



**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

Poděkování.

Název práce: Vizuální browser grafových dat

Autor: Štěpán Stenclák

Katedra: Katedra softwarového inženýrství

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

Abstrakt: Abstrakt.

Klíčová slova: klíčová slova

Title: Visual browser of graph data

Author: Štěpán Stenclák

Department: Name of the department

Supervisor: doc. Mgr. Martin Nečaský, Ph.D., department

Abstract: Abstract.

Keywords: key words

# 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 Rozšíření požadavků . . . . .	9
2.1.2 Meta konfigurace . . . . .	9
2.1.3 Konfigurace . . . . .	10
2.1.4 ViewSet . . . . .	10
2.1.5 View . . . . .	10
2.1.6 Expansion . . . . .	11
2.1.7 Preview . . . . .	11
2.1.8 Detail . . . . .	11
2.1.9 Dataset . . . . .	12
2.1.10 Visual style sheet . . . . .	12
2.2 Server . . . . .	14
2.3 Uživatelské a systémové požadavky . . . . .	15
2.3.1 Konfigurace a stylesheet . . . . .	15
2.3.2 Vložení vrcholů . . . . .	15
2.3.3 Detail vrcholu . . . . .	15
2.3.4 Skrývání uzlů . . . . .	16
2.3.5 Expanze . . . . .	16
2.3.6 Filtrování vrcholů . . . . .	16
2.3.7 Vícejazyčné uživatelské rozhraní . . . . .	17
2.3.8 Stažení grafu do souboru . . . . .	17
2.3.9 Odstranění vrcholu . . . . .	19
2.3.10 Vyhledávání vrcholů s pomocí autocomplete . . . . .	19
2.3.11 Podpora layoutů . . . . .	19
2.3.12 Seskupování vrcholů . . . . .	19
<b>3 Návrh architektury</b>	<b>21</b>
3.1 Server . . . . .	21
3.1.1 Jazyková podpora . . . . .	22
3.1.2 API . . . . .	22
3.2 Klient . . . . .	25
3.2.1 Moduly . . . . .	25

<b>4</b>	<b>Implementace</b>	<b>28</b>
4.1	Vue.js framework . . . . .	28
4.1.1	Vuex . . . . .	28
4.1.2	Vue framework . . . . .	29
4.1.3	Loaders . . . . .	30
4.2	Programátorská dokumentace . . . . .	32
4.2.1	Vstupní skript a pomocné soubory . . . . .	32
4.2.2	Komponenta Application . . . . .	32
4.2.3	Interface file-save/ObjectSave . . . . .	34
4.2.4	Třída Graph . . . . .	35
4.2.5	Třída NodeCommon . . . . .	36
4.2.6	Třída Node . . . . .	38
4.2.7	Třída NodeGroup . . . . .	39
4.2.8	Třídy EdgeCommon, Edge, GroupEdge . . . . .	40
4.2.9	Třída NodeViewSet . . . . .	40
4.2.10	Třída NodeView . . . . .	41
4.2.11	Třída Expansion . . . . .	41
4.2.12	Komponenta GraphArea . . . . .	42
4.2.13	Komponenty GraphElementNode, GraphElementNodeGroup a GraphElementNodeMixin . . . . .	43
<b>5</b>	<b>Uživatelské testování</b>	<b>45</b>
5.1	System Usability Scale (SUS) . . . . .	45
	<b>Závěr</b>	<b>46</b>
	<b>Seznam použité literatury</b>	<b>47</b>
	<b>Seznam obrázků</b>	<b>48</b>
	<b>Seznam tabulek</b>	<b>49</b>
	<b>Seznam použitých zkratk</b>	<b>50</b>
<b>A</b>	<b>Přílohy</b>	<b>51</b>
A.1	První příloha . . . . .	51

# 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.

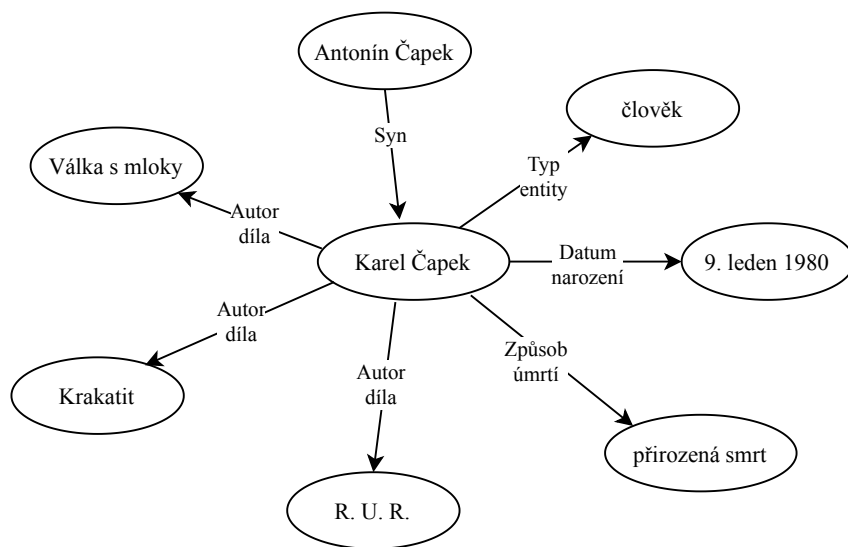
Možností řešení tohoto problému je zveřejňovat data i ve strojové podobě. Nabízí se například tabulky ve formátu CSV, nebo komplikovanější data ve formátu JSON a XML. Zde je již snadné číst data a dále je zpracovávat, ale programátor je stále nucen pochopit a přizpůsobit program na rozhraní těchto dat.

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.

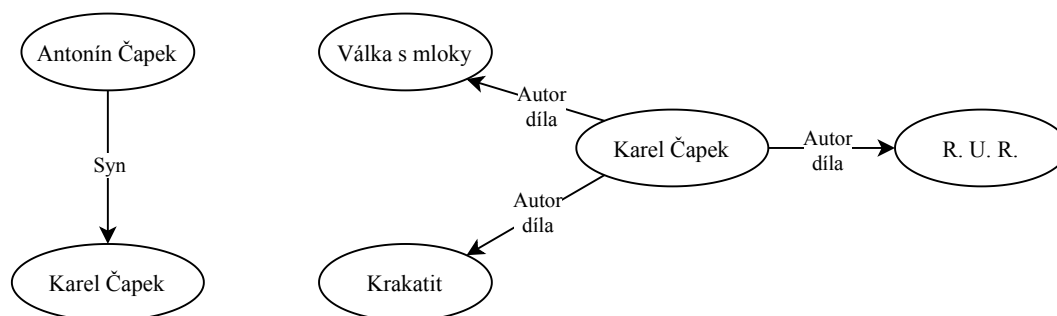


Obrázek 1: Ukázka části grafu jež může reprezentovat Karla Čapka.

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. Možností řešení tohoto problému je dívat se na entity pouze určitým pohledem, a tedy zobrazit pouze ty vztahy k vrcholu, jež odpovídají 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 bylo 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 z Wikidat** - 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 (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 přidat IRI uzlům různé hodnoty a podporují právě i multijazyčnost. Taktovému grafy pak dokáží 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>1</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 **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><https://www.w3.org/TR/turtle/>

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 .
```

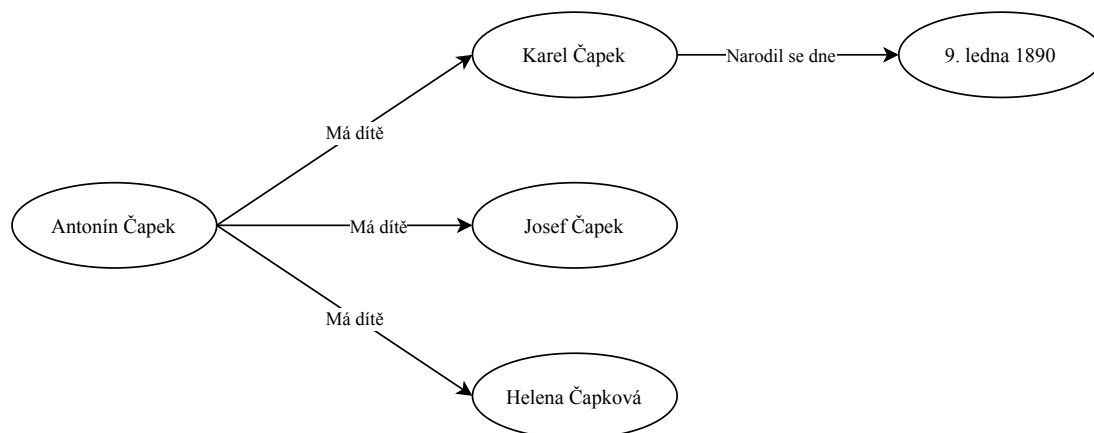
Kód popisuje jedno tvrzení, kdy se Karlu Čapkovi <https://www.wikidata.org/wiki/Q155855> přiřadí přes property datum narození <https://www.wikidata.org/wiki/Property:P569> literál s jeho datem narození, jež má typ <http://www.w3.org/2001/XMLSchema#dateTime>.

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.

```
@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

Jakýk SPARQL<sup>2</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. Výhoda tohoto typu dotazu je ta, pokud neznáme strukturu grafu a chceme tedy získat „nějaké“ informace o vrcholu.

---

<sup>2</sup><https://www.w3.org/TR/sparql11-overview/>

**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.

## 2. Analýza požadavků

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

Jak již bylo zmíněno v úvodu, cílem práce je vyrobit webovou aplikaci, jež komunikuje se serverem a je schopna číst konfigurace na základě kterých pak vizualizuje grafová data. Konfigurace a serverová část byla napsána mým vedoucím a tedy sloužila jako specifikace systémových požadavků na klientskou část. V následující podkapitole nejprve popíšu strukturu konfigurace.

### 2.1 Konfigurace

#### 2.1.1 Rozšíření požadavků

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 mým vedoucím jsem se rozhodl konfigurace rozšířit o tyto parametry (název, počáteční vrcholy) a přidat meta konfigurace, jež slouží jako složky pro konfigurace. Níže je tedy popsán již nový návrh konfigurací.

V následujících sekcích jsou použity tyto RDF namespaces:

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

#### 2.1.2 Meta konfigurace

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

- **dct:title** Název metakonfigurace (*je možné zadat ve více jazycích*)
- **dct:description** Detailnější popis, co daná metakonfigurace obsahuje (*je možné zadat ve více jazycích*)
- **browser:image** Odkaz na URL soubor s obrázkem reprezentujícím 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 metakonfiguraci. (*je očekáváno více objektů; popsáno dále*)

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

### 2.1.3 Konfigurace

Konfigurace popisuje, jak by měla aplikace procházet datasety. 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 uzly v aplikaci vypadat. *(popsáno dále)*
- **browser:startingNode** Doporučený uzel nebo uzly, kde 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 ještě než se pošle požadavek.
- **browser:hasViewSet** View sety. *(popsáno dále)*
- **browser:autocomplete** JSON soubor se seznamem RDF uzlů podle kterých probíhá hledání. *(je očekáváno více objektů; popsáno dále)*

### 2.1.4 ViewSet

View set reprezentuje skupinu pohledů. Právý smysl pohledů pochopí čtenář dále v textu. 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 uzel.
- **browser:hasDataset** Dataset vůči kterému probíhá ASK dotaz. *(popsáno dále)*

### 2.1.5 View

Konkrétní pohled na uzel jež byl vysvětlen v úvodu. 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** Odkaz na expanzi - určuje jaké uzly lze získat z daného uzlu (*popsáno dále*)
- **browser:hasPreview** Odkaz na preview - určuje jaká data se mají získat pro ostylování konkrétního uzlu (*popsáno dále*)
- **browser:hasDetail** Odkaz na detail - určuje která data se zobrazí v detailu konkrétního uzlu (*popsáno dále*)

### 2.1.6 Expansion

Expansion popisuje jak lze daný uzel expandovat, tedy jedná se o operaci kdy se stahují nové uzly jež jsou nějak příbuzné expandovanému uzlu. Expanze vrací graf, tedy expandované uzly nemusí být přímými sousedy expandovaného. 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 SELECT dotaz který bude spuštěn na endpointu datasetu.

### 2.1.7 Preview

Preview popisuje která data v rámci daného pohledu (view) popisují daný uzel. Popisem se myslí taková data, která mají na svědomí stylování uzlu. V aplikaci je možné mít uzly různých barev a tvarů, to právě popisuje preview. Obdobně jako expanze, preview patří ke konkrétnímu pohledu.

Preview má stejné vlastnosti jako expanze. **browser:query** v tomto případě popisuje SPARQL dotaz, který vrátí graf obsahující daný uzel a jeho literály a z těchto literálů bude sestaven preview.

Konkrétně pro preview je predikát **browser:class** považován na třídy uzlu a ty jsou nastaveny jako třídy v knihovně Cytoscape<sup>1</sup>, která je využívána v klientské části na kreslení grafů.

### 2.1.8 Detail

Detail je poslední z trojice a poskytuje dodatečné informace k uzlu. Může se jednat o literály které nemá smysl v dané konfiguraci vykreslit do grafu jako uzly, proto budou zobrazeny v bočním panelu aplikace.

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

---

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

### 2.1.9 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 SPARQL dotazy
- `browser:accept` Popisuje HTTP Accept type v jakém formátu by měl endpoint svá data poskytnout. **TODO: nemělo by to čistě záležet na serveru?**

### 2.1.10 Visual style sheet

Popisuje jakž styl mají mít uzly v dané konfiguraci. Popis je založen na datech pocházejících právě z **preview**. Forma zápisu stylů aktuálně odpovídá stylům jaké používá knihovna Cytoscape.

Visual style sheet má pouze jednu vlastnosti

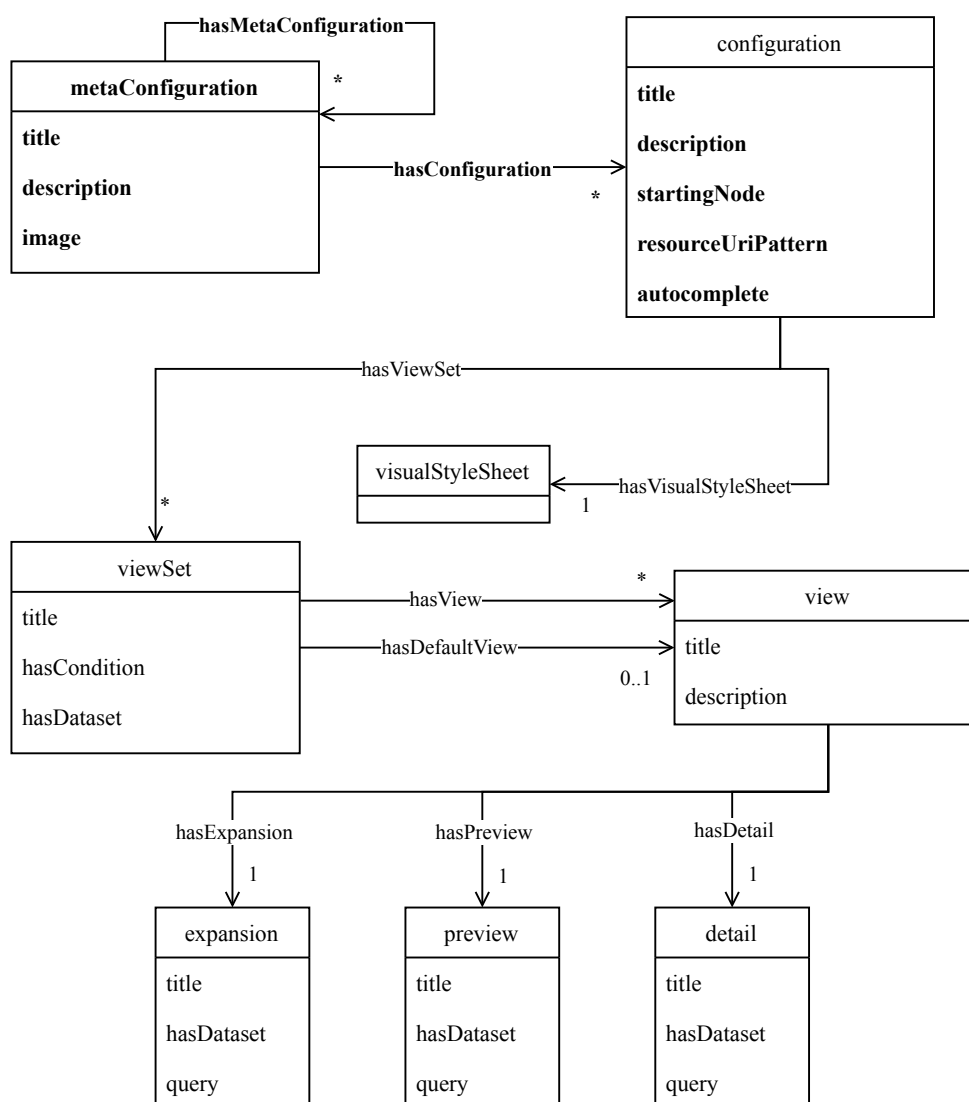
- `browser:hasVisualStyle` Jedno pravidlo jak se má provést stylování, odkazuje na **Visual style**.

#### Visual style

Visual style má pak:

- `browser:hasSelector` Selector pro Cytoscape knihovnu, jež vybírá na které uzly nebo hrany daný styl bude aplikován.
- `browser:*` Jednotlivá pravidla jak má být daný uzel nebo hrana ostylevána. Používají se přesně ty názvy, které používá knihovna Cytoscape, kupříkladu `browser:border-width` nebo `browser:background-color`.





Obrázek 2.1: Class diagram konfigurací. Tučně jsou zvýrazněné části jež nebyli součástí původní specifikace.

## 2.2 Server

Funkcionálním požadavkem bylo, aby 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.

### 2.3.1 Konfigurace a stylesheet

Uživatel musí být schopen vybrat IRI konfigurace a 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. Není tedy možné mít dvě konfigurace v jednom grafu a změnou konfigurace tedy dojde ke smazání původního grafu.

### 2.3.2 Vložení vrcholů

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

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

- 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 bude viditelně zvýrazněn. 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.

### 2.3.3 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.

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

### 2.3.4 Skrývání uzlů

Uživatel může skrývat uzly v panelu s detailem. Skrytý uzel se v grafu skryje společně se všemi hranami a bude možné zobrazit seznam skrytých uzlů ze kterého bude možné 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.

### 2.3.5 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 byla provedena.

**Poznámka** Aktuálně se expanze ukládají k vrcholům a je tedy možné zjistit, které vrcholy vznikly ze které expanze bez nutnosti znovu expanzi volat. Bohužel tato funkcionality ještě není implementována a je řešena v poslední kapitole.

### 2.3.6 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.

### 2.3.7 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 nový jazyk. Podrobněji je toto rozebráno v kapitole s implementací.

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 může být procházení měst v Japonsku anglicky mluvícím uživatelem, kdy je žádoucí, aby názvy měst byly v originále.

### 2.3.8 Stažení grafu do souboru

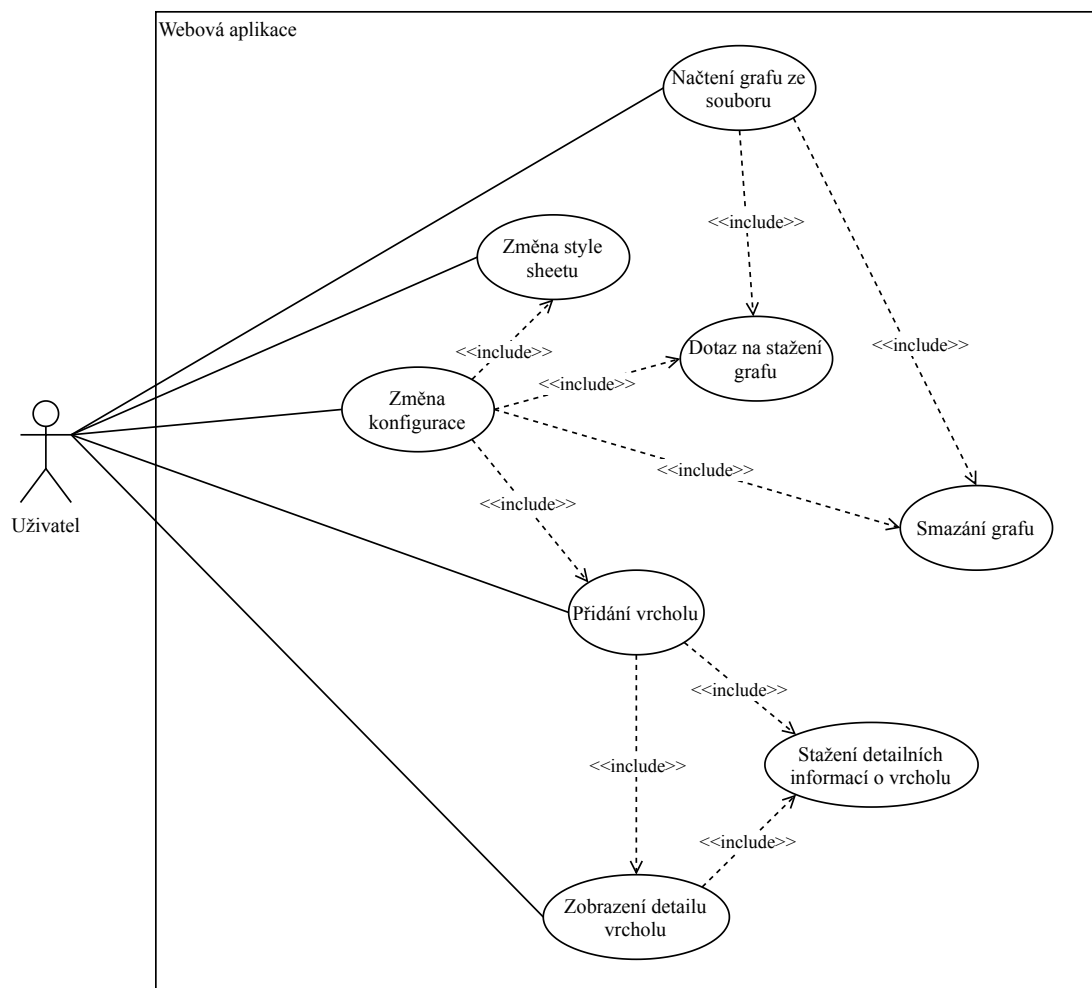
Uživatel má možnost stáhnout aktuální graf do souboru a později ho ze souboru načíst zpět.

**Analýza** Protože je aplikace ve vývoji, je třeba dbát na zpětnou kompatibilitu. Při každé nové verzi je tedy třeba ověřit, že 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. Toto je opět řešeno v poslední kapitole.

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.



Obrázek 2.2: Use case diagram vytváření grafu a základní práce s grafem.

### 2.3.9 Odstranění vrcholu

Uživatel může z grafu vrchol odstranit.

**Analýza** Odstraněním vrcholu se musí odstranit i všechny hrany patřící vrcholu.

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

### 2.3.10 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.

### 2.3.11 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 nebo pravým kliknutím myši. 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.

### 2.3.12 Seskupování vrcholů

Vrcholy bude možné seskupovat do skupin které budou ve vizuálním grafu reprezentovány jedním uzlem. 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 odstraněny. Obdobně toto platí i mezi dvěma skupinami. Pokud je ve skupině vrchol skrytý (ať uživatelem, 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.

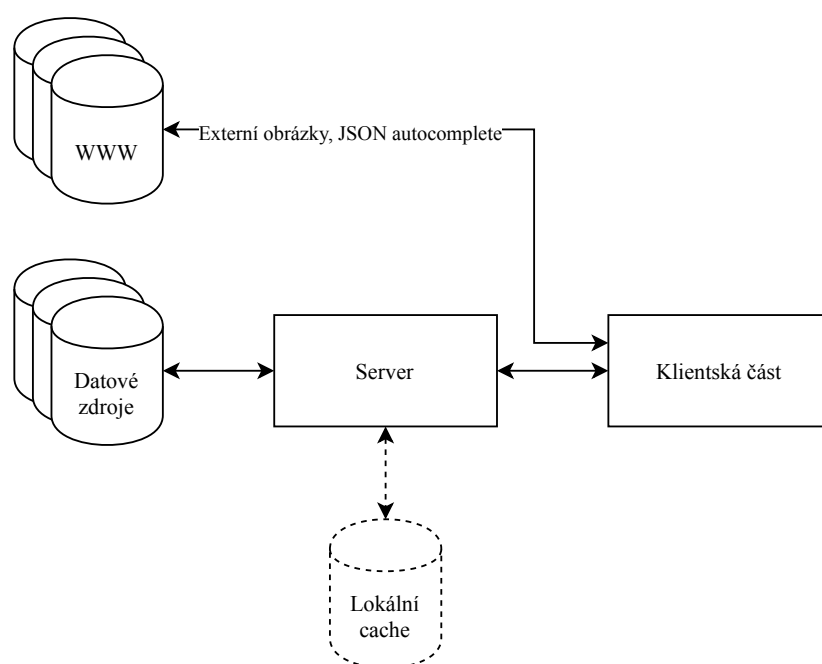


## 3. Návrh architektury

Bylo zamýšleno, že klientská aplikace bude komunikovat se serverovou částí, která bude stahovat data z různých datových zdrojů a výsledky posílat zpět klientské aplikaci. Ta v případě potřeby bude stahovat přímo z internetu obrázky, pokud na ně uzly odkazují. Idea rozdělení klienta a serveru byla v tom, že v budoucnu může být na serveru implementováno chachování, aby aplikace byla plynulejší a neprobíhalo zatěžování datových zdrojů.<sup>1</sup>

Server tak slouží na odstínění od link datové vrstvy a v budoucnu pro chachování dat.

Chachování aktuálně implementováno ještě není, server je tedy kompletně bezstavový.



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

### 3.1 Server

Jak již bylo zmíněno dříve, převážná část serveru byla napsána mým vedoucím Martinem Nečaským a sloužila i jako technická specifikace pro klientsou část. Server je napsaný v JavaScriptu a běží v Node.js<sup>2</sup>. Server je bezstavový a odpovídá na požadavky zmíněné dále.

<sup>1</sup>Při příliš rychlém stahování z Wikidat lze bohužel snadno dostat IP ban, tedy alespoň pro Wikipedii je cachování nezbytné.

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

### 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 jejíž 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.

**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.

#### **/metaconfiguration**

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

Vrátí informace o metakonfiguraci zadané podle `iri`.

Vrátí všechna data (viz kapitola 2.1.2) o metakonfiguraci, veškerá data o dceřiných konfiguracích (viz kapitola 2.1.3) a základní data o dceřiných metakonfiguracích (vše kromě seznamu konfigurací a metakonfigurací).

```
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 `/metaconfiguration` o svých sceřiných konfiguracích.

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

```
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 2.1.10) 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 2.1.4) 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 4.2.10) 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 4.2.10) 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 2.1.6) 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 Klient

### 3.2.1 Moduly

#### Připojení na server

Tento modul řeší komunikaci se serverem.

#### Graf

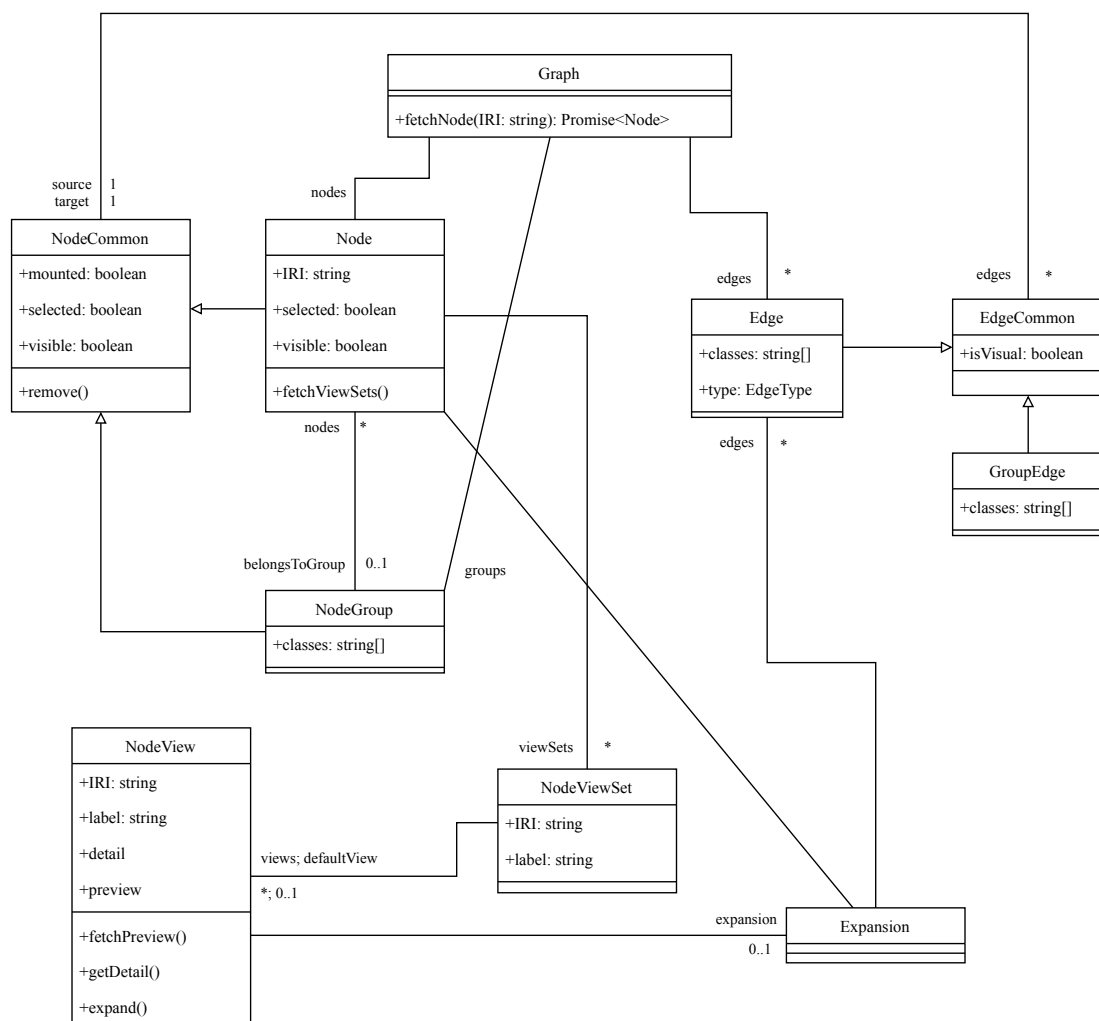
Tato část aplikace reprezentuje stažený graf a všechny jeho data. Má metody pro práci s grafem, jeho modifikaci a stahování nových dat ze serveru.

#### Filtrování

Tento modul integruje možnost filtrování vrcholů v aplikaci na základě různých grafových a sémantických vlastností. Modul se dá rozšířit o další filtry i za runtime a nabízí tedy možnost instalace pluginů.

#### Layoutování

Modul řeší uspořádání vrcholů v grafu. Layoutování reaguje na významné události z grafu. Jedná se například o vložení nového vrcholu, expanzi, přesunutí vrcholů, zamčení vrcholů atp. Stejně jako filtrování, jednotlivé layouty lze přidávat za runtime a je tedy možné s pomocí pluginů přidat nové layouty.



Obrázek 3.2: Class diagram části aplikace jež pracuje s grafem.

## **Vyhledávání**

Modul vyhledávání vrací vrcholy na základě textového řetězce. Vyhledává v grafu, z autocomplete nebo se pokusí IRI sestavit z hledaného výrazu. Jednotlivé vyhledávače lze rozšířit o další stejně jako u filtrování a layoutování.

## **Ukládání do souboru**

Řeší jak uložit a znova načíst stav aplikace do a ze souboru.

## 4. Implementace

### 4.1 Vue.js framework

Klientská část aplikace je postavena nad Vue.js frameworkem, 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>1</sup> (Facebook) nebo Angular<sup>2</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>3</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 takovouto hodnotu se v aplikaci mohu ptát libovolně krát, ale počítá se pouze tehdy, když se množina sousedních uzlů vrcholu změní, nebo se změní právě označení uzlu z množiny.

---

<sup>1</sup><https://reactjs.org/>

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

<sup>3</sup><https://vuex.vuejs.org/>



## 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 **oklonováním 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.

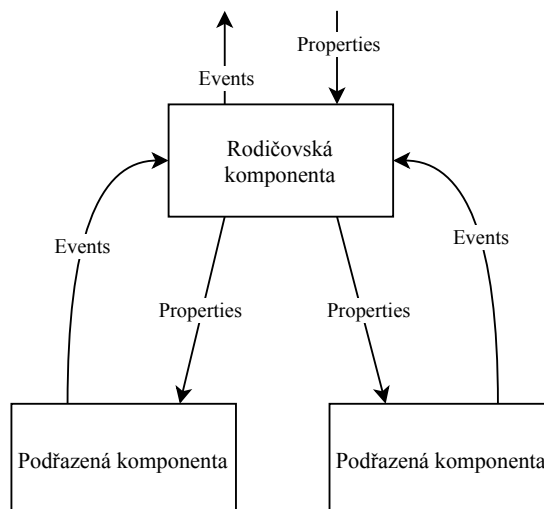
### 4.1.2 Vue framework

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 vyrobiť pod sebou další komponenty a předat jim část dat, která dostala.

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 pseudoselektor `::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 Programátorská dokumentace

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

### 4.2.1 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.

- `$t_literal(translations): string|undefined` - Očekává objekt popsáný v kapitole 3.1.1 a vybere z něj nejvhodnější překlad podle pravidel popsáných ve zmíněné kapitole. Vrací `undefined` pokud žádný překlad není.
- `$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.

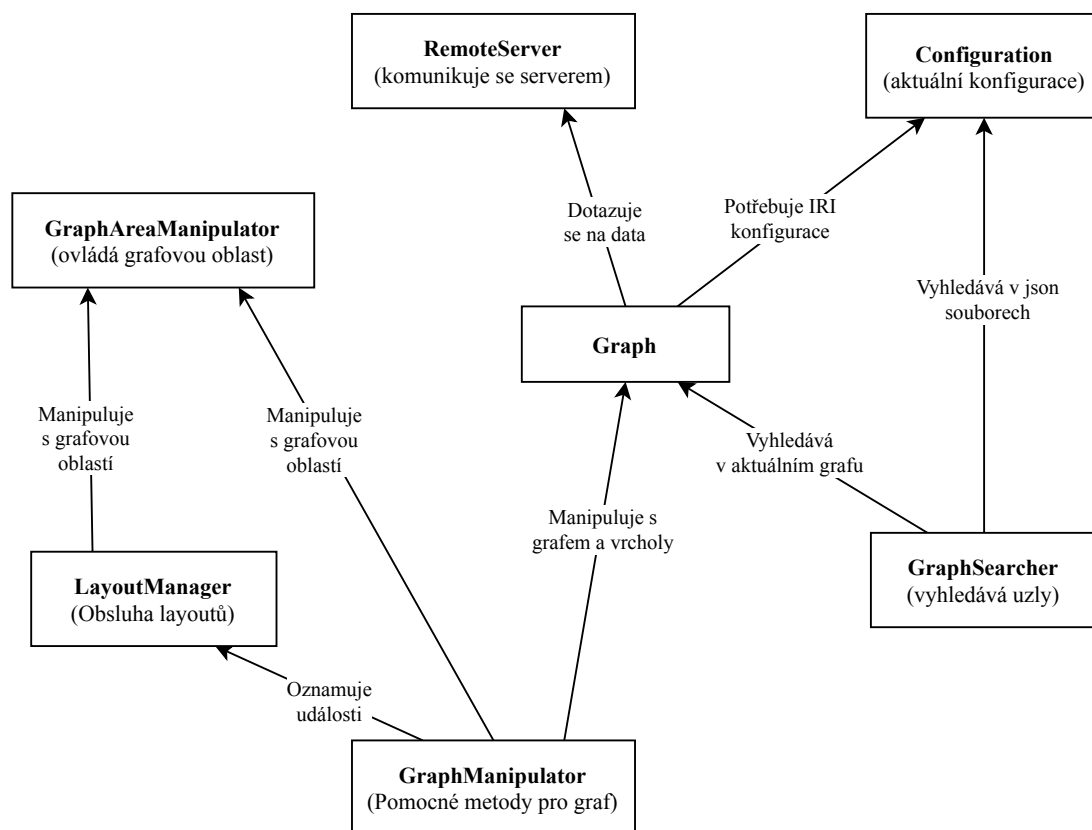
### 4.2.2 Komponenta Application

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.kgwb` 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 2.1.3.
- `graph: Graph` - Aktuální graf jež závisí na `configuration`.
- `areaManipulator: GraphAreaManipulator` - Třída která obsahuje metody pro práci s grafovou oblastí jako zoomování, 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`.



Obrázek 4.2: Závislost některých modulů aplikace na sobě

- **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 metakonfigurace ze serveru.
- **visualStyleSheet: ResponseStylesheet** - Aktuální stylesheet 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 stylesheet 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()` - Vyrobit soubor se stavem aplikace který je možné stáhnout.

## 4.2.3 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 oklonovány, 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.

#### 4.2.4 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.
- `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`.

### 4.2.5 Třída NodeCommon

Třída `NodeCommon` je společný předek pro třídy `Node` a `NodeGroup` a zaštiťuje 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 vykreslet, 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 skupinu 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 taktovýchto 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 uzel 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)
```

## 4.2.6 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 promisu z prvního stahování. Takto nedojde k zatížení serveru a datových zdrojů. Ukázka metody je zobrazena na obrázku 4.2.6.
- `async useDefaultView(): Promise<NodeView>` - Metoda, v případě že vrchol nemá nastaven pohled, stáhne pohledy a nastaví výchozí.

```

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** 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.

### 4.2.7 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 volí 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.

**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 ke znovupřepočítání 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 chache 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 chachován, tentokrát pomocí computed properties a tedy k přepočítání dojde pouze tehdy, změní-li se graf.

### 4.2.8 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.

### 4.2.9 Třída NodeViewSet

Třída reprezentuje view set popsáný v kapitole 2.1.4. 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(URI: 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`.

#### 4.2.10 Třída `NodeView`

Tato třída odpovídá pohledům tak, jak jsou definovány v kapitole 2.1.5.

##### Fieldy

- `detail: DetailValue[]` - Představuje pole detailů které lze získat ze serveru voláním `detail` a odpovídají . Interface `DetailValue` pak obsahuje `type`, `URI` 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>` - Proveďte expanzi a vrátí ji.

Všechny tři funkce mají podobnou ochranu proti znovuzavolání jako metoda `fetchViewSets` jejíž ukázka je na obrázku 4.2.6. 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.

#### 4.2.11 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 částech práce bude popsáno, jak je tento modul grafu integrován do Vue frameworku tak, aby se samy vytvářely a updatovaly vrcholy a hrany kdykoli dojde k jejich změně.

### 4.2.12 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 obsluhuje 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 emitu jež je schopna pracovat právě s grafovou oblastí.

#### GraphAreaStylesheetMixin

Komponenta používá `GraphAreaStylesheetMixin` kde je oddělena logika zpracování stylesheetů.

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`. Jamile dojde k jeho změně, zavolá se metoda která předá knihovně (`this.cy`) nový stylesheet. Nesmíme zapomenout objekt oklonovat, protože nemáme zaručeno, že ho knihovna nebude upravovat. Pokud by kupříkladu knihovna objekt vždy upravila, například z optimalizačních důvodů, Vue framework by zachytil změnu stylu a znoa by zavolal tuto funkci a situace by se opakovala.

Komponenta pak ke každému vrcholu, hraně a skupině vytvoří vlastní komponentu 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"

...
/>

```

### 4.2.13 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 komponenty `GraphElementNodeMixin`, `GraphElementNode` a `GraphElementNodeGroup` jež si navzájem odpovídají.

Mezi významné metody patří

- `mount()` - Metoda se volá automaticky když je komponenta vytvořena v této metodě probíhá registrace vrcholu v Cytoscape knihovně. Kromě vytvoření uzlu zde probíhá i registrace důležitých eventů a registrace knihovny Popper<sup>4</sup><https://popper.js.org/> jež má na starosti pozicování elementů vůči různým objektům, zde právě vůči vrcholům na plátně. Tak jsme schopni vykreslit k vrcholům ikonky, pokud jsou vrcholy uzamčeny.
- `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. Opětovným odstraněním uzlu z kontejneru dojde k jeho smazání.

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);
}

```

<sup>4</sup>`\unskip\protect\penalty\M\vrulewidth\z@height\z@depth\dp,`

```
} ...
```

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.



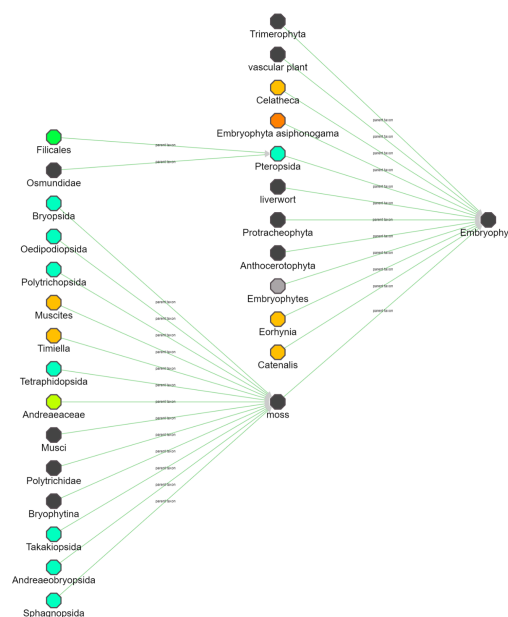
# 5. Uživatelské testování

## 5.1 System Usability Scale (SUS)

System Usability Scale 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.



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“.
5. Bez aktualizace stránky změňte konfiguraci na „Poslanecká sněmovna parlamentu České republiky“ a vyberte libovolný uzel.

# Závěr

Anděl (2007, str.29)

# Seznam použité literatury

ANDĚL, J. (2007). *Základy matematické statistiky*. Druhé opravené vydání. Matfyzpress, Praha. ISBN 80-7378-001-1.

# 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. Tučně jsou zvýrazněné části jež nebyli součástí původní specifikace. . . . .	13
2.2	Use case diagram vytváření grafu a základní práce s grafem. . . .	18
3.1	Komunikace mezi klientem, serverem a datovými zdroji. Cachování na serveru ještě implementováno není. . . . .	21
3.2	Class diagram části aplikace jež pracuje s grafem. . . . .	26
4.1	Doporučený způsob komunikace mezi komponentami ve Vue frameworku . . . . .	30
4.2	Závislost některých modulů aplikace na sobě . . . . .	33
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. . . . .	39

# Seznam tabulek

# Seznam použitých zkratek

## A. Přílohy

### A.1 První příloha