FAKULTA
INFORMAČNÍCH
TECHNOLOGIÍ
ČVUT V PRAZE

ZADÁNÍ DIPLOMOVÉ PRÁCE

Název:	Webová aplikace pro podporu práce se sémantickými daty
Student:	Bc. Michal Kopecký
Vedoucí:	Ing. Jaroslav Kuchař, Ph.D.
Studijní program:	Informatika
Studijní obor:	Webové a softwarové inženýrství
Katedra:	Katedra softwarového inženýrství
Platnost zadání:	Do konce letního semestru 2018/19

Pokyny pro vypracování

Cílem této práce je analyzovat současnou situaci, navrhnout a implementovat webovou aplikaci umožňující správu obsahu v podobě sémantických dat. Tato data umožní i ve vhodných formátech prezentovat/publikovat a zapojit je tak do současného webu dat.

1. Seznamte se se stávajícími řešeními pro vytváření strukturovaných dat v nejrozšířenějších redakčních systémech jako je Wordpress a jemu podobné. Zmapujte ale také i méně známé CMS, které se na tvorbu sémantických dat specializují.
2. Na základě analýzy navrhněte vlastní řešení, které by mělo zjištěné nedostatky odbourat. Zejména se zaměřte na:
   a. tvorbu více specifických datových entit
   b. možnosti modelování ontologie
   c. podporu více datových formátů na výstupu
3. Řešení implementujte jako vhodné rozšíření některého z existujících řešení.
4. Implementované řešení podrobte uživatelskému testování.
5. Zhodnoťte jeho přínos pro podporu tvorby a publikace sémantických dat a diskutujte další možný rozvoj.

Seznam odborné literatury

Dodá vedoucí práce.

Ing. Michal Valenta, Ph.D.
vedoucí katedry

doc. RNDr. Ing. Marcel Jiřina, Ph.D.
děkan

V Praze dne 10. února 2018

FAKULTA
INFORMAČNÍCH
TECHNOLOGIÍ
ČVUT V PRAZE

Diplomová práce

Webová aplikace pro podporu práce se sémantickými daty

Bc. Michal Kopecký

Katedra Softwarového inženýrství
Vedoucí práce: Ing. Jaroslav Kuchař, Ph.D.

3. května 2018

Poděkování

Rád bych na prvním místě poděkoval panu Ing. Jaroslavu Kuchařovi, Ph.D., který byl ochoten se ujmout mého vlastního tématu a pomohl mi s jeho formulací. Také věnoval svůj čas konzultacím, na kterých mě posunul správným směrem při řešení některých problémů, jež se při řešení práce vyskytly. Tím celkově přispěl k zdárnému dokončení práce.

Dále bych rád poděkoval Ing. Radimu Haškovi, Bc. Marku Hanušovi, Bc. Matěji Fantovi a Bc. Lubomíru Koblásovi, kteří otestovali praktickou část této práce a pomohli mi tak s jejím vyladěním.

V neposlední řadě bych rád poděkoval své rodině a přítelkyni, že naslouchaly problémům spojeným s tvorbou mojí práce a byly mi v průběhu jejího tvoření velkou oporou.

Prohlášení

Prohlašuji, že jsem předloženou práci vypracoval(a) samostatně a že jsem uvedl(a) veškeré použité informační zdroje v souladu s Metodickým pokynem o etické přípravě vysokoškolských závěrečných prací.

Beru na vědomí, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., autorského zákona, ve znění pozdějších předpisů. V souladu s ust. § 46 odst. 6 tohoto zákona tímto uděluji nevýhradní oprávnění (licenci) k užití této mojí práce, a to včetně všech počítačových programů, jež jsou její součástí či přílohou a veškeré jejich dokumentace (dále souhrnně jen „Dílo“), a to všem osobám, které si přejí Dílo užít. Tyto osoby jsou oprávněny Dílo užít jakýmkoli způsobem, který nesnižuje hodnotu Díla, avšak pouze k nevýdělečným účelům. Toto oprávnění je časově, teritoriálně i množstevně neomezené.

V Praze dne 3. května 2018 .....

Odkaz na tuto práci

Kopecký, Michal. *Webová aplikace pro podporu práce se sémantickými daty.* Diplomová práce. Praha: České vysoké učení technické v Praze, Fakulta informačních technologií, 2018.

Abstrakt

Tato práce se zabývá analýzou stávajících řešení pro tvorbu sémenatických dat v redakčních systémech. Dále pak navrhuje vlastní řešení, které by mohlo proces tvorby strukturovaných dat v redakčních systémech usnadnit a zpřístupnit i širší skupině uživatelů, kteří o nich nemají takový přehled.

Tato vylepšení poté implementuje formou vhodného rozšíření CMS Strapi. Do něj přidává grafický nástroj pro modelování ontologií založený na slovníku Schema.org. Zároveň v systému umožňuje sémantická data vytvářet a také publikovat.

V závěrečné fázi toto rozšíření podrobuje uživatelskému testování, které ověřuje, zda-li došlo ke kýženému výsledku zjenodušení tvorby sémantických dat a nastiňuje jeho další možný rozvoj.

Klíčová slova web dat, tvorba sémantických dat, sémantické CMS, JSON-LD API, Strapi

Abstract

This thesis deals with analysis of existing solutions for semantic data creation in content management systems. It also proposes its own solution that could make the process of creating structured data in CMS easier and more accessible to a wider group of users who do not have such an overview about semantic web.

These improvements are then implemented in the form of an appropriate extension of CMS Strapi. It adds a graphical tool for ontology modeling based on the Schema.org dictionary, and also allowing semantic data to create and publish.

In the final phase, this extension is subject of user testing, which verifies whether the result made semantic data creation easier and outlines its further possible development.

Keywords web of data, semantic data creation, semantic CMS, JSON-LD API, Strapi

Obsah

Seznam obrázků

Úvod

O webu dat se častokrát hovoří v souvislosti s budoucím směřováním vývoje internetu. S tímto pojmem se pojí i výskyt sémantických dat na webu. I přes již několikaletou existenci specifikací však spousta webů neobsahuje strukturovaná data a nedošlo zatím k jejich masivnějšímu rozšíření. Jedny z faktorů brzdících jejich větší rozvoj můžou být i uživatelská nepřívětivost tvorby a publikace samotných dat.

Cílem práce je nejprve tyto možné problémy a nedostatky tvorby sémantických dat v stávajících redakčních systémech analyzovat. Hlavně se zaměřuje jednak na pluginy, které slouží pro automatické generování strukturovaných dat v nejrozšířenějších redakčních systémech jako je Wordpress a jemu podobné, ale také monituruje CMS, které se na tvorbu strukturovaných dat přímo specializují. Výstupem této části je zhodnocení výhod a nevýhod obou přístupů a jejich srovnání.

Na základě této provedené analýzy v další části práce navrhuji vlastní řešení a postupy, které by měly zpřístupnit snadnější, ale zároveň i komplexnější vytváření sémantických dat i pro běžné uživatele, neznalé všech standardů, které se se sémantickými daty pojí.

Tato vylepšení poté implementuji formou vhodného rozšíření CMS Strapi, do kterého přidávám nástroj umožňující modelování sémantických ontologií na základě slovníku Schema.org. Zároveň ho také rozšiřuji o funkcionality pro správu a publikaci sémantických dat.

Implementované řešení poté podrobuji uživatelskému testování, kterým se snažím ověřit, zda-li se mi skutečně povedlo proces tvorby sémantických dat do systému integrovat a udělat ho pro uživatele srozumitelným.

V závěrečné fázi zhodnocuji přínos práce a diskutuji další možný rozvoj a nové funkce, kterými by bylo řešení možné ještě vylepšit.

Kapitola 1

Web dat

Následující kapitola je věnována vysvětlení pojmu Web dat, popsání nejdůležitějších technologií sémantického webu, o které se implementační část této práce opírá, a také by měla být objasněním toho, k čemu může jeho větší rozšíření přispět a proč je důležité se o toto téma zajímat.

1.1 Motivace k zavedení webu dat

Problémem současného webu je, že na něm vzniká každým dnem velké množství dat, ale tato data často nemají žádnou pevně danou strukturu ani sémantiku, zároveň jsou publikována v mnoha rozličných a i proprietárních formátech. Tato data jsou sice díky internetu veřejně komukoliv na světě přístupná a v těchto datech jsou uložené mnohdy cenné informace, ale právě vyhledávání těchto relevantních informací v tak velkém množstvím dat, které na internetu existuje, není jen výpočetně velmi náročný úkol. Tyto problémy se snaží řešit dnešní vyhledávače, ale i za pomoci strojového učení a různých pokročilých algoritmů je stále velice obtížné, aby programy porozuměly významu dat, která jsou určena primárně pro lidské publikum. Rozlišení relevantního obsahu na položený dotaz od toho nerelevantního je proto stále obtížná úloha.

Další motivací k zavedení standardů webu dat do praxe může být také zajištění lepší inteoperability mezi jednotlivými programy. V dnešní době asi nejpopulárnější REST API je sice dobrým nástrojem pro datovou výměnu, ale jeho použití může být ještě lepším, pokud pomocí něj budeme poskytovat data ve strukturované podobě. V té chvíli můžou být data jím poskytovaná samopopisná a nemusíme k němu například vytvářet dokumentaci s popisem významu jednotlivých položek.

1.2 Co to je web dat

Pojem Web dat nemůžeme chápat jen jako nějakou novou technologii, kterou jednoduše nasadíme do provozu. Web dat musíme chápat jako proces transformace stávajícího webu jakožto distribuovaného souborového systému neuspořádaných dokumentů na distribuovanou databázi znalostí. [1]

Tento proces se snaží zavést sadu technologií a specifikací, které by umožnily vnést do dat na internetu pořádek. Přiřadily jim konkrétní sémantiku čitelnou i pro stroje, a tak zpřístupnily jejich porozumění i pro různé programy. Zároveň by tím mezi nimi zajistily i lepší interoperabilitu a docílily tak celkového propojení dat vyskytujících se na internetu.

1.2.1 Web dat, semánatický web a linked data

Ve spojitosti s pojmem web dat asi nejčastěji uslyšíte také pojem sémantický web, který jsem záměrně také hned na začátku použil. Tyto pojmy jsou podle mě do určité míry zaměnitelné, ale zároveň v nich spatřuji drobné odlišnosti.

Na Web dat pohlížím jako na pojem zastřešující celou tuto oblast, který svým významem vyzařuje nutnost změny pohledu na web ne jen jako změť HTML dokumentů, ale jakožto jednu velkou databázi dat.

Oproti tomu historicky starší pojem Sémantický web je pro mě hlavně označením pro sadu technologií, které umožňují data na internetu publikovat v strukturované, chcete-li sémantické podobě.

Pojem Linked data pro mě pak znamená označení způsobu, jakým by data na webu měla být publikována, tedy ve strukturované podobě a ta související by měla být mezi sebou provázána pomocí odkazů, aby celý web tvořil nejlépe jednu velkou souvislou databázi. [2]

1.3 Historie a vývoj

Už v roce 2000 přišel Tim Berners-Lee s myšlenkou sémantického webu. Upozorňoval na to, že web je jen stále větší změť webových stránek, mezi kterými je stále těžší najít relevantní informace. A řekl také, že v kontextu sémantického webu slovo sémantický znamená strojově zpracovatelný, a že se tedy nemá jednat o sémantiku přirozeného jazyka. [3]

Dnes je myšlenka sémantického webu stará více než 18 let, a dalo by se tedy hovořit o tom, že je už dospělá. Za dobu její existence vzniklo mnoho technologií a specifikací, díky nimž se má aplikovat tato myšlenka v praxi. Zmiňované technologie i specifikace se však neustále ještě vyvíjejí a zdokonalují. Co se však týče jejich uplatňování, zdá se, že jsme skoro teprve na samotném začátku.

Jak jsem již zmiňoval, jedná se o proces transformace celého webu, který je největším zdrojem informací na světě a dat na něm neustále exponenciálně přibývá. Není tedy reálné si myslet, že k tomu může dojít ze dne na den.

Dokonce se opovažuji tvrdit, že tento proces nebude nikdy dokončen, protože dokud budou vznikat nová, neuspořádaná data, nemůžeme hovořit o jeho konci.

Cílem tedy není transformovat celý web, ale co největší jeho část a vzhledem k tomu, že data na webu přibývají exponenciálně, nemusíme se dokonce ani pokoušet transformovat už ta stávající. Teoreticky by úplně stačilo, aby už jen nově vznikající data byla všechna sémantická. A právě k tomuto bych chtěl napomoci i svou diplomovou prací.

1.4 Technologie a specifikace webu dat

Web of data je pojem, který zaštiťuje celou řadu technologií a specifikací, které ho vymezují. O tyto specifikace se povětšinou stará konsorcium W3C a jednou z těchto stěžejních specifikací je právě i RDF.

1.4.1 RDF

Resource Description Framework je standardní datový model pro výměnu dat na webu, doporučovaný konsorciem W3C. RDF rozšiřuje odkazovou strukturu webu a používá URI identifikátory k pojmenování vztahů mezi entitami a také entit samotných. Celý datový model je založen na tzv. trojicích: subjekt - predikát - objekt. Pomocí tohoto jednoduchého modelu umožňuje strukturované a polostrukturované údaje kombinovat, vystavovat a sdílet mezi různými aplikacemi. [4]

Subjekt v tomto modelu představuje entitu reprezentovanou URI. Podobně objekt může být buď další entitou také definovanou vlastní URI, nebo literálem. To je jednoduchý datový typ jakožto string, date, number atd. Predikát je pak vyjádření vztahu mezi subjektem a objektem, který je také reprezentován vlastní URI, která pevně definuje jeho význam.

Výhodou tohoto modelu je, že ke svému aplikování nepotřebuje zavádět žádnou novou technologii co se síťové úrovně webu týče a není ani potřeba žádné rozšiřování HTTP protokolu. Jeho aplikování do praxe by tedy mělo být tímto ulehčeno.

RDF je však pouze datový model, jeho konkrétní reprezentace je určena datovým formátem, často se také v tomto významu hovoří o serializaci. V dnešní době existuje už celá řada těchto formátů. Každý z nich má svoje specifika, výhody a oblasti vhodného použití. Mělo by však platit, že jednotlivé datové formáty jdou mezi sebou převést, protože se jedná jen o jiný způsob reprezentace RDF grafu. Mezi nejvýznamnější z nich patří Turtle, N-triples, JSON-LD či RDFa.

Nad tímto datovým modelem je také možné se dotazovat, což je jednou z hlavních výhod sémantických dat. Jako standardní dotazovací jazyk W3C doporučuje používat SPARQL, jehož syntaxe částečně vychází z SQL, kterou všichni vývojáři důvěrně znají.

1.4.2 JSON-LD

Jedním z nejvýznamnějších formátů pro serializaci RDF je i JSON-LD. Jedná se o datový formát založený na obyčejném JSON a každý JSON-LD dokument je zároveň validním JSON dokumentem. To s sebou přináší celou řadu výhod. Například řada JSON parserů může být využita i pro parsování JSON-LD dokumentů. Zároveň je díky populárnosti JSON mezi vývojáři snadno adaptovatelný. V neposlední řadě se v porovnání s ostatními formáty jedná o velice úspornou reprezentaci, co se týče datové velikosti. V poslední době ho také velmi propaguje Google. Budu se mu zde jako jedinému z formátů podrobněji věnovat, protože ho hojně využívám v praktické části této práce a je pro ni stěžejní.

JSON-LD se oproti standardnímu JSON hlavně liší zavedením takzvaného kontextu a dalších rozšiřujících vlastností. Všechny tyto vlastnosti se označují prefixem @, a je tedy hned na první pohled patrné, které klíče nesou speciální význam. JSON-LD definuje několik desítek významových vlastností, v mé práci však používám jen některé z nich, jejichž význam zde proto vysvětlím. Nejvýznamnější z nich jsou *@context* a *@type* [1].

1.4.2.1 Kontext

Když spolu dva lidé komunikují, konverzace se odehrává ve společném prostředí, které se nazývá "kontext konverzace". Tento sdílený kontext umožňuje používat různé zkratky, jako například používání pouze křestního jména společného přítele a konverzace tak probíhá rychleji, aniž by byla ztracena nějaká informace. Kontext v JSON-LD funguje obdobným způsobem. Umožňuje používání zkratek při komunikaci mezi dvěma aplikacemi, aniž by docházelo ke ztrátě jakýchkoli informací. [5]

Kontext se zpravidla uvádí hned jako první vlastnost JSON dokumentu a jeho obsah může být buď uveden přímo v dokumentu, nebo se na něj může odkazovat pomocí URI. Kontextů může být v jednom JSON-LD dokumentu hned několik, v podstatě každý objekt může mít svůj vlastní, jediným omezením je, že kontext nesmí být v objektu přímého potomka kontextu.

Můžeme v něm definovat URI jednotlivých klíčů a nebo můžeme využít syntaxe *@vocab* k definování hlavního prefixu. Ten je použit pro všechny vlastnosti, které obsahují relativní URI a jejichž typ není přímo definován jinde v kontextu.

1.4.2.2 Hodnota s přidruženým typem

Přidružený typ hodnoty je určen zadanou URI, která reprezentuje jednotlivé datové typy. V JSON-LD může být určení tohoto typu zapsáno třemi způsoby,

[1] @type značí sémantický typ entity a jedná definován pomocí URL `http://www.w3.org/1999/02/22-rdf-syntax-ns#type`. Pokud dále v textu budu používat zkratku *@type* myslím tím právě tento typ.

a to:

1. Použitím klíčového slova @type v definování výrazu v sekci *@context*
2. Použitím hodnotového objektu (value object), což je JSON objekt, který obsahuje vlastnost *@value*
3. Použitím nativního datového JSON typu, jako je číslo nebo booleovská hodnota.

Určení tohoto typu je důležité, protože právě díky tomu víme, jakého sémantického typu je daný objekt a jeho subjekty.

1.5 Náklady na vytváření a využití sémantických dat

Uživatel, jenž chce data poskytovat v sémantické podobě, by měl nejprve porozumět datovému modelu RDF, data poté převést do některé z jeho serializací, pak je publikovat a podle definice propojených dat ještě nejlépe také propojit s jinými již na webu existujícími daty.

Je tedy vidět, že takto surová tvorba sémantických dat bohužel znamená pro jejich tvůrce i spoustu práce navíc oproti publikaci dat v nestrukturované podobě. [6]

Avšak i nestrukturovaná data uživatelé na webu většinou vytvářejí pomocí specializovaných nástrojů v podobě různých redakčních systémů, které uživatele odstiňují od porozumění syntaxe HTML atd.

Podobně by proto měli mít uživatelé i možnost vytvářet strukturovaná data pomocí CMS uzpůsobených k tomuto účelu a být tak odstíněni od porozumění RDF modelu a jeho konkrétní reprezentace. I tak ovšem vytváření sémantických dat představuje složitější úlohu, neboť je potřeba data více strukturovat podle jednotlivých položek, těm přiřazovat konkrétní významy, což je právě činnost, kterou zatím neumějí programy zvládat tak precizně, a vyžadují proto lidskou činnost. Podrobněji se tomuto tématu budu věnovat v následující kapitole, kde budu zkoumat již existující nástroje pro vytváření strukturovaných dat.

Práce navíc, kterou vytváření sémantických dat obnáší, pak musí být vykoupena nějakou přidanou hodnotou, kterou uživatel získá jako odměnu za svoji snahu, neboť bez toho by nebyl dostatečně motivován a k rozšíření webu dat by s těží někdy došlo.

1.5.1 Využití sémantických dat

Jedním z hlavních přínosů sémantických dat a tak i jejich možného využití je porozumění významu v nich uložených i pro různé algoritmy. Jak jsem již zmiňoval, právě náročnost, až skoro nemožnost porozumění přesnému významu

dat je dnes velkým problémem a právě metadata obsažená v sémantických datech jsou jeho řešením. Toto ale není primárně věc, ze které může těžit tvůrce oněch dat. Z tohoto faktu těží hlavně ostatní uživatelé, kteří chtějí data takto publikovaná využívat. Pokud ale právě o to tvůrci dat jde, aby se jeho data snadno a hojně používala, jedná se o sekundární efekt, který může posloužit jako motivace k jejich publikování ve strukturované podobě.

Pro tvůrce sémantických dat může být také motivací to, že pokud sémantická data vystaví pomocí API ve strukturované podobě, může je sám snadněji využít v rozličných aplikacích. Jeden centrální zdroj dat tak může posloužit například jak pro webovou stránku, tak i pro mobilní aplikaci.

V neposlední řadě může být také motivací pro publikování sémantických dat na webu jejich využívání pomocí vyhledávačů a zvýhodňování těchto webů ve výsledcích vyhledávání. To může na stránky přilákat více potencionálních zákazníků a nebo tvůrci přinést více financí plynoucích například z reklamy na nich zobrazovaných.

Kapitola 2

Analýza stávajících systémů pro tvorbu sémantických dat

V této kapitole se budu věnovat existujícím řešením pro tvorbu strukturovaných dat. V první části se podívám na rozšíření pro stávající CMS systémy, které původně nevznikly pro tvorbu a práci se sémantickými daty, ale byly k tomuto účelu rozšířeny. V druhé části se pak zaměřím na specializované redakční systémy, které za účelem tvorby strukturovaných dat přímo vznikly a nejsou tak ničím limitovány, ale také nejsou tolik rozšířeny. V závěrečné fázi kapitoly oba tyto postupy zhodnotím a vyzdvihnu jejich klady a zápory.

2.1 Analýza tradičních CMS systémů

Content management systémy začaly vznikat už v pozdních 90. letech minulého století za účelem toho, aby usnadnily uživatelům správu struktury webu, medií a v neposlední řadě tvorbu obsahu stránek. Myšlenka sémantických dat v té době ještě neexistovala a tyto systémy se tak zaměřovaly hlavně na publikaci dat jakožto HTML dokumentů, a proto je i jejich struktura navržena k tomuto účelu.

Takovýchto systémů existují stovky a ve své analýze nejsem určitě schopen zmapovat všechna možná rozšíření pro všechny tyto redakční systémy, a proto bych se rád zaměřil jen na tři nejrozšířenější. V dnešní době (dle dat za rok 2017) mezi ně patří Wordpress, Joomla a Drupal. Wordpress se skoro 60 procentním podílem na celkovém trhu a s 26 700 000 aktivními instalacemi (zdroj dat `https://websitesetup.org/popular-cms/`) můžeme bezesporu označit jako nejúspěšnější z nich. Joomla je pak dvojkou a Drupal trojkou na trhu. Zajímavé je, že všechny tři jsou psané v PHP a zároveň se jedná o open-source. Právě tento fakt nejspíše přispěl k jejich masivnímu rozšíření.

2.1.1 Pluginy pro Wordpress

Wordpress vznikl v roce 2003. V době jeho vzniku tedy už sice Tim Bernsers Lee začal veřejně mluvit o myšlence sémantického webu, ale krom existence pojmu ještě ani pořádně neexistovaly jeho specifikace. Kromě toho Wordpress původně vznikl jen jako jednoduchý sytém pro publikaci článku ve formě blogu a jeho autoři zřejmě vůbec netušili, že se jednou stane tak populárním redakčním systémem a bude využíván i na mnohem komplexnější stránky.

Worpdress je open-source software a umožňuje také rozšíření jeho funkcionalit pomocí pluginů. Vzhledem k tomu, že na tomto redakčním systému běží více než polovina všech webů, zdařilá implementace podpory sémantických dat by mohla významně přispět k celkovému rozšíření webu dat. Bohužel jak už jsem zmiňoval, systém k tomu není původně navržen, tudíž se tato rozšíření musí potýkat s celou řadou návrhových omezení, která jim tento systém přináší. I přes tyto problémy vzniklo hned několik rozšíření, která se zaobírají přidáním podpory pro strukturovaná data do tohoto oblíbeného systému. Některá jsou více, jiná méně zdařilá a podporu pro publikaci strukturovaných dat provádí na různých stupních integrace. Ty nejjednodušší z nich zavádí pouze automatickou transformaci existujícího obsahu do RDF a omezují se jen na typy entit jako je stránka a nebo blogový příspěvek. Tím je sice splněna podmínka toho, že data jsou publikována v RDF, ale nic moc navíc se nedozvíme o jejich významu a přínos takovéhoto řešení je tedy minimální. Pokročilejší pluginy umí generovat obsah přímo pro konkrétní datové entity, avšak zase chybí jakékoliv propojení na existující obsah, který se používá pro běžné uživatele. Data jsou tedy redundována.

2.1.1.1 Wp-linked-data

Jedním ze zástupců skupiny pluginů založených na automatické transformaci již existujícího obsahu do RDF je právě i plugin wp-linked-data. Ten využívá obsah stávajících entit, se kterými Wordpress pracuje, tedy stránek a příspěvků a automaticky z nich generuje strukturované entity typu `http://rdfs.org/sioc/types#BlogPost` a `http://rdfs.org/sioc/ns#Item`. Zároveň transformuje údaje o profilu uživatele, který daný příspěvek vytvořil na entitu typu `http://rdfs.org/sioc/ns#UserAccount`.

Možnosti konfigurace tohoto pluginu jsou minimální. V administraci pouze nabízí volbu URI pro profil uživatele, aby si mohl vybrat stránku, kterou chce být reprezentován. Bohužel však už neumožňuje žádnou možnost konfigurace datových typů entit tak, aby mohly lépe odpovídat skutečnému obsahu webu.

RDF také umí reprezentovat jen ve formátu Turtle, přičemž patřičnou verzi dokumentu dostaneme standardním způsobem jako odpověď na požadavek, kde uvedeme hlavičku *Accepted: turtle/text*. Jednotlivé verze stránek tedy koexistují na těch samých URL.

```
@prefix sioct: <http://rdfs.org/sioc/types#> .
@prefix dc: <http://purl.org/dc/terms/> .
@prefix sioc: <http://rdfs.org/sioc/ns#> .
@prefix xsd: <http://www.w3.org/2001/XMLSchema#> .

<http://wordpress-test.local/ahoj-vsichni#it>
a~sioct:BlogPost ;
dc:title "Nadpis prispevku" ;
sioc:content "Text prispevku lorem ispum dolor sit amet." ;
dc:modified "2018-01-29"^^xsd:date ;
dc:created "2018-01-29"^^xsd:date ;
sioc:has_creator <http://wordpress-test.local/author/admin#account> ;
sioc:has_container <http://wordpress-test.local#it> .

<http://wordpress-test.local/author/admin#account>
a~sioc:UserAccount ;
sioc:name "admin" .

<http://wordpress-test.local#it> a~sioct:Weblog .
```

Ukázka kódu 2.1: Ukázka odpovědi pluginu Wp-linked-data

Jeho rozšíření není také nikterak velké a podle webu Wordpressu aktuálně běží jen cca 30 aktivních instalací.

2.1.1.2 Schema

Jako nejúspěšnější plugin pro Wordpress, alespoň co se počtu aktivních instalací týče, je právě Schema. Schema.org je oblíbený sémantický slovník a tento plugin se zaměřuje právě na jeho podporu pro systém Wordpress. Podobně jako předchozí případ však řeší hlavně tvorbu strukturovaných dat z již existujícího obsahu, a to automatickou transformací. Sémantická data pak publikuje ve formátu JSON-LD přímo jako součást standardní odpovědi typu *text/html*.

Na rozdíl od předchozího pluginu však disponuje možnostmi konfigurace. V nastavení informací o organizaci si můžete vybrat ze dvou datových typů a to *schema:Person* [2] nebo *schema:Organization* tak, aby lépe odpovídaly skutečnosti. Pak zde také můžete vyplnit jméno, URL webové stránky a logo.

Na další záložce Korporativní kontakty pak můžete vyplnit telefon a URL adresy různých sociálních sítí. Nastavení těchto údajů je potom využito pro generování Google rich snippetů. Dalším možným nastavením je povolení strukturované syntaxe pro drobečkovou navigaci, komentáře u příspěvků, videa a audio objekty.

U jednotlivých entit (post type) si můžete v nastavení pluginu zvolit, který typ daná entita reprezentuje. Nedostatkem však je, že k dispozici jsou jen pod-

[2] Pokud jste si všimli zkratky schema:* jedná se o označení URL adresy http://schema.org/*, v tomto konkrétním případě se tedy jedná o URL `http://schema.org/Person`. Toto označení budu v práci v souladu s tímto významem hojně používat.

typy typu *schema:Article*. Plugin tedy sice umožňuje zvolit si více specifický typ, přidaná hodnota této volby je ovšem pro výstupní data stejně minimální, protože se jedná vždy jen o nějaký typ článku. U konkrétního příspěvku však můžeme v textovém poli vyplnit URL adresy, pomocí kterých můžeme specifikovat i další typy této entity díky vlastnosti *owl:sameAs*. Dále zde můžeme zaškrtnutím checkboxu vynutit nepoužívání sémantických dat pro jednu konkrétní entitu.

2.1.1.3 WP-LDP

Tento plugin je nejlepší co se komplexnosti tvorby sémantických dat ve Wordpressu týče. Není však ani zdaleka tak rozšířený jako jeho předchůdci, má totiž méně než deset aktivních instalací. Tento fakt přisuzuji tomu, že běžný uživatel Wordpressu chce, aby kliknutím na tlačítko aktivovat plugin měl vše připravené, ale právě komplexnost tohoto pluginu je zřejmě odrazujícím faktorem, neboť vyžaduje další nastavení.

Na rozdíl od předchozích řešení není určen k automatické transformaci již existujících dat do RDF, ale naopak k vytváření nezávislého modelu na struktuře Wordpressu a editaci dat zcela nových už sémantických. Data na rozdíl od předchozích pluginů poskytuje jen pomocí REST API ve formátu JSON-LD.

Plugin v globálním nastavení potřebuje pouze odkaz na soubor s definovaným kontextem v syntaxi JSON-LD, pomocí něhož jsou definovány veškeré prefixy.

Při tvorbě samotných dat je pak nejprve potřeba vytvořit takzvané kontejnery, které definují typy datových entit. Zde se nejprve definuje *rdf:type*, kterým říkáme jakého typu daný kontejner bude. Dále pak vytváříme Model za pomocí syntaxe JSON, ve kterém definujeme jednotlivé podporované property entity v poli fields. Jak toto probíhá by mělo být patrno z ukázky kódu 2.2.

Posledním dílem do skládačky jsou zdroje (resources), které pak reprezentují jednotlivé instance entit. Zde už se jedná o klasickou editační stránku, podobnou jako tomu je při editaci post type, ale doplněnou o vygenerované vlastní pole přesně podle definice modelu z daného kontejneru.

2.1.2 Pluginy pro Joomla

Joomla vznikla oproti Worpdressu o dva roky později, tedy v roce 2005 a nikdy se jí nepodařilo dohnat náskok svého staršího kolegy a na trhu se tedy nikdy nestala jedničkou. Joomla je oproti Wordpressu více technicky propracovaná a se svým 6,6% zastoupením na celkovém trhu je poměrně velkým hráčem. Není proto divu, že i pro ni vznikl plugin pro podporu sémantických dat. Záměrně mluvím v jednotném čísle, protože se mi opravdu podařilo najít jen

```
{
  "artwork": {
    "fields": [
    {
      "label": "Photos, images",
      "data-property": "foaf:img",
      "type": "url",
      "multiple": "true"
    },
    {
      "label": "Nom",
      "data-property": "foaf:name"
    },
    {
      "label": "Description (140 characters max)",
      "data-property": "pair:shortDescription",
      "type": "textarea"
    },
    ]
  }
}
```

Ukázka kódu 2.2: Ukázka definice modelu kontejneru

jeden jediný plugin, který se zabývá podporou implementace strukturovaných dat do tohoto systému, a tím je Google structured data markup.

Mohl bych ještě zmínit plugin Rich snippets vote, který se však vymezuje jen na naprosto jednoduchou funkcionalitu, a to zobrazování uživatelského hodnocení ve formátu strukturovaných dat. Tento plugin nemá, co se týče podpory strukturovaných dat, žádné větší ambice, a ve své analýze se mu proto nevěnuji.

2.1.2.1 Google Structured Data Markup

Jak už je z názvu tohoto pluginu patrné, snaží se o zavedení podpory strukturovaných dat pro Google Rich Snippets. Celá funkcionalita pluginu je tedy silně ovlivněna tím, jaké entitní typy právě Rich Snippets interpretují.

Jedná se o jeden z pluginů, které primárně převádějí obsah automatickými tranformacemi na strukturovaná data, avšak oproti pluginům, které takto fungovaly pro Wordpress, má tento mnohem pokročilejší možnosti konfigurace.

Pokročilé možnosti konfigurace jsou však vykoupeny zpoplatněním tohoto pluginu. Naštěstí však obsahuje i free verzi ochuzenou o některé funkcionality. Měl jsem tedy alespoň možnost hlouběji prozkoumat tuto verzi.

V plné verzi svým uživatelům nabízí tyto typy obsahů:

• Články • Kurzy • Události • Ověření faktů • Produkty • Recepisy • Hodnocení • Videa

Ve volné verzi bohužel podporuje jen entity typu Články, takže se dostane na podobnou úroveň přidané hodnoty, jakou měly pluginy ve Wordpressu. Oproti pluginům pro Wordpress však na transformaci jednotlivých příspěvků nedochází zcela automaticky. To je právě kvůli možnosti volby z více typů datových entit. U jednotlivých příspěvků je nejprve potřeba zvolit v patřičném nastavení typ entity. U každého z nich je pak ještě možnost ovlivnit i obsah jednotlivých property, kterými daný typ disponuje. Například u entit typu Článek můžeme vyplnit titulek a popisek a sémantický obsah těchto polí se tak může lišit od toho pro normální uživatele. Pokud však nic nevyplníme, použije se ten z běžného obsahu.

Kromě typů příspěvků ještě podporuje několik globálních strukturovaných dat, která se týkají celého webu. Jedná se o tato data:

Stránka ve smyslu domény, jež je reprezentována enitou *schema:WebSite*.

Drobečková navigace reprezentována jako entita typu *schema:BreadcrumbList* a jednotlivé položky jako *schema:ListItem*.

Logo reprezentováno jako property *schema:logo* entity *schema:Organization*.

Vyhledávač reprezentován jako entita *schema:SearchAction*.

Sociální sítě - ty jsou však dostupné jen v placené verzi pluginu.

Dalším pozitivem tohoto pluginu je i fakt, že podporuje integraci dalších oblíbených rozšíření pro Joomla. Pomocí těchto rozšíření totiž mohou uživatelé také utvářet obsah webu. Plugin Google Structured Data Markup pak však zajišťuje i automatickou transformaci obsahu vytvořeného za pomoci těchto rozšíření. Tato funkcionalita je bohužel obsažena jen v placené verzi.

Strukturovaná data jsou publikována ve formátu JSON-LD přímo jako součást standardní odpovědi serveru. To je výhodné, protože tento plugin je určen primárně pro použití v Google Rich Snippets a ty tento způsob publikace strukturovaných dat doporučují.

2.1.3 Drupal

Drupal je ze zmiňovaných CMS systémů nejstarším, jeho první verze vyšla už v roce 2000. Na rozdíl třeba od Wordpressu se nesnaží zachovávat zpětnou kompatibilitu. Každá majoritní verze činí radikální změny, které na ni nemusí brát ohled. Právě díky této strategii je v mnoha ohledech architektonicky Drupal ze zmiňovaných systémů nejpokročilejší a nejmodernější. Zároveň také primárně cílí na pokročilejší a řekněme technicky zdatnější uživatele.

Pravděpodobně díky těmto okolnostem je právě Drupal, co se podpory sémantických dat týče, nejdále. Od majoritní verze 7 totiž obsahuje přímo ve

svém jádru modul RDF. Dá se tedy hovořit, že Drupal je už napůl specializovaný CMS systém pro strukturovaná data, protože integrace této funkcionality do jádra je jistě nespornou výhodou oproti rozšíření jen pomocí funkcionality pluginů. Navíc od té doby v nejnovější verzi 8 přistoupil ještě na těsnější integraci sémantických dat a jejich podpory přímo do jádra. Část funkcionalit, které ještě ve verzích 7.x byly řešeny za pomocí rozšiřovacích pluginů, jsou nyní již také součástí přímo core modulu RDF. Je tedy vidět, že v Drupal se o sémantická data opravdu zajímají a záleží jim na jejich integraci.

Přímo po čisté instalaci Drupal se už obsah webu publikuje automaticky v RDF, a to konkrétně v RDFa serializaci. I přes velkou podporu strukturovaných dat přímo v jádru tohoto systému existuje stejně jedno jeho velmi užitečné rozšíření. Tím je Schema.org configuration tool - RDF UI, které si může uživatel teprve volitelně nainstalovat. Pokud však chcete strukturovaná data v tomto systému využívat naplno, tak se bez něj téměř neobejdete.

2.1.3.1 Schema.org configuration tool - RDF UI

Tento plugin totiž umožňuje i ruční mapování ostatních content type na více specifické datové entity, než je jen výchozí mapování na entitu typu *schema:Article*. Plugin je rozdělen podle funkčnosti do dvou částí, a to RDF UI a RDF UI builder.

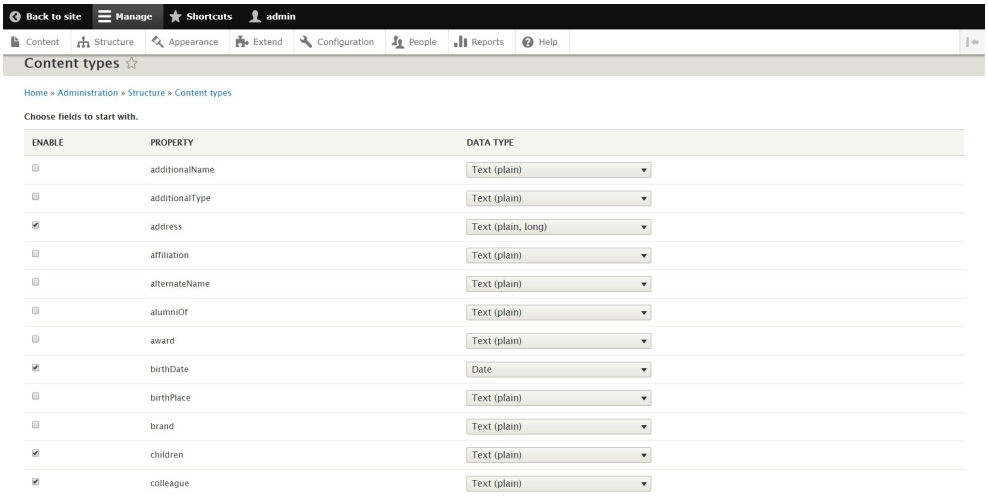
RDF UI poskytuje uživatelské rozhraní pro mapování content type a jejich jednotlivých polí na příslušné datové entity a property. Jak už je patrné z názvu pluginu, jedná se o entity ze slovníku Schema.org.

V praxi to vypadá tak, že pokud uživatel zakládá nový, nebo edituje stávající content type, má možnost si vybrat v select boxu typ entity ze seznamu, který obsahuje všechny entity ze slovníku Schema.org. Podobně pak u jednotlivých polí má uživatel na výběr ze všech vlastností, které slovník danému typu entity definuje.

Takto dojde k manuálnímu namapování a core modul RDF pak zajistí vypsání dat v RDFa. Problém tohoto řešení je, že mapování probíhá ručně a nekontroluje validitu příslušných datových typů.

RDF UI builder pak poskytuje uživatelské rozhraní k vytváření content type už přímo v sémantické podobě. Jedná se tedy o zjednodušení procesu mapování.

Tvorba nového content type probíhá tak, že si uživatel nejprve zvolí ze seznamu entitních typů a na základě toho se mu vygeneruje formulář s jednotlivými property. Vzhledem k tomu, že každý typ entity má několik desítek možných property, ale ne všechny chceme v našich datech podporovat. U každé vlastnosti je proto checkbox, pomocí něhož určujeme, zda-li chceme danou vlastnost použít u našeho nového content type nebo ne.



Obrázek 2.1: Výběr vlastností u entity typu *schema:Person*

Zároveň je u jednotlivých properties i selectbox, který umožňuje výběr datového typu, jímž bude vlastnost reprezentována. Toto typování probíhá sice automaticky, ale jako hlavní nedostatek musím zmínit, že funguje jen na jednoduché datové typy jako je číslo, datum atp. Nezvládá tedy mapovat properties typu entity. Ty mapuje pouze na textová pole. Jak celý tento proces probíhá, je zobrazeno na obrázku 2.1.

2.2 Analýza specializovaných CMS systémů

Tradiční CMS systémy trpí při podpoře sémantických dat nedostatky, které pramení z toho, že původně vůbec k tomuto účelu nebyly tyto systémy navrženy. Sémantická data vyžadují zásah přímo do datového modelu a to standardní systémy většinou formou rozšíření pomocí pluginů neumožňují nebo jen velmi omezeně. Výjimku tvoří Drupal, protože ten je k této funkcionalitě upraven přímo jeho autory, a je tedy někde na pomezí specializovaných a tradičních CMS systémů. I tak ale trpí některými nedostatky, a člověka proto napadne, zda-li nejde vytvořit systém přímo určený a od základu navržený pro správu obsahu v podobě sémantických dat. Objevil jsem však pouze jediné existující řešení a tím je Webnodes Semantic CMS.

2.2.1 Webnodes Semantic CMS

Webnodes Semantic CMS vznikl přímo za účelem podpory sémantických dat. Na rozdíl od všech předchozích systémů, které jsem zmiňoval, se nejedná o open-source. Od roku 2010 je vyvíjen společností Webnodes AS. Tato společnost dokonce obdržela na jeho vývoj granty od institucí Inovation Norway a The Research Council of Norway, aby ho vyvinuly jako sémantické CMS

postavené na technologii ASP.NET. Na redakčním systému je vidět, že jeho vývoj stál nemalé peníze a celý je spíše stavěn pro enterprise řešení.

Webnodes CMS je data centricky zaměřený systém. Jako vysvětlení tohoto pojmu nejlépe poslouží přeložená citace přímo z oficiálních stránek výrobce: Většina CMS systémů se soustředí na správu obsahu webových stránek v podobě hierarchické struktury. Problémem je, že obsah, který chcete na webu prezentovat, není přirozeně strukturován jako stránky. Když zobrazujete obsah dat jako internetovou stránku, jeho přirozená struktura a mínění je ztraceno. Ve Webnodes definujete nejprve svůj obsahový model (ontologii) tak, aby odpovídal přirozeným vztahům mezi jednotlivými typy obsahu. Obsah sám o sobě a to, jaké vazby mají jednotlivé typy obsahu mezi sebou, je to hlavní, o co v našem systému jde. Nejde tedy o to, jak poté chcete obsah použít na internetových stránkách. [7]

2.2.1.1 Ontology editor plugin

Od verze 5 obsahuje Webnodes CMS nástroj k modelování ontologií v podobě pluginu do Microsoft Visual Studio. V tomto pluginu může uživatel modelovat Obsahové třídy, což je analogie k datovým Entitám. U každé entity může přidávat properties a zároveň také nastavovat Schema.org type URI, což je přesně místo, kde probíhá provázání typu entity se sémantickým katalogem.

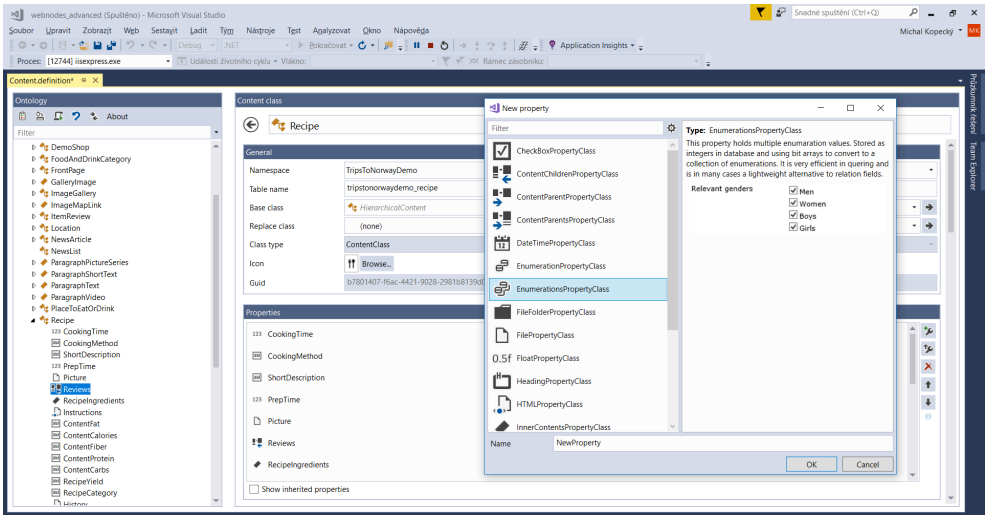
U každé property pak definujeme její datový typ. Zde je na výběr jednak z primitivních datových typů, typu vazby (1:1, 1:N, N:1), ale třeba také z typu vnořené entity nebo výčtového typu. Tento proces je zachycen na obrázku 2.2. Každé vlastnosti může uživatel nastavit také Schema.org property URI, čímž obdobně jako u entity dojde k provázání a určení jejího sémantického významu.

V další sekci pluginu si uživatel může modelovat již zmiňované enumerace, kde může nadefinovat seznam a jeho jednotlivé položky, na které se pak mohou vázat properties tohoto typu.

Poslední zajímavou částí je modelování vztahů (relací) mezi entitami. Pokud totiž vybereme property typu relace, musíme ji pak navázat na nějaký existující objekt, jenž relaci reprezentuje. Právě tento objekt totiž definuje význam vztahu mezi entitami.

2.3 Zhodnocení výhod a nevýhod sémantických rozšíření pro tradiční CMS systémy

V této sekci bych rád zhodnotil výhody a nevýhody rozšíření pro tradiční CMS systémy, které jsem podrobil svojí analýze. Ve své podstatě existují jen 3 typy těchto rozšíření. První jsou založené na automatické transformaci stávajících dat. Druhé vyžadují manuální mapování uživatelem a třetí nabízí sémantický obsah, jenž je úplně oddělen od obsahu pro uživatele.



Obrázek 2.2: Plugin do Visual Studia pro modelování ontologií

2.3.1 Výhody automatických řešení

Nespornou výhodou automatických řešení převodu obsahu do jeho sémantické podoby je to, že nestojí žádné nebo skoro žádné úsilí. Pro uživatele, kteří se o sémantická data zajímají jen kvůli SEO a chtějí pouze splnit požadavek, aby jejich stránka obsahovala nějaká sémantická data, která by mohla posloužit pro Google Rich Snippets, jsou tyto pluginy v podstatě ideální. Pouhým aktivováním pluginu všechno najednou funguje, a už je moc nezajímá, že pouze velmi omezeným způsobem.

2.3.2 Nevýhody automatických řešení

Problémem jednoduchých řešení založených na automatickém převádění existujícího obsahu do sémantické podoby je, že původní obsah skoro žádnou pevně danou strukturu nemá. Pokud se tedy pokusíme o její automatické rozpoznávání, nejsme na tom o moc lépe než dnešní vyhledávače. Naši výhodou oproti nim je, že máme alespoň nějaké informace o struktuře. Například v případě Wordpressu máme tři různé druhy entity: příspěvek, stránka a autor. U jednotlivých typů entit ještě například víme, který údaj je titulkem (jménem), co je jejím hlavním obsahem a který obrázek ji reprezentuje. Přesně tyto informace analyzované pluginy využívají a na základě toho generují příslušná sémantická data.

Je tedy jasné, že pokud chceme nabízet data lépe strukturované, musíme se zaměřit už na proces jejich vytváření, aby strukturovaná již vznikala. Pouze tehdy se vyhneme řešení problému, že stroje neumějí dost dobře porozumět jejich významu.

Pokud ještě zůstanu u automatického převádění, tyto převody nejspíše

skončí převedením na entity typu *schema:Article* nebo na nějaký z jeho podtypů. Pokud se však zamyslíme nad obsahem internetových stránek, nejsou to jen obyčejné články. Tento datový typ je tedy vhodný možná pro stránku typu blog nebo pro firemní novinky, ale pro většinu jiných stránek je to typ naprosto nevhodný a o ničem nevypovídající. Většina internetových stránek pojednává o nějaké konkrétní entitě, v případě výpisových stránek se jedná o agregaci podle jejich typu, či o agregace více různých typů entit.

2.3.3 Výhody manuálního mapování

Výhodou ručního mapování obsahové struktury na sémantická data je to, že uživatel má plně pod kontrolou, jakým způsobem jsou data a vztahy mezi nimi reprezentovány. Administrátor sám nejlépe zná strukturu svých dat a tu je schopen zachytit pomocí mapování na sémantické typy. Právě to je klíčové proto, aby později byla data dobře čitelná i pro programy.

Při procesu mapování obsahu nedochází k jeho redundanci, a uživatel ho tedy edituje jen na jedno místě pro obě jeho verze interpretace. Proces mapování je také užitečný v tom, že část obsahu může záměrně zůstat nestrukturovaná, pokud nedává smysl ji semantizovat, nebo zda-li ji záměrně chceme vynechat ze sémantické podoby, ať už je k tomu naše motivace jakákoliv.

2.3.4 Nevýhody manuálního mapování

Jak už to s manuálními činnostmi bývá, jsou náchylné k uživatelské chybě a ani tento postup není výjimkou. Properties jednotlivých entitních typů často vyžadují přiřazení hodnoty jiného specifického datového typu. Toto přiřazení ovšem žádný z analyzovaných pluginů nekontroluje, takže se snadno může stát, že uživatel udělá mapování nekorektně a poskytovaná data jsou tak sice strukturovaná, ale nevalidní.

Další nevýhodou je to, že manuální mapování vyžaduje, i když jsou k tomu zkoumané pluginy celkem dobře uzpůsobeny, nějakou činnost navíc, což může hodně uživatelů, kteří nevidí ve strukturovaných datech dostatečný přínos, odradit.

Poslední a asi největší nevýhodou je, že uživatel musí mít alespoň základní povědomí o sémantických datech a o slovnících, pomocí kterých mapování provádí. Aniž by tomuto porozuměl, nemůže manuální mapování provést.

2.3.5 Výhody odděleného sémantického obsahu

Výhodou odděleného sémantického obsahu tak, jak to nabízí například plugin WP-LDP popisovaný v 2.1.1.3, je to, že můžeme poskytovat sémantický obsah i ve chvíli, kdy už struktura nesémantického byla vytvořena a její zpětné namapování není možné.

Výhodou je také to, že můžeme poskytovat obsah v jeho opravdové struktuře, co se i vazeb na ostatní entity týče. Při manuálním mapování existujících

entit jsme totiž stále omezeni strukturou webu a podobou, v jaké se dané entity prezentují. Oproti tomu, pokud jsme oproštěni od zobrazovaného obsahu webu jako HTML stránek, máme ruce volné a můžeme namodelovat ontologii dat tak, jak skutečně má být.

2.3.6 Nevýhody odděleného sémantického obsahu

Největší nevýhodou odděleného sémantického obsahu je právě jeho oddělenost. Z této podstaty plyne, že obsah musí být vyplňován a upravován na dvou různých místech, což je náročné na jeho údržbu a konzistenci, a je to tedy další práce pro uživatele navíc.

Další nevýhodou je, obdobně jako u ručního mapování, že i zde proces modelování vyžaduje uživatelskou znalost sémantických dat a sémantických slovníků. Tady to platí o to více, protože uživatel nemá možnost se opřít o strukturu již existujícího obsahu, takže musí dobře vědět, co dělá.

Nevýhodou je také to, že jednotlivé reprezentace těch samých dat jsou odděleny na jiných URL. Taková data bez další integrace do stávajícího obsahu tedy neposlouží ani jako pomůcka pro Google či jiné vyhledávače. To bývá hlavní motivací, proč se o strukturovaná data pokoušet. Toto řešení je spíše vhodné, pokud svoje data potřebujete poskytnout formou API a dále je konzumovat.

2.4 Zhodnocení výhod a nevýhod Webnodes Semantic CMS

Webnodes Semantic CMS je zde jako jediný zástupce systému vyvinutého originálně pro podporu práce se sémantickými daty. Tento fakt je na CMS vidět a oproti tradičním CMS má nespornou výhodu, protože byl navržen právě za tímto účelem.

2.4.1 Výhody Webnodes Sematnic CMS

Jako hlavní výhodu spatřuji to, že data jsou uložena přímo v sémantické podobě v přirozené struktuře. Při jejich vytváření i editaci tedy nejsou nijak deformována do modelu stránek, tak, jak se tomu děje v případě tradičních CMS systémů. Díky tomu může CMS sloužit nejen pro webové stránky, ale i jako centrální úložiště dat. Díky poskytovanému API v podobě SPARQL a OData endpointu může posloužit i jako zdroj dat pro mobilní aplikace a ostatní systémy.

Další výhodou je to, i když to může být sporné, že k reprezentaci sémantiky dat používá svůj vnitřní mechanismus modelování ontologií a nezávisí tak na žádném konkrétním sémantickém slovníku. Kapacita těchto slovníků totiž bývá někdy dosti omezena. Na druhou stranu obsahuje ale plně integrova-

nou podporu pro slovník Schema.org a nabízí jednoduchý nástroj k mapování jednotlivých typů pomocí tohoto slovníku.

2.4.2 Nevýhody Webnodes Semantic CMS

Možná není úplně fér tomuto redakčnímu systému vyčítat hned na první místě to, že se nejedná o open-source. Vzhledem k ceně jeho licencí začínajících po přepočtu na skoro 180 000 Kč si myslím, že to je jeden z největších problémů, proč toto CMS nemůže být nikdy globálně rozšířeno, i kdyby bylo sebelepší. V praxi se totiž ukazuje, že nejrozšířenější CMS systémy jsou open-source, které jsou zcela zdarma.

Další negativní věcí, kterou bych tomuto CMS vytkl, je zvolená technologie. Ne samozřejmě kvůli tomu, že by ASP.NET bylo špatné nebo méněcenné, ale opět kvůli ceně a rozšíření hostingů podporujících ASP. Toto řešení tak podle mého názoru může být možná ideální pro enterprise zákazníky, ale je zkrátka nedostupné pro většinu okolí a jeho přínos k celkovému rozšíření sémantického webu se limitně blíží k nule.

S tím se pojí i to, že nástroj pro modelování ontologií je realizován v podobě pluginu pro Microsoft Visual Studio. Chápu, že tento nástroj je ve Windows světě standardem pro vytváření .NET aplikací, ale jen ještě více umocňuje závislost Webnodes CMS na konkrétní platformě, a dokonce ještě na vývojovém prostředí.

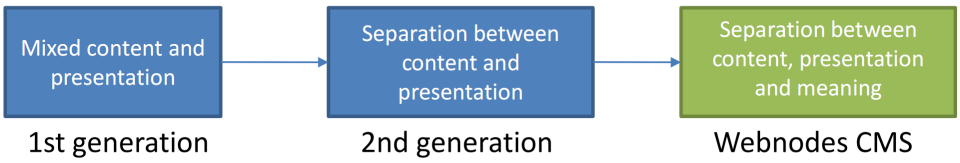
2.5 Srovnání tradičních CMS systémů a Webnodes Semantic CMS

Vzhledem k tomu, že Webnodes vzniklo jako sémantické CMS, dá se očekávat, že oproti tradičním stránkově založeným CMS bude mít výhodu. V čem ale ona výhoda spočívá a proč je vůbec dobré o CMS systému přemýšlet trochu jinak než doposud? Právě na tyto otázky by měla odpovědět následující sekce.

2.5.1 Datacentrický versus stránkový pohled na data

Nechci rozhodně tvrdit, že Webnodes CMS je dokonalé a že ostatní systémy by si z něj měly brát jen příklad. Co ale musím vyzdvihnout je právě jeho datacentrický přístup. Největším rozdílem oproti běžným CMS je, že na začátku si musíte namodelovat ontologii svých dat. Ta je výchozím bodem a další úkony, jako tvorba dat a jejich publikace, vychází právě z vytvořené ontologie. To považuji za největší výhodu tohoto systému. Při práci se sémantickými daty proces jejich tvorby působí zcela přirozeně, protože se s jejich sémantikou počítá už od začátku a systém k tomu není nijak dodatečně přizpůsobován.

Oproti tomu tradiční redakční systémy, které jsem analyzoval, jsou stále založeny převážně na editaci obsahu v podobě HTML dokumentů. I nejpokročilejší systém Drupal se musí potýkat s tím, že data jsou do sémantické podoby



Obrázek 2.3: Vývoj generací CMS systémů podle Webnodes semantic CMS

mapována dodatečně, což je omezující, protože struktura HTML stránek velmi často neodpovídá struktuře dat. Toto mapování v nich navíc probíhá většinou až po instalaci nějakého rozšíření, a to se tak potýká s omezeními, na která v daném systému naráží.

2.5.2 Headless vs. klasické CMS

Jako další výhodu Webnodes CMS spatřuji i to, že se jedná o tzv. headless CMS. Redakční systém poskytuje data v něm uložená jen pomocí API a chybí mu front-end. Takový systém může sloužit jako centrální úložiště dat nejen pro webovou aplikaci, ale zároveň i pro mobilní, nebo například pro nějaký interní systém typu CRM. Tím, že nenabízí žádný front-end, nikoho ani nenapadne o plnění dat přemýšlet jako o nějakých webových stránkách, ale spíše jako o jakési databázi. To je výhodné, protože právě sémantická data jsou spíše jakousi speciální databází znalostí.

Oproti tomu, i když už i řada tradičních systémů poskytuje data také za pomoci API, jejich struktura bývá stále omezena tradičním pojetím redakčních systémů jako prostředků pro správu stránek a ne přímo dat. Většina uživatelů těchto systémů také tuto funkcionalitu nevyužívá, protože typičtí uživatelé používají hotová řešení, což jsou šablony postavené na klasickém front-endu.

Kapitola 3

Návrh vlastního řešení rozšíření redakčních systémů

V analytické části práce jsem zjistil, že současná situace pro podporu sémantických dat v redakčních systémech není až tak špatná, jak jsem si původně myslel. Pokud o ně mají uživatelé zájem, existuje už celá řada nástrojů, jak sémantická data v redakčních systémech spravovat.

Na druhou stranu, ve většině případů se jedná o volitelná rozšíření, a pokud jejich uživatelé nemají o sémantických datech povědomí, těžko si některé z nich nainstalují. Zářnou výjimku tvoří Drupal, který podporu sémantických dat integroval přímo do svého jádra, ale i v něm jsou pokročilejší funkcionality dostupné až po nainstalování dalšího rozšíření.

Různé redakční systémy a jejich rozšíření řeší integraci sémantických dat na různé úrovni a odlišnými přístupy. Co však mají společného je to, že jejich možnosti jsou stejně dosti omezené a limitované systémem, do kterého byly implementovány. Oproti tomu Webnodes CMS je možná až moc komplexním nástrojem a je určen spíše enterprise zákazníkům. K celkovému rozšíření sémantických dat proto podle mě také moc nepřispívá.

Mým cílem je se poučit z nedostatků rozšíření pro klasické redakční systémy a zkusit navrhnout vlastní řešení, které by bylo od nich oproštěno. Nechci se například omezovat jen na primitivní entitní typy jako je *schema:Article*, ale chtěl bych poskytnout nástroj pro modelování sémantických dat opravdu podle skutečných entit z reálného světa. A právě tomu návrhu je věnována následující kapitola.

3.1 Datový model

Jak už jsem zmiňoval, sémantická data potřebují zasáhnout přímo do datového modelu a právě to je největší překážkou zdárné integrace do již existujících redakčních systémů.

Klasické webové stránky jsou totiž reprezentovány naprosto jednoduchým datovým modelem, který zná jen jediný typ entity, a to dokument. Tyto entity navíc mezi sebou nerozlišují různé druhy vazeb a pouze se mezi nimi můžeme utvářet relace pomocí odkazů. HTML 5 sice zavedlo další významové tagy, které nám dovolují členit dokumenty na další sémantické sekce, a můžeme tedy hovořit i o menší granualitě, ale i tak je tento model dosti omezen. V dalších verzích HTML ovšem nemá smysl zavádět nové významové tagy, protože jejich konečný výčet nikdy nemůže uspokojit všechny typy entit z reálného světa.

Konceptuální datový model každé domény je však daleko složitější. Vezmu například jednoduchou firemní stránku. Stránka jako celek bude pojednávat o dané firmě, tedy entitě typu *schema:Organization*. Na kontaktní podstránce se budou vyskytovat kontaktní osoby tedy entity typu *schema:Person*, a pak místa, kde má firma pobočky, tedy entity typu *schema:ContactPoint*. Firma bude prostřednictvím stránek nabízet svoje produkty, tedy typ *schema:Product*. A takto bych mohl pokračovat.

Je vidět, že jednotlivé dokumenty mohou najednou pojednávat o více datových entitách, a to i různých typů. Můžeme hovořit o tom, že internetová stránka jakožto dokument je pouze náhledem na určitou výseč dat z konceptuálního datového modelu domény. Důležitá je proto podle mě separace konceptů. Na konceptuální model a na to, jak jsou data v něm uložena, bychom měli pohlížet jako na jednu vrstvu. Ta obsahuje data v přirozené struktuře a tvoří tak graf. Na zobrazení dat pomocí dokumentů bychom měli pohlížet jako na druhou vrstvu, která definuje konkrétní projekci jejich určité výseče. Zde bych ještě rád zdůraznil, že pojmem zobrazení nemyslím v tomto kontextu interpretaci dat jen v nějakém konkrétním datovém formátu například v HTML, ale opravdu zobrazení části jejich grafu, který nemůžeme a ani nechceme reprezentovat celý v rámci jednoho dokumentu.

Abychom mohli sémantická data vytvářet a spravovat, měli bychom to dělat právě na této konceptuální úrovni. Tak bych to chtěl dělat i ve svém řešení. Vhodným výchozím systémem by mělo být proto headless CMS, které poskytuje data v něm uložená pomocí API. Právě to je totiž případ kdy redakční systém, ač nesémantický, data edituje v přirozené podobě podle jejich významu a ne podle zobrazení v rámci HTML stránek. Toto zobrazení častokrát řeší až cílová aplikace, která dat z API využívá a mixuje je do sebe. Té tedy můžeme poskytnout data sémantická, ze kterých sestaví výsledný pohled na graf v podobě HTML dokumentů a zároveň je vloží do dokumentu ve formátu JSON-LD.

3.2 Modelování ontologií

K vytváření konceptuálního modelu potřebujeme prostředek, pomocí něhož ho budeme definovat, nebo ještě lépe uživatelské rozhraní, díky kterému si ho nakonfigurujeme v přehledné grafické podobě. Ostatně na tomto principu je

založeno právě i ono specializované Webnodes CMS a obdobně slouží například i nástroj pro mapování obsahových typů v Drupal.

Pokud chceme udělat nástroj pro modelování ontologií, musíme pracovat s těmito třemi stavebními kameny:

1. Entitní typy
2. Atributy
3. Vazby

Entita je definována jako libovolný objekt tvořící součást reálného světa, který je zároveň obsažen v datovém modelu. Může jim být například člověk, zvíře, věc nebo jev. Typy entit jsou množiny objektů totožného typu se stejnými atributy, podle kterých jsou určeny. [8]

A právě tento typ je důležitý k definování významu jednotlivých entit v sémantických datech. Tyto typy pak obsahují atributy, jinými slovy properties, které určují jednotlivé vlastnosti entit. Atributy pak mohou nabývat hodnot jiných entit nebo jednoduchých datových typů jako číslo, datum atd. v RDF známých jako literály.

Posledním stavebním prvkem jsou vazby, které určují násobnost relací mezi jednotlivými entitami. Stejně jako v databázovém modelu existují 3 typy vazeb a to: 1:1, 1:N a N:N. Reprezentace modelu tak musí být schopna zachytit všechny tyto variace, které mohou vzniknout kombinováním těchto tří stavebních kamenů. Tento proces můžeme nazvat modelováním ontologie.

Proto, aby se takto utvořený datový model dal později snadno jednoduše použít pro generování sémantických dat, bych chtěl při tomto procesu zohlednit podporu entit z nějakého konkrétního sémantického slovníku. Hlavní výhodou tohoto přístupu je to, že uživatelé i neznalí sémantických dat budou mít k dispozici předdefinované entitní typy i jejich atributy a nebude od nich tak vyžadována větší znalost principů strukturovaných dat při modelovacím procesu.

Jako ideální kandidát se mi jeví slovník Schema.org, který definuje základní entity z reálného světa, jejich jednotlivé atributy a relace mezi nimi. Volba na tento slovník padla také mimo jiné proto, protože je hojně podporován Google, který umí pomocí jeho syntaxe generovat takzvané Rich Snippets, které se zobrazují ve výsledcích vyhledávání a mohou obsahovat další užitečné informace o obsahu stránky. Zároveň tento slovník podporovala i celá řada analyzovaných rozšíření, což svědčí o jeho významu.

3.2.1 Grafický nástroj pro modelování ontologií

Aby redakční systém skutečně zjednodušoval práci se sémantickými daty, je podle mě důležité, aby byla tato skutečnost, že s nimi pracuje, co nejvíce na pozadí. V ideálním případě by ani neměl rozpoznat, že vykonává nějakou

činnost navíc, než by dělal, kdyby pracoval s daty nesémantickými. Právě k tomuto komfortu by měl sloužit grafický nástroj pro modelování ontologií a měl by být součástí i mého řešení. To potvrzuje i plugin pro Visual studio, pomocí něhož modeluje ontologii Webnodes CMS a pak také plugin RDF UI Builder pro Drupal.

Moje vize je, aby si na začátku uživatel zvolil typ entity, který chce vytvářet, ze seznamu typů. Ten se bude dynamicky generovat ze slovníku Schema.org. Z tohoto typu se bude také automaticky generovat název entity, ale uživatel bude mít možnost tento název manuálně upravit, neboť třeba entity typu *schema:Person* může používat v různých významech, například jako Kontaktní osoby nebo třeba Poradci.

Díky této volbě bude v druhém kroku dostupný seznam atributů, jimiž daný typ disponuje. Vzhledem k tomu, že těchto vlastností některé typy entit obsahují i několik desítek, bude si uživatel moci vybrat jen ty, které chce opravdu pro svoji konkrétní entitu používat. Podobně jako u volby typu entity i zde bude název atributu automaticky vygenerován z typu property, ale zároveň si ho bude moci manuálně pojmenovat sám. Toto pojmenování je totiž důležité už i pro uživatele, který bude data plnit, a je tedy závislé na jazyku i na konkrétní doméně, kde se bude model využívat.

Při vytváření atributů ovšem nastává trochu problém, protože mohou být nejen jednoduchého datového typu, ale také typu jiné entity, která v době zakládání nemusí vůbec v našem modelu existovat. Aby toho nebylo málo, vstupuje zde do hry ještě také násobnost vazeb. Nejlépe budu vše ilustrovat na konkrétním příkladu.

Představte si, že třeba chcete založit entitu typu *schema:Movie*. Tento typ entity má mimo jiné i property *schema:actor*, která může obsahovat jen entity typu *schema:Person*. V jednom filmu může hrát více herců a zároveň jeden herec může hrát v několika různých filmech. Vazba, která tyto dva typy entit pojí, je tedy typu M:N. K tomu, abychom vazbu mezi nimi mohli utvořit, potřebujeme, aby existoval jak typ *schema:Person*, tak typ *schema:Actor*. Pokud chceme v našem nástroji namodelovat ontologii obsahující tuto vazbu, musíme to udělat po částech. Pokud se nejprve snažím založit entitu typu film a mezi nabízenými vlastnostmi si vyberu, že chci podporovat vlastnost herec, dostanu buď na výběr ze všech entit typu *schema:Person*, které v mé onotologii již existují. Pokud žádná neexistuje, musí ve stejný okamžik uložení entity typu film vzniknout i nová prázdná entita typu *schema:Person*, jejíž atributy si budu moci nakonfigurovat později.

Upozorňoval jsem zde i na problém násobnosti těchto vazeb. Tu slovník Schema.org nikde bohužel neurčuje a všude může být teoreticky mnohonásobná. V praxi to však u některých vlastností nedává smysl a bylo by lepší ji definovat. Při hledání řešení tohoto problému jsem narazil na seznam properties a odpovídajících definic toho, zda-li jsou násobné nebo ne. Nejedná se však o oficiální knihovní záležitost a nemusí tedy obsahovat všechny definované typy properties.

V mém řešení by bylo dobré vyžadovat určení této násobnosti vazeb ručně, případně lze použít přednastavení typu vazby podle zmiňovaného seznamu. Toto určení však nemůže být závazné a uživatel musí mít možnost si ho změnit.

3.3 Tvorba a editace dat ve strukturované podobě

Na základě namodelované ontologie by se mělo generovat i uživatelské rozhraní pro editaci obsahu. Podobně jako ve Wordpressu, kde existují takzvané post type, které definují typy příspěvků a tento typ je pak patřičně zobrazen v menu, tak i v mém rozšíření by podle zvoleného entitního typu měla být v menu položka pro každý tento typ.

Po kliknutí na tuto položku by měl být zobrazen seznam všech příspěvků patřících k dané entitě. V tomto výpisu by mělo fungovat stránkování a vyhledávání tak, jak to je v podobných systémech dobrým zvykem. Při kliknutí na řádek reprezentující konkrétní instanci entitního typu by měl být zobrazen formulář pro její editaci. Zároveň by zde mělo být tlačítko pro vytvoření nové instance, které povede na obdobný formulář, jako je ten pro její editaci.

3.3.1 Formulář pro editaci obsahu instance entity

Tento formulář by měl být dynamický, generovaný na základě namodelované ontologie. Měl by obsahovat pole pro všechny vlastnosti, které si při modelování ontologie uživatel zvolil jako podporované.

Zde je potřeba rozlišit mezi atributy jednoduchých datových typů, jejichž vstup se dá zajistit jen pomocí jednoho patřičného inputu, a atributy typu entity. Například properties typu *schema:Date* se dají vyplňovat za pomoci HTML inputu typu date, avšak vlastnosti typu *schema:Person* odkazují na složený datový typ z dalších mnoha položek. Tyto vlastnosti se pak mohou spravovat pomocí výběru z listu entit jiných typů a podle typu vazby je umožněn mnohonásobný nebo jednonásobný výběr přiřazené entity.

Vzhledem k tomu, že atributy v sémantických datech bývají dost často typu jiné entity, pouze tento způsob by vedl na potřebu velkého množství různých entitních typů a jejich správa a plnění daty by se tak staly nepohodlnými.

Pro entity s vazbou 1:1 je proto možné zavést vnořený formulář pro tuto entitu a plnit její obsah v strukturované podobě přímo v rámci formuláře nadřazené entity.

3.4 Publikování sémantických dat

Všechna analyzovaná řešení nabízela publikování strukturovaných dat jen v jednom formátu, což mi přijde omezující. Napříč řešeními byly sice zastoupeny různé datové formáty, avšak žádné z nich je nenabízelo současně. Jednotlivé serializace RDF by přitom měly být mezi sebou převoditelné, a neměl by být

tedy problém uživateli poskytnout data rovnou ve formátu, o který si řekne za pomoci nastavení hlavičky Accepted ve svém požadavku.

Ve svém řešení bych toto rád napravil a podporoval serializace JSON-LD, Turtle a N-Triples, které považuji za nejrozšířenější. Oproti tomu se plánuji vyhnout formátu RDFa, který slouží jako rozšíření pro formát HTML, protože ten je už závislý na konkrétním zobrazení dat pomocí HTML šablon.

Kapitola 4

Výběr vhodného CMS, návrh a implementace jeho rozšíření

Cílem mojí práce je mimo jiné i implementovat řešení pro podporu sémantických dat jako vhodné rozšíření některého z existujících CMS systémů. Různých redakčních systémů ale existují stovky a otázkou tedy je, který si k rozšíření vybrat. V předchozí kapitole jsem navrhoval způsob, jakým by moje řešení mělo fungovat, což určitě tento výběr také ovlivňuje. Před začátkem implementace bylo potřeba určit, který redakční systém bude nejvhodnější k tomu, abych ho rozšířil o tuto funkcionalitu.

4.1 Zvolení vhodného CMS k rozšíření

Úplně na začátku jsem přemýšlel o svém řešení jako o možném dalším rozšíření pro Wordpress. To mě lákalo hlavně z důvodu, že jeho úspěšná implementace by znamenala potencionálně velký záběr mnoha uživatelů. Když jsem ale vyšel z poznatků analytické části, tak aby bylo moje řešení v něčem přínosné a neomezovalo se jen na automatické transformace jako již existující pluginy, muselo by být podporováno i přímo konkrétní používanou šablonou. Takové řešení by tedy stejně nakonec nezasáhlo mnoho uživatelů, protože pro použití mého pluginu by musela být vždy šablona na míru a její tvůrce by musel se sémantickými daty počítat už od začátku. O výhodu velkého počtu uživatelů bych tedy stejně přišel.

Přemýšlel jsem i nad jinými kritérii, která by mi umožnila zvolit vhodné CMS, o jehož rozšíření bych se pokusil. Z poznatků analytické části a mnou navržených principů, jak bych si přál, aby mé řešení fungovalo, mi vyplynuly následující požadavky.

4.1.1 Kritéria výběru vhodného CMS

Konfigurovatelný datový model
Zvolené CMS by mělo obsahovat konfigurovatelný datový model, protože pokročilejší integrace sémantických dat vyžaduje jeho modifikaci. Zároveň je důležitá i jeho dostatečná flexibilita, aby umožňoval vytváření libovolného počtu entitních typů a relací mezi nimi, protože právě sémantická data jsou v tomto ohledu velmi rozmanitá.

Nástroj pro modelování ontologií
Z podstaty principu fungování mého navrženého rozšíření je také potřeba, aby šel zvolený redakční systém rozšířit o nástroj pro modelování ontologií.

Jako nejvhodnější kandidát se jeví ten, který už ve svém základu vytváření datového modelu po svých uživatelích vyžaduje. Moje řešení by pak mohlo být jen rozšířením tohoto procesu, aby zohledňoval i sémantiku dat, a to by se tak mohlo stát jeho přirozenou součástí.

Databázová vrstva
Další věcí, kterou bych při výběru vhodného CMS chtěl zohlednit, je databázová vrstva. Pro sémantická data totiž existuje serializace JSON-LD, která se dá nativně uložit do dokumentově orientovaných databází jako je MongoDB. Vzhledem k tomu, že každý JSON-LD je zároveň validní JSON, data v tomto formátu díky tomu mohou být uložena přímo v sémantické podobě, což považuji za další výhodu. [9]

Otevřená licence
Redakční systém, který budu rozšiřovat, by měl být pod otevřenou licencí. To jednak z důvodu, abych mohl v případě potřeby zasáhnout i do kódu aplikace, ale hlavně kvůli tomu, aby moje rozšíření bylo dostupné pro co největší počet uživatelů.

Headless CMS
Jako poslední požadavek jsem si vytyčil, že CMS, které budu rozšiřovat, by mělo být takzvané headless, tedy bez front-endu. Headless CMS poskytují data v nich spravovaná ve formě API a právě v této situaci je nejlépe využitelné to, že jsou data ve strukturované podobě.

Zároveň toto řešení využívá i Webnodes Semantic CMS, což jen potvrzuje výhodnost tohoto přístupu ve spojení se sémantickými daty.

4.2 Strapi

Právě díky těmto požadavkům jsem narazil mimo jiné i na CMS Strapi. To jsem si nakonec vybral nejen proto, že splňovalo veškeré mé požadavky, ale

také proto, že je postavené na NodeJS. Osobně této technologii do budoucna věřím a chtěl jsem se sní více seznámit.

V porovnání třeba s Wordpressem se jedná o vcelku mladý projekt, jehož první release vyšel v říjnu 2015. Za jeho vývojem stojí společnost Strapi Solutions, avšak díky jeho otevřenému kódu se komunita vývojářů rozrostla i mimo ni.

Projekt je v současné době stále pod aktivním vývojem a tento fakt je ještě umocněn tím, že jsem ve své práci upravoval verzi 3.x, která oproti předchozí prošla výraznými změnami a je teprve v alfa stadiu vývoje. Na druhou stranu díky tomu, že jeho vývoj je stále aktivní, by integrace mého rozšíření pro podporu sémantických dat mohla být do jisté míry přímo součástí jádra systému, což by bylo snazší prosadit než v případě už zaběhlých systémů.

Strapi je postaveno na moderních technologiích a kromě serverové části běžící na NodeJs má administraci napsanou v React. Celé CMS je napsané v JavaScript. Co se týče databázové vrstvy, tak mimo mnou požadované MongoDB podporuje také standardní relační databáze jako MySQL či PostgreSQL. Sami autoři však doporučují použití právě s MongoDB.

Strapi samo o sobě tvrdí, že je nejpokročilejším open-source content management frameworkem. Nejedná se tedy o úplně klasický redakční systém, jako byly ty, které jsem analyzoval, ale jedná se o takzvané headless CMS a poskytuje data v něm spravovaná jen pomocí REST API.

Strapi se také ukázalo jako ideálním kandidátem k rozšíření o sémantická data i kvůli tomu, že už v jeho základu obsahuje plugin. Pomocí něho uživatelé modelují jakousi ontologii, a vytváření modelu dat je tedy pro jeho uživatele přirozenou součástí.

Další výhodou je také to, že se jedná o modulární systém a i core funkce jsou odděleny do jednotlivých modulů, což znamená jednoduchou rozšířitelnost a zaměnitelnost těchto součástí.

4.2.1 Datový model

Základní částí Strapi je právě datový model, který definuje jednak uživatelské rozhraní pro plnění dat, a také REST API, které posléze systém poskytuje.

Datový model se skládá z takzvaných obsahových typů (content type), vazeb mezi nimi a z atributů.

Obsahové typy můžeme chápat jako jednotlivé typy entit. Atributy mohou být různých typů například string, number, text či třeba obrázek a dalších. Jednotlivé atributy jsou povětšinou reprezentovány primitivními datovými typy a můžeme je chápat jako properties entitních typů.

Všechny modely jsou uloženy po složkách pojmenovaných podle názvu obsahového typu ve složce *api*. V každé z těchto složek najedeme složky obsahující definice modelů, konfigurace routerů, kontroléry a služby.

```
1
2  {
3    "connection": "default",
4    "collectionName": "",
5    "info": {
6      "name": "event",
7      "description": "events in the region"
8    },
9    "options": {
10     "timestamps": true
11   },
12   "attributes": {
13     "eventName": {
14       "multiple": false,
15       "type": "string",
16     },
17     "location": {
18       "multiple": false,
19       "type": "text",
20     },
21     "actor": {
22       "collection": "actors",
23       "via": "actIn",
24     },
25   }
26 }
```

Ukázka kódu 4.1: Ukázka modelu entity Event

4.2.1.1 Popis struktury modelu

O popis struktury modelu se stará soubor *NazevEntity.settings.json* ve složce *models*. Strukturu popisuje ve formátu JSON a způsob tohoto zápisu je znázorněn v následující ukázce kódu 4.1.

Součástí definice content type je název entity, její popis, možná nastavení a dále zde pak může volitelně definovat i jiný název pro kolekci pro fyzické uložení v databázi (název tabulky).

Asi nejdůležitější částí popisu modelu je část attributes. V té se definují jednotlivé atributy modelu, jejich název, typ, unikátnost a další parametry specifické pro zvolený typ atributu.

Pomocí atributu jsou zachyceny i vazby mezi jednotlivými content type. K jejich utváření totiž dochází vždy přes atributy na obou stranách relace. V tomto speciálním typu atributu je zachycena vazba na jiný content type přes jeho název, který je přiřazen buď k property *model* nebo k *collection* v závislosti na násobnosti vazby. Pomocí property *via* je označen atribut, přes který je vazba v cílovém content type utvořena.

4.2.1.2 Kontroléry a služby

Složka *controllers* obsahuje další důležitý soubor *NazevEntity.js*, který definuje kontroléry, jež jsou tvořeny funkcemi. Tyto funkce jsou namapovány pomocí souboru *routes.json* na jednotlivé URL, které poté slouží jako REST API.

Neméně důležitým je také stejně nazvaný soubor ve složce *services*, který obsahuje definice služeb. To jsou opět obyčejné funkce, které však představují výkonnou část kódu starajících se o jednu určitou funkcionalitu. Jednotlivé kontroléry tyto funkce poté provolávají, dávají výsledky z nich dohromady a zaštiťují tak požadovanou funkcionalitu.

4.2.1.3 Vytváření modelu

V Strapi je na začátku model skoro prázdný. Obsahuje jen předdefinované entitní typy Users, Permission a Role. Ty slouží k interním účelům, a to pro správu uživatelů, jejich rolí a oprávnění přístupu k jednotlivým funkcím administračního prostředí. Poté ještě může obsahovat model File, pokud máte nainstalovaný plugin Upload.

K tomu, abyste mohli API Strapi využívat, je potřeba si na začátku vytvořit model podle potřeb konkrétní aplikace. Právě k tomuto účelu obsahuje nástroje, které toto vytváření mají co nejvíce usnadnit.

Jednak obsahuje takzvané generátory, což jsou konzolové aplikace, které umožňují definování struktury specifického content type. Ty také zároveň vygenerují i celou strukturu složky, včetně předdefinovaných kontrolérů, servis a konfigurací.

Druhým způsobem, jak modely vytvářet, je grafický nástroj v podobě pluginu Content Type Builder. Ten je obsažen již v základní instalaci Strapi a poskytuje k vytváření modelů grafické rozhraní.

4.2.2 Content Type Builder

Plugin Content Type Builder slouží k tvorbě a správě content type a jejich atributů. Můžeme ho nazývat nástrojem sloužícím k modelování ontologií.

Jak jsem již zmiňoval, jedná se o grafické rozhraní, jenž na pozadí provolává generátory a poskytuje tak uživatelům komfortnější a přehlednější vytváření modelů. Právě existence tohoto modulu učinila ze Strapi jednoho z hlavních kandidátů na implementaci mého rozšíření, protože právě díky němu jsou uživatelé používající Strapi zvyklí modelovat ontologii svých dat.

Modelování content type probíhá ve dvou krocích. To má svůj důvod, který vzápětí popíšu. V prvním kroku uživatel definuje údaje o samotném content type jako je jeho název, popisek atd. Poté se přesune k druhému kroku, ve kterém definuje jednotlivé atributy. Ty mohou být různých typů reprezentující buď jednoduché datové typy, jakožto text, číslo, datum, anebo také vazby na další obsahové typy.

Rozdělení tohoto procesu do dvou kroků je jednak kvůli přehlednosti, a jednak také kvůli vazebním atributům mezi dvěma entitními typy. Ty musí být zadefinovány v obou modelech současně. To je zajištěno právě díky tomu, že v době zadání názvu typu už může na pozadí content type vzniknout, i když zatím jen dočasně v paměti.

4.2.3 Content Manager

Dalším důležitým pluginem je Content Manager, který se stará o plnění uživatelských dat. Na základě vygenerovaných modelů utváří hlavní menu, kde si uživatel nejprve volí content type, který chce spravovat. Po zvolení konkrétního obsahového typu se dostane na výpis všech záznamů, které daný typ obsahuje. Ty posléze může editovat, mazat a nebo vytvářet nové.

Jak k editaci, tak k vytváření konkrétní entity slouží stránka s formulářem, která je automaticky generována na základě namodelovaných atributů. Podle zvoleného datového typu atributu se použije vhodný formulářový prvek a proběhne jakési automatické mapování mezi datovými typy a jednoduchými HTML inputy.

Výjimku tvoří atributy typu vazby mezi obsahovými typy. Pro ty Strapi definuje svůj speciální formulářový prvek, jenž umožňuje výběr více či jedné entity jiného typu v závislosti na násobnosti vazby. Tyto formulářové prvky jsou odděleny i graficky a nachází se v jiné části formuláře.

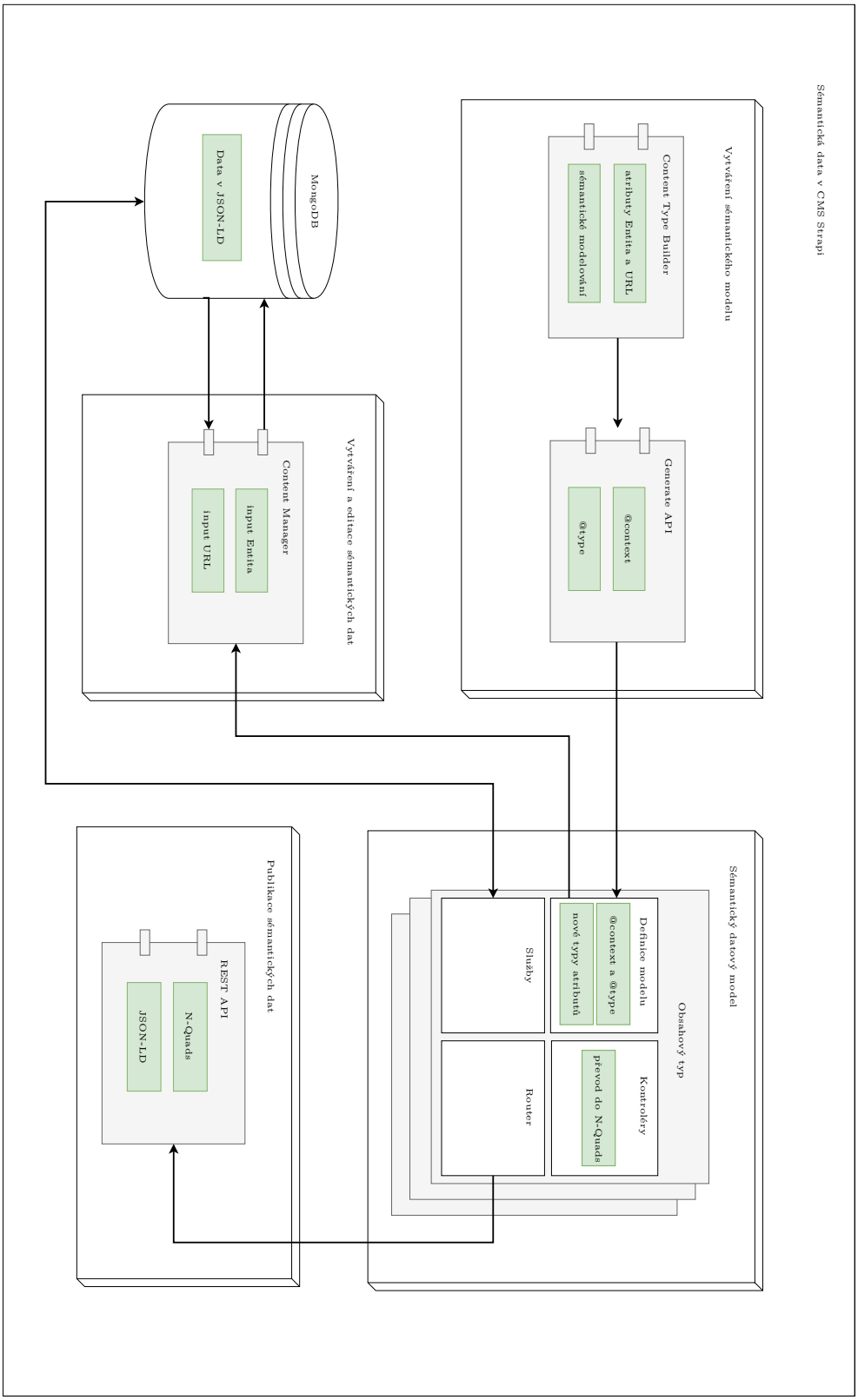
4.3 Implementace sémantického rozšíření do Strapi

K tomu, abych do Strapi přidal podporu pro sémantická data, jsem musel učinit celou řadu úprav, které se bohužel neobešly bez zásahu přímo do kódu aplikace. Upravil jsem celkem pět různých pluginů, které jsou součástí základní instalace Strapi. To je však nutná daň za těsnou integraci.

Tyto úpravy jsou však vesměs spíše rozšiřujícího charakteru a měly by být zpětně kompatibilní s originální verzí Strapi. Vzhledem k tomu, že je mnou rozšiřovaná verze stále v alfa stadiu vývoje, je tato skutečnost velmi nepříjemná a komplikuje udržení kroku s aktuální oficiální vývojovou verzí, protože v kódu může snadno dojít ke konfliktům bránícím jednoduchému sloučení.

O začlenění mnou provedených změn jsem jednal s komunitou okolo Strapi. V případě, že by alespoň některé změny začlenily přímo do svého kódu, situace by se zlepšila a otevřely by se i nové možnosti vydání upravené verze pluginu Content Type Builder jako vlastního.

Sémantická data však pro ně nejsou prioritou číslo jedna, takže jim nepřikládají velkou váhu. Jednání je proto zdlouhavé a nepodařilo se mi ho před odevzdáním této diplomové práce dotáhnout do zdárného konce. V současné době je tedy implementace pouze dostupná jako fork verze oficiální.



Obrázek 4.1: Diagram sémantických dat v Strapi

Mnou implementované úpravy jsou součástí originálních kódů Strapi, které určitě nechci vydávat za vlastní. Co je a co není součástí mé praktické práce je nejlépe vidět na následující URL adrese `https://github.com/strapi/strapi/compare/master...kopecmi8:master`, kde je vidět přesný diff kódů obou verzí.

Schéma mých úprav jednotlivých částí a modulů Strapi můžete také zhlédnout na nákresu 4.1

4.4 Úpravy datového modelu

Jednou z nejdůležitějších změn, které jsem v rámci rozšiřování Strapi udělal, byly úpravy v datovém modelu, respektive ve způsobu definice datových modelů jednotlivých content types.

4.4.1 Přidání kontextu a typu

Jednotlivé obsahové typy jsem se podle svého návrhu rozhodl mapovat na entitní typy ze slovníku Schema.org. Abych toho, docílil přidal jsem do souboru s definicí modelu entity atribut *@type* po vzoru JSON-LD specifikace. Ten nyní definuje sémantický význam obsahového typu.

Další věcí, kterou bylo do definice modelu potřeba přidat, je definice kontextu. Ten se posléze používá při vypisování dat v souladu s použitím kontextu v JSON-LD specifikaci. V rámci něj používám vlastnost *@vocab*, která umožňuje zadefinovat výchozí prefix URL, která je použita pro všechny properties. To je pro mě výhodné, protože jsem se ve své implementaci zaměřil na používání jednoho slovníku Schema.org a stačí mi tak jeho URL zadefinovat právě jen v této vlastnosti a všude jinde už můžu používat zkratky jednotlivých typů bez uvádění tohoto prefixu.

4.4.2 Přetěžování typu property

Občas se také může stát, že v rámci modelu potřebujeme mít atribut jinak pojmenovaný, než je jeho přesné znění v URL. To je typický příklad už předdefinovaných atributů. V Strapi například všechny uložené záznamy obsahují identifikátor pojmenovaný jako *id*. S využitím zadefinované vlastnosti *@vocab* jako `http://schema.org/` tedy vznikne url `http://schema.org/id`. To je ale neplatné označení, slovník Schema.org totiž definuje pro identifikátory url `http://schema.org/identifier`. Kvůli tomu musím u některých již definovaných atributů typ explicitně určit. Způsob takovéhoto určení je znázorněn v ukázce kódu zjednodušené definice modelu File z pluginu Upload 4.2.

4.4.3 Kombinování i s nesémantickými properties

V rámci kontextu také využívám vlastnosti formátu JSON-LD a některým atributům nastavuji hodnotu *null*. Tento zápis totiž umožňuje, aby byl atribut ignorován při parsování podle JSON-LD specifikace, avšak byl stále součástí a čitelný jako JSON dokument [10].

To je výhodné, protože model Strapi obsahuje některé výchozí atributy, které nemají odpovídající sémantický typ a nešel by jim ani dodatečně přiřadit. Tohoto zápisu také využívám v případě mixování sémantických a nesémantických atributů. Styl toho zápisu můžete opět vidět na ukázce kódu4.2.

4.4.4 Modifikace jednotlivých atributů

Jak jsem již zmiňoval v části, která popisuje model Strapi, jednotlivé atributy jsou definovány v objektu attributes. Ten obsahuje objekty, jejichž název se originálně používá jak pro uložení property v databázi, tak také k automatickému generování popisků uživatelských vstupů v pluginu Content Manager.

Jednotlivé atributy ve svém sémantickém modelu mapuji na properties entit, a jejich název tedy musí přesně korespondovat s názvem property ve slovníku. Zvolený název ve slovníku však může být někdy nevhodný pro označení polí pro uživatele, je totiž jen v angličtině a někdy i ne zcela šťastně zvolený.

Do modelů jsem proto přidal vlastnost *label*, pomocí které může uživatel explicitně definovat popisky uživatelských polí. Názvy objektů se mi tak uvolnily a mohu je použít pro namapování na jednotlivé sémantické properties.

Další věcí, o kterou byly definice jednotlivých atributů potřeba rozšířit, je, že jednotlivé properties vyžadují položky specifického typu. Tato vlastnost je v slovníku označena jako *http://meta.schema.org/rangeIncludes* a právě po jejím vzoru jsem do definice atributu přidal objekt *range*, ve kterém se specifikuje zvolený typ, jež má property obsahovat.

Zmiňované úpravy nejen definic atributů jsou pak nejlépe patrny na ukázce modelu entity Person 4.3.

4.4.5 Přidání nových typů atributů

V sémantických datech může být property buď primitivního datového typu, ale častou situací je také to, že je typu jiné entity. V základním verzi svého modelu Strapi umožňuje řešit tuto situaci vytvořením vazby mezi dvěma obsahovými typy přes inverzní properties.

To je sice elegantní způsob, ale vzhledem k četnosti výskytu properties těchto komplexních typů v sémantických slovnících by tímto způsobem vznikalo spousty nových content types a administrace by se tak stala velmi nepřehlednou. I samotná editace dat by nebyla pro uživatele komfortní, protože by při plnění obsahu musel stále mezi jednotlivými typy přepínat.

```
{
  "info": {
    "name": "file",
  },
  "@context": {
    "@vocab": "http://schema.org/",
    "id": "http://schema.org/identifier",
    "mime": "htpp://schema.org/fileFormat",
    "size": "htpp://schema.org/contentSize",
    "hash": null,
    "ext": null,
  },
  "@type": "MediaObject",
  "attributes": {
    "name": {
      "type": "string",
      "configurable": false,
      "required": true
    },
    "hash": {
      "type": "string",
      "configurable": false,
      "required": true
    },
    "ext": {
      "type": "string",
      "configurable": false
    },
    "mime": {
      "type": "string",
      "configurable": false,
      "required": true
    },
    "size": {
      "type": "string",
      "configurable": false,
      "required": true
    }
  }
}
```

Ukázka kódu 4.2: Ukázka využití NULL hodnot v kontextu a definování typů jednotlivých properties

Atribut typu Entity Z těchto důvodů jsem rozšířil datový model o nový typ atributu Entita. Jedná se v podstatě o vnořený obsahový typ v rámci atributu. Jeho definice obsahuje objekt pojmenovaný *entity*, v rámci něhož opět můžeme definovat jednotlivé atributy, *@context* a typ entity (*@type*).

Tento nový atribut má sloužit pro entity, mezi nimiž je vazba 1:1. Díky tomuto novému atributu může být pak uživateli umožněno vyplňovat formulář strukturovaný podle jednotlivých properties vnořené entity a vytvořit tak properties neprimitivního datového typu bez zakládání nového obsahového typu.

Atribut typu URL Dalším typem atributu, o který jsem Strapi rozšířil, je URL. Jedná se v podstatě jen o rozšíření jednoduchého atributu typu string o validaci vstupu podle patternu URL adresy. Obdobně funguje i již existující typ atributu E-mail.

Využití tohoto atributu je podobné jako atributu typu Entita. URL adresa je totiž univerzálním identifikátorem pro entitu, a díky ní můžeme vyplnit property netriviálního typu, aniž bychom museli zakládat nový obsahový typ.

Dále pak tento typ atributu slouží ještě pro mapování properties typu *schema:URL*, které bych jinak musel mapovat na atribut typu text. V tom případě by nebylo zaručeno, že se skutečně jedná o URL adresu, a vstup by tak byl náchylný k uživatelským chybám.

4.5 Úpravy pluginu Content Manager

Jak jsem již zmiňoval v části věnované popisu Strapi, podle popisu datového modelu plugin Content Manager generuje uživatelské rozhraní pro plnění obsahových dat. Vzhledem k tomu, že jsem způsob popisu modelu rozšiřoval, musel jsem v patřičné návaznosti rozšířit a upravit i tento plugin.

4.5.1 Ukládání @context a @type

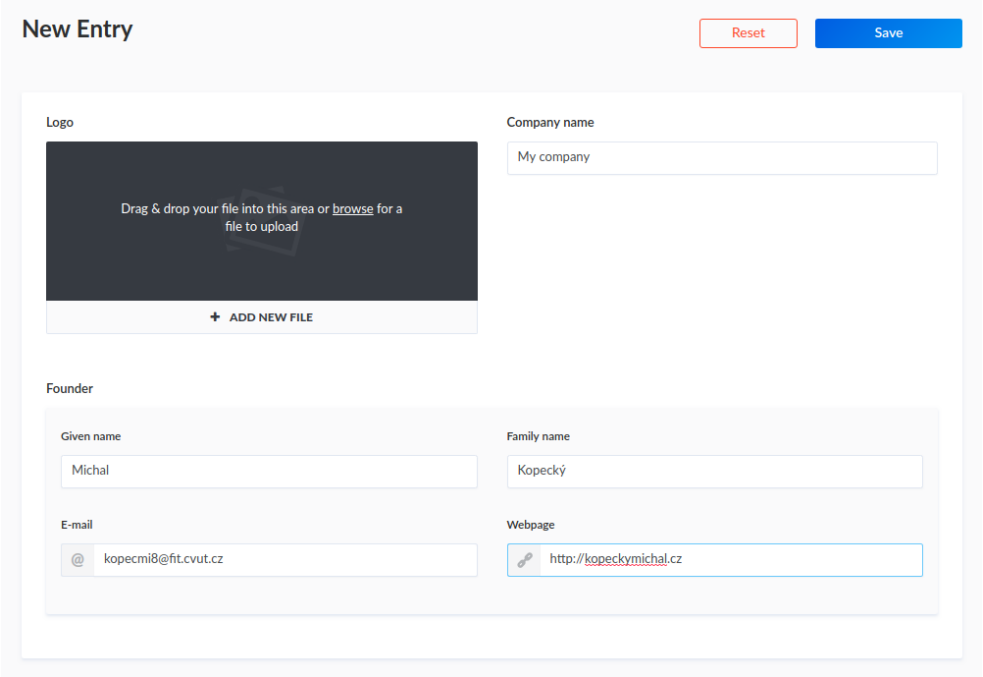
Jedním z hlavních důvodů, proč jsem chtěl, aby CMS, které budu rozšiřovat, používalo jako databázovou vrstvu MongoDB, byl fakt, že se v ní dá uložit přímo nativně JSON-LD. Toho jsem využil právě v úpravách tohoto pluginu.

Aby se tomu tak stalo a JSON, který Strapi ve svém základu ukládá do databáze, byl povýšen na JSON-LD, musel jsem začít spolu s daty ukládat mezi jednotlivé položky i *@context* a také *@type*, které jsou definovány v popisech jednotlivých modelů.

Velikost dat ukládaných v databázi sice tímto krokem naroste, ale i kdybychom nepoužili REST API, které generuje Strapi ze svého modelu, a napojili jinou aplikaci přímo na databázi, dostaneme data už sémantická, což považuji za větší výhodu, než kdybych *@context* a *@type* k datům přikládal až na aplikační úrovni při jejich vypisování.

```
{
  "connection": "default",
  "collectionName": "person",
  "info": {
    "name": "person",
    "description": "Family members"
  },
  "@context": {
    "@vocab": "http://schema.org/",
    "_id": "http://schema.org/identifier",
    "__v": null
  },
  "@type": "http://schema.org/Person",
  "attributes": {
    "givenName": {
      "multiple": false,
      "label": "Name",
      "type": "string",
      "range": "http://schema.org/Text"
    },
    "familyName": {
      "multiple": false,
      "label": "Surname",
      "type": "string",
      "range": "http://schema.org/Text"
    },
    "address": {
     "entity": {
       "@type": "http://schema.org/PostalAddress",
       "attributes": {
         "streetAddress": {
           "range": "http://schema.org/Text",
           "type": "string"
         },
         "postalCode": {
           "range": "http://schema.org/Text",
           "type": "string"
         }
       }
     },
     "type": "entity",
     "range": "http://schema.org/PostalAddress"
    },
    "worksFor": {
      "range": "http://schema.org/Organization",
      "model": "company",
      "via": "employees",
      "label": "Company"
    }
  }
}
```

Ukázka kódu 4.3: Ukázka rozšířeného modelu o nové atributní typy a vlastnosti



Obrázek 4.2: Formulář pro vyplnění vnořené entity typu schema:Person

Při editaci a vytváření jednotlivých položek jsou proto vždy načteny jak *@context*, tak i *@type* z modelu obsahového typu a jsou uloženy spolu s obsahovými daty přímo do databáze.

4.5.2 Přidání podpory pro nové typy atributů

Kvůli lepší správě strukturovaných dat jsem vymyslel dva nové typy atributů URL a Entita. Pro ně bylo také potřeba patřičně rozšířit tento plugin, aby umožňoval editaci jejich obsahu.

Zatímco přidání podpory pro typ URL bylo celkem triviální rozšíření typu string o validaci tvaru vyplněných dat podle regulárního výrazu odpovídajícího URL, přidání podpory pro typ Entita bylo mnohem náročnější.

Definice atributu entity obsahuje totiž definice ostatních atributních typů, tedy až na typ atributu vazby na jiný obsahový typ. To je z důvodů, že při vytváření vazby se musejí oba obsahové typy provázat přes jednotlivé properties a to by ve vnořených poperties nefungovalo.

Vnořený atribut však může být opět typu Entita a takto můžeme postupovat hlouběji a hlouběji. Samotné vykreslování tohoto formulářového prvku je proto potřeba provádět rekurzivně. Jednotlivé formulářové prvky odpovídajícím vnořeným atributním typům jsou proto sdružené do skupiny, která reprezentuje vždy jednu entitní property. Jak takovýto formulářový prvek může vypadat nejlépe uvidíte na obrázku 4.2.

Data z něj jsou poté ukládány ve formátu JSON, který svou strukturou vytváří objekt, jenž kopíruje jména jednotlivých vnořených atributů.

Vzhledem k tomu, že Strapi podporuje kromě MongoDB i klasické relační databáze, tento prvek by se v nich musel ukládat do jednoho sloupce s datovým typem JSON. Toto použití jsem však neřešil a pro podporu i ostatních databází by bylo potřeba ještě doplnit kód aplikace o jakýsi adaptér. Proto je nutné zatím moji verzi Strapi používat výhradně s MongoDB.

4.6 Úpravy pluginu upload

Jak už z názvu toto pluginu vyplývá, jeho účelem je nahrávání souborů a jejich přikládání k jednotlivým příspěvkům. Tento plugin tvoří defaultní content type nazvaný *File*. Instance tohoto obsahového typu pak reprezentují jednotlivé soubory. Stejně jako u uživatelských obsahových typů i tento model je definován JSON souborem. Toho jsem využil ve svých úpravách k tomu, abych tento model obohatil také o *@content* a *@type* a udělal z něj také typ sémantický.

Tento model reprezentuje jakýkoliv typ souboru. Zvolil jsem proto jeho *@type* jako *schema:MediaObject*, což je typ, který je v podstatě abstraktním předkem konkrétních typů, jako jsou *schema:AudioObject*, *schema:VideoObject* a *schema:ImageObject*.

U tohoto content type pak probíhá určení specifičtějšího typu, na rozdíl od všech ostatních, ještě při nahrávání konkrétního souboru podle MIME type. Pokud se však nepodaří MIME type rozpoznat a určit ho mezi typy Audio, Video nebo Obrázek, je použit výchozí typ *schema:MediaObject*, který je abstraktní reprezentací souboru.

4.6.1 Typ atributu Media

Mít jen centrální úložiště souborů by však nebylo moc užitečné. Tento plugin proto také přidává do Strapi nový typ atributu Media, díky kterému pak můžeme jednotlivé soubory provazovat s uživatelskými obsahovými typy. Toto propojování na pozadí probíhá obdobně jako tvorba relací mezi dvěma uživatelskými obsahovými typy. Zde si však uživatel může pouze zvolit, zda-li chce podporovat nahrávání více souborů, a je odstíněn od dialogu pro tvorbu relací.

Tento plugin je však velmi mladý a jeho vývoj dokonce započal až v době tvorby mé práce. Jeho možnosti jsou proto prozatím dost omezené. V rámci atributu Media například bohužel neumožňuje určení typu souborů, které mohou být ke konkrétnímu obsahovému typu nahrávány.

To má za následek to, že pokud property jiné entity jako svůj obor hodnot definuje některý z konkrétních podtypů, jako je například *schema:ImageObject*, nejsem schopen zabránit tomu, aby nebyl přiřazen do tohoto atributu i objekt jiného typu, například *schema:VideoObject*. Je tedy jen na uživateli, aby do-

držel odpovídající typ souborů, které k property patří, a není to zatím nijak programově omezeno.

4.7 Úpravy pluginu Content Type Builder

Nejvýznamnější a největší částí mojí praktické práce jsou úpravy právě tohoto pluginu. Tento plugin slouží už v základní verzi Strapi k modelování ontologií. Mým cílem bylo upravit tento plugin tak, aby byl schopen generovat mnou rozšířené definice sémantických modelů jednotlivých obsahových typů, ale také aby zároveň tvorba modelů probíhala podle entitních typů ze slovníku Schema.org. Cílem mých úprav bylo, aby tento plugin poskytoval uživatelům grafický nástroj zjednodušující modelování ontologií založených na sémantických datech.

Původně jsem zamýšlel tuto funkcionalitu udělat jako plugin úplně nový, abych nebyl při jeho návrhu ničím omezován a limitován. Nejprve jsem řešení takto začal i implementovat.

V průběhu tvorby jsem však narazil na to, že vytváření modelů je závislé i na dalších core modulech, jako je například generátor API, jejichž funkcionalitu jsem musel také upravit. Pouhé vytvoření uživatelského rozhraní nového pluginu proto nestačí.

Další věcí hovořící ve prospěch úprav tohoto pluginu bylo i to, že je součástí každé instalace Strapi. Pokud bych si vystačil pouze s jeho úpravami, tak bych dosáhl užší integrace sémantických dat přímo do jádra systému.

Třetím faktorem, co ovlivnilo moje rozhodování, byla úvaha i o budoucí udržitelnosti rozvoje. Vzhledem k tomu, že moje verze je nyní fork té originální, můžu díky tomu, že upravuji původní kód, stále celkem jednoduše zahrnout i případné aktualizace tohoto core pluginu.

V případě potřeby se ostatně dá z pluginu udělat i jiný, nový přejmenováním patřičných namespaces. Takto osamostatněný plugin však stejně nemůže fungovat bez patřičně upravených i dalších pluginů, a jeho osamostatnění je tedy v současném stavu zbytečné.

Nakonec mi cesta úpravy stávajícího pluginu dávala proto větší smysl a svoje řešení jsem implementoval jako jeho rozšíření. Tolik tedy k úvodu a teď už k samotným úpravám.

4.7.1 Definování sémantického typu

Jak jsem již zmiňoval, originální obsahové typy mapuji ve svém řešení na jednotlivé entitní typy ze slovníku Schema.org. Při zakládání obsahového typu musí tedy uživatel jasně zvolit přidružený sémantický typ. Za tímto účelem jsem do formuláře pro vytváření a editaci obsahových typů přidal komponentu selectbox, která uživateli dovoluje právě tento typ zvolit. Tyto typy získávám díky JS knihovně schema.org.

Vzhledem k tomu, že entitních typů existuje ve slovníku Schema.org několik stovek, standardní formulářový HTML prvek select by byl velice nepřehledný. Zakomponoval jsem tedy za tímto účelem do Strapi komponentu React-Select, která umožňuje mezi prvky vyhledávat, díky čemuž uživatel snadněji nalezne odpovídající sémantický typ. Tato komponenta umožňuje také při patřičné konfiguraci vícenásobný výběr, což se mi hodilo i v další části úprav.

Volbu sémantického typu uživatel může provést však jen při zakládání nového obsahového typu a nemůže ho už měnit při jeho úpravách. Ve chvíli, kdy už je typ založen a obsahuje nějaké atributy, ty by se mohly pro různé entitní typy rozcházet a muselo by se tedy detekovat, které se mají smazat a které ne. Toto řešení by bylo zbytečně komplikované a pro uživatele i matoucí.

Tuto volbu jsem dal jako povinnou. A to i přes to, že jsem musel v Strapi zachovat podporu i nesémantických content types kvůli existenci těch výchozích, které sémantický typ nemají určený a je i nevhodné ho určovat, protože slouží jen k interním účelům. Po vzoru slovníku Schema.org je totiž každý typ potomkem typu *schema:Thing*, což je nejabsrtaktnější typ a dá se tedy použít pro jakýkoliv content type.

4.7.2 Vybírání typu atributu

Po založení obsahového typu má uživatel v originální verzi Strapi při vytváření nového pole k dispozici nejprve dialog s možností volby typu atributu. V mé sémantické verzi jsem podobně udělal dialog pro vytváření sémantické property.

V něm však potřebuji nejprve znát typ property, na kterou sé ma daný atribut vztahovat. Obdobně jako u volby enitního typu i zde používám nový formulářový prvek selectbox s vyhledáváním, do kterého dynamicky načítám všechny properties, kterých může daný typ nabývat.

Tato data získávám však přímo požadavkem na patřičnou URL adresu konkrétního sémantického typu. Dochází zde proto k mírné latenci, protože na rozdíl od získávání jednotlivých typů musí tato komunikace proběhnout přes internet. K získávání těchto dat využívám odpověď ve formátu JSON-LD, kterou patřičně zpracuji a přetvořím na seznam všech dostupných properties pro daný sémantický typ.

Po zvolení konkrétní property mohou nastat dvě situace. Slovník Schema.org totiž definuje obor hodnot, kterého může konkrétní properta nabývat. To však může být jedna nebo více hodnot.

Pokud property může nabývat více hodnot, uživatel si musí zvolit konkrétní typ, který chce používat. V opačném případě, kdy obor hodnot obsahuje jen jeden možný typ, není potřeba nic zadávat a obor hodnot je vybrán automaticky a uživateli je jen zobrazena informace, který obor hodnot se používá.

Tak či tak výsledkem je zvolený obor hodnot, což je opět URL adresa nějakého typu entity ze slovníku Schema.org. Zde přichází na řadu mapování

těchto typů oboru hodnot na jednotlivé typy atributů. To probíhá následujícím způsobem:

schema:Text se mapuje na atributní typy *string*, *text* nebo *e-mail*. Podle typu property však později ještě tyto typy atributů filtruji, a pokud se jedná o property typu E-mail, ponechám pouze atributní typ *e-mail*. U ostatních typů ho naopak vynechám.

Rozdíl mezi typy String a Text je jediný. Pro typ String se používá formulářový prvek input typu text a pro Text se používá formulářový prvek typu textarea, který lépe poslouží pro zadávání delších textových vstupů a zároveň může být zobrazen formou WYSIWYG editoru.

schema:URL se mapuje na atributní typ *URL*, který jsem mimo jiné vytvářel i za tímto účelem. V tomto případě se jedná o jednoduchý formulářový prvek input typu URL, který obsahuje validaci vstupu podle patternu URL adresy.

schema:Number*, *schema:Float* a *schema:Integer tyto typy se mapují na atributní typ *number*, u kterých pak už ručně uživatel musí zvolit příslušný číselný typ. Zde je proto prostor pro uživatelskou chybu a v budoucnu by se dalo toto určování provádět automaticky.

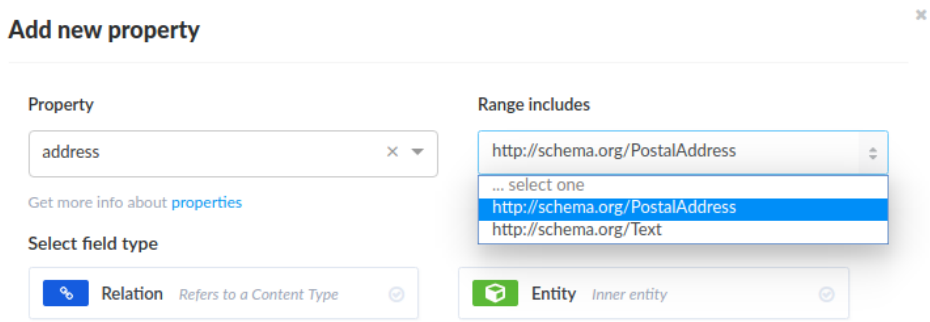
schema:Boolean*, *schema:True* a *schema:False se mapují na atributní typ *boolean*, který je poté reprezentován jako formulářový prvek checkbox, jehož zaškrtnutím či nezaškrtnutím určujeme příslušnou boolenovskou hodnotu. V případě typů *schema:True* nebo *schema:False* by se například mohla vyplňovat výchozí hodnota tohoto atributu, to se však v současné verzi implementace neděje.

schema:Date*, *schema:DateTime* a *schema:Time se mapují na atributní typ *date*. I zde by byl prostor pro zlepšení. Atributní typ *date* totiž ve skutečnosti nejlépe odpovídá typu *schema:DateTime*, pro zbylé dva by se dalo omezit vybírání uživatelského vstupu jen na jednu složku.

schema:MediaObject*, *ImageObject*, *VideoObject* a *AudioObject se mapují na atribut typu *media*, který je obsluhován pomocí pluginu Upload a slouží pro nahrávání souborů různých datových typů.

Všechny ostatní entity se pak mapují buď na atributní typ *relation* nebo *entity*, které slouží k pokrytí neprimitivních datových typů.

Po zvolení typu property a výběru jejího oboru hodnot je proces tvorby obsahového typu napojen zpět na původní princip fungování. Uživatel si tedy musí zvolit typ atributu, seznam je však nyní vyfiltrovaný podle příslušného oboru hodnot. Má tedy na výběr jen atributní typy, které dávají pro danou property smysl. Jak takový dialog vypadá, nejlépe uvidíte na obrázku 4.3.



Obrázek 4.3: Výběr typu property

Až na poslední dva zmiňované atributní typy je následující proces tvorby atributů skoro shodný s originální verzí Strapi. Jediným výraznějším rozdílem je to, že název atributu je již předurčen a nelze měnit.

Dialogy jak pro tvorbu sémantické property, tak pro tvorbu nesémantického atributu jsem ještě rozšířil o input pro zádávání label, protože ten může nově sloužit k explicitnímu zadání popisku atributu v administraci.

4.7.3 Tvorba atributu typu relace

V originální verzi Strapi sice tento typ atributu existuje a moje řešení z něho také vychází, v nesémantické podobě modelování však není potřeba rozlišovat, mezi kterými obsahovými typy je možné relaci utvořit, a dialog proto nabízí možnosti volby z jakéhokoliv obsahového typu.

To je ovšem nevhodné v sémantické podobě, kdy je relace pevně spjata s významovým typem atributu. Výběr obsahových typů, se kterými může být relace utvořena, je tedy v případě tvorby sémantické relace filtrován podle zvoleného oboru hodnot dané property, díky definovanému *@type* u jednotlivých content types.

Díky tomu se však na rozdíl od originální verze může stát, že vhodný cílový obsahový typ ještě neexistuje a je potřeba ho založit, aby bylo možné relaci vytvořit. K tomu účelu jsem dialog upravil tak, že pokud neexisuje žádný odpovídající obsahový typ, je v něm zobrazeno tlačítko. To uživateli otevře standardní dialog k založení nového obsahového typu, avšak s již předvoleným *@type* na odpovídající hodnotu, kterou si vynucuje daná property.

Relace jsou vždy vytvářeny mezi dvěma obsahovými typy a jsou možná trochu překvapivě mapovány přes atributy na každé straně relace i v případě, že se jedná o relaci typu 1:N. To má za následek to, že je potřeba vždy volit také inverzní property odpovídající významu dané relace na entitě, na kterou naší vazbou cílíme.

```
{
"@context": {
"@vocab": "http://schema.org/",
"writeABook": {
"@reverse": "http://schema.org/author"
}
},
"@type": "http://schema.org/Person",
"name": "Michal Kopecky",
"writeABook": {
"@type": "http://schema.org/Book",
"name": "Diplomova prace"
}
}
```

Ukázka kódu 4.4: Ukázka využití vlastnosti @reverse

K tomuto účelu jsem přidal do dialogu výběrový box, který obsahuje všechny properties cílové entity, jejichž obor hodnot může nabývat typu stejného jako má počáteční entita. Uživatel si sám musí zvolit property, která nejlépe odpovídá sémantickému typu vazby.

V mém rozšíření datového modelu do každé property ukládám i informaci o oboru hodnot, kterého má property nabývat. Při vytváření vazby je proto potřeba do cílového atributu zanést také informaci o jeho oboru hodnot. Ten je automaticky určen a odpovídá právě typu entity, ze které vazba vede. Díky této vlastnosti se pak může relace spravovat z obou stran v příslušných obsahových typech.

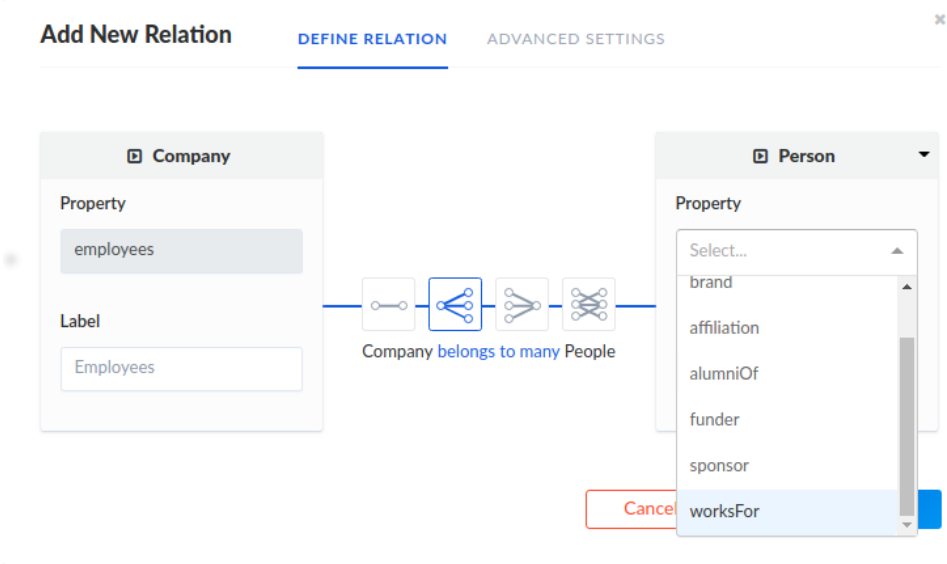
4.7.3.1 Neexistující inversní property

Ve slovníku Schema.org však častokrát narazíme na to, že property existuje jen u počáteční entity a u cílové entity odpovídající inverzní vlastnost vůbec neexistuje.

Například typ *schema:Book* má property typu *schema:author*, jejíž obor hodnot může být buď *schema:Person*, nebo *schema:Organization*. Oba dva zmiňované typy cílové entity však nemají žádnou property, jejíž obor hodnot by odpovídal typu *schema:Book*. To má za následek to, že uživateli ve výběrovém boxu pro výběr inverzní property nemám co zobrazit, a uživatel by tak relaci nemohl vůbec vytvořit.

Naštěstí i na toto je syntaxe JSON-LD připravena. V tomto případě se využije vlastnosti *@reverse*, díky níž mohu do *@context* uložit informaci o tom, že daná property je inversní k té na počáteční straně relace.

Když toto nastane, uživatel si v dialogu nevolí cílovou property z výběrového boxu, ale místo toho mu zobrazím textové pole, pomocí něhož vyplňuje svůj vlastní název, který může být v podstatě libovolný.



Obrázek 4.4: Tvorba relace mezi dvěma obsahovými typy

Budu-li se držet příkladu s knihami a pokud si uživatel zvolí jako obor hodnot cílové entity typ *schema:Person*, může například nastavit název property na *writeABook* a do kontextu se pak uloží informace v podobě *"@reverse": {"writeABook": "http://schmea.org/author"}*. Díky tomu se pak při vypisování dat vymění objekt se subjektem a použije se místo naší vlastní property *writeABook* jako predikát property reversní, tedy *shema:author*. Tento příklad je také zobrazen v ukázce kódu 4.4.

4.7.3.2 Násobnost relace

Dále je také potřeba zvolit násobnost relace. To ostatně umožňuje Strapi už ve své základní verzi. Násobnosti jednotlivých properties ovšem nejsou přímo ve slovníku Schema.org nijak definovány, a proto se o jejich automatické určování nepokouším.

Jejich předvyplnění není nereálný úkol. Násobnost relace by se dala odvozovat například z názvů properties, které mohou nést význam o násobnosti vazby v podobě množného či jednotného čísla. Také jsem objevil JSON soubor, kde se někdo pokusil pro všechny properties jejich násobnost určit manuálně. Vzhledem ke statické povaze tohoto souboru však není zaručeno, že obsahuje opravdu všechny properties, protože slovník Schema.org se neustále vyvíjí.

I z těchto důvodů jsem ve svém řešení ponechal tuto volbu na samotném uživateli, tak jak to je i v originální verzi a může to být předmětem dalších vylepšení. Jak celý upravený dialog pro tvorbu relací vypadá, můžete zhlédnout na obrázku 4.4.

4.7.4 Tvorba atributu typu entita

Atributní typ *Entita* je mnou přidaný nový typ v základní verzi pluginu Content Type Builder, tedy uživatel nemá možnost vůbec vytvářet atributy tohoto typu. Tímto atributem je uživateli umožněno vytvářet pouze sémantické properties.

Dialog pro tento atributní typ jsem navrhl zcela nový. V rámci něho je potřeba uživateli umožnit vytvářet v podstatě celý vnořený obsahový typ.

K určení sémantického typu (*@type*) vnořeného obsahového typu se použije zvolený obor hodnot pro danou property. Dopředu v dialogu můžeme načíst všechny properties, kterými daný typ disponuje. Uživateli zobrazuji podobně jako při tvorbě samotného atributu výběrový box, kde je umožněno mezi těmito properties vyhledávání a několikanásobný výběr. Díky tomu uživatel může v rámci dialogu provést snadný výběr odpovídajících properties, které chce ve vnořeném obsahovém typu poskytovat.

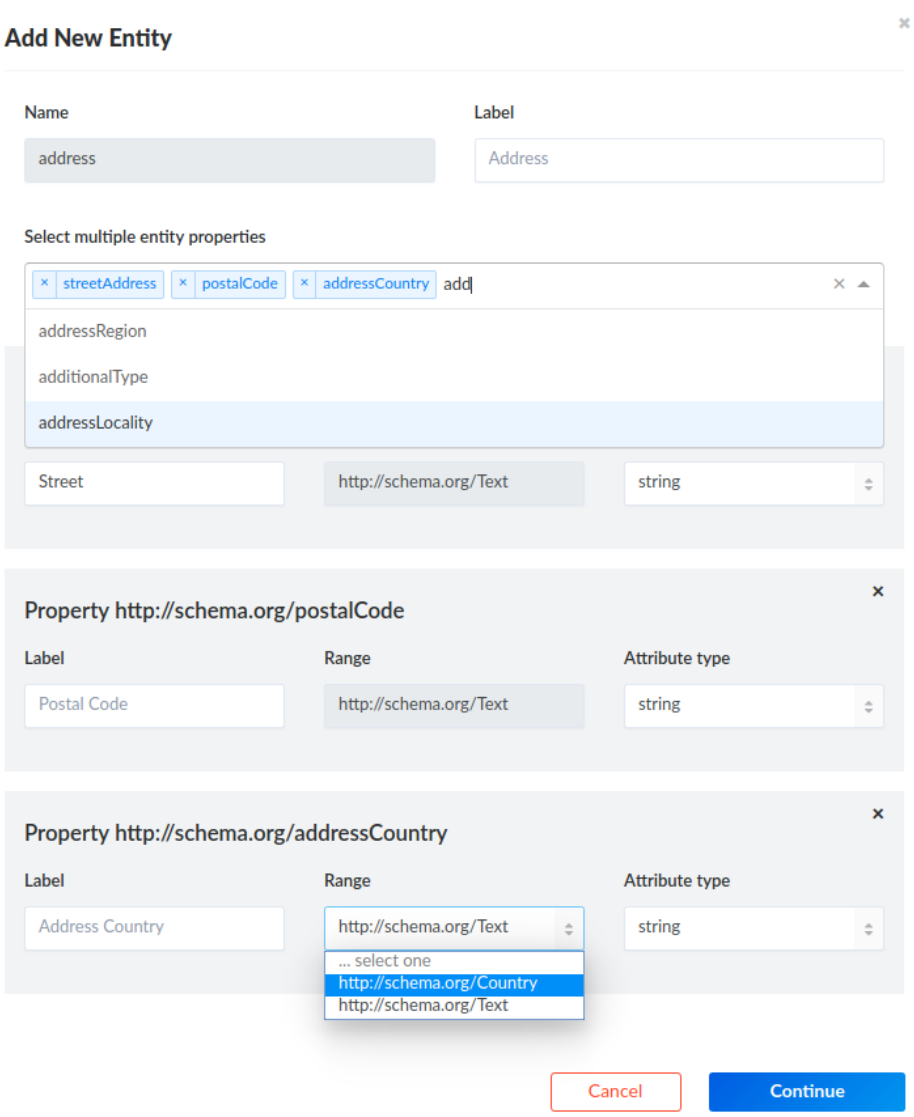
Podle těchto zvolených properties se pak dynamicky generuje formulář pro jednotlivou property. V této opakující se části formuláře je uživateli umožněno zvolit si jednak label property, a pokud property může nabývat více typů, tak i obor hodnot pomocí výběrového boxu. Podle tohoto zvoleného oboru hodnot, podobně jako při vytváření property u obsahového typu, nabízím výběrový box pro zvolení typu atributu.

Možnosti volby typu atributu jsou na této úrovni mírně omezeny a není zde možné si zvolit atribut typu relace a entity. Typ relace je zde zakázán, protože Strapi potřebuje k utvoření vazby dva inverzní atributy na každé straně obsahového typu a tady se už vyskytujeme ve vnořeném atributu. Zde by těchto relací mohlo teoreticky vznikat více. Navíc obsah celého tohoto atributu je poté ukládán jako jeden velký JSON a vytvoření vazby do jeho části by prostě nešlo. Typ atributu relace je proto na této úrovni vyfiltrován.

Situace je trochu jiná u atributu typu Entita. To, jak jsem ho vytvořil, totiž umožňuje klidně rekurzivní zanořování těchto typů atributu do sebe. Problém však nastává při jejich vytváření a editaci v rámci modelování content types.

Zde bych musel skládat dialogy navrch přes sebe a nebylo by to pro uživatele vůbec přehledné. V grafické podobě vytváření modelů jsem se tedy rozhodl, že nebudu tuto možnost povolovat, avšak znalý uživatel si stále může takto model namodelovat manuálně pomocí konfiguračního JSON souboru.

Tím, že jsem ale zakázal použití těchto dvou atributních typů, bych se ovšem skoro připravil o možnost řešení složitých datových typů, které se nedají mapovat na primitivní formulářové prvky. Právě za tímto účelem pak používám také vlastní nový typ atributu URL, protože právě URL adresa může v případě potřeby reprezentovat celou další entitu.



Obrázek 4.5: Dialog pro vytváření property typu entita

4.8 Úpravy modulu pro generování API

Posledním modulem, který jsem musel v rámci rozšíření o sémantické funkce upravit, byl modul Strapi Generate API. Ten, jak už název napovídá, slouží pro generování souborů jednotlivých obsahových typů a jejich API. Zde jsem nejprve musel zahrnout změny, které jsem udělal v rámci úprav pluginu Content Type Builder, protože právě pomocí tohoto modulu se jednotlivé modely generují. V rámci toho jsem upravil šablonu pro vytváření konfiguračního souboru modelu hlavně tak, aby podporovala přidání *@context* a *@type*.

Další změnou toho modulu byla úprava podoby generovaných funkcí kontrolérů jednotlivých modelů. Ty jsou totiž mapovány přes konfigurační soubor na jednotlivé URL a slouží k poskytování dat ze systému pomocí REST API. Právě zde bylo proto potřeba upravit příslušné funkce, abych byl schopen navracet obsah i v jiném formátu, než je jen výchozí JSON-LD. Využil jsem proto oficiální implementační balíček jsonld pro JavaScript, který umožňuje nejen snadný převod mezi jednotlivými reprezentacemi JSON-LD (compacted, expanded, flattened), ale také převod do serializace N-Quads.

API Strapi tedy nadále poskytuje data ve výchozím nastavení ve formátu JSON-LD, avšak pokud s naším požadavkem odešleme i příslušnou hlavičku *Accept* nastavenou na *application/n-quads*, dostaneme data převedená právě do této serializace. To podle mě zvyšuje využitelnost API, protože nenutí vývojáře k použití jednoho konkrétního formátu, a mohou si vybrat formát, který je pro jejich aplikaci přirozený a nemusejí se starat o převod dat uvnitř vlastní aplikace.

Ve svém návrhu jsem původně počítal s podporou ještě více serializací, od které jsem nakonec upustil. To i kvůli tomu, že zmiňované změny kontrolérů jsem musel provést opět v rámci šablon a připojit kvůli tomu patřičný balíček *jsonld* přímo do konfiguračního souboru *package.json*, který slouží pro celou aplikaci. Pro podporu ostatních formátů bych musel připojit další balíčky, a celá aplikace by tak měla zbytečné závislosti, které by většina uživatelů ani nevyužila.

Beru proto převod do serializace *n-quads* jako vhodnou ukázku. V případě potřeby si mohou samotní uživatelé připojit i další balíček, který převod do jimi požadované serializace podporuje. Samotný převod si mohou doimplementovat podle sebe v rámci jednotlivých API modelů, a nemusejí proto stále implementovat převod až v rámci vlastní aplikace.

```json
{
  "@context": {
    "@vocab": "http://schema.org/",
    "_id": "http://schema.org/identifier",
    "__v": null,
    "createdAt": null,
    "updatedAt": null
  },
  "@type": "http://schema.org/Person",
  "_id": "5aeb1f75d2181769ecfa4c0a",
  "givenName": "Michal",
  "familyName": "Kopecky",
  "birthDate": "1994-07-18T00:00:00.000Z",
  "address": {
    "streetAddress": "Horni Rybniky",
    "postalCode": "54941",
    "addressLocality": "Zabrodi",
    "@type": "http://schema.org/PostalAddress"
  },
  "createdAt": "2018-05-03T14:40:53.971Z",
  "updatedAt": "2018-05-03T14:40:53.971Z",
  "__v": 0
}
```

Ukázka kódu 4.5: Ukázka výstupu REST API ve formátu JSON-LD

```
_:b0 <http://schema.org/address> _:b1 .
_:b0 <http://schema.org/birthDate> "Mon Jul 18 1994 02:00:00 GMT+0200 (CEST)
    " .
_:b0 <http://schema.org/familyName> "Kopecky" .
_:b0 <http://schema.org/givenName> "Michal" .
_:b0 <http://schema.org/identifier> <http://localhost:1337/person/5
    aeb1f75d2181769ecfa4c0a> .
_:b0 <http://www.w3.org/1999/02/22-rdf-syntax-ns#type> <http://schema.org/
    Person> .
_:b1 <http://schema.org/addressLocality> "Zabrodi" .
_:b1 <http://schema.org/postalCode> "54941" .
_:b1 <http://schema.org/streetAddress> "Horni Rybniky" .
_:b1 <http://www.w3.org/1999/02/22-rdf-syntax-ns#type> <http://schema.org/
    PostalAddress> .
```

Ukázka kódu 4.6: Ukázka výstupu REST API ve formátu N-QUADS

Kapitola 5

Testování

Implementované změny jsem chtěl podrobit uživatelskému testování, které by mělo ověřit, zda-li se mi povedlo udělat řešení, které je pro uživatele pochopitelné a použitelné. Dlouho dobu jsem přemýšlel z jakého konce toto testování uchopit. Nakonec jsem se rozhodl pro testování použitelnosti. V rámci něho jsem otestoval hlavně upravený plugin Content Type Builder, pomocí něhož testeři vytvářeli netriviální konceptuální model.

Trik spočíval v tom, že jsem jim před testováním nesdělit, že se jedná o sémantickou aplikaci. Tím jsem chtěl mimo jiné docílit toho, zda-li vůbec uživatelé musejí poznat, že jsou na pozadí strukturovaná data, a jestli kvůli tomu tvorba sémantického modelu musí být nutně pracnější.

5.1 Příprava testování

V rámci příprav bylo zapotřebí vytvořit hlavně testovací scénář, podle kterého testování probíhalo. V tomto scénáři testerům nastiňuji modelovou situaci, kde si mají představit, že jsou webovými vývojáři a dostali za úkol namodelovat konceptuální model pojednávající o filmové databázi pomocí grafického nástroje, který jim k tomu Strapi poskytuje.

Tento model je v rámci scénáře podrobně popsán a vyžaduje důkladné použití mnou upraveného pluginu Content Type Builder. V rámci scénáře by testeři měli narazit i na krajní situace, kdy se musejí potýkat s neexistující inversní property, zakládat nové content types a také použít můj dialog pro vytváření vnořené entity. Přesné znění testovacího scénáře můžete zhlédnout v příloze C.

Vzhledem k tomu, že testeři budou simulovat činnost webového vývojáře, by se mělo jednat o programátory nebo alespoň osoby s nadprůměrnou znalostí IT. S prosbou o otestování jsem proto oslovil své spolužáky a kolegy.

5.2 Průběh testování použitelnosti

Testování jsem prováděl na svém osobním počítači se Strapi rozběhlém na lokálním serveru. Pro každého uživatele jsem měl připravenou úplně čistou instalaci. K testování se mi podařilo sehnat 5 lidi, nejedná se tedy o příliš velký vzorek, ale i tak si myslím, že se testováním odhalilo spoustu nedostatků, které bych si sám těžko uvědomil.

U testování jsem byl vždy přítomen a fungoval jsem jednak jako jeho moderátor, ale zároveň jsem i bedlivě sledoval, jak testeři při práci s aplikací postupují. Do testování jsem se snažil co nejméně zasahovat, a byl jsem ku pomoci, až když si o to testovaná osoba vyloženě řekla.

Samotné testování průměrně trvalo cca 30 minut. Na jeho konci si pak uživatelé měli ještě možnost zkusit proklikat vygenerované uživatelské rozhraní na základě jimi vytvořeného modelu. Zpětně jsem jim pak řekl, že testovali sémantickou aplikaci, a ukázal jsem jim základní myšlenku fungování na pozadí.

5.3 Vyhodnocení testování a zjištěné problémy

Díky uživatelskému testování jsem narazil na řadu problémů, které bych při práci s aplikací sám jen těžko odhalil. I když se jednalo vesměs o drobnosti, ve svém součtu a důsledku někdy mátly uživatele, a proces tvorby ontologie se proto neobešel bez obtíží.

Ne všechny tyto problémy byly však způsobeny přímo mými změnami. Vzhledem k tomu, že používám upravený plugin Content Type Builder, uživatelé častokrát naráželi i na problémy, které obsahuje přímo originální verze tohoto pluginu.

I přes tyto problémy si ale myslím, že proces tvorby sémantických ontologií byl pro uživatele vcelku srozumitelný a všem se zdárně podařilo namodelovat sémanticky správný model, aniž by o sémantických datech cokoliv předtím věděli. To považuji za úspěch a za potvrzení toho, že proces modelování sémantických ontologií jde před uživatelem skrýt.

V rámci testování jsem odhalil i několik chyb ve funkcionalitě programu, těm se zde ale nechci věnovat. V následující sekci popíšu objevené chyby v návrhu uživatelského rozhraní aplikace a možnosti jejich možného odstranění.

5.3.1 Špatný popisek tlačítka při zakládání nového Content Type

Hned na začátku testování častým problémem bylo, že při zakládání nového content type uživatele mátl fakt, že tato akce je vícekroková. Poté, co správně vyplnili *@type* a název, hledali marně v dialogu ještě prostředek na vytváření jednotlivých atributů tohoto nového obsahového typu.

Z toho jsem usoudil, že může být matoucí, že tlačítko dialogu neslo název Uložit. To je sice věc, která vzešla přímo z originální verze Strapi, ale kvůli tomu, jaká to je drobnost, jsem se rozhodl tento popisek změnit na Pokračovat, aby lépe odpovídal skutečné akci, která se poté děje.

Tuto drobnou úpravu jsem zahrnul už do druhé vlny testování a skutečně to pomohlo v chápání vytvářecího procesu. Uživatelé se už nebáli na tlačítko kliknout a postupovali rychleji v procesu vytváření.

5.3.2 Vytváření relace s ještě neexistujícím content type

Pokud uživatel vytváří relaci mezi dvěma content types, kvůli vynucování sémantického typu se v mé verzi může stát, že odpovídající typ obsahu ještě neexistuje. Z těchto důvodů jsem upravil dialog k vytváření relací tak, že nabídne uživateli tlačítko, které mu otevře dialog pro založení nového obsahového typu s předvyplněným sémantickým typem.

Co jsem už nedomyslel, je to, že property, kterou začal u prvního content type zakládat, se mu nikde neuloží. To je z důvodu, že k samotnému provázání může dojít, až když oba content types existují. Uživatelé byli ale tímto velmi zmateni a měli pocit, že by se property měla vytvořením nového obsahového typu také vytvořit.

Možným řešením tohoto problému by mohlo být uložení si stavu o začátku vytváření relace do lokálního úložiště. Při návratu na stránku původního content type, který uživatel opustil kvůli zakládání nového, by se pak dialog automaticky zobrazil už předvyplněný a obnovený z původního stavu.

Lepší řešení mě nenapadá, protože vytváření vazby totiž nemohu automaticky dokončit kvůli tomu, že uživatel si musí ještě manuálně zvolit inverzní property na nově vzniklém content type. Ani toto řešení jsem však zatím nestihl implementovat.

5.3.3 Nemožnost vrátit se o krok zpět

Vytváření atributu je dvoukrokové, což vychází ze způsobu, jakým tento proces probíhá v originálním Strapi. Při testování jsem však narazil na to, že uživatel tím, jak se ještě moc neorientoval ve všech možnostech atributů, častokrát zvolil jiný typ, než chtěl skutečně použít. Když pak ve druhém kroku klikl na tlačítko Cancel, očekával, že se vrátí zpět do prvního dialogu, což se ovšem nestalo. Pokud chtěl tedy udělat nějakou změnu v kroku číslo jedna, musel proces výběru property a typu atributu opakovat.

Tuto chybu nepovažuji za nijak kritickou a jako její řešení bych spíše volil přidání vhodně umístěného tlačítka Zpět, které by umožňovalo návrat do prvního kroku při vytváření atributu. Tlačítko Cancel totiž slouží k vystornování úprav i při modifikaci atributu a tam už není návrat do prvního kroku možný. Bylo by podle mě tedy matoucí jeho funkcionalitu odlišovat podle toho, zda-li se jedná o zakládání nebo editaci atributu.

5.3.4 Neukládání změn při přechodu mezi jednotlivými content type

Tato chyba není také primárně způsobena mými úpravami, avšak narazili na ni všichni testeři a považuji ji za celkem zásadní. Plugin totiž vytvořené properties ihned neukládá do konfiguračních JSON souborů, ale ponechává si je v paměti prohlížeče. To je z důvodu, že uložení do konfiguračních souborů potřebuje resetovat server, aby naběhl s novou konfigurací, a tato operace je tedy náročná a provádí se až explicitně na vyžádání uživatele při stisku tlačítka uložit.

Problém nastává, že pokud upravujeme obsahový typ, stačí se například překliknout do stránky s jiným content type, a o neuložené úpravy bez jakéhokoliv varování přijdeme. To považuji za nepřijatelné a Strapi by mělo alespoň o možné ztrátě dat varovat. Situace je ještě nepříjemnější v tom, že pokud zakládám nový content type, který ještě neexistuje, tak v tomto jediném případě ke ztrátě dat nedojde, protože data content type jsou uložena v lokálním úložišti prohlížeče. O to víc je pak však uživatel zmatený, když ke ztrátě neuložených úprav dojde u již existujícího obsahového typu.

Po testování jsem tedy aplikaci upravil tak, aby když k této situaci dojde, byl uživatel varován modálním oknem s textem, že může dojít ke ztrátě neuložených úprav a umožní mu akci ještě zavčasu zrušit a provedené změny si uložit.

5.3.5 Možnost výběru více properties v dialogu vnořené entity

V rámci testovacího scénáře jsem uživatele navedl k tomu, aby vytvářeli adresu jako strukturovaný typ *schema:PostalAddress* právě pomocí mnou vytvořeného dialogu pro vnořenou entitu. Chtěl jsem, aby u adresy evidovali jak číslo ulice, tak PSČ atd. To vyžadovalo volbu více properties, což jim měl umožnit formulářový prvek multi select.

Všichni testeři se v tomto kroku na chvíli zasekli. Po přidání první property totiž nejprve nepoznali, že ten samý formulářový prvek mohou použít k volbě více hodnot. Nebyl to nijak zásadní zádrhel a všem testerům to po chvíli došlo.

Abych uživatelům systému pomohl, změnil jsem popisek tohoto inputu, který nyní zní: *Select **multiple** properties.* Vnořenou entitu zakládáte právě kvůli tomu, že u ní potřebujete evidovat více properties. Od nového popisku si slibuji, že to díky němu uživatelé nyní lépe poznají.

Kapitola 6

Přínos implementace a její další možný rozvoj

Jak už jsem několikrát zmiňoval, ve své implementaci jsem se zaměřil hlavně na podporu slovníku Schema.org. Právě díky němu moje implementace ulehčuje modelování sémantického modelu. Při vytváření svých modelů uživatelé prakticky nemusí mít žádné znalosti o sémantických datech, a přesto jim je umožněno vytvořit je tak, aby později pomocí nich mohli vytvářet a publikovat data v sémantické podobě.

Díky definovaným oborům hodnot v rámci tohoto slovníku si například na rozdíl od pluginu sloužícího k modelování ontologií v Drupal hlídám i přiřazené typy k jednotlivým atributům, aby skutečně odpovídaly jejich požadovanému sémantickému typu. Myslím si proto, že moje řešení je méně citlivé na chyby, kterých se v ostatních systémech může neznalý uživatel snadno dopustit, a díky tomu nevytváří nevalidní sémantická data.

Jako další přínos implementace hodnotím i to, že data s využitím serializace JSON-LD a použití databáze MongoDB v ní ukládám už přímo v sémantické podobě. Díky tomu skoro odpadají starosti při jejich následné publikaci. Data takto uložená se pak dají i strojově zpracovávat, i když nebudou poskytována přes nějakou konkrétní aplikaci.

Myslím si také, že můj nový typ atributu Entita řeší elegantním způsobem podstatný a častý problém sémantických dat, a to, že property je typu další netriviální entity. Právě díky němu uživatelé mohou svá sémantická data vytvářet v rámci formuláře pro jednu entitu, i když nastane tato situace.

6.1 Další možný rozvoj implementace

I přes to, že jsem úspěšně Strapi rozšířil o podporu sémantických dat, považuji svoje řešení stále ještě za prototyp. Při jeho implementaci jsem narazil na spoustu problémů, díky kterým jsem ale přišel i na další nápady, o které by

se řešení dalo ještě vylepšit. Implementaci některých vylepšení jsem ještě stihl zařadit nad rámec svých prvoplánových úprav, jiných však už ne.

A právě o těchto dalších možných nápadech, jak by se implementace v rámci CMS Strapi dala vylepšit a posunout k lepší uživatelské spokojenosti, bude pojednávat následující sekce.

6.1.1 Použití i dalších sémantických slovníků

Ve své práci jsem se zaměřil na implementaci podpory slovníku Schema.org. Ač ho považuji za jeden z nejlepších, rozsah typů, které jsou v něm definovány, určitě nepokrývá všechny možné typy entit, které se na světě vyskytují. Současná verze implementace pluginu pro tvorbu ontologií má dokonce tímto slovníkem docela dost svázané ruce, protože svým uživatelům umožňuje výběr typů a vlastností právě jen z tohoto slovníku. Pokud tedy uživatel chce namodelovat něco nad rámec typů, které ve slovníku existují, už si s tímto nástrojem nevystačí.

V tom případě musí místo sémantické property přidat nesémantický atribut, který je ze sémantických dat při parsování vyřazen. Uživatel sice může jít přímo do definic jednotlivých modelů pomocí JSON souborů a patřičně si v kontextu dodefinovat sémantický typ atributu. To už však vyžaduje uživatelovu pokročilou znalost sémantických dat a způsob, jakým to lze provést, je i nekomfortní.

Dokážu si proto představit nějaké pokročilejší uživatelské rozhraní, kde by uživatel u jednotlivých obsahových typů mohl například editovat přímo obsah kontextu v grafické podobě, a tím mohl rozšířit seznam properties, které by mu Content Type Builder při vytváření atributů nabízel.

6.1.1.1 Rozšíření funkcionality atributního typu Entita

Vnořený atributní typ Entita řeší jednoduché vazby 1:1, kterých je v sémantických datech kvůli silnému typování veškerých properties plno. Tento atribut se hodí ve chvíli, kdy jednotlivé položky vnořeného entitního typu nepotřebujeme editovat na jednom místě, a nemusíme proto kvůli této vazbě zakládat hned nový content type.

Podobnou analogií by se dala vyřešit i vazba 1:N, kdybychom také nepotřebovali k navázaným entitám přistupovat jednotlivě. Toho by se docílilo rozšířením tohoto atributního typu tak, že by uživatel mohl zadefinovat, že chce u vnořené entity umožnit její opakování.

V pluginu Content Manager by pak v editačním formuláři měl u příslušného atributu tlačítko *Přidat další entitu*. Kliknutím na něj by se formulář dynamicky rozšířil o celou skupinu prvků

vnořené entity. Právě díky tomu by mohl vytvořit až N prvků. Takto získaná data by se jednoduše ukládala do pole, což by nebyl problém, protože atribut vnořené entity je už teď uspořádaný jako JSON objekt.

6.1.1.2 Podpora více specifických primitivních datových typů

V kapitole popisující implementaci mého řešení jsem se zmiňoval o automatickém mapování typů reprezentujících jednoduché datové typy na jednotlivé typy atributů. Toto mapování není v současné fázi zcela dokonalé a obsahuje ještě prostor pro vylepšení.

V rámci tohoto mapování přiřazuji všechny typy čísel, jako je například *schema:Float*, na abstraktní typ atributu number. Neurčuji tedy konkrétní číselný formát. To už si musí zajistit uživatel sám, a může zde proto dojít k určité nekonzistenci dat. Zde by bylo dobré toto určování provádět automaticky a předejít tak těmto možným problémům.

Podobně se tomu děje i v případě datumů. Zde také všechny tři různé typy *schema:Date*, *schema:DateTime* i *schema:Time* mapuji na typ atributu date. Ten však ve skutečnosti odpovídá jen typu *schema:DateTime*, protože umožňuje zadání jak data, tak času. Zde by bylo dobré atribut vhodně rozšířit a automaticky omezovat uživatelský vstup jen na jednu složku v případě více specifických typů.

Posledním případem je také atribut typu Media. Ten se používá pro reprezentace typů *schema:MediaObject*, *schema:ImageObject*, *schema:VideoObject* a *schema:AudioObject*. Při vybírání přiřazeného media však nedochází ke kontrole jeho typu, a může se proto snadno stát, že uživatel ve svých datech například přiřadí k atributu video, i když by měl obsahovat jen entity typu obrázek.

To by se opět dalo vyřešit rozšířením atributního typu a podle zvoleného oboru hodnot si vynucovat jen konkrétní typ medií.

6.1.1.3 Automatické určování typu relace

Jak už jsem zmiňoval v kapitole věnující se implementaci mého rozšíření, slovník Schema.org nedefinuje násobnost jednotlivých properties. Právě tato informace mi pak chybí k tomu, abych mohl určovat typ vazeb mezi dvěma obsahovými typy automaticky.

Při pátrání po řešení tohoto problému jsem narazil na tento dokument `https://github.com/geraintluff/schema-org-gen/blob/master/property-multiplicity.json`, kde se někdo pokusil násobnost jednotlivých properties určit ručně. Tento dokument je bohužel několik let neaktualizovaný, a některé properties v něm tedy vůbec nemusí být uvedeny.

Automatické určování násobnosti property by se dalo činit podle jednotného či množného čísla jejího názvu. Například u typu *schema:Person* existují jak properties *schema:children* i *schema:childrens*. Zde je tedy situace, zda-li má být daná property násobná nebo ne, velmi jednoduchá a šlo by tak automaticky učinit.

Když ale vezmu jiný příklad property *schema:alumniOf*, situace už je nejednoznačná. Jeden člověk sice obecně může být členem více organizací

a spolků, ale v případě konkrétního modelu můžeme chtít i integritní omezení, které by determinovalo příslušnost jen na jednu společnost.

Myslím si proto, že rozšíření o automatické předurčování typu relace je sice možné, ale nemělo by být určujícím a uživatel by ho měl být schopen vždy změnit ještě ručně.

6.1.1.4 Rozšíření JS knihovny schema.org o definice properties

Pro načítání jednotlivých sémantických typů používám JS knihovnu schema.org, která je částečnou implementací slovníku Schema.org.

Tato knihovna bohužel ale neobsahuje definice properties jednotlivých sémantických typů, pro jejich získávání proto musím na pozadí aplikace provolávat přímo server `http://schema.org`. To ale způsobuje v mém řešení mírnou latenci, a je proto uživateli potřeba na více místech zobrazovat komponentu loaderu, který signalizuje dotahování dat na pozadí. Zároveň je moje rozšíření závislé na fungování tohoto serveru, a v případě jeho výpadku se nedá využívat.

Kvůli snížení této latence, lepšímu uživatelskému zážitku a zbavení se závislosti na aktuálním stavu serveru Schema.org, by bylo dobré knihovnu rozšířit i o definice jednotlivých properties, které by pak mohlo využívat i moje rozšíření Strapi, obdobně jako používá informace o jednotlivých sémantických typech.

6.1.2 Vybírání přiřazených entit

Při plnění obsahu konkrétní entity pomocí pluginu Content Manager, Strapi ve svém základu nabízí výběrový box, který umožňuje uživateli výběr entit jiného obsahového typu. V tomto seznamu je tedy potřeba reprezentovat cizí entitu jedním textovým identifikátorem.

Pokud obsahový typ obsahuje atribut typu String, použije se on. Pokud jich je více, využije se první z nich. Pokud neobsahuje žádný atribut typu String, použije se automaticky generovaný identifikátor položky.

Díky sémantickému určení typů jednotlivých properties bychom však byli schopni lépe vybrat atribut, kterým chceme reprezentovat celou entitu. A to například jako property typu *schema:name*. Nezáleželo by tedy na pořadí atributů, v jakém byly do content type přidány, ale na jejich sémantickém významu.

6.2 Další vývoj Strapi

Jak už jsem zmiňoval, svoji práci jsem implementoval jako odnož oficiální verze Strapi, protože úpravy mnou provedené si vyžadovaly zásah přímo do datových modelů a nemohly být proto realizovány pouhým přidáním pluginu.

To si však nese svou daň a takovýto kód je mnohem těžší udržovat. Jen za dobu implementace mojí práce vyšly čtyři nové verze. K tomu ještě aktualizace mého kódu na ně nemohla být provedena pouhým sloučením kódů, ale vyžadovala nepříjemné řešení konfliktů. Vzhledem k tomu, že je mnou upravovaná verze Strapi stále v alfa stadiu, pro další použití jsou však tyto aktualizace nezbytné.

Jednal jsem proto v rámci issue na GitHub s tvůrci Strapi, zda-li by nebyli ochotni začlenit části mé práce přímo do knihovního kódu. Tato jednání se mi však po dobu konání práce nepovedla dovést k úspěšnému konci mimo jiné i proto, že tvůrci o sémantická data nejeví velký zájem.

Do budoucna však ani nepředpokládám, že by byl celý kód začleněn. chtěl bych však, aby byly začleněny alespoň některé úpravy v core pluginech, bez kterých se moje rozšíření neobejde. Díky tomu bych pak získal podporu přímo v jádru aplikace.

Samotný plugin Content Type Builder bych pak mohl převést na plugin vlastní Semantic Content Type Builder, který by byl už pro uživatele volitelný. Část funkcionality sémantických dat by byla obsažena už přímo v jádru. Uživatelé, kteří by měli o strukturovaná data větší zájem, by si pak k modelování sémantických dat stáhli můj plugin Semantic Content Type Builder, který by nahrazoval ten originální. Ostatně tak to má přesně udělané i Drupal. O těchto možnostech budu nadále jednat.

Další nepříjemnou věcí je, že do té doby má moje řešení jen malý dosah, než by mělo kdyby bylo součástí přímo oficiální verze. Pro jeho větší využití v praxi je proto podle mě důležité tato jednání dotáhnout do konce. V případě neúspěchu bych CMS vydal jako vlastní odnož pojmenovanou například Semantic Strapi. Stále však věřím, že když se mi řešení podaří ještě vylepšit alespoň o některé navrhované úpravy a zajistím aby bylo kompatibilní s představami vývojářů Strapi, tak se část míchúprav nakonec může dostat přímo do oficiální verze.

Závěr

V první části práce jsem analyzoval tři v současné době nejrozšířenější redakční systémy a jejich rozšíření, která se zabývají přidáním podpory sémantických dat do těchto systémů. Analyzoval jsem také Webnodes CMS jakožto zástupce redakčních systémů specializovaných na sémantická data. V rámci této analýzy jsem pak zhodnotil výhody a nevýhody jednotlivých přístupů a zároveň jsem si díky ní rozšířil obzory o tom, jak implementace sémantických dat v redakčních systémech může vypadat.

Na základě toho jsem pak v další části práce navrhl vlastní způsob řešení, jak si představuji, že by se sémantická data v redakčních systémech mohla vytvářet, spravovat a publikovat. CMS systém by měl tedy poskytovat grafický nástroj pro modelování ontologií na základě nějakého sémantického slovníku, data by měla být spravována na jejich konceptuální úrovni a publikována pomocí REST API.

Podle těchto návrhů jsem pak vybral vhodné CMS k rozšíření, kterým se stalo Strapi mimo jiné kvůli tomu, že se jedná o headless CMS, podporuje MongoDB a už v základu disponuje pluginem pro vytváření datových modelů.

Toto CMS jsem patřičně upravil tak, aby jeho datový model mohl pojmout sémantická data. Zároveň jsem přetvořil plugin pro grafickou tvorbu modelů na nástroj pro modelování ontologií na základě sémantických typů ze slovníku Schema.org. Dále jsem ho rozšířil o další typy atributů, které jsou využívány při vytváření a editaci sémantických dat. A v neposlední řadě jsem provedl úpravy, aby umožňoval publikaci dat nejen ve formátu JSON-LD.

Implementované řešení jsem poté podrobil uživatelskému testování, v kterém jsem převážně zkoumal to, jak se mi podařilo udělat nástroj pro modelování ontologií srozumitelný i pro běžné uživatele neznalé sémantických dat. Na základě tohoto testování jsem aplikaci ještě upravil, abych předešel některým problémům, na které jsem při testování narazil.

V poslední části práce nastiňuji další možná rozšíření vlastní implementace, která se mi v rámci této práce už nepodařilo uskutečnit, ale myslím si, že by procesu vytváření sémantických dat ještě prospěla.

Literatura

[1] SPIVACK, N.: The Semantic Web, Collective Intelligence and Hyperdata [online]. http://novaspivack.typepad.com/nova_spivacks_weblog/2007/09/hyperdata.html, 2007, [Citováno 2018-01-31].

[2] BERNERS-LEE, T.: Linked Data - Design Issues [online]. https://www.w3.org/DesignIssues/LinkedData.html, 2006, [Citováno 2018-04-29].

[3] DUMBILL, E.: Berners-Lee and the Semantic Web Vision [online]. https://www.xml.com/pub/a/2000/12/xml2000/timbl.html, 2000, [Citováno 2018-01-31].

[4] working group RDF: RDF - Semantic Web Standards [online]. https://www.w3.org/RDF/, 2014, [Citováno 2018-01-30].

[5] SPORNY, M.; LONGLEY, D.; KELLOGG, G.; aj.: JSON-LD 1.0 [online]. https://www.w3.org/TR/json-ld/, 2014, [Citováno 2018-03-04].

[6] HAUSENBLAS, M.: 5-star Open Data [online]. http://5stardata.info/en/, 2012, [Citováno 2018-04-28].

[7] Product info - Webnodes ASP.Net CMS [online]. https://www.webnodes.com/product-info, 2010, [Citováno 2018-03-18].

[8] Co je to Entita? - IT Slovník [online]. https://it-slovnik.cz/pojem/entita, 2008, [Citováno 2018-04-26].

[9] Kellogg, G.: JSON-LD and MongoDB [online]. https://www.slideshare.net/gkellogg1/jsonld-and-mongodb, 20012, [Citováno 2018-04-21].

[10] SPORNY, M.; LONGLEY, D.; KELLOGG, G.; aj.: JSON-LD 1.1 [online]. https://json-ld.org/spec/latest/json-ld/#dfn-null, 2018, [Citováno 2018-04-29].

Příloha **A**

Seznam použitých zkratek

CMS Content Management System

RDF Resource Description Framework

JSON JavaScript Object Notation

JSON-LD JSON for Linked Data

XML Extensible markup language

API Application programming interface

HTML HyperText Markup Language

REST REpresentational State Transfer

MIME type Multipurpose Internet Mail Extensions

WYSIWYG What You See Is What You Get

@type `http://www.w3.org/1999/02/22-rdf-syntax-ns#type`

schema: http://schema.org/

Příloha **B**

Instalační příručka

Praktická část mojí práce vyžaduje ke svému zprovoznění specifické systémové požadavky a také netriviální instalační postup. Těmto požadavkům a návodu jak aplikaci rozchodit na lokálním serveru je věnována tato příloha.

B.1 Systémové požadavky

Strapi ke svému fungování vyžaduje nainstalované:

- **Node.js** ve verzi minimálně **9.0**
- **NPM** ve verzi minimálně **5.0**
- **MongoDB** ve verzi minimálně **3.4**

Strapi v produkční verzi funguje na jakémkoli systému podporujícím Node.js, praktickou část práce však budete muset spouštět ve vývojářském módu, jehož instalační script funguje pouze v systémech typu **UNIX**.

B.2 Postup instalace a spuštění

1. Stáhněte si zdrojové kódy buď z přiloženého CD nebo z repozitáře na Github: `https://github.com/kopecmi8/strapi.git`
2. Spusťte příkaz *sudo npm run setup* v adresáři se zdrojovými kódy (pravděpodobně strapi). Instalace může v závislosti na výkonu počítače trvat několik minut.
3. Po dokončení instalace přejděte do složky, ze které chcete vygenerovat nový projekt.
4. Nový projekt vygenerujete pomocí příkazu *strapi new nazev-projektu --dev*

5. Postupujte podle instrukcí v instalačním procesu. Jako DB ponechte podle výchozího nastavení MongoDB.
6. Po skončení procesu přejděte do nově vytvořené složky pojmenované podle názvu projektu.
7. V podsložce admin spusťte následující příkaz *npm start*. Ten spustí buildovací proces.
8. Otevřete si novou konzoli a přejděte opět do složky nově vytvořeného projektu. Zde přímo v rootu spusťte příkaz *strapi start*

B.3 Po spuštění

Aplikace ve vývojářském módu běží na dvou různých portech 4000 a 1337. Administrace aplikace je dostupná na adrese `http://localhost:4000/admin`, kde při prvním spuštění musíte nejprve založit nový administrátorský účet. Pomocí něho budete do administrace automaticky přihlášen.

Na adrese `http://localhost:1337` běží front-end Strapi. Zde zatím na začátku není nic k vidění. Nejprve je nutné namodelovat nějaké content types, pomocí pluginu Content Type Builder. Díky ním se pak vygeneruje REST API. Po dokončení modelování nového content type je zároveň nutné povolit přístup k jeho API v pluginu Roles&Premmissions. To se provádí v konfiguraci role *Public*. V příslušném menu byste pro jednotlivé vytvořené content types měli mít možnost povolit všechny CRUD operace: create, destroy, find, finone a update.

Podle názvu vytvořeného content type se generuje i URL adresa API. Například content type nazvaný person tak bude po povolení v pluginu Roles&Premmissions dostupný na adrese `http://localhost:1337/person`.

Jako podrobnější návod může také posloužit stránka `https://strapi.io/documentation/getting-started/quick-start.html`, která však popisuje zprovoznění produkční a originální verze Strapi. Některé kroky se proto liší a doporučuji se řídit touto instalační příručkou.

Příloha C

Testovací scénář

C.1 Úvod

Strapi je webový redakční systém sloužící primárně k editaci obsahu a vytváření dat na webu. Kromě samotného plnění dat však poskytuje i nástroj pro vytváření datového modelu (modelování ontologií), jenž je potřeba vytvořit na začátku jeho používání podle specifické domény, pro kterou chcete tento systém využívat. Vaším úkolem bude právě tvorba takového modelu pomocí grafického nástroje, který k tomuto účelu Strapi poskytuje. Díky tomuto procesu vygenerujete i uživatelské rozhraní pro běžné uživatele, kteří budou data poté v redakčním systému plnit.

Pokud vám to nevadí, pokuste se říkat nahlas, co si právě myslíte, co se chystáte udělat a proč. S řešením jednotlivých úkolů nespěchejte. Pokud se budete cítit ztracen, zkuste si nejprve prosím přečíst i znění dalšího kroku, neboť jednotlivé kroky jsou mezi sebou provázány.

C.2 Scénář: Tvorba datového modelu pro doménu filmové databáze

Pokud jím nejste, představte si, že jste webový vývojář. Dostal jste za úkol vytvořit webovou aplikaci, která má sloužit jako databáze filmů a herců, kteří v nich hrají. Rozhodl jste se k tomu využít CMS Strapi, které jste si právě nainstaloval na lokální server. Vaším úkolem je nyní vytvořit datový model, díky němuž pak budou moci běžní uživatelé databázi plnit jednotlivými filmy a herci. Všechny názvy, které budete vyplňovat, prosím zadávejte v angličtině.

Úvodní stav Úvodní obrazovka aplikace po přihlášení

Koncový stav Obrazovka s editací položky konkrétního obsahového typu

1. Právě jste poprvé spustili aplikaci Strapi a nacházíte se v administrační části na její úvodní obrazovce po přihlášení. Vaším úkolem je nyní vytvořit datový model tak, aby vyhovoval všem požadavkům zadavatele.

2. Plugin Content Type Builder je tím pravým nástrojem, který by Vám měl k tomu účelu posloužit. Nejprve založte nový obsahový typ (content type), který bude reprezentovat entity typu Film (Movie). Je u něj potřeba evidovat následující údaje:

   - název filmu
   - datum premiéry (publikace)
   - herce, kteří ve filmu hráli

   Herce (Actors) chcete evidovat jako vlastní obsahový typ (content type), neboť ve své aplikaci chcete, aby měl každý herec i vlastní stránku. Při vytváření vazby vycházejte z toho, že jeden herec může hrát v mnoha filmech a zároveň v jednom filmu hraje více herců.

3. Při přidávání atributu *herce* jste pravděpodobně narazil na problém, kvůli kterému nemůžete property typu *herec* vytvořit. Pokuste se vyřešit tento problém vytvořením nového obsahového typu. U jednotlivých herců pak evidujte následující údaje:

   - jméno
   - příjmení
   - datum narození

   Nezapomeňte také dokončit předchozí krok, že herci a filmy mezi sebou mají být provázány.

4. Nyní byste měl být ve své aplikaci schopen plnit jednotlivé filmy a spravovat herce, kteří v nich hrají. Zadavatel však přišel se změnovým požadavkem, a chce po Vás, abyste ještě u jednotlivých filmů evidoval i filmová studia (Production company), která je vyrobila. U nich pak evidujte následující údaje:

   - název studia
   - herce, kteří pro studio pracují
   - adresu, kde studio sídlí.

   Adresu filmového studia potřebujete zadávat ve strukturované podobě a evidovat u ní: ulici, poštovní směrovací číslo a zemi. Při vytváření tohoto atributu vycházejte z toho, že jednotlivá studia nikdy nesídlí na stejném místě a je tedy zbytečné je evidovat jako vlastní obsahový typ.

5. Tímto je testování u konce, v levém menu si ještě nyní můžete zkusit rozkliknout některý z vytvořených obsahových typů a podívat se na výsledné vygenerované uživatelské rozhraní.

Příloha D

Obsah přiloženého DVD

- `src` ..... zdrojové kódy implementace a textu práce
  - `impl` ..... implementace
    - `strapi` ..... zdrojové kódy implementace
    - `instalacniPrirucka.pdf` ..... instalační příručka
    - `screencast.mp4` ..... Screencast zachycující modelování ontologií
  - `thesis` ..... zdrojová forma práce ve formátu LaTeX
    - `img` ..... obrázky použité v textu
- `text` ..... text práce
  - `thesis.pdf` ..... text práce ve formátu PDF
  - `thesis.ps` ..... text práce ve formátu PS