Univerza v Ljubljani
Fakulteta za računalništvo in informatiko

Tine Mislej

Semantična kompozicija spletnih storitev REST

magistrsko delo

magistrski program druge stopnje
računalništvo in informatika

Mentor: doc. dr. Dejan Lavbič

Ljubljana, 2018

Zahvala

Zahvaljujem se družini za neizmerno podporo skozi študijska leta in mentorju doc. dr. Dejanu Lavbiču za izjemno mentorstvo pri magistrskem delu.

Tine Mislej, 2018

Kazalo

Seznam uporabljenih kratic

kratica	angleško	slovensko
JSON-LD	JSON Linked Data	JSON za povezane podatke
API	Application Programming Interface	Programski vmesnik
WSMO	Web Service Modeling Ontology	Ontologija za modeliranje spletnih storitev
SAWSDL	Semantic Annotations for WSDL and XML Schema	Semantične značke za WSDL in shemo XML
SLA	Service Level Agreement	Sporazum o ravni storitve
SCG	Service Composition Graph	Graf kompozicije spletnih storitev
SPM	Shortest Predecessor Matrix	Matrika najkrajših predhodnikov
IRI	Internationalized Resource Identifier Interface	Mednarodni označevalnik vira
WSDL	Web Service Description Language	Opisni jezik spletnih storitev
SPARQL	Simple Protocol And RDF Query Language	Protokol SPARQL in poizvedovalni jezik RDF
SOAP	Simple Object Access Protocol	Protokol za izmenjavo podatkov med spletnimi storitvami
BFS	Breadth-First Search	Iskanje v širino
RDF	Resource Description Framework	Ogrodje za opisovanje virov

kratica	**angleško**	**slovensko**
XML	Extensible Markup Language	Razširljivi označevalni jezik
REST	Representational state transfer	Arhitektura za izmenjavo podatkov med spletnimi storitvami
URL	Uniform Resource Locator	Enotni naslov vira

Povzetek

Naslov: Semantična kompozicija spletnih storitev REST

V delu smo se osredotočili na semantično razširjanje standarda Swagger z namenom podpore semantični kompoziciji. Eden izmed izzivov sodobnega razvoja spletnih storitev je semantično opismenjevanje z namenom učinkovitejšega obdelovanja definicij in uporabe takih storitev. V okviru magistrskega dela smo razširili obliko JSON standarda Swagger različice 2.0 s semantičnimi značkami, ki pomensko opisujejo vhode in izhode spletnih storitev. Za razširjanje smo uporabili format JSON-LD in upoštevali omejitve standarda ter tako zagotovili popolno skladnost s standardom. Na osnovi razširjenih opisov smo razvili algoritme, potrebne za vizualizacijo semantične kompozicije. Razvili smo razčlenjevalnik razširjenih opisov, algoritem za gradnjo kompozicijskega grafa, algoritem za iskanje kandidatov v kompozicijskem grafu, algoritem za predlaganje rešitev v primeru neuspešne kompozicije in logiko, potrebno za semantično presojanje. Razvili smo podporno spletno aplikacijo za vizualizacijo semantične kompozicije, ki uporablja omenjene algoritme, in vzpostavili potrebno infrastrukturo. Delo poda osnovo za pristop k semantični kompoziciji spletnih storitev, opisanih v razširjenem standardu Swagger.

Ključne besede

Swagger, semantika, kompozicija, REST, JSON-LD

Abstract

Title: Composition of semantic REST web services

The thesis focuses on the semantic extensions for Swagger standard to support semantic composition. One of the challenges in modern web service development is semantic annotation to support efficient processing and the use of web services. In the thesis, JSON form of Swagger standard 2.0 was extended with semantic annotations which define the semantics of the inputs and outputs of web services. The extensions are based on JSON-LD format and comply with the standard's limitations, resulting in a fully compatible standard extension. Based on the extended standard definitions the algorithms needed for visualizing semantic composition were developed. We also developed a parser for documents described in extended form of the standard, an algorithm for building a composition graph, an algorithm for searching composition candidates in the graph, an algorithm for suggesting possible solutions in case of failed composition and the logic necessary for semantic reasoning. Besides developing a support web application for semantic composition visualization that uses developed algorithms, we also set up the necessary infrastructure.
The thesis proposes a possible approach to semantic composition of web services described in extended Swagger standard form.

Keywords

Swagger, semantics, composition, REST, JSON-LD

Poglavje 1

Uvod

Spletne storitve - sistemi, v osnovni namenjeni komunikaciji med stroji oz. računalniškimi sistemi - so danes eden izmed najpomembnejših konstruktov v procesu razvoja aplikacij. To so tisti gradniki, ki strežejo informacije in zahtevke aplikacijam, ki nenazadnje služijo nam, uporabnikom. Še več, spletne storitve so tudi tisti gradniki, ki so pogosto uporabljeni kot povezovalni deli med različnimi, heterogenimi računalniškimi sistemi in procesi. Njihova že tako pomembna vloga v procesu razvoja pa danes še bolj pridobiva pomen. V zadnjem desetletju smo namreč priča eksponentni rasti podatkov in informacij, ki jih v spletu ustvarjamo tako ljudje kot tudi naprave in različni računalniški procesi. Po nekaterih informacijah [15] naj bi dnevno v spletu nastalo 2,5 kvintilijona bajtov novih podatkov. Hitrost akumuliranja podatkov v spletu samo še narašča, predvsem s pojavom interneta stvari (angl. Internet of Things), upravljanje teh podatkov pa je v večji meri podprto z uporabo spletnih storitev.

Razne pametne naprave uporabljajo spletne storitve za obdelovanje in objavljanje podatkov. Na primer, merilci hitrosti burje svoja merjenja v spletu objavljajo z uporabo spletnih storitev. Ljudje (ali pa tudi drugi računalniški procesi), kot porabniki teh informacij, do njih prav tako pogosto dostopamo prek spletnih storitev. Nadalje, družbena omrežja, kot so Facebook, Twitter, Instagram itd., velike količine zbranih podatkov prav tako izpostavljajo prek

spletnih storitev.
Spletne storitve so torej zelo pomemben vmesnik med takimi agregacijami podatkov in porabniki le-teh in, kot že zapisano, različnimi deli heterogenih računalniških sistemov. Eden izmed večjih izzivov za razvijalce je uporaba obstoječih spletnih storitev za razvoj novih storitev in aplikacij. Ker je množica objavljenih spletnih storitev ogromna, se pojavlja težnja po učinkovitem upravljanju, iskanju in kompoziciji obstoječih spletnih storitev.

Semantične spletne storitve [22] so običajne spletne storitve, katerih podatki so pomensko označeni in tako omogočajo višjo raven (računalniško podprtega) presojanja in uporabe. Na področje semantičnih spletnih storitev spada tudi naše delo. Ukvarjali smo se s semantičnim razširjanjem standarda Swagger [16] za potrebe semantične kompozicije ter razvili podporno storitev in potrebne algoritme. V nadaljevanju predstavimo motivacijo za svoje delo in v 2. poglavju pregledamo sorodna dela. V nadaljevanju dela predstavimo kratek pregled ključnih tehnologij, ki jih uporablja naša rešitev. V 5. in 6. poglavju predstavimo razširitev standarda Swagger in semantično kompozicijo. V nadaljevanju z analizo SWOT ovrednotimo rešitvi in v zadnjem, 8. poglavju podamo sklepne ugotovitve.

1.1 Motivacijski primer

V tem delu predstavimo motivacijski primer za svoje delo. Predpostavimo obstoj repozitorija spletnih storitev, ki vsebuje tudi naslednje storitve, z definiranimi vhodnimi in izhodnimi koncepti.

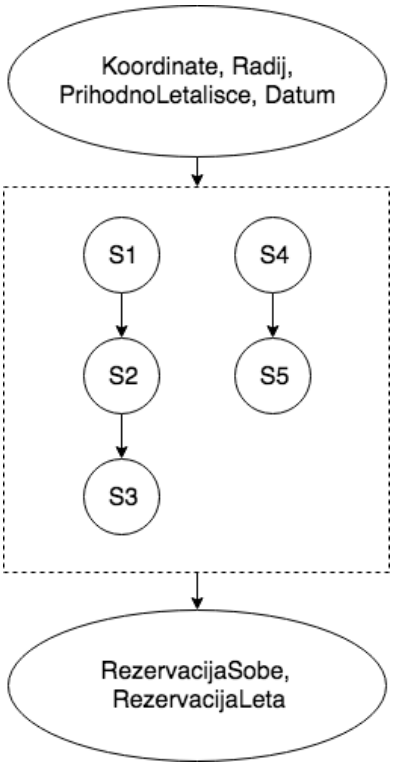
Izpis 1.1: Poenostavljen primer semantičnih spletnih storitev

```
S1 = (vhod=[Koordinate, Radij], izhod=[Hotel])
S2 = (vhod=[Hotel], izhod=[HotelskaSoba])
S3 = (vhod=[HotelskaSoba, izhod=[RezervacijaSobe]])
S4 = (vhod=[Koordinate], izhod=[Letalisce])
S5 = (vhod=[Letalisce, PrihodnoLetalisce, Datum],
      izhod=[RezervacijaLeta])
```

Denimo, da razvijalec razvija turistično aplikacijo, katere ena izmed njenih funkcionalnosti bo tudi rezervacija hotelske sobe in leta do hotelu najbližjega letališča. Kot vhod storitvi razvijalec želi podati `Koordinate`, `Radij`, `PrihodnoLetalisce` in `Datum`, iskana izhoda pa sta `RezervacijaSobe` in `RezervacijaLeta`.
Običajno je najprej treba najti storitev, ki reši dani problem. V našem primeru mora razvijalec v repozitoriju poiskati zahtevano storitev, kar pa je lahko precej dolgotrajen in težaven proces. Temu navadno sledi še proučevanje storitve in načina vključitve v aplikacijo. Delo postane še težje, če iskana storitev ne obstaja, kar se pogosto pripeti. Tedaj mora razvijalec v repozitoriju poiskati več storitev, ki bodo skupaj lahko rešile dani problem. Razvijalec mora ponovno najti in preučiti posamezno storitev ter, če najdene storitve lahko rešijo problem, storitve sestaviti v delujočo celoto.

Iz izpisa 1.1 je razvidno, da so storitve semantično skladne in so dobri kandidati za kompozicijo. Slika 1.1 prikazuje možno kompozicijo omenjenih storitev.

Cilj našega dela je razširitev standarda Swagger s podporo za semantično opismenjevanje storitev. V splošnem lahko potem storitve predstavimo na podoben način, kot so zapisane v izpisu 1.1. Aplikacija za podporo semantični



Slika 1.1: Primer kompozicije storitev iz izpisa 1.1

kompoziciji uporablja razširjene opise Swagger in na zahtevo išče storitve ter vizualizira kompozicijo - podobno, kot to prikazuje slika 1.1. V zgornjem primeru obstaja samo en možen način kompozicije za doseganje zastavljenega cilja, običajno pa je teh več - torej lahko obstaja več potencialnih kandidatov za kompozicijo. Naša aplikacija je zmožna iskanja takih kandidatov ter razvrščanja glede na njihove morebitne nefunkcionalne parametre. Še več, če kompozicija ni možna, naša aplikacija predlaga čim boljše nove storitve, katerih vključitev bo omogočila kompozicijo.

1.2 Pristop k delu

Delo je potekalo v več fazah. Najprej smo znaten del časa namenili raziskovanju področja in že obstoječih rešitev ter se z njimi seznanili. V tej fazi

je bilo ključno identificiranje prednosti in slabosti obstoječih pristopov, kar je zelo pripomoglo k jasnejšemu snovanju razširitve standarda Swagger. V nadaljevanju smo se posvetili študiji standarda in snovanju razširitve. Pri snovanju rešitve smo si zadali naslednje smernice:

- Razširitev naj bo skladna s standardom.
- Obstoječe definicije Swagger naj bo enostavno nadgraditi oz. razširiti.
- Razširitev naj za implementacijo zahteva čim manj dodatnega znanja.

Ugotovili smo, da bomo naštetim smernicam lahko sledili z uporabo notacije JSON-LD [14]. Z znanim ciljem in načinom smo začeli s snovanjem razširitve. Le-to je potekalo v dveh iteracijah, rezultat te faze pa je razširitev standarda Swagger, predstavljena v nadaljevanju tega dela.
Naslednja faza dela sta bila načrtovanje in implementacija podporne storitve za semantično kompozicijo. Cilj te faze je bil razvoj podporne spletne aplikacije, ki bo vizualizirala kompozicijo, temelječo na definicijah storitev v razširjenem standardu Swagger. Najprej smo začeli z načrtovanjem in prototipiranjem algoritmov, potrebnih za kompozicijo. To je potekalo v več iteracijah. Ključna je bila odločitev, da problem prevedemo na teorijo grafov, kjer spletne storitve in njihove vhode in izhode definiramo kot usmerjen graf vozlišč dveh tipov. V prvih iteracijah smo razvili več prototipov algoritmov. Nekateri izmed njih niso v celoti rešili naših problemov, zato smo jih zavrgli. Rezultat nadaljnih iteracij pa je zbirka algoritmov, potrebnih za iskanje potencialnih kandidatov za kompozicijo.
Sledili sta integracija razvitih algoritmov in implementacija dveh spletnih storitev. Namen prve storitve sta nalaganje in razčlemba razširjenih datotek Swagger, druga pa je zadolžena za dejansko iskanje kandidatov za kompozicijo v skladu z uporabnikovo poizvedbo.
V zadnji fazi dela smo razvili uporabniški vmesnik podporne (spletne) aplikacije za vizualizacijo kompozicije. Za razvoj spletnega vmesnika smo uporabili ogrodje Vue [17], za vizualizacijo kompozicije pa knjižnico D3 [18].

1.3 Prispevki magistrskega dela

V okviru magistrskega dela smo raziskovali možne pristope k semantični kompoziciji spletnih storitev REST. Razširili smo de-facto standard Swagger, namenjen opisovanju in definiranju spletnih storitev REST. Standard smo razširli s semantičnimi značkami in dodatnimi, nefunkcionalnimi opisi operacij spletnih storitev. Razširitev je v celoti skladna z izvorno definicijo standarda, zato je že obstoječe definicije Swagger možno relativno enostavno nadgraditi - brez večjih sprememb in prilagajanj obstoječe infrastrukture in orodij.
Na podlagi razširjenih definicij Swagger smo razvili potrebne algoritme za podporo semantični kompoziciji. Razvili smo algoritme za:

- razčlenjevanje razširjenih definicij;
- gradnjo kompozicijskega grafa;
- iskanje kandidatov oz. rešitev kompozicije v kompozicijskem grafu;
- predlaganje manjkajočih spletnih storitev v primeru neuspešne kompozicije;
- modificiran algoritem za gradnjo kompozicijskega grafa, potreben za podporo algoritmu iz prejšnje točke;
- logiko za ocenjevanje primernosti spletne storitve glede na definirane nefunkcionalne parametre.

Algoritme smo združili v celoto in njihove funkcionalnosti izpostavili v obliki dveh spletnih storitev REST.[1] Poleg tega smo razvili spletni vmesnik[2] za vizualizacijo semantične kompozicije.

[1] `https://bitbucket.org/tinemislej/swsc-webservice`
[2] `https://bitbucket.org/tinemislej/swsc-app`

Poglavje 2

Pregled sorodnih del

V tem poglavju pregledamo sorodna dela na področju semantične kompozicije spletnih storitev. Področje je precej dejavno. Raziskanih je bilo že veliko pristopov, takih, ki temeljijo na umetni inteligenci, pa tudi takih, ki, kot naš pristop, temeljijo na teoriji grafov. Najprej je bilo dejavno področje spletnih storitev SOAP/XML. Za potrebe semantičnega opismenjevanja in kompozicije je bilo razvitih veliko standardov, kot npr. standard SAWSDL [3], ki definira množico razširitvenih atributov za WSDL in standard WSMO [4]. Razvitih je bilo več pristopov, ki pa so v osnovi nekompatiblni zaradi uporabe različnih predstavitvenih jezikov in konceptualnih razlik [2]. Večina pristopov temelji na bogatenju spletnih storitev WSDL, te pa danes ne prevladujejo več [2]. V delu smo se osredotočili na danes najbolj uporabljane spletne storitve REST.

V delu [1] predstavijo pristop k semantični kompoziciji spletnih storitev, ki tako kot naš temelji na teoriji grafov. Semantična razmerja med spletnimi storitvami modelirajo z uporabo usmerjenega grafa. V tem grafu nato s pomočjo razširjenega Floyd-Warshallowega algoritma izračunajo najkrajše poti med vsemi vozlišči (spletnimi storitvami) in tako skušajo rešiti problem semantične kompozicije.
Ta pristop temelji na predstavitvi semantičnih razmerji v obliki usmerjenega

grafa (SCG[1]), zgrajenega že v času objave spletne storitve [1]. Za preverjanje semantične skladnosti preverjajo skladnost posameznih parametrov. Parametra sta skladna, če je en parameter splošnejši od drugega. Vedno, ko je nova storitev objavljena, posodobijo graf SCG in matriko SPM. Matrika SPM hrani informacije o najkrajših poteh med vsemi vozlišči v grafu SCG. Te informacije so izhod Floyd-Warshallovega algoritma, ki se na grafu SCG izvede vsakič, ko je v graf dodana nova storitev oz. vozlišče. Z izračunom najkrajših poti in gradnjo grafa SCG v času objave storitve so avtorji pristopa precej zmanjšali računsko zahtevnost v času izvajanja in posledično zmanjšali čas, potreben za kompozicijo.
V času izvajanja kompozicije najprej identificirajo začetna (Vstart) in končna vozlišča (Vgoal), ki ustrezajo vhodom oz. izhodom uporabnikove poizvedbe. Nato iz matrike SPM izvlečejo vse najkrajše poti med začetnimi in končnimi vozlišči - to so vse najkrajše poti med vhodnimi in izhodnimi parametri. Tu gre za delne kompozicije, ki imajo lahko skupna vozlišča [1]. Končni graf kompozicije je v osnovi graf vseh najkrajših poti (delnih kompozicij), z odstranjenimi podvojenimi potmi.

Avtorji dela [5] predlagajo nov pristop h kompoziciji spletnih storitev RESTful. Bistvo njihovega pristopa je razširitev spletnih virov z deskriptorjem, ki vsebuje metapodatke o viru in informacije o povezanih virih.
Pristop temelji na opisu spletnih virov z namenom enotnega procesa odkrivanja notranjih virov in zunanjih storitev. Ko je URI vira na voljo, je možno pridobiti tudi njegov deskriptor in na ta način slediti splošnemu vzorcu interakcije. Interkacijo omogočajo z uporabo zaglavja HTTP LINK in lastnosti `describedBy`, izražene v RDF [5]. Na ta način deskriptor povežejo z eksplicitnim semantičnim opisom vira.
Deskriptor opisuje vire, ki jim pripada in v osnovi opisuje, katere operacije HTTP so na voljo na spletnem viru, ter dodatne informacije o (pod)virih in drugih zunanjih virih, ki so na voljo. Vsi deskriptorji so tudi viri v skladu z

[1]Graf semantično povezanih storitev.

načeli REST - posledično je ta mehanizem rekurzivno veljaven.
Pristop torej temelji na deskriptorjih in se zanaša, da ima vsak vir (vedno) tudi svoj deskriptor z metapodatki, ki ga običajno pridobimo z operacijo HEAD ali GET na viru. Odjemalec dostopa do sorodnih virov z informacijami o vsakem izmed njih in se tako odloča, po kateri poti iti naprej. Informacija o drugem viru mora biti odjemalcu v pomoč, da se lahko odloči o uporabi določenega vira. Opisovati mora semantiko povezave in katerekoli druge uporabne informacije. Hkrati teh informacij ne sme biti preveč. Velika količina teh informacij lahko privede do podatkovne odvečnosti ali težav, povezanih s pasovno širino [5]. Te informacije so običajno opisi operacij HTTP na spletnem viru, opisi podatkovnih modelov vhodov in izhodov spletnega vira itd. V prihodnosti nameravajo razširiti tudi semantične opise o sorodnih virih in na ta način natančneje opisati, kako se lahko drugi viri uporabijo v kombinaciji s trenutnim [5].

V delu [6] avtorji predstavijo nekoliko drugačen pristop za kompozicijo JSON API-jev, ki tudi temelji na teoriji grafov. Njihov pristop je zmožen prepoznati povezave med shemami različnih API-jev. Z informacijami o omenjenih povezavah in shemah API-jev zgradijo graf, kjer posamezne poti predstavljajo kompozicije. Na tem grafu je moč prepoznati načine za kompozicijo posameznih API-jev. Pristop predstavlja informacije o domeni storitev kot diagrame razredov - vključujoč koncepte in njihova razmerja, kompozicijske povezave pa so predstavljene kot razmerja med koncepti iz različnih domen [6].
Učenje domene API-ja poteka z združevanjem domen posameznih storitev, ki jih ponuja. Domeno posamezne storitve ugotavljajo z analizo podatkov JSON, uporabljenih kot vhod oz. izhod storitve [6]. To je prva faza - iskanje domene posamezne storitve (angl. single-service discovery), katere rezultat je model, ki predstavlja domeno storitve. V drugi fazi (angl. multi-service discovery) z modeli prve faze sestavijo nov, širši model, ki predstavlja celotno domeno API-ja.

Iskanje kompozicijskih povezav med API-ji poteka z iskanjem ujemanja konceptov njihovih domen in preverjanjem, ali so del vhodov storitev posameznih API-jev. Ta proces iskanja kompozicijskih povezav torej analizira posamezne domene ter išče podobnosti in razlike [6].
Z algoritmi za iskanje poti v grafu je potem možno najti poti med določenimi vhodnimi in izhodnimi koncepti. V delu se avtorji dotaknejo še kompozicije, ki upošteva cene določenih poti. Ceno določenih poti, tako kot v našem delu, določajo nefunkcionalni parametri, kot je npr. cena uporabe storitve.

Naše delo v primerjavi z drugimi temelji na dokumentih opisov storitev, skladnih z razširjenim standardom Swagger. V zgled nam je bilo delo [1]. Naš pristop k iskanju algoritmov je soroden in se zgleduje po njihovem pristopu gradnje grafa SCG. Kompozicijski graf gradimo v času izvajanja in le s storitvami, ki so pomembne za določeno kompozicijo, medtem ko v delu [1] graf SCG gradijo v času dodajanja storitve. Naš pristop se posledično razlikuje tudi v načinu izvajanja kompozicije in iskanju kandidatov. V delu [6] se učijo domen posamezne operacije in celotne spletne storitve, medtem ko v svojem delu tega ne počnemo, saj je pomen posameznih parametrov že določen s semantičnimi značkami. V svojem delu podobno kot v delu [6] izkoriščamo definirane nefunkcionalne parametre. V delu [6] jih sicer uporabljajo za uteževanje poti, medtem ko smo jih mi uporabili za ocenjevanje posameznih kandidatov za kompozicijo.

Poglavje 3

Uporabljene tehnologije

3.1 Swagger

Swagger (oz. specifikacija Open Api) je odprta specifikacija za definiranje in dokumentiranje REST API-jev. Swagger je jezikovno neodvisen vmesnik do RESTful API-jev, ki omogoča ljudem in računalnikom enostavno odkrivanje in razumevanje zmožnosti spletnih storitev brez dostopa do izvorne kode, dokumentacije ali pregledovanja omrežnega prometa [16]. Dokument Swagger je na področju REST API-jev ekvivalent dokumentu WSDL na področju spletnih storitev SOAP. Dokument določa seznam virov, dostopnih na REST API-ju in operacije, ki jih lahko izvajamo na njih [7].

Izpis 3.1: Primer definicije Swagger

```
{
  "swagger": "2.0",
...,
  "paths": {
    "/pets": {
      "get": {
        "summary": "List all pets",
        "operationId": "listPets",
        "tags": [
          "pets"
        ],
```

```
      "parameters": [
        {
          "name": "limit",
          "in": "query",
          "description": "How many items to return at one time",
          "required": false,
          "type": "integer",
          "format": "int32"
        }
      ],
      "responses": {
        "200": {
          "description": "An paged array of pets",
          "headers": {
            "x-next": {
              "type": "string",
              "description": "A link to the next page of responses"
            }
          },
          "schema": {
            "$ref": "#/definitions/Pets"
          }
        }
      }
    }
  }
},
"definitions": {
  "Pet": {
    "required": [
      "id",
      "name"
    ],
    "properties": {
      "id": {
        "type": "integer",
        "format": "int64"
      },
```

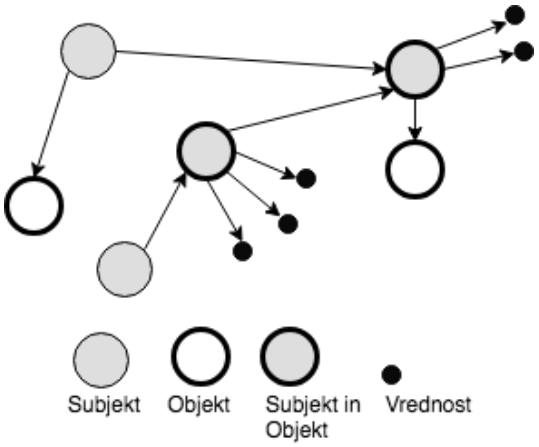
```
        "name": {
          "type": "string"
        },
        "tag": {
          "type": "string"
        }
      }
    },
    "Pets": {
      "type": "array",
      "items": {
        "$ref": "#/definitions/Pet"
      }
    }
  }
}
```

3.2 JSON-LD

JSON-LD [14] je lahek format za opisovanje povezanih podatkov (angl. Linked Data). Temelji na splošno sprejetem formatu JSON in je posledično skladen z njim ter tudi enostaven za branje in pisanje. JSON-LD ni namenjen samo izražanju povezanih podatkov, temveč tudi dodajanju semantike že obstoječim dokumentom JSON. Zasnovan je bil kot preprost, jedrnat in berljiv za ljudi. Še več, pri snovanju je bil eden izmed glavnih ciljev to, da od razvijalcev zahteva kar se da malo truda, da preproste dokumente JSON pretvorijo v semantično bogate dokumente JSON-LD [8].

V osnovi je podatkovni model, opisan z dokumentom JSON-LD označen usmerjeni graf. Graf vsebuje vozlišča, povezana s povezavami. Vozlišča so tipično podatki, npr. nizi (angl. string), števila, tipizirane vrednosti ali IRI, povezave pa so tipično označene z IRI-jem. Vozlišča v grafu so subjekti (angl. subjects) ali objekti (angl. objects), povezave pa lastnosti (angl. properties). Subjekt je vozlišče z vsaj eno izhodno povezavo, medtem ko je objekt vozlišče

z vsaj eno vhodno povezavo. Posledično je vozlišče lahko objekt in subjekt hkrati. Da je subjekt nedvoumno prepoznaven in naslovljiv, je zaželeno, da je označen z IRI-jem. JSON-LD podpira tudi neoznačena vozlišča in na ta način podpira primere, kjer je potrebno samo lokalno naslavljanje podatkov [8]. Podobno velja tudi za lastnosti, če so označene z IRI-jem, jih lahko naslavljamo tudi iz drugih dokumentov, sicer gre pa za običajne lastnosti JSON.



Slika 3.1: Podatkovni model, prirejeno po [8].

Nekoliko drugače je pri objektih. Če je objekt označen z IRI-jem, ga imenujemo objekt, sicer pa, če je označen z nečim drugim (npr. številom), ga imenujemo vrednost [8].

Prednost formata JSON-LD, poleg popolne združljivosti z običajnim formatom JSON, je tudi preprostost uporabe. Za osnovno uporabo je dovolj, če razvijalec poleg formata JSON pozna še ključni besedi `@context` in `@id` [8].

Kombinacija preproste uporabe in popolne združljivosti s formatom JSON je glavna prednost formata JSON-LD. Predvsem je pomembno to, da je možno obstoječe dokumente JSON oz. strukture nadgraditi brez večjih težav, spreminjanja poteka dela, uvajanja novih orodij in ogrodij itd.

Osnovna ideja formata JSON-LD je torej ustvarjanje opisov podatkov v obliki konteksta [8].

Izpis 3.2: Primer dokumenta JSON-LD

```
{
  "@context": "http://schema.org/",
  "@id": "http://tinemislej.com/mscthesis",
  "@type": "Thesis",
  "author": {
    "@type":"Person",
    "name":"Tine Mislej"
  },
  "inSupportOf": "Semantic Composition",
  "name": "Composition of semantic REST web services",
  "sourceOrganization": "University of Ljubljana"
}
```

Kontekst, definiran s ključno besedo `@context`, je lahko definiran v dokumentu, skupaj s podatki, ali pa ločeno, in je naslovljen prek povezave URL. Povezavo do dokumenta s kontekstom je mogoče umestiti tudi v zaglavje odgovorov HTTP in na ta način nadgraditi obstoječe dokumente JSON, brez posegov v njihovo strukturo.

3.3 RDF

RDF (angl. Resource Description Framework) je standard W3C, prvotno namenjen standardizaciji definicij in uporabe metapodatkovnih opisov spletnih virov, vendar je prav tako primeren za opisovanje poljubnih podatkov [9]. Postal je priljubljen podatkovni model za izpostavljanje javnih podatkov na internetu. RDF je prilagodljiv mehanizem za opisovanje entitet, tako da lahko različni izdajatelji dodajo nove informacije o isti entiteti ali pa ustvarijo nove povezave med ločenimi entitetami [10].
RDF-ov podatkovni model je graf. Vozlišča v grafu so entitete, katerih razmerja so predstavljena s povezavami. RDF sestavljajo trije osnovni elementi:

- viri - stvari, ki jih opisujemo;
- lastnosti - razmerja med stvarmi, ki jih opisujemo;
- vrednost lastnosti - dejanska vrednost, določene lastnosti.

Ti elementi sestavljajo **izjavo** RDF v obliki trojke `<subjekt, lastnost, objekt>`.

Izpis 3.3: Primer izjave

```
<http://example.org/thesis/mscthesis> <schema:creator>
"Tine Mislej"
```

RDF definira množico lastnosti, kot so `type`, `range`, `domain`, `subClassOf`, `subPropertyOf` itd. Lastnosti `subClassOf` in `subPropertyOf` sta tudi pomembna sestavna dela naše rešitve. S poizvedbami, ki uporabljajo omenjeni lastnosti, iščemo t. i. podkoncepte, tj. koncepte, ki izhajajo iz določenega koncepta. Na ta način pridobivamo dodatna znanja, potrebna za presojo o semantični skladnosti spletnih storitev.

3.4 SPARQL

SPARQL je semantični poizvedovalni jezik za podatke, shranjene v formatu RDF, standardiziran v okviru W3C. SPARQL deluje po načelu ujemanja grafov. Poizvedba SPARQL je sestavljena iz vzorcev, katerih ujemanje se izvede na viru podatkov. Vrednosti, ujemajoče s poizvedbo, so obdelane tako, da vrnejo želeni odgovor.

Izpis 3.4: Primer poizvedbe SPARQL

```
PREFIX o: <http://dbpedia.org/ontology/>
PREFIX r: <http://dbpedia.org/resource/>
SELECT ?s WHERE {
 ?s a o:City ;
    o:country r:Slovenia
}
```

3.5 Apache Jena Fuseki

Apache Jena Fuseki je samostojni strežnik SPARQL. V našem primeru smo strežnik uporabili za gostovanje ontologije in podpiranje poizvedb SPARQL v algoritmih za kompozicijo. Strežnik teče na virtualnem strežniku Windows Azure in je ločen od strežnikov za spletni storitvi in spletno aplikacijo.

Poglavje 4

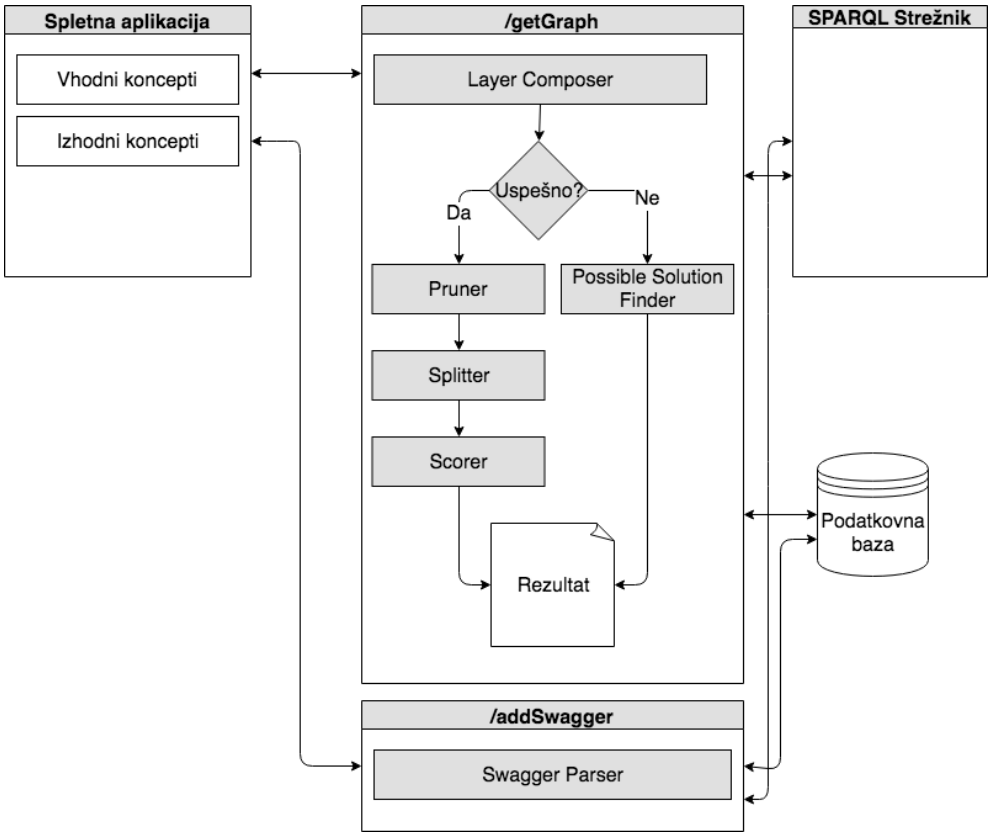
Arhitektura

V tem poglavju opišemo arhitekturo podporne aplikacije za semantično kompozicijo spletnih storitev. Gre za trinivojsko arhitekturo, kjer med seboj komunicirajo trije neodvisni deli - ospredje, tj. vmesnik spletne aplikacije, izpostavljene spletne storitve REST in strežnik SPARQL.

Aplikacija je zasnovana modularno. Vsaka raven je ločena od preostalih dveh in teče na namenskemu strežniku. Modularno so sestavljene tudi posamezne ravni in tako omogočajo enostavno zamenjavo obstoječih komponent z alternativnimi implementacijami.

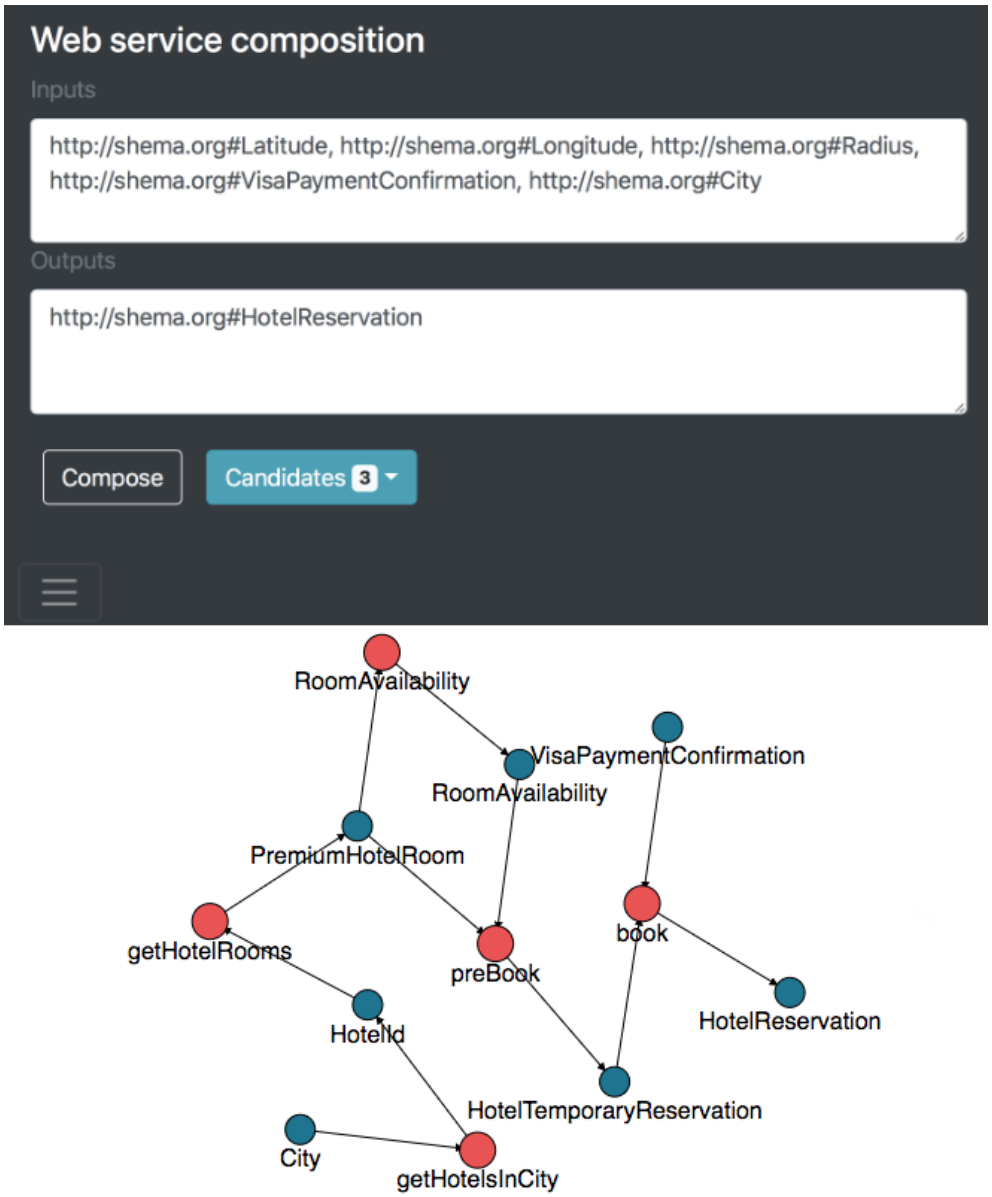
4.1 Spletna aplikacija

Gre za spletni vmesnik, implementiran s pomočje ogrodja Vue. Vmesnik omogoča vnos vhodnih in izhodnih parametrov in komunicira s spletno storitvijo `/getGraph`. Če je kompozicija možna oz. uspešna, je rezultat poizvedbe množica kandidatov. V nasprotnem primeru storitev odgovori z množico manjkajočih preslikav (oz. predlogov spletnih storitev).
Vsak kandidat je skupek informacij, ki definirajo določen graf. Ta je aciklični usmerjeni graf vozlišč, ki predstavljajo vhodne in/ali izhodne parametre ter storitve. Spletna aplikacija vizualizira vse kandidate, začenši z najprimer-



Slika 4.1: Arhitektura podporne aplikacije

nejšim - ocenjevanje primernosti kandidatov je opisano v naslednjih poglavjih. Informacije o posameznem kandidatu uporabimo za generiranje grafa, ki ga vizualiziramo s knjižnico D3.



Slika 4.2: Spletna aplikacija

4.2 Spletni storitvi

Razvili smo dve spletni storitvi REST. Storitev `addSwager` omogoča nalaganje in razčlenjevanje dokumenta, definiranega v razširjenem standardu Swagger. Razčlenjeni podatki so shranjeni v podatkovno bazo SQL in kasneje uporabljani v času izvajanja kompozicije. Te podatke uporablja storitev `getGraph`, ki izvaja kompozicijo. Izhod te storitve je seznam kandidatov za kompozicijo oz., ko kompozicija ni možna, seznam predlogov rešitev. Storitev `getGraph` je sestavljena iz naslednjih modulov (slika 4.1):

- *LayerComposer* - modul za gradnjo kompozicijskega grafa;
- *Pruner* - modul za odstranjevanje odvečnih storitev iz kompozicijskega grafa;
- *Splitter* - modul za iskanje kandidatov v kompozicijskem grafu;
- *Scorer* - modul za ocenjevanje kandidatov;
- *PossibleSolutionFinder* - modul za iskanje potencialnih rešitev kompozicije.

Spletna aplikacija in storitvi gostujejo na spletni platformi Heroku [20].

4.3 Strežnik SPARQL

Storitvi `addSwager` in `getGraph` za svoje delovanje potrebujeta strežnik SPARQL (Apache Jena Fuseki), ki omogoča presojanje o konceptih. Strežnik SPARQL smo namestili na operacijski sistem Ubuntu [21], ki teče na oblačni platformi Microsoft Azure [23]. Platforma v okviru oblačne hrambe hrani tudi datoteko, v kateri je definirana ontologija. Dostop do datoteke potrebuje algoritem za iskanje predlogov rešitev, ki preverja sorodnost konceptov.

Poglavje 5

Razširitev standarda Swagger

Eden izmed glavnih prispevkov dela je razširitev standarda Swagger. Dokumenti Swagger so običajno zapisani v notacijah JSON ali YAML. V delu smo se omejili na različico JSON standarda Swagger 2.0. Glavna motivacija za izbiro tega standarda sta bili njegova široka sprejetost v skupnosti in enostavnost uporabe.

5.1 Zahteve in vodila

Enostavnost in berljivost sta bili tudi eni izmed glavnih vodil snovanja razširitve. Glavna zahteva razširitve je bila **popolna skladnost** s standardom. S tem smo želeli dopustiti odprte poti pri sprejemanju te rešitve oz. njeni uporabi kot osnovi za nove razširitve. Neskladnost oz. delna skladnost bi po našem mnenju pomenila le nov poskus definiranja semantičnih spletnih storitev - neskladen s standardom Swagger. Druga zelo pomembna zahteva je bila **enostavnost nadgradnje**. Želeli smo, da je obstoječe dokumente Swagger mogoče semantično razširiti z minimalnim vložkom, brez dodatnih ogrodij in orodij. Kot zadnjo glavno zahtevo, navezujočo na zahtevo po enostavnosti nadgradnje, smo definirali **količino dodatnega znanja**, potrebnega za razširitev dokumenta. Menimo, da predvsem pri uporabi semantičnih tehnologij prihaja do določene stopnje odpora za njeno uporabo. V članku

[8] avtorji navajajo pojav *semafobije (angl. semaphobia)*, pri katerem gre za strah pred uporabo semantičnih tehnologij, saj so te praviloma zaznane kot kompleksne. To smo poskusili preprečiti z uporabo formata JSON-LD. Ta omogoča semantično razširitev, ki terja relativno majhno količino potrebnega znanja za izvedbo.

5.2 Razširitev

V naslednjih poglavjih je opisana razširitev standarda, ki je sestavljena iz dveh delov. **Semantične značke** opisujejo oz. razširjajo v dokumentu definirane zahtevke (angl. request) oz. možne odgovore (angl. response) operacij na spletnih storitvah. **Nefunkcionalni parametri** opisujejo nefunkcionalne lastnosti operacije. Te lastnosti so lahko cena uporabe, varnost spletne storitve itd.

5.2.1 Semantične značke

Namen semantičnega označevanja je dodajanje pomena parametrom in odgovorom spletnih storitev. Brez semantičnih označb so ti konstrukti le vrsta znakov, brez kakršnega koli pomena za programsko opremo, ki jih (samodejno) obdeluje. Pomen posameznih parametrov in odgovorov spletnih storitev je torej dokumentiran izven definicije storitve, v primeru odsotnosti take dokumentacije pa je delo inženirja, da razbere pomen posameznih konstruktov. Pogosto to pomeni, da prihaja do precej tesne sklopljenosti med storitvami in odjemalci, ki jih uporabljajo. Dejstvo je, da je razumevanje takih konstruktov vkodirano v odjemalca, ki je posledično precej neprilagodljiv. Cilj našega dela je bil razširitev definicij parametrov in odgovorov v standardu Swagger s semantičnimi značkami.

Semantične značke omogočajo višjo stopnjo opismenjevanja pomena parametrov in odgovorov spletnih storitev, natančnejše semantično presojanje o pomenu storitev in posledično semantično kompozicijo.

Izpis 5.1 prikazuje izsek iz dokumenta Swagger, razširjenega s semantičnimi značkami.

Izpis 5.1: Izsek razširitve s semantičnimi značkami

```
{
  "description": "Hotel Booking",
  "schema":{
    "type": "object",
    "properties":{
      "from":{
        "type":"string",
        "format":"date-time"
      },
      "to":{
        "type":"string",
        "format":"date-time"
      },
      "hotel_id":{
        "type":"string",
        "x-semantics": {
          "@type": "http://shema.org#HotelId"
        }
      },
      "room_id":{
        "type":"string",
       "x-semantics": {
          "@type": "http://shema.org#HotelRoomId"
        }
      }
```

```
    },
    "x-semantics": {
      "@type": "http://shema.org#HotelReservation"
    }
  }
}
```

Semantična značka je preprost objekt JSON-LD, definiran pod ključno besedo `x-semantics`. Najosnovnejša oblika značke zahteva samo definicijo (semantičnega) tipa značke - rezervirana ključna beseda `@type`. Značka je razširljiva, saj je vanjo možno vključiti dodatne informacije. Ker gre za običajen objekt JSON-LD, lahko definiramo kontekst (`@context`) ali pa informacije vključimo v običajnem načinu JSON.

Semantične značke uporabljamo na ravni definicije zahtevka oz. odgovora določene spletne storitve. Z njimi označujemo potrebne dele lastnosti oz. različne podobjekte, kot to prikazuje izpis 5.1. Oblika oz. zasnova semantičnih značk je pogojena z omejitvami standarda Swagger. V prvi fazi smo opise Swagger želeli razširiti s formatom JSON-LD, kar pa za enkrat še ni možno zaradi različnih omejitev. Standard Swagger omogoča razširjanje le na določenih mestih v dokumentu in z vnaprej določeno obliko poimenovanja razširitev. Zaradi tega je bila problematična izvedba prve ideje, ki je bila razširjanje celotne definicije zahtevkov oz. odgovorov spletne storitve s formatom JSON-LD. Standard Swagger ne dopušča poljubnih lastnosti v določenih objektih JSON, ki so sestavni del dokumenta Swagger. Zaradi tega je tu nemogoče umestiti že kontekst (`@context`) objekta JSON-LD. Zaradi teh omejitev smo prvo idejo opustili in nadaljevali v smeri semantičnih značk.

5.2.2 Nefunkcionalni parametri

Ta razširitev določa nefukncionalne lastnosti operacije spletne storitve, ki so v praksi pomemben dejavnik pri uporabi spletne storitve. Te so lahko SLA, varnost, cena, omejitev dostopa itd. S to razširitvijo smo želeli opisati doda-

tne informacije o storitvi (oz. operaciji) ter jih ponuditi v človeku prijazni in hkrati programsko berljivi obliki. Nefunkcionalne parametre v podporni aplikaciji uporabljamo za ocenjevanje primernosti kandidatov za kompozicijo. Tudi ta razširitev temelji na formatu JSON-LD in uporabi ontologije, kjer so pomeni nefukncionalnih zahtev natančneje definirani.

Izpis 5.2: Primer nefunkcionalnih parametrov

```
x-nonfunctional-parameters": {
  "@context": {
    "pricing": "http://sws.org#Pricing",
    "request_limit": "http://sws.org#Request_Limit",
    "security": "http://sws.org#WebServiceSecurity"
  },
  "@type": "http://sws.org#NonFunctionalParameters",
  "pricing": "http://sws.org#Free",
  "request_limit": {
    "@context": {
      "daily": "http://sws.org#DailyLimit",
      "limit": "http://sws.org#Limit"
    },
    "@type": "daily",
    "limit": 1000
  },
  "security": "http://sws.org#HighSecurity"
}
```

Izpis 5.2 prikazuje primer razširitve operacije, definirane v dokumentu Swagger z nefunkcionalnimi parametri. Pomeni posameznih parametrov in njihovih vrednosti so določeni z IRI-jem, kar pomeni, da so edinstveni in praviloma nedvoumni. Razširitev iz izpisa 5.2 navaja tri nefunkcionalne parametre, ki dodatno opisujejo dotično operacijo. Objekt JSON pod ključno

besedo `@context` določa kontekst razširitve. Posameznim označbam, uporabljenim v razširitvi, so določeni IRI-ji, ki določajo njihov pomen. V nadaljevanju so definiranim označbam dodeljene vrednosti. Vrednost oz. njeno obliko (sintakso) določa ontologija, v kateri so koncepti definirani.
V primeru izpisa 5.2 sta vrednosti parametrov pricing in security (`http://sws.org#Pricing` in `http://sws.org#WebServiceSecurity`) vedno določeni z IRI-jem. Vrednost parametra pricing je lahko `http://sws.org#Free` ali `http://sws.org#Paid`, vrednost parametra security pa `http://sws.org#HighSecurity`, `http://sws.org#MediumSecurity` ali `http://sws.org#LowSecurity`.
Vrednosti parametrov pa niso lahko le IRI-ji, temveč katerekoli veljavne vrednosti JSON. Vrednost parametra `request_limit` je tako objekt JSON-LD, ki natančneje opisuje ta parameter. Tip objekta je določen z IRI-jem `http://sws.org#DailyLimit` in vrednostjo lastnosti limit 1000.
Razširitev `x-non-functional-parameters` je veljaven objekt JSON-LD. Njena oblika je v večji meri pogojena z definicijami konceptov v ontologiji, ki jo uporablja.

Poglavje 6

Semantična kompozicija

V tem poglavju opišemo svoj pristop k razvoju podporne storitve za semantično kompozicijo spletnih storitev, opisanih v razširjenem standardu Swagger. Naš cilj je bil razvoj spletne aplikacije za pomoč razvijalcem pri semantični kompoziciji spletnih storitev.

6.1 Iskanje spletnih storitev

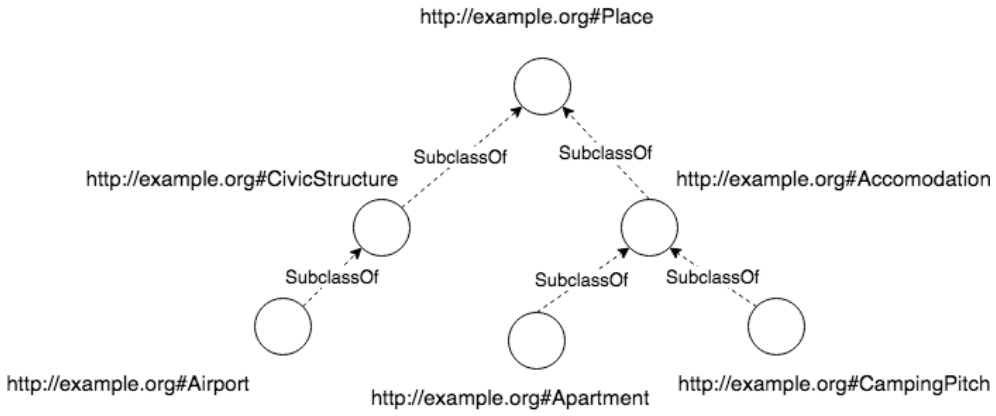
Temeljna funkcionalnost, potrebna za (semantično) kompozicijo, je iskanje oz. odkrivanje spletnih storitev, ki je podprto z uporabo konceptov. Koncept je abstrakcija pomena, predstavljena s točno določenim URL-jem. Z vidika aplikacije je spletna storitev definirana kot $S = (In_S, Out_S)\ \epsilon\ R$, kjer je In_S množica zahtevanih vhodnih konceptov spletne storitve S, Out_S množica izhodnih konceptov spletne storitve S, R pa množica vseh spletnih storitev. Vsak vhodni oz. izhodni koncept je definiran v ontologiji O. Velja torej $In_S \cup Out_S \subset O$.
Vloga vhodnih oz. izhodnih konceptov je torej dvojna. V prvi fazi služijo za iskanje primernih storitev, v drugi pa za omogočanje same kompozicije.
Definiramo lahko več stopenj semantičnega ujemanja [11]:

- **Točna (angl. Exact)**: Izhodni koncept out_{S1} storitve S1 je enak (isti) vhodnemu konceptu in_{S2} storitve S2.

- **Vključujoča (angl. Plugin)**: Izhodni koncept out_{S1} storitve S1 se ujema z vhodnim konceptom in_{S2} storitve S2. Pri tem gre za vključujoče ujemanje - in_{S2} je nadkoncept koncepta out_{S1}. Praviloma velja, da je koncept in_{S2} bolj splošen od koncepta out_{S1}.

- **Podvržena (angl. Subsume)**: V primerjavi z vključujočo stopnjo gre za obratno stopnjo ujemanja. Izhodni koncept out_{S1} storitve S1 se ujema z vhodnim konceptom in_{S2} storitve S2, vendar je out_{S1} nadkoncept koncepta in_{S2}.

- **Spodletela (angl. Fail)**: Ko nobena od prejšnjih stopenj ujemanj ni možna, govorimo o spodletelem ujemanju. V osnovi to pomeni, da sta koncepta semantično neskladna.



Slika 6.1: Preprosta ontologija.

Predpostavimo primer preproste ontologije, predstavljene na sliki 6.1, in množico storitev:
$S1 = (In = \emptyset, Out = \{http://example.org\#CivicStructure\})$,
$S2 = (In = \{http://example.org\#CivicStructure\}, Out = \emptyset)$,
$S3 = (In = \emptyset, Out = \{http://example.org\#Airport\})$,
$S4 = (In = \emptyset, Out = \{http://example.org\#Place\})$.

Storitvi S1 in S2 sta semantično skladni. Izhodni koncept storitve S1 je semantično skladen z vhodnim konceptom storitve S2 - gre za **točno skladnost**, saj gre za isti koncept. Prav tako sta skladni storitvi S3 in S2. V tem primeru gre za **vključujoč** tip skladnosti, saj je izhodni koncept storitve S3 `http://example.org#Airport` podkoncept vhodnega koncepta storitve S2 `http://example.org#CivicStructure`.
Iz primera je razvidno, da sta v primeru semantične kompozicije smiselni le točna in vključujoča stopnja skladnosti. Samo v tem primeru lahko drugo storitev kličemo z uporabo izhodnega koncepta prve storitve. V primeru podvržene stopnje skladnosti (storitvi S4 in S2) ne moremo z gotovostjo trditi, da lahko drugo storitev kličemo z izhodnim konceptom prve storitve, medtem ko pri spodleteli stopnji skladnosti ne moremo govoriti o skladnosti - posledično kompozicija ni možna.

Semantično skladnost konceptov c_1 in c_2 torej definiramo kot

Definicija 1. $c_1 \equiv c_2 \vee c_1 \subseteq c_2$

Rezultat preverjanja skladnosti pa ne nujno velja tudi v nasprotni smeri. Torej, če je koncept c_1 semantično skladen s konceptom c_2, ni nujno, da je koncept c_2 semantično skladen s konceptom c_1 - operacija preverjanja semantične skladnosti torej ni komutativna.

Semantično skladnost preverjamo v času gradnje splošnega grafa kompozicije - to je v času izvajanja aplikacije, ko uporabnik sproži zahtevo za kompozicijo. Presojanje o podkonceptih določenega koncepta se izvede že v času objave storitve. Operacijo preverjanja semantične skladnosti, definirane v definiciji 1, smo podprli z uporabo strežnika SPARQL. Ta gosti ontologijo, v kateri so definirani koncepti vseh storitev.
Za preverjanje semantične skladnosti je v aplikaciji zadolžen primerek razreda `Matcher` (slika 4.1). Za poizvedovanje po podkonceptih določenega koncepta uporabljamo naslednjo poizvedbo SPARQL.

Izpis 6.1: Poizvedba SPARQL

```
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
SELECT DISTINCT ?class
WHERE {
    { ?class rdfs:subClassOf+ <%s> }
        UNION { ?class rdfs:subPropertyOf+ <%s> }
}
```

6.2 Kompozicija

Kompozicija je potrebna v primeru, ko repozitorij ne vsebuje iskane oz. želene spletne storitve - iskane na način, opisan v prejšnjem poglavju. Kompozicija spletnih storitev je v osnovni sestavljanje oz. združevanje spletnih storitev z namenom ustvarjanja nove spletne storitve, ki rešuje dani problem. Za združene spletne storitve praviloma velja, da je izhod ene storitve (lahko) uporabljen kot vhod v drugo storitev. Spodnja definicija definira zmožnost kompozicije dveh spletnih storitev.

Definicija 2. *Če je izhod storitve W_1 koncept O_1 in storitev W_2 porabi O_1 kot svoj vhod, potem lahko sklenemo, da sta storitvi sestavljivi. Kompozicijo spletnih storitev torej lahko definiramo kot samodejno iskanje končnega zaporedja spletnih storitev v registru. [12]*

Problem kompozicije je torej doseči zadani cilj, določen v začetnem zahtevku, brez razkrivanja podrobnosti kompozicije [12]. Uporabniki naše aplikacije cilj opredelijo v obliki poizvedbe, ki določa vhodne in izhodne semantične parametre sestavljene storitve.

Spletno storitev definiramo kot S = [I_S, O_S], uporabnikovo poizvedbo pa kot Q = [I_Q ,O_Q]. Kompozicija je veljavna, ko so uresničena naslednja merila: [12]

1. $\exists S_i(I_Q \subseteq I_{S_i})$

2. $\exists S_i(O_Q \subseteq O_{S_i})$

3. $\forall S_i(S_i \rightarrow S_j)$ - obstaja vsaj ena pot iz S_i do S_j.

V prvi fazi kompozicije lahko nastopajo samo spletne storitve, katerih vhodi so podani v zahtevku (I_Q). V zadnji fazi mora kompozicija vsebovati vse izhodne parametre, določene v poizvedbi (O_Q). V vmesnih fazah lahko nastopajo katerekoli storitve, ki zagotavljajo vhod za neko drugo storitev ali pa je njihov izhod del izhodnih parametrov O_Q. Storitve, ki s stališča kompozicije nimajo dodane vrednosti (npr. so odvečne - njihovi izhodi niso uporabljeni v kompoziciji), ne smejo biti del kompozicije.

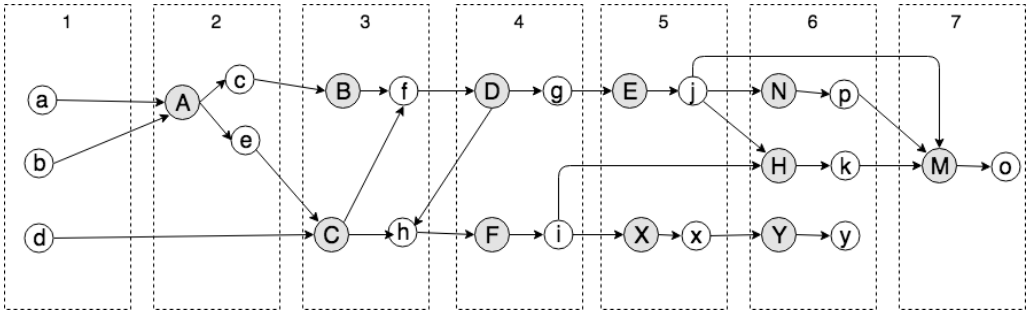
6.2.1 Kompozicijski graf

Najpomembnejši del kompozicije je gradnja kompozicijskega grafa. Kompozicijski graf je usmerjen aciklični graf vseh storitev in njihovih parametrov, ki potencialno lahko sodelujejo v kompoziciji. Cikel je pot poljubne dolžine med vozlišči v grafu, ki vodi nazaj v izvorno vozlišče. Aciklični graf je torej graf, ki takih poti ne dovoljuje oz. vsebuje.
Gradnja kompozicijskega grafa se izvede za vsako zahtevo po kompoziciji - tj. v času izvajanja kompozicije. Kompozicijski graf G = (V, E) je acikličen usmerjeni graf, kjer velja:

1. $V = S \cup C$ je množica vseh vozlišč v grafu, kjer je `S` množica vseh vozlišč, ki predstavljajo storitev, `C` pa množica vseh vhodnih in izhodnih parametrov storitev iz množice `S`.

2. $E = E_{in} \cup E_{out}$ je množica vseh povezav med vozlišči. E_{in} je množica vseh povezav med vozliščem C_i, ki predstavlja vhodni parameter storitve in vozliščem S_i, ki predstavlja storitev, katere vhodni parameter je C_i.

Slika 6.2 prikazuje primer kompozicijskega grafa za poizvedbo `Q = (in=[a,b,d], out=[o])`. Temnejša vozlišča predstavljajo storitve v



Slika 6.2: Kompozicijski graf za poizvedbo `Q = (in=[a,b,d], out=[o])`

grafu, svetlejša in manjša pa vhodne in/ali izhodne parametre teh storitev. Skladno z zgoraj naštetimi pravili ta graf vsebuje vse storitve, ki jih lahko posredno ali neposredno kličemo z vhodnimi parametri poizvedbe `Q`.

Gradnja kompozicijskega grafa poteka postopno - v plasteh (na sliki 6.2 so to pravokotniki označeni z 1, 2 ..., 7). Vsaka plast vsebuje tiste storitve (in pripadajoče izhodne parametre), ki jih lahko v tej plasti kličemo s parametri, zagotovljenimi v prešnjih plasteh - torej so v danem trenutku na voljo vsi vhodni parametri določene storitve. Posebni plasti sta prva in zadnja plast. Prva plast vsebuje samo vhodne parametre poizvedbe `Q` in predstavlja vhod nove sestavljene storitve. Zadnja plast vsebuje vse storitve iz registra, katerih izhod je vsaj eden izmed izhodnih parametrov poizvedbe `Q` in tako ponazarja izhod sestavljene storitve. Vsaka vmesna plast N lahko uporablja parametre iz plasti 1...N-1.

Algoritem 1 prikazuje psevdokodo za generiranje kompozicijskega grafa. Vhod v algoritem sta seznama vhodnih in izhodnih parametrov sestavljene storitve. Parametri, predstavljeni kot nizi, so URL-ji, ki določajo koncepte v ontologiji. V začetni fazi izvajanja algoritma le-ta komunicira s strežnikom SPARQL, z namenom pridobitve konkretnejših tipov vhodnih in izhodnih parametrov (funkcija `getSubconcepts` v 10. vrstici) za potrebe nadaljnega presojanja o uporabnosti konceptov v kompoziciji. V tej fazi so v graf postavljena vozlišča, ki predstavljajo vhodne parametre. Gre za prvo fazo gradnje

kompozicijskega grafa, ki je na sliki 6.2 označena s številko 1. V nadaljnih fazah v graf dodajamo in povezujemo vozlišča, ki predstavljajo storitve, ki jih lahko kličemo s koncepti, ki so na voljo v določeni fazi - shranjeni v množici `availableConcepts`. Storitev se dojema kot pripravljena za klicanje, če so vsi njeni vhodni parametri na voljo in v skladu z definicijo 1. Vozlišča ustreznih storitev dodamo v graf in povežemo z njihovimi vhodnimi koncepti. Na tej točki v graf dodamo tudi vozlišča, ki predstavljajo izhodne parametre in ustrezno povežemo z vozlišči, ki predstavljajo storitve. Zadnja stvar vsake faze je razširjanje množice `availableConcepts` z izhodnimi koncepti vsake dodane storitve, saj so ti na voljo šele v naslednji fazi.

Gradnja kompozicijskega grafa se zaključi, ko v repozitoriju ni več drugih storitev, ki jih lahko kličemo z razpoložljivimi koncepti - vhodnimi koncepti in tistimi, pridobljenimi v vmesnih fazah. Izhod algoritma je odvisen od uspešnosti gradnje kompozicijskega grafa. Če je gradnja uspešno zaključena, torej je kompozicija možna, je izhod algoritma kompozicijski graf. V primeru neuspešne kompozicije (zahteve izhodnih parametrov niso izpolnjene) algoritem začne z gradnjo grafa v nasprotni smeri - od izhodnih parametrov nazaj. Gradnja poteka po istih načelih, spremenjena je le njena smer. Izhod algoritma v tem primeru je par dveh grafov. Grafa sta potrebna za primerjanje razlik in iskanje potencialnih preslikav med njima. V primeru neuspešne kompozicije ju uporabimo za predlaganje novih storitev, ki bi omogočile kompozicijo. Ta postopek predstavimo v nadaljevanju.

Algoritem 1 Psevdokoda algoritma za gradnjo kompozicijskega grafa

Vhod:

1: $inputConcepts : List(String)$ ▷ seznam vhodnih konceptov
2: $outputConcepts : List(String)$ ▷ seznam izhodnih konceptov

Izhod:

3: graph ▷ kompozicijski graf - v primeru uspešne gradnje
4: reversed ▷ del kompozicijskega grafa, grajen v obratni smeri - v primeru neuspešne gradnje

5: $graph \leftarrow \emptyset$
6: $ended \leftarrow false$
7: $availableConcepts \leftarrow \emptyset$
8: $outputs \leftarrow \emptyset$
9: **for all** outputConcept in outputConcepts **do**
10: $concept \leftarrow newConcept(outputConcept, getSubconcepts(outputConcept))$
11: $outputs \leftarrow outputs + concept$
12: **end for**
13: **for all** inputConcept in inputConcepts **do**
14: $concept \leftarrow newConcept(inputConcept, getSubconcepts(inputConcept))$
15: $availableConcepts \leftarrow availableConcepts + concept$
16: $paramNode \leftarrow newParameterNode(concept)$
17: $addVertex(graph, paramNode)$
18: **end for**
19: **repeat**
20: $outputConcepts \leftarrow \emptyset$
21: **for all** nonVisitedEndpoint in repository **do**
22: **if** can call nonVisitedEndpoint with availableConcept **then**
23: $invocationConcepts \leftarrow nonVisitedEndpoint.inputParameters$
24: $endpointNode \leftarrow newEndpointNode(nonVisitedEndpoint.name)$
25: $addVertex(endpointNode, graph)$
26: **for all** invocationConcept in invocationConcepts **do**
27: **if** invocationConcept exists in graph **then**
28: $addEdge(invocationConcept, endpointNode)$
29: **else**
30: $paramNode \leftarrow newParametrNode(invocationConcept)$
31: $addVertex(graph, paramNode)$
32: $addEdge(paramNode, endpointNode)$
33: **end if**
34: **end for**
35: $p \leftarrow nonVisitedEndpoint.outputParameters$
36: $outputConcepts \leftarrow outputConcepts + p$

Algoritem 1 Psevdokoda algoritma za gradnjo kompozicijskega grafa (nadaljevanje)

```
37:             for all outputConcept in nonVisitedEndpoint.outputConcepts do
38:                 if outputConcept is in nonVisitedEndpoint.inputConcepts then
39:                     continue
40:                     if outputConcept exists in graph then
41:                         addEdge(endpointNode, outputConcept)
42:                     else
43:                         outpuConceptNode ← newParameterNode(outputConcept)
44:                         addVertex(graph, outputConceptNode)
45:                         addEdge(endpointNode, outputConceptNode
46:                     end if
47:                 end if
48:             end for
49:         end if
50:     end for
51:     availableConcepts ← availableConcepts + outputConcepts
52:     ended ← shouldEnd(availableConcepts)
53: until !ended
54: if hasEndedSuccessfuly then
55:     return graph
56: else
57:     reverse ← reverseBuild(outputConcepts)
58:     return(graph, reverse)
59: end if
```

6.3 Predlaganje rešitev

Nezmožnost kompozicije je relativno pogost pojav. Razlog za to so praviloma manjkajoče spletne storitve. V primeru neuspešne kompozicije aplikacija skuša predlagati primerne preslikave, ki bi želeno kompozicijo omogočile. Preslikava je model spletne storitve ali storitev, ki bi kompozicijo omogočile. V primeru neuspešne kompozicije bo izhod algoritma za gradnjo kompozicijskega grafa dvodelen. Prvi del je nepopoln kompozicijski graf, grajen od

vhodnih parametrov proti izhodnim. Drugi del rezultata je prav tako nepopoln kompozicijski graf, grajen v obratni smeri, od izhodnih konceptov proti vhodnim. Med omenjenima deloma skušamo najti čim bolj optimalne preslikave iz enega v drugega. Število preslikav je odvisno od oblike delov. Preprosta, sekvenčna kompozicija navadno zahteva eno preslikavo, medtem ko je v kompleksnejših primerih potrebno tudi več različnih preslikav.
Naš algoritem poišče možne preslikave med deloma in predlaga najboljše. Predlogi rešitev so združeni glede na število potrebnih preslikav, ki sestavljajo posamezni predlog. Ti so med seboj razvrščeni glede na oceno primernosti posameznega predloga. Največje število predlogov je parameter delovanja, hevristično nastavljen na vrednost 2. Aplikacija torej predlaga najboljša dva predloga iz vsake skupine.
Iskanje predlogov poteka v naslednjih korakih:

1. **iskanje možnih predlogov**;
2. **izračun semantičnih razdalj med preslikavami**;
3. **razvrščanje predlogov**.

6.3.1 Iskanje možnih predlogov

Iskanje predlogov možnih rešitev je dokaj pomemben vidik aplikacije, saj zaradi manjkajočih rešitev kompozicija pogosta ni možna. V takih primerih je smiselno, da uporabniku skušamo predlagati možne rešitve, ki bodo kompozicijo omogočile, saj se pogosto izkaže, da v množici storitev za kompozicijo manjka le malo storitev ali pa zgolj ena sama.
Predlog rešitve so preslikave med koncepti, ki jih imamo na voljo, in koncepti, ki jih potrebujemo. Preslikava `P=[`In_P`,` Out_P`]` je definirana podobno kot semantična spletna storitev. Cilj algoritma za iskanje možnih predlogov je najti take preslikave oz. kombinacije le-teh, ki bodo omogočile kompozicijo.
Prva faza iskanja možnih predlogov je iskanje takih kombinacij konceptov,

ki lahko ponudijo rešitev. Najosnovnejši tak primer je kar izhod Out_Q poizvedbe `Q`. Vse nadaljne kombinacije izhajajo iz te najosnovnejše kombinacije in kombinacij, ki jih dobimo v naslednjih korakih te faze. V posameznih korakih te faze vsak koncept iz posamezne kombinacije skušamo zamenjati z novim konceptom tako, da bo kombinacija še vedno veljavna. Nov koncept poiščemo med storitvami, ki jih imamo na voljo - to so tiste storitve, ki jih lahko kličemo z vhodnimi parametri oz. tistimi parametri, pridobljenimi v vmesnih korakih gradnje nepopolnega kompozicijskega grafa. Rezultat te faze je množica kombinacij, ki lahko v sklopu te nepopolne kompozicije vrne želeni rezultat. Pomembno je poudariti, da so nekatere kombinacije samozadostne - torej ne potrebujejo nobenih storitev iz nepopolne kompozicije. Take kombinacije dejansko že predstavljajo sestavljeno storitev. Druge kombinacije pa bodo omogočile kompozicijo v kontekstu trenutne nepopolne kompozicije. V osnovi gre torej za preslikave, ki bodo v nepopolnem kompozicijskem grafu omogočile kompozicijo.

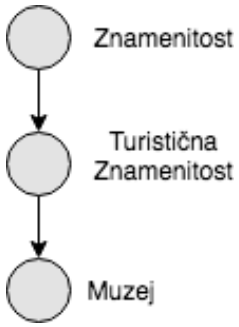
V drugi fazi uporabimo najdene kombinacije in začnemo iskati preslikave. Preslikava slika vhodne koncepte v izhodne. V našem primeru slikamo iz konceptov na levi strani grafa (to, kar imamo na voljo) v koncepte iz najdenih kombinacij (to, kar potrebujemo).

Najprej generiramo pare (preslikave). Za vsak koncept v kombinaciji generiramo pare med konceptom iz kombinacije in vsakim konceptom na levi strani. Za vsak par izračunamo tudi semantično razdaljo. Vmesni rezultat te faze je množica parov za vsak koncept iz trenutne kombinacije. Za dobljene množice parov generiramo kombinacije in tako dobimo predloge rešitev za trenutno kombinacijo. Ta postopek ponovimo za vsako kombinacijo, najdeno na začetku.

Algoritem išče vse možne kombinacije in je posledično kombinatorično zahteven. Število predlogov je odvisno od oblike levega in desnega dela, predvsem pa od števila storitev in konceptov v posameznih delih. Število predlogov je običajno veliko, zato uporabniku predlagamo le najboljše.

6.3.2 Izračun semantičnih razdalj med preslikavami

Semantična razdalja je dolžina najkrajše poti med konceptoma v ontologiji. V osnovi je ontologija graf, koncept pa določeno vozlišče v njem. Pomensko sorodni koncepti so tudi v grafu blizu skupaj. Njihova razdalja je relativno kratka, medtem ko je razdalja med daljno sorodnimi koncepti večja. Med določenima konceptoma v ontologiji taka razdalja lahko tudi ne obstaja. V tem primeru gre za različna koncepta, ki pomensko praviloma nimata nič skupnega. Semantične razdalje omogočajo boljše predlaganje rešitev, saj tako boljše presojamo o primernosti posamezne rešitve.
Slika 6.3 prikazuje del grafa ontologije. Semantična razdalja med konceptom `Znamenitost` in konceptom `Turistična Znamenitost` je 1, med `Znamenitost` in `Muzej` pa 2. V obratni smeri koncept razdalje praviloma ne velja (koncepta sta sicer še vedno semantično povezana) - torej med konceptom `Turistična Znamenitost` in konceptom `Znamenitost` razdalje ni, saj pot med njima ne obstaja oz. je enosmerna. Prav tako semantična razdalja ne obstaja, če sta koncepta v različnih komponentah grafa - torej nepovezana.



Slika 6.3: Primer dela grafa ontologije

V naši aplikaciji torej uporabljamo semantično razdaljo za presojanje o primernosti kandidatov. Predlog rešitve z najmanjšo vsoto semantičnih razdalj posameznih preslikav se dojema kot najboljši. Semantično razdaljo računamo med konceptoma `a` in `b` v obe smeri. Torej med `a` in `b` ter `b` in `a`. Upoštevana je vedno najkrajša razdalja.

Algoritem dostopa do datoteke na strežniku, kjer je definirana ontologija. Z iskanjem v širino išče najkrajše poti med konceptoma v grafu ontologije in v primeru več poti vrne poljubno. Pot je seznam izjav RDF, ki ustrezajo filtru. Upoštevamo tiste izjave, katerih ime predikata je bodisi `subClassOf` bodisi `subPropertyOf`.

6.3.3 Razvrščanje predlogov

Število predlogov je odvisno od kompleksnosti obeh delov nepopolnega kompozicijskega grafa. Število predlogov se hitro povečuje s številom spletnih storitev v obeh delih. Še več, na število predlogov vpliva predvsem število izhodnih, pa tudi vhodnih konceptov storitev v obeh delih.
Velike količine predlogov po našem mnenju lahko precej okrnejo izkušnjo, pa tudi večje dodane vrednosti nimajo. Zaradi tega jih najprej združimo skupaj, glede na število preslikav v predlogih. Znotraj vsake skupine predloge uredimo glede na seštevek semantičnih razdalj v vsakem predlogu. Tako aplikacija v primeru neuspešne kompozicije iz vsake skupine predlaga (največ) dve najboljši rešitvi.

6.4 Dodajanje pomožnih vozlišč

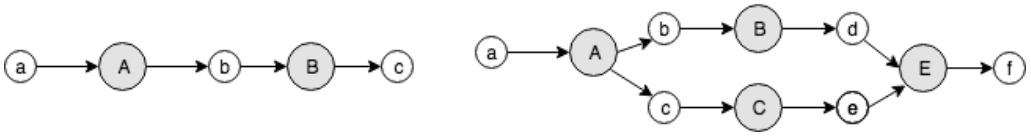
V primeru uspešne kompozicije za potrebe nadaljne obdelave kompozicijskega grafa vanj dodamo pomožni vozlišči. Gre za vozlišči, imenovani `START` in `END`. Vozlišči predstavljata pomožni storitvi. START predstavlja storitev, definirano kot $START = (\emptyset, I_Q)$. Gre za storitev brez vhodnih parametrov, katere izhod so vhodni parametri poizvedbe `Q`. Podobno definiramo vozlišče END, ta predstavlja storitev $END = (O_Q, \emptyset)$ - tj. storitev, katere vhod je izhod poizvedbe `Q`, njen izhod pa je prazen.
Vozlišči ne nastopata v predstavitvah potencialnih kandidatov za kompozicijo. Uporabljamo ju pri iskanju kandidatov in odstranjevanju odvečnih vozlišč - storitev iz kompozicijskega grafa.

6.5 Odstranjevanje odvečnih vozlišč

Kot zapisano, odvečne storitve ne smejo biti del kompozicije, saj take storitve nimajo nobene dodane vrednosti - kvečjemu le nižajo vrednost potencialnega kandidata za kompozicijo. Slika 6.2 prikazuje kompozicijski graf za poizvedbo `Q = (in=[a,b,d], out=[o])`. Opazimo lahko, da storitvi `X` in `Y` nimata dodane vrednosti za kompozicijo. Izhod storitve X je vhod v storitev Y, katere izhod pa ni uporabljen. Storitve, katerih noben izmed izhodnih parametrov ni potreben v kompoziciji, smatramo kot **odvečne**. Odstranjevanje odvečnih vozlišč poteka od konca - izhodnih vozlišč proti začetku - vhodnim vozliščem. Posledica odstranitve odvečne storitve je lahko nova odvečna storitev. V primeru slike 6.2 to velja za storitev `X` po odstranitvi storitve `Y` - zatorej, odstranjevanje poteka rekurzivno od konca proti začetku.

6.6 Iskanje kandidatov za kompozicijo

Iskanje kandidatov za kompozicijo se izvaja na optimiziranem kompozicijskem grafu. Ta graf ne vsebuje odvečnih vozlišč, ima pa dodani obe pomožni vozlišči.
Pri iskanju kandidatov smo podprli dve ključni obliki kompozicije - to sta **zaporedje** in **vejitev**. Ti obliki modelirata zaporedno oz. vzporedno izvajanje spletnih storitev in tako obravnavata večino praktičnih primerov kompozicije.



Slika 6.4: Zaporedje (levo) in vejitev (desno)

Iskanje kandidatov za kompozicijo je v osnovi problem iskanja različnih podgrafov med določenimi vozlišči. Cilj našega algoritma je poiskati vse ti-

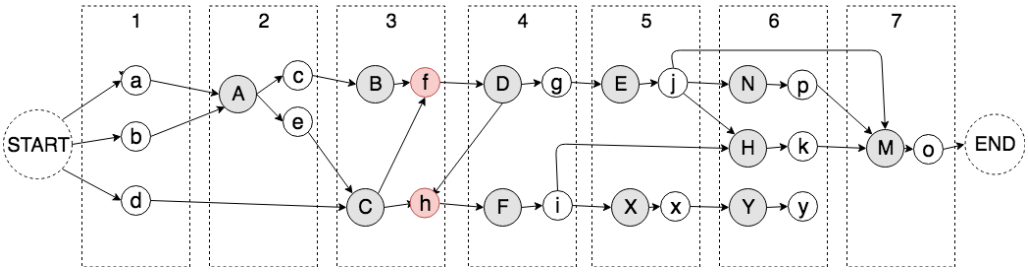
ste podgrafe med vozlišči, ki predstavljajo vhodne in izhodne koncepte, ki predstavljajo možnega kandidata za kompozicijo.
Najpreprostejša oblika kompozicije je zaporedje storitev poljubne dolžine, kot je predstavljeno na sliki 6.4, in zato najenostavnejša oblika za iskanje kandidatov, saj gre v osnovi za problem iskanja vseh poti v grafu. Kompozicijski graf pa je običajno kompleksnejši, sestavljen iz zaporedij in zahtevnejših vejitev.
Iskanje kandidatov poteka z gradnjo usmerjenih acikličnih grafov od končnega pomožnega vozlišča - `END` proti začetnemu pomožnemu vozlišču - `START`. Gre za algoritem BFS, ki v graf kandidata dodaja vozlišča iz kompozicijskega grafa. Algoritem beleži zadnja dodana vozlišča in ob dodajanju novih upošteva naslednji ključni pravili:

1. Če zadnje dodano vozlišče predstavlja parameter oz. koncept:
   - če ima vozlišče enega predhodnika, dodaj predhodnika v graf,
   - sicer razdeli graf.
2. Če zadnje dodano vozlišče predstavlja storitev,
   - dodaj predhodnike v graf.

V primeru vozlišča, ki predstavlja storitev, je v graf treba dodati vse predhodnike nazadnje dodanega vozlišča, saj predstavljajo vhodne parametre za dotično storitev. V nasprotnem primeru, ko je zadnje dodano vozlišče koncept, v graf dodajamo novo storitev. V tem primeru je nadaljna gradnja odvisna od števila predhodnikov prej dodanega parametra oz. koncepta. Če ima parameter samo enega predhodnika, vemo, da je ta parameter izhodni parameter ene storitve, ki jo tudi dodamo v graf. Če je predhodnikov več, lahko sodimo, da je ta parameter izhodni parameter več različnih storitev, torej je ta parameter mogoče pridobiti na več različnih načinov. V tem primeru bo potrebno deljenje grafa.
Deljenje grafa se zgodi v primeru, ko ima prej dodani parameter vsaj dva

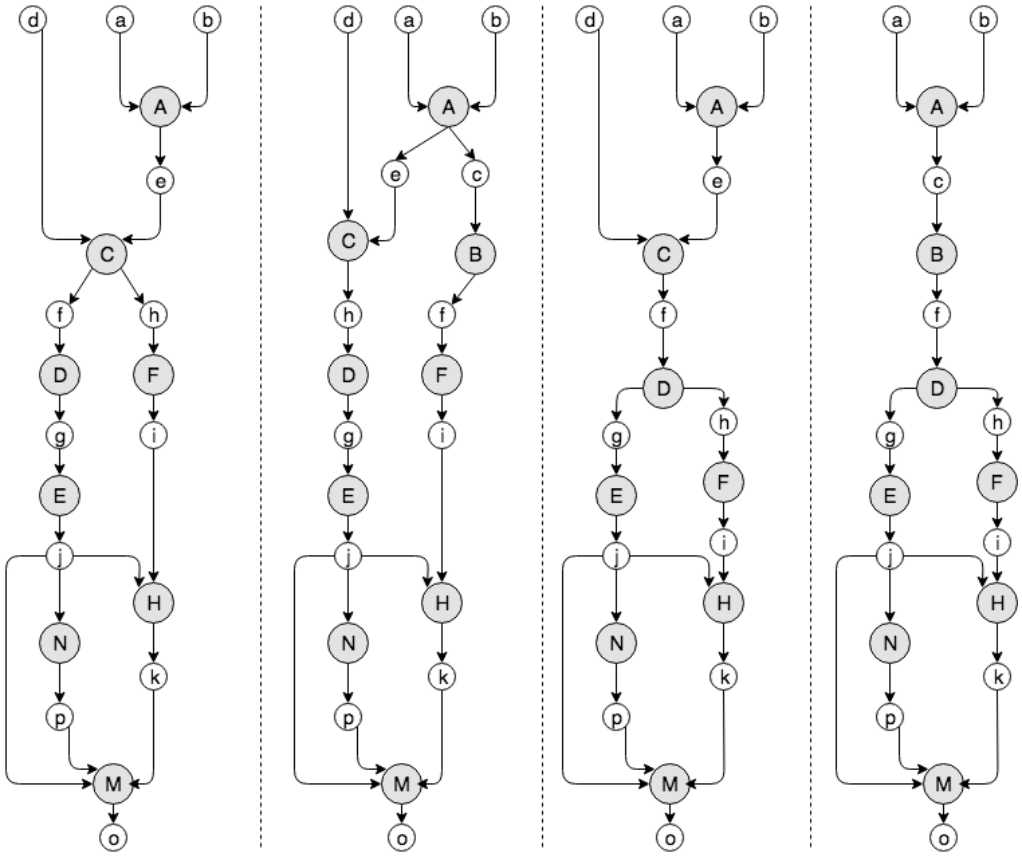
predhodnika. Iz tega sledi, da je parameter izhod vsaj dveh storitev, torej lahko na več načinov pridobimo isti parameter oz. koncept. Rezultat deljenja je več novih grafov, vsak izmed njih pa vsebuje enega izmed predhodnikov koncepta, pri katerem je bilo potrebno deljenje. Nadaljna gradnja se izvaja na vseh novih grafih, pridobljenih v dosedanjih točkah delitve, za vsakega izmed novih grafov pa veljajo ista pravila kot za prvotnega. Algoritem se izvaja vse, dokler vsi grafi ne dosežejo pomožnega vozlišča `START`.

Slika 6.5 prikazuje kompozicijski graf, z dodanimi pomožnimi vozlišči, pripravljen za iskanje kandidatov za kompozicijo.



Slika 6.5: Kompozicijski graf za poizvedbo `Q = (in=[a,b,d], out=[o])`

Vozlišči, obarvani v rdeče, sta tisti vozlišči, v katerih bo prišlo do deljenja grafa. Opazimo, da je vozlišče `f` izhod storitev `B` in `C` ter vozlišče `h` izhod storitev `C` in `D`. Obe vozlišči torej lahko dobimo na dva različna načina, kar pomeni, da bomo skupno imeli štiri kandidate, prikazane na sliki 6.6.



Slika 6.6: Kandidati, najdeni v kompozicijskem grafu iz slike 6.5

Algoritem 2 Psevdokoda dela algoritma za iskanje kandidatov

Vhod:

1: $graph \leftarrow compositionGraph$ ▷ kompozicijski graf
2: $compositionCandidate$
3: $lastNodes$

Izhod:

4: $compositionCandidate$
5: $newLastNodes \leftarrow \emptyset$
6: **for all** node in lastNodes **do**
7: **if** node is endpoint **then**
8: $inputs \leftarrow getIncomingVerticesOf(node)$
9: **for all** input in inputs **do**
10: $addNode(node, compositionCandidate)$
11: $addEdge(input, node, compositionCandidate)$
12: **if** newLastNodes doesNotContain input and compositionCandidate doesNotContain input **then**
13: $newLastNodes \leftarrow newLastNodes + input$
14: **end if**
15: **end for**
16: **else**
17: **if** shouldSplitAt(node, graph) **then**
18: $endpoints \leftarrow getIncomingVerticesOf(node)$
19: $endpoint \leftarrow endpoints.first$
20: $splits \leftarrow performSplit(compositionCandidate, graph, node, endpoints, newLastNodes, lastNodes.from(node.index, lastNodes.size))$
21: **if** newLastNodes doesNotContain endpoint and compositionCandidate doesNotContain endpoint **then**
22: $newLastNodes \leftarrow newLastNodes + endpoint$
23: **end if**
24: $addNode(endpoint, compositionCandidate)$
25: $addEdge(endpoint, node, compositionCandidate)$
26: $newLastNodes \leftarrow newLastNodes + lastNodes.from(node.index, lastNodes.size)$
27: return splits

Algoritem 2 Psevdokoda dela algoritma za iskanje kandidatov (nadaljevanje)

```
28:         else
29:             endpoints ← getIncomingVerticesOf(node)
30:             for all endpoint in endpoints do
31:                 addNode(endpoint, compositionCandidate)
32:                 addEdge(endpoint, node, compositionCandidate)
33:                 if newLastNodes doesNotContain endpoint
                            and compositionCandidate doesNotContain endpoint then
34:                     newLastNodes ← newLastNodes + endpoint
35:                 end if
36:             end for
37:         end if
38:     end if
39: end for
40: lastNodes ← newLastNodes
41: return ∅
```

Poglavje 7

Vrednotenje

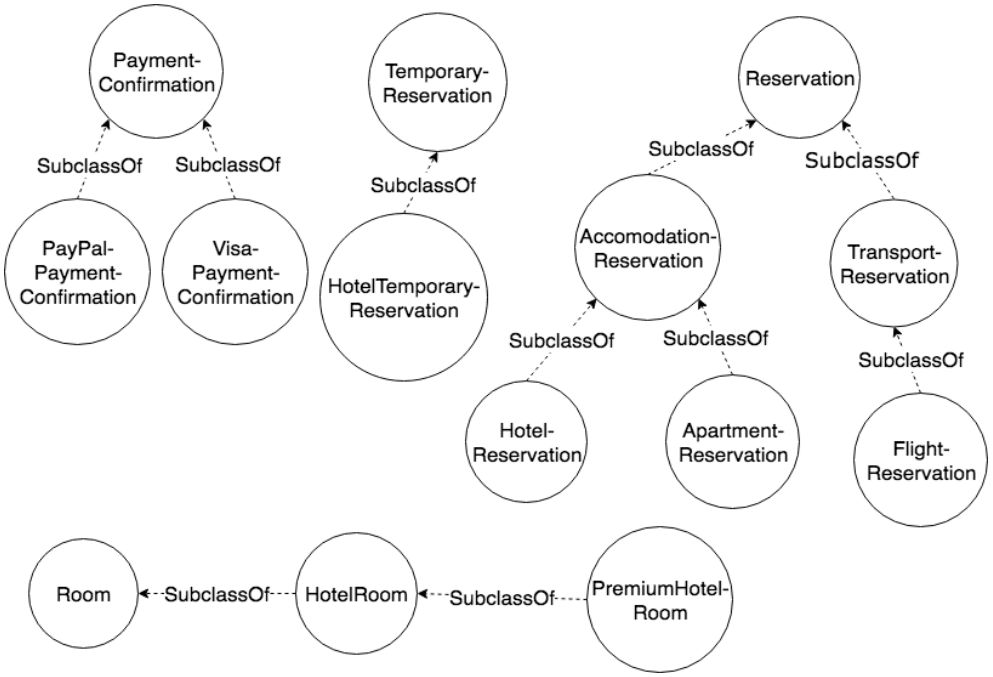
V tem poglavju opišemo študijo primera in v sklopu le-te predstavimo določene prednosti in slabosti našega pristopa. V nadaljevanju poglavja v obliki analize SWOT [13] ovrednotimo svoj pristop k razširitvi standarda Swagger in semantični kompoziciji. Analiza SWOT je ena izmed najpogosteje uporabljenih oblik analize, predvsem v poslovanju, apliciramo pa jo lahko tudi na druga področja.

7.1 Študija primera

Predpostavimo spletni repozitorij, ki vsebuje tudi naslednje storitve:

- `getHotels = (vhod=[Latitude, Longitude, Radius], izhod=[Hotel], nefunkcionalni parametri = [pricing: Free, requestLimit = (daily, 1000), security: HighSecurity])`
- `getHotelsInCity = (vhod=[City], izhod=[Hotel], nefunkcionalni parametri = [pricing: Free, security: HighSecurity])`
- `getApartments = (vhod=[Latitude, Longitude, Radius], izhod=[Apartmentl])`
- `getHotelRooms = (vhod=[HotelId], izhod=[PremiumHotelRoom])`
- `topRatedHotelRooms = (vhod=[City], izhod=[PremiumHotelRoom])`
- `roomAvailability = (vhod=[HotelRoom], izhod=[RoomAvailability])`
- `preBook = (vhod=[HotelRoom, RoomAvailability], izhod=[HotelTemporaryReservation])`
- `book = (vhod=[HotelTemporaryReservation, VisaPaymentConfirmation], izhod=[HotelReservation])`
- `pay = (vhod=[CreditCardInformation], izhod=[PaymentConfirmation])`

Slika 7.1 prikazuje del uporabljene ontologije

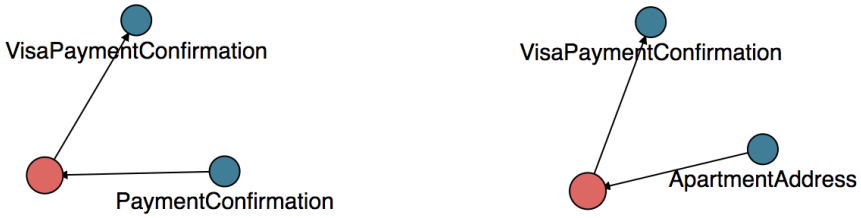


Slika 7.1: Del uporabljene ontologije.

Denimo, da razvijalec razvija funkcionalnost turistične aplikacije, ki omogoča tudi rezervacijo (bližnjega) hotela. Predpostavimo naslednji zahtevek po kompoziciji:

$$Q_1 = (vhod = [Latitude, Longitude, Radius, CreditCardInformation],\\ izhod = [HotelReservation])]$$

Iz zgornjih definicij spletnih storitev je moč ugotoviti, da kompozicija ni možna, kar dokazuje tudi vizualizacija iz spletne aplikacije na sliki 7.2, ki predstavlja dve rešitvi, ki lahko kompozicijo omogočita.



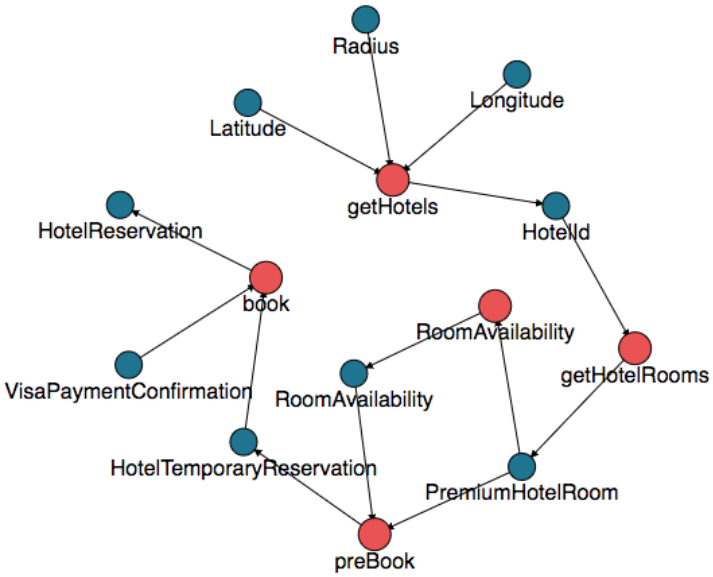
Slika 7.2: Predlagane rešitve

Iz definicij spletnih storitev in slike 7.2 je očitno, da bo kompozicija možna, če bo na voljo podatek oz. koncept `VisaPaymentConfirmation`. Preslikava iz levega dela slike 7.2 ponazarja možno rešitev. Gre za storitev, ki koncept `PaymentConfirmation` preslika v koncept `VisaPaymentConfirmation` in tako omogoči kompozicijo. Ta rešitev je z vidika našega pristopa najboljša, saj sta koncepta `PaymentConfirmation` in `VisaPaymentConfirmation` semantično sorodna. `PaymentConfirmation`, ki je izhod storitve `pay`, je splošnejši koncept od koncepta `VisaPaymentConfirmation` - torej gre za nadkoncept. Desni del slike 7.2 predstavlja drugo možno rešitev, ki je slabša, saj koncepta nista semantično povezana.

Na podlagi ugotovitev iz rezultata kompozicije, pogojenega z zahtevkom Q_1, lahko zahtevo spremenimo in definiramo novo:

$$Q_2 = (vhod = [Latitude, Longitude, Radius, VisaPaymentConfirmation], \\ izhod = [HotelReservation])]$$

Opazimo, da je kompozicija spletne storitve, definirana z zahtevo Q_2, možna. Edini možen kandidat je predstavljen na sliki 7.3.



Slika 7.3: Kandidat za kompozicijo

Opazimo lahko natančnost semantičnih značk. Koncept `VisaPaymentConfirmation` natančno opredeljuje pomen parametra. Predpostavimo, da parameter, opredeljen s konceptom `VisaPaymentConfirmation`, zamenjamo s parametrom, opredeljenim kot `PaymentConfirmation`. Algoritmi, ki se skušajo naučiti pomenov iz API-jeve domene oz. imena parametra, bi potencialno lahko sklepali, da sta ta dva koncepta sorodna, če ne celo ista in na ta način bi omogočili kompozicijo z enim in drugim. Naš pristop to loči in ugotovi, da kompozicija s konceptom `PaymentConfirmation` ni možna. Tu se kaže velika natančnost oz. nedvoumnost semantičnih značk, a za ceno splošnosti algoritma.

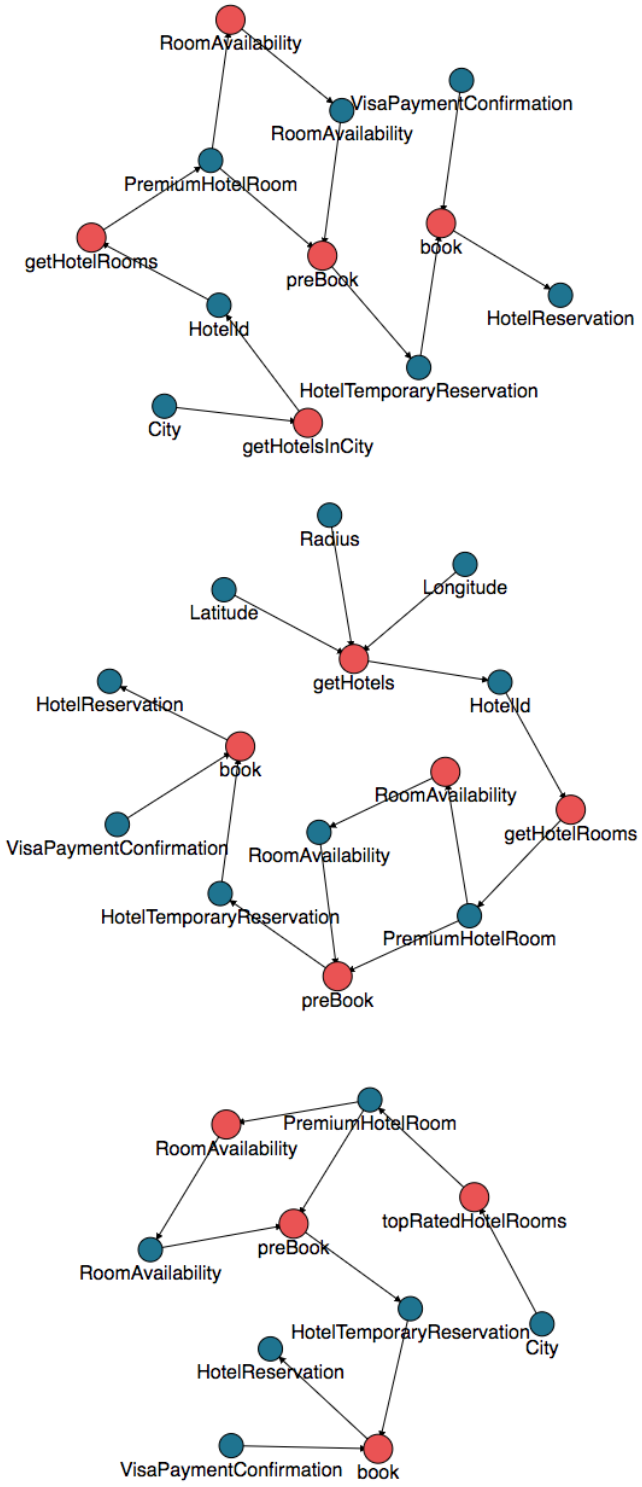
Natančnost semantičnih značk posledično omogoča tudi natančno preverjanje semantične skladnosti. Iz definicij spletnih storitev opazimo, da je izhod storitev `getHotelRooms` in `topRatedHotelRooms` parameter, opredeljen s konceptom `PremiumHotelRoom`, ki je uporabljen kot vhod storitev `preBook` in `roomAvailability`. Storitvi se ujemata, saj gre za vključujoče semantično ujemanje - koncept `PremiumHotelRoom` je konkretnejši (podkoncept) od koncepta `HotelRoom`. Manjša natančnost oz. dvoumnost semantičnih značk bi

onemogočala zanesljivo preverjanje semantičnega ujemanja.

Denimo, da zahtevek še posplošimo in ga definiramo kot:

$$Q_3 = (vhod = [Latitude, Longitude, Radius, City, \\ VisaPaymentConfirmation], izhod = [HotelReservation])]$$

Vhodnim parametrom zahtevka Q_3 je dodan parameter, katerega pomen definira koncept `City`. Zahtevek lahko opišemo kot zahtevek po kompoziciji spletne storitve, ki bo rezervirala hotelsko sobo v bližini oz. podanem mestu. Iz definicij spletnih storitev lahko opazimo, da je kandidatov za tako kompozicijo več (slika 7.4).



Slika 7.4: Kandidati za kompozicijo

Ker je kandidatov več, uporabimo nefunkcionalne parametre storitev v kompoziciji za razvrščanje oz. ocenjevanje primernosti kandidatov. Ti so na sliki 7.4 glede na primernost razvrščeni po vrsti od zgoraj navzdol. Vrednosti posameznih nefunkcionalnih parametrov so interni parametri delovanja aplikacije.
Najboljši kandidat v tem primeru zbere dve točki, saj sta nefukncionalna parametra storitve `getHotelsInCity` skupaj vredna dve točki, druge storitve pa nimajo definiranih nefunkcionalnih parametrov. Podobno velja za drugega kandidata. Vrednosti nefunkcionalnih parametrov `pricing` in `security` prinašata dve točki, vendar je zaradi nefunkcionalnega parametra `request limit` vreden le eno točko. Storitve v zadnjem kandidatu nimajo definiranih nefunkcionalnih parametrov, zato kandidat ne zbere nobene točke in je v tem primeru uvrščen kot zadnji.

Naša aplikacija uporablja nefuncionalne parametre za ocenjevanje in razvrščanje kandidatov za kompozicijo, vendar to ni edini možni način uporabe. V primeru integracije z repozitoriji spletnih storitev bi lahko omogočili tudi iskanje oz. filtriranje po nefunkcionalnih parametrih.

7.2 Analiza SWOT

V prejšnjem poglavju smo poudarili nekatere prednosti in slabosti svojega pristopa. V nadaljevanju nadaljujemo vrednotenje pristopa v obliki analize SWOT. Ta poda prednosti in slabosti našega pristopa, ki so notranji dejavniki, na katere imamo vpliv. Priložnosti in nevarnosti pa so zunanji dejavniki, na katere ni mogoče vplivati. Ti vidiki so zbrani v tabeli 7.1 in natančneje opisani v nadaljevanju.

Tabela 7.1: Analiza SWOT

Prednosti	**Slabosti**
• Natančnost semantičnih značk • Šibka sklopljenost • Podpora za nefunkcionalne parametre • Uporaba širše podprtih tehnologij • Preprosta nadgradnja	• Gradnja kompozicijskega grafa v času izvajanja • Centraliziranost • Brez podpore YAML • Odvisnost od definirane ontologije
Priložnosti	**Nevarnosti**
• Podpora ekonomiji API • Cenejša nadgradnja • Repozitoriji spletnih storitev • Schema.org	• Slabša sprejetost semantičnih tehnologij

7.2.1 Prednosti

Natančnost semantičnih značk

Natančnost oz. nedvoumnost semantičnih značk je prav gotovo ena izmed bistvenih prednosti našega pristopa. V spletu je unikatnost ključna lastnost URL-jev. Vsak dokument v spletu je enolično in nedvoumno določen z URL-jem. Naše semantične značke so v osnovi določene s koncepti, ki so v ontologiji predstavljeni z URL-ji in tako posledično dedujejo unikatnost URL-ja. Pomen natančnosti semantičnih značk se izkaže predvsem pri iskanju storitev in kompoziciji. Unikatnost URL-jev poskrbi za precej natančno presojanje o konceptih in tako zelo zmanjša možnost lažnih pozitivnih ujemanj. Posledično je iskanje storitev precej natančno, rezultati kompozicije pa v veliki meri pravilni.

Šibka sklopljenost

Podporna storitev za kompozicijo je zasnovana modularno. Pomembni medseboj neodvisni deli so modul za gradnjo kompozicijskega grafa, modul za iskanje kandidatov za kompozicijo, modul za iskanje predlogov rešitev itd. Modularna zasnova omogoča relativno enostavno nadgradnjo in razvoj posameznih modulov. Storitev je tako primerna za nadaljni razvoj.

Uporaba širše podprtih tehnologij

Razširitev standarda Swagger in tudi podporna storitev za kompozicijo uporabljajo široko podprte tehnologije. JSON-LD je standard W3C, ki ga uporabljata *Schema.org* in *Google Knowledge Graph* ter je v splošnem precej uporabljan na področju optimizacije spletnih iskalnikov. Pod okriljem organizacije W3C se razvija tudi standard SPARQL za poizvedovanje po grafih RDF.
Dejstvo, da se dva ključna standarda, na katerih temelji naše delo, razvijata pod okriljem organizacije W3C, je prav gotovo velika prednost. Oba standarda sta tudi širše sprejeta in uporabljana.

Preprosta nadgradnja

Razširitev standarda Swagger temelji na uporabi sintakse JSON-LD. Dokument JSON-LD je še vedno popolnoma veljaven in skladen z običajnim dokumentom JSON. Posledično nadgradnja dokumenta JSON v JSON-LD ne zahteva dodatnih orodij in organizacijskih prilagajanj oz. njihovih načrtovanj. Podobno velja tudi glede učne krivulje sintakse JSON-LD. Za osnovno uporabo te sintakse je potrebno relativno malo učenja in le poznavanje koncepta sintakse JSON-LD ter ključnih besed `@context` in `@type`. Ta dva konstrukta sta tudi temelj naše razširitve.
Danes je precej pomemben vidik izpostavljanja podatkov. Veliko podjetij omogoča uporabo podatkov, podprto s spletnimi storitvami, kar običajno prinaša tudi konkurenčno prednost na trgu. Menimo, da semantična nadgradnja teh storitev lahko prispeva še več dodane vrednosti, zaradi česar je preprosta nadgradnja toliko bolj pomembna.

Podpora za nefunkcionalne parametre

Standard smo razširili tudi z možnostjo definiranja nefunkcionalnih parametrov. Ti naslavljajo druge, nefunkcionalne vidike spletnih storitev, kot so varnost, SLA, razpoložljivost, cena itd. Sami jih sicer uporabljamo za razvrščanje kandidatov za kompozicijo, vendar menimo, da je njihova uporabnost širša.

7.2.2 Slabosti

Gradnja kompozicijskega grafa v času izvajanja

Naša implementacija podporne storitve kompozicijski graf gradi v času izvajanja, za vsako zahtevo posebej. V primeru relativno majhnega števila storitev v repozitoriju je ta slabost sicer precej zanemarljiva, česar pa ne moremo trditi v primeru velikih repozitorijev. Ta del kompozicije lahko zahteva precej časa za izvedbo, predvsem pa računalniških virov (pomnilnik, procesorska moč ...).

To slabost bi bilo vredno poskusiti izboljšati s prilagoditvijo algoritma za gradnjo kompozicijskega grafa. V osnovi bi ta algoritem lahko shranjeval vmesne rezultate oz. dele grafa in jih v primernih trenutkih (npr. ko gre za enako ali zelo podobno zahtevo) ponovno uporabljal. Podobno težavo v delu [1] rešujejo s shranjevanjem vseh najkrajših poti med vozlišči v matriko in te, prej izračunane informacije, v času izvajanja uporabijo.

Centraliziranost

Centraliziranost je ena izmed večjih pomanjkljivosti. Trenutna implementacija predvideva uporabo ene storitve SPARQL, katere namen je dostop do ontologije in presojanja o različnih konceptih.
Težave oz. nezmožnost presojanja o konceptih in posledično semantične kompozicije lahko povzroči izpad strežnika SPARQL. Izpad lahko povzroči tudi sprememba naslova strežnika, zato bi bilo treba raziskati možne izboljšave oz. načine prilagajanja aplikacije v takih primerih.

Odvisnost od definirane ontologije

Sorodna omejitev centraliziranosti je tudi odvisnost od definirane ontologije. Podporna storitev za kompozicijo in razširitev standarda v trenutni različici predpostavljata uporabo natanko ene ontologije. To lahko omejuje zmožnost opisovanja in izražanja konceptov v semantičnih značkah. Zmožnost izražanja različnih konceptov igra ključno vlogo pri semantični kompoziciji, saj so le bogate ontologije tako privlačne kot tudi uporabne.

Odpravljanje oz. prilagajanje te omejitve bi bilo vredno raziskati v smeri podpore definiciji oz. opisu strežnika SPARQL ali lokacije ontologije. V razširitvi standarda bi lahko podprli definiranje lokacije ontologije oz. storitve SPARQL. Te informacije bi v osnovi izkoriščala (podporna) storitev za kompozicijo - z njimi bi lahko razvijalec vplival na vir informacij za presojanje o konceptih, uporabljenih v kompoziciji. V osnovi bi bila taka podpora mogoča z dodajanjem novih informacij v obliki razširitve, katerih komple-

ksnost je odvisna od prepoznanih potreb. Definicijo take razširitve bi bilo mogoče definirati tudi na strežniku. Kot razširitev v standardu bi bil zadosten že spletni naslov, prek katerega bi storitev za kompozicijo z določeno akcijo HTTP (npr. HEAD) pridobila dodatne informacije (npr. URL strežnika SPARQL).

Brez podpore YAML

Dokumenti Swagger so praviloma definirani v formatu JSON ali jeziku YAML. Naša razširitev predpostavlja uporabo formata JSON in ni skladna oz. mogoča v dokumentih, napisanih v jeziku YAML. V okviru nadaljnega dela bi bilo treba razviti razširitev in razčlenjevalnik za dokumente Swagger, napisane v jeziku YAML. S tem bi zagotovili popolno semantično razširitev standarda in možnost omenjenih prednosti tudi za opise storitev v jeziku YAML.

7.2.3 Priložnosti

Podpora ekonomiji API

Danes spletne storitve niso več samo programski konstrukti in implementacijske rešitve v računalniških sistemih, temveč igrajo tudi pomembno vlogo v organizacijskem, podjetnem smislu, kar je privedlo do pojava ekonomije API. Ekonomija API je komercialna izmenjava poslovnih funkcij, zmožnosti in kompetenc v obliki spletnih storitev oz. API-jev [19]. Ta lahko precej pozitivno vpliva na poslovanje, rast in inovativnost podjetja oz. organizacije. Menimo, da so semantične razširitve že v osnovi dobrodošel dodatek k spletnim storitvam, ki pa lahko dodano vrednost prek ekonomije API prenesejo tudi na druge vidike v podjetju.

Cenejša nadgradnja

Med prednosti smo uvrstili tudi preprosto nadgradnjo. Ta je mogoča zaradi uporabe relativno poznanih tehnologij, skladnih in široko uporabljenih tehnologij. Posledica preproste nadgradnje pa je tudi cenejša nadgradnja.

Ta je precej pomembna z vidika podjetij, ki bi potencialno lahko uporabila razširitev standarda, saj lahko pozitivno vpliva na sprejemanje odločitev o vpeljavi razširitve. K temu gotovo pripomore dejstvo, da je razširitev mogoče realizirati brez dodatnih orodij in precej verjetno z uporabo že obstoječega znanja v organizaciji. Cenejša nadgradnja pozitivno vpliva tudi na vidike podpore ekonomiji API.

Repozitoriji spletnih storitev

Repozitoriji spletnih storitev so pomemben vidik, ki je bil tudi eden izmed glavnih vodil pri našem delu. To so platforme, ki v osnovi omogočajo iskanje relevantnih spletnih storitev. Običajno je iskanje v repozitorijih podprto z iskanjem glede na ključne besede, kategorije itd., ki pa je lahko precej nerodno oz. neoptimalno. Semantična razširitev lahko izboljša samo iskanje in tudi izkušnjo pri uporabi spletnih repozitorijev, saj je iskanje praviloma natančnejše in podprto z uporabo relativno dobro definiranih konceptov v ontologijah. Semantično razširitev oz. podporno storitev za semantično kompozicijo bi bilo mogoče uporabiti tudi kot dodatno plast v implementacijah takšnih repozitorijev in tako semantično kompozicijo izpostaviti kot dodatno funkcionalnost na takih platformah.

Schema.org

Schema.org je iniciativa, ki se ukvarja z ustvarjanjem, vdrževanjem in promocijo strukturiranih shem za označevanje podatkov. Gre za skupen trud organizacij Google, Bing, Yandex in Yahoo!. Schema.org definira besednjake, ki definirajo različne entitete, njihove lastnosti in razmerja med njimi, ki jih lahko prosto uporabljamo pri označevanju različnih strukturiranih podatkov in je kompatibilna s formatoma RDF in JSON-LD. Besednjak se hitro razvija, ima relativno veliko skupnost in ga podpirajo večja podjetja in organizacije, zato menimo, da bo lahko imel precej veliko in pozitivno vlogo pri semantičnem opisovanju strukturiranih podatkov.

7.2.4 Nevarnosti

Slabša sprejetost semantičnih tehnologij

V delu [8] avtorja navajata pojav semafobije (angl. semaphobia) kot strah pred uporabo semantičnih tehnologij. Kot razloge za to navajata nerazumevanje področja semantičnih tehnologij, prepričanje, predvsem na ravni podjetij, da je za uvedbo semantičnih tehnologij potrebno ogromno dela, in splošno nenaklonjenost semantičnim tehnologijam. Taka nenaklonjenost ima lahko negativen vpliv na razširitev in njen nadaljni razvoj.

Poglavje 8

Sklepne ugotovitve

Cilja našega dela sta bila razširitev standarda Swagger s semantičnimi opisi spletnih storitev in razvoj podporne storitve za vizualizacijo semantične kompozicije. Na začetku smo raziskali področje semantične kompozicije in skušali na to področje pristopiti z novim pristopom. Naš cilj je bil razvoj razširitve za že obstoječe tehnologije, ki so danes v uporabi. Standard Swagger je de-facto standard za opis spletnih storitev, široko uporabljan, predvsem pa jasen in razumljiv, zato smo želeli svojo rešitev temeljiti na tem standardu.

Opise storitev v tem standardu je moč opisati v formatu JSON ali jeziku YAML. Pri svojem delu smo se odločili za razširitev oblike JSON, možno pa bi bilo tako razširitev razviti tudi v jeziku YAML. Na začetku smo iskali najprimernejšo obliko razširitve in se tu odločili za uporabo formata JSON-LD. Ta format je danes že zelo zrel, precej poznan na področju semantičnih tehnologij in primeren za realizacijo naše razširitve. Na začetku smo hoteli standard precej bolj razširiti, s potrebnimi informacijami na različnih mestih v dokumentu. Tu smo naleteli na prvo večjo oviro, saj standard ne omogoča poljubnih razširitev, temveč predvideva oz. dovoljuje razširitve na točno določenih mestih in s poimenovanjem, kot ga predpisuje standard. Zaradi tega ni bilo mogoče razširiti formata z uporabo JSON-LD in hkrati ohranjati skladnosti s standardom Swagger. Tu smo sprejeli verjetno eno izmed najpomembnejših odločitev v delu. Odločili smo se za razvoj razširitve standarda,

ki bo popolnoma skladna s standardom. V nasprotnem primeru bi bil končni rezultat dela nekakšen predlog standarda, ki temelji na standardu Swagger. Odločili smo se za razširitev v obliki semantičnih značk, kjer standard to dovoljuje. Semantične značke so še vedno veljavni objekti JSON-LD, zato je te v prihodnje moč nadalje razširjati in dodati morebitne nove semantične oz. kontekstne informacije. Za enkrat taka oblika semantičnih značk zadošča zadanim ciljem. Drugi cilj je bil razvoj podporne storitve za semantično kompozicijo. Razvili smo spletno aplikacijo in spletni storitvi, ki združujeta potrebne algoritme za iskanje kandidatov za semantično kompozicijo. Razvili smo algoritem za gradnjo kompozicijskega grafa, algoritem za iskanje kandidatov v kompozicijskem grafu in algoritem za predlaganje rešitev (storitev), ki bodo omogočile semantično kompozicijo. Ti algoritmi so združeni v celoto v obliki spletne storitve, ki ji kot vhod pošljemo vhodne in izhodne koncepte želene sestavljene storitve. Rezultat storitve so možni kompozicijski kandidati oz. predlogi za rešitve neuspešne kompozicije. Druga storitev omogoča nalaganje in razčlenjevanje dokumentov, opisanih v razširjenem standardu Swagger.

Menimo, da je naše delo dobra osnova za nadaljni razvoj razširitve oz. nasploh razvoja na področju semantičnih spletnih storitev.

V nadaljnjem delu bi se bilo smiselno osredotočiti na odpravo pomanjkljivosti, najdenih v analizi SWOT. Ključno bi bilo odpraviti problem centraliziranosti in odvisnosti od ene ontologije. Verjamemo, da lahko na tem področju pomembno vlogo igra besednjak Schema.org, ki se dejavno razvija pod okriljem največjih spletnih organizacij. V času pisanja tega dela JSON Schema še nima podpore za semantično označevanje, vendar obstajajo ideje o tem. Menimo, da ima lahko realizacija te ideje zelo velike pozitivne učinke na razvoj semantičnega označevanja dokumentov JSON ter konkretneje tudi definicij Swagger in oblike realizirane razširitve.

Z razvojem podporne storitve je bil naš namen razviti aplikacijo in algoritme, ki bodo uporabljali semantično razširitev. V prihodnosti bi bilo dobro razviti in integrirati vmesno programsko opremo (angl. middleware) za re-

pozitorij spletnih storitev in tako na ravni repozitorija preveriti in omogočiti semantično kompozicijo ponujenih spletnih storitev. Poseben poudarek bi bilo treba nameniti optimizaciji algoritmov, ki delujejo na grafih (iskanje rešitev v primeru neuspešne kompozicije in iskanja kandidatov) in gradnji kompozicijskega grafa, ki se trenutno dogaja v času izvajanja posameznega zahtevka.

Literatura

[1] H. Elmaghraoui, I. Zaoui, D. Chiadmi, L. Benhlima "Graph based E-Government web service composition", CoRR, zv. abs/1111.6401, 2011

[2] W3C, "Unified Lightweight Semantic Descriptions of Web APIs and Web Services". Dostopno na: `https://www.w3.org/2011/10/integration-workshop/p/OpenUniversity.pdf` (pridobljeno 20.8.2018)

[3] W3C, "SAWSDL". Dostopno na `https://www.w3.org/TR/sawsdl/` (pridobljeno 20.8.2018)

[4] J. de Brujin, C. Bussler, J. Domingue, D. Fensel, M. Hepp, U. Keller, M. Kiffer, B. König-Ries, J. Kopecky, R. Lara, H. Lausen, E. Oren, A. Polleres, D. Roman, J. Scicluna, M. Stollberg, "WSMO". Dostopno na https://www.w3.org/Submission/WSMO/ (pridobljeno 20.8.2018)

[5] M. Bennara, M. Michael, Y. Amghar, "Composing RESTful Linked Services on the Web", Proceedings of the 23rd International Conference on World Wide Web, WWW '14 Companion, Seul, Republika Koreja, str. 977-982, 2014

[6] J.L. Cánovas Izquierdo, J. Cabot, "Composing JSON-Based Web APIs", Web Engineering: 14th International Conference, ICWE 2014, Toulouse, Francija, str. 390-399, 2014

[7] V. Surwase, "REST API Modeling Languages - A Developer's Perspective", IJSTE - International Journal of Science Technology and Engineering, št. 10, zv. 2, str 634-637, 2016

[8] M. Lanthaler, C. Gütl, "On Using JSON-LD to Create Evolvable RESTful Services", Proceedings of the Third International Workshop on RESTful Design, WS-REST '12, str. 25-32, 2012

[9] J. Broekstra, A. Kampman, F. van Harmelen, "Sesame: A Generic Architecture for Storing and Querying RDF and RDF Schema", The Semantic Web - ISWC 2002, str. 54-68, 2002

[10] J. Huang, D.J. Abadi, K. Ren, "Scalable SPARQL Querying of Large RDF Graphs", PVLDB, zv. 4, str. 1123-1134, 2011

[11] M. Paolucci, T. Kawamura, T.R. Payne, K. Sycara, "Semantic Matching of Web Services Capabilities", The Semantic Web - ISWC 2002, str. 333-347, 2002

[12] Y-J. Lee, "Semantic-Based Web API Composition for Data Mashups", J. Inf. Sci. Eng., zv. 31, str. 1233-1248, 2015

[13] N. Fallon, "SWOT Analysis: What It Is and When to Use It". Dostopno na: `https://www.businessnewsdaily.com/4245-swot-analysis.html` (pridobljeno 27.8.2018)

[14] JSON-LD. Dostopno na: `https://json-ld.org/` (pridobljeno 20.8.2018)

[15] B. Marr, "How Much Data Do We Create Every Day? The Mind-Blowing Stats Everyone Should Read. Dostopno na: `https://www.forbes.com/sites/bernardmarr/2018/05/21/how-much-data-do-we-create-every-day-the-mind-blowing-stats-everyone-should-read/#2fbd432360ba` (pridobljeno 20.8.2018)

[16] SmartBear Software, “Swagger”. Dostopno na: `https://swagger.io/` (pridobljeno 20.8.2018)

[17] E. You, ”Vue.js”. Dostopno na: `https://vuejs.org/` (pridobljeno 20.8.2018)

[18] M. Bostock, ”D3.js”. Dostopno na: `https://d3js.org/` (pridobljeno 20.8.2018)

[19] (2014) Stepping Forward into the API Economy. Dostopno na: `http://www.redbooks.ibm.com/redpapers/pdfs/redp5164.pdf` (pridobljeno 20.8.2018)

[20] Heroku. Dostopno na: `https://www.heroku.com/` (pridobljeno 20.8.2018)

[21] Ubuntu. Dostopno na: `https://www.ubuntu.com/` (pridobljeno 20.8.2018)

[22] Semantic Web Services. Dostopno na: `https://en.wikipedia.org/wiki/Semantic_web_service` (pridobljeno 27.8.2018)

[23] Microsoft Azure. Dostopno na: `https://azure.microsoft.com/en-gb/` (pridobljeno 20.8.2018)