Microservice-based Architecture for the Integration of Data Backends and Dashboard Applications in the Energy and Environment Domains

Jannik Sidler, Eric Braun, Christian Schmitt, Thorsten Schlachter, and Veit Hagenmeyer

Institute for Automation and Applied Informatics (IAI)
Karlsruhe Institute of Technology (KIT)
Karlsruhe, Germany
jannik.sidler@kit.edu
https://www.iai.kit.edu

Abstract. This article presents a software architecture based on the onion architecture that uses the concept of application microservices in order to integrate data backends with dashboard applications. Its main goal is to reduce the complexity in the architecture's frontend and therefore to increase the performance of the application for the user. The concept of the added application layer as well as its interaction with the other parts of the architecture is described in detail. Then an evaluation of its advantages is presented which shows the benefits of the concept regarding performance and simplicity using a real-world use case in the energy and environmental domains.

Keywords: environmental information systems, energy dashboards, web application, software architecture, application microservice, onion architecture

1 Introduction

Nowadays, climate change is one of the dominating and most difficult challenges mankind has to face. In this context, many measures are in progress of being accomplished in order to support the energy transition and the protection of the environment. During this process, the requirements for software engineers began to rise steadily, since one of the most important goals on the road to a successful deceleration of climate change is making the world's population understand how their behaviour can be relevant in the process of deceleration. To improve this understanding, there are various possibilities. As the internet has a great relevance in the process of gathering information, a great potential lies in the numerous websites that deal with topics such as environment protection. Consequently, an important aspect in the process of the population's understanding of the changes our world succumbs is the utility of these websites. Therefore,

tools and applications are required which make the usage of renewable energy sources, the effects of the energy transition and the change of different sectors of the environment visible for everyone. Additionally, the influences of climate change on the environment have to be measured and visualised.
In the present article, a software architecture is presented which supports the formerly mentioned scenarios. The architecture consists of three parts. The first part is a set of web services which is called Generic Microservice Backend (GMB), using the microservice design pattern ([5] and [6]). GMB's competence consists in the provision of generic master data and timeseries data that are usable in different contexts and applications. GMB is based on previous work described in [1] and is used as a foundation in the present article. In general, these web services used in GMB are derived from a software architecture called *onion architecture* [2]. They fulfill the tasks of the domain services which are described in Section 2.
The second part is an application layer that also contains a web services which are located on top of the previously described domain services. These web services are called application services in the onion architecture [2]. In contrast to the domain services, they perform tasks which are more specific to an existing application, where the domain services perform tasks that are of a more general nature which can be reused in multiple other scenarios. the application services also use the microservice design pattern [5][6].
The third part is a flexible and reusable framework [7]. Its main goal is the visualisation of data by using lightweight web components which provide easy integration into existing websites and other web frameworks.
The main goals of this article can be summarised into three different aspects:

- Usage of application services to enhance the efficiency of an existing architecture
- Reusability of components
- Support for applications in the energy- and environment domain

The structure of this article is as follows: In Section 2, foundations and related work, which are principally connected to the presented work, are described and compared with the architecture at hand. In the third section, the mentioned architecture is described in detail and core concepts are explained. In Section 4, an evaluation regarding the usefulness of the achieved work is provided. Additionally, improvements of the current architecture are explained and discussed. The last section summarises the presented concepts and approaches and provides an outlook on further work that still has to be done in this context.

2 Foundations and Related Work

This section presents foundations of the given software architecture as well as similar work from different contexts. One important aspect in the scope of the present article is the term *microservice*. According to [5] and [6], microservices

are a design pattern in software engineering enhancing the encapsulation of functionality in distributed logical units and furthermore, the independence of these logical units of other functionality used in the same context. According to Fowler, microservices are "an approach to develop a single application as a suite of small services, each running in its own process and communicating with lightweight mechanisms" [5]. This approach has several advantages. First, the reusability of microservices is high. They provide an API that can be easily reworked, according to the context's requirements. Furthermore, they can be reused easily in different contexts due to their modular design. Second, the maintainability of microservices is very good. As they are independent of other microservices, they can be maintained without creating undesired effects on other components, as the only point of access to the microservice is its API. Finally, microservices fit very well in modern software deployment infrastructures. As they encapsulate only a certain, manageable amount of functionality, they can easily be deployed "by a fully automated deployment machinery" [5]. For these reasons, the architecture presented in this paper uses the microservice design pattern.
Moreover, the core concept used in the present article are the application services given by the onion architecture [2]. The onion architecture is a pattern which compares the structure of a software architecture with the one of an onion. In Figure 1, this reference architecture is depicted in detail. The domain model is located in the middle. It is the core of the architecture, as every behaviour and state of the architecture depends on the way the domain is organised. The layer around the domain model are the domain services which are closely related to the domain model and implement the behaviour of the domain. Further outside, the application services connect the user interface and infrastructure (which is the layer on the outer edge) to the domain services. They contain business logic which defines the functionality of the application that is build using the domain layer. The outer layers are coupled with the next inner layer in an unidirectional manner, which means that the user interface is indirectly coupled to the domain services as well, but not vice versa.
In the present article, the onion architecture and the microservice design pattern are combined. This has already been done in [3]. This is derived by using domain-driven design (DDD) [4] which strongly encourages using microservice architectures and can additionally be mapped to the onion architecture. Consequently, the application services will be adressed more specifically as application microservices in the following sections and in the context of the presented architecture.

3 Architecture for Applications using Data Backends

In this section, the architecture for applications using generic data backends is explained. Figure 2 shows that the architecture is grouped into three different layers.
The domain layer mainly consists of three microservices [5]: The Master Data

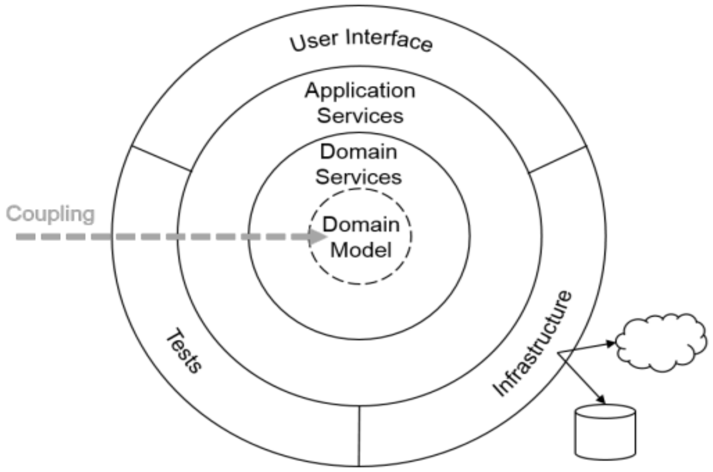


Fig. 1: The onion architecture [3].

Service (MD) is responsible for managing master data, including functionality to read and write them from/to a persistent database. To ensure this functionality, the MD service has an appropriate REST API, providing all the functionality needed by applications, e.g. providing requested data by using filters for more specific search queries and by providing the ability to sort and compute aggregations [1]. The Semantic Service's (SEM) goal is the validation of master data objects by using predefined schemas as references. These schemas define the structure of master data objects of a specific category. Therefore, the SEM service is required to verify the validity of master data objects [1]. Furthermore, the SEM service adds semantic information to master data objects, as for instance additional descriptions, data types and further meta information. The Timeseries Service (TS) is needed for managing timeseries data, e.g. measurements. Therefore, it uses optimised database tools for this kind of data. Additionally, its API provides all necessary functionality for aggregating and filtering time series data according to the application's requirements [1].
The user interface [7] represents the application that provides the top-level components which are responsible for user interactions. It is implemented by using a flexible framework which has been developed previously. More details can be obtained from [7].
The application layer is located between the previously described layers. The concept of the application microservices based on the onion architecture [2] has been described in Section 2. In the present paper, this concept is used mainly to increase the performance of the user interface by reducing its complexity and responsibilities. This extension is described in the following.
In order to improve performance, there are two main aspects which have to be considered. The first one is data processing. Components requesting data po-

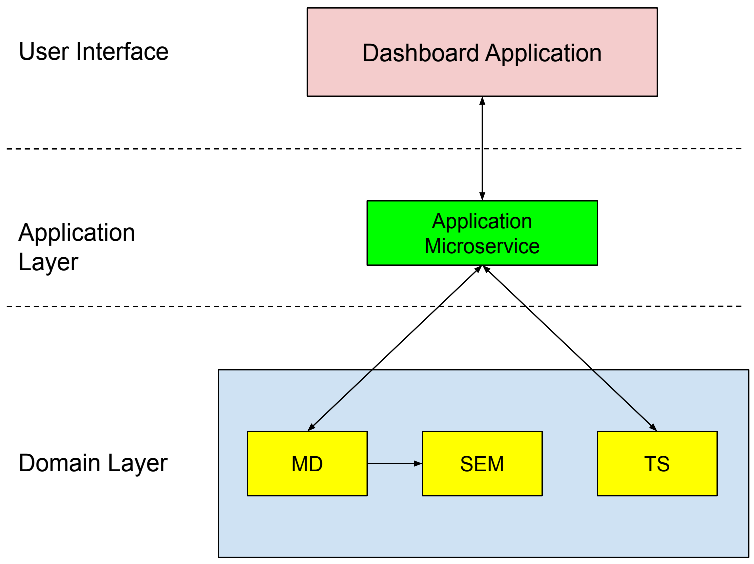


Fig. 2: Architecture of the given dashboard application.

tentially need specific formats containing particular attributes and/or a fixed number of values. These special data formats usually cannot be provided by domain services, which means that a certain number of processing steps has to be added to the application layer. This data processing, if executed in the user interface, has to be performed by the end user's hardware resources, as without these processing steps, the data cannot be visualised by the user interface components. Consequently, if the user cannot afford these hardware requirements, the user interface will be slow and it takes longer until changes are visible, which leads to the user's dissatisfaction. Additionally, integrating business logic in the user interface is not consistent with best practices such as the model-view-controller pattern (MVC) [8]. The solution for this problem is to shift these data processing tasks from the user interface (and from the user's hardware) to the backend side, precisely to the application layer. The application microservice(s) can be used to accomplish processing tasks and consequently provide the specific data formats for the applications.
The second aspect that has to be considered in order to improve the user interface's performance is the network aspect. Without an application layer (or a similar gateway component), the user interface of the microservice-based architecture has to make many requests to get the required data from the backend since the data is distributed across different microservices and resources within these services. Furthermore, in this case, the overhead from the data retrieving is much higher compared to retrieving the data with less requests. This leads to a higher network load, as constantly high numbers of data requests have to be made in the background. Combined with the previously described aspect of data

processing, this leads to a significantly reduced performance. This problem can be tackled by using an additional application layer, too. The number of requests the user interface has to make can be reasonably reduced when it does not have to organise the required data by itself. Alternatively, the application microservice is used to orchestrate the different requests to the domain microservices. Afterwards, the data is filtered by accomplishing necessary processing tasks and by generating the required data format. At this point, an improvement regarding performance of the user interface and complexity of the single component layers is reached. Instead of assigning the mentioned tasks to the user interface, an additional application layer is included being explicitly responsible for application-specific tasks (e.g. creating particular data formats for the user interface) that are independent of the components used in the user interface.
Figure 3 shows the structure of the application microservice. It consists mainly of three components. First, there is a REST API which provides endpoints encapsulating a certain set of functionality. In the context of this architecture, one endpoint is used to update the state of the whole application. Within this REST API which can be called by the user interface, certain other functions and classes are called containing the particular logic required for data processing. So on the one hand, there is a functional block which contains all necessary steps to perform the data processing and to create the appropriate data format. On the other hand, the data for this tasks is provided by the domain services. The requests to the domain services are orchestrated in a third part of the application microservice.

4 Evaluation

In the evaluation section, two different applications are presented and their purpose and functionality are described.
The first application that is shown in Figure 4 is a measurement net application with the purpose to visualize data from measurement sensors which measure different air pollution parameters. The application allows the visualisation of different parameters, such as particular matter, nitrogen dioxide and ozone. Additionally, an air quality index can be selected which provides a calculation of the air quality based on the formerly mentioned air pollutants. These parameters can be selected by the corresponding dropdown menu (”Luftschadstoffe” → ”air pollutants”)[1]. Similarly, the location of a measurement station can be selected by using another dropdown menu (”Messstellenauswahl” → ”choice of measurement station”). Alternatively, the location can also be selected by clicking a measurement station on the map component provided by the application. After choosing parameters for both menus, a legend appears right to the map, providing particular values that have been measured recently. Furthermore, the legend explains the colours of the measurement stations on the map, which provide an evaluation of the recently measured value, leading from very good (”sehr gut” → ”very good”, colour deep blue) to very bad (”sehr schlecht” → ”very bad”,

[1] Pattern for die translations: ”german term” → ”english term”

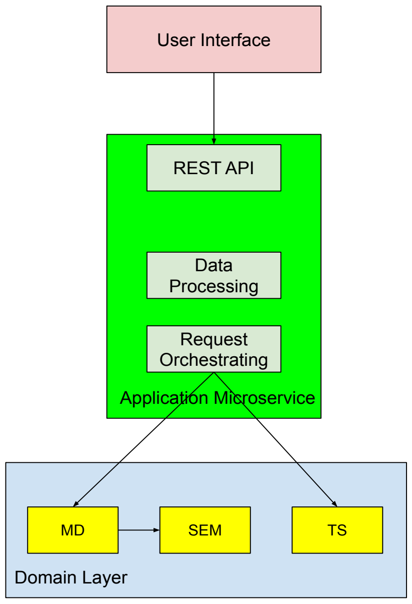
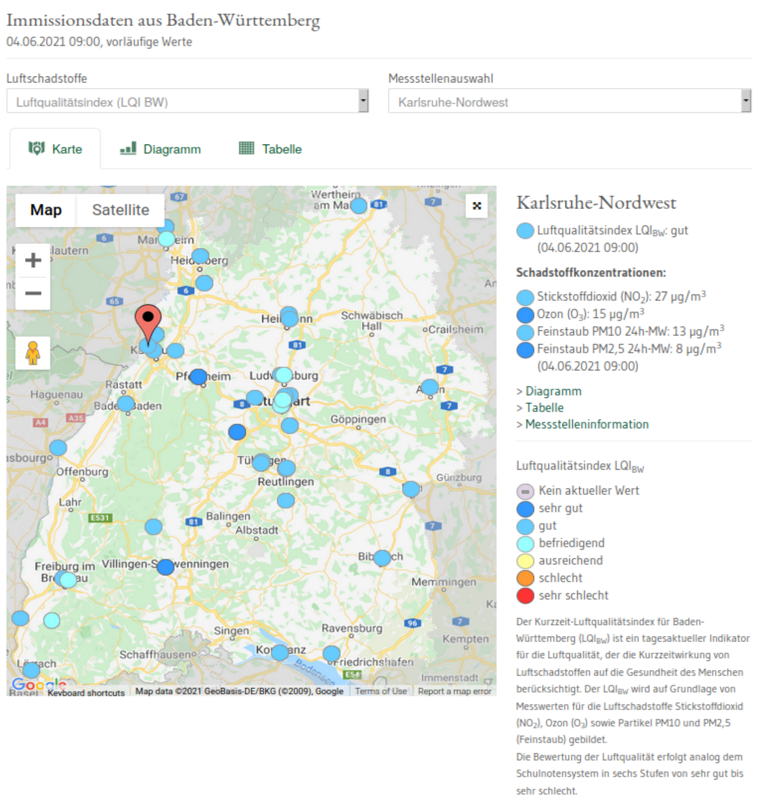


Fig. 3: Structure of the application microservice in collaboration with user interface and domain layer.

colour red). Furthermore, there are two different views that can be selected. On the one hand, the diagram ("Diagramm" → "diagram") view provides an overview over measured values of one measurement station from a certain time period (e.g., last seven days until today). On the other hand, the table ("Tabelle" → "table") view displays the measured values of all stations in a table which makes comparing values from different stations more comfortable.

This application does not use application microservices (further referred to as AMS), which results in a high network load for clients who use the application. Additionally, after making all required requests, the user interface (further referred to as UI) still does data processing tasks which mainly means generating data formats for the different components. On the one hand, this process violates the MVC pattern [8], as the view components should not implement any application logic, nor should they do any processing tasks on the data they visualise. On the other hand, the high network load combined with the mentioned processing tasks slow the application and lead to an increased initial loading time.

The second application presented in this context is a dashboard application of the EnergyLab 2.0, a research project at Karlsruhe Institute of Technology (KIT), which is used for the evaluation of the concept presented in Section 3 and which shows the advantages of the concept compared with the formerly presented environmental application. Figure 5 shows the dashboard application with its different components. It has several components that visualise data of


Immissionsdaten aus Baden-Württemberg
04.06.2021 09:00, vorläufige Werte

Luftschadstoffe: Luftqualitätsindex (LQI BW)
Messstellenauswahl: Karlsruhe-Nordwest

Karlsruhe-Nordwest

● Luftqualitätsindex LQI_{BW}: gut (04.06.2021 09:00)

Schadstoffkonzentrationen:

● Stickstoffdioxid (NO_2): 27 µg/m^3
● Ozon (O_3): 15 µg/m^3
● Feinstaub PM10 24h-MW: 13 µg/m^3
● Feinstaub PM2,5 24h-MW: 8 µg/m^3
(04.06.2021 09:00)

> Diagramm
> Tabelle
> Messstelleninformation

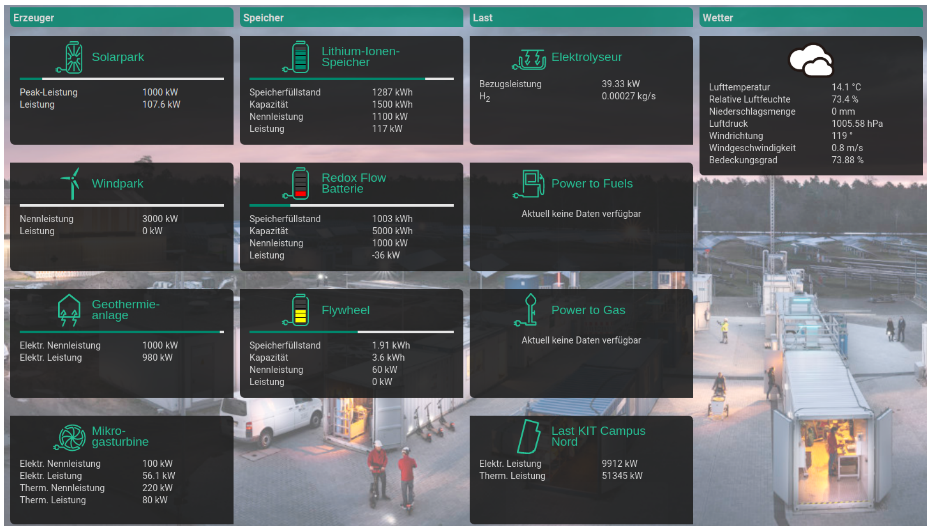
Luftqualitätsindex LQI_{BW}

● Kein aktueller Wert
● sehr gut
● gut
● befriedigend
● ausreichend
● schlecht
● sehr schlecht

Der Kurzzeit-Luftqualitätsindex für Baden-Württemberg (LQI_{BW}) ist ein tagesaktueller Indikator für die Luftqualität, der die Kurzzeitwirkung von Luftschadstoffen auf die Gesundheit des Menschen berücksichtigt. Der LQI_{BW} wird auf Grundlage von Messwerten für die Luftschadstoffe Stickstoffdioxid (NO_2), Ozon (O_3) sowie Partikel PM10 und PM2,5 (Feinstaub) gebildet.
Die Bewertung der Luftqualität erfolgt analog dem Schulnotensystem in sechs Stufen von sehr gut bis sehr schlecht.

Fig. 4: Screenshot of the air measurement application [in German].

different energy storage systems and energy plants that make use of renewable energy technologies, e.g. a solar park, a redox flow battery and a wind park. They retrieve their data from sensors that are located in the field (in the area of KIT) which are connected to a backend described in the following. Furthermore, the user interface contains one or more data source components. Data source components are located in the background and therefore, they are invisible for the client. Their purpose is the fetching of data. This dashboard application is the user interface of the architecture presented in Section 3.
The domain layer (backend) consists of the microservices presented and described previously. Additionally, the application uses an AMS that orchestrates requests to the domain layer and creates a suitable data format for the UI based on the data collected from the domain microservices.
The main advantage of using an AMS in this architecture lies in the cost reduction of requests made by the UI. Requests can have different costs, based on the physical distance between the corresponding devices and on the data they contain. For example, a request from one device to another device in the same network is relatively cheap, while a request from one network to another one is



Erzeuger

Solarpark

Peak-Leistung	1000 kW
Leistung	107.6 kW

Windpark

Nennleistung	3000 kW
Leistung	0 kW

Geothermie-anlage

Elektr. Nennleistung	1000 kW
Elektr. Leistung	980 kW

Mikro-gasturbine

Elektr. Nennleistung	100 kW
Elektr. Leistung	56.1 kW
Therm. Nennleistung	220 kW
Therm. Leistung	80 kW

Speicher

Lithium-Ionen-Speicher

Speicherfüllstand	1287 kWh
Kapazität	1500 kWh
Nennleistung	1100 kW
Leistung	117 kW

Redox Flow Batterie

Speicherfüllstand	1003 kWh
Kapazität	5000 kWh
Nennleistung	1000 kW
Leistung	-36 kW

Flywheel

Speicherfüllstand	1.91 kWh
Kapazität	3.6 kWh
Nennleistung	60 kW
Leistung	0 kW

Last

Elektrolyseur

Bezugsleistung	39.33 kW
H_2	0.00027 kg/s

Power to Fuels

Aktuell keine Daten verfügbar

Power to Gas

Aktuell keine Daten verfügbar

Last KIT Campus Nord

Elektr. Leistung	9912 kW
Therm. Leistung	51345 kW

Wetter

Lufttemperatur	14.1 °C
Relative Luftfeuchte	73.4 %
Niederschlagsmenge	0 mm
Luftdruck	1005.58 hPa
Windrichtung	119 °
Windgeschwindigkeit	0.8 m/s
Bedeckungsgrad	73.88 %

Fig. 5: Screenshot of the Energy Lab 2.0 dashboard.

more expensive. In the following, AMSs are always considered to be in the same network as the corresponding domain layer, which makes requests between UI and AMS more expensive than requests between AMS and the domain layer.
Moreover, there are different strategies how the UI is updated. As described previously, the UI consists of components hat can potentially be dependant on different backend (domain layer) resources in the domain layer. Therefore, there are various approaches how the data in the components can be updated. To evaluate the previously presented concept of using AMSs, two different approaches are considered. The first one is a synchronous updating strategy. In this case, the UI components are updated together by using one data source component. The second strategy is an asynchronous update, where the components can be updated independently of each other. Each strategy results in a different total number of requests that is required to accomplish the update. Figure 6 shows the number of requests in the synchronous case without (a) and with using an AMS (b). The bold arrows show expensive requests, while the slim arrows show cheap requests. As an example scenario, an update of the four dashboard components solar park, lithium ion battery, wind park and weather is considered. While the solar park and the wind park components show the currently injected electrical power, the weather component shows local weather data and the lithium ion battery component shows the current level of power the battery still has, as well as the maximal capacity, the nominal capacity and general power.
The dashboard depicted in Figure 5 implements the synchronous update strategy and uses an AMS. Compared to the case without a such one, the number of expensive requests is higher, as the data source has to request all necessary resources by itself, which results in a potentially higher number of expensive

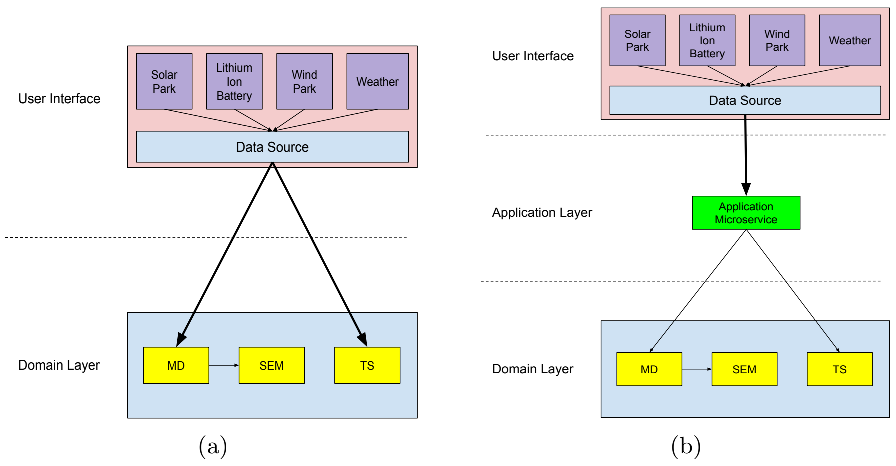


Fig. 6: Possible synchronous request orchestration without (a) and with AMS (b).

requests, depending on the number of resources that has to be requested. In the case with the AMS, the UI's data source component has to make only one expensive request per update.
Figure 7 shows the same scenario by implementing the asynchronous updating strategy. Although this updating concept is not used in the dashboard application shown in Figure 5, it still can be reasonable, especially when there are components that need more frequent updates then other ones within the same application. In this case, the components are updated individually, which means that every component has its own data source, as depending on the update frequency, fewer data sources may become a bottleneck in the updating process. In this scenario, the number of expensive requests is higher as the data sources may have to request different resources from the domain layer, depending on the corresponding requirements of the requesting component.
As previously stated, the expensive requests are much more significant than cheap requests in terms of network connection. Therefore, the cheap requests are neglected and the focus of the following description lies in the expensive requests. To evaluate the presented concept from Section 3, it is assumed that each UI component accesses different resources from the domain layer which is the worst-case scenario as it requires the biggest number of expensive requests. Table 1 shows the number of expensive requests for the different scenarios. For the asynchronous cases, the usage of an AMS reduces the number of expensive requests by 50%, since it makes cheaper requests to the domain layer. In the synchronous case, without an AMS the UI has to make 8 requests, since it is assumed that each UI component requires different resources. By using an AMS, the number of expensive requests can be reduced by 87,5%. This is, when considering only the value, a great reduction of the network load. Still, it has to be

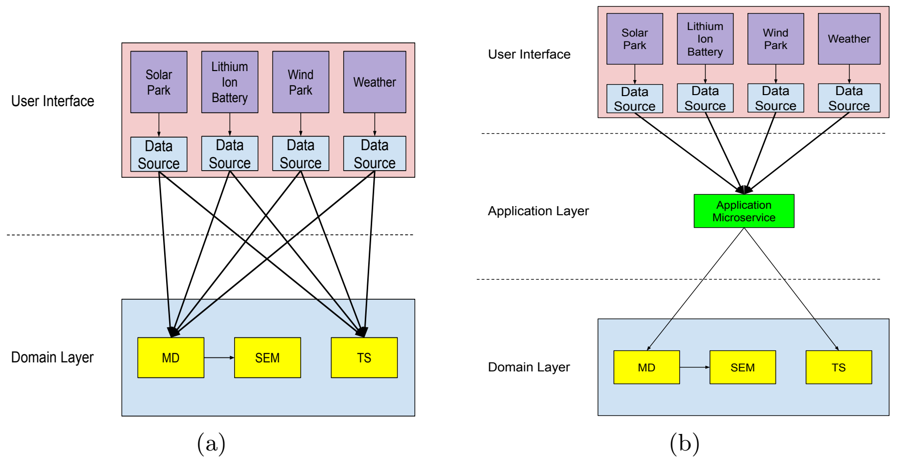


Fig. 7: Possible asynchronous request orchestration without (a) and with AMS (b).

mentioned that the increase of performance is most significant when the UI has to be updated frequently. When only sporadic updates are required, the saved costs and the relative increase of performance decrease. Regarding the two presented use cases from Figure 4 and 5, it can be stated that the usage of the presented concept leads to an increase of performance, as regular updates are necessary in both cases. The result is a significant reduction of data transfer and therefore, a reduced network load for the client using the application. By achieving this reduction, and consequently by using the proposed architecture, the power consumption of client devices can be reduced.

Type	Number of Requests
asynchronous update without AMS	8
asynchronous update with AMS	4
synchronous update without AMS	8
synchronous update with AMS	1

Table 1: Number of expensive requests for the different updating scenarios.

5 Conclusion and Outlook

The present article describes the usage of application microservices, originally derived from the application services from the onion architecture [2] within a

software architecture that formerly encapsulated application-specific tasks in its user interface. In this concept, the application microservices perform application-specific tasks as well as tasks usually executed by a gateway component, which are the orchestration of requests. By using this design pattern, the number of requests sent simultaneously from the user interface to the backend (domain layer) is reduced significantly. Additionally, data processing tasks are encapsulated in the application microservice to a high degree, which results in an improved performance for the the application. However, the processing efforts transferred from the user interface to the backend have to be made available on the server infrastructure. Moreover, two real applications (one from the energy domain, one from the environmental domain) are shown and discussed and the benefit of the provided concept is stated in their corresponding context.
Future tasks based on the presented concept are the design of a formal engineering process integrating the modified pattern into a microservice architecture.
Furthermore, the effort required to implement application microservices has to be examined. There are scenarios where many of them are required while the tasks they perform are relatively similar, so there can be a certain overhead that has to be compared with the benefit that is originally created by the usage of application microservices.

References

1. Braun, Eric et al.: A Generic Microservice Architecture for Environmental Data Management. International Symposium on Environmental Software Systems (ISESS) 2017, pp. 383 - 394
2. Palermo, Jeffrey: The Onion Architecture (2008).
https://jeffreypalermo.com/2008/07/the-onion-architecture-part-1/ (last accessed: July 16, 2021)
3. Hippchen, Benjamin et al.: Designing Microservice-Based Applications by Using a Domain-Driven Design Approach. International Journal on Advances in Software (2017), pp. 432 - 445.
4. Evans, Eric: Domain-Driven Design: Tackling Complexity in the Heart of Software. Addison-Wesley, 2004.
5. Fowler, Martin and Lewis, James: Microservices – definition of this new architectural term.
https://martinfowler.com/articles/microservices.html (last accessed: May 25, 2021)
6. Newman, Sam: Building Microservices. O'Reilly Media Inc. (2015)
7. Braun, Eric et al.: A Lightweight Web Components Framework for Accessing Generic Data Services in Environmental Information Systems.
From Science to Society (2017), pp. 191-201
8. Fowler, Martin: GUI Architectures (2006).
https://martinfowler.com/eaaDev/uiArchs.html (last accessed: June 9, 2021)