Design of a Web-Service for Formal Descriptions of Domain-Specific Data

Jannik Sidler, Eric Braun, Thorsten Schlachter, Clemens Düpmeier, Veit Hagenmeyer

Design of a Web-Service for Formal Descriptions of Domain-Specific Data

Jannik Sidler[1], Eric Braun[1], Thorsten Schlachter[1], Clemens Düpmeier[1], and Veit Hagenmeyer[1]

Institute for Automation and Applied Computer Science, Karlsruhe Institute of Technology, Karlsruhe, Germany
{jannik.sidler, eric.braun2, thorsten.schlachter, clemens.duepmeier, veit.hagenmeyer}@kit.edu

Abstract. The growing relevance of Big Data and the Internet of Things (IoT) leads to a need for an efficient handling of this data. One key concept to achieve efficient data handling is their semantic description. In the environmental and energy domain, these issues become more relevant since there are measurement stations that produce large amounts of data that software systems have to deal with. In the context of cloud-based infrastructure and virtualisation via containers, microservice architectures and scalability become important aspects in software engineering. This article presents the design of a web service providing software systems with semantic descriptions of data fostering a microservice architecture. It implements key concepts such as domain modelling, schema versioning and schema modularisation. It is evaluated and demonstrated in the context of a current environmental use case.

Keywords: JSON Schema · Semantic Description · Schema Service · Semantic Web Services · Big Data · Internet of Things

1 Introduction

In the past few years, Big Data and the Internet of Things (IoT) have become topics of big significance. Consequently, many resources are invested in research of concepts and technology to improve the overall ability to make use of them in an efficient manner. One important aspect is the management and storage of Big Data that comes along with modern data platforms. Problems of Big Data storage are often symbolised by four words beginning with the letter "v": *volume*, *velocity*, *variety* and *veracity*. Especially, the variety of data semantics is a key problem since Big Data storage solutions have to cope with many different types of data, where each type has its own semantic structure. Therefore, most Big Data storage solutions, for example NoSQL database systems such as time series [13, ?] or document-oriented databases [15, ?], are schema-less, i.e. do not enforce static schemas for data storage. Still, Big Data applications often require a certain degree of structural and semantic understanding of the data, which cannot be acquired by internal database schemas anymore. However, a

formal description of the structure of data can also be achieved by externalizing the semantic description and creating additional metadata schemas of the respective data. This data is stored in an external schema service. This service can deliver a description of how data of a certain type has to be interpreted and therefore simplify data management and processing.

Furthermore, an external semantic schema description covers another important aspect: connections and relationships between data items. If relations are not formally defined, it is difficult for humans and even more difficult for machines to identify them. In software systems (for example search engines), identifiable relations between data lead to a better search interface e.g. connecting search results in a knowledge graph. By using schemas, such relations can be formalised and consequently used by software that has the need for such information. Linked metadata schemas can be used to create Linked Data [19, 20], e.g. the data providing services annotate the data with semantic description elements from the schema descriptions. Externalised schemas are frequently used in semantic web applications, in the context of IoT as described in [5–7], in the context of big datasets [8], or for referring to comparability of data [9].

Another very popular approach in software development is the microservice architecture [24]. The main idea of this architecture is to divide functionality into blocks of a reasonable small size and make these blocks part of a greater, overall functionality, but keeping them independent of each other. This principle contrasts to monolithic architectures, where functional blocks are not implemented independently and the implementations of different functionalities are closely linked to each other. While separating functionality of an application into separate services, single microservices can be far more generic. They can be used even in different application contexts. This advantage of the microservice architecture becomes relevant in many projects and applications today. In environmental projects, it is often necessary to measure various properties of the environment, for example, air pollution properties, such a carbon dioxide emissions, water quality or radioactivity. If the application-specific part of the semantics of such measurements is separated from the meta information such as a metric identifier (identifying the property), a timestamp and a value, a generic time series service can be used to store many different kinds of measurements. Similarly, the important aspects of the measured physical property (name, unit identifier, relation to a measurement device) can be stored in a separate generic service called master data service (more details can be taken from [23]). Therefore, a schema service can provide a formal semantic description of how time series data is structured within the time series service [23], how corresponding master data is structured in the master data service and how certain master data is related to a time series metric. By doing this, it is possible to describe the semantic interpretation context of the time series data, i.e. the physical property and related information which is associated with time series data corresponding to a certain metric.

Motivated by the previously mentioned problem setting, the goal of this article is to describe the design and basic concepts of a microservice that is able to

manage and administer schemas describing the application semantics of specific application domains. Such meta knowledge about data is relevant for many modern application areas like Big Data and IoT, as mentioned before [5–9] and furthermore, the importance of generic microservices which can be used, deployed and executed in a cloud-based environment in a generic application independent way rises continuously. For the design of such microservices and their interoperability with a schema service adding application-specific meta knowledge, there are several requirements that have to be considered.

The first requirement is related to the basic functionality of the service. It is intended to be used in a productive environment, which implies that it is necessary to offer basic data operations for users who work with it. These operations are derived from the CRUD principle (Create, Read, Update and Delete). In the context of this article, schemas have to be accessible via an appropriate REST API [12, 17] that offers such operations.

Additionally, it is necessary to control the formal correctness of metrics and data in general. For this purpose, an automatic schema/metadata validation mechanism is necessary. This mechanism has to supervise the formal correctness of available data by validating it against an appropriate schema before a create/update operation is executed. Consequently, an update of data is only allowed to be executed if the validation of the data is successful.

Another requirement is the need for a versioning concept. Client applications may have to work with a specific version of data, structured accordingly to a certain version of the application schemas, while higher versions of the schema are already created for working with next generation clients. In this case, in order to provide backward compatibility, it is necessary that older data versions are supported although a newer version is already available (and possibly even recommended for usage). Versioning also guarantees the availability of a history feature, which is crucial for supporting old data that still can be of interest. If the data format changes in a certain period of time, old versions of data may not be possible to be processed correctly anymore. In this case, the availability of older schema versions is necessary to process old data.

To model relations between different schemas, it is necessary to include references that point to related schemas. This is supported by the usage of a specific vocabulary, for example JSON Schema [1, 2] It allows the usage of a special keyword which offers the inclusion of external data in order to mark relations between schemas and to enable a modular schema structure with low redundancy. Linked resources are identified by a Uniform Resource Identifier (URI). For an efficient usage of references, it is mandatory to have a suitable domain concept that relates every resource to its corresponding domain.

The remainder of this article is organised as follows: Section 2 deals with related work that examines similar content referring to this article. Section 3 describes solutions for the main concepts mentioned above. Section 4 evaluates the presented concepts. In Section 5, a conclusion is given as well as an outlook for further work.

2   Related Work

This section deals with publications that are related to the general idea of the present article and to parts of the requirements.

The article by Chervenak, Foster & Co. [8] deals with the management of datasets of large volume in scientific contexts. They describe the design of a data grid and suggest a concept for a schema service describing the data in which they decide to distinguish schema information between payload and metadata. The payload in this case is the actual content of the schema that describes the underlying data, whereas the metadata is a piece of information that defines meta attributes of the schema. As reasons for this distinction, they mention increased flexibility regarding the storage system implementation and less effort when changing behaviours that affect one metadata or payload description. Additionally, metadata is divided into different kinds of metadata which are application metadata, replica metadata and system configuration metadata, where each of these respectively covers different tasks. The article emphasizes that the separation of metadata and payload is an important matter for many years and that the design of a metadata service in combination with Big Data was already reasonable in the year 2000. A similar approach can also be found in [6]. The approach described later divides information in schema documents into "payload" and "metadata".

Another related article is given by Krylovskiy, Jahn and Patti [5]. t deals with the design of a smart city IoT platform by using the microservice architecture. The presented platform architecture consists of applications, a service platform, containing middleware services and smart city services, and information models. Additionally, it contains components for the management of platform metadata. Data can be accessed by a client application via a REST API and is stored in a document-oriented database. Data with more semantic structure is stored in a triplestore database and can be accessed by a semantic web client. The article describes the advantages of the microservice architecture in the context of its service platform, the most important ones are the componentisation of functionality, decentralised governance and data management, which lead to technology heterogeneity, resilience, good scaling and composability. The separation of metadata and payload consequently can be covered by using a microservice architecture, which may lead to a separate service only dealing with metadata. However, the work presented in [5] describes metadata in an abstract way, and does not mention aspects like an appropriate metadata model or a distinction of metadata depending on their respective domain.

Additionally, related to the present article is the work given by Agocs and Le Goff [10]. It deals with the architecture of a web service using a REST API and JSON Schema to construct knowledge graphs for data visualisation. They describe the need for descriptors that are used to validate data. Moreover, they suggest an ontology-like hierarchy as data structure. The latter requirement is applied by using JSON Schemas referencing functionality. Agocss and Le Goffs design of the web service is similar to the one that is described in the present

article as they use a microservice architecture as well as a REST API, which provides basic CRUD operations for applications using the knowledge graph.

However, concepts that are not discussed in Agocss and Le Goffs work are a versioning concept, which will be included in the present work. Additionally, the creation and management of domain concepts for different applications in the same schema service and a metadata model are not addressed by Agocss and Le Goffs work but will be discussed in the following chapters as part of the solution presented in this paper.

3   Concept and Architecture

In this chapter, a solution for the problems described in the introduction is presented.

3.1   Domain Model

First, the term **Domain-specific Data** is discussed. It refers to different categories of data for different application domains, for example environmental data or energy data. These two application domains serve as examples in the context of the present article as they already have good and well known semantic models for their data. Domain-specific data is hierarchically categorised according to its domain-specific meaning. A category defines a more specific type of data which can be divided into more specific subgroups on its own. This process is repeated until the scope of the grouped data items is specific enough that the structure of the data can be defined by a formal description. The category names can be associated with a more formal definition of a vocabulary of domain terms with precise semantics within the application domain, which can also be described by a thesaurus. Adding structural information to certain categories, results in a domain model. Figure 1 shows an example of such a hierarchical categorisation and is a visualisation of the air/climate domain model given by the Umweltbundesamt [18]. The air/climate domain model contains more terms in the hierarchy, which are not depicted in Figure 1 to keep it clear.

At the level of gaseous pollutant, i.e. emissions of gas into the air, an associated data schema can be basically defined by the name or type of the pollutant (ozone, nitrogen dioxide) and its concentration, which can be seen as a measurement value MV (see Figure 1) if there are means for measuring or calculating it from measurements. This contrasts to air pollutants which are not gaseous but particles (e.g. particulate matter). The size of the particle and the particle type mix is important besides the concentration. Therefore, both concepts lead to different schemas.

A set of such schemas which define the data semantics of all data belonging to certain domain terms (e.g. gaseous pollutant) is called **schema domain**. The schema service discussed in the present article allows to create as many schema storage containers as required to provide schema domains as sets of schemas to different applications. These applications can have different application domains
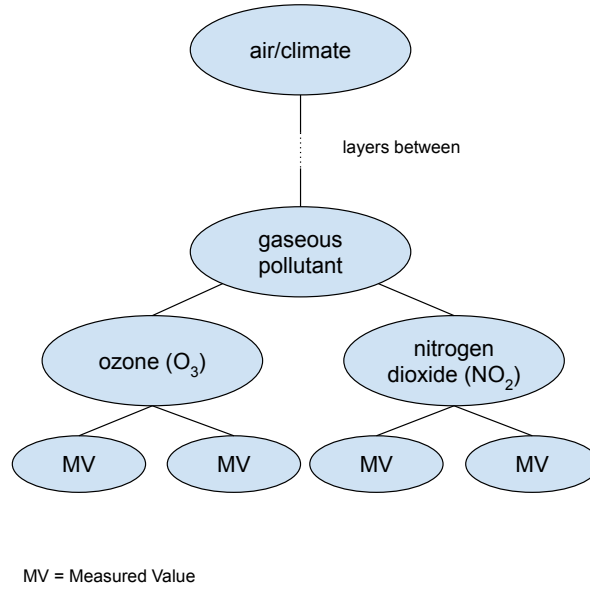
Fig. 1. Excerpt from the air/climate domain model, extended by measured values.

as well. Each schema domain can contain many schemas which define the structure of certain types of data within the domain.

As schemas can reference other schemas to implement relations between them, schemas defining the data structure of a certain application domain are closely related to each other. If the structure of schemas is enhanced over time, new versions of schemas are created, and not all versions of different schemas are compatible with each other regarding their relationships. Therefore, it is necessary to have a versioning concept for schema domains and for single schemas.

3.2 Versioning Concept

In this section, the versioning concept is discussed. Principally, there are different methods and use cases how versioning can be applied to schemas. In the present article, three approaches are discussed. The first one is Domain-specific Versioning, which attaches a version number to a whole schema domain. In this approach, all schemas that belong to the same domain have the same version number as the schema domain. Consequently, updating a single schema in a (sub)domain leads to an update of the version number of all schemas in this domain. The mechanism is depicted in Figure 2. It shows an update request which is handled by an interface managing the update of the domain. This leads to a consistent version number in the entire domain, which is a crucial feature for using software applications. However, this uniform version comes along with a disadvantage. To keep the version consistent, every update leads to a large
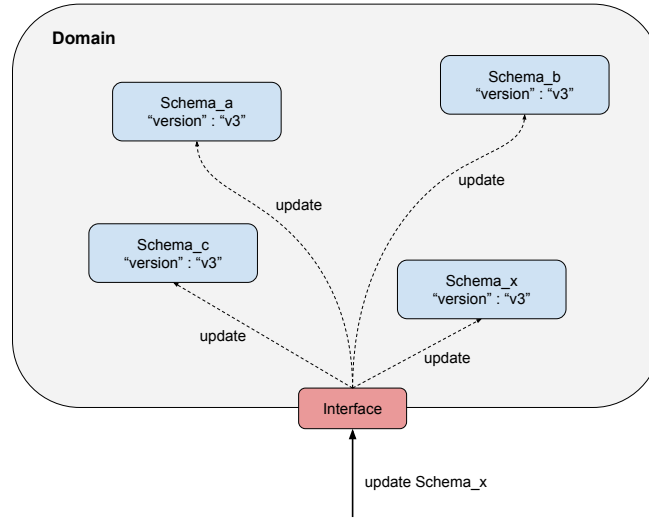
Fig. 2. Schema updating with Domain-specific Versioning.

number of update requests. Even if a schema contains no changes for a new version, the version number must be incremented. Depending on the frequency of updates, the effort for incrementing the version number may be too high to be negligible.

This problem is the motivation for a second versioning approach, the Schema-specific Versioning. In this approach, every schema has its own version number, which implies that every schema can be updated independently of each other. Figure 3 shows the updating of a schema using the Schema-specific Versioning method. As depicted, the updating of a single schema does not affect other schemas in the domain. The advantage of this strategy lies in the efficiency as only the affected schema is updated. This concept is well-known from versioning source code files in software development processes [21, 22] and suitable for authoring schemas since changes of schemas are tracked by the revision number and different revisions can be compared to each other. However, as a consequence, there is no consistent and uniform version number which may lead to difficulties as schemas are linked to each other, and applications have no precise view on which version of a schema is linked to which version of another schema.

For this reason, the third approach combines both formerly presented concepts. The combination is similar to the versioning concepts applied to software code where each source code file has a revision number. Each schema (e.g. analogous to a source code file) has an internal version that is called revision number. It is only relevant for authoring and managing schemas and schema domains but not propagated to applications that are working with the data. Additionally, there is a domain version applied to a schema domain as a whole which

can be considered as the version number of a schema domain release, which can be a set of consistent schema definitions that are used by applications. The domain version is relevant for applications and users working with schemas to access a consistent set of schemas. The schema service itself internally manages a mapping which revision number of a certain schema belongs to a given schema domain release. Figure 4 shows an illustration of this concept.
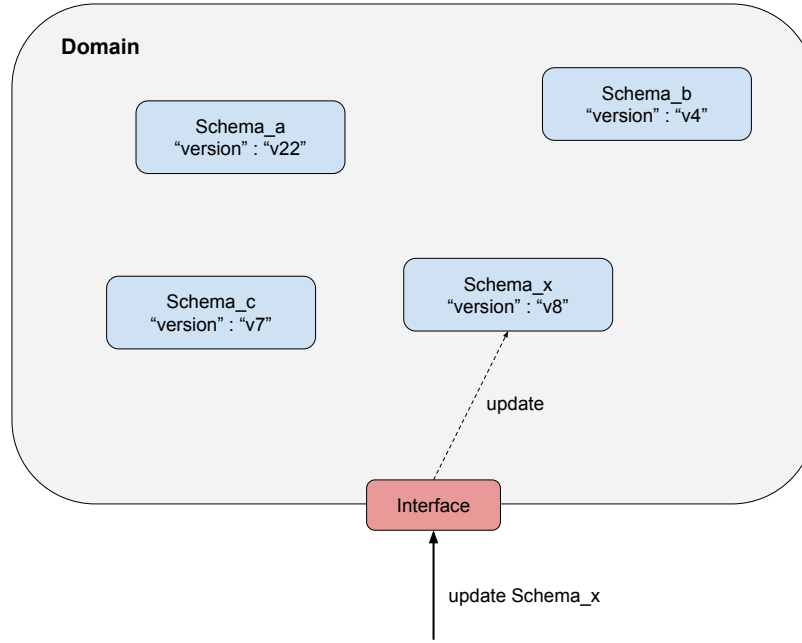

Fig. 3. Schema updating with Schema-specific Versioning.

Related to this approach is the question of an updating strategy. Typically on demand of application developers, schema authors have to evolve schemas to add new functionalities. This is performed by preparing new schema domain releases. The combined version approach supports this: schema authors work on new releases by committing new versions of single schemas or single schema sets analogous to the versioning of software source code which results in new instances of the schema objects internally having an incremented revision number. When a new set of consistent schema instances is finalised, a new schema domain release is prepared by assigning the corresponding revision numbers of the schemas to the new schema domain release. Afterwards, the consistent set of schemas is released to be usable for applications. The applications refer to the new release by the new schema domain version number. To provide backward compatibility, the schema service has to provide more than one release of the same schema

domain to clients according to the version number the client application uses. For maintaining consistency across all schemas of a domain release, it is important that schema revisions are fixed and not changeable anymore when they are assigned to a released schema domain version.

Fig. 4. Updating using combined strategy of Domain-specific Version and Schema-specific Version.

3.3   Modularisation

Another important topic in the context of this article is the modularisation of schemas. Modularisation is a consequence of using references in schemas to divide large schemas into smaller pieces. This approach has various advantages. Redundancy is significantly reduced by using references. This leads to a concept where schema information is stored exactly once, which means that schemas are reusable. Consequently, updating is less expensive since an update request affects only smaller parts of a schema. Strongly related to this is reusability which is a desired feature as it reduces the efforts for updating and further editorial work. Such reusable, "common" schemas are helpful for authors who need them as they can include them instead of creating them again. Additionally, schemas become more readable for humans.

Related to the usage of references is their resolving. A reference is a URI that points to a specific schema at a specific location described by the reference itself. The resolving indicates the process of replacing the references URI by the referenced schema itself. As there may be applications that are not able to resolve references by themselves, the service contains a functionality that performs the resolving on demand. As internal references (where the referenced schema is part of the schema itself, in which it may be used multiple times) can be resolved implicitly by the usage of JSON Schema [1, 2], external references (where the referenced schema is located in a separate document) have to be treated differently. JSON Schemas "$ref" keyword uses URIs to define the location of a specific linked schema. To resolve external references, an algorithm is needed that locates all the corresponding references, queries the linked schemas using

the URI and writes them to the correct location in the schema. The algorithm exactly fulfills the described requirements by recursively iterating through the schema, detecting all references, querying the respective reference schema and editing the base schema correctly. Whenever it detects the "$ref" keyword in a schema, it uses the value of this key to query the corresponding schema from the database and writes it to the proper location, adding all necessary syntactical characters. Whenever another keyword is detected, it is checked if there is a nested schema. The complexity of the algorithm depends on the number of nested schemas that are located in the main schema. The more nested the schema structure is, the more recursive steps the algorithm has to perform.

Related to the modular schema structure that uses references is the usage of a classification concept which divides the set of schema documents belonging to a schema domain into more modular parts (in the following called *package*). In many (sub)domains, schemas can be divided into several groups of reusable base schemas, for example basic data attribute definitions, basic data objects, such as measurements or more complex application object schemas. For this type of classification as well as for assigning internal revision numbers to schemas, metadata attributes are required to be assigned to schema documents. As discussed before, it is desirable to separate payload and metadata in a schema document. By using JSON Schema, a possible representation of the schema document structure is shown in Listing 1.1. This example contains the different sections for metadata and payload (schema).

```
"metadata" : {
        "class": "measurement",
        "package": "DO",
        "revisionNumber": "r44",
        ...
}
"schema": {
        ...
}
```

Listing 1.1. Structure of schema documents within a storage container of a document-oriented database of the schema service (related to one schema domain).

The metadata section contains three properties:

 – the "class" property, which describes the type of the schema and the derived objects which are instances of that schema
 – the "package" property, which defines the package to which a schema definition belongs to (e.g. DO for Data Objects)
 – the "revisionNumber" property, which represents the internal revision version

Moreover, Figure 5 models the validation process that is used in JSON Schema and in the schema service. It consists of three different layers. The lowest one

is the object layer, where objects are given in the JSON data format [3]. They are validated against a certain JSON Schema that serves as formal prototype for the object. This schema, on the other hand, defines the structure of the objects. JSON Schemas are the middle layer in the model given by Figure 5. They are validated themselves against the upper layer, the JSON Meta Schema or JSON Schema Draft. This draft defines the keywords and their functionality and thereby, it defines the JSON Schemas in the middle layer. Figure 6 depicts
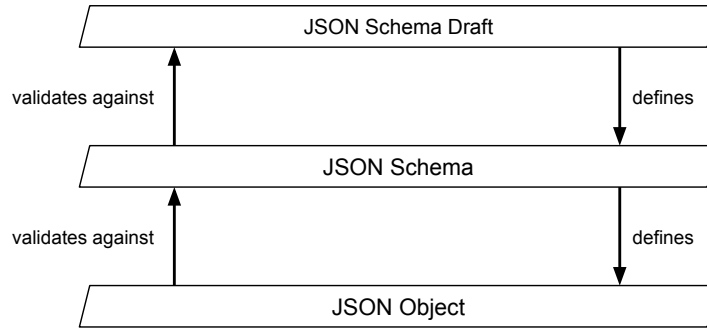

Fig. 5. Hierarchy in the validation process with JSON Schema.

the architecture of the schema service. It consists of different components containing different tasks needed in the context.

The service is used by client applications. Examples for client applications are dashboards or other visualisation components. For the applications, the service provides a REST Interface (REST API) containing the necessary functionality to process client requests. Requests are received and processed by the Application Controller (AC). The AC uses a configuration file to manage necessary system parameters, for example ports or authentication/authorisation information. If the request contains a request body (in case of create/update requests), it is validated by the validator first. If the validation is successful, the AC uses a database interface to translate the request to the corresponding database query. The query is sent to the database where the desired data is stored. If the validation fails or the requested data is not available, the client receives an error request with the corresponding HTTP status code.

3.4   Prototype Architecture

Figure 6 depicts the architecture of the schema service. It consists of different components performing different tasks needed in the context. The service is used by client applications. Examples for client applications are dashboards or other visualisation components. For the applications, the service provides a REST

Interface (REST API) containing the necessary functionality to process client requests. The REST API is designed accordingly to [12]. Requests are received and processed by the Application Controller (AC). The AC uses a configuration file to manage necessary system parameters, for example ports or authentication/authorisation information. If the request contains a request body (in case of create/update requests), it is validated by the validator first. If the validation is successful, the AC uses a database interface to translate the request to the corresponding database query. The query is sent to the database where the desired data is stored. If the validation fails or the requested data is not available, the client receives an error request with the corresponding HTTP status code.
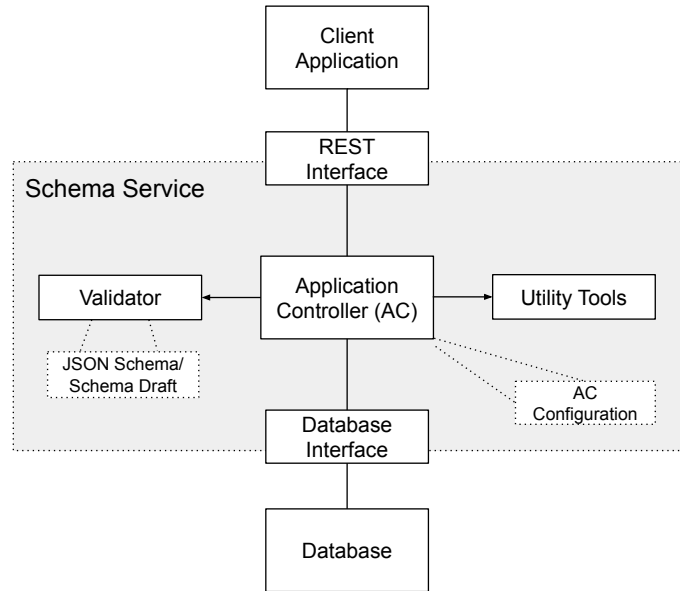

Fig. 6. Architecture of the the service and conntected systems.

4   Evaluation

To evaluate the concept provided within this article, the service was tested within an application of the Landesanstalt für Umwelt Baden-Württemberg (LUBW), Germany. The application beside other usages instruments a google maps chart with an additional layer that shows the nitrogen dioxide content in the air (see Figure 7) at different measurement points in Baden-Wrttemberg. On the right side of the figure, the meaning of the different measurement point colors is shown, which changes with a rising or falling value of nitrogen dioxide in the air depending on which predefined range of values contains the value.
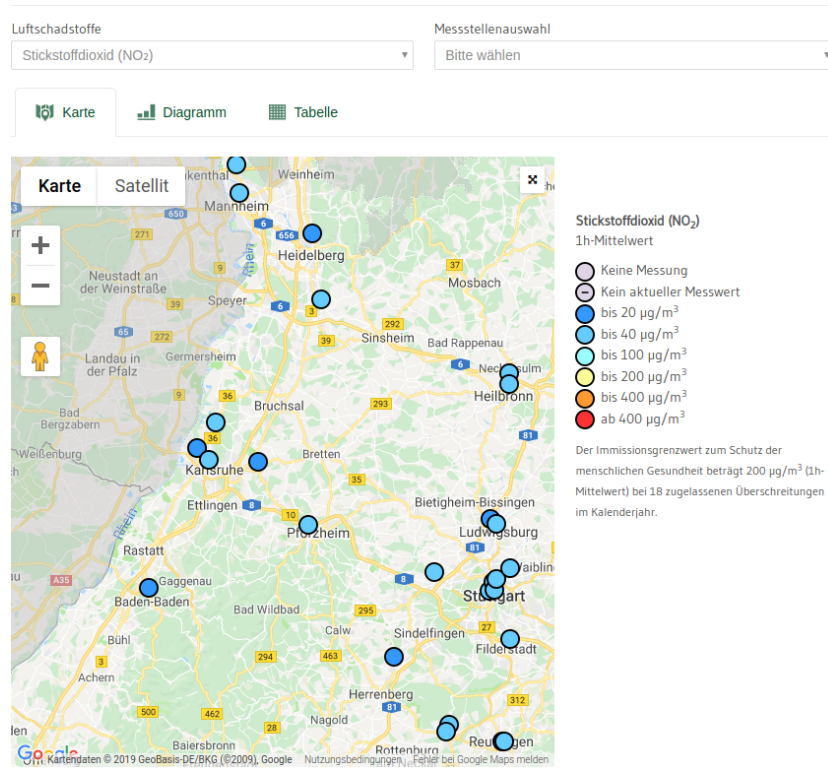
Fig. 7. Map with measurement data that shows the nitrogen dioxide (NO$_2$) content in the air.

In this example context, the schema service provides advantages for the application. On the one hand, data that is stored in the system can be validated. This helps to reduce the existence of error values which contain an illegal format or illegal values. On the other hand, time series data used in the measurements can be schematically linked with master data objects which provide the domain specific interpretation context to the measurement values. They share information about the chemical property measured (as nitrogen dioxide), the unit of the measurement value, the time resolution, the measurement environment (measurement station equipment) and the location of the station. The service that delivers the data to the map client component is able to resolve the references for concrete instance data and to provide an aggregated data object, which contains all the information beside the measurement values that is required to have the coloring information and the legend information available to render the data on the map. Additionally, schemas are useful for preconfiguring components as selectors (e.g. for filtering data) based on classification information according to the schema of the corresponding dataset.

5   Conclusion

In the present article, the need for semantic descriptions of data objects in an application domain and the usefulness of an external schema service for it were motivated. Afterwards, related works and needed functionalities of such a service were discussed. First, the versioning of semantic descriptions were discussed and an appropriate concept was presented which is analogous to the versioning of source code and releases in software development. Second, the data format of a schema document within the schema service and its metadata for management of schemas were described. Furthermore, a short overview of the overall architecture of the schema service was presented. Finally, the evaluation described in the evaluation chapter showed that a larger environmental software project can benefit from the presented concepts of the schema service in different ways. It provides stricter checking of data consistency, can link data to meta information given an interpretation context for the data which can be used by an application without hardcoding the interpretation knowledge into the application itself. Thus, it helps to implement advanced, but helpful functionalities for users, such as filtering of data or navigation between data within the application.

Important further work lies in the extension of the service API. A basic set of functions that is required for the usage of the service has already been implemented. Still, additional features such as extended filtering and an extended search would improve the API. Additionally, a more powerful user interface is needed for updating and managing schemas. It simplifies the verification process of the service API and makes it more reliable. Moreover, different data formats can be considered. One of the services limitations is that it works with JSON/JSON Schema only, which are the most widespread data formats in the context of web engineering. Still, it may be useful to support other data formats as well, for example XML/XSD or RDF/OWL. Especially, schema information returned to the client can be augmented with semantic annotations leading to Linked Data using JSON-LD.

References

1. JSON Schema Homepage, http://json- schema.org. Last accessed 13 Sep. 2019.
2. Zyp, K., Court, G., Galiegue, F.: JSON Schema: core definitions and terminology, Internet Engineering Task Force, Internet-Draft draft-zyp-json-schema-04, Aug. 2013, https://tools.ietf.org/html/draft-zyp-json-schema-04. Last accessed 20 Aug. 2019.
3. The JSON Data Interchange Format, 1st ed. ECMA International, October 2013, http://www.ecma- international.org/publications/files/ECMA-ST/ECMA-404.pdf. Last accessed 23 Aug. 2019.
4. Bray, T., The JavaScript Object Notation (JSON) Data Interchange Format, IETF RFC 7158, Oct. 2015, https://rfc-editor.org/rfc/rfc7158.txt. Last accessed 23 Aug. 2019.
5. Krylovskiy, A., Jahn, M., Patti, E.: Designing a Smart City Internet of Things Platform with Microservice Architecture. In: 2015 3rd International Conference on Future Internet of Things and Cloud, Rome, August 24-26 2015. https://doi.org/10.1109/FiCloud.2015.55

6. Mattmann, C., Crichton, D., Medvidovic, N., Hughes, S.: A Software Architecture-Based Framework for Highly Distributed and Data Intensive Scientific Applications. In: ICSE '06 Proceedings of the 28th international conference on Software engineering, Pages 721-730, Shanghai, May 20-28, 2006.

7. Kolchin, M., Klimov, N., Shilin, I., Garayzuev, D., Andreev, A., Mouromtsev, D.: SEMIOT: An Architecture of Semantic Internet of Things Middleware. In: 2016 IEEE International Conference on Internet of Things (iThings) and IEEE Green Computing and Communications (GreenCom) and IEEE Cyber, Physical and Social Computing (CPSCom) and IEEE Smart Data (Smart Data). https://doi.org/10.1109/iThings-Green-CPSCom-SmartData.2016.98

8. Chervenak, A., Foster, I., Kesselman, C., Salisbury, C., Tuecke, S.: The data grid: Towards an architecture for the distributed management and analysis of large scientific datasets. In: Journal of Network and Computer Applications 23, 187-200 (2000). https://doi.org/10.1006/jnca.2000.0110

9. Kettouch, M., Luca, C., Hobbs, M.: Using Semantic Similarity for Schema Matching of Semi-structured and Linked Data. In: 2017 Internet Technologies and Applications (ITA). https://doi.org/10.1109/ITECHA.2017.8101923

10. Agocs, A., Le Goff, J.-M.: A web service based on RESTful API and JSON Schema/JSON Meta Schema to construct knowledge graphs. In: 2018 International Conference on Computer, Information and Telecommunication Systems (CITS). https://doi.org/10.1109/CITS.2018.8440193

11. Braun, E., Schlachter, T., Duepmeier, C., Stucky, K.-U., Suess, W.: A Generic Microservice Architecture for Environmental Data Management. In: Environmental Software Systems. Computer Science for Environmental Protection. ISESS 2017. IFIP Advances in Information and Communication Technology, vol 507. Springer, Cham. https://doi.org/10.1007/978-3-319-89935-0_32

12. Giessler, P., Gebhart, M., Steinegger, R., Abeck, S.: Checklist for the API Design of Web Services based on REST. In: International Journal on Advances in Internet Technology, vol. 9, no. 3 & 4, 2016.

13. Jensen, S.-K., Pedersen, T.-B., Thomsen, C.: Time Series Management Systems: A Survey. In: IEEE Transactions on Knowledge and Data Engineering, vol. 29, no. 11, November 2017. https://doi.org/10.1109/TKDE.2017.2740932

14. Influx DB: https://www.influxdata.com/. Last accessed: 09 Sep. 2019.

15. MongoDB: https://www.mongodb.com/. Last accessed: 09 Sep. 2019.

16. Elasticsearch: https://www.elastic.co/. Last accessed: 09 Sep. 2019.

17. Fielding, R.: Architectural Styles and the Design of Network-based Software Architectures. University of California, Irvine, 2000.

18. Environment Thesaurus of the Umweltbundesministerium. https://sns.uba.de/umthes/de/hierarchical_concepts.html. Last accessed: 27 Nov. 2019.

19. Leadbetter, B., Smyth, D., Fuller, R., OGrady, E., Shepherd, A.: Where Big Data meets Linked Data: Applying standard data models to environmental data streams. In: 2016 IEEE International Conference on Big Data (Big Data). https://doi.org/10.1109/BigData.2016.7840943

20. Al Rasyid, M., Syarif, I., Putra, I.: Linked Data for Air Pollution Monitoring. In: 2017 International Electronics Symposium on Knowledge Creation and Intelligent Computing (IES-KCIC). https://doi.org/10.1109/KCIC.2017.8228565

21. Hildenbrand, T., Rothlauf, F., Geisser, M., Heinzel, A., Kude, T. Approaches to Collaborative Software Development. In: 2008 International Conference on Complex, Intelligent and Software Intensive Systems. https://doi.org/10.1109/CISIS.2008.106

22. Hata, H., Mizuno, O., Kikuno, T. Historage: Fine-grained Version Control System for Java. In: IWPSE-EVOL '11: Proceedings of the 12th International Workshop on Principles of Software Evolution and the 7th annual ERCIM Workshop on Software Evolution, September 2011. https://doi.org/10.1145/2024445.2024463
23. Prasad, S., Bhole, A. Application of Polyglot Persistence to Enhance Performance of the Energy Data Management Systems. In: 2014 International Conference on Advances in Electronics, Computers and Communications (ICAECC). https://doi.org/10.1109/ICAECC.2014.7002444
24. Newman, S.: Building Microservices. OReilly Media Inc., 2015.