Univerzita Hradec Králové
Fakulta informatiky a managementu
Katedra informatiky a kvantitativních metod

Vizualizace vztahů ve sdružení NBS

Diplomová práce

Autor: Bc. Pavel Bořík
Studijní obor: Aplikovaná informatika

Vedoucí práce: Mgr. Jiří Haviger, Ph.D.
Odborný konzultant: Ing. Stanislav Mikulecký, Ph.D.

Hradec Králové Duben 2019

Prohlášení:
Prohlašuji, že jsem diplomovou práci vypracoval samostatně a uvedl jsem všechny použité prameny a literaturu.

V Hradci Králové dne 14. dubna 2019 Bc. Pavel Bořík

Poděkování:
Děkuji vedoucímu diplomové práce Mgr. Jiřímu Havigerovi, Ph.D. za odborné vedení a cenné rady při tvorbě této práce. Dále chci poděkovat zaměstnancům firmy Unicorn Systems Ing. Stanislavu Mikuleckému, Ph.D. a Ing. Petru Havelkovi za podnětnou a bezproblémovou spolupráci.

Anotace

Tato práce se věnuje vývoji vizualizačního nástroje, který zobrazuje vztahy mezi entitami na energetickém trhu ve formě orientovaného grafu. Cílem práce je navrhnout a implementovat front-endovou aplikaci v programovacím jazyce JavaScript za využití knihovny React a knihovny pro vizualizaci grafů Vis.js. Vytvořená aplikace je schopna přehledně vizualizovat vztahy mezi stovkami entit a je možné ji nasadit v již existujícím informačním systému.

Annotation

Title: Visualization of relationships in the NBS group

This Diploma Thesis deals with the development of a visualization tool which displays relationships between entities on an energy market as a directed graph. The goal of the thesis is to design and implement a front-end application while using the JavaScript programming language along with the React and Vis.js libraries. The created application is able to visualize the relationships between hundreds of entities in a well-arranged way and it's possible to deploy it as a part of an already existing information system.

Obsah

1 Úvod **1**

2 Cíl práce **2**
2.1 Základní požadavky . . . 2

3 Grafy a jejich vizualizace **3**
3.1 Graf . . . 3
3.2 Reprezentace grafů . . . 5
3.3 Vizualizace grafů . . . 8

4 Sdružení Nordic Balance Settlement **13**
4.1 Činnost NBS . . . 13
4.2 Participující subjekty . . . 13
4.3 Ostatní entity . . . 14
4.4 Struktura NBS . . . 15
4.5 Mohutnost vztahů . . . 17

5 Použité technologie **19**
5.1 JavaScript . . . 19
5.2 React . . . 20
5.3 Další podpůrné technologie . . . 25

6 Knihovny pro vizualizaci grafových struktur **27**
6.1 Požadavky na vybranou knihovnu . . . 27
6.2 Přehled vybraných zástupců . . . 28
6.3 Zhodnocení vybraných zástupců . . . 32

7 Vývoj aplikace **34**
7.1 Inicializace projektu . . . 34
7.2 Návrhová část . . . 36
7.3 Implementační část . . . 41

8 Shrnutí výsledků **55**

9 Závěr **56**

Literatura **57**

Přílohy **60**

Seznam obrázků

3.1 Neorientovaný graf 3
3.2 Orientovaný graf 4
3.3 Multigraf 4
3.4 Kořenový strom 5
3.5 Definice grafu pomocí seznamu sousedů 6
3.6 Příklad matice sousednosti a matice incidence popisující ukázkový graf 7
3.7 Ortogonální vizualizace grafu 9
3.8 Vrstvení grafu s využitím pomocných vrcholů pro lepší vizualizaci hran 10
3.9 Graf vykreslený pomocí Lombardiho metody 11

4.1 Princip vztahů mezi obchodníky, BRP a MGA 15
4.2 Informační propojení mezi různými subjekty v NBS 16
4.3 Schéma vztahů mezi entitami v NBS 17

5.1 Počet stažení různých front-endových technologií z databáze Node package manager mezi roky 2013 až 2018 21
5.2 Životní cyklus React komponent při jejich vytváření a aktualizaci 26

7.1 Základní adresářová struktura projektu po instalaci závislostí 36
7.2 Ukázková struktura testovacích dat exportovaných do podoby databázových tabulek 37
7.3 Návrh uživatelského rozhraní 39
7.4 Struktura React komponent v aplikaci 44
7.5 Ukázka vykresleného grafu s legendou 49
7.6 Graf s vyznačenými hranami 50
7.7 Průběh odkrytí části grafu po poklikání na cluster 53
7.8 Průběh vytvoření subclusterů po poklikání na cluster 54

Seznam ukázek kódu

3.1 Reprezentace grafu ve formátu GraphML . . . 7
3.2 Reprezentace grafu ve formátu DOT . . . 8
5.1 Možnosti vytváření React elementů . . . 23
5.2 Funkcionální komponenta . . . 23
5.3 Komponenta jako ES6 třída . . . 24
5.4 Typické využití metody *componentDidUpdate()* . . . 25
6.1 Přidání vrcholů do grafu v D3.js . . . 29
6.2 Ukázka funkcionality clusteringu v knihovně Vis.js . . . 30
6.3 Příkad použití funkce Dijkstrova algoritmu nabízené knihovnou Cytoscape.js . . . 32
7.1 Ukázka souboru package.json . . . 35
7.2 Část konfiguračního objektu ze serverové odpovědi popisující rozdělení vrcholů do clusterů pomocí stromové struktury . . . 40
7.3 Část objektu popisující vrchol grafu, ukazující vlastnosti *id*, *label* a *clustering* . . . 41
7.4 Základní operace nutné pro spuštění testovacího serveru . . . 42
7.5 Funkce koncového bodu *getData* . . . 42
7.6 Získání dat ze serveru v komponentě *Dashboard* . . . 45
7.7 Změna vybraného časového data v komponentě *EnhancedDatePicker*, podle kterého jsou zobrazeny vztahy mezi entitami v pohledu *Moment* . . . 46
7.8 Získání detailních informací ze serveru při výběru vrcholu v grafu v komponentě *InfoCardDetail* . . . 47
7.9 Využití serverové odpovědi k vytvoření informační karty v komponentě *InfoCardDetail* 47
7.10 Základní integrace knihovny Vis.js do prostředí React v komponentě *VisWrapper* . 48
7.11 Vytvoření legendy ze serverových dat a její vykreslení na element Canvas v komponentách *GraphViewTimeFrame* a *GraphViewMoment* . . . 49
7.12 Obarvení hran představující vztah, který neplatí po celý interval vybrané časové validity, v komponentě *GraphViewTimeFrame* . . . 50
7.13 Filtrování hran a vrcholů na základě jejich validity v konkrétním časovém okamžiku v komponentě *GraphViewMoment* . . . 51
7.14 Struktura funkce vytvářející clustery v komponentách *GraphViewTimeFrame* a *GraphViewMoment* . . . 52
7.15 Vytvoření clusterů nejvyšší úrovně v komponentách *GraphViewTimeFrame* a *GraphViewMoment* . . . 52
7.16 Implementace funkce reagující na událost kliknutí na vrchol v grafu v komponentách *GraphViewTimeFrame* a *GraphViewMoment* . . . 53
7.17 Funkce určující vytváření subclusterů podle definice v serverové odpovědi v komponentách *GraphViewTimeFrame* a *GraphViewMoment* . . . 54

Kapitola 1

Úvod

S rozvojem informačních technologií a postupným zvětšováním objemu generovaných dat začalo být nutné využívat nástroje a techniky, pomocí kterých je možné uložená data snadno zkoumat a pochopit vztahy mezi nimi. Jedním z takových nástrojů je datová vizualizace.

Vizualizace dat v různých formách existuje na světě již mnoho staletí – v současnosti ji lze často vidět ve formě například sloupcových nebo koláčových grafů. Existuje ovšem i oblast vizualizace týkající se vztahů mezi objekty. Vizualizace v tomto případě nabývá formy grafů složených z vrcholů, které jsou spojeny hranami.

Tato práce se věnuje právě vizualizaci grafů představující vztahy – konkrétně vztahy mezi entitami na energetickém trhu reprezentující strukturu trhu v daném regionu. Konkrétní doménou, pro kterou je vizualizace vytvářena, je sdružení Nordic Balance Settlement, které má na starosti udržování výkonové rovnováhy elektrické sítě ve Skandinávii. Cílem je vytvořit nástroj ve formě webové aplikace, který bude výše zmíněné obchodní struktury odpovídajícím způsobem vizualizovat jako orientovaný graf, a který bude možné použít v již existujícím informačním systému i dalších podobných systémech pracujících se složitější reprezentací energetického trhu. Vývoj aplikace probíhal ve spolupráci se zaměstnanci společnosti Unicorn Systems, která toto zadání vytvořila.

V rámci této práce je nejprve v kapitole 2 stanoven cíl práce a stručně popsána řešená problematika. Následně se práce v kapitole 3 věnuje základním pojmům a principům vizualizace grafových struktur. Dále je v kapitole 4 představena struktura a činnost uskupení Nordic Balance Settlement. Následně jsou v kapitolách 5 a 6 popsány technologie využité při tvorbě aplikace, včetně stručného srovnání dostupných knihoven pro grafovou vizualizaci. Samotný návrh řešení a implementace aplikace jsou zpracovány v kapitole 7.

Kapitola 2

Cíl práce

Cílem práce je navrhnout a implementovat nástroj, který bude vizualizovat vztahy mezi entitami na energetickém trhu. Vývoj v této práci se vztahuje zejména k doméně sdružení Nordic Balance Settlement (NBS), pro kterou byla zadavatelem poskytnuta testovací data. Nástroj by měl být následně využit jako součást stávajícího informačního systému Balance Settlement System (BASSE) dodávaného firmou Unicorn Systems. BASSE zajišťuje podporu byznys procesů v rámci NBS a je tak důležitým prvkem fungování energetického trhu ve Skandinávii. Mezi procesy patří například plánování produkce a spotřeby energie, její směna a fakturace nebo měření současného stavu. BASSE tvoří rozhraní pro operátory a účastníky trhu, kteří zde mohou zadávat a kontrolovat data nutná pro správné fungování NBS. Dále tato data spravuje a v rámci funkcionality je poskytuje externím aplikacím. Uživatelské rozhraní informačního systému BASSE lze vidět v příloze A na obrázku A.1.

V rámci působení NBS vstupuje do procesu několik druhů entit s různými typy odpovědnosti – od účastníků na trhu po elektrárny samotné. V informačním systému BASSE ovšem dosud neexistuje žádná možnost grafické reprezentace vazeb mezi těmito entitami. Vytvořený vizualizační nástroj by měl rozšířit informační systém o tuto funkcionalitu a usnadnit tak uživatelům orientaci ve struktuře energetického trhu.

2.1 Základní požadavky

Požadavky na aplikaci byly postupně doplňovány a upřesňovány během konzultací se zaměstnanci firmy Unicorn Systems. Jak bylo zmíněno výše, aplikace by měla být nasazena jako součást informačního systému BASSE. Proto byla jako forma zvolena webová aplikace, která bude nasazena na webovém serveru a bude tak dostupná přes odkaz z informačního systému. Jako prostředek pro implementaci byl stanoven programovací jazyk JavaScript s využitím knihovny React.js. K zobrazování grafových struktur bylo dále nutné vybrat a použít vhodnou vizualizační knihovnu. Pro korektní získávání dat ze serveru bylo také nezbytné navrhnout strukturu odpovědi ve formě objektu JSON.

V práci jsou dále řešeny i požadavky týkající se podoby vztahů mezi entitami na energetickém trhu. Implementovaná vizualizace musí být schopna zpracovat stovky až tisíce entit tak, aby informace o jejich vztazích byly stále přehledně rozeznatelné. Tyto vztahy se dále mění v závislosti na čase, při návrhu tak bylo nutno brát v potaz i tuto problematiku.

Kapitola 3

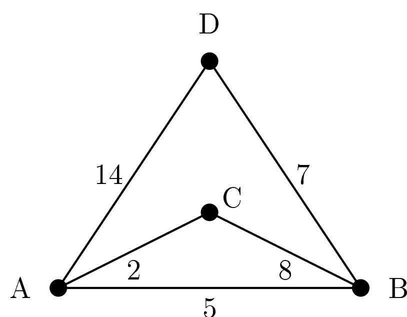
Grafy a jejich vizualizace

Jelikož se tato práce zabývá vizualizací grafových struktur, je vhodné uvést základní pojmy týkající se grafů a grafové vizualizace. Zmíněny budou různé typy grafů, možnosti reprezentace grafové struktury v počítači a také metody jejich vykreslení.

3.1 Graf

Graf je v kontextu teorie grafů diskrétní matematická struktura skládající se z vrcholů a hran, které tyto vrcholy propojují. Grafová struktura může vyjadřovat souvislosti a vztahy mezi různými objekty – například vztahy v sociálních sítích, závislosti molekul v biologii nebo abstrakci energetické sítě. Vrcholy zde představují entity, hrany pak vyjadřují vztahy mezi nimi. Grafy si pro svoji názornost a snadné zpracování počítačem vydobyly velkou oblíbenost v informatických oborech.

Obyčejný graf lze formálně definovat jako uspořádanou dvojici $G = (V, E)$, kde V značí neprázdnou množinu vrcholů, E jako množina hran přestavuje množinu vybraných dvouprvkových podmnožin množiny V [1]. Příklad takového grafu je na obrázku 3.1.



Obrázek 3.1: Neorientovaný graf

Dalším základním pojmem je kružnice v grafu. Kružnicí rozumíme graf na n vrcholech ($n \geq 3$), kde jsou navzájem různé vrcholy postupně spojeny n hranami a poslední vrchol je spojen s prvním. Kružnice se značí C_n, kde n je její délka.

V grafu může být také někdy zkoumán počet sousedů určitého vrcholu. Tento počet se nazývá stupeň vrcholu. Stupněm vrcholu v v grafu G tedy rozumíme počet hran obsahujících vrchol v, a značíme ho $deg_G(v)$. S tímto pojmem souvisí významná vlastnost grafů

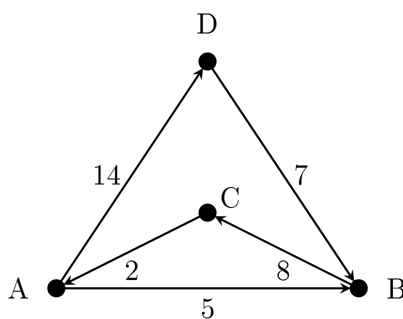
– princip sudosti. Ten říká, že pro každý graf $G = (V, E)$ platí:

$$\sum_{v \in V} deg_G(v) = 2|E|.$$

Důsledkem tohoto tvrzení je, že počet vrcholů lichého stupně v každém grafu je sudé číslo [2].

3.1.1 Orientovaný graf

V případě nutnosti vyznačit směr vztahu mezi vrcholy (například jednosměrka v silniční síti) lze definovat orientovaný graf, ve kterém jsou hrany uspořádané dvojice vrcholů. V obrázku jsou pak hrany kresleny jako šipky. Formálně tedy lze tedy orientovaný graf definovat jako uspořádanou dvojici $G = (V, E)$, kde $E \subseteq V \times V$ [1]. Příklad orientovaného grafu je na obrázku 3.2.

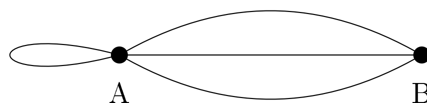


Obrázek 3.2: Orientovaný graf

Ve znační hran je nutné dbát na pořadí vrcholů – například hrana (a, b) z grafu G začíná ve vrcholu a a končí ve vrcholu b. Hrana (b, a) je od hrany (a, b) různá, na rozdíl od neorientovaného grafu. S tím také souvisí rozlišování vstupního a výstupního stupně vrcholů v grafu. Vstupní stupeň je značen $deg_G^+(v)$, výstupní $deg_G^-(v)$. Celkový stupeň vrcholu je pak vypočten jako jejich součet.

3.1.2 Multigraf

Speciálním případem grafu je tzv. multigraf, kde mezi dvěma vrcholy může vést několik stejně orientovaných (nebo neorientovaných) hran. Hrany také mohou začínat i končit ve stejném vrcholu. Multigraf je vyobrazen na obrázku 3.3.

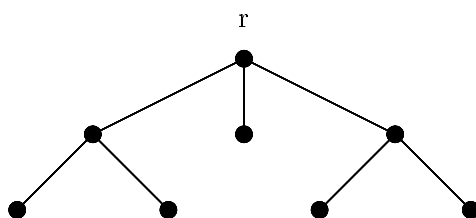


Obrázek 3.3: Multigraf

3.1.3 Strom

Strom lze definovat jako souvislý graf T, kde T neobsahuje žádnou kružnici. Speciálním typem stromu je kořenový strom, což je dvojice (T, r), kde $r \in V(T)$ je jeden zvolený vrchol zvaný kořen. Pro takový graf pak platí, že z určeného kořene vede do každého vrcholu právě jedna cesta. Sousední vrcholy na této cestě jsou pak označovány jako předchůdce a následovník, rodič a potomek apod. Takový strom lze vidět na obrázku 3.4. Speciálním typem kořenového stromu je binární strom, pro jehož vrcholy platí, že každý má maximálně dva následovníky. [1]

Stromové struktury můžeme najít v mnoha aplikacích, známým příkladem mohou být rodokmeny nebo struktura souborového systému. Stromy jsou také spojeny s významnou problematikou teorie grafů – kostrou grafu. Kostra souvislého grafu G je takový podgraf v G, který obsahuje všechny vrcholu grafu G a zároveň je stromem [1].



Obrázek 3.4: Kořenový strom

Pro každý graf stromu také platí, že je rovinný – lze ho nakreslit (v rovině) bez křížení hran.

3.2 Reprezentace grafů

Při vytváření grafů v počítačových aplikacích je nutné využít vhodné struktury k jejich definici. Zde je nutné rozlišit uložení grafu v datových strukturách programovacích jazyků a textové formáty generované počítačovými programy pro další použití. První skupina je založena na matematických principech a využívá struktur jako pole nebo matice. Druhá skupina reprezentuje grafy v textové podobě, kterou lze použít například pro sdílení vytvořeného grafu mezi počítačovými programy podporujícími daný formát.

3.2.1 Datové struktury

Grafy uložené v datových strukturách programovacích jazyků se liší principy fungování dané datové struktury, což má za následek rozdílnou časovou náročnost při manipulaci s grafem a paměťovou náročnost uložení v dané datové struktuře. Zde jsou představeny možnosti uložení grafu pomocí matic sousednosti a incidence, seznamu sousedů a seznamu hran.

3.2.1.1 Seznam hran

Nejjednodušší možnou strukturou k uložení grafu je pole párů, kde jednotlivé hodnoty v párech označují spojené vrcholy. Tato struktura je omezena principy implementace pole

jako datové struktury – například pro zjištění existence spojení dvou vrcholů hranou by bylo nutné iterovat v nejhorším případě přes celé toto pole (výpočetní složitost $\mathcal{O}(|E|)$).

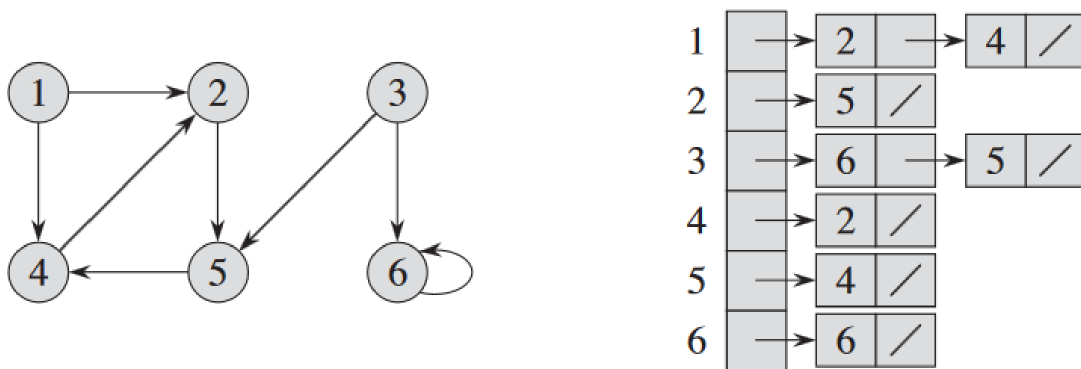
3.2.1.2 Matice sousednosti

Matice sousednosti představuje dvourozměrné pole A, které obsahuje $n \times n$ (n značí počet vrcholů) booleovských hodnot (nul a jedniček). Hrana mezi uzly i a j je pak definována jedničkou na pozici A_{ij}. Pokud jsou na hranách zaznamenávány určité hodnoty (např. vzdálenost mezi uzly), pak jsou jedničky v matici nahrazeny těmito hodnotami. Při popisu neorientovaného grafu je výsledná matice symetrická – lze tak brát v potaz pouze odpovídající trojúhelníkovou matici. [3]

Ačkoliv má kontrola existence hran, jejich přidávání a odebírání v této reprezentaci výpočetní složitost $\mathcal{O}(1)$, přidání každého vrcholu způsobí zvětšení matice o řádek i sloupec. Tato reprezentace s paměťovou náročností $|V^2|$ tak není vhodná pro definici rozsáhlých grafů. Ukázku matice sousednosti obsahuje obrázek 3.6.

3.2.1.3 Seznam sousedů

Další možností reprezentace grafu je seznam sousedů. Jedná se o datovou strukturu, kde pro každý vrchol grafu existuje spojový seznam jeho sousedů. Pokud tedy existuje hrana z vrcholu i do vrcholu j, pak je vrchol j uveden v seznamu vrcholu i. Oproti matici sousednosti je seznam sousedů méně náročný na paměť – jeho náročnost je $|V| + |E|$. Tato výhoda se prohlubuje v případě grafu, kde jsou vrcholy propojeny pouze řídce. Přidání elementů je provedeno se složitostí $\mathcal{O}(1)$ – jedná se pouze o prodloužení pole vrcholů nebo jejich příslušných seznamů [3]. Ukázka uložení grafu jako seznam sousedů je vyobrazeno na obrázku 3.5.

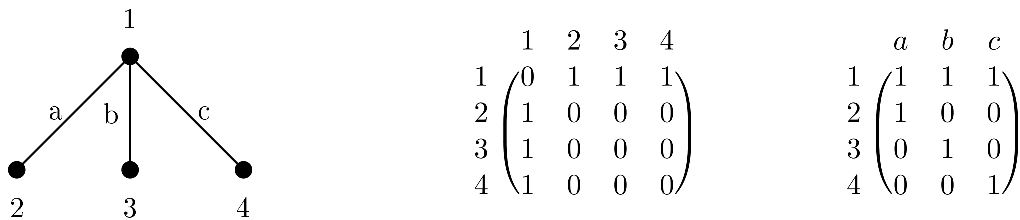


Obrázek 3.5: Definice grafu (vlevo) pomocí seznamu sousedů (vpravo), zdroj: [4]

3.2.1.4 Matice incidence

Poslední zmíněnou formou uložení grafu je matice incidence. Jedná se o dvojrozměrné pole, kde tentokrát existuje n řádků a m sloupců. Řádky reprezentují vrcholy, sloupce pak hrany. Existence hrany je znovu určena nenulovou hodnotou. Ukázka matice incidence je na

obrázku 3.6. Nevýhodou je stejně jako u matice sousednosti náročnost na paměť (náročnost $|V| \cdot |E|$), jelikož je nutné rozšiřovat matici při přidávání nových elementů. [3]



$$\begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \end{array} \begin{array}{c} \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \begin{pmatrix} 0 & 1 & 1 & 1 \\ 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{pmatrix} \end{array} \qquad \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \end{array} \begin{array}{c} \begin{array}{ccc} a & b & c \end{array} \\ \begin{pmatrix} 1 & 1 & 1 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} \end{array}$$

Obrázek 3.6: Příklad matice sousednosti (uprostřed) a matice incidence (vpravo) popisující ukázkový graf (vlevo)

3.2.2 Textové formáty

Textové formáty představují standardizovaný prostředek pro reprezentaci grafových struktur. Uživatelé mohou soubory v daném formátu využít pro vložení grafu do aplikace pro tvorbu a zpracování grafů, pokud tato aplikace tento formát podporuje. Samotné zpracování grafu ve zdrojovém kódu, například pomocí datových struktur uvedených výše, je pak v každé aplikaci implementováno různě – textové formáty tak představují určitou abstrakční vrstvu pro sdílení vytvořených grafů.

Mnoho textových formátů je založeno na struktuře Extensible Markup Language (XML). Známými zástupci takových formátů jsou:

- GraphML – univerzální XML formát, jehož vývoj začal v roce 2000, a jehož struktura byla navržena v rámci konference Graph Drawing Symposium ve Vídni v roce 2001. Tento formát byl navržen i s ohledem na použití ve webových službách. [5]
- GEXF – tento formát, jehož vývoj začal v roce 2007, je využíván ve známém vizualizačním programu Gephi. [6]
- GXL – formát GXL byl poprvé představen v roce 2000 jako standard pro sdílení informací mezi nástroji reinženýringu. Pro jeho obecnost ho ovšem následně bylo možné použít i v rámci aplikací pro grafovou vizualizaci. [7]

Příklad formátu GraphML je uveden v ukázce 3.1.

```
<graphml>
  <graph id="G" edgedefault="undirected">
    <node id="n0"/>
    <node id="n1"/>
    <edge id="e1" source="n0" target="n1"/>
  </graph>
</graphml>
```

Ukázka kódu 3.1: Reprezentace grafu ve formátu GraphML

Existují i textové formáty s vlastní syntaxí nebo ve formátech, které nebyly založeny na formátu XML. Mezi vybrané zástupce patří následující:

- jazyk DOT – DOT je ústředním formátem pro reprezentaci grafu v programu GraphViz. Soubory vytvořené v tomto formátu mají koncovku *gv* nebo *dot*. Tyto soubory pak mohou být dále zpracovány nástroji pro vytváření layoutů – například dot, neato, twopi nebo circo. [8]
- GML – formát GML byl dalším pokusem o zavedení standardizovaného formátu pro účely reprezentace libovolných datových struktur, stejně jako formát GXL je ale vhodný i pro popis grafů. Původní návrh byl předložen již v roce 1997, i přesto je ale tento formát podporován i v současnosti (například v aplikaci Cytoscape). [9]
- JSON formát – v prostředí JavaScriptových knihoven se často využívá reprezentace grafu ve formátu souboru JSON. To usnadňuje práci s grafem, jelikož není třeba soubor transformovat po jeho přenosu skrz síť.

Příklad definice pomocí jazyka DOT je v ukázce 3.2. Lze vidět, že v notaci DOT lze definovat i struktury jako podgraf nebo cluster.

```
digraph UkazkovyGraf {
  subgraph cluster_0 {
      label="Subgraph A";
      a -> b;
      b -> c;
      c -> d;
  }
  subgraph cluster_1 {
      label="Subgraph B";
      a -> f;
      f -> c;
  }
}
```

Ukázka kódu 3.2: Reprezentace grafu ve formátu DOT

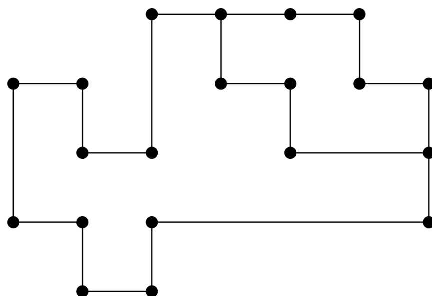
3.3 Vizualizace grafů

Vizualizace grafových struktur může být pro pozorovatele účinným prostředkem k pochopení vztahů v zachycené oblasti. Samotnou vizualizaci je ovšem nutno provést správným způsobem vzhledem ke kontextu zkoumaných vztahů. Například síť metra je nutné zobrazit s ohledem na skutečné umístění stanic ve městě, rodokmen je zase vhodné zobrazovat důsledně v hierarchické struktuře se vztahy uspořádanými dle jejich stáří. Tyto aspekty jsou zkoumány ve vědní oblasti s anglickým názvem *Graph drawing*. Ta se zabývá algoritmy pro tvorbu rozložení vrcholů a hran v grafu, vzhledem elementů, a klade důraz na celkovou kvalitu vizualizace z hlediska předání informací pozorovateli.

V průběhu rozvoje oboru grafové vizualizace vzniklo nepřeberné množství algoritmů sloužících k optimálnímu rozložení vrcholů (vytvoření layoutu). Existuje několik stěžejních metod vykreslování, do kterých lze jednotlivé algoritmy zařadit – mezi nejznámější patří ortogonální, hierarchické nebo vykreslení za pomoci analogie fyzikálních sil.

3.3.1 Ortogonální vykreslení

Metoda ortogonálního vykreslení grafu je využívána v oblastech, které upřednostňují snadné rozlišení vizualizovaných hran. Hrany v ortogonální reprezentaci mohou vést pouze horizontálně a vertikálně. Pokud hrana mění směr, vytvoří se v tomto místě zlom. Tento styl vizualizace je využíván například k zobrazení integrovaných obvodů, jelikož dává důraz na to, aby nedocházelo ke křížení hran [10]. Ukázka grafu v ortogonální reprezentaci je na obrázku 3.7.



Obrázek 3.7: Ortogonální vizualizace grafu, zdroj: [10]

Ortogonální vykreslování je tak spjato s oblastí rovinných grafů. Řešena je problematika zakódování rovinných grafů a rovinných grafů v ortogonálním tvaru – zde jsou navíc potřeba informace o zlomech na hranách. Dále existují algoritmy řešící problém rovinného grafu, ve kterém existuje vrchol se stupněm větším než 4 – takový graf nelze nakreslit bez křížení hran při požadavku na vrchol, se kterým jsou hrany spojeny pouze horizontálním nebo vertikálním směrem. Mezi další zkoumané problematiky patří v této oblasti například algoritmy k minimalizaci počtu zlomů. [10]

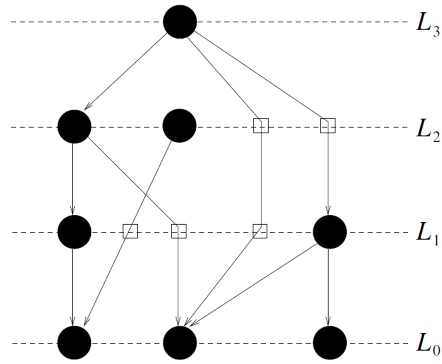
3.3.2 Hierarchické vykreslení

Tato metoda grafové vizualizace se zabývá rozmístěním vrcholů orientovaného grafu do vrstev, ideálně podle hierarchické struktury, pokud taková existuje. Důraz je dán na orientaci hran stejným směrem. Metoda je známá v anglické literatuře i pod jinými názvy – *layered graph drawing* nebo *Sugiyama-style graph drawing*. Právě Kozo Sugiyama byl prvním vědcem, který tuto problematiku zpracoval ve své práci z roku 1981.

Sugiyamaova metoda zpracování grafu do hierarchické struktury se skládá ze čtyř hlavních kroků. Nejprve je z původního grafu vytvořen acyklický orientovaný graf. To je dosaženo buď dočasným obrácením orientace hran vytvářejících cyklus, nebo jejich odstraněním. Tato operace je nutná pro druhý krok metody – rozmístění vrcholů do vrstev. Po rozmístění je možno vrátit odstraněné nebo otočené hrany do původního stavu. Pro dlouhé hrany je možné využít pomocných vrcholů, které vytvoří zlomy na hranách a zlepší tak výslednou vizualizaci. Tento krok lze vidět na obrázku 3.8. [11]

Počet pomocných vrcholů by měl být co nejmenší, jelikož jejich vysoký počet výrazně zpomaluje další kroky vizualizace. Toho je dosaženo například metodami lineárního programování. [11]

Po dokončení rozvrstvení následuje nepovinný krok, ve kterém dochází ke zmenšení



Obrázek 3.8: Vrstvení grafu s využitím pomocných vrcholů pro lepší vizualizaci hran, zdroj: [11]

hustoty hran mezi sousedními úrovněmi. To je dosaženo přidáním nových vrcholů, které fungují jako pomocné uzly pro přerozdělení hran. [11]

Posledním důležitým krokem je redukce křížení hran. Toho je dosaženo přeuspořádáním vrcholů v jednotlivých vrstvách. Jelikož je nalezení minimálního počtu křížení NP-úplný problém, jsou v tomto kroku využívány různé heuristiky. [11]

3.3.3 Vykreslení pomocí fyzikálních analogií

Využití fyzikálního modelu pro vytvoření layoutu je flexibilní metodou grafové vizualizace. V anglické literatuře je tato oblast známa pod názvy *force-directed* algoritmy nebo *spring embedders*. Základní myšlenkou je zavedení působení fyzikálních sil mezi vrcholy a hranami a následná minimalizace energie v tomto systému. Vizualizace vytvořené touto metodou většinou dávají dobré estetické výsledky, vytvářejí symetrické rozložení a jsou schopny vytvořit zobrazení bez křížení hran u rovinných grafů. [12]

Základní princip ve fyzikálním modelu spočívá v odpudivém působení vrcholů, hrany se naopak chovají jako pružiny s určitou přitažlivou silou. Jelikož jsou tyto výpočty prováděny pomocí matematických rovnic, existuje i mnoho modifikací s různým chováním. Mezi známé zástupce algoritmů patří například:

- Fruchterman-Reingold – tento algoritmus se řídí podle dvou pravidel – vrcholy spojené hranou by měly být blízko, ale zároveň by neměly být blízko až příliš. Algoritmus zavádí novou proměnnou nazvanou *teplota*, která kontroluje rozmístění vrcholů v grafu. Proměnná začíná na určité hodnotě a s probíhajícím algoritmem se tato hodnota postupně snižuje. Zároveň, čím lepší je rozložení layoutu, tím menší změny teploty nastávají. [12]

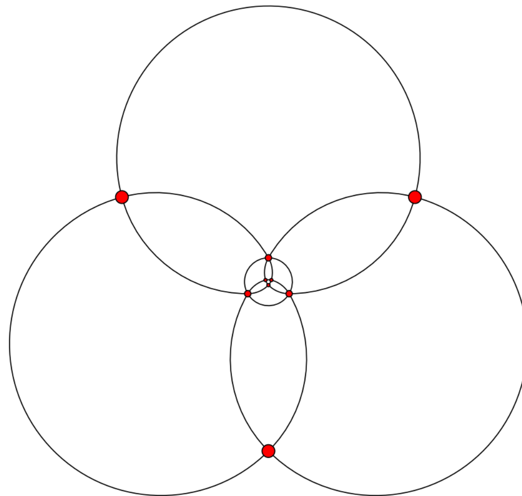
- Kamada-Kawai – na rozdíl od dvou pravidel zmíněných v předchozím algoritmu je zde rozmístění vrcholů založeno na výpočtu teoretických (euklidovských) vzdáleností. Tyto vzdálenosti jsou vypočteny jako nejkratší cesty mezi každými dvěma vrcholy. Tato operace je ovšem výpočetně náročná - její složitost je až $\mathcal{O}(|V|^3)$ při použití Floyd-Warshallova algoritmu. [12]

- Davidson-Harel – podobně jako Fruchterman-Reingold pracuje s teplotou grafu, dále

přidává požadavky na minimalizaci křížení hran a zabránění vrcholům být příliš blízko nesousedních hran. Algoritmus využívá techniky simulovaného žíhání, což je pravděpodobnostní optimalizační metoda hledající globální extrémy funkce. Simulované žíhání je využito k nalezení minima vybrané funkce energie působící na vrcholy a hrany. [12]

- ForceAtlas2 – tento poměrně nový algoritmus byl vytvořen pro potřeby vizualizačního programu Gephi. Podle tvůrců ho lze použít pro grafy s až 100 000 vrcholy. Zajímavostí tohoto algoritmu je, že jeho výpočty probíhají až do explicitního zastavení uživatelem – v rámci vizualizace lze tedy sledovat celý průběh rozmisťování vrcholů grafu. ForceAtlas2 ve výpočtech působících sil bere v potaz stupeň vrcholu, cílem je držet vrcholy s nízkým stupněm blízko vrcholů s vysokým stupněm. To je dosaženo snížením odpudivé síly mezi těmito dvěma typy vrcholů. Pokud chce uživatel ovlivnit rozložení grafu, může využít několika nabízených nastavení, jako změna gravitace nebo možnost zahrnout do výpočtu layoutu váhy na hranách. [13]

Existují i další přístupy a algoritmy využívané při vykreslování pomocí fyzikálních analogií (zmíněny v [12]). Barycentrická metoda využívá k výpočtu soustavu lineárních rovnic, kde je každý vrchol reprezentován jako konvexní kombinace poloh jeho sousedů (algoritmus Tutte). Neeuklidovský přístup neprovádí výpočet rozložení vrcholů v rovině, ale je využíváno například sférické nebo hyperbolické geometrie. Lombardiho metoda vyobrazuje hrany jako kruhové oblouky, čímž maximalizuje úhly mezi hranami každých dvou vrcholů. Ukázka vykreslení pomocí této metody je na obrázku 3.9. Navrženy byly dále i přístupy pro rozsáhlé grafy (např. algoritmus Harel-Koren) a dynamické grafy.



Obrázek 3.9: Graf vykreslený pomocí Lombardiho metody, zdroj: [14]

3.3.4 Vlastnosti kvalitní grafové vizualizace

Při vytváření co nejkvalitnější vizualizace grafu je nutno brát ohled na několik společných kritérií. Některé z nich byly již zmíněny při popisu výše zmíněných přístupů vytváření layoutu grafu při jeho vykreslení. Tato společná kritéria shrnuje [15]:

- Minimalizace křížení hran – pokud zobrazovaný graf není rovinný, je vhodné omezit počet zkřížených hran na minimum, jelikož znesnadňují orientaci v grafu.
- Omezení zlomů a ohybů na hranách – pro pozorovatele je snazší sledovat hrany, které mají rovný tvar. Toho se využívá u ortogonálních layoutů a výsledná vizualizace pak může být využita například pro zobrazení elektrického obvodu.
- Minimalizace zabírané oblasti – výsledný graf by měl zabírat co nejmenší oblast nutnou k přehledné vizualizaci. Také hustota vrcholů v prostoru by měla být homogenní. Jako příklad lze uvést kapesní vizualizaci dopravní mapy.
- Omezení překrývajících se hran – překrývání hran může způsobit problémy zejména na přístrojích s malým rozlišením. Úhel svíraný dvěma křížícími se hranami by tedy měl být co největší.
- Minimalizace délky hran – délku hran je vhodné zmenšit na nezbytně nutnou vzdálenost.
- Symetrie – pokud graf obsahuje informace o symetričnosti, je vhodné ji zobrazit i ve vizualizaci. Symetrie mohou obsahovat například technické výkresy.
- Zvýraznění skupin – při vizualizaci objemných grafů je vhodné vyznačit související skupiny entit.

Posledním zde zmíněným kritériem přispívajícím k uživatelsky přívětivé vizualizaci je redukce složitosti grafu. Tuto problematiku je nutné řešit u rozsáhlých grafů, které by při zobrazení v celé své velikosti zabránily uživateli v efektivní vizuální analýze. Příkladem oboru řešící tento problém je biologie, kde jsou grafy využívány například k vizualizaci proteinových struktur, jak ukazují autoři v [16].

Redukce složitosti grafu může být v menší míře redukována pomocí technik jako je přibližování a oddalování objektů nebo pohybem mezi částmi grafu. Existují ovšem i algoritmy, které mění strukturu celého grafu. Autoři v [17] představují tři možné způsoby, pomocí kterých lze vhodně upravit grafovou strukturu – jedná se o metodu rozbalování a sbalování vrcholů (*expand / collapse*), skládání vrcholů (*folding*) a skrytí nevýznamné části topologie.

První způsob využívá rozdělení vrcholů do podgrafů, které lze v rámci interakce sbalit do jediného elementu, který ve vizualizaci zachová všechny hrany představující spojení sbalených vrcholů s vrcholy vnějšími.

Technika skládání vrcholů je založena na seskupení vrcholů s určitou charakteristikou do jednoho vrcholu (na rozdíl od předchozí techniky nemusí být vrcholy členy jednotlivých podgrafů). Tento princip je využit i v rámci této práce pod názvem *clustering*.

Metoda skrytí nevýznamné části topologie představuje buď úplné odstranění vrcholů z grafu nebo využití metody pojmenované autory jako *ghosting*, která vizuálně sníží důležitost určených vrcholů například pomocí jejich zprůhlednění.

Kapitola 4

Sdružení Nordic Balance Settlement

Vizualizační nástroj, který je vytvářen v rámci této práce, je zaměřen na zobrazování vztahů mezi subjekty participujícími v rámci sdružení Nordic Balance Settlement (dále jen NBS). V této kapitole bude popsán účel tohoto uskupení, jeho účastníci a vztahy mezi nimi. Tato kapitola byla zpracována zejména podle dokumentu NBS Handbook [18] a informací poskytnutých firmou Unicorn Systems.

4.1 Činnost NBS

Uskupení NBS započalo své fungování v roce 2010 po dohodě třech provozovatelů přenosových soustav – Fingrid (Finsko), Svenska kraftnät (Švédsko) a Statnett (Norsko). Činnost těchto provozovatelů spočívá v udržování výkonové rovnováhy v rámci elektrické sítě. Tu je nutné udržovat neustále, aby nedocházelo k výpadkům elektřiny, nebo naopak přetěžování přenosové soustavy. V běžném provozu ovšem nastávají výkyvy mezi spotřebou a dodávkou elektrické energie. Dodávky může ovlivnit například špatné plánování, poruchy při vytváření energie nebo problémy v přenosové soustavě. NBS tedy poskytuje společný systém pro měření, vypořádání nerovnováhy, fakturování a reportování pro všechny účastníky působící ve Finsku, Švédsku a Norsku.

4.2 Participující subjekty

V rámci NBS participuje velký počet organizací podílejících se na fungování energetického trhu. Každá organizace může na trhu plnit některou z rolí (nebo i více z nich), které jsou popsány v této kapitole. Pro usnadnění orientace v obrázcích a později testovacích datech vizualizačního nástroje jsou ponechány názvy subjektů včetně jejich zkratek v anglickém jazyce.

4.2.1 Transmission System Operator (TSO)

V překladu již zmínění provozovatelé přenosových soustav. Tito operují na soustavě vysokého napětí a koordinují poptávku a dodávku v síti tak, aby bylo dosaženo požadované frekvence 50 Hz. Řídí se zákony jednotlivých zemí. Uvnitř NBS se tedy jedná o společnosti Fingrid, Svenska kraftnät a Statnett.

4.2.2 Imbalance Settlement Responsible (ISR)

Tuto roli má pouze finská firma eSett. Ta byla založena v roce 2017 a je vlastněna rovným dílem třemi TSO zmíněnými výše. Firma eSett provádí činnosti spojené se sbíráním, validací a řízením dat fungování NBS a tato data poskytuje ostatním účastníkům na energetickém trhu. Tento trh dále monitoruje a také poskytuje zákaznickou podporu a reporting. Firmu eSett lze tedy označit jako společné rozhraní poskytující služby jménem třech výše zmíněných vlastníků.

4.2.3 Balance Responsible Party (BRP)

BRP je subjekt, který má platnou dohodu s firmou eSett a místním TSO. Jeho závazkem je průběžně plánovat a dosahovat rovnováhy mezi spotřebovanou a dodanou energií mezi producenty, spotřebiteli a obchodníky. Pokud BRP produkuje méně elektřiny, než bylo naplánováno, musí zbytek potřebného objemu nakoupit od firmy eSett. Při přebytku naopak BRP svoji energii prodává. Subjekt BRP je tedy finančně odpovědný za rozdíly v realizované produkci a spotřebě v porovnání s plánem. Požadavky na tuto činnost jsou definovány určeným TSO v konkrétní oblasti působnosti.

4.2.4 Retailer (RE)

Zde jde o obchodníky s elektřinou, kteří se zabývají jejím nákupem a prodejem zákazníkům. Obchodníci musí mít dohodu s BRP pro produkci a spotřebu energie v každé měřící oblasti (viz. MGA), kde operují. Výjimkou je Finsko, kde obchodníkům stačí mít dohodu s jiným obchodníkem, který má dohodu s BRP. Obchodníci mohou mít dohodu s různými BRP v různých měřících oblastech jak pro spotřebu, tak i pro produkci. Princip těchto vztahů je ukázán na obrázku 4.1 na příkladu Finska (pouze ukázka – nejde o reálnou strukturu).

4.2.5 Distribution System Operator (DSO)

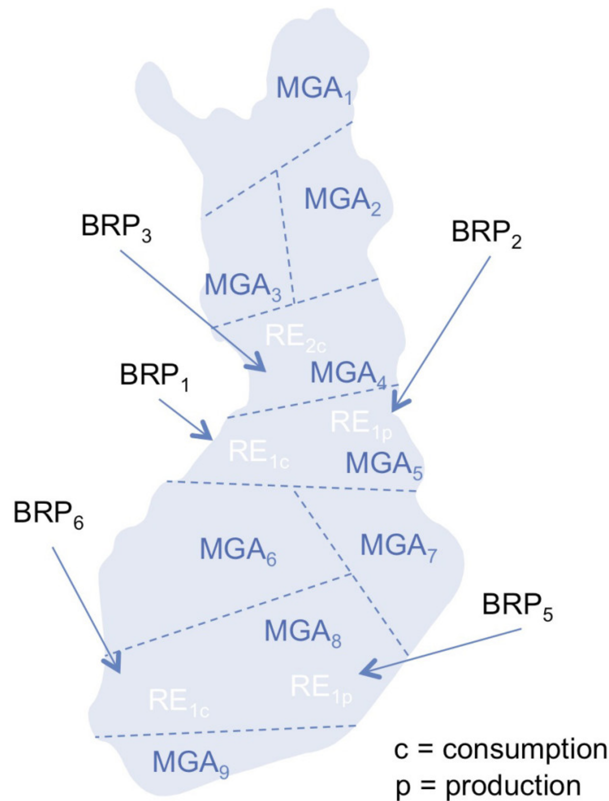
Zkratka DSO označuje provozovatele distribuční sítě. Každý provozovatel má na starost distribuci elektřiny od producentů k zákazníkům. Dále obsluhuje měřící systém v jednotlivých MGA a naměřené hodnoty odevzdává oprávněným účastníkům. Na základě těchto dat vypočítá eSett nerovnováhu za každou BRP.

4.3 Ostatní entity

V rámci fungování energetického trhu jsou součástí NBS další entity, pomocí kterých lze specifikovat další informace týkající se produkce a spotřeby elektrické energie.

4.3.1 Metering Grid Area (MGA)

MGA je fyzická oblast, kde může být měřena spotřeba a (nebo) produkce energie. Oblast se vymezuje na národní úrovni podle platné legislativy. V rámci MGA existují měřící přístroje, které mají na starost sbírání dat ze tří hlavních oblastí – toky energie mezi sousedními MGA, hodinová produkce uvnitř MGA a hodinová spotřeba energie v dané



Obrázek 4.1: Princip vztahů mezi obchodníky (RE), BRP a MGA, zdroj: [18]

MGA. Jelikož jsou naměřená data důležitá ke správnému vypořádání odchylek, je jejich kvalita monitorována firmou eSett.

4.3.2 Market Balance Area (MBA)

MBA je fyzická oblast, ve které se nachází jedna nebo více MGA. V celé této oblasti je stanovena stejná tržní cena elektřiny, která je použita k prodeji přebytků nebo nákupům pro pokrytí deficitu jednotlivých BRP.

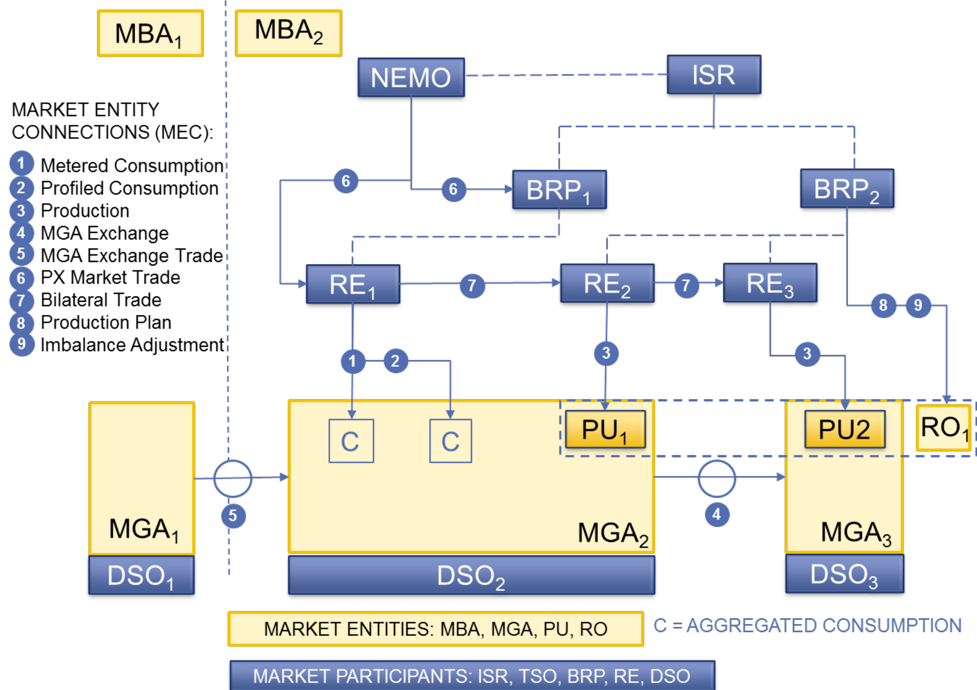
4.3.3 Production Unit (PU) a Regulation Object (RO)

Zde jde o objekty, kterých se týká výroba elektřiny. Produkční jednotky (PU) jsou generátory v rámci jedné elektrárny. Dělí se na dva typy podle objemu produkce – *normal* a *minor*. Regulační objekt (RO) se potom skládá z několika PU, které mají ovšem mají stejnou technologii výroby energie (solární, jaderná apod.).

4.4 Struktura NBS

Jak bylo popsáno výše, struktura NBS je složena ze dvou hlavních částí – subjektů participujících na trhu s elektrickou energií a ostatních entit sloužících ke specifikaci dalších informací o produkci a spotřebě. Tyto dvě skupiny generují data, která jsou pak sdílena jak uvnitř, tak i s druhou skupinou. Tyto vztahy jsou nazývány *Market Entity Connections (MEC)*. Schéma s jednotlivými vztahy MEC mezi různými subjekty je vyobrazeno

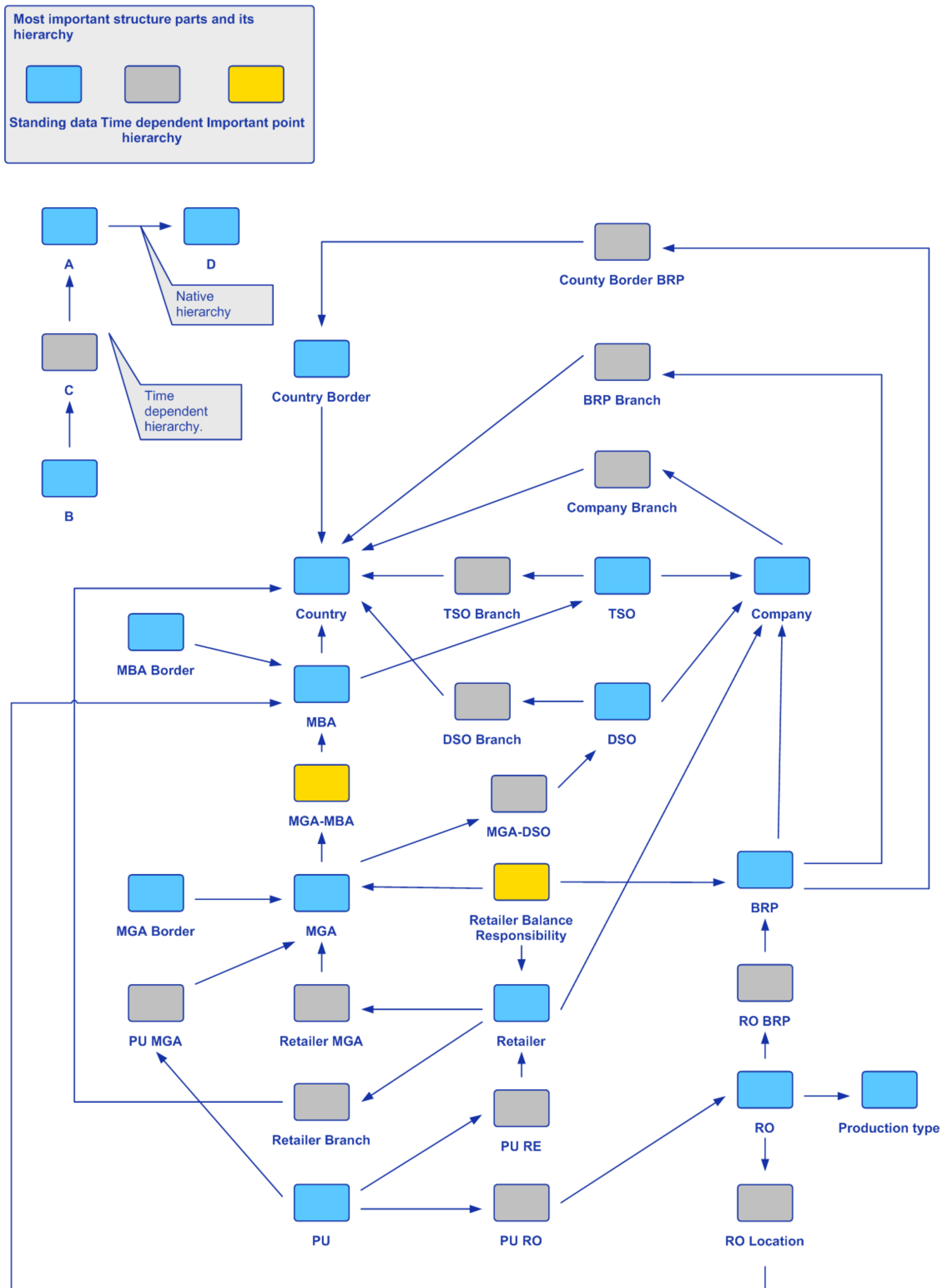
na obrázku 4.2. Součástí obrázku je také dosud nezmíněný subjekt NEMO – *Nominated Electricity Market Operator*. Ten má na starost kalkulaci cen energie na trhu v rámci jedné MBA a následně podává tyto údaje firmě eSett. Dále dohlíží na probíhající obchodování a dodržování regulací a pravidel této činnosti.



Obrázek 4.2: Informační propojení mezi různými subjekty v NBS, zdroj: [18]

Informace o struktuře vztahů účastníků je klíčovou součástí fungování NBS. Každý účastník je tedy zodpovědný za uvedení aktuálních informací o vztazích s jinými subjekty v informačním systému poskytnutém firmou eSett. Důležitým faktem je, že strukturální informace má časově omezenou platnost, v jejímž průběhu je subjekt aktivním účastníkem v systému vypořádání nerovnováhy.

Od firmy Unicorn Systems bylo poskytnuto celkové schéma vytvořené podle testovacích dat používaných pro potřeby vývoje. Schéma je na obrázku 4.3 a jsou na něm vyznačeny časové závislosti vztahů mezi subjekty.



Obrázek 4.3: Schéma vztahů mezi entitami v NBS, zdroj: poskytnuto firmou Unicorn Systems

4.5 Mohutnost vztahů

Z hlediska vývoje vizualizačního nástroje jsou důležité počty vztahů mezi entitami, jež budou muset být zobrazovány. Společností Unicorn Systems byly poskytnuty orientační

očekávané hodnoty – jejich neúplný výčet je uveden v tabulce 4.1. V tabulce je uvedeno deset vztahů s největší mohutností. V tabulce lze také vidět, zda je vztah přímý (nepřímý vztah je možno odvodit ze struktury vztahů – lze vidět na obrázku 4.3).

Vztah	**Kardinalita vztahu**	**Max. hodnota M**	**Max. hodnota N**	**Přímý vztah**
MBA – PU	1:N	1	1000	Ne
Country – PU	1:N	1	1000	Ne
BRP – RE	M:N	10	750	Ne
BRP – PU	1:N	1	568	Ne
MBA – MGA	1:N	1	500	Ano
Country – MGA	1:N	1	500	Ne
RE – PU	1:N	1	489	Ano
Country – Company	M:N	4	379	Ano
Country – RE	M:N	4	357	Ano
RO – PU	1:N	1	295	Ano

Tabulka 4.1: Deset nejmohutnějších vztahů mezi entitami NBS.

V tabulce lze vidět, že mohutnost vztahů může dosáhnout až několika set, nejvyšší hodnota kardinality vztahu mezi dvěma entitami by neměla přesáhnout počet 1000. Z hlediska vizualizace je tak nutné najít vhodný způsob, který zachová dobrou pochopitelnost vztahů mezi entitami. Návrh takového způsobu je popsán v části 7.2.2.

Kapitola 5

Použité technologie

Tato kapitola se zabývá technologiemi, které byly použity při vývoji vizualizačního nástroje. Představen bude programovací jazyk JavaScript včetně jeho ekosystému, dále bude popsána knihovna React spolu s principy, které se v ní využívají. Nakonec budou zmíněny ostatní podpůrné knihovny.

5.1 JavaScript

JavaScript (často zkracován na JS) je jazykem moderního webu. Jedná se o interpretovaný, dynamický, slabě typovaný skriptovací jazyk, který od svého prvního vydání v roce 1995 sloužil zejména v klientské části k vytváření interaktivních webových stránek. V posledních letech se ovšem JavaScript začal prosazovat i ve vývoji serverové části – lze s ním tak vytvořit plně funkční webovou aplikaci. JavaScript lze najít i mimo oblast tvorby webu – například v aplikacích společnosti Adobe nebo některých NoSQL databázích.

Vývoj JavaScriptu a jeho standardizace jsou řízeny společností ECMA International. Jazyk standardizovaný touto společností se nazývá ECMAScript – samotný JavaScript pak lze označit jako implementaci tohoto standardu. Názvy konkrétních vývojových verzí mají dvě podoby, dle roku vydání nebo dle pořadí vydané edice. Například standard z roku 2015 je tak označován buď jako ECMAScript 2015 nebo ECMAScript 6 (běžně zkracováno jako ES6).

Právě verze ES6 přinesla mnoho změn, které měly za cíl zjednodušit dříve používané přístupy a umožnit také využívání prvků funkcionálního programování. Novinky verze ES6 lze najít i ve zdrojovém kódu této práce – jedná se například o [19]:

- Nové definice proměnných a konstant – namísto definice proměnných pomocí klíčového slova *var* je vhodné využít buď slova *const* (proměnné lze přiřadit hodnotu pouze jednou) nebo *let* (hodnotu lze přiřadit vícekrát).

- Šipkové funkce – dovolují vytvořit funkci bez specifikování klíčového slova *function*. Za určitých podmínek není třeba ani příkazu *return*, což snižuje objem nutného kódu. Šipkové funkce lze běžně vidět při implementaci metod *map*, *reduce* nebo *filter*, které slouží k transformaci prvků v poli.

- Moduly – jakýkoliv JavaScriptový datový typ lze přenášet mezi soubory pomocí klíčových slov *export* a *import*. Tento přístup nahrazuje předchozí formát CommonJS.
- Třídy – ve verzi specifikace ES6 se do JavaScriptu dostal i tento prvek známý zejména z objektového přístupu. Třídy podporují dědičnost i vytváření instancí pomocí konstruktorů a v rámci této práce jsou využity v komponentách knihovny React.

5.1.1 Podpůrné nástroje JavaScriptového ekosystému

Při vývoji JavaScriptových aplikací je běžně využíváno technologií, které ulehčují podpůrné činnosti vývoje, jako je příprava zdrojového kódu do produkčního stavu nebo správu závislostí mezi externě využívanými knihovnami. Zde jsou zmíněny technologie využívané v rámci vizualizačního nástroje vyvíjeného v této práci – jedná se o Babel, Webpack, Node.js a Node package manager.

Babel a Webpack jsou důležitými nástroji pro zpracování vytvořeného JavaScriptového zdrojového kódu. Babel se stará o jeho transformaci mezi různými specifikacemi ECMA (tento proces se nazývá transpilování). Jedná se zejména o převod z novější specifikace do starší z důvodu kompatibility s webovými prohlížeči. Interprety webových prohlížečů ve starších verzích totiž nedokáží zpracovat kód vytvořený v nejnovější verzi ECMA specifikaci. [21]

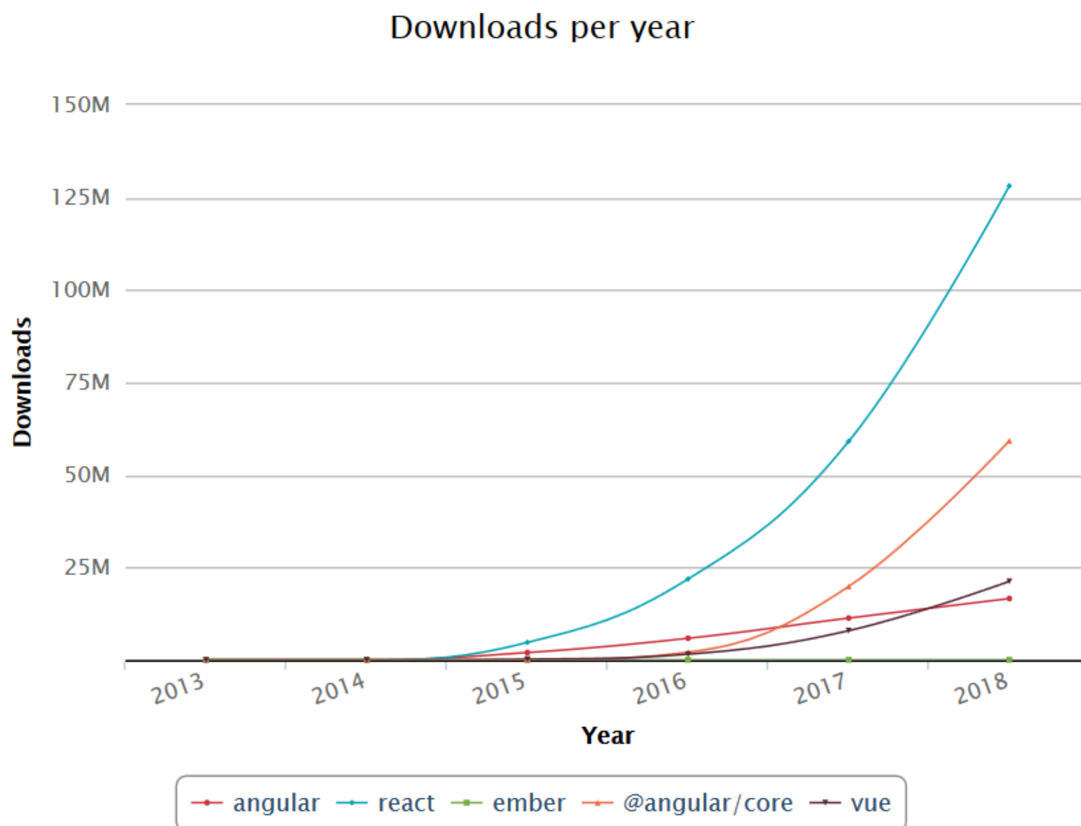
Webpack je sestavovací nástroj, který dokáže zabalit zdrojový kód vytvořený v modulech a připravit ho pro přenos po síti. Pro tento přenos dokáže Webpack zajistit minifikaci kódu (odstranění mezer, zalomení řádků a dalšího nepotřebného kódu) a jeho rozdělení na části, které jsou načítány dle potřeby (code splitting). [19]

Node.js je technologií, která tvoří tzv. run-time prostředí, ve kterém lze spouštět JavaScriptový kód mimo webový prohlížeč. To umožňuje využít JavaScript i k vytvoření plnohodnotných webových serverů, což bylo uplatněno i v rámci této práce. Node.js sice není potřeba k používání knihovny React, nicméně jeho součástí je systém pro správu balíčků Node package manager (npm), který je i jedním z možných prostředků ke správě závislostí (alternativou je například správce balíčků Yarn). Součástí npm je online databáze veřejných i privátních placených balíčků nazvaná npm registry. Z ní lze balíčky stahovat buď programově nebo přes webové stránky.

5.2 React

React je technologie, která slouží k tvorbě uživatelského rozhraní na webových stránkách. Poprvé byla představena v roce 2013 společností Facebook a od té doby se stala oblíbeným front-endovým nástrojem ve webových aplikacích, které pracují s dynamickým zpracováním dat. Popularitu dokládá graf na obrázku 5.1, ve kterém je znázorněn vývoj počtu stažení z registru npm mezi front-endovými technologiemi React, Angular.js (v tabulce jako angular), Ember, Vue a jádrem nové verze Angularu založené na jazyce Typescript (@angular/core).

React, ačkoliv často srovnáván s dalšími JavaScriptovými frameworky jako Angular nebo Vue, je pouze malou knihovnou představující zobrazovací vrstvu v pomyslné ar-



Obrázek 5.1: Počet stažení různých front-endových technologií z databáze Node package manager mezi roky 2013 až 2018, zdroj: server `npm-stat.com`

chitektuře Model-View-Controller. Pro další potřebnou funkcionalitu jako routing nebo správu stavu aplikace je nutné využít dalších knihoven. React dále dává větší důraz na využití prvků funkcionálního programování než na objektový přístup. Výhodou je lepší testovatelnost kódu i vyšší výkon. [19]

5.2.1 React a Document object model

React je knihovnou využívanou při vytváření tzv. Single page aplikací (SPA). SPA je webová aplikace, která funguje na principu přepisování obsahu právě načtené stránky, raději než načítání nové stránky ze serveru. Změny obsahu stránky jsou vykonány pomocí knihovny ReactDOM, která dokáže manipulovat s Document object modelem (DOM) a renderovat tak nadefinované uživatelské rozhraní. ReactDOM byla od samotné knihovny React oddělena ve verzi 0.14, a to z důvodu možnosti znovupoužití vytvořeného uživatelského rozhraní mimo webový prohlížeč – konkrétně ve frameworku React Native při tvorbě mobilních aplikací. [19]

Samotný Document object model představuje hierarchickou reprezentaci HTML dokumentu, kde každý vrchol popisuje určitý HTML element. Hierarchický model je vytvořen webovým prohlížečem po načtení HTML struktury. Dále poskytuje aplikační rozhraní pro manipulaci se svojí strukturou, které lze následně využít například při vytváření skriptů měnící obsah, styl apod. [20]

Manipulace s obsahem DOM je tedy základním principem při vytváření Single page aplikací, na kterou se zaměřuje výše zmíněná knihovna ReactDOM. Ta dále zavádí další koncept, virtuální DOM, který reprezentuje vytvořené uživatelské rozhraní (konkrétně elementy knihovny React) v paměti počítače a který je synchronizován s reálným DOM. React elementy mají formu JavaScriptových objektů (viz. následující podkapitola) a manipulace s nimi je méně náročná na výkon než manipulace s reálným Document object modelem. Knihovna ReactDOM zajišťuje synchronizaci virtuálního DOM s reálným, a je tak schopna efektivně propagovat změny pouze v té části hierarchické reprezentace, kde nastala změna. [19]

5.2.2 Vytváření uživatelských rozhraní

Existuje několik možných způsobů, jak vytvořit výše zmíněné React elementy. Prvním způsobem je využití funkce *React.createElement()*. Tato funkce vyžaduje tři parametry. První určuje typ požadovaného HTML elementu, druhý parametr obsahuje tzv. vlastnosti objektu, které představují další atributy elementu jako id, className apod. Poslední parametr se využívá k zanoření dalších požadovaných elementů neboli potomků.

Druhým způsobem je využití tzv. factories, což jsou speciální objekty, které dokáží obstarat vytvoření nových elementů. Pomocí factories lze vytvořit všechny běžně podporované HTML a SVG elementy a lze je také využít při vytváření nových komponent (komponenty dále popisuje následující podkapitola). Inicializační funkce tentokrát přijímá pouze dva parametry, první pro přidání vlastností a druhý pro definici potomků.

Poslední zmíněnou možností, která se v rámci této práce objevuje nejčastěji, je využití speciálního rozšíření JavaScriptu nazvaného JSX. JSX byl představen s vydáním Reactu za účelem snadnější definice objektů Document object modelu webové stránky. Syntax využívá HTML tagů, uvnitř kterých je možné snadno definovat další vnořenou strukturu. Lze zde také vykonat příkazy JavaScriptového kódu, pokud jsou ohraničeny ve složených závorkách. Výsledek provedených příkazů je automaticky escapován, takže je aplikace chráněna před potenciálním narušením pomocí Cross-site scriptingu. Části kódu v jazyce JSX je nutné před zobrazením transpilovat pomocí nástroje Babel, jelikož interpretace samotného JSX není ve webových prohlížečích podporována. [19]

Všechny tři výše zmíněné možnosti vytváření React elementů jsou ukázány v následujícím výpisu kódu 5.1 na příkladu odrážkového seznamu.

```
// Funkce createElement()
let example1 = React.createElement("ul", {"className": "myList"},
        React.createElement("li", null, "myListItem1"),
        React.createElement("li", null, "myListItem2")
);

// Objekt factory
let example2 = React.DOM.ul({"className": "myList"},
        React.DOM.li(null, "myListItem1"),
        React.DOM.li(null, "myListItem2")
)

// Notace JSX
let exampleName = "myListItem1"
let example3 = <ul><li>{exampleName}</li><li>myListItem2</li></ul>
```

Ukázka kódu 5.1: Možnosti vytváření React elementů

5.2.3 Komponentový přístup

Komponenty jsou základním stavebním kamenem uživatelského rozhraní vytvářeného za pomoci knihovny React. Jedná se o objekty chovající se podobně jako JavaScriptové funkce – mohou přijímat parametry ve formě objektu nazvaného *props* (z angl. properties) a vracejí React elementy popsané v minulé podkapitole. Pomocí komponent lze rozdělit uživatelské rozhraní na menší části, což přináší výhody v podobě větší přehlednosti a znovupoužitelnosti kódu.

Komponenty lze definovat buď jako pravé JavaScriptové funkce, nebo pomocí JavaScriptového standardu ES6, kde jsou komponenty definovány jako třídy. Definice pomocí funkce je vhodná pro jednoduché komponenty bez složité vnitřní logiky, jejich zápis je stručný a nároky na výkon nižší. Příklad funkcionální komponenty lze vidět na výpisu 5.2. Předání objektu *props* je provedeno při volání komponenty podle klíče *name*.

```
function Example(props) {
    return <h1>Hello, {props.name}</h1>;
}

// Nasledne volani komponenty
<Example name="Anna" />
```

Ukázka kódu 5.2: Funkcionální komponenta

Naopak, pokud je třeba řešit složitější logiku uvnitř komponenty, je možno definovat komponentu jako třídu. Příklad takové definice je v následující ukázce kódu 5.3.

```
export default class Example extends React.Component {
  constructor(props) {
    super(props)
    this.state = {...}
  }
  componentDidMount() {...}
  componentDidUpdate(prevProps, prevState, snapshot) {...}

  render() {
    return (<div><h1>Hello, {props.name}</h1></div>)
  }
}
```

Ukázka kódu 5.3: Komponenta jako ES6 třída

V takto definované komponentě je možné upravit chování v průběhu jejího životního cyklu (funkce *constructor()*, *componentDidMount()*, *componentDidUpdate()*), a také kontrolovat její stav (objekt *state*). Stav je místo, kde jsou uložena data dané komponenty, která se mění například v závislosti na interakci uživatele s rozhraním v prohlížeči. Pokud taková změna nastane, je volána funkce *render()*, pomocí které je vykreslena nová podoba komponenty. Při změně stavu je využito principu imutability – není doporučeno měnit stav přímo, jelikož by mohlo dojít k narušení fungování ostatních metod životního cyklu nebo k přepsání provedených změn (jediné místo, kde lze modifikovat stav komponenty přímo, je její konstruktor, viz. dále). Ke změně stavu je v Reactu zabudována funkce *setState()*, která spustí asynchronní operaci sloučení starého a nového stavu, a zahájí tak překreslení dané komponenty, včetně jejích potomků. [21]

Další speciální metody se týkají životního cyklu komponenty. První z nich, *constructor()*, je první volanou funkcí při procesu přidávání komponenty do Document object modelu (anglicky je tento proces označován jako *mounting*). V konstruktoru lze inicializovat proměnné uvnitř stavu na zvolené hodnoty a registrovat funkce definované v rámci komponenty do její instance. Pokud je v konstruktoru třeba přistupovat k objektu *props*, lze toho dosáhnout pomocí předání tohoto objektu v parametru konstruktoru a následným voláním funkce *super(props)*. [22]

Po provedení operací v konstruktoru je volána metoda *render()* a je upraven Document object model. Ihned po vložení nové komponenty do DOM je volána funkce *componentDidMount()*, která může znovu spustit metodu *render()*, pokud došlo ke změně ve stavu komponenty. Výhodou je, že druhé překreslení proběhne ještě před zobrazením komponenty ve webovém prohlížeči, jedná se tak o vhodné místo například k získání dat ze vzdáleného API, jelikož uživatel uvidí až výslednou podobu s načteným obsahem (pokud je tento obsah získán dostatečně rychle). [22]

Druhá část životního cyklu komponenty se týká jejího aktualizování. Aktualizace je spuštěna změnou v objektech *state* nebo *props*. Lze ji také vyvolat ručně pomocí funkce *forceUpdate()*. Při aktualizaci React podobně jako v předchozím případě spustí *render()* metodu a upraví Document object model. Následně je volána metoda *componentDidUp-*

date(). Tato metoda je přetížena několika parametry – dostupné jsou prevProps, prevState a snapshot. Tyto parametry obsahují informace z doby před aktualizací komponenty a lze je využít pro další operace. Pokud tato operace znovu aktualizuje komponentu, je nutné ji provádět pouze v podmínce, která zajistí, že dojde jen k jedné aktualizaci – v opačném případě by vznikl nekonečný aktualizační cyklus. Typické využití funkce *componentDidUpdate()* lze vidět v následujícím výpisu kódu, kde aktualizace komponenty nastane pouze v případě, že se změnilo uživatelské ID v příchozím parametru.

```
componentDidUpdate(prevProps) {
  // Typicke vyuziti (je nutne porovnat novy objekt props s jeho
     stavem pred aktualizaci):
  if (this.props.userID !== prevProps.userID) {
    this.fetchData(this.props.userID);
  }
}
```

Ukázka kódu 5.4: Typické využití metody *componentDidUpdate()*

V průběhu životního cyklu komponent lze využít ještě dalších funkcí, které dosud nebyly zmíněny, nicméně které jsou používány spíše v krajních případech. Jedná se o [22]:

- *getDerivedStateFromProps(newProps, prevState)* – tato funkce je společná jak pro vytváření, tak aktualizaci komponenty. Je využívána v případě, že stav komponenty je závislý na nově příchozích parametrech v *props*. Návratovou hodnotou je tedy JavaScriptový objekt, který modifikuje stav (nebo jeho část) komponenty, nebo hodnota *null*.

- *shouldComponentUpdate(nextProps, nextState)* – tato funkce dovoluje manuálně zastavit aktualizaci komponenty a zamezit tak novému překreslení. To může zajistit vyšší výkon aplikace. Návratová hodnota funkce je *true* pro pokračování v procesu aktualizování nebo *false* pro její zastavení.

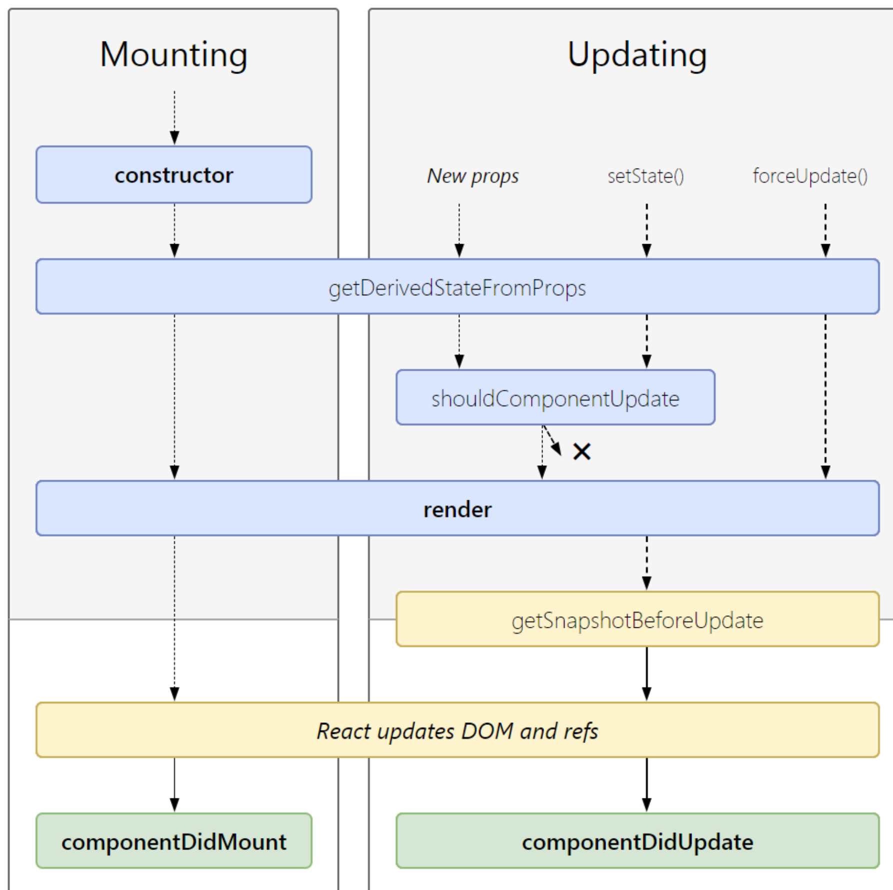
- *getSnaphotBeforeUpdate(prevProps, prevState)* – pomocí této metody lze získat informace z Document object modelu (např. pozici na stránce) těsně před tím, než jsou hodnoty v DOM přepsány. Návratová hodnota je předána funkci *componentDidUpdate()* v parametru snapshot.

Celý průběh životního cyklu včetně těchto funkcí lze vidět na obrázku 5.2.

5.3 Další podpůrné technologie

V implementaci projektu bylo využito několika dalších externích knihoven:

- **Express.js** – tento framework byl využit v serverové části, která připravuje testovací data a vystavuje je ve formě objektu JSON. Express poskytuje rozhraní, pomocí kterého lze snadno definovat routování, tedy definice URL cest koncových bodů, a také zpracování HTTP požadavků a vytváření odpovědí.



Obrázek 5.2: Životní cyklus React komponent při jejich vytváření a aktualizaci, zdroj: [23]

- **Material-UI** – tento framework slouží k vytváření uživatelského rozhraní. Material-UI poskytuje soubor prvků uživatelského rozhraní, které splňují specifikaci Google Material Design. Prvky mají podobu React komponent, které dále mají své vlastní rozhraní sloužící k jejich editaci a stylování.

- **Moment.js** – tato knihovna slouží k manipulaci s datumy a časovými údaji v prostředí JavaScriptu. Umožňuje automatické parsování řetězců v různých formátech (například ISO 8601 nebo RFC 2822), definovat lze i vlastní formát struktury datumu. Parsované řetězce jsou převedeny na speciální objekt typu *moment*. Tyto objekty lze mezi sebou pomocí vestavěných funkcí porovnávat, formátovat je nebo přičítat a odčítat časové hodnoty.

- **React Date Picker** – malá knihovna poskytující implementaci kalendáře jako React komponenty. Kalendář nabízí mnoho možností úprav včetně zvýraznění zvolených dnů, různé možnosti zobrazení při výběru datumů nebo vyznačení vybraného časového rozmezí mezi dvěma kalendáři.

Kapitola 6

Knihovny pro vizualizaci grafových struktur

V této kapitole budou představeny dostupné knihovny pro vizualizaci grafových struktur. Jelikož je pro vývoj aplikace nutná možnost nasazení knihovny v JavaScriptové aplikaci, byl i výběr zástupců omezen pouze na knihovny implementované v tomto programovacím jazyce. Cílem této kapitoly je vytvoření stručného srovnání vybraných zástupců vzhledem k požadavkům vytvářené aplikace a poskytnutí základních informací o knihovnách pro potencionálního zájemce.

6.1 Požadavky na vybranou knihovnu

Z hlediska vytvářené aplikace byla stanovena základní kritéria, podle kterých jsou vybraní zástupci srovnáváni. Jedná se o:

- Jednoduchost použití – objem již předpřipravené funkcionality, možnosti přenastavení a jednoduchost ovládání aplikačního rozhraní.
- Podpora a oblíbenost projektu – hodnotí podporu ze strany vývojářů a uživatelskou aktivitu v jednotlivých repozitářích.
- Kvalita dokumentace a vzorových příkladů – knihovny mohou kromě dokumentace nabízet i vzorové implementace, které potenciálnímu zájemci ulehčí vytváření základní funkcionality.
- Metody renderování – metoda renderování grafických elementů může být prováděna různými způsoby (pomocí SVG elementů, využitím HTML elementu Canvas nebo za pomoci grafické akcelerace ve Web Graphics Library – WebGL).
- Podpora clusteringu – hodnotí, zda knihovna obsahuje funkcionalitu clusteringu zmíněnou v části 3.3.4.

Tato kritéria jsou následně vyhodnocena v části 6.3, kde je také vybrán vhodný zástupce pro vývoj vizualizačního nástroje.

6.2 Přehled vybraných zástupců

Do výběru pro bližší zkoumání byly vybrány knihovny Sigma.js, D3.js, Vis.js, ECharts a Cytoscape.js. Tento výběr se snažil zahrnout co možná nejpoužívanější zástupce (hodnoceno subjektivně podle aktivity a oblíbenosti jednotlivých GitHub repozitářů).

6.2.1 Sigma.js

Sigma.js je knihovna vytvořená pod záštitou francouzské společnosti Sciences Po médialab. Zaměřuje se pouze na vykreslování grafových struktur. Knihovnu v době posledního sběru dat (březen 2019) již nelze nazvat aktivně podporovanou – poslední stabilní verze byla vydána v říjnu 2017. Podle informací v GitHub repozitáři sice v současné době probíhá vývoj nové verze, která bude lépe přizpůsobena moderní JavaScriptové specifikaci ECMAScript 6, nicméně tato verze je zatím bez jakékoliv dokumentace.

Sigma.js v základní verzi nenabízí kompletní sadu nástrojů pro ovládání grafu – chybí například možnost manipulace s vrcholy nebo možnost nastavení force-directed layoutu. Tento problém lze překonat použitím některého z mnoha pluginů, které jsou ke knihovně dostupné – kromě výše zmíněných funkcionalit lze mezi pluginy najít například podporu pro grafovou databázi Neo4j, animace grafu nebo zpracování souboru GEXF. Bez využití pluginů knihovna i tak nabízí poměrně rozsáhlé API, týkající se zachycení událostí, úprav elementů v grafu, rendererů nebo celkového nastavení. Nepodporuje však funkcionalitu clusteringu.

Dokumentace základní části knihovny (viz. [24]) je dostatečná jak v úplnosti, tak přehlednosti. Jelikož je ovšem velká část funkcionality implementována ve výše zmíněných pluginech, je zde kvalita dokumentace závislá i na autorech jednotlivých pluginů. Nevýhodou je také nedostatek interaktivních příkladů – autoři odkazují pouze na vzorové soubory v jejich repozitáři, což snižuje názornost pro potenciálního zájemce.

Knihovna nabízí dva renderery pro vykreslení grafu – HTML Canvas a renderer založený na WebGL. První způsob využívá samostatné Canvas elementy pro vrcholy, hrany, popisky a události. Vrcholy zde prostupují několika funkcemi, které určí jejich výsledné vlastnosti a souřadnice. Výhodou je možnost úpravy těchto funkcí, ovšem za cenu nižšího výkonu. Druhý způsob naopak provádí maticové výpočty transformací. Ty lze akcelerovat na grafické kartě, což má za následek lepší výkon při vykreslování, je ovšem složité do tohoto procesu vstoupit a provádět změny. V rámci knihovny existuje i možnost použít svůj vlastní renderer.

6.2.2 D3.js

Knihovna D3.js s podnázvem Data-Driven Documents je zřejmě nejznámější JavaScriptovou knihovnou používanou pro vizualizaci dat. S více než 77 000 hvězdami na serveru GitHub tak nabízí obrovské množství tutoriálů, příkladů a aktivní komunitu. D3.js lze použít pro vizualizaci libovolného diagramu, grafu nebo tabulky – vše záleží pouze na vlastní implementaci uživatelem. [25]

Knihovna pro vizualizaci využívá HTML a CSS, samotné prvky jsou vykresleny jako SVG elementy. Data jsou nejprve svázána s Document object modelem stránky, poté jsou na ně aplikovány animace, přechody a další transformace.

Ačkoliv D3.js přináší možnost implementovat jakýkoliv vizualizační návrh, je nutné poznamenat vysokou složitost zejména pro nové uživatele. Vizualizace začíná opravdu „od nuly“ a veškerou funkcionalitu je nutné implementovat. V případě vykreslování grafů se jedná například o zoom, manipulaci s vrcholy na plátně nebo přidání orientace hranám – takové operace jsou většinou v ostatních knihovnách již připraveny k použití.

V dokumentaci je aplikační rozhraní perfektně popsáno, nicméně pro nové uživatele může být obtížné zorientovat se v obrovském počtu poskytovaných metod. Výhodou D3.js je galerie, ve které jsou desítky implementovaných interaktivních příkladů se zdrojovými kódy [26].

Implementační jazyk knihovny je podobný syntaxi jQuery. Využívá možností selektorů, podle kterých pak pomocí funkcí mění vlastnosti elementů. Dále odděluje metody pro vkládání (funkce *enter()*), modifikaci a odstranění (funkce *remove()*) dat. Přidání vrcholů do grafu může tedy vypadat tak, jak ukazuje výpis 6.1.

```
...
var node = svg.append("g")
    // zakladni vyber dat do grafu
    .attr("class", "nodes")
    .selectAll("circle")
    .data(graph.nodes)
    .enter().append("circle")
    .attr("r", 15)
    // stylovani
    .attr("fill", function (d) { return color(d.group); })
    .attr("stroke", "#fff")
    .attr("stroke-width", 1.5)
    // funkce pro manipulaci vrcholu - nutno implementovat
    .call(d3.drag()
        .on("start", dragstarted)
        .on("drag", dragged)
        .on("end", dragended));
...
```

Ukázka kódu 6.1: Přidání vrcholů do grafu v D3.js

6.2.3 Vis.js

Vis.js je knihovna vytvořená holandskou firmou Almende B.V. Její oficiální vývoj byl ukončen v únoru roku 2019, další podpora tak bude závislá na aktivitě komunity. Kromě podpory vykreslení grafu vrcholů a hran (modul Network) nabízí také implementaci pro vizualizaci časové řady (modul Timeline), sloupcových a spojnicových grafů (modul Graph2d) nebo grafů ve trojrozměrném prostoru (modul Graph3d). Pro vykreslování využívá HTML

element Canvas.

Z pohledu vizualizace grafů je tato knihovna velmi propracovaná a nabízí všechny potřebné funkce pro interakci s elementy bez nutnosti použití jakýchkoli pluginů. Jedná se například o manipulaci s vrcholy, možnost přidání vrcholů do výběru nebo registrace umístění kurzoru myši nad určitým elementem. Tyto funkce lze ovládat pomocí proměnné nastavení, která má formu JavaScriptového objektu. Samotné API knihovny lze tedy hodnotit jako poměrně bohaté.

Dokumentace je na dobré úrovni, knihovna také poskytuje webovou stránku s mnoha interaktivními příklady ukazující její možnosti [27].

Vis.js aplikuje na rozložení vrcholů hierarchický nebo force-directed layout s propracovaných fyzikálním systémem, jehož chování (odpor mezi vrcholy, síla a délka pružin, gravitace atd.) lze měnit pomocí konstant v nastavení. Využít lze implementaci BarnesHut, Force Atlas 2 nebo Kamada Kawai algoritmu.

Další funkcí, kterou knihovna podporuje, je možnost clusteringu na základě určité vlastnosti vrcholu (např. barva). Všechny vrcholy s uvedenou vlastností jsou při využití této funkce sloučeny do jednoho rozkliknutelného vrcholu. Příklad takové operace lze vidět v ukázce 6.2.

```
var nodes = [
  {id: 4, label: 'Node 4'},
  {id: 5, label: 'Node 5'},
  {id: 6, label: 'Node 6', cid:1},
  {id: 7, label: 'Node 7', cid:1}
]
var options = {
  joinCondition:function(nodeOptions) {
    return nodeOptions.cid === 1;
  }
}
network.clustering.cluster(options);
```

Ukázka kódu 6.2: Ukázka funkcionality clusteringu v knihovně Vis.js

Struktura grafu je ve Vis.js uložena ve speciální datové struktuře nazývané DataSet. Ten umožňuje dynamicky zobrazovat změny v grafu po přidání, odebraní nebo úpravě vrcholů a hran. Data je také možné pomocí nabízených funkcí filtrovat nebo v nich vyhledávat podle unikátního identifikátoru. Samotné elementy jsou v DataSetu uloženy jako JavaScriptové objekty, pomocí kterých lze měnit vlastnosti zobrazovaných vrcholů a hran (barva, velikost, popisek atd.). To lze docílit definováním klíčů a jejich hodnot (nebo vnořených objektů), které jsou specifikovány v API knihovny. Příklad všech definovatelných vlastností lze vidět v [28]. Ke speciálním vlastnostem objektu lze přidat i vlastní vlastnosti s dodatečnými informacemi.

Vis.js podporuje různé možnosti nahrání vstupních dat – kromě klasického vstupu z JavaScriptového objektu lze využít také vestavěného parseru pro JSON výstup z aplikace Gephi, podporován je i vstup ve formátu jazyka DOT.

6.2.4 ECharts

ECharts je knihovna aktivně vyvíjená čínskými vývojáři. Knihovna má i početnou čínskou uživatelskou komunitu, což lze vidět v GitHub repozitáři, kde je velká část diskuze vedena v čínském jazyce, což ovšem může být překážkou při hledání řešení problémů.

Nabízené možnosti vizualizace jsou dynamické a bohaté na animace. Podporováno je bezmála třicet druhů 2D i 3D vizualizací, zajímavostí může být například podpora vyobrazení dat na mapovém podkladu (například tako teplotní mapa). Knihovna nabízí možnost renderovat vizualizace buď s použitím HTML Canvasu nebo jako SVG elementy.

Modul pro vizualizaci grafů je velmi podobný modulu z Vis.js. Nabízí mnohá nastavení, včetně možnosti manipulace s vrcholy, změnu barev a tvaru vrcholů nebo možnosti animací. Unikátní funkcionalitou je zde automatická tvorba legendy ze struktury nastavených kategorií. Kategorii lze popsat jako předpis pro skupinu vrcholů – jedna kategorie má například vlastní barvu a tvar. Výsledná legenda je interaktivní, po kliknutí na prvek legendy lze všechny vrcholy a příslušné hrany z dané kategorie v grafu skrýt, opětovným kliknutím poté znovu zobrazit. Prvky v grafu lze uspořádat buď pomocí force-directed layoutu nebo vrcholy uspořádat do kruhu. Implementace force-directed layoutu zde ovšem není na dobré úrovni – při interakci s grafem totiž ostatní vrcholy nemění svoji pozici plynule. Připravené layouty je ovšem možné nepoužít a souřadnice vrcholů předat přímo v datech.

Dokumentace knihovny (viz. [29]) je do určité míry nepřehledná, jelikož popisuje všechny moduly knihovny dohromady. Její výhodou jsou naopak příklady využití metod objevující se přímo v jejich popisu. Galerii s názornými příklady implementace knihovny lze najít v [30].

6.2.5 Cytoscape.js

Cytoscape je nástroj vyvíjen za účelem vizualizace molekulárních struktur – svoje užití tak nachází zejména v institucích zaměřujících se na biologický výzkum. Nástroj má jak svoji desktopovou verzi vyvíjenou v jazyce Java, tak i JavaScriptovou verzi Cytoscape.js, kterou je možné využít při vývoji webové aplikace. Cytoscape.js je nástupce nástroje Cytoscape Web, který byl založen na teď již zastaralé technologii Adobe Flash. Knihovnu lze spustit buď jako vizualizační nástroj v okně prohlížeče, nebo ji používat pro výpočty na serveru bez grafického rozhraní (tzv. headless mode). [31]

Architektura knihovny je složena ze dvou hlavních částí – samotné jádro aplikace (core API) a kolekce zpracovaných dat. Metody z jádra aplikace slouží k manipulaci s grafem jako celkem a některé z nich vrací zmíněnou kolekci. Kolekce má svoje vlastní rozhraní, které nabízí funkce pro zjištění informací o elementech, filtrování nebo průchod grafem. Pro Cytoscape.js jsou podobně jako u knihovny Sigma.js vyvíjeny pluginy, které nabízejí nové funkce nebo layouty. [31]

Cytoscape.js je knihovna orientovaná na teorii grafů, čemuž odpovídá i její nabídka funkcí. Kromě klasické podpory uživatelské manipulace s grafem je implementováno mnoho využitelných layoutů, například mřížka, kruh, náhodné rozložení, Cose layout a jiné. Další layouty jsou dostupné ve formě rozšíření.

Jedinečnou částí knihovny je nabídka algoritmů. Dostupné jsou metody pro průchod

grafem – prohledávání do šířky a hloubky, nalezení nejkratší cesty, kostry grafu i míry centrality nebo mezipolohy. V ukázce 6.3 je ukázáno nalezení nejkratší cesty a vzdálenosti z vrcholu e do vrcholu j, přičemž váha hrany je dána proměnnou *weight*.

```
var dijkstra = cy.elements().dijkstra('#e', function(edge){
  return edge.data('weight');
});

var pathToJ = dijkstra.pathTo( cy.$('#j') );
var distToJ = dijkstra.distanceTo( cy.$('#j') );
```

Ukázka kódu 6.3: Příkad použití funkce Dijkstrova algoritmu nabízené knihovnou Cytoscape.js

Všechny poskytované funkce lze najít v přehledné dokumentaci, která obsahuje u každé funkce i příklad jejího použití [32].

6.3 Zhodnocení vybraných zástupců

Všechny knihovny byly krátce otestovány a následně ohodnoceny dle stanovených kritérií. Výsledky lze vidět v tabulce 6.1. Hodnocení bylo provedeno známkami 1 až 5 (jako ve škole).

	Sigma.js	**D3.js**	**Vis.js**	**ECharts**	**Cytoscape.js**
Vybavenost a jednoduchost použití	2	4	1	2	2
Podpora a oblíbenost	4	1	3	3	1
Dokumentace a příklady	3	1	1	2	1
Metody renderování	Canvas WebGL	SVG	Canvas	Canvas SVG	Canvas
Podpora clusteringu	Ne	Ne	Ano	Ne	Ne

Tabulka 6.1: Srovnání vybraných knihoven pro grafovou vizualizaci

Knihovna Sigma.js (včetně svých pluginů) určitě poskytuje dostatečné aplikační rozhraní pro práci s grafem a jako jediná poskytuje akceleraci vykreslování pomocí rendereru založeného na WebGL. Z hlediska využití v této práci ovšem nemohla být využita kvůli chybějící funkcionalitě clusteringu a nedostatečné podpoře kvůli ukončení vývoje.

D3.js se ukázala být knihovnou s vysokou učící křivkou a její zobrazování prvků jako SVG elementy se ukázalo být výkonově nedostatečné již při zobrazení grafu s několika desítkami vrcholů a orientovanými hranami. Tuto knihovnu tak lze doporučit uživatelům, kteří potřebují vytvořit vizualizaci přesně na míru nejlépe s malým počtem vizualizovaných elementů.

Knihovna ECharts je zástupcem s dobře vybaveným aplikačním rozhraním, dojem nicméně zhoršuje nedostatečně kvalitní implementace force-directed layoutu pro grafovou vizualizaci. Knihovna dále nepodporuje funkcionalitu clusteringu.

Uživateli, který by ve své aplikaci potřeboval využít funkce z teorie grafů, lze jed-

noznačně doporučit knihovnu Cytoscape.js. Tato knihovna nabízí jak rozsáhlé aplikační rozhraní, tak i mnoho dostupných layoutů pro grafovou vizualizaci. Její nevýhodou může být určitá nepřehlednost zapříčiněná syntaxí selektorů, pro využití v této práci také chyběla podpora clusteringu.

Pro použití v této práci tak byla vybrána knihovna Vis.js, která jako jediná nabízí funkcionalitu clusteringu. Možnost vytváření clusterů se ukázala být stěžejním prvkem v návrhu zpracování velkého množství vizualizovaných entit jak z hlediska přehlednosti, tak i plynulosti vizualizace ve webovém prohlížeči. Knihovna dále využívá JavaScriptové objekty pro definici vizualizovaných vrcholů, hran i samotné konfigurace knihovny, což je ideální formát pro komunikaci se serverem. V serverové části je tak možné přímo generovat konfiguraci objektů bez nutnosti transformace kódu do odlišné struktury. Dalším kladným bodem je kvalitní implementace fyzikálního modelu, který poskytuje plynulé animace a činí tak interakci s grafem uživatelsky přívětivou.

Kapitola 7

Vývoj aplikace

Tato kapitola se věnuje popisu realizace projektu, včetně jeho inicializace a nutných prerekvizit, jako je příprava testovacích dat. Popsána je struktura projektu, návrh způsobu vizualizace i návrh uživatelského rozhraní a struktury dat při komunikaci se serverem. Kapitola se následně věnuje postupu při realizaci navržených částí, tedy implementaci. Implementace probíhala za použití technologií popsaných v kapitole 5 – jazyka JavaScript spolu s knihovnou React – a vybrané vizualizační knihovny Vis.js představené v kapitole 6. V kapitole budou zmíněny i některé kroky vývoje testovací serverové části, která, ačkoliv nebyla požadována jako výsledný produkt této práce, byla nedílnou součástí vývoje při návrhu a testování navrženého způsobu komunikace.

7.1 Inicializace projektu

K inicializaci základní struktury projektu bylo využito správce balíčků Node package manager (npm), představeného v kapitole 5. Npm je součástí instalace běhového prostředí Node.js, které je nutné pro inicializaci a spuštění testovacího serveru. Pro inicializaci serverové části bylo využito příkazu *npm init*, pomocí kterého lze v příkazové řádce interaktivně nastavit název projektu, verzi, popis nebo typ licence. Tento příkaz vytvoří soubor *package.json*, který je nedílnou součástí projektu. Tento soubor kromě obecných údajů o projektu drží informace o použitých balíčcích včetně jejich verzí, což je využito Node package managerem při správě závislostí. Uvnitř *package.json* lze také definovat uživatelské skripty, které lze následně spustit příkazem *npm run* <*název skriptu*>.

Ukázka souboru *package.json* využitého v serverové části projektu je na následujícím výpisu. V ukázce si lze všimnout definovaného skriptu *dev*, který slouží ke startu serveru i front-endové části zároveň (s využitím knihovny *Concurrently*), čehož bylo využíváno po celou dobu tvorby aplikace při vývoji funkcionality obou zmíněných částí.

```
{
  "name": "graphapp",
  "version": "1.0.0",
   "author": "Pavel Borik",
   ...
  "scripts": {
    "client-install": "cd client && npm install",
    "start": "node server.js",
    "server": "nodemon server.js",
    "client": "npm start --prefix client",
    "dev": "concurrently \"npm run server\" \"npm run client\""
  },
  "devDependencies": { ... },
  "dependencies": { ... }
}
```

Ukázka kódu 7.1: Ukázka souboru package.json

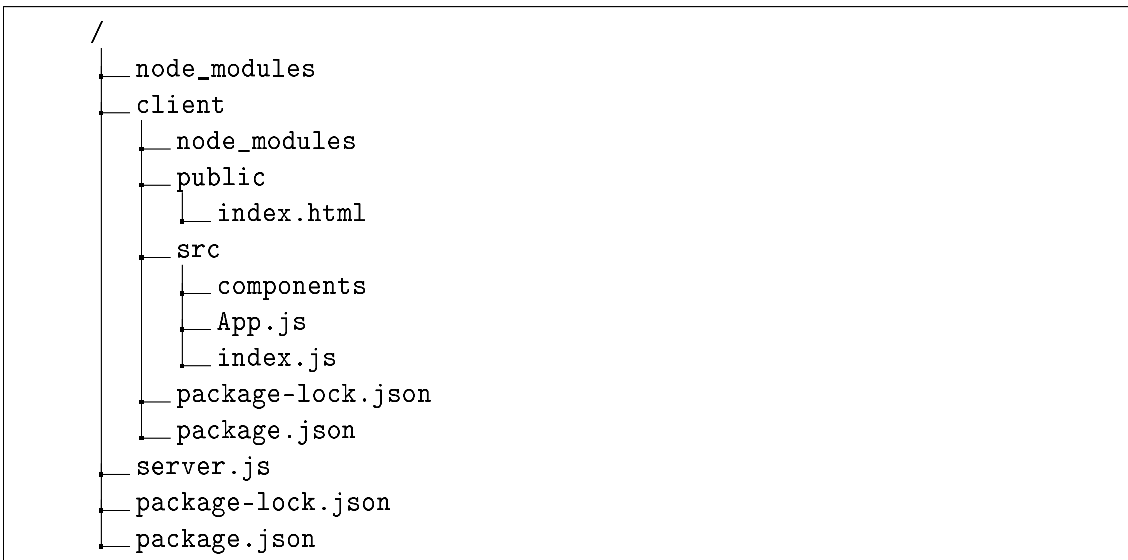
Pro vytvoření základní struktury front-endové části bylo využito nástroje *Create React App*, který slouží k automatické inicializaci React aplikace. Vytvořen je jak výše zmíněný soubor *package.json*, tak i hlavní HTML stránka, ve které jsou následně propagovány změny Document object modelu (podle principu Single page aplikace). Dále je vytvořena ukázková komponenta v souboru *App.js* a také soubor *index.js*, který tuto komponentu zavede (vyrenderuje) do hlavní HTML stránky.

Důležitou součástí Create React App je dále automatické nastavení podpůrných nástrojů Babel a Webpack, což umožní využívat moderní prvky JavaScriptu a zajistí automatické transpilování a vytvoření balíčku se zdrojovým kódem pro spuštění aplikace ve webovém prohlížeči.

7.1.1 Struktura projektu

Struktura projektu má dvě části - v kořenu projektu se nachází serverová část tvořená souborem *server.js*, ve složce *client* se nachází front-endová část tvořená React aplikací. Pomocí příkazu *npm install* spuštěném ve složce se souborem *package.json* je vytvořena složka *node_modules*, kam jsou staženy potřebné soubory knihoven, a také soubor *package-lock.json*. Tento soubor popisuje strom závislostí a slouží například nástroji npm k optimalizaci procesu instalování nových knihoven.

Ve složce *client* pak lze vidět soubory zmíněné výše – hlavní HTML stránka a další její pomocné soubory (například favicona) se nachází ve složce *public*, ve složce *src* jsou pak umístěny JavaScriptové soubory včetně všech React komponent. Základní struktura projektu je vyobrazena na obrázku 7.1



Obrázek 7.1: Základní adresářová struktura projektu po instalaci závislostí

7.1.2 Příprava testovacích dat a jejich struktura

Pro potřeby vývoje byla společností Unicorn Systems poskytnuta neprodukční data sloužící pro testování implementované funkcionality. Data byla předána v dokumentu Excel ve formátu *xls*. Jelikož bylo pro vývoj nutné převést tato data do testovacího databázového systému, byl využit nástroj MySQL for Excel, který rozšiřuje funkcionalitu programu Excel o grafické rozhraní pro komunikaci s databázovým systémem MySQL. Nástroj umožňuje import, úpravu a zejména export dat z tabulkového procesoru včetně vytvoření potřebných tabulek v databázi.

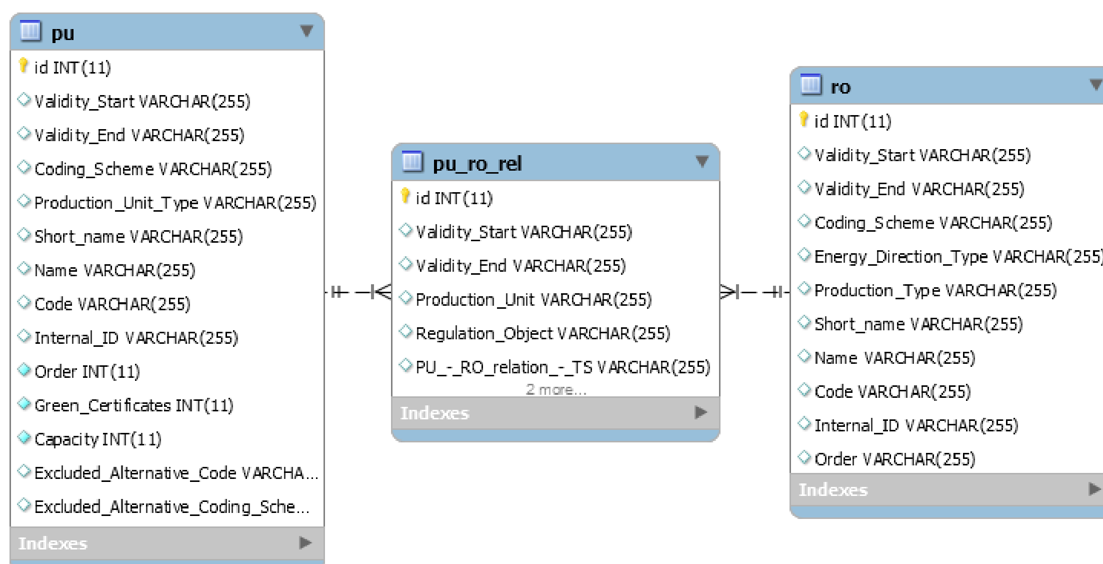
Testovací data poskytují pohled na reálnou strukturu uchovávaných informací o entitách. Na obrázku 7.2 lze vidět příklad vytvořených tabulek v databázi – jedná se o tabulky uchovávající data o entitách PU (production unit), RO (regulation object) a vztahu mezi nimi (pu_ro_rel). Ze struktury lze vidět například začátky a konce časové validity jak jednotlivých entit, tak vztahů mezi nimi. Tabulky uchovávající data o dalších entitách NBS byly pro přehlednost vypuštěny.

7.2 Návrhová část

V této kapitole jsou popsány konkrétní navržená řešení problematiky vizualizace, návrh vzhledu uživatelského rozhraní a také návrh struktury dat, které jsou předávány při komunikaci se serverem.

7.2.1 Obecný postup tvorby vizualizace v rámci aplikace

Před začátkem návrhu a implementace byl stanoven obecný popis kroků, které jsou nutné ke spuštění nástroje a získání potřebných dat nutných pro tvorbu grafové vizualizace. Ke spuštění nástroje by mělo dojít v informačním systému BASSE, který byl zmíněn v kapitole 2. Kroky nutné k provedení vizualizace jsou následující:



Obrázek 7.2: Ukázková struktura testovacích dat exportovaných do podoby databázových tabulek, zdroj: vlastní zpracování v nástroji MySQL Workbench

1. Spuštěny budou dva servery, jeden pro front-endovou React aplikaci, druhý pro poskytování potřebných dat.
2. Uživatel v systému BASSE vybere požadované časové rozmezí v detailu určité entity a spustí webovou stránku s vizualizačním nástrojem.
3. Je spuštěn vizualizační nástroj, který vyšle dotaz na server obsahující jednoznačný identifikátor entity, vybraný časový interval a identifikátor pohledu, podle kterého jsou vygenerována data.
4. Server zpracuje dotaz a vrátí data ve formátu JSON v požadované struktuře.
5. Data jsou uložena ve stavu React komponenty (ve webovém prohlížeči).
6. Komponenta předá data knihovně Vis.js, která zajistí vytvoření vizualizace grafové struktury.

7.2.2 Návrh řešení vizualizace a doplnění dalších požadavků

Jak bylo nastíněno v kapitole 2.1, bylo nutné navrhnout takový princip vizualizace grafové struktury, který umožní zachytit proměnlivé časové vazby mezi entitami sdružení NBS. Pro tuto problematiku byly navrženy dva uživatelsky přepínatelné pohledy na graf. Tyto pohledy mají společnou část, a to výběr časového rozsahu, do kterého se promítají změny ve vztazích mezi entitami – pokud tedy validita vztahu mezi dvěma entitami skončí před začátkem nebo začne až po konci vybraného časového intervalu, do grafu tyto entity nevstupují, jelikož jsou vyfiltrovány na straně serveru.

První pohled načítá do grafu všechny entity s validitou vztahu projevující se ve vybraném časovém rozsahu. Pokud se vazba mezi entitami mění (začíná nebo končí) ve vybraném

rozsahu, je hrana v grafu představující tuto vazbu označena červenou barvou. Tento pohled je v uživatelském rozhraní i zdrojovém kódu označován jako **Time frame**.

Druhý pohled na graf zobrazuje vztahy validní pouze v určitém momentu, který je uvnitř výše zmíněného vybraného časového rozsahu. Uživatel může buď vybrat moment ručně, nebo využije tlačítek sloužících k přeskakování mezi takovými časovými momenty, kdy nastala změna vztahu mezi entitami – princip lze přirovnat k pohybu po časové ose. Tento pohled je dále označován jako **Moment**.

Další návrh řešení se týká problematiky zmíněné v kapitole 4.5, a to zobrazování entit s velkým množstvím vazeb na jiné entity. Využito bylo speciální funkcionality **vytváření clusterů**, kterou poskytuje knihovna Vis.js. To uživateli umožní skrýt pro něj nezajímavou část grafové struktury a zlepší jak přehlednost, tak i výkon celé aplikace ve webovém prohlížeči. Skrytá část grafu zde má podobu vrcholu, který je po rozkliknutí nahrazen vnitřní strukturou – buď dalšími subclustery, nebo již konkrétními entitami. Dělení clusterů lze definovat v serverové odpovědi – je tak možné určit počet úrovní tak, aby poslední cluster po rozkliknutí odhalil takový počet konkrétních entit, který zachová rozumnou přehlednost grafu. Jedním z nápadů na skladbu clusterů bylo dělení dle abecedy – první úroveň *A-Z*, druhá úroveň *A-L* a *M-Z*, v dalších úrovních analogicky.

7.2.2.1 Další požadavky

V průběhu konzultování práce byly specifikovány další požadavky týkající se celkové funkcionality. Pro funkcionalitu clusteringu bylo nutné zajistit, aby uživatel mohl vrátit zpět akce rozkliknutí vrcholů, což zajistí pohodlné prozkoumávání grafu. Dále byly specifikovány požadavky na interakci s grafem – zobrazování informační karty o entitě NBS po kliknutí na příslušný vrchol. V informační kartě se dále zobrazují tlačítka, které lze směrovat na libovolný odkaz, například tedy zpět do informačního systému. Definovat tak lze například spuštění formuláře pro úpravu dané entity. Pro interakci s grafem bylo dále zajištěno zobrazení dalších volitelných informací o entitě ve formě vyskakovacího okna, pokud uživatel umístí kurzor nad určitý vrchol grafu.

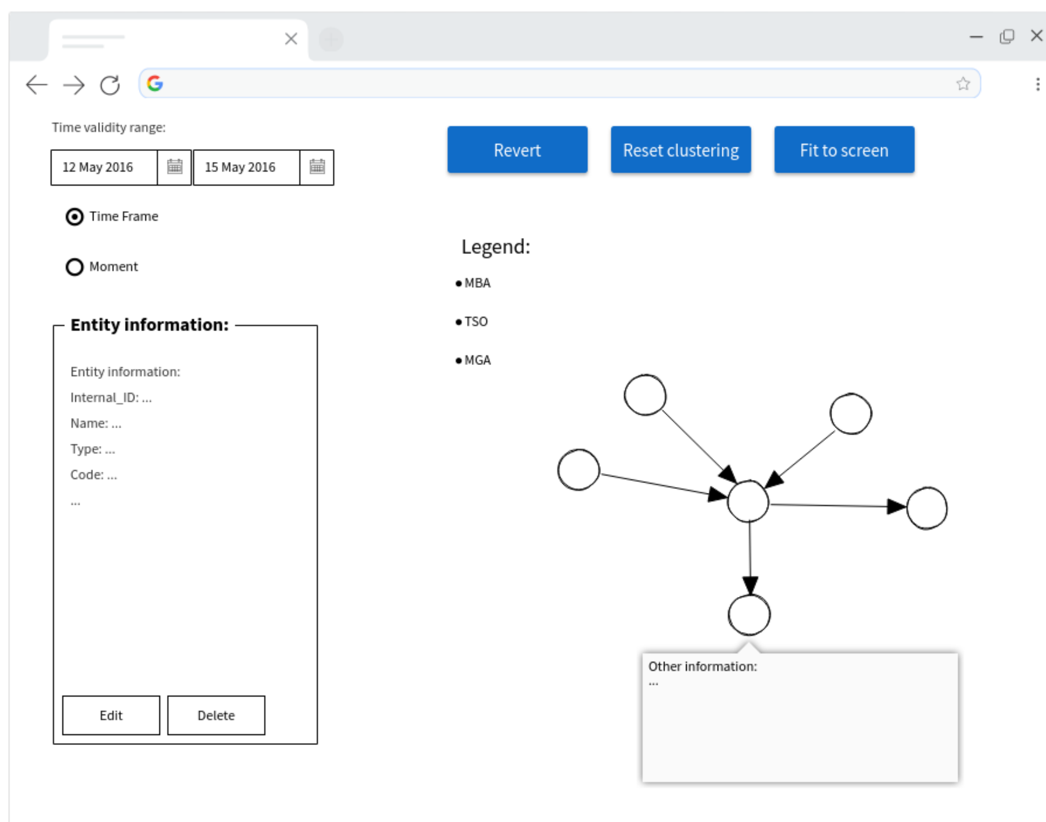
7.2.3 Uživatelské rozhraní

Navržené uživatelské rozhraní rozděluje obrazovku na tři hlavní části – samotnou vizualizaci grafové struktury, nastavení parametrů pro kontrolu časové validity a informační kartu zobrazující informace o vybrané entitě.

Plátno s vizualizovaným grafem zabírá zhruba dvě třetiny obrazovky, a je na něm také umístěna legenda popisující barvu právě zobrazovaných entit. Nad grafem jsou umístěna tři tlačítka, která lze využít k navrácení akce rozkliknutí vrcholů představujících clustery (tlačítko Revert), obnovení clusterů do základní úrovně (tlačítko Reset clustering) a obnovení přiblížení a umístění grafu do základní pozice, kde graf zabírá co největší plochu plátna (tlačítko Fit to screen).

Část uživatelského rozhraní poskytující nastavení pohledu na graf se nachází v levé horní části obrazovky. Obsahuje dva elementy kalendáře, pomocí kterých lze vybrat začátek a konec časové validity a také přepínač pohledů (*Time frame* a *Moment*). Po přepnutí typu

pohledu na *Moment* je zobrazen třetí element kalendáře se šipkami, které umožní rychlý pohyb po časové ose změn v grafové struktuře. Celý návrh uživatelského rozhraní lze vidět na obrázku 7.3.



Obrázek 7.3: Návrh uživatelského rozhraní, zdroj: vlastní zpracování na serveru `mockflow.com`

7.2.4 Návrh způsobu získávání dat

Komunikace se serverem funguje na principu REST API, kde po volání metody GET zašle server data potřebná pro vizualizaci pomocí HTTP. Definovány byly dva koncové body – první pro získání popisu celé grafové struktury, druhý pro získání detailu o vybrané entitě. Data poskytovaná serverem mají formát JSON.

7.2.4.1 Koncový bod getData

První jmenovaný koncový bod má unikátní identifikátor getData a jako parametry požadavku přijímá unikátní identifikátor entity, počáteční a konečné datum časové validity a také identifikátor pohledu, podle kterého je na serveru vygenerována struktura grafu (tímto identifikátorem se myslí interní značka, podle které skript na serveru vybírá vztahy k zahrnutí do grafové struktury, a ne pohledy na graf, které byly zmíněny v předchozích kapitolách).

Odpověď ze serveru obsahuje tři hlavní části – část týkající se konfigurace grafu, informace o entitě, pro kterou je vytvářena vizualizace, a samotný popis vrcholů a hran grafu. Úplná struktura odpovědi tohoto koncového bodu je uvedena v příloze B. Je vhodné

poznamenat, že struktura odpovědi byla navrhována tak, aby z části odpovídala konfiguračnímu objektu knihovny Vis.js. Ve front-endové části aplikace tak není potřeba složitě transformovat obdržená data.

Konfigurační část obsahuje objekt *groups*, ve kterém jsou definovány typy entit vstupující do vizualizace. Uvedeny jsou jejich jména a barvy v grafu, tyto informace jsou využity také při vytváření legendy.

Další částí konfigurace je objekt *clustering*. Tento objekt popisuje celou strukturu dělení clusterů, pomocí které lze následně tuto funkcionalitu implementovat ve vizualizaci. Rozdělení má strukturu stromu, kde každý objekt popisující subcluster obsahuje klíč *parent*, pomocí kterého se odkazuje na svého rodiče ve vyšší úrovni. Princip lze vidět ve výpisu 7.2, kde je definován rodičovský cluster *g1* se dvěma potomky.

```
"config": {
    ...
    "clustering": {
        "g1": {
            "name": "Cluster g1 contains: {count}"
        },
        "g1_1": {
            "name": "Subcluster g1_1 with {count} items",
            "parent": "g1"
        },
        "g1_2": {
            "name": "Subcluster g1_2 with {count} items",
            "parent": "g1"
        }
        ...
    },
},
```

Ukázka kódu 7.2: Část konfiguračního objektu ze serverové odpovědi popisující rozdělení vrcholů do clusterů pomocí stromové struktury

Poslední částí konfigurace je objekt *range*, který obsahuje počáteční a koncovou hranici vybraného intervalu časové validity. Těmito hodnotami jsou naplněny kalendáře po prvotním načtení webové stránky.

Mimo konfigurační objekt serverová odpověď obsahuje dále objekt *queriedEntity*, jehož struktura je téměř shodná s odpovědí druhého koncového bodu *getDetail* (viz. následující podkapitola 7.2.4.2). Tento objekt je zde zakomponován kvůli zrychlení načítání stránky – není třeba vysílat dva HTTP požadavky.

Poslední částí odpovědi koncového bodu *getData* je objekt popisující konektivitu vizualizovaného grafu. Tento objekt se nazývá *graph* a je složen ze dvou polí – pole *nodes* popisující vrcholy a pole *edges* popisující hrany. Odpověď obsahuje základní informace o vrcholech, které jsou následně využívány komponentami v aplikaci. Některé vlastnosti objektu jsou využívány přímo knihovnou Vis.js – jedná se například o nutný unikátní identifikátor (vlastnost *id*), popisek vrcholu v grafu (vlastnost *label*) nebo obsah vyskakovacího

okna, jež může být definován i přímo pomocí značkovacího jazyka HTML (vlastnost *title*).

Za zmínku dále stojí vlastnost *group*, jejíž hodnota odkazuje na jednu ze skupin v objektu konfigurace popsaného výše. To zajistí korektní stylování vrcholu v grafu. Poslední důležitou vlastností je pole *clustering*, které obsahuje klíče clusterů z definovaného objektu *clustering*. Pole každého objektu obsahuje klíče odkazující na všechny clustery, kam daný vrchol v rámci stromové struktury definice clusterů patří – toto lze vidět ve výpisu 7.3.

```
...
{
    "id": "ac3f6ea5-9bbf-11e8-a86c-1c6f65c3aae2",
    "label": "SC R0106",
    ...
    "clustering": [
        "g1_1",
        "g1"
    ]
},
...
```

Ukázka kódu 7.3: Část objektu popisující vrchol grafu, ukazující vlastnosti *id*, *label* a *clustering*

Druhou částí objektu popisující konektivitu grafu je pole *edges*, ve kterém jsou definovány hrany mezi vrcholy. Základními vlastnosti objektu popisující hranu jsou zde dvě reference na unikátní identifikátory vrcholů, které určují odkud a kam hrana povede (vlastnosti *from* a *to*). Objekt dále obsahuje údaje o začátku a konci validity tohoto konkrétního vztahu, a také údaj, zda se tato validita mění v rámci intervalu validity vybraného při spuštění aplikace nebo v uživatelském rozhraní (vlastnost *validityChanges*).

7.2.4.2 Koncový bod getDetail

Tento koncový bod je volán po kliknutí na vrchol v grafu pro získání dat do informační karty. Obsah tohoto koncového bodu je triviální, obsahuje pouze vlastnost *actions*, což je pole definující tlačítka, která jsou zobrazena na informační kartě, a vlastnost *detail*, která obsahuje řetězec složený z HTML značek, který je přímo vložen do informační karty. Ukázku struktury lze vidět v příloze B.

7.3 Implementační část

V této kapitole jsou popsány části zdrojového kódu aplikace. Stručně je zmíněna implementace testovacího serveru, zbytek kapitoly se věnuje zejména React komponentám frontendové části.

7.3.1 Testovací server

Zprovoznění serveru poskytujícího testovací data sestávalo ze dvou hlavních částí – základního nastavení serveru včetně připojení k databázovému systému a nastavení routování. Jak bylo zmíněno v kapitole 5.3, k vytvoření serveru bylo využito frameworku Express.js.

První část lze vidět ve výpisu 7.4. Nejprve je nutné importovat potřebné moduly pomocí klíčového slova *require*. Následně je možné vytvořit serverovou aplikaci voláním metody *express()*. Aplikace je spuštěna příkazem *listen()*, ve kterém je také nastaven aplikační port. Pro připojení k databázi je využito metod *createConnection()*, kde jsou nastaveny přihlašovací údaje z environmentálních proměnných, a *connect()*, která vytvoření databázové spojení.

```
const express = require('express');
const mysql = require('mysql');

const app = express();
const db = mysql.createConnection({
  host: `${process.env.DB_HOST}`, port: `${process.env.DB_PORT}`,
     user: `${process.env.DB_USER}`,  password: `${process.env.
     DB_PASS}`,  database: `${process.env.DB_SCHEMA}`
});

db.connect((err) => { if (err) { throw err; }
    console.log('Mysql connected')
});
...
const port = 5000;
app.listen(port, () => `Server running on port ${port}`);
```

Ukázka kódu 7.4: Základní operace nutné pro spuštění testovacího serveru

Nastavení routování je prováděno funkcemi frameworku Express, jejichž název odpovídá HTTP požadavkům – v případě ukázky 7.5, která ukazuje koncový bod *getData*, se jedná o metodu *get()*. V parametru metody lze specifikovat cestu ke koncovému bodu, druhým parametrem je callback funkce přijímající parametry představující objekty HTTP požadavku (*req*) a HTTP odpovědi (*res*).

```
app.get('/api/getdata', (req, res) => {
  const query = req.query; // Parametry URL
  ...
  db.query(queryString, queryParams, (err, rows) => {
    if (err) throw err;
    ...
    // Ziskane radky z databaze lze upravovat
    rows = rows.filter(node => moment(node.validityStart).isBefore(
       moment(validityEnd)));
    ...
    // Odeslani HTTP odpovedi
    res.json({...});
  });
});
```

Ukázka kódu 7.5: Funkce koncového bodu *getData*

Objekt *req* obsahuje různé vlastnosti týkající se HTTP požadavku – části URL adresy, parametry, IP adresu požadavku, obsah HTTP hlavičky apod. Právě získání parametrů z URL bylo v tomto projektu důležité, jelikož tyto parametry obsahují informace o vyhledávané entitě, časové validitě a dalších potřebných údajích. K těmto údajům lze přistoupit pomocí klíče *query* objektu *req*.

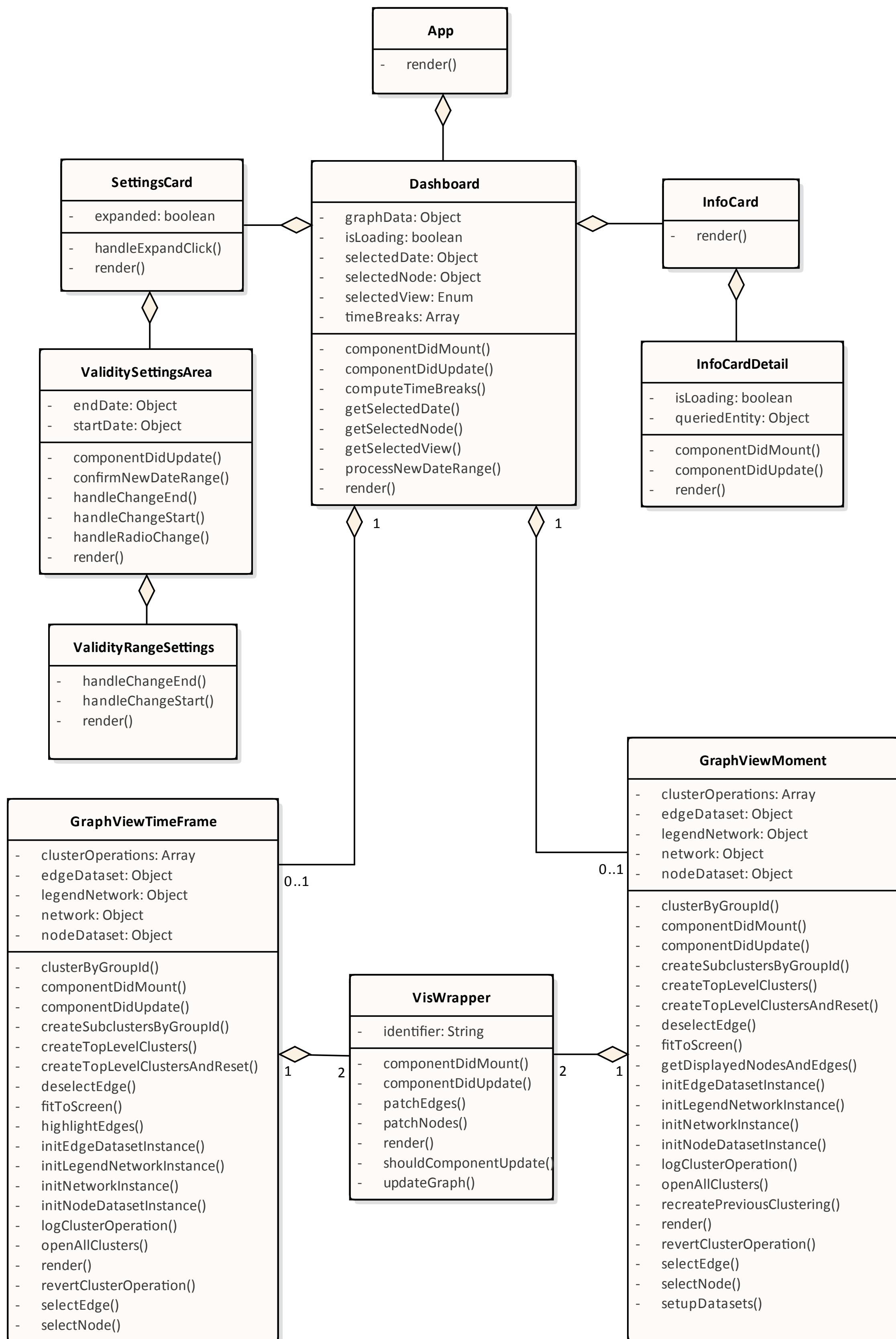
Objekt *res* představuje HTTP odpověď odeslanou aplikací *Express*, v ukázce lze vidět odeslání dat ve formě objektu JSON. Volání je provedeno uvnitř callback funkce databázového dotazu, který byl nutný k získání dat z testovací databáze.

7.3.2 Struktura komponent ve front-endové části

Při vývoji aplikace bylo využito komponentového přístupu frameworku React k rozdělení zodpovědností za různé části uživatelského rozhraní. Na obrázku 7.4 lze vidět diagram zachycující strukturu vztahů mezi komponentami vizualizačního nástroje. V diagramu jsou kromě funkcí implementovaných danými komponentami také atributy představující stavové nebo instanční proměnné daných komponent.

Počátkem struktury je komponenta *App*, která je vygenerována při inicializaci aplikace. Jejím potomkem je pak komponenta *Dashboard* – tato komponenta ve svém stavu udržuje data získaná ze serveru a také určuje rozmístění svých potomků na webové stránce.

Zbývající část je rozdělena na tři části – *SettingsCard* se svými potomky zodpovídá za část rozhraní, které zajišťuje možnost změny časové validity vztahů a přepínání pohledu. Právě pohled na graf zajišťuje jedná z komponent *GraphViewTimeFrame* a *GraphViewMoment* – podle toho, která z nich je aktivní. Zde je také nutné zmínit komponentu *VisWrapper*, která je spouštěna ve dvou instancích – jedna pro zobrazení legendy, druhá k zobrazení samotného grafu. Informační kartu o vybrané entitě mají pak na starost komponenty *InfoCard* a *InfoCardDetail*.



Obrázek 7.4: Struktura React komponent v aplikaci, zdroj: vlastní zpracování

7.3.3 Získání dat ze serveru

Data vytvořená v serverové části jsou získána v komponentě Dashboard pomocí metody *fetch*. Tato metoda umožňuje vyslat HTTP požadavek na server a vrací objekt typu *Promise*, který představuje budoucí dokončení asynchronní operace. V ukázce 7.6 lze vidět průběh získávání dat ze serveru, kde metoda *fetch* přijímá parametry URL adresy. V projektu je URL adresa speciálně složena z proměnné prostředí určující doménu a port serveru, umístění na serveru (*pathname*) a parametrů URL (*search*). Objekt *location* je speciálním objektem nástroje *React router*, který dokáže modifikovat URL adresu v prohlížeči a získat její zmíněné části. Po vykonání asynchronního volání jsou data ve formě JSON uložena ve stavu komponenty pomocí funkce *setState()*. Nakonec jsou z těchto dat určeny časové okamžiky, kdy došlo ke změnám v časové validitě vztahů mezi získanými entitami, pomocí funkce *computeTimeBreaks()*. Získané časové okamžiky jsou využity v pohledu na graf **Moment** pro rychlé přeskakování mezi změnami ve struktuře vztahů.

```
componentDidMount() {
     const { location } = this.props;
     this.setState({ isLoading: true });
     fetch(`${process.env.REACT_APP_API}${location.pathname}${
         location.search}`)
       .then(res => {if (res.ok) return res.json()})
       .then(data => this.setState({
         graphData: data,
         ...
       }, () => this.computeTimeBreaks()));
}
```

Ukázka kódu 7.6: Získání dat ze serveru v komponentě *Dashboard*

7.3.4 Implementace uživatelského rozhraní

Jak již bylo zmíněno, obrazovka aplikace se skládá ze třech hlavních částí – vizualizace grafu, části nastavení a informační karty s detaily o právě vybrané entitě. Tyto tři části jsou rozmístěny v komponentě *Dashboard*, která ve svém stavu udržuje velký podíl informací, s nimiž tyto části pracují.

Uživatelské rozhraní aplikace bylo vytvořeno za pomoci komponent z JavaScriptového frameworku Material-UI, který nabízí již předpřipravené prvky uživatelského rozhraní splňující specifikaci Google Material Design. Komponenty frameworku umožňují změnu jejich vzhledu pomocí poskytovaného rozhraní, Material-UI dále umožňuje stylovat komponenty přímo pomocí JavaScriptu, což je využíváno v tomto projektu spolu se stylováním pomocí souborů *CSS*. Výslednou podobu uživatelského rozhraní lze vidět v příloze A na obrázku A.2.

Část nastavení je implementována v komponentách *SettingsCard*, *ValiditySettingsArea* a *ValidityRangeSettings*. Část nastavení, která umožňuje přeskakování po časových momentech, je implementována v komponentě *EnhancedDatePicker*, která zajišťuje funkcionalitu

kalendáře spolu s ovládacími tlačítky. Pomocí vnějších ovládacích tlačítek (šipky vlevo a vpravo) lze přeskakovat po časových momentech, kdy nastala změna ve struktuře grafu, vnitřní tlačítka (šipky nahoru a dolu) umožňují pohyb po jednotlivých dnech.

V ukázce 7.7 lze vidět právě zmíněný posun o jeden den v metodě *increaseDate()*. V ní jsou použity metody knihovny Moment.js – nejprve je kvůli zamezení modifikaci vytvořena kopie instance objektu *Moment*, jehož časová hodnota je následně zvýšena o jeden den pomocí metody *add()*. Nové datum je předáno do funkce, která změní stav v komponentě *Dashboard*, v jejímž stavu je tento údaj uložen. V ukázce lze dále vidět vytváření tlačítek jako klikatelné ikony a také podmínku pro vypnutí tlačítka, pokud vybraný časový moment dosáhl horní hranice vybrané validity vztahů v grafu.

```
...
increaseDate = () => {
  const newDate = this.props.selectedDate.clone().add(1, 'd');
  this.props.getSelectedDate(newDate);
};
...
render() {
...
  <IconButton disabled={!!selectedDate.clone().add(1, 'd').isAfter(
      maxValidity, 'd')} onClick={this.increaseDate}>
    <ArrowUp />
  </IconButton>
...
}
```

Ukázka kódu 7.7: Změna vybraného časového data v komponentě *EnhancedDatePicker*, podle kterého jsou zobrazeny vztahy mezi entitami v pohledu *Moment*

Informační karta o vybrané entitě je implementovaná v komponentě *SettingsCard* a jejím potomku *SettingsCardDetail*. Právě druhá jmenovaná komponenta se stará o zobrazení detailních informací o entitě v informační kartě. Potřebná data může získat buď z hlavní serverové odpovědi koncového bodu *getData* (nastává vždy při prvotním načtení stránky nebo při změně časové validity), pokud ovšem uživatel interaguje s grafem a vybírá vrcholy, komponenta kontaktuje server pomocí HTTP požadavku na koncový bod *getDetail*, který potřebný obsah poskytne. To lze vidět ve výpisu 7.8, který ukazuje implementaci funkce *componentDidUpdate()*.

```
componentDidUpdate(prevProps) {
  const { selectedNode, isInfoIncluded } = this.props;
  const { internalId, type } = selectedNode;
  if (prevProps.selectedNode.internalId !== internalId) {
    if (!isInfoIncluded) {
      this.setState({ isLoading: true });
      fetch(`${process.env.REACT_APP_API}/getdetail?id=${internalId
         }&type=${type}`)
        .then(...)
        .then(res => this.setState({ queriedEntity: res.
           queriedEntity, isLoading: false }));
    } else {
      this.setState({ queriedEntity: selectedNode });
    }
  }
}
```

Ukázka kódu 7.8: Získání detailních informací ze serveru při výběru vrcholu v grafu v komponentě *InfoCardDetail*

Následující výpis 7.9 ukazuje způsob vložení části serverové odpovědi, ve které je definován obsah o detailu entity přímo jako řetězec ve formátu HTML. Vložení dovnitř informační karty je prováděno ve funkci *render()* do HTML elementu *div* pomocí atributu *dangerouslySetInnerHTML*.

```
render() {
  ...
  actionsComponent = queriedEntity.actions.map(action => (
      <Button ... href={action.url}>
        {action.name}
      </Button>
      ));
  ...
  return (
    ...
    <Fragment>
      <div className="detail" dangerouslySetInnerHTML={{ __html:
         queriedEntity.detail }} />
      <div className="actions-container">{actionsComponent}</div>
    </Fragment>
    ...
  );
}
```

Ukázka kódu 7.9: Využití serverové odpovědi k vytvoření informační karty v komponentě *InfoCardDetail*

7.3.5 Zobrazení grafu a legendy

Základem pro zobrazení grafové struktury v aplikaci je komponenta *VisWrapper*. Jedná se o upravený soubor z volně dostupného nástroje *react-graph-vis*, který slouží k integraci objektů samotné knihovny Vis.js do React komponenty. Ukázku integrace lze vidět ve výpisu 7.10, kde jsou po načtení komponenty vytvořeny potřebné objekty DataSet a Network. V ukázce lze dále vidět registraci událostí (např. kliknutí na vrchol v grafu) definovaných v rodičovských komponentách.

```
componentDidMount() {
  this.edges = new vis.DataSet();
  this.nodes = new vis.DataSet();
  ...
  this.Network = new vis.Network(
    container,
    Object.assign({}, this.props.graph, {
      edges: this.edges,
      nodes: this.nodes
    }), options
  );
  ...
  const events = this.props.events || {};
  for (const eventName of Object.keys(events)) {
    this.Network.on(eventName, events[eventName]);
  }
}
```

Ukázka kódu 7.10: Základní integrace knihovny Vis.js do prostředí React v komponentě *VisWrapper*

Pro manipulaci s grafem komponenta *VisWrapper* poskytuje dva způsoby. Prvním způsobem je zpracování nových dat předaných z rodičovské komponenty, tedy v objektu *props*. *VisWrapper* dále implementuje metodu *shouldComponentUpdate()*, ve které porovnává současné a nové objekty popisující graf. V metodě je tak určeno, které vrcholy (nebo hrany) byly odstraněny, přidány nebo upraveny, a v DataSetu jsou provedeny příslušné operace (přidání, upravení, smazání), což zajistí plynulé překreslení grafu na obrazovce. Tento přístup se ovšem ukázal být nevhodný pro vysoký počet entit vizualizovaných v rámci této práce. Porovnávání všech objektů je totiž výpočetně náročné – je nutné projít rekurzivně všechny zkoumané JavaScriptové objekty a porovnávat obsah jejich vlastností. To způsobovalo zamrznutí uživatelského rozhraní až na stovky milisekund.

Proto bylo nutné v implementaci využít druhý možný způsob, a to předání instancí objektů Network i Dataset do rodičovské komponenty. Tam je následně využito funkce *setData()* definované v rozhraní objektu Network pro přímé nastavení zobrazovaných vrcholů a hran.

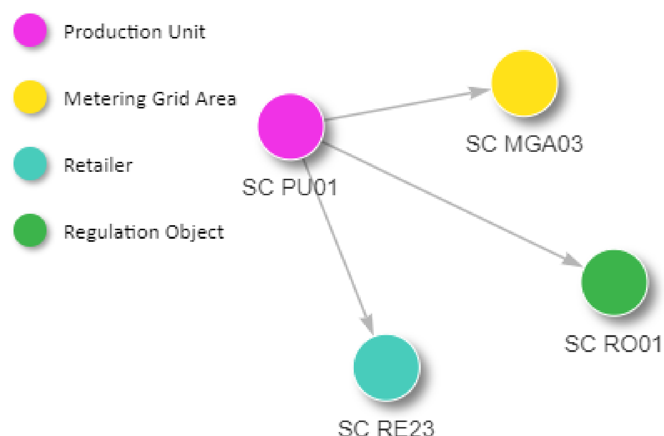
V aplikaci je komponenta *VisWrapper* využita dále k vykreslení legendy. Instance objektu Network totiž dokáže poskytnout přístup ke kontextu HTML elementu Canvas, na kterém je vykreslován graf. Zde je tak možné využít JavaScriptových metod k vykreslení

legendy grafu, což lze vidět ve výpisu 7.11. Metoda *initLegendNetworkInstance* se stará o předání reference objektu Network do instanční proměnné *legendNetwork*, jak bylo zmíněno v předchozím odstavci. Tento objekt pak poskytuje funkci *on()*, která zaregistruje posluchač události spouštějící se před prvotním vykreslením. Zde je pak možné provádět změny v elementu Canvas.

```
initLegendNetworkInstance = networkInstance => {
  networkInstance.on('beforeDrawing', ctx => {
    ...
    Object.values(this.props.data.config.groups).forEach(group => {
      if (Object.prototype.hasOwnProperty.call(group, 'parent'))
         return;
      const coords = this.legendNetwork.DOMtoCanvas({ x: baseX, y:
         baseY });
      ctx.beginPath();
      ctx.arc(coords.x, coords.y, 10, 0, 2 * Math.PI, false);
      ...
      ctx.fillStyle = group.color.background;
      ctx.fill();
      ...
    });
  });
  this.legendNetwork = networkInstance;
};
```

Ukázka kódu 7.11: Vytvoření legendy ze serverových dat a její vykreslení na element Canvas v komponentách *GraphViewTimeFrame* a *GraphViewMoment*

Výsledek vizualizace lze vidět na obrázku 7.5. Vrcholy jsou obarveny podle jejich zařazení skupin definovaných v serverové odpovědi. V legendě je pak barevná vysvětlivka pro jednotlivé skupiny entit sdružení NBS.



Obrázek 7.5: Ukázka vykresleného grafu s legendou

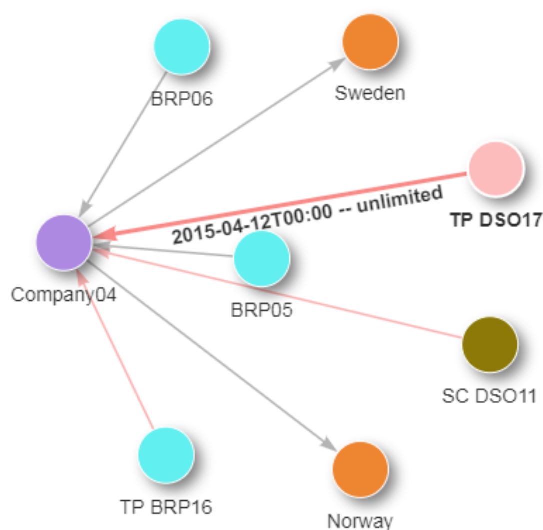
7.3.6 Pohledy na graf

Pohledy na graf jsou implementovány v komponentách *GraphViewTimeFrame* a *GraphViewMoment* podle návrhu funkcionality popsané v části 7.2.2. Aktivní je vždy pouze jedna z těchto dvou komponent, v závislosti na hodnotě přepínače pohledu v kartě nastavení. Výběr aktivní komponenty je prováděn v komponentě *Dashboard*, která také poskytuje potřebná data získaná ze serveru.

V komponentě *GraphViewTimeFrame* je důležitá zejména funkce *highlightEdges()*, která zajišťuje obarvení hran představující vztah, který neplatí po celý interval vybrané časové validity (podle vlastnosti *validityChanges*). Hrany jsou obarveny modifikací objektu, ve kterých jsou pomocí funkce *Object.assign()* definovány vnořené objekty popisující obarvení hran. Upravené pole hran je nakonec zaregistrováno v objektu Network knihovny Vis.js pomocí funkce *setData()*. Tento průběh lze vidět ve výpisu 7.12. Výslednou podobu grafu s vyznačenými hranami lze pak vidět na obrázku 7.6.

```
highlightEdges = () => {
  const { nodes, edges } = this.props.data.graph;
  const highlightedEdges = edges.map(edge => {
    if (edge.validityChanges === true)
      return Object.assign(edge, {
        color: { color: '#ff6363', opacity: 0.5, highlight: '#
           fb1414' }
     });
    return edge;
  });
  this.network.setData({ nodes: nodes, edges: highlightedEdges });
};
```

Ukázka kódu 7.12: Obarvení hran představující vztah, který neplatí po celý interval vybrané časové validity, v komponentě *GraphViewTimeFrame*



Obrázek 7.6: Graf s vyznačenými hranami

Zobrazení grafu v konkrétním časovém okamžiku v komponentě *GraphViewMoment* je složitější – je totiž nutné provést filtrování vrcholů a hran na základě jejich validity přímo v klientské části aplikace. Tuto funkcionalitu zajišťuje funkce *getDisplayedNodesAndEdges()*, kterou lze vidět ve výpisu 7.13. Pro zjištění, zda zobrazit daný vztah, je využito metody *isBetween()*, která zjišťuje, zda interval validity daného vztahu obsahuje vybraný časový moment. Pokud je tato podmínka splněna, hrana je zařazena do vizualizovaného výběru. Následně jsou podle vyfiltrovaných hran určeny také zobrazované vrcholy. Výsledné objekty jsou následně (mimo ukázku) využity k nastavení vizualizovaných dat pomocí funkce *setData()*, podobně jako v předchozí ukázce.

```
getDisplayedNodesAndEdges() {
    const { nodes, edges } = this.props.data.graph;
    const displayedEdges = edges.filter(edge => {
      return this.props.selectedDate.isBetween(
        moment(edge.validityStart),
        edge.validityEnd !== 'unlimited' ? moment(edge.validityEnd)
            : moment('2100-01-01'), 'h', '[)');
    });
    const displayedNodes = nodes.filter(node => {
      const edgeCount = displayedEdges.filter(edge => {
        return edge.from === node.id || edge.to === node.id;
      });
      return edgeCount.length > 0;
    });
    return { displayedNodes, displayedEdges };
  }
```

Ukázka kódu 7.13: Filtrování hran a vrcholů na základě jejich validity v konkrétním časovém okamžiku v komponentě *GraphViewMoment*

7.3.7 Clustering

K vytvoření clusterů z vrcholů je využito funkce *cluster()* objektu Network. Tato funkce přijímá jako parametr objekt, ve kterém je vytvoření clusteru definováno – lze vidět ve výpisu 7.14. Zařazování do clusteru je určeno ve vlastnosti *joinCondition*, která iteruje přes všechny vrcholy (jejichž vlastnosti jsou přístupné parametrem nodeOptions) a vrací vrcholy splňující danou podmínku. V případě ukázky se zjišťuje, zda obsah pole vrcholu, kde je (v serverové odpovědi) definována příslušnost do clusterů, obsahuje klíč shodný s příchozím parametrem funkce *clusterByGroupId()*. Modifikovat lze i nastavení vrcholu představující cluster, a to buď dynamicky ve vlastnosti *processProperties*, nebo staticky ve vlastnostech *clusterNodeProperties* a *clusterEdgeProperties*. V případě první možnosti lze pomocí parametrů přistoupit k objektům, které obsahují informace o potomcích uvnitř clusteru. Takto lze například do popisu clusteru vložit informaci o počtu obsažených vrcholů, viz. zmíněný výpis.

```
clusterByGroupId = (styleGroupId, clusterGroupId) => {
  const groupInfo = this.props.data.config.clustering[
     clusterGroupId];
  const clusterOptionsByData = {
    joinCondition: nodeOptions => {
      return nodeOptions.clustering.includes(clusterGroupId);
    },
    processProperties: (clusterOptions, childNodes, childEdges) =>
    {
      const countPlaceholder = '{count}';
      let label = groupInfo.name;
      if (label.includes(countPlaceholder)) {
        label = label.replace(countPlaceholder, childNodes.length);
      } else {
        label = `Contains: ${childNodes.length.toString()}`;
      }
      clusterOptions.label = label;
      return clusterOptions;
    },
    clusterNodeProperties: {...},
    clusterEdgeProperties: {...}
  };
  this.network.cluster(clusterOptionsByData);
};
```

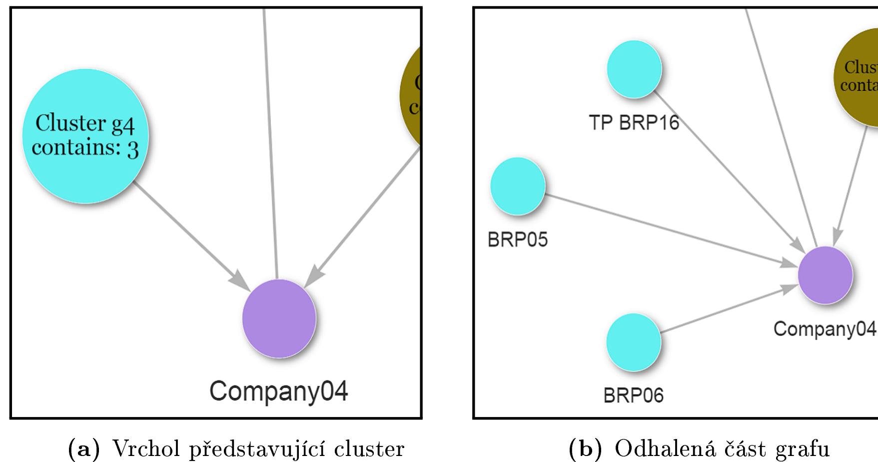
Ukázka kódu 7.14: Struktura funkce vytvářející clustery v komponentách *GraphViewTimeFrame* a *GraphViewMoment*

Příklad využití funkce z předchozí ukázky lze vidět ve výpisu 7.15, kde jsou vrcholy zařazovány do clusterů nejvyšší úrovně. Ve funkci jsou nejprve otevřeny všechny clustery, následně je procházen objekt ze serverové odpovědi, a nakonec je volána funkce vytvářející clustery s parametrem označení clusteru nejvyšší úrovně (takové clustery neobsahují vlastnost *parent*).

```
createTopLevelClusters = () => {
  this.openAllClusters();
  Object.entries(this.props.data.config.clustering).forEach(
     clusterDescription => {
    if (!Object.prototype.hasOwnProperty.call(clusterDescription
       [1], 'parent'))
      this.clusterByGroupId(clusterDescription[0],
         clusterDescription[0]);
  });
};
```

Ukázka kódu 7.15: Vytvoření clusterů nejvyšší úrovně v komponentách *GraphViewTimeFrame* a *GraphViewMoment*

Výsledek lze vidět na obrázku 7.7 – vrchol představující cluster zde ukrývá část grafu, která je po poklikání na tento vrchol odhalena.



(a) Vrchol představující cluster **(b)** Odhalená část grafu

Obrázek 7.7: Průběh odkrytí části grafu po poklikání na cluster

7.3.7.1 Princip vytváření subclusterů

Způsob vytvoření subclusterů při interakci s grafem je zachycen ve výpisu 7.16. Funkce *selectNode* implementuje událost volanou po kliknutí na vrchol v grafu, objekt *event* je poskytnut knihovnou Vis.js a obsahuje pole jedinečných identifikátorů kliknutím aktivovaných hran a vrcholů, což je využito k přiřazení identifikátoru vybraného vrcholu do proměnné *clickedNode*. Pomocí funkcí knihovny Vis.js – *isCluster()* a *openCluster()* – je zjištěno, zda je vybraný vrchol clusterem, a pokud ano, je tento cluster otevřen.

```
selectNode = event => {
  const { nodes } = event;
  const clickedNode = nodes[0];
  if (this.network.isCluster(clickedNode) === true) {
    const clusterNodeInfo = this.network.clustering.body.nodes[
       clickedNode];
    ...
    this.network.openCluster(clickedNode);
    this.createSubclustersByGroupId(
      clusterNodeInfo.options.group,
      clusterNodeInfo.options.clusterGroupId
    );
  }
  ...
};
```

Ukázka kódu 7.16: Implementace funkce reagující na událost kliknutí na vrchol v grafu v komponentách *GraphViewTimeFrame* a *GraphViewMoment*

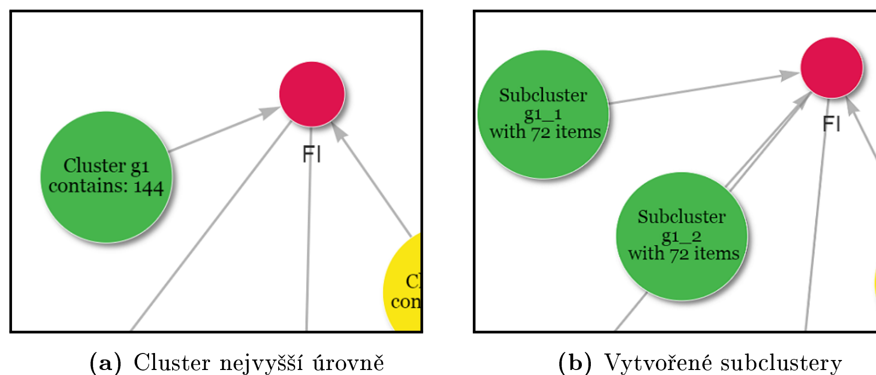
Po otevření clusteru je ihned volána funkce *createSubclustersByGroupId()*, kterou lze vidět ve výpisu 7.17. V ní je procházena definice v serverové odpovědi, a jsou vytvářeny

nové clustery, které jsou podle definice potomky právě otevřeného clusteru. Pokud takoví potomci neexistují, cluster zůstane otevřen, a dojde k odhalení konkrétních entit představující skrytou část grafu.

```
createSubclustersByGroupId = (styleGroupId, clusterGroupId) => {
  Object.entries(this.props.data.config.clustering).forEach(
      clusterDescription => {
    if (Object.prototype.hasOwnProperty.call(clusterDescription[1],
        'parent')) {
      if (clusterDescription[1].parent === clusterGroupId)
        this.clusterByGroupId(styleGroupId, clusterDescription[0]);
    }
  });
};
```

Ukázka kódu 7.17: Funkce určující vytváření subclusterů podle definice v serverové odpovědi v komponentách *GraphViewTimeFrame* a *GraphViewMoment*

Při využití tohoto přístupu tak není nutné vytvářet složitou strukturu zabalení všech vrcholů do struktury vnořených clusterů ještě před vizualizací grafu (přestože i takový přístup lze uplatnit), ale je možné je vytvářet dynamicky při poklikání na vrcholy. Prodleva mezi otevřením rodičovského clusteru a vytvořením jeho subclusterů je totiž pro uživatele nepostřehnutelná. Výsledné chování lze vidět na obrázku 7.8.



(a) Cluster nejvyšší úrovně **(b)** Vytvořené subclustery

Obrázek 7.8: Průběh vytvoření subclusterů po poklikání na cluster

Kapitola 8

Shrnutí výsledků

Výsledkem této práce je aplikace, která je připravena k propojení se serverem poskytující produkční data a k následnému nasazení do informačního systému. Byla vyřešena problematika zachycení časových validit vazeb mezi entitami sdružení Nordic Balance Settlement i logika přehledného zobrazení stovek vrcholů pomocí techniky clusterování.

Právě zpracování velkého počtu entit ukázalo limity jak webového prohlížeče (lze pozorovat zpomalení renderování grafu při zobrazení více než zhruba sta jednotlivých vrcholů), tak i samotných knihoven pro práci s grafy. V knihovně Vis.js byl objeven výkonnostní problém při vytváření clusterů, pokud jsou v DataSetu knihovny uloženy řádově stovky vrcholů. Tento problém byl sice opraven ve vývojové větvi projektu v repozitáři GitHub, nicméně v poslední verzi stažitelné z databáze npm tato oprava dosud vystavena nebyla – v aplikaci tak bylo využito upraveného zdrojového kódu. I přesto lze knihovnu Vis.js doporučit pro její jednoduchost nastavení a mnoho implementovaných funkcí pro práci s grafy, zejména při práci s řádově desítkami vrcholů.

Pro vytvoření uživatelského rozhraní bylo využito knihovny React, která se ukázala být spolehlivým nástrojem. I přes její rychlý vývoj je její dokumentace srozumitelná, a díky její rozšířenosti není problém vyhledat řešení k mnoha problémům. V závěru tvorby aplikace ovšem vyšlo najevo, že by bylo vhodné použít také stavový manažer (například knihovna Redux). To by zabránilo nutnosti využívat komponentu *Dashboard* jako most pro komunikaci ostatních komponent a nebylo by tak nutné předávat informaci o změně stavu přes celý komponentový podstrom.

Kapitola 9

Závěr

V této práci byl ve spolupráci s firmou Unicorn Systems navržen a implementován vizualizační nástroj ve formě aplikace, která zobrazuje vztahy mezi entitami energetického trhu ve formě orientovaného grafu. Aplikace byla vytvořena za použití programovacího jazyka JavaScript a knihoven React, Vis.js a Material UI. Při tvorbě byly řešeny dvě hlavní oblasti – způsob a obsah komunikace se serverem poskytujícím data a tvorba front-endové části.

Při řešení první oblasti byla navržena struktura požadavků i odpovědi pro komunikaci se serverem v architektuře REST API. Navržená struktura byla otestována na vytvořeném pokusném serveru. Toto rozhraní bylo vytvořeno na základě testovacích dat pro doménu sdružení Nordic Balance Settlement, nicméně s malými úpravami by bylo možné nástroj nasadit i k vizualizaci jiné oblasti energetického trhu.

Vizualizační část aplikace musela překonat dva hlavní problémy, a to proměnlivost struktury vztahů mezi entitami v čase a nutnost přehledně vizualizovat stovky až tisíce entit sdružení NBS. Pro to byly vytvořeny dva různé pohledy na data, které zachycují jejich časové validity. Velké množství entit je přehledně zobrazováno za využití techniky clusteringu.

Vytvořená aplikace je připravena stát se součástí informačního systému BASSE vyvíjeném firmou Unicorn Systems jako nástroj, který umožní uživatelům přehledné zobrazení strukturálních informací o trhu s elektrickou energií. Jako námět pro další výzkum lze uvést například spojení grafové vizualizace s mapovými podklady, již výše zmíněné úpravy pro vizualizaci odlišné domény energetického trhu nebo užší propojení s již existujícím informačním systémem, které by uživateli poskytovalo například možnost úprav vybrané entity.

Literatura

[1] HLINĚNÝ, Petr. *Základy teorie grafů.* Elportál, Brno: Masarykova univerzita, 2010. ISSN 1802-128X.

[2] MATOUŠEK, Jiří, NEŠETŘIL, Jaroslav. *Kapitoly z diskrétní matematiky.* Karolinum Praha, 2000. ISBN 80-246-0084-6.

[3] HOLUBOVÁ, Irena, KOSEK, Jiří, MINAŘÍK, Karel, NOVÁK, David. *Big Data a NoSQL databáze.* 1. vydání. Praha: Grada Publishing, a.s., 2015, 288 s. ISBN 978-80-247-5938-8.

[4] CORMEN, Thomas H. *Introduction to algorithms.* 3rd ed. Cambridge, Mass.: MIT Press, 2009. ISBN 978-0-262-03384-8.

[5] BRANDES, Ulrik et al. GraphML progress report structural layer proposal. In: *International Symposium on Graph Drawing.* Berlin, Heidelberg: Springer, 2001, p. 501-512.

[6] GEXF Working Group. *Gexf 1.2 draft primer.* 2012. Dostupné z: `https://gephi.org/gexf/1.2draft/gexf-12draft-primer.pdf`

[7] WINTER, Andreas. Exchanging graphs with GXL. In: *International Symposium on Graph Drawing.* Berlin, Heidelberg: Springer, 2001, p. 485-500.

[8] Documentation. *Graphviz – Graph Visualization Software* [online]. [cit. 18. 8. 2018]. Dostupné z: `https://graphviz.gitlab.io/documentation/`

[9] HIMSOLT, Michael. *GML: A portable graph file format* [online]. Universität Passau, 1997, [cit. 19. 8. 2018]. Dostupné z: `http://www.uni-passau.de/fileadmin/files/lehrstuhl/brandenburg/projekte/gml/gml-technical-report.pdf`

[10] EIGLSPERGER, Markus et al. Orthogonal graph drawing. In: *Drawing graphs.* Berlin, Heidelberg: Springer, 2001, p. 121-171.

[11] HEALY, Patrick, NIKOLOV, Nikolov S. Hierarchical Drawing Algorithms. *Handbook of Graph Drawing and Visualization.* Hoboken, NJ: Chapman & Hall/CRC, 2013, s. 409-446.

[12] KOBOUROV, Stephen G. *Spring embedders and force directed graph drawing algorithms.* arXiv preprint arXiv:1201.3011, 2012.

[13] JACOMY, Mathieu et al. *ForceAtlas2, a Continuous Graph Layout Algorithm for Handy Network Visualization Designed for the Gephi Software* [online]. 2014, [cit. 29. 7. 2018]. Dostupné z: `http://journals.plos.org/plosone/article?id=10.1371/journal.pone.0098679`

[14] DUNCAN, Christian A. et al. Lombardi Drawings of Graphs. In: *J. Graph Algorithms Appl.* 16(1). 2012, s. 85-108.

[15] FLEISCHER, Rudolf, HIRSH, Colin. Graph drawing and its applications. In: *Drawing graphs.* Berlin, Heidelberg: Springer, 2001, p. 1-22.

[16] WANG, Rui et al. Open source libraries and frameworks for biological data visualisation: A guide for developers. *PROTEOMICS* [online]. 2015, 15(8), 1356-1374 [cit. 24. 3. 2019]. DOI: 10.1002/pmic.201400377. ISSN 16159853. `Dostupnéz:http://doi.wiley.com/10.1002/pmic.201400377`

[17] DOGRUSOZ, Ugur, GENC, Burkay. A multi-graph approach to complexity management in interactive graph visualization. *Computers & Graphics* [online]. 2006, 30(1), 86-97 [cit. 24. 3. 2019]. DOI: 10.1016/j.cag.2005.10.015. ISSN 00978493. Dostupné z: https://linkinghub.elsevier.com/retrieve/pii/S0097849305002128

[18] eSett – Handbook. *eSett* [online]. 2018, [cit. 26. 8. 2018]. Dostupné z: `https://www.esett.com/handbook/`

[19] BANKS, Alex, PORCELLO, Eve. *Learning React: functional web development with React and Redux.* Sebastopol, CA: O'Reilly Media, 2017. ISBN 978-1-491-95462-1.

[20] Introduction to the DOM. *MDN web docs* [online]. 2018, [cit. 18. 01. 2019]. Dostupné z: `https://developer.mozilla.org/en-US/docs/Web/API/Document_Object_Model/Introduction`

[21] ACCOMAZZO, Anthony, MURRAY, Nathaniel, LERNER Ari. *Fullstack React: The Complete Guide to ReactJS and Friends.* Fullstack.io, 2017. ISBN 978-0991344628.

[22] React.Component. *React* [online]. [cit. 18. 01. 2019]. Dostupné z: `https://reactjs.org/docs/react-component.html`

[23] React Lifecycle Methods diagram. *GitHub* [online]. [cit. 18. 01. 2019]. Dostupné z: `https://github.com/wojtekmaj/react-lifecycle-methods-diagram`

[24] Home – jacomyal/sigma.js Wiki *GitHub* [online]. 2018, [cit. 3. 8. 2018]. Dostupné z: `https://github.com/jacomyal/sigma.js/wiki/`

[25] D3.js – Data-Driven Documents. *D3.js – Data-Driven Documents.* [online]. 2018, [cit. 29. 7. 2018]. Dostupné z: `https://d3js.org/`

[26] Gallery – d3/d3 Wiki. *GitHub* [online]. [cit. 31. 3. 2019]. Dostupné z: `https://github.com/d3/d3/wiki/Gallery`

[27] Network examples. *Vis.js* [online]. [cit. 31. 3. 2019]. Dostupné z: http://visjs.org/network_examples.html

[28] Network – nodes. *Vis.js* [online]. [cit. 2. 8. 2018] Dostupné z: http://visjs.org/docs/network/nodes.html

[29] Docs – API. *ECharts* [online]. 2018, [cit. 8. 8. 2018]. Dostupné z: https://ecomfe.github.io/echarts-doc/public/en/api.html#echarts

[30] Get Started – Examples. *ECharts* [online]. 2018, [cit. 8. 8. 2018]. Dostupné z: https://ecomfe.github.io/echarts-examples/public/index.html

[31] FRANZ et al. *Cytoscape.js: a graph theory library for visualisation and analysis.* Bioinformatics, vol. 32, no. 2, 15 January 2016, p. 309–311. Dostupné z: https://doi.org/10.1093/bioinformatics/btv557

[32] Cytoscape.js. *Cytoscape.js* [online]. [cit. 10. 8. 2018]. Dostupné z: http://js.cytoscape.org/

Přílohy

Příloha A: Snímky z aplikací

Production Units Add production unit

Production Unit | View all PU | Code: Test_PU_2069519306 × | RE

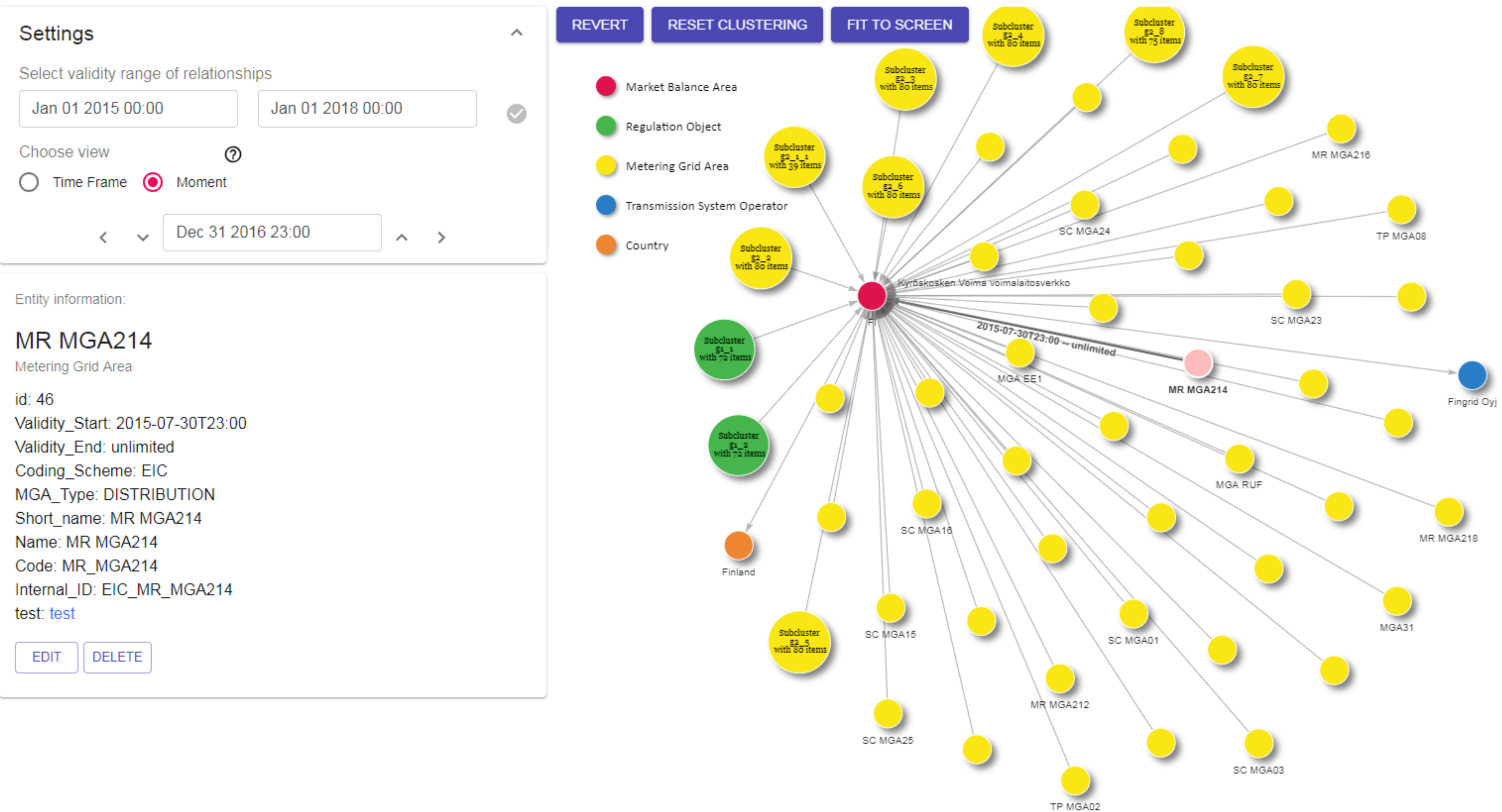
Production Type | MGA | DSO | Type

Valid From	Valid To	Production Unit	Code	Coding Scheme
24.02.2018 00:00	01.06.2018 24:00	Test PU 2069519306	Test_PU_2069519306	EIC

Days	24.02.2018 00:00 - 30.03.2018 24:00	31.03.2018 00:00 - 01.06.2018 24:00	
Valid From	24.02.2018 00:00	31.03.2018 00:00	
Valid To	30.03.2018 24:00	01.06.2018 24:00	
Name	Test PU 2069519306		
Code	Test_PU_2069519306		
Coding Scheme	EIC		
Alternative Code	(not set)		
Alternative Coding Scheme	(not set)		
PU Type	Hydro		
Capacity	100,000		Change
Production Type	Normal		Change
MGA	MGA11		Change
DSO	DSO11		
MBA	NO4		
RE	(not set)	RE12	Change
BRP	(not set)	BRP10	
Regulation Object	(not set)		Change
Green Certificate	(not set)		Change
Internal ID	EIC_TEST_PU_2069519306		

Edit | Change Validity

Obrázek A.1: Ukázka informačního systému BASSE, zdroj: poskytnuto firmou Unicorn Systems



Obrázek A.2: Celá obrazovka vizualizačního nástroje

Příloha B: Struktura API

B.1 Struktura odpovědi koncového bodu getData

```
{
"config": {
    "groups": { // definition of node groups
        "g1": { // group key
            "name": "Regulation Object", // group name, displayed
               in the legend
            "color": { // group color styling
                "background": "#3cb44b", // node background color
                "highlight": {
                    "background": "#ffbcbc" // node background
                       color when clicked on
                }
            }
        },
        "g2": {
            "name": "Metering Grid Area",
            "color": {
                "background": "#ffe119",
                "highlight": {
                    "background": "#ffbcbc"
                }
            },
        },
        // ...other group definitions...
    },
    "clustering": {  // description of clustering. Top level
       clusters have no parent key, subclusters do
        // !!! ID of top level cluster must be the same as some key
             in the groups object for correct styling !!!
        "g1": {
            "name": "Cluster g1\ncontains: {count}" // label of the
                 cluster displayed on the cluster node
                                                  // {count} is
                                                     replaced by
                                                     the number
                                                     of nodes in
                                                     the cluster
        },
        "g2": {
```

```
                "name": "Cluster g2\ncontains: {count}"
            },
            "g1_1": {
                "name": "Subcluster\ng1_1\n with {count} items",
                "parent": "g1" // reference of the parent's key
            },
            "g1_2": {
                "name": "Subcluster\ng1_2\n with {count} items",
                "parent": "g1"
            },
            "g2_1": {
                "name": "Subcluster\ng2_1\n with {count} items",
                "parent": "g2"
            },
            "g2_1_1": {
                "name": "Subcluster\ng2_1_1\n with {count} items",
                "parent": "g2_1"
            },
            // ...
        },
        "range": { // validity dates, used by datepickers,
           YYYYMMDDTHHmm format
            "validityStart": "20150101T0000",
            "validityEnd": "20180101T0000"
        }
    },
    "queriedEntity": { // entity information displayed in the info card
        - for optimization
        "name": "NO3", // entity name displayed in the info card header
        "typeFullName": "Market Balance Area", // entity type name
           displayed in the info card header
        "actions": [ // corresponding buttons are created from this
           array
            {
                "name": "Edit", // name label on the button
                "url": "http://localhost:3000" // target url
            },
            // ...other action definitions...
        ],
        "detail": // string composed of HTML markup, displayed in the
           info card
    },
    "graph": { // connectivity information
        "nodes": [
            {
                "id": "ac3f6ea5-9bbf-11e8-a86c-1c6f65c3aae2", // UUID -
                    required by Vis.js
                "internalId": "EIC_SC_R0106", // identifier used for
```

```
            composing getDetail query URL (should be able to
            query this id in the database)
        "label": "SC R0106", // label of the entity - displayed
             under the node in the vizualization
        "name" : "SC R0106", // name of the entity - displayed
            in the info card header
        "group": "g1", // reference of the group key in the
            config part - used for styling
        "type": "ro", // type identifier used for composing
            getDetail query URL (this should determine the
            target database table)
        "title": "<h3> ac3f6ea5-9bbf-11e8-a86c-1c6f65c3aae2 </
            h3><ul class=\"tooltip-list\"><li>Validity start:
            2017-05-01T00:00</li><li>Validity end: 2018-09-01T00
            :00</li></ul>", // tooltip content (displayed after
            hovering over the node), in HTML markup
        "typeFullName": "Regulation Object" // displayed in the
             info card header
        "clustering": [ // Array of the whole subtree of (sub)
            clusters the node belongs to
             "g1_1",
             "g1"
        ]
    },
    {
        "id": "337ca5f2-ac58-11e8-81d2-1c6f65c3aae2",
        "internalId": "NFI_KY0001",
        "label": "Kyroskosken Voima voimalaitosverkko",
        "name": "Kyroskosken Voima voimalaitosverkko",
        "group": "g2",
        "type": "mga",
        "title": "<h3> Kyroskosken Voima voimalaitosverkko </h3
            ><ul class=\"tooltip-list\"><li>Type: mga</li></ul>"
            ,
        "typeFullName": "Metering Grid Area",
        "clustering": [
            "g2_1_2",
            "g2_1",
            "g2"
        ]
    },
    // ...other nodes definitions...
],
"edges": [
    {
        "from": "ac3f6ea5-9bbf-11e8-a86c-1c6f65c3aae2", // id
            of the node
        "to": "337ca5f2-ac58-11e8-81d2-1c6f65c3aae2", // id of
```

```
        the node
      "validityStart": "2017-05-01T00:00", // starting date
        of the relationship validity, YYYY-MM-DDTHH:MM
        format, used for the filtering
      "validityEnd": "2018-09-01T00:00", // end date of the
        relationship validity, YYYY-MM-DDTHH:MM format, used
         for the filtering
      "hiddenLabel": "2017-05-01T00:00 -- 2018-09-01T00:00",
        // used for displaying edge label after clicking on
        it
      "validityChanges": true // true if validity of this
        relationship STARTS AFTER the queried validity start
         date or ENDS BEFORE the end of the validity end
        date
    },
    // ...other edges definitions...
  ]
}
}
```

Ukázka kódu B.1: Struktura odpovědi koncového bodu getData

B.2 Struktura odpovědi koncového bodu getDetail

```
{
"queriedEntity": {
  "actions": [ // corresponding buttons are created from this
     array
    {
      "name": "Edit", // name label on the button
      "url": "http://localhost:3000" // target URL
    },
    {
      "name": "Delete",
      "url": "http://localhost:3000"
    }
  ],
  "detail": ""  // string composed of HTML markup, displayed in
     the info card
}
}
```

Ukázka kódu B.2: Struktura odpovědi koncového bodu getDetail

Univerzita Hradec Králové
Fakulta informatiky a managementu
Akademický rok: 2018/2019

Studijní program: Aplikovaná informatika
Forma: Prezenční
Obor/komb.: Aplikovaná informatika (ai2-p)

Podklad pro zadání DIPLOMOVÉ práce studenta

PŘEDKLÁDÁ:	ADRESA	OSOBNÍ ČÍSLO
Bc. Bořík Pavel	Nové Město nad Metují	I1700480

TÉMA ČESKY:

Vizualizace vztahů ve sdružení NBS

TÉMA ANGLICKY:

Visualization of relationships in the NBS group

VEDOUCÍ PRÁCE:

Mgr. Jiří Haviger, Ph.D. - KIKM

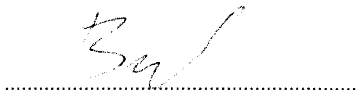
ZÁSADY PRO VYPRACOVÁNÍ:

Cílem práce je vytvoření front-endové části nástroje, který bude zobrazovat vztahy mezi subjekty participujícími ve sdružení Nordic Imbalance Settlement (NBS) ve formě orientovaného grafu. Nástroj by měl být následně využitelný ve webové aplikaci, která funguje jako informační systém NBS.

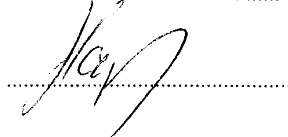
Osnova:
1. Úvod, 2. Grafy a jejich vizualizace, 3. Sdružení Nordic Imbalance Settlement, 4. Analýza vizualizačního nástroje, 5. Použité technologie, 6. Knihovny pro vizualizaci grafových struktur, 7. Vývoj aplikace, 8. Shrnutí výsledků, 9. Závěr

SEZNAM DOPORUČENÉ LITERATURY:

HLINĚNÝ, Petr. Základy teorie grafů. Elportál, Brno: Masarykova univerzita, 2010. ISSN 1802-128X.
TAMASSIA, Roberto. Handbook of Graph Drawing and Visualization. 2014.
BANKS, Alex, PORCELLO, Eve. Learning React: functional web development with React and Redux. Sebastopol, CA: O'Reilly Media, 2017. ISBN 978-1491954621.

Podpis studenta: 

Datum: 2. 10. 2018

Podpis vedoucího práce: 

Datum: 2. 10. 2018