



**MATEMATICKO-FYZIKÁLNÍ  
FAKULTA**  
Univerzita Karlova

## **BAKALÁŘSKÁ PRÁCE**

Štěpán Stenclák

# **Vizuální browser grafových dat**

Katedra softwarového inženýrství

Vedoucí bakalářské práce: doc. Mgr. Martin Nečaský, Ph.D.

Studijní program: Informatika

Studijní obor: Softwarové a datové inženýrství

Praha 2020

Prohlašuji, že jsem tuto bakalářskou práci vypracoval(a) samostatně a výhradně s použitím citovaných pramenů, literatury a dalších odborných zdrojů. Tato práce nebyla využita k získání jiného nebo stejného titulu.

Beru na vědomí, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., autorského zákona v platném znění, zejména skutečnost, že Univerzita Karlova má právo na uzavření licenční smlouvy o užití této práce jako školního díla podle §60 odst. 1 autorského zákona.

V ..... dne .....

Podpis autora

Chtěl bych poděkovat svému vedoucímu doc. Mgr. Martinovi Nečaskému, Ph.D. za jeho čas a pomoc, kterou mi poskytl při vypracování této práce. Současně mu děkuji za pomoc s výběrem tohoto tématu, které mě bavilo řešit. Dále děkuji své rodině, která mi umožnila v klidu práci dokončit.

Název práce: Vizuální browser grafových dat

Autor: Štěpán Stenclák

Katedra: Katedra softwarového inženýrství

Vedoucí bakalářské práce: doc. Mgr. Martin Nečaský, Ph.D., Katedra softwarového inženýrství

Abstrakt: Jedním ze způsobů, jak publikovat data ve strojově čitelné podobě na internetu, je formou grafu. Taková data je pak velmi snadné propojovat mezi různými datovými zdroji a vytvořit tak velkou síť propojených dat. Abychom tyto data mohli vizualizovat, musíme používat nástroje na procházení grafu, které mohou být často nepraktické, neboť nám obvykle zobrazí veškeré informace, kterých mohou být stovky.

Cílem této práce je vytvořit webovou aplikaci, která je schopna tyto grafová data vizualizovat, poskytovat k nim informace a procházet je pomocí takzvaných konfigurací. Konfigurace popisuje, jaké data z velké množiny dat na internetu chceme zobrazit, jak je vizualizujeme a jak se na ně můžeme dívat. Dovolí nám tedy odfiltrovat stovky nezajímavých informací a odstíní tak uživatele od složité sítě propojených dat, aby se mohl věnovat pouze těm, která ho zajímají.

Klíčová slova: RDF databáze graf linked data webová aplikace

Title: Visual browser of graph data

Author: Štěpán Stenclák

Department: Department of Software Engineering

Supervisor: doc. Mgr. Martin Nečaský, Ph.D., Department of Software Engineering

Abstract: One way to publish data in a machine-readable form on the Internet is in the form of a graph. Such data are easy to interconnect between different data sources and thus create a large network of linked data. In order to visualize this data, we need to use graph tools, which can often be impractical because they usually show us all the information about the graph.

The aim of this work is to create a web application that is able to visualize those graph data, provide information about them, and browse them using so-called configurations. The configuration describes what data from a large set of data on the Internet we want to display, how we visualize them, and how we can look at them. It allows us to filter out hundreds of uninteresting information, thus shielding users from a complex network of linked data so that users can focus on the data that interests them.

Keywords: RDF database graph linked data web application

# Obsah

<b>Úvod</b>	<b>3</b>
0.1 Cíl práce . . . . .	4
<b>1 Technické předpoklady</b>	<b>5</b>
1.1 Resource Description Framework . . . . .	5
1.1.1 Turtle jazyk . . . . .	5
1.1.2 SPARQL jazyk . . . . .	7
<b>2 Analýza požadavků</b>	<b>9</b>
2.1 Konfigurace . . . . .	9
2.1.1 Meta konfigurace . . . . .	11
2.1.2 Konfigurace . . . . .	11
2.1.3 ViewSet . . . . .	12
2.1.4 View . . . . .	12
2.1.5 Expanze . . . . .	12
2.1.6 Preview (náhled) . . . . .	13
2.1.7 Detail . . . . .	13
2.1.8 Dataset . . . . .	13
2.1.9 Visual style sheet . . . . .	13
2.2 Server . . . . .	15
2.3 Uživatelské a systémové požadavky . . . . .	16
<b>3 Návrh architektury</b>	<b>22</b>
3.1 Server . . . . .	22
3.1.1 Jazyková podpora . . . . .	22
3.1.2 API . . . . .	23
3.2 Klientská část aplikace . . . . .	27
3.2.1 Moduly . . . . .	27
<b>4 Implementace</b>	<b>31</b>
4.1 Vue.js framework . . . . .	31
4.1.1 Vuex . . . . .	31
4.1.2 Vue framework . . . . .	32
4.1.3 Loaders . . . . .	33
4.2 Cytoscape knihovna . . . . .	35
4.3 Prostředí . . . . .	36
4.3.1 Instalace . . . . .	36
4.3.2 Adresářová struktura . . . . .	37
4.4 Programátorská dokumentace . . . . .	38
4.4.1 Vstupní skript a pomocné soubory . . . . .	38
4.4.2 Komponenta Application . . . . .	38
4.4.3 Interface file-save/ObjectSave . . . . .	40
4.4.4 Třída Graph . . . . .	40
4.4.5 Třída NodeCommon . . . . .	43
4.4.6 Třída Node . . . . .	44

4.4.7	Třída NodeGroup . . . . .	45
4.4.8	Třídy EdgeCommon, Edge, GroupEdge . . . . .	47
4.4.9	Třída NodeViewSet . . . . .	47
4.4.10	Třída NodeView . . . . .	47
4.4.11	Třída Expansion . . . . .	48
4.4.12	Komponenta GraphArea . . . . .	49
4.4.13	Komponenty GraphElementNode, GraphElementNodeGroup a GraphElementNodeMixin . . . . .	50
4.4.14	Definice filtrů . . . . .	51
4.4.15	Definice Layoutů . . . . .	53
4.4.16	Abstraktní třída Layout . . . . .	53
4.4.17	LayoutManager . . . . .	54
4.4.18	Vyhledávání vrcholů . . . . .	54
<b>5</b>	<b>Uživatelské testování</b>	<b>56</b>
5.1	System Usability Scale (SUS) . . . . .	56
5.1.1	Výsledky testování SUS . . . . .	57
5.2	Výsledky obecných otázek . . . . .	57
<b>Závěr</b>		<b>58</b>
5.3	Možnosti vylepšení . . . . .	58
5.4	Aplikace . . . . .	60
<b>Seznam použité literatury</b>		<b>63</b>
<b>Seznam obrázků</b>		<b>64</b>
<b>Seznam tabulek</b>		<b>65</b>
<b>Seznam použitých zkratk</b>		<b>66</b>
<b>A Přílohy</b>		<b>67</b>
A.1	První příloha . . . . .	67

# Úvod

Na internetu je dnes možné najít téměř cokoli, kupříkladu stav počasí, encyklopedické informace, odborné publikace, jízdní řády a podobně. Tyto informace jsou uloženy jako webové stránky systému WWW (World Wide Web) a kterýkoli uživatel internetu má k nim přístup a může z nich čerpat.

Pro lidi je systém WWW vyhovující zdroj informací, nicméně narazíme na problém, pokud chceme tyto informace číst strojově. WWW totiž nepopisuje, jak by měly být informace na internetu prezentovány. Každý publikovatel si může data zveřejnit jinak, například tabulkou, odrážkovým seznamem, nebo ve větách. Tyto informace jsou stále snadno čitelné pro člověka, ale obtížně čitelné pro stroj.

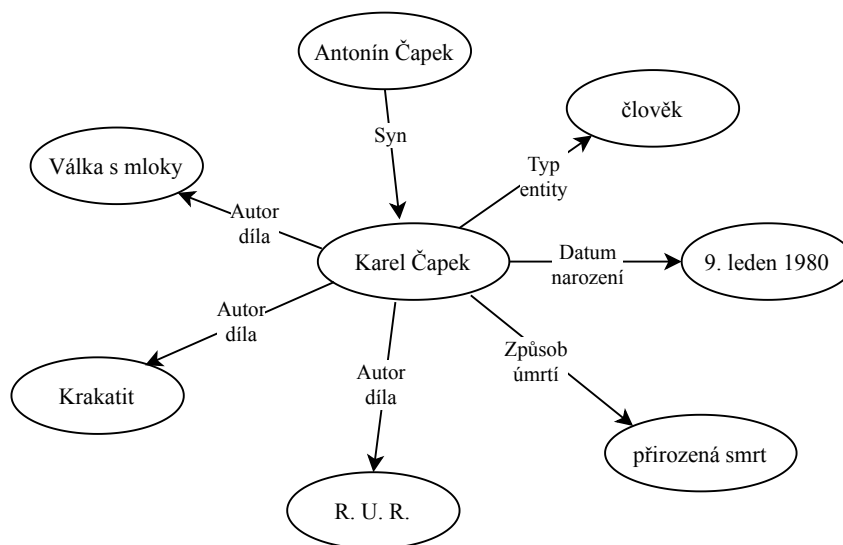
Možností řešení tohoto problému je zveřejňovat data i ve strojové podobě. Nabízí se například tabulky ve formátu CSV, nebo komplikovanější data ve formátu JSON a XML. Zde je již snadné číst data a dále je zpracovávat, ale programátor je stále nucen pochopit a přizpůsobit program na rozhraní těchto dat.

Možným řešením se nabízí popsat data pomocí RDF frameworku, jež je důkladněji rozebrán v následující kapitole.

Tato práce se zabývá reprezentací dat popsaných právě pomocí RDF. RDF reprezentuje entity z reálného světa jako vrcholy grafu a jejich vlastnosti pomocí hran propojující tyto entity. Data uložená v grafových databázích jsou snadno čitelná počítačovými programy a je jednoduché propojovat různé datové zdroje do větších celků a provádět nad nimi dotazy.

Jako veškerá strojová data je potřeba i tyto grafová data vizualizovat. Příkladem může být datový analytik, který se chce přesvědčit, že jsou data uložena tak, jak bylo zamýšleno.

Jako modelový příklad vizualizace uvedme entitu Karla Čapka a některá jeho data, jež má o něm uložená Wikipedie.

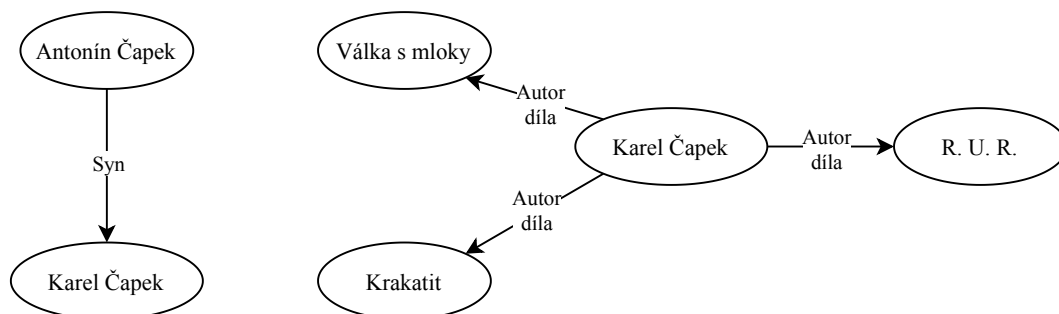


Obrázek 1: Ukázka části grafu jež může reprezentovat Karla Čapka.

Jak vidíme z obrázku 1, zobrazit všechna data k uzlu může být velmi nepřehledné a takovýchto uzlů může být stovky. Pokud nás například zajímají pouze

díla Karla Čapka, určitě v grafu nebudeme chtít mít informaci, že je Karel Čapek člověk. Tuto situaci můžeme vyřešit tak, že se na entity grafu budeme dívat pouze určitým pohledem a tedy zobrazíme pouze ty vztahy k vrcholu, jež odpovídají danému pohledu.

Kupříkladu pokud bychom chtěli procházet rodokmenem, je vhodné se na Karla Čapka dívat jako na osobu, jež má rodiče a sama může být rodičem ostatních osob. V tuto chvíli nás nezajímají ostatní vlastnosti jako knihy, které napsal, nebo ocenění, která získal. Na Karla Čapka se ale můžeme dívat i jako na spisovatele, kde nás pak zajímají pouze jeho díla a rodinné vztahy můžeme skrýt. Příklad takovýchto pohledů je na obrázku 2.



Obrázek 2: Pohled na Karla Čapka jako na osobu mající rodinu (vlevo) a na spisovatele jež je autorem literárních děl (vpravo)

Tento způsob procházení grafových dat sice vyžaduje předem nadefinovat pohledy, ale procházení dat je pak jednoduché a velmi přehledné.

## 0.1 Cíl práce

Cílem této práce je vyrobit webovou aplikaci, jež by na základě předem definovaných pohledů vizualizovala grafová data a umožňovala uživateli procházet tento graf a objevovat nové vrcholy. Protože je žádoucí mít více pohledů na konkrétní typ vrcholu, pohledy jsou seskupeny do takzvaných konfigurací. Konfigurace pak popisuje nejen pohledy na různé typy vrcholů, ale i jaké vrcholy lze takto vizualizovat a z jakých zdrojů čerpat.

Konfigurace, jež jsou použité v aplikaci jsou například:

- **Procházení živočichů na Wikidatech** - Uživatel může vizualizovat jednotlivé rostlinné a živočišné druhy a procházet je podle rozdělení do taxonů.
- **Procházení slavných osobností na Wikidatech** - Uživatel může procházet slavné osobnosti a nechat si načíst jejich filmová a literární díla a rodinné vztahy.

Kromě procházení si bude uživatel moci zobrazit detailní informace ke konkrétnímu vrcholu v závislosti na pohledu.



# 1. Technické předpoklady

## 1.1 Resource Description Framework

Resource Description Framework, zkráceně RDF, je rodina specifikací<sup>1</sup> (tedy sada pravidel), která se používá na popis informací na internetu. RDF popisuje data jako graf, konkrétně vrcholy grafu jsou entity, jež ztvárňují nějaké věci (fyzické předměty, díla, myšlenky) a hrany tyto entity propojují a dávají jim vztahy.

Základem je takzvaný **statement** (česky tvrzení). Tvrzení je vyjádřeno formou trojice v tomto pořadí ze subjektu, predikátu a objektu, přičemž subjekt a objekt jsou vrcholy (anglicky **nodes**) a predikát je orientovanou hranou jdoucí od subjektu k objektu.

Vrcholem v RDF může být IRI, literál, nebo prázdný uzel.

Jako vrchol IRI (Internationalized Resource Identifier) se rozumí vrchol, jež přiřazuje entitě nějaký identifikátor ve tvaru IRI. Tímto jsme schopni v rámci WWW identifikovat různé entity a pracovat s nimi napříč různými zdroji. Ku příkladu Karel Čapek, zmíněný v úvodu, je v rámci Wikidat jednoznačně identifikován jako <https://www.wikidata.org/wiki/Q155855>. IRI kromě identifikátoru nenese žádné další informace včetně jména, reprezentuje tedy nějakou entitu, která musí být popsána tvrzeními, aby dostala význam.

Literálem je pak uzel nesoucí nějakou hodnotu. Může se jednat o číslo popisující věk osoby, její jméno atp. V rámci RDF kromě hodnoty má literál i svůj typ, jež je opět vyjádřen pomocí IRI. Uvedme kupříkladu <http://www.w3.org/2001/XMLSchema#integer> jež vyjadřuje obecně celé číslo. Speciálním typem je <http://www.w3.org/1999/02/22-rdf-syntax-ns#langString>, který vyžaduje ještě takzvaný **language tag** a popisuje text v nějakém konkrétním jazyce.

Literály nám tedy dávají možnost popsat IRI uzly (entity) pomocí různých vlastností, které podporují právě i multijazyčnost. S pomocí takovýchto grafů pak dokážeme popsat celou řadu věcí z reálného světa.

Nakonec, hrana je opět popsána pomocí IRI, jež reprezentuje typ vztahu.

### 1.1.1 Turtle jazyk

RDF graf může být popsán pomocí Turtle jazyka<sup>2</sup>. Ten je využit při zápisu konfigurací, jež jsou popsány dále v tomto dokumentu.

Trojice (tvrzení) se zapisují jako tři slova oddělená mezerami a zakončená tečkou. Chceme-li zapsat IRI, musíme jej dát do špičatých závorek <>. Na začátku dokumentu lze definovat prefixy, které pak umožňují zkrátit zapisované IRI do namespace a zbylé části jako **namespace:zbytek**.

Pokud se nám u trojic opakuje subjekt a predikát, můžeme jednotlivé objekty oddělovat čárkou (,). Obdobně, pokud se nám opakuje jen subjekt, můžeme dvojice predikát-objekt oddělovat středníkem (;).

---

<sup>1</sup>Raimond a Schreiber (2014)

<sup>2</sup>Prud'hommeaux a Carothers (2014)

Uvedme několik příkladů RDF grafu.

**Příklad.** Tvrzení „Karel Čapek se narodil 9. ledna 1890“ vyjádřené v rámci Wikidat.

```
@prefix wd: <https://www.wikidata.org/wiki/> .  
@prefix wdt: <https://www.wikidata.org/wiki/Property:> .  
@prefix xsd: <http://www.w3.org/2001/XMLSchema#> .  
  
wd:Q155855 wdt:P569 "1890-01-09"^^xsd:dateTime .
```

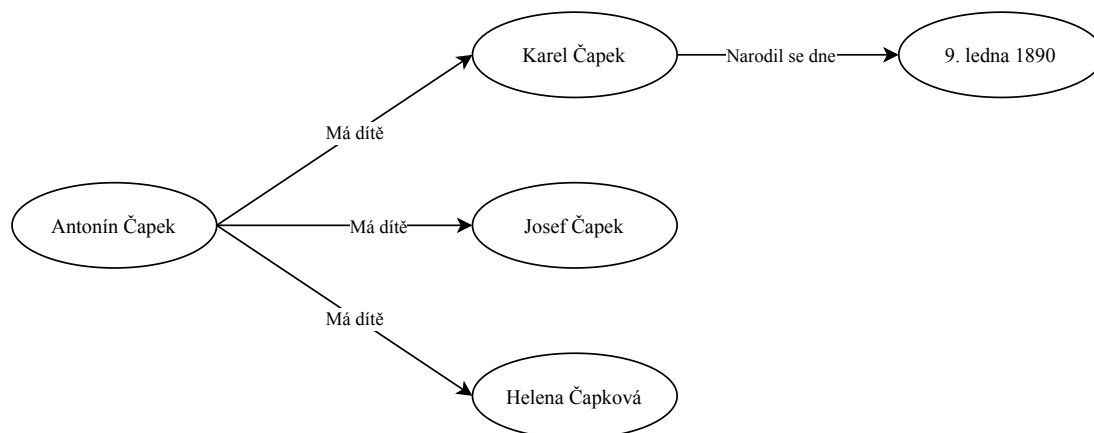
Kód popisuje jedno tvrzení, kdy se Karlu Čapkovi <https://www.wikidata.org/wiki/Q155855> přiřadí přes property datum narození <https://www.wikidata.org/wiki/Property:P569> literál s jeho datem narození, jež má typ <http://www.w3.org/2001/XMLSchema#dateTime>.

Jak lze vidět z příkladu, tvrzení nemusí odkazovat jen na svůj dataset, ale i mimo něj. To nám dovoluje stavět již na existujících datasetech a jednoduše se na ně odkazovat přes IRI. Můžeme tak mít vlastní knihovní databázi, jež ke knize přiřadí autora z datasetu Wikidat. Tímto jsme propojili dva datasety, což nám umožní nad nimi provádět dotazy. Příkladem takového dotazu by mohlo být „Chci seznam knih v naší knihovně, jež byly napsány českými autory.“ Takový dotaz pak současně prohledá dva různé datasety a vrátí očekávané výsledky.

Uvedme ještě příklad tvrzení, jež se odkazuje na další entity.

**Příklad.** Tvrzení „Děti Antonína Čapka jsou Karel, Josef a Helena“ vyjádřené v rámci Wikidat.

```
@prefix wd: <https://www.wikidata.org/wiki/> .  
@prefix wdt: <https://www.wikidata.org/wiki/Property:> .  
  
wd:Q6657059 wdt:P40 wd:Q155855 ,  
                wd:Q454568 ,  
                wd:Q4532606 .
```



Obrázek 1.1: Příklad grafu, který získáme z předešlých dvou ukázek.

Pro úplnost zmiňme ještě následující graf popisující vztahy mezi lidmi, které jsou vyjádřeny ontologií FOAF (friend of a friend). Pokud by všechny zdroje popisující lidi využívaly tuto ontologii, měli bychom jednotné rozhraní, jak přistupovat k lidským vztahům.

Ontologie je slovník obsahující formalizovaný seznam pojmů na definici kategorií a vztahů z určitého oboru.

```
@prefix foaf: <http://xmlns.com/foaf/0.1/> .
@prefix example: <http://example.org/> .

<http://example.org/Jan-Novák> foaf:name "Jan Novák" .
<http://example.org/Ondřej-Novák> foaf:name "Ondřej Novák" .

example:Pavel-Novotný foaf:name "Pavel Novotný" ;
                      foaf:knows example:Jan-Novák ,
                               example:Ondřej-Novák .
```

### 1.1.2 SPARQL jazyk

Jazyk SPARQL<sup>3</sup> slouží k definici dotazů nad RDF daty a jejich manipulaci. SPARQL má 4 typy dotazů:

- **SELECT** dotaz vrací data z databáze ve formě tabulky obdobně jako u tabulkových databází. Můžeme tak například k jedné entitě vrátit hned několik vlastností jako různé sloupce tabulky.
- **CONSTRUCT** dotaz vrací data ve formě RDF grafu. Tento dotaz tedy použijeme, pokud chceme s výsledkem dále pracovat jako s grafem.
- **ASK** dotaz vrací pravdivostní hodnotu ano/ne podle toho, zda dotaz uspěl. Může být využit například ke zjištění, zda se konkrétní data již v databázi nacházejí.
- **DESCRIBE** dotaz obdobně jako **CONSTRUCT** vrací RDF graf s tím rozdílem, že zde určuje databáze, jaká data na dotaz dostaneme. Tento dotaz použijeme, pokud neznáme strukturu grafu a chceme získat „nějaké“ informace o vrcholu.

---

<sup>3</sup>Seaborne a Harris (2013)

**Příklad.** Dotaz, jež nám vrátí seznam děl Karla Čapka a jejich data vydání z Wikidat.

```
PREFIX wd: <https://www.wikidata.org/wiki/>
PREFIX wdt: <https://www.wikidata.org/wiki/Property:>

SELECT ?title ?publication_date
WHERE
{
    wd:Q155855 wdt:P800 ?work .
    ?work wdt:P1476 ?title ;
        wdt:P577 ?publication_date .
}
```

Jedná se o **SELECT** dotaz, tedy jako výsledek dostaneme tabulku, jež má sloupce `title` a `publication_date`.

Dotaz se nejprve zeptá na všechny vrcholy, které dostaneme přes vlastnost `wdt:P800`, kterou Wikidata definují jako „dílo - významné vědecké, umělecké či jiné dílo“. Tyto vrcholy jsou pak reprezentovány proměnnou `?work`. Na dalších dvou řádcích se pak dotazujeme, jaký titulek (`wdt:P1476`) má `?work` a uložíme ho do `?title`, který je i prvním sloupcem výstupu. Dále se pak obdobně ptáme na datum publikace, které je také součástí výsledku.

**Příklad.** Dotaz, zda vrchol identifikován jako „Q7377“ je taxon, nebo klad (dle Wikidat definovaný jako „systematická jednotka v biologii zahrnující společného předka a všechny z něho vzešlé potomky“).

```
ASK {
    wd:Q7377 wdt:P31 ?type .
    FILTER(?type IN (wd:Q16521, wd:Q713623))
}
```

Protože jako odpověď očekáváme ano/ne, jedná se o **ASK** dotaz. Obdobně jako v minulém příkladu ve **WHERE** části popisujeme podmínky, které pak určí, zda existuje nějaká trojice a tedy zda dotaz vrátí kladný výsledek. Všimněme si funkce **FILTER**, která dovoluje typu mít jen jednu ze dvou hodnot.

**Příklad.** Dotaz, který vrací graf dětí Antonína Čapka (Q6657059) v ontologii <http://purl.org/vocab/relationship/>.

```
PREFIX relationship: <http://purl.org/vocab/relationship/>
CONSTRUCT {
    ?child relationship:childOf wd:Q6657059 .
} WHERE {
    wd:Q6657059 wdt:P40 ?child .
}
```

Nejprve specifikujeme, jak chceme, aby výsledný graf vypadal. Poté ve **WHERE** části určíme podmínky a tím i proměnné použité v první části dotazu.

## 2. Analýza požadavků

*Tato kapitola popisuje požadavky, jež byly kladeny na vývoj systému.*

Cílem práce je vyrobit webovou aplikaci, která vizualizuje grafová data dle předem definovaných konfigurací a je schopna s těmito daty pracovat. Konfigurace popisují jak a která data mají být vizualizována a jak je možné graf dále rozšiřovat. Celá aplikace bude rozdělena na klientskou a serverovou část, přičemž server odstiňuje klienta od RDF modelu a vrací mu již zpracovaná data.

Součástí požadavků na webovou aplikaci bylo vedoucím této práce připraveno několik konfigurací a webový server s určeným rozhraním, se kterým má klientská aplikace komunikovat.

V následujících kapitolách je nejprve popsána struktura konfigurací, rozhraní serveru a následně uživatelské požadavky na klientskou část aplikace.

### 2.1 Konfigurace

**Konfigurace** popisuje určitý pohled na data a definuje, jaká data lze v rámci této konfigurace vizualizovat a jakými způsoby. Konfigurace má **množiny pohledů**, kdy každá množina je aplikovatelná jen na určitou skupinu typů vrcholů podporovaných konfigurací. Množina pohledů pak obsahuje jednotlivé **pohledy**, které popisují, jak se na konkrétní vrchol můžeme dívat. Každý pohled pak určuje **náhled** (preview), **detail** a **expanzi**. Náhled poskytuje základní informace o vrcholu v rámci pohledu a určuje, jak má být vrchol v grafu vykreslen. Detail pak tyto informace rozšiřuje o další, které mohou být vypsány v tabulce. Expanze popisuje RDF graf, který doplňuje konkrétní vrchol o nové vztahy v rámci pohledu.

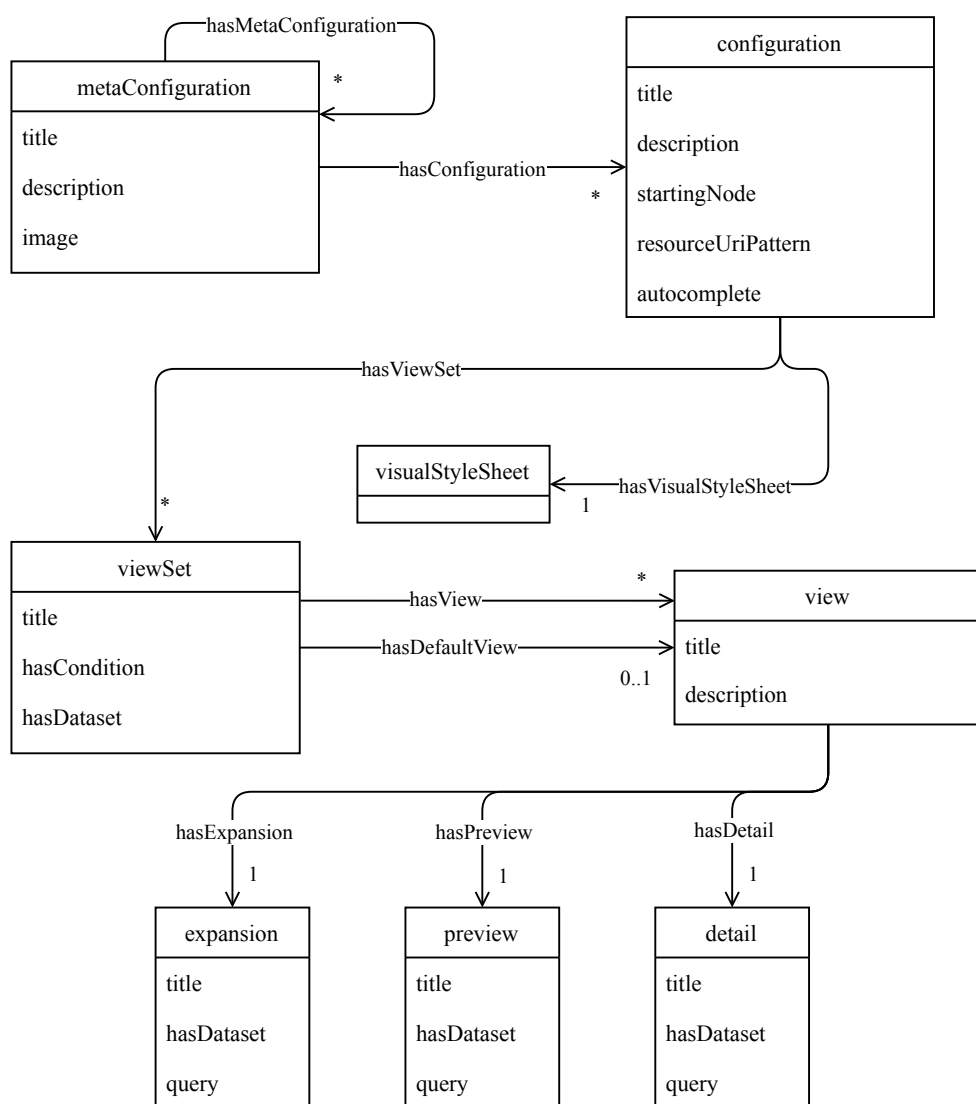
Konfigurace také určuje **počáteční vrchol** a **stylesheet**, tedy pravidla, jak mají být uzly v aplikaci vizualizovány.

Nakonec, konfigurace mohou být uloženy v rámci **meta konfigurací**, které slouží jako složky pro konfigurace.

Obrázek 2.1 popisuje UML diagram konfigurací.

V následujících sekcích jsou použity tyto RDF namespaces:

Prefix	IRI
browser	<code>https://linked.opendata.cz/ontology/knowledge-graph-browser/</code>
dct	<code>http://purl.org/dc/terms/</code>



Obrázek 2.1: Class diagram konfigurací včetně pozdější implementace meta konfigurace a rozšíření konfigurace.

### 2.1.1 Meta konfigurace

Meta konfigurace je skupina pro další konfigurace a meta konfigurace. Díky tomuto může uživatel procházet desítky různých konfigurací uspořádaných do složek obdobně jako v souborovém systému na počítači. Meta konfigurace má tyto vlastnosti:

- **dct:title** Název meta konfigurace. *(je možné zadat ve více jazycích)*
- **dct:description** Detailnější popis, co daná meta konfigurace zahrnuje. *(je možné zadat ve více jazycích)*
- **browser:image** URL adresa s obrázkem reprezentující meta konfiguraci. Obrázek je pak zobrazen v aplikaci při procházení konfigurací.
- **browser:hasMetaConfiguration** Meta konfigurace, které spadají pod tuto meta konfiguraci. *(je očekáváno více objektů)*
- **browser:hasConfiguration** Konfigurace, které spadají pod tuto meta konfiguraci. *(je očekáváno více objektů; popsáno dále)*

Meta konfigurací může být například „Wikidata“ nebo „Otevřená data ČR“.

### 2.1.2 Konfigurace

Konfigurace popisuje způsoby, jakými se lze dívat na data. Dvě konfigurace již byly zmíněny v úvodní kapitole. Aktuálně může mít konfigurace tyto vlastnosti:

- **dct:title** Název konfigurace. *(je možné zadat ve více jazycích)*
- **dct:description** Detailnější popis čeho je možné s konfigurací dosáhnout. *(je možné zadat ve více jazycích)*
- **browser:hasVisualStyleSheet** Určuje, jak mají být uzly v aplikaci vizualizovány. *(popsáno dále)*
- **browser:startingNode** Doporučený uzel nebo uzly, se kterými začít s procházením grafu. *(je očekáváno více objektů)*
- **browser:resourceUriPattern** Regulární výraz popisující, jak by mělo vypadat IRI uzlu. Používá se v aplikaci jako nápověda uživateli, zda zadal správné IRI dříve, než se pošle požadavek. Také se používá pro sestavení IRI z vyhledávaného dotazu, pokud lze dotazem nahradit část výrazu a získat tak validní IRI.
- **browser:hasViewSet** Seznam množin pohledů které tato konfigurace podporuje. *(popsáno dále)*
- **browser:autocomplete** JSON soubor se seznamem RDF uzlů podle kterých probíhá hledání. *(je očekáváno více objektů; popsáno dále)*

### 2.1.3 ViewSet

View set reprezentuje skupinu pohledů. Skupina pohledů je vždy aplikovatelná na určité typy vrcholů v rámci konfigurace. View set má následující vlastnosti:

- `dct:title` Název view setu. *(je možné zadat ve více jazycích)*
- `browser:hasView` Pohledy, které patří pod tento view set. *(je očekáváno více objektů; popsáno dále)*
- `browser:hasDefaultView` Výchozí pohled ze seznamu výše.
- `browser:hasCondition` SPARQL ASK dotaz jež určí, zda tato množina pohledů je aplikovatelná na konkrétní vrchol.
- `browser:hasDataset` Dataset, vůči kterému probíhá ASK dotaz. *(popsáno dále)*

### 2.1.4 View

Konkrétní pohled na vrchol, který určuje náhled, detail a expanzi. Jak již bylo zmíněno v úvodu, pohledem může být například „člověk, jež je autorem literárních děl“, nebo „člověk jako součást rodokmenu“. View má následující vlastnosti:

- `dct:title` Název pohledu. *(je možné zadat ve více jazycích)*
- `dct:description` Popis pohledu. *(je možné zadat ve více jazycích)*
- `browser:hasExpansion` Expanze - popisuje graf který rozšiřuje vrchol o nové vrcholy v rámci daného pohledu. *(popsáno dále)*
- `browser:hasPreview` Preview (náhled) - určuje data která popisují vrchol v rámci daného pohledu a na základě kterých se vrchol vizuálně vykreslí *(popsáno dále)*
- `browser:hasDetail` Detail - určuje dodatečné informace o vrcholu v rámci daného pohledu. *(popsáno dále)*

### 2.1.5 Expanze

Expanze popisuje, jak lze daný vrchol rozšířit o nové vrcholy, které s ním souvisí. Expanze vrací graf, tedy expandované uzly nemusí být přímými sousedy expandovaného vrcholu. Jako expanzi si můžeme představit například „Zobraz všechny knihy, co napsala daná osoba“. Expanze formálně patří k pohledu (view).

- `dct:title` Název expanze. *(je možné zadat ve více jazycích, aktuálně se nepoužívá)*
- `browser:hasDataset` Popisuje dataset, vůči kterému se dotazuje na data. *(popsáno dále)*
- `browser:query` Popisuje SPARQL CONSTRUCT dotaz, který bude spuštěn na endpointu datasetu a vrátí výsledný graf.



### 2.1.6 Preview (náhled)

Preview určuje, která data v rámci daného pohledu (view) popisují konkrétní vrchol. Popisem se myslí taková data, podle kterých se určí, jak bude vrchol na grafu vizuálně vykreslen. Obdobně jako expanze, preview patří ke konkrétnímu pohledu.

Preview má stejné vlastnosti jako expanze. `browser:query` v tomto případě popisuje SPARQL CONSTRUCT dotaz, který vrátí graf obsahující daný uzel společně s literály ze kterých bude sestaven preview.

Konkrétně pro preview je predikát `browser:class` považován za třídu uzlu. Podle těchto tříd je pak možné nastavovat vizuální styly pro konkrétní vrcholy.

### 2.1.7 Detail

Detail poskytuje dodatečné informace k uzlu. Může se jednat o literály, které nemá smysl v dané konfiguraci vykreslit do grafu jako uzly, a proto budou zobrazeny v bočním panelu aplikace po kliknutí na uzel.

Detail má stejné vlastnosti včetně `browser:query` jako preview.

### 2.1.8 Dataset

Dataset popisuje SPARQL endpoint vůči kterému probíhá dotazování na data.

- `dct:title` Název datasetu. (*aktuálně se nepoužívá*)
- `void:sparqlEndpoint` URL adresa SPARQL endpointu na kterou se posílají dotazy.
- `browser:accept` Popisuje HTTP Accept type - v jakém formátu by měl endpoint svá data poskytnout.

### 2.1.9 Visual style sheet

Popisuje sadu pravidel, podle kterých budou vizuálně vykresleny vrcholy v aplikaci. Pravidla popsaná v rámci style sheetu mohou pracovat náhledem (preview) včetně zmíněných tříd. Forma zápisu stylů aktuálně odpovídá stylům, jaké používá knihovna Cytoscape<sup>1</sup>, která bude popsána v další kapitole.

Visual style sheet má pouze jednu vlastnost

- `browser:hasVisualStyle` Pravidlo popisující, jak se má provést stylování. Odkazuje na **Visual style**.

#### Visual style

Visual style má pak:

- `browser:hasSelector` Selector pro Cytoscape knihovnu, jež vybírá, na které uzly nebo hrany bude daný styl aplikován.

---

<sup>1</sup><https://js.cytoscape.org/>

- **browser:\*** Konkrétní styly, jak má být daný uzel vykreslen. Používají se přesně ty názvy, které používá knihovna Cytoscape. Kupříkladu `browser:border-width` nebo `browser:background-color`.

Vedoucím projektu byly dodány pouze konfigurace. Výše sepsaná dokumentace konfigurací je již součástí této práce.

Původní návrh konfigurací byl mírně jednodušší a považoval konfigurace pouze jako odkazy na seznam pohledů. Klientská aplikace tedy musela mít seznam konfigurací s jejich názvy, počátečními vrcholy a dalšími parametry.

Na základě debaty s vedoucím byla konfigurace rozšířena o meta konfigurace a o parametry u konfigurace, jako název, popis, počáteční vrcholy atp. To nám umožnilo mít v rámci klientské aplikace pouze jeden odkaz na IRI meta konfigurace, ze které se dají získat další meta konfigurace a konfigurace. Je tedy možné přidávat konfigurace bez nutnosti měnit klientskou část aplikace.

**Příklad.** Na závěr uvedme část konfigurace popisující procházení taxonů živočichů a rostlin na Wikidatech. Tato konfigurace je dostupná pod IRI <https://linked.opendata.cz/resource/knowledge-graph-browser/configuration/wikidata/animals>. Prefix `rb` definujeme jako <https://linked.opendata.cz/resource/knowledge-graph-browser/>.

### Definice konfigurace

```
rb:configuration/wikidata/animals
  a browser:Configuration ;
  dct:title "Taxonomy of animals and plants"@en ,
           "Taxonomie rostlin a živočichů"@cs ;
  browser:hasVisualStyleSheet
    rb:wikidata/animals/style-sheet ;
  browser:startingNode <http://www.wikidata.org/entity/Q7377> ,
                     <http://www.wikidata.org/entity/Q192154> ;
  browser:resourceIriPattern
    "^http://www\\.wikidata\\.org/entity/Q[1-9][0-9]*$" ;
  browser:hasViewSet
    rb:view-set/wikidata/animals/taxon .
```

### Definice pohledu

```
rb:view/wikidata/animals/taxon/narrower a browser:View ;
  dct:title "Child taxons"@en ;
  browser:hasExpansion rb:expansion-query/animals/taxon/narrower ;
  browser:hasPreview rb:preview-query/animals/taxon/basic ;
  browser:hasDetail rb:detail-query/animals/taxon/basic .
```

### Definice datasetu

```
rb:dataset/wikidata a void:Dataset ;
  dct:title "Wikidata SPARQL endpoint" ;
  void:sparqlEndpoint <https://query.wikidata.org/sparql> ;
  browser:accept "application/sparql-results+json" .
```

## Definice stylu

```
rb:wikidata/animals/style/genus a browser:VisualStyle ;  
    browser:background-color "#ffbf00" ;  
    browser:hasSelector ".genus" .
```

## 2.2 Server

Dalším požadavkem bylo, aby klientská aplikace komunikovala se serverem, který provádí dotazy a vrací data pro aplikaci. Server bude detailněji popsán v příští kapitole.

Server zprostředkovává následující požadavky:

- `/meta-configuration` - Vrací informace o meta konfiguraci.
- `/configuration` - Vrací informace o konfiguraci.
- `/stylesheet` - Vrací visual style sheet.
- `/view-sets` - K vrcholu a konfiguraci vrátí seznam view setů.
- `/preview` - K vrcholu a pohledu vrátí preview.
- `/detail` - K vrcholu a pohledu vrátí detail.
- `/expand` - K vrcholu a pohledu vrátí expandované uzly.

## 2.3 Uživatelské a systémové požadavky

Níže je sepsán seznam původních požadavků na klientskou aplikaci.

### Konfigurace a stylesheet

Uživatel musí být schopen vybrat IRI konfigurace a IRI visual style sheetu a lze je kdykoli změnit. IRI konfigurace může být vybráno jen jedno.

**Analýza** Konfigurace popisuje, jak je graf zobrazen a které pohledy jsou na uzly aplikovatelné. Ačkoli by teoreticky bylo možné podporovat dvě konfigurace, nebudeme toto implementovat. Změna konfigurace tedy smaže aktuální graf a vytvoří graf nový.

### Vložení vrcholů

Uživatel může ručně zadat IRI vrcholu, který se následně zobrazí v grafu.

**Analýza** Pro úspěšné zobrazení vrcholu musí aplikace znát jeho **preview**. To lze ale získat pouze z konkrétního pohledu a je tedy nutné nejprve stáhnout **view-sets** daného vrcholu, z nich následně vybrat defaultní a zvolit výchozí pohled. Pak je možné zavolat metodu **preview** na serveru a získat data o vrcholu.

- IRI vrcholu bude považováno za chybné, pokud server vrátí prázdnou množinu na dotaz **view-sets**. V takovém případě totiž nejde aplikovat žádné pohledy na uzel, a tedy zřejmě nepatří do dané konfigurace.
- Vrchol může být již v grafu přítomen, pak se označí a přesune se na něj obrazovka. Pokud je vrchol skrytý, bude odkryt. Pokud jsou zapnuté filtry a vrchol je skrytý filtrem, vrchol se v grafu nezobrazí. Pokud se vrchol podaří načíst, nebo již existuje, bude vybrán a zobrazí se jeho detail v pravém panelu.

### Detail vrcholu

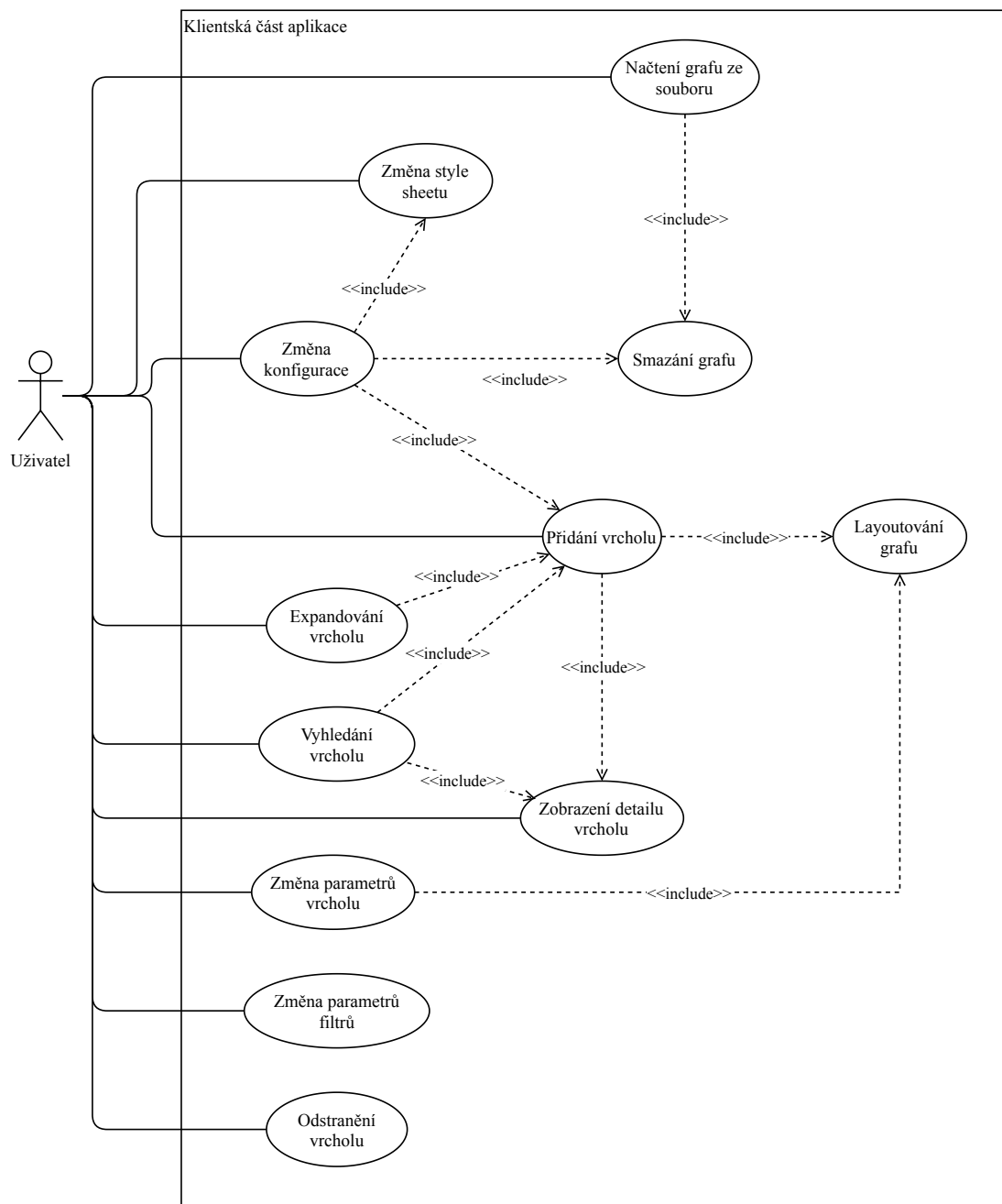
Pokud uživatel klikne na vrchol, zobrazí se panel s podrobnými informacemi o uzlu. Bude zobrazen detail uzlu voláním metody **detail** na serveru a budou zobrazeny veškeré pohledy vrcholu, možnost je přepínat a provádět expanze podle daného pohledu.

Detail bude zobrazen jako dvousloupcová tabulka klíč-hodnota.

Panel bude zobrazovat také další možné akce k vrcholu.

**Analýza** Vrchol nemusí mít načtené **view-sets**, **preview** ani **detail**, je tedy nutné po zobrazení panelu tyto informace stáhnout a během stahování zobrazit informaci, že se data stahují.

Může se stát, že **view-sets** vrátí prázdný výsledek. V takovém případě vrchol v grafu necháme i přes to, že podle původního požadavku bychom jej smazali. Taková situace nicméně implikuje chybně napsanou konfiguraci a uživatel bude o tomto informován chybovou hláškou.



Obrázek 2.2: Use case diagram dle uživatelských požadavků.

Mezi akcemi bude lokalizace vrcholu, smazání, opětovné načtení všech dat, zafixování pozice.

## Skrývání uzlů

Uživatel může skrývat uzly v panelu s detailem. Skrytý uzel se v grafu skryje společně se všemi hranami. Bude možné zobrazit seznam skrytých uzlů ze kterého půjde skryté uzly opět zviditelnit.

**Analýza** Bude přidáno tlačítko k detailu pro skrytí/zobrazení uzlu. Současně pro větší přehlednost bude do detailu přidána hláška informující uživatele, že je uzel skrytý.

Skryté uzly nebude možno lokalizovat, ale stále bude možné s nimi dále pracovat. Podle prvního požadavku se uzel opět zobrazí, pokud ho uživatel bude chtít explicitně vložit.

V panelu se skrytými uzly bude možnost uzly přímo zobrazit, nebo si zobrazit jejich detail. Detail skrytého uzlu funguje stejně jako pro viditelné uzly.

## Expanze

Po dvojkliku na uzel se uzel expanduje podle aktuálního pohledu. Uzel je také možné expandovat v panelu s detailem kliknutím na tlačítko expanze u příslušného pohledu. Expanzí se zavolá metoda `expand` na serveru a v grafu se zobrazí nové vrcholy.

**Analýza** Pokud přidávaný vrchol v grafu již je a je skrytý, pak skrytým zůstane. Na rozdíl od přidávání jednoho vrcholu totiž explicitně neříkáme, že chceme daný vrchol přidat. Protože je možné vrcholy mazat, povolíme uživateli provádět expanzi znova, i když již byla provedena.

Budeme si také pamatovat které vrcholy a hrany vznikly ze které expanze pro snazší práci s expanzemi.

## Filtrování vrcholů

Uživatel může přidat filtr, jež skryje nevyhovující vrcholy. Takovéto skryté vrcholy se pak chovají stejně jako uživatelem skryté vrcholy. Požadované filtry jsou

- Zobraz jen ty vrcholy, jež mají stupeň v daném intervalu nebo rozmezí.
- Zobraz vrcholy jen s konkrétním typem nebo třídou.

**Analýza** Aplikace by měla podporovat snadné přidání filtrů do budoucna a podporu pro pluginy, které dodávají vlastní možnosti filtrování. Aby bylo možné vyjádřit libovolné filtry, každý filtr by měl mít přístup k celému grafu a všem vrcholům.

Uzel může být skrytý jak filtrem, tak i uživatelem. Pro přehlednost by měl být uživatel informován, že nemůže ručně zobrazit vrchol který je skrytý filtrem. Tyto

vrcholy budou stále zobrazeny v seznamu skrytých vrcholů pro možnost přístupu k nim.

Každý filtr určí, zda je vrchol podle tohoto filtru viditelný. Vrchol pak bude viditelný, pokud všechny filtry určí, že viditelný je. Jedná se tedy o konjunkci.

Typy a třídy vrcholů nejsou známy předem a API serveru neumožňuje získat celou množinu typů a tříd. Je tedy nutné přizpůsobit filtry tak, aby si dokázaly poradit s novými uzly. Proto filtry na typ a třídu budou mít dva módy. V jednom módu explicitně skrývají zvolené vlastnosti a ve druhém explicitně zobrazují vrcholy s danými vlastnostmi. Toto chování pak ovlivní přidání nových, neznámých, vrcholů. V prvním případě nový vrchol bude zobrazen (pokud jeho typ, resp. třídy ještě nejsou známy) a ve druhém případě bude ihned skryt.

## Vícejazyčné uživatelské rozhraní

Uživatel může přepínat mezi více jazyky uživatelského rozhraní. Aktuálně bude podporována pouze angličtina a čeština.

**Analýza** Kromě uživatelského rozhraní by měl být vícejazyčný i graf. Aktuální API serveru toto ještě neumožňuje a problém je předmětem poslední kapitoly. Multijazyčnost zatím podporují metody `meta-configuration` a `configuration` na serveru. Protože obecně můžou existovat překlady do spousty světových jazyků, je vhodné stahovat pouze požadovaný jazyk a při jeho změně stáhnout data s novým jazykem znovu.

Pro velké grafy může být překlad problémem, protože změna jazyku může vyvolat spoustu požadavků na datové zdroje. Pro překlad grafu by tedy bylo nejlepší stáhnout překlad až když si uživatel zobrazí detail, nebo explicitně neoznačí vrcholy na překlad. Taktéž je vhodné oddělit výběr jazyka pro obsluhu aplikace a výběr konfigurací od překladu jednotlivých vrcholů. Příkladem situace, kde je oddělení žádoucí, může být procházení měst v Japonsku anglicky mluvícím uživatelem. V této situaci bychom požadovali, aby názvy měst byly v originálním jazyce.

Překlady do konkrétního jazyka by měly být v jednom souboru, nebo adresáři a přidávání nových jazyků by mělo být snadné, nejlépe bez zásahu do kódu aplikace.

## Stažení grafu do souboru

Uživatel má možnost stáhnout aktuální graf do souboru a později ho ze souboru načíst zpět.

**Analýza** Protože je aplikace ve vývoji, je třeba dbát na zpětnou kompatibilitu. Při každé nové verzi je tedy třeba ověřit, zdali soubor pochází ze staré verze a použít staré metody pro jeho zpracování a převedení do nového systému. Výsledný soubor může být zkomprimován pro menší velikost a nabízí se i možnost stáhnout jen základní informace tak, aby zbytek mohl být při načtení stažen ze serveru.

Pokud uživatel bude chtít zavřít aplikaci, bude dotázán, zda chce aktuální graf uložit do souboru. Uložený graf již nebude blokovat stránku o uzavření, dokud

uživatel nesmaže, nebo nepřidá nové uzly. Stažení detailu, nebo změna pohledu nebude považována za změnu hodnou k uložení.

Pokud uživatel zvolí načtení nového souboru, starý graf bude zahozen a uživatel tedy bude požádán o uložení.

Kromě otevření souboru ze systému bude možnost soubor načíst i z webu.

## Odstranění vrcholu

Uživatel může z grafu vrchol odstranit.

**Analýza** Odstraněním vrcholu se musí odstranit i všechny hrany patřící vrcholu. Vrchol by také měl být odstraněn ze všech expanzí tak, aby nedocházelo k únikům paměti.

Odstranit vrchol bude možné pomocí tlačítka v panelu s detailem, popřípadě skupinu vrcholů bude možné odstranit obdobným způsobem.

## Vyhledávání vrcholů s pomocí autocomplete

Pokud to konfigurace umožňuje, bude možné přidávat nové uzly do grafu s pomocí autocomplete.

**Analýza** Data pro vyhledávání se budou stahovat až když uživatel bude chtít poprvé vyhledávat.

Návrh vyhledávače by měl podporovat více vyhledávacích zdrojů a vhodně kombinovat nalezené výsledky v případě, že více zdrojů vrátí stejné vrcholy.

Kromě vyhledávání v JSON souboru se pak nabízí hledat i v aktuálním grafu a umožnit uživateli zadat do vyhledávacího pole přímo IRI uzlu, popřípadě její část, ze které se aplikace pokusí sestavit celou IRI.

## Podpora layoutů

Aplikace bude umožňovat několik způsobů layoutování grafu, které si bude moci uživatel volit.

- Pokud to layout umožňuje, bude možné ukotvit vrchol. To lze provést přesunutím vrcholu, pravým kliknutím myši, nebo z nabídky u detailu vrcholu. U ukotvených vrcholů bude zobrazena ikonka. Takovéto vrcholy pak nebudou layoutem ovlivněny.
- Layout reaguje na různé události, jako vytvoření skupiny, expanze vrcholu, přidání nového vrcholu do grafu, přesunutí vrcholu a na základě těchto událostí provádí layoutování grafu.
- Pokud to layout umožňuje, bude v pravém dolním rohu obrazovky tlačítko, které spustí layoutování explicitně.
- Layout má vlastní nastavení.



## Seskupování vrcholů

Vrcholy bude možné seskupovat do skupin, které budou ve vizuálním grafu reprezentovány jedním vrcholem. Pokud expanze vrátí větší množství nových vrcholů, vzniknou jako skupina. Skupinu bude možné rozbít dvojklikem.

- Z grafového hlediska skupina vznikne kontrakcí hran mezi vrcholy skupiny. To znamená, že pokud vedla hrana mezi vrcholem mimo skupinu a vrcholem ve skupině, povede tato hrana mezi vrcholem mimo skupinu a skupinou. Všechny násobné hrany stejného typu budou nahrazeny jednou hranou. Obdobně toto platí pro hrany mezi dvěma skupinami. Pokud je ve skupině vrchol skrytý (uživatel, nebo filtrem), jeho hrany se nepodílejí na utváření skupiny. Pokud jsou všechny vrcholy skupiny skryty, je skrytá i skupina.
- Skupina se na grafu chová jako obyčejné vrcholy, je ji možné skrýt, přesouvat, ukotvit atp.
- Označením několika vrcholů se nabídne možnost vytvořit skupinu. Tato skupina pak vznikne na místě, kde byly původní vrcholy.
- Rozbít skupinu je možné dvojklikem, nebo tlačítkem z detailu skupiny. Vrchol skupiny bude odstraněn a vzniknou místo něj původní vrcholy, které budou layoutovány.

## Kompaktní mód

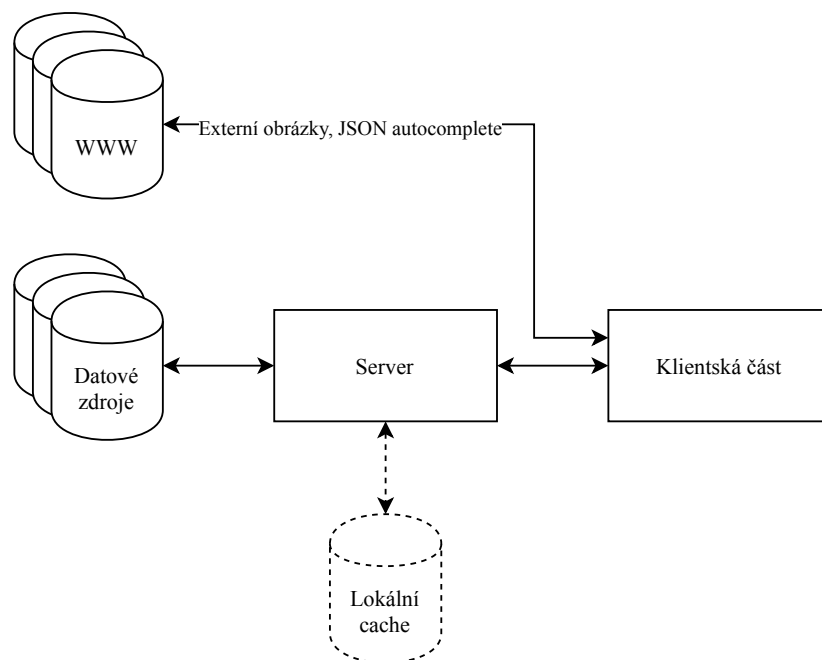
Uživatel bude mít možnost zapnout kompaktní mód. Během něj se ve středu obrazovky zobrazí pouze zvolené vrcholy a jejich přímí sousedi. Tyto vrcholy budou layoutovány bez ohledu na ostatní vrcholy, které budou skryté. Uživatel pak bude mít možnost graf v tomto módu procházet klikáním na sousedy aktivního vrcholu.

**Analýza** Kompaktní mód se vypne, pokud nebude vybrán žádný vrchol. Při kompaktním módu se bude obrazovka neustále zaměřovat na vrcholy účastníci se kompaktního módu. Jakmile vrchol kompaktní mód opustí, měl by se vrátit na svou původní pozici. Po skončení kompaktního módu se původní rozložení grafu nezmění. Nově vytvořené vrcholy v rámci kompaktního módu musí být i správně umístěny, jakmile bude mód ukončen.

## 3. Návrh architektury

Aplikace je rozdělena na serverovou a klientskou část. Server slouží pro odstínění klienta od RDF databází, komunikuje tedy přímo s datovými zdroji a zpracovaná data předává klientské aplikaci. Klient pak v případě potřeby z internetu stahuje externí obrázky (příkladem jsou konfigurace Wikidat, které obsahují URL odkazy) a autocomplete soubory.

Rozdělení aplikace na klient-server také umožňuje cachování na straně serveru, které ještě implementováno není. Tento problém je diskutován v poslední kapitole.



Obrázek 3.1: Komunikace mezi klientem, serverem a datovými zdroji. Cachování na serveru ještě implementováno není.

### 3.1 Server

Jak již bylo v dokumentu zmíněno, převážná část serveru byla dodána jako specifikace pro klientskou aplikaci.

#### 3.1.1 Jazyková podpora

Server jsem částečně přepsal, aby podporoval dotazování na data z více světových jazyků. Některé požadavky přijímají parametr `languages` obsahující čárkou (,) oddělené ISO 639-1 jazykové kódy. Server pak vrací objekty, jejichž klíčem je jazykový kód a hodnotou daný překlad do jazyka. Pokud překlad neexistuje, hodnotou je `null`. V případě, že na všechny jazyky bylo vráceno `null`, server se pokusí přidat další jazyk, který existuje. Který jazyk takto bude vybrán, není určeno.

Tato implementace umožňuje stahování vícejazyčných dat tak, že jsou staženy jen žádané jazyky, což může značně ušetřit přenos dat v některých případech, ale

současně dojde ke stažení alespoň jednoho podporovaného jazyka, pokud je to možné.

**Příklad.** Pro `languages=cs,en` může server vrátit například

```
{
  cs: null,
  en: "Kankakee County"
}
```

ale pokud nezná překlad ani do češtiny, ani do angličtiny, může vrátit

```
{
  cs: null,
  en: null,
  sk: "Jazero Beňatina"
}
```

### 3.1.2 API

Server vrací data ve formátu JSON, požadavky jsou posílány metodou GET a parametry jsou kódovány do URL adresy.

#### **/meta-configuration**

**parametry:** `iri` a `languages`

Vrátí informace o meta konfiguraci zadané podle `iri`.

Vrátí všechna data (viz kapitola 2.1.1) o meta konfiguraci, veškerá data o dceřiných konfiguracích (viz kapitola 2.1.2) a základní data o dceřiných meta konfiguracích (vše kromě seznamu konfigurací a meta konfigurací).

```
interface ResponseMetaConfiguration extends
ResponseMetaConfigurationBase {
    has_meta_configurations: ResponseMetaConfigurationBase[],
    has_configurations: ResponseConfiguration[],
}

interface ResponseMetaConfigurationBase {
    iri: string,
    title: {[language: string]: string},
    description: {[language: string]: string},
    image: string,
}
```

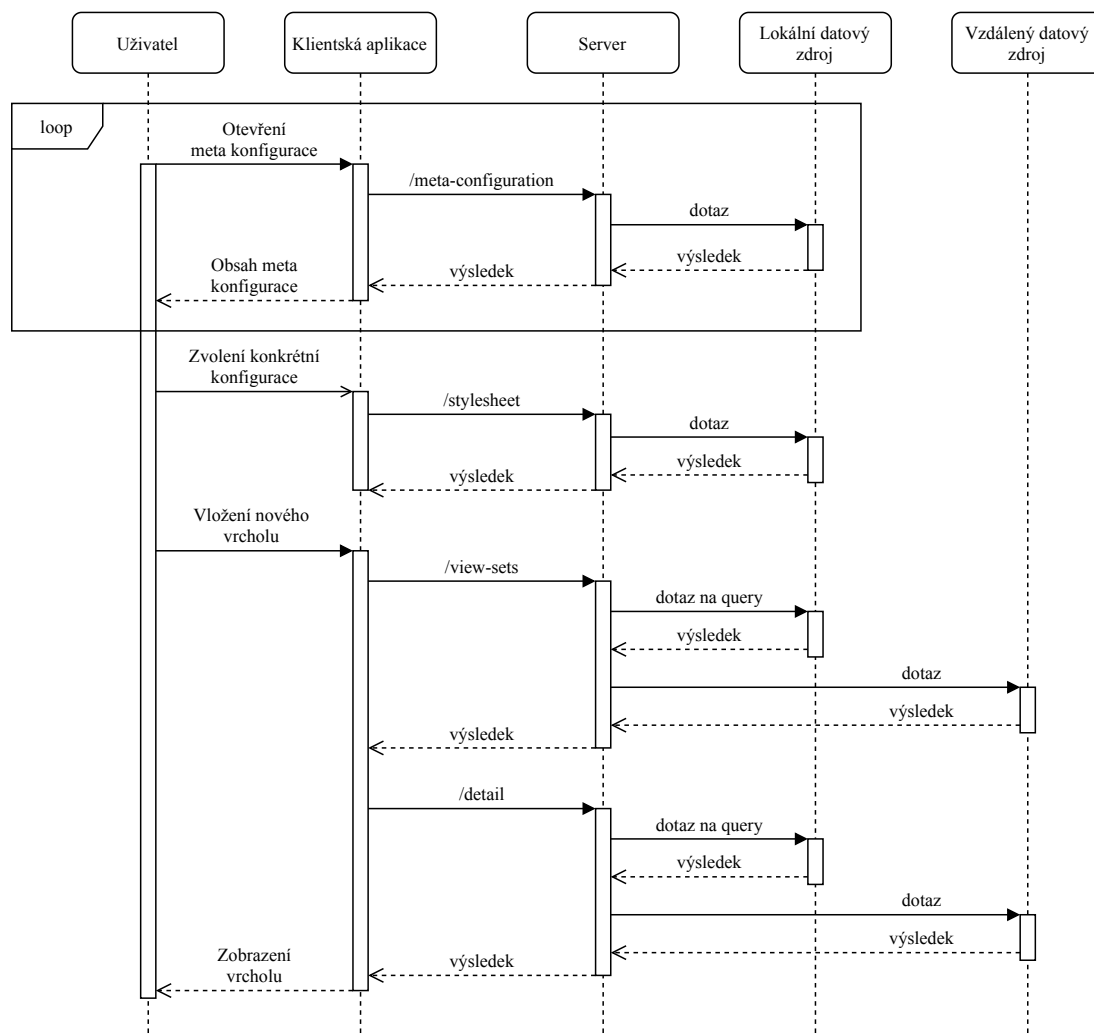
#### **/configuration**

**parametry:** `iri` a `languages`

Vrátí informace o konfiguraci zadané podle `iri`.

Vrátí stejná data jako `/meta-configuration` o svých dceřiných konfiguracích.

Toto volání se používá pouze když uživatel ručně zvolí IRI konfigurace, v opačném případě si aplikace vystačí s voláním `/meta-configuration`.



Obrázek 3.2: Příklad komunikace mezi klientem, serverem a RDF databází při spuštění aplikace. Na začátku klient nejprve vybírá konfiguraci. (Přičemž je dostačující volat pouze **meta-configuration**, protože výsledek dotazu obsahuje i konfiguraci.) Poté začne stahování style sheetu a prvního vrcholu.

Při stahování vrcholu se nejprve stáhnou **view-sets** a poté z výchozího pohledu **detail** a vrchol se zobrazí na grafu. Případné **expand** a **preview** vypadají obdobně.

```
interface ResponseConfiguration {
    iri: string,
    stylesheet: string[],
    title: {[language: string]: string},
    description: {[language: string]: string},
    autocomplete: string[],
    starting_node: string[],
    resource_pattern: string|null,
}
```

## /stylesheet

**parametry:** stylesheet

Vrátí kompletní visual style sheet (viz kapitola 2.1.9) na základě jeho IRI jako parametr stylesheet.

```
interface ResponseStylesheet {
    styles: {
        selector: string;
        properties: {
            [property: string]: string;
        }
    }[];
}
```

## /view-sets

**parametry:** config a resource

Vrátí seznam možných view setů, (viz kapitola 2.1.3) které odpovídají uzlu s IRI resource při dané konfiguraci config.

```
interface ResponseViewSets {
    viewSets: {
        iri: string;
        label: string;
        defaultView: string;
        views: string[];
    }[];
    views: {
        iri: string;
        label: string;
    }[];
}
```

## /preview

**parametry:** view a resource

Vrátí data z dotazu preview (viz kapitola 4.4.10) na uzel s IRI resource při daném pohledu view.

```
interface ResponsePreview {
    nodes: ResponseElementNode[];
    types: ResponseElementType[];
}
```

### **/detail**

**parametry:** view a resource

Vrátí data z dotazu detail (viz kapitola 4.4.10) na uzel s IRI **resource** při daném pohledu **view**.

```
interface ResponseDetail {
    nodes: {
        iri: string;
        data: {
            [IRI: string]: string;
        };
    }[];
    types: ResponseElementType[];
}
```

### **/expand**

**parametry:** view a resource

Vrátí expandované uzly (viz kapitola 2.1.5) pro uzel s IRI **resource** při daném pohledu **view**. Tyto expandované uzly již obsahují data o detailu a tedy není třeba žádného dalšího volání.

```
interface ResponseExpand {
    nodes: ResponseElementNode[];
    edges: ResponseElementEdge[];
    types: ResponseElementType[];
}
```

Mezi pomocná rozhraní pak patří

```
interface ResponseElementType {
    iri: string;
    label: string;
    description: string;
}

interface ResponseElementEdge {
    source: string;
    target: string;
    type: string;
    classes: string[];
}

interface ResponseElementNode {
    iri: string;
    type: string;
    label: string;
    classes: string[];
}
```

## 3.2 Klientská část aplikace

### 3.2.1 Moduly

Následující text popisuje rozvržení klientské části do modulů a jejich vzájemnou integraci.

#### Připojení na server

Modul řeší komunikaci se serverem. Obsahuje metody odpovídající API serveru, které vracejí data ve správném interface. V případě chyby vrátí jako odpověď false. Vyžaduje URL adresu serveru, se kterým má komunikovat.

#### Graf

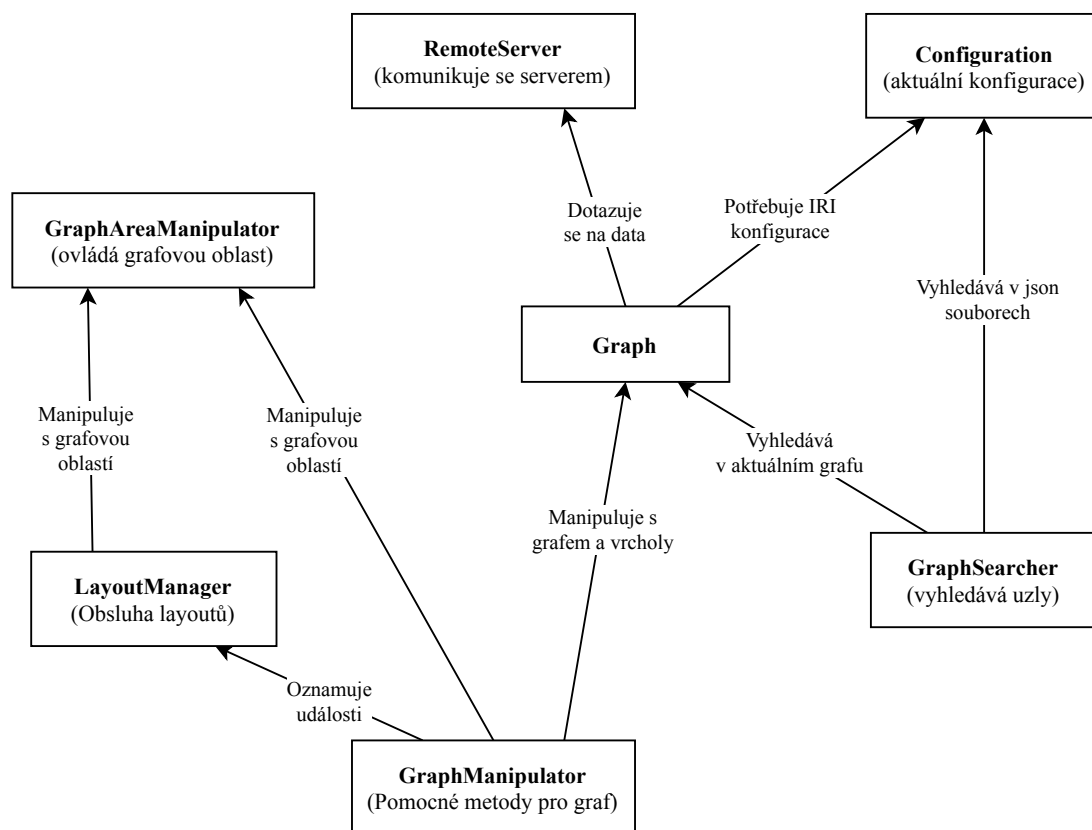
Tato část aplikace reprezentuje stažený graf a všechny jeho data. Má metody pro práci s grafem, jeho modifikaci a stahování nových dat ze serveru.

Využívá modul pro připojení na server.

#### Práce s grafovou oblastí

Tento modul odpovídá za práci s grafovou oblastí, tedy s plátnem, na které se vykresluje graf. Řeší manipulaci s grafovou oblastí, správné umístění nových vrcholů v rámci grafu a podobně.

Využívá modulu Graf pro práci s grafem.



Obrázek 3.3: Závislost modulů a dalších částí aplikace na sobě. **TODO:** Mohl bych se zeptat? Počítá se toto jako popis vazeb mezi moduly? Tento obrázek popisuje skutečné závislosti a používá skutečná jména tříd z aplikace. Můžu ho tedy v této části použít (i když ne vždy odpovídá textu v této kapitole), nebo ho mám dát k implementaci a tady dát něco jiného?

## Filtrování

Modul integruje možnost filtrování vrcholů v aplikaci na základě různých grafových a sémantických vlastností.

Obsahuje množinu filtrů, která může být za běhu aplikace měněna a tedy umožňuje aplikaci rozšířit o nové filtry pomocí pluginů. Každý filtr pak

- Zodpovídá za vykreslení uživatelského rozhraní pro nastavení parametrů filtru.
- Má vlastní parametry
- Obsahuje třídu, která bude instanciována pro každý vrchol a na základě parametrů filtru a dat vrcholu rozhodne, zda je vrchol viditelný.

Filtrování tedy potřebuje přístup ke grafu.

## Layoutování

Modul řeší uspořádání vrcholů v grafu, přičemž může reagovat na významné události grafu, jako přidání nového vrcholu, expanzi atp.



Obdobně jako filtrování, modul layoutování obsahuje množinu layoutů, která může být za běhu aplikace měněna. Každý layout pak

- Zodpovídá za vykreslení uživatelského rozhraní pro nastavení parametrů layoutu.
- Má vlastní parametry
- Provádí layoutování, pokud je daný layout aktivní.
- Může vykreslit dodatečná tlačítka do uživatelského rozhraní pro snazší obsluhu layoutování.

Vždy je aktivní právě jeden layout. Pokud dojde ke změně layoutu, je o tom starý, i nový layout informován. Aktivní layout pak přijímá události z aplikace, na které může reagovat. Využívá právě modulu práce s grafovou oblastí.

## Vyhledávání

Modul vyhledávání vrací IRI vrcholy konfigurace na základě textového řetězce. Vyhledává v grafu, z autocomplete nebo se pokusí IRI sestavit z hledaného výrazu.

Modul má několik vyhledávačů a každý se pokusí na základě hledaného výrazu vrátit nalezené vrcholy. Ty jsou pak vhodně sloučeny a seřazeny do jednoho seznamu. Modul vyhledávání sám o sobě nezávisí na žádné jiné části aplikace, nicméně jednotlivé vyhledávače ano.

Některé vyhledávače můžou mít větší šanci nalézt vrchol, ale menší dodat k němu podrobné informace. Příkladem může být vyhledávač, který sestavuje IRI vrcholu z hledaného výrazu. V takovém případě je vyhledávač schopen poskytnout pouze IRI, nicméně jiný vyhledávač může vrchol znát, ale nemusel ho být schopen podle hledaného výrazu najít.

Proto vyhledávání probíhá ve dvou fázích. V první fázi se na základě hledaného řetězce sestaví množina IRI adres vrcholů. Ty jsou následně předány ve druhé fázi všem vyhledávačům, které se pak snaží doplnit informaci k těmto vrcholům, například jejich popisem. Protože vyhledávače mohou využívat internet, je třeba výsledky vhodně přepočítat kdykoli nějaký z vyhledávačů vrátí výsledek.

Postup vyhledávání je následující

1. Dotaž se všech vyhledávačů současně na IRI, která odpovídají hledanému výrazu.
2. Když vyhledávač odpoví, přidej výsledná IRI k již nalezeným (od jiných vyhledávačů) v rámci daného hledaného výrazu.
3. Dotaž se všech ostatních vyhledávačů současně na ty IRI, která byla do seznamu přidána poprvé.
4. Když vyhledávač odpoví, přidej informace o daných vrcholech a aktualizuj vyhledávané výsledky.

## Ukládání do souboru

Řeší ukládání stavu aplikace do souboru a jeho následnou obnovu. Každý modul aplikace, konkrétně třídy, co nesou data, mají metody na serializaci a deserializaci jejich vnitřního stavu. Jednotlivé stavy jsou pak získány ze všech modulů a uloženy do souboru. Obdobně opačným způsobem se soubor načte a jednotlivé objekty popisující stavy jsou zpět rozdistribuovány po aplikaci na obnovení stavu.

## Uživatelské rozhraní

TODO: Můžu to považovat za modul? Je to vlastně určitá část aplikace co něco dělá. Poskytuje uživateli informace o vrcholech a umožňuje mu interagovat s grafem.

## 4. Implementace

### 4.1 Vue.js framework

Klientská část aplikace je postavena nad Vue.js frameworkem<sup>1</sup>, jež je populární JavaScriptový framework na stavbu uživatelských rozhraní. Protože některé z jeho funkcionalit byly použity v klíčových částech aplikace, je nutné čtenáře obeznámit alespoň se základním principem fungování frameworku.

#### 4.1.1 Vuex

Vue.js framework, podobně jako konkurenční React<sup>2</sup> (Facebook) nebo Angular<sup>3</sup> (Google), využívají principu sledování stavu aplikace (jejich dat) pro automatickou změnu DOMu webové stránky. V praxi to znamená, že programátor může velice snadno napsat kód, který generuje uživatelské rozhraní na základě dat, která mohou být libovolně měněna bez nutnosti řešit problém, zda ke změně vůbec došlo a které části aplikace mají být o změně stavu informovány. Ve Vue tuto funkcionalitu zastává právě Vuex<sup>4</sup>, jež je možný používat samostatně.

Vuex drží stav aplikace jako jeden objekt (tedy slouží jako centrální úložiště dat pro celou aplikaci). Tento objekt se nazývá **store**. Změny ve storu mohou být sledovány Vuexem pro vykonání libovolných akcí, například překreslení textu na stránce, jež byl vykreslen Vue frameworkem.

Vrátíme-li se k původnímu příkladu, programátorovi stačí přiřadit do proměnné, jež je spravovaná Vuexem, novou hodnotu a Vuex se postará o zavolání všech komponent, které tuto proměnnou využívají a tyto komponenty na stránce překreslí původní hodnotu na novou. Překreslení přitom proběhne až poté, co skončí průběh aktuální funkce. Tohoto je docíleno pomocí

`Window.requestAnimationFrame()`. Díky tomuto můžeme stav v rámci průběhu jedné funkce modifikovat vícekrát se skoro nulovým dopadem na celkový výkon aplikace.

#### Computed properties

Kromě této funkcionality Vuex nabízí takzvané **getter**y, jež jsou ve Vue frameworku nazývány jako **computed properties**. Jedná se o funkce, které využívají data ze storu pro výpočet dat nových. Výhoda takovýchto getterů je ta, že Vuex dokáže výsledky těchto funkcí cachovat a přepočítává je pouze tehdy, změní-li se data původní. Interně gettery fungují tak, že při zavolání klientské funkce Vuex sleduje, které části storu byly dotázány a ty pak sleduje na změnu jež invaliduje cache konkrétního getteru. Při příštím požadavku na hodnotu se pak klientská funkce volá znovu a celá operace se opakuje.

Tyto computed properties jsou v aplikaci využívány často. Kupříkladu funkce, která počítá, zda je sousední uzel vybrán. Na takovou hodnotu se v aplikaci

---

<sup>1</sup><https://vuejs.org/>, You (2020)

<sup>2</sup><https://reactjs.org/>

<sup>3</sup><https://angularjs.org/>

<sup>4</sup><https://vuex.vuejs.org/>

mohu ptát libovolně krát, ale počítá se pouze tehdy, když se množina sousedních uzlů vrcholu změní, nebo se změní právě označení uzlu z množiny.

## Watchers

Ve Vue lze využívat i **watchery**, které umožňují registraci callbacku na změnu určité proměnné ve stavu. Watchery má smysl využívat tam, kde již data přestávají být spravována Vue frameworkem, tedy u knihoven třetích stran. Watchery, podobně jako překreslení komponent, jsou volány až po skončení probíhající funkce.

## Změna stavu

Jak již bylo zmíněno, stav lze měnit přiřazením do proměnné, popřípadě voláním metod jako `.push()` na poli. Vuex dokáže tyto změny sledovat nahrazením původní proměnné (máme na mysli položku objektu) JavaScriptovým setterem. U polí dojde k obalení metody `.push()`, jež registruje změnu stavu.

Tento přístup má několik nevýhod, které se promítly i při vývoji klientské aplikace:

- Vue není plně kompatibilní s novými **ES6 kontejnery** `Map` a `Set` a proto jsou v aplikaci používány jen v rámci lokálních proměnných a mapa je nahrazena klasickým objektem.
- Protože je v JavaScriptu nemožné sledovat **vytvoření nové property objektu**, musí programátor v tomto případě volat ručně `Vue.set(target, propertyName/index, value)` a `Vue.delete`, popřípadě vytvořit nový objekt, který nahradí ten původní.
- Je důležité mít na paměti, že data ve storu již nejsou původními objekty v pravém slova smyslu, ale veškeré fieldy a metody u polí byly nahrazeny, jak je zmíněno výše. Proto předání objektů a polí ze storu knihovnám třetích stran je nutné ošetřit **klonováním objektu**, jinak může dojít k zaseknutí aplikace.
- Instance tříd **knihoven třetích stran může zaseknout aplikaci**, pokud ji uložíme do storu. Bohužel, tohoto je velmi snadné ve Vue frameworku dosáhnout omylem. Problém je rozebrán v následující kapitole.

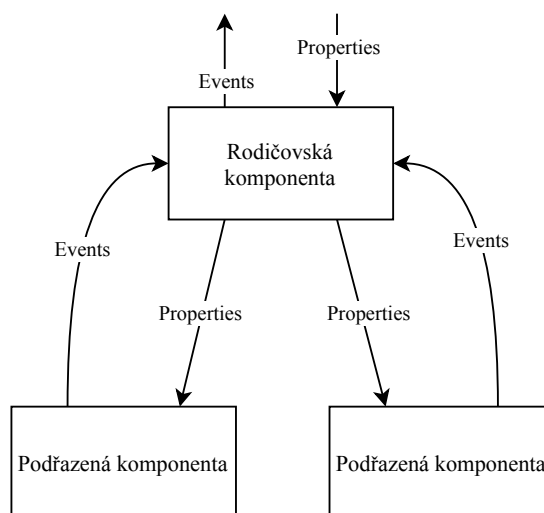
### 4.1.2 Vue framework

Vue framework využívá takzvané komponenty. Komponentou se rozumí prvek na stránce, se kterým má smysl pracovat samostatně. Každá komponenta má vlastní HTML, CSS a JS. Komponenty se mohou do sebe zanořovat a vytvářet tak větší komponenty z menších. Příkladem může být komponenta *seznam* jež dokáže na stránku vykreslit seznam prvků. Takováto komponenta by pak mohla mít podkomponenty jako *prvek seznamu*.

Každé komponentě lze předat data formou **properties** přičemž komponenta na základě těchto dat může vyrobit pod sebou další komponenty a předat jim část dat, která dostala.

Někdy může být žádoucí, aby vícero komponent sdílelo stejnou množinu funkcí. V klasickém programování toto odpovídá dědičnosti tříd. Vue má takzvané Mixiny. Mixin je objekt, který komponentě dodává logiku navíc. Mixiny se dají použít jak v případě dědičnosti, kdy komponenty řeší podobný problém, tak i v případě, kdy kód jedné komponenty je příliš dlouhý a vyplatí se ho rozdělit na více částí. Mixiny totiž umožňují i vícenásobnou dědičnost.

Tyto předávána data jsou právě data ze storu. Vue framework doporučuje, aby data, co komponenta dostane formou properties, neupravovala přímo, ale místo toho posílala události rodičovské komponentě, která data upraví. Ve své práci jsem se **rozhodl toto doporučení ignorovat**, neboť by tímto vzrostla náročnost na správu aplikace.



Obrázek 4.1: Doporučený způsob komunikace mezi komponentami ve Vue frameworku

### 4.1.3 Loaders

Kód Vue komponent se zapisuje do souborů s příponou `.vue`. Při sestavování aplikace se pak použije `vue-loader`, který ze souboru vyextrahuje zvlášť CSS, JS a HTML a ty předá dál na zpracování. HTML kód komponent není ve skutečnosti pravý HTML. Jedná se nadstavbu umožňující psát speciální značky, jež rozhodují kolikrát a jestli vůbec se tag na stránce vyrenderuje. Tato HTML nadstavba je pak předána `vue-template-compiler`, který vyrobí optimalizovaný JS kód jež renderuje HTML na základě stavu komponenty.

**Příklad.** Ukázka jednoduché Vue komponenty `IndexedList`, která má parametr `list` očekávající pole stringů. Tato komponenta vypíše pole v odrážkovém seznamu ve formě `index: hodnota`. Komponenta se sama stará o překreslení DOMu, když se změní data. Komponentu můžeme v jiné komponentě použít vložením `<indexed-list :list="inputData"/>`, kde `inputData` je proměnná obsahující pole stringů.

```
<template>
  <ul>
    <li v-for="(item, index) in list" :key="index">
      {{index}}: {{ item }}
    </li>
  </ul>
</template>

<script lang="ts">
  import {Component, Prop} from "vue-property-decorator";
  import Vue from "vue";
  @Component
  export default class IndexedList extends Vue {
    @Prop() private list: string[];
  }
</script>

<style scoped lang="scss">
  ul {
    color: red;
  }
</style>
```

## Scoped styly

Můžeme si povšimnout `scoped` stylů v ukázce. Vue má mechanismus, že styly, které zde nastavíme se aplikují jen na tuto komponentu. Nastavením červené barvy na seznam jsme tedy skutečně nastavili červenou barvu jen této komponentě a ostatní seznamy jsou netknuté. Toto má nespornou výhodu pokud pracujeme s velkým množstvím komponent a hrozilo by, že bychom museli používat složité pojmenované css třídy aby nedošlo ke kolizi.

Pokud bychom chtěli ovlivnit styly vnořených komponent a máme nastavené `scoped` styly, musíme použít pseudo selektor `::v-deep`, kupříkladu `.actions ::v-deep .v-input-selection-controls`. Tohoto je hodně používáno, pokud je potřeba upravit styly Vuetify frameworku (viz dále).

## 4.2 Cytoscape knihovna

Cytoscape.js<sup>5</sup> je JavaScriptová knihovna na kreslení grafů s pomocí technologie canvas<sup>6</sup>, která byla využita v tomto projektu.

Cytoscape umožňuje definovat graf pomocí vrcholů a hran, který bude vykreslen na obrazovku, kde může uživatel přesouvat jednotlivé jeho části, měnit přiblížení grafu, označovat vrcholy a podobně. Kromě toho podporuje širokou škálu možností, jak vizuálně obarvit vrcholy. Tato pravidla jsou dodávána jako style sheet a odpovídají těm, které byly zmíněné v kapitole 2.1.9.

V knihovně lze zaregistrovat callbacky na různé události grafu a tímto jsme schopni na ně v aplikaci reagovat a propojit knihovnu i s Vue frameworkem. Umožňuje také měnit stav grafu průběžně a vytvářet tak animace, kupříkladu přiblížení grafu pak bude plynulé.

Cytoscape podporuje layoutování grafu s pomocí pluginů. V této práci byl využit Cola layout<sup>7</sup>, který využívá fyzikální simulaci na esteticky příjemné uspořádání vrcholů. Původní plugin byl v mírně upraven, aby vyhovoval požadavkům na uzamykání pozic vrcholů. Kromě Cola je v aplikaci využit Dagre layout, který vrcholy uspořádá do stromové struktury.

**Třídy** Cytoscape podporuje přidávání tříd vrcholům obdobně, jako třídy v HTML. Na tyto třídy je pak možné nastavit pravidla style sheetů. Toho je v aplikaci právě využito tak, že náhled (4.4.10) vrací seznam tříd, které budou nastaveny vrcholům a podle kterých budou barevně a jinak vizuálně rozlišeny.

**Příklad.** Ukázka kódu, který vytváří nový vrchol. Jak lze vidět, vytvářenému vrcholu nastavujeme ihned pozici a třídy. Kromě tříd může nést další data jako popisek (label), který se pak zobrazí jako text u vrcholu. Vrcholu jsou nastavena všechna data z `preview`, na které lze reagovat a používat je v rámci style sheetů.

```
this.element = this.cy.add({
  group: 'nodes',
  data: {
    label: '-',
    ...clone(this.node.lastPreview),
    id: this.node.IRI
  },
  classes: this.classList,
  position,
});
```

*Za povšimnutí stojí i funkce clone, která je nutná, neboť předáváme objekt z Vue frameworku.*

<sup>5</sup><https://js.cytoscape.org/>, Franz a kol. (2015)

<sup>6</sup>Canvas umožňuje kreslit bitmapové obrázky na stránku. Alternativou této technologie je SVG, která pracuje s vektorovými objekty. Nicméně pro větší grafy může limitovat výkon aplikace.

<sup>7</sup><https://github.com/cytoscape/cytoscape.js-cola>

## 4.3 Prostředí

Klientská část aplikace je napsána v jazyce TypeScript<sup>8</sup>, což je typovaná nadstavba JavaScriptu. Projekt využívá balíčkovací nástroj npm<sup>9</sup> a je sestavován pomocí @vue/cli-service. Ten interně využívá Webpack<sup>10</sup> pro sestavení celého projektu.

### 4.3.1 Instalace

Pro úspěšné přeložení aplikace je nutné vytvořit soubor `./conf.yaml`, který přepisuje nastavení ze souboru `./conf.default.yaml`. V tomto souboru lze nastavit výchozí jazyk aplikace, výchozí IRI meta konfigurace a URL adresu, kde se nachází server.

Pro přeložení projektu je pak nutné spustit

1. `npm install` pro stažení a případnou aktualizaci balíčků, na kterých aplikace závisí.
2. `npm run build` pro přeložení projektu.

Výsledné soubory jsou umístěny v adresáři `./dist` včetně souboru `index.html`, odkud se aplikace spouští. Webový server by tedy měl odkazovat právě do tohoto adresáře.

### Jazykový překlad aplikace

Aplikaci lze přeložit do více světových jazyků přidáním souboru `<jazykový-kód>.yaml` do adresáře `./locales`. Soubor vytvořte jako kopii souboru `en.yaml`, jež obsahuje anglické překlady aplikace. Aplikaci pak stačí znova přeložit a vytvořený jazyk bude v aplikaci dostupný.

---

<sup>8</sup><https://www.typescriptlang.org/>

<sup>9</sup><https://www.npmjs.com/>

<sup>10</sup><https://webpack.js.org/>



### 4.3.2 Adresářová struktura

```
/
├── dist .... Adresář s vyrobenými .js a .css soubory. Obsahuje také výchozí
│   │   │   index.html soubor.
├── locales ..... Adresář s .yaml překlady aplikace.
├── public ..... Adresář se statickými soubory.
│   └── index.html ..... Šablona pro soubor index.html.
├── src
│   ├── @types ..... Definice typů pro knihovny, jež nemají oficiální typy pro
│   │   │   Typescript.
│   ├── component ..... Obsahuje Vue komponenty.
│   │   └── Application.vue ..... Top level Vue komponenta. Zde se vše
│   │       registruje.
│   ├── configurations ..... Obsahuje logiku pro správu konfigurací a meta
│   │   │   konfigurací.
│   ├── file-save ..... Interface pro třídy které obsahují uložitelná data.
│   ├── filter ..... Logika filtrování a jednotlivé filtry na skrývání uzlů.
│   │   └── filters ..... Obsahuje adresáře s jednotlivými filtry.
│   ├── graph ..... Třídy reprezentující lokální graf.
│   ├── layout ..... Logika layoutů.
│   │   └── layouts ..... Obsahuje adresáře s jednotlivými layouty.
│   ├── remote-server ..... Třídy pro stahování dat ze vzdáleného serveru.
│   ├── searcher ..... Logika vyhledávání.
│   │   └── searchers ..... Obsahuje jednotlivé vyhledávače.
│   ├── LiteralTranslator.ts ..... Pomocná metody pro překlad.
│   ├── conf.ts ..... Pomocný export soubor pro konfiguraci.
│   ├── i18n.ts ..... Inicializuje překlad aplikace.
│   └── main.ts ..... Počáteční soubor aplikace.
├── conf.default.yaml ..... Výchozí nastavení aplikace.
├── tsconfig.json ..... Konfigurační soubor pro Typescript.
└── vue.config.js ..... Konfigurační soubor pro balíčky aplikace.
```

## 4.4 Programátorská dokumentace

V této sekci budou postupně rozebrány klíčové třídy a komponenty aplikace.

### 4.4.1 Vstupní skript a pomocné soubory

Vstupním skriptem celé aplikace je soubor `main.ts`, ve kterém je inicializován Vuetify a Vue framework, který spouští komponentu `Application`, jež je výchozí komponentou celé aplikace.

V tomto souboru je také includován soubor `LiteralTranslator.ts`, který přidává pomocné globální metody (v rámci Vue frameworku jsou dostupné pod `this`) pro překlad literálů z grafu.

- `$t_literal(translations): string|undefined` - Očekává objekt popsáný v kapitole 3.1.1 a vybere z něj nejvhodnější překlad podle pravidel popsáných ve zmíněné kapitole. Vrací `null`, pokud žádný překlad není.
- `$te_literal(translations): boolean` - Vrací `true`, pokud by předchozí metoda našla překlad.
- `$i18nGetAllLanguages(): string[]` - Vrací jazyky, které jsou požadovány ze serveru. Pokud se jazyk aplikace změní a na již stažená data vrátí `$te_literal false`, pak se data stáhnou znovu s již správným jazykem. Tato logika je zatím implementována pouze u meta konfigurací a konfigurací.

### 4.4.2 Komponenta Application

Jedná se kořenovou komponentu celé aplikace, která drží základní třídy a moduly a řídí logiku celé aplikace. Tato komponenta registruje veškeré dialogové okna a grafické prvky.

Pro debugování aplikace je komponenta přístupná pod `window.kgvb` a je tedy možné zasahovat do jakékoli části aplikace.

Mezi důležité fieldy patří (veškeré tyto třídy budou popsány dále v textu):

- `server: RemoteServer` - Třída komunikující se serverem. Obsahuje metody, které zavolají na serveru konkrétní požadavek a vrátí výsledek ve správném interface nebo `false`, pokud nastala chyba.
- `configuration: Configuration` - Aktuální konfigurace grafu ve smyslu konfigurace z kapitoly 2.1.2.
- `graph: Graph` - Aktuální graf jež závisí na `configuration`.
- `areaManipulator: GraphAreaManipulator` - Třída, která obsahuje metody pro práci s grafovou oblastí jako přibližování, přesouvání pohledu atp.
- `manipulator: GraphManipulator` - Třída, která obsahuje metody pro složitější práci s grafem. Na rozdíl od `Graph`, metody v této třídě více odpovídají akcím uživatele a obvykle volají další moduly jako `LayoutManager`.

- **viewOptions: ViewOptions** - Jednoduchá třída mající stav, jak pohlížet na graf. Řeší, zda u hran mají být popisky a zda vůbec mají být hrany viditelné. U vrcholů pak řeší také viditelnost popisků a zda se mají zobrazit jako malé tečky.
- **filter: FiltersList** - Modul řešící filtrování.
- **layouts: LayoutManager** - Modul řešící layoutování grafu v závislosti na určitých akcích uživatele.
- **configurationManager: ConfigurationManager** - Drží všechny načtené konfigurace a meta konfigurace ze serveru.
- **visualStyleSheet: ResponseStylesheet** - Aktuální style sheet pro Cytoscape knihovnu.
- **graphSearcher: GraphSearcher** - Třída schopná vyhledávat vrcholy v grafu a z autocomplete souborů jež jsou specifikovány v konfiguraci.

## Závislost modulů

Značná část modulů v aplikaci je závislá na jiných. Kupříkladu **Graph** je závislý na **RemoteServer** a částečně na **Configuration**. Na třídě **Graph** pak závisí **GraphAreaManipulator** na které závisí **GraphManipulator**. Protože těchto závislostí je hodně, některé třídy byly určeny jako readonly a tedy se může měnit pouze jejich stav. Jedná se například o třídu **RemoteServer** které lze měnit URL adresu serveru. Dalšími třídami jsou **GraphAreaManipulator**, **ViewOptions**, **FiltersList**, **LayoutManager**, **ConfigurationManager**. Ostatní třídy jsou pak měněny pouze když se mění konfigurace, v tu chvíli se starý graf zahazuje a vytváří se nový.

Mezi důležité metody patří

- **async loadStylesheet()** - Načte style sheet do proměnné **visualStyleSheet** podle aktuální konfigurace **configuration**.
- **changeConfiguration()** - Nastaví novou konfiguraci **configuration** a zavolá metodu **createNewGraph**.
- **createNewGraph(loadStylesheet: boolean = true)** - Na základě konfigurace **configuration** vytvoří nový, prázdný graf (startý zahodí) a nastaví závislosti mezi moduly. Tato metoda pak ještě resetuje nastavení filtrů a plátna kde se vykresluje graf.
- **updateGraphSearcher()** - Pomocná metoda pro **createNewGraph**, která na základě konfigurace a grafu sestaví třídu schopnou vyhledávat nové vrcholy v grafu.

Kromě těchto metod komponenta obsahuje ještě Vue metodu **mounted**, která skryje uvítací obrazovku aplikace až se komponenta (a tedy i celá aplikace) inicializuje. Kontroluje také URL adresu, zda neobsahuje parametry **load**, **meta-configuration** nebo **configuration** které přimějí aplikaci načíst graf z url respektive načíst meta konfiguraci respektive konfiguraci.

## ApplicationLoadStoreMixin

`ApplicationLoadStoreMixin` rozšiřuje komponentu o načítání a ukládání grafu ze a do souboru.

### `askForSaveAndPerformAction(modal: boolean, callback: Function)`

Tato funkce zkontroluje, zda jsou v grafu nějaké neuložené změny a pokud ano, otevře `SaveDialog` komponentu. Podle odpovědi na dialog pak graf uloží a v případě kladného potvrzení zavolá `callback` funkci, která pak může například vytvořit nový graf. Pokud žádné změny nejsou, `callback` je volán okamžitě.

`loadFromFile(file: File)`, `loadFromUrl(url: string)` - Metoda otevře soubor respektive soubor z URL a přečte ho jako JSON. Výsledný objekt pak předá metodě `ObjectSave::restoreFromObject`.

`saveToFile()` - Vyrobí soubor se stavem aplikace který je možné stáhnout.

### 4.4.3 Interface file-save/ObjectSave

Interface `ObjectSave` předepisuje dvě metody `saveToObject(): object` a `restoreFromObject(object: any): void` které uloží stav třídy do serializovatelného objektu, respektive tento stav obnoví. Tento interface obsahují všechny třídy jež drží stav aplikace který je třeba uložit do souboru, pokud o to uživatel požádá.

Tento interface implementuje i komponenta `Application` jež tyto metody volá na modulech a jednotlivé výsledky pak spojí do jednoho objektu (v případě metody `saveToObject`). Výsledek je pak serializován do JSONu a uložen do počítače. V opačném případě je JSON deserializován a předán metodě `restoreFromObject` která volá tuto metodu na modulech které ji mohou volat na podtřídách.

**Poznámka:** Je třeba mít na paměti, že metoda `saveToObject` by měla vracet plain Javascript objekt, tedy veškeré neprimitivní typy jako objekty a pole musí být klonovány, pokud jsou ve vlastnictví Vue frameworku. Taktéž je třeba dbát na zpětnou kompatibilitu u metody `restoreFromObject`.

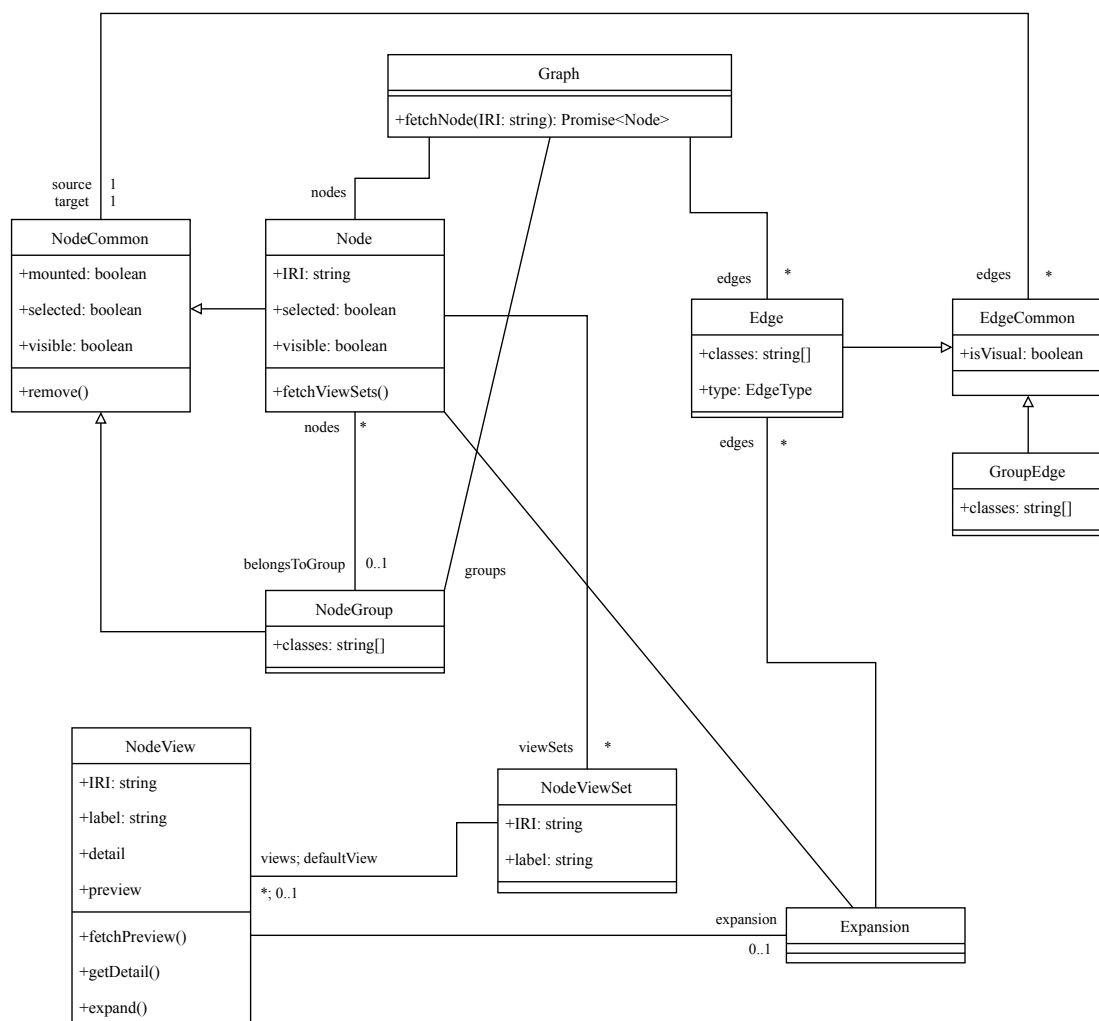
Do budoucna se nabízí tento interface rozšířit o více módů ukládání. Tento problém je popsán v poslední kapitole.

### 4.4.4 Třída Graph

Třída `Graph` umístěna v `graph/` spravuje stažený graf pod konkrétní konfigurací. Závisí na `server: RemoteServer` a `configuration: Configuration`. Pokud se konfigurace mění, nebo se načítá nový graf ze souboru, je tato třída zahozena. Ačkoli třídy jako `Node` nebo `Edge` obsahují fieldy pro obsluhu viditelného grafu, třída `Graph` a ostatní nemusí být explicitně použity na vykreslení grafu na plátno, ale mohou být použity pro držení obecného grafu.

#### Fieldy

- `nodes` objekt tříd `Node` - Seznam všech vrcholů grafu.



Obrázek 4.2: Class diagram tříd, které pracují s grafem.

- `edges` objekt tříd `Edge` - Seznam všech hran stažených ze serveru.
- `groups: NodeGroup[]` - Seznam všech neprázdných skupin vrcholů.
- `nodesVisual: NodeCommon[]` - Všechny `Node` a `NodeGroup` které jsou aktuálně viditelné v grafu. *Viditelnost vrcholů je popsána dále.*
- `groupEdges: GroupEdge[]` - Hrany které propojují skupiny uzlů nebo skupinu a normální uzel. Tyto hrany vznikly sloučením několika hran a existují pouze díky existenci nějaké skupiny vrcholů.
- `edgesVisual: EdgeCommon[]` - Všechny `Edge` a `GroupEdge` které jsou aktuálně viditelné v grafu. *Viditelnost hran je popsána dále.*

Poslední tři zmíněné fieldy jsou gettery. Protože se aplikace často na tyto proměnné dotazuje, je využito Vuexu a tyto hodnoty jsou ve skutečnosti computed properties. Protože třída `Graph` není komponentou, existuje komponenta `vuexComponent: GraphVuex` která se vytvoří automaticky společně s grafem a tyto zmíněné fieldy počítá. Tak docílíme cachování těchto hodnot a k jejich přepočítání dojde pouze tehdy, změní-li se hrany, nebo vrcholy v grafu.

## Metody

- `getNodeByIRI(IRI: string): Node|null` - Vrátí vrchol dle jeho IRI nebo `null` v případě, že neexistuje.
- `createNode(IRI: string): Node` - Vytvoří a zaregistruje nový vrchol v grafu.
- `createEdge(source: Node, target: Node, type: EdgeType): Edge` - Vytvoří a zaregistruje novou hranu v grafu.
- `createGroup(): NodeGroup` - Vytvoří a zaregistruje prázdnou skupinu vrcholů.
- `getAllTypes(): Set<NodeType>` a `getAllClasses(): Set<string>` - Pomocné metody které projdou všechny uzly v grafu a shromáždí jejich typy, respektive třídy. Těchto metod je využíváno při filtrování kdy bohužel nejsme schopni ze serveru získat kompletní množinu typů a tříd. Metody jsou volány když uživatel otevře okno s možnostmi filtrů.
- `fetchNode(IRI: string): Promise<Node>` - Vrchol je definován pouze svým IRI a tedy pro vytvoření vrcholu se aplikace nemusí ničeho dotazovat serveru. Tato metoda kromě vytvoření takového vrcholu ještě stáhne jeho `view-sets`, zvolí výchozí pohled a stáhne `preview` vrcholu. V případě že se nepodaří stáhnout tyto informace, vrchol do grafu nebude přidán a metoda vrátí `null`.
- `getOrCreateNode(IRI: string): Promise<Node>` - Obdobná metoda metodě výše. Pokud uzel neexistuje, volá předchozí metodu. Pokud uzel existuje, stáhne jeho `view-sets` a `preview` obdobně jako předchozí metoda a vrátí ho.

**Poznámka k viditelnosti** Veškeré metody které vytvářejí vrcholy, skupiny nebo hrany tyto prvky nevytvoří viditelnými. Aby mohl být prvek viděn v grafu, musí mu být nastaven `field mounted = true`.

#### 4.4.5 Třída `NodeCommon`

Třída `NodeCommon` je společný předek pro třídy `Node` a `NodeGroup` a zaštiťje především jejich vizuální vlastnosti.

##### Fieldy

- **`mounted: boolean`** - Určuje, zda má být vrchol viditelný v aplikaci. Na rozdíl od ostatních úrovní viditelností je tato myšlena tak, že uzel který má tuto hodnotu `false` v grafu vůbec nefiguruje a není jej možné ani najít v jiných částech aplikace (například mezi skrytými vrcholy). Využití najde v případě, kdy server pošle více vrcholů než uživatel žádal, nebo ho lze použít v případě browsingu seznamem, který je popsán v poslední kapitole.
- **`onMountPosition: [number, number]`** - Pomocný field který určuje, kde má být vrchol na plátně vykreslen, až bude nastaven na `mounted`.
- **`visible: boolean`** - Nastavuje uživatelskou viditelnost uzlu. Uživatel se může rozhodnout ručně skrýt uzel. Takový uzel si pak zachovává svou pozici na plátně a je v seznamu mezi ostatními skrytými uzly.
- **`get isVisible: boolean`** - Pomocný getter který určí, jestli je uzel viditelný.  
Pro `NodeGroup` se počítá jako `visible AND „alespoň jeden vrchol skupiny je isVisible“`.  
`Node` pak viditelnost počítá jako `visible AND „žáden filtr nezakazuje jeho viditelnost“`.
- **`selected: boolean`** - Pokud je vrchol vybrán, je zobrazen jeho detail v pravém panelu aplikace. Je možné vybrat více vrcholů. Vybrání vrcholů je napojeno na vybírání vrcholů na plátně. Lze vybrat i skryté vrcholy (uživatel, nebo filtrem). Pokud vrchol není `mounted`, je tato hodnota ignorována.
- **`get neighbourSelected: boolean`** - Pokud je vrchol vybrán, je zobrazen jeho detail v pravém panelu aplikace. Je možné vybrat více vrcholů. Vybrání vrcholů je napojeno na vybírání vrcholů na plátně. Lze vybrat i skryté vrcholy (uživatel, nebo filtrem). Pokud vrchol není `mounted`, je tato hodnota ignorována.
- **`get identifier: string`** - Jednoznačný identifikátor pro potřeby Cytoscape instance.
- **`get selfOrGroup: NodeCommon`** - Pomocný field který vrátí buď sebe, nebo skupinu do které vrchol patří. Využití má čistě pro zjednodušení programování a využívá se například v částech kódu, kde se řeší kontrakce hran.

- `lockedForLayouts: boolean` - Pokud daný layout podporuje zamykání pozic vrcholů, tato proměnná určuje, jestli je jeho pozice zamčená.

## Metody

- `remove()` - Smaže vrchol z grafu společně s jeho hranami. V případě skupiny smaže skupiny a vrcholy v ní.
- `selectExclusively()` - Nastaví `selected` pouze pro tento vrchol. V praxi to znamená, že se zobrazí detail tohoto vrcholu v pravém panelu.

**Poznámka** Díky Vue frameworku je hodně akcí vyvoláno právě nastavením nějaké proměnné. Kupříkladu proměnná `mounted` místo metody `mount()`. Vue framework automaticky při změně takovýchto proměnných provede příslušnou akci. Tento přístup má několik výhod, kupříkladu můžeme napojit proměnnou `visible` na checkbox a tak propojit viditelnost vrcholu se zaškrtačím políčkem v obou směrech. Nevýhoda tohoto přístupu je ta, že skutečná akce bude provedena až po skončení funkce. Pokud bychom například chtěli počkat, až se uzal namountuje a pak provést nějakou akci, musíme počkat na další `AnimationFrame`. Následující kód toto předvádí na akci, kdy chceme vrchol zobrazit v grafu a pak na něj přesunout pohled.

```
node.mounted = true;
await Vue.nextTick(); // Mounting function is called
this.area.fit(node)
```

## 4.4.6 Třída Node

Třída `Node` rozšiřuje třídu `NodeCommon` a reprezentuje vrchol získaný ze serveru, tedy entitu z RDF databáze.

### Fieldy

- `get classes: string[]` - Seznam tříd z posledního kompletního pohledu. (*viz dále*)
- `get edges: Edge[]` - Seznam hran příslušících vrcholu.
- `filters` jako objekt - Objekt jehož klíčem jsou identifikátory filtrů a hodnotou je `boolean` zda konkrétní filtr povoluje viditelnost vrcholu. Objekt je aktualizován kdykoli se změní stav vrcholu který daný filtr využívá nebo se změní nastavení filtrů. Všechny hodnoty nastavené na `true` jsou nutnou podmínkou viditelnosti vrcholu.
- `get shownByFilters: boolean` - Zdali všechny hodnoty objektu výše jsou nastaveny na `true`.
- `currentView: NodeView` - Určuje aktuální pohled na vrchol.



- `lastFullView: NodeView | null` - Tato proměnná odkazuje na poslední pohled na vrchol který měl kompletní **preview**. Změna pohledu je totiž okamžitá ale změněný pohled ještě nemusí mít stažený **preview**. To by způsobilo zbytečné „blikání“ uzlu když by uživatel měnil jeho pohledy. Na krátkou chvíli by totiž neměl žádné třídy a tedy by se ztratily jeho styly.
- `viewSets` jako objekt `NodeViewSet` - Seznam view setů. Může být `null`, pak ještě nebyly staženy.
- `belongsToGroup: NodeGroup | null` - Určuje zda vrchol patří do skupiny. Pokud ano, pak nebude zobrazen na plátně.

## Metody

- `async fetchViewSets(): Promise<void>` - Metoda asynchronně stáhne view sety. Tato metoda (společně s dalšími) je navržena tak, že druhým voláním nezačíná druhé stahování, ale vrátí `Promise` z prvního stahování. Takto nedojde k zatížení serveru a datových zdrojů. Ukázka metody je zobrazena an obrázku 4.4.6.
- `async useDefaultView(): Promise<NodeView>` - Metoda, v případě že vrchol nemá nastaven pohled, stáhne pohledy a nastaví výchozí.

**Poznámka** Aktuální interface serveru vrací při expanzi detail vrcholů. Detail formálně patří k pohledu ale v odpovědi ze serveru není určeno o jaký pohled se jedná. Takový pohled pak je nastaven jako aktuální, ale při volání metody `useDefaultView` bude ignorován a přepsán.

### 4.4.7 Třída `NodeGroup`

Třída `NodeGroup` rozšiřuje třídu `NodeCommon` a reprezentuje skupinu vrcholů `Node`. Aktuálně není podporováno, aby skupina obsahovala další skupiny. Všechny pomocné metody pak řeší přeskupování tak, že prvně vrcholy ze skupiny odstraní a vloží je do jiné.

## Fieldy

- `get classes: string[]` - Vrátí průnik tříd všech vrcholů které obsahuje. Takto lze docílit, že skupina podobných vrcholů bude mít stejný styl jako jednotlivé vrcholy.

## Metody

- `addNode(node: Node, overrideExistingGroup: boolean = false)`  
Vloží vrchol do skupiny.
- `checkForNodes()` - Pomocná metoda pro kontrolu, zda skupina vůbec nějaké vrcholy obsahuje. Pokud tomu tak není, odstraní se. Pokud obsahuje jen jeden vrchol, odstraní se a vrchol odstraní ze skupiny.

```

private fetchViewSetsPromise: Promise<void> = null;

async fetchViewSets(): Promise<void> {
    let asynchronouslyFetchViewSets = async () => {
        let result = await this.graph.server.getViewSets(...);

        if (result) {
            this.viewSets = ...;
        }
        this.fetchViewSetsPromise = null;
    }

    if (!this.viewSets) {
        if (!this.fetchViewSetsPromise) {
            this.fetchViewSetsPromise = asynchronouslyFetchViewSets();
        }

        return this.fetchViewSetsPromise;
    }
}

```

Obrázek 4.3: Příklad kódu na stažení view setů. Metoda má vevnitř další metodu která je volána pouze tehdy, neskončila-li předchozí Promise. Takto zařídíme pouze jeden požadavek na server současně.

**Poznámka** Tato operace ve skutečnosti může být kontrolována Vue frameworkem. Nicméně kvůli jednoduchosti ruční implementace jsou skupinu spravovány mimo framework.

## Generování GroupEdges

**NodeGroup** seskupuje několik vrcholů a z grafového hlediska provádí jejich kontrakci. To znamená, že musíme sjednotit několik hran do jedné.

Třída obsahuje field `groupEdgesCache` který udržuje tyto virtuální hrany. Kdykoli dojde ke změně v grafu, dojde k opětovnému přepočítání těchto virtuálních hran a pokud již existují, budou použity tyto existující. Pokud vznikne nová hrana, bude uložena do této cache a pokud nějaká hrana zanikla, bude z této cache odstraněna.

Tímto způsobem je docíleno automatického generování těchto hran přičemž existující hrany se nemění (jejich instance zůstává stejná). Celý proces je spravován Vue frameworkem a tedy se děje všechno automaticky.

Třída má field `get visibleGroupEdges: GroupEdge[]` který vrací všechny hrany kromě těch, které vycházejí z jiné **NodeGroup**. Jednoduchým sjednocením pak tedy dostaneme všechny hrany a každou právě jednou. Hrany se vypočítávají funkcí `getGroupEdgesInDirection` která kvůli komplexnosti počítá hrany jen v jednom směru a musí se tedy volat dvakrát. Jak již bylo zmíněno, tato funkce používá cache aby vracela již existující hrany. Výsledek funkce je pak ještě jednou cachován, tentokrát pomocí computed properties a tedy k přepočítání dojde pouze

tehdy, změní-li se graf.

#### 4.4.8 Třídy `EdgeCommon`, `Edge`, `GroupEdge`

Hrany RDF grafu jsou pak reprezentovány třídou `Edge` a hrany mezi skupinou a vrcholem, nebo dvěma skupinami třídou `GroupEdge`.

Společný předek předepisuje `get isVisual: boolean` který určuje, zda je hrana přítomná na plátně. Hrana je `isVisual` pokud její oba vrcholy jsou `mounted` a nepatří do skupiny. Obdobně pro `GroupEdge` je podmínka splněna pokud jsou oba vrcholy `mounted`.

##### Fieldy třídy `Edge`

- `type: EdgeType` - Typ hrany který určuje její label.
- `classes: string[]` - Třídy hrany.

#### 4.4.9 Třída `NodeViewSet`

Třída reprezentuje view set popsáný v kapitole 2.1.3. Jedná se o kontejner pro pohledy. Z hlediska implementace ej třída velmi jednoduchá. Obsahuje seznam pohledů jako `views` a výchozí pohled `defaultView: NodeView`.

Třída obsahuje metodu `createView(IRI: string): NodeView` která vytvoří a vhodně zaregistruje nový pohled. Protože jednotlivé pohledy jsou součástí jednoho požadavku (server na `view-sets` vrátí i pohledy) je logika vytváření pohledů ve třídě `Node`.

#### 4.4.10 Třída `NodeView`

Tato třída odpovídá pohledům tak, jak jsou definovány v kapitole 2.1.4.

##### Fieldy

- `detail: DetailValue[]` - Představuje pole detailů které lze získat ze serveru voláním `detail` a odpovídají . Interface `DetailValue` pak obsahuje `type`, `IRI` a `value`. Aktuálně je `value` v rámci aplikace považována za textový řetězec. Do budoucna je možné aplikaci rozšířit o více typů. Tomuto tématu se opět věnuje poslední kapitola.
- `preview: NodePreview` - Obsahuje základní informace o vrcholu které řídí jeho vykreslení na obrazovce. Preview lze získat voláním `preview` na serveru a odpovídá definici v .
- `expansion: Expansion` - Expanze dle tohoto pohledu.

##### Metody

- `async getDetail(): Promise<DetailValue[]>` - Stáhne, uloží a vrátí detail.

- `async fetchPreview(): Promise<NodePreview>` - Stáhne, uloží a vrátí preview.
- `async expand(): Promise<Expansion>` - Prove expanzi a vrátí ji.

Všechny tři funkce mají podobnou ochranu proti opětovnému volání, jako metoda `fetchViewSets`, jejíž ukázka je na obrázku 4.4.6. Metoda `expand` používá třídu `Graph` a vytváří nové vrcholy a nastavuje jim `preview`. Nově vytvořené vrcholy nemají nastavené `mounted`, takže lze snadno určit nově vytvořené vrcholy a ty vhodně layoutovat.

#### 4.4.11 Třída `Expansion`

Aktuálně třída `Expansion` je v aplikaci využívána pouze během procesu expanze. Do budoucna ji lze použít například pro znázornění vrcholů, které vznikly z jiných vrcholů. Obsahuje seznam vrcholů `nodes: Node[]` a hran `edges: Edge[]`

Veškeré třídy zde zmíněné pro práci s grafem implementují rozhraní `ObjectSave` a tedy je možné na třídě `Graph` volat metody pro uložení a obnovení stavu.

V následujících kapitolách práce bude popsáno, jak je tento modul grafu integrován do Vue frameworku tak, aby se samy vytvářely a aktualizovaly vrcholy a hrany, kdykoli dojde k jejich změně.

#### 4.4.12 Komponenta GraphArea

Tato komponenta je definována v souboru `src/component/graph/GraphArea.vue` a reprezentuje právě plátno na které se vykresluje graf. Kromě plátna ještě vyrábí tlačítka v pravém dolním rohu na jeho ovládání a vyhledávací políčko v levém horním rohu.

Komponenta přijímá spoustu properties, nejdůležitější jsou však `graph: Graph` a `stylesheet: ResponseStylesheet`. Jakmile se komponenta vyrobí, vrátí rodičovské komponentě `Application` instanci třídy `GraphAreaManipulator` formou Vue emitu, jež je schopna pracovat právě s grafovou oblastí.

##### GraphAreaStylesheetMixin

Komponenta používá `GraphAreaStylesheetMixin` kde je oddělena logika zpracování style sheetů.

Předtím, než jsou styly předány Cytoscape knihovně, jsou použity výchozí styly z `defaultStyles` které nastavují základní parametry. Poté jsou aplikovány styly které komponenta dostala od rodiče a ten si je stáhl ze serveru. Nakonec jsou použity styly z `viewOptionsStyles` které na základě `ViewOptions` přepisují základní pravidla.

Pokud se například rozhodneme v aplikaci skrýt hrany, právě poslední pravidlo je skryje.

Tato logika je sestavena v getteru `get finalStylesheet` kde jsou ještě přidány pomocné styly jako `:selected`. Nakonec je použit watcher který tuto proměnnou sleduje a když dojde ke změně předá tyto styly Cytoscape knihovně. Logika této funkce je v následujícím kódu

```
@Watch('finalStylesheet')
protected stylesheetUpdated() {
  this.cy.style(clone(this.finalStylesheet));
}
```

S pomocí dekorátoru nastavíme sledování `finalStylesheet`. Jakmile dojde k jeho změně, zavolá se metoda, která předá knihovně (`this.cy`) nový style sheet. Nesmíme zapomenout objekt klonovat, protože nemáme zaručeno, že ho knihovna nebude upravovat. Pokud by knihovna objekt vždy upravila, například z optimačních důvodů, Vue framework by zachytil změnu stylu a znovu by zavolal tuto funkci.

Komponenta pak ke každému vrcholu, hraně a skupině vytvoří vlastní podkomponentu, která tento prvek reprezentuje a spravuje. Jako příklad uvedme vytvoření komponent reprezentujících vrchol.

```
<graph-element-node
  v-for="node in graph.nodes"
  v-if="node.mounted && !node.belongsToGroup"
  :node="node"

  ...
/>
```

Mezi tlačítka v pravém dolním rohu se vykreslí i tlačítka aktuálního layoutu, pokud to daný layout podporuje. Layout v takovém případě musí dodat právě komponentu, která se zde vloží. V následujícím kódu se ověří, že `buttons` není prázdné a použije se jako ona komponenta. Té je pak předán jeden parametr, a to aktuální layout.

```
<component
  v-if="layoutManager.currentLayoutData.buttons"
  :is="layoutManager.currentLayoutData.buttons"
  :layout="layoutManager.currentLayout" />
```

#### 4.4.13 Komponenty `GraphElementNode`, `GraphElementNodeGroup` a `GraphElementNodeMixin`

Jak již bylo zmíněno v popisu Vue frameworku a z příkladu výše, framework automaticky pro každý vrchol grafu, který je `mounted` a není ve skupině, vytvoří komponentu `GraphElementNode`. Ta se pak stará o zobrazení tohoto vrcholu v rámci Cytoscape knihovny.

Obdobně jako jsme měli třídy `NodeCommon`, `Node` a `NodeGroup` i zde máme komponenty `GraphElementNodeMixin`, `GraphElementNode` a `GraphElementNodeGroup`, jež si navzájem odpovídají.

Mezi významné metody patří

- `mounted()` - Metoda se volá automaticky, když je komponenta vytvořena. V této metodě probíhá registrace vrcholu v Cytoscape knihovně, registrace důležitých událostí a registrace knihovny Popper<sup>11</sup>, jež má na starosti pozicování elementů vůči různým objektům, zde právě vůči vrcholům na plátně. Tak jsme schopni vykreslit k vrcholům ikonky. Prozatím existují dvě ikonky - uzamčení vrcholu a že je vrchol skupinou.
- `beforeDestroy()` - Metoda je volána před tím, než Vue framework zničí komponentu. V této části probíhá odstranění vrcholu z grafu.

**Souhrn** Díky těmto dvěma metodám a kompletnímu managementu Vue frameworku nám stačí do třídy `Graph` přidat nový uzel a ten se automaticky vykreslí do grafu. Odstraněním uzlu z kontejneru pak dojde k jeho smazání.

<sup>11</sup><https://popper.js.org/>

Komponenty obsahují spoustu dalších metod pro aktualizaci stylů, pozic vrcholů a podobně. Uvedme ještě konkrétní příklad pro vybírání uzlů.

```
@Watch('node.selected')
protected selectedChanged() {
  if (this.node.selected) {
    this.element.select();
  } else {
    this.element.unselect();
  }
}

mounted() {
  ...

  this.element.on("select", () => this.node.selected = true);
  this.element.on("unselect", () => this.node.selected = false);

  ...
}
```

Jak lze vidět z ukázky, první metodou zařídíme, že událost vybrání vrcholu putuje z Vue frameworku do Cytoscape knihovny. Ve druhé metodě pak definujeme opačný směr a máme tak docíleno, že pokud uživatel klikne na vrchol, dojde k nastavení `selected`, což může otevřít kupříkladu panel s detailem o vrcholu.

Hrany jsou pak vykresleny komponentou `GraphElementEdge`.

#### 4.4.14 Definice filtrů

Jednotlivé filtry jsou definovány v adresáři `src/filter/filters/<jméno-filtru>`. Každá definice filtru je pak soubor, který exportuje objekt odkazující na třídy a komponenty příslušného filtru. Tento objekt musí odpovídat rozhraní `FilterDefinition`, které je definováno následovně:

- **name: string** - Jednoznačný identifikátor filtru podle kterého se jednotlivé filtry identifikují a data filtru ukládají do souboru. Pro každý filtr by měl být neměnný a nesmí nastávat kolize.
- **component: typeof Vue** - Jedná se o Vue komponentu, která se napojí na filtr a jeden vrchol a provádí filtrování tohoto vrcholu. Výhoda použití komponenty místo klasické třídy je ta, že komponenta si může snadno zaregistrovat watchery a aktualizuje se pouze tehdy, když se změní sledovaný stav. Této komponentě je pak předán jeden vrchol `node: Node` a data filtru `filter`. Komponenta na základě dat filtru sleduje vrchol a zapisuje do `node.filters[name]` pravdivostní hodnotu, zda je vrchol dle tohoto filtru viditelný.

Jedná se o takzvanou renderless komponentu. Komponenta nemá žádný HTML výstup a pouze využívá Vue frameworku, který ji automaticky vytvoří a zničí, když se odstraní vrchol.

- **filter: Filter** - Nese data filtru, která se pak dají uložit do souboru. Interface **Filter** tedy rozšiřuje rozhraní **ObjectSave**. Kromě toho má metodu **reset()**, která nastaví filtr do výchozí hodnoty. Toho je využíváno při vytváření nového grafu, neboť většina filtrů je závislých právě na konkrétním grafu (například filtrování podle typu nemá smysl zachovávat, když měníme konfiguraci). Nakonec, toto rozhraní ještě předepisuje **readonly active: number**, jež je použit jako getter a říká „kolik filtrů je aktivních“. Tyto čísla jsou pak zobrazena v uživatelském rozhraní pro přehled uživatele.
- **tabs** - Jedná se o pole jednotlivých sekci filtrů. Každý filtr totiž může v rámci uživatelského rozhraní mít více oken pro nastavení. Každé okno je pak reprezentováno jedním prvkem této položky.
  - **component: typeof Vue** - Komponenta, která vykreslí okno s nastavením filtru.
  - **icon: string** - Data ikony, která bude zobrazena u názvu karty.
  - **active: (filter: Filter) => bool** - Vrací logickou hodnotu zdali je daná část filtru aktivní.
  - **text: string** - Kód reprezentující text, který se zobrazí v seznamu karet. Podle tohoto kódu se pak provede překlad.

## Komponenta VueFilterComponentCreator

Tato komponenta očekává **graph: Graph** a seznam filtrů **filter: FiltersList**. Pro každý filtr a každý vrchol pak vytvoří komponentu, (viz výše) která dostane zmíněný vrchol a graf. Tato komponenta je vytvořena v komponentě

**Application**.

Příklad definice filtru.

```
export default {
  name: "propertyFilter",
  component: PropertyFilterComponent,
  filter: new PropertyFilterData(),
  tabs: [{
    component: PropertyFilterSettingsTabClass,
    icon: mdiFormatListBulletedType,
    active: (filter: PropertyFilterData) => filter.class.active,
    text: "filters.propertyFilter.class.tab",
  }, {
    component: PropertyFilterSettingsTabType,
    icon: mdiFormatListBulletedType,
    active: (filter: PropertyFilterData) => filter.type.active,
    text: "filters.propertyFilter.type.tab",
  }],
} as FilterDefinition;
```



### 4.4.15 Definice Layoutů

Struktura layoutů je velmi podobně koncipována jako struktura filtrů. Layouty se registrují v rámci třídy `LayoutManager`. Jednotlivé layouty jsou definované jako objekty implementující rozhraní `LayoutData`. Toto rozhraní je definováno následovně:

- `name: string` - Unikátní identifikátor layoutu podle kterého se ukládá stav layoutu do souboru.
- `layout: Layout` - Třída nesoucí stav layoutu a provádí samotné layoutování.
- `settingsComponent: typeof Vue` - Vue komponenta, která vykreslí stránku s nastavením layoutu. Této komponentě je předán parametr `layout` a data tohoto layoutu upravuje. Na rozdíl od filtrů, každý layout může vyrobit jen jednu stránku s nastavením. Titulek této stránky je pak sestaven ze jména `name` jako kód na přeložení.
- `buttons?: typeof Vue` - Volitelný parametr jako komponenta, která vykreslí dodatečná tlačítka v pravém dolním rohu aplikace. Této komponentě je opět předán parametr `layout`. Ukázka integrace této komponenty je na konci kapitoly 4.4.12.

### 4.4.16 Abstraktní třída Layout

Tuto třídu rozšiřují třídy, které jsou schopny provádět layoutování vrcholů grafu. Layoutování přitom reaguje na různé události aplikace tak, že jsou na aktivním layoutu volány odpovídající metody. Layout se pak může rozhodnout tyto metody přepsat a tedy na ně reagovat. Jedná se o následující metody:

- `onAddedNodes()` - Voláno, když je do grafu přidán vrchol uživatelem. **Poznámka:** Při expanzi tato funkce není volána.
- `onExpansion(expansion: Expansion)` - Voláno při expanzi, která je předána jako parametr. Nově přidáné vrcholy nemají nastaveno `mounted` a tedy je lze rozlišit v rámci funkce.
- `onDrag(isStartNotEnd: boolean)` - Metoda je zavolána s kladným parametrem, pokud uživatel začal přesouvat vrcholy v grafu. Při skončení je funkce zavolána se záporným parametrem.
- `onCompactMode(nodes: NodeCommon[] | null, edges: EdgeCommon[] | null)` - Metoda je volána při zapnutí kompaktního módu. V takovém případě je předán seznam vrcholů a hran, které se kompaktního módu účastní. Opětovným zavoláním lze změnit množinu prvků účastnících se kompaktního módu. Jakmile je kompaktní mód ukončen, oba parametry budou nastaveny na `null`.
- `onLockedChanged()` - Voláno při změně zafixování vrcholů na grafu.

- `onGroupBroken(nodes: Node[], group: NodeGroup)` - Voláno při rozbití skupiny `group` na samostatné vrcholy `nodes`.

Jakmile je layout aktivován, je na něm volána metoda `activate`. Naopak, pokud je nastaven layout jiný, je voláno `deactivate`, jež by mělo zrušit veškeré probíhající animace na grafu a přestat layoutovat. Když layout přestane být aktivní, nebudou již na něm volány předešlé metody při významných událostech. Při změně layoutu je nejprve na starém voláno `deactivate` a pak na novém layoutu `activate`.

Pro explicitní spuštění layoutu je možné volat funkci `run`.

Layout implementuje `ObjectSave`.

#### 4.4.17 LayoutManager

Třída spravuje všechny layouty. Je konstruována s polem objektů `LayoutData`, tedy při jejím vytvoření dostane seznam podporovaných layoutů. Tento seznam může být později měněn.

Metoda `switchToLayout(name: string)` pak po zadání identifikátoru layoutu správně provede jeho změnu.

#### 4.4.18 Vyhledávání vrcholů

##### Interface Searcher

Všechny vyhledávače implementují následující funkce:

- `query(query: string)` vrací mapu z IRI do `SearcherResult` nebo tu samou mapu v Promise - Na základě vyhledávaného výrazu `query` funkce vrátí nalezené vrcholy jako JavaScriptovou mapu, kde klíčem je IRI vrcholu a hodnotou je objekt `SearcherResult`, který představuje dodatečné informace o nalezeném vrcholu. Metoda také může vrátit stejnou mapu obalenou v Promise, pokud se například dotazuje na internetu.
- `getByIRI(IRIs: IterableIterator<string>)` vrací to samé, jako předchozí metoda - Tato funkce vrátí informace o vrcholech zadaných pomocí jejich IRI. Obdobně jako předchozí funkce může vracet Promise.

##### Interface SearcherResult

- `readonly IRI: string` - IRI vrcholu, který objekt reprezentuje.
- `text: string | string[]` - Text, jež bude zobrazen u nalezeného vrcholu jako „popisek“ tohoto vrcholu. Pokud se jedná o `string`, bude normálně zobrazen. To se použije v případě, kdy známe název vrcholu. Pokud se jedná o pole, první prvek reprezentuje kód pro překlad a další prvky pak parametry. Tohoto lze využít, pokud název vrcholu neznáme a chceme tedy vypsát obecnou hlášku, například „Vrchol s identifikátorem XYZ“.
- `icon: string` a `color: string` - Popisuje, jak má vypadat ikonka u vyhledávaného výsledku. Používá se pro rozlišení více typů vyhledávačů, například v autocomplete a v grafu.

## GraphSearcher

Tato třída pak reprezentuje modul vyhledávání. Drží si jednotlivé vyhledávače a je schopna v nich vyhledávat. Obsahuje metodu `search`, jejímž prvním parametrem je vyhledávaný řetězec a druhým je callback, který je volán **několikrát** se seřazeným seznamem výsledných vrcholů a informací, zda vyhledávání stále probíhá (například při dotazování na server).

## GraphSearcherQuery

Předchozí třída při vyhledávání vytvoří tuto třídu, která reprezentuje konkrétní vyhledávání.

- **MAX\_RESULTS\_PER\_SEARCHER**: *number* - Představuje maximální počet výsledků, které vyhledávání vrátí.
- **nodes** - Již nalezené vrcholy z první fáze vyhledávání.
- **firstPhase()** - Tato metoda na všech vyhledávačích zavolá první metodu `query` a shromáždí výsledky následovně. Výsledky, které byly navraceny hned (tedy nejedná se o Promise) jsou sjednoceny, uloženy do privátní proměnné jako nalezené vrcholy a je zavolána druhá fáze. Jakmile se dokončí nějaká Promise z vyhledávačů, které nevrátily výsledek ihned, aktualizuje se seznam nalezených vrcholů a opět se zavolá druhá fáze.
- **secondPhase()** - Druhá fáze je volána vždy, když se aktualizoval seznam nalezených vrcholů z první fáze. Vrcholy, které nebyly prohledávány v rámci druhé fáze se pak postupně předají všem vyhledávačům pomocí druhé metody `getByIRI` a jakmile ta vrátí výsledek, (buď okamžitě, nebo až po splnění Promise) zavolá se callback s aktualizovaným seznamem nalezených vrcholů.

V aplikaci jsou implementovány následující vyhledávače

- **IRIConstructorSearcher** - Pokud hledaný výraz odpovídá regulárnímu výrazu, pak se tento výraz obalí prefixem a sufixem a takto bude vytvořeno IRI. Dá se využít kupříkladu u Wikidat, jejichž IRI mají tvar jednotného prefixu následován výrazem `Q<číslo>`.
- **IRIIdentitySearcher** - Pokud hledaný výraz odpovídá regulárnímu výrazu popisující, jak by mělo vypadat IRI, bude vrácen výsledek s tímto IRI označen jako „podporované IRI“. V případě, že dotaz neodpovídá regulárnímu výrazu, bude opět navracen jeden výsledek s IRI rovnou vyhledávanému dotazu a označen jako „nepodporované IRI“.
- **LocalGraphSearcher** - Vyhledává v grafu podle popisku (label pod preview). Ignoruje velikost písmen.
- **SimpleJsonSearcher** - Stáhne z internetu soubor ve formátu JSON-LD se seznamem vrcholů a jejich popisky. Po stažení je index uložen a tedy další dotazy již nevrací Promise, ale přímo výsledky.

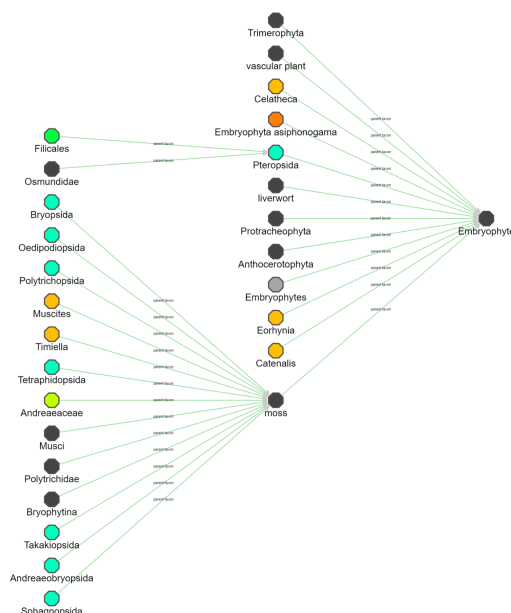
# 5. Uživatelské testování

## 5.1 System Usability Scale (SUS)

System Usability Scale<sup>1</sup> je jednoduchý test na přívětivost uživatelského rozhraní aplikace. Cílem tohoto testování je seznámit uživatele s aplikací a pak mu položit 10 otázek, které SUS definuje. Na základě odpovědí na tyto otázky je pak spočteno skóre uživatelské přívětivosti. Aby testování bylo objektivní, je potřeba aplikaci testovat na osobách, jež se nepodílely na žádné části softwarového vývoje.

Testovaným osobám bude vysvětlen základní princip fungování aplikace a poté jim budou určeny jednoduché úkoly, které musí splnit. Znění těchto úkolů bude úmyslně napsáno velmi obecně a nebudou použity termíny, jež jsou použity v aplikaci. Testovaná osoba pak musí sama přijít na to, jak danou akci provést a díky tomu bude schopna objektivně zhodnotit přívětivost uživatelského rozhraní.

1. Zjistěte, které taxony řadíme pod Vyšší rostliny (latinsky Embryophyta).
2. Pokuste se vyrobit graf podobný grafu na obrázku. Toto uspořádání vrcholů do stromu se nazývá **dagre**. Konkrétní pořadí vrcholů není důležité. Místo ručního hledání Pteropsidy a moss se je pokuste vyhledat pomocí jejich názvu.



3. Stáhněte si aktuální graf do souboru, stránku aktualizujte a graf ze souboru opět načtěte.
4. Nechte si zobrazit ze všech vrcholů jen ty, co mají třídu „genus“ (tedy chceme ty taxony, jež reprezentují rod).
5. Odstraňte z grafu libovolný vrchol.

<sup>1</sup><https://www.usability.gov/how-to-and-tools/methods/system-usability-scale.html>

### 5.1.1 Výsledky testování SUS

TODO: Zde budou zpracované otázky a výsledné skóre. Možná nějaké tabulky atp.

## 5.2 Výsledky obecných otázek

V rámci testování SUS byli respondenti navíc dotázáni, jaké části aplikace pro ně byly uživatelsky nepřívětivé a čemu neporozuměli. Výsledky jsou sepsány v následujícím seznamu.

- Tlačítka v pravém dolním rohu grafové oblasti sloužící na obsluhu grafu (layoutování, zobrazení celého grafu, kompaktní mód) nejsou popsány a respondent odpověděl, že měl problém zjistit, co dělají.

Tlačítkům bude vhodné přidat tooltipy vysvětlující jejich funkce obdobně, jak je tomu v jiných částech aplikace.

- Respondent nepochopil uživatelské rozhraní pro výběr layoutu. Neuvedl si, že jednotlivé karty odpovídají konkrétním layoutům a vždy je aktivní pouze jeden layout. Měnil nastavení jiného layoutu, než toho, který byl aktivní.

Zvážit kompletní změnu uživatelského rozhraní, které uživatele nutí nejprve zvolit layout a poté měnit jeho nastavení.

Úkoly popsané v rámci SUS průzkumu sloužily také jako určitá forma uživatelského testování aplikace. I když nebylo jasné definováno, jak by aplikace měla na konkrétní úkoly reagovat, můžeme předpokládat, že úkoly proběhly úspěšně, neboť je všichni z respondentů dokončili. Měřením code coverage bylo zjištěno, že provedením všech úkolů bylo pokryto 77% kódu. Většina aplikace tedy byla těmito úkoly úspěšně otestována. „Provedením úkolů“ je myšlena nejjednodušší cesta, jak lze daný úkol v aplikaci provést.

Měření bylo prováděno s pomocí IDE WebStorm<sup>2</sup> od JetBrains pod prohlížečem Google Chrome. Code coverage se měří z řádků, které byly v aplikaci spuštěny, přičemž se do výpočtu podílu nezahrnují komentáře, prázdné řádky a definice rozhraní, které se do JavaScriptu nepřekládají.

---

<sup>2</sup><https://www.jetbrains.com/webstorm/>

# Závěr

V rámci této práce byla dle zadaných požadavků navržena, implementována a otestována webová aplikace pro procházení a vizualizaci dat z grafových databází. To, jak jsou data zobrazena a jakým způsobem je lze procházet popisuje konfigurace.

Uživatel si může do aplikace přidat vrcholy reprezentující existující entity a ty může expandovat a rozšiřovat tak graf o nové vrcholy. Aplikace poskytuje uživateli dodatečné informace o vrcholech, obarvuje je podle jejich typu, umožňuje filtrovat graf a podporuje několik layoutů, jak může být graf zobrazen. Pro snazší práci lze vrcholy seskupovat do skupin, a je možné si vrcholy zobrazit tak, aby je bylo snadné procházet po jejich sousedech.

## 5.3 Možnosti vylepšení

Tato kapitola rozebírá možné způsoby, jak aplikaci rozšířit do budoucna a možnou implementaci, jak tyto rozšíření provést.

### Rozhraní mezi serverem a klientem

Aktuálně je rozhraní navrženo tak, že serveru lze posílat požadavky samostatně. Toto řešení je nicméně nevyhovující v případě, kdy bude klient vyžadovat odpovědi na více požadavků současně. Budou-li poslány v jedné zprávě, kromě snížení komunikace mezi serverem a klientem bude moci server optimalizovat dotazy na datové zdroje. Kupříkladu dvě expanze lze teoreticky spojit do jednoho SPARQL dotazu.

Server také může posílat data, o která klient nežádal, v případě, že se bude domnívat, že je klient může požadovat. Například když klient požádá o **view-sets**, může server rovnou poslat **preview** na výchozí pohled, neboť o něj bude klient zřejmě žádat.

### Cachování

Problém cachování již v této práci rozebírán byl. S každým požadavkem klienta se server dotazuje datových zdrojů. Tato operace je časově náročná a zbytečně dochází k vytěžování datových zdrojů.

Kupříkladu SPARQL endpoint Wikidat má omezení na počet požadavků a při vysoké aktivitě klienta může server dostat IP ban.

Možnosti cachování jsou následující:

- Cachovat výsledky jednotlivých dotazů. Například budou cachována data expanze pro každý vrchol kompletně zvlášť. Toto řešení je jednoduché, nicméně náročné na paměť, neboť jednotlivé vrcholy budou uloženy několikrát.
- Cachovat malé bloky a ty pak sestavovat. V případě s expanzí bude uložen pouze seznam IRI vrcholů a data k vrcholům budou uložena samostatně. Tak dojde k omezení všech redundantních informací. U každého bloku bude

uložen čas stažení, který bude kontrolován a v případě příliš starého výsledku se cache zahodí.

- Cachovat kompletní graf. Místo ukládání dat, která jsou výsledky jednotlivých dotazů, stáhneme část grafu a dotazy nad tímto grafem budou provedeny lokálně. Tímto jsme schopni z cache získat i taková data, na která se dotazujeme poprvé.

## Detail

Detail zobrazuje podrobné informace o vrcholu, které získá z různých literálů. Aktuálně je ignorován typ těchto literálů, je tedy možné s ním pracovat a na základě tohoto typu pak data zpracovávat. Kromě jednoduchých typů jako jsou čísla, můžeme zpracovávat například telefonní čísla, e-mailové adresy, nebo URL adresy.

Typem položky detailu může být i vrchol, přes který se pak bude moci uživatel dostat na skutečný vrchol v grafu. Tento vztah pak nahrazuje klasickou hranu v grafu a dá se použít tam, kde je podle autora konfigurace nevhodné, aby byly vrcholy spojené skutečnou hranou.

Detail je možné rozšířit i tak, aby neobsahoval jen skalární hodnoty, ale i pole a objekty. Toto opět nahrazuje klasický grafový přístup, kdy místo několika vrcholů (literálů) vypíšeme seznam těchto hodnot v detailu.

## Práce s daty v detailu

V případě, že budeme mít typované položky detailu, nabízí se s těmito daty pracovat.

Jednou možností je zvolit množinu vrcholů a jejich data vykreslit do grafu (myšleno jako koláčový/sloupkový graf). Uvedme konkrétní příklad, opět s Karlem Čapkem. Necháme si vrchol expandovat a zobrazit si všechna jeho díla. Za předpokladu, že u každého vrcholu díla je položka „rok publikace“, můžeme z těchto dat sestavit histogram a podívat se, v jakém období Karel Čapek psal a kdy byl vrchol jeho kariéry.

Pokročilejším grafem by pak mohlo být zanesení roku díla na osu  $x$  a popularitu tohoto díla na osu  $y$ .

## Textové procházení grafem

Pro některé scénáře může být pro uživatele nepraktické si vrcholy zobrazovat v grafu a hledat jednotlivé vrcholy v něm. Někdy může být žádoucí tyto vrcholy do grafu vůbec nepřidávat. V takovém případě by bylo možné přejít na textové procházení grafem, kdy se jednotlivé expanze vypíší jako seznam vrcholů a uživatel jimi bude moci procházet obdobně jak je tomu u webových stránek. Všechny tyto vrcholy by se implicitně v grafu nezobrazovaly (tedy by nebyly *mounted*) a až konečný vrchol by uživatel mohl převést do grafu.

## Nástroj pro úpravu a návrh konfigurací

Aktuálně jsou konfigurace definovány pomocí Turtle notace v RDF grafech. To bohužel znemožňuje snadné vytváření konfigurací běžnými uživateli. Nabízí se tedy vytvořit nástroj, který by tyto konfigurace vytvářel a ukládal je do úložiště konfigurací, aby nebylo potřebné je ukládat ručně.

### 5.4 Aplikace

*Tato sekce obsahuje snímky z aplikace a velmi stručný návod na její obsluhu. Cílem této sekce není poskytnutí uživatelské dokumentace. Pouze popisuje její základní principy.*

Po spuštění aplikace je uživatel vyzván k výběru konfigurace. V tomto okně je také možné zadat IRI konfigurace a meta konfigurace ručně, nebo otevřít existující graf ze souboru. Jakmile je uživatelem vybrána konfigurace, je vyzván k výběru prvního vrcholu, který bude přidán do grafu. To je možné provést ze seznamu předdefinovaných vrcholů, nebo může použít vyhledávací pole.

Hlavní okno aplikace má uprostřed grafovou oblast. S tou je možné manipulovat pomocí myši, nebo červených tlačítek v pravém dolním rohu. Po kliknutí na vrchol, označení více vrcholů, nebo skupiny se zobrazí pravý boční panel s detailními informacemi. Levý panel se automaticky skrývá a obsahuje další akce, které je možné s grafem provést. V levém horním rohu se také nachází pole, přes které lze přidávat a vyhledávat vrcholy v grafu.


#### Výběr konfigurace

##### Podporované konfigurace

Seznam všech podporovaných konfigurací knowledge graph browseru.



**Státní správa  
České republiky**  
Procházení dat  
popisujících veřejnou  
správu České republiky.



**Wikidata**  
Procházení dat největší  
online encyklopedie.

NASTAVIT RUČNĚ   OTEVŘÍT ZE SOUBORU / URL

ZAVŘÍT

čeština

Obrázek 5.1: Úvodní obrazovka s výměrem mezi dvěma meta konfiguracemi. Je také možné ručně zadat IRI konfigurací nebo stáhnout graf ze souboru.



## Výběr konfigurace

 [Podporované konfigurace](#)

### Wikidata

Procházení dat největší online encyklopedie.

**Taxonomie rostlin a živočichů**

Možnost procházet sítě taxonů všech zvířat a rostlin jež jsou na Wikipedii.

**Slavné osobnosti**

Procházení filmů, herců a spisovatelů.

[NASTAVIT RUČNĚ](#)

[OTEVŘÍT ZE SOUBORU / URL](#)

[ZAVŘÍT](#)

čeština

Obrázek 5.2: Obrazovka s výběrem konkrétních konfigurací v rámci Wikipedie (Wikidat).

## Slavné osobnosti

Procházení filmů, herců a spisovatelů.

### Vyberte počáteční uzel

Můžete zadat IRI identifikátor uzlu nebo se ho pokusit vyhledat, pokud to daná konfigurace umožňuje.

 Zadejte IRI, název uzlu, kód...

#### Předdefinované uzly

Bob Kahn

human

Marie Curie-Skłodowská

entity

Albert Einstein

human

Tim Berners-Lee

human

Vint Cerf

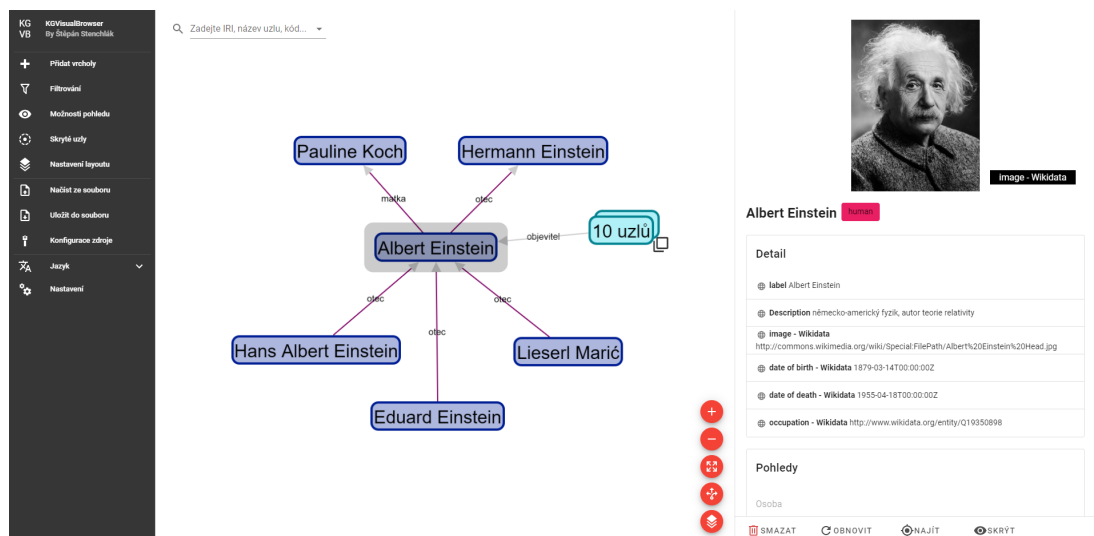
human

[ZPĚT](#)

[JÍT KE GRAFU](#)

čeština

Obrázek 5.3: Obrazovka s výběrem počátečního vrcholu. Ten je možné vybrat z předdefinovaných, nebo je možné použít vyhledávací pole.



Obrázek 5.4: Hlavní okno aplikace s již načteným grafem a vybraným vrcholem. Levý panel je zde rozevřený.

The 'Nastavení layoutu' dialog box is shown. It has a title bar 'Nastavení layoutu' and a tabbed interface with 'COLA' selected. A warning message states: 'Cola layout může značně ovlivnit výkon aplikace na větších grafech kvůli složitým výpočtům. Pro větší grafy zvažte použití jiného, méně náročného, layoutu.' The settings include:
 

- ☒ Animovat: Nastaví, zda výpočet layoutu bude animován, nebo se počká až na koncový stav.
- ☒ Spustit layoutování pokud uživatel přetahuje uzly: Kdykoli uživatel přesune uzel spustí se layoutování celého grafu.
- ☐ Při expanzi layoutovat pouze uzly expanze: Pokud uživatel provede expanzi, musí proběhnout layoutování. Tímto přepínačem zvolíte, zda se má layoutovat celý graf, nebo jen část, kde proběhla expanze.
- Extra prostor pro uzel: 10 px. Nastaví kolem uzlu extra prostor, ve kterém se nesmí nacházet žádný jiný uzel.
- Délka hran: 200 px. Nastaví optimální délku hran. Hodí se zvýšit v případě, že popisky uzlů a hran splývají.
- ☒ Expandované uzly seskupovat do skupiny: Nastaví, jaký je minimální počet nových uzlů z expanze, aby byly seskupeny do skupiny.

 At the bottom are buttons: 'PONECHAT PŮVODNÍ LAYOUT' and 'NASTAVIT TENTO LAYOUT'.

Obrázek 5.5: Dialog pro výběr a nastavení layoutů.

# Seznam použité literatury

System usability scale (sus) | usability.gov. URL <https://www.usability.gov/how-to-and-tools/methods/system-usability-scale.html>.

FRANZ, M., LOPES, C. T., HUCK, G., DONG, Y., SUMER, O. a BADER, G. D. (2015). Cytoscape.js: a graph theory library for visualisation and analysis. *Bioinformatics*, **32**(2), 309–311. ISSN 1367-4803. doi: 10.1093/bioinformatics/btv557. URL <https://doi.org/10.1093/bioinformatics/btv557>.

PRUD'HOMMEAUX, E. a CAROTHERS, G. (2014). RDF 1.1 turtle. W3C recommendation, W3C. <https://www.w3.org/TR/2014/REC-turtle-20140225/>.

RAIMOND, Y. a SCHREIBER, G. (2014). RDF 1.1 primer. W3C note, W3C. <https://www.w3.org/TR/2014/NOTE-rdf11-primer-20140624/>.

SEABORNE, A. a HARRIS, S. (2013). SPARQL 1.1 query language. W3C recommendation, W3C. <https://www.w3.org/TR/2013/REC-sparql11-query-20130321/>.

YOU, E. (2020). *Vue js framework*. URL <https://vuejs.org/>.

# Seznam obrázků

1	Ukázka části grafu jež může reprezentovat Karla Čapka. . . . .	3
2	Pohled na Karla Čapka jako na osobu mající rodinu (vlevo) a na spisovatele jež je autorem literárních děl (vpravo) . . . . .	4
1.1	Příklad grafu, který získáme z předešlých dvou ukázek. . . . .	6
2.1	Class diagram konfigurací včetně pozdější implementace meta konfigurace a rozšíření konfigurace. . . . .	10
2.2	Use case diagram dle uživatelských požadavků. . . . .	17
3.1	Komunikace mezi klientem, serverem a datovými zdroji. Cachování na serveru ještě implementováno není. . . . .	22
3.2	Příklad komunikace mezi klientem, serverem a RDF databází při spuštění aplikace. Na začátku klient nejprve vybírá konfiguraci. (Příčemž je dostačující volat pouze <code>meta-configuration</code> , protože výsledek dotazu obsahuje i konfigurace.) Poté začne stahování style sheetu a prvního vrcholu. Při stahování vrcholu se nejprve stáhnou <code>view-sets</code> a poté z výchozího pohledu <code>detail</code> a vrchol se zobrazí na grafu. Případné <code>expand</code> a <code>preview</code> vypadají obdobně. . . . .	24
3.3	Závislost modulů a dalších částí aplikace na sobě. <b>TODO: Mohl bych se zeptat? Počítá se toto jako popis vazeb mezi moduly? Tento obrázek popisuje skutečné závislosti a používá skutečná jména tříd z aplikace. Můžu ho tedy v této části použít (i když ne vždy odpovídá textu v této kapitole), nebo ho mám dát k implementaci a tady dát něco jiného?</b> . . . . .	28
4.1	Doporučený způsob komunikace mezi komponentami ve Vue frameworku . . . . .	33
4.2	Class diagram tříd, které pracují s grafem. . . . .	41
4.3	Příklad kódu na stažení view setů. Metoda má vevnitř další metodu která je volána pouze tehdy, neskončila-li předchozí Promise. Takto zařídíme pouze jeden požadavek na server současně. . . . .	46
5.1	Úvodní obrazovka s výměrem mezi dvěma meta konfiguracemi. Je také možné ručně zadat IRI konfigurací nebo stáhnout graf ze souboru. . . . .	60
5.2	Obrazovka s výběrem konkrétních konfigurací v rámci Wikipedie (Wikidat). . . . .	61
5.3	Obrazovka s výběrem počátečního vrcholu. Ten je možné vybrat z předdefinovaných, nebo je možné použít vyhledávací pole. . . . .	61
5.4	Hlavní okno aplikace s již načteným grafem a vybraným vrcholem. Levý panel je zde rozevřený. . . . .	62
5.5	Dialog pro výběr a nastavení layoutů. . . . .	62

# Seznam tabulek

# Seznam použitých zkratek

## A. Přílohy

### A.1 První příloha