Microservice-based Reference Architecture for Semantics-aware Measurement Systems

Zur Erlangung des akademischen Grades eines

Doktors der Ingenieurwissenschaften

von der KIT-Faktultät für Informatik
des Karlsruher Instituts für Technologie (KIT)

genehmigte

Dissertation

von

M. Sc. Eric Braun

Tag der mündlichen Prüfung:	27.10.2020
Hauptreferent:	Prof. Dr. Veit Hagenmeyer
Korreferent:	Prof. Dr. Sebastian Abeck

Abstract

Cloud technologies have become more important than ever with the rising need for scalable and distributed software systems. A pattern that is used in many such systems is a microservice-based architecture (MSA). MSAs have become a blueprint for many large companies and big software systems. In many scientific fields like energy and environmental informatics, efficient and scalable software systems with a primary focus on measurement data are a core requirement. Nowadays, there are many ways to solve research questions using data-driven approaches. Most of them have a need for large amounts of measurement data and according metadata. However, many measurement systems still follow deprecated guidelines such as monolithic architectures, classic relational database principles and are missing semantic awareness and interpretation of data. These problems and the resulting requirements are tackled by the introduction of a reference architecture with a focus on measurement systems that utilizes the principles of microservices.

The thesis first presents the systematic design of the reference architecture by using the principles of Domain-driven Design (DDD). This process ensures that the reference architecture is defined in a modular and sustainable way in contrast to complex monolithic software systems. An extensive scientific analysis leads to the core parts of the concept consisting of the data management and semantics for measurement systems. Different data services define a concept for managing measurement data, according meta data and master data describing the business objects of the application implemented by using the reference architecture. Further concepts allow the reference architecture to define a way for the system to understand and interpret the data using semantic information. Lastly, the introduction of a frontend framework for dashboard applications represents an example for visualizing the data managed by the microservices.

Contents

1 Introduction

1.1 Motivation

The past years have shown a drastic increase in the complexity of software systems, especially applications with backend servers. With the increasing focus on web-based applications, client-server architectures became more popular. Further, Web 2.0 and HTML5 technologies [ZC07] increased the amount of software systems which implement their business logic in a server or service backend. Architectures required for such software systems often follow the well-known model-view-controller pattern [GH+94]. Furthermore, the vast change from desktop PCs to mobile devices led to an even higher increase in applications that have a web-based frontend in order to run the client application in the browser. These applications are highly compatible with nearly every device that runs a browser, since modern browsers mostly follow the same standards[1]. Since software in general increases in complexity over time and many companies have quite large code bases in use, the complexity of possible monolithic systems is very hard to maintain. Besides the increase in complexity, the hardware that is nowadays available to deploy such backend systems on has become more efficient and cheaper. This combination shows a trend away from legacy server systems to distributed software systems that run on cloud infrastructures [KQ17]. Another drastic change that took place over the last decade is the amount of data that is collected by e.g. sensors, simulations or users. This implies that storage, processing and distribution of data require more and more resources to satisfy the desired speed, volume and security of information. Cloud systems can be scaled horizontally and vertically. However, vertical scaling is limited by the amount of resources a cluster node provides. Therefore, horizontal scaling (i.e. adding more nodes) can not be ignored in order to fulfill the mentioned goals. A key concept for the design of modular and distributed systems that has grown in popularity over the last years is the microservice-based architecture [Ne15, FL14]. It emphasizes the modularity and loose coupling of software systems in contrast to monolithic backends.

Especially with the introduction of container virtualization, continuous integration/deployment (CI/CD) and DevOps [KL+16a], software engineers got many tools changing the way of

[1] W3C Standards: https://www.w3.org/standards/

architecting and developing software. The change from monolithic software to modular and scalable microservices allows developers to use the technology stack and even programming language of their choice. Each microservice can be deployed separately using container virtualization. When it comes to the designing and testing of software, there exist many tools that help software engineers to create modern distributed architectures [Ev03, HG+17] and to test their software in an automated way using CI/CD [Vi15]. This allows architects and developers to work more efficient, since a lot of processes can be automated. Besides the development and operation of software systems, also the design process of microservice architectures is evolving and getting more attention. Especially the principles of domain-driven design [Ev03, HG+17] are becoming a promising technique in order to design microservice architectures. The main idea is to focus on the domain of the software and the relationships between the different parts of this domain. Once this process is finished, different artifacts help to better understand how to realize the architecture. This process is used in the present thesis in order to design the reference architecture.

In the fields of energy and environmental informatics, which are in the focus of the thesis, scientists collect a lot of measured data that have to be stored, managed and retrieved efficiently. Besides measurements from real sensors, also simulations produce time series data that have to be treated in a similar fashion. Many current systems in science and industry in these fields store such data as files without any management or in a monolithic data management system that does not scale well and is hard to maintain [VG+15]. Another common problem with current measurement management systems is that, often, classic relational databases (RDBMS) are used to store time series and their metadata [JP+17]. Such database systems have limited features when it comes to time series data and do not support scalability across multiple nodes in a cluster. Therefore, these databases are not suitable for distributed management of large measurement data sets. The architectures of measurement systems and IoT (Internet of Things) platforms, therefore, follow the technological trend of the mentioned cloud technologies. In combination with the advantages of microservices [VL+14, KJ+15], modern measurement systems have a good foundation to tackle the challenges of distributed systems collecting data from sensors or simulations and manage these data in an efficient way. The mentioned trend also introduces many new database technologies that try to solve the problem of scalability for such measurement systems. Literature already shows many approaches that present a microservice architecture to solve the challenges of modern measurement data management [VL+14, GM+17, SL+17]. However, there is still a gap when it comes to measurement systems that have to manage different types of time series data and their respective metadata. Time series can come in many different types such as univariate and multivariate types but also as data sets that are of moderate volume and that have a very high resolution. These data sets, called sampled values [Om19], have very special requirements. Additionally, the semantics of time series and the respective metadata coming from sensors or simulation setups are important in order to classify and understand the actual data. Finally, most scientific articles, e.g. [[DB18, GM+17, KJ+15, KK+16]] are not mentioning how to integrate such an architecture in an application that also introduces a suitable concept for its frontend.

1.2 Scientific Placement

As already mentioned this thesis focuses on two scientific subfields of computer science: the energy and environmental domain. From the digital point of view, research in the energy field is facing an important step. With the potential of renewable energies and the increase of electric vehicles the energy sector has a much higher need for information technology than in the past. Monitoring, control and simulation of smart grids [GG11] will be a big challenge in the future. For such systems, it is indispensable that they can manage measurement data and their associated metadata (e.g. unit of measurement, multiplication factor or location of measurement) at a high level of efficiency. Additional data might also be stored and managed, which might include describing data for measurement devices, machines that are monitored or controlled (e.g. PV panels, battery storage systems or a simple temperature sensors monitoring critical temperatures of the system) or other assets of the energy domain. A third type of data that has to be managed are files and binary data sets which just have to be stored and retrieved on demand. An example for such data are images, documentation in form of Microsoft Word or PDF, or simulation models coming as executables, or binary files that can be executed by a specific software. This combination of requirements for the management of data often leads to software storing lots of data which is hard to keep track of and to classify. The reference architecture conceptualized in the present thesis is defined as a generic measurement system able to manage data without knowing the semantic description of this data beforehand. This allows users and applications to bring their own data to the system. The semantic description of this data still has to be defined in order to understand the information that is managed. Since this semantic description is not included in the API or the source code of the system, it has to be managed separately. The energy domain has a bunch of standards describing the different aspects of the field. The *CIM* (common information model) standard [US+12], for example, defines an ontology for electrical substations and the *IEC 61850* standard [Ma06] defines semantics for measurement devices and a communication protocol. These standards are very important for the future and therefore more and more adopted by the industry. The managed data have to be compatible with these standards in order to efficiently manage each data set and to keep the data compatible with third party software systems and protocols.

A category of software systems in the energy field that clearly shows the need for multifarious data management are control center applications used by transmission system operators in order to monitor and control the electricity grid on a regional or national level. The main characteristic of such a control center is a wall of displays showing a large visualization of the electricity grid and its status. This visualization is fed from many different measurements across the whole grid. The amount of data needed for such a scenario shows the need for not only databases being able to handle the load and diversity of the data but also providing an efficient management of it. Besides the electricity grid, also heat and gas grids are looked at in case it is of importance for the operator. They have similar requirements.

The second scientific field of interest for this thesis is environmental informatics. In many aspects it is very similar to the energy domain. Furthermore, it is closely linked with the energy sector since the positive or negative effects from the energy transition can be seen and measured in the environmental sector. Monitoring the environment is essential in order to inform citizens (e.g. regulation in Germany: [Deu19]) and to warn people of critical events such as flooding situations, bad air quality or high radiation. In contrast to the wall of screens in a control center, environmental data is usually visualized as publicly accessible web pages reachable from any device in a flexible and intuitive way. This introduces the need for a web-based frontend that is needed in order to visualize the data. There still are many environmental information systems presenting their data by means of static images (e.g. JPEG or PNG). This is quite outdated for the current state of technology since responsive design [VS+15] and the compatibility of multiple devices is nowadays indispensable for web applications. As for data types, besides time series data, environmental applications also need metadata for the measurements as well as other assets. Similar to the energy domain, there exist standards defining semantics[2].

After the introduction of the more specific domains, different problems can be identified that will be discussed in the next section.

[2]ISO 19115: Geographic information – Metadata, ISO 19119: Geographic information – Services

1.3 Problem Description

There are already many publications on different aspects of microservice architectures for the mentioned two subfields in computer science. However, a further analysis shows different problems that have not yet been solved entirely. The following subsections provide an overview and discussion of these problems. In the rest of the thesis solutions for the problems are presented in the form of different contributions. The problems are referred to as P1 to P4 respectively.

1.3.1 P1: Monolithic Measurement Systems

As already mentioned, many software systems in the energy and environmental fields are still developed as monolithic applications. On the one hand monolithic systems can have advantages when it comes to simplicity in development and management of the code since there is only one code base. Also testing and deployment is easier in the sense that there is only one system and not many different systems that have to be tested and deployed separately. On the other hand there are a lot of downsides when a code base becomes larger and the development process therefore becomes more complex. It is also hard to scale a large monolithic system since it has to be deployed as one software system and therefore there is no scalability for a specific part of it. Another disadvantage of monolithic measurement systems is that they are hard to maintain. Adding new features or adopting new technologies is very complex since the whole code base might be affected. There are often no clear boundaries between the different parts of the software. Additionally, many architectures are not conceptualized using a design process or the architectures have been refactored and the changes have not been well documented [TL18]. This can lead to architectures that show bad smells and are hard to maintain.

1.3.2 P2: Heterogeneous Data Management

In the past, relational database management systems (RDBMS) proved to be a very good solution to manage data by storing different data sets in different databases and tables. As long as the data comes with no special requirements such as large volume or high velocity, RDBMS are preferred since they are well understood, well documented and easy to use. Quite the contrary applies if the data has more restrictive requirements such as time series data, especially when the time series data is larger or has to be ingested at a high velocity. For measurement data there exist many software projects [JP+17] that might fill this gap. Using a time series database (TSDB) means that additional information such as metadata or semantics often have to be stored in a separate database since most TSDBs only can store the actual measurements but no metadata. This quickly leads to a heterogeneous data management which is far more complex and requires new innovative concepts.

1.3.3 P3: Missing Semantic Information and Interpretation of Data

Similar to the TSDBs not allowing to store semantics or metadata for each time series, measurement systems in general often neglect the necessity to store semantics in order to identify and further also classify data. Since each domain has its own standards defining ontologies and semantics, it is important to link these to the actual data they describe and hence put in a specific context. Additionally, relations between different data sets are also often needed. The article [CG16] gives a good overview of the semantics (in the context of semantic web) and presents the importance of semantics. For example simulation results have to be stored in a way that each simulation run should be attached to the same experiment which defines for example its date and the used parameters. Nowadays this is often achieved by storing results in files and using the filename or the folder structure to create an implicit connection between them. However, this does not scale for larger, more complex systems. Additionally, the data is often only useful if it is interpreted correctly. This arises the need for data analysis for the heterogeneous data management mentioned in the previous problem.

1.3.4 P4: Missing Flexibility and Modularity in Frontends

The frontend and especially the visualization of data is very important since it is the main tool to provide information to the user. Current energy dashboard solutions are often statically defined and offer little flexibility. The configuration of such dashboards can take a lot of time since many frontends are manually implemented and configured. Moreover such control center dashboards are configured for one specific electrical grid. Adapting the grid to a new configuration is a lot of work since it has to be manually reconfigured. For control centers that display a simulated grid that can completely change within short time intervals due to a different simulation setup, the frontend has to be much more flexible and modular in order to adapt to new grid setups in a reasonable amount of time.

For environmental web pages, visualizations are often manually programmed using frontend libraries (e.g. Highcharts[3], Leaflet[4], etc.) or visualizations are even shown as static images. This leads to the downside that each visualization has to be developed or statically created.

[3]https://www.highcharts.com/
[4]https://leafletjs.com/

The latter has the additional disadvantage that it does not follow the responsive design principles for the Web.

Web pages consist often of different applications that are embedded on the same web presence. A problem that is common for such pages is that different domain experts are responsible for the applications and therefore it is complex to ensure that all of the applications follow the same guidelines and design principles.

1.4 Contributions

After discussing the identified scientific problems in the previous section, the following sections describe the main contributions to solving these problems. The contributions are named C1 - C4 respectively.

1.4.1 C1: Design of a Microservice-based Reference Architecture for Measurement Systems

The first contribution of the thesis is dedicated to the reference architecture itself and most importantly its design. The reference architecture is modeled using a technique that is known as Domain-driven Design [Ev03]. It allows to create microservice-based architectures using a proven workflow that ensures the breakdown of a concept into the right services. This workflow and its artifacts are described by Eric Evans in his book from 2003, which became very popular with the increase of cloud computing and the popularity of distributed systems. The main idea of the presented reference architecture is to use microservices in order to model different distributed and independent parts of the software. The concept of microservices solves the problems of a cumbersome monolithic software system. The independent services allow to develop and deploy them separately and therefore to scale them independently.

This contribution solves the first problem P1 by introducing the *measurement system* domain. Additionally C1 will give an overview of the final reference architecture with its different services.

1.4.2 C2: Distributed Data Management for Measurement Systems

The most important part of a modern measurement system is its data management. This contribution presents different services that all implement a small but specific part of the overall distributed measurement data management. As already presented in popular concepts such as the data grid [CF+00] each entity of data has to be somehow described by additional metadata. This is realized by complementing measurement services with different other services. These additional services focus on storing structured business data in order to describe measurements and the measuring system. This data management allows to have much more flexibility than a classic database containing all the information in one place. Additionally, the separation in different services allows an independent scaling and different specialized databases for each task. Furthermore the data services have to define relationships between the data.

This contribution solves P2 by presenting a heterogeneous data management concept including the mentioned services and their APIs that are realized using REST and streaming technologies. Since the semantics of the data, especially the description of the data models, are not included in the services and the data is not further analyzed in the data management services itself, the next contribution presents solutions for these challenges.

1.4.3 C3: Semantics and Analysis Service

When it comes to semantics, two problems have to be dealt with for distributed data management. First, the data has to be put into a semantic context. This additional semantic information, that is needed to describe the context of measurements and other assets in the measurement system, can be managed in an own microservice. The service ensures that the measurement system is semantically-aware of the data that it manages. The information contained in the system is self-describing without any expert knowledge that comes from the user-side or customer-side.

A second service that supports the main data management services is the analysis service that allows to perform data analysis tasks on the data managed by the different services defined in C2.

Both services solve the third problem P3 and complete the full picture of the reference architecture.

1.4.4 C4: Frontend Framework for Dashboard Applications

The last contribution deals with a frontend that allows to evaluate the reference architecture. Furthermore, the frontend presented in C4 solves the problem P4 with a modern frontend architecture. With the idea of separation and distribution for microservices in mind, the architecture of the frontend for modern measurement systems needs flexibility and modularity.

In order to solve the last problem, a frontend framework, based on the web components standard [W3C19] is presented which allows modular definition of frontend components and is highly customizable. In contrast to the already mentioned static and manually configured dashboards, the presented solution will allow to set up dashboards that are highly customizable and therefore easily adapt to new requirements. Those requirements might vary in data sources, frontend interactions or styling changes. One key advantage of the concept presented is that all of the mentioned customization can be done without programming. The concept can be integrated with the microservice backend without additional effort of the frontend user who configures it.

1.5 Limitations

Since the scientific range of microservice architectures for measurement systems is quite broad, the following sections will define limitations that are applied for the thesis.

1.5.1 Focus on Applied Energy and Environmental Computer Science

In general, the presented reference architecture is valid in a generic way for any computer science field that has needs for microservice-based measurement systems. Since the field is very broad, the present thesis focuses on applied energy and environmental computer science. This allows much more narrow analysis of related work, more precise examples and more specific evaluation. For the reference architecture this does not exclude the fact that it is a generic design and might also be applied in other fields.

1.5.2 Focus on the Dissemination of Data

Two main requirements for measurement systems are the gathering and management of data. For the presented reference architecture the focus is on the latter. Furthermore, not only the provision of data but also data visualization is of importance in the thesis. The gathering of data through sensors or simulations and the transformation of data are not discussed in detail. However, the data management also includes the modification of data that is ingested into the system.

1.5.3 Generic Platform for Different Requirements

A third limitation is that the requirements for the applications, that are implemented using the reference architecture, are not known before the software system is used. For many applications, all of the requirements are identified during the development process and therefore included in the implementation. When such an application is up and running, all of the features are available. In contrast to such applications, this thesis presents a reference architecture that describes a system which is developed in a generic way. The reason behind this is that the requirements are not necessarily known during the development process and therefore the platform has to cover a broad range of possible requirements. This can be achieved by designing a generic architecture that is highly customizable in order to adapt to new requirements during the runtime of the software.

1.6 Structure of the Thesis

The thesis is partitioned in the following nine chapters:

1. Introduction
2. Fundamentals
3. Requirements and Related Work
4. Design of a Microservice-based Reference Architecture for Measurement Systems
5. Distributed Data Management for Measurement Systems
6. Semantics and Analysis Service
7. Frontend Framework for Dashboard Applications
8. Evaluation
9. Conclusion

Chapter 1 motivates and presents the topic of the thesis. Subsequently, the problems that are pointed out in the motivation are clearly presented. Each problem is then addressed by a contribution and the goals of the thesis are defined.

Chapter 2 defines the most important terms and technologies in order to explain the fundamentals for the following sections.

Chapter 3 presents requirements for the reference architecture and analyzes relevant articles. The resulting gaps that no article tackled yet are the open problems that are solved by the four following contributions.

Chapter 4 is the first contribution and focuses on the general architecture. First, the reference architecture is modeled using the domain-driven design approach [Ev03]. The main artifacts are presented: the domain model and the context map of the microservice architecture. Furthermore an overview of the resulting reference architecture is given with services for data management, semantics and a frontend concept.

In Chapter 5 the data management aspects of the reference architecture are illustrated and discussed in detail as a second contribution. The focus lies not only on measurement data but also metadata and other accompanying data.

The third contribution presented in Chapter 6 consists of the description of semantic services complementing the data management. These services are used by the data services to get additional information for the data that they manage and to validate new data sets. Additionally, the semantic services provide general semantics and contextual information for frontend applications. The services presented are: the semantics service which manages semantic descriptions of data objects and the analysis service that allows performing data analysis jobs providing statistical information about the data in the different services.

Chapter 7 presents the frontend concept which completes the whole reference architecture. It consists of the presentation of modular and customizable frontend components and a dashboard concept that combines these components and allows for additional configuration in order to create a flexible and semantics-aware dashboard.

After the four contributions, Chapter 8 evaluates the reference architecture with the help of the dashboard application presented in Chapter 7. On the one hand, a theoretical evaluation is discussed, and on the other hand, prototypes are evaluated.

The last chapter (Chapter 9) concludes the thesis with a summary of the main solutions for the scientific challenges and an outlook for possible further scientific investigations.

2 Fundamentals

This chapter describes the fundamental concepts and technologies in order to have a better understanding for the rest of the thesis. First, the main concepts of microservices and microservice architectures are presented. Second, further technologies that cover the different contributions of the thesis are presented.

2.1 Microservice-based Architecture

The proposed reference architecture is defined as a microservice-based architecture and therefore, the term microservice itself and other concepts around it are described.

2.2 Microservice

The first term that is the most important for the thesis is *microservice*. It originates from a discussion at a workshop of software architects near Venice in May 2011 [FL14]. Martin Fowler and James Lewis describe the term *microservice architecture* in an article that is a main reference for the community [FL14]. In the following a definition by Martin Fowler and James Lewis is given.

Definition 1 *In short, the microservice architectural style is an approach to developing a single application as a suite of small services, each running in its own process and communicating with lightweight mechanisms, often an HTTP resource API. These services are built around business capabilities and independently deployable by fully automated deployment machinery. There is a bare minimum of centralized management of these services, which may be written in different programming languages and use different data storage technologies.[FL14]*

As stated in [FM+17], [AA+16] and [PC+16] the term "microservice architecture" became popular in the scientific community in the year 2015. The popularity increased further in 2016. The three cited articles give a good overview of the current state-of-the-art of microservices.

They show that current articles discuss a broad range of topics around microservices. Key terms extracted from most of the papers are scalability, independence and modularity and therefore show the relevance of this architecture for the presented platform. Additionally, the book “Building Microservices” by Sam Newman [Ne15] is a detailed reference for the microservice architecture and the environment around it.

2.2.1 Microservice Architecture vs. Monolithic Architecture

As stated in the introduction there is a shift from classical monolithic architectures to microservice architectures. Even though the latter is not always the best choice, many complex systems can profit from this modern architectural style. The main differences are depicted in detail in [Ba17]. Table 2.1 shows an overview of a comparison between both software architecture styles.

The comparison of both architectural styles shows that, for complex systems with a large code base, the microservice architecture reduces the complexity and allows modular development, testing, deployment and maintenance of services. The downside of the microservice architecture is that the distributed deployment of each service raises the need for a smart infrastructure that keeps track of the services and supports the communication between the services in order to reply to complex frontend requests. The background of such an infrastructure is discussed in the following sections.

2.2.2 Microservice Infrastructure

In order to benefit from all the advantages that a microservice architecture brings to the table, a microservice infrastructure has to be set up. This infrastructure has to fulfill different requirements. Figure 2.1 shows a general microservice reference architecture with the main components that such a system defines.

Category	**Monolithic Architecture**	**Microservices Architecture**
Code	A single code base for the entire software.	Each microservice has its own code base.
Understandability	Often confusing and hard to maintain.	Much better readability and much easier to maintain.
Deployment	Complex deployments with maintenance windows and scheduled downtimes.	Simple deployment as each microservice can be deployed individually, with minimal if not zero downtime.
Programming Language	Typically entirely developed in one programming language.	Each microservice can be developed in a different programming language.
Scaling	Requires you to scale the entire application even though bottlenecks are localized.	Enables you to scale bottle-necked services without scaling the entire application.
Documentation	Grown documentation that becomes harder to read the older the monolith is.	Simple documentation due to small code base and feature set. Standardized documentation of API interface (e.g. OpenAPI[1]).
Testing	For large software systems testing becomes very hard. Code coverage is nearly impossible.	Microservices can be tested automatically using a DevOps pipeline. Tests are easier to maintain, since the codebase is small.
Complexity for small code base	The architecture performs good if the software has a rather small code base because there is no overhead.	The microservice architecture comes with a lot of infrastructure that enables the operation of the services. For small software systems this architecture is therefore not suited.

Table 2.1: Comparison of Monolithic and Microservice Architectures (extension of [Ba17])

The system is divided in the inner architecture that describes the microservices themselves. This part should mainly contain the interface definition, its business logic and possible

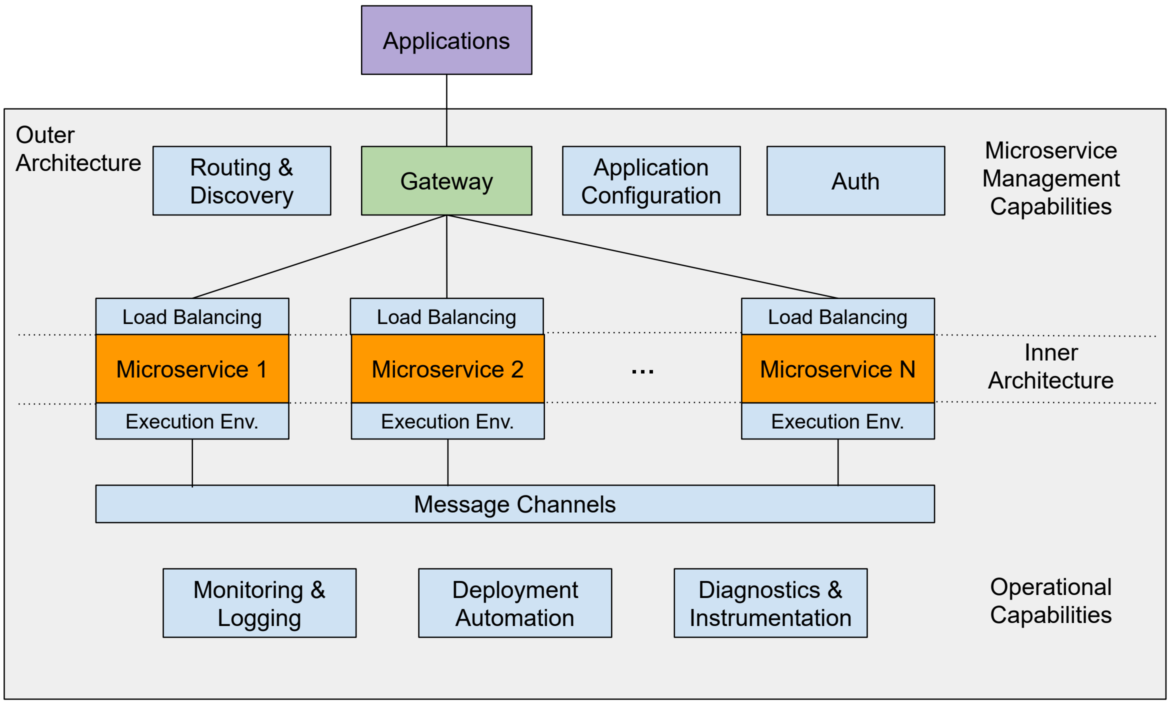


Figure 2.1: General Microservice Reference Architecture

additional software that is needed e.g. databases. The more interesting part for this section is the outer architecture. It consists of two main aspects: the microservice management capabilities and operational capabilities. The first aspect defines components that help to keep track of services also known as service discovery [An16], feature load balancing [TP+17], circuit breaking [Ma15] and authorization [CP+15, EM+17]. Additionally, the communication with external services and users is ensured by featuring a gateway that usually allows synchronized HTTP(S) requests which are routed to the right service using service discovery tools. For the internal communication each microservice can also use the HTTP(S) API of the other services, but this means that there is a synchronized connection between both services, which introduces an additional dependency between these services. Such dependencies contradict the microservice principles that request low coupling between services. Therefore, each microservice is connected to a messaging system that allows services to communicate internally using asynchronous message channels. The second aspect of the outer architecture consists of the operational capabilities. These define tools that help to monitor services [BM17] and manage logs at a central place. Also the deployment and orchestration of services has to be supported by the infrastructure in order to have a short deployment time and automated processes that make sure that each service operates as expected.

2.2.3 Microservice Architecture Design

In order to define the concepts for the architecture and the operation of microservices, first, the architecture has to be designed. The most important part of the design of a microservice architecture is to properly define the different services and their responsibilities. A very popular process to help with the design of such an architecture is called *Domain-Driven Design* (DDD). DDD has been defined in detail by Eric Evans in his book with the title *Domain-Driven Design: Tackling Complexity in the Heart of Software* [Ev03]. The idea is to involve the customers and/or users of the software in the development process in order to identify the terms and common language that defines the system that the software implements: the *domain*. Once defined, the domain can be cut into different bounded contexts that each define a separated part of the whole domain. The different bounded contexts and the relations between them are described in a *Context Map* which is one of the resulting artifacts of DDD. A second artifact that defines the domain in detail is the *domain model*. The next step is to define the technical details behind each bounded context which define the use of databases, data formats, interfaces, etc.

Having seen the different microservice concepts that are relevant for this thesis, the next section will present further concepts and technologies that help tackle the different scientific challenges.

2.3 Communication Concepts

In the following sections, two different communication concepts are presented. First, the concept of RESTful interfaces is discussed. Second, different streaming technologies are presented.

2.3.1 REST (Representational State Transfer)

The most important technology behind microservices is REST. It is the technology that is used to realize most of the interfaces of modern microservices architectures. The term *Representational State Transfer* was introduced by Roy Fielding in his dissertation [Fi00]. Nowadays, it is used in a wide range of applications since it defines a lightweight communication for web applications and services.

A REST API provides an interface that allows operating on resources using the HTTP protocol. A lot of frameworks support REST client and/or server functionalities. Furthermore, REST APIs are easy to document. A RESTful service provides basic CRUD (Create, Read, Update and Delete) functionalities by instrumenting the HTTP methods as follows. The GET method is used to request one or multiple resources. The PATCH method is used for partial updates of a resource and the PUT method is used to fully replace a resource. The POST method is used to create a new resource and returns the location of the new resource as a header field. Lastly, the DELETE method fully deletes a resource. Furthermore, the POST method can be used for more complex operations for example as trigger starting the computation of a data analysis job.

The main disadvantage of RESTful communication is that it is a synchronous request/response communication. This means that the client sends a request and waits for the response. Since many applications are dependent on high asynchronous behavior and since microservices should not be coupled synchronously, a REST API is often not the only interface a service provides.

2.3.2 Streaming

The asynchronous counterpart of RESTful communication is streaming. It allows a much more flexible form of communication. There are three types of streaming technologies relevant for the present thesis. First, the communication of microservices with frontend web-based applications. The most popular technologie for this use case is the WebSocket [WS] technology. It allows the browser to open a socket connection to the backend and send data bidirectionally. The second type of streaming are so called message servers. They often implement the publish/subscribe pattern [Ta12] and allow to asynchronously publish data to a channel or subscribe for such a channel and receive published data. Examples for such message server protocols are AMQP and MQTT [DC+19]. A software like RabbitMQ[2] can handle both protocols. The last type of streaming are RPC [Bl92] technologies. These are continuously improved and a modern protocol called gRPC[3] developed by Google uses the HTTP/2 protocol to allow bidirectional streaming of data using the underlying Protocol Buffer [ProtoBuf] message format.

2.4 Container Virtualization and Orchestration

The orchestration of services is a challenge for many companies and therefore many different tools were developed in the past years. An established technology for the operation of

[2]https://www.rabbitmq.com/
[3]https://grpc.io/

services is container virtualization. With Docker [JN+16], as the leading software for container virtualization, this new technology has spread in many different IT domains. Especially, the microservice domain can profit from this technology. Each microservice can be deployed and operated as a separate container. This makes it fairly easy to start multiple services of the same kind in order to sustain more load from client requests. Docker provides tools for building container images and testing the deployment locally but it is missing a production-ready container orchestration platform that allows to deploy containers on multiple nodes forming a cluster. An example for such a platform that fulfills those requirements is Kubernetes [Be14], which is the platform that gets the most attention at the moment.

2.5 Types of Different Data

The thesis proposes a data centric architecture that defines features for specific types of data. The three important categories are measurement data, master data and digital assets. These will be defined in the following in order to have a clear understanding for the rest of the thesis.

2.5.1 Measurement Data

First, the concept of a time series is formally defined. Let T be the set of all possible timestamps ordered such as $t_i < t_{i+1} \forall t_i \in T \land i \in \mathbb{N}$. V_j is the set of possible values for index j. In that case a time series can be defined by a relation f.

$$f : T \mapsto V_1 \times ... \times V_n, n \in \mathbb{N}$$

The relation f is an observation of one or more values at a specific point in time. The time series is called univariate in the case of $n = 1$ and multivariate in the case of $n > 1$. Furthermore, if the different timestamps are all equidistant $t_{i+1} - t_i = t_{i+2} - t_{i+1} \forall t_i \in T$, the time series is of type *sampled values* which is a term commonly used in the energy field. Sampled values are often used when many values are recorded in a short amount of time. An example for such a measurement is the data that is recorded by a phasor measurement unit (PMU) that observes multiple important metrics of an electrical line.

2.5.2 Master Data

Besides the actual measurement data that a measurement system has to manage, also the business objects that are important in the application's domain have to be stored and operated on. These objects are called *master data* of an application. They represent data that are structured and strictly typed. In the following a master data object is defined. Let D be the set of the abstract definition of a business object. Let I be the set of possible identifiers for each business object definition d. Each business object definition has multiple master data instances that are each identified by a specific identifier. Additionally, let P be the key-value pairs for each business object instance.

$$f : \{(d,i) \mid d \in D \wedge i \in I\} \mapsto \{(p_1,...,p_n) \mid p_j \in P_{(d,i)} \wedge n \in \mathbb{N}\}$$

Then, f is the relation that maps a specific business object definition and identifier to a set of key-value pairs. Note that the key-value pairs depend on the selected business object instance. Next, P is defined in more detail. Let K_d be the set of possible keys that the business object has. Each key has to be unique and can therefore not be defined more than once. Additionally, W_{K_d} is the set of possible values for this key.

$$P_{(d,i)} = \{(k_d, w_{(d,i)}) \mid k_d \in K_d \wedge k_m \neq k_o, m \neq o, 1 \leq m \leq n, 1 \leq o \leq n \wedge w_{k_d} \in W_{K_d}\}$$

Furthermore, the set W is defined. Let V be the set of possible values according to the type t of possible data types in T. Possible data types are in this case at least the following: Boolean, Integer, Float, String, Array, Object. If the type is one of the latter two, v is defined as a new subset of values.

$$W = \{(v,t) \in V \times T\}$$

This allows a generic description of a master data object without a specific application context. The application defines the business object definitions. These definitions are also called schemas in the following.

2.5.3 Digital Asset

A digital asset is a document or file that can be considered an asset for an application. The structure or content of the file is not important. Therefore the only relevant attribute of a digital asset is its name and file path. Some examples for digital assets are the following:

- image file
- audio file
- word document
- PDF document
- simulation model

Let P be the set of possible file paths that identify a digital asset and let A be the set of those digital assets. The relation f defines a mapping between a file path and a digital asset.

$$f : P \mapsto A$$

2.6 Semantics

Semantics are an important part of IT and always needed when a system has to understand the interpretation of the data. Often this concept can be realized with the help of an ontology. Such an ontology is a definition of concepts and entities that define the semantics of a specific domain. This definition includes schemas of master data objects, their properties and the relations between them. Such an ontology allows applications to understand the schemas of master data objects and therefore the semantics of their properties and how they are related to each other. More information on semantics and especially semantic web can be found in [CG16].

The present thesis will not go into that much depth and the focus lies on the definition of schemas. Such schemas can be described using different strategies. The basics of schemas are defined in the following section.

2.6.1 Schema

A schema is a formal description of a document or data entity (in this case master data object). The description includes the structure, data types and human readable descriptions of the different properties of the schema. The most common schema formats are XML Schema that describes an XML document and JSON schema that describes a JSON document. These schemas are tailored to the specific document formats. They allow applications to understand the semantics of such a document. Since JSON and XML are the most common formats for building RESTful APIs, the respective schemas are also the most popular.

2.7 Frontend Concept

The main frontend concept needed for the present thesis is the web components standard. It includes different separate standards that are needed for the definition of web components. They are specified by the World Wide Web Consortium [W3CWebComp]. In the following they are defined.

2.7.1 Custom Element

Definition 2 *This specification describes the method for enabling the author to define and use new types of DOM elements in a document.*

These elements can be used as any other DOM element. The content and behavior is fully customizable. The only condition is that the name of a new element *must contain a U+002D HYPHEN-MINUS character, and must not contain any uppercase ASCII letters* [W3CWebComp]. Furthermore, a few names are blacklisted because they are already defined in certain standards (e.g. font-face in the SVG specification). The biggest advantage of custom elements is that the developer can hide complex content behind a simple HTML element.

2.7.2 Shadow Dom

Definition 3 *This specification describes a method of combining multiple DOM trees into one hierarchy and how these trees interact with each other within a document, thus enabling better composition of the DOM.*

The shadow DOM feature allows a DOM tree to be attached as child to another DOM tree. The <content> element can be used as an insertion point for such a shadow DOM tree. Shadow DOM structures should be hidden from the end user to follow the principle of “information hiding” for web components.

2.7.3 HTML Templates

Definition 4 *This specification describes a method for declaring inert DOM subtrees in HTML and manipulating them to instantiate document fragments with identical contents.*

HTML templates can be used to define the shadow DOM of a custom element. The template can be reused every time a custom element of this definition is imported and added to the document. These standards can be used with plain JavaScript, however there exist multiple frameworks that help efficiently implementing web components. One example for such a framework is Polymer[4].

[4]https://www.polymer-project.org/

3 Requirements and Related Work

In order to evaluate the related work and the proposed reference architecture in the present thesis, the requirements for the reference architecture will be deduced from the analysis of two domain-specific applications and discussed in the first part of this chapter. Second, the most relevant publications are presented and analyzed with respect to these requirements. Finally, resulting gaps are identified.

3.1 Requirement Analysis of Domain-specific Applications

In order to define the generic requirements for the reference architecture, an analysis of two different applications is presented. This will show similarities but also differences which will lead to the generic requirements in Section 3.2. The scientific fields of both applications (energy and environment) are presented in Section 1.2 already. Therefore, this section focuses on the requirements.

3.1.1 Energy Application

A typical energy application with need for a measurement system is a control center software. As already mentioned in the introduction, the most important part of a control center application are visualizations on large control center display walls.

Therefore, the first requirement is the visualization of data. An example of such a visualization is a single line diagram of the electrical grid that shows the current network's condition, the status of different generators or a map visualization of the electrical grid. To create such visualizations, first of all, the data has to be gathered and stored efficiently. Therefore, a few more requirements are needed.

Second, the measurement data that has to be stored is not necessarily of the same type. In the following, two examples are presented. The first example is data that is monitored on a constant basis (e.g. one recording per second). This means that the data is stored as a stream of values at a low or medium velocity. This is a very basic example that is present in nearly every measurement system. The second example is data that is recorded using a PMU (Phasor Measurement Unit). Such a device only records data on special occasions for example when an event occurs such as a failure of a system or an alarm from a generator in the electrical grid. However, it records the data at a very high velocity for a short amount of time (up to 10.000 values per second). This data can get very large on the one hand and has very fine grained resolution on the other hand. Therefore a measurement system has to be able to cope with such situations as well.

Another requirement is the management of master data. This includes all the data describing the different business objects in the electrical grid including the measurement devices. This data is important in order to have meaningful visualizations and to know which devices have recorded which measurement. Furthermore, an energy system usually has not only one PMU or sensor but many at different points in the grid. This leads to the need for a scalable system.

A last, very important, aspect of control center applications is the need for a graphical user interface operable without any IT knowledge since mostly electrical engineers without deeper IT knowledge will operate the control center. Therefore, the last requirement defines the need for a good user experience and a supporting graphical user interface. The following list shows an overview of the core requirements:

- visualization of data
- management of different types of measurement data
- management of master data
- scalable system
- user friendly graphical user interface

3.1.2 Environmental Application

The second area that is analyzed in this thesis is the environmental field. The application that is looked at is a web site that is meant to inform citizens about their environment. The application ranges from information about air pollution to water levels of rivers and radioactive immissions.

Similar to the previous application, the citizen wants to see different visualizations showing the information mentioned before. The measurement data of this application is more simple since the data is usually not recorded at a high velocity and there are no complex types of measurement data besides the normal basic form also described in the energy application. A difference however is the need for correction of measurement data. The problem with sensors or measurement networks is that sometimes data has bad quality or is erroneous. In that case the data should be corrected if possible. For the energy application this is no big problem since the devices are much more expensive and therefore most of the time calibrated in a way that errors lie in very small bands.

Another similar requirement is the management of master data since the citizen wants to see information of the measurement stations or static information of their environment (the location and shape of a lake for example). Furthermore, it is also necessary to analyze measurements and master data since the citizen might want to know the closest forest or lake from his/her current location. This shows the need for an analysis of all the forests and lakes and a classification of the results.

The last requirement is also similar to the energy application, even if the environmental measurement system does not have to store and manage that much data it is still important for the end user to have a fast and responsive web site, therefore scalability is important.

The following list shows an overview of the core requirements for this second application:

- web-based data visualization
- management of measurement data
- correction of measurement data
- management of master data
- analysis of measurements and master data
- scalable system

3.2 Requirements for the Proposed Reference Architecture

The requirements for the microservice-based reference architecture for semantics-aware measurement systems are derived from the previously presented domain-specific requirements. Since both representative applications are defining different requirements from different domains, this subsection unifies the requirements in order to get the generic requirements the reference architecture has to fulfill.

3.2.1 Description of the Requirements

Each requirement is identified by a number in order to reference them in the evaluation of the related work and in the following chapters of the thesis.

R1 Architecture Based on Microservices with RESTful and Streaming APIs

The proposed reference architecture is supposed to be based on the architectural style of microservices. One of the most important parts of microservices is the API which is typically realized using RESTful interfaces or streaming technologies. Both concepts are introduced in Chapter 2. There are many ways to design an API, therefore this requirement aims to ensure that best practices for REST API design [GG+16] and microservices in general are used for the reference architecture.

R2 Scalability

One advantage of microservice-based architectures is that they are easy to scale. As identified in the domain-specific requirements this is a central requirement. However, the architecture only scales well if the services as well as the underlying infrastructure are designed in that way. Modularity and low coupling are important in order to scale different services independently from each other which is crucial for an infrastructure that has uneven load across the services. Since there exist multiple ways of scaling, for example vertical and horizontal scaling [Mo16], or the three axes of the *Scale Cube* from [AF15], the right concept in order to decide between them or combine them is important. Considering a microservice architecture, it is not only necessary that the database systems scale but the whole infrastructure has to be scalable including databases, microservices and auxiliary services (e.g. gateway, IAM, service discovery).

R3 Generic and Reusable Services

Since the requirements of specific energy or environmental applications can be very versatile (as seen in Section 3.1) and are not known before the design of the reference architecture (see Section 1.5), the design has to be generic in order to handle a broad range of possible requirements within the scope of the present thesis. Hence to design generic services, they have to be reusable for different use cases. Otherwise each use case has to be implemented separately with customized functionality.

R4 Various Measurement Types

As already seen in Chapter 2, the term time series which is closely linked to the term measurement is not only restricted to one specific type of data that has a temporal dimension. Multiple time series types can be defined for different types of measurements. A simple temperature measurement for example might be represented by a univariate time series that only consists of the timestamp and temperature value. A phase voltage measurement for all three phases might be implemented as multivariate time series such that each timestamp is assigned to all three phase voltage measurements. Furthermore, a more complex example are measurements that are recorded in very high frequency that are usually gathered in equidistant time intervals. It is not suitable to treat this data as conventional time series since the data volume is too large and might be stored in a whole different way than other measurement types. All different types have to be supported in an efficient and generic way.

R5 Master Data

Besides the mentioned measurement data, the reference architecture has to provide the ability to manage master data objects. These objects usually describe the different business objects in the application's context. In the field of energy this might be for example the descriptions of different energy consumers and generators. Such data might also be related to the different measurements. Each master data object has to be identified uniquely in the microservices. Additionally, the API for querying such objects has to provide the ability to filter the objects following specific rules (e.g. a query that only fetches energy generators that use renewable resources). To provide such filters, the corresponding service has to know the structure of each master data object. The basic requirements for the API on the other hand define basic CRUD operations. In the case of a domain-specific application such objects can be seen as domain objects of the architecture and can be defined in the design phase. However, since the proposed reference architecture will be designed in a generic way, this domain knowledge has to be treated as data in order to maintain sufficient flexibility. Data can be seen as user input of the measurement system. Therefore this requirement is important in order to obtain a generic way of defining master data objects.

R6 Semantics

As mentioned in the previous requirement, the management of master data has to be aware of each object's structure in order to provide a useful API for operating on the data. This additional information can be provided by semantic metadata in the form of schemas. These schemas have to define the structure of different categories of master data objects (can be seen as a class definition in object-oriented programming). This includes the relations between different categories and the data types of the properties that belong to a master data object category. Besides the data structure, also semantic descriptions of master data objects expressed in natural language are part of the schemas. Both types of information help users (descriptions in natural language) and algorithms (structure and data types) to understand the managed data.

Also the schemas have to be managed, therefore a respective service is needed. Similar to the previous requirements, this service's API has to feature CRUD operations and advanced filter operations. The architecture benefits from the semantic context in a way that the software understands the data and can associate each master data object with a corresponding schema describing the object. A frontend user interface can use this information to create a generic view component, based on this schema. The user is then served with the view that belongs to this schema without the explicit implementation of each of these views.

R7 Data Analysis

For now, the requirements focussed on the persistence and the management of data. However, in order to get more insight into the data, applications need to perform data analysis tasks. Especially, data analysis for measurements is needed since they contain the most information in a measurement system. However, also other data besides measurements might benefit from data analysis. In order to fulfill this requirement, a new component has to be introduced in the reference architecture that allows to perform arbitrarily data analysis jobs. The concepts have to be aware of the fact that such jobs can take a larger amount of time than blocking RESTful requests.

R8 Semantics-aware Frontend

Most of the preceding requirements are backend requirements. This last requirement focuses on the frontend. A generic frontend can use the different data components and the semantics in order to visualize the data. Additionally, the user can be supported by the semantic contexts and results of data analysis jobs. Since the frontend is, similar to the backend, not application-specific but has to be designed in a generic way, the frontend architecture also has to be defined modular and reusable. Furthermore, as already mentioned in the first section, the frontend has to be web-based in order to comply with a broad range of devices.

R9 List of all Requirements

The following table (3.1) summarizes the eight requirements presented in this section. This analysis defines the fundamental gaps that will be solved in the contributions C1 to C4.

Number	Title
R1	Architecture Based on Microservices with RESTful and Streaming APIs
R2	Scalability
R3	Generic and Reusable Services
R4	Various Measurement Types
R5	Master Data
R6	Semantics
R7	Data Analysis
R8	Semantics-aware Frontend

Table 3.1: List of requirements

3.3 Related Work

In the following, related work for the thesis is analyzed and assessed using the different requirements defined in the last section. Among a larger literature research, only the most relevant articles were selected.

3.3.1 A Software Architecture-based Framework for Highly Distributed and Data Intensive Scientific Applications [MC+06]

The article [MC+06] presents a framework called *OODT* (Object-Oriented Data Technology). Nine challenges are addressed with the framework: Complexity, Heterogeneity, Location Transparency, Autonomy, Dynamism, Scalability, Distribution, Decentralization and Performance. Based on those challenges, the authors describe four guiding principles that were used to design the architecture. The first is *division of labor* which is also one of the main principles for microservice-based architectures. The second principle is *technology independence*, which is common for the separation of architectures from the implementation. Furthermore, this also leads to an architecture where each component can be implemented using a different programming language/technology as long as the interfaces of the different components are mutually compatible. The third guiding principle is distinguishing between metadata and actual data, which is also defined in the article by Chervenak et al. [CF+00].

Finally, the last principle describes the separation of the data models from the software. At first glance this contradicts domain driven design since the software can not be implemented following a specific domain which would lead to specific data models that are implemented in software. However, in the context of this thesis, this fits perfectly with the third limitation (see 1.5.3). The first three principles are closely related to microservice-based architectures and therefore the architecture shows characteristics of the service-oriented architectural style. The underlying infrastructure of the framework is designed in a distributed way. As depicted in Figure 3.1, the framework runs on different machines in so called *Sandboxes* which can be compared to virtual machines nowadays. The concept of the architecture is the *Product* which is defined as *a unit of data*. It has two components in the architecture: the *Product Server*, which is included in distributed instances of OODT, and the *Product Client* that is at the users site. Both communicate over the messaging layer using HTTP. The next component is the equivalent for handling metadata called *Profile Client/Server*. The managed metadata is divided in three parts: *Housekeeping Information* (including *ID*, *Last Modified* ...), *Resource Information* (*Title*, *Author* ...) and *Domain-Specific Information*. The third component is the *Query Client/Server* that is responsible for searching in the product and profile server. The last component that is not as relevant for this thesis is the *Catalog and Archive Client/Server* that are responsible for ingestion. However this is excluded from focus by the second limitation (see 1.5).

Analysis

The architecture is designed with many principles of microservice architectures. Additionally, the communication is implemented using HTTP. Therefore, the first requirement (R1) is partially met. The infrastructure is designed in a distributed way that allows the framework to scale (R2). The article does not mention support for measurements, however the product server can manage any kind of data in a generic way (R3). The product server component does only support one kind of data (defined as product) therefore the support of different types of measurements is not possible in a way that the architecture benefits from it (R4). The management of master data is possible when master data are mapped to a product (R5). The data requirements of the thesis are not fully covered, since the system only has one component handling all of the data and not being able to have custom APIs for different types of data. The profile server fulfills the need to add a semantic context to the data (R6). The article does not mention data analysis or a visualization frontend (R7 + R8).

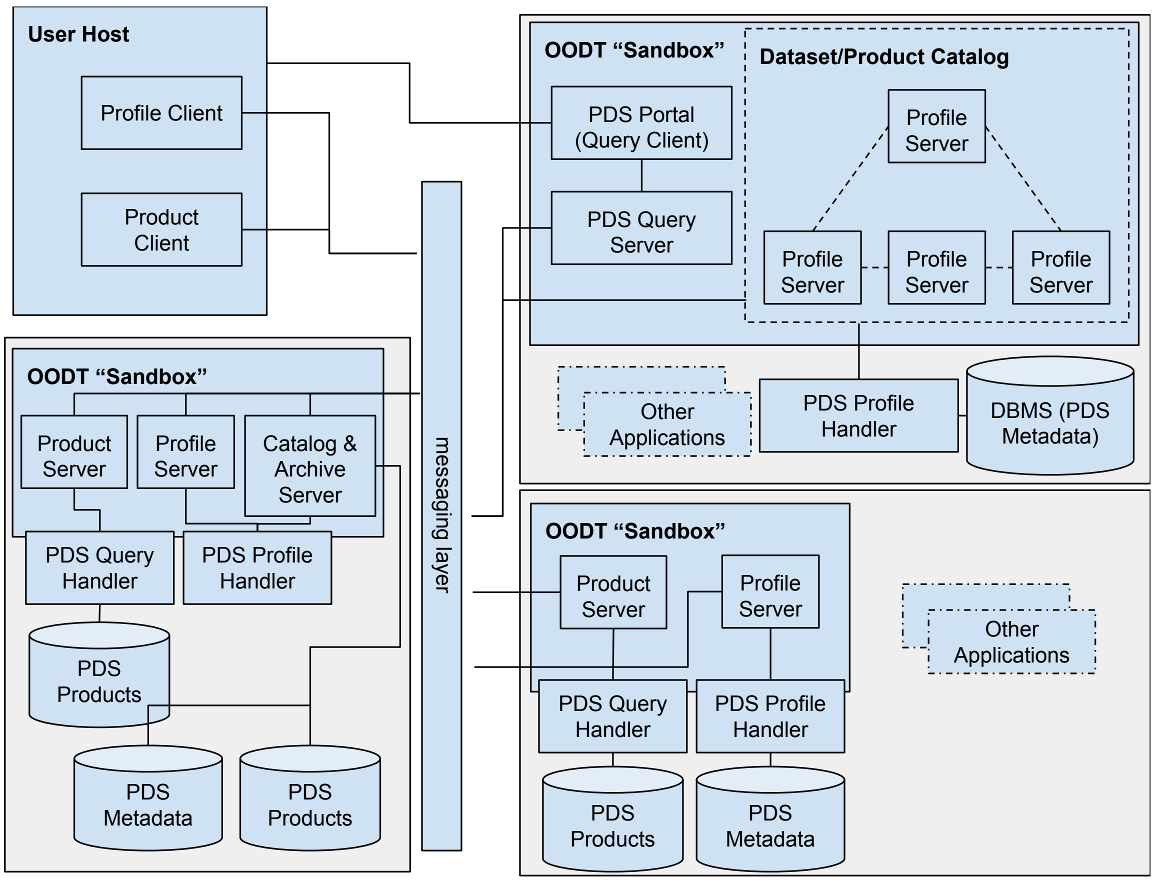


Figure 3.1: Overview of the Framework's Architecture from [MC+06]

3.3.2 Experience on a Microservice-Based Reference Architecture for Measurement Systems [VL+14]

A second architecture is presented in [VL+14]. This article focuses on a measurement system defined by a microservice-based architecture. The architecture is designed fulfilling the requirements presented in table 3.2.

1	Integration of heterogeneous systems to provide the basis for different metrics and visualizations.
2	Fast and up-to-date recognition and update of the metrics on a change in an integrated system.
3	Clear separation of system integration, calculation and visualization.
4	Be robust to avoid a complete system failure if a small part of the system fails. Additionally, the failure of the infrastructure should not result in a failure of the integrated systems.
5	Be flexible to support evolution of metrics, integrated systems, and visualizations.
6	Offer dedicated operation and development support.

Table 3.2: List of requirements from [VL+14]

The architecture defines a way to heterogeneously link different data sources to the system using so called data adapters. This concept allows the architecture to define data sources in a very generic way, since each adapter is an own piece of software that is closely attached to the data source and can convert it into a format that is supported by the system. The disadvantage is that a new data adapter has to be developed for each new type of data source. The architecture defines a dashboard and its communication with the backend, however the frontend itself is not further specified. The focus of the article is on the data communication from the data source up to the visualization.

Analysis

The architecture is based on microservices. The dashboard uses the REST API of the metric kernels (R1). The article does not mention scalability, even though the architecture defines modular microservices (R2). The architecture describes a measurement system that is generic and very flexible (R3). Multiple measurements and data sources are supported, however, all measurements are stored in the same place (see *Measurement Cache*). Therefore, the persistence does not support different types of measurements as defined in R4. The article

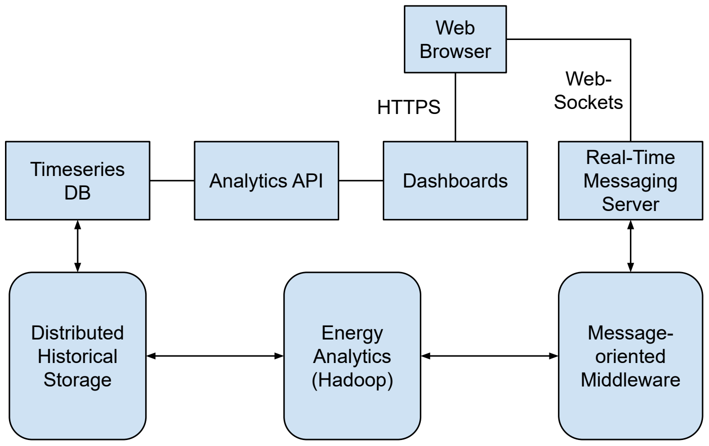

Web Browser
HTTPS
Web-Sockets
Timeseries DB
Analytics API
Dashboards
Real-Time Messaging Server
Distributed Historical Storage
Energy Analytics (Hadoop)
Message-oriented Middleware

Figure 3.2: Simplified Overview of the Energy Cloud architecture from [SC+14]

does not mention the management of master data or additional semantic information (R5, R6). The publication describes metric kernels that also perform calculations. Therefore, the requirement of data analysis is fulfilled (R7). Finally, a visualization of data is described. However, there is no mention of semantics used in the frontend (R8).

3.3.3 Energy Cloud: Real-time Cloud-native Energy Management System to Monitor and Analyze Energy Consumption in Multiple Industrial Sites [SC+14]

In the next article [SC+14] an architecture with the focus on energy measurement data is presented. First, the article presents different industry challenges and how cloud computing can solve them. Then, multiple energy management systems (EMS) are compared and analyzed. Figure 3.2 shows an overview of the architecture. It uses the lambda architecture [Ma13] as basis and it's optimized to process time series data. In contrast to the previous articles, it is no microservice-based architecture. The system includes storage for time series data, batch and real-time processing of data, ingestion of energy data and a visualization component. The implementation makes use of the Hadoop ecosystem[1] which is a broad range of open source tools for the storage and processing of big data.

Analysis

The architecture presented in the article uses REST APIs and WebSockets to communicate with the client. However, it is no microservice architecture, therefore the first requirement is

[1]https://hadoop.apache.org/

only partially fulfilled (R1). The infrastructure that is used is highly scalable since scalability is one of Hadoop's main features (R2). The architecture is defined in a generic way since most of the components can be used generically and independent from a specific application or data. However, the analysis is quite specific for energy management. Therefore, R3 is only partially fulfilled. R4 to R6 are not satisfied since no data management is specified other than time series data. Data analysis is the main focus of the publication and therefore fulfills requirement R7. The system has a visualization dashboard included, however, does not mention the use of any semantics (R8).

3.3.4 Next-Generation, Data Centric and End-to-End IoT Architecture Based on Microservices [DB18]

The next article presents an IoT architecture that is based on microservices. First, the article presents the well known IoT challenges that are scalability, interoperability, and integration of heterogeneous devices and protocols. In order to tackle these challenges the article proposes microservices and micro data as core concepts. Micro data is data exchanged among services and devices. Figure 3.3 shows a high-level overview of the components included in the architecture. The two main components are the microservices deployed on a cloud infrastructure and the microservices deployed on the edge server. The main interfaces of the architecture are realized using REST. However, the communication between cloud and edge server is also implemented using MQTT and CoAP protocols. The data exchange between the different services of the cloud backend are defined using the semantic data format JSON schema. This allows to ensure that the micro data that is exchanged follows a specific format. This allows every service to interpret the data in a uniform way.

Analysis

The article presents a microservice-based architecture that features RESTful APIs and uses different other communication protocols. Therefore requirement R1 is met. The scalability (R2) of the system is a challenge that the article presented as such and uses the architecture shown in 3.3 to solve it. Next, requirement R3 is also met since the architecture has does not define application-specific components and seems therefore reusable and generic. The Next two requirements concerning the support of different measurements types and the support for master data management is not mentioned explicitly. The concept of *micro data* might be used to build such a system. However, both requirements are not fulfilled since the publication does not mention these measures. In contrast to the last two requirements, R6 is

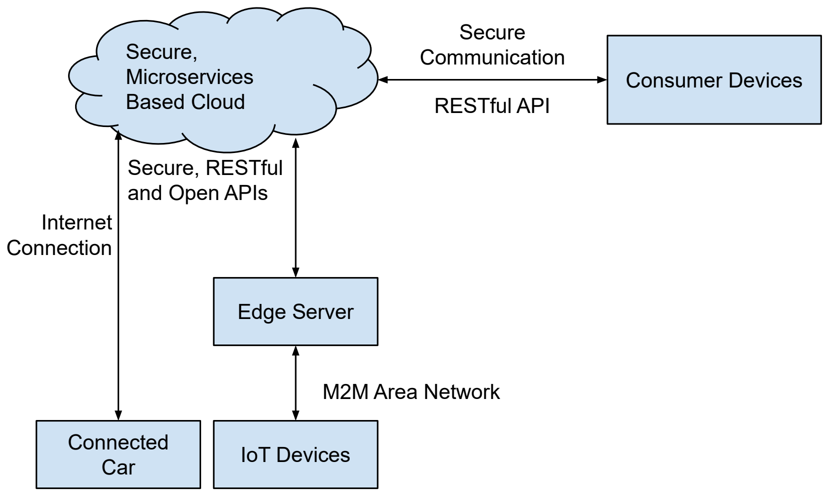


Figure 3.3: Overview of the Architecture form [DB18]

met since the article presents the usage of JSON schema in order to semantically define the micro data. Since the article does not mention data analysis or frontend visualizations, the last two requirement (R7 and R8) are not met.

3.3.5 Reference architecture for open, maintainable and secure software for the operation of energy networks [GM+17]

The reference architecture presented in [GM+17] describes a service-oriented architecture (shown in Figure 3.4) that defines an enterprise service bus as internal communication channel and REST APIs for external communication between clients and the services. The architecture is separated in two parts: *core modules* defining general services such as authentication and monitoring and *domain modules* describing for example services for data management and data analysis. This separation is closely related to the general microservice architecture presented in Chapter 2. The core modules are similar to the outer architecture and the domain modules to the inner architecture respectively. The architecture defines different data management services for different types of data, especially the management of measurements. In order to store semantic information about the topology of an energy network a separate service is designed.

Analysis

The article does not mention that the architecture is microservice-based. However, it is very close to such an architecture. Furthermore, each service has a REST API (R1). Similar to

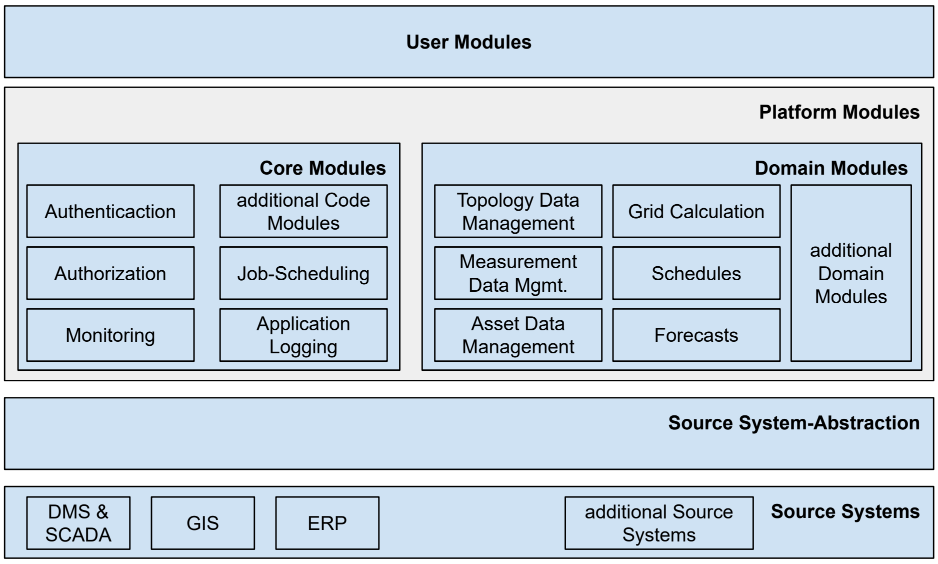


Figure 3.4: Multilayer Architecture from [GM+17]

R1, there is no mention of a scalable infrastructure for the architecture (R2). The architecture includes a service for the management of measurement data. This service seems to be generically reusable for multiple energy applications (R3). The article does not mention the support for multiple measurement types (R4). The requirements R5, R6 and R8 are not fulfilled since the article does not define the management of master data, semantic information or frontend visualizations or dashboards. However, the architecture defines different computation modules for example a grid calculation module and a forecast module which are fulfilling the requirement R7.

3.3.6 Designing a Smart City Internet of Things Platform with Microservice Architecture [KJ+15]

The next architecture presented in [KJ+15] is microservice-based and defines a platform with the name DIMMER. It is a platform that manages IoT devices and their data in the context of smart cities. The main characteristics for the architecture are the following:

- Componentization via Services
- Organization around Business Capabilities
- Smart endpoints and dump pipes
- Decentralized Governance

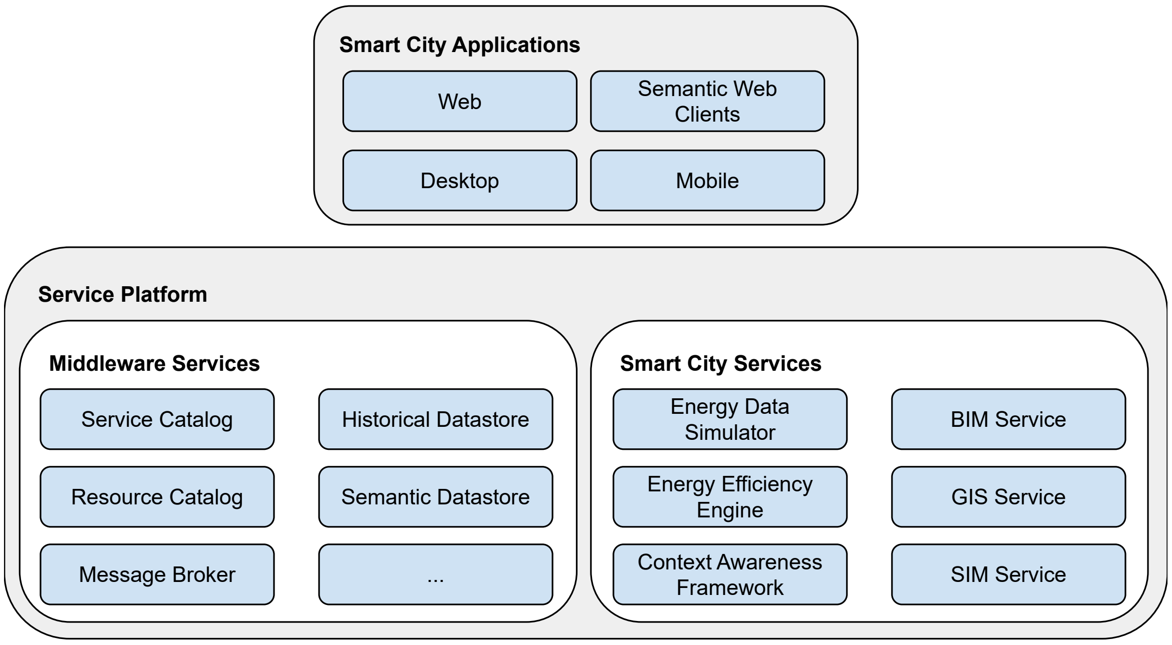


Figure 3.5: Architecture of the DIMMER Platform from [KJ+15]

- Decentralized Data Management
- Evolutionary Design

They are very similar to preceding articles. Next, the benefits from an architecture implementing those characteristics are discussed. The benefits are *technology heterogeneity*, *resilience*, *scaling*, *organizational alignment* and *composability*. Figure 3.5 shows the resulting architecture. It consists, on the one hand, of different middleware services, which are a generic abstraction of the IoT devices connected to the platform. On the other hand, domain-specific services are defined. For the present thesis, the middleware services are of special interest since they are designed generically and not domain-specific. The middleware separates the incoming data in measurement data stored in the historical datastore and semantics stored in the semantic datastore. This composition is similar to the data grid [CF+00]. The other middleware services handle service discovery and ingestion of data, which is not in the focus of the present thesis.

Analysis

Since it is a microservice-based architecture including REST APIs, the first requirement (R1) is met. The platform is designed in a scalable way which is confirmed by the authors

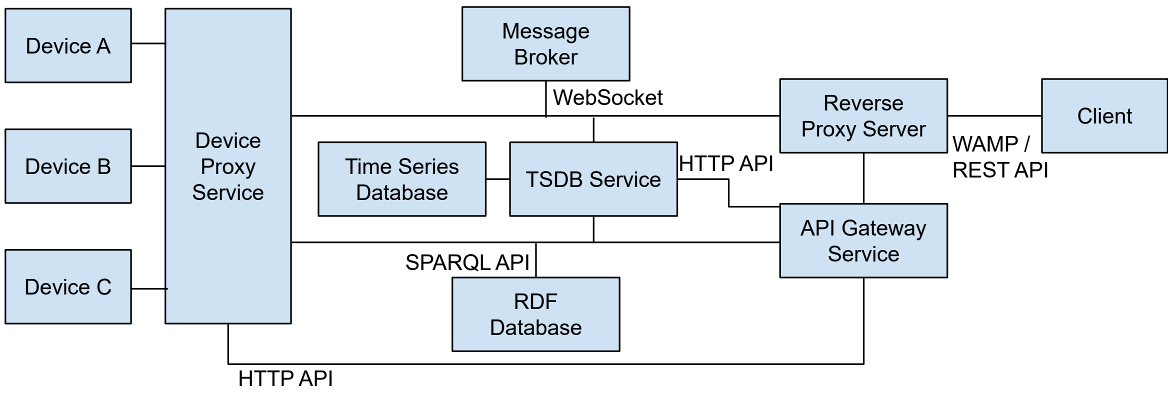


Figure 3.6: Overview of the Architecture from [KK+16]

(R2). The architecture supports management of measurements in an abstract and generic way (R3). Furthermore, since the architecture only features a single service dedicated to measurement data and the authors do not mention the distinction of different measurement types requirement R4 is not fulfilled. Since one service provides the support to manage metadata of measurements and of related entities, the requirements R5 and R6 are met. The article does not mention any analysis components (R8). The same goes for the frontend (R9).

3.3.7 SEMIOT: An Architecture of Semantic Internet of Things Middleware [KK+16]

The second last article [KK+16], that was selected for the present section, defines an architecture describing a middleware for IoT. Very similar to the previous publication, the architecture separates measurement data from semantic data. Additionally, the semantic data management is build on top of existing semantic ontologies with appropriate interfaces (SPARQL API) and storage. The architecture provides a REST API and a message broker in order to communicate with clients. Figure 3.6 illustrates the design.

Analysis

The article does not mention the term *microservices*, even though the architecture seems to be very similar to a microservice-based one (R1). The article presents experimental evaluation which shows that the architecture is scalable (R2). The presented system features a time series database with a service providing a REST API. This service offers a generic measurement management for all IoT devices (R3). The authors do not mention support

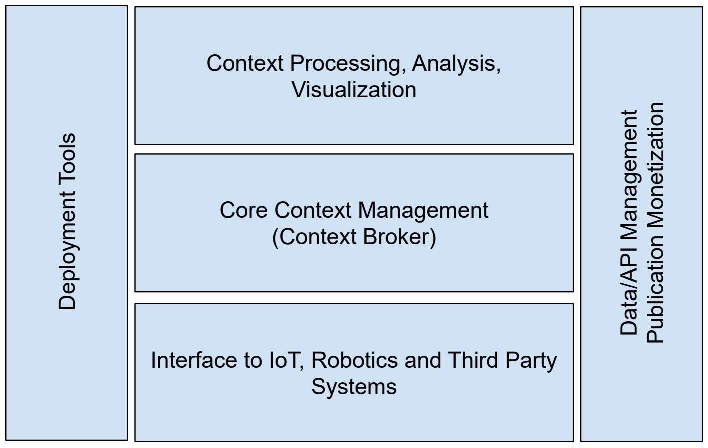

Deployment Tools
Context Processing, Analysis, Visualization
Core Context Management (Context Broker)
Interface to IoT, Robotics and Third Party Systems
Data/API Management Publication Monetization

Figure 3.7: Overview of the Different FIWARE Components from [FIWARE]

for multiple measurement types or master data management (R4, R5). With the support of ontologies however, the architecture fulfills the requirement R6. Finally, no data analysis or frontend is mentioned which excludes R8 and R9.

3.3.8 FIWARE [FIWARE]

The last related work is an European project called FIWARE. It is defined as follows:

Definition 5 *FIWARE is an open source initiative defining a universal set of standards for context data management which facilitate the development of Smart Solutions for different domains such as Smart Cities, Smart Industry, Smart Agrifood, and Smart Energy. [FIWARE]*

The main component that is also the only mandatory one is the context broker. It is a component that allows to manage so called context information. This information is the relevant business data for the application using the FIWARE system. There are different context brokers available that each use the standardized NGSIv2 REST API that is part of the project. This API allows operating on the business objects called entities. Each entity might include measurement data in the form of instantaneous values. Figure 3.7 shows an overview of the different high level components such as the context broker. Each of these components have different implementations which are published through the community of the FIWARE project. A combination of these different components can be put together in order to form a platform managing the application-relevant data.

Analysis

The first requirement R1 is partially fulfilled since the open source initiative defines a REST API that is used by the different implementations. Streaming is defined by special context brokers, however not every context broker is able to stream for example time series data. Additionally, the platform is not built with microservices as a main idea. Since each component is isolated and can be deployed separately, there are however many similarities with a microservice architecture. However, the initiative does not define the components as microservices.

The second requirement is the need for scalability. The core component with its default implementation the Orion Context Broker can be deployed multiple times and therefore scale horizontally. However it is complex to analyze every single implementation of the different components whether it is scalable or not. However, since the central component is scalable, requirement R2 is considered fulfilled.

The third requirement is met since the different components are all separately deployable and there are different implementations for each component which show the reusability. Furthermore, the components can be used for different domains and use cases which shows that they are generic. R3 is therefore fulfilled.

The context broker QuantumLeap is an implementation that supports two time series databases (TimescaleDB and CrateDB). It implements the NGSIv2 REST API but it does not allow to query time series data directly. The user has to use the database API in order to query data. Furthermore, there is no context broker that supports multiple measurement types. However, the generic definition of components allows to implement one or many new components that realize this support. Therefore, R4 is considered partially met.

The context broker with its standardized API manages entities. These entities are the same as master data objects. Additionally, the context broker supports basic CRUD operations. This fulfills the fifth requirement.

The sixth and seventh requirement (R6, R7) define the need for semantics and data analysis. Both of these requirements are met by the FIWARE ecosystem. FIWARE introduces the concept of data models which defines the semantics of the data managed by the context broker. Furthermore, the component *Context Processing, Analysis and Visualization* has multiple implementations that provide different forms of data analysis.

The last requirement is requesting a frontend that is web-based and closely coupled to the rest of the system allowing it to semantically understand the information stored in the system. FIWARE has a component for visualization and the implementation called WireCloud. It is similar to a portal system. The user can create different display components on a dashboard and connect it to the data stored in the system. This fulfills the last requirement (R8).

3.3.9 Unconsidered Articles

The literature research for the present thesis included many more articles, however, not all of the article were considered to be analyzed in detail in the current chapter. Therefore, additional articles will be briefly mentioned in this article. Additionally, the reason why they were no considered is given.

First, a few articles define a microservice-based architecture, however, they do not specify the concept in enough detail in order to evaluate it in this thesis [KS+17, GC+18, WC+18, SH+19, ZL+19, RE+10, SL+17]. Second, different articles define a concept that is very technical or focused on hardware rather than software systems [VC16, GB+18, BM+16]. Finally, some articles have a focus that is not of interest in this thesis [LH+17, Al18].

3.4 Summary of the Analysis

Table 3.3 shows an overview of the requirements that are fulfilled for each publication that have been analyzed. It shows that no article presents a solution fulfilling all requirements.

The analysis of the related work shows that most of the articles describe service-oriented architectures that are scalable and have the ability to manage measurement data. The requirements R5, R6 and R7 are covered by some of the publications but it is rare that an architecture is presented that combines measurement data, semantics and data analysis in a service-oriented style. Additionally, a few articles define a frontend component, however they lack the semantic context in order to have a frontend that is aware of data semantics. This is a first gap to consider. Requirement R4 is not met by any article which shows a second gap. Since many of the publications describe a measurement system some might also support different measurement types, however, no publication describes a specific concept for it. FIWARE is closest since it is very generic. This shows that there is a gap for an architecture that defines a measurement system with support for multiple measurement types including semantic descriptions and a semantics-aware frontend.

Article	**R1**	**R2**	**R3**	**R4**	**R5**	**R6**	**R7**	**R8**
A Software Architecture-based Framework for Highly Distributed and Data Intensive Scientific Applications [MC+06]	(X)	X	(X)		X	X		
Experience on a Microservice-Based Reference Architecture for Measurement Systems [VL+14]	X	X	X	(X)			X	(X)
Energy Cloud: Real-time Cloud-native Energy Management System to Monitor and Analyze Energy Consumption in Multiple Industrial Sites [SC+14]	(X)	X	(X)				X	(X)
Next-Generation, Data Centric and End-to-End IoT Architecture Based on Microservices [DB18]	X	X	X			X		
Reference architecture for open, maintainable and secure software for the operation of energy networks [GM+17]	(X)		X			X		
Designing a Smart City Internet of Things Platform with Microservice Architecture [KJ+15]	X	X	X		X	X		
SEMIOT: An Architecture of Semantic Internet of Things Middleware [KK+16]	(X)	X	X			X		
FIWARE [FIWARE]	(X)	X	X	(X)	X	X	X	X

Table 3.3: Overview of Requirements that are Fulfilled by Related Work

4 Design of a Microservice-based Reference Architecture for Measurement Systems

This chapter is the first of four chapters presenting scientific contributions. After a clarification of the most important terms, an analysis of the scientific background, the reference architecture is discussed in more detail in the following sections. Next, the process of designing the reference architecture step by step following the guidelines from [Ev03] and [HG+17] will be presented. These steps discuss different artifacts that give insight in the ubiquitous language, the domain which the reference architecture resides in and the context map that shows the different relationships between identified bounded contexts. Second, a technical view is presented and the design of the reference architecture is finalized.

4.1 Definitions

Before the actual scientific analysis is presented, a few terms are introduced and possible ambiguous definitions are clarified. Since the present chapter heavily uses the term *domain*, the term is defined and ambiguous meanings between two interpretations of the term are discussed. The first occurrence of the term is in the *domain-driven design* approach which is used in the present chapter to design the reference architecture. In this context, the term is used to describe the area that the architecture is realized in. For the present thesis, this area called domain is the *measurement system*.

However, the different services dealing with measuring data also deal with different domains defining the *area in which the observations or metadata originate*. This can be a domain from e.g. the energy or the environmental field. The *measurement system domain* is more generic and abstracts specific domains for different applications. In order to avoid ambiguous usage of the term, the domain for a specific application will be called *business domain*. Figure 4.1 illustrates the difference between both terms.

The measurement system domain is presented with two generic types of data: *measurement data* and *master data*. A closer definition of these types is given later in this chapter. The important aspect at this point is that the data is specified in a generic way.

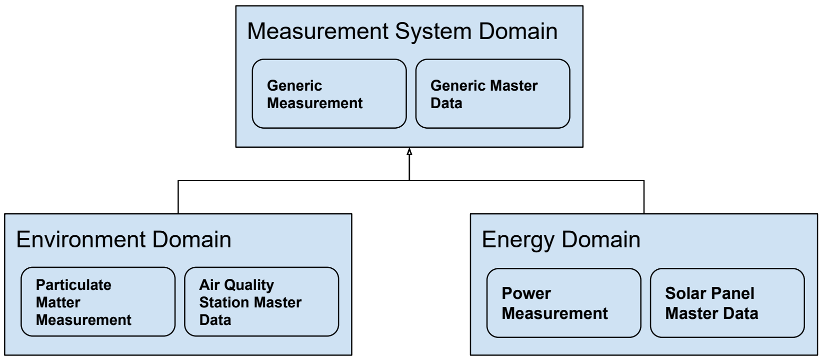


Figure 4.1: Definition of the Terms Domain and Business Domain

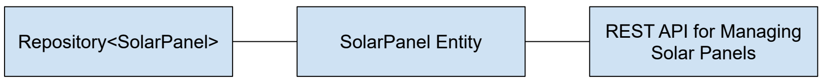


Figure 4.2: Example for a Microservice Modeled After a Specific Business Domain

Furthermore, there are two possible business domains which can be integrated and make use of the generic architecture. The *environment domain* defines for example air quality data which can be separated in a particulate matter (PM10) measurement and the information of the measurement station. As for the energy domain, a measurement of the power generation and the according information of the generating solar panel is given. Both business domains define a specific context. However, both can be generically described by the measurement system domain.

Since the reference architecture presented in the current thesis is designed to be able to work with many different business domains at the same time, the only way is to design it according to the much more generic measurement system domain. This definition is closely related to the limitations given in Section 1.5.

A second term that might be used ambiguously is the term *semantics*. For a simple application implemented using a microservice backend, semantics are typically included in the design of the service, it's API and the source code of each microservice. An example is an application showing information on air quality in a specific region. In that case, a microservice might be responsible for the management of air quality measurement stations. The data structure, data types and additional semantic information needed for the modelling of such a station are usually defined in an entity class of the microservice's source code. This entity is usually managed by a repository which represents the persistence API connecting the service to a database.

Figure 4.2 shows such a model with the mentioned components. An important condition for such a design is that the definition of the domain entity is known and defined *before* the design and implementation of the service. If this condition is not fulfilled, the semantic information is not known during the design phase and can not be integrated into the software's source code. In that case, the information has to be ingested into the software at runtime and the semantics have to be defined by data objects. This is why, a separate microservice has to be defined in order to manage the business domain's semantics. However, the design of the measurement system domain has its own, more generic semantics, which are integrated in the software similar to the model shown in Figure 4.2.

Both terms, domain and semantics define two layers of abstraction for the present thesis which are important to distinguish. First, the more generic measurement system defining an abstract semantic description of the system. Second, the business domains with their more specific semantics that are not known before the implementation of the software and have to be managed by the software at runtime.

4.2 Scientific Analysis

As seen in the introduction (Chapter 1), measurement systems are shifting from monolithic systems storing all the data and its semantics into single traditional databases towards systems distributed among many microservices shipped via containers to a cloud infrastructure. Such systems require new concepts in order to have the traditional data management mapped to the new technologies and environments. Since each microservice is responsible for a single task, main concepts such as the management of data and the management of the interpretation of this data, which turns raw data into information, are separated from each other. In publications such as [CF+00, MC+06] different approaches are already defined how such a separation can be applied on an abstract level. Therefore, a concept for flexible management of measurement data and the definition of semantics for this data and other measurement systems is needed. This concept can be divided into two parts.

The first part of the concept is the management of data using separate services. The most important type of data is measurement data gathered by sensors, simulations or other means of observation. A second type is master data that describes business objects in the domain that an application resides in. These two types are also described in [PA14].

However, this publication does not take into account the second part of the concept which is the management of semantics describing the mentioned data. Semantics are important in order to describe the data for a specific context. Applications can only use the data if such a context is described. The context allows the data to be interpreted as information.

This very simple definition of data services is further extended and analyzed in the publication [SH+19]. Furthermore, this publication presents an architecture with semantic services. One main feature is the enrichment of a non-semantic dataset based on an ontology containing the vocabulary to semantically describe the dataset. The dataset itself is converted into RDF triples. For the energy and environmental applications considered in the present thesis, the data services have to be optimized in order to meet the appropriate performance. Therefore, a transformation as described by the publication is not possible. Additionally, the microservice architecture separates the data from its semantics. This also leads to a mapping between semantics and the data but it has to be a mapping that references data sets and their semantics across different services. Such a separation of services is similarly done in [GM+17, KJ+15, KK+16]. In all of these publications, there are separate services for the management of data (master data or/and time series data) and for the management of its semantics. Two downsides of [KK+16] are that the architecture is specific for the IoT domain and does not support the storage of master data which is of importance in the present thesis.

Most of the relevant publications (e.g. [RS+18, MJ+16, WC+18, KI+18, GC+18]) define a microservice architecture that is implemented with a specific application or business domain in mind. However, this is not suitable for the present thesis since the reference architecture is meant to be used by multiple applications across different energy or environmental business domains (see Section 1.5). Therefore, the domain of the reference architecture has to be of a more abstract nature than the business domains in the mentioned articles. This leads to a microservice-based architecture that can be used by multiple applications which do not necessarily have to share their business domain. Additionally, this abstraction also allows to define the more concrete business domain knowledge in the semantic services. As already defined in the previous section, the domain of the reference architecture is called *measurement system domain*.

All of the mentioned publications in this section are missing a description of their architectural design of using specific standards, best practices or references from literature. Such a design can be supported by development principles e.g. domain-driven design (DDD). DDD is getting more and more attention [HG+17, MM+18, RS+18] seen by a rising number of scientific publications. DDD is used in the next sections in order to design a reference

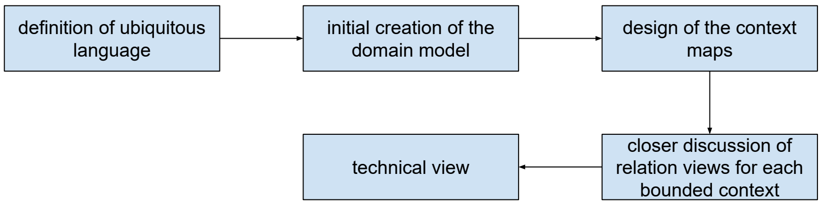

definition of ubiquitous language
initial creation of the domain model
design of the context maps
closer discussion of relation views for each bounded context
technical view

Figure 4.3: Overview of the Design Process Following the Principles of DDD

architecture that can deal with measurement data and its semantics in an abstract way. Figure 4.3 shows an overview of the design process and therefore the structure of this chapter.

First, a language is defined that is understood by every participant of the software development process as well as the end users and/or customers. This artifact makes sure that misunderstandings and ambiguous use of terms are avoided.

The second step is the creation of the domain model. This model is similar to a class diagram known from classical software engineering processes but it has not that much detail and does not represent the different classes that are implemented. It represents different components of the semantic domain(s) of one or many embedded applications.

The third step further enhances the domain by defining bounded contexts grouping different components of the domain. Each bounded context is again embedded in a part of the domain. Finally, the context map defines relationships between the bounded context in order to define how they interact with each other.

It has to be noted, that in literature, the second and third step are often in the reverse order. This is justified if there is already an existing domain model and an existing context map. In this case the context map can be extended in order to decide if a new bounded context is needed or if the new parts of the domain are embedded in existing bounded contexts. However, if there is no domain yet, the order is not that clear and in the present thesis the domain model was first designed and based on that artifact the context map and the bounded contexts were defined.

A fourth step in the design process is the closer discussion of each bounded context and a closer look at the relevant domain model the bounded context includes.

Finally, the design process is completed by a mapping from the strategic modeling to a technical definition of the reference architecture.

4.3 Ubiquitous Language

In order to find the appropriate *terms* that each developer and software architect understands in the same way, firstly, a ubiquitous language is defined. Figure 4.4 shows a language sketch that defines all of the important terms and their relationships to each other. The most important term in the diagram is *Measurement*. This term is also used in [VL+14, GM+17]. It represents a generic measurement that is recorded or is stored in the measurement system. A scientific gap that is identified in Chapter 3 is that in most of the publications only one form of measurement is considered. It is not distinguished between different types which is important for the support of different energy applications. Therefore, the three most common types of measurement are defined in the language sketch: univariate and multivariate measurements and sampled values. These are further described in Chapter 2. Despite different aspects of the various types, there are terms that are always valid for any kind of measurement.

The first is that each of them consists of multiple measured values. Usually these are multiple values that are grouped in different columns called *Fields*. Each field measures a single metric that has its defined *Unit* and possible other properties. Depending on the measurement type, the multiplicity of fields changes. For univariate measurement, there is exactly one field. The same applies to sampled values. In the case of multivariate measurements, there can be a variable number of fields with a minimum of one. Additionally, measurements can be aggregated by their time resolution which is parameterized by a defined *Resolution Interval*. Furthermore, the timestamp of the observation or recording has to be defined in order to map each value to a point in time. Last, each measured value is associated with one or more *tags* that further describe the measured value and is mainly used for the filtering of measurements.

A second important term is *Master Data*. It describes data that usually provide information for objects that are relevant for the business domain. An example for such an object is a *Measurement Device*. It has different *Properties* that define the device. Listing 4.1 shows an example of such properties in JSON representation. Furthermore, a master data object can be related to other objects, measurements and digital assets.

Besides the properties of a master data object, it also needs a *Semantic Description*. This description validates the objects and adds additional semantic information e.g. data types and human readable descriptions of the actual data. In order to organize different descriptions, they are grouped in semantic classes also called *Schema Classes* in the present thesis.

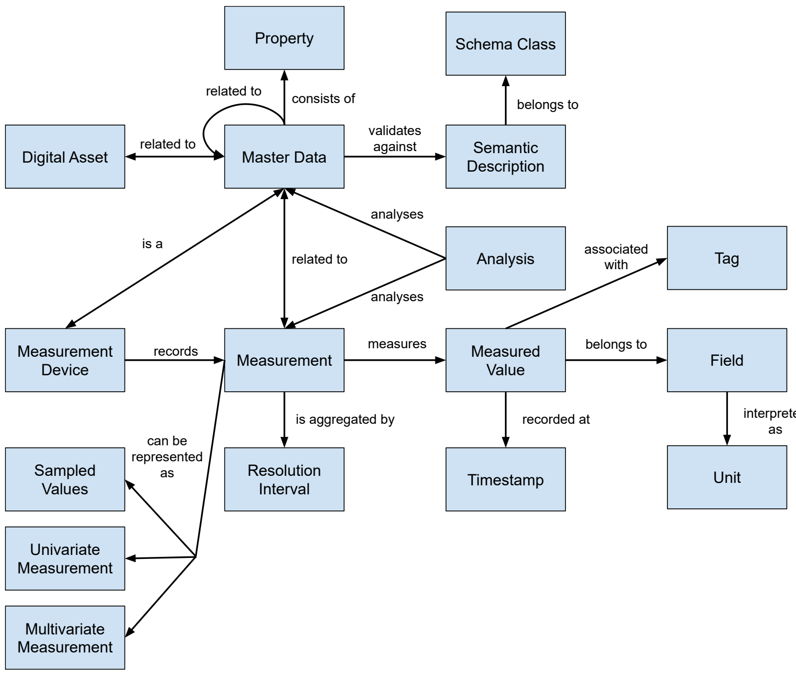


Figure 4.4: Language Sketch to Define the Ubiquitous Language

Listing 4.1: Example Properties for the *Measurement* Device Object

```
{
  "name": "Device-001",
  "description": "temperature sensor in room 231",
  "location": "room-231",
  ...
}
```

The last two terms that are relevant for the measurement system domain are *Digital Asset* and *Analysis*. Digital assets describe binary data that is usually represented as a file (e.g. PDF or binary executable of a simulation). They can be related to master data objects. The *Analysis* performs computations on master data and/or measurements and provides statistical insight into the data. An example is the minimum value of a specific field for a measurement in a specific time frame.

After the presentation of a ubiquitous language, the domain can be further designed. The following section focuses on an overall design of the measurement system domain.

4.4 Domain Model

The domain model shown in Figure 4.5 is an artifact defined in the domain-driven design (DDD) as presented in [Ev03]. Most of the domain vocabulary used in Figure 4.5 is already defined in the language sketch. However, it enriches the language sketch by defining DDD types for the different domain objects, multiplicities and further detail (such as attributes and methods) for each object. More detail on the description of the diagram type can be found in Chapter 2. Similar to the language sketch, the main objects in the domain model are Measurement, Master Data and Digital Assets.

The most central part of the measurement system is the Measurement entity. It is responsible for the creation and management of different measurements. The entity has to be able to create a measurement. This process consists of defining a unique measurement id and the fields that are observed. After this first step, new values can be added, rectified or removed. The main task of the entity however, is to provide a rich API in order to return a specific time series (as defined in Chapter 2). Each measurement entity represents one of the three types: univariate measurement, multivariate measurement or sampled values. As already mentioned in the previous section, each measurement has a variable amount of fields depending on the measurement type. Each field represents values in a specific timestamp. The diagram shows three possible forms however many more are imaginable. The measurement entity is related to the master data entity since each master data object can have a relationship to one or more measurements. Furthermore, the entity is related to the analysis entity in order to get complex statistical information about each measurement.

The second most important part is the Master Data entity. It features basic CRUD operations in order to manage the different master data objects. Similar to the measurement entity, a rich API for requesting and filtering a list of master data objects is needed. Each object is

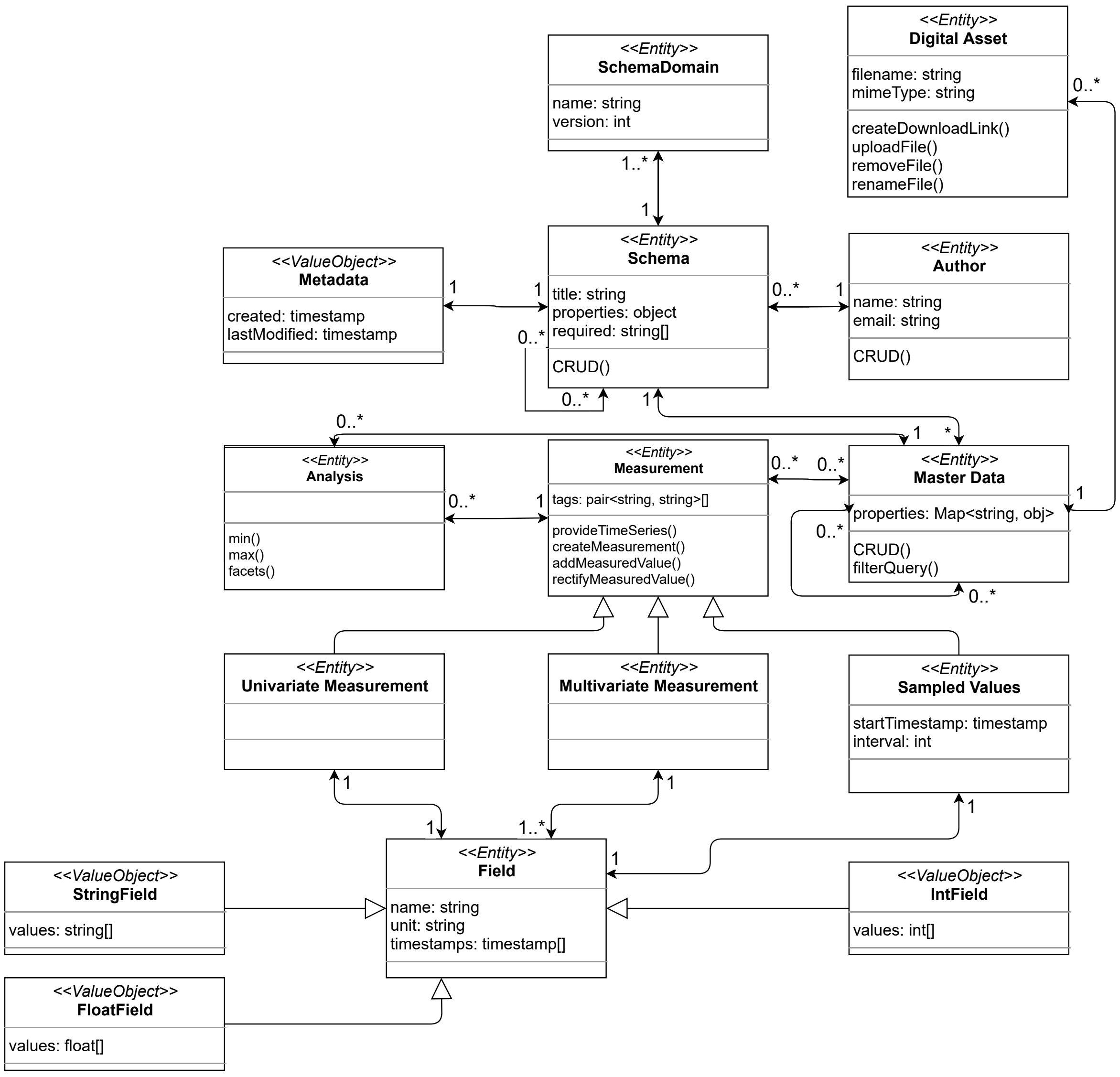


Figure 4.5: Domain Model of the Measurement System Reference Architecture

mainly characterized by its defined properties. The possible properties, their data types and human readable explanations are defined by semantic descriptions which are managed by the schema entity. A second relationship is the relation to digital assets. Each master data object might be related to one or more digital assets.

The schema entity which is responsible for defining the semantics of the master data objects has basic semantic properties (similar to JSON Schema, see Chapter 2). The methods are basic CRUD operations. Furthermore, each schema has an author, who is responsible for one or a set of schemas. The author entity also provides CRUD operations. Furthermore, each schema has metadata describing e.g. the creation time or, last modification time. The last relation is the schema class that is related to a group of schemas. Each class has a name and a specific version in order to keep track of changes.

The last important part is the Digital Asset entity. The representation of a file is implemented with properties that describe the file name and the MIME type of the file. The operations allow a file to be uploaded, removed and renamed. Additionally a download link can be created in order to access the file using a remote device.

Another entity belonging to the domain is the Analysis entity. It allows to compute complex statistical information for measurements and master data. For each measurement the analysis entity might compute minimum or maximum values for specific time intervals and fields. Another example is the computation of statistical distribution for specific properties of a master data object.

After having discussed the domain model, the context map is presented in the next section.

4.5 Context Map

The context map shown in Figure 4.6 uses the information of the domain model and summarizes the diagram in subdomains and bounded contexts (BD). This results in a much cleaner diagram with less information. It is important in order to define the main areas of the domain and the relationships between the different BDs.

The subdomains are divided into two *core subdomains* that are dedicated to the two most important parts of the domain model: the management of measurement data (*MeasurementManagement*) and the management of master data (*MasterDataManagement*). A third subdomain, called Analysis, serves as **supporting subdomain** for the first two.

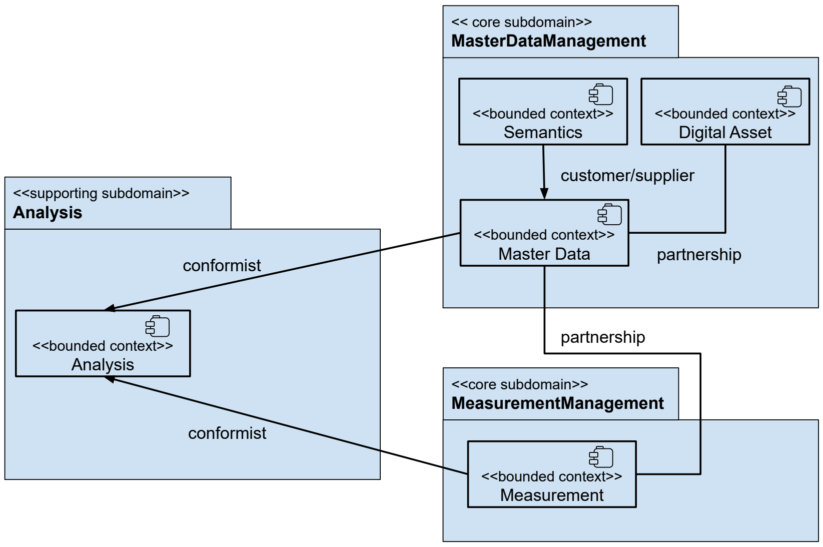


Figure 4.6: Context Map of the Measurement System

The measurement management subdomain consists of the measurement BD. It has a partnership relation with the master data BD since both manage data that can be related. The analysis BD on the other hand uses the measurement BD as is and therefore is in a conformist relationship with it. The master data management subdomain consists of the master data BD and the semantics BD. Both are in a customer/supplier relationship since the master data BD uses information of the semantics BD but not the other way around. Furthermore the subdomain also includes the digital asset BD which is, similar to the measurement relation, in a partnership with the master data BD.

4.6 Relation Views of Bounded Contexts

The domain model was already presented in Figure 4.5, however this section presents a more detailed view of each bounded context and especially the connection to the entities of the related bounded contexts.

Figure 4.7 shows the relation view of the measurement bounded context. The changes between Figure 4.5 and Figure 4.7 are that there is a relation with the analysis entity. However, the measurement entity does not need the facets() operation since this is meant to be used by the master data entity. The master data entity is not reduced in attributes or properties in contrast to the shared entity since the measurement entity might need all of the functionality.

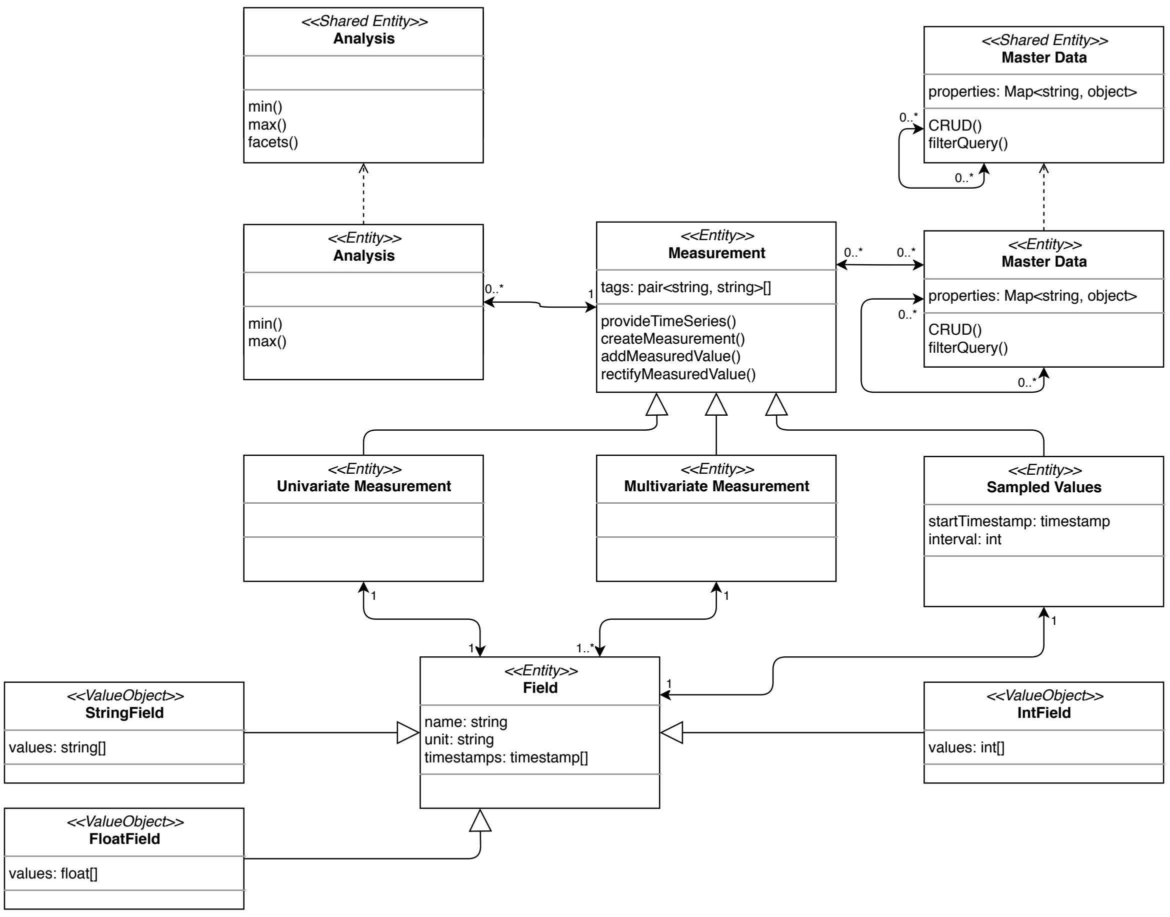


Figure 4.7: Relation View of Measurement Bounded Context

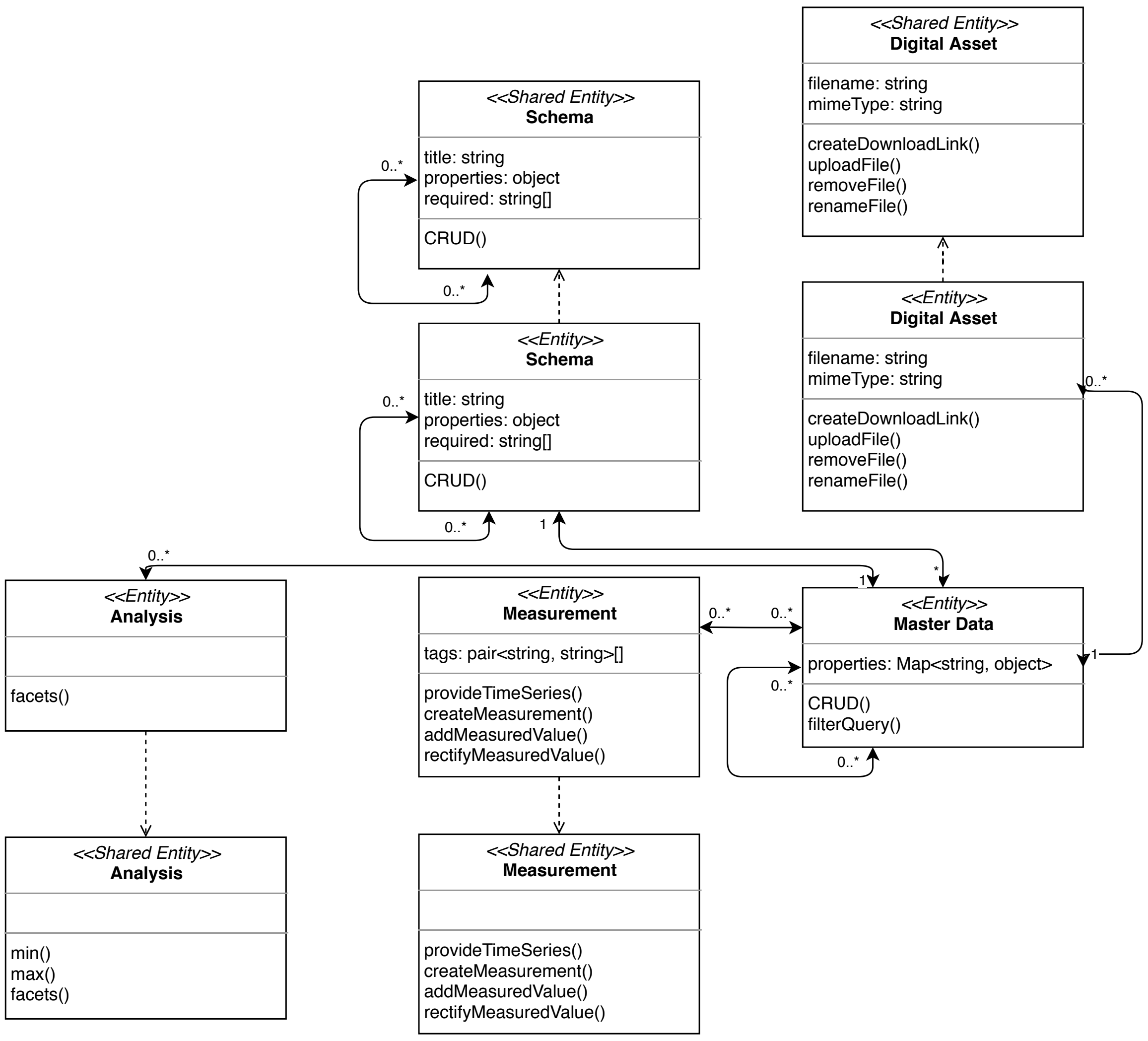


Figure 4.8: Relation View of the Master Data Bounded Context

Figure 4.8 shows the relation view of the master data bounded context. The schema and the digital asset entities are shared across the two other bounded contexts from the master data subdomain. The measurement entity is shared with the other core subdomain. Similar to the previous domain model, the statistic entity only needs the relevant operations for master data analysis but not the measurement analysis operations. The relation view clearly shows that the master data entity is the most complex since it has the most relations to other bounded contexts.

The next Figure (Figure 4.9) shows the relation view of the semantics bounded context. It is the only relation view that has no shared entities. It is completely independent from any other bounded context from this diagram's point of view (the relation between the schema

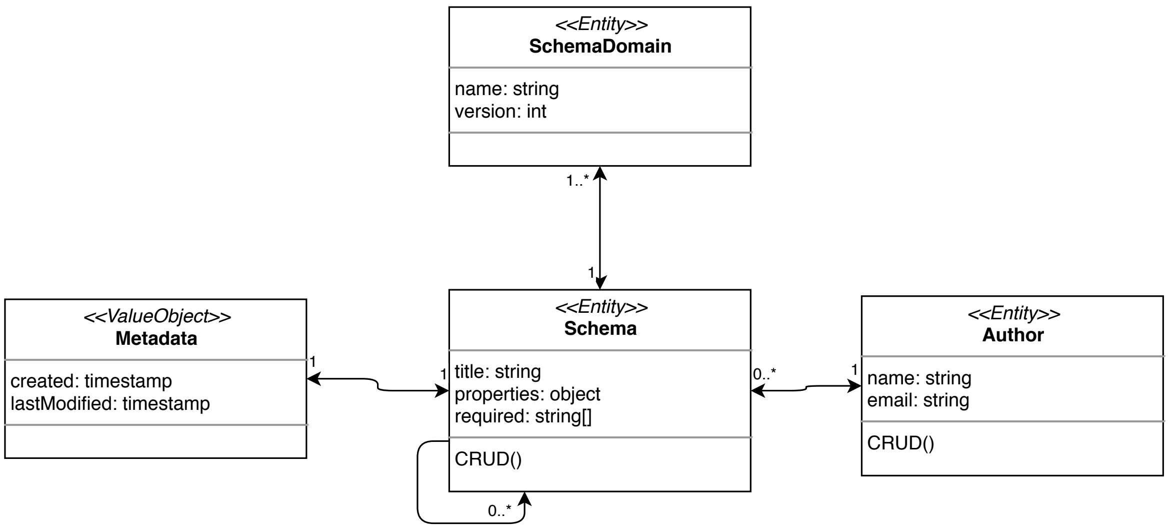


Figure 4.9: Relation View of the Semantics Bounded Context

BD and the master data BD is unidirectional from supplier to customer).

The next relation view (Figure 4.10) describes the domain of the digital asset bounded context. It only consists of a single entity and a single relation to the master data entity which therefore is the smallest domain model.

The final relation view (Figure 4.11) shows the extract for the analysis bounded context. As it is responsible for the analysis of measurements and master data, it is related to those two entities and therefore bounded contexts. The features of both services are not needed in their entirety but only the read operations.

4.7 Technical View

After having discussed the tactical and strategic design process, the first technical view has to be established. This is the first time in the design of the reference architecture that potential technologies are discussed. Therefore, possible restrictions or extensions to the current state of the concept are not ruled out. The main focus of this section is the definition of the different microservices and to ensure that each service is cut appropriately. Figure 4.12 shows a first possible mapping from each bounded context to exactly one microservice. This first strategy is a good starting point since the bounded contexts are already identified and provide a good separation of concerns. However, there are a few design choices that have to be discussed before the reference architecture is further defined.

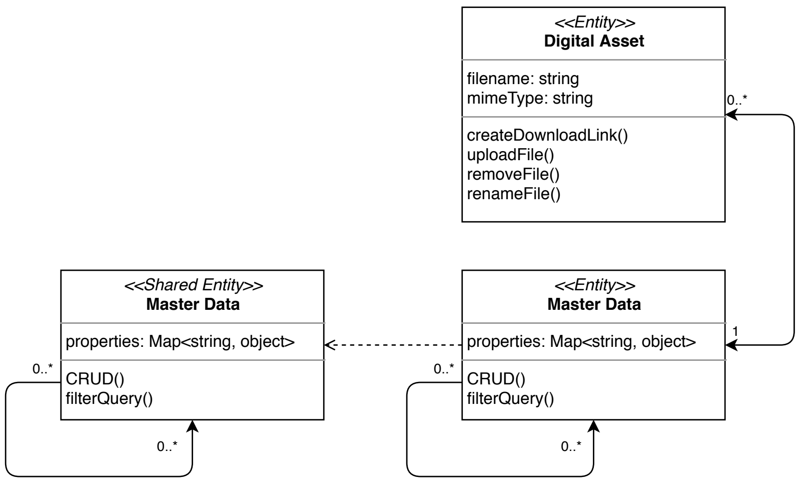


Figure 4.10: Relation View of the Digital Asset Bounded Context

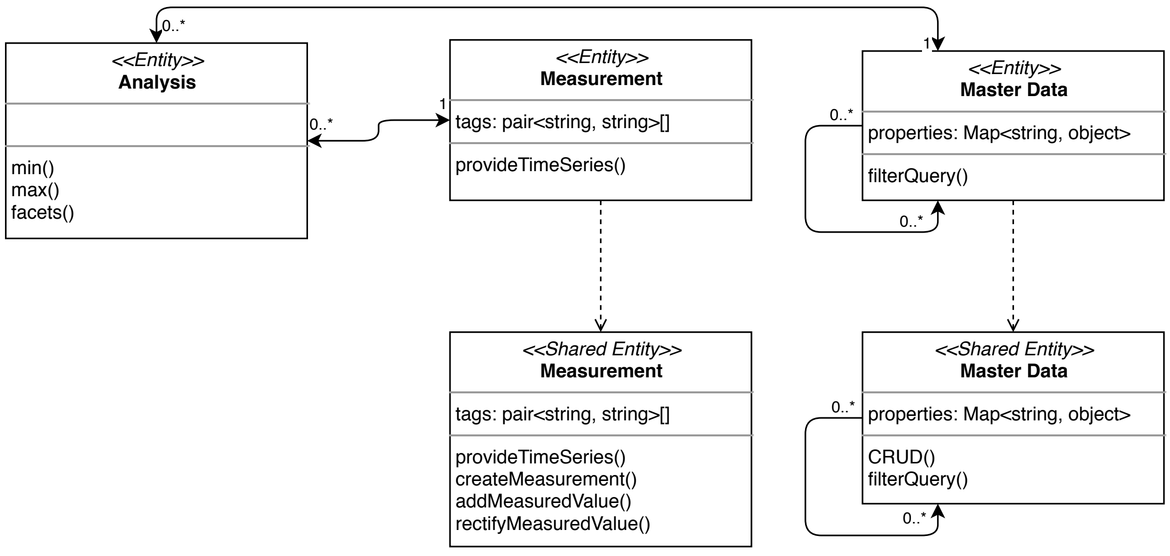


Figure 4.11: Relation View of the Analysis Bounded Context

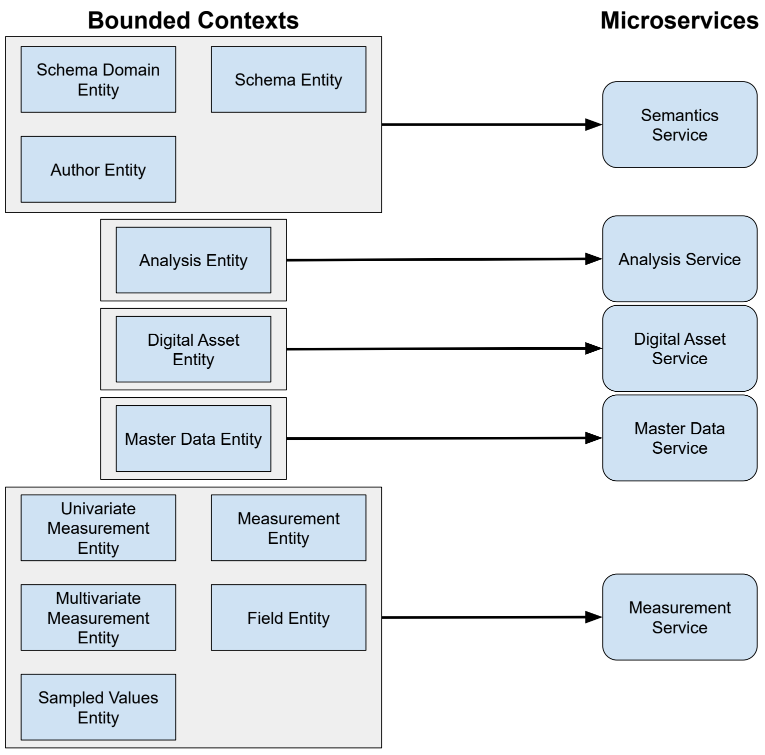


Figure 4.12: Mapping of Bounded Contexts to Microservices

First, the measurement service has a lot of responsibility since there are different types of measurements that have to be treated differently. Having a look at time series databases[1] one can see that most of them can handle univariate and multivariate time series (a counter example is OpenTSDB which can only handle univariate time series). However, most of the time series management systems have no support for very fine grained measurements (below microseconds) and therefore have no good support for sampled values which typically have a resolution below one value per microsecond. It might be reasonable to use different time series management systems for those different types. This further increases the complexity of the potential measurement service shown in 4.12. A solution for this problem is to define multiple services that each handle a measurement type. Univariate and multivariate measurements are also considered different since there are time series databases that can only store univariate measurements but a unification of both cases will not be excluded at this point. However, in order to have a more generic reference architecture, the measurement service is not omitted but serves as a facade [GH+94] for the three more specific services. The change to the measurement service is depicted in Figure 4.13.

[1]Different time series databases are introduced in Section 5.4.1.

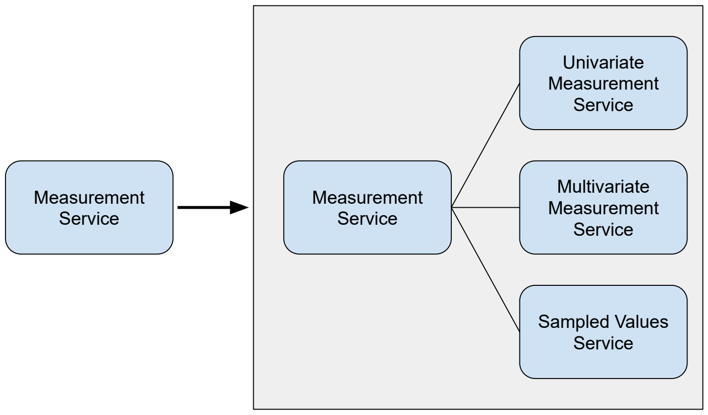

Measurement Service
Measurement Service
Univariate Measurement Service
Multivariate Measurement Service
Sampled Values Service

Figure 4.13: Separation of Responsibilities of the Measurement Service

The second change of design addresses the analysis service. As seen in the domain models, the analysis service, among other things, analyzes measurements and computes statistics. The measurement service on the other hand provides management for measurements. This is a clean separation of responsibilities and functionality. Looking at time series databases, raises the problem that they often already provide a range of analysis features for the data. Since each microservice has its own database and does not share the database with another service, it is not reasonable that the analysis service copies this functionality. Therefore, the analysis service and the measurement service have to share the responsibility for data analysis. A possible solution for the problem is that the measurement service provides simple analysis and the analysis service is responsible for more complex computations.

Considering the presented changes, Figure 4.14 shows the actual resulting technical view of the reference architecture. The diagram additionally shows a gateway to emphasis that it is a microservice-based reference architecture providing a single point of entry. Each microservice has its own database in the background to store and retrieve data. In the following, the seven services will be shortly presented with their basic functionality based on the domain model and the design changes in this section.

- The **Semantics Service** provides basic CRUD operations for the management of schemas master data objects. These schemas consist of semantic descriptions in form of data types and human readable descriptions. Each schema belongs to a class, is connected to an author and has additional metadata. This allows the service to manage the domain knowledge of specific use cases while still being of generic nature. The measurement system has to cope for example with both business domains presented at the beginning of Chapter 3 and the semantics service provides this functionality.

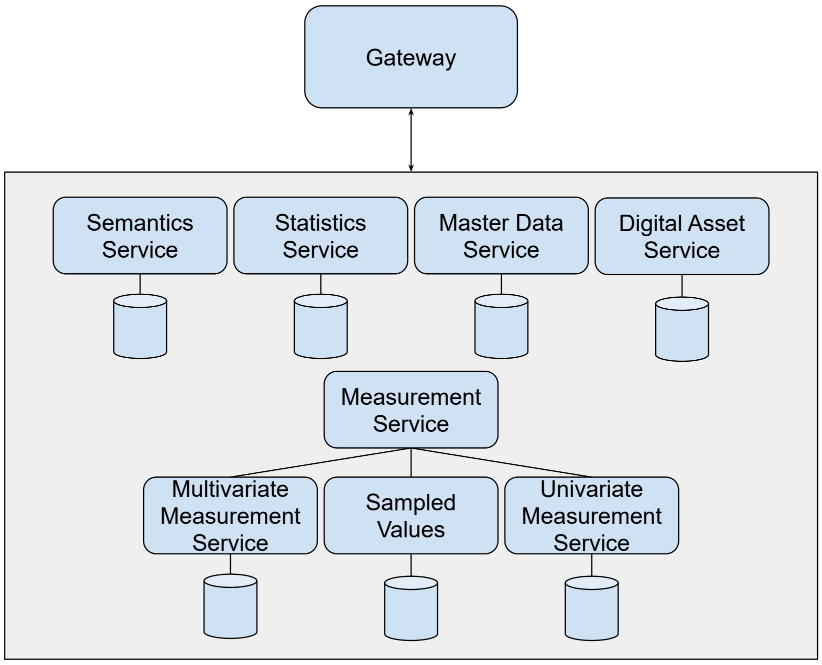


Figure 4.14: Overall Reference Architecture for Measurement Systems

- The **Analysis Service** analyzes data of the measurement service and the master data service. However, as mentioned before, basic operations might already be implemented in the measurement service itself depending on the database technology in use. The analysis can be useful for users to get insight into the data which is especially important for measurement data of larger volume. Furthermore, the analysis of master data can support user interfaces providing the user with additional information about the overall data of a specific context.

- The **Master Data Service** manages master data that is relevant for applications. As already defined in the domain model, the service provides basic CRUD operations for the management of the data. Furthermore, the service provides more complex read operations in order to filter a list of master data objects. This allows applications to make complex queries and only get the needed results.

- The **Digital Asset Service** manages data that is classified as digital asset as defined in the language sketch. In the domain model, a range of operations is already defined that provide management for the stored files.

- The three measurement services have the responsibility to provide the functionality that is included in the measurement bounded context. Since it is too versatile, not

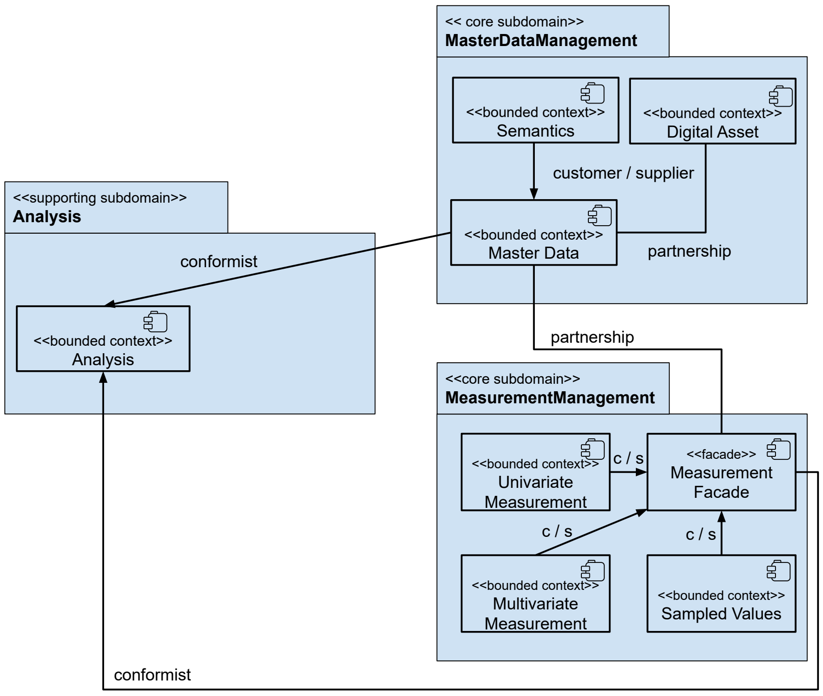


Figure 4.15: Updated Context Map of the Measurement System

a single service is defined but one for each measurement type. However, all three services are hidden behind a unified service facade that is defined by the measurement service. For now, the following three types are defined: /textbfunivariate measurement, *multivariate measurement* and **sampled values**. The first two measurement services deal with measurements that have one or more values for a specified timestamp (see Chapter 2). The third service manages sampled values which have the specialty that they are recorded at equidistant points in time and only an initial timestamp is stored. Those measurements are most of the time recorded in a very fine grained resolution and in high volume.

After having defined the technical view and after the discussion of the different changes, the context map can be updated. This process of updating the context map is very useful in order to keep the artifacts in line with the actual concept. In Figure 4.15 the new context map is depicted. The only change is that the measurement bounded context is replaced by a measurement facade that is of the type facade. This ensures that the other bounded contexts do not have to deal with more complexity than in the initial context map. All three measurement bounded contexts are in a customer/supplier relationship with the facade.

4.8 Conclusion

The first of four contributions presents the design of the overall microservice-based reference architecture. First, the ubiquitous language with all relevant terms is discussed in order to have a clear understanding of the domain language. The second design artifact that is presented is the domain model. It defines the different components of the domain including their attributes, operations and multiplicities. This model ensures that every aspect of the reference architecture is defined and assigned to the correct domain object. The next step is the definition of subdomains and within those different bounded contexts that each consist as a part of the domain model. The resulting artifact is the context map. Additionally, the context map shows the relationships between the different bounded contexts. After the definition of the overall domain model, the different domain models of the bounded contexts are discussed and especially the interfaces to related bounded contexts.

These design artifacts defined the context in which the reference architecture is developed. Furthermore, these tactical and strategic views are mapped to a technical view. This step is done by including technology aspects. In this case, these are mainly the databases behind each microservice. At the end of the last section, the microservice-based reference architecture is presented.

5 Distributed Data Management for Measurement Systems

This chapter covers the second contribution and therefore the first describing the reference architecture in more detail. The main focus lies on distributed data management. The design of the architecture (Chapter 4) showed the need for multiple data management services which consist of the following: master data service, measurement services, digital asset service. For these services scalability is crucial as requirement R2 describes. Especially the measurement services have to be highly scalable in order to fulfill the requirements for the storage and provision of different measurements. The following sections give an overview of the data management part of the architecture and discuss the different services and the appropriate concepts in detail.

5.1 Scientific Analysis

The idea to separate the data management in a microservice-based architecture (MSA) is not completely new and can be found in literature for example in [PA14]. The authors of this publication present an energy data management system. They classify the data relevant for their application into two categories: master data and time series data. However, they do not define the design or the structure of the architecture. Additionally, they discuss the challenges of traditional single RDBMS and the need for polyglot data persistence [Fo11]. Furthermore, they present a prototype implemented using MySQL, MongoDB and OpenTSDB to evaluate the proposed concepts.

A second publication [MJ+16] presents a web service providing meter data management in the field of smart home. The service features a Web API based on RESTful principles. This has become the standard for microservices as well and is used for every service discussed in the present thesis. However, it is crucial to use these principles in the right way following best practices [AA+18, GG+16]. Another important aspect treated in the publication are basic analysis functionalities. Two of those are shortly discussed in the following. First, the ability to filter measurement data based on time intervals and other criteria. This is needful in order to provide a specific subset of the measurement data that an application is interested in. An important sidenote of the publication is that the timestamp has to be indexed in the respective database. Additionally, as seen in the section presenting the measurement service in the current chapter, some other criteria might be indexed in order to improve the performance of filter queries. Such filter functionality is also useful for the master data service and the concept can be extended as shown below in the current chapter. A second analysis function that is discussed in the presented publication is the so-called condensation of data. It allows the reduction of data points that are in an interval of time in order to minimize the volume of data requested. This method is called *aggregation* in the present thesis. The publication further explains how such a functionality can be implemented.

A third publication [HS+18] evaluates REST APIs and message queues. It shows the importance of having both communication methods in an MSA in order to deal with a high number of requests. Additionally, an important property of measurement data is that it is often gathered with high velocity and has to be processed as fast as possible. This is another criteria that has to be analyzed in the current thesis which is not covered by the publication.

Furthermore, there are many publications defining measurement services with different emphasis on various aspects. In [FU+18], for example, a microservice for data storage and a microservice for analytics is defined. However, a single service for data storage is not suitable for the present thesis since such a service has too many responsibilities and is not designed in a sustainable way. Another publication defining a measurement service is [GM+17]. However, the architecture does not include a service allowing the storage of additional master data.

At last, the standard SensorThings API [HH+19, OGCSensorThings] from the Open Geospatial Consortium (OGC) is analyzed. The standard describes concepts to connect IoT devices, manage data and provide interfaces for web applications. The standard focuses on IoT devices. Each device is represented as a thing and is related to datastreams that are measured by sensors. A REST API is built around these entities and provides management capabilities for each entity. The API includes multiple concepts of the present thesis in a single API. Especially, the master data objects for the IoT domain and measurements are integrated in the API. The semantic description of the data is further realized by supporting linked data [SW+18]. The standard implements very similar approaches to the present thesis, however, the focus lies on IoT and it is not generically usable for other domains having e.g. the need for more specific measurement management or for master data management within other than IoT domains. The fact that the present thesis abstracts from a specific business domain allows to manage data in a more flexible way.

After the analysis of literature, a clear gap is seen for measurement systems which is the lack of heterogeneity of data services with different databases and their specific capabilities. There is no publication describing multiple measurement services discriminating different types of measurement data. Additionally, the service APIs are mostly not defined in detail. However, the API is the most important part of an MSA since it is the only interface to client applications. Furthermore, most of the applications define application-specific services. The present thesis however, defines a reference architecture for many applications requiring generic services. The following sections describe this architecture with a focus on the data services and closes the gaps identified in literature.

5.2 Overview of Data Services

First, an overview of the services that are presented and discussed is given. Figure 5.1 shows the architectural overview of the data services. This simplified presentation serves to give an overview of the next sections and how they are structured. First, the master data service will be presented. It is, as already mentioned in Chapter 4, responsible for the management of the structured and more static master data of different domains. The semantic description of the master data objects is extracted from the semantics service which is defined in more detail in Chapter 6.

The second part of this chapter presents and discusses the measurement services. They are depicted as a single block in the figure. However, as designed in the last chapter, each form of measurement has a dedicated service implementing the specific features tailored to the measurements that are expected. An API features read operations that support applications with the delivery of measurement data including basic statistics.

Last, the digital asset service manages additional data that is stored as unstructured and binary files. The API of this service does not focus on the manipulation or filtering of the data, however, it provides an interface to manage these files. Both, the measurement services and the digital asset service have a master data object describing the relationship with possible master data objects. The definition of such a master data object is given in the respective sections in this chapter. Each service features a RESTful interface. The APIs are defined similarly in order to have unified conventions. They are designed following best practices [GG+16]. Each service has its own database as it is defined by Sam Newman [Ne15]. The

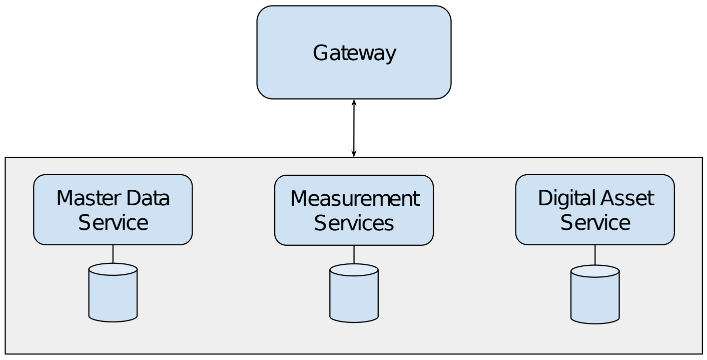


Figure 5.1: Overview of Data Services

master data service will be presented first since it has the most influence on other services as it can be seen in the relation view of the master data bounded context in Section 4.8.

5.3 Master Data Service

This section presents the first detailed concept of a service. The master data service is one of three data managing services (the measurement service is seen as one service for a better overview). The first part describes the data format and standards used and the second part discusses the design of the RESTful API in detail.

5.3.1 General Concept

In Chapter 2 the term master data has already been introduced. The service managing the master data is responsible for the storage and provision of the data. The API provides different mechanisms that allow an application or a user to query and filter the data accordingly. Master data objects are only annotated with their type. However, the type can include much more information in order to add a semantic context to the associated master data object. Master data is clearly structured using the JSON format. However, besides the hierarchical structure, there is no semantic information attached to it. A promising standard that helps to provide this information is JSON Schema. It provides the application with data types, human readable descriptions and a clear identifier. As described in Chapter 6, there is a dedicated service to manage this information. However, the semantics service is discussed in Chapter 6 in more detail. The next section describes the design of the API in detail.

5.3.2 API design

There are many best practices and API design principles already defined. Therefore, it is highly favorable to use such principles and avoid designing the whole API from scratch. The following section will present a few API standards and selects the most suitable for the master data service. The following standards will be analyzed:

- **JSON API**: The JSON API [JSONAPI] standard was first drafted in 2013 and is closely related to the popular JavaScript framework Ember.js[1]. It defines Hypermedia as the Engine of Application State [HATEOAS] aspects using a link property. Additionally relationships can be defined and also included as sub resources. Some other features include pagination, filtering and sorting. However, the standard does not define the REST API URL formats. It only defines the layout of the request body. The advantage is that the standard is quite mature and provides lot of features. However, the downside is that the format is quite verbose.

- **OData**: OData [PH+14] is a standard of the OASIS consortium. Additionally it is ISO/IEC approved. The standard has a more lightweight data format that is not as verbose compared to JSON API. It also supports to fetch nested resources. A difference in regard to JSON API is that the URL design is completely included. However, this restricts the capabilities of the service to the ones of the standard. Additionally, the API can not be freely designed following best practices.

- **GraphQL**: The GraphQL standard [PB18] is the youngest of the three standards. It has a different approach that is not directly related to REST. It defines an alternative to REST mainly using the body of the HTTP requests to control the requests or actions that have to be triggered.

After a short presentation, an analysis shows that all three standards are viable, however, the most flexible in terms of supported features and design of a RESTful API is the JSON API standard. It defines a format that allows to communicate generic JSON objects including optional relations to other objects. Furthermore, the standard does not define the format of the URLs which keeps the design open and flexible. Hence, the basis of the API design is defined by the standard. However, since it does not define all features the master data service requires, the standard is only a basis and the API has to be derived from it in order to be able to fulfill all the features and to have an API that is fully defined. Furthermore, JSON API does not completely define how pagination, filter etc. work. Therefore the next paragraph

[1]https://emberjs.com/

Listing 5.1: Example of a Master Data Object Representing a Measurement Station

```json
{
  "data": [{
    "type": "building",
    "id": "1",
    "attributes": {
      "name": "Office Building",
      "number": 449,
      "campus"; "campus north"
    }
  }]
}
```

will introduce additional concepts expanding the basics in order to get a complete design for the master data service. In the following, the example depicted in Listing 5.1 is used in order to better understand the different API aspects. The figure shows a single building master data object in the JSON API format.

5.3.2.1 Pagination

Starting with the first feature that only has a small concept: pagination. The pagination usually needs two parameters in order to get a part of a list. There are different possible solutions. The JSON API standard suggests three different strategies:

- page[number] and page[size]
- page[offset] and page[limit]
- page[cursor]

All three strategies are viable. However, defining an offset and limit leaves the frontend the possibility to work with an index-based system or to locally compute pages and their size. The third strategy needs more management code in the frontend. However, it might be a good strategy for other services defined in this thesis. Therefore, the second strategy is used and also the query parameter called page will be used as suggested by the standard. Figure 5.2 shows an example of a request that has an offset of 10 and a limit of 20 therefore returning a list of master data objects from the 10th item to the 29th item. The base URL is in this case only a placeholder since it is not yet discussed (see Section 5.3.3 for more detailed information). The square brackets are displayed non-encoded for better readability. They have to be percent-encoded in order to define a valid URI.

```
/masterdataBaseURL?page[offset]=10&page[limit]=20
```

Figure 5.2: Example of the Pagination Feature

```
/masterdataBaseURL?sort=name,-number
```

Figure 5.3: Example of the Sort Feature

5.3.2.2 Sorting

The second feature is sorting. It is important to be able to sort the master data objects according to specific JSON properties or even nested ones (properties inside a nested object). As query parameter, the standard suggests the *sort* parameter which is also used in the presented concept. Furthermore, the concept allows to sort multiple fields which are denoted by the field names separated by commas. Another important aspect when sorting lists is the definition of the order (ascending or descending). The order is always ascending unless a minus is placed before a property which changes it to descending for this property. Figure 5.3 shows an example that sorts the result list ascending by name for results with the same name these sublists are sorted descending by number. It is important to define the properties in the correct order to achieve the desired result.

5.3.2.3 Field Selection

The third feature is the ability to select properties that are included or ignored. The JSON API standard suggests the query parameter name *field*. Each property added to the list, is included in the returned objects. This results in a whitelist which means that no other properties are included except the ones that are mentioned. Similar to the sort feature, the field feature also has the ability to set a minus sign. This results in the properties not being included (blacklist functionality). The minus sign in front of the list changes the list into a blacklist. A mixture of white- and blacklist has no defined behavior and results in an error. Figure 5.4 shows an example of the feature. The example shows a request that only returns building objects that have their *name* field and their *number* field included. The figure also shows the response of the request. It illustrates that only these two fields are returned. Note that this object is not a valid response since it is not JSON API compliant.

```
/masterdataBaseURL?field=name,number

{
  "name": "Office Building",
  "number": 449
}
```

Figure 5.4: Example of the Field Feature including a Response Object

5.3.2.4 Filter Concept

The next API feature is the *filter* functionality. It is crucial in order to reduce the amount of objects requested based on the filter criteria. The JSON API specification does not give any suggestions for this query parameter besides the name. Additionally, it is the most complex one in the presented concept. The parameter is used to select a subset of the requested list of master data. There are several rules, mainly based on the type of the field that is used to filter the list. First, the following list shows different types of fields that each have a specific grammar that defines how to filter according to the type:

- string
- enumeration
- boolean
- number
- datetime
- geo

First the main structure of the filter is described. Figure 5.5 shows this structure. The filter is always applied to one specific field therefore the field name is in brackets after the filter. The first part of the query parameter's value is the name of the filter operation. E.g. for the type *string* there is the operation *equal* (or in short **eq** is available). This operation can have multiple parameters which are defined after the operation and put into brackets. After the operation the value is defined. This value depends on the filter operation and the field type. For string and the equal operation the value is the string that is used to compare the field

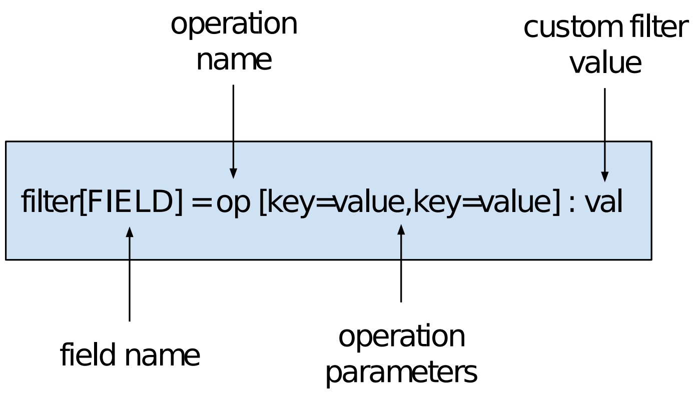


Figure 5.5: Structure of the Filter Feature

value with. In the following the different types and their filters and operations are presented and discussed.

String Filter

This is the most common filter since the string type is the most common type for master data properties. Table 5.1 shows an overview of operations and example values. Most of the operations are basic string operations and self-explanatory. The *like* operation has the option to define wildcards that allow a broader search. The optional parameter **ic** stands for ignore case. It allows to define if the operations ignore the case sensitivity of the provided value.

Operations	**Operation Desc.**	**Example Value**	**Parameters**
eq	field is equal to value	f[name]=eq:John	ic=true/false
ne	field is not equal to value	f[name]=ne:Smith	
start	field starts with value	f[name]=start:Jo	
end	field ends with value	f[name]=end:hn	
contains	field contains value	f[name]=contains:oh	
like	field is like value	f[name]=like:*h*	

Table 5.1: Overview of Possible String Filter Operations

The default operation is always emphasized in the tables for the following definitions. By omitting the operation name in the API, this default operation is implicitly chosen. Additionally, in the following tables, the *filter* key of the API is abbreviated by *f* in order to safe space in the tables. However it is not intended that this is a valid API definition.

Enumeration Filter

The filter for enumerations is not very complex since only the operations checking for equality and inequality are available. In Table 5.2, the values RED and GREEN are part of an enumeration.There are no parameters defined.

Operations	**Operation Desc.**	**Example Value**	**Parameters**
eq	field is equal to value	f[color]=eq:RED	/
ne	field is not equal to value	f[color]=ne:GREEN	

Table 5.2: Enumeration Filter Operations

Boolean Filter

The boolean filter is the most basic. It only consists of one operation (see Table 5.3). There are also no parameters defined.

Operations	**Operation Desc.**	**Example Value**	**Parameters**

eq	field is equal to value	f[valid]=eq:true
		f[valid]=eq:false
			/

Table 5.3: Boolean Filter Operations

Number Filter

The next filter is the number filter. The operations (see Table 5.4) compare the value to the specified field. No parameters are defined.

Operations	**Operation Desc.**	**Example Value**	**Parameters**
eq	field is equal to value	f[number]=eq:42	
ne	field is not equal to value	f[number]=ne:3	
lt	field is less than value	f[number]=lt:43.7	/
le	field is less or equal than/to value	f[number]=le:42	
gt	field is greater than value	f[number]=gt:5	
ge	field is greater or equal than/to value	f[number]=ge:12	

Table 5.4: Number Filter Operations

Datetime Filter

The datetime filter is important in order to have the complex datetime operations (Table 5.5) as a modular concept. This leads to a single place to convert different date and time formats. The operations compare dates and check if a value is in between two moments in time. There are two parameters. The first allows to specify the format that the date and time is represented in. This leads to the support of any datetime format. The second parameter allows to change the delimiter that is used to separate the two dates specified for the between operation.

Operations	Operation Desc.	Example Value	Parameters
eq	field is equal to value	f[time]=eq[format=unix]: 1488812400	/
before	field is a date before value	f[time]=before:2017-03-06 17:00 +01:00	
after	field is a date after value	f[time]=after:2017-03-06 15:00	
between	field is a date between the first part and the second part of the value	f[time]=between:2017-03-06 15:00,2017-03-06 17:00	

Table 5.5: Datetime Filter Operations

Geo Filter

An important aspect of master data objects in the environmental and energy fields is that the different sensors, measurement devices and many other assets have a specific location or even geometry (e.g. building). This location can be interpreted as spatial information. The geo filter supports queries that target such information. There are different types of geographical constructs. The different geo types include points, lines, polygons, multiple polygons and circles. The first column of Table 5.6 describes the allowed type for the specified field of the geo filter.

With the geo filters, all filters are discussed. The concept of filters might be extended by additional types, operations and parameters. However, it is important that filters are not mixed up with aggregations. The difference is that filters only select entries from a list whereas aggregations change the result according to the specified aggregation type. In the following the aggregation concept is presented. Since the operations are much more complex than the previous ones, this filter has no default operation.

Type of Field	Operations	Operation Desc.
all	intersects	field intersects with value
all	within	field is located within the value
all	disjoint	field is disjoint with the value
all	contains	field contains the value
point	contains	field has a specific distance to value

Table 5.6: Geo Filter Operations

5.3.2.5 Aggregations

Aggregations defined for the master data service are only viable when the technology behind the microservice has already built-in functionality for that aggregation. The same goes for the other services. All aggregations that are not supported by the database of persistence technology used by the microservice should be shifted to the statistics service which has the primary role to compute such data. These considerations have to be done each time a service is implemented using a different database technology.

The master data service has currently one aggregation that might be provided by the service considering a database technology like Elasticsearch[2] that has built-in support for it. The aggregation that is defined for the master data service is the ability to compute summaries (called terms aggregation) based on a query. The aggregation of terms is a concept that is important for applications with need for summaries of large data sets. It shows the different values for a specific property and their number of occurrence in the requested list. In Figure 5.6, the structure of an aggregation request is shown.

The structure of the aggregation query parameter is very similar to the filter parameter except that the name of the query parameter changed from *filter* to *aggregation* and that there is no value that has to be specified since the aggregation is always performed exclusively on the content of the fields. All arguments of the aggregation are provided using parameters. In Table 5.7 the terms aggregation is illustrated. Since the operation has no parameters that have to be defined, the operation term is the only value of the query parameter.

[2]https://www.elastic.co/products/elastic-stack

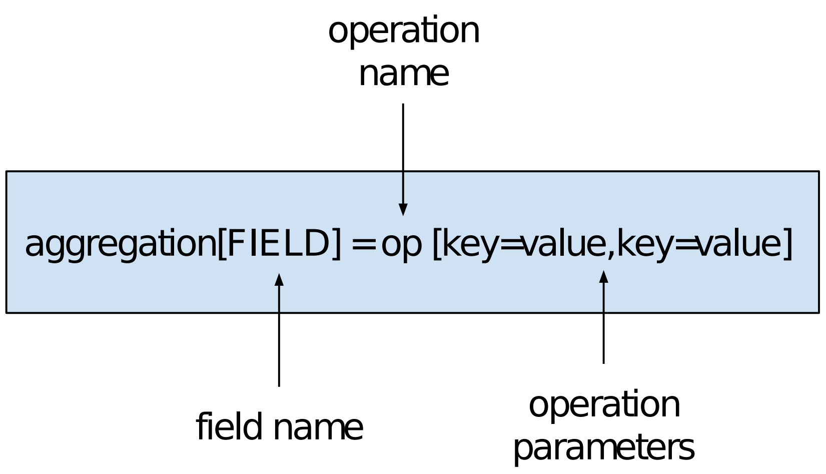


Figure 5.6: Structure of an Aggregation Query Parameter

Listing 5.2: Terms Aggregation Response Object

```
{
  "aggregation": {
    "terms": {
      "RED": 9,
      "GREEN": 10,
      "BLUE": 2,
      "BLACK": 42
    }
  }
}
```

Operations	Example Value
terms	aggregation[color]=terms

Table 5.7: Terms Aggregation

Listing 5.2 shows a response object. The response object consists of the term aggregation result. In this case, this is the frequency of occurrence of the different color terms.

5.3.2.6 Relationships

Besides the filters and aggregations of masterdata objects, also their relationships among themselves are important. In order to define the concept of relationships between master data objects it is crucial to present the JSON API closer. Therefore, Listing 5.3 shows an example

Listing 5.3: Example of a Master Data Object

```json
{
  "data": [{
    "type": "building",
    "id": "1",
    "attributes": {
      "name": "Office Building",
      "number": 449,
      "campus": "CAMPUS NORTH"
    },
    "relationships": {
      "rooms": {
        "data": [
          {"id": "599", "type": "campus;1.0.0;room"},
          {"id": "600", "type": "campus;1.0.0;room"}
        ]
      }
    }
  }]
}
```

representing a response body with a master data object according to the JSON API standard. The example shows a complete response object describing a particular building. JSON API defines the different properties giving each object a specific structure. The attributes represent the actual data and the relationships are the relations the building has. In this case the building is related to two different rooms. Each room represents another master data object and is described by the type *campus;1.0.0;room*. This representation defines the name of the schema (*room*), the schema class (*campus*) and the version of the schema class (*1.0.0*). These concepts are further described in Chapter 6.

A downside of the JSON API is that a relationship can only be defined on the root object. In other words, a relationship can not be defined on a property that is in a nested object. The solution for such a nested relationship is to define another intermediate object. This object can have the relationship in question at its root level. On the one hand, this leads to more separated objects but on the other hand each object is clearly structured. Each relationship defined in this way is a reference to another master data service. However, a concept is needed to define a reference to data from other services, e.g. one of the measurement services or the digital asset service. The next paragraph defines this concept.

To set a reference to external data, there are two viable possibilities. The first option is expanding the JSON API by defining a way to have external relationships besides the existing

Listing 5.4: Example of a Datasource Definition

```
{
  "data": [{
    "type": "datasource",
    "id": "2",
    "attributes": {
      "url": "/dataservice/{{id}}",
      "parameters": {
        "id": "89"
      }
    },
    "relationships": {}
  }]
}
```

relationship concept. The downside of this option is, however, that the standard has to be changed. The second possibility is to define a specific master data type that represents the external relationship as actual data. This preserves the standard and therefore allows to use the existing features of the master data service to manage external references. In Listing 5.4 an example for an external relationship object is given. The datasource is its own type that defines attributes representing a RESTful request. In the example the two most important attributes are shown: the URL-template and parameters that might be included in it. The template can be parametrized in order to give an application the possibility to change them and use the information in a more flexible way. *URL* and *parameters* are only the basic attributes. Additional attributes might be specified in order to define the request in more detail. For example the HTTP method or HTTP header fields might be of importance. However, these can be omitted if the request uses the GET method and no special header fields.

5.3.3 REST API

After having defined all of the features of the master data service, this section describes the final REST API. The first part of the REST API is the base URL. Figure 5.7 defines the structure of the base URL. The domain name is just a placeholder for the entry point of the deployment environment. The service name is abbreviated with the name of the service but without the explicit mention of the word service since it would be too long and unnecessary in this context. The next part is the version of the service. The complete version consists of three numbers (major, minor and subminor) but only the major version is included in the URL since only the major version is incremented with API breaking changes. The next two

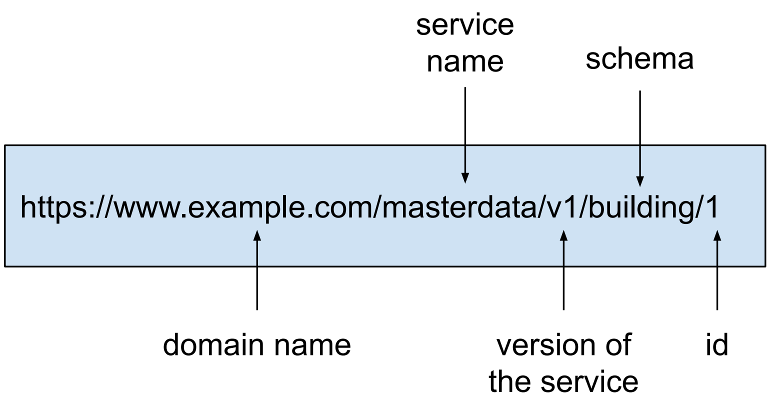


Figure 5.7: Structure of Master Data URL

parts of the URL is the type and id that are also included in the responses of the service that was already presented. Besides the latter, the base URL of the other services is defined very similar.

The CRUD operations for the management of master data are defined following best practices for RESTful services. Table 5.8 gives a short overview of these operations.

Operation	**HTTP Method**	**Example URL**
Create	POST	https://www.example.com/masterdata/v1/building/
Read	GET	https://www.example.com/masterdata/v1/building/1
Read All	GET	https://www.example.com/masterdata/v1/building/
Update	PUT	https://www.example.com/masterdata/v1/building/1
Delete	DELETE	https://www.example.com/masterdata/v1/building/1

Table 5.8: Master Data Service RESTful API Requests

The create and update requests include a body that follows the JSON API specification. The same applies for the response of the read and read all operations. The read operation is of course simplified in the table since all of their features have already been presented in the last section. After having discussed the whole design of the master data service API, the next service is presented.

5.4 Measurement Services

This section describes the concept of the three measurement services that were identified during the design process in Chapter 4. First, a common concept including the API design for the different services will be presented. The different services are not discussed in detail. The separation is meant to improve performance and allows each service to select the optimal persistence layer. However, the specific services are only adapters to a general API defined in the next sections. The goal is to keep the concept simple but at the same time provide high performance.

5.4.1 General Concept

In Chapter 2 three different types of time series have been presented. These three types are the technical reasons why the three services are suggested by the design process of the reference architecture. In order to have three services that are each optimized for the specific type of time series, it is important that they use an appropriate database for their needs. Table 5.9 shows an overview of possible databases for each type.

Type	**Database Examples**
univariate time series	*OpenTSDB*, InfluxDB, TimescaleDB
multivariate time series	*InfluxDB*, TimescaleDB
sampled values	InfluxDB, TimescaleDB, *HDFS*, S3

Table 5.9: Time Series Databases for Different Types of Measurements

The table is only a suggestion for databases. There exist a large number of time series databases which will not be evaluated by this thesis in order to find the most suitable database for each service. The table shows that each time series type has different options. The most suitable after a brief evaluation are highlighted. For univariate time series database OpenTSDB[3] is suitable since the database is optimized for univariate time series. Multivariate time series are the most common and, therefore, the most popular database technologies might be a suitable choice. For sampled values, it is not suitable to store the values in a time series database since the amount of values is very large. Therefore, a distributed file system or object store is more suitable. In this case HDFS is selected since it also has the possibility to get insight from the data using data analytics advantages of a Hadoop cluster. This short analysis shows that for all three types different databases with their specific features are selected. However, the services have a lot of common functionality, since all are working with time series after all. Therefore, the following concept introduces a common service that provides a RESTful interface and a streaming API that can be used for all three measurement

[3]http://opentsdb.net/

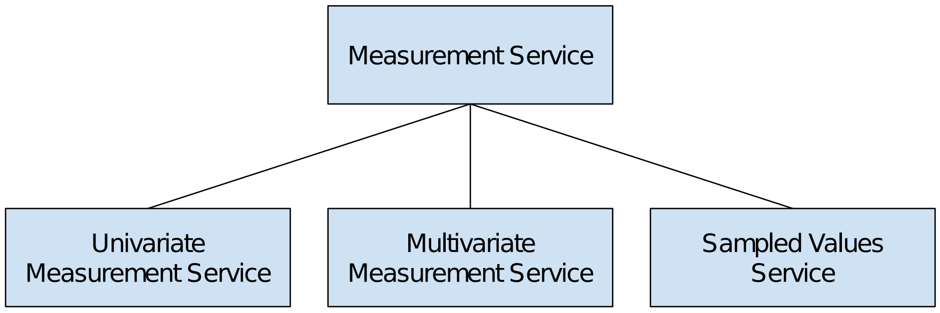

Measurement Service
Univariate Measurement Service
Multivariate Measurement Service
Sampled Values Service

Figure 5.8: Overview of the Abstract Measurement Service and the Specific Services

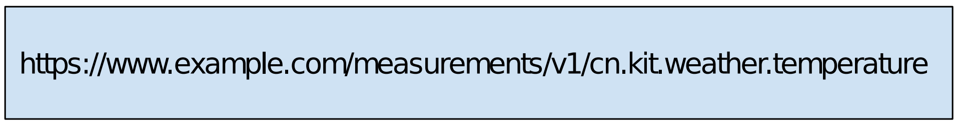

https://www.example.com/measurements/v1/cn.kit.weather.temperature

Figure 5.9: Base URL of the Measurement Service

types and their respective services. The APIs hide the complexity of dealing with a specific database technology.

As mentioned in the introduction of this chapter, a single API for all of the measurement services is favorable. Therefore the concept in Figure 5.8 describes an additional service that serves as a facade for the three specific services. This concept allows the measurement service to provide a general API that is suitable for simple requests that are valid for all three services. The API requests are forwarded depending on the type of measurement that is operated on.

5.4.2 API Design

Unlike the API of the masterdata service, it is not derived from a standard. Therefore, modern time series databases were analyzed in order to design an API that uses best practices. In the following the RESTful GET request will be described. Similar to the master data service it is the most complex request. The measurement service follows a similar concept for the API design. First, Figure 5.9 illustrates the base URL. The structure is the same as the structure for the master data service. The exemplary identifier is in this case a typical identifier for measurements as used in multiple time series management systems: a set of strings joined with dots that describe the nature, the location and kind of the measurement.

5.4.2.1 Measurement Operations

Next, the query parameters of the GET request will be presented and discussed. In order to efficiently query measurement data, the operations have to be identified. As already partly identified in the ubiquitous language the following operations are of importance:

- **time interval**: Define an interval of the data that is requested.
- **downsampling**: Controls the amount of data points a requested interval has. This parameter allows to reduce the amount of data points by indicating a time interval for which all data points are bundled. This parameter is typically used in conjunction with an aggregation that computes ,for example, an average or maximal value for each of these time intervals. Most of the time series databases provide this operation.
- **tag**: Defines a key/value pair for a measurement point. This helps the measurement service to find related data points. Tags are used when very similar measurements are recorded that logically belong to the same kind. In that case, all measurements might be stored with the same identifier but different tags. An example is the measurement of two temperatures inside two different rooms in the same building. Since both parts of the measurement are recording the same unit and in the same context (building), they have the same identifier which can include an identifier of the building. A suitable tag for this case might be the pair ”*room*” = *ROOM_NUMBER*. Tags can help to keep data together in the database in order to perform efficient aggregations. However, it is important to understand when to use tags.
- **reduce operation**: Most of the time series databases that were looked at are able to perform simple aggregations e.g. the computation of a maximum or minimum value or the average of the requested time series. These simple operations can be summarized as operations that reduce a sequence of values to a single value.
- **rate**: This parameter is important when the measurement is a value that represents for example a counter of a specific metric and the rate of change is needed. In that case the parameter can derive the measurement to get the change over time.

Time Interval Filter

In the following these operations have to be mapped to the API of the measurement service. In order to have a similar API design as master data service, it is favorable to reuse the

```
filter[time] = between[format=unix]:605025480,636561480
```

Figure 5.10: Time Interval Filter using the Between Operation

```
filter[room] = eq:231
```

Figure 5.11: Example of the Filter by Tag Operation

concepts of fields, filters and aggregations. First, we look at the time interval operation. Since it selects a specific range of the whole measurement, it is a value along the time axis. Figure 5.10 shows an example for this filter. As seen in the section of the master data service, the filter of datetime fields allows to select values that lie in between two timestamps. Most of the time series databases work with unix timestamps, therefore, this representation is also used in the present concept. However, this raises the need to store additional information of the measurement at a second place, for example the used time zone. A metadata concept for this will be described later in this chapter.

Tag Filter

The next filter operation is the selection of measurements by tag. Since the tag is a key-value combination, again filters can be used. In this case, the tag key is used as field name and the tag value is compared to the given value. Figure 5.11 shows an example of the filter with the concrete example of the building and room mentioned above. The tag fields are of data type string and therefore all string operations are permitted for this type of filter. However, it depends on the underlying database if all operations are supported.

Downsampling

The next operation is downsampling. Since it changes the data and is not only a selection of a subset of the data, it belongs to the category of aggregations. Figure 5.12 shows an example of such a query parameter. The aggregation is performed on the time field and the name of the operation is *downsample*. This operation has two parameters which are exemplarily shown in the figure. The interval controls the size of each bucket that is bundled to a single

```
aggregation[temperature] =downsample[interval=30m,fill=0]
```

Figure 5.12: Example of the Downsample Aggregation

value. In this case 30m means 30 minutes. For each 30 minute interval, the values are grouped together following one of the aggregation functions presented in the next paragraph. When no aggregation function reducing the amount of data points to one is indicated, the computation of the average is used as default. The second parameter defines the behavior when having no data points in the interval at all. The different options are the following:

- **none**: leaves the interval empty
- **null**: maps the keyword null to each interval that is empty
- **linear**: performs a linear interpolation in order to determine the missing value
- **previous**: copies the previous value
- **[any number]**: fills in the specified number

In the figure, all empty intervals are filled with the number 0. Note that *linear* and *any number* are only possible if the measurement has numbers (float, double, integer, ...) as data type for their values. The data type string for example is not possible and returns an error.

Reduce Aggregation

The next aggregation is a range of different aggregations that all reduce the requested interval (or intervals in case of downsampling) to one value. An example for this aggregation is given in Figure 5.13 with the max aggregation that reduces the data set to its maximum value. Similar to the max aggregation there are a number of such simple computations e.g. average, min, count, etc. The available aggregations depend on the data type of the requested value. String aggregations are also possible. The count aggregation is independent of the data type.

```
aggregation[temperature] = max
```

Figure 5.13: Example of the Max Aggregation

```
aggregation[temperature] = rate
```

Figure 5.14: Example of the Rate Aggregation

Rate Aggregation

The last aggregation that can be performed on the data is the rate aggregation. The fields this aggregation is applied to, have to be of a numeric data type. Figure 5.14 shows an example. This aggregation does not change the number of values, however, it changes the absolute values to the rate of change.

Fields Query Parameter

Another query parameter that is defined for the master data service and is also needed for the measurement service is the selection of fields. The parameter has the same semantics as in the master data service. The fields that can be included or excluded are the measurement fields, the time field and the tag fields.

5.4.2.2 Data Format

For now, the presented API is specifically designed to be the same for all measurement types. However, at some point the different types of measurements have to be treated separately in order to have efficient services. This separation happens on the level of the data format used to transport data from client applications to the services and vice versa. Since univariate and multivariate measurements have very similar structures, both use a similar format. In Listing 5.5 this format is presented. It is verbose but has a clear structure. The timestamp is in the unix format and considered to be in the UTC timezone. Each field has a name and a value. The amount of fields in the object is the only difference between a univariate and a multivariate measurement. A possible simplification of the format of a univariate measurement is to omit the fields object and only add a key called "value" that holds the only

Listing 5.5: Example of Univariate Data Format

```
[{
  "time": 605025480,
  "fields": {"temperature": 23.6},
  "tags": {"room": "231"}
},{
  "time": 605026380,
  "fields": {"temperature": 23.4},
  "tags": {"room": "231"}
}{
  ...
}]
```

value. The presented data format is used to write data to the measurement service and to read data from it.

5.4.2.3 Separation of Services

First, writing data to the measurement service in this format needs some more information in order to determine the specific service responsible for the request. Therefore, new custom content types are defined that are sent using the HTTP Content-Type header field. The value of the content type controls which service is used to process the data. The following content types are defined:

- application/vnd+gmb.univariate-measurement+json
- application/vnd+gmb.multivariate-measurement+json
- application/vnd+gmb.sampled-values+json

The content types give an idea how the concept works. However, more content types might be added in the future, especially to support more efficient data formats. For sampled values, the data format might even be binary files. The presented approach defines the handling of RESTful requests for different services. In the next paragraph, streaming is defined.

5.4.2.4 Streaming

In order to support streaming for the different services, a similar concept is used. Figure 5.15 shows the communication flow that initiates a stream. First, the client sends a synchronous RESTful request to the measurement service in order to request the stream information. This

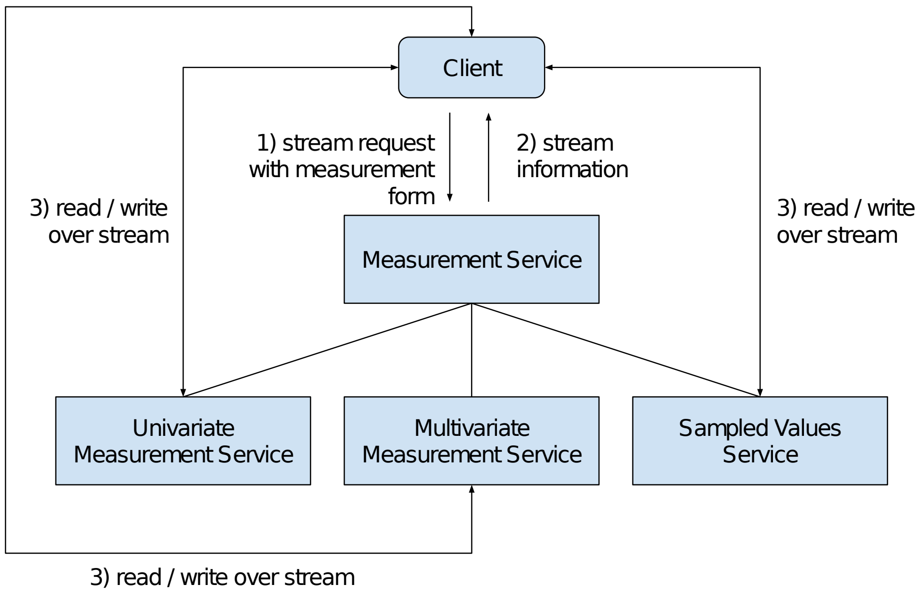

Client
1) stream request with measurement form
2) stream information
3) read / write over stream
3) read / write over stream
Measurement Service
Univariate Measurement Service
Multivariate Measurement Service
Sampled Values Service
3) read / write over stream

Figure 5.15: Initiation of Stream Communication

request includes the measurement type that requested using the Accept-Header similar to the previous section. The client can select between one of the three measurement services in the background. The response of the measurement service includes all information in order to contact the specific service. Bypassing the measurement service facade for streaming allows the client to efficiently communicate in the data format that is directly supported by a specific service. This concept allows, on the one hand, to have a comfortable access point with the Restful API of the measurement service facade for simple applications that have no special performance requirements. On the other hand, the streaming features of the specific measurement services can be used for faster and a more compact communication.

5.4.2.5 Response Format

Another part of the concept that was not yet discussed is the response formats for aggregations. Listing 5.5 showed the general format that might also be used as basic (verbose) response format. However, aggregations especially reduce aggregations have other semantics and return a different response. The format of such a response is illustrated in Listing 5.6. In this example, two aggregations were requested. The *max* aggregation on the temperature field returns the maximum value which is in this case 23.6. The timestamp is the associating timestamp for the max value. The *avg* aggregation returns the average of all values, in this case 23.5. The timestamp can not be defined since the average is computed on the whole

Listing 5.6: Response Format of a Reduce Aggregation Request

```
[{
  "time": 605025480,
  "max": 23.6,
  "field": "temperature",
  "tags": {"room": "231"}
},
{
  "avg": 23.5,
  "field": "temperature",
  "tags": {"room": "231"}
}]
```

interval that was set in the filter. In both scenarios, the tags that were used as filters in the request are added.

5.4.2.6 Relationships

The last missing piece of the concept is the relationship with other data and the management of additional metadata. As we have seen above, the time zone, for example, is not stored in the measurement services. Additionally, information like the unit of measurement and possible multiplication factors are not stored in the measurement services directly. Since the master data service already has a way to store structured data with a predefined schema, also the metadata is stored in this service. This also immediately solves the relationship problem since the master data service has the data source concept to reference external services such as the measurement service. However, the metadata is not included in the data source object but there are two different objects that reference each other. Listings 5.7 and 5.8 illustrates an example with both master data objects and their respective references.

5.4.3 REST API

To wrap up this measurement service section, the final REST API methods are presented. Table 5.10 shows the most important REST operations. The domain is omitted in the example URLs for better readability. Similar to the master data service, best practices are satisfied. The POST request is used to create new data points and also new measurements (that are empty when first created). In this case, the measurement id is sent in the body of the request. The GET request querying a specific time series is discussed in detail above. A second GET request fetches all of the measurements and gets an overview. The PUT request corrects data

Listing 5.7: Example of Measurement Metadata

```
{
  "data": [{
    "type": "measurement-metadata",
    "id": "2",
    "attributes": {
      "unit": "°C",
      "factor": "1",
      "time-zone": "+02:00"
    },
    "relationships": {
      "datasource": {
        "data": {
          "type": "datasource", "id":2
        }
      }
    }
  }]
}
```

Listing 5.8: Example of Measurement Data Source

```
{
  "data": [{
    "type": "datasource",
    "id": "2",
    "attributes": {
      "url": "/measurement/v1/{{id}}",
      "parameters": {
        "id": "cn.kit.weather.temperature"
      },
      "mimeType":
        "application/vnd+gmb.univariate-measurement+json"
      },
    },
    "relationships": {}
  }]
}
```

points which can be overwritten in that case. Finally, the DELETE request deletes all the data points and the measurement itself.

Operation	HTTP Method	Example URL
add new measurement	POST	/measurement/v1
add new data points	POST	/measurement/v1/cn.kit.weather.temperature
query a time series	GET	/measurement/v1/cn.kit.weather.temperature
get a list of all measurements	GET	/measurement/v1
correct a data point	PUT	/measurement/v1/cn.kit.weather.temperature
delete a measurement	DELETE	/measurement/v1/cn.kit.weather.temperature

Table 5.10: Measurement Service RESTful API Requests

5.5 Digital Asset Service

The third type of data that was identified in the environmental and energy context is the digital assets type. It summarizes all data representing files or binary files. These files are considered to be stored without necessarily knowing their structure or providing specific features for a specific type of file. Other than the previous services, this concept is quite short since there are already very promising technologies that can be used. The first section presents the short concept and the second section discusses the integration in the rest of the architecture with a relationship concept.

5.5.1 Concept and API

A standard that already fulfills all of the requirements for the management of digital assets is the Content Management Interoperability Services (CMIS) standard [CMIS2012]. The mature standard drafted by OASIS defines a rich API allowing the management of documents. There are many applications that implement the protocol both open source and commercial ones. The API is based on REST and supports the JSON format. Besides simple CRUD operations, the standard features access control mechanisms, versioning, folder structure and relationships between objects. Not every application implements the entire features. However, the standard is a good fit for the reference architecture since it already defines many useful features and does not require the design of a new concept for the digital asset service.

Listing 5.9: Example of a Data Source Pointing to a Digital Asset Document

```
{
  "data": [{
    "type": "datasource",
    "id": "2",
    "attributes": {
      "url": "/digitalasset/v1/cmis/versions/1.0/atom/content?id
          ={{id}}",
      "parameters": {
        "id": "48805f6f-2b5b-4554-9802-2fc362b520ff"
      }
    },
    "relationships": {}
  }]
}
```

5.5.2 Integration into the Rest of the Architecture

Similar to the measurement services, the digital asset service uses the same data source object in the master data service to create a relationship between both services. Listing 5.9 shows an example of a data source that includes a GET request pointing to the digital asset service. The further reference architecture of this service is defined in the CMIS standard.

5.6 Conclusion

This chapter presents the core of the reference architecture in detail. The data services consist of three types: the master data service, the measurement service and the digital asset service. The separation of services has been defined in the previous chapter (Chapter 4). Both, the master data service and the measurement service have complex APIs to access the data from a client application. An additional streaming concept is presented for specific measurement services. The REST APIs consist of different concepts that introduce mechanisms for filtering and aggregating data. The services partly profit from mature standards. Especially the digital asset service is defined to use the CMIS standard that completely describes the API. Another concept that has been presented is the definition of relationships between master data objects and also data from different services. The chapter solves some of the requirements that were presented in Chapter 3. The rest of the requirements are solved in the following two chapters (Chapter 6 and Chapter 7). The services are evaluated in more detail in Chapter 8. The next chapter completes the reference architecture with the presentation of two additional services that support the applications with additional semantic and analytical information.

6 Semantics and Analysis Service

The remaining parts of the reference architecture designed in Chapter 4 are the semantics service and the analysis service. These two services supporting the data services with additional functionality will be discussed in this chapter. First the semantics service will be presented that stores additional semantic information in order to support the master data service and client applications. Second, the analysis service provides analytical functionality in order to compute statistics for the master data service and the measurement service. Such information is important for applications in order to get additional insight into the data in a quick and efficient way.

6.1 Scientific Analysis

As seen in the previous chapter, the data services, that are designed in a generic manner, allow to manage data from different applications since they are not tied to specific requirements of a single application. The measurement system domain shows a clear need for semantics describing this data in order to have the possibility of interpreting the data in a specific context. There are publications that suggest microservice architectures with separate semantic-aware services. After an analysis of these publications, two types of architectures were identified.

First there are architectures that implement so called semantic enrichment [SH+19, JA+18]. These are services that map the semantic information to the according data. This enriches the raw data with a semantic context. The advantage of such an architecture is that every data object is accompanied by its semantic description and has a clear meaning for the requesting application. A disadvantage is that the semantic services that implement the enrichment are closely coupled to the data services. The architecture is only working when the semantic services are available since no application use the raw data without semantic context.

A second type of semantic-aware architectures implement the according services loosely coupled and independent from the data services [GM+17, KJ+15, KK+16]. These architectures allow applications to separately access the raw data and their according semantic information. Such an architecture is more flexible and versatile. However, applications have to be smarter in order to implement the semantic enrichment or similar functionalities themselves. This second architectural style, however, is also compatible with the backend for frontend or API gateway principles [Ne15] that allow to compute enrichments or aggregations in an abstract way without the need of strong coupled domain services.

As already figured out in Chapter 4, the presented reference architecture provides a semantics service that is separate from data services. Therefore, the second type relates to the present thesis. A second observation that was made during the analysis of publications is that the articles focusing on semantics-aware microservices are often using a full stack of ontology technologies. This includes the file format RDF, triple store databases, the web ontology language (OWL) and SPARQL which is an RDF query language. Such a stack allows for very complex queries and exploration of the ontology. However, in the present thesis the semantics-aware microservice is only a supporting one which does not require such complex technologies. Therefore a more lightweight alternative is suggested that manages the semantic information in the form of schemas. Each schema describes a specific object instead of managing a whole ontology. Designing the reference architecture in this way still allows to expand a specific architecture by both, the semantic enrichment features using an API gateway and the ontology management by adding services implementing the OWL and SPARQL features.

A second analysis of the literature has been done for the data analysis part. Similar to the previous analysis, the focus is on the measurement data and not on data analysis. However, first an overview of the different relevant literature is given.

The first two publications [YS+15, SC+14] define an architecture that uses a Big Data environment as backend in order to perform specific data analysis algorithms. However the architecture in [SC+14] additionally defines a microservice that encapsulates analytics as a service. The API is not further discussed in the publication. In both cases, the Big Data environment is set-up using the Apache Hadoop ecosystem. This ecosystem features many different tools for processing large volumes of data at a high velocity. Such an architecture requires a specific infrastructure that is not necessarily the same as the infrastructure needed for the deployment of microservices. However, there are already ideas on how to combine Big Data technologies and container virtualization environments [Na17].

The next two publications [KJ+15, GM+17] don't define such a general data analytics service or Big Data environment but define application-specific components that perform a very specific computation, e.g. the computation of energy related forecasts or calculations for an electricity grid. However, if the requirements are not known beforehand, an architecture that is meant to provide data analysis functionality has to be designed to be more generic than the architectures presented in these publications.

As already mentioned, the focus in the present thesis lies on the management of the measurement data and the data analysis is located in a supporting domain which suggests a single service that is an interface for different analysis queries. For more complex data analysis found in literature, the reference architecture can be expanded. However, this is not the focus of the current thesis.

6.2 Semantics Service

In energy and environmental applications, measurement data and associated master data are crucial for the users to get a better understanding of domain specific systems, processes and simulations. However, it is equally important to have a clear understanding of the structure of the data and additional semantic information in order to completely understand the data. This applies for users working with the applications as well as for the application itself to properly provide functionality to the users. In this section the concept of the semantics service providing this additional information is presented. First, the used data format is discussed, then the advanced concepts for schema versioning and referencing are presented and lastly the REST API is designed.

6.2.1 Data Format

As already identified in the design of the reference architecture, the semantics service has to manage three different domain objects: the schema, associated metadata and the author who is responsible for the schema. The schema is the core component. It defines data types, data structures and additional semantic information such as semantic descriptions. A data format, incorporating all three components in a JSON object is presented in Listing 6.1.

First, the metadata includes a version (note that the versioning concept is still to be discussed), the author's identifier and timestamps that represent the time the schema was created and last modified.

Second, the schema itself is included in the data format which validates against the JSON schema draft [JSON-Schema] indicated using the $*schema* property. Listing 6.1 illustrates an exemplary schema from the IEC 61850 [Ma06] standard is part of the International Electrotechnical Commission's (IEC) Technical Committee 57 reference architecture for

Listing 6.1: Structure of the Semantics Service Data Format

```
{
  "metadata" : {
    "version" : "v1" ,
    "author" : "fce34f99-3949-4dc6-965c-c48a6b35620d",
    "created" : "2017-03-06 16:00",
    "lastModified" : "2017-04-12 9:00"
  },
  "schema " : {
    "$schema" : "https://json-schema.org/draft/2019-09/schema",
    "title": "IEC 61850 - MMXU",
    "type " : "object",
    "properties": {
      "TotW": {"type": "number"}
      "TotVAr": {"type": "number"}
      "Hz": {"type": "number"}
    },
    "required": ["Hz"]
  }
}
```

electric power systems. In the example, an extract from the logical node[1] MMXU which represents measurements for electrical systems is presented. This logical node defines multiple properties but only three of them are mentioned in the schema. The following list briefly describes the three properties.

- **TotW**: The total real power (P) in a three-phase circuit, in watt (W).
- **TotVAr**: The total reactive power (Q) in a three-phase circuit, in volt-ampere reactive (VAr).
- **Hz**: The frequency of the power system, in hertz (Hz).

The logical node MMXU is a typical semantic resource that is used in energy systems to work with measurements. In the example the frequency is mandatory and the other two are optional which is only exemplary and not conform to the standard. A schema identifier is missing at this point since the structure of the identifier will be discussed later in this chapter. In the next section the concept for schema versioning is discussed.

[1]Top Level Categories of the IEC 61850 Data Model

6.2.2 Schema Classes and Versioning

In order to change or update a schema, it has to be ensured that the changes do not break applications using it. This can be achieved preserving the backward compatibility of the different schemas. However, if the schema changes and breaks the backward compatibility, the applications still need a possibility to access the previous version of a schema. There are multiple ways to implement versioning. The same problems are identified for the schema classes which bundle multiple schemas of a same context. Three versioning concepts are described in the following.

6.2.2.1 Schema-specific Versioning

The first option is to introduce a version for each schema. This allows for the most flexibility since all of the schemas have their own version and are therefore independent from each other. Any change to a schema results in a version update and applications decide which version to choose. A disadvantage of this method is that applications have to manage each version separately therefore resulting in a large version management on the client-side. Since applications might want to use the latest schemas in order to support the newest additions, they have to update their versions often which is nearly impossible when too many schema updates occur. Another problem is that the semantics service has to manage a lot of update operations. This also leads to inconsistent versions across the different schemas. One schema might have the version $v20$ and another might still be at version $v1$. Those two numbers have nothing in common since they are independent. This makes a consistent update of schemas hard and confusing. The second option tries to counter these disadvantages.

6.2.2.2 Schema Class Versioning

The next versioning option for the semantics service is to only define a version for a whole schema class. This has the implication that all schemas have the version of their class. This option reduces the management complexity to the amount of classes. Additionally, the schema classes are usually low coupled and therefore their different versions are not as confusing than different versions for each schema within one class. There are, however, also clear disadvantages of this method. An update of a schema results in the update of the whole class, which might lead to a high number of version updates. In this case, the applications still have a hard time to manage every new version and upgrade the software to implement changes. The number of updates has to be reduced. This is also used for standards or norms which also only have one version per document and not for every single schema. The last method tries to further improve this point.

6.2.2.3 Schema Revisioning

This versioning method is an addition to the schema class versioning. The idea is to combine the two previous methods and get all of the advantages. A downside of the schema class versioning is that there might be many updates of the schema class and therefore implicitly all schemas within. In order to counter this disadvantage, each schema can have a temporary version that is not visible for consuming applications but allows users to update a schema step by step without the need to create a new schema class version every time. This temporarily version is called *revision*. This method gives the application more flexibility when updating schemas. Applications consuming the schemas have stable versions for a longer period of time and a version update only occurs when it makes sense to reflect the changes of different schemas to a new schema class version.

This concept does not solve the problem of too many versions entirely since the user is still able to create many version for a schema class in quick successions. However, is not necessarily a larger problem since the JSON format only implies breaking changes if a key or a data type of an existing key changes. This means that adding new properties to a schema does not break any applications since they can just ignore these newly added properties.

The update to a new version might get triggered manually by an application or user using the REST API of the service. This allows to have full control over the mechanism. However, a more automated process is also possible which triggers a version update after a specific amount of time or after a certain number of changes. Such an automated process makes sense when the schemas are not supervised by users that can manually initiate version updates. Furthermore, the format defining the metadata has to be adapted to the new concept. Listing 6.2 shows that the property *version* (mentioned in Listing 6.1) was substituted with the property *revisionNumber*. The schema class version is not included in the metadata since it is the same for the whole class. It is included in the URI of a schema. This will be presented later in the current chapter.

6.2.3 Schema Referencing

For complex schemas, it is important to have a mechanism that allows large schemas to be broken down into smaller ones using a modular concept. JSON schema already provides

Listing 6.2: Updated Metadata Format Including the Revision Concept

```
{
  "metadata" : {
    "revisionNumber" : "r21" ,
    "author" : "fce34f99-3949-4dc6-965c-c48a6b35620d",
    "created" : "2017-03-06 16:00",
    "lastModified" : "2017-04-12 9:00"
  }
}
```

some concepts in order to reference other schemas. Those will be presented in order to discuss the final referencing concepts for the service. The first functionality of JSON schema is the ability for in-schema referencing. Listing 6.3 shows an example of a schema describing a building located on a campus. The building has some properties e.g. a number, a name and additionally a complex property (an *address*) that is referenced in the document via the $\$ref$ keyword. The reference points to the **definitions** object which is included in the document. This allows to create modular and reusable schemas that are still included in a single JSON file. This is especially interesting for applications that only request one schema file but still want to make use of the modular concept. The reference keyword can, however, also be used for external schema referencing. In this case the value of a reference is the URI of the referenced schema. An example URI is presented in Figure 6.1. The basic structure remains the same compared to the other services. However, the last part of the URI is defined by the schema class identifier, the schema class version and the schema identifier. This URI can be used to directly address a specific schema. These referencing options provided by JSON schema are used by the semantics service to provide a rich and flexible API.

The semantics service has three strategies to include references in schemas or the referenced schemas directly. The first is the use of external references. This can be done using the $\$ref$ keyword and the URI of a schema. This is the easiest strategy for the semantics service since it uses the schemas in a modular way and does not merge any data.

The second strategy is to integrate the references in the schema and reference them using local references (as seen in Figure 6.3).

The third strategy is to resolve the references by producing one large schema including all of the referenced schemas at the position that the reference was indicated. The result is one JSON schema document without any $\$ref$ keys. This is important for applications that do

Listing 6.3: Example of In-schema Referencing

```
{
  "$schema" : "https://json-schema.org/draft/2019-09/schema",
  "title": "Campus Building",
  "type " : "object",
  "properties": {
    "number": {"type": "number"},
    "name": {"type": "number"},
    ...
    "address": {"$ref": "#/definitions/address"}
  },
  "definitions": {
    "address": {
      "type": "object",
      "properties": {
        "street": { "type": "string" },
        "city": { "type": "string" }
        ...
      },
      "required": ["street", "city"]
    }
  }
}
```

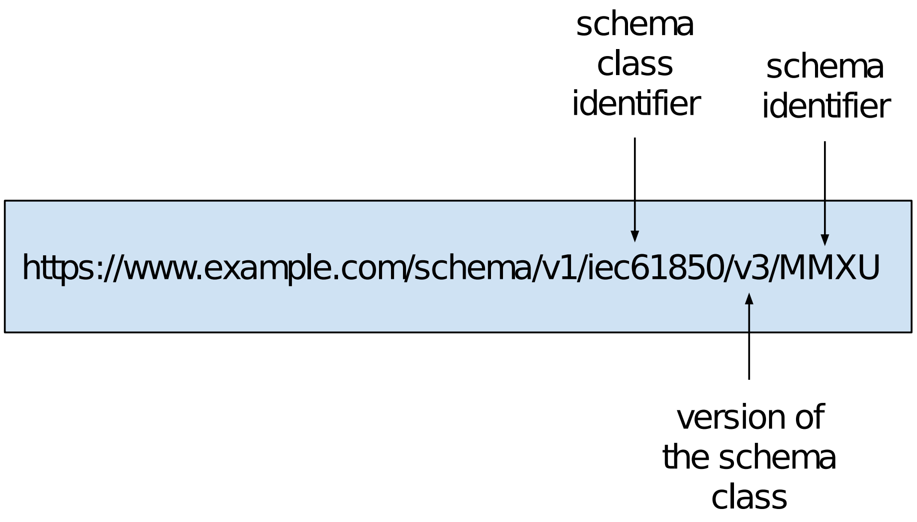

schema class identifier
schema identifier
https://www.example.com/schema/v1/iec61850/v3/MMXU
version of the schema class

Figure 6.1: URI of a Schema

not support referencing. After having seen the different concepts, the next section presents the design of the API.

6.2.4 RESTful API

Similar to the data services, the GET request is the most complex. In the following the concepts previously discussed are mapped to the API. First, the field query parameter is defined. It is used to decide whether the schema, the metadata or a specific property should be included in the metadata object. This allows applications to precisely select the information they need in order to get the semantic information. The following list shows a few examples.

- **field=schema**: returns the schema information
- **field=metadata**: returns the metadata of the schema
- **field=metadata.author,metadata.created**: returns the author and created field of the metadata object
- **field=-schema**: excludes the schema information (same as the second example)

Furthermore, there is an alternative for an application that wants to fetch a schema in the JSON schema format. This can be achieved by setting the HTTP accept header to the value *application/schema+json*. In that case the field query parameter is ignored.

Last, the REST API has to allow to define a way to include or resolve references. The default is that only the requested object is returned and no references. The JSON API standard the master data service is based on defines a mechanism that allows to include related objects. This parameter is suitable here and therefore used for including such objects. The value of the *include* query parameter is a list of keys that are added to the schema additional to the requested object. Considering the example depicted in Figure 6.3, the address can be included by the query parameter in Figure 6.2. The include query parameter allows to set two different operators. The **resolve** operator implements the third strategy by resolving the address reference and therefore placing the schema at the place of the $\$ref$ key. The **append** operator includes the schema in a new top level object as seen in Figure 6.3. The name of the key can be modified. The default is, as the JSON schema draft suggests, **definitions**.

After having seen the GET request, Table 6.1 shows an overview of all the RESTful requests. Most of the requests are self-explanatory or already presented. The responses of the first two

```
include=resolve:address
include=append[key=definitions]:address
```

Figure 6.2: Examples for the Include Query Parameter

GET requests return a paginated list which can be browsed in the same way as the list of master data objects in the master data service. The PUT request has to include the changes in the request body and a new revisionNumber is returned. Deleting a whole schema class also deletes all of the schemas included in this schema class version. When omitting the version, the whole schema class with all the schemas can be deleted.

Operation	**HTTP Method**	**Example URL**
add a schema class	POST	/schema/v1/
add a schema	POST	/schema/v1/class/classVersion/
fetch all schema classes	GET	/schema/v1/
fetch all schemas of a specific class and version	GET	/schema/v1/class/classVersion/
fetch a schema	GET	/schema/v1/class/classVersion/schema
update a schema	PUT	/schema/v1/class/classVersion/schema
delete a schema class	DELETE	/schema/v1/class/classVersion/
delete a schema	DELETE	/schema/v1/class/classVersion/schema

Table 6.1: semantics service RESTful API Requests

6.3 Analysis Service

The analysis service is mentioned by the requirement R8. Applications have the need for insight into the data stored in the different data services. This section focuses on statistical data for the measurement service and the master data service. As already presented in Chapter 5, the data services already provide certain aggregation functionality if the underlying database supports the functionality. In that case it is more efficient for the data service to directly provide the aggregations. However, it is not viable to include all of the aggregation functionality in the data services. In that case, the services have to much responsibility which is a contradiction to the microservice design principles. The analysis service provides more complex aggregations that are not provided by the services directly. Since the service uses https://www.example.com/analysis/v1/

Figure 6.3: Base URL of the Analysis Service

the data from the data services, it does not require an own database. The only exception might be in order to support caching of results. In the following, the service base URL, main concepts and the REST API are presented.

6.3.1 Base URL

The base URL of the service is very simple. Since the requests are too complex to be integrated in the URI, the requests are defined in the request body. Figure 6.3 shows the structure of the URL which is identical to the data services'. The functionality to compute statistics implemented mainly using the request body is discussed in the following.

The main request of the analysis service is a POST request that triggers the computation of statistics based on the query that is transmitted using the body of the request. Since GET request usually have no body, a POST request is used. Furthermore, a request that triggers a computation is not only a read request even if the results of the computation are returned. This further justifies the choice of using the POST method. Listing 6.4 shows the structure of the request body. The example fetches a list of buildings based on a filter and transforms the location field to the geographic coordinate system (using lat and lon parameters). The JSON object consists of two parts: the service object that describes the data that will be requested in order to perform an analysis and the analysis part that describes the computation that is performed. The vocabulary of the service part is the same as in the corresponding services. The example results in the following query that is performed by the analysis service in order to fetch the desired dataset:

/masterdata/v1/building?filter[number]=eq:449&fields=number,name,location

The second part is the analysis that will be performed. The name and the field that the analysis is applied to have to be specified. Additionally, parameters that depend on the type of the analysis have to be set. In the example the target format has to be set in order for the analysis to know which transformation has to be performed. The parameters depend on the analysis type that is requested similar to the filter and aggregation operations.

Listing 6.4: Example of a Request Body for the Analysis Service

```
{
  "service": {
    "name": "masterdata",
    "version": "v1",
    "schema": "building",
    "filter": [{
      "field": "number",
      "operation": "eq",
      "parameters": [],
      "value":"449"
    }],
    "fields": {
      "whitelist": true,
      "values":["number", "name", "location"]
    }
  },
  "analysis": {
    "name": "transformLocation",
    "field": "location",
    "parameters": {
      "targetFormat": "lat/lon"
    }
  }
}
```

Listing 6.5: Data Format for an Asynchronous Request

```
{
  "service": {...},
  "aggregation": {...},
  "async": true
}
```

6.3.2 Async Requests

As mentioned, some analysis types are very complex and might take a long time to execute. Therefore, the request can be set to the asynchronous mode (see Listing 6.5) which immediately returns the request. The response includes a URL that allows the application to request the status of the analysis. This is useful for tasks that take longer than the application can wait for the synchronous RESTful request to succeed. When the *async* key is omitted the default behavior is that the request is not asynchronous.

6.3.3 Master Data Service

This section discusses possible analysis types for the master data service. As already mentioned, a very important analysis is the terms aggregation that provides a summary of a specific field of master data objects that have the same schema. If the database behind the master data service already supports this functionality, the analysis service can call the API of the master data service and forward the response to the client. Otherwise, the analysis service implements the functionality by requesting the relevant objects and computing the summary. This operation might be complex and both time and memory consuming. Therefore it makes sense to cache such a response. Such a cache can speed up additional requests of the same analysis. A second example is the transformation of locations from one map projection to another. In Figure 6.4 this analysis is already shown. It has a parameter that defines the target map projection. The map projection that the data is represented in before the transformation can be either detected automatically or define using a second parameter. The field that the analysis is applied can be validated using the corresponding schema of the master data object. This allows to make sure that the field is a location and map projection it represents. This shows another advantage of having the semantics service which its explicitly defining the semantics of each master data object.

The analysis service can define many more types of analysis for the master data service. However, this chapter only presents the concept and some examples in order to have a reference for implementing such a service.

6.3.4 Measurement Service

The measurement service can also be analyzed using the same data format presented in Figure 6.4. The only change is the service information that has to be conform to the measurement service API capabilities. Similar to the master data service, the aggregations provided by the measurement service itself can also be requested using the analysis service. In that case the request and response are forwarded. This is the case for the basic reduce aggregations and the downsampling and rate aggregations (see Chapter 5). Another example for a more complex analysis is the extraction of the trend or the seasonality of a time series. Both extractions can get the application important insights in the data. Furthermore, a common use case is to have data with outliers or errors. In that case data analysis can help in order to remove outliers. A method to do that is called hampel filter [PN+16]. The service computes such a filter and returns the result. The measurement service can have very complex analysis types including possible machine learning or Big Data algorithms. In that case, the analysis service might need some other services or tools in order to perform such analysis. Many more data analysis methods might be discussed, however this is out of focus for the present thesis.

6.3.5 RESTful API

The RESTful API (Table 6.2) only features two methods. The first method is already presented in detail in this chapter. The second request is used to get a status of an analysis that was requested asynchronously. When the computation succeeded, the result is returned. The identifier that is needed is provided in the response of the first request.

Operation	HTTP Method	Example URL
perform analysis	POST	/analysis/v1/
get information about async analysis	GET	/analysis/v1/analysisIdentifier/

Table 6.2: Analysis Service RESTful API Requests

6.4 Conclusion

This chapter presented the supporting services of the reference architecture. First, the semantics service was presented that provides the semantic information that is missing in the data services. This information can be especially useful for applications that work with the data service and have the need for additional information about the data. The service has different concepts that allows to manage so called schemas which represent the semantic definition of data objects. Mainly the master data service profits from the semantics service since each object is described by exactly one schema. The semantics service allows to have different versions for schema classes and each schema can have an internal revision number in order to keep track of changes. The data format and the REST API were designed and discussed in this chapter. Second, the analysis service was presented. It provides applications with further insight in order to request data analysis computations that are triggered by a request and can be executed synchronously or asynchronously. In the latter case, the application can send a GET request to get a status report about the computation's progress.

7 Frontend Framework for Dashboard Applications

After having seen the concepts for the reference architecture's backend and its design process, this chapter focuses on the frontend. In Chapter 1 and 2 two example applications have been presented. First, a control center software that shows visualizations about energy grids on large display walls was presented. Second, environmental applications visualizing data about the citizen's environment on web pages were discussed. Both of these application types have a frontend that can be described as *dashboard* showing customizable visualizations about the data stored in the backend systems. This chapter presents a framework that applies the generic and modular design ideas of the backend to the frontend. The functionality the framework provides is a generic dashboard system that allows to create a fully customizable dashboard application. The goal of the chapter is to define the concept for an application that preserves the generic functionality and customizability of the backend and to demonstrate an application that allows to evaluate the whole reference architecture. The first section gives a brief analysis of two relevant publications that build the foundation of the framework presented in the current chapter.

7.1 Scientific Analysis

A first publication [MW17] presents a similar frontend application as presented in this chapter. It is a dashboard application showing different web-based visualizations. It visualizes monitoring data that is collected by a microservice monitoring and logging infrastructure. The publication can be seen as a related example application that is mapped to the energy domain for the current chapter and the evaluation. The authors define different visualization components on a single web page. Different configurations are called *views*. Additionally, the publication shows different examples of a prototypical implementation.

In a second publication [KL+16b] a frontend architecture based on the web component technology is presented. The concept is fairly abstract, but it also uses the idea presented in this chapter to create a dashboard from different web components. The different components connected to their appropriate data including the semantic context. The authors use linked data principles. for the implementation of this concept.

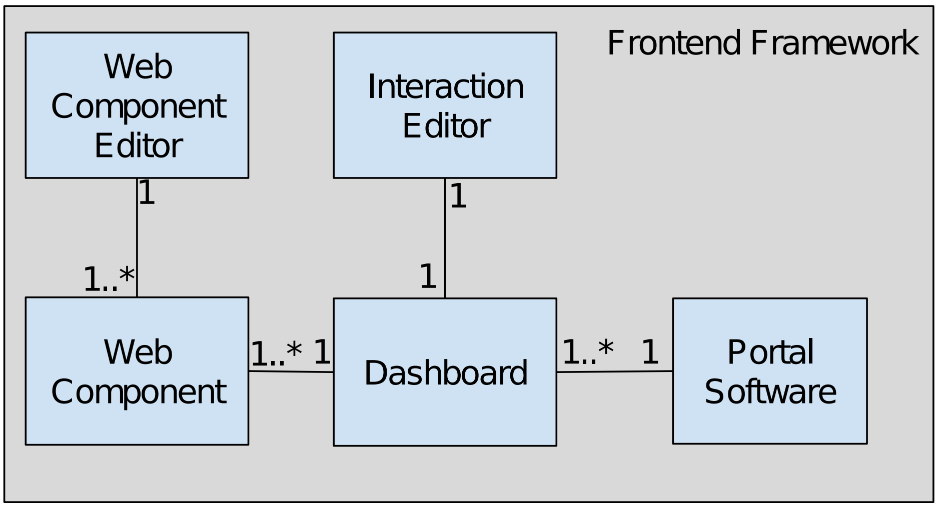


Figure 7.1: Overview of the Concept for the Frontend Framework

These two publications build the foundation of the frontend architecture. However, this chapter goes into more detail on how to realize such a dashboard application allowing the user to create different dashboards in an abstract way without knowing the specific domain beforehand. The next section gives an overview of the concept.

7.2 Overview of the Framework Concepts

The different parts of the concept are illustrated in Figure 7.1. The main idea of a framework that allows to create dashboards and is highly customizable arose from the article [BD+17a]. It describes the early concepts that only focused on data visualization. The idea is to select a specific visualization type (for example a bar chart or a line chart) and to configure the visualization to the own needs. This customization can be done using an editor that allows to configure the visualization without the need of programming knowledge. This allows even non-IT experts to create their own visualizations. The technological basis of the frontend framework presented in the article are web components (see Chapter 2 for more information). A second article [BD+17b] showed that even complex visualization types with 3D visualization requirements can be created and customized using the same concepts. These scientific works showed that the web component technology is suitable for such generic frontend frameworks and has more potential than only the creation of data visualizations. The overall web components concept is used in many fields of web application development and use cases which shows that the concept presented in both articles is suitable to be expanded to a general framework with the web component technology as a basis.

A second step is the formal concept of the usage of the web components. Similar to services, a web component has defined interfaces that can be used to integrate them in a larger concept in a modular way. A sustainable approach allows to integrate web components developed for the framework as well as also third party components that implement the W3C standard [W3C19]. However, the concept presents methods to improve the integration of components into the framework presented in the next sections. These methods include the improvement of documentation and the improved interface of the backend services described in Chapter 5 and 6. Additionally, semantic information of the components is used to support the third step of the frontend concept: a graphical user interface (also called *web component editor* in this chapter) which allows to configure each component. The idea of both preceding articles mentioned in the first paragraph (Section 7.2), is used in order to keep the advantages. With the web component editor and a second editor, called *interactions editor*, the user is able to configure a component and the interactions between different components without the need of programming knowledge. With the additional information of semantic descriptions of the web components the user has an improved user experience.

After the illustration of the concept of a single web component, the whole dashboard concept is presented and the interaction of multiple components in the context of a web page is discussed. This concept includes the usage of backend services, definition of the communication between components and possible coordination of this step. At last, a discussion about the integration in existing portal system solutions is discussed. A portal system plays the role of a container application for one or multiple dashboards.

7.3 Web Components

As already described in Chapter 2, the web components technology is a standard expanding HTML5 [W3C19]. Besides the different parts the standard consists of (Shadow DOM, HTML Imports, Custom Elements), each web component is defined by a specific set of properties, methods, outgoing events and slots. Figure 7.2 illustrates the interface of a web component. The properties are the main interface which allows the developer to define key/value pairs that are directly available as variables inside the web component's source code. Two-way binding concepts allow to react to value changes which result in dynamic changes of the style and behavior of a component. Most of the communication with a component is done via the configuration of the properties. However, sometimes a more complex operation has to be provided by a component, for example the ability to set large amounts of data which is not suitable to be serialized into the value of an HTML attribute (which is the formal

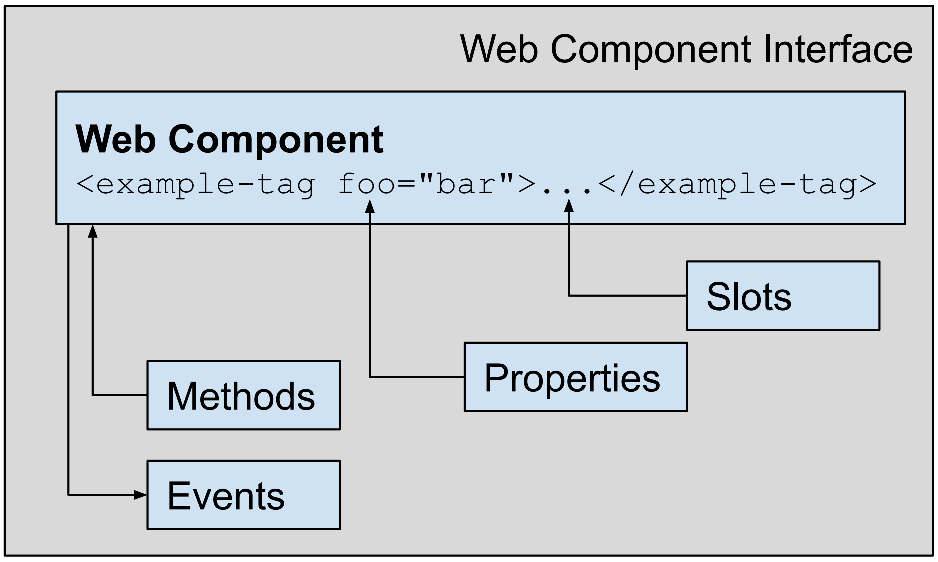


Figure 7.2: The Interface of a Web Component

definition of a web component property). Another example is an operation that triggers a complex behavior or a get-operation that returns an internal data structure. These operations are better defined as methods that can be triggered from the outside. In order to use a method, some custom implementation is needed, however, this is reasonable for such a complex task. Both ways of interaction with the component are triggered from outside of the component. The only way for a component to communicate autonomously with the rest of a web page is the trigger of events. These are ignored by the web page until a matching event listener is registered which processes the event and reacts to it. Both directions of communication are indicated by the direction of the arrows in Figure 7.2. The last part of the interface is the definition of slots. They define a place in the web component's DOM where additional content can be embedded. However, this does not directly influence the behavior of the actual component. An example for the utilization of slots is a web component that shows a news article entry with custom header and footer slots. Both slots can be bound to customized HTML. This allows developers to further design the DOM that resides inside a web component.

The web component editor which will be defined later in this section uses the information of defined properties, methods and events in order to enable the user to configure them appropriately. However, depending on the library that is used to develop the web component, it is hard to extract meaningful semantic information about the interface. Therefore, the following paragraph presents conventions that lead to better documented and semantically-aware web components.

Listing 7.1: First Example of a Property Definition

```json
{
  "name": "offset",
  "desc": "The amount by which the element is out of line.",
  "schema": {
    "type": "integer",
    "defvalue": "0"
  },
  "required": true
}
```

Listing 7.2: Second Example of a Property Definition

```json
{
  "name": "complex",
  "desc": "An object that holds complex information.",
  "schema": {
    "type": "object",
    "properties": {
      "color": {
        "type": "color"
      },
      "text": {
        "type": "string"
      }
    },
    "required": ["text"]
  },
  "required": true
}
```

First, looking at the standard, the developer has no standardized way to define any information about a property. Not even the existence of a property can be defined properly. Having a look at the Polymer[1] library, the developer of the component has the ability to define a data type for each property. However only basic data types are allowed since this information is only used to parse the value appropriately. Such parsers are needed since the value of the property, which is inserted into the DOM tree when an element is rendered, is always a string because it is a value of an XML attribute. A similar functionality is provided by the library LitElement[2]. In order to enhance the description of each property the configuration depicted in Listings 7.1 and 7.2, including two examples, is used. The description includes multiple custom fields that all further define the property. First, the type of the property can be defined by setting the schema field which not only supports JavaScript types but also types that are recognizable by the user interface such as the type *color*. The information that the property is a color allows the web component editor to show a color picker in order to give the user an easier way to select a value for it. JSON schema [JSON-Schema] is used as specification with some possible custom fields or custom types if needed. JSON schema also allows to define a default value (*defvalue*) and information about *required* fields if the type of the schema is an object. The *required* field on the top level is used to define if the property as a whole is mandatory or optional. The *desc* field contains a human readable text that describes the property in one or more sentences. This additional information for each property can be used to increase the user experience of the web component editor and to create a generated documentation. Last, the *name* defines the string that is used as HTML attribute name.

Second, the description of a method is analyzed. The common libraries do not allow to define names of available methods, their descriptions or the arguments that exist. However, these are crucial information that again an editor needs in order to support the user in configuring the component appropriately. Similar to the definition of a property, the description of a method can be defined in the JSON format. An example is provided in Listing 7.3. Similar to the property definition, a *name* and a description (*desc*) can be set. A new field is called *args*. It defines an array of arguments a method can be called with. Again a name and a description are used to provide meaningful text to the user. In order to have metadata that defines the usage of each argument, a *schema*, a *required* field and a default value (*defvalue*) can be set.

The third part of the interface is information on supported events. Analogous to the two previous paragraphs, events are also poorly supported when it comes to semantic information.

[1]https://www.polymer-project.org/
[2]https://lit-element.polymer-project.org/

Listing 7.3: Example of the Definition of a Method

```
{
  "name": "setInput",
  "desc": "Setter method for complex input data",
  "args": [{
    "name":"val",
    "desc": "value to set",
    "schema": ...,
    "required": true,
    "defvalue": undefined
  }]
}
```

Listing 7.4: Example of the Definition of an Event

```
{
  "name": "textChanged",
  "desc": "Sent when the text inside the component changed.",
  "schema": ...
  "example": {
    "value":"Example Text"
  }
}
```

Listing 7.4 shows an example of an event definition. Again, an event is described by the fields *name* and *desc*. Additionally, a schema can be set defining the structure of the event payload. Furthermore, an *example* can be set in order to give a description of how the event's payload might look like. This information can be used in order to document the component. However, it is not used in the web component editor since it is meant to be used by other components or code that resides outside of the triggering component. Hence, the information can be used in the interactions editor described later in this chapter.

The last part of the interface that has to be described are the available slots. The slots are special in the way that their description is only used in order to generate documentation for the web component. There is no editor planned in the concept that allows to configure the behavior of slots. Therefore, as seen in Listing 7.5, only the fields *name* and *desc* have to be defined.

After having discussed the additional information that can be defined in order to improve the generated documentation and user experience, the storage options of this information

Listing 7.5: Example of the Definition of a Slot

```
{
  "name": "slotName",
  "desc": "This slot allows the placement of additional
      components"
}
```

are analyzed. One obvious possibility is that the information is included in the source code as additional JSON properties, custom variables or as comments similar to JavaDoc [3] or Doxygen [4]. The advantage is that the developer can add the description without the need for additional files or another version control system repository. A downside for the usage of comments, however, is that comments are often missing. Therefore, an alternative is to add the whole interface description in a single JSON file that can be stored besides the web component's source code within the same repository. An advantage of this method is that the whole information is in a machine readable file at a specific place. Since some web components libraries, as mentioned, have already implemented parts of the solution, a combination is also possible which allows to automatically create parts of the JSON file based on the description that the developer has to provide in order to use the library. In that case, the information can be extracted from the source code and added to the file. The combination described last is considered as the solution for the rest of the concept.

The next paragraph presents the concept for the web component editor. The editor provides the user a tool to configure a single web component. Since only one web component is in the focus, the editor only uses the property information. The rest of the interface is related to the communication or coordination of a web component in the context of a web page with multiple other components. Since each property ideally has a specific type definition and schema attached, the editor makes use of this information in order to show the user an input allowing the configuration of the property that is tailored to each property type. A property that represents a color, for example, does not only show a text input that allows to set the hex code of an HTML color but the user is also presented a color picker which lets the user pick a color very intuitively. Additionally, each property can be described by its human readable semantic description if available. This is an extension of the concept described in [BD+17a] by the ability to configure any web component and not only data visualization components. After having seen the concept of the expansion of a single web component by the ability to

[3]https://docs.oracle.com/javase/8/docs/technotes/tools/windows/javadoc.html
[4]http://www.doxygen.nl/

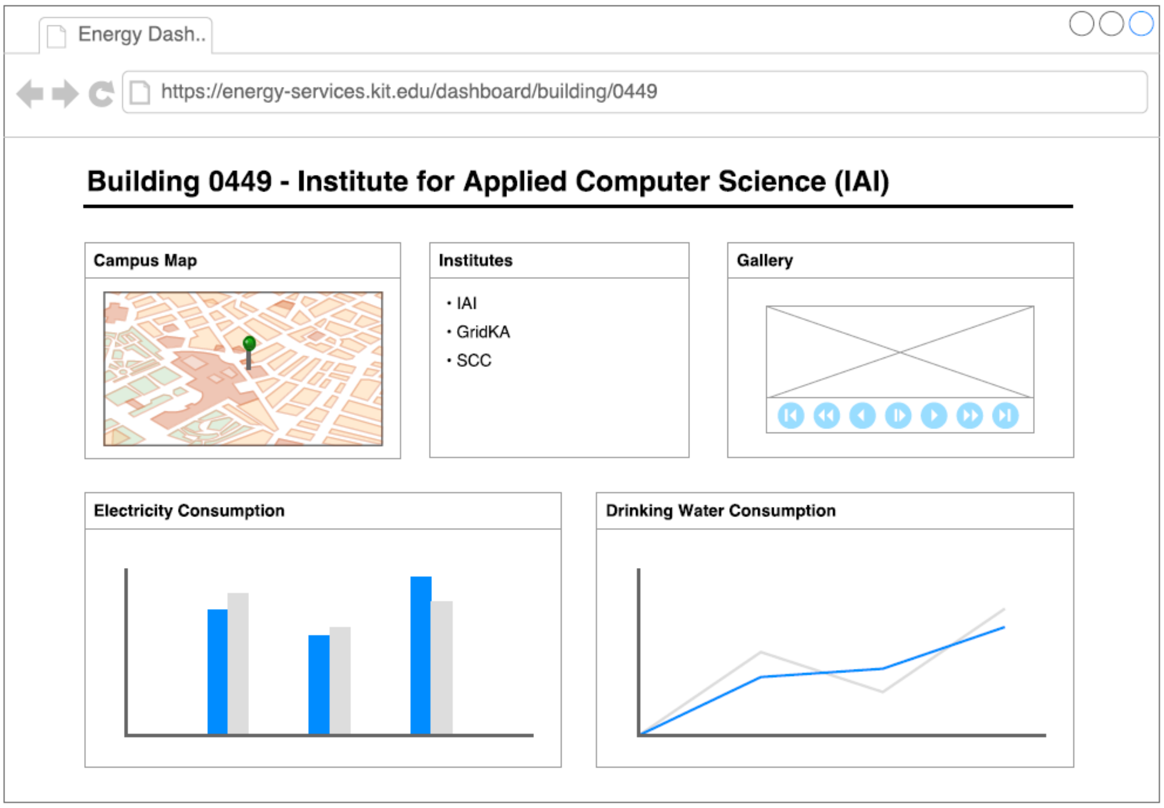

Energy Dash..
https://energy-services.kit.edu/dashboard/building/0449
Building 0449 - Institute for Applied Computer Science (IAI)
Campus Map
Institutes
• IAI
• GridKA
• SCC
Gallery
Electricity Consumption
Drinking Water Consumption

Figure 7.3: Mockup of an Energy Dashboard Monitoring Buildings from [Pa16]

better document and semantically describe its interface, this part of the frontend concept is embedded into the overall concept of having a dashboard of multiple web components. In the following section this dashboard concept is presented. However, first, two scenarios are presented that will motivate the dashboard concept.

7.4 Scenarios

In order to create a control center application or an application with information about the environment, a single web component is not enough. The developers or operators of these systems need the ability to create a complete page or grid of web components. The position and configuration of each web component has to be defined and, most importantly, the communication and interaction between the components has to be configured. First of all, two scenarios will be presented that each describe an application for a dashboard. More details about the examples can be found in [Pa16].

In the following, a simple example dashboard (Figure 7.3) is used to further illustrate the

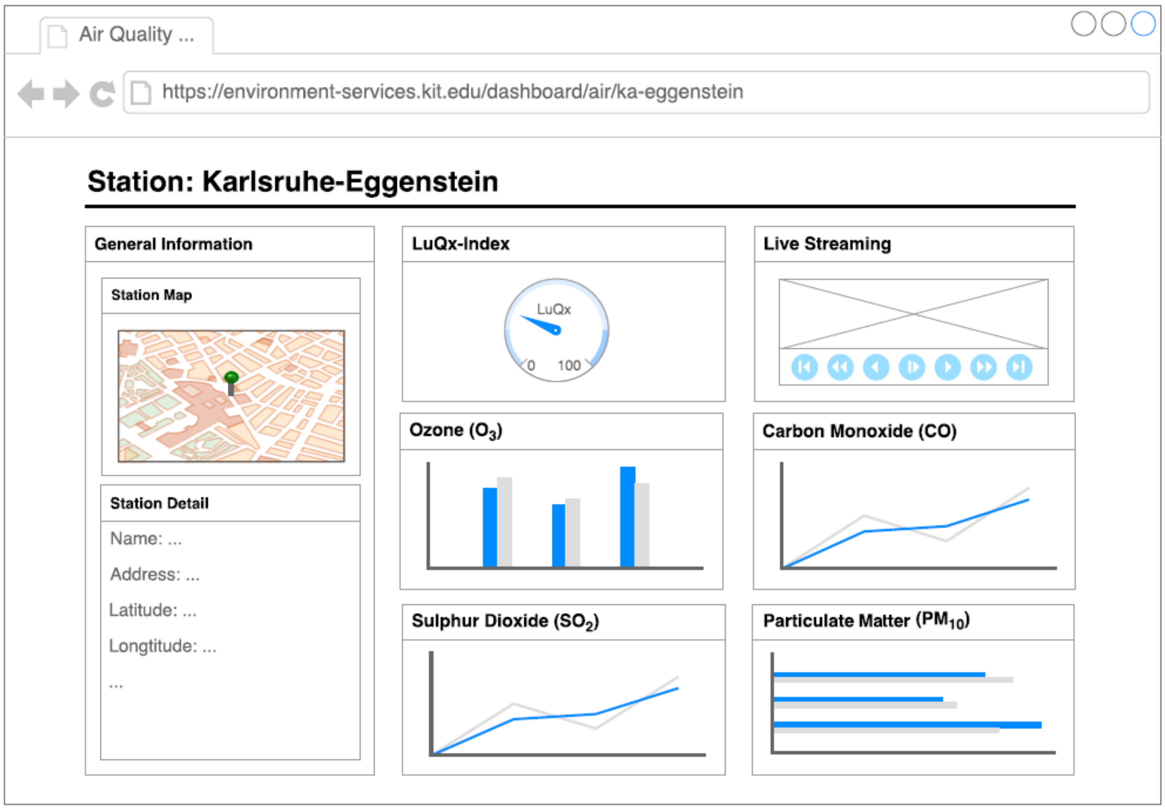


Figure 7.4: Mockup of an Environment Dashboard that Displays Air Quality Information from [Pa16]

concept. The mockup shows a dashboard that displays information of buildings and their energy consumption. The dashboard consists of five different parts each visualizing a specific set of information. The first is a map showing the different buildings situated on a campus. The second is a table displaying information about the building. This information is usually typical master data. The third part displays an image of the building which is a typical digital asset. The last two parts of the mockup visualize the eletricity and drinking water consumption data as line and bar chart respectively.

Each part can be modelled using the concept presented in the last section using the web component technology. However, the developer/operator has more needs than just looking at each component separately. He wants to see the components interact with each other. For example, a click on a building on the map should trigger the other components to update their information according to the selected building.

A second example (Figure 7.4) shows a dashboard displaying information on air quality. Again, a map is displayed showing the location of the currently selected air measurement station and lets the user select other stations. A second component shows the metadata of the station with its name, address and location. A third component displays a live stream that shows the surrounding area of the measurement station for security reasons. The rest of the components display measurement data for different air pollutants.

These two example scenarios lead to important objectives a generic dashboard application has to tackle. First, each component has to be placed in a grid or some other form of structure in order to have an arrangement and alignment on the dashboard area as needed. Second, each component has to be configured in a UI integrated in the dashboard concept in order to have a single place for configuration. This might be implemented by having a dialog displaying the web component editor. Third, the interaction of the different components has to be defined. The next challenge is that data has to get to the frontend dashboard in order to be visualized. Therefore, all of the data services presented in Chapter 5 are needed to implement both dashboards. In order to tackle these challenges, the following dashboard concept will be presented.

7.5 Dashboard Concept

The concept for the configuration of a single web component has already been presented. However, having multiple web components on a single web page raises the need for additional concepts. Especially, the concept of dealing with the communication between two or more components. Figure 7.8 shows two web components and how each part of a component of the dashboard interacts with another. First, when a web component is placed in a dashboard also its configuration, previously defined using the editor, has to be applied too. Otherwise the communication can not be coordinated. Therefore, each web component is wrapped inside a generic web component called *Flex Component* (short for flexible component). This is a special web component that is used to coordinate a previously configured component. A second wrapper might be used when using a portal system (*Portal System Wrapper*) to support the dashboard components with additional portal-specific functionalities. This wrapper can be used to store and load configurations when the developer/operator configuring a component changes it or when a user loads the dashboard. The configuration of a web component is (once loaded from the portal system wrapper) used to initialize the web component or to reflect changes the user has done in the web component editor. Besides the management of the configuration, each flex component is responsible for the coordination of interactions between the different web components. This includes receiving and sending events. An example for such interactions in the first scenario is a user who clicks on a building in the map

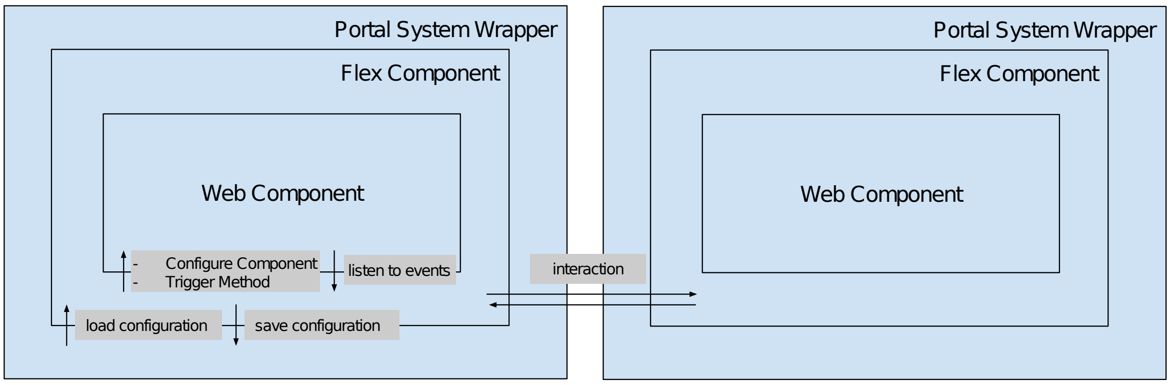


Figure 7.5: Dashboard Communication and Wrapper Concept

and updates the other components accordingly. In the following, these different aspects of the concept are discussed in detail. First, the flex component will be presented and discussed.

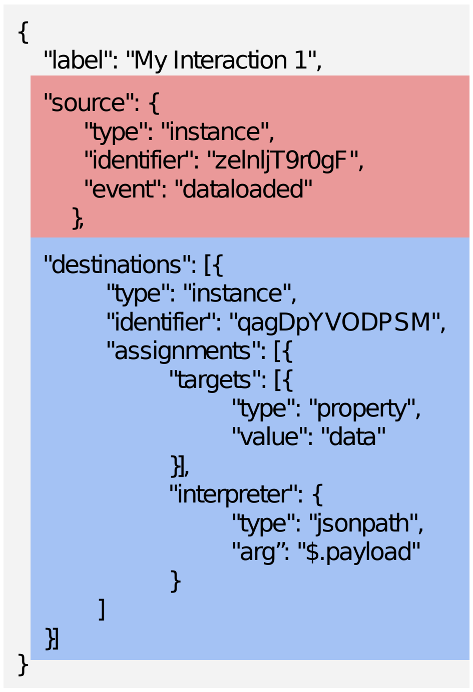
7.5.1 Flex Component

The core part that is needed in order to have a working concept for the display and interactive behavior of the dashboard is the flex component. It provides the connection between the plain web components, the dashboard framework and the editors (or to be more precise for the latter: the configuration files that are created/edited by the editor). The most important aspect of the flex component is how it manages the web component (one flex component is responsible for exactly one web component since it is wrapped around it). However, first, the concept of interactions has to be defined in order to completely present the concept of web component management.

7.5.1.1 Interactions

As already mentioned, in order to configure the interactive functionality of a dashboard, interactions have to be defined. Interactions are operations happening between two or multiple components. Such an operation has a defined source that triggers it and a one or more defined destinations that are being triggered. Source and destination is each a web component. The interaction is realized as an event that is sent between the source flex component and one or multiple destination flex components. Similar to the web component, each interaction is configured using the interaction editor. An example configuration is shown in Figure 7.6.

The configuration of an interaction is divided into three parts. A general part including metadata defines the root of the object and includes a human readable label for the interaction



```
{
   "label": "My Interaction 1",
   "source": {
        "type": "instance",
        "identifier": "zeInljT9r0gF",
        "event": "dataloaded"
     },
   "destinations": [{
           "type": "instance",
           "identifier": "qagDpYVODPSM",
           "assignments": [{
                   "targets": [{
                           "type": "property",
                           "value": "data"
                   }],
                   "interpreter": {
                           "type": "jsonpath",
                           "arg": "$.payload"
                   }
           ]
   }]
}
```

Figure 7.6: Interaction Config Example

in order to make it easily identifiable in the editor. Additional metadata can be extended if needed. The second part of the configuration is the definition of a source for the interaction. This is the web component or web component type that is the sender of the event. The *type* of the source is either a specific web component within the dashboard or a web component type, for example all maps within the dashboard. In the first case, the *identifier* field is the identification of this specific component. The latter case changes the semantics of the identifier field to the identification of the component type. An example of the second case is a dashboard showing two maps with the same content in two different styles or at two different positions within the dashboard. However, both should trigger the same interaction when the user clicks on an element of that map. In that situation a single interaction can solve the problem. Without the support of component types as source, two interactions had to be defined to realize this. The *event* field selects the correct event the component emits. In order to get the event to interact with other components, the destination part has to be set up. It is more complex since it is not only the component that has to be selected but also the action that should be triggered. The *type* and *identifier* have the same meaning as in the source block of the configuration. For a specific destination, multiple *assignments* can be defined. An assignment consists of specific *targets* that can be a method or a property (controlled using the *type* field). The *value* is the name of the property or the method. Those two properties control how the event triggers the destination component(s). The last and most complex part of the assignment is the *interpreter*. It is responsible for the selection or transformation of data that is set to a property or method argument. The *type* can be a range of transformation operations. The most common is the JsonPath [5] transformation which allows the assignment to select a specific part of the event object. In the example of Figure 7.6, only the content of the event's *payload* property is set to the property *data* of the destination component. Another flexible transformation is the ability to select a plugin. In that case, the value is a key/value list of the parameters the plugin requires. The concept allows to expand the possible assignment types in order to provide frequently used operations. The goal is not to have a plugin for every case at the end but to have plugins only for special cases and specific assignment operations for the common ones. An example is a plugin that changes the timezone of a datetime parameter.

The interaction editor needs all the information of all the web component specifications in order to help the developer/operator with the selection of a component that is situated on the dashboard or a list of events that a web component supports. Since the web component configuration file the web component editor creates has a field for a page id, the interaction editor can know which components are present on the dashboard.

7.5.1.2 Interaction Coordination

After having discussed the structure of an interaction and the process of changing its configuration, this paragraph focuses on the execution of an interaction when a user loads the dashboard or triggers an event. As seen in Figure 7.5, the flex component listens to each event that participates in an interaction. This allows the flex component to coordinate a proper event communication. A challenge that has to be tackled is that the events of web components are send asynchronously and, therefore, events might get lost or sent in a wrong order (an order that the user feels uncomfortable with). Two examples will illustrate an erroneously behavior. First, the flex component has to be loaded by the browser and next, the flex component has to load the actual web component. This happens once per web component that is included in the dashboard. However, when an event that is part of an interaction has already been sent before the destination flex component is ready, this event will not be processed by it. In that case the interaction will never occur and the dashboard is in a wrong state. A second error might occur when multiple events are sent nearly simultaneously but their processing is slower in some cases. In that situation, the user might see unwanted side effects. Both examples show

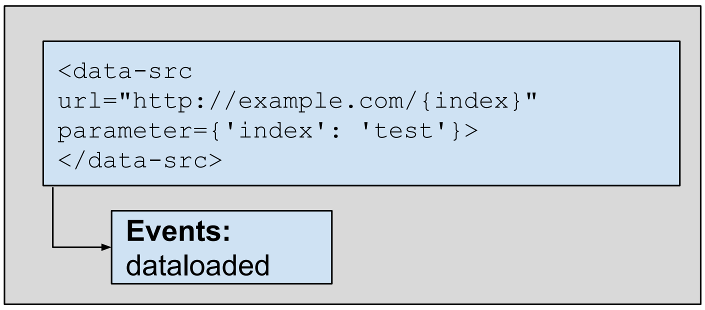
[5] https://github.com/json-path/JsonPath

that interactions have to be coordinated correctly. In order to avoid events getting lost or race conditions, the event processing has to be done centrally on a dashboard. The flex component exists multiple times on a page but the event progressing has to be implemented as a singleton in order to be managed at one place. As for the missing events, the interactions can only be processed when all flex components are ready. When events are emitted before this point in time, these events have to be halted for the time in between. Another challenge that has to be tackled is the need for some events to be processed synchronously. When the user selects a measurement station on the map and wants all the visualizations and the live stream to update simultaneously, a further coordination strategy has to be set up. Since the events for each interaction are sent asynchronously and some data have to be fetched (which is typically also an asynchronous request), the application of each event to the destination component has to be coordinated again. This can be solved by specifying interactions that belong together and have to be processed as a unit. This solution works only with central coordination. This last challenge is especially important for a dashboard with multiple components all showing information for a single context. In that case, it is mostly certain that the user would like to see an update of the whole dashboard when something changes and not an update as soon as it is processed since then each component is updated in short time intervals one after the other which might be perceived by the user as a bad experience (temporarily inconsistent views). As already briefly mentioned, in order to visualize data, this data has to be fetched from somewhere as asynchronous request in beforehand. The next section will discuss this task.

7.5.1.3 Data Management

There are two possible solutions for the task of fetching data from a service and make it accessible in the dashboard. It is a question about which part of the concept is responsible for handling data requests. Data handling can be done by the flex component similar to the event processing or a web component might be responsible for the handling of one or multiple data requests. The advantage of integrating the data handling in the flex component is that a central management is possible which leads to a better caching infrastructure and less requests that have to be sent. The advantages of the introduction of a separate web component that sends data requests is that no additional logic inside the flex component is required which would not increase its complexity. Additionally, the component can be naturally integrated in the interaction concept and make use of its central architecture. Additionally, the web component iron-ajax[6] implemented by Google already shows a proof of concept for a component which

[6]https://www.webcomponents.org/element/@polymer/iron-ajax



```
<data-src
url="http://example.com/{index}"
parameter={'index': 'test'}>
</data-src>
```

Figure 7.7: Data Source Web Component

is responsible for sending data requests. Since the advantages of the latter dominate, a new web component, called data source component (*data-src*), is introduced that is responsible for the management of data requests. Figure 7.7 shows the most important parts of the interface. The data source component has no visible part that is shown in the dashboard. The connection to the interaction concept is realized by emitting an event called *dataloaded* each time data is available or the response of a request was returned. There might be multiple events depending on the status and success of the request. The *parameters* and *URL* of the request are set using the data source component's properties. In the figure, the URL includes a single parameter. Additionally, realizing the data management as web component and emitting events with the results, allows to get the same advantages of the solved coordination challenges mentioned in the last section.

Multiple configurations have been presented. The two different editors and the flex component need all this information in order to be able to configure and provide the whole dashboard functionality. Hence, the management of these configurations is still to discuss. Three configurations have been presented:

- **Web Component Definition**: The web component definition is used to describe a web component's interface and some additional metadata, for example the page identifier. This definition is required by the editors in order to support the user in the configuration of web components and interactions. The definition has to be provided to the flex component since information such as the default value for a property is included.
- **Web Component Configuration**: The web component configuration is created and edited by the web component editor and used by the flex component to create the web component instance with correct values for their properties.
- **Interaction Configuration**: The interaction configuration is created and edited by the

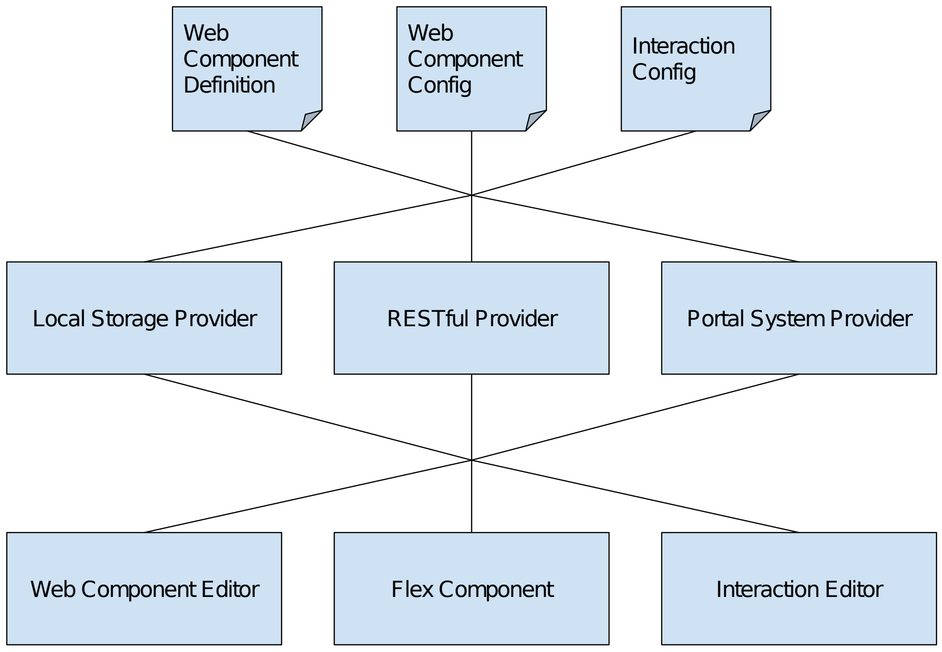


Figure 7.8: Management of Configurations

interaction editor. The configuration exists once for each dashboard or website. It is used by the flex components to manage the interactions.

Since the editors and the flex component have to have access to the configurations, it is useful to define a concept to store the different information in a consistent way. The concept of this additional part of the framework is called *provider*. Figure 7.8 shows the different configurations and possible providers that can be used. Each provider needs the ability to save and load the three configuration types. The most generic case is the RESTful provider which is a service that provides all of the functionality. However, it is not always useful to have a separate service. Therefore, a provider for the portal system the dashboard resides in is also defined. This allows to utilize the storage infrastructure of this system. A third provider allows to store all of the configuration locally in the web browser in order to work offline or to do prototypical changes. However, the latter provider has only impact on the dashboard in the context of a specific browser since the local storage of a browser is not shared across clients or different browser software.

7.5.2 Portal System Integration

The concept presented in this chapter does not include all aspects of a web-based application frontend such as security, user management, layout and positioning of the components, navigation bars, etc... This functionality is usually implemented by portal systems (such as Liferay[7]). Such software usually allows to integrate web components or HTML, CSS and JavaScript in general. Liferay, for example, allows to integrate any kind of web content. Additionally, the editors can be embedded into the portal systems settings which gives the user a better and familiar experience. In order to hide the editors from normal users, the security concept of the portal can be used. In this case only special users (having a specific authorization) are allowed to change dashboard configurations. The layout functionality can be used to rearrange the web components. Additionally, the navigation elements of the portal system can be used to navigate between dashboards. As presented in the last section, there is a portal system provider which has to be implemented specific for one software since it has to integrate into the software's storage system. Since the whole dashboard concept works without any additional software, the integration comes with not much overhead. The only additional work that has to be done is when the user experience has to be increased such as the integration of the editors into the portal system setting.

7.6 Conclusion

This chapter first defined a concept extending the web component standard. A concept for describing the interface of a web component is presented. This description leads to improved generated documentation and a web component editor supporting the user by making use of the semantic description. Next, two scenarios were presented and analyzed in order to understand the objectives of a generic dashboard. Then, a concept has been presented that defines this generic solution. The main part of the concept is the flex component which wraps around each web component and provides a specific set of features. Besides the application of the web component configuration to its properties, the component makes use of defined interactions in order to create a dynamic and interactive dashboard. Each interaction is configured using a second specific editor for this task. Additional challenges were discussed including the coordination of interactions and the management of data. Furthermore, the storage of the different configuration files was discussed. Last, the integration into a portal system was analyzed in order to benefit from the functionality of such a system.

Overall, the presented frontend concept allows to create flexible and generic dashboards by configuring different components and their interactions with editors that are semantically enhanced for improved user experience.

[7]https://www.liferay.com/

8 Evaluation

This chapter evaluates the concepts defined in the four scientific contributions. There exist different strategies and methods to evaluate an architecture. Therefore, the current chapter will first give a brief overview of different methods. Next, two prototypical implementations are presented that allow further evaluation of the reference architecture. Then, the reference architecture is analyzed using both implementations based on different quality attributes. Furthermore, the REST APIs are evaluated and finally the eight requirements defined in Chapter 3 are discussed.

8.1 Architecture Evaluation Methods

According to literature and publications, there are two main methods that define how to evaluate an architecture. These methods are not designed for a specific architecture and can be used in general. The first method is called *Software Architecture Analysis Method* (SAAM). It is first defined in the publication by Kazman et al. [KB+94]. The *Architecture Tradeoff Analysis Method* (ATAM) is based on SAAM. It is closer described in a second publication by Kazman et al. [KK+00]. Both methods are so-called scenario-based methods. They evaluate a specific architecture by analysing different scenarios that each cover a functionality or a behavior of the architecture. Both methods are defined in detail in [CK+01]. The book defines different case studies that further clarify the methods.

Additionally to the basic literature and publications, there are different publications that further analyze the topic of software architecture evaluation/analysis. In [CA13] a systematic mapping study is presented analyzing the years after the publication of ATAM. The relevant publications are mostly proposing new methods based on ATAM. These methods mainly focus on a specific attribute of the architecture in question. These attributes are specific quality attributes (e.g. from the standard [ISO25010] which is use in the present thesis) focusing on a specific part of the architecture, for example the hardware. This publication clearly shows the relevance of ATAM.

The next publication [PS15] presents a survey analyzing different methods. Besides both methods already presented, the survey discussed multiple other methods. On the one hand there are methods that extend SAAM by different aspects and on the other hand methods evaluating a specific aspect of an architecture instead of the architecture as a whole. A last method can be used in order to compare different software architectures from different domains. This publication clearly shows that there are many different methods, however, most of them are based on ATAM and SAAM as also presented in the previous article. A second survey [DN02] that was conducted a decade before the first came to similar findings.

The publications presented in the previous paragraphs propose software architecture analysis methods that do not focus on specific architectures but are meant for architectures in general. However, since the present thesis is about a microservice architecture the following publication presents a process for evaluating microservice-based architectures. The article defines multiple principles for the evaluation of the architecture. Each of these principles is mapped to metrics that can be measured and evaluated. Additionally, a tool is presented allowing to extract these metrics.

Since the present thesis defines a reference architecture, this chapter will define two specific applications in the next sections that are prototypically implemented. This allows a specific evaluation based on the presented literature. Furthermore, quality attributes for the prototypes are defined in order to analyze and evaluate the architecture in a third section.

8.2 Prototypical Implementation

In order to evaluate the presented reference architecture it is important to have a (prototypical) implementation of it. In the following the prototypical implementations are described. This includes the used technologies and tools for the infrastructure, persistence layer and service implementation. First, a basic setup is defined. In a next step, this setup then is further detailed for two different applications. The first application is from the environmental domain, the second is from the energy domain. Both implementations consist of a backend setup implementing the concepts of the first three contributions and a frontend based on the concepts of the fourth contribution.

8.2.1 Basic Structure of the Prototypes

Since the reference architecture presented in the previous contributions is based on microservices, the properties related to microservices are described in figure 8.1. The structure illustrated in the figure is inspired by the reference for microservice architectures presented by Gartner [Ol15]. First, the architecture is separated into two parts: the inner architecture describes the architecture of the different services and the outer architecture describes all the components needed in order to have a stable distributed architecture. The components of the outer architecture are described in the following list including specific technologies that are used for the presented prototypes.

- **Gateway**: The gateway is the single point of entry for a microservice architecture. It allows the backend to have a single external IP and domain name. For the prototypes the infrastructure layer is realized with the container orchestration software Kubernetes. It features so-called *Ingress controllers* implementing the gateway functionality.

- **Routing & Discovery**: The routing is usually integrated in the gateway and ensures that the requests from clients are redirected to the right service. The Kubernetes Ingress controller takes care of this feature.

  Service Discovery is an important aspect for scalability. It is implemented in Kubernetes and is part of the service concept. A service in Kubernetes is a facade that uses *service discovery* and *load balancing* techniques in order to forward the routed request to a specific container instance.

- **Application Configuration**: The ability to create configurations and make them available for an application is provided by Kubernetes' feature called *Config Maps* which allow configurations to be passed inside a Container using different mechanisms.

- **Auth**: For the prototypes, security is not considered as important and, therefore, it is not in the focus. However, a concept is briefly presented in the following. The authentication and authorization of a microservice architecture can be implemented using tools evolving around the concepts of single sign-on. This concept allows the user to log-in once and access all of the services using that single authentication. In order to manage user credentials and access roles, an authentication provider is used. For the following prototypes, the software Keycloak[1] is used. The authorization is the second part of the security concept. It defines which resources a user is allowed to access. This access can have different meanings for example read only or read/write access. On the one hand, authorization can be implemented at a single point in the system and define which user has access to which service and in which manner. On the other hand, authorization can be implemented at each service separately. This allows

[1]https://www.keycloak.org/

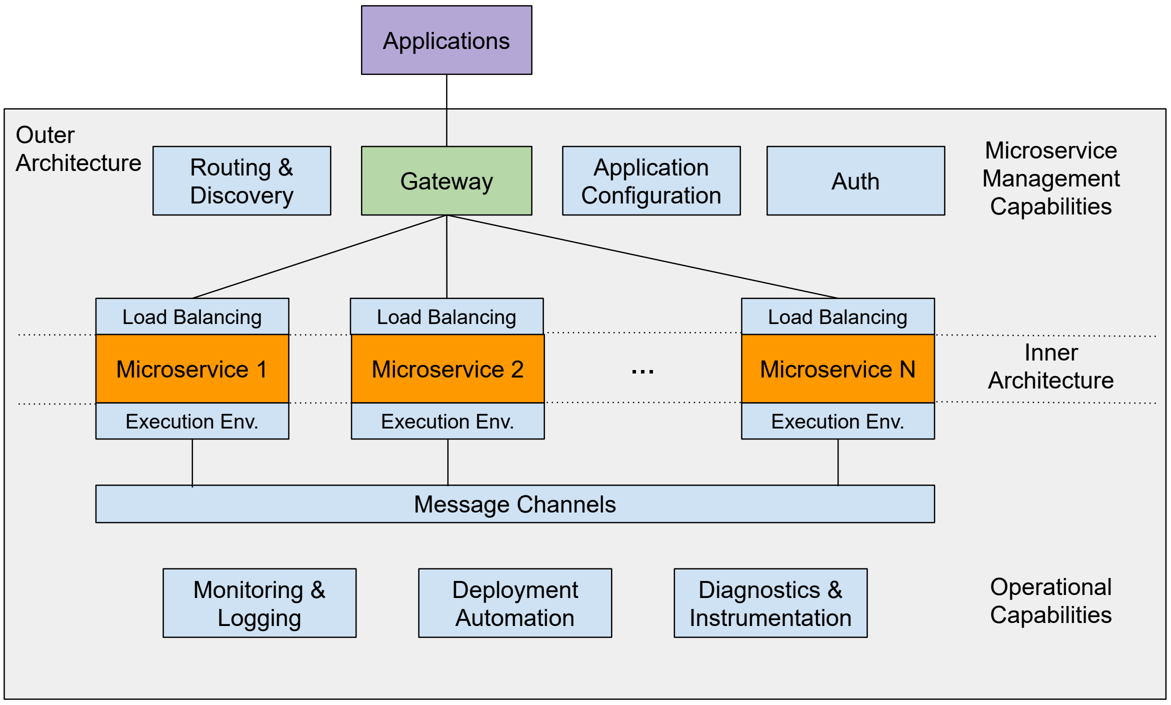


Figure 8.1: Overview of a General Microservice Architecture inspired by Gartner [Ol15]

each service to handle their own authorization autonomously. The second method is implemented for both prototypes.

- **Monitoring & Logging**: Since the prototypes are not deployed as productive software this is also not as important. However a basic monitoring using Prometheus[2] and a central logging system using Graylog[3] is implemented.
- **Deployment Automation**: Continuous integration and deployment is a very popular technique and an important part of the operation of a microservice architecture. However, since the present thesis does not cover operation aspects this is also not further evaluated.
- **Diagnostics & Instrumentation**: This is also a part of the operation of the architecture and not important for the present chapter.
- **Message Channel**: The message channel allows the service to communicate asynchronously. A good choice for this feature is RabbitMQ[4].

[2]https://prometheus.io/
[3]https://www.graylog.org/
[4]https://www.rabbitmq.com/

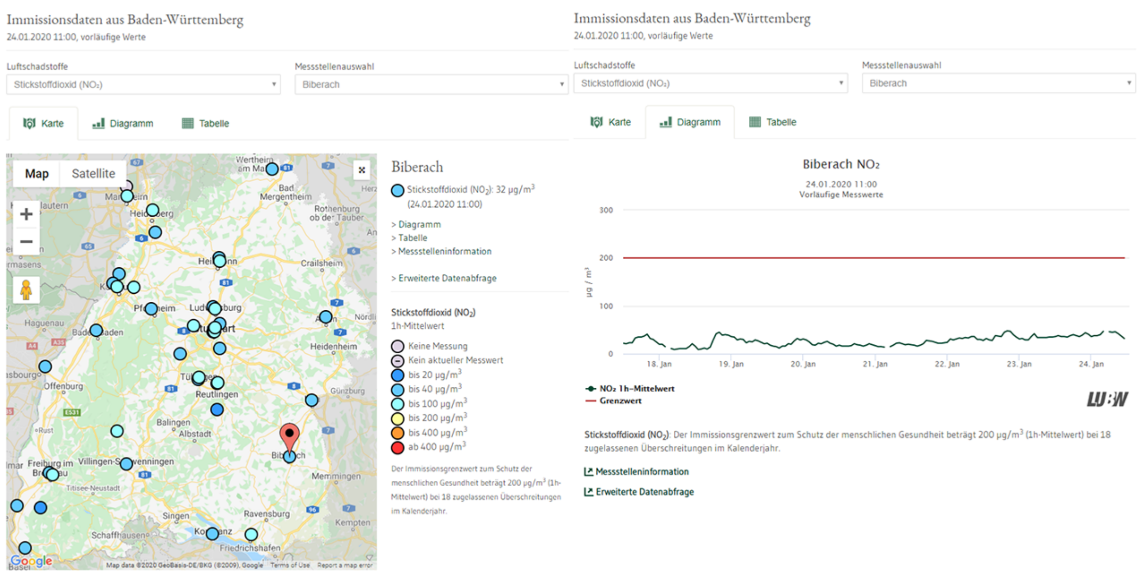
This list of microservice architecture components gives a rough overview of which tools and technologies are used for the prototypes in order to evaluate them. The last component that has to be closer looked at are the microservices themselves. They form the inner architecture. The framework used for the prototypical implementation of the services is Spring Boot[5]. It is one of the most stable and mature systems that is built for the Java programming language. Data persistence is defined abstractly, so each microservice can use a different database. Each service is virtualized in a Docker container and is deployed on a Kubernetes cluster.

After having defined the main components of the backend, the frontend is defined. The framework makes use of the web component standard as defined in the fourth contribution (Chapter 7). In order to support the implementation the Polymer library is used. It adds many convenient functionalities to improve the efficient development of web components. As described in the fourth contribution, a content management system or portal system is suitable to complement the framework's features. For the presented prototypes, Liferay CE 7[6] is used as a portal server in order to provide a central login, persistence of configuration files and an initial layout and design of the web pages.

8.2.2 Environmental Measurements Applications

The first prototype is a measurement system monitoring different aspects of the including air quality, water quality and radioactivity. Each of these fields are considered a separate application. However, all applications are embedded in the same Liferay portal server instance. The applications are run in a productive manner and operated by the LUBW (State Institute for Environmental Protection Baden-Württemberg). The infrastructure is Google Cloud and the services are deployed using Docker containers. The frontend serves as information dashboards for citizens. Figure 8.2 shows the air quality application. The left screenshot shows a map visualizing the different measurement stations at their geographical location. The colors of the stations correspond to the air pollutant that is selected in the dropdown located above the map. On the right of the map, additional information about the measurements as well as a legend are displayed. The second screenshot shows the same pollutant and measurement station dropdowns but the main content is a visualization of the historical measurement values using a line chart. Additionally, a limit is shown indicating a critical threshold. Both applications are implemented using the framework presented in Chapter 7. Additionally, the backend services are used to provide the data. The measurement

[5]https://spring.io/projects/spring-boot
[6]https://www.liferay.com/



Stickstoffdioxid (NO$_2$): 32 µg/m^3 (24.01.2020 11:00)

Der Immissionsgrenzwert zum Schutz der menschlichen Gesundheit beträgt 200 µg/m^3 (1h-Mittelwert) bei 18 zugelassenen Überschreitungen im Kalenderjahr.

Stickstoffdioxid (NO$_2$): Der Immissionsgrenzwert zum Schutz der menschlichen Gesundheit beträgt 200 µg/m^3 (1h-Mittelwert) bei 18 zugelassenen Überschreitungen im Kalenderjahr.

Figure 8.2: Screenshot of Air Quality Application

stations are stored as master data in the respective service and the historical values are stored in the univariate measurement service. The semantics service is used in order to describe the semantics of a measurement station.

8.2.3 Energy Dashboard Application

The second prototypical implementation is from the energy domain. The application shows a dashboard consisting of different dashboard components that visualize information about a specific building selected on a map. The dashboard is implemented using the same framework as for the first prototype. Figure 8.3 shows a dashboard consisting of four components. The first component is a map showing a campus. The different buildings of the campus are highlighted on the map and can be selected using mouse or touch input. After selecting a building, the other components update their state according to the selected building. The second component shows an overview of departments residing in the building. The two next components are dropdowns allowing the user to select different measurement devices associated with the building. These devices measure different energy related metrics (e.g. heat or electricity consumption). The last two components are visualizing these metrics for a specific time interval. This prototype is an early version and it is neither fully developed nor styled at all. However, the technologies discussed in the present thesis are all used in the background. The buildings and the measurement devices are each represented by master data objects. The time series data is managed in the measurement services and linked to

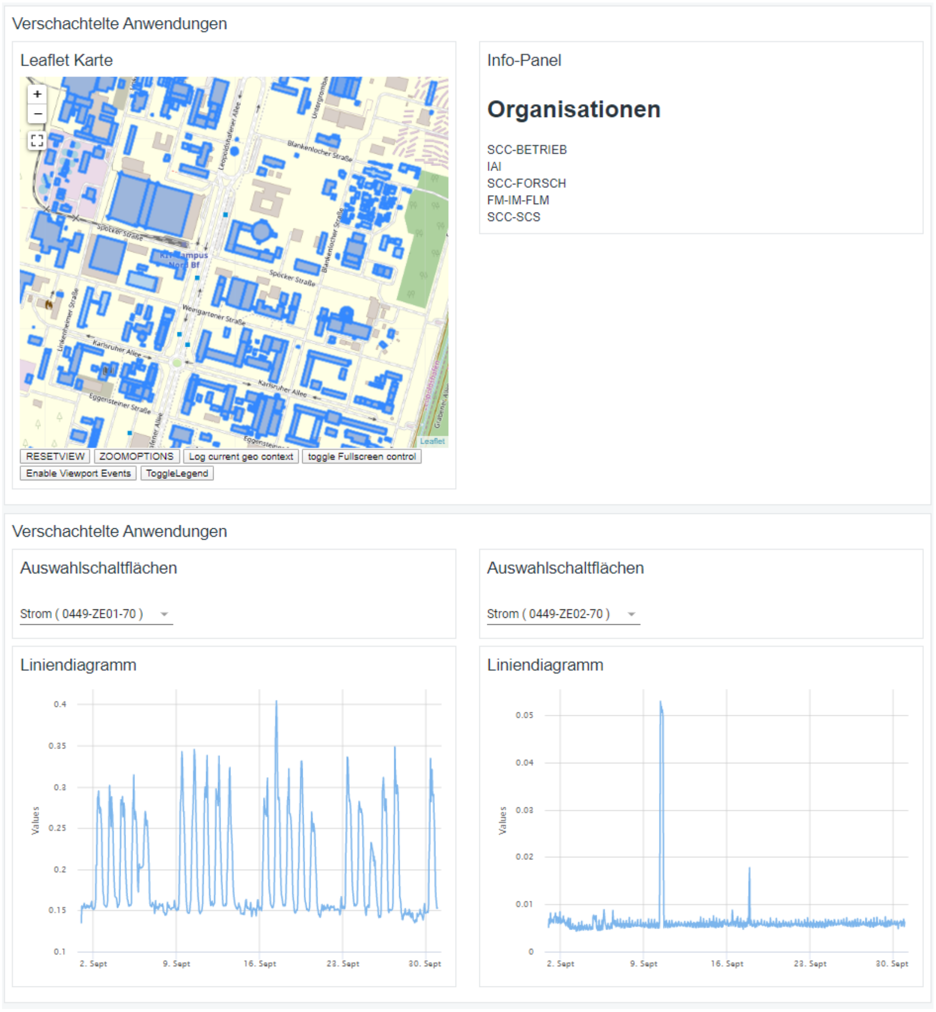


Figure 8.3: Screenshot of Building Dashboard

the measurement devices. Additionally, the semantics are included in the semantics service and describe the different master data objects. The frontend framework uses the semantic information in order to give the dashboard customization options that are aware of these semantics. Chapter 7 discusses these features in more detail.

8.3 Quality Attribute Analysis

This section analyzes the reference architecture by looking at different quality attributes for both implementations. The quality attributes are selected based on the ISO/IEC 25010:2011 standard as already mentioned in Section 8.1. The quality attributes are further divided in different sub quality attributes that allow a more detailed analysis. In the following, each quality attribute is discussed separately in order to evaluate the reference architecture.

8.3.1 Functional Suitability

Functionality is the capability of the software architecture to provide the required functions. The functionality mainly relies on the fulfillment of the requirements. However, this is discussed later in this chapter. The attribute can, however, be analyzed in a more abstract way. Since the reference architecture is based on microservices, the functionality mainly depends on this architectural style. Microservices are designed to be technology independent which allows to implement them using different programming languages, libraries and databases. The presented reference architecture preserves these qualities. Each service can be implemented with the technology and databases of choice. Since each service has its own database, persistence is not shared by multiple services. This avoids one of the bad smells defined in [TL18].

Additionally, the microservice architecture allows for optimal distribution of the functionality on multiple services and, therefore, teams developing specific services. The structure of the organization using the reference architecture can therefore be used optimally. The cut of the services is also thoroughly done and presented in Chapter 3.

Another bad smell is the usage of too many standards. The data services and the supporting services are all designed using the JSON data format. Most of the other standards, for example JSON schema and JSON API, are used in the same way and have good synergy with JSON itself. The exception is the digital asset service that uses another well established standard. However, it is completely independent from the other services and the implemented CMIS API is an ideal match for the requirements. The rest of the used standards, libraries and technologies will be defined by the implementation and are not part of the reference architecture. Section 8.5 discusses the requirements defined in chapter 3 in more detail.

8.3.2 Maintainability

Following the standard [ISO25010], maintainability describes the capability to modify the software. In order to modify the software in a sustainable way, it is very important for microservice architectures to ensure specific quality attributes.

One of those attributes is *modularity*, i.e. that different services are independent from each other. This is a very crucial aspect since a highly coupled network of services is not maintainable in the long run. The services of the reference architecture have separated

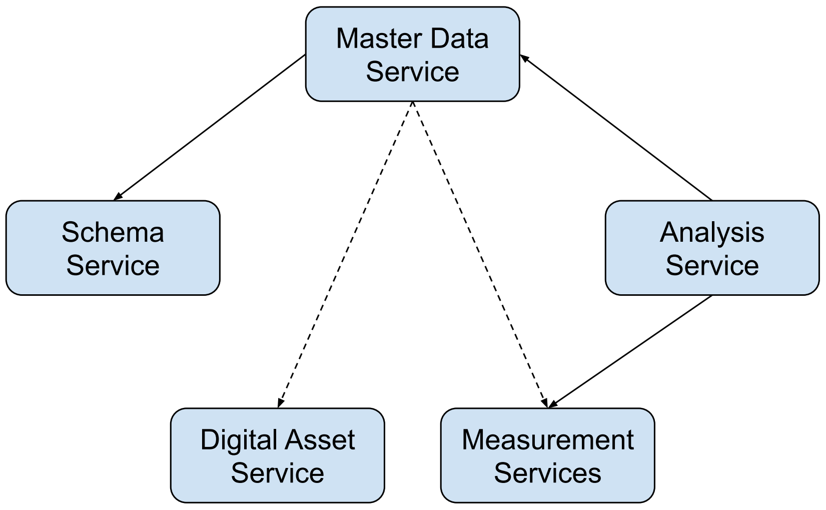


Figure 8.4: Data and Service Dependencies

responsibilities and are designed to be as modular as possible. The data stored in the services has dependent relationships between each other, however, the master data service has a data source of each data object which is managed in one place. This allows the master data service to operate independently if the applications requesting the data are not interested in the actual data of the referenced services.

The first type of dependency is, therefore, that the applications have to rely on the consistency and availability of the other services' data. This dependency is shown in Figure 8.4 with dashed arrows. A second type of dependency is the service dependency which are defined between the semantics service and the master data service and between the analysis service and both the master data service and the measurement service as seen in Figure 8.4 with solid arrows.

However, the domain-driven design allows to control the relationship defining a customer/supplier relationship between the semantics service and the master data service and a conformist relationship between the analysis service and the two data services that are analyzed. Additionally, between the master data service and the respective other data services, partnership relationships are defined.

This allows the software engineers to control maintainability since the relationships are clearly defined. A bad smell that has to be avoided is cyclic dependencies [TL18] between services. This can be excluded since the dependency graph does not have such a cycle. This low dependency of the different services shows that the reference architecture is modular and the services are low coupled.

A second sub quality attribute besides modularity is *reusability*. It helps maintaining the software since components can be reused instead of only implementing new components. In order for the reference architecture to be reusable, it is important that it is not bound to a specific domain (e.g. energy or environmental domain). This is achieved by defining a generic domain for the reference architecture which is compatible with many specific sub domains. By separating the microservices according to different data types, each service can be used very generically as long as the data used by the application has one of the types supported by the architecture.

Analysability is a third sub quality attribute that describes the ability of logging and monitoring the system. This leads to metrics that can be measured and logs that can be analyzed in order to facilitate maintenance. Especially errors or problems with the software can be found and fixed faster with the appropriate information.

Besides fixing bugs or solving errors, an architecture is also supposed to be updated and changed in order to feature new functionality. The according sub quality attribute that defines changing software is *modifiability*. Since the reference architecture is based on microservices, modifiability is facilitated. Each service can be updated or even replaced separately.

8.3.3 Reliability

Reliability is one of the most important attributes. Distributed systems such as microservices are designed to be scalable and fault tolerant. There are different mechanisms that ensure fault tolerance. First, there is a common strategy to cope with requests that are sent to a service that is not working correctly. The strategy is called circuit breaker and it ensures that such requests are redirected to a working instance of the service and the applications that sent the request never notices the redirection except a possible longer delay than usual. Second, fault tolerance is often solved on the database or even file system level. These mechanisms lead to a sub quality attribute that is very important for reliable systems: availability. The requirement for systems to be accessible 24/7 is getting more and more important which primarily requires the system to be available.

The next attribute that ensures high availability is *scalability*. Many systems, especially measurement systems, have different amounts of load over time and therefore have to scale accordingly in order to have enough resources to cope with the specific amount of load. A Kubernetes cluster has different important features that favor scalability of the deployed microservices. First, each Kubernetes cluster allows to scale microservices horizontally by

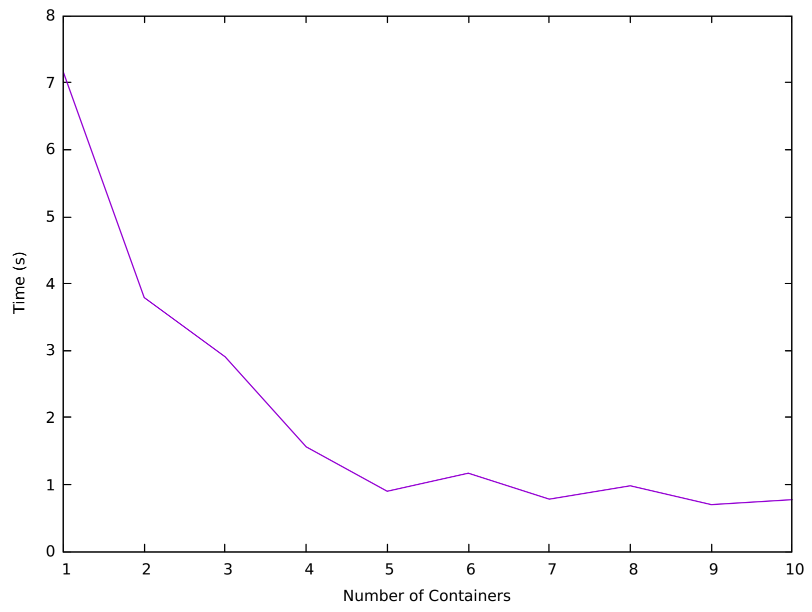


Figure 8.5: Experiment: Scalability of Read Operations

running multiple instances of the same service in parallel. The services are distributed to the different cluster nodes in order to get the most out of the cluster's performance. Second, each Kubernetes cluster has built-in load balancing strategies routing requests to the different instances of a service.

In order to further evaluate the scalability of the reference architecture, two experiments are presented in the following. For both experiments it is expected that the time the read and write requests take is reduced by deploying more containers each running an instance of the multivariate measurement service. The time should decrease by 50% if the amount of containers is doubled. Both experiments are executed on a four node cluster. Each node consists of 32 CPU cores and 128GB RAM.

The first experiment focuses on read operations. The containers provide a time series that consists of the amount of data points that is gathered by a measurement device that records one value each second for a year. The data is requested in parallel by 32 clients generating the load for the experiment. Figure 8.5 shows the results of the experiment. The curve clearly shows that the time is reduced by increasing the amount of containers.

The second experiment evaluates the scalability of the multivariate measurement service in regard to write operations. Figure 8.6 shows a similar behavior than the previous one. The amount of data that is written is one million data points distributed over 32 different clients.

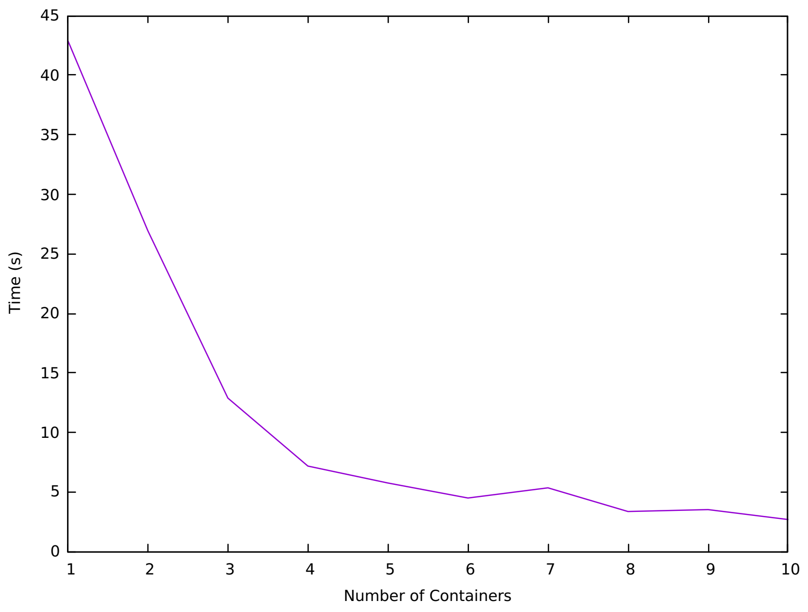


Figure 8.6: Experiment: Scalability of Write Operations

This amount is taken from a publication [SC+14] that uses 24 cluster nodes for a scenario that has six million smart meters. Since the cluster that the current experiments are executed on only consists of four cluster nodes, the number of devices is reduced to one million smart meters.

Both experiments show that the multivariate measurement service is scalable on an appropriate cluster.

8.3.4 Compatibility

This quality attribute is an addition to the ISO/IEC 25010:2011 standard (in contrast to its predecessor ISO/IEC 9126). The *compatibility* is closely related to the *deployment of the services and the infrastructure*. In the world of cloud computing it is very important to support different infrastructures and the interoperability of the deployment of software components to these infrastructures. It is a common use case that a system is deployed to both an infrastructure for testing purposes and an infrastructure for production. It is therefore important that the system consisting of the different services and components is deployable to both infrastructures. The prototypical implementations meet this need by making use of container virtualization. Each service and additional components such as microservice management components or frontend components are deployed as different containers. Having the container virtualization as an abstraction layer, the system is compatible with any infrastructure supporting container virtualization. This allows the reference architecture to be highly compatible with different infrastructures.

Additionally, a second form of compatibility is the ability for *third-party applications* to communicate with the architecture. This is favored by the REST APIs the different services provide. An application only has to use the HTTP(S) protocol or streaming protocols. Since these protocols are very popular in every programming language it is no big obstacle for applications to be compatible with the communication technologies of the reference architecture.

Finally a last aspect of compatibility is that the reference architecture is *domain independent* and therefore compatible with different domains that have the requirement to manage measurement data. The different data services and the semantics service allow ingesting and managing different semantics and therefore different business domains.

8.3.5 Performance Efficiency

The *performance efficiency* is a very important quality attribute to ensure that the system is able to cope with the desired amount of data (read and write operations) in an adequate time interval. For the reference architecture especially the measurement services are reliant on the performance of the system. For the evaluation of the performance, different experiments are presented and discussed in the following. All of the experiments were executed on a single cluster node that features 32 CPU cores and 128GB RAM. Note that these are not all the possible experiments but only a representative selection.

The first experiment analyzes the performance of the RESTful interface in comparison to the streaming interface. Both interfaces are implemented for the measurement services as described in the concept. The hypothesis is that the RESTful interface is much slower than the streaming interface. Especially for large data sets, this gap should increase.

The analyzed measurement service is the univariate measurement service that is implemented with both interfaces. The data sent for the experiment ranges from 100.000 to 1.000.000 data points (these numbers are chosen based on the scenario described in the experiments in Section 8.3.3). As seen in Fig. 8.7, the RESTful API is slower. This is the case since the RESTful interface has much more communication overhead than the streaming interface, which is a single TCP connection that is open for the entire duration of the stream. Additionally, the gap increases with the number of data points. This can be explained by

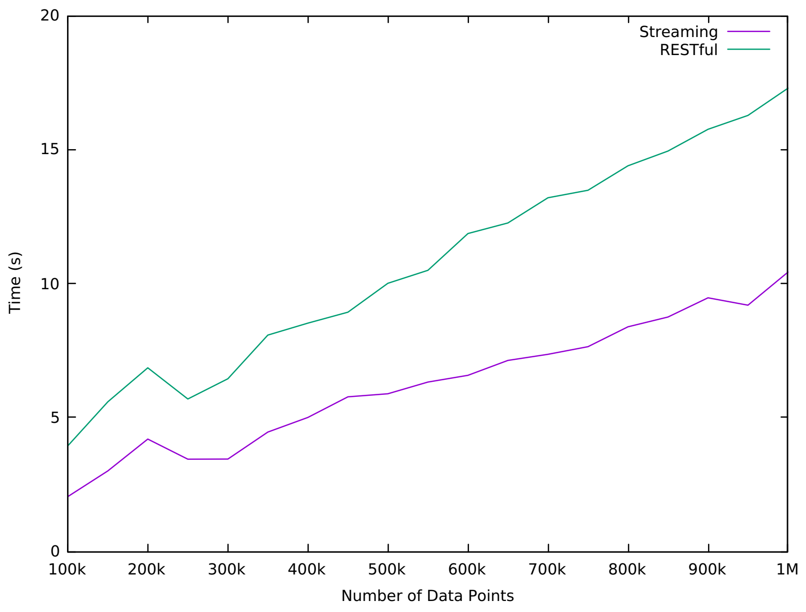


Figure 8.7: Experiment: RESTful Interface vs Streaming Interface

the fact that the data can not be sent with a single requests since it is too large. Hence, each request adds more overhead to the overall experiment.

The second experiment analyzes the overhead the measurement services have while writing data. The hypothesis is that the communication and management overhead of the service is low and not a bottleneck for writing data. The experiment compares the time that the service needs with the time that only the database needs and uses this to determine the overhead. The streaming capabilities of the service are used in order to have a higher throughput as seen in the first experiment. For the streaming API, the technology GRPC[7] is used. There are different supported implementation strategies. For the evaluation, two different ways have been analyzed. Both strategies are depicted in Fig. 8.8.

The first strategy confirms each data set that is sent synchronously and is written to the database immediately. This strategy gives constant feedback and builds a very robust system. The second strategy sends the data asynchronously. The confirmation is done once at the end of the stream. The data is written continuously to the database without waiting for new data or sending confirmations. The results of both studies are depicted in Fig. 8.9 and 8.10. For both experiments, the database InfluxDB[8] was deployed as a single node installation. The metric that is measured for the evaluation of the experiment is the time t the whole request

[7]https://grpc.io/
[8]https://www.influxdata.com/

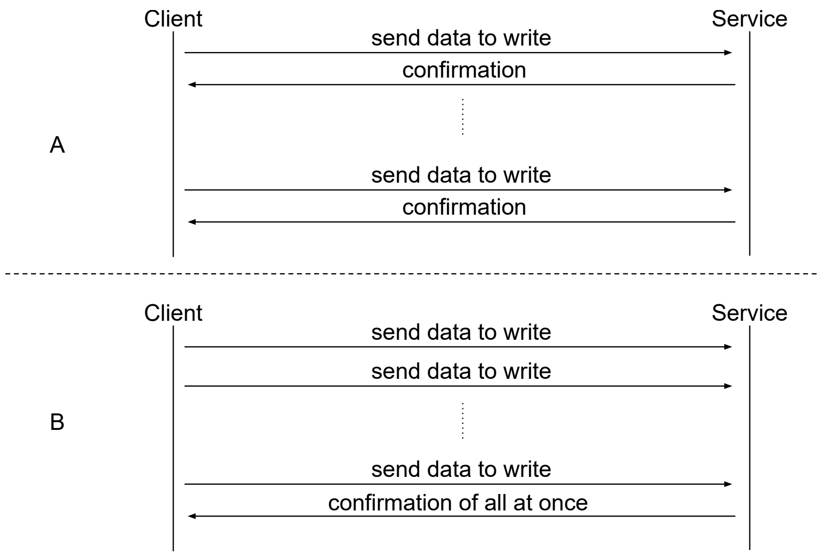


Figure 8.8: Two Strategies to Implement Streaming with GRPC

takes. t is divided in the overhead t_o and the time it takes to write the data to the database t_{db}.

$$t = t_o + t_{db}$$

Fig 8.9. shows the experiment for the synchronous strategy. It is clear that the overall request time is dominated by the database write time. However, the overhead is a substantial amount of the time and is growing in a linear way similar to the overall time. This behavior can be explained by the synchronous aspect that requires a confirmation for each data set that is sent. This leads to a high communication overhead.

Fig. 8.10 shows the same experiment with the second, asynchronous, strategy. The overhead is much lower and nearly non-existent in contrast to the total time. This behavior can be explained by the fact that the overhead that is measured for every data set for the first strategy of the experiment is only measured once for the second strategy. The second strategy fulfills the hypothesis.

The last two experiments are evaluating the semantics service. First, the write performance is measured. It is expected that schemas can be written in a reasonable amount of time and that the time increases linearly with the amount of schemas written. However, schemas are usually not changing much and have different requirements than measurement data. The prototypical

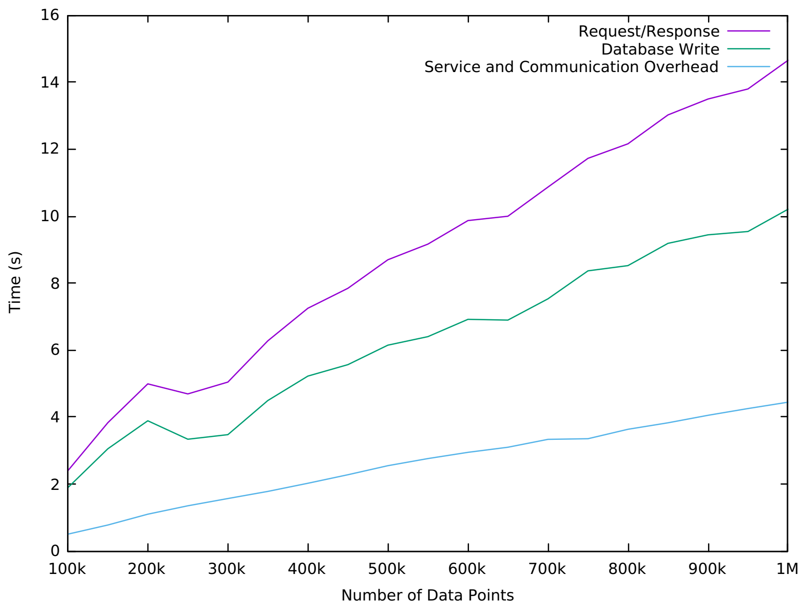


Figure 8.9: Experiment: Synchronous Data Stream

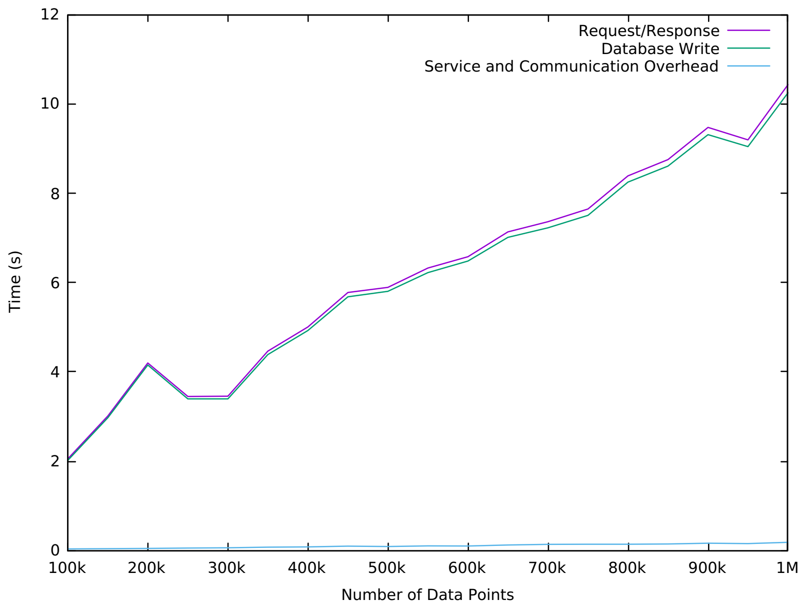


Figure 8.10: Experiment: Asynchronous Data Stream

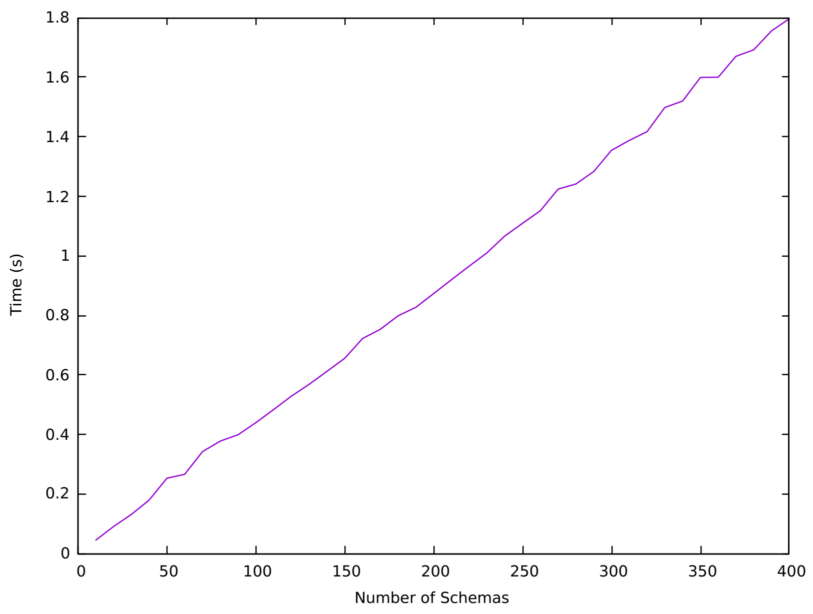


Figure 8.11: Experiment: Schema Write Performance

semantics service uses MongoDB[9] as a backend database (also deployed on a single cluster node). Fig. 8.11 shows the result of this first experiment. The curve rises as expected and 400 schemas can be written in 1.8 seconds which is reasonable for the semantics service. The number 400 is chosen since the standards IEC61850-7-2, IEC61850-7-3 and IEC61850-7-4, which form the core of the IEC61850 standard [Ma06], consist of around 400 schemas if one maps the standard to the semantics service.

The second and last experiment evaluates the schema resolving feature. Since schemas can be linked in a complex way, the resulting schema tree can be quite large. It is, however, important that the resolving of such complex schemas is also done in reasonable time. It is expected that schemas can be resolved in a few seconds in order to avoid large delays in applications. The experiment uses a schema tree with a depth of 10 to 400 schemas. Fig. 8.12 clearly shows that 400 schema resolve operations are executed in less than 1 second. Since not many changes happen to schemas in a short period of time, such schemas can easily be cached once they are resolved. This last experiment also fulfills the expectations.

These experiments showed that the quality attribute performance efficiency is covered and the results are satisfactory.

[9]https://www.mongodb.com/

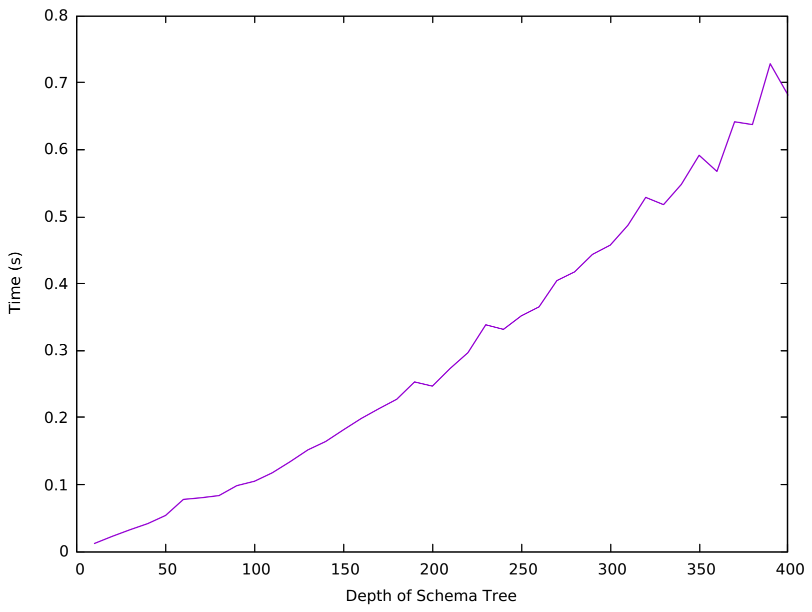


Figure 8.12: Experiment: Performance of Schema Resolving

8.3.6 Portability

Portability is the capability of changing the environment of an architecture. This mainly depends on the deployment of the software and its compatibility with different infrastructures. The portability is closely related to the first aspect discussed for the compatibility attribute. This aspect already showed that the container virtualization of every component of the architecture leads to high compatibility with different infrastructures and therefore also a high portability to different environments. The only mandatory requirement is that the environment supports container virtualization in general. This aspect also favors the installability of the system.

8.3.7 Usability

First, the *usability* of the backend part of the reference architecture is discussed. The usability of the backend services targets the interfaces, primarily the RESTful APIs. For the measurement service also the streaming API is part of the architecture's interface. The reference architecture defines the APIs for each service in detail. They are defined in a way that they use a uniform structure of the URLs, query parameters, semantics and data formats. This creates a high synergy among the services and a good practice to give users and applications a flat learning curve. In the next section, the REST APIs are evaluated in more detail.

The second part of the reference architecture is the frontend framework that also has a need for high usability. The presented editors provide help for the admin/operator by different semantic descriptions of the web components and their interactions. Additionally, the fact that no programming skills are required in order to create dashboard applications further improves the usability of the frontend.

8.3.8 Security

As a new addition to the top quality attributes of the 25010:2011 standard, *security* became obviously more important over the years. The present thesis does not focus on security, however, a basic concept for authentication and authorization is briefly described in Section 8.2.1.

8.4 Evaluation of RESTful APIs

After having seen the analysis based on different quality attributes targeting both general software systems and microservices in particular, this section will evaluate the RESTful API since it is the most important part of a microservice. Most of the time it is the only interface that is seen from the outside and it is the only way to interact with the service. Therefore it is very important that the RESTful APIs are designed following best practices. The publication [GG+16] gives a very good overview on different quality aspects for a RESTful API. In the following, the checklist that is presented in the publication is used to evaluate the APIs of the different services.

8.4.1 Versioning

The *version* of the services is structured in a consistent way across the reference architecture. A major version breaking the API is included in the URL. The rest of the service's version (minor and subminor parts) is defined not to be API breaking and therefore not included. However, this concept diverges from the suggestion of completely leaving out any versioning and relying on hypermedia in order to provide the applications the right URLs and version of the services. Each application, however, has to cope with a specific version that it is implemented for. Therefore the choice to only integrate an API breaking version in the URL is not contradictory to the best practice [GG+16].

8.4.2 Naming

The *naming* of the REST API is mainly based on the domain model and the ubiquitous language which defines the resource names used in the base URLs of the services. The rest of the naming is depending on the data itself. The master data service, for example, includes the schema name in the URL which is defined by the user or application creating or updating schemas. Therefore, the applications having this responsibility also have to follow the best practices [GG+16].

8.4.3 Resource Identification

Additionally to the previous item on the checklist, not only the naming but also the *identification of a resource* has to be checked. In the presented reference architecture, each REST API uses identifiers for the resources they manage. The identifiers are chosen to be unique for each resource instance.

8.4.4 Error Handling and Troubleshooting

The thesis does not focus on this part, however the prototypical implementations realize basic error handling following the best practices [GG+16].

8.4.5 Documentation

The documentation of the REST API is also not in the focus of the thesis. However, Chapter 7 presents the importance of documentation and how to implement it using principles of generated documentation that does not turn obsolete. This can be transferred to the REST API concepts as well by automatically generating documentation based on the source code or some describing artifacts. An example technology for this is the OpenAPI standard[10] and the Swagger tools supporting this concept. A possible combination with the HATEOAS principle proposed in [GG+16] has to be further investigated in future work.

[10]https://swagger.io/specification/

8.4.6 Usage of Query Parameters

The concept around query parameters is the main part of the RESTful API design in this thesis. The complexity of the interface is implemented using query parameters. The presented concept reduces some of the complexity by distributing it to different query parameters. The main query parameters e.g. for sorting, filtering, pagination, etc. are identically defined in [GG+16]. The only difference lies in some details. Additionally, most of the ideas for the concepts find their roots in the JSON API standard. This confirms that the query parameters are conceptualized according to best practices [GG+16].

8.4.7 Interaction with Resources

The different services all use the semantics of the HTTP methods to use them appropriately. The GET method is only used to return data. This operation is idempotent which conforms with the best practices [GG+16]. The POST method is mainly used to create a new resource or for more complex operations such as the execution of an analysis, which which is conform with the lack of idempotency for this method. The PUT and DELETE methods are used to update and delete resources. Both of these actions are idempotent. The PATCH method is not used at all in any of the services since only the PUT method is used in order to update resources. Additionally, the best practice for the GET methods is to only return data if it was modified [GG+16]. This can be achieved with an empty response and the HTTP status code 204 (Not Modified). The second way is to cache the result and return it with minimal effort for the service. This is included in the concept of the analysis service which is the service with more complex operations.

8.4.8 MIME types

The services of the reference architecture use the JSON format as a basis wherever it is possible. The measurement services use the MIME type in order to route the request to the appropriate service behind the facade. The semantics service defines more MIME types allowing applications to use content negotiation to select the right data format. For specific applications more MIME types can be added if needed.

8.5 Requirements Analysis

This section evaluates the reference architecture and the frontend framework based on the requirements defined in Chapter 3.

8.5.1 R1 Architecture Based on Microservices with RESTful and Streaming APIs

The first requirement needs the proposed reference architecture to be based on microservices. To achieve a proper microservice architecture, the first contribution methodically designs the microservices with the help of domain-driven design principles. First a ubiquitous language is defined in order to define a precise and well understood vocabulary. In a second artifact, this language is used to define the different domain objects that are part of the whole architecture. This allows the domain to be cut in different subdomains and define their relationships in the context map. The different parts that are identified to be different bounded contexts are then mapped to microservices. This process ensured the microservices to be well cut and to have responsibilities that are independent from the other services. A loosely coupled architecture is developed and the microservice part is therefore fulfilled. Additionally, R1 requires the services to support RESTful APIs and streaming APIs. This is also the case and therefore R1 is completely met.

8.5.2 R2 Scalability

The second requirement needs a scalable system that allows to cope with many requests and high velocity. The choice of having a microservice-based architecture is already a good start since it allows to scale each service independent from each other. Additionally, the domain-driven design approach leads to an architecture which has separated the responsibilities in a way that each bounded context manages different types of data or information which is a good separation in order to scale the architecture based on the data type. This allows, for example, to have the measurement services scaled up since they probably need more resources because of the velocity or volume of the data. The semantics service for example probably needs less resources since the operations are more basic and not requested as frequent as measurement data. Furthermore, since each service has its own database, also the underlying persistence layer can use scalable databases and yet independent from the persistence layers of other services. The definition of RESTful and streaming interfaces support the scalability since there are a lot of options to communicate with the services. This short analysis shows that each layer of a service is defined to be scaled and together with the results presented in Section 8.3.3 the second requirement is fulfilled.

8.5.3 R3 Generic and Reusable Services

In the first and second contribution the measurement services are presented. The RESTful API is designed to be generic since it is not defined for a specific type of time series or a specific domain. The measurement service was introduced in order to create a facade for concrete measurement services managing specific types of measurements. Additionally, the API of this abstract service was designed following best practices from multiple time series databases and general RESTful API best practices [GG+16]. This also counts for the other services and therefore concludes that the requirement is met.

8.5.4 R4 Various Measurement Types

In order to fulfill requirement R3, the generic measurement services are defined. As mentioned the service facade abstracts from specific services that support different types of measurements. The concept defines three such services. First, a service is defined that allows the management of simple, univariate measurement data. Second, a service that manages multivariate measurements is defined. The third service is meant to process more complex measurements called sampled values. These three services show that the reference architecture supports multiple types of measurements. The advantage is that each service has its own database by design and therefore each measurement service can use the optimal database for that type. This also concludes the fulfillment of the fourth requirement.

8.5.5 R5 Master Data

As shown in Chapter 3 besides measurement data and the corresponding services, a second important data type has to be supported: master data. Master data management is very important in order to store information about business objects. Especially objects that are linked to measurement data are important in order to store additional data besides the raw measurement data. Chapter 5 describes the functionality of the master data service in detail.

Next, the second contribution introduces special master data objects representing metadata for measurements or digital assets. This concept is called *data source*. A data source represents a reference pointing to a specific measurement or digital asset. This relationship connects a measurement or digital asset to a master data object defining the metadata further describing the primary data. The same relationship concept is used for master data objects that represent business objects. Additionally, the master data service defines a CRUD-based API that provides functionality to manage master data. Especially the read operations offer a wide range of filter methods allowing applications to highly customize their queries. These concepts lead to a fulfillment of the requirement R5.

8.5.6 R6 Semantics

The next requirement focuses on the semantics service. The third contribution presents concepts that meet this requirement. The semantics service is based on multiple concepts supporting the other services of the reference architecture and frontend applications that need semantic information and descriptions about the data stored in the measurement system. These concepts include a rich API to request schemas and their metadata, a concept for grouping multiple schemas in so-called classes, a versioning of those classes and a way to reference other schemas as well as to resolve those references if needed.

8.5.7 R7 Data Analysis

The following requirement defines the need for an analysis service. This service uses the data services and processes their data in order to compute statistics or run data analysis jobs. In Chapter 6 the concept for this service is presented in detail. A flexible API is presented that allows the computation of a broad range of algorithms. Since these computations can take a lot of time which might be an obstacle for some applications, the service provides an additional option to compute the jobs asynchronously. The third contribution fulfills therefore this requirement.

8.5.8 R8 Semantics-aware Frontend

The last requirement that is discussed targets the frontend. The fourth contribution presents a framework that has been conceptualized as one possible application using the backend architecture. The frontend framework allows operators to create dashboards in a flexible and highly customizable way. The domain the application is implemented for is not important since all semantic information is provided by the backend services and can be processed in the semantics-aware frontend. Additionally, the frontend is implemented in a modular and generic way using the web component technology. The requirement is met.

8.5.9 Summary

Table 8.1 shows the analysis of the related work and visualizes that the present thesis fulfills all of the requirements which is the aim of the evaluation.

Article	**R1**	**R2**	**R3**	**R4**	**R5**	**R6**	**R7**	**R8**
A Software Architecture-based Framework for Highly Distributed and Data Intensive Scientific Applications [MC+06]	(X)	X	(X)		X	X		
Experience on a Microservice-Based Reference Architecture for Measurement Systems [VL+14]	X	X	X	(X)			X	(X)
Energy Cloud: real-time cloud-native Energy Management System to monitor and analyze energy consumption in multiple industrial sites [SC+14]	(X)	X	(X)				X	(X)
Next-Generation, Data Centric and End-to-End IoT Architecture Based on Microservices [DB18]	X	X	X			X		
Reference architecture for open, maintainable and secure software for the operation of energy networks [GM+17]	(X)		X			X		
Designing a Smart City Internet of Things Platform with Microservice Architecture [KJ+15]	X	X	X		X	X		
SEMIOT: An Architecture of Semantic Internet of Things Middleware [KK+16]	(X)	X	X			X		
FIWARE [FIWARE]	(X)	X	X	(X)	X	X	X	X
Present Thesis	X	X	X	X	X	X	X	X

Table 8.1: Overview of Requirements that are Fulfilled by Related Work

8.6 Conclusion

The evaluation chapter analyzes the reference architecture based on two specific applications that are implemented using the reference architecture as a basis. In order to have a detailed analysis, the chapter looked at different aspects: different quality attributes, best practices for RESTful APIs and the requirements that are described in Chapter 3. These different aspects are analyzed according to well known standards, methods and publications in order to have meaningful results.

The presented prototypes are covering both domains the present thesis focuses on: the energy and the environmental domain. The result of this chapter clearly shows that the four contributions fulfill the eight different requirements. Additionally, the analysis of the quality attributes shows that the overall reference architecture not only fulfills the specified requirements but also a row of non-functional requirements that are very important for a microservice-based architecture.

This chapter shows that the scientific problems and the resulting requirements are fulfilled and therefore the goal of the thesis is reached.

9 Conclusion

In the following, the major results of the thesis are summarized and an outlook is given. First, the thesis presents specific problems identified in the context of microservice-based architectures for measurement systems. The focus lies on the energy and environmental domain, however, the reference architecture is conceptualized generically. The problems are further analyzed in detail by defining different needed requirements for both domains. In a next step, these requirements are used in order to analyze the most relevant work in the form of scientific articles and publications. The result of this analysis is that not all of the requirements are fully met by a single reference architecture in the literature and therefore four scientific contributions are presented which form the core of the present thesis.

The first contribution introduces the design process of the reference architecture. This process is supported by the principles of domain-driven design (DDD). The first DDD artifact is the ubiquitous language that builds the basis for the reference architecture by defining the most important terms. Next, the domain model of a measurement system is deduced from this first artifact. The domain model defines the different aspects of the reference architecture in more detail without any technical context. This domain model is furthermore refined by different relation views showing the relationships between the different entities of the domain model. Additionally, the context map is the artifact that bundles these entities in bounded contexts. The context map also shows the relationships between these bounded contexts in order to define the governance of the interfaces. Last, the first contribution presents a technical view that also suggests changes to the context map. The final reference architecture is then presented in more detail in the following contributions.

The data services of the measurement system are presented in the second contribution. The main concept is that each data type (measurement, master data and digital asset) is managed by a separate service. Since there are different measurement types that are identified in the thesis, each type is mapped to a separate measurement service. In order to have a clear interface to the outside as well as to the other bounded contexts, these measurement services are connected to a facade that unifies the REST APIs of the specific measurement services. These REST APIs and also the interface of the master data service are further described and discussed. The second contribution introduces a central API concept that is used by all of the services. The digital asset is an exception since it is defined by using an already existing standard API without any changes to it.

The third contribution introduces a core concept that covers the semantics of the measurement system. With the requirement of having generic services which are not built for a specific business domain, the semantics of business data is not included in the service implementations. This fact raised the need for a service for the management of semantic information in order to have a measurement system not only managing data but also providing information on structure and semantics of the data. A second service complementing the other services is the analysis service. It is also generic and performs any data analytics jobs using the content of the data services in the measurement system. The five services are all important for the energy and environmental domains the reference architecture focuses on.

A last contribution presents an application that uses the reference architecture. It is also conceptualized in a generic way. It allows to evaluate the reference architecture in more detail. The application consists of a frontend architecture that allows to build generic dashboards for a specific business domain using the different backend services. The data services are used in order to visualize the data and the semantics service is used in order to interpret the data on the frontend side. This allows to increase the user experience by supporting the creation and customization of the dashboards.

The last chapter evaluates the reference architecture using well known methods and standards as well as the requirements that are presented in the third chapter. The analysis of the reference architecture shows that it fulfills not only the functional requirements but also non functional requirements such as scalability and performance efficiency. Additionally, also the REST APIs are evaluated in more detail and this analysis shows satisfying results. Furthermore, the performance efficiency is validated by conducting different experiments. The results of this chapter show that the contributions define a reference architecture and a frontend application that fulfills the goal of the thesis.

The present thesis defines many new concepts that can be further extended in future work. An example for such an extension is the rework of the digital asset service in order to integrate it more in the reference architecture by defining an alternative REST API that follows the unified concept of the other services. This allows the realization of a more lightweight service that does not have to implement the complex CMIS standard. In this case, the meta data of the digital assets can be managed by the master data service similarly to the measurement services.

A further extension of the thesis is the introduction of the backend for frontend (BFF) concept in the reference architecture. This allows to create application-specific backend services that collect the necessary data or bundles requests and manages the application context. Furthermore, this allows to get rid of some of the complexity in the frontend applications and to have APIs that are using the semantics of a specific application.

A last example for future work is the analysis of the DevOps concepts and an introduction of recommendations for the operation of measurement systems using CI/CD concepts and the advantages of automation. This automation can even lead to the generation of documentation and code. The introduction of such operation concepts might lead to new scientific topics that are very important for the future since many legacy applications have to be migrated to cloud technologies and these aspects support this migration in many aspects.

References

[AF15] Martin L. Abbott, Michael T. Fisher: The Art of Scalability: Scalable Web Architecture, Processes, and Organizations for the Modern Enterprise, Addison-Wesley Professional, 2nd ed., ISBN 0134032802, 9780134032801, 2015.

[Al18] E. Al-Masri: QoS-Aware IIoT Microservices Architecture, in 2018 IEEE International Conference on Industrial Internet (ICII), pages 171–172, 2018.

[AA+16] N. Alshuqayran, N. Ali, R. Evans: A Systematic Mapping Study in Microservice Architecture, in 2016 IEEE 9th International Conference on Service-Oriented Computing and Applications (SOCA), pages 44–51, 2016.

[An16] C. Anderson: James Phillips on Service Discovery, IEEE Software, vol. 33, no. 6, pages 117–120, 2016.

[AA+18] A. Archip, C. Amarandei, P. Herghelegiu, C. Mironeanu, E. şerban: RESTful Web Services – A Question of Standards, in 2018 22nd International Conference on System Theory, Control and Computing (ICSTCC), pages 677–682, 2018.

[Ba17] K. Bakshi: Microservices-based software architecture and approaches, in 2017 IEEE Aerospace Conference, pages 1–8, 2017.

[BM+16] Kaibin Bao, Ingo Mauser, Sebastian Kochanneck, Huiwen Xu, Hartmut Schmeck: A Microservice Architecture for the Intranet of Things and Energy in Smart Buildings: Research Paper, in Proceedings of the 1st International Workshop on Mashups of Things and APIs, MOTA '16, ACM, New York, NY, USA, ISBN 978-1-4503-4669-6, pages 3:1–3:6, URL `http://doi.acm.org/10.1145/3007203.3007215`, 2016.

[Be14] D. Bernstein: Containers and Cloud: From LXC to Docker to Kubernetes, IEEE Cloud Computing, vol. 1, no. 3, pages 81–84, 2014.

[Bl92] John Bloomer: Power programming with RPC, O'Reilly & Associates, Sebastopol, CA, ISBN 9780937175774, 1992.

[BM17] M. Brattstrom, P. Morreale: Scalable Agentless Cloud Network Monitoring, in 2017 IEEE 4th International Conference on Cyber Security and Cloud Computing (CSCloud), pages 171–176, 2017.

[BD+17a] Eric Braun, Clemens Düpmeier, Daniel Kimmig, Wolfgang Schillinger, Kurt Weissenbach: Generic Web Framework for Environmental Data Visualization, in Volker Wohlgemuth, Frank Fuchs-Kittowski, Jochen Wittmann (editors), Advances and New Trends in Environmental Informatics, Springer International Publishing, Cham, ISBN 978-3-319-44711-7, pages 289–299, 2017.

[BD+17b] Eric Braun, Clemens Düpmeier, Stefan Mirbach, Ulrich Lang: 3D Volume Visualization of Environmental Data in the Web, in Jiří Hřebíček, Ralf Denzer, Gerald Schimak, Tomáš Pitner (editors), Environmental Software Systems. Computer Science for Environmental Protection, Springer International Publishing, Cham, ISBN 978-3-319-89935-0, pages 457–469, 2017.

[CA13] C. Catal, M. Atalay: A Systematic Mapping Study on Architectural Analysis, in 2013 10th International Conference on Information Technology: New Generations, pages 661–664, 2013.

[CF+00] Ann Chervenak, Ian Foster, Carl Kesselman, Charles Salisbury, Steven Tuecke: The data grid: Towards an architecture for the distributed management and analysis of large scientific datasets, Journal of Network and Computer Applications, vol. 23, no. 3, pages 187 – 200, URL `http://www.sciencedirect.com/science/article/pii/S1084804500901103`, 2000.

[CP+15] S. Cirani, M. Picone, P. Gonizzi, L. Veltri, G. Ferrari: IoT-OAS: An OAuth-Based Authorization Service Architecture for Secure Services in IoT Scenarios, IEEE Sensors Journal, vol. 15, no. 2, pages 1224–1234, 2015.

[CK+01] Paul Clements, Rick Kazman, Mark Klein: Evaluating Software Architectures: Methods and Case Studies, SEI Series in Software Engineering, Addison-Wesley, Boston, MA, ISBN 978-0-201-70482-2, 2001.

[CMIS2012] CMIS: Content Management Interoperability Services (CMIS) Version 1.1, URL `http://docs.oasis-open.org/cmis/CMIS/v1.1/csprd01/CMIS-v1.1-csprd01.pdf`, 2012.

[CG16] Andrew W. Crapo, Steven Gustafson: Semantics: Revolutionary Breakthrough or Just Another Way of Doing Things?, chap. 5, Springer International Publishing, Cham, ISBN 978-3-319-16658-2, pages 85–118, URL `https://doi.org/10.1007/978-3-319-16658-2_5`, 2016.

[DB18] S. K. Datta, C. Bonnet: Next-Generation, Data Centric and End-to-End IoT Architecture Based on Microservices, in 2018 IEEE International Conference on Consumer Electronics - Asia (ICCE-Asia), pages 206–212, 2018.

[Deu19] Bundesrepublik Deutschland: Umweltinformationsgesetz, URL `https://www.bgbl.de/xaver/bgbl/start.xav?jumpTo=bgbl104s3704.pdf`, 2004.

[DC+19] Jasenka Dizdarević, Francisco Carpio, Admela Jukan, Xavi Masip-Bruin: A Survey of Communication Protocols for Internet of Things and Related Challenges of Fog and Cloud Computing Integration, ACM Comput. Surv., vol. 51, no. 6, URL `https://doi.org/10.1145/3292674`, 2019.

[DN02] L. Dobrica, E. Niemela: A survey on software architecture analysis methods, IEEE Transactions on Software Engineering, vol. 28, no. 7, pages 638–653, 2002.

[EM+17] O. Ethelbert, F. F. Moghaddam, P. Wieder, R. Yahyapour: A JSON Token-Based Authentication and Access Management Schema for Cloud SaaS Applications, in 2017 IEEE 5th International Conference on Future Internet of Things and Cloud (FiCloud), pages 47–53, 2017.

[Ev03] Evans: Domain-Driven Design: Tacking Complexity In the Heart of Software, Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, ISBN 0321125215, 2003.

[Fi00] Roy Thomas Fielding: REST: Architectural Styles and the Design of Network-based Software Architectures, Doctoral dissertation, University of California, Irvine, URL `http://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm`, 2000.

[FIWARE] FIWARE: The Open Source platform for our smart digital future, URL `https://www.fiware.org/`.

[Fo11] Martin Fowler: PolyglotPersistence, URL `https://martinfowler.com/bliki/PolyglotPersistence.html`, 2011.

[FL14] Martin Fowler, James Lewis: Microservices, URL `http://martinfowler.com/articles/microservices.html`, 2014.

[FM+17] P. D. Francesco, I. Malavolta, P. Lago: Research on Architecting Microservices: Trends, Focus, and Potential for Industrial Adoption, in 2017 IEEE International Conference on Software Architecture (ICSA), pages 21–30, 2017.

[FU+18] Y. Y. Fridelin, M. R. Ulil Albaab, A. R. Anom Besari, S. Sukaridhoto, A. Tjahjono: Implementation of Microservice Architectures on SEMAR Extension for Air Quality Monitoring, in 2018 International Electronics Symposium on Knowledge Creation and Intelligent Computing (IES-KCIC), pages 218–224, 2018.

[GC+18] A. Galletta, L. Carnevale, A. Buzachis, A. Celesti, M. Villari: A Microservices-Based Platform for Efficiently Managing Oceanographic Data, in 2018 4th International Conference on Big Data Innovations and Applications (Innovate-Data), pages 25–29, 2018.

[GH+94] Erich Gamma, Richard Helm, Ralph Johnson, John M. Vlissides: Design Patterns: Elements of Reusable Object-Oriented Software, Addison-Wesley Professional, 1 ed., ISBN 0201633612, URL `http://www.amazon.com/Design-Patterns-Elements-Reusable-Object-Oriented/dp/0201633612/ref=ntt_at_ep_dpi_1`, 1994.

[GB+18] A. S. Gaur, J. Budakoti, C. Lung: Design and Performance Evaluation of Containerized Microservices on Edge Gateway in Mobile IoT, in 2018 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 (SmartData), pages 138–145, 2018.

[GG11] H. Gharavi, R. Ghafurian: Smart Grid: The Electric Energy System of the Future [Scanning the Issue], Proceedings of the IEEE, vol. 99, no. 6, pages 917–921, 2011.

[GG+16] P. Giessler, M. Gebhart, R. Steinegger, S. Abeck: Checklist for the API Design of Web Services based on REST, International Journal on Advances in Internet Technology, vol. 9, no. 3 & 4, pages 41 – 51, 2016.

[GM+17] A. Goering, J. Meister, S. Lehnhoff, P. Herdt, M. Jung, M. Rohr: Reference architecture for open, maintainable and secure software for the operation of energy networks, CIRED - Open Access Proceedings Journal, vol. 2017, no. 1, pages 1410–1413, 2017.

[HATEOAS] HATEOAS: HATEOAS Driven REST APIs – REST API Tutorial, `https://restfulapi.net/hateoas/`, (Accessed on 09/26/2019).

[HH+19] Philipp Hertweck, Tobias Hellmund, Hylke van der Schaaf, Jürgen Moßgraber, Jan-Wilhelm Blume: Management of Sensor Data with Open Standards, in Zeno Franco, José J. González, José H. Canós (editors), Proceedings of the 16th International Conference on Information Systems for Crisis Response and Management, València, Spain, May 19-22, 2019, ISCRAM Association, pages 1126–1139, URL `http://idl.iscram.org/files/philipphertweck/2019/1859_PhilippHertweck_etal2019.pdf`, 2019.

[HG+17] Benjamin Hippchen, Pascal Giessler, Roland Steinegger, Michael Schneider, Sebastian Abeck: Designing Microservice-Based Applications by Using a Domain-Driven Design Approach, International Journal on Advances in Software (1942-2628), vol. 10, pages 432 – 445, 2017.

[HS+18] X. J. Hong, H. Sik Yang, Y. H. Kim: Performance Analysis of RESTful API and RabbitMQ for Microservice Web Application, in 2018 International Conference on Information and Communication Technology Convergence (ICTC), pages 257–259, 2018.

[ISO25010] ISO/IEC 25010: ISO/IEC 25010:2011, Systems and software engineering — Systems and software Quality Requirements and Evaluation (SQuaRE) — System and software quality models, 2011.

[JN+16] D. Jaramillo, D. V. Nguyen, R. Smart: Leveraging microservices architecture by using Docker technology, in SoutheastCon 2016, pages 1–5, 2016.

[JA+18] M. A. Jarwar, S. Ali, I. Chong: Exploring Web Objects enabled Data-Driven Microservices for E-Health Service Provision in IoT Environment, in 2018 International Conference on Information and Communication Technology Convergence (ICTC), pages 112–117, 2018.

[JP+17] S. K. Jensen, T. B. Pedersen, C. Thomsen: Time Series Management Systems: A Survey, IEEE Transactions on Knowledge and Data Engineering, vol. 29, no. 11, pages 2581–2600, 2017.

[JSON-Schema] JSON-Schema: JSON Schema | The home of JSON Schema, `https://json-schema.org/`, (Accessed on 09/16/2019).

[JSONAPI] jsonapi.org: JSON:API — A specification for building APIs in JSON, `https://jsonapi.org/`, (Accessed on 09/27/2019).

[KL+16a] H. Kang, M. Le, S. Tao: Container and Microservice Driven Design for Cloud Infrastructure DevOps, in 2016 IEEE International Conference on Cloud Engineering (IC2E), pages 202–211, 2016.

[KI+18] S. Kanti Datta, M. Irfan Khan, L. Codeca, B. Denis, J. Härri, C. Bonnet: IoT and Microservices Based Testbed for Connected Car Services, in 2018 IEEE 19th International Symposium on "A World of Wireless, Mobile and Multimedia Networks" (WoWMoM), pages 14–19, 2018.

[KB+94] R. Kazman, L. Bass, G. Abowd, M. Webb: SAAM: a method for analyzing the properties of software architectures, in Proceedings of

16th International Conference on Software Engineering, pages 81–90, 1994.

[KK+00] Rick Kazman, Mark Klein, Paul Clements: ATAM: Method for Architecture Evaluation, Tech. Rep. CMU/SEI-2000-TR-004, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA, URL `http://resources.sei.cmu.edu/library/asset-view.cfm?AssetID=5177`, 2000.

[KL+16b] Ali Khalili, Antonis Loizou, Frank van Harmelen: Adaptive Linked Data-Driven Web Components: Building Flexible and Reusable Semantic Web Interfaces, in Harald Sack, Eva Blomqvist, Mathieu d'Aquin, Chiara Ghidini, Simone Paolo Ponzetto, Christoph Lange (editors), The Semantic Web. Latest Advances and New Domains, Springer International Publishing, Cham, ISBN 978-3-319-34129-3, pages 677–692, 2016.

[KS+17] K. Khanda, D. Salikhov, K. Gusmanov, M. Mazzara, N. Mavridis: Microservice-Based IoT for Smart Buildings, in 2017 31st International Conference on Advanced Information Networking and Applications Workshops (WAINA), pages 302–308, 2017.

[KK+16] M. Kolchin, N. Klimov, I. Shilin, D. Garayzuev, A. Andreev, D. Mouromtsev: 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 (SmartData), pages 416–419, 2016.

[KQ17] Nane Kratzke, Peter-Christian Quint: Understanding cloud-native applications after 10 years of cloud computing - A systematic mapping study, Journal of Systems and Software, vol. 126, pages 1 – 16, URL `http://www.sciencedirect.com/science/article/pii/S0164121217300018`, 2017.

[KJ+15] A. Krylovskiy, M. Jahn, E. Patti: Designing a Smart City Internet of Things Platform with Microservice Architecture, in 2015 3rd International Conference on Future Internet of Things and Cloud, pages 25–30, 2015.

[LH+17] D. Lu, D. Huang, A. Walenstein, D. Medhi: A Secure Microservice Framework for IoT, in 2017 IEEE Symposium on Service-Oriented System Engineering (SOSE), pages 9–18, 2017.

[Ma06] R. E. Mackiewicz: Overview of IEC 61850 and benefits, in 2006 IEEE Power Engineering Society General Meeting, pages 8 pp.–, 2006.

[Ma15] V. Malcher: Design Patterns in Cloud Computing, in 2015 10th International Conference on P2P, Parallel, Grid, Cloud and Internet Computing (3PGCIC), pages 32–35, 2015.

[Ma13] Nathan Marz: Big data: principles and best practices of scalable realtime data systems, O'Reilly Media, [S.l.], ISBN 978-1617290343, URL `http://www.amazon.de/Big-Data-Principles-Practices-Scalable/dp/1617290343`, 2013.

[MC+06] Chris A. Mattmann, Daniel J. Crichton, Nenad Medvidovic, Steve Hughes: A Software Architecture-based Framework for Highly Distributed and Data Intensive Scientific Applications, in Proceedings of the 28th International Conference on Software Engineering, ICSE '06, ACM, New York, NY, USA, ISBN 1-59593-375-1, pages 721–730, URL `http://doi.acm.org/10.1145/1134285.1134400`, 2006.

[MW17] B. Mayer, R. Weinreich: A Dashboard for Microservice Monitoring and Management, in 2017 IEEE International Conference on Software Architecture Workshops (ICSAW), pages 66–69, 2017.

[MJ+16] S. A. Mikkelsen, R. H. Jacobsen, A. F. Terkelsen: DB A: An Open Source Web Service for Meter Data Management, in 2016 IEEE Symposium on Service-Oriented System Engineering (SOSE), pages 4–13, 2016.

[Mo16] Daniel Moldovan: On Monitoring and Analyzing Elastic Cloud Systems, Ph.D. thesis, -, 2016.

[MM+18] I. J. Munezero, D. Mukasa, B. Kanagwa, J. Balikuddembe: Partitioning Microservices: A Domain Engineering Approach, in 2018 IEEE/ACM Symposium on Software Engineering in Africa (SEiA), pages 43–49, 2018.

[Na17] N. Naik: Docker container-based big data processing system in multiple clouds for everyone, in 2017 IEEE International Systems Engineering Symposium (ISSE), pages 1–7, 2017.

[Ne15] Sam Newman: Building Microservices, O'Reilly Media, Inc., 1st ed., ISBN 1491950358, 9781491950357, 2015.

[OGCSensorThings] OGC: OGC SensorThings API, URL `https://www.ogc.org/standards/sensorthings`, 2016.

[Ol15] Gary Olliffe: Microservices : Building Services with the Guts on the Outside, URL `https://blogs.gartner.com/gary-olliffe/2015/01/30/microservices-guts-on-the-outside/`, 2015.

[Om19] OMICRON: Sampled Values in IEC 61850 environments, URL `https://www.omicronenergy.com/en/applications/power-utility-communication/sampled-values-in-iec-61850-environments/`, 2019.

[PC+16] Claus Pahl, Pooyan Jamshidi: Microservices: A Systematic Mapping Study, in Proceedings of the 6th International Conference on Cloud Computing and Services Science - Volume 1 and 2, CLOSER 2016, SCITEPRESS - Science and Technology Publications, Lda, Portugal, ISBN 978-989-758-182-3, pages 137–146, URL `https://doi.org/10.5220/0005785501370146`, 2016.

[Pa16] K. Pathomkeerati: A new generic approach for web based dashboard solutions in a microservice architecture, Diploma thesis, Karlsruhe Institute of Technology, 2016.

[PS15] A. Patidar, U. Suman: A survey on software architecture evaluation methods, in 2015 2nd International Conference on Computing for Sustainable Global Development (INDIACom), pages 967–972, 2015.

[PN+16] Ronald K. Pearson, Yrjö Neuvo, Jaakko Astola, Moncef Gabbouj: Generalized Hampel Filters, EURASIP Journal on Advances in Signal Processing, vol. 2016, no. 1, page 87, URL `https://doi.org/10.1186/s13634-016-0383-6`, 2016.

[PH+14] Michael Pizzo, Ralf Handl, Martin Zurmuehl: OData Version 4.0 Part 1: Protocol, `http://docs.oasis-open.org/odata/odata/v4.0/os/part1-protocol/odata-v4.0-os-part1-protocol.html`, (Accessed on 09/27/2019), 2014.

[PB18] Eve Porcello, Alex Banks: Learning GraphQL: Declarative Data Fetching for Modern Web Apps, O'Reilly Media, Inc., 1st ed., ISBN 1492030716, 9781492030713, 2018.

[PA14] S. Prasad, S. B. Avinash: Application of polyglot persistence to enhance performance of the energy data management systems, in 2014 International Conference on Advances in Electronics Computers and Communications, pages 1–6, 2014.

[ProtoBuf] ProtoBuf: Protocol Buffers Version 3 Language Specification, URL `https://developers.google.com/protocol-buffers/docs/reference/proto3-spec`.

[RS+18] F. Rademacher, J. Sorgalla, S. Sachweh: Challenges of Domain-Driven Microservice Design: A Model-Driven Perspective, IEEE Software, vol. 35, no. 3, pages 36–43, 2018.

[RE+10] S. Rusitschka, K. Eger, C. Gerdes: Smart Grid Data Cloud: A Model for Utilizing Cloud Computing in the Smart Grid Domain, in 2010 First IEEE International Conference on Smart Grid Communications, pages 483–488, 2010.

[SW+18] Sherif Sakr, Marcin Wylot, Raghava Mutharaju, Danh Le Phuoc, Irini Fundulaki: Linked Data : Storing, Querying, and Reasoning, SpringerLink, Springer, Cham, ISBN 9783319735153, URL `https://doi.org/10.1007/978-3-319-73515-3`, 2018.

[SH+19] I. L. Salvadori, A. Huf, F. Siqueira: Semantic Data-Driven Microservices, in 2019 IEEE 43rd Annual Computer Software and Applications Conference (COMPSAC), vol. 1, pages 402–410, 2019.

[SC+14] Hugo Sequeira, Paulo Carreira, Thomas Goldschmidt, Philipp Vorst: Energy Cloud: Real-Time Cloud-Native Energy Management System to Monitor and Analyze Energy Consumption in Multiple Industrial

Sites, in Proceedings of the 2014 IEEE/ACM 7th International Conference on Utility and Cloud Computing, UCC '14, IEEE Computer Society, Washington, DC, USA, ISBN 978-1-4799-7881-6, pages 529–534, URL `https://doi.org/10.1109/UCC.2014.79`, 2014.

[SL+17] L. Sun, Y. Li, R. A. Memon: An open IoT framework based on microservices architecture, China Communications, vol. 14, no. 2, pages 154–162, 2017.

[TL18] D. Taibi, V. Lenarduzzi: On the Definition of Microservice Bad Smells, IEEE Software, vol. 35, no. 3, pages 56–62, 2018.

[Ta12] Sasu Tarkoma: Publish / Subscribe Systems: Design and Principles, John Wiley & Sons, URL `https://www.ebook.de/de/product/19239255/sasu_tarkoma_publish_subscribe_systems.html`, 2012.

[TP+17] S. Tripathi, S. Prajapati, N. A. Ansari: Modified optimal algorithm: For load balancing in cloud computing, in 2017 International Conference on Computing, Communication and Automation (ICCCA), pages 116–121, 2017.

[US+12] Mathias Uslar, Michael Specht, Sebastian Rohjans, Jörn Trefke, José M. González: The Common Information Model CIM: IEC 61968/61970 and 62325 - A Practical Introduction to the CIM, Springer Publishing Company, Incorporated, ISBN 3642252141, 9783642252143, 2012.

[VL+14] M. Vianden, H. Lichter, A. Steffens: Experience on a Microservice-Based Reference Architecture for Measurement Systems, in 2014 21st Asia-Pacific Software Engineering Conference, vol. 1, pages 183–190, 2014.

[VG+15] M. Villamizar, O. Garcés, H. Castro, M. Verano, L. Salamanca, R. Casallas, S. Gil: Evaluating the monolithic and the microservice architecture pattern to deploy web applications in the cloud, in 2015 10th Computing Colombian Conference (10CCC), pages 583–590, 2015.

[Vi15] M. Virmani: Understanding DevOps bridging the gap from continuous integration to continuous delivery, in Fifth International Conference on the Innovative Computing Technology (INTECH 2015), pages 78–82, 2015.

[VS+15] J. Voutilainen, J. Salonen, T. Mikkonen: On the Design of a Responsive User Interface for a Multi-device Web Service, in 2015 2nd ACM International Conference on Mobile Software Engineering and Systems, pages 60–63, 2015.

[VC16] T. Vresk, I. Čavrak: Architecture of an interoperable IoT platform based on microservices, in 2016 39th International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO), pages 1196–1201, 2016.

[W3C19] W3C: Web Components, URL `https://github.com/w3c/webcomponents`, 2019.

[W3CWebComp] W3C: W3C Webcomponents, URL `https://github.com/w3c/webcomponents`, original-date: 2013-11-11T04:00:30Z, 2020.

[WC+18] Y. Wang, B. Cheng, M. Niu: An End-user Microservice-Based Lightweight Architecture for IoT, in 2018 14th International Computer Engineering Conference (ICENCO), pages 68–72, 2018.

[WS] WebSocket: The WebSocket API (WebSockets), URL `https://developer.mozilla.org/en-US/docs/Web/API/WebSockets_API`.

[YS+15] N. Yu, S. Shah, R. Johnson, R. Sherick, M. Hong, K. Loparo: Big data analytics in power distribution systems, in 2015 IEEE Power Energy Society Innovative Smart Grid Technologies Conference (ISGT), pages 1–5, 2015.

[ZC07] J. S. Zepeda, S. V. Chapa: From Desktop Applications Towards Ajax Web Applications, in 2007 4th International Conference on Electrical and Electronics Engineering, pages 193–196, 2007.

[ZL+19] J. Zhou, L. Li, N. Zhou: Research and Application of Battery Production Data Management System Based on Microservice, in 2019 IEEE 9th International Conference on Electronics Information and Emergency Communication (ICEIEC), pages 1–5, 2019.

List of Publications

2017 Braun, E.; Düpmeier, C.; Kimmig, D.; Schillinger, W. & Weissenbach, K. Generic Web Framework for Environmental Data Visualization Advances and New Trends in Environmental Informatics, Springer International Publishing, 2017, 289-299

2017 Braun, E.; Schlachter, T.; Düpmeier, C.; Stucky, K.-U. & Suess, W. A Generic Microservice Architecture for Environmental Data Management Environmental Software Systems. Computer Science for Environmental Protection, Springer International Publishing, 2017, 383-394

2017 Braun, E.; Düpmeier, C.; Mirbach, S. & Lang, U. 3D Volume Visualization of Environmental Data in the Web Environmental Software Systems. Computer Science for Environmental Protection, Springer International Publishing, 2017, 457-469

2017 Schlachter, T.; Braun, E.; Düpmeier, C.; Schmitt, C. & Schillinger, W. A Generic Web Cache Infrastructure for the Provision of Multifarious Environmental Data Environmental Software Systems. Computer Science for Environmental Protection, Springer International Publishing, 2017, 360-371

2018 Braun, E.; Radkohl, A.; Schmitt, C.; Schlachter, T. & Düpmeier, C. A Lightweight Web Components Framework for Accessing Generic Data Services in Environmental Information Systems From Science to Society, Springer International Publishing, 2018, 191-201

2018 Schlachter, T.; Braun, E.; Düpmeier, C.; Müller, H. & Scherrer, M. From Sensors to Users—Using Microservices for the Handling of Measurement Data From Science to Society, Springer International Publishing, 2018, 203-213

2018 Ordiano, J. Á. G.; Bartschat, A.; Ludwig, N.; Braun, E.; Waczowicz, S.; Renkamp, N.; Peter, N.; Düpmeier, C.; Mikut, R. & Hagenmeyer, V. Concept and benchmark results for Big Data energy forecasting based on Apache Spark Journal of Big Data, Springer Science and Business Media LLC, 2018, 5

2018 Liu, J.; Braun, E.; Düpmeier, C.; Kuckertz, P.; Ryberg, D. S.; Robinius, M.; Stolten, D. & Hagenmeyer, V. A Generic and Highly Scalable Framework for the Automation and Execution of Scientific Data Processing and Simulation Workflows 2018 IEEE International Conference on Software Architecture (ICSA), 2018, 145-155

2018 Khalloof, H.; Jakob, W.; Liu, J.; Braun, E.; Shahoud, S.; Duepmeier, C. & Hagenmeyer, V. A Generic Distributed Microservices and Container Based Framework for Metaheuristic Optimization Proceedings of the Genetic and Evolutionary Computation Conference Companion, Association for Computing Machinery, 2018, 1363–1370

2019 Liu, J.; Braun, E.; Düpmeier, C.; Kuckertz, P.; Ryberg, D.; Robinius, M.; Stolten, D. & Hagenmeyer, V. Architectural Concept and Evaluation of a Framework for the Efficient Automation of Computational Scientific Workflows: An Energy Systems Analysis Example Applied Sciences, MDPI AG, 2019, 9, 728

2020 Sidler, J.; Braun, E.; Schlachter, T.; Düpmeier, C. & Hagenmeyer, V. Design of a Web-Service for Formal Descriptions of Domain-Specific Data Environmental Software Systems. Data Science in Action, Springer International Publishing, 2020, 201-215

Curriculum Vitæ

since 10/2015	Research associate and PhD candidate
	Karlsruhe Institute of Technology (KIT), Karlsruhe, Germany
10/2012 - 09/2015	M. Sc. in Computer Science
	Karlsruhe Institute of Technology (KIT), Karlsruhe, Germany
03/2011 - 01/2014	Software Engineer
	Wildenmann Tools & Services GmbH & Co KG, Ettlingen, Germany
10/2009 - 09/2012	B. Sc. in Computer Science
	Karlsruhe Institute of Technology (KIT), Karlsruhe, Germany
09/2001 - 07/2009	Enseignement secondaire classique
	Lycée Aline Mayrisch, Luxembourg, Luxembourg
09/1994 - 07/2000	Enseignement fondamental
	École Am Sand, Niederanven, Luxembourg