.org/documentation/streams/.

[58] Confluent, Inc. *Schema Registry Overview* [software]. [vid. 28. 3. 2023]. Dostupné na https://docs.confluent.io/platform/current/schema-registry/index.html.

[59] The Apache Software Foundation. *Airflow* [software]. [vid. 27. 3. 2023]. Dostupné na https://airflow.apache.org.

[60] The Apache Software Foundation. *Operators* [software]. [vid. 27. 3. 2023]. Dostupné na https://airflow.apache.org/docs/apache-airflow/stable/core-concepts/operators.html.

[61] The Apache Software Foundation. *Apache Spark Operators* [software]. [vid. 27. 3. 2023]. Dostupné na https://airflow.apache.org/docs/apache-airflow-providers-apache-spark/stable/operators.html.

[62] The Apache Software Foundation. *Sensors* [software]. [vid. 27. 3. 2023]. Dostupné na https://airflow.apache.org/docs/apache-airflow/stable/core-concepts/sensors.html.

[63] *Implementations* [software]. [vid. 30. 5. 2023]. Dostupné na https://json-schema.org/implementations.

[64] *JSON schema Specification* [online]. [vid. 10. 10. 2023]. Dostupné na https://json-schema.org/specification.

[65] VMware, Inc. *Spring Framework* [software]. [vid. 28. 3. 2023]. Dostupné na https://spring.io/projects/spring-framework.

[66] Wicksell, Taylor, Tom Cellucci, Howard Yuan, Asi Bross, Noel Yap a David Liu. *Netflix OSS and Spring Boot*. [vid. 1. 4. 2023]. Dostupné na https://netflixtechblog.com/netflix-oss-and-spring-boot-coming-full-circle-4855947713a0.

[67] VMware, Inc. *Spring Boot Actuator* [software]. [vid. 10. 5. 2023]. Dostupné na https://docs.spring.io/spring-boot/docs/current/reference/html/actuator.html.

[68] Broadcom Inc. *Micrometer* [software]. [vid. 5. 10. 2023]. Dostupné na https://micrometer.io.

[69] Netflix, Inc. *Eureka* [software]. [vid. 1. 4. 2023]. Dostupné na https://github.com/Netflix/eureka.

[70] Micronaut Foundation. *Micronaut* [software]. [vid. 10. 5. 2023]. Dostupné na https://micronaut.io.

[71] InfluxData, Inc. *InfluxDB* [software]. [vid. 21. 3. 2023]. Dostupné na https://www.influxdata.com.

[72] The Apache Software Foundation. *Apache HBase Reference Guide*. [vid. 21. 3. 2023]. Dostupné na https://hbase.apache.org/book.html.

[73] *Json Schema* [software]. [vid. 30. 5. 2023]. Dostupné na https://json-schema.org.

[74] VMware, Inc. *Spring Cloud* [software]. [vid. 28. 3. 2023]. Dostupné na https://spring.io/projects/spring-cloud.

[75] *git* [software]. [vid. 28. 3. 2023]. Dostupné na https://git.kernel.org/pub/scm/git/git.git/.

[76] VMware, Inc. *Project Reactor* [software]. [vid. 27. 4. 2023]. Dostupné na https://projectreactor.io.

[77] Eclipse Vert.x. *Vert.x Json Schema* [software]. [vid. 30. 5. 2023]. Dostupné na https://vertx.io/docs/vertx-json-schema/java/.

[78] FasterXML, LLC. *Jackson* [software]. [vid. 28. 3. 2023]. Dostupné na https://github.com/FasterXML/jackson.

[79] VMware, Inc. *Spring for GraphQL* [software]. [vid. 8. 11. 2023]. Dostupné na https://spring.io/projects/spring-graphql/.

[80] creativeLabs. *Core UI* [software]. [vid. 8. 11. 2023]. Dostupné na https://coreui.io.

[81] *NgRx* [software]. [vid. 8. 11. 2023]. Dostupné na https://ngrx.io.

[82] Ricadat, Pierre. *Caliban* [software]. [vid. 8. 11. 2023]. Dostupné na https://ghostdogpr.github.io/caliban/.

[83] Lightbend Inc. *sbt* [software]. [vid. 8. 11. 2023]. Dostupné na https://www.scala-sbt.org.

[84] Docker Inc. *Docker Compose* [software]. [vid. 30. 4. 2023]. Dostupné na https://docs.docker.com/compose/.

[85] Cohn, Mike. *Succeeding with Agile: Software Development Using Scrum*. Addison-Wesley, 2010. ISBN 978-0-321-57936-2.

[86] Vocke, Ham. *The Practical Test Pyramid* [online]. [vid. 1. 5. 2023]. Dostupné na https://martinfowler.com/articles/practical-test-pyramid.html.

[87] Fernandez, Tomas a Dan Ackerson. *The Testing Pyramid: How to Structure Your Test Suite* [online]. [vid. 1. 4. 2023]. Dostupné na https://semaphoreci.com/blog/testing-pyramid.

[88] Fowler, Martin. *Continuous Integration* [online]. [vid. 10. 5. 2023]. Dostupné na https://martinfowler.com/articles/continuousIntegration.htm.

[89] Pascal, Aurlane. *Happy & Unhappy Paths: Why You Need to Test Both* [software]. [vid. 10. 5. 2023]. Dostupné na https://cucumber.io/blog/test-automation/happy-unhappy-paths-why-you-need-to-test-both/.

[90] Fowler, Martin. *Unit Test* [online]. [vid. 10. 5. 2023]. Dostupné na https://martinfowler.com/bliki/UnitTest.html.

[91] Fowler, Martin. *Integration Test* [online]. [vid. 10. 5. 2023]. Dostupné na https://martinfowler.com/bliki/IntegrationTest.html.

[92] *Selenium* [software]. [vid. 10. 5. 2023]. Dostupné na https://www.selenium.dev.

[93] *Postman* [software]. [vid. 10. 5. 2023]. Dostupné na https://www.postman.com.

[94] VMware, Inc. *Spring Boot testing* [online]. [vid. 10. 5. 2023]. Dostupné na https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#features.testing.

[95] Bechtold, Stefan, Sam Brannen, Johannes Link, Matthias Merdes, Marc Philipp, Juliette de Rancourt a Christian Stein. *JUnit User Guide* [online]. [vid. 10. 5. 2023]. Dostupné na https://junit.org/junit5/docs/current/user-guide/.

[96] Fowler, Martin. *Xunit* [online]. [vid. 10. 5. 2023]. Dostupné na https://martinfowler.com/bliki/Xunit.html.

[97] *Mockito* [software]. [vid. 10. 5. 2023]. Dostupné na https://site.mockito.org.

[98] SmartBear, Inc. *Cucumber* [software]. [vid. 10. 5. 2023]. Dostupné na https://cucumber.io.

[99] SmartBear, Inc. *Behaviour-Driven Development* [online]. [vid. 10. 5. 2023]. Dostupné na `https://cucumber.io/docs/bdd/`.

[100] VMware, Inc. *Spring GraphQL Tests* [online]. [vid. 12. 5. 2023]. Dostupné na `https://docs.spring.io/spring-graphql/reference/testing.html`.

Příloha A
Seznam použitých zkratek

API	Aplication Interface
BDD	Behavior Driven Development
CI/CD	Continuous Integration / Continuous Delivery
DI	Dependency Injection
DS	Distribuční Soustava
DSL	Domain Specific Language
EAN	European Article Number
EDC	Elektroenergetické Datové Centrum
ERÚ	Energetický Regulační Úřad
E2E	End-to-End
F	Funkční požadavek
FVE	Fotovoltaická Elektrárna
HTTPS	Hypertext Transfer Protocol Secure
IoC	Inversion of Control
JDBC	Java Database Connectivity
JSON	JavaScript Object Notation
JVM	Java Virtual Machine
MPO	Ministerstvo Průmyslu a Obchodu
MPPT	Maximum Power Point Tracking
N	Nefunkční požadavek
OM	Odběrné Místo
OMp	Odběrné Místo přidružené
OMv	Odběrné Místo vůdčí
OZE	Obnovitelné Zdroje Energie
PDS	Provozovatel Distribuční Sítě
POJO	Plain Old Java Object
PS	Přenosová Soustava
SPA	Single Page Application
SSL	Secure Sockets Layer
TCP	Transmission Control Protocol
TDD	Test Driven Development
TLS	Transport Layer Security
UC	Use Case
UID	Unique Identifier
UML	Unified Modeling Language
URL	Uniform Resource Locator
UUID	Universal Unique Identifier
VQL	Verta Query Language
XML	Extensible Markup Language

Příloha B
Obsah přiloženého paměťového média

```
.
|-- README.md                          popis archivu digitální přílohy
|-- src/                                    adresář se zdrojovými kódy
|   |-- thesis/                            zdrojový kódy textu práce
|   `-- impl/                    zdrojové kódy implementovaného systému
`-- thesis.pdf                              text práce ve formátu PDF
```

Příloha C
Uživatelská příručka

Uživatelská příručka

systém pro podporu správy komunitních FVE

1 Obsah

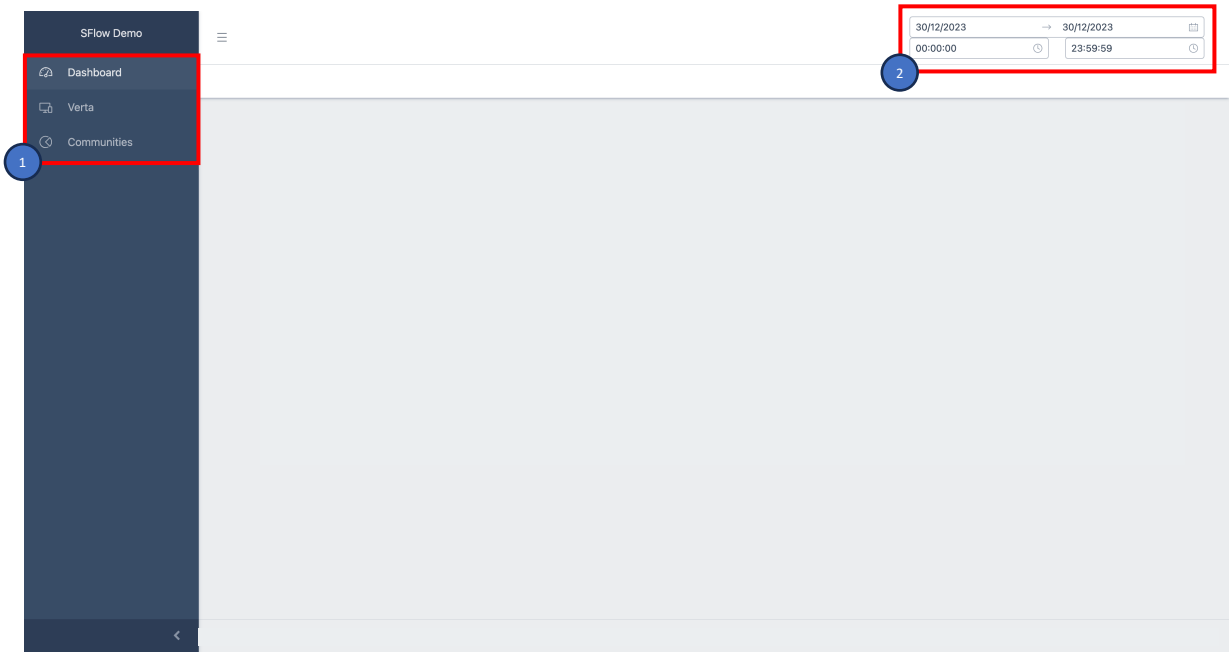
2 Popis funkcionalit

Systém pro podporu správy komunitních FVE umožňuje evidování datových objektů, které mohou reprezentovat např. jednotlivé komponenty FVE, ale také abstraktní struktury, jako jsou zákazníci nebo odběrná místa. Dále umožňuje k jednotlivým objektům zobrazovat naměřená data – časové řady. Chování všech částí aplikace lze ovlivňovat pomocí globálního časového filtru (sekce 3.2).

Všechny poskytované funkcionality slouží pouze pro zobrazování dat (případně spouštění úloh). Přes webovou aplikaci **nelze přidávat** nová data do systému.

3 Globální funckionality

V této sekci jsou popsány globální funkcionality, které lze využívat v libovolné části webové aplikace. Těmito funkcionalitami lze ovlivňovat chování všech dále popsaných částí aplikace.



Obrázek 1 Globální funkcionality

3.1 Navigace

V levé části aplikace je vždy zobrazena hlavní nabídka obsahující tři prvky: *Dashboard, Verta, Communities* (vizte Obrázek 1, číslo 1). Jednotlivé položky lze vybrat kliknutím.

- *Dashboard* zobrazí přehled časových řad pro objekt definovaný v environment property. Při běhu aplikace toto chování nelze nijak měnit.
- *Verta* zobrazí přehled všech datových objektů evidovaných v systému. Odtud je poté možné zobrazit vlastnosti (a jiné detaily) pro jednotlivé objekty včetně vazeb (více v sekci 4).
- *Communities* zobrazí výběr z instancí sdílení elektrické energie (komunit). Odtud jde poté zobrazit detaily sdílení – kolik elektrické energie bylo vyrobeno, kolik bylo spotřebováno na jednotlivých odběrných místech atp. (více v sekci 6).

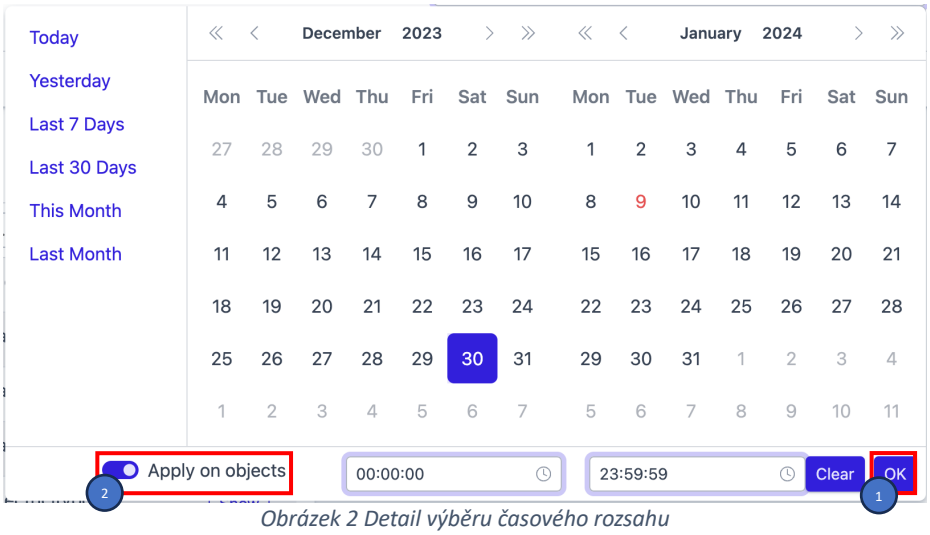
3.2 Filtrování podle času

Systém udržuje historii všech změn, které byly provedeny. Pro jejich procházení slouží filtr v levé horní části webové aplikace (vizte Obrázek 1, číslo 2). Tento filtr respektují **všechny operace**, které lze v aplikaci provést. Stejně tak má tento výběr pro každé zobrazení stejný

význam, a to: *Zobrazovaná data musela být platná po celou dobu vybraného úseku.* Technicky to znamená, že musela být vytvořena dříve, než je začátek zadaného intervalu, a smazána (invalidována) později, než je konec zadaného intervalu.

Výběr časového rozsahu lze provést kliknutím na počáteční nebo koncové datum v pravém horním rohu (zvýrazněno na Obrázek 1, číslo 2). Následně se zobrazí nabídka pro výběr data a času (vizte Obrázek 2). Výběr je nutné potvrdit tlačítkem *Ok* (zvýrazněno v pravém dolním rohu na Obrázek 2, číslo 1).

Přepínačem *Apply on objects* (Obrázek 2, číslo 2) je možné potlačit použití časového filtru pro objekty. Poté je vybraný časový rozsah aplikován pouze na data časových řad (měření).



Obrázek 2 Detail výběru časového rozsahu

4 Zobrazení přehledu všech evidovaných objektů

Přechod do seznamu všech evidovaných komponent se provede kliknutím na položku *Verta* v hlavním (levém) menu aplikace (vizte sekci 3.1). Tato položka je zvýrazněna na Obrázek 3 číslem 1. Po kliknutí zobrazí webová aplikace přehled všech evidovaných objektů formou tabulky (vizte Obrázek 3, číslo 1). Zobrazená data respektují časový filtr v levé horní části (vizte sekci 3.2).

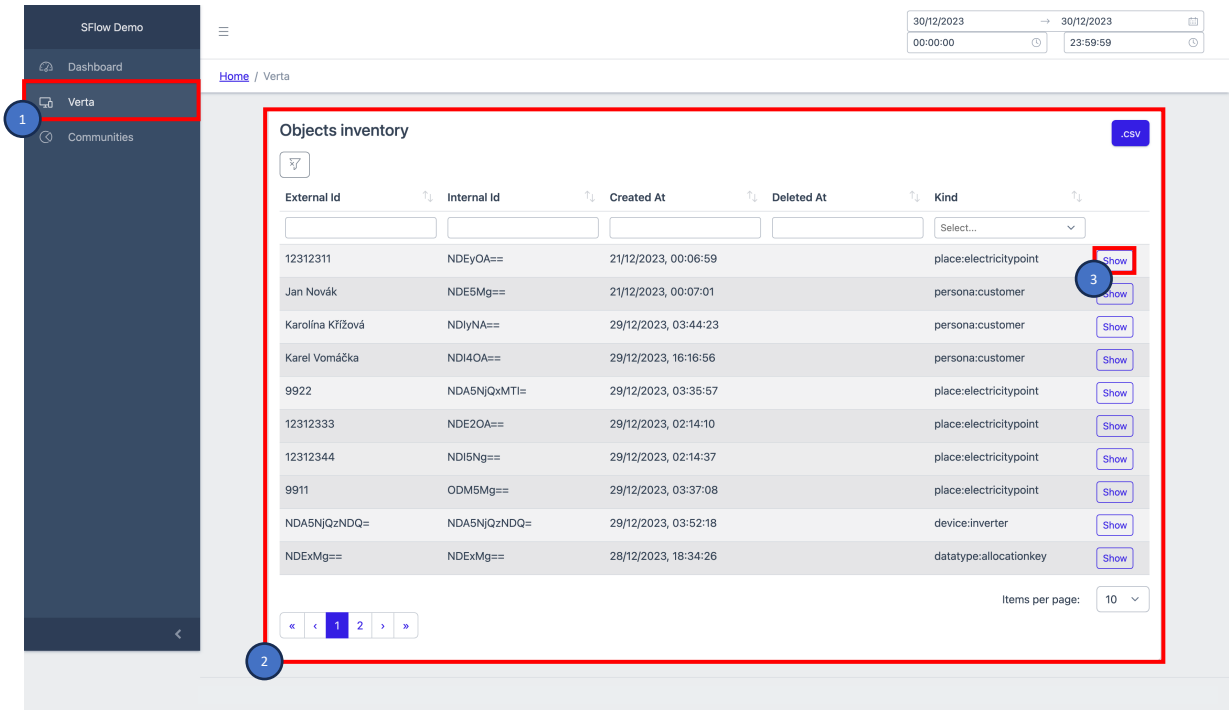
Pozor tedy, vytvoří-li se nový objekt a časový rozsah začíná dříve, než byl tento objekt vytvořen, aplikace ho v této tabulce nezobrazí, protože nesplňuje časový filtr! Obdobně, smaže-li (invaliduje) se objekt a časový rozsah končí až po čase, ve kterém byl objekt smazán, aplikace ho v tabulce také nezobrazí!

Objekty, které jsou zobrazeny, ale byly smazány (tato situace může nastat při prohlížení historie), jsou označeny v tabulce značkou na červeném pozadí s textem *deleted.* deleted

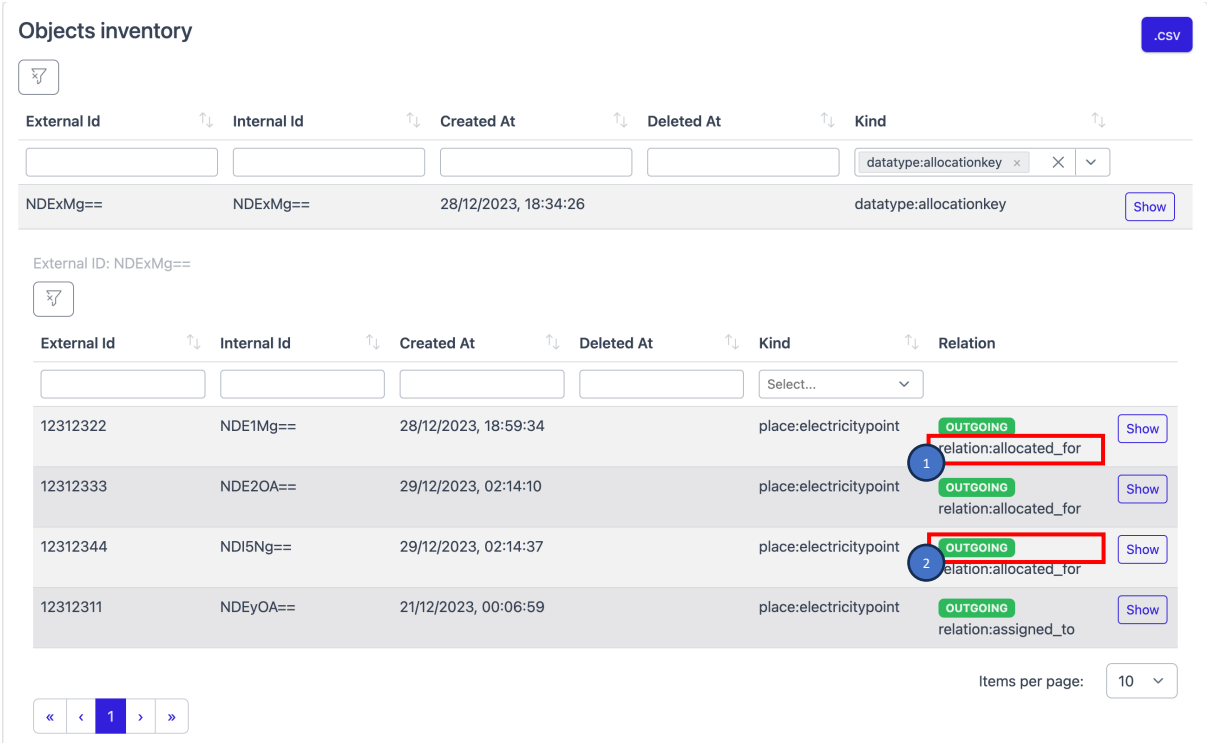
Tabulka je stránkována, ve výchozím nastavení zobrazuje 10 položek na jednu stránku. V tabulce lze filtrovat podle všech sloupců. První čtyři lze filtrovat zadáním textu do příslušných polí. Poslední sloupec je možné filtrovat výběrem typu datového objektu. Kliknutím na tlačítko *Show* (číslo 3) lze zobrazit detail datového objektu (vizte sekci 5).

4.1 Zobrazení vazeb datového objektu

Každý záznam v tabulce lze rozkliknout (kliknutím kdekoliv na řádek, kromě tlačítka *Show* v pravé části). Po rozkliknutí aplikace zobrazí přehled všech objektů, které mají vazbu s vybraným objektem. Detail vazeb pro datový objekt je možné vidět na Obrázek 4 Zobrazení detailu vazeb evidovaného datového objektu. V tomto detailu je pro každou vazbu uveden její typ (číslo 1) a směr (číslo 2). *Outgoing* značí, že daná vazba začíná v objektu, jehož detail je zobrazen, *ingoing* naopak značí, že vazba do tohoto objektu vede.



Obrázek 3 Zobrazení všech evidovaných objektů



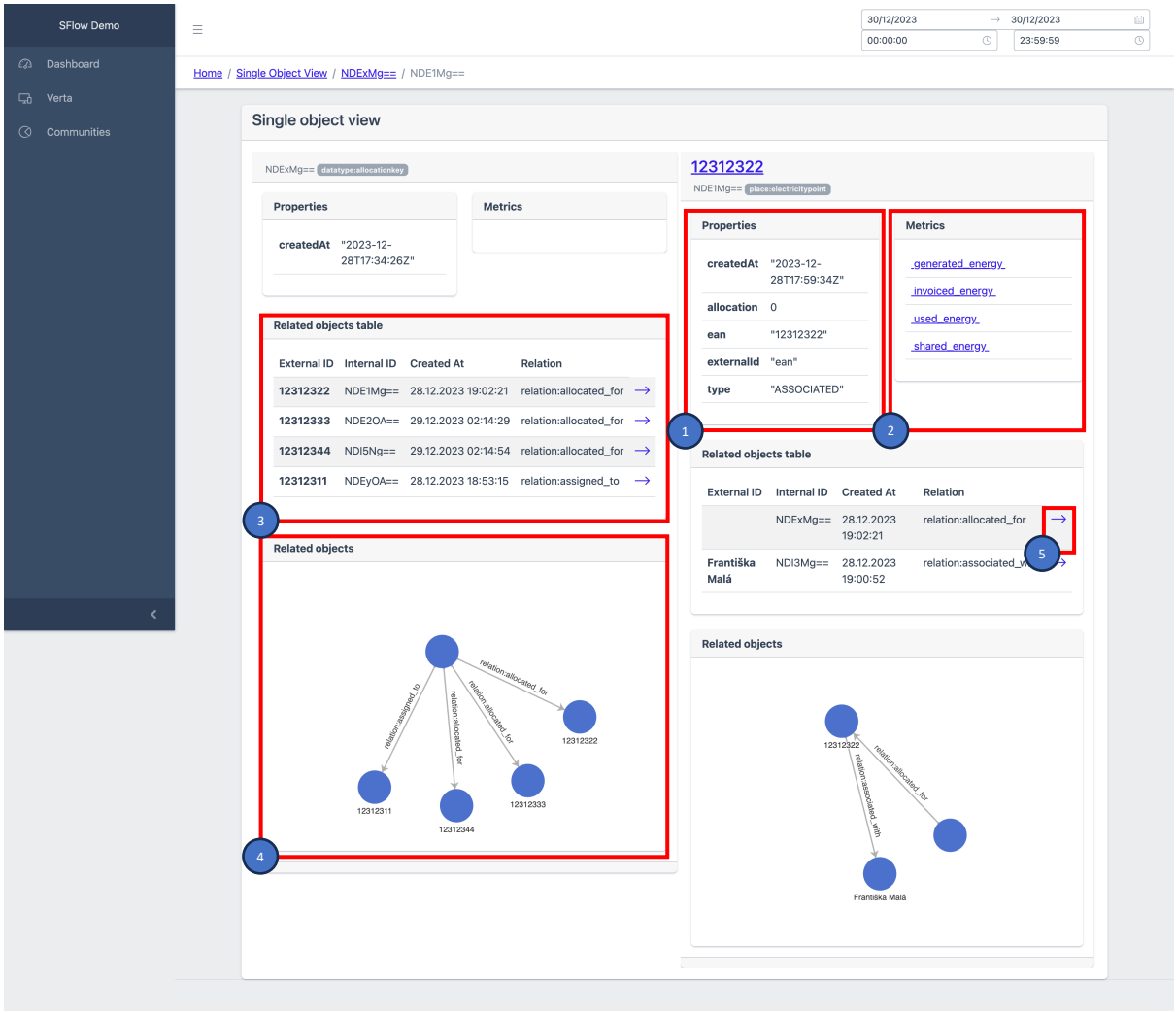
Obrázek 4 Zobrazení detailu vazeb evidovaného datového objektu

5 Zobrazení detailu datového objektu

Detail konkrétního datového objektu lze zobrazit kliknutím na tlačítko *Show* (vizte sekci 4). Pro vybraný objekt jsou zobrazeny všechny jeho vlastnosti (Obrázek 5, číslo 1). Dále jsou zobrazena všechna měření (časové řady) evidované pro tento objekt (Obrázek 5, číslo 2). Kliknutím na název časové řady je možné zobrazit její data v patnáctiminutových intervalech v zadaném časovém rozsahu (globální filtr).

Dále jsou k vybranému objektu zobrazeny jeho vazby (Obrázek 5, číslo 3). U každé z nich je uveden její typ a objekt, se kterým je vybraný objekt spojen touto vazbou. Stejná informace je poté vizualizována pod touto tabulkou (Obrázek 5, číslo 4). Na konci každého řádku tabulky je tlačítko se znakem šipky vpravo (Obrázek 5, číslo 5). Po kliknutí na tuto šipku je zobrazen daný objekt vpravo od objektu, na který bylo kliknuto. *Na Obrázek 5 by popsaná situace vypadala následovně: Nejprve by byl zobrazen pouze objekt typu datatype:allocationkey. Poté by bylo kliknuto na šipku na řádku v tabulce seznamu vazeb reprezentující vazbu s datovým objektem s externím identifikátorem 12312322. Po provedení těchto akcí by aplikace zobrazovala stejnou obrazovku, jako je na Obrázek 5.*

Takto lze řetězit za sebou zobrazení několika datových objektů. Díky tomu je poté snazší prohledávat uložená data a získat přehled o jejich topologii.

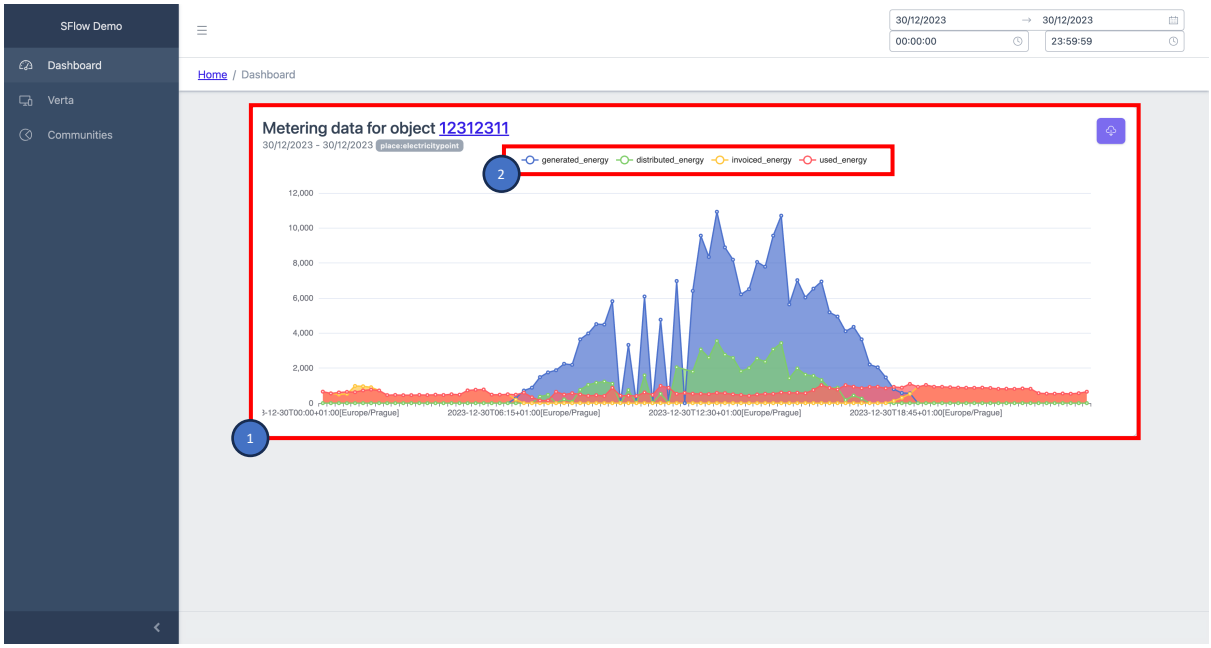


Obrázek 5 Detail vybraného datového objektu (resp. několika objektů)

6 Zobrazení časových řad (dat měření)

Obrazovka zachycená na Obrázek 6 ukazuje detailní zobrazení časových řad pro vybraný datový objekt. Na tuto obrazovku lze přejít z detailu zobrazení datového objektu (vizte sekci 5). Případně z hlavního menu, potom je zobrazen detail staticky definovaného objektu v environment property (vizte sekci 3.1).

Zobrazení časových řad respektuje globální časový filtr (vizte sekci 3.2). Data časových řad jsou zobrazena do grafu (Obrázek 6, číslo 1) agregovaná součtem do patnáctiminutových intervalů. Jednotlivé časové řady je možné zobrazovat a schovávat kliknutím na jejich název (Obrázek 6, číslo 2). Kliknutím na externí identifikátor objektu je možné přejít na jeho detail – sekce 5.



Obrázek 6 Zobrazení časových řad (dat měření) pro vybraný datový objekt

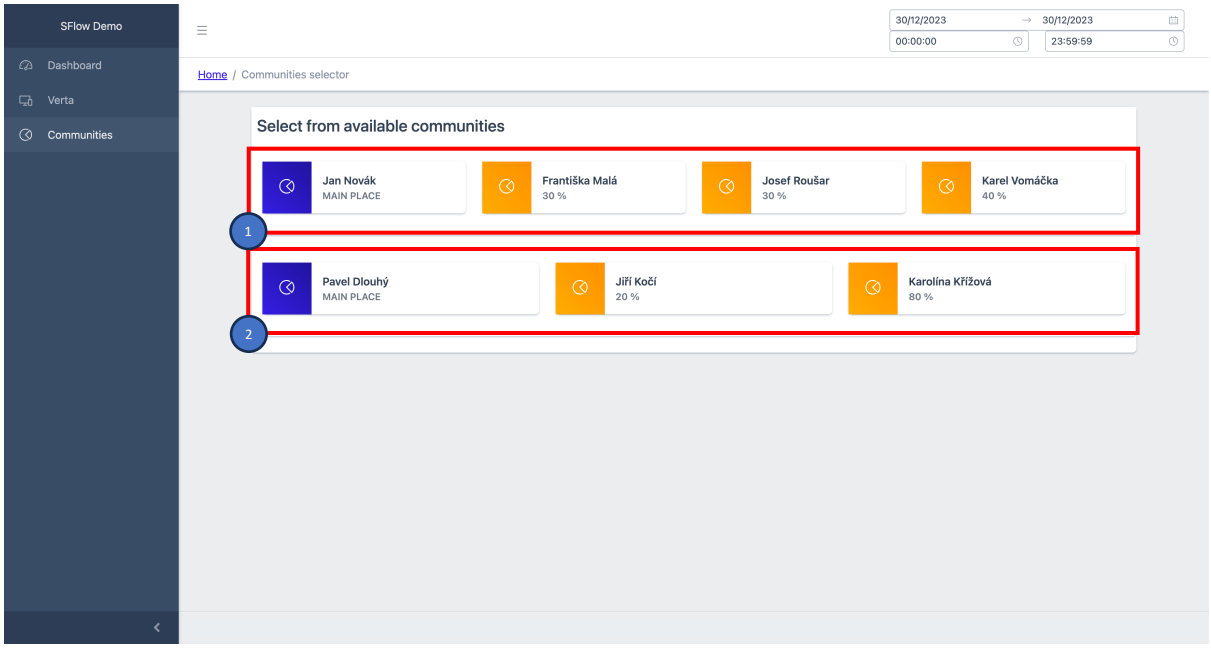
7 Zobrazení stavu sdílení elektrické energie

Přistoupit na obrazovku s výběrem instance sdílení (alokačního klíče) je možné z hlavního menu v levé části aplikace kliknutím na tlačítko *Communities* (vizte sekci 3.1).

7.1 Výběr instance sdílení elektrické energie

V systému je možné evidovat více instancí sdílení elektrické energie, přičemž každá má své vůdčí odběrné místo, přidružená odběrná místa, alokační klíč atd. Při výběru, pro kterou instanci má být zobrazen detail, je v seznamu uveden vždy seznam všech míst, která se podílejí na sdílení (resp. jsou zobrazena jména zákazníků, kteří jsou s těmito místy spjati). Dále je pro každé místo uvedeno, kolik procentních bodů z vyrobené energie má dané odběrné místo (resp. zákazník) alokováno.

Na Obrázek 7 je možné vidět obrazovku pro popsaný výběr pro dvě instance sdílení (čísla 1 a 2). Požadovanou instanci lze vybrat kliknutím na jedno z jejích odběrných míst. Aplikace poté zobrazí detail sdílení pro danou instanci (vizte sekci 7.2).



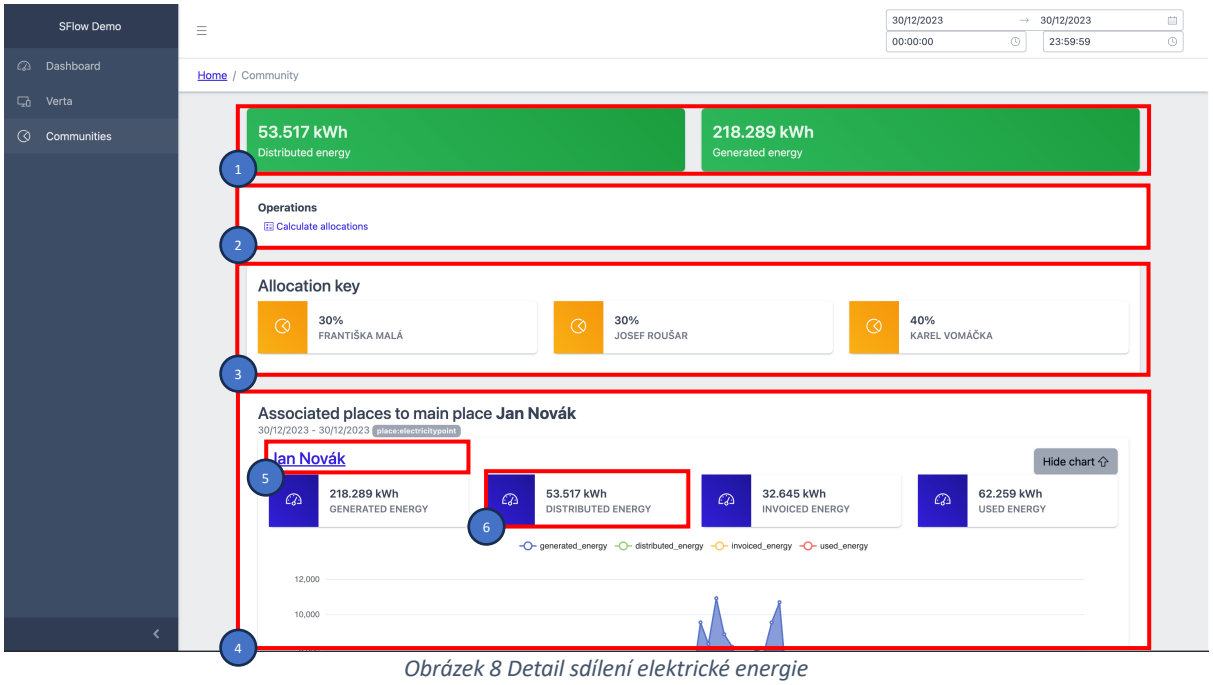
Obrázek 7 Výběr instance sdílení (alokačního klíče) pro zobrazení

7.2 Detail instance sdílení elektrické energie

Po výběru konkrétního alokačního klíče (vizte sekci 7.1) aplikace zobrazí detailní stav sdílení elektrické energie. V horní části (Obrázek 8) jsou zobrazeny souhrnné informace o množství distribuované (dodané do sítě) a vygenerované energie (číslo 1) . Za tímto následují akce, které je možné s daty v sdílení elektrické energie provést (číslo 2). Dále je uveden alokační klíč (číslo 3). Za těmito souhrnnými informacemi následují detaily jednotlivých odběrných míst počínaje vůdčím odběrným místem (číslo 4).

Jednotlivé části umožňují interaktivní přechod do jiných zobrazení. Žádným z těchto přechodů není ovlivněn globální časový filtr. Všechny navazující obrazovky využívají stále stejný časový rozsah. Kliknutím na odběrné místo v zobrazení alokačního klíče (číslo 3) je možné přejít do zobrazení detailu daného odběrného místa (obrazovka popsaná v sekci 5). Kliknutím na jméno zákazníka v detailu každého odběrného místa (číslo 5) je možné přejít na zobrazení všech časových řad evidovaných k vybranému odběrnému místu (resp. zákazníkovi). Jedná se o obrazovku popsanou v sekci 6. Obdobným způsobem se lze dostat na detailní zobrazení pouze jedné časové řady pro vybrané odběrné místo. Místo jména je však nutné kliknout na požadovanou metriku v souhrnu informací pro požadované odběrné místo (číslo 6). Po této volbě bude zobrazena obrazovka popsaná v sekci 6.

Na Obrázek 9 je vidět celá obrazovka detailu sdílení, včetně detailů všech odběrných míst. Pro všechna přidružená místa jsou zobrazované informace stejné. Jedná se o souhrny měření (vždy jde o součet naměřených hodnot daného měření v rozsahu specifikovaného globálním filtrem – vizte sekci 3.2).

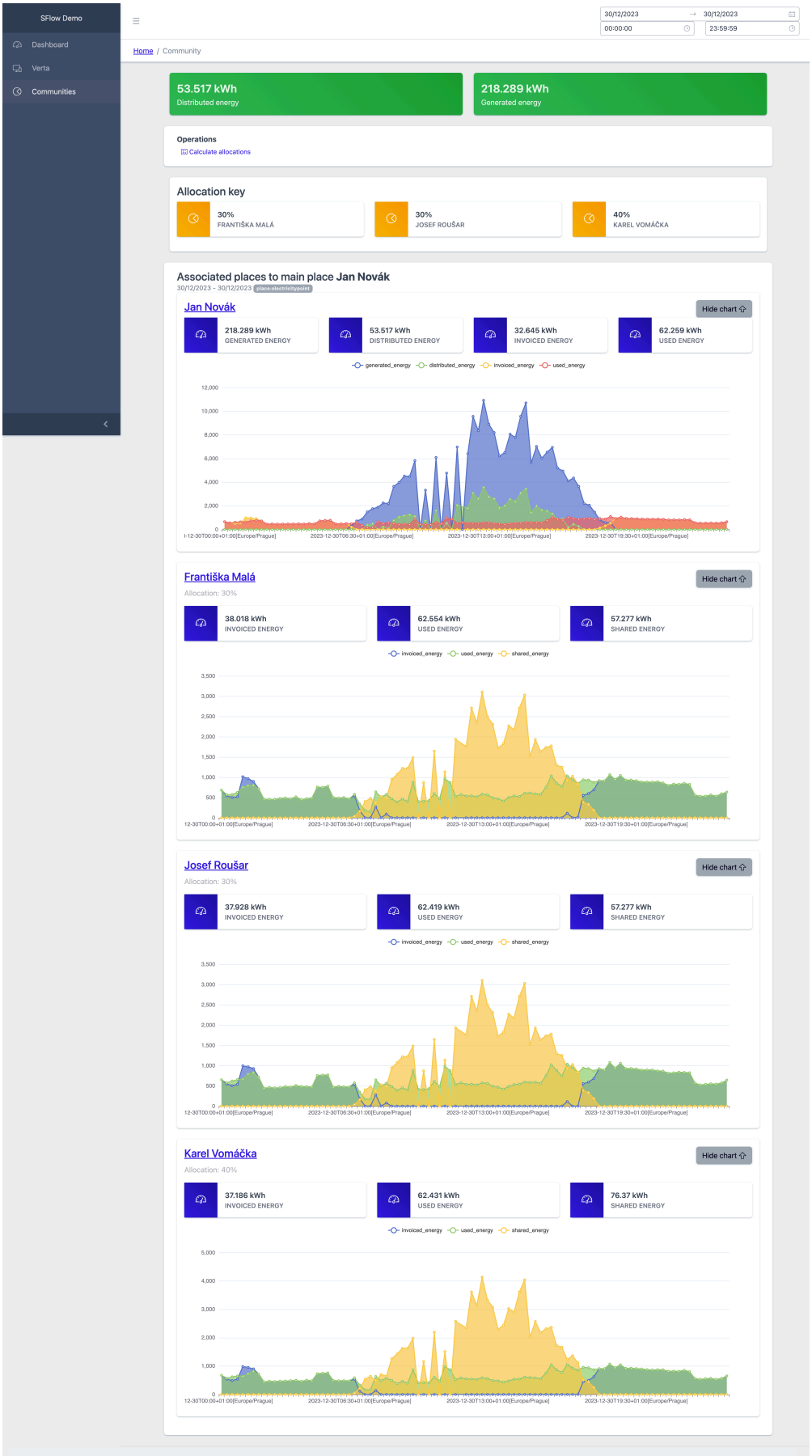


Obrázek 8 Detail sdílení elektrické energie

7.3 Ruční spuštění rozpočtu vyrobené energie

Systém automaticky rozpočítává vyrobenou elektrickou energii mezi jednotlivá odběrná místa podle alokačního klíče. Tento výpočet je prováděn v konfigurovatelných intervalech, ale lze jej vyvolat bezodkladně kliknutím na tlačítko *Calculate allocations* (v části s číslem 2).

Pozor, je však nutné myslet na to, že samotný výpočet není okamžitý a výsledky nejsou do systému ukládány synchronně. Proto samotné přijetí požadavku ještě neznamená jeho dokončení a data se tak mohou v systému objevit se zpožděním.



Obrázek 9 Detail sdílení elektrické energie (kompletní obrazovka)

Příloha D
Reporty testování

Tato příloha obsahuje reporty pokrytí implementace pomocí testů pro jednotlivé komponenty systému. K vytvoření reportů byl použit plugin Jococo.

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods	Missed	Classes
com.thybit.sflow.common.hbase.tsdb		70%		60%	14	41	46	160	5	23	1	5
com.thybit.sflow.common.hbase.tsdb.query		70%		62%	13	39	45	134	5	23	2	9
com.thybit.sflow.common.hbase.context		57%		53%	19	29	6	25	5	14	1	4
com.thybit.sflow.common.hbase.model		73%		75%	7	22	10	53	6	20	0	6
com.thybit.sflow.common.hbase.types		87%		60%	17	69	15	100	10	59	0	11
com.thybit.sflow.common.hbase.bytes		86%		50%	19	66	16	105	4	44	0	2
com.thybit.sflow.common.hbase.matcher		95%		79%	14	63	11	150	3	34	1	9
com.thybit.sflow.common.hbase.exception		29%		n/a	4	6	8	12	4	6	1	3
com.thybit.sflow.common.hbase.tsdb.exception		0%		n/a	4	4	8	8	4	4	2	2
com.thybit.sflow.common.hbase.mapping		95%		100%	2	13	2	27	2	12	0	3
com.thybit.sflow.common.hbase.tsdb.query.model		88%		62%	3	5	2	7	0	1	0	1
com.thybit.sflow.common.hbase.util		98%		n/a	1	12	1	26	1	12	0	4
com.thybit.sflow.common.hbase.client		100%		n/a	0	1	0	2	0	1	0	1
Total	808 of 4,162	80%	76 of 212	64%	117	370	170	809	49	253	8	60

Created with JaCoCo 0.8.9.202303310957

Obrázek D.1. Report pokrytí implementace testy pro databázi časových řad

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods	Missed	Classes
com.thybit.sflow.repository.controller		96%		100%	1	17	3	50	1	14	1	2
com.thybit.sflow.repository.service		93%		66%	2	12	1	21	0	9	0	2
com.thybit.sflow.repository.dto		77%		n/a	2	5	2	5	2	5	2	5
com.thybit.sflow.repository		58%		n/a	1	3	2	4	1	3	0	1
com.thybit.sflow.repository.exceptions		72%		n/a	1	3	2	6	1	3	0	1
com.thybit.sflow.repository.dao		100%		n/a	0	3	0	11	0	3	0	1
Total	29 of 461	93%	2 of 12	83%	7	43	10	97	5	37	3	12

Created with JaCoCo 0.8.9.202303310957

Obrázek D.2. Report pokrytí implementace testy pro komponentu `Schema Repository`

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods	Missed	Classes
com.thybit.sflowdatastasher.configuration		0%		0%	45	45	41	41	31	31	7	7
com.thybit.sflowdatastasher.model		24%		0%	39	54	3	10	12	27	0	2
com.thybit.sflowdatastasher.hbase		1%		0%	11	12	61	62	10	11	1	2
com.thybit.sflowdatastasher.enhacer		13%		0%	9	12	23	28	8	11	1	2
com.thybit.sflowdatastasher.kafka		57%		50%	4	10	21	44	2	8	0	1
com.thybit.sflowdatastasher.enhacer.uid		0%		n/a	3	3	8	8	3	3	2	2
com.thybit.sflowdatastasher.exceptions		0%		n/a	6	6	12	12	6	6	2	2
com.thybit.sflowdatastasher.controller		0%		n/a	3	3	8	8	3	3	1	1
com.thybit.sflowdatastasher.hbase.model		36%		n/a	3	4	3	4	3	4	3	4
com.thybit.sflowdatastasher		0%		n/a	4	4	7	7	4	4	2	2
com.thybit.sflowdatastasher.bootstrap		0%		n/a	2	2	4	4	2	2	1	1
Total	1,146 of 1,366	16%	88 of 90	2%	129	155	191	228	84	110	20	26

Created with JaCoCo 0.8.9.202303310957

Obrázek D.3. Report pokrytí implementace testy pro komponentu `Datastasher`

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods	Missed	Classes
com.thybit.sflow.verta.service		57%		43%	23	55	86	217	12	39	0	5
com.thybit.sflow.verta.controller		33%		0%	29	45	80	128	25	41	1	7
com.thybit.sflow.verta.model.dto.request.matcher.visitor		37%		20%	14	30	55	86	6	20	0	3
com.thybit.sflow.verta.model.serde		70%		59%	13	45	37	119	5	31	0	9
com.thybit.sflow.verta.service.validator		70%		48%	13	37	20	68	2	19	0	3
com.thybit.sflow.verta.model.dto.request.property.value.type		69%		50%	28	55	8	69	8	35	0	7
com.thybit.sflow.verta.model.dto.request.property.value.factory		72%		55%	16	45	21	72	9	36	0	8
com.thybit.sflow.verta.model.dto.request		22%		n/a	7	9	7	9	7	9	7	9
com.thybit.sflow.verta.model.dto.request.property		19%		0%	3	4	9	10	2	3	1	2
com.thybit.sflow.verta.model.dto.response		89%		85%	6	21	5	50	4	14	2	5
com.thybit.sflow.verta.model.dto.request.matcher.property.factory		73%		n/a	10	24	10	32	10	24	0	8
com.thybit.sflow.verta		28%		n/a	5	7	5	7	5	7	2	3
com.thybit.sflow.verta.model.dto.request.property.value		72%		0%	3	5	4	16	1	3	0	1
com.thybit.sflow.verta.utils		92%		83%	7	37	2	41	2	22	2	4
com.thybit.sflow.verta.model.dto.request.matcher.property.type		63%		n/a	4	10	8	20	4	10	2	8
com.thybit.sflow.verta.model.dto.request.matcher.traversal.type		42%		n/a	2	4	4	8	2	4	0	2
com.thybit.sflow.verta.service.validator.rules		93%		100%	1	10	3	22	1	8	1	4
com.thybit.sflow.verta.model.dto.request.matcher.property		64%		n/a	1	2	3	7	1	2	1	2
com.thybit.sflow.verta.model.dto.request.matcher.logical.factory		84%		n/a	2	6	2	8	2	6	0	2
com.thybit.sflow.verta.model.dto.request.matcher.label.factory		84%		n/a	2	6	2	8	2	6	0	2
com.thybit.sflow.verta.model.dto.request.matcher.traversal.factory		84%		n/a	2	6	2	8	2	6	0	2
com.thybit.sflow.verta.model.dto.request.matcher.label.type		50%		n/a	1	2	2	4	1	2	1	2
com.thybit.sflow.verta.hash		91%		n/a	0	5	2	7	0	5	0	1
com.thybit.sflow.verta.model.dto.request.matcher.id.factory		92%		n/a	1	3	1	6	1	3	0	1
com.thybit.sflow.verta.model.dto.request.matcher		100%		n/a	0	5	0	10	0	5	0	2
com.thybit.sflow.verta.model.dto.request.matcher.id		100%		n/a	0	1	0	4	0	1	0	1
com.thybit.sflow.verta.model.dto.request.matcher.traversal		100%		n/a	0	1	0	4	0	1	0	1
com.thybit.sflow.verta.model.dto.request.matcher.label		100%		n/a	0	1	0	4	0	1	0	1
com.thybit.sflow.verta.model.dto.request.matcher.logical		100%		n/a	0	1	0	4	0	1	0	1
com.thybit.sflow.verta.model.dto.request.matcher.logical.type		100%		n/a	0	2	0	4	0	2	0	2
com.thybit.sflow.verta.model.dto.request.matcher.id.type		100%		n/a	0	1	0	2	0	1	0	1
Total	1,977 of 5,135	61%	110 of 229	51%	193	485	378	1,054	114	367	20	109

Created with JaCoCo 0.8.9.202303310957

Obrázek D.4. Report pokrytí implementace testy pro komponentu `Verta`

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods	Missed	Classes
com.thybit.sflow.queryresolver.controller.graphql		24%		10%	63	79	95	134	35	51	0	7
com.thybit.sflow.queryresolver.model.graphql		42%		50%	21	32	23	36	19	30	10	19
com.thybit.sflow.queryresolver.graphql		34%		11%	16	27	35	57	7	18	0	6
com.thybit.sflow.queryresolver.model		22%		0%	16	17	16	22	6	7	4	5
com.thybit.sflow.queryresolver.controller		61%		69%	8	20	7	18	2	7	0	1
com.thybit.sflow.queryresolver.serde		40%		n/a	1	6	8	16	1	6	0	2
com.thybit.sflow.queryresolver.model.graphql.value		60%		25%	5	9	2	8	2	6	0	1
com.thybit.sflow.queryresolver.service		46%		n/a	1	2	1	4	1	2	0	1
com.thybit.sflow.queryresolver.clients		100%		n/a	0	4	0	8	0	4	0	3
Total	1,275 of 1,923	33%	85 of 113	24%	131	196	187	303	73	131	14	45

Obrázek D.5. Report pokrytí implementace testy pro komponentu `Query resolver`