POSER: A Semantic Payload Lowering Service

Daniel Spieldenner
Agents and Simulated Reality,

Keywords: Semantic Web, Interoperability, Mapping, Ontology, JSON.

Abstract: Establishing flexible, data source and structure independent interoperability between databases, web services or devices is an ubiquitous problem in today's digitized, connected world. The concept of semantic interoperability is a promising approach to abstract communication from concrete protocols and data structures to a more meaning driven data representation. While transforming structured data into semantic knowledge graphs is a well investigated problem, actually using semantic data in an ecosystem of established legacy services, often providing only a standard structured data API, is still an open issue. In this paper we propose an approach on how to formally describe possible mappings between a higher level semantic data representation onto syntactically fixed structured data objects, introduce an algorithm describing how to generate structured data objects from semantic input using mapping rules following these concepts, and illustrate the approach with an example implementation for a possible interoperability layer service, connecting a semantic input data set to a JSON API.

1 INTRODUCTION

Since its early days, the *Semantic Web* (Berners-Lee et al., 2001) has developed to a vast field of not only research but also applications. Developers in domains that demand a high level of interoperability, such as Smart Cities or Smart Homes, have understood the beauty of semantically describing and exchanging data.

Instead of having fixed data structures that do not convey any meaning of the data exchanged, formats like *RDF* (Group, 2014) allow for describing data in a not only machine readable, but also human understandable manner. Being able to not only describe existing structured data in a semantic way, but also using the vast amount of knowledge contained in knowledge graphs to be used in existing applications would allow for a higher level of interoperability in terms of service composition and data exchange (Sadeghineko and Kumar, 2021), (de Mello et al., 2022), (Rejeb et al., 2021), (Bröring et al., 2018) However, established applications and interfaces are rarely tailored to a complex format such as RDF. Using semantic data to feed regular APIs that rely on structured data formats such as JSON is not possible out of the box.

While the transformation of structured data into semantic data (*lifting* the data to a semantic level) or the access to structured data sources in a semantic fashion is widely investigated ((Dimou et al., 2014), (Gupta et al., 2012), (Spieldenner, 2020), (Michel et al., 2014)), the inverse direction of how to make semantic data accessible for established structured data legacy APIs is still mostly an open question.

In this paper, we present an approach to *lower* data from a semantic level to structured data objects by describing a target interface and its structure we want to address in a semantic manner. To this end we first introduce a formal definition of the mapping process between a semantic knowledge graph and a structured data object. Based on that we derive an algorithm how to actually generate a nested structured data object from a semantic input source and finally provide an example implementation illustrating the concepts introduced before.

After giving a brief overview of existing approaches and their potential shortcomings in section 2 we give a formal definition of the problem we want to solve: finding mappings between a semantic knowledge graph and a given non-semantic data structure (section 3). In section 4 we propose a slim ontology to semantically describe structured data, taking JSON as an example, in such a way that we include the concepts of section 3. Section 5 introduces an algorithm describing a possibility to apply the aforementioned mapping rules, enriched with the knowledge expressed via the ontology presented before, to generate nested

Spieldenner, D.
POSER: A Semantic Payload Lowering Service.
DOI: 10.5220/0011510000003318
In *Proceedings of the 18th International Conference on Web Information Systems and Technologies (WEBIST 2022)*, pages 249-256
ISBN: 978-989-758-613-2; ISSN: 2184-3252
Copyright © 2022 by SCITEPRESS – Science and Technology Publications, Lda. All rights reserved

structured data objects. *POSER*, a prototype implementation of a web service taking a knowledge graph as input and producing a JSON text as output, following the concepts introduced before, is illustrated in section 6. We conclude the paper and discuss open questions in section 7.

2 RELATED WORK

Interoperability is a thoroughly investigated topic, not only in the IoT domain (Lee et al., 2021), (Amjad et al., 2021) (Bröring et al., 2017) but also in larger scale smart environments like smart cities, smart homes or smart grids. (Brutti et al., 2019) (Ahlgren et al., 2016) (Ma et al., 2013) (Uslar and Engel, 2015) (Perumal et al., 2010)

The motivation behind interoperability solutions is mostly the same: connecting systems in a data source, manufacturer or provider independent fashion and allowing communication between data "silos", enclosed systems with vertical communication channels that are difficult to connect to other, similarly enclosed, data sources. Examples for such silos are certain sets of sensors in smart homes of a specific vendor that are incompatible to interfaces of other devices, or data sets of different domains in a smart city environment. (Gardner, 2005) (Stonebraker et al., 2018)

Semantic interoperability has become a promising approach to overcome this restriction to silo like architectures (Heiler, 1995) (Ganzha et al., 2017). Instead of communicating with purely structured, "meaningless" data, data is semantically annotated to allow data exchange on the basis of the actual meaning of the data instead of just a string of characters.

However, as existing legacy APIs are not tailored to this kind of communication, we need to provide ways to not only enrich data with semantic meaning, but also need methods to exploit this semantic knowledge when addressing legacy interfaces expecting plain structured data.

While approaches to generate RDF data from relational databases or structured data sources like RML (Dimou et al., 2014) exist and are widely used, the inverse way how to use this semantic information when addressing actual interfaces is less thoroughly investigated.

The *Web of Things*[1] (Zeng et al., 2011) initiative provides a W3C standard for describing IoT interfaces in an interoperable fashion and adding semantic information via annotations, however the possibilities to actually map complex payloads to achieve full interoperability between services with incompatible structured data APIs is limited.

With *JSON-LD* (Sporny et al., 2014), a JSON format incorporating semantic information is already available. However, it is rather meant as a serialization format for semantic data, not as a detailed concept to describe an actual JSON structure. Just serializing RDF to JSON-LD will not yield the results necessary to cater for a given JSON-API. RDF to JSON approaches like (Alexander, 2008) mostly focus on serializing RDF to JSON, less on actually being able to lower the semantic data back onto API level.

With *OBA* (Garijo and Osorio, 2020) Osorio and Garijo present a framework to generate API descriptions from an ontology with the possibility to serve the client JSON text according to an ontology defined structure. While they bridge the gap between the semantic world and plain JSON, the approach is meant to access an ontology, with the API structure given by that ontology, rather than allowing semantic data and the knowledge connected to it to be fed to an existing legacy JSON API. Moreover, OBA does not provide any specific description of JSON text itself, the output is generated based on the OpenAPI description derived from the input ontology.

The task of actually creating documents with a specific structure was investigated by Allocca and Gougousis (Allocca and Gougousis, 2015), analyzing the RDF generated by a given RML mapping and deriving inverse mapping rules for the given RML mappings, resulting in a structured data document of the form initially taken as input for the RML mapping step. The approach only works on a subset of possible RML mapping rules, and only generating CSV data was investigated.

When it comes to describing actual JSON objects, JSON schema [2] is a widely used well established standard, used, for example, by Barbaglia et al. for describing REST services (Barbaglia et al., 2017). Reading a JSON schema definition of a complex object can get quite cumbersome for a human reader and provides only little semantic information about the JSON object itself and how it links to other semantic data.

3 FORMAL DEFINITION

In order to derive an algorithm to generate structured data from RDF, in our case JSON, we aim to describe the *JSON text* we want to generate in terms of RDF triples. A JSON text in this context represents a series of valid JSON *tokens* as defined in section 2 of the

[1] https://www.w3.org/WoT/

[2] https://json-schema.org/

JSON specification (Sporny et al., 2014).

With such a semantic description of the desired JSON text at hand, lowering the actual data from the knowledge graph to the JSON text then boils down to finding proper mappings between the graph describing our JSON text and the knowledge graph containing the RDF data.

Let Γ be a set of graphs representing semantic data. Let S be a set of structured data objects that can be serialized into a proper JSON text. Our goal is to find a mapping λ for a graph $\gamma \in \Gamma$ to create an object s so that $s \in S$ and

$$\exists \lambda : \lambda(\gamma) \mapsto s \tag{1}$$

In our case, S consists of all possible syntactically correct JSON texts. For JSON, we mainly need to differentiate between two possible types of data created: *primitives* $P \subset S$ and *compound data types* $C \subset S$. In addition, we can consider the special *null* value $null \in S$.

Primitives consist of the datatypes *string*, *number* and *boolean*. The actual values of the primitive data types will be determined by evaluating statements $\sigma \in \Gamma$, mapping knowledge from a knowledge graph $\gamma \in \Gamma$ to an object $s \in S$. We can define a mapping in a recursive manner as explained in the following.

3.1 Primitives

Let $\sigma \in \Gamma$ be a statement consisting of subject, predicate, object $\sigma_s, \sigma_p, \sigma_o$. Let

$$\lambda(\sigma) = I(\sigma_o) = p \in P \tag{2}$$

where $I(\sigma_o)$ is the *interpretation* of σ_o. The interpretation I depends on the json data type of the primitive p:

- If p is a string, $p = \sigma_o$, i.e. the actual string value in the object position of statement σ. If σ_o consists of an URI pointing to another resource, or a number, parse the value into a string nevertheless.
- If p is a number, σ_o is parsed into a number format, if possible, and *null* otherwise.
- If p is boolean, $p = true$ if $\sigma_o = true \vee \sigma_o = 1$ and $p = false$ if $\sigma_o = false \vee \sigma_o = 0$. If σ_0 is not interpretable in a boolean fashion, then set $p = null$

The outcome of the interpretation $I(\sigma_o)$ then is in any case either a *string*, a *number* or the *null* value, and with that, we have created a valid JSON text according to (Bray, 2014) .

3.2 Objects

Let $\gamma \in \Gamma$ be a graph representing semantic data. Our goal is to map this graph in such a way that we receive a JSON object $o \in S$, defined by a key/value pair (k, v), i.e. $\exists \lambda(\gamma) : \lambda(\gamma) \mapsto o$. By definition, $v \in S$, and $k \in P$, as keys are strings and as such can be interpreted as primitive objects. In order to map γ to an object, we thus must map γ to (k, v), so we are looking for mappings λ_1, λ_2 with $o = (\lambda_1(\gamma), \lambda_2(\gamma))$.

Finding a mapping $\lambda_1(\gamma))$ is trivial as $\lambda_1(\gamma)) = k$ by definition is a primitive and the mapping rules for primitives as given by equation 2 apply.

For $\lambda_2(\gamma) = v$ we either can use the mapping rules for primitives again, resulting in an object `{o, v}` with $o, v \in P$. Or we take $v = o' \in C$ with its own specific mapping

$$\gamma' \in \Gamma' \subset \Gamma, \lambda' : \Gamma' \subset \Gamma \to S, \lambda'(\gamma') = o' \tag{3}$$

and proceed recursively.

We have shown that we can create arbitrarily deeply nested objects from a given graph. Now for the inverse direction, we show that we can define a mapping λ for a given, fixed object $o \in C$.

Let $o = (k, v), o \in C$ be an object consisting of a key/value pair (k, v). We need to specify a mapping λ so that $o = \lambda(\gamma), \gamma \in \Gamma$ with γ being a given graph representing semantic data. If $v \in P$, we need to specify a statement $\sigma = (\sigma_s, \sigma_p, \sigma_o) \in \Gamma$ with $\lambda(\sigma) = p$ as given in equation 2 and we are done.

If we aim for $v \in C$, so v being a compound object, we select a subset $\Gamma' \subset \Gamma$ and set

$$v = \lambda'(\gamma'), \gamma' \in \Gamma'. \tag{4}$$

In this fashion, we can recursively define nested objects until finally we arrive at a literal value, optionally narrowing down the search space in the semantic representation when proceeding to lower hierarchy levels.

3.3 Arrays

Let $a \in C$ be a data structure representing a collection of objects $o_1, o_2, ..., o_n \in S$. A mapping $\lambda : \Gamma \to S, \gamma \in \Gamma, a \in S, \lambda(\gamma) = a$ can be found by giving specific mappings for every element of a:

$$\begin{aligned} \lambda(\gamma) &= (\lambda_1(\gamma_1), \lambda_2(\gamma_2), ..., \lambda_n(\gamma_n)) \\ &= (o_1, o_2, ..., o_n) \\ &= a \end{aligned} \tag{5}$$

In the inverse direction, if we have a specific array $a = (o_1, o_2, ...o_n)$ we want to express as a mapping $\lambda(\gamma))$, we can do so by mapping each $o_1, o_2, ...o_n)$ using either mappings of the form of equation 2 or 3.

4 SEMANTIC REPRESENTATION OF JSON

As a proof of concept we show how the previously introduced formalism can be used to map data from RDF to JSON. In order to describe the structure of the resulting JSON text and to provide the links to the input data where we want to have semantic data mapped to our JSON values, we introduce a slim ontology for describing JSON texts. Describing a JSON API in terms of RDF makes it easy for us to express the data to be mapped to our JSON objects in terms of RDF statements, as explained in section 3 . A visualisation of the ontology is shown in figure 1 .

Mainly we make use of the following classes that represent all the valid data types for a JSON object:

Text: The entire JSON text, consisting of objects and values

Value: The super class for all possible JSON values. *Value* has two subclasses, *Compound* and *Primitive*, to distinguish JSON values that themselves can be made up of more complex objects again, and those representing a single literal value.

Compound: Subclass of *Value*, super class of *Object* and *Array*. Compound values can contain other JSON values in one way or another.

Object: Standard JSON object, consisting of a key/-value pair and enclosed in curly braces.

Array: A JSON array containing a set of other JSON *Values*.

String: Primitive representing a string

Number: Primitive representing a number

Boolean: Primitive representing a boolean value

Null: null value

In addition, we defined we following properties to express further relations in the JSON document:

hasRoot: $Text \longrightarrow Value$, the top most element in the JSON hierarchy that also acts as an entry point for the lowering algorithm.

hasKey: $Object \longrightarrow String$, defines the key for the current JSON object

hasValue: $Compound \longrightarrow Value$, the value of the current JSON object

isExpressedBy: $Primitive \longrightarrow rdfs : Statement$ The RDF statement modelling the data representing the primitive.

isRepresentedBy: $Compound \longrightarrow rdf : Class$ The class in the incoming semantic data to which this compound object refers.

5 ALGORITHM

Our goal is to express knowledge encoded in a given RDF source in terms of a *JSON value*. According to the JSON specification (Bray, 2014) , a value must be a *JSON object, array, number, string*, one of the boolean values *true* or *false*, or the special literal *null*. A *JSON text* is defined as a *serialized value*, however, in our case we limit ourselves to the case of JSON texts representing *objects* expressed as *key/value pairs* surrounded by curly braces. Covering the full range of possible JSON values can easily be done using the same algorithm as described below with minimal implementation changes.

JSON texts result in a document with a tree like structure: we have one *root element* that contains at least one key/value pair. The value itself can either consist of a *primitive*, an another *object* or an *array*, a list of values.

In order to build the actual JSON structure, we will start at the designated root element, read the string for the key from our semantic API description and check the value to use. In case of the value pointing to another JSON object, we repeat the above steps recursively, using the resulting JSON object as the value for the corresponding key. In detail, we perform the steps as described in the following.

5.1 Initializing the JSON Structure

The document we describe is represented by a RDF resource of the type `json:Text` that we can retrieve by querying the document for an RDF resource of that type. As defined before, this element must have a property `json:hasRoot` with range `json:Object`. Create a new JSON object as explained below.

5.2 Creating JSON Objects

A JSON Object is defined as such by using the RDF type `json:Object`. As given in the JSON specification, each JSON object consists of a key/value pair. The key is defined by the property `json:hasKey`, as defined in section 4. The value of this property can be immediately used as the string for the key. As defined in the ontology, an object of type `json:Object` *must* have exactly one property `json:hasValue` with range again `json:Object`. Create the JSON object defined by the referenced resource as described here and append it to the *value* position of this key/value pair.

If the JSON object is supposed to represent data of a given type θ from the semantic data layer, expressed by the property `json:isRepresentedBy`, we query

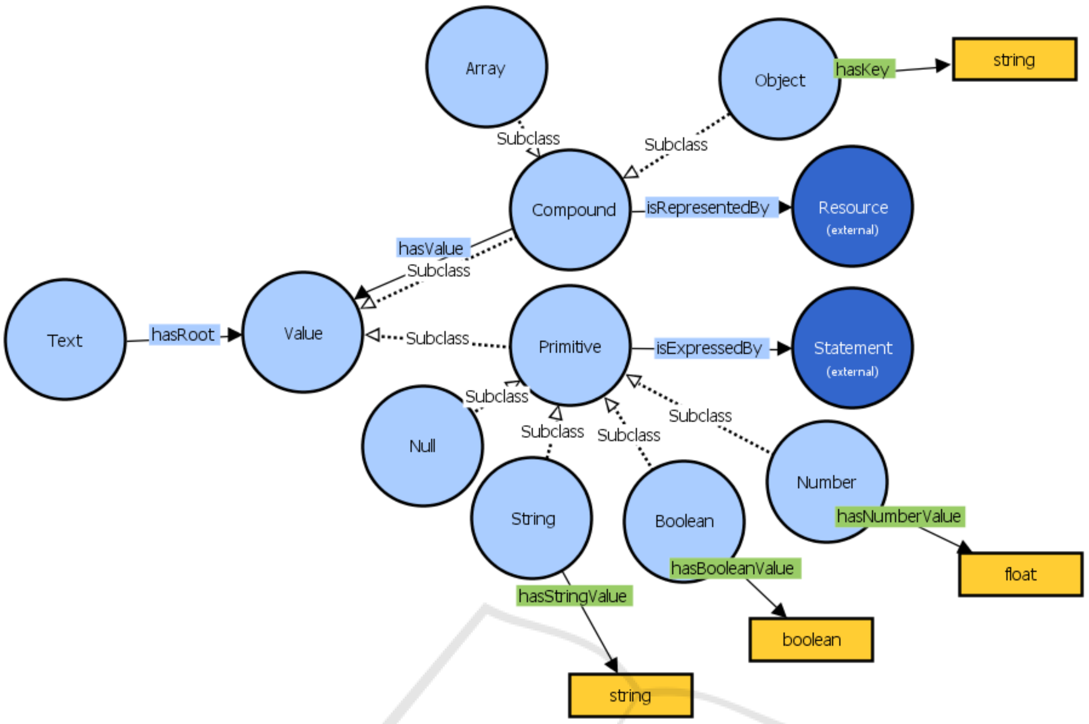

Figure 1: Visualisation of the JSON ontology.

the input data set Γ for a subgraph Γ′ that contains every path from a resource of the specified type θ. Using this subgraph Γ′ in subsequent recursive steps for nested objects, we realize equation 3 of the formal definition given in section 3 .

5.3 Creating JSON Primitives

JSON primitives are defined by the RDF classes `json:String`, `json:Number` or `json:Boolean`. Again, a primitive consists of a key/value pair, but this time we know that no JSON object will appear in the position of the value. Thus, the actual value of this resource is defined by our input data. Therefore, an entity of type `json:Primitive` *must* contain a property `json:isExpressedBy`. The resource defining the primitive to be generated is given by a *RDF statement* defining the class of resources from the input data to use for retrieving the values, along with the property to take the value from. What we receive as result of mapping the JSON primitive is an *interpretation* $I(\sigma_o)$ of the statement $\sigma = (\sigma_s, \sigma_p, \sigma_o)$ as defined in section 3 .

5.4 Creating JSON Arrays

JSON arrays are described with their own class `json:Array` in the ontology introduced in 4. A JSON value defined as `json:Array` in the mapping file will basically be treated like an object, with the difference that multiple matching values will be accumulated and the array itself will be enclosed by square brackets instead of curly braces. In the end, an array will consist of a collection of mapping results $\lambda_1(\gamma_1), \lambda_2(\gamma_2), \ldots \lambda_n(\gamma_n)$ as defined in section 3 .

6 IMPLEMENTATION

As a proof of concept, we provide an example JAVA implementation [3], making use of the *Spring Boot* [4] framework. The resulting JSON object can either be retrieved in the response body of a REST POST request or can immediately be forwarded to another REST endpoint. In the latter case, the data flow between the client and the target service is managed making use of *Spring Cloud Gateway* [5].

To describe the JSON API we want to use, we use *lowering templates* written in Turtle RDF, consisting of a JSON API description using the JSON ontology described in section 4 and a *reification header* to define the RDF statements to be used for JSON primitives.

[3]https://github.com/spidan/poser
[4]https://spring.io/projects/spring-boot
[5]https://spring.io/projects/spring-cloud-gateway

Algorithm 1: POSER Algorithm.

```
Require: Semantic Input data $input$
Require: JSON Payload Model $model$
 1: $result = emptyJSON$
 2: $root \leftarrow JsonRoot(model)$
 3: $result = BuildJSON(input, root, result)$
 4: procedure BUILDJSON($input, jsonModel, result$)
 5:     if jsonModel is Compound then
 6:         if model representedBy subInput then $input \leftarrow subInput$
 7:         end if
 8:         $children = getValues(jsonModel)$
 9:         for all $child \in children$ do
10:             $result \leftarrow BuildJSON(input, child, result)$
11:         end for
12:     end if
13:     if jsonModel is Primitive then
14:         $Value \leftarrow reify(input)$
15:         $result \leftarrow (key, value)$
16:     end if
17:     return result
18: end procedure
```

6.1 Template Files

Templates are written in Turtle RDF and stored locally in the web service by sending a POST request to the `/storeTemplateFile` endpoint of the service. In order to reference a specific template when requesting a mapping from RDF to JSON, provide the name of the specific template file as URL parameter.

The template itself consists of two named graphs: the `json:ReificationHeader` describes how to map actual data properties to the corresponding primitives in the JSON document, and the `json:ApiDescription`, providing a semantic representation of the API itself.

6.1.1 Reification Header

The reification header contains the definition for the RDF statements representing actual data to be used in the generated JSON file. It consists of resources of the class `rdf:Statement`, the class of the corresponding resource in the input data set and a property defining the predicate to use for querying the desired property. For an example, see listing 1.

```
@prefix ex: <https://www.example.org/> .
@prefix json: <http://some.json.ontology/> .

json:ReificationHeader {
 ex:TimeDataSource a
  iots:TimeData;
  rdf:predicate time:dateTime .
}
```

Listing 1: Reification header of a simple lowering template with one primitive value to be mapped.

6.1.2 JSON API Description

The second graph, named `json:ApiDescription`, contains the actual semantic representation of the API along with references to the semantic data sources, following the ontology introduced in section 4. Listing 2 provides an example for a simple JSON text to be generated, consisting of one object with key *data* that contains a string primitive *timestamp* (see listing 3)

```
@prefix ex: <https://www.example.org/> .
@prefix json: <http://some.json.ontology/> .
@prefix iots: <http://iotschema.org/> .

json:ApiDescription {

 ex:JsonModel a json:Text;
  json:hasRoot ex:Data .

 ex:Data a json:Object ;
  json:key "data" ;
  json:value ex:TimeValue ;
  json:isRepresentedBy iots:TimeSeries .

 ex:TimeValue a json:String ;
  json:key "timestamp"^^xsd:string ;
  json:isExpressedBy ex:TimeDataSource .
}
```

Listing 2: API description of a JSON text with one object containing one string primitive.

```
{
 data: {
  timestamp: "2021-01-10T19:58:49.294909Z"
 }
}
```

Listing 3: The JSON object to be created by the above API description.

6.2 Mapping Step

The central part of the service is built on the *RDF4J* [6] library for parsing, processing and writing RDF. The mapping procedure takes two inputs: the *input model*, a RDF model representing the input data, and the *json model*, a RDF model describing the semantic representation of the JSON object we want to generate.

As described in section 5, the first step is determining the root object. For performance reasons we are using the *filter* method of the *RDF4J Model API* [7] instead of storing the data in a local triple store and using SPARQL queries. Retrieving the object of the triple resulting from filtering the input for all statements with `json:hasRoot` in the predicate position yields the unique root value, as by definition there is only one root present in the JSON model description.

Starting from this root object, we recursively call the actual mapping method, taking three parameters: the *object model*, which is the semantic representation of the current JSON object to be generated, the *input model*, which represents the subset of the input data to be used in this mapping step, determined as given in section 5 depending on the presence of a `json:isRepresentedBy` property, and the *result object*, the JSON object to append the mapping results of the current step to and to be returned as the result of the current mapping step.

In each recursion step, we first check whether the current object is a primitive or a compound object, again by using the RDF4J filter function, checking for *rdf:type* in predicate position, to determine the RDF type of the current object.

In case of a primitive we check the corresponding `json:isExpressedBy` statement in the reification header, perform a pattern matching in the input data set, append the result to the current result object and terminate by returning that object.

In case of a compound object, we check for any `json:isRepresentedBy` properties and narrow down the input data set for upcoming recursion steps by finding each path from entities of the specified class in the input data set to a leaf node in the RDF graph. The resulting model functions as the input model for the next recursion step, as object model we choose the resource referenced by `json:hasValue`, again by filtering using the RDF4J model API, and as resultObject we provide the JSON object we have obtained as a result of the previous mapping steps during the recursion.

[6]https://rdf4j.org/

[7]https://rdf4j.org/documentation/programming/model/

7 CONCLUSION AND DISCUSSION

In this paper we have given a formal definition of a possible mapping approach to reduce a semantic knowledge graph to a structured data object. In order to be able to express the mapping rules in a human and machine readable fashion we have introduced a slim, lightweight ontology to describe the structure of an existing JSON API in a semantic linked data fashion and proposed a way how to tie these API descriptions to actual data in RDF format.

Given such an API representation and a set of suitable data queries, we have shown how to generate actual JSON payload, following a fixed given structure, by means of an example implementation. This prototype allows for generating JSON texts from semantic input data and for forwarding this JSON texts to third party JSON APIs, providing an actual example of a possible interoperability layer between the semantic world and a structured data legacy service.

For the sake of simplicity, we have omitted some concepts that were not needed for our test use cases both on the RDF as well as the JSON side for our prototype implementation. First of all, we ignore sets defined in RDF, such as *bags* or *collections* .

Furthermore, arrays in JSON are by definition ordered lists. RDF graphs however are by definition unordered, so while the resulting mappings with our approach will be semantically correct, our goal is to provide a means for realizing an interoperability layer between legacy APIs, some of which may rely on the order of elements in an array. One solution to ensure a certain ordering would be introducing an `orderBy` property to the `json:Array` class which will be added in future releases of the prototype.

In the current version of the prototype there is quite some effort left to the user to write the lowering template files. While some of this work is unavoidable due to the higher expressiveness of linked data and the necessity to give the user the possibility to specify the actual correspondences between the non-semantic structured data and the semantic knowledge graph, especially the semantic representation of the structure of a JSON document as such could be determined automatically, leaving only the task of connecting data to objects and values to the user.

ACKNOWLEDGEMENTS

This work has partially received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No. 857237 (Interconnect)

REFERENCES

Ahlgren, B., Hidell, M., and Ngai, E. C.-H. (2016). Internet of things for smart cities: Interoperability and open data. *IEEE Internet Computing*, 20(6):52–56.

Alexander, K. (2008). Rdf in json: a specification for serialising rdf in json. *SFSW 2008*.

Allocca, C. and Gougousis, A. (2015). A preliminary investigation of reversing rml: From an rdf dataset to its column-based data source. *Biodiversity data journal*, (3).

Amjad, A., Azam, F., Anwar, M. W., and Butt, W. H. (2021). A systematic review on the data interoperability of application layer protocols in industrial iot. *IEEE Access*.

Barbaglia, G., Murzilli, S., and Cudini, S. (2017). Definition of rest web services with json schema. *Software: Practice and Experience*, 47(6):907–920.

Berners-Lee, T., Hendler, J., and Lassila, O. (2001). The semantic web. *Scientific american*, 284(5):34–43.

Bray, T. (2014). The javascript object notation (json) data interchange format. Technical report.

Bröring, A., Schmid, S., Schindhelm, C.-K., Khelil, A., Käbisch, S., Kramer, D., Le Phuoc, D., Mitic, J., Anicic, D., and Teniente, E. (2017). Enabling iot ecosystems through platform interoperability. *IEEE software*, 34(1):54–61.

Bröring, A., Ziller, A., Charpenay, V., Thuluva, A. S., Anicic, D., Schmid, S., Zappa, A., Linares, M. P., Mikkelsen, L., and Seidel, C. (2018). The big iot api-semantically enabling iot interoperability. *IEEE Pervasive Computing*, 17(4):41–51.

Brutti, A., Sabbata, P. D., Frascella, A., Gessa, N., Ianniello, R., Novelli, C., Pizzuti, S., and Ponti, G. (2019). Smart city platform specification: A modular approach to achieve interoperability in smart cities. In *The internet of things for smart urban ecosystems*, pages 25–50. Springer.

de Mello, B. H., Rigo, S. J., da Costa, C. A., da Rosa Righi, R., Donida, B., Bez, M. R., and Schunke, L. C. (2022). Semantic interoperability in health records standards: a systematic literature review. *Health and Technology*, pages 1–18.

Dimou, A., Vander Sande, M., Colpaert, P., Verborgh, R., Mannens, E., and Van de Walle, R. (2014). Rml: a generic language for integrated rdf mappings of heterogeneous data. In *Ldow*.

Ganzha, M., Paprzycki, M., Pawłowski, W., Szmeja, P., and Wasielewska, K. (2017). Semantic interoperability in the internet of things: An overview from the inter-iot perspective. *Journal of Network and Computer Applications*, 81:111–124.

Gardner, S. P. (2005). Ontologies and semantic data integration. *Drug discovery today*, 10(14):1001–1007.

Garijo, D. and Osorio, M. (2020). Oba: An ontology-based framework for creating rest apis for knowledge graphs. In *International Semantic Web Conference*, pages 48–64. Springer.

Group, W. R. W. (2014). Rdf - resource description framework.

Gupta, S., Szekely, P., Knoblock, C. A., Goel, A., Taheriyan, M., and Muslea, M. (2012). Karma: A system for mapping structured sources into the semantic web. In *Extended Semantic Web Conference*, pages 430–434. Springer.

Heiler, S. (1995). Semantic interoperability. *ACM Computing Surveys (CSUR)*, 27(2):271–273.

Lee, E., Seo, Y.-D., Oh, S.-R., and Kim, Y.-G. (2021). A survey on standards for interoperability and security in the internet of things. *IEEE Communications Surveys & Tutorials*, 23(2):1020–1047.

Ma, R., Chen, H.-H., Huang, Y.-R., and Meng, W. (2013). Smart grid communication: Its challenges and opportunities. *IEEE transactions on Smart Grid*, 4(1):36–46.

Michel, F., Montagnat, J., and Zucker, C. F. (2014). *A survey of RDB to RDF translation approaches and tools*. PhD thesis, I3S.

Perumal, T., Ramli, A. R., Leong, C. Y., Samsudin, K., and Mansor, S. (2010). Interoperability among heterogeneous systems in smart home environment. In *Web-Based Information Technologies and Distributed Systems*, pages 141–157. Springer.

Rejeb, A., Keogh, J., Martindale, W., Dooley, D., Smart, E., Simske, S., Fosso-Wamba, S., Breslin, J., Yapa, K., Thakur, S., et al. (2021). The big picture on semantic web and interoperability. what we know and what we don't.

Sadeghineko, F. and Kumar, B. (2021). Application of semantic web ontologies for the improvement of information exchange in existing buildings. *Construction Innovation*.

Spieldenner, T. (2020). On the fly sparql execution for structured non-rdf web apis. In *WEBIST*, pages 243–252.

Sporny, M., Longley, D., Kellogg, G., Lanthaler, M., and Lindström, N. (2014). Json-ld 1.0. *W3C recommendation*, 16:41.

Stonebraker, M., Ilyas, I. F., et al. (2018). Data integration: The current status and the way forward. *IEEE Data Eng. Bull.*, 41(2):3–9.

Uslar, M. and Engel, D. (2015). Towards generic domain reference designation: How to learn from smart grid interoperability. *DA-Ch Energieinformatik*, 1:1–6.

Zeng, D., Guo, S., and Cheng, Z. (2011). The web of things: A survey. *J. Commun.*, 6(6):424–438.